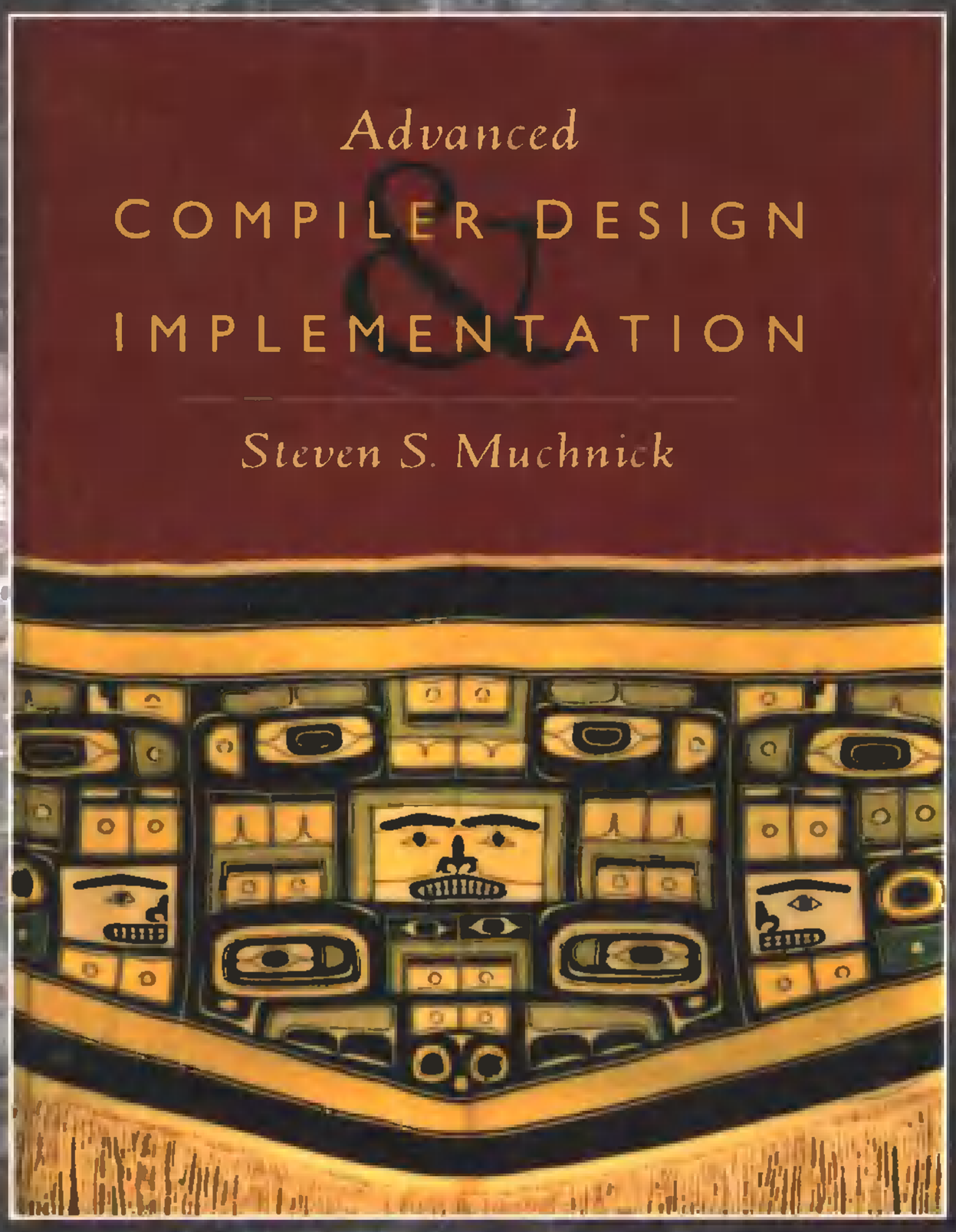
计 算 机 科 学 丛 书

高级编译器设计与实现

(美) Steven S. Muchnick 著　赵克佳　沈志宇　译



Advanced Compiler Design and Implementation

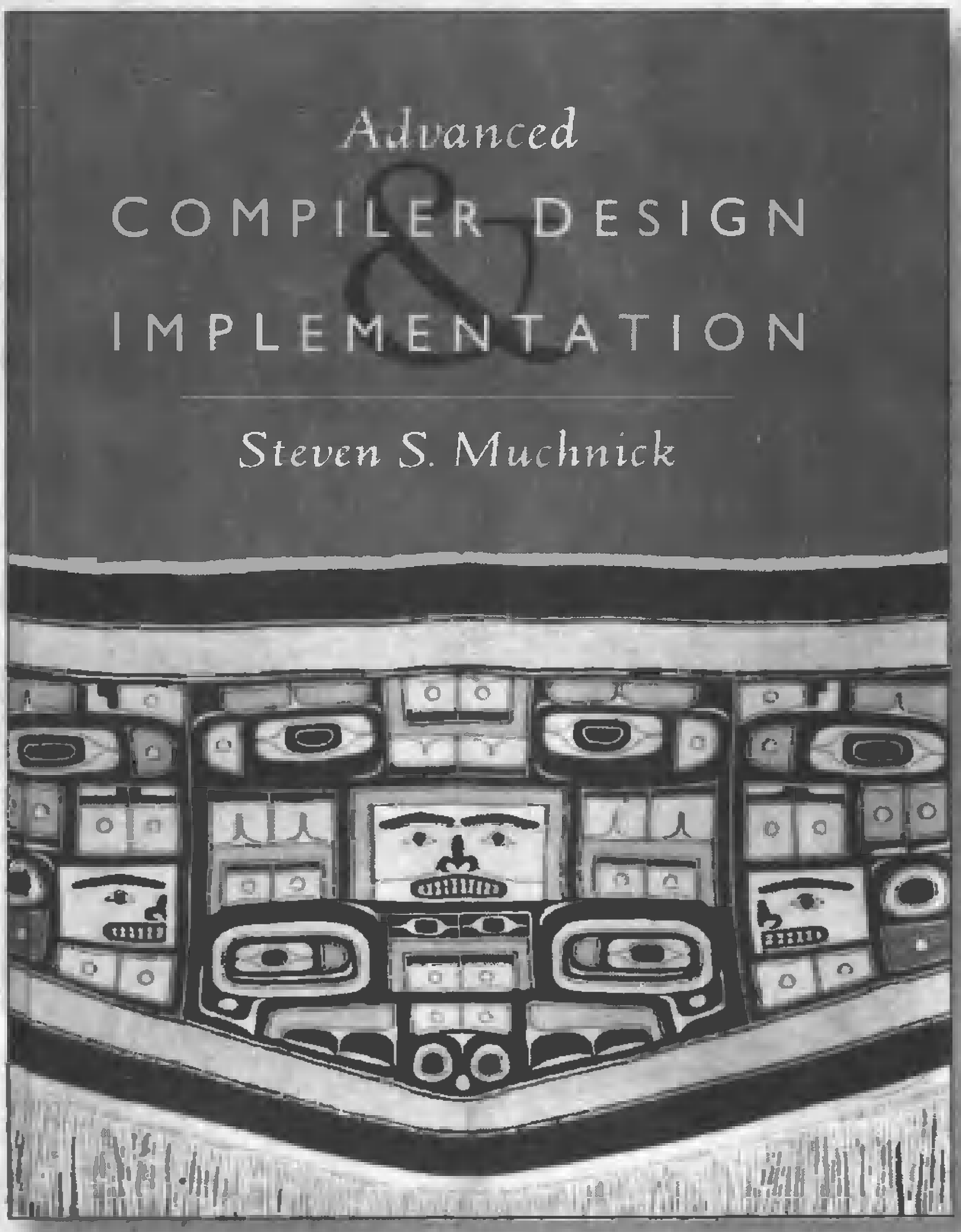
机械工业出版社
China Machine Press

bbs.theithome.com

计　算　机　科　学　丛　书

高级编译器设计与实现

（美） Steven S. Muchnick 著　赵克佳　沈志宇　译



Advanced Compiler Design and Implementation

机械工业出版社
China Machine Press

bbs.theithome.com

超星公司制作 请尊重作者版权

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

bbs.theithome.com

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

超星公司制作 请尊重作者版权

bbs.theithome.com

版权所有 请尊重作者版权

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

bbs.theithome.com

版权所有 请勿制作盗版

译 者 序

本书被称为 “鲸书”，这一方面是因为它的英文原版书的封面是一张奇尔卡特毛毯的照片，而这张毛毯的图案是一条在水中潜游的鲸鱼；另一方面则是因为本书是有关编译技术的重要著作之一，被人们誉为与编译技术的经典代表作“龙书”（《Compilers: Principles,Techniques,and Tools》）齐名。

本书的作者Steven S. Muchnick在编译技术方面有着深厚的理论基础，又具有丰富而广博的经验。他以这样两方面的功力写成的这本书，对于从事编译器设计和实现的科技人员具有无法衡量的价值。本书的内容主要集中在编译器的后端以及编译优化实现方面，全面阐述了在编写一个真实的编译器时遇到的各种关键问题及解决方法，反映了近十多年来针对现代计算机体系结构而出现的各种新的编译优化技术，同时作者对仍在研究中的编译实现和优化热点问题也给出了很好的综述和参考文献。注重真实语言和真实体系结构的编译实现方法，注重对现代体系结构性能有重要影响的各种优化是本书的特点，而这正是国内已出版的编译方面的著述所缺乏的，该书的翻译出版填补了国内编译著作在这一方面的空白。对于已经学习过编译原理课程的高年级本科生和研究生而言，书中的内容为他们进一步了解真实编译器中的设计问题和实现技术开拓了视野，而对于想实现一个优化编译器和在编译技术领域进行深入研究的科研人员而言，作者在书中指出的工程上必须注意但容易被忽视的许多问题以及给出的大量有价值的实用方法，为他们进行工程实践提供了指导，也为他们指出了深入研究的方向。

能充分发挥现代并行体系结构性能的优化编译器是最复杂的软件系统之一。设计和实现一个优化编译器既是一项工程，又是一种艺术创造。而翻译这本关于高级编译技术的书，对于我们而言，既是一次学习，又是一种欣赏。通过翻译本书，我们对高级编译技术又进行了一次系统的学习，收获颇丰。编译器的设计和实现要遵循计算机科学技术的规律，编译器的功能要能适应体系结构和编程语言的特点，充分发挥它们的特性，而编译器内部各组件要能有机协调地工作。我们在研究和编写一个好的编译器时，就在欣赏着科学技术的美感。在翻译本书的过程中，我们也钦佩作者杰出的聪明才智，体会到了自然的辩证法。

在自然科学领域，计算机科学技术是一门很年轻的学科，但在计算机科学技术中，编译技术却是一个相对“古老”的研究方向。新技术层出不穷，引人入胜，需要有人去研究。但同时我们也认为，编译技术是计算机技术的重要组成部分，随着计算机新技术新理论的不断涌现，编译技术本身也在不断发展，也还有着无穷的奥秘等待着人们去探索。信息化是全面建设小康社会的必由之路，实现信息化必须真正掌握发展计算机技术的主动权，而编译技术如同微处理器和操作系统技术等一样，是计算机技术的关键，必须掌握在自己手中。通过翻译本书，我们愿为我国编译技术的发展献上一份绵薄之力。

本书的前言和第1、9、10、11章由沈志宇翻译，其余由赵克佳翻译。全书由沈志宇审校。国防科大软件研究所的黄春、王锋、张小强等老师，以及郭学鹏、张定飞等研究生阅读了本书的部分译稿，并给出了不少有价值的修改意见，特此致谢。

由于我们才疏学浅，在翻译中不当之处还恳请读者诸君不吝赐教。

译 者

2005年3月于国防科学技术大学

bbs.theithome.com

超星公司制作 请尊重作者版权

序

从20世纪50年代中期以来，编译器设计就一直是计算机研究和开发中的活跃主题。第一个广泛使用的高级语言Fortran的成功在很大程度上归功于其早期高质量的编译器。John Backus和他在IBM的同事们认识到，除非编译器生成的代码与手工书写的机器代码的性能非常接近，否则程序员就不会放弃他们在使用汇编语言时所习惯采用的细节设计控制方法。Backus的小组创造了若干关键概念，这些概念构成了本书所研究问题的基础。这些概念包括循环优化中数组索引的处理和局部寄存器分配方法等。从那时起，研究人员和开发人员就一直不断用更有效的方法来改善和替代旧的方法。

既然编译器设计已有很长的历史，并且是一门相对成熟的计算技术，人们可能会问，为什么还要写一本有关此领域的新书？回答是明显的，编译器是一种生成由源程序至机器指令的高效映射的工具。语言的设计不断在变化，目标机体系结构也不断在变化，程序越来越复杂，其规模也越来越大。尽管编译器设计问题在高级层次上没有变化，但当我们深入其内部研究就会发现，它其实也一直在变化。此外，我们能够供编译器本身使用的计算资源也在增加。因此，现代编译器可以采用比以前更耗费时间和空间的算法。当然，研究人员也在继续开发新的、更好的技术来解决传统的编译器设计问题。事实上，本书中的所有主题都是计算机体系结构变化的直接结果。

本书迎接现代语言和体系结构的挑战，帮助读者做好准备，去应对将来难免要遇到的编译器设计的新问题。例如，第3章在读者把握了符号表和局部作用域的知识后，讲述了如何处理Ada、Modula-2和其他现代语言中的导入和导出作用域。而且，由于运行时的环境基本上规定了源语言的动态语义，第5章关于运行时支持的高级问题（如编译共享对象）的讨论也是特别有价值的。第5章还讨论了某些现代语言丰富的类型系统和现代体系结构所要求的种种参数传递策略。

任何一部关于编译器设计的书，如果没有介绍代码生成，它就是不完整的。早期的代码生成研究提供了设计手工书写指令选择例程的方法，以及如何在进行指令选择的同时管理寄存器的方法。本书第6章在讨论代码生成时，论述了基于模式匹配的自动化技术。这种自动化技术之所以成为可能，其原因不仅仅是编译器研究的进步，而且也是因为指令集已经变得更为简单和更为规整，以及编译器已经能够构造和遍历中间代码树。

优化是高级编译器设计的核心，也是本书的重点。有许多理论性的成果是关于程序分析的。程序分析的目的既是为了优化的安全，也是为了其他目的。本书第7~10章回顾了迄今为止的各种经典分析方法，同时也介绍了以前只在研究论文中出现过的更新和更有效的方法。这种综合本身就是对编译器设计的一个重要贡献。随后的许多章节都要使用这些分析来完成各种优化方法。

近代计算机系统具备由较多数量的寄存器构成的寄存器集合，这促成了第16章关于寄存器分配的讨论。这一章概括了近十年来在寄存器分配问题上的算法和启发式方法。另外，计算机速度提高的一个重要的原因是并行性，即同时做若干件事情的能力。为了将串行程序转换为可以利用硬件并行性的并行程序，编译器可能需要以既保证正确性又增加并行性的方式重排部分计算。尽管完整的并行处理超出了本书的范围，但本书集中讨论了指令级并行，这导致了第9章关于依赖分析和第17章关于代码调度的关键问题的讨论。

bbs.theithome.com

第20章论述存储层次优化，这也是由现代目标机引起的。为了克服处理器和存储器访问速度之间的差距，现代目标机引入了不同的数据访问速度。从本书出版商的网站上还可以得到额外的一章，它讨论了目标代码转换。这种转换是基于编译技术的，它将一个程序在已有体系结构上的目标代码转换为新体系结构的目标代码，即使该程序在新体系结构上的源程序还未出现。

由于新语言的设计一直鼓励程序员对大程序的结构化使用更为成熟的方法，过程间分析和优化的重要性也随之增加了。随着分析方法的不断精炼和调整，以及更快的计算机使得所需要的分析工作已经能够在可接受的时间内完成，这种分析和优化的可行性也增加了。第19章专门讨论过程间信息的确定和使用。

编译器设计本质上是一种工程活动，它所使用的方法必须很好地解决现实（即，用真实的语言书写的且在真实的机器上执行的真实的程序）中出现的各种翻译问题。多数情况下，书写编译器的人必须接受他们面对的语言和机器，很少能够影响和改善这两者的设计。做什么样的分析和转换，以及什么时候做它们都是工程上的选择，但正是这些选择决定了一个优化编译器的速度和质量。这些设计选择在贯穿全书的优化方法和第21章的实例研究中都得到了极为重要的处理和体现。

作者Steven S. Muchnick最大的优势之一是具有丰富而广博的经验。他早期曾是计算机科学教授，后来作为惠普的PA-RISC和Sun的SPARC两种计算机体系结构开发团队的核心成员，将自己的知识和经验应用于编译器设计。每一种体系结构的初始工作完成之后，他就担任了该系统的高级编译器设计与实现小组的负责人。这些职业经历对他确定读者需要知道高级编译器设计的哪些内容很有帮助。他在研究和亲身开发方面的双重经验，对于指导读者做出编译器设计决策极具价值。

Susan Graham
加利福尼亚大学，伯克利分校

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

前　言

本书讨论单机编译器设计和实现技术领域的前沿问题，重点讨论编译优化技术（超过了本书60%的篇幅）。我们考虑了支持指令级并行的机器，但几乎完全忽略了大规模并行处理和向量处理的有关问题。

本书首先讨论编译器的结构、符号表管理（包括那些允许导入和导出作用域的语言）、中间代码结构、运行时支持问题（包括可以在运行时链接的共享对象），以及根据机器描述自动产生代码生成器等。之后，探讨过程内的（通常称为“全局的”）控制流分析、数据流分析、依赖关系分析和别名分析的各种方法，并介绍一系列的全局优化，包括那些作用于程序不同成分（从单个表达式到整个过程）的优化。接下来本书讲述过程间的控制流分析、数据流分析和别名分析，以及过程间优化和如何应用过程间信息来改善全局优化。然后，讨论有效利用层次存储系统的优化技术。最后，详细介绍4个分别来自DEC、IBM、Intel和Sun 微系统公司的商业化编译系统，以提供编译器结构、中间代码设计、优化策略和效果的专门例子。如我们将看到的，这些编译系统采用的技术具有广泛的代表性，并用不同的方法获得了类似的效果。

本书的写作过程

1990年6月到1991年，我在Sun微系统公司担任高级工程师。在ACM SIGPLAN的程序语言设计和实现技术年会上，我用半天时间作了一个名为“RISC系统高级编译技术”的讲座。在这个讲座上，我用大约130张幻灯片讲解了RISC体系结构和相关的编译技术，特别是优化技术。在那次讲座之后，我有了一个想法，即讲座材料可能是一颗渴望着阳光、土壤和水分，期待着长成参天大树的种子（实际上，在我的脑海中是一颗橡树种子），而这棵大树就是你面前的这本书。一年多后，我与Wayne Rosing讨论了这个想法，然后又向Sun微系统实验室主任作了汇报，几周后他就决定支持这本书的写作，并给予了一年半的时间和部分资助。

本书的第一稿中除了高级编译技术外，还包含了相当多的RISC体系结构的内容。不久我就认识到（在3位评阅者的帮助下），不必要写这么多体系结构的内容。新的RISC体系结构不断被开发出来，大多数大学的体系结构课程都介绍了研究编译技术所需要的相关知识，本书的重点应该是编译技术。

这导致了本书写作方向的一次重大改变。体系结构方面的大多数内容都被删除了，仅仅保留了体系结构中与编译过程决策有关的内容。编译器的材料也拓宽为包含了有关CISC的编译技术，同时决定只集中讨论单机编译技术，将并行化和向量化的内容留待其他书去讨论。有关编译技术的内容更加集中和深入，有些方面缩小了范围，有些方面又扩大了范围（例如，有关手工代码生成的内容几乎全部删除了，但添加了有关先进调度技术的内容，如踪迹调度和渗透调度）。这样就形成了你面前的这本书。

关于本书的封面

本书封面的图片是从作者的西北海岸民间艺术收藏中选取的，这是一张奇尔卡特毛毯的照片。这块毛毯是在19世纪晚期，由美国阿拉斯加东南部的一个特里吉特妇女，用红松内层树皮

bbs.theithome.com

超星公司制作 请尊重作者版权

制成的非常细的绳子和山羊毛线编织的。编织这样一块毛毯通常需要6~9个月。这块毛毯的图案分为3个部分。中间的一块描绘了一条在水中潜游的鲸鱼；鲸鱼头位于底部，是一个割裂开了的图形；中间有着鲸鱼面部的那个图形是鲸鱼的身体（在这类绘画中，看起来像鲸鱼面部的图形并不表示鲸鱼的面部）；鲸鱼的侧鳍在身体的两边；而顶部是鲸鱼的尾鳍。这个设计中的每一部分，就本身而言，都是功能上的，并没有表达什么含意；但它们按正确的方式组合起来，就描绘了一条在水中潜游的鲸鱼，显示了拥有这条毛毯的村长的权力和特权。

类似地，一个编译器的每个组件有着某种功能，但仅当这些组件以适当的方式组合在一起时，才能完整地实现编译器的功能。设计和编织这样一块毛毯需要技巧，同样，构造工业水准的编译器也需要技巧。每个行业都有一组特定的工具、材料、设计要素和总体模式，而所有这一切都必须按满足预期用户的需要和愿望的方式组合到一起。

本书的读者

本书预期的读者是需要了解设计和构造单机高级编译器有关问题的计算机专业人员、研究生和高年级本科生。我们假定读者已经选修了数据结构、算法、编译器设计和实现、计算机体系结构、汇编语言程序设计等课程，或已经具有相当的工作经验。

内容概览

全书分为21章，另有3个附录。

第1章　高级主题介绍

这一章介绍本书的主题，即高级编译器的设计和构造，同时讨论编译器的结构和编译优化的重要性，并介绍本书内容的组织结构。

第2章　非形式化编译算法表示

第2章介绍了一种称为ICAN的非形式化的程序设计表示方法，并给出了有关例子。这种表示方法用于表示本书中的编译算法。在介绍了这种语言的语法符号之后，我们给出了ICAN的概述，接着详细描述了该语言。读者读了ICAN的概述，就足以读懂本书中的大多数算法，只是在很少的情况下，才需要了解ICAN的全部细节。

第3章　符号表结构

第3章首先讨论变量的属性，例如存储类、可见性、易变性、作用域、大小、类型、对齐、结构、寻址方法等。然后给出了高效地构造和管理局部和全局符号表的方法，包括导入的和导出的作用域（在Ada、Mesa、Modula-2中存在这种情况）、存储绑定，以及在考虑上述属性的前提下产生取、存指令的方法。

第4章　中间表示

这一章集中介绍中间语言的设计、本书专用的三种中间语言，以及编译器中可能使用的中间代码的其他基本形式。我们使用了密切相关的高、中、低三种形式的中间语言，以便能够深入说明书中讨论的所有优化算法。我们还讨论了本书使用的中间语言和其他中间语言的相对重要性以及它们的用途。

另外两种更详细的中间代码形式，即静态单赋值（SSA）和程序依赖图分别在8.11节和9.5节中讨论。

bbs.theithome.com

版权公司制作 请勿盗作者版权

第5章　运行时支持

第5章关注的问题主要与运行时支持用高级程序设计语言编写的程序有关，我们讨论了数据表示、寄存器使用、运行时栈和栈帧的设计、参数传递、过程结构和链接、过程值变量、代码共享、位置无关代码，以及为支持符号和多态语言涉及的一些问题。

第6章　自动产生代码生成器

第6章讨论根据机器描述自动产生代码生成器的方法。我们详细给出了Graham-Glanville的语法制导技术，并介绍了另外两种方法，即语义制导分析和树模式匹配。

第7章　控制流分析

这一章和随后的三章讨论作用于过程的4种分析方法，这些分析对于实现正确和有效的程序优化至关重要。

第7章重点讨论确定过程中的控制流和构造控制流图（CFG）的方法。我们首先综述了若干种可能使用的方法，然后详细讨论其中的三种。第一种是采用深度为主搜索和必经结点的传统方法。在讨论它的同时，我们也将讨论流图的遍历，如CFG的前序遍历和后序遍历，以及查找CFG的强连通子图。另外两种方法依赖于可归约性的概念，它使得过程的控制流图可以层次式地构成。这两种方法分别称为区间分析和结构分析，两者之间的不同在于它们辨别的结构单元的类型。我们也讨论了用所谓的控制树表示一个过程的层次结构方法。

第8章　数据流分析

第8章讨论确定过程中数据流的方法。它从一个例子开始，然后讨论数据流分析所依赖的基本数学概念，即格、流函数和不动点。接下来给出数据流问题和解法的分类，然后详细讨论与前一章3种控制流分析方法对应的3种数据流分析方法。第一种方法是迭代数据流分析，它对应于使用深度为主搜索和必经结点的控制流分析。另两种数据流分析方法与基于控制树的两种控制流分析对应，并分别具有与它们相同的名字：区间分析和结构分析。在此之后，我们概述一种少见的称为位置式分析的新技术，并描述表示数据流信息的方法，即du链、ud链、网（web）和静态单赋值形式（或SSA）。最后考虑有关数组、结构和指针的处理，并以关于数据流分析器自动构造方法的讨论结束本章。

第9章　依赖关系分析和依赖图

第9章关注依赖关系分析，以及与依赖分析密切相关，称为程序依赖图的中间代码形式，其中的依赖关系分析是对数组和低级存储引用进行数据流分析的一种简易方法。我们首先讨论依赖关系，然后讨论如何计算基本块中的依赖关系，这些内容对于第17章中要介绍的代码调度技术至关重要。接着讨论循环中的依赖关系以及依赖关系测试方法，它们是第20章讨论数据存储优化的基础。最后讨论程序依赖图，这是一种直接表示控制和数据依赖的中间代码形式，用它来进行一系列的程序优化比用隐含着依赖信息的中间代码来进行程序优化要更为有效。

第10章　别名分析

第10章讨论别名分析，这种分析确定一个存储单元是否有可能由多个访问路径访问，例如既可用名字也可通过指针来访问。我们首先讨论在Fortran 77、Pascal、C和Fortran 90等语言中别名对程序的影响。接着讨论一种非常通用的对过程内出现的别名进行判定的方法，这种方法由一个与语言相关的别名收集器和一个与语言无关的别名传播器组成。可以对别名收集器和传播器作裁剪来提供信息，这些信息可以是依赖于或不依赖于过程控制流的信息，也可以是用其

bbs.theithome.com

版权所有 请勿制作盗版

他多种方式提供的信息，从而使得它成为一种能够适应特定程序设计语言编译器需要和时间限制的通用方法。

第11章　优化简介

第11章介绍代码优化的内容，它首先讨论了这样一个事实，即，尽管大多数优化的适用性和有效性是递归不可判定的，但仍然值得对于适用性和有效性是确定的程序施加这些优化，并且对第12~18章涉及的过程内优化技术作了一番快速浏览。然后讨论了流敏感性和可能与一定信息，如何将这些信息应用于优化，特定优化的相对重要性，以及它们应有的执行顺序。

第12章　前期优化

本章讨论在优化过程的较早阶段实行的一些优化技术，包括聚合量的标量替代、局部和全局值编号（施加于SSA形式的代码）、局部和全局复写传播、稀有条件常数传播（同样施加于SSA形式的代码）。本章还将讨论常数表达式求值（或常数折叠）和代数化简，以及如何在优化器中以子程序的形式实现它们，以便可随时根据需要而调用。

第13章　冗余删除

第13章关注几种类型的冗余删除。冗余删除的基本功能是删除过程中一条路径上多于一次执行的计算。本章描述了局部和全局公共子表达式删除、向前替换、循环不变代码外提、部分冗余删除和代码提升。

第14章　循环优化

第14章研究施加于循环的优化技术，包括归纳变量的识别、强度削弱、归纳变量删除、线性函数测试替换和不必要边界检查的删除。

第15章　过程优化

第15章给出施加于过程的优化技术，讨论了尾调用优化（包括尾递归删除）、过程集成、内嵌扩展、叶例程优化和收缩包装。

第16章　寄存器分配

第16章关注过程内的寄存器分配和指定。我们首先讨论局部的、基于代价的方法，然后讨论图着色法。我们讨论了作为寄存器分配对象的网、冲突图、冲突图的着色以及溢出代码的生成。在此之后，简要给出了使用过程的控制树进行寄存器分配的方法。

第17章　代码调度

第17章集中讨论局部和全局指令调度，这种调度通过重排指令的顺序来发挥现代处理器中流水线的最佳性能。局部调度有两种途径，一种是表调度，它处理一个基本块内的指令序列；另一种是分支调度，它处理基本块之间的连接。接着我们考察跨基本块边界的调度方法，包括前瞻取和提升。然后讨论两种软流水方法，称为窗口调度和展开-压实。接下来讨论循环展开、变量扩展、寄存器重命名和层次归约技术，前三种技术增加了调度的自由度，层次归约技术处理嵌在循环中的控制结构。最后以踪迹调度和渗透调度两种全局调度方法来结束本章。

第18章　控制流和低级优化

第18章研究控制流优化和通常施加于低级形式的中间代码上或类汇编语言表示上的一些优化。这些优化包括不可到达代码的删除、伸直化、if化简、循环化简、循环倒置、无开关化、

bbs.theithome.com

超星公司制作 请尊重作者版权

分支优化、尾融合（又称交叉转移）、条件传送指令的使用、死代码删除、内嵌扩展、分支预测、机器方言、指令归并，以及寄存器合并或归并。

第19章　过程间分析与优化

第19章关注将分析和优化技术推广到处理整个程序的技术。它讨论过程间的控制流分析、数据流分析、别名分析以及各种变换，给出两种方法，分别用于流不敏感副作用分析和别名分析，同时给出一种计算流敏感性副作用和进行过程间常数传播的算法。然后讨论过程间的优化，并将过程间的信息应用于过程内的优化，之后是过程间寄存器分配和全局引用的聚合。最后讨论将过程间分析和优化集成到编译器优化序列中时应有的顺序。

第20章　存储层次优化

第20章讨论发挥大多数系统都具有的层次存储器，特别是多级高速缓存效率的优化技术。我们首先讨论数据和指令高速缓存对程序性能的影响。然后考虑指令高速缓存的优化，包括指令预取、过程排序、过程和基本块的放置、过程内的代码安置、过程分裂，以及过程内和过程间方法的结合。

接下来我们讨论数据高速缓存的优化，主要集中于循环变换。我们不全面地研究这个问题，而是选择部分问题进行详细讨论，例如数组元素的标量替换和数据预取，并且对于其他问题只概要性地给出定义、术语和技术，它们涉及到的问题，以及相关技术的若干例子。这些技术包括数据重用、局部化、循环铺砌、标量优化与面向存储器的优化之间的相互作用。我们这样写是因为这一方面的技术非常新，以至于难以准确地选择权威性的优化算法。

第21章　编译器实例分析与未来的发展趋势

第21章给出4个商业化编译器的实例，它们涉及了目标机体系结构、编译设计技术、中间代码表示和优化技术的各个方面。这4种体系结构是：Sun 微系统的SPARC、IBM的POWER（和PowerPC）、Digital Equipment的Alpha以及Intel 386系列。对于其中的每一种，我们首先简要讨论它的体系结构，然后介绍硬件厂家为它提供的编译器。

附录A　本书使用的汇编语言指南

附录A对本书使用的每一种汇编语言给出一个简短的描述，使读者能更容易地理解书中的例子。它包含了关于SPARC、POWER（和PowerPC）、Alpha、Intel 386体系结构系列以及PA-RISC等汇编语言的讨论。

附录B　集合、序列、树、DAG和函数的表示

附录B对书中大多数抽象数据结构的具体表示作一番快速的回顾，它不能代替数据结构课程，但可以作为一些问题和方法的备用参考。

附录C　软件资源

附录C给出可通过匿名FTP或万维网访问的一些软件，这些软件可作为教学中的编译器实现课题的基础，在某些情况下，这些软件甚至可作为公司研制编译器的基础。

参考文献

最后的参考文献列出了针对本书所涉及问题的大量进一步阅读材料。

索引

书末附有一个索引，其中列出了本书涉及的各种语法符号和各个主题。

bbs.theithome.com

超星公司制作 请尊重作者版权

补遗、资源和网络扩充的内容

每一章的结尾都给出了若干习题。高级习题在左页边标注了ADV，研究性的习题标注了RSCH。部分习题解答的电子版可以从原出版社获得。与本书相关的资料可以在关于本书的如下网址找到：http:// books.elsevier. com/us//mk/us /subindex.asp? maintarget =companions/defaultindividual.asp&isbn=1558603204。教师如需本书习题的解答，请直接与原出版社联系。

资源

如上所述，附录C给出了可用于构建编译器项目的自由软件及获取的途径。从上述网址中也可获得整个附录，连同它给出的那些资源的地址链接。

网络扩充的内容

可以从出版社的上述网址得到关于目标代码转换的补充材料，这是一种从一种体系结构的目标代码（而不是从源程序）得到另一种体系结构的目标代码的转换。该网址上的扩充内容讨论了目标代码编译过程的原理，然后集中讨论了3个例子，即Hewlett-Packard的HP3000到PA-RISC的变换器OCT、数字设备公司（DEC）的VAX VMS到Alpha的变换器VEST和MIPS到Alpha的变换器mx。

致谢

首先，也是最重要的，我要感谢我以前在Sun 微系统公司的同事们，其中主要有Peter Deutsch、Dave Ditzel、Jon Kannegaard、Wayne Rosing、Eric Schmidt、Bob Sproull、Bert Sutherland、Ivan Sutherland以及程序设计语言部的成员们，尤其是Sharokh Mortazavi、Peter Damron和Vinod Grover。没有他们的支持和鼓励，就不可能有这本书。我还要特别感谢Wayne Rosing在本书写作的头一年半时间中给予的资助。

其次，我要感谢本书的评阅人J. Randy Allen、Bill Appelbe、Preston Briggs、Fabian E. Bustamante、Siddhartha Chatterjee、Bob Colwell、Subhendu Raja Das、Amer Diwan、Krishna Kunchithapadam、Jim Larus、Ion Mandoiu、Allan Porterfield、Arch Robison、Francisco J. Torres-Rojas和Michael Wolfe，他们都对本书的草稿提出了有价值的意见。

对其他公司和组织中与我共享了他们的编译器和其他技术信息的同行们，我也心怀感激，无论这些信息是否直接被本书采用。他们是：Cygnus Support的Michael Tiemann、James Wilson和Torbjörn Granlund，Digital Equipment的Richard Grove、Neil Faiman、Maurice Marks和Anton Chernoff，Green Hills Software的Craig Franklin，Hewlett-Packard的Keith Keilman、Michael Mahon、James Miller和Carol Thompson，Intel的Robert Colwell、Suresh Rao、William Savage和Kevin J. Smith，IBM的Bill Hay、Kevin O'Brien和F. Kenneth Zadeck，MIPS Technologies的Fred Chow、John Mashey和Alex Wu，Open Software Foundation的Andrew Johnson，Rice大学的Keith Cooper和他的同事们，斯坦福大学的John Hennessy、Monica Lam和他们的同事们，以及英国国防研究署的Nic Peeling。

我特别感激我的父母Samuel和Dorothy Muchnick，他们为我创造了一个充满希望的家庭氛围，鼓励着我提问和尝试回答艰难的问题，使我更加欣赏世间万物的美丽，无论它们是简单的还是复杂的，是自然的还是人造的。

Morgan Kaufmann出版社的工作人员以他们熟练的技巧和沉着的作风指导了这本书的出版。他们是：Bruce Spatz、Doug Sery、Jennifer Mann、Denise Penrose、Yonie Overton、Cheri

bbs.theithome.com

超星公司制作 请尊重作者版权

Palmer、Jane Elliott和Lisa Schneider。我感谢他们并期待着与他们再度合作。

Windfall Software的排字员Paul Anagnostopoulos设计了ICAN中用的特殊符号，细心而有主见地完成了本书的排版。一天上午他发给我一封电子邮件说，他期待着阅读我们共同劳动的成果，对我而言，这一天就具有了十分重要的意义。

最后，我要感谢Eric Milliren，他在过去的5年中让我花费了这么多的时间来写这本书，为我提供了滋润的家庭生活，并推迟了他自己的几个项目直到这本书即将完成。

bbs.theithome.com

版权公司制作 请尊重作者版权

本书涵盖了现代微处理器编译器的设计和实现方面的所有高级主题。

本书首先介绍编译器的结构、符号表管理、中间代码结构、运行时支持等问题，探讨过程内的控制流分析、数据流分析、依赖关系分析和别名分析的各种方法，并介绍一系列的全局优化。接下来，讲述过程间的控制流分析、数据流分析和别名分析，以及过程间优化和如何应用过程间信息来改善全局优化。然后，讨论有效利用层次存储系统的优化技术。最后，详细介绍4种商业化编译系统，以提供编译器结构、中间代码设计、优化策略和效果的专门例子。

本书适合作为高等院校计算机专业研究生和高年级本科生的教材，也适合需要了解高级编译器设计和构造有关问题的计算机专业人员参考。

Steven S. Muchnick: Advanced Compiler Design and Implementation (ISBN 1-55860-320-4).

Copyright © 1997 by Elsevier Science (USA).

Translation Copyright © 2005 by China Machine Press.

All rights reserved.

本书中文简体字版由美国Elsevier Science公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2003-8117

图书在版编目（CIP）数据

高级编译器设计与实现 /（美）马其尼克（Muchnick, S. S.）著；赵克佳，沈志宇译. －北京：机械工业出版社，2005.7

（计算机科学丛书）

书名原文：Advanced Compiler Design and Implementation

ISBN 7-111-16429-6

Ⅰ.高…　Ⅱ.①马…　②赵…　③沈…　Ⅲ.编译码器－程序设计－教材　Ⅳ.TN762

中国版本图书馆CIP数据核字（2005）第030408号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：朱起飞

北京慧美印刷有限公司印刷·新华书店北京发行所发行

2005年7月第1版第1次印刷

787mm × 1092mm 1/16　·40.25印张

印数：0 001-4 000册

定价：75.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

bbs.theithome.com

{ — ° ç Ô s < œ

ì { 2 ! > h Ú t X + h t ÿì U ì Y ' q Ł \ ÿÔ h t ô! l F '

r Ł ¸ & ì · ÿp ¨ M äh r u , ÿì p Q ' l p » ÿq Ó{ h 2 % 8 1 ¸ X

+

§ » öl Ó« · ¸ ì ì ÿ& ö÷ì O u ÷Ñr : h Ø Ò[v h ßO - ¸ [r : h { òV Œ

l 2 ì Þ ßO - ÿ ¸ [! . 3 + / Ø Ò[v K [v = l ‹ { { 2 2 C ÿ{ l Ó ¢ ‘ ÿÕÞ

ì k ¸ Àt ! l » ÿq Ó{ h ì F O L @ K J A P l B ? R > ÿ " \ î À l ‡ Ò ° ÖN â ì

Þ» W ì F O L @ K J A P À ï f fl ÿ_ w ı ý t »

r ¸ ÿ Æ H { 2 § \ t r ! t b < § h 1 ü + $! t b § { Òh

ÿw ì ' ì = O L J A P * ì ¨ y Ü ! J A P F A A L D L § h Úh 4 ¶ N _ ðl h r

ÿe v y Y u · ì ÿO M H O A N R A N ? O O = F = T v ß [> h H t ß ì W Ö ø | l

H t ßW ” ª l ¸ y ˇ ˇ » + r W r § ˇ ˇ ' æ ? ì ’ æ q ? Úæ

l ? ì ! ') h § J h l ') Þs ¥ ÿx Ł ðÓÓ b r k k ‡ à ! e

Æ Úh « J A P ì Þ ˇ h Æ F A A ÿs 7 h Æ = L E { 2 h Ø Ò[v % ‡ [y 4 ì

ÿ¢ ı r Ł ¸ & ì · ÿM ä l » ÿq Ó{ ! ¸ À

´ ¨ J A P F = R = ÿN _ ? t Øõ ! | { Ø Ò[v l ° t ÕÞ l » ! f ¸ » ÿh Óh

ý ¨ t ý ! ¨ h ' l B ? J A P p ‹ Ù¶ Ö . . . § k æÞ h f ÿÚh ° » k Ü t h ! '

r àk àt ! l » Þ t b ï ÿV Œ t b ÿV ' ì J A P ÿ Æ 9 ÀÆ Ø e Ù¶

÷t ÷ð ¥ ü r ð t b § l t âð Ó ° N _ Ót b ï ÿV Œ Ók Ü t

h ° ýß E ÷ðÿh t ¨ ¥ N _ ï < h ° Ó Üï ÿV Œ y W ð Àf ÿÚ

Ó{ ¾ ÿ – w J Ó ß ÿ » h ß > ÷Ñr : O ì ÿt Þh x 2 C ÿ{ J ˇ ⅜ x J

‹ S D W ` u U ÿ÷Ñr : ß > t ÜØ Ò[v h x J Ül N _ ’ þÿV Œ ï · k » Ø Ò[v

ÿ t Ü| { h { òV Œ h x b 2 C E ˇ Ó¸ Ó¸ Ø h t { ° òÿ –

ß > ! ì Þ» ÖN â W ì J A P h F = R = I # b y ÀÚh # s + h ï · [ÿ ' , — ¸ h w t °

' þh q h I R ? K N I I A : h Óh Àð t ý § ðh Úl p Àh Ú ì Þ Ó«

‹ { h 4 ¶ ’ 2 h Œ ß ^ R I » ì Þ W I k ‡ Öt b E p b

¸ t Ó¸ þx æ þ ' , P , # 2 6 + * I I ' , / ¨ ß ¥ h k ß > , ÿ » t ß h ì '

p ƒ p ˇ ÖJ h h ! & h t ! K ÿ “ ÿ I þ¸ ÿ ' , Û Ł ¸ F ÿV Œ h Ó f

ÿì ' , H s C h ß ß ÿ « » ï ì Þh ! l » h ÓÕÞe þ ' , f

N â { 2 E ÷òì ! ! [v = L E { 2 h ï · ØF F h ì fl · Æ ¥ Z V Œ ì Þ ? !

{ 2 » h W ì I B ? J A P I Às 7 5 7 h w f . ß äŒ ì 4 { 2 § Ó h ì » 2

4 ! ÿ 2 C Œ ß§ Ó > e ß > Óòì 4 h 4 ì y h t À h H > h ^ C 4 h ß > òì

y h 4 ì 4 h Ł Óh § Ó ’ ì h “ 9 ¸ k

{ 2 Þs k - h ß> ì J A P I B ? R > I þ¸ ÿø Þ ’ þ ° - ‰ R h » ÿì - h q

‰ R - t ß t k ° ÿ f ¸ ¥ Àà fi ´ ¯ Æ b § k ! ôô h ¯ b

ôÿ ¸ ‹ 4 ¯ h / ß k - l h ô ¾ 1 o y N âk Õ= ÷Õ âÿ I ¢ U ÿ

- h } À ¸ ˙ œ ¯ Y . r k H \ ´ Óÿ p ° . - Ñ] H ' y h n k ˘ k ‹ ÿi

- h æ g k Õ ¯ Æ b ´ Ós ô ÿ » ÿh q ÓÿÖt { 2 Þs ° [h L » Þ !

Ö À H Ôÿh w h # ° h ˘ Ò # 8 q I F ï + t ! ð t ° ¢ ÿt ì h

· m ‡ ÕÞÿ»

I ¥ ÿ; Õh üÿt f ¥ Z h ¸ k ‡ I & ì · ÿp ¨ h Ý óy ¸ Ö À! ¸ À

ÿ ÿÚ¸ h « I & ì · ÿM äÿø B ł Œ L @ B ¸ ' s ‹ s 6 2 % ¸ ! 8 1 L @ B

“ › D P P L > > O P D A E P D K I A ? K I N A = @ D P I P E @ D P I H

ü I ¸ “ Ø¸ Ó Þ ß ÿ» h t Óh · R { ÿ' , h Þ» W ì e ' ,

X . ” ª h / ÿ_ À Þÿ

» I q Óx J ì Þï öl Úh ì Þï · E ÷ Él » 2 [ÿÿ‹ ¸ Ö p ÿ § ì O

Ø { 2 ç – ¸ ö þ¸ “ o ' + Ø ‡ q + ÿ— – p I I p ÿ § t ¤ 8 1 · Þ» h w

· ÿ2 C ¥ ï ï ‡ ¸ ÿ_ h Ýà 6 ól § y h · ÿ2 C + À= † ı ¨ I

§ ? t Þ ! î \ h ¥ ÿ¸ ì ðl \ ô0 ÿ t ¸ I Y å{ Œ Ø , ! 4 ôÀ

! ^ ! M Ö Ø ‡ h ¸ œ w Æ Ø ì I § h Àì ÿ . ß Ô ! % I J

‡ y ÿì s ß ÿ»

+ Ø ‡ ×§ j > > O P D A E P D K I A ? K I

ßO - ì » t — ß ÿ" \ ¨ ÿ{ h Ó q -

ì Ú† p ÿ § h ¨ E § h x J k -

ì Ý×Ö ÿ“ ÿh ì Þh t Àe ^

q + B Æ Ø h v h Àì ÿ . ß Ô

Æ Ø ÿk p ÿ § h q Ö+ k Àâ · a î q + Æ Ø f ! ÿ

超星公司制作 请尊重作者版权

目　录

出版者的话
专家指导委员会
译者序
序
前言
第1章　高级主题介绍……1
1.1　编译器结构回顾……1
1.2　基本问题中的高级论题……2
1.3　代码优化的重要性……4
1.4　优化编译器的结构……5
1.5　激进型优化编译器中各种优化的位置……7
1.6　本书各章的阅读流程……10
1.7　本书没有涉及的相关主题……10
1.8　例子中所用的目标机……11
1.9　数的表示与数据的大小……11
1.10　小结……11
1.11　进一步阅读……12
1.12　练习……12
第2章　非形式化编译算法表示……13
2.1　扩展的巴科斯-诺尔范式语法表示……13
2.2　ICAN简介……14
2.3　ICAN概貌……16
2.4　完整的程序……17
2.5　类型定义……18
2.6　声明……18
2.7　数据类型和表达式……19
2.7.1　一般简单类型……20
2.7.2　枚举类型……20
2.7.3　数组……21
2.7.4　集合……21
2.7.5　序列……22
2.7.6　元组……23
2.7.7　记录……23
2.7.8　联合……24
2.7.9　函数……24
2.7.10　编译专用的类型……24
2.7.11　值nil……25
2.7.12　size运算符……25
2.8　语句……25
2.8.1　赋值语句……26
2.8.2　过程调用语句……27
2.8.3　返回语句……27
2.8.4　goto语句……27
2.8.5　if语句……27
2.8.6　case语句……27
2.8.7　while语句……28
2.8.8　for语句……28
2.8.9　repeat语句……28
2.8.10　ICAN的关键字……28
2.9　小结……29
2.10　进一步阅读……29
2.11　练习……29
第3章　符号表结构……31
3.1　存储类、可见性和生命期……31
3.2　符号属性和符号表项……32
3.3　局部符号表管理……34
3.4　全局符号表结构……35
3.5　存储绑定和符号寄存器……38
3.6　生成取数和存数指令的方法……42
3.7　小结……46
3.8　进一步阅读……46
3.9　练习……47
第4章　中间表示……49
4.1　与中间语言设计有关的问题……49
4.2　高级中间语言……50
4.3　中级中间语言……51
4.4　低级中间语言……52
4.5　多级中间语言……52
4.6　我们的中间语言：MIR、HIR和LIR……53
4.6.1　中级中间表示（MIR）……53

bbs.theithome.com

超星公司制作请尊重作者版权

4.6.2 高级中间表示（HIR）……56
4.6.3 低级中间表示（LIR）……57
4.7 用ICAN表示MIR、HIR和LIR ……58
4.7.1 用ICAN表示MIR……59
4.7.2 用ICAN表示HIR……62
4.7.3 用ICAN表示LIR……64
4.8 管理中间代码的若干数据结构和例程的ICAN命名……67
4.9 其他中间语言形式……70
4.9.1 三元式……70
4.9.2 树……71
4.9.3 无环有向图（DAG）……72
4.9.4 前缀波兰表示……73
4.10 小结……74
4.11 进一步阅读……74
4.12 练习……74
第5章 运行时支持……77
5.1 数据表示和指令……77
5.2 寄存器用法……80
5.3 局部栈帧……81
5.4 运行时栈……83
5.5 参数传递规则……84
5.6 过程的入口处理、出口处理、调用和返回……86
5.6.1 用寄存器传递参数：平面寄存器文件……87
5.6.2 用运行时栈传递参数……88
5.6.3 用具有寄存器窗口的寄存器传递参数……89
5.6.4 过程值变量……91
5.7 代码共享与位置无关代码……91
5.8 符号和多态语言支持……94
5.9 小结……96
5.10 进一步阅读……96
5.11 练习……97
第6章 自动产生代码生成器……99
6.1 简介代码生成器的自动生成……100
6.2 语法制导技术……100
6.2.1 代码生成器……102
6.2.2 代码生成器的产生器……103
6.2.3 删除链循环……110
6.2.4 删除句法阻滞……112
6.2.5 最后的考虑……115
6.3 语义制导的分析介绍……115
6.4 树模式匹配和动态规划……116
6.5 小结……120
6.6 进一步阅读……120
6.7 练习……121
第7章 控制流分析……123
7.1 控制流分析的方法……125
7.2 深度为主查找、前序遍历、后序遍历和宽度为主查找……128
7.3 必经结点和后必经结点……132
7.4 循环和强连通分量……139
7.5 可归约性……143
7.6 区间分析和控制树……144
7.7 结构分析……147
7.8 小结……156
7.9 进一步阅读……157
7.10 练习……157
第8章 数据流分析……159
8.1 一个例子：到达-定值……159
8.2 基本概念：格、流函数和不动点……163
8.3 数据流问题及其解决方法的分类……166
8.4 迭代数据流分析……168
8.5 流函数的格……171
8.6 基于控制树的数据流分析……172
8.7 结构分析……172
8.7.1 结构分析：向前问题……172
8.7.2 结构分析：向后问题……178
8.7.3 结构分析方程的表示……180
8.8 区间分析……181
8.9 其他方法……182
8.10 du链、ud链和网……183
8.11 静态单赋值形式……184
8.12 数组、结构和指针的处理……188
8.13 数据流分析器的自动构造……188
8.14 更贪婪的分析……189
8.15 小结……191
8.16 进一步阅读……192
8.17 练习……192

bbs.theithome.com

第9章 依赖关系分析和依赖图 ……195
9.1 依赖关系 ……195
9.2 基本块依赖DAG ……196
9.3 循环中的依赖关系 ……200
9.4 依赖关系测试 ……204
9.5 程序依赖图 ……207
9.6 动态分配的对象之间的依赖关系 ……209
9.7 小结 ……210
9.8 进一步阅读 ……211
9.9 练习 ……211
第10章 别名分析 ……213
10.1 各种现实程序设计语言中的别名 ……215
10.1.1 Fortran 77中的别名 ……216
10.1.2 Pascal中的别名 ……216
10.1.3 C中的别名 ……217
10.1.4 Fortran 90中的别名 ……218
10.2 别名收集器 ……218
10.3 别名传播器 ……222
10.4 小结 ……227
10.5 进一步阅读 ……227
10.6 练习 ……228
第11章 优化简介 ……229
11.1 第12~18章讨论的全局优化 ……230
11.2 流敏感性和可能与一定信息 ……231
11.3 各种优化的重要性 ……231
11.4 优化的顺序与重复 ……232
11.5 进一步阅读 ……235
11.6 练习 ……235
第12章 前期优化 ……237
12.1 常数表达式计算（常数折叠） ……237
12.2 聚合量标量替代 ……238
12.3 代数化简和重结合 ……240
12.3.1 地址表达式的代数化简和重结合 ……241
12.3.2 对浮点表达式应用代数化简 ……246
12.4 值编号 ……247
12.4.1 作用于基本块的值编号 ……247
12.4.2 全局值编号 ……251
12.5 复写传播 ……256
12.6 稀有条件常数传播 ……261
12.7 小结 ……267
12.8 进一步阅读 ……269
12.9 练习 ……269
第13章 冗余删除 ……271
13.1 公共子表达式删除 ……271
13.1.1 局部公共子表达式删除 ……272
13.1.2 全局公共子表达式删除 ……276
13.1.3 向前替代 ……284
13.2 循环不变代码外提 ……284
13.3 部分冗余删除 ……292
13.4 冗余删除和重结合 ……298
13.5 代码提升 ……299
13.6 小结 ……302
13.7 进一步阅读 ……302
13.8 练习 ……304
第14章 循环优化 ……305
14.1 归纳变量优化 ……305
14.1.1 识别归纳变量 ……306
14.1.2 强度削弱 ……312
14.1.3 活跃变量分析 ……319
14.1.4 归纳变量删除和线性函数测试替换 ……320
14.2 不必要边界检查的消除 ……325
14.3 小结 ……327
14.4 进一步阅读 ……329
14.5 练习 ……329
第15章 过程优化 ……331
15.1 尾调用优化和尾递归删除 ……331
15.2 过程集成 ……334
15.3 内嵌扩展 ……337
15.4 叶例程优化和收缩包装 ……338
15.4.1 叶例程优化 ……338
15.4.2 收缩包装 ……339
15.5 小结 ……341
15.6 进一步阅读 ……343
15.7 练习 ……343
第16章 寄存器分配 ……345
16.1 寄存器分配和指派 ……345
16.2 局部方法 ……346
16.3 图着色 ……347
16.3.1 图着色寄存器分配概述 ……347
16.3.2 顶层结构 ……349
16.3.3 网，可分配对象 ……350

超星公司制作 请尊重作者版权

bbs.theithome.com

16.3.4 冲突图 ……354
16.3.5 冲突图的表示 ……355
16.3.6 寄存器合并 ……358
16.3.7 计算溢出代价 ……359
16.3.8 修剪冲突图 ……361
16.3.9 指派寄存器 ……363
16.3.10 溢出符号寄存器……365
16.3.11 图着色寄存器分配的两个例子 …367
16.3.12 其他问题 ……375
16.4 基于优先级的图着色……376
16.5 其他寄存器分配方法……377
16.6 小结 ……377
16.7 进一步阅读 ……378
16.8 练习 ……380
第17章 代码调度 ……381
17.1 指令调度 ……381
17.1.1 分支调度 ……382
17.1.2 表调度……385
17.1.3 自动生成指令调度器 ……390
17.1.4 超标量实现有关的调度 ……390
17.1.5 基本块调度中的其他问题 ……390
17.1.6 跨基本块边界的调度 ……392
17.2 前瞻取和上推 ……392
17.3 前瞻调度 ……393
17.4 软流水 ……393
17.4.1 窗口调度 ……395
17.4.2 展开-压实软流水 ……397
17.4.3 循环展开 ……400
17.4.4 变量扩张 ……403
17.4.5 寄存器重命名 ……404
17.4.6 软流水的其他方法 ……407
17.4.7 层次归约 ……407
17.5 踪迹调度 ……408
17.6 渗透调度 ……409
17.7 小结 ……411
17.8 进一步阅读 ……413
17.9 练习 ……413
第18章 控制流和低级优化 ……415
18.1 不可到达代码的删除……415
18.2 伸直化 ……417
18.3 if化简 ……419
18.4 循环化简 ……420
18.5 循环倒置 ……421
18.6 无开关化 ……422
18.7 分支优化 ……422
18.8 尾融合或交叉转移 ……423
18.9 条件传送 ……424
18.10 死代码删除 ……425
18.11 分支预测……429
18.12 机器方言和指令归并 ……430
18.13 小结 ……433
18.14 进一步阅读 ……433
18.15 练习 ……435
第19章 过程间分析与优化 ……437
19.1 过程间控制流分析: 调用图 ……438
19.2 过程间数据流分析 ……445
19.2.1 流不敏感副作用分析 ……445
19.2.2 流敏感副作用: 程序概要图 ……455
19.2.3 副作用计算中的其他问题 ……458
19.3 过程间常数传播 ……458
19.4 过程间别名分析 ……461
19.4.1 流不敏感别名分析 ……462
19.4.2 传值和传指针语言的过程间别名分析 ……471
19.5 过程间优化 ……473
19.6 过程间寄存器分配 ……475
19.6.1 连接时的寄存器分配 ……475
19.6.2 编译时的过程间寄存器分配 ……477
19.7 全局引用的聚合 ……477
19.8 过程间程序管理中的其他主题 ……478
19.9 小结 ……478
19.10 进一步阅读 ……480
19.11 练习 ……480
第20章 存储层次优化 ……483
20.1 数据和指令高速缓存的影响 ……484
20.2 指令高速缓存优化 ……485
20.2.1 利用硬件辅助: 指令预取 ……485
20.2.2 过程排序 ……485
20.2.3 过程和基本块的放置 ……489
20.2.4 过程内的代码安置 ……489
20.2.5 过程分裂 ……492
20.2.6 过程内和过程间方法的结合 ……492

版权所有 侵权必究

bbs.theithome.com

20.3 数组元素的标量替换 ……493
20.4 数据高速缓存优化 ……496
20.4.1 过程间的数据安排 ……497
20.4.2 循环转换 ……498
20.4.3 局部性与循环铺砌 ……502
20.4.4 利用硬件辅助：数据预取 ……504
20.5 标量优化与面向存储器的优化 ……505
20.6 小结 ……506
20.7 进一步阅读 ……508
20.8 练习 ……508
第21章 编译器实例分析与未来的发展趋势 ……509
21.1 Sun用于SPARC的编译器 ……510
21.1.1 SPARC体系结构 ……510
21.1.2 Sun SPARC编译器 ……511
21.2 IBM POWER和PowerPC体系结构的XL编译器 ……517
21.2.1 POWER和PowerPC体系结构 ……517
21.2.2 XL编译器 ……518
21.3 DEC用于Alpha的编译器 ……524
21.3.1 Alpha体系结构 ……524
21.3.2 Alpha的GEM编译器 ……525
21.4 Intel 386体系结构上的Intel参考编译器 ……530
21.4.1 Intel 386体系结构 ……530
21.4.2 Intel编译器 ……531
21.5 小结 ……538
21.6 编译器设计和实现未来的趋势 ……539
21.7 进一步阅读 ……539
附录A 本书使用的汇编语言指南 ……541
附录B 集合、序列、树、DAG和函数的表示 ……549
附录C 软件资源 ……557
参考文献 ……561
索引 ……579

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第1章　高级主题介绍

我们首先回顾编译器的结构，然后给出本书后续各章讨论编译器的设计和实现高级主题所需要的基础知识。具体地，我们首先回顾与编译器结构有关的基本知识，概述第3章到第6章中关于符号表的结构与访问、中间代码的形式、运行时的表示以及代码生成器的自动生成等高级论题。然后，我们论述优化对于提高代码运行速度的重要性，介绍优化编译器的几种可能的结构，以及激进型优化编译器中各种优化的组成。接下来是关于本书各章的阅读流程。后面三节列出了本书没有包含的相关主题、例子中所用的目标机和数的表示，以及我们用于表示各种数据大小的名字。最后是小结、进一步阅读的文献以及除了最后一章之外每一章都有的练习。

1.1　编译器结构回顾

严格地说，编译器是一个将高级语言编写的程序转换成能在一台计算机上执行的等价目标代码或机器语言程序的软件系统。因此，一个特定的编译器可以在一台IBM兼容的个人计算机上运行，并将Fortran 77编写的程序转换为能在PC上运行的Intel 386体系结构的目标代码。这个定义可以扩展到包含将一种高级语言程序转换成另一种高级语言程序的系统，从一种机器语言程序转换成另一种机器语言程序的系统，从一种高级语言程序转换成一种中间语言程序的系统，等等。例如，在这种广义定义下，我们可能有一个在Apple Macintosh机上运行的编译器，它将Motorola M68000 Macintosh的目标代码转换成能在基于PowerPC微处理器的Macintosh机上运行的PowerPC的目标代码。

在狭义的定义下，一个编译器由一系列的阶段组成，这些阶段从要编译的源程序的字符序列开始，依次对一个给定形式的程序进行分析，并产生一种新的形式，在大多数情况下最终产生一个可以与其他目标代码链接，并装入一台机器的存储器中执行的可重定位目标模块。如同任意一本编译器基础教科书所述，这一编译过程至少有如图1-1所示的4个阶段，这些阶段分别是:

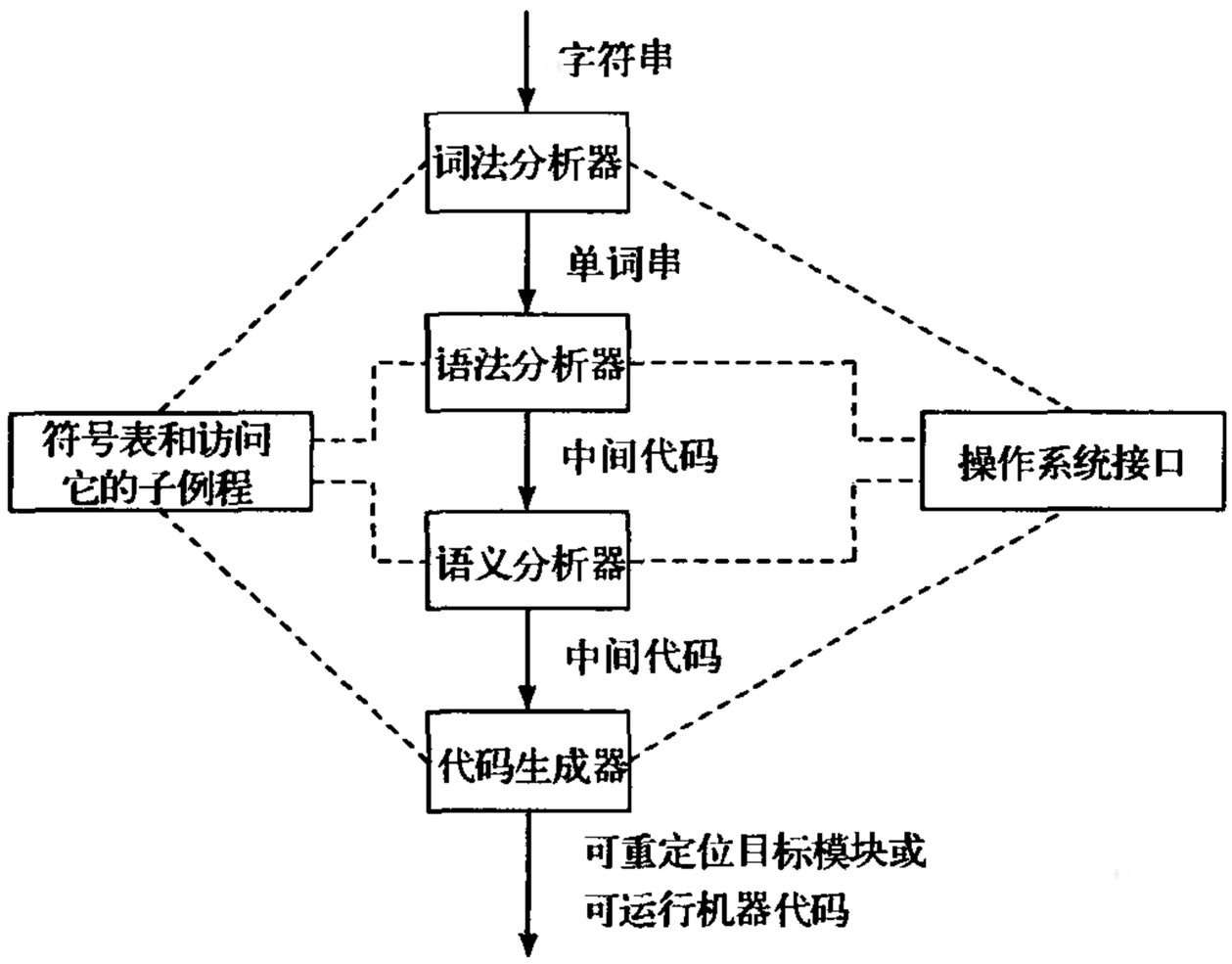


图1-1　一个简单编译器的高层结构

bbs.theithome.com

1. 词法分析(lexical analysis)，它分析提交的字符串，将其划分为一组符合编程语言词法规范的单词（如果字符串不能转换为一串合法的单词串，将报告错误信息）。

2. 语法分析(syntactic analysis or parsing)，它处理单词串，产生中间表示，如语法分析树或连续的中间代码（作为一个例子，参见4.6.1节中MIR的定义），以及符号表。符号表记录程序中所使用的标识符和它们的属性（如果单词串中有语法错误，将报告错误信息）。

3. 静态语义检查(static-semantic validity or semantic checking)，它以中间代码和符号表为输入，确认程序是否满足源语言要求的静态语义属性，也就是说，标识符的声明和使用是否一致
2 （如果程序在语义上不一致或在其他某一方面不满足源语言定义的要求，将报告错误信息）。

4. 代码生成(code generation)，它将中间代码转换成可重定位目标模块或可直接运行的目标代码形式的等价机器代码。

检测出来的错误可能是警告性的或致命性的，在后一种情况下，编译过程可能终止。

除了这4个阶段，编译器还包含一张符号表和访问该表的若干例程，以及与操作系统和用户环境的接口（用来读写文件、读取用户的输入、向用户输出信息，等等），它们在编译过程的所有阶段都能使用（如图1-1所示）。在构造与操作系统和用户环境的接口时采取一定措施，就可以支持编译器在不同的操作系统上运行，而不用改变编译器与操作系统之间的接口。除了第21章外，下面编译器的结构图中将不再包括符号表或操作系统接口。

对于许多高级语言而言，编译器的4个阶段可以组合成一遍，从而构成一个快速的一遍编译器。这样的编译器对于要求不高的用户是完全可以满足的，它也可作为软件开发环境中进行增量编译的一个选择，目的是在编辑－编译－调试循环中，对程序的修改能快速地进行编译和调试。然而，通常不能指望这样的编译器产生非常高效的代码。

另一种选择是将词法分析和语法分析阶段组合成第一遍，产生符号表和某种形式的中间代码[⊖]。语义检查和由中间代码生成目标代码作为第二遍，或各自作为独立的一遍（也可以在第一遍中完成大部分的语义检查）。编译器生成的目标代码可以是可重定位的目标机器代码或汇编语言，汇编语言就需要再用汇编器来生成可重定位目标代码。一个程序或一部分程序被编译后，通常需要进行链接，使它们与相关的库例程连接起来，再由装载程序将其读入至存储器中并重定位生成可运行的映像。链接可以在运行前进行（静态地），也可在运行中进行（动态地），或分两阶段进行，即静态链接用户程序的各部分，动态链接系统的库例程。

1.2 基本问题中的高级论题

本节就第3章到第6章讨论的符号表设计和访问、中间代码设计、运行时表示和代码生成器的自动生成等高级论题作一个简介。这些论题中的基本问题通常在编译器设计和实现的早期课程中就重点讨论过。然而，当考虑支持一个其风格不如教科书所给那么规整的语言时，这些问
3 题就变得复杂起来了。

符号表的设计一直以来在很大程度上说它是一种玄妙的艺术比说它是科学原理更合适。这样说的部分理由是因为符号表的属性随语言的不同而变化，也因为全局符号表最重要的方面是它的接口和性能。对于不同的基础数据结构，如栈、各种树、散列表等，组织一个能快速查填的符号表的方法也各不相同，这些方法都很值得介绍。我们在第3章将介绍其中的一

⊖ 某些语言，例如Fortran，为了正确地分析程序，要求词法分析和语法分析协同进行。例如，对于一行Fortran代码“do 32 i=1”，不进行超前搜索，就不能确定等号之前的字符是3个单词还是一个单词。如果1之后是一个逗号，则等号前是3个单词，这个语句是一个循环的开始。如果1是这一行的结束，则等号之前是一个单词，它可以写作“do32i”，这个语句是一个赋值语句。

bbs.theithome.com

超星公司制作 请尊重作者版权

些方法，但重点介绍一种栈、散列和链表组合的方法，这种方法能够通过扩展Fortran、C和Pascal等语言的简单栈模式，满足Ada、Mesa、Modula-2和C++等语言可导入和导出作用域的要求。

中间代码的设计在相当大的程度上也是一种技巧，而不是科学。有人说有多少种编译器套件，就有多少种中间语言设计，但这种估计可能比实际的少了一半左右。事实上，由于许多编译器使用两种以上不同的中间代码，中间代码设计方案数可能两倍于编译器套件种类数!

因此第4章研究中间代码设计中遇到的有关问题，讨论各种设计方案的优缺点。为了便于进一步讨论作用于中间代码的各种优化算法，我们最终必须选择一种或几种中间代码作为具体的例子。我们选用如下4种中间代码：HIR，一种高级中间表示，它保留了循环结构、循环上下界和数组下标，用于与数据高速缓存相关的优化；MIR，一种中级中间表示，本书在大多数情况下使用这种中间语言；LIR，一种低级中间表示，用于那些必须涉及真实机器特征的优化，但用于全局寄存器分配时有一点变化，即它涉及的是符号寄存器，而不是真实的机器寄存器；SSA形式，它可以看作是增加了一个额外操作的MIR中间语言版本，这个操作即ϕ函数，我们几乎总是只在流程图中使用它，而不在顺序程序中使用它。图1-2给出了一个HIR循环、它的MIR形式，以及它使用符号寄存器的LIR形式。

```
for v ← v1 by v2 to v3 do
   a[i] ← 2
endfor
```

a)

```
    v ← v1
    t2 ← v2
    t3 ← v3
L1: if v > t3 goto L2
    t4 ← addr a
    t5 ← 4 * i
    t6 ← t4 + t5
    *t6 ← 2
    v ← v + t2
    goto L1
L2:
```

b)

```
        s2 ← s1
        s4 ← s3
        s6 ← s5
    L1: if s2 > s6 goto L2
        s7 ← addr a
        s8 ← 4 * s9
        s10 ← s7 + s8
        [s10] ← 2
        s2 ← s2 + s4
        goto L1
    L2:
```

c)

图1-2 用a）HIR、b）MIR和c）使用符号寄存器的LIR表示的一个代码段（假设v2的值为正）

图1-3给出了该循环的SSA形式，除了被转换为流程图形式，并且符号寄存器s2被分为3个，即$s2_1$、$s2_2$、$s2_3$，而且在基本块B2的开始处，即$s2_1$和$s2_3$的汇合点增加了一个ϕ函数之外，它的代码与图1-2c相同。SSA的主要优点是使以前用于基本块或扩展基本块[⊖]的几种优化也能很好地用于整个过程，这常常能显著地改善程序的性能。

另一种对好几种优化非常有用的中间代码形式，即程序依赖图，将在9.5节讨论。

第5章集中讨论构造程序执行所处的运行时环境的问题。运行时环境同代码生成一起，是保证正确和高效地实现一个语言的少数几个关键环节之一。如果所设计的运行时模式不支持语言中的某些操作，这个设计显然就是失败的。如果能够支持语言的运行，但比需要的要慢，就会对程序的性能产生较大的影响，而这种影响一般不能通过程序优化来弥补。运行时环境的基本问题包括源程序中的数据类型的表示、寄存器的分配和使用、运行时栈空间的布局、对非局

⊖ 扩展基本块是一棵由基本块组成的树，这棵树只能从根结点进入、从某个叶结点离开。

bbs.theithome.com

部符号的访问，以及过程的调用、入口、出口和返回。另外，我们也将讨论运行时模式为支持位置无关代码所需的扩展，以及如何运用这些扩展来实现共享代码对象运行时的高效动态链接。对于编译动态和多态语言时必须考虑的一些额外问题，例如递增地改变运行代码，堆存储管理和运行时类型检查（以及运行时类型检查的优化，或在可能时类型检查的消除），我们给出了一个综述。

最后，第6章考虑根据机器描述自动产生代码生成器的方法。这一章详细描述了一种技术，即语法制导的Graham-Glanville方法，并简要介绍了另外两种方法，即Ganapathi和Fischer的语义制导分析方法以及Aho、Ganapathi和Tjiang的twig方法。twig方法采用树模式匹配和动态规划。与手工编写的代码生成器相比，这些方法的好处在于它们使得程序员能在较高层次上思考，并且能方便地随时修改代码生成器，不论是改错、改进，还是使其适应新的目标机。

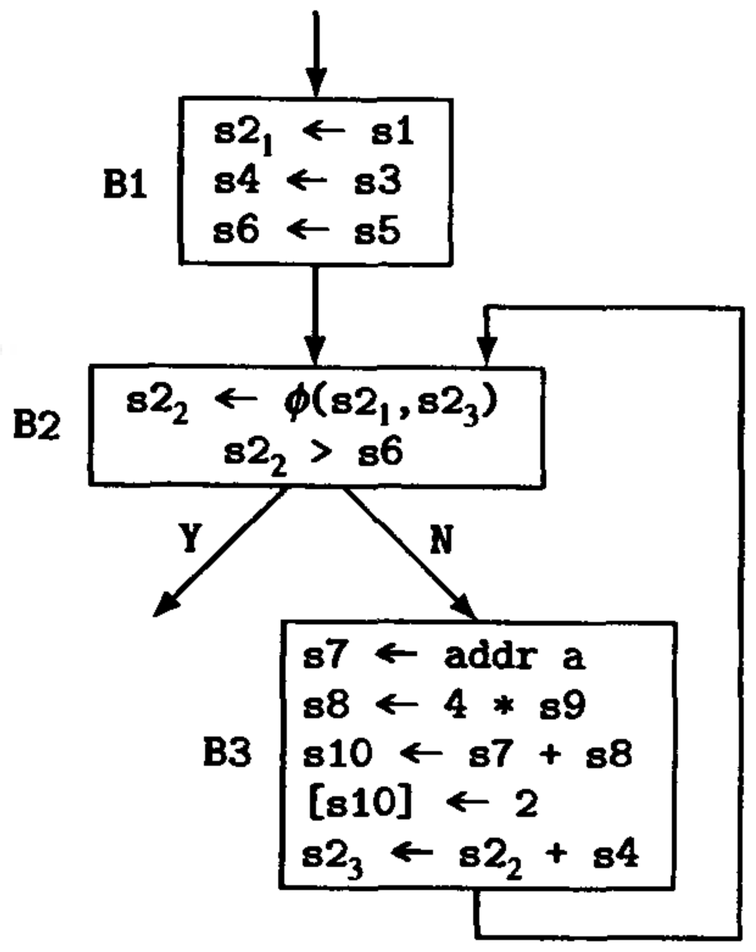


图1-3 图1-2中例子的SSA形式。注意，s2被分为$s2_1$、$s2_2$和$s2_3$三个变量，在基本块B2开始处增加了ϕ函数

1.3 代码优化的重要性

通常，使用如1.1节描述的一遍编译器生成的目标代码效率较低，如果进行更多的编译，效率就会有较大的提高。例如，一遍编译器可能逐条表达式地顺序生成目标代码，使得图1-4a的C语言代码段可能生成如图1-4b所示的SPARC汇编代码[⊖]。如果该编译器对这段代码进行优化，包括将变量分配到寄存器中，就可以得到如图1-4c所示更高效的代码。即使不将变量分配到寄存器中，至少也可以避免多余地取c的值。在早期典型的单标量SPARC实现中，执行图1-4b中的代码需要10个时钟周期，而图1-4c中的代码仅需要2个时钟周期。

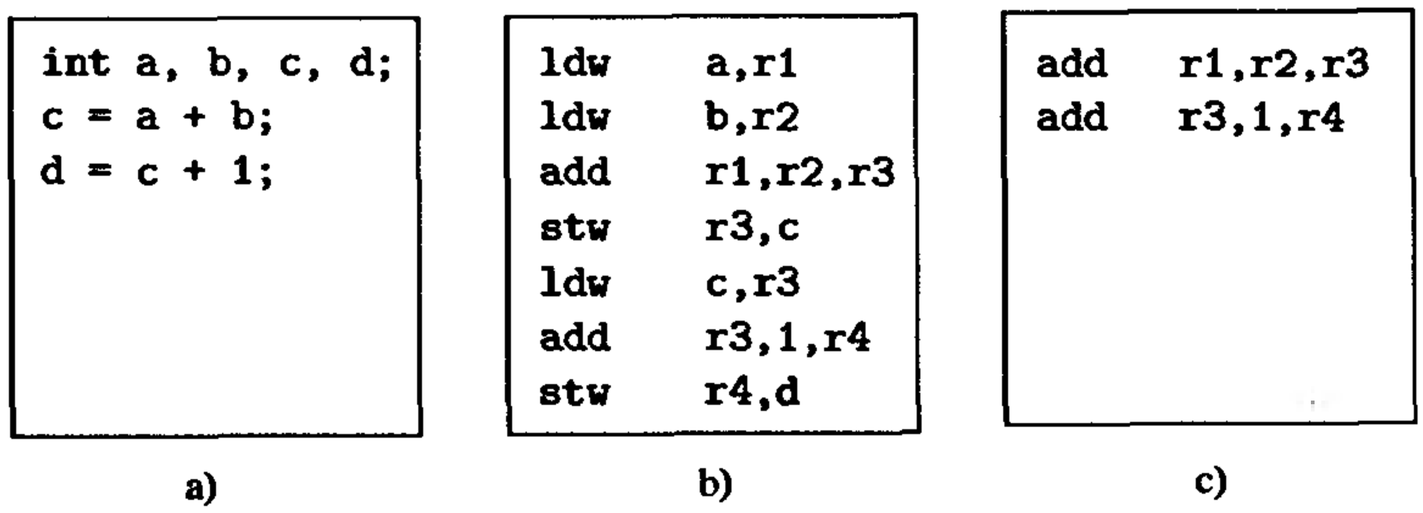


图1-4 a) 一个C代码段，b) 生成的原始SPARC代码，c) 优化的代码

一般而言，最重要的优化是与循环有关的优化（例如循环不变计算外提、简化或消除归纳变量计算）、全局寄存器分配和指令调度，所有这些都在第12~20章进行了讨论（同时也讨论了许多其他的优化技术）。

然而，有许多优化技术的作用只对特定的程序才显示出来，还有一些优化技术的作用随着程序结构和细节的不同而变化。

例如，对一个高度递归的程序施行尾调用优化可能很有效果（见15.1节），这种优化将递归转换为循环，再经过循环优化就提高了程序的性能。另一方面，对于一个循环次数不多，而

⊖ SPARC汇编语言指南请查阅附录A。

bbs.theithome.com

超星公司制作 请尊重作者版权

循环中的基本块非常大的程序，循环分布（它将一个循环分裂成多个，每个新的循环完成原循环的部分工作）或寄存器分配优化可能非常见效，但其他循环优化对这种程序可能见效不大。类似地，过程集成或内联，即用子程序体置换子程序调用，不但减少了子程序调用的开销，而且使过程内优化能够施加于子程序体，从而得到在没有内联或不进行（开销很大的）过程间分析和优化（见第19章）的情况下不可能得到的显著改善。但是，内联通常增大了程序的体积，这可能对程序的性能有负作用，即增加了高速缓存的不命中率。因此，对于每一个程序，应当评估编译器提供的优化选项的效果，选用那些能够实现程序最佳性能的优化。

5
~
6

上述优化和其他一些优化能使程序的性能产生很大的变化——程序的执行速度常常能提高1~2倍，在某些情况下甚至更高。

对于大型软件项目，包括编译器，一个重要的设计原则是用较小的、功能不同的模块构成大的程序，每个模块尽量简单，以方便设计、编码、理解和维护。这样，用不进行优化的编译器生成类似图1-4b的局部代码是完全可能的，但要产生图1-4c中更高效的代码就必须进行优化。

1.4 优化编译器的结构

一个能生成高效目标代码的编译器包含优化器组件。优化编译器的结构主要有两种模式，如图1-5a和b所示[⊖]。在图1-5a中，源代码被转换成类似LIR（见4.6.3节）的低级中间代码，所有优化都施加于这种低级形式的中间代码，我们称这为优化的低级模式（low-level model）。在图1-5b中，源代码被转换成类似MIR（见4.6.1节）的中级中间代码。对中级中间代码进行的优化主要是体系结构无关的优化，然后中级代码被转换成低级代码，再进一步优化，主要是体系结构相关的优化，我们称此为优化的混合模式（mixed model）。在两种模式中，优化器的各阶段对中间代码进行分析和转换，以消除无用代码和提高任务的执行速度。例如，优化器可能确认循环中的一个计算在每一个迭代中都产生同样的结果，因此将这个计算移到循环外就可提高程序的执行速度。在混合模式中，由所谓的后遍（postpass）优化器来执行低级优化，例如利用目标机特有的指令和寻址模式，而在低级模式中这是由单一的优化器来完成的。

7

混合模式的优化器有可能能更好地适应新的体系结构，且编译效率可能更高。而低级模式的优化器可能会难以移植到另一种体系结构，除非第二种体系结构与第一种非常相似，例如，第二种是第一种向上兼容的扩展。选择混合模式还是低级模式主要取决于投资和开发所关注的重点。

采用混合模式的编译器有Sun 微系统公司SPARC上的编译器（见21.1节）、数字设备公司（DEC）支持Alpha的编译器（见21.3节）、Intel支持386体系结构系列的编译器（见21.4节）和Silicon Graphics支持MIPS的编译器。采用低级模式的编译器有IBM支持POWER和PowerPC的编译器（见21.2节），以及Hewlett-Packard支持PA-RISC的编译器。

低级模式能较好地避免优化顺序产生的问题，能使整个优化器都看到所有的地址计算。根据这些还有其他一些理由，我们建议在从头开始构建一个优化器时就采用低级模式，除非日后希望把它移植到另一种显著不同的体系结构上。但无论怎样，本书中描述的优化既可以运用于中级中间代码，也可如同运用于中级中间代码一样运用于低级中间代码。经过简单的改编，这些优化就能适用于低级中间代码。

如前面所述，Sun和Hewlett-Packard的编译器代表了与低级模式相反的做法。Sun的全局优化器最初是为基于Motorola MC68010微处理器的Sun-2系列工作站的Fortran 77编译器编写的，当时就确信它将要移植到未来的体系结构上。后来它被移植到使用相同中间表示的其他几个编

⊖ 也可将词法分析、语法分析、语义分析和其他转换或中间代码生成放在一个步骤中。

bbs.theithome.com

译器中，然后又被移植到十分类似的基于MC68020微处理器的Sun-3系列，最近又移植到了SPARC和SPARC-V9。虽然花费了相当大的投资来使该优化器能对SPARC进行非常有效的优化，特别是将代码生成前的一些优化转移到代码生成后进行，但该优化器针对新的体系结构的移植大部分还是比较容易的。

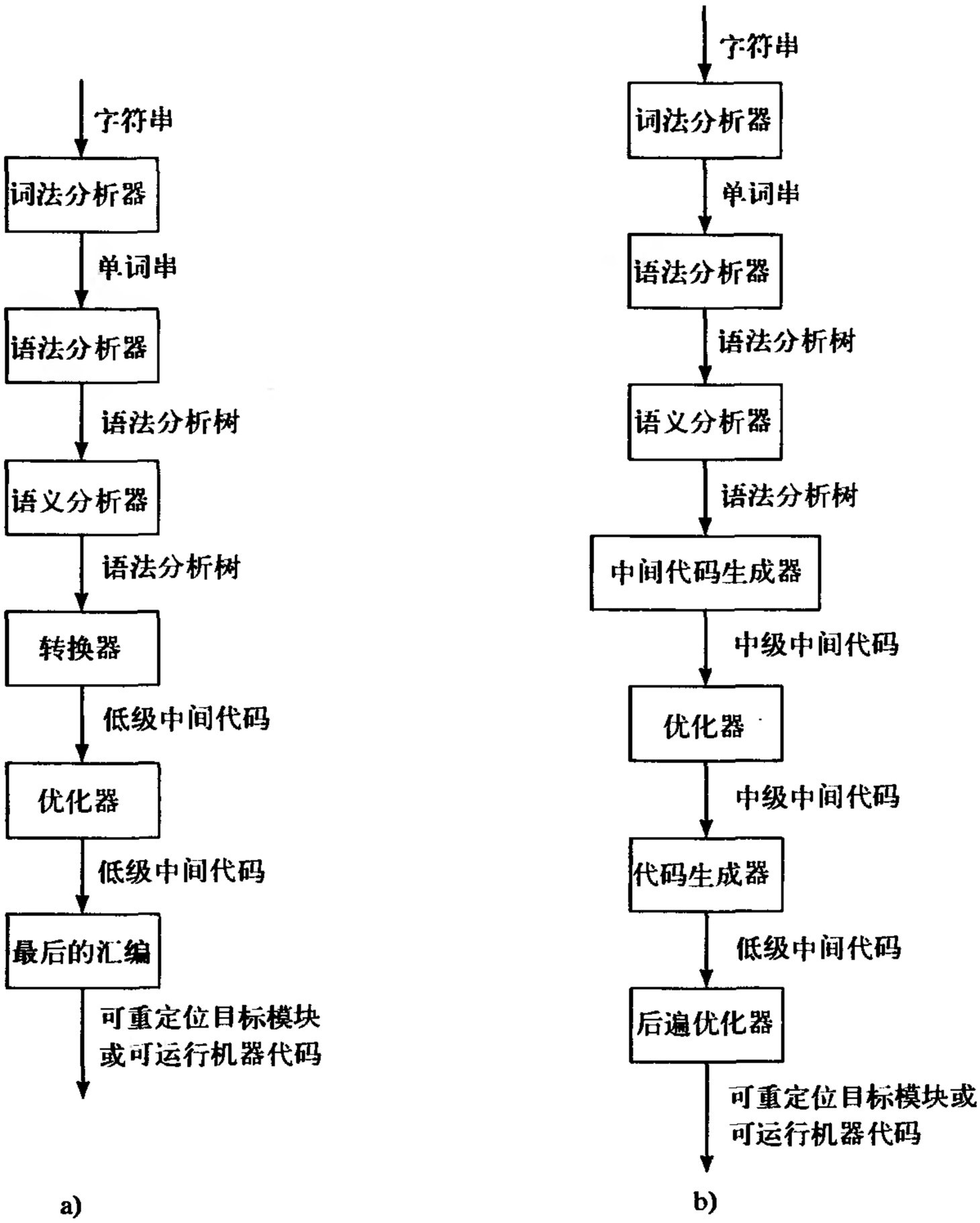


图1-5 优化编译器的两种高层结构：a) 低级模式，所有优化都在低级中间代码上完成；b) 混合模式，优化分成两个阶段，一个阶段的优化在中级中间代码上进行，另一个阶段的优化在低级中间代码上进行

另一方面，Hewlett-Packard支持PA-RISC的全局优化器是一项大投资的一部分，这项投资的目的是：围绕一种新的体系结构统一该公司的大多数计算机产品。采用单一的优化器和统一各方力量的好处使得公司有充分的信心专门为PA-RISC设计一个全局优化器。

除非一个体系结构只准备用于特殊的用途，如作为嵌入式处理器，否则不能只支持一种程序设计语言。这使得人们希望一种体系结构能够尽可能地共享多种编译器的组件，既减少编写和维护编译器的工作量，又能从改善被编译代码性能方面获得最广泛的效益。在这点上，无论是混合模式还是低级模式优化都是相同的。因此，第21章中讨论的所有实际的编译器都是特定体系结构的编译器套件中的成员。这些编译器套件中的各个编译器共享许多组件，包括代码生成器、优化器、汇编器以及其他可能的组件，但它们具有不同的前端来处理不同语言的词法、

bbs.theithome.com

超星公司制作 请尊重作者版权

语法和静态语义。

另一种情况是一个软件厂家为多种体系结构提供同一种语言的编译器。这时我们就能看到，这些编译器使用相同的前端，优化器组件通常也是相同的，但代码生成器是不同的，也可能还有另外的优化阶段来处理每种体系结构的特征。在这种情况下采用混合模式优化更为合适。代码生成器的结构常常是相同的，其结构采用一种既适合于源语言，或更多地适合于相同中间代码的方式，而与具体的目标机无关，不同的只是为每一种目标机生成的指令不同。

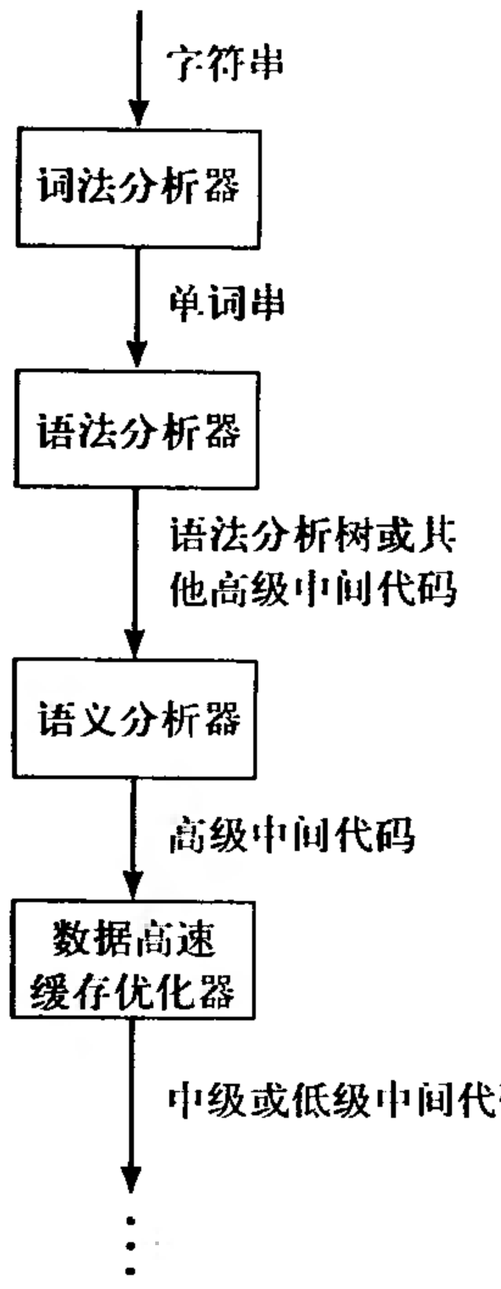


图1-6 增加数据高速缓存优化至优化编译器中。最后的箭头在图1-5a低级模式中指向转换器，在图1-5b混合模式中指向中间代码生成器

还有一种选择是采用预处理器(preprocessor)将一种语言编写的程序转换为另一种语言的等价程序，然后进行编译。C++ 的早期实现就是用一个称为cfront的程序将C++ 代码转换为C代码，在这个过程中（除了进行其他转换外）进行一种称为*名字重塑*(name mangling)的转换，将易读的C++ 标识符转换为不可读、但能编译的C标识符。另一个例子是用预处理器将Fortran程序转换为另一种能更好地发挥向量或多处理机系统性能的程序。第三个例子是将一个还没有研制成功的处理器的目标代码转换为一个已有机器的代码，以实现对预期系统的模拟。

一个在有关优化器结构和它在编译器或编译器套件中的位置的讨论中被忽略的问题是：有些优化，特别是在20.4节中将讨论的数据高速缓存的优化，施加于源语言或高级中间语言（例如HIR，见4.6.2节）时更为有效。这种优化可以作为优化过程的第一步，如图1-6所示，图中最后的箭头在低级模式中指向转换器，在混合模式中指向中间代码生成器。IBM支持POWER和PowerPC的编译器使用的是另一种方法，它首先将源程序转换为一种低级代码（称为XIL），然后再由它产生一种高级表示（称为YIL）来进行数据高速缓存优化。经过数据高速缓存优化后，YIL代码再转换回到XIL代码。

8
~
10

1.5 激进型优化编译器中各种优化的位置

上一节讨论了优化在整个编译过程中的位置。在以后关于优化的各章的小结中，都有一个类似于图1-7的图，它说明了本书中讨论的几乎所有优化在激进型优化编译器中的一个比较合理的执行顺序。注意，用“激进型”这个词是因为我们假定其目的是在不破坏正确性的前提下尽可能地优化性能。在每一章相应的图中，所讨论的优化都用黑体字标示出来。另外要说明的是，这些图中只包含了各种优化，没有包含编译的其他阶段。

图1-7中左边的字母说明它右边的优化通常施加于何种类型的代码，如下所述：

- **A** 这些优化通常施加于源代码，或施加于基本按其源代码形式保留了循环结构和顺序、数组访问的高级中间代码。在执行这类优化的编译器中，这些优化通常在编译过程的前期进行，因为随着编译过程从上一遍进行到下一遍，代码的层次也趋于越来越低。

bbs.theithome.com

超星公司制作 请尊重作者版权

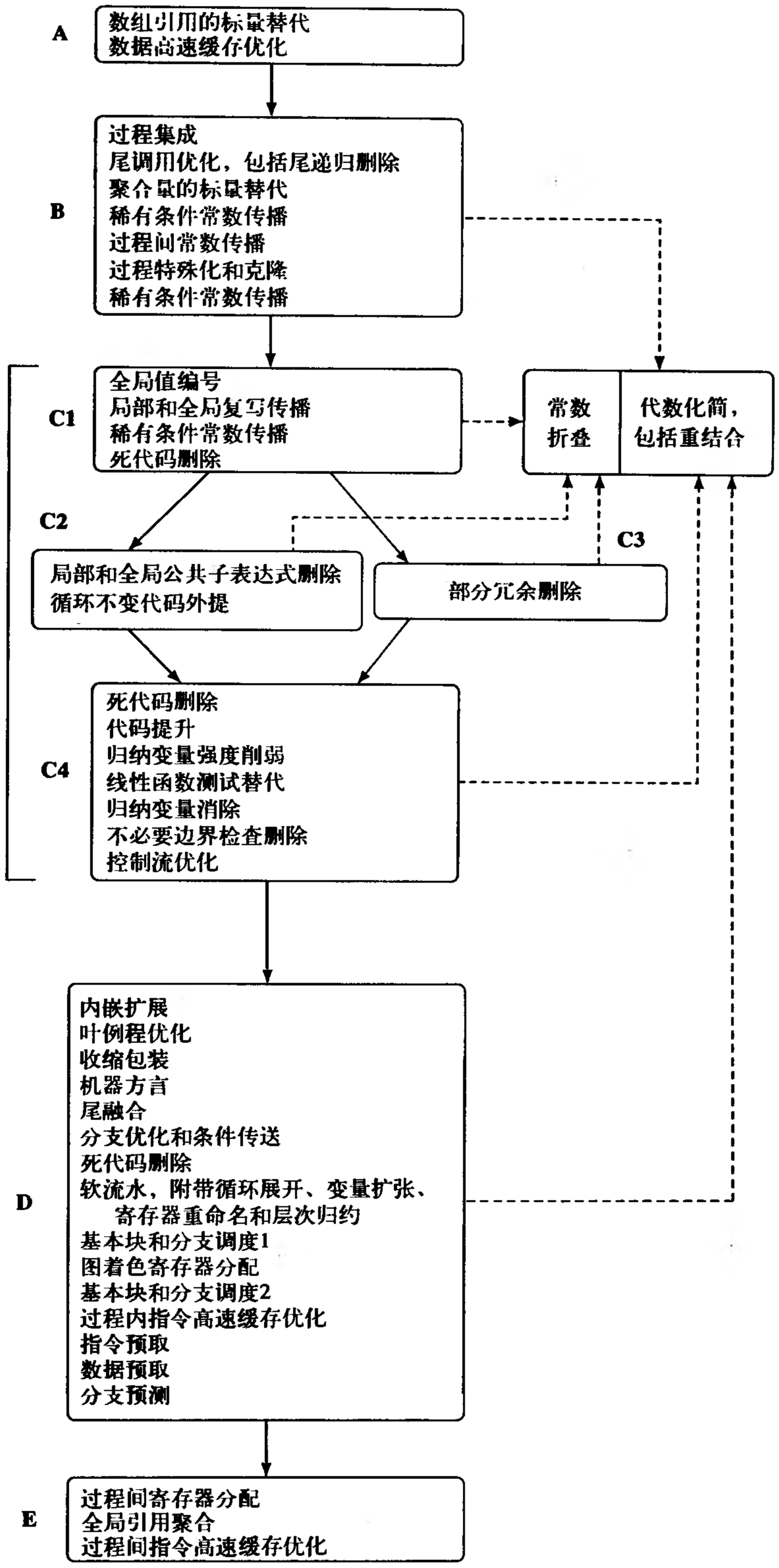


图1-7 各种优化的顺序

bbs.theithome.com

超星公司制作 请尊重作者版权

- B,C　这些优化通常施加于中级或低级中间代码，具体取决于所采用的是混合模式还是低级模式。
- D　这些优化几乎总是施加于低级形式的代码——这种形式可能在很大程度上是机器相关的（例如一种结构化的汇编语言），或者稍微通用一些，像本书中的LIR代码——因为这些优化要求地址被转换成“基址寄存器＋偏移”的形式（或类似形式，取决于目标处理器提供的有效寻址模式），也因为这些优化中有一些需要有低级控制流代码。
- E　这些优化在链接时进行，所以它们施加于可重定位的目标代码。这一方面一个令人感兴趣的项目是Srivastava和Wall的OM系统，它是一种在链接时进行所有优化的编译系统，也是这一领域中的一个开创性研究。

图1-7中的方框除了表示相应优化适合的代码层次外，还表示了各种优化之间的整体流程。例如，常数折叠和代数化简所在的方框由一根虚线箭头与其他优化阶段相连，因为最好是将它们构造为子程序，以便可以在任何需要的地方调用。

从C1到C2或C3的分支表示系统在进行基本相同的优化（即，在不改变程序语义的情况下，将计算移到执行次数较少的位置）时对不同方法的选择。这个分支也表示了对优化所使用的不同数据流分析方法的选择。

方框内的详细流程比方框间的流程更为自由。例如，方框B中，在稀有条件常数传播之后进行聚合量的标量替代使系统能决定标量替代是否值得进行，而在常数传播之前进行标量替代将使得常数传播更有效。前一种情况的例子如图1-8a所示，后一种如图1-8b所示。在a)中，通过将赋给a的值1传播给测试a=1，可以确定基本块B1将取分支Y。同样地，聚合量的标量替代导致第二遍常数传播能确定出基本块B4取分支Y。在b) 中，聚合量的标量替代使得稀有条件的常数传播确定了基本块B1取分支Y。

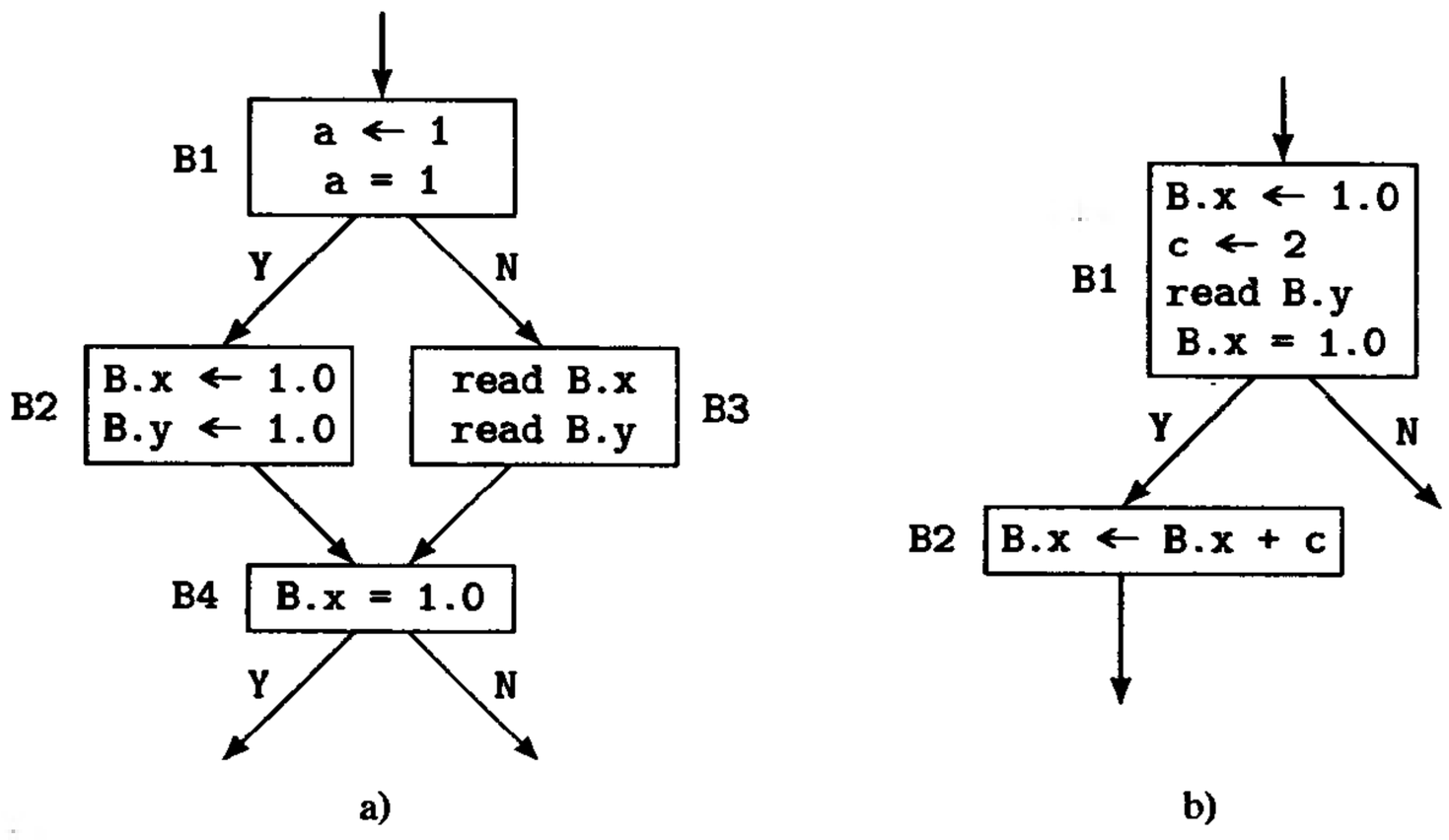


图1-8　a) 在常数传播之后和 b) 在常数传播之前进行聚合量标量替代的效果的例子

类似地，全局值编号、全局和局部的复写传播，以及稀有条件常数传播的某种顺序可能对某些程序很有效，而另一种顺序对其他一些程序更有效。

进一步研究图1-7后，我们建议进行3次稀有条件常数传播和死代码删除，做2次指令调度。这样做的理由在每种情况下略有不同:

1. 稀有条件常数传播发现每次使用时其值都是常数的操作数。在进行过程间常数传播之前进行稀有条件常数传播，有利于将其值为常数的参数传送给其他过程，而过程间常数又有利于发现更多的过程内的常数。

bbs.theithome.com

版权所有 侵权必究

2. 我们建议重复进行死代码消除，因为有几种优化和几组优化经常产生死代码，应该尽可能及时地消除这类代码，以便适当地减少其他编译阶段——如优化或其他任务，例如将代码转换为较低层次的形式——所要处理的代码量。

11 ~ 13

3. 建议在寄存器分配之前和之后都进行指令调度，因为第一遍指令调度可以利用代码中使用了很多符号寄存器（而不是只有几个实际寄存器）的相对自由度。而第二遍指令调度可以处理可能被寄存器分配插入的寄存器溢出和恢复代码。

最后要强调的是，实现图1-7的所有优化功能，将构成一个对于单处理机系统能生成高性能的代码，同时又是非常庞大的激进型编译器，但这还完全没有涉及支持并行或向量计算机的有关问题，如代码重组等。

1.6 本书各章的阅读流程

读者可以根据自己的知识基础、需要和其他因素，用不同的方法来阅读这本书。图1-9给出了几种可选的阅读本书的流程，如下所述。

1. 首先，我们建议读者阅读本章和第2章。它们对本书其他部分作了简要的介绍，定义了编写本书所有算法的ICAN语言。

2. 如果读者想阅读全书，我们建议按顺序阅读后续的各章。虽然也可以按其他次序来读，但这是我们写作本书各章时考虑的阅读次序。

14

3. 如果读者需要更多关于编译器设计和实现基础方面较先进的信息，但可以忽略其他方面，我们建议继续阅读第3章到第6章。

4. 如果读者主要关注优化，那么你的兴趣是针对存储器层次结构进行的数据相关的优化，同时还有其他类型的优化吗？

a) 如果是，继续阅读第7章到第10章，接着阅读第11章到第18章，以及第20章。

b) 如果不是，继续阅读第7章到第10章，接着阅读第11章到第18章。

5. 如果读者对过程间优化感兴趣，请阅读第19章，它囊括了过程间的控制流、数据流、别名和常数传播等分析，以及其他几种形式的过程间优化，特别是过程间寄存器分配。

6. 接着阅读第21章，它给出了Digital Equipment Corporation、IBM、Intel和Sun Microsystems的4种产品化编译器套件的描述，给出了关于其他中间代码的设计、优化的选择和进行优化的顺序，以及用于实现优化和编译过程中其他一些任务的技术的例子。当阅读其他章时，你也可能会希望参考第21章中的这些例子。

15

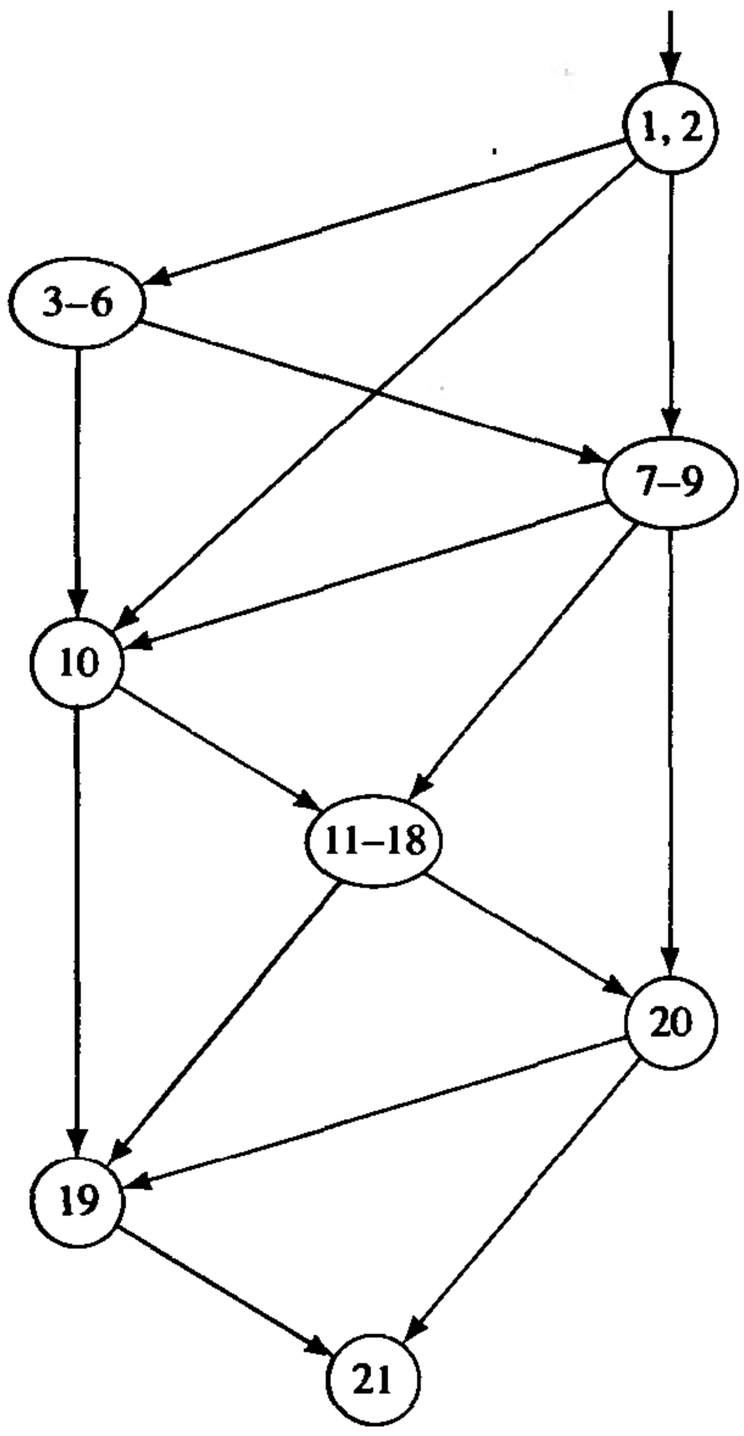


图1-9 本书各章的阅读流程

最后，3个附录给出了本书的支撑材料，包括本书中使用的汇编语言、数据结构的具体表示，以及可通过ftp和万维网访问的编译器项目的资源。

1.7 本书没有涉及的相关主题

本书原本还可以覆盖其他一系列的主题，但我们没有这样做。原因是多方面的，例如不能

bbs.theithome.com

增加本来就相当大了的本书的篇幅，有些问题只是与本书的主题稍微沾点边，或在其他书中已经很好地讨论过（1.11节给出的参考文献可作为研究这些问题的入门材料）。本书没有涉及的相关论题是：

1. 优化和调试的相互作用（interaction of optimization and debugging）是从1980年左右就开始研究的一个问题。前期研究的进展一直相当缓慢，但近期有了令人印象深刻的改变。Adl-Tabatabai和Gross及Wismüller的工作为这方面的入门提供了极好文献。

2. 并行化和向量化（parallelization and vectorization）以及它们与标量优化的关系在本书中没有讨论，因为讨论这些问题需要更大的篇幅，同时关于并行化和向量化已经有了几本很好的书，例如Wolfe、Banerjee、Zima和Chapman的书。然而，在第9章讨论的依赖关系分析和在20.4.2节讨论的循环转换技术是并行化和向量化的基础。

3. 剖面分析反馈式编译过程（profiling feedback to the compilation process）是一种非常重要的编译方式，本书也几次涉及了这个问题。Ball和Larus的著作，以及Wall的著作对这些技术、剖面分析器产生的结果给出了有关解释，并对它们在编译器中的应用作了很好的介绍，同时也给出了之前有关工作的参考文献。

1.8 例子中所用的目标机

本书中大多数目标机代码的例子是用SPARC汇编语言编写的。我们使用了一个没有寄存器窗口的简化版本。偶尔也有SPARC-V9的例子或其他汇编语言的例子，例如POWER或Intel 386体系结构系列的汇编语言等。为使读者理解本书的例子，附录A对本书涉及的所有汇编语言都作了相当好的描述。

1.9 数的表示与数据的大小

术语字节（byte）和字（word）通常用于表示一个字符的大小和特定系统中一个寄存器数据的大小。我们主要考虑8位为一个字节的32位系统，将64位系统作为32位系统的扩展，因此统一使用表1-1所示的术语来表示存储大小。

16

表1-1 数据类型的大小

术语	大小（位）
字节	8
半字	16
字	32
双字	64
四字	128

本书中几乎所有的数都是十进制表示法并按通常的方式来书写。我们偶尔也使用十六进制表示。一个十六进制的整数写法为0x之后跟着一串十六进制数字（0~9，a~f或A~F），且总是解释为无符号数（除非特别地说明了表示一个有符号的值），其长度是从最左边的第一个“1”数到最后一位的位数。例如，0x3a是十进制数58的6位表示，0xfffffffe是十进制数4294967294 $=2^{32}-2$的32位表示。

1.10 小结

本章集中回顾了编译器的一些基本问题，为进入后续的章节打下基础。

在讨论了第3章到第6章包含的内容之后——这些内容通常认为是基本问题中的一些高级方面，我们描述了优化的重要性，优化编译器的结构，包括混合和低级模式，以及在激进型优化编译器中各种优化的组织。

然后我们讨论了本书各章可能的阅读流程，在结束本章之前以两小节的篇幅提及了本书没有涉及的相关论题，以及例子中使用的目标机，数的表示和用于表示各种数据大小的名称。

bbs.theithome.com

超星公司制作 请尊重作者版权

从本章应当学到的5点主要启示是:

1. 在通常认为是编译器构造的基本问题中，有一些在设计和构建能实际使用的编译器时需要认真考虑的高级问题，例如作用域的导入和导出的处理、中间语言的设计或选择、位置无关代码和共享代码对象等;

17 2. 组织整个编译器和编译器中的优化组件存在着若干种合理的方式;

3. 划分优化过程有2种主要的模式（混合的和低级的），通常采用低级模式更好，但根据特定的编译器和优化项目情况的不同，可以有不同的选择;

4. 新的优化技术和对传统方法的改进不断出现;

5. 还有一些重要论题需要仔细研究，如优化代码的调试、标量优化与并行化和向量化的集成，但这些超出了本书的范围。

1.11 进一步阅读

关于程序设计语言的历史有2本非常优秀的著作，即[Wexe81]和[BerG95]，但关于编译器历史的出版物极少。Knuth在[Knut62]一书中给出了编译器开发早期的历史，一些近期的材料则包含在上面提过的2本著作中，即[Wexe81]和[BerG95]。

在近期关于编译器构造的著作中，较好的有[AhoS86]和[FisL91]。

在将C++ 程序转换为C程序的过程中所使用的名字重塑技术，[Stro88]中将它描述为“函数名编码方案”。

[GhoM86]描述了Sun的全局优化器的起源、结构和功能，而[JohM86]描述了Hewlett-Packard支持PA-RISC的全局优化器的结构和性能。

[SriW93]描述了Srivastava和Wall的OM系统。

关于Adl-Tabatabai和Gross在优化和调试方面的工作的入门参考文献是[AdlG93]。Wismüller的[Wism94]是关于这个领域另一分支工作的参考文献。

关于并行编译系统的著作有[Bane88]、[Bane93]、[Bane94]、[Wolf96]和[ZimC91]。

Ball和Larus关于剖面分析的工作包含在[BalL92]中。Wall在[Wall91]中考虑了剖面分析反馈重编译的作用和对性能的影响。

1.12 练习

1.1 确定和描述你的计算环境中使用的一个编译器的总体结构和中间代码。该编译器的哪些部分是在用户的控制下有选择地执行的?

RSCH 1.2 从[GhoM86]、[JohM86]、[SriW93]、[AdlG93]、[Wism94]、[BalL92]和[Wall91]中选取一篇论文，写一个3~5页的大纲，说明论文涉及的问题、研究成果或结论，以

18 及论文中对这些成果或结论给出的论据。

bbs.theithome.com

超星公司制作 请尊重作者版权

第2章　非形式化编译算法表示

本章讨论非形式化编译算法表示，即ICAN（Informal Compiler Algorithm Notation），本书中用它来表示编译算法。我们首先讨论扩展的巴科斯-诺尔范式，它用于表示ICAN的语法和下一章讨论的中间语言。随后简要介绍ICAN语言，以及它与其他常见程序设计语言的关系，非形式化地概述该语言的全貌，给出ICAN的形式化语法描述，并讲述其语义的非形式化英语描述。虽然非形式化的概述对于读者理解ICAN程序已经足够，但本书仍给出了ICAN完整的定义，以应对概述中没有涉及的情况。

2.1　扩展的巴科斯-诺尔范式语法表示

为了描述程序设计语言的语法，我们采用巴科斯-诺尔范式的一种称为“扩展的巴科斯-诺尔范式”的版本，简称XBNF。在XBNF中，终结符用印刷字体（例如，“type”和“[”），非终结符用斜体且第一个字符为大写字母，散布在其中的其他大写字母用于改善可读性（例如，用“*ProgUnit*”，而不用“*Progunit*”）。产生式由非终结符后随一个长右箭头（“→”）以及非终结符、终结符和运算符的序列组成。符号“ϵ”∈表示空字符串。

表2-1　扩展的巴科斯-诺尔范式语法描述中使用的运算符

符　号	含　义
|	分隔可选项
{和}	分组
[和]	可选
*	零次或多次重复
+	至少一次以上重复
⋈	左操作数重复至少一次以上，且重复之间用右操作数分隔

运算符列在表2-1中。上标运算符“*”、“+”以及运算符“⋈”都具有比连接运算符更高的优先级，连接运算符则具有比选择运算符“|”更高的优先级。花括号“{”…“}”和方括号“[”...“]”都用作分组运算符，但方括号还表示括号内的内容是可选的。注意，XBNF运算符采用正文所用的字体书写。当同样的符号以印刷字体出现时，它们代表被定义语言的终结符。因此，

KnittingInst → {{ knit | purl } *Integer* | castoff}$^{+}$

表示*KnittingInst*是由三种可能的选择（即，knit后随一个整数，purl后随一个整数，或castoff）中的任意一至多个对象组成的序列；而

Wall → brick ⋈ mortar | cementblock ⋈ mortar

表示*Wall*是一个由brick组成且被mortar分隔（或更恰当地说是由mortar连接）的序列，或者是一个由cementblock组成的序列，cementblock之间用mortar分隔。

作为一个更合适的例子，考虑下面的产生式，

ArrayTypeExpr → array [*ArrayBounds*] of *TypeExpr*

ArrayBounds → {[*Expr*]⋯ [*Expr*]} ⋈ ,

bbs.theithome.com

第一行描述*ArrayTypeExpr*的组成是：一个关键字array之后跟着一个左方括号“[”，其后跟着一个与*ArrayBounds*语法相符的成分，一个右方括号，再跟关键字of，最后是一个与语法*TypeExpr*相符的成分。第二行描述*ArrayBounds*是由逗号“,”分隔的、一个以上的三元组组成的序列，其中三元组的形式为：一个可选的*Expr*，其后接着“..”，最后是一个可选的*Expr*。下面是*ArrayTypeExpr*的例子：

```
array [··] of integer
array [1··10] of real
array [1··2,1··2] of real
array [m··n+2] of boolean
```

2.2 ICAN简介

本书用一种非形式化[⊖]但相对清晰的表示方法来书写算法，这种表示方法称为非形式化编
20 译算法表示，简称为ICAN（Informal Compiler Algorithm Notation）。它汲取了几种程序设计语言的特点，如C、Pascal和Modula－2，并扩充了诸如集合、序列、函数、数组，以及编译专用的一些类型。图2-1和图2-2给出了两个ICAN代码的例子。

```
1     Struc: Node ⟶ set of Node
2
3     procedure Example_1(N,r)
4        N: in set of Node
5        r: in Node
6     begin
7        change := true: boolean
8        D, t: set of Node
9        n, p: Node
10       Struc(r) := {r}
11       for each n ∈ N (n ≠ r) do
12          Struc(n) := N
13       od
14       while change do
15          change := false
16          for each n ∈ N - {r} do
17             t := N
18             for each p ∈ Pred[n] do
19                t ∩= Struc(p)
20             od
21             D := {n} ∪ t
22             if D ≠ Struc(n) then
23                change := true; Struc(n) := D
24             fi
25          od
26       od
27    end     || Example_1
```

图2-1 ICAN全局声明和过程之例（左列的行号不是代码的一部分）

ICAN的语法设计成每一种类型的复合语句都以一个终止分界符终止，例如“fi”终止一个if语句。这样，在语句之间就不再需要另外的分隔符。但是，作为改善可读性的一个约定，当两

⊖ ICAN的非形式性的一种度量是该语言的许多被认为是错误的地方，例如用一个超过数组维界的下标访问一个数组，其作用是不确定的。

bbs.theithome.com

超星公司制作 请尊重作者版权

条以上的语句出现在同一行时，我们用分号来分隔它们（如图2-1中第23行）。类似地，如果一个定义、声明或语句超过了一行，续行采用缩进格式书写（如图2-2中第1行和第2、4和5行）。

注释以分界符“||”开始直至该行结尾（如图2-1中第27行）。

词法上，ICAN程序是ASCII字符组成的序列。制表符、注释、行结束符以及一至多个空格符组成的序列均称为“空白符”。每一个空白符都可以转换为单个空格符而不影响程序的含义。关键字前后必须跟有空白符，但运算符不必如此。


```
 1    webrecord = record {defs: set of Def,
 2                        uses: set of Use}
 3
 4    procedure Example_2(nwebs,Symreg,nregs,
 5       Edges) returns boolean
 6       nwebs: inout integer
 7       nregs: in integer
 8       Symreg: out array [1··nwebs] of webrecord
 9       Edges: out set of (integer × integer)
10    begin
11       s1, s2, r1, r2: integer
12       for r1 := 1 to nregs (odd(r1)) do
13          Symreg[nwebs+r1] := nil
14       od
15       Edges := ∅
16       for r1 := 1 to nregs do
17          for r2 := 1 to nregs do
18             if r1 ≠ r2 then
19                Edges ∪= {⟨nwebs+r1,nwebs+r2⟩}
20             fi
21          od
22       od
23       for s1 := 1 to nwebs do
24          repeat
25             case s1 of
26    1:          s2 := s1 * nregs
27    2:          s2 := s1 - 1
28    3:          s2 := nregs - nwebs
29                return false
30    default:    s2 := 0
31             esac
32          until s2 = 0
33          for r2 := 1 to nregs do
34             if Interfere(Symreg[s1],r2) then
35                goto L1
36             fi
37          od
38    L1:   od
39          nwebs += nregs
40          return true
41       end     || Example_2
```

图2-2 ICAN代码之例2（左列的行号不是代码的一部分）

词法分析从左至右读字符，并按能够构成单词的最长字符序列形成词法单词。例如，如下代码:

bbs.theithome.com

```
for I37_6a:=-12 by 1 to n17a do
```

由下面9个单词组成:

22

```
for  I37_6a  :=  -12  by  1  to  n17a  do
```

2.3 ICAN概貌

本节给出ICAN的概貌，这对于帮助读者开始阅读和理解本书中的程序应当足够了。后面几节将形式化地定义ICAN的语法，并非形式化地定义其语义。

ICAN程序由类型定义序列、跟随在其后的变量声明序列、过程声明序列以及一个可选的主程序组成。

类型定义由类型名后跟一个等号和一个类型表达式组成，例如:

```
intset = set of integer
```

类型可以是一般类型，也可以是编译专用类型；可以是简单类型，也可以是构造类型。一般简单类型是boolean、integer、real和character。类型构造符列在下表中:

构造符	名　称	声明示例
enum	枚举	enum {left,right}
array ... of	数组	array [1··10] of integer
set of	集合	set of MIRInst
sequence of	序列	sequence of boolean
×	元组	integer × set of real
record	记录	record {x: real, y: real}
∪	联合	integer ∪ boolean
⟶	函数	integer ⟶ set of real

变量声明由变量名后面依次跟随一个可选的初始值、冒号和该变量的类型组成，例如，

```
is := {1,3,7}: intset
```

过程声明由过程名、包含在括号中的过程参数列表、可选的返回类型、过程参数声明和过程体构成。参数声明由逗号分开的变量名序列、冒号、in（传值调用）或out（传结果调用）或inout（传值和结果调用）三者之一，以及这些参数的类型依次组成。过程体由关键字begin、变量声明序列、语句序列以及关键字end依次组成。例如，图2-1、图2-2、图2-3都是过程声明的例子。

```
procedure exam(x,y,is) returns boolean
   x, y: out integer
   is: in intset
begin
   tv := true: boolean
   z: integer
   for each z ∈ is (z > 0) do
      if x = z then
         return tv
      fi
   od
   return y ∈ is
end    || exam
```

图2-3 ICAN过程声明之例

bbs.theithome.com

超星公司制作 请尊重作者版权

表达式可以是常数、变量、nil、一元运算符后跟一个表达式、二元运算符分开的两个表达式、括号表达式、数组表达式、序列表达式、集合表达式、元组表达式、记录表达式、过程或函数调用、数组元素、元组元素、记录域、size表达式或受限表达式。操作数和运算符必须在类型上兼容。

23

2.7节给出了与具体类型相适应的运算符。其中有几个含义不太明显的运算符在下面讨论。以下是构造类型常数的例子:

类　型	常量示例
array [1··2,1··2] of real	[[1.0,2.0],[3.0,4.0]]
sequence of integer	[2,3,5,7,9,11]
integer × boolean	⟨3,true⟩
set of (integer × real)	{⟨3,3.0⟩,⟨2,2.0⟩}
record {x: real, y: real}	⟨x:1.0,y:-1.0⟩
(integer × real) ⟶ boolean	{⟨1,2.0,true⟩,⟨1,3.0,false⟩}

integer和real类型包含∞和-∞。空集用∅表示，空序列用[]表示。值nil是每一种类型的成员，并且是每一个未初始化变量的初值。如果*x*是某个构造类型的成员，表达式|*x*|产生*x*的大小——例如，集合的基、序列的长度等。

当作用于集合时，一元运算符“◆”，产生该集合的任意一个成员。表达式*sq* ↓ *i*产生序列*sq*的第*i*个元素，*sq*1⊕*sq*2产生*sq*1与*sq*2的连接，而*sq* ⊖ *i*的结果是从*sq*中去掉第*i*个元素；如果*i*为负数，它从*sq*的末尾往回数。表达式*tpl* @ *i*生成元组*tpl*的第*i*个元素。

编译专用的类型根据需要来定义，例如，

```
Block = array [..] of array [..] of MIRInst
Edge = Node × Node
```

语句包括赋值语句、调用语句、返回语句、goto语句、if语句、case语句以及for、while和repeat循环。基本的赋值运算符是“:=”。同C语言一样，冒号也可以用满足后面条件的任何二元运算符替代，这种二元运算符的左操作数与赋值结果操作数相同；例如，下面的两条赋值语句含义相同:

24

```
Seq := Seq⊕[9, 11]
Seq ⊕= [9, 11]
```

每一个复合语句有一个终止分界符。复合语句的开始、内部和终止分界符如下所列:

开始	内部	终止
if	elif, else	fi
case	of, default	esac
for	do	od
while	do	od
repeat		until

case标号也是case语句的内部分界符。

该语言中使用的所有关键字都是保留字——它们不能作为标识符。关键字列在表2-8中。

下面几节详细介绍ICAN。

2.4 完整的程序

ICAN程序由类型定义序列、随后的变量声明序列、过程声明序列以及一个可选的主程序组成。主程序的形式与过程体相同。表2-2给出了ICAN程序的语法。

bbs.theithome.com

版权所有 侵权必究

表2-2 完整ICAN程序的语法

Program	→	*TypeDef** *VarDecl** *ProcDecl** [*MainProg*]
MainProg	→	*ProcBody*

2.5 类型定义

类型定义由类型名、其后的一个等号和它的定义组成（图2-2中第1、2行）。类型定义的语法在表2-3中给出。类型定义可以递归。由递归类型定义所定义的类型是满足该定义的最小集合，即，该定义的最小不动点。例如，由下式

```
IntPair = integer U ( IntPair × IntPair )
```

定义的类型是包含所有整数和所有其元素要么是整数要么是该类型的元素组成的偶对的偶对的集合。

25

表2-3 ICAN类型定义的语法

TypeDef	→	{*TypeName* =}* *TypeExpr*
TypeName	→	*Identifier*
TypeExpr	→	*SimpleTypeExpr* | *ConstrTypeExpr* | (*TypeExpr*)
		| *TypeName* | ∅
SimpleTypeExpr	→	**boolean** | **integer** | **real** | **character**
ConstrTypeExpr	→	*EnumTypeExpr* | *ArrayTypeExpr* | *SetTypeExpr*
		| *SequenceTypeExpr* | *TupleTypeExpr* | *RecordTypeExpr*
		| *UnionTypeExpr* | *FuncTypeExpr*
EnumTypeExpr	→	**enum** { *Identifier* ⋈ , }
ArrayTypeExpr	→	**array** [*ArrayBounds*] **of** *TypeExpr*
ArrayBounds	→	{[*Expr*] ·· [*Expr*]} ⋈ ,
SetTypeExpr	→	**set of** *TypeExpr*
SequenceTypeExpr	→	**sequence of** *TypeExpr*
TupleTypeExpr	→	*TypeExpr* ⋈ ×
RecordTypeExpr	→	**record** { {*Identifier* ⋈ , : *TypeExpr*} ⋈ , }
UnionTypeExpr	→	*TypeExpr* ⋈ ∪
FuncTypeExpr	→	*TypeExpr* ⋈ × ⟶ *TypeExpr*

2.6 声明

表2-4给出了ICAN变量和过程声明的语法。语法中包括了非终止符*ConstExpr*，但在该文法中没有它的定义，它代表一个不含变量的表达式。

表2-4 ICAN声明的语法

VarDecl	→	{*Variable* [:= *ConstExpr*]} ⋈ , : *TypeExpr*
Variable	→	*Identifier*
ProcDecl	→	**procedure** *ProcName ParamList* [**returns** *TypeExpr*]
		ParamDecls ProcBody
ProcName	→	*Identifier*
ParamList	→	([*Parameter* ⋈ ,])
Parameter	→	*Identifier*
ParamDecls	→	*ParamDecl**
ParamDecl	→	*Variable* ⋈ , : {**in** | **out** | **inout**} *TypeExpr*
ProcBody	→	**begin** *VarDecl** *Statement** **end**

bbs.theithome.com

超星公司制作 请尊重作者版权

变量声明由所声明的标识符的名字、一个可选的初值、冒号和该变量的类型组成（图2-1中第1、4、5和7~9行）。数组的维界是其类型的一部分，用跟随在关键字array之后、放置在方括号内的下标值范围表来说明（图2-2中第8行）。标识符的初值通过其后跟随的赋值运算符“:=”和值（它可以是任何单一由常数构成的表达式，只要是正确的类型）来说明（图2-1中第7行）。在一个声明中可以声明多个相同类型的标识符，用逗号分隔这些标识符名字和它的可选初值（图2-1中第8和9行）。

过程声明由过程关键字procedure、过程名、置于括号之中的参数表、可选的返回类型、声明参数类型的若干缩进行（图2-1中第4和5行），以及过程体组成（图2-1中第6~27行）。返回类型由关键字returns后随类型表达式组成（图2-2中第5行）。参数可声明为“in”（传值调用）、“out”（传结果调用）或“inout”（传值和结果调用）（参见图2-2中第6~9行）。过程的正文在关键字begin和end间缩进（图2-1中第6~27行）。

将类型定义或变量声明置于一组过程之前，可以使它们全局于该组过程（图2-1中第1行和图2-2中第1、2行）。

2.7 数据类型和表达式

表2-5和表2-6分别给出了ICAN的表达式和一般简单常数的语法。

表2-5 ICAN表达式语法

Expr	→	*Variable* | *SimpleConst* | (*Expr*) | *UnaryOper Expr*
		| *Expr BinaryOper Expr* | *ArrayExpr* | *SequenceExpr*
		| *SetExpr* | *TupleExpr* | *RecordExpr* | *ProcFuncExpr*
		| *ArrayEltExpr* | *SizeExpr* | *QuantExpr* | *Expr* ∈ *TypeExpr*
		| nil
UnaryOper	→	! | - | ◆
BinaryOper	→	= | ≠ | & | ∨ | + | - | * | / | % | ↑ | < | ≤ | > | ≥ | ∪ | ∩
		| ∈ | ∉ | × | ⊕ | ↓ | ⊖ | ⊗ | .
ArrayExpr	→	[*Expr* ⋈ ,]
SequenceExpr	→	[*Expr* ⋈ ,] | " *ASCIICharacter** "
SetExpr	→	∅ | { *Expr* ⋈ , | [*Variable* ∈] *Expr* **where** *SetDefClause* }
TupleExpr	→	⟨ *Expr* ⋈ , ⟩
RecordExpr	→	⟨ {*Identifier* : *Expr*} ⋈ , ⟩
ProcFuncExpr	→	*ProcName ArgList*
ArgList	→	([*Expr* ⋈ ,])
ArrayEltExpr	→	*Expr* [*Expr* ⋈ ,]
QuantExpr	→	{∃ | ∀} *Variable* ∈ [*Expr* | *TypeExpr*] (*Expr*)
SizeExpr	→	| *Expr* |
Variable	→	*Identifier*

表2-6 ICAN一般简单类型的常数的语法

SimpleConst	→	*IntConst* | *RealConst* | *BoolConst* | *CharConst*
IntConst	→	0 | [-] *NZDigit Digit** | [-] ∞
NZDigit	→	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Digit	→	0 | *NZDigit*
RealConst	→	[-] {*IntConst* . [*IntConst*] | [*IntConst*] . *IntConst*}
		[E *IntConst*] | [-] ∞
BoolConst	→	**true** | **false**
CharConst	→	' *ASCIICharacter* '
Identifier	→	*Letter* {*Letter* | *Digit* | _ }*
Letter	→	a | . . . | z | A | . . . | Z

类型是具有相同值属性的成员的集合。它可以是简单类型，也可以是构造类型；可以是一般类型，也可以是编译专用的类型。

一般简单类型是boolean、integer、real和character。

构造类型用一至多个类型构造符来定义。类型构造符是enum、array...of、set of、sequence of、record、“U”、“×”和“→”。符号“U”构造联合类型，“×”构造元组类型，“→”构造函数类型。

表达式可以是常数、变量、nil、一元运算符后跟一个表达式、二元运算符分开的两个表达式、括号表达式、数组表达式、序列表达式、集合表达式、元组表达式、记录表达式、过程或函数调用、数组元素、受限表达式（quantified expression）或size表达式。操作数和运算符必须在类型上兼容，后面将给予说明。

2.7.1 一般简单类型

boolean类型取值为true和false。右表的二元运算符作用于布尔类型变量。

操　作	符　号
等于	=
不等于	≠
逻辑与	&
逻辑或	V

前缀一元“非”运算符“！”作用于布尔变量。

受限表达式具有布尔值。它由符号“∃”或“∀”依次后随一个变量、“∈”、类型值或集合值表达式，以及括号内的布尔值表达式组成。例如，

```
∃v∈Var (Opnd(inst, v))
```

是一个受限表达式，它的结果是当且仅当存在某个变量v使得Opnd（inst，v）为真时，它的值才为真。

操　作	符　号
加	+
减	−
乘	*
除	/
模	%
取数	↑
等于	=
不等于	≠
小于	<
小于等于	≤
大于	>
大于等于	≥

28

整数值可以是0，可选的一个负号后跟一至多个十进制数字（但打头数字不是0）组成的数字串，∞或者−∞。

实数值有三种形式：一个整数后跟一个小数点和一个整数（这两个整数可以缺其中的一个，但不可以全部都缺）以及一个可选的指数，∞或−∞。[⊖] 一个指数是字母E之后跟随一个整数。

右表的二元运算符作用于有穷整数和实数。前缀一元“取负”运算符“−”作用于有穷整数和实数。只有关系运算符作用于无穷值。

字符值是置于单引号之内的可见ASCII字符，例如，'a'或'G'。可见ASCII字符（在语法中用未定义的非终止符*ASCIICharacter*代表它们）是所有可打印ASCII字符、空格符、制表符以及回车符。这些字符中有几个需要用右表所示转义序列来表示。

转义序列	表　示
%r	Carrriage return
%"	"
%'	'
%%	%

二元运算符等于（“=”）和不等于（“≠”）作用于字符。

2.7.2 枚举类型

枚举类型是标识符的有限非空集合。用如下形式的声明来声明变量*var*为枚举类型：

var: enum {$element_1$, ... ,$element_n$}

⊖ ICAN实数不是浮点数——它们是数学意义上的实数。

bbs.theithome.com

超星公司制作 请尊重作者版权

其中每一个$element_i$是标识符。

下面的例子声明action是一个枚举类型的变量:

```
action: enum (Shift, Reduce, Accept, Error)
```

二元运算符等于（“=”）和不等于（“≠”）作用于枚举类型的各成员。

枚举类型的元素可以作为case的标号（见2.8.6节）。

2.7.3 数组

用如下形式的声明来声明变量*var*为数组类型:

var: array [*subslist*] of *basetype*

其中*subslist*是逗号分开的下标范围表，每一个下标可以只由“..”组成。*basetype*是数组元素的类型。例如，代码段

```
U: array [5··8] of real
V: array [1··2,1··3] of real
   .
   .
   .
U := [1.0,0.1,0.01,0.001]
V := [[1.0,2.0,3.0],[4.0,5.0,6.0]]
```

声明U是一维数组，其下标范围从整数5到8；V是一个二维数组，它的第一个下标范围从整数1到2，第二个下标范围从整数1到3；并且这两个数组的元素都是实数类型。这段声明同时指定了具体的数组常数作为这两个数组的值。

当然，数组也可以看成是从若干整数序列的乘积到其他类型的一种有限函数。

一维数组类型常数是置于方括号中、用逗号分开的元素类型的常数序列，其长度是$hi - lo + 1$，其中*lo*和*hi*分别是下标的最小值和最大值。n（$n > 1$）维数组类型常数是置于方括号中、用逗号分开的$n-1$维数组类型的常数序列，此$n-1$维数组是删除第一维下标后获得的数组。这串常数的长度是$hi - lo + 1$，其中*lo*和*hi*分别是第一维下标的最小值和最大值。因此，数组是以行为主的。

二元运算符等于（“=”）和不等于（“≠”）作用于整个数组。当n维数组值表达式后跟一个置于方括号中、用逗号分开且至多n个下标值列表时，这也是一个表达式。

注意，下面两个数组类型

```
array [1..10, 1..10] of integer
array [1..10] of array [1..10] of integer
```

是不同的，即使这两个类型的常数没有区别。第一个是二维数组，而第二个是由一维数组组成的一维数组。

2.7.4 集合

集合类型的变量*var*可以用它的基类型的任意子集作为它的值，其声明具有如下形式:

var: set of *basetype*

其中，*basetype*是集合元素的类型。

集合常数可以是空集“∅”、置于花括号中用逗号分开的一串元素，或者有意定义的集合常数。所有元素必须具有相同的类型。例如，下面是一些集合常数:

```
∅    {1}    {1, 2, 100}    {<true, 1.0>, <false, -2.3>}
{n ∈ integer where 0 ≤ n & n ≤ 20 & n % 2 = 0}
```

还有,在下面的程序段中，

bbs.theithome.com

```
B: set of integer
B := {1,4,9,16}
B := {n ∈ integer where ∃m ∈ integer (1 < m & m < 4 & n = m * m)}
```

两个赋值都是合法的，且都赋予B相同的值。

右表的二元运算符作用于集合:

操　作	符　号
并	∪
交	∩
差	−
积	×
成员	∈
非成员	∉

其中，最后两个运算符“∈”和“∉”的左操作数是类型为*ty*的值，右操作数是类型为“set of *ty*”的值，并产生一个布尔类型的结果。对于“∈”，如果左操作数是右操作数的成员，其结果为true；否则为false。对于“∉”，如果左操作数是右操作数的成员，结果为false；否则为true。

前缀一元运算符“◆”从它的集合操作数中随机地选取一个元素，这个被选中的元素就是结果。例如，◆{1，2，7}的结果可以是1、2、7中任意之一。

注意，二元运算符“∈”与用于for循环迭代中的相同符号是不同的。前者产生一个布尔值，后者是一个能够产生一系列值的较大表达式的一部分，如图2-4所示。图a）中的代码在含义上与图b）的代码等价，其中Tmp是与A具有相同类型的一个新临时变量。

```
for each a ∈ A do
    body
od
```

a)

```
Tmp := A
while Tmp ≠ ∅ do
    a := ◆Tmp
    Tmp -= {a}
    body
od
```

b)

图2-4 a) 在集合上进行迭代的ICAN循环，b) 等价的while循环

另外，在语法描述中未定义的非终止符*SetDefClause*是一个从句，它有针对性地定义集合中的元素，例如下面代码中where和“}”之间的部分就是这样的一个从句:

```
31    S := {n ∈ N where ∃e ∈ E (e @ 2 = n)}
```

注意，含有*SetDefClause*的集合定义总是可以用嵌套循环来替换。例如，对于上面这个赋值语句，假设E的类型是N的类型与自身的积，则可用下面的代码来替换:

```
S := ∅
for each n ∈ N do
    for each e ∈ E do
        if e@2 = n then
            S ∪= {n}
        fi
    od
od
```

2.7.5 序列

如下形式用来声明一个变量*var*为序列类型:

var: sequence of *basetype*

其中，*basetype*是此序列的元素的类型，它们的成员可作为*var*的值。

序列类型常数由置于方括号内、用逗号分开的一组有限基类型成员组成。所有这些成员必须具有相同的类型。空序列记为“[]”。例如，下面是一些序列常数:

```
[]   [1]   [1, 2, 1]   [true, false]
```

二元运算符“⊕”表示序列的连接。二元运算符“↓”作用于一个序列和一个非零整数时，表示从该序列中选择一个元素。具体而言，正整数*n*选择序列中的第*n*个元素，负整数−*n*选择

bbs.theithome.com

序列的倒数第n个元素。例如，[2,3,5,7]↓2=3，[2,3,5,7]↓-2 = 5。二元运算符e作用于一个序列s和一个非零整数n时，生成从该序列删除第n个元素后形成的一个序列。例如，[2,3,5,7]e2=[2,5,7]，[2,3,5,7]e-2=[2,3,7]。

类型CharString是sequence of character的缩写。例如，"ab CD"与['a','b',' ','C','D']相同。可以互换[]和""来表示空的CharString，并且对于任意字符x，有['x']= "x"。

注意，数组常数是序列常数的子集——惟一的不同是，在数组常数中，每一嵌套层内的所有成员必须具有相同的长度。

2.7.6 元组

如下形式用来声明变量var为元组类型:

var: $basetype_1 \times \ldots \times basetype_n$

其中，$basetype_i$是第i个元素的类型。

元组常数是置于尖括号内、用逗号分开且长度固定的表。考虑下面的元组类型例子:

```
integer × integer × boolean
```

这个类型的元素是三元组，其第一个和第二个元素是整数，第三个元素是布尔值，例如<1，7，true>。下面也是元组常数的例子:

```
<1>  <1, 2, true>  <true, false>
```

当作用于一个元组和一个正整数索引时，二元运算符 @ 产生元组中具有该索引的元素。例如，<1，2，true>@3 = true。

2.7.7 记录

如下形式的声明将变量var声明为记录类型:

var: record {$idents_1$: $basetype_1$，...，$idents_n$: $basetype_n$}

其中，$idents_i$是逗号分开的成员选择符表，$basetype_i$是对应成员的类型。

记录常数是一个元组，其中每一个元素是标识符（称为选择符）和值组成的偶对，中间用冒号分开。一个特定记录类型的所有值都必须具有相同的标识符集合，并且，对于每一个标识符，这些值必须具有相同类型；但是，与不同选择符对应的值可以是不同的类型。例如，由如下类型定义

```
ibpair = record {int: integer,
                 bool: boolean}
```

定义的类型ibpair以形如 <int:i, bool:b> 的偶对作为其成员，其中i是整数，b是布尔值。在记录类型成员中，偶对的顺序无关紧要，因此，<int:3，bool:true>和<bool:true，int:3>是相同的记录常数。下面是记录常数的例子:

```
<r1:1.0, im:-1.0>   <left:1, right:2, val:true>
```

将二元运算符“.”作用于一个记录和一个其值是记录选择符之一的表达式时，其结果是该选择符对应成员的值。例如，在下面的代码段中，

```
ibpair = record {int: integer,
                 bool: boolean}
. . .
b: boolean
ibp: ibpair
```

bbs.theithome.com

超星公司制作 请尊重作者版权

```
. . .
ibp := <int:2,bool:true>
ibp.int := 3
b := ibp.bool
```

ibp和b最后的值分别是 <int:3, bool:true> 和true。

2.7.8 联合

联合类型是构成该联合的各种类型的值的集合的并集。如下形式的声明用来声明变量*var*为联合类型:

var: $basetype_1 \cup \ldots \cup basetype_n$

其中，$basetype_i$是构成该联合类型的类型集合中的一种类型。

作为联合类型的例子，考虑integer∪boolean，这个类型的元素要么是整数，要么是布尔值。

所有能作用于集合的运算符都能作用于联合。如果构成联合类型的集合是不相交的，那么联合中的元素所属的集合可以用成员运算符“$\in$”来确定。

2.7.9 函数

函数类型有一个定义域类型（写在箭头的左边）和一个值域类型（写在箭头的右边）。

如下形式用来声明一个变量*var*是函数类型:

var: $basetype_1 \times \cdots \times basetype_n \rightarrow basetype_0$

其中，对于$i = 1, \cdots, n$，$basetype_i$是定义域的第i个成员的类型，$basetype_0$是值域的类型。

有n个参数的函数常数是一个集合，它的每一个元素是一个$(n+1)$元组。此$(n+1)$元组的前n个成员分别是定义域的第一至第n个类型，第$(n+1)$个成员是值域类型。要成为一个函数，元组集合必须是单值的，即，如果两个元组的前n个成员相同，这两个元组的第$(n+1)$个成员也必须相同。

作为函数类型的一个例子，考虑boolean→integer。此函数类型的变量或常数是一种
34 偶对的集合：其第一个成员是布尔值，第二个成员是整数。也可以用赋值或含类型名的赋值来表示。例如，给定声明

```
A: boolean → integer
```

于是，为了指定A的一个具体函数值，可以写为:

```
A := {<true, 3>, <false, 2>}
```

或写为:

```
A(true):= 3
A(false):= 2
```

函数不必对每一个定义域元素有定义值。

2.7.10 编译专用的类型

编译专用的类型都以首字母大写的形式命名，并且在正文中根据需要定义。根据需要，它们可以是简单的，也可以是构造的。例如，类型Var、Vocab和Operator都是简单类型，而下面定义的类型Procedure、Block和Edge都是构造的。

```
Block = array [··] of array [··] of MIRInst
Procedure = Block
Edge = Node × Node
```

bbs.theithome.com

其中类型MIRInst也是构造的，它在4.6.1节定义。

2.7.11 值nil

值nil是每一种类型的成员。它是所有未初始化变量的初值。在大部分上下文中，在一个表达式中使用nil作为操作数将导致结果也为nil。例如，3+nil等于nil。

用nil作为表达式的操作数而不产生nil结果的惟一表达式是等于和不等于比较，如右表所示。

表达式	结　果
nil = nil	true
a = nil	false
nil ≠ nil	false
a ≠ nil	true

其中，*a*是非nil的任意值。

另外，nil可以出现在赋值语句的右端（图2-2中第13行），也可以作为过程的参数或返回值。

35

2.7.12 size运算符

运算符“|| ”作用于所有构造类型的对象。在所有情况下，它的值是其操作数的元素个数，只要其操作数大小是有限的。例如，若A声明为:

```
A: array [1..5, 1..5] of boolean
```

并且f被声明和定义为:

```
f: integer × integer ⟶ boolean
. . .
f(1,1) := true
f(1,2) := false
f(3,4) := true
```

则

```
|{1,7,23}| = 3
|['a','b','e','c','b']| = 5
|<rl:1.0,im:-1.0>| = 2
|A| = 25
|f| = 3
```

若*x*的大小是无穷的，则|*x*|是不确定的。

2.8 语句

语句包括赋值语句（例如图2-1中第12和19行）、过程和函数调用语句（图2-1中第19行和图2-2中第34行）、返回语句（图2-2中第29和第40行）、goto语句（图2-2中第35行）、if语句（图2-1中第22~24行）、case语句（图2-2中第25~31行）以及for循环（图2-1中第16~25行）、while循环（图2-1中第14~26行）、repeat循环（图2-2中第24~32行）。表2-7给出了它们的语法。

表2-7　ICAN语句的语法

Statement	⟶	*AssignStmt* | *ProcFuncStmt* | *ReturnStmt*
		| *GotoStmt* | *IfStmt* | *CaseStmt* | *WhileStmt*
		| *ForStmt* | *RepeatStmt* | *Label* : *Statement*
		| *Statement* ;
Label	⟶	*Identifier*
AssignStmt	⟶	{*LeftSide* :=}* *LeftSide* {: | *BinaryOper*} = *Expr*
LeftSide	⟶	*Variable* | *ArrayElt* | *SequenceElt* | *TupleElt* | *RecordElt*
		| *FuncElt*
ArrayElt	⟶	*LeftSide* [*Expr* ⋈ ,]

bbs.theithome.com

（续）

SequenceElt	⟶	*LeftSide* ↓ *Expr*
TupleElt	⟶	*LeftSide* @ *Expr*
RecordElt	⟶	*LeftSide* . *Expr*
FuncElt	⟶	*LeftSide ArgList*
ProcFuncStmt	⟶	*ProcFuncExpr*
ReturnStmt	⟶	`return` [*Expr*]
GotoStmt	⟶	`goto` *Label*
IfStmt	⟶	`if` *Expr* `then` *Statement** {`elif` *Statement**}*
		[`else` *Statement**] `fi`
CaseStmt	⟶	`case` *Expr* `of` {*CaseLabel* : *Statement**}+
		[`default` : *Statement**] `esac`
WhileStmt	⟶	`while` *Expr* `do` *Statement** `od`
ForStmt	⟶	`for` *Iterator* `do` *Statement** `od`
Iterator	⟶	{*Variable* := *Expr* [`by` *Expr*] `to` *Expr*
		| `each` *Variable* ⋈ , ∈ {*Expr* | *TypeExpr*}} [(*Expr*)]
RepeatStmt	⟶	`repeat` *Statement** `until` *Expr*

语句可以带有一个标号（图2-2中第38行）。语句标号是一个标识符后跟一个冒号。每个结构化的语句体用诸如`if`和`fi`的关键字来界定。

2.8.1 赋值语句

赋值语句由一至多个左部和一个右部组成，其中，每一个左部其后跟随有一个赋值运算符（图2-1中第10、12、15、19和21行）。每一个左部可以是变量名或变量的元素名，如记录、数组、序列或元组的成员，或函数值。每一个左部的赋值运算符，除了最后一个之外，都必须是“:=”。最后一个赋值运算符可以是“:=”（图2-1中第10和17行，图2-2中第26~28行），也可以是扩展的赋值运算符——这种赋值运算符的冒号被一个其左操作数与结果操作数相同的二元运算符所替代（图2-1中第19行，图2-2中第19和39行）。例如，下面的所有赋值都是合法的:

```
recex = record {lt,rt: boolean}
i, j: integer
f: integer ⟶ (boolean ⟶ integer)
g: integer ⟶ sequence of recex
p: sequence of integer
t: boolean × boolean
r: recex
. . .
i := 3
j := i += 1
f(3)(true) := 7
g(0)↓2.lt := true
p↓2 := 3
t@1 := true
r.rt := r.lt := false
```

36
~
37

赋值语句的右部可以是任何类型兼容的表达式（参见2.7节）。当一个扩展赋值运算符作用于原来的赋值形式时，赋值语句的左部和右部必须具有相同的类型。跟在扩展赋值运算符后面的右部的计算就好像右部带有括号一样。例如，赋值语句

```
S := S1 ∪= {a}∩X
```

等价于

bbs.theithome.com

超星公司制作 请尊重作者版权

```
S := S1 := S1∪((a}∩X)
```

它又等价于

```
S1 := S1∪((a}∩X)
S := S1
```

但不等价于

```
S1 := (S1∪(a}) ∩X
S := S1
```

2.8.2 过程调用语句

过程调用语句具有过程表达式的形式，即表2-5中的*ProcFuncExpr*。它由过程名后跟置于括号中、用逗号分开的参数表组成。它导致用指定的参数调用所命名的过程。

2.8.3 返回语句

返回语句由关键字return后跟一个可选的表达式组成。

2.8.4 goto语句

goto语句由关键字goto后跟一个标号组成。

2.8.5 if语句

if语句具有如下形式:

```
if condition0 then
    then_body
elif condition1 then
    elif_body1
. . .
elif conditionn then
    elif_bodyn
else
    else_body
fi
```

38

其中，elif和else部分是可选的。条件*condition*是布尔值表达式，并且按捷径方式来计算，即，给定$p \vee q$，当且仅当p求值为false时才计算q。每一个if体是零个或多个语句的序列。

2.8.6 case语句

case语句具有如下形式:

```
    case selector of
label1:   body1
label2:   body2
. . .
labeln:   bodyn
default:  body0
    esac
```

其中default部分是可选的。每一个标号是一个与*selector*表达式同类型的常数，*selector*表达式必须是简单类型或枚举类型。每个执行体是零个或多个语句的序列。同Pascal一样，在执行了其中的一个选择对应的执行体之后，控制从闭分界符“esac”之后的语句继续执行。case语句必须至少有一个非default的case标号和对应的执行体。

bbs.theithome.com

超星公司制作 请尊重作者版权

2.8.7 while语句

while语句具有如下形式:

```
while condition do
    while_body
od
```

其中，条件*condition*是布尔值表达式，并且按捷径方式计算。while循环体是零至多条语句的序列。

2.8.8 for语句

for语句具有如下形式:

```
for iterator do
    for_body
od
```

其中，迭代符*iterator*可以是数值的，也可以是枚举的。数值迭代符依次指定一个变量、一个值域以及一个括号内的布尔表达式，如“i := n by -1 to 1 (A[i]=0)”(参见图2-2中第12~14行)。如果“by”部分是“by 1”，则它是可选的。布尔表达式也是可选的，如果它没有出现，则使用值true。变量i的值在循环体内不可以被改变。

39

枚举迭代符，例如“each n∈N (n≠abc)”(参见图2-1中第11~13行和第16~25行)或“each p, q∈T (p≠q)”，只选择集合中所有那些满足某种条件的元素，选择的顺序不确定，具体条件则由集合操作数之后括号内的布尔表达式指明。在没有带括号的布尔表达式的情况下，使用true值作为选择条件。如果变量部分有多个元素，它们都必须满足相同的标准。例如，“each m, n∈N (1≤m& m≤n & n≤2)”导致变量偶对 <m, n> 的定义域按某种顺序为<1, 1>、<1, 2>和<2, 2>。对于任何出现在迭代符中的集合*S*，for语句的循环体不能改变*S*的值。

循环体由一至多条语句组成。

2.8.9 repeat语句

repeat语句具有如下形式:

```
repeat
    repeat_body
until condition
```

其中，循环体由一至多条语句组成。条件*condition*是一个按捷径方式求值的布尔值表达式。

2.8.10 ICAN的关键字

ICAN的关键字列在表2-8中，它们都是保留字，不能用作标识符。

表2-8 ICAN的关键字

array	begin	boolean	by
case	character	default	do
each	elif	else	end
enum	esac	false	fi
for	goto	if	in
inout	integer	nil	od
of	out	procedure	real
record	repeat	return	returns
sequence	set	to	true
until	where	while	

40

bbs.theithome.com

版权所有 侵权必究

2.9 小结

这一章主要描述ICAN，这是用于表示本书算法的一种非形式化表示。

这种语言包含了一套丰富的预定义和构造类型，包括那些专门用于编译器结构的类型，而且这种语言能够有效地表示各种表达式和语句。它的每一种复合语句都用闭分界符终止，其中有些复合语句，如while和case语句还有内部的分界符。

该语言的非形式性主要体现在它的各种结构的语义不是特别详细而精确，当一个结构在语法上合法时，在语义上它可能有二义性、不确定性，也可能是非法的。

2.10 进一步阅读

没有ICAN的参考文献，因为它是专门为本书而发明的。

2.11 练习

2.1 (a) 描述如何将XBNF语法表示转换成只使用连接的表示。(b) 应用你的方法重写下面的XBNF描述。

$$
\begin{array}{lcl}
E & \longrightarrow & V \mid AE \mid (\ E\ ) \mid ST \mid SE \mid TE \\
AE & \longrightarrow & [\ \{\ E \mid \mathtt{nil}\ \}\ ] \\
ST & \longrightarrow & \text{"}\ AC^{+}\ \text{"} \\
SE & \longrightarrow & \emptyset \mid \{\ E^{*}\ \} \\
TE & \longrightarrow & \langle\ E \bowtie ,\ \rangle
\end{array}
$$

2.2 描述如何用不包含数组、集合、记录、元组、乘积和函数的ICAN版本来实现数组、集合、记录、元组、乘积和函数，以及它们的运算（即，利用联合和序列构造符来表示其他构造类型），并以此说明它们都只是ICAN的“句法上的修饰成分”。

2.3 (a) 已知结点集合N $\subseteq$ Node、一个无向弧集合E $\subseteq$ Node × Node，以及开始与结束结点start，goal $\in$ N，写一个走迷宫的ICAN算法。算法应当返回一个结点表，表内的结点构成了从start到goal的一条路径；如果不存在这种路径，返回nil。(b) 你的这个算法关于n=|N|和e=|E|的时间复杂度是多少?

2.4 利用上一习题的算法解货郎担问题，即，返回一个开始并结束于start，经过所有结点（start除外）一次且仅一次的路径组成的结点表。如果不存在此路径，返回nil。

2.5 已知集合A的二元关系R $\subseteq$ A × A，写出一个计算此关系的自反传递闭包的ICAN过程RTC(R,x,y)。自反传递闭包R，记为R*，当且仅当a=b满足aR*b；或者存在一个c，使得aRc且cR*b。因此，若xR*y，RTC(R,x,y)返回true，否则返回false。 41

ADV2.6 我们在ICAN中有意省去了指针，因为指针会引起一些严重的问题，而不使用指针就可以完全避免这些问题。这些问题包括指针别名和可能创建环形结构。指针别名是指两个以上的指针指向同一个对象，因此改变其中一个指针的引用会影响其他指针的引用。环形结构是一个指针序列，该指针序列经过一系列的引用后又回到其开始处。但是，不使用指针会降低算法应有的效率。假设我们决定将ICAN扩充为包含指针的PICAN，(a) 列出这样做的优缺点；(b) 讨论此语言所需要的扩充，以及这些扩充对程序员和语言实现所产生的问题。 42

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第3章　符号表结构

本章探讨有关符号表构造的问题，包括如何设计反映现代程序设计语言特征的符号表，以及如何使它们对于这些语言的编译实现更具效率等。

我们首先讨论符号可能属于的存储类，以及支配符号在程序各个部分中可见性的规则，也即作用域规则。接着讨论符号属性和构造局部符号表的方法，局部符号表适用于其作用域单一的符号。然后讲述全局符号表的表示，包括作用域的导入和导出，并给出全局和局部符号表的程序设计接口，以及用ICAN表示的、根据属性生成存取变量的子程序。

3.1　存储类、可见性和生命期

多数程序设计语言允许给变量指定存储类（storage classes）。存储类规定了变量的作用域、可见性和生命期等特征。正如后面将看到的，支配作用域的规则也决定了符号表的构造规则和运行时访问变量的表示规则。

作用域（scope）是其内说明了一个以上变量的静态程序结构单元。在许多语言中，作用域可以嵌套。Pascal中有过程作用域，C中有块、函数、文件作用域。与作用域密切相连的一个概念是变量的可见性（visibility），可见性指出变量名引用的特定实例处在什么作用域内。例如，在Pascal中，如果变量a在最外层作用域中声明，那么，它在该程序的所有地方都是可见的[⊖]，但不包括在同样声明了变量a的函数内，也不包括在该函数所嵌套的函数内；这些函数见到的是局部变量a（除非它又被同名的另一个变量声明所取代）。如果最内层作用域的变量使得外层作用域中的同名变量暂时变为不可见的，则称内层变量遮盖（shadows）了外层变量。

变量的生命期（lifetime）是声明此变量的程序的一段执行时期，它从变量第一次可见时开始，到最后一次可见时结束。因此，Pascal程序中最外层声明的变量的生命期为程序的整个执行期间，而在嵌套过程内声明的变量可以有多次生命期，每次生命期从过程入口开始，到过程退出时结束。Fortran中那些具有save属性的变量，或C中那些具有static属性的局部变量具有不连续的生命期——若它们在过程f()中声明，它们的生命期由f()的各次执行期组成，并且它们的值从一次执行期保存到下一次执行期。

几乎所有的语言都有全局的（global）存储类。全局存储类赋予变量的生命期为程序的整个生命期（全局作用域），即，它使得整个程序都能见到该变量，或者，在可见性规则允许一个变量遮盖另一个变量的语言中，它使得该变量在未被遮盖的地方处处可见。全局作用域的例子包括C中用extern声明的变量，以及Pascal中在最外层声明的变量。

Fortran有公用存储类，它与多数作用域的概念不同之处在于，Fortran的公用对象在多个程序单元可见，这些程序单元相互并无嵌套，并且这些对象可以有不同的名字。例如，假设在子程序f1()和f2()中分别有如下公用说明:

⊖ 在许多语言中，如C，变量的作用域在代码中从它的声明点开始，一直延续到程序单元的结束；而另一些语言，如PL/I，变量的作用域包含整个相关的程序单元。

bbs.theithome.com

```
common /block1/i1,j1
```

和

```
common /block1/i2,j2
```

则变量i1和i2在它们各自的子程序内引用同一个存储单元，变量j1和j2也一样，并且它们的生命期是程序的整个执行期。

有些语言，如C，具有文件或模块存储类，这种存储类使得变量只在某个文件或模块内可见，但生命期为程序的整个执行期。

大多数语言支持自动存储类或栈存储类。这种存储类将变量的作用域指定为声明它的程序单元，生命期是该程序单元的特定活跃期。这种存储类也可与过程相关（如Pascal），或同时与过程和过程内的块相关（如C和PL/I）。

有些语言允许这种存储类，它们由限定前面所述的存储类为静态的而来。例如，C允许将变量声明成static，这使得在程序的整个执行期内都保存分配给该变量的存储空间，即使它们的声明在某个函数之内。这些变量只能在该函数内被访问，但它们的值从函数的一次调用一
44 直保存到下一次调用，就像Fortran的save变量一样。

有些语言允许数据对象（有几种语言还允许变量名）具有动态生命期，即从它的（隐式的或显式的）分配点开始至它的释放点结束。有些语言，尤其是LISP，允许动态作用域(dynamic scoping)，即作用域可以按照调用关系嵌套而不是静态嵌套。在动态嵌套的情况下，如果过程f()调用g()，而g()内使用了未加声明的变量x，则g()将使用f()中声明的x；或者，如果f()也没有声明x，则使用f()的调用者的x，如此递推，而与程序的静态结构无关。

有些语言，如C，有显式的volatile存储类修饰符，它指出声明为volatile的变量可能会异步地被修改，例如，被I/O设备修改。这种声明限制了对包含访问这种变量的结构进行优化。

3.2 符号属性和符号表项

程序中的每一个符号有一系列相关联的属性，这些属性既来源于源语言的语法和语义，也来源于程序对符号的具体声明和使用。典型的属性中有一些是很明显的，如符号的名字、类型、作用域和大小；其他一些属性，如符号的寻址方法，则不是很明显。

本节试图枚举出各种可能的属性，并对那些不太明显的属性进行解释。我们也用这种方式来讲述符号表的内容，即符号表项。*符号表项*（symbol-table entry）以一种便于设置和查找的方式集中了具体符号的各种属性。

表3-1列出了符号的典型属性集合。为了便于表示压缩的和非压缩的数据，一方面提供了size和boundary，另一方面又提供了bitsize和bitbdry。

表3-1 符号表项中典型的域

域 名	类 型	含 义
name	字符串	符号的标识符
class	枚举	存储类
volatile	布尔值	被异步访问的量
size	整数	字节大小
bitsize	整数	非整字节数时的位大小
boundary	整数	字节对齐单位
bitbdry	整数	非整字节数时的位对齐单位

bbs.theithome.com

超星公司制作 请尊重作者版权

（续）

域　名	类　型	含　义
type	枚举或类型引用	源语言数据类型
basetype	枚举或类型引用	构造类型的元素
machtype	枚举	机器类型，它与源语言的简单类型，或构造类型的元素类型对应
nelts	整数	元素个数
register	布尔值	若该符号的值在寄存器中，其值为真
reg	字符串	包含该符号值的寄存器名字
basereg	字符串	计算该符号的地址所使用的基址寄存器名字
disp	整数	该符号的存储单元相对基址寄存器之值的位移

类型引用（type referent）或者是一个指向表示构造类型结构的指针，或者是表示一个构造类型结构的名字（ICAN中没有指针，因此，我们将使用后者）。提供type、basetype和machtype使得我们对如下的Pascal类型:

```
array [1..3, 1..5] of char
```

能够指明其type域是一个诸如t2的类型引用，与它相关联的值是<array，2，[<1，3>，<1，5>]，char>，指明其basetype域为char，machtype域为值byte。此外，它的nelts的值是15。basereg和disp域的存在使我们能够表示：为了访问这个Pascal数组的起始地址，如果basereg是r7，disp是8，则应当形成地址[r7+8]。

符号表记录最复杂的方面通常是type属性的值。一般说来，源语言类型由预定义的类型（如Pascal的integer、char、real等）和构造类型（如Pascal的枚举、数组、记录和集合类型）组成。预定义的类型可以用枚举来表示，而构造类型则可用元组来表示。因此，Pascal类型模板

```
array [t1, ..., tn] of t0
```

可用一个元组来表示，该元组由一个表示其构造符（array）的枚举值、维数、表示*t*1到*tn*每一维的一个序列，以及基类型*t*0的表示组成，即ICAN元组

```
<array, n, [t1, ..., tn], t0>
```

类似地，记录类型模板

```
record f1:t1;...;fn:tn end
```

可以用这样一个元组来表示，该元组的组成是：一个表示其构造符（record）的枚举值、域的个数以及域标识符*fi*和类型*ti*组成的偶对的一个序列，即，

```
<record, n, [<f1, t1>, ..., <fn, tn>]>
```

在类型定义中使用正在构造的这个类型的名字是通过引用它的类型定义来表示的。作为一个具体的例子，考虑图3-1a中给出的Pascal类型声明，图3-1b是它们的ICAN表示。注意，它们是递归的：t2的定义包含了对t2的引用。

bbs.theithome.com

```
t1 = array [0..5,1..10] of integer;
t2 = record
          t2a: integer;
          t2b: ↑t2;
          t2c: array [1..3] of char;
       end;
t3 = array [1..100] of t2;
```

a)

```
t1 = ⟨array,2,[⟨0,5⟩,⟨1,10⟩],integer⟩
t2 = ⟨record,3,[⟨t2a,integer⟩,⟨t2b,⟨pointer,t2⟩⟩,
     ⟨t2c,⟨array,1,[⟨1,3⟩],char⟩⟩]⟩
t3 = ⟨array,1,[⟨1,100⟩],t2⟩
```

b)

图3-1 a) 一组Pascal类型声明，b) 它们的ICAN表示

我们知道，在所有语言中，预定义的类型都是全局的，而用户定义的类型遵循与变量标识符相同的作用域规则。所以，可以用与后面将讨论的局部和全局符号表结构相同的类型表（或图）来表示它们。

3.3 局部符号表管理

下面我们讨论如何管理特定过程的局部符号表，这是任何代码生成方法都会遇到的一个基本问题。

下面是一组创建、释放和管理符号表及其表项的过程接口（SymTab是表示符号表的类型，Symbol是符号的类型）：

```
New_Sym_Tab: SymTab→SymTab
```

创建一个新的局部符号表，给定的符号表作为其父亲；或者，如果没有给定符号表的话，其父亲为nil（参见3.4节全局符号表结构的讨论），并返回新的（空的）局部符号表。

45 ~ 47

```
Dest_Sym_Tab: SymTab→SymTab
```

释放当前局部符号表，并返回它的父亲（或nil，如果没有父亲）。

```
Insert_Sym: SymTab × Symbol→boolean
```

插入给定符号的符号表项至指定的符号表，并返回true；或者,如果该符号已经在表中，则不插入新项，然后返回false。

```
Locate_Sym: SymTab × Symbol→boolean
```

在符号表中查找给定的符号，如果它在表中则返回true；否则返回false。

```
Get_Sym_Attr: SymTab × Symbol × Attr→Value
```

返回指定符号表中与指定符号的指定属性相连的值，前提是此符号在表中；否则，返回nil。

```
Set_Sym_Attr: SymTab × Symbol × Attr × Value→boolean
```

如果指定的符号在指定的符号表中，则为其指定属性设置指定值，并返回true；否则返回false；表3-1中列出的域都视为是属性，如需要还可以有其他属性。

```
Next_Sym: SymTab × Symbol→Symbol
```

按某种顺序查找符号表，返回跟在第二个参数指定符号之后的那个符号；当第二个参数为nil时，它返回符号表的第一个符号，或者如果符号表为空，返回nil。当第二个参数设置为符号表中的最后一个符号时，也返回nil。

bbs.theithome.com

版权所有 侵权必究

```
More_Syms: SymTab × Symbol → boolean
```

如果存在多个要查看的符号，返回true；否则返回false；如果符号参数为nil，当符号表非空时它返回true，如果符号表为空，它返回false。

注意，在Get_Sym_Attr()和Set_Sym_Attr()定义中类型Value的用法。我们想让它成为这样一个联合类型，这个联合类型包含了各种属性可能具有的所有类型。还要注意的是，最后两个例程可按下面的方法用于逐个查看符号表中的符号:

```
s := nil
while More_Syms(symtab,s) do
   s := Next_Sym(symtab,s)
   if s ≠ nil then
      process symbol s
   fi
od
```

设计符号表时主要的考虑是要能使符号和属性的插入和提取都尽可能地快。我们本可以将符号表构造成由表项组成的一维数组，但那样会使得查找相当慢。另外有两种选择，即平衡二叉树或散列表，二者都提供了快速的插入、查找和提取，但在保持树的平衡和计算散列值上会有一些代价。如后面3.4节将讨论的，一般更好的方法是采用对每一个散列值带有一系列项的散列表。对于诸如Modula-2、Mesa、Ada和面向对象语言，最适合于处理作用域规则的符号表 48
实现方法是将散列与数组结合起来使用。图3-2给出了说明这种符号表是如何组织的示意图(“项*i*. *j*”表示在第*i*条散列链中的第*j*项)。所选择的散列函数应当使标识符相对平均地分布于各个散列值。

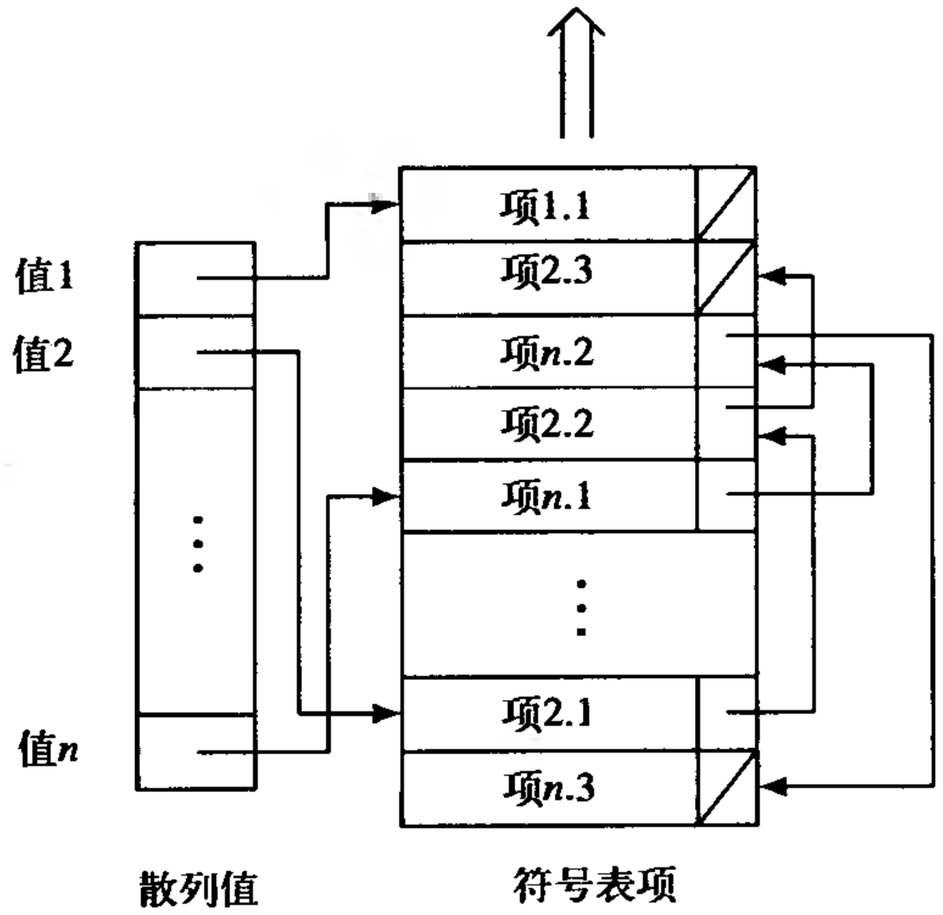


图3-2 每一个散列值具有散列桶链的局部符号散列表

3.4 全局符号表结构

源语言的作用域和可见性规则隐含地确定了全局符号表的结构。对于许多语言，如ALGOL 60、Pascal和PL/I，作用域规则具有将整个全局符号表构造成其结点是局部符号表的一棵树的作用，其中，树根是具有全局作用域的局部表，而那些被嵌套其内的作用域的局部表则作为包含它们的作用域对应表的儿子。因此，图3-3所示的一段Pascal程序结构，其对应的全局符号表结构如图3-4所示。

bbs.theithome.com

不过，对于这种语言，编译器可以采用更简单的结构。因为在编译过程的任意一点，所处理的只是这棵树上的某个特定结点，并且只需要访问从那个结点开始至根结点路径上的符号表。于是，用栈来表示一条路径上包含的局部符号表就足够了。当进入新的较深一层的嵌套作用域时，将其局部表压入栈中，而当退出此作用域时，则从栈中弹出它的局部表。图3-5展示了处理图3-3程序期间出现的全局符号表序列。

49 这样，任何在当前局部作用域找不到的变量都可以在其祖先作用域中查找。例如，考查过程i()。其内引用的变量b是在i()中声明的局部变量b，而a引用的是在g()中声明的局部变量b，c引用的是在最外层作用域e()中声明的全局变量c。

局部符号表栈可以用一个数组组成的栈和一组散列表来实现，数组组成的栈和散列表的实现方法如3.3节所述。更好的一种结构是使用两个栈，一个存放标识符，另一个指明每一张局部表的基址（这个栈叫做块栈），并且同前面一样，辅以一张散列表来散列符号名。如果采用这种方法，则过程New_Sym_Tab()压一个新项至块栈，Insert_Sym()在符号栈顶添加一项并将它链在散列链的开始，Locate_Sym()只需沿符号的散列链进行查找。Dest_Sym_Tab()从所有链中删除块栈当前登记项之上的那些登记项，那些登记项都位于链的前面，并由此而释放块栈栈顶的登记项。图3-6给出了这种作用域模型的例子，其中假定f和e具有相同的散列值。

余下的一个问题是如何构造那些不符合前面所讨论的树结构的作用域，例如Modula-2的import和export、Ada的package和use语句、C++的继承机制等。对于前两种语言，编译系统必须提供一种机制分别用于保存和提取关于模块和包的定义。下面要讨论的作用域模式基于Graham、Joy和Roubine[GraJ79]提出的开放作用域和闭作用域的区别。对于开放作用域（open scope），可见性规则直接对应于作用域的嵌套，而对于闭作用域（close scope），可见性规则在程序中显式地指明。对于开放作用域，用前面描述的机制就够了。

```
program e;
   var a, b, c: integer;
   procedure f;
      var a, b, c: integer;
   begin
      a := b + c
   end;
   procedure g;
         var a, b: integer;
      procedure h;
         var c, d: integer;
      begin
         c := a + d;
      end;
      procedure i;
         var b, d: integer;
      begin
         b := a + c
      end;
begin
   b := a + c
end;
procedure j;
   var b, d: integer;
begin
   b := a + d
   end;
begin
   a := b + c
end .
```

图3-3 嵌套结构的Pascal程序

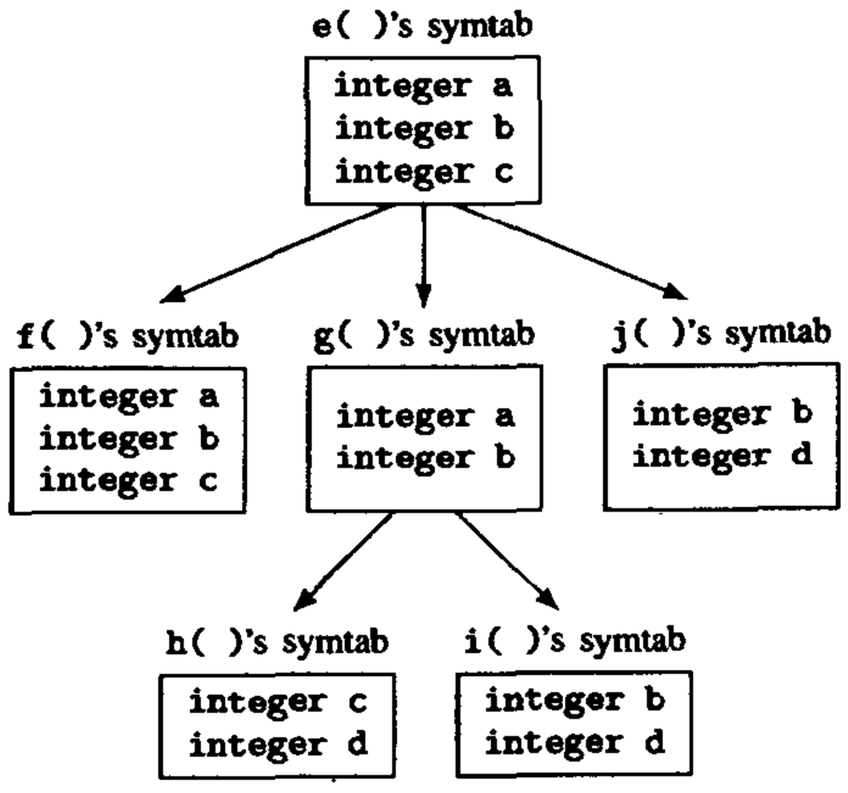


图3-4 图3-3的Pascal程序对应的局部符号表树

但是，闭作用域可以使得一组名字能在某个另外的作用域中可见，而与嵌套无关。例如，在Modula-2中，一个模块可以包含一列它要导出的对象名，其他模块可以通过明显地列出它们而导入所有名字或其中的任何名字。Ada的包机制允许一个包导出一组对象，并允许其他模块

bbs.theithome.com

通过明显地指明要使用这个包而导入它们。这两种作用域机制和其他的显式作用域机制可采用栈＋散列符号表模型，通过为每个标识符保存一张表以记录该标识符可见的作用域的层次号来实现。这样做可能易于维护，不过也可以用空间开销较少的方法来实现。

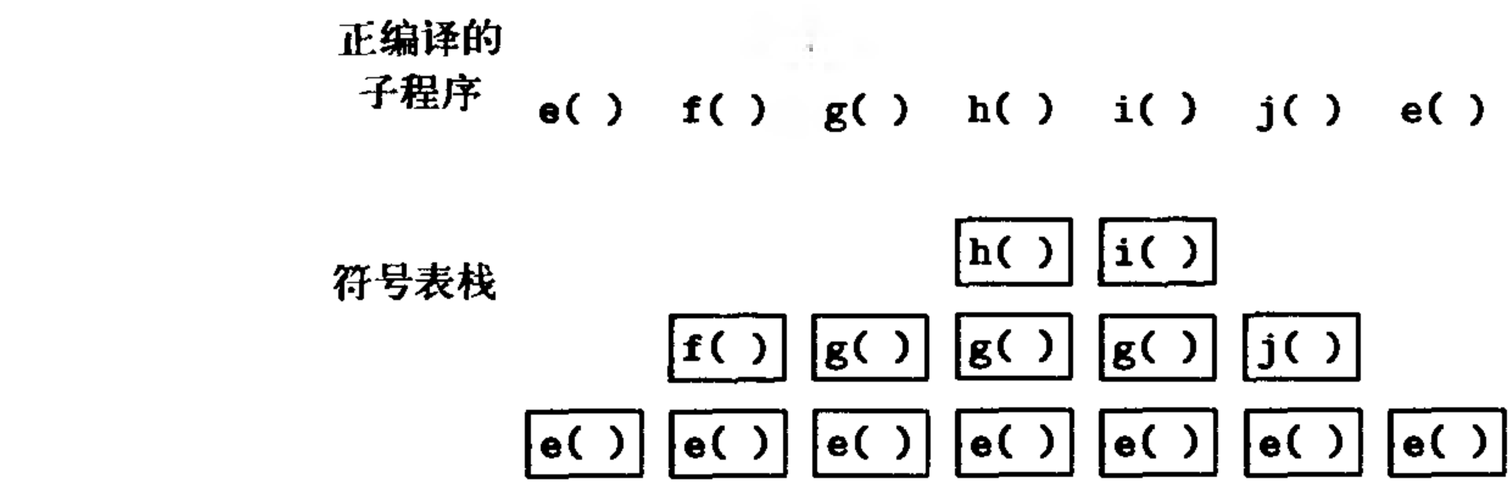


图3-5 在编译图3-3所示Pascal代码过程中出现的符号表栈

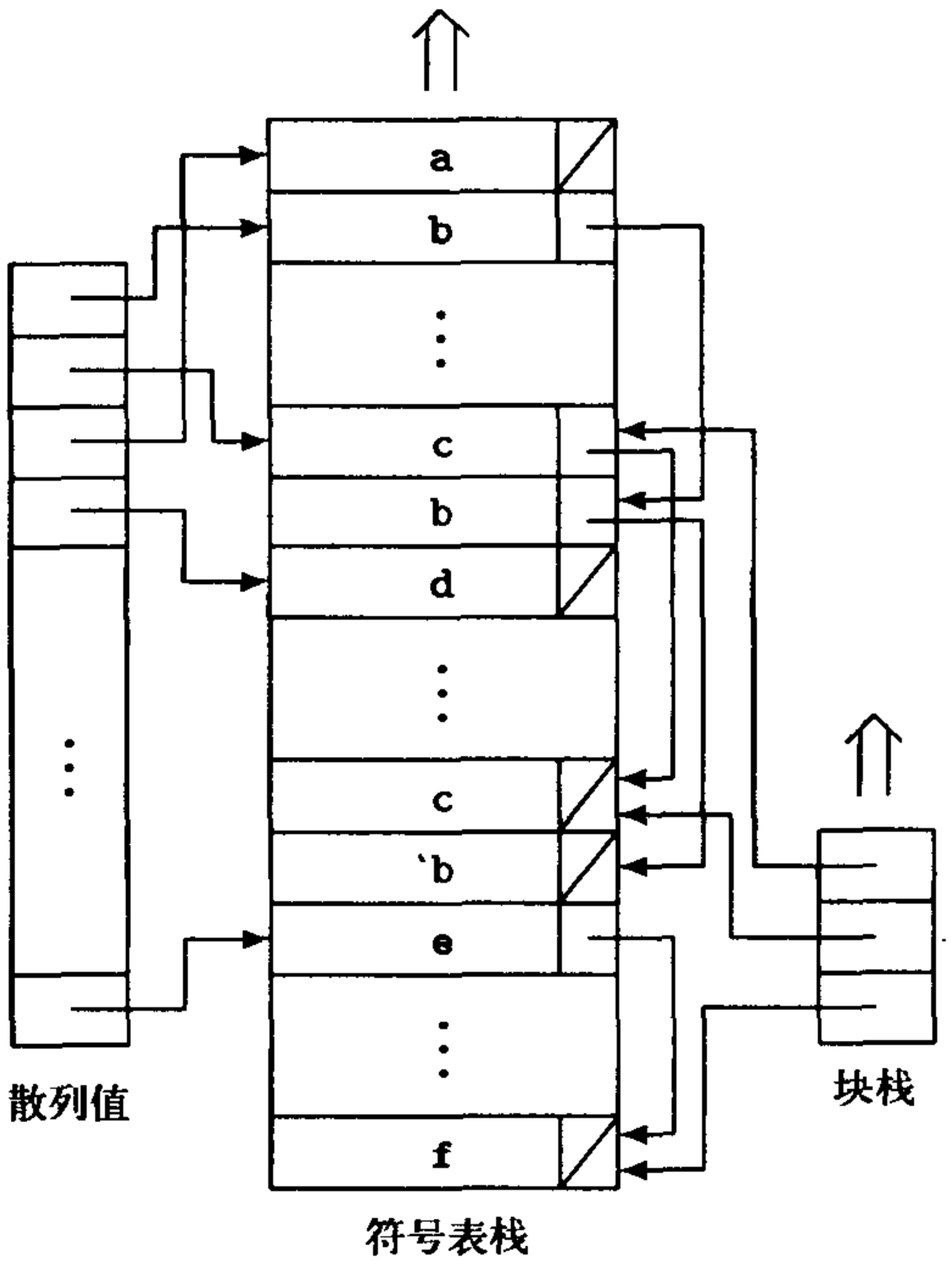


图3-6 带有块栈的散列全局符号表

这种简化源于人们意识到在这种表中层号一定是连续的，因为作用域能导出一个标识符，仅当该标识符在其内是可见的；能导入一个标识符仅当该标识符是显式说明的，或在包含它的作用域中是可见的。如果保证每个登记项在可见它的最外层作用域退出的同时也从符号表中被删除，则处理还可以进一步简化。我们只需要记录它们在其中可见的最内层作用域的层号，并在进入和退出一个作用域时更新此层号。这种方法能快速实现作用域的登记、符号的插入、属性的设置和提取等操作，但在退出作用域时会需要查找整个符号表，以找到那些需要删除的登记项。

为适应闭作用域而对栈＋散列表模式所作的最有效的修改方法是，使用栈结构来反映符号说明所在的作用域或导出此符号的最外层作用域，同时用散列结构来实现可见性规则。它按这样一种顺序重排散列链中的元素：将要插入的导入符号插入在链的前部，之后设置它的最内层作用域层号，对该作用域导出的每一个符号也按同样的顺序重排，然后将局部说明的符号插入

bbs.theithome.com

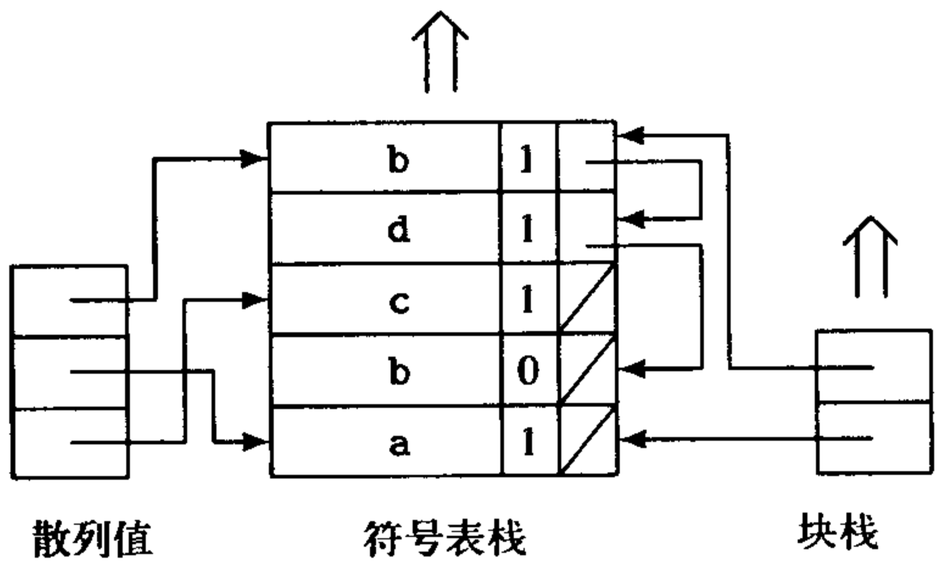
在那些已在散列链中的符号之前。读者可以证明，在每一条链中的符号保持了反映可见性规则的顺序。当退出一个作用域时，我们必须从散列链中删除那些非导出的局部符号，这些符号登记项的作用域层号与当前作用域的相同。如果一个符号是在包含当前作用域的外层作用域中说明的，或是导出到这个作用域的（从这个符号的栈结构可以判明），该符号将保留在散列链中，并且其最内层作用域层号减1；否则，从散列表中删除此符号。

50
~
52

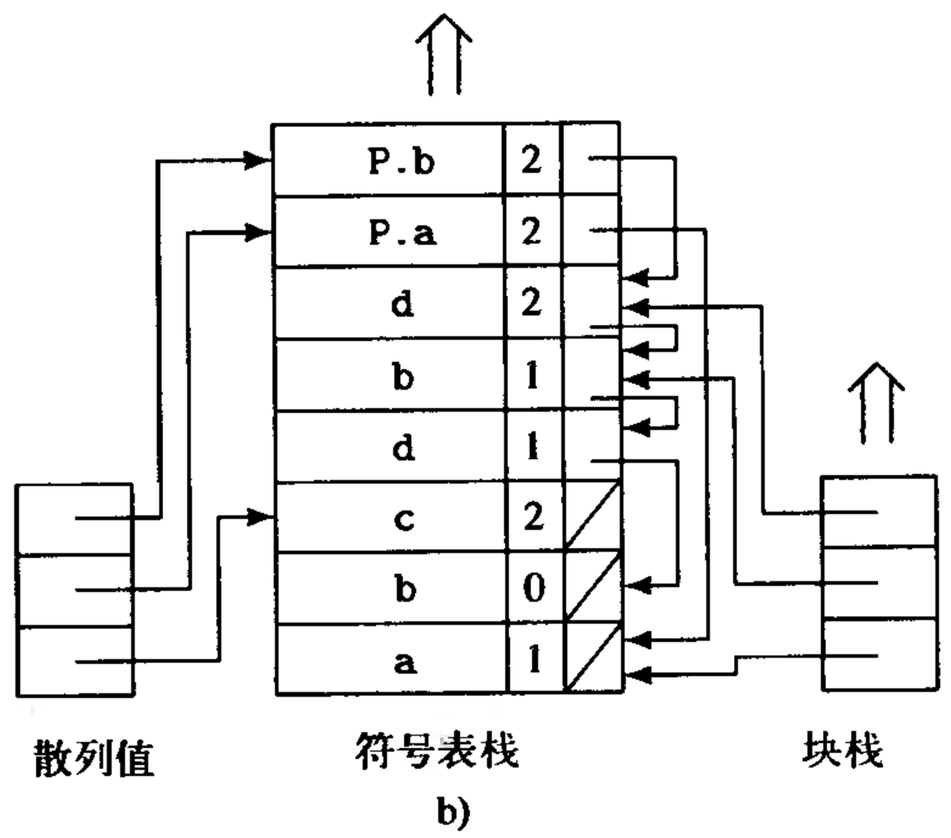
作为一个例子，考虑图3-7所示代码。在刚进入过程f()时，符号表如图3-8a所示。在刚进入g()时，有新说明的变量d和从包P中导入的变量a和b（注意，我们假设b和d具有相同的散列值），由此新形成的符号表结构如图3-8b所示。注意在图a)中原来包含两个b和一个d的散列链已被扩展成导入的符号b在前，后随新说明的符号d，然后才是以前登记的那三个符号。最内层层号也已经调整成指明导入的符号a和b，g()的符号d，以及全局符号a都是g()中可见的符号。为了回到以前的状态，即图a)表示的状态，只要从块栈中弹出栈顶登记项，并从符号栈中弹出它所指登记项之上的所有符号项，并调整散列链反映这些已删除的符号。因为我们在进入g()时并没有为闭作用域重排散列链，因此这就已经将符号表恢复到了原状。

```
program
   var a, b, c, d
   procedure f( )
      var b
      procedure g( )
         var d
         import a, b from P
      end
   end
end
package P
   export a, b
end
```

图3-7 导入代码例子



a)



b)

图3-8 a）含最内层层号的散列全局符号表，b) 在进入g()之后，含局部变量d和导入的符号a和b的一个作用域

为了支持全局符号表结构，我们在3.3节所描述的接口上增加了两个子程序:

```
Encl_Sym_Tab: SymTab × Symbol → SymTab
```

返回说明了第二个参数的最内层的符号表，如果无此符号表，则返回nil。

```
Depth_Sym_Tab: SymTab → integer
```

返回给定符号相对当前符号表的深度。此时约定当前符号表的深度为0。

3.5 存储绑定和符号寄存器

存储绑定（storage binding）将变量名转换为地址，即给变量分配地址。这是代码生成之中或之前必须进行的处理过程。在我们的中间语言层次系统中，存储绑定是MIR至LIR转换过程的一部分（参见第4章），它除了将名字转换为地址之外，还将MIR的赋值展开为取数、运算和存储指令，并且展开调用为一系列的指令。

每一个变量都被指定一个适合于其存储类的地址，或更确切地说，是指定一种寻址方法。

bbs.theithome.com

超星公司制作 请尊重作者版权

我们采用后一个术语。因为，给过程内的局部变量指定的不是固定的机器地址（或相对于一个模块的基址的固定地址），而是相对某个寄存器的偏移来访问的一个栈地址，这个寄存器的值通常随过程调用而改变。

对于存储绑定，变量分为四种主要的类别：全局变量、全局栈变量、栈变量和栈静态变量。在那些允许导入变量或导入整个作用域的语言中，我们还必须考虑导入变量或导入作用域的情况。

全局变量和具有静态存储类的变量通常分配的是固定的可重定位地址，或是相对一个基址寄存器的偏移，这个基址寄存器称为全局指针[⊖]。有些RISC编译器，如MIPS公司为MIPS体系结构提供的编译器，对全局变量尽可能多地使用相对基址寄存器的偏移地址，以便使所有落在有效范围内的全局变量都能用单条指令来访问。栈变量分配的是相对栈指针或帧指针的偏移地址，因此它们可以随过程的调用而出现或消失，而且它们的位置也会随调用不同而改变。

53
~
54

在许多语言中，堆对象的空间是动态分配的，并通过存储分配子程序设置的指针来访问。但是，在有些语言中，如LISP，这种对象可以是“interned”，即这些空间有名字，并可通过名字直接访问它们。

另一种处理栈变量（在有些情况下也用于全局变量）的方法是给它们分配寄存器而不是存储单元。当然，存放在寄存器中的变量通常不能通过索引来寻址，因此这种方法不能用于数组。同样，在代码生成之前或代码生成过程中，一般也不能给很多变量分配寄存器，因为只有数量有限的可用寄存器。但是我们可以给标量变量指定符号寄存器，这种符号寄存器没有个数的限制，它只是一个名字。然后，在编译过程的较后阶段再为这个名字分配真实的寄存器或存储单元。采用图着色全局寄存器分配算法（在16.3节讨论）的编译器就使用了这种方法。符号寄存器的分配简单地通过累加计数器来实现，第一个变量分配s0，下一个为s1，依次类推。寄存器分配过程还能够将已分配存储单元的若干变量封装到几个寄存器中，并由此删除相关的存储器访问，就像16.4节基于优先级的图着色方法所做的那样。

图3-9给出了一个名为Bind_Local_Vars()的ICAN子程序。该程序使用3.3节和3.4节描述的符号表管理子程序来给变量绑定存储单元。符号表项中至少包含表3-1中描述的域。

```
procedure Bind_Local_Vars(symtab,Initdisp)
   symtab: in SymTab
   Initdisp: in integer
begin
   symclass: enum {local, local_static}
   symbasetype: Type
   i, symsize, staticloc := 0, stackloc := Initdisp,
      symnelts: integer
   s := nil: Symbol
   while More_Syms(symtab,s) do
      s := Next_Sym(symtab,s)
      symclass := Get_Sym_Attr(symtab,s,class)
      symsize := Get_Sym_Attr(symtab,s,size)
      symbasetype := Get_Sym_Attr(symtab,s,basetype)
      case symclass of
local:   if symbasetype = record then
            symnelts := Get_Sym_Attr(symtab,s,nelts)
            for i := 1 to symnelts do
```

图3-9 局部变量存储绑定子程序

⊖ 在位置无关代码中使用后一种方法，参见5.7节。

bbs.theithome.com

```
                symsize := Get_Sym_Attr(symtab,s,⟨i,size⟩)
                || allocate local symbols at negative offsets
                || from the frame pointer
                stackloc -= symsize
                stackloc := Round_Abs_Up(stackloc,symsize)
                Set_Sym_Attr(symtab,s,reg,"fp")
                Set_Sym_Attr(symtab,s,⟨i,disp⟩,stackloc)
            od
        else
            stackloc -= symsize
            stackloc := Round_Abs_Up(stackloc,symsize)
            Set_Sym_Attr(symtab,s,reg,"fp")
            Set_Sym_Attr(symtab,s,disp,stackloc)
        fi
local_static:
        if symbasetype = record then
            symnelts := Get_Sym_Attr(symtab,s,nelts)
            for i := 1 to symnelts do
                symsize := Get_Sym_Attr(symtab,s,⟨i,size⟩)
                || allocate local static symbols at positive offsets
                || from the beginning of static storage
                staticloc := Round_Abs_Up(staticloc,symsize)
                Set_Sym_Attr(symtab,s,⟨i,disp⟩,staticloc)
                staticloc += symsize
            od
        else
            staticloc := Round_Abs_Up(staticloc,symsize)
            Set_Sym_Attr(symtab,s,disp,staticloc)
            staticloc += symsize
        fi
    esac
  od
end     || Bind_Local_Vars

procedure Round_Abs_Up(m,n) returns integer
  m, n: in integer
begin
  return sign(m) * ceil(abs(float(m)/float(n))) * abs(n)
end     || Round_Abs_Up
```

图3-9 （续）

Bind_Local_Vars()给每个静态变量指定一个位移，给每个分配在栈中的变量指定一个位移，并用帧指针作为其基址寄存器。对于记录，则从第一个元素到最后一个元素，依次给每个元素指定位移和基址寄存器。initdisp的值是在栈帧内能够分配的第一个位置的位移。我们假定initdisp、栈帧的基址以及静态存储区都是双字对齐的。注意，这里同后面第5章所讨论的一样，我们给局部变量分配的是相对fp的负偏移，给静态变量分配的是正偏移，但忽略了5.1节所讨论的封装记录的可能性。Round_Abs_Up()用来保证适当的边界对齐。函数abs()返回其参数的绝对值，函数ceil返回大于或等于其参数的最近整数。符号寄存器的分配实际上与此相同，而全局变量的处理也类似，不同的只是它可能需要跨多个编译单元，这取决于源语言的结构。

图3-10a中的MIR代码段是一个存储绑定的例子，其中a是全局整变量，b是局部整变量，

bbs.theithome.com

而c[0..9]是一个局部整数组变量。令gp是指向全局区的寄存器，fp是栈的帧指针。于是可以分配gp之后8字节位移的地址给a，分配fp-20给b，fp-60给c（注意，取c[1]导致从位置fp-56取数；这是因为c[0]被分配在fp-60并且c[]的元素的字长为4字节）；在这段MIR代码中给每一个变量绑定地址将产生图3-10b所示的LIR代码（当然，第4条指令是冗余的，因为它装入的值是刚由前一条指令存入的。MIR-LIR转换器能识别这种情形，否则这一任务可遗留给后面的优化，如8.11节所述）。 55

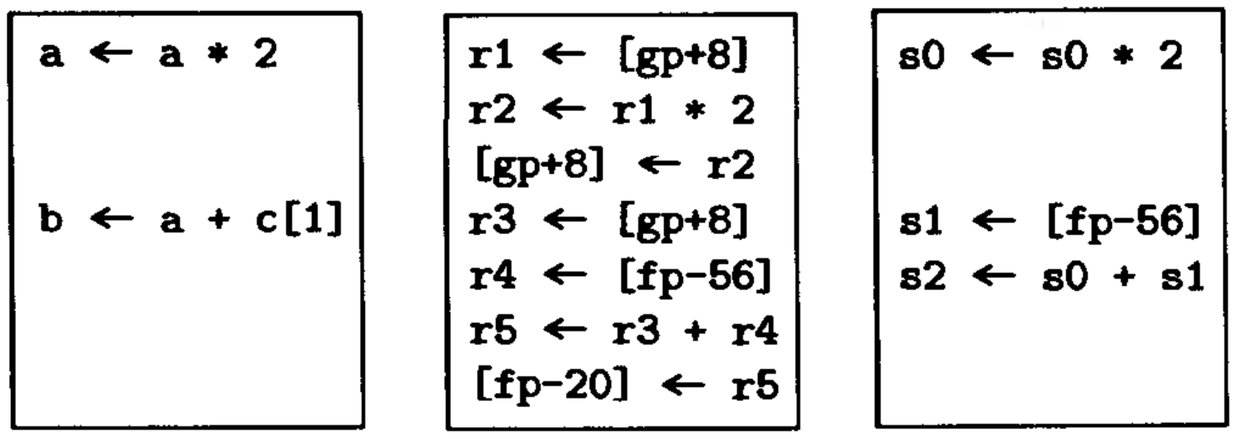


图3-10 a) MIR代码段和到LIR的两种转换，其中b) 将变量名与存储位置进行了绑定，c) 将简单变量与符号寄存器进行了绑定

如果使用符号寄存器，b将得到一个符号寄存器，比如说s2，全局变量a也可以得到一个，这取决于寄存器分配器是否给全局变量分配符号寄存器。我们这里假定它分配，所产生的LIR代码如图3-10c所示。注意，没有给c[1]分配符号寄存器，因为它是一个数组元素。

将局部变量分配到栈帧有若干种可能的方法。我们可以简单地为变量连续分配栈帧内的单元，每个变量都有足够容纳它们的空间，但这要求将变量放置在适合于快速访问它们的存储边界上——例如，如果将一字大小的对象放置在半字边界上，则在最坏情况会需要两条取半字的取数指令、一条移位指令和一个“或”操作，而不是一条取整字的指令。如果没有取半字的指令，如在Alpha体系结构中，还会需要“与”操作。通过保证每一个栈帧开始于双字边界，并在对象之间保留间隙，使得它们开始于适当的边界，我们可以解决这一问题。但这样做浪费了空间，这可能是严重的，因为RISC机器的取数和存储指令只提供较短的偏移，而CISC机器的指令可能需要使用较长的偏移，并且可能影响高速缓存的性能。 56

通过按变量所需要的对齐方式由大至小地对它们排序，并保证每一个栈帧都对齐在提供最有效访问的存储边界上，我们可以实现更好的存储绑定。如果栈帧的开始是双字对齐的，我们可以首先从栈帧起始处安排所有要求双字对齐的对象，后面接着安排要求字对齐的对象，然后是半字对象，最后是字节对象，这就保证了每一个变量都在适当的边界上。例如，已知图3-11所示的C变量声明，我们可以按它们的声明顺序存储它们，如图3-12a所示，也可以按大小排序方式存储它们，如图3-12b所示。注意，对它们排序不仅使得访问它们更快，而且节省空间。这种排序是安全的，因为我们知道的语言定义中，没有一种语言有依赖于局部变量存储安排的语义。

```
int i;
double float x;
short int j;
float y;
```

图3-11 C局部变量声明

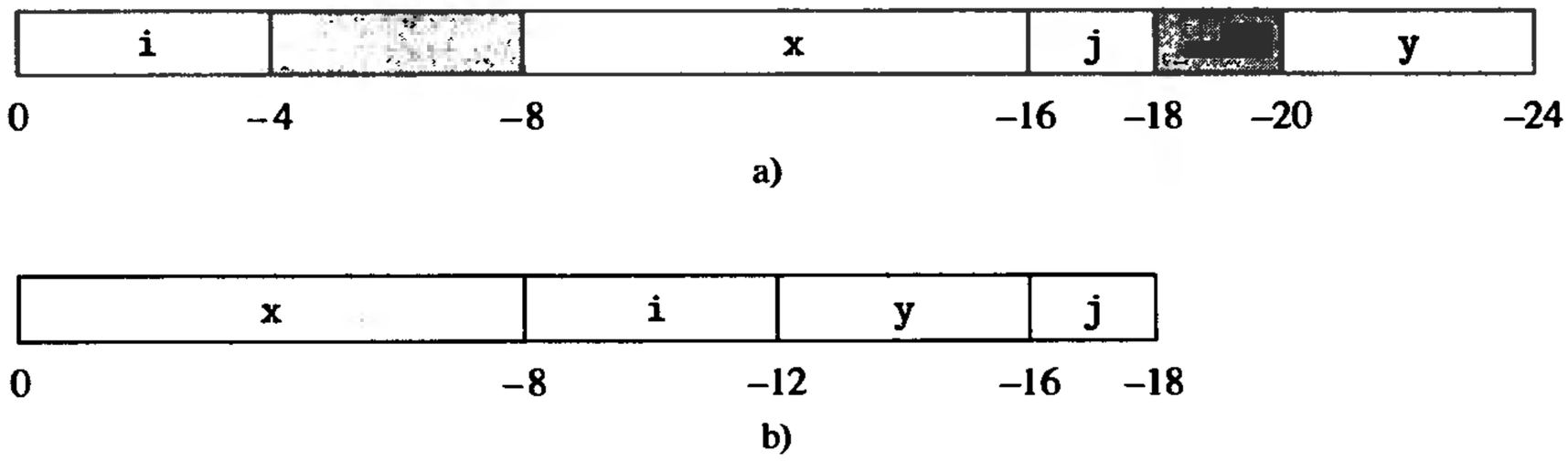
第三种方法是按大小排序，但先分配最小的，并遵循边界对齐要求。这可能需要比前一种方法稍多一点的空间，但对于大栈帧而言，它能使更多的变量用较短的偏移来访问。

如何存储较大的局部数据结构（如数组）需要更多的考虑。我们可以直接将它们存储在栈帧内，有些编译器就是这样做的，但这可能导致访问数组元素和其他变量所需要的偏移量超过指令直接数域所能表示的值。注意，如果把较大的对象放在栈帧起始处，则其他的变量就会需

bbs.theithome.com

越是公司制作请尊重作者版权

要相对fp的较大的偏移；而如果把它们放在栈帧的尾部，相对sp又会发生同样的情况。我们本可以分配第二个基寄存器（或者更多寄存器）来扩充栈帧，但即使那样也不能使它足够大。一种可选的方法是分配较大的对象于栈帧的中间（或其他地方），并存储指向它们的指针于栈帧内相对fp较小的偏移处。为了访问一个数组，这会需要额外的一条取数指令，但这条取数指令的开销可以被多次的数组访问而冲销，尤其是当数组在一个或多个循环之内频繁使用时。



57
~
58

图3-12 图3-11所示声明的栈帧安排，a) 无排序但有边界对齐，b) 排序（偏移以字节为单位）

3.6 生成取数和存数指令的方法

为了具体起见，从现在起我们假定生成32位SPARC-V9系统的汇编代码，但与该系统不同的是，我们使用平面寄存器文件而不是寄存器窗口。

下面的过程生成取值到寄存器的取指令和将计算好的值存储至存储器适当位置的存指令。它们也可能会修改对应的符号表项，以反映每一个值的位置，即这个值是否在寄存器中以及在哪一个寄存器，并在需要时生成整数寄存器与浮点寄存器之间的传送指令，以便当使用这些值时它们已经在所需要类型的寄存器中。

```
Sym_to_Reg: SymTab × Var → Register
```

生成一条从变量的存储位置取数至适当类型寄存器的取数指令，寄存器可能是一个寄存器、两个寄存器或四个寄存器，并返回第一个寄存器的名字。此过程用到的全局数据类型和结构如图3-13所示；Sym_to_Reg()的一个直观版本如图3-14所示，该过程使用的附属例程如图3-15所示。变量GloSymtab以全局符号表作为其值，变量StaticLinkOffset存放当前过程的静态链相对寄存器fp的偏移。该过程的代码用到了下面的函数：

1. Locate_Sym(*symtab*，*v*)，如果变量*v*在符号表*symtab*中，返回true；否则返回false（参见3.3节）。

2. Encl_Sym_Tab(*symtab*，*v*)从*symtab*向外查找，直至找到含有变量*v*的符号表并返回此符号表；如果没有找到，返回nil（参见3.4节）。

3. Depth_Sym_Tab(*symtab1*，*symtab*)返回从当前符号表*symtab*到包含在其内的符号表*symtab1*之间的深度差（参见3.4节）。

4. Get_Sym_Attr(*symtab*，*v*，*attr*)返回变量*v*在符号表*symtab*中属性*attr*的属性值（参见3.3节）。

5. Short_Const(*c*)如果*c*能用13位（SPARC短常量操作数的长度）来表示，返回true；否则返回false。

6. Gen_Inst(*opc*，*opds*)输出一条具有操作码*opc*和参数表*opds*的指令。

7. Reg_Char(*reg*)将Register操作数*reg*转换为对应的字符串。

8. Find_Opcode(*stype*)返回数组LdStType中具有类型*stype*的登记项的索引。

bbs.theithome.com

超星公司制作 请尊重作者版权

Sym_to_Reg_Force : SymTab × Var × Register

生成从变量的存储位置取数至所命名寄存器的取数指令，寄存器可能是一个寄存器、两个寄存器或四个寄存器。这个例程可以强制过程参数至适当的寄存器。

Alloc_Reg: SymTab × Var → Register

分配一个、一对或两对适当类型的寄存器来存放参数指定变量的值，设置此变量的符号表项的reg域，除非它已经分配过寄存器，在两种情况下均返回第一个寄存器的名字。 59

Reg_to_Sym: SymTab × Register → Var

生成将第二个参数的值（一个寄存器名）存至变量存储单元的存储指令，图3-14给出了Reg_to_Sym()的直观版本，它使用了图3-13中给出的全局类型和数据结构，以及图3-15中的附属例程。

Alloc_Reg_Anon: enum{int, flt} × integer → Register

分配一个、一对或两对适当类型的寄存器（根据第二个参数的值，它可以是1、2或4），并返回第一个寄存器的名字。与Alloc_Reg不同，它并不使寄存器与符号相连。

Free_Reg: Register → ∅

将参数给定的寄存器送回到可用寄存器池。

```
SymType = enum {byte, uns_byte, short, uns_short, int,
   uns_int, long_int, uns_long_int, float, dbl_float,
   quad_float}
LdStType = array [1··11] of record
                {type: SymType,
                 LdOp, StOp: CharString}
LdStType :=
   || types, load instructions, and store instructions
   [⟨type:byte,          LdOp:"ldsb",  StOp:"stsb"⟩,
    ⟨type:uns_byte,      LdOp:"ldub",  StOp:"stub"⟩,
    ⟨type:short,         LdOp:"ldsh",  StOp:"stsh"⟩,
    ⟨type:uns_short,     LdOp:"lduh",  StOp:"stuh"⟩,
    ⟨type:int,           LdOp:"ldsw",  StOp:"stsw"⟩,
    ⟨type:uns_int,       LdOp:"lduw",  StOp:"stuw"⟩,
    ⟨type:long_int,      LdOp:"ldd",   StOp:"std"⟩,
    ⟨type:uns_long_int,  LdOp:"ldd",   StOp:"std"⟩,
    ⟨type:float,         LdOp:"ldf",   StOp:"stf"⟩,
    ⟨type:dbl_float,     LdOp:"lddf",  StOp:"stdf"⟩,
    ⟨type:quad_float,    LdOp:"ldqf",  StOp:"stqf"⟩]
GloSymtab: SymTab
StaticLinkOffset: integer
Depth: integer
```

图3-13 用于生成存取指令的全局类型和数据结构

```
procedure Sym_to_Reg(symtab,v) returns Register
   symtab: in SymTab
   v: in Var
begin
   symtab1: SymTab
   symdisp: integer
   Opcode: CharString
```

图3-14 取一个变量的值到一个寄存器、一对寄存器、两对寄存器，以及将这些寄存器中的值保存到变量的例程

bbs.theithome.com

```
    symreg: Register
    symtype: SymType
    symtab1 := Find_Sym_Tab(symtab,v)
    if Get_Sym_Attr(symtab1,v,register) then
       return Get_Sym_Attr(symtab1,v,reg)
    fi
    symtype := Get_Sym_Attr(symtab1,v,type)
    Opcode := LdStType[Find_Opcode(symtype)].LdOp
    symreg := Alloc_Reg(symtab1,v)
    symdisp := Get_Sym_Attr(symtab1,v,disp)
    || generate load instruction and return loaded register
    Gen_LdSt(symtab1,Opcode,symreg,symdisp,false)
    return  symreg
end     || Sym_to_Reg

procedure Reg_to_Sym(symtab,r,v)
    symtab: in SymTab
    r: in Register
    v: in Var
begin
    symtab1: SymTab
    disp: integer
    Opcode: CharString
    symtype: SymType
    symtab1 := Find_Sym_Tab(symtab,v)
    symtype := Get_Sym_Attr(symtab1,v,type)
    Opcode := LdStType[Find_Opcode(symtype)].StOp
    symdisp := Get_Sym_Attr(symtab1,v,disp)
    || generate store from register that is the value of r
    Gen_LdSt(symtab1,Opcode,r,symdisp,true)
end     || Reg_to_Sym
```

图3-14 （续）

```
procedure Find_Sym_Tab(symtab,v) returns SymTab
    symtab: in SymTab
    v: in Var
begin
    symtab1: SymTab
    || determine correct symbol table for symbol
    || and set Depth if neither local nor global
    Depth := 0
    if Locate_Sym(symtab,v) then
       return symtab
    elif Locate_Sym(GloSymtab,v) then
       return GloSymtab
    else
       symtab1 := Encl_Sym_Tab(symtab,v)
       Depth := Depth_Sym_Tab(symtab1,symtab)
       return symtab1
    fi
end     || Find_Sym_Tab

procedure Find_Opcode(symtype) returns integer
```

图3-15 生成存取指令时用到的附属例程

版权所有 翻印必究

bbs.theithome.com

```
            symtype: in SymType
         begin
            for i := 1 to 11 do
               if symtype = LdStType[i].type then
                  return i
               fi
            od
         end    || Find_Opcode
procedure Gen_LdSt(symtab1,OpCode,reg,symdisp,stflag)
   symtab1: in SymTab
   OpCode: in CharString
   reg: in Register
   symdisp: in integer
   stflag: in boolean
begin
   i: integer
   reg1, regc: CharString
   if symtab1 = GloSymtab then
      || set reg1 to base address
      reg1 := "gp"
      regc := Reg_Char(reg)
   else
      reg1 := "fp"
      if stflag then
         reg := Alloc_Reg_Anon(int,4)
      fi
      regc := Reg_Char(reg)
      || generate loads to get to the right stack frame
      for i := 1 to Depth do
          Gen_Inst("lduw","[" ⊕ reg1 ⊕ "+"
             ⊕ StaticLinkOffset ⊕ "]," ⊕ regc)
          reg1 := regc
      od
      if stflag then
         Free_Reg(reg)
      fi
fi
|| generate load or store
if Short_Const(symdisp) & stflag then
   Gen_Inst(Opcode,regc ⊕ ",[" ⊕ reg1 ⊕ "+" ⊕ Int_Char(symdisp) ⊕ "]");
   return
elif Short_Const(symdisp) then
   Gen_Inst(Opcode,"[" ⊕ reg1 ⊕ "+" ⊕ Int_Char(symdisp) ⊕ "]," ⊕ regc);
   return
fi
|| generate sethi and load or store
Gen_Inst("sethi","%hi(" ⊕ Int_Char(symdisp) ⊕ ")," ⊕ reg1)
if stflag then
   Gen_Inst(Opcode,regc ⊕ ",[" ⊕ reg1 ⊕ "+%lo(" ⊕ Int_Char(symdisp)
      ⊕ ")]")
else
   Gen_Inst(Opcode,"[" ⊕ reg1 ⊕ "+%lo(" ⊕ Int_Char(symdisp) ⊕ ")],"
      ⊕ regc)
fi
end    || Gen_LdSt
```

图3-15 （续）

bbs.theithome.com

与简单地在使用一个值之前取它到寄存器和计算好一个值便立即存储它的方法不同，我们不难提供这些存/取寄存器的例程的更成熟版本。第一种改进是对每一个寄存器的内容进行追踪，如果需要的值已在适当类型的寄存器中，就无需冗余地取它而直接使用该寄存器；如果值已经在一个寄存器中，但是类型不符，则当目标体系结构支持整型和浮点型寄存器之间的数据移动时，可以生成传送指令而不是取数指令。如果寄存器已用完，我们可以选择一个来重新使用（我们将这一任务分派给Alloc_Reg()）。类似地，我们不是每计算一个值后就立即存储它，而是推迟到基本块结束时或寄存器用完时再存储。如果需要的寄存器个数多于可用的寄存器个数，我们还可以采用这样的策略来实现Reg_to_Sym()，即尽可能合理地及时存储每一个计算出的量，以便使得需要的存数指令条数和使用的寄存器个数都保持最少。

60

后两种策略还可以更进一步，我们可以通过考虑一个基本块入口处各个寄存器的状态来对多个基本块进行寄存器分配。如果一个基本块只有一个前驱，基本块入口时的寄存器状态就是其前驱出口时的状态。如果它有多个前驱，比较恰当的选择是取这些前驱出口寄存器状态的交集。这种寄存器分配方法对于诸如if-then-else的结构是相当有效的，能使存取指令最少；但对循环没有帮助，因为循环体的前驱之一是循环体本身。要做得更好，我们需要第16章讨论的全局寄存器分配技术，那一章更详细地讨论了这里刚描述的局部寄存器分配方法。

另一种选择是在代码生成时分配符号寄存器，在需要时由全局寄存器分配器为这些符号寄存器分配存储单元。

对于导入到当前作用域中的闭作用域中的变量，生成它们的存取指令相对容易，只要给每一个符号表（不是符号表项）增加若干属性指出它所表示的是哪一种作用域，以及它用哪个寄存器（即非fp和gp的寄存器）来访问其内的符号即可。

61~63

3.7 小结

本章讨论了如何构造适应现代程序设计语言特点的局部和全局符号表，以及在实现这些语言的编译器中如何更有效地使用它们等问题。

我们首先讨论了存储类和可见性或作用域规则，然后讨论了符号属性以及如何构造局部符号表，介绍了一种组织全局符号表的方法，这种方法支持诸如Modula-2、Ada和C++等语言中的导出和导入作用域。

接着，给出了全局和局部符号表的程序设计接口，这些接口使得人们可以随意按自己的需要来构造全局和局部符号表，只要满足这些接口规定就可以。在此之后，我们探讨了给变量绑定存储单元和符号寄存器的有关问题，并且给出了若干ICAN例程，这些例程生成与变量符号属性和前面提到的符号表接口相一致的关于变量的存取指令。

这一章学到的主要内容是：（1）存在多种实现符号表的方法，有一些方法更有效，（2）使符号表的操作具有效率是非常重要的，（3）仔细考虑所使用的数据结构能得到既相对简单又高效的实现，（4）通过一种十分特殊的接口，可以对编译器的其他部分隐藏符号表的结构，（5）存在若干种生成存取指令的方法，这些方法在追踪寄存器的内容和最小化存储交换方面，其效率上各有高低。

3.8 进一步阅读

[MitM79]中描述了程序设计语言Mesa。

[Knut73]提供了丰富的有关设计散列函数的信息和技术。

本节描述的作用域模型基于Graham、Joy和Roubine写的论文[GraJ79]。

bbs.theithome.com

超星公司制作 请尊重作者版权

3.9 练习

3.1 给出实际程序设计语言中与下表列出的每一种作用域和生命期相匹配的例子。对于每一种情形，指明其语言并给出代码例子。 64

	整个程序	文件或模块	过程集	过程	块
整个执行期					
模块执行期					
过程的所有执行期					
过程的单次执行期					
块执行期					

3.2 写一个ICAN例程Struct_Equiv(*tn*1, *tn*2, *td*)，它以用ICAN表示的两个类型名*tn*1和*tn*2，以及一个Pascal类型定义集合*td*作为其参数，其中*tn*1和*tn*2要么是简单类型，要么是在*td*中定义的类型之一。如果*tn*1和*tn*2在结构上等价，该例程返回true；如果它们不等价，则返回false。*td*是一个ICAN的偶对表，偶对的每一个成员由一个类型名和一个类型表示组成，例如，图3-1b的类型定义可用如下列表来表示:

```
[⟨t1,⟨array,2,[⟨0,5⟩,⟨1,10⟩],integer⟩⟩,
 ⟨t2,⟨record,3,[⟨t2a,integer⟩,⟨t2b,⟨pointer,t2⟩⟩,
        ⟨t2c,⟨array,1,[⟨1,3⟩],char⟩⟩]⟩⟩,
 ⟨t3,⟨array,1,[⟨1,100⟩],t2⟩⟩]
```

两个类型在结构上等价是指: (1) 它们都是同一种简单类型，或者(2) 除了对组成它们的类型可能使用不同的类型名之外，它们的定义相同。例如，给定Pascal类型定义

```
t1 = integer;
t2 = array [1..10] of integer;
t3 = array [1..10] of t1;
t4 = record f1: integer; f2: ↑t4 end;
t5 = record f1: t1;      f2: ↑t4 end;
t6 = record f1: t2;      f2: ↑t4 end;
```

偶对t1和integer、t2和t3、t4和t5，每一对在结构上都是等价的，而t6与所有其他类型不等价。

3.3 写一个计算栈变量偏移量的子程序，其中偏移量是相对于栈帧指针的，并且栈变量按如下不同顺序排列:

(a) 按符号表给定的顺序; (b)按数据长度的顺序，最长的在前; (c)按数据长度的顺序，最短的在前。

3.4 写出(a) Sym_to_Reg()、(b)Reg_to_Sym()和(c)Alloc_Reg()的寄存器追踪版本。

3.5 设计一个ICAN符号表项，它至少包括3.2节开始处描述的域; 并用ICAN写出例程Get_Sym_Attr()和Set_Sym_Attr()。

3.6 利用上一练习的符号表项设计，设计一个局部符号表结构，并用ICAN实现Insert_Sym()、Locate_Sym()、Next_Sym()和More_Syms()。

3.7 利用上一练习的局部符号表设计，设计一个全局符号表结构，并用ICAN实现支持闭作用域的New_Sym_Tab()、Dest_Sym_Tab()、Encl_Sym_Tab()和Depth_Sym_Tab()。 65 ~ 66

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第4章 中间表示

本章探讨中间代码表示和设计中涉及的问题，如同我们即将看到的一样，中间代码结构有许多可行的选择。

在讨论了各种中间代码形式和它们各自的优点之后，我们最终需要选择使用一种具体的中间代码设计，以便用它来表示有关优化和代码生成的问题。我们主要使用的中间语言称为中级中间表示，简称MIR。本章还描述一种级别稍高一点的形式，称为高级中间表示，简称HIR，以及一种级别稍低一点的形式，称为LIR的低级中间表示。基本MIR适合于大多数优化（LIR也如此）；HIR则用于依赖分析和基于依赖分析的某些代码转换；LIR用于那些需要明确指明寄存器和地址的优化。

4.1 与中间语言设计有关的问题

中间语言设计在很大程度上是艺术而不是科学，可以应用若干设计原则，也有许许多多的经验可以借鉴，但不管怎样都要决定是否采用一个现存的表示。如果不用现存的语言，在新的设计中就会有许多需要决定的问题。如果使用现存的语言，就需要考虑它对新编译器的各种适应性问题——既包括被编译的语言也包括目标机体系结构，还需要考虑实现它所产生的移植代价与重用现有设计和代码所固有的开销节省之间的权衡。新的中间语言形式是否适合所要实现的优化种类和级别也是需要考虑的问题。对于某种给定的中间语言，有的优化可能很难实现，而另一些优化则可能花费比用其他中间表示更长的时间。例如，UCODE中间语言（PA-RISC和MIPS编译器所使用的一种中间语言形式）就设计得非常适合于那种用栈对表达式求值的体系结构，因为栈就是其表达式求值模型。但是它不太适合那种没有计算栈，而是具有大量寄存器集合，并且需预先将数据取至寄存器，然后将结果存回内存的体系结构。因此，Hewlett-Packard的编译器和MIPS的编译器为了进行优化，都将UCODE转换为另一种形式。HP将它转换为一种级别非常低的表示，而MIPS在优化器中将它转换为一种中间级别的三元式形式，并基于这种形式进行优化，然后，再将这种形式转换回UCODE进行代码生成。

如果要设计一种新的中间表示，其中涉及的问题包括：它的级别（更具体地，即它依赖机器的程度）、它的结构、它的表达能力（即覆盖适当种类的语言所需要的各种结构）、它对某种优化或某些特殊优化的适应性，以及它对目标机体系结构或多种目标机体系结构生成代码的适应性。

一个编译器也可能使用多种中间形式，在编译过程中从一种形式转换至另一种形式，以便保护已有的技术投资，或能够在相应的时间内完成适合于不同级别形式的功能。一个编译器也可以有多级的中间形式。前者是Hewlett-Packard在其PA-RISC芯片编译器上的做法。该编译器首先将程序转换成UCODE形式，这是前一代HP3000系列所使用的中间形式（UCODE非常适合它们，因为它们是栈式机器）。然后，它们经过两个步骤而被转换成一种称为SLLIC[⊖]的非常低级的表示，所有优化实质上都是基于这种表示实现的。这种做法的好处是，既能在这种非常低的级别上有效地实现优化，同时保存了语言前端和中间阶段所收集的关于代码的信息。

⊖ SLLIC是Spectrum Low-Level Intermediate Code的缩写。Spectrum是PA-RISC在其开发时期的名字。

bbs.theithome.com

在后一种方法中，典型地，对某些语法结构可能会有多种表示，并且每一种表示可能只适合于某一种特定的任务。这种情况的一个常见例子是分别用下标表（相对高级的形式）和线性表达式（低级形式，它明确地计算相对数组基地址或其他元素地址的存储偏移）来表示下标表达式。表的形式要适合进行依赖分析（参见9.1节），有多种优化要基于依赖分析。而线性形式适合于常数折叠、强度削弱、循环不变代码外提，以及其他更为基础的优化。

例如，使用后面4.6节所描述的表示，对于具有声明

```
float a[20][10]
```

的C表达式a[i][j+2]，可以表示成图4-1a所示的高级形式（HIR），图4-1b所示的中级形式（MIR）和图4-1c所示的低级形式（LIR）。高级和中级形式中变量名的使用指明该变量的符号表项，中级形式中的一元运算符addr和“*”分别表示取一个对象的地址和指针间接引用。高级形式指明它所引用的是一个具有两个下标的数组元素，其中第一个下标是i，第二个下标是表达式j+2。中级形式计算

68

```
(addr a)+ 4 * ( i * 20 + j + 2)
```

作为该数组元素的地址，并且将此地址中的值取至临时变量t7。低级形式从存储器取i和j的值（假定分别在相对栈帧指针寄存器fp偏移为-4和-8的位置），计算此数组元素在数组内的偏移，然后将该数组元素的值取到浮点寄存器f1。

```
t1 ← a[i,j+2]
```

a)

```
t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t4
t7 ← *t6
```

b)

```
r1 ← [fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3 * 20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp - 216
f1 ← [r7+r6]
```

c)

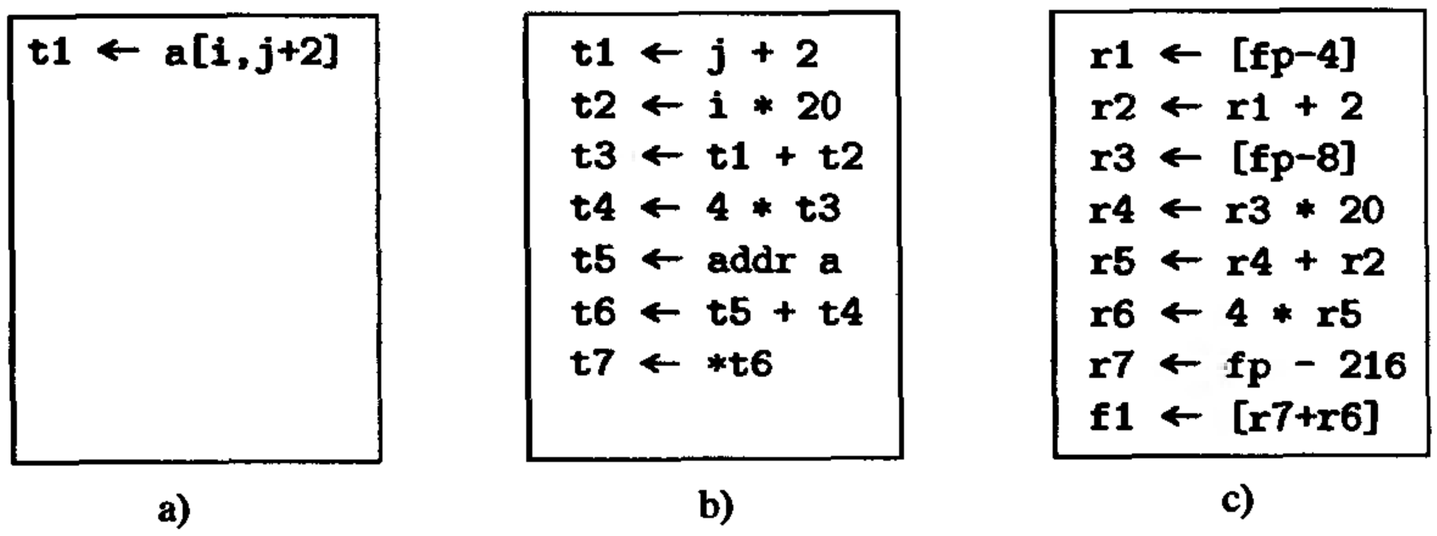


图4-1 C数组引用的a) 高级、b) 中级和c) 低级表示

注意，为了提高编译速度和代码的紧凑性，中间代码的表示在编译器内部一般是二进制形式，而符号（如变量）则表示为指向符号表项的指针。在本书的例子中，我们使用便于阅读的外部正文表示。大多数编译器都有为了满足编译器设计者自己的需要而开发的中间代码的调试输出。在某些情况下，如在那些保存中间代码以便能够跨模块过程集成（参见15.2节）的编译器中，它不仅是用于调试目的的打印形式，而且还是可以被再读回去并使用的形式。在设计这种外部表示时遇到的三个主要问题是：(1) 在外部表示中如何以位置无关的方式来表示指针？(2) 如何以惟一的方式来表示在模块中出现的编译生成的对象，如临时变量？(3) 如何使得外部表示既是紧凑的又能被快速读写？一种可能的途径是使用两种外部表示——基于字符的表示给人阅读，而二进制形式用于支持跨模块过程集成和其他的过程间分析和优化。在二进制形式中，可通过使指针保持与它所引用对象的相对位置而使指针与位置无关。为了使得每一个临时变量对使用它的模块是惟一的，可以让临时变量名中包含模块名。

4.2 高级中间语言

高级中间语言几乎总是用于编译过程的最早阶段，或用于编译之前的预处理器中。在前一种情况下，它们由编译前端产生，并且一般在此之后不久便被转换成低级形式。在后一种情况下，它们常常被转换回到原先的语言或另一种语言的源代码形式。

69

bbs.theithome.com

超星公司制作 请尊重作者版权

最常见的一种高级中间语言形式是抽象语法树，它保持了明显的程序结构，并含有能够将它重构到源代码形式或其近似形式的足够信息。抽象语法树主要用于语言敏感的或语法制导的程序设计语言编辑器中，在这种编辑器中，它们通常就是程序的标准内部表示。例如，考虑图4-2中的简单C例程和图4-3中该例程的抽象语法树表示。这棵树（包括其中指出变量类型的符号表）提供了重构源代码所需的一切信息（源代码行的布局细节信息除外，但如果需要，它可以作为注解包含进来）。

70

```
int f(a,b)
int a, b;
{   int c;
    c = a + 2;
    print(b,c);
}
```

图4-2 一个小C例程，其抽象语法树在图4-3中给出

将抽象语法树转换为中级中间代码形式（如下一节讨论的那些形式），只需要由一个知道源语言语义的编译组件对树做一次遍历即可。

高级中间语言的另一种形式是9.1节将讨论的用于依赖关系分析的形式。这种中间语言与树不同，它通常是线性的，但它保留了源语言的某些特征，如数组下标和循环结构（基本上是其源代码的形式）。后面4.6.2节将描述的HIR具有这种高级特征，但也包含了许多中级特征。

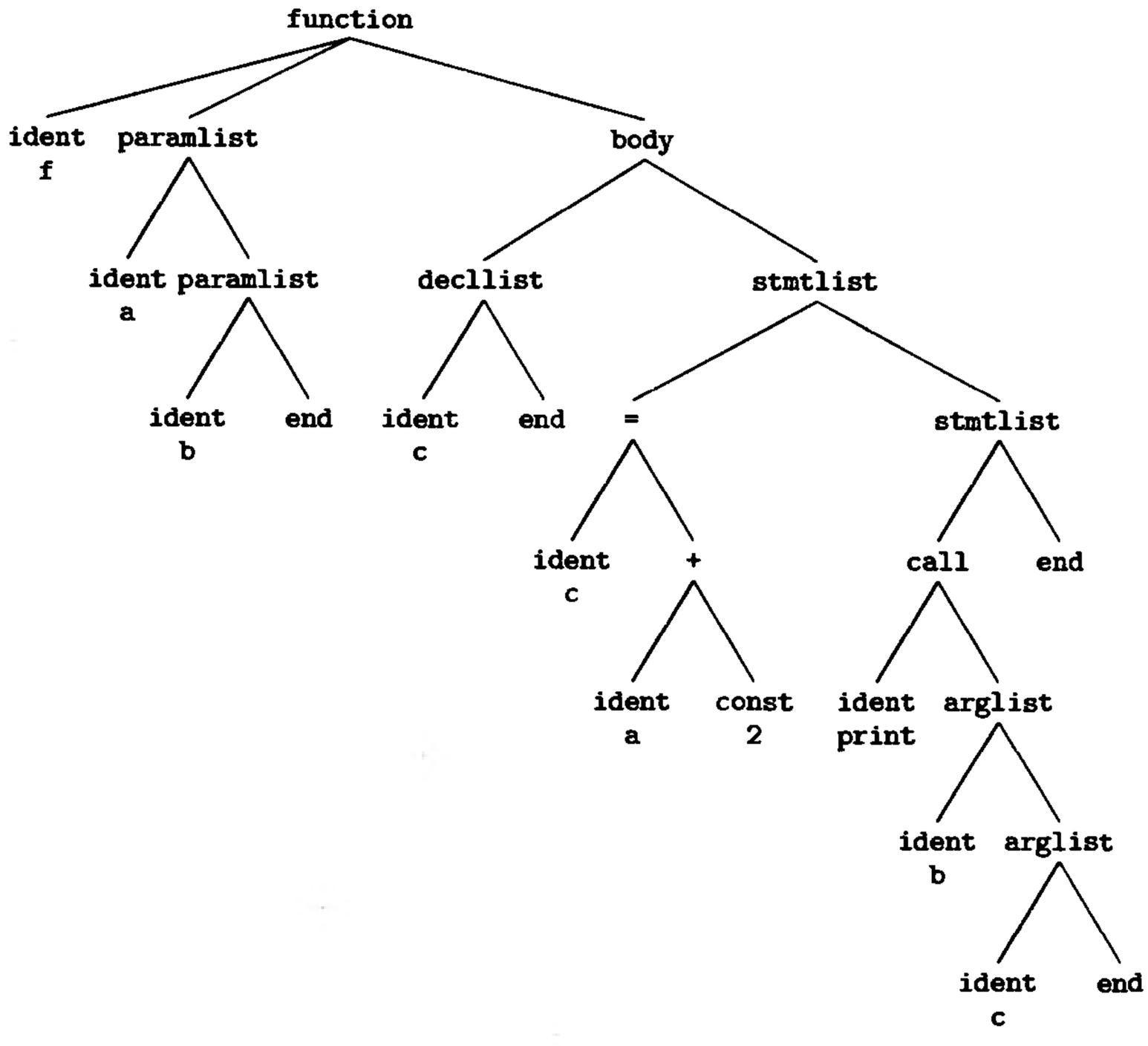


图4-3 图4-2 中的C例程的抽象语法树

4.3 中级中间语言

中级中间语言的设计通常既要能够以一种与语言无关的方式在一定程度上反映源语言集合的特征，又要能够适应多种体系结构并能有效生成它们的机器代码。中间语言能够表示源程序变量、临时变量和寄存器，能够将控制流归约为简单的有条件和无条件分支、调用和返回，还能够用明显的操作来支持块结构和过程。

bbs.theithome.com

我们的MIR（4.6.1节）和Sun的IR（21.1节）都是中间语言的好例子。

中级中间语言适合于多数编译优化，如公共子表达式删除（13.1节）、代码外提（13.2节）和代数化简（12.3节）。

4.4 低级中间语言

低级中间语言与目标机指令几乎是一一对应的，因此也经常与体系结构相关。与目标机指令不一一对应的情形一般有两种。一种发生在中间代码可能生成多种最有效的代码时。例如，低级中间语言可能有一个整数乘法运算符，但目标机体系结构可能没有乘法指令，或者虽有乘法指令，但对于某种操作数组合而言却不是要生成的最好代码选择。另一种情况是中间代码可能只有简单的寻址方式，如寄存器＋寄存器和寄存器＋常量，而目标体系结构有多种复杂的方式，如带缩放的变址或变址寄存器修改方式。在这两种情况下，为中间代码选择要生成的适当指令或指令序列，就成为了编译过程中最终指令选择遍或优化之后的处理遍的功能。使用这种
71 表示既允许对中间代码执行尽可能多的优化，又允许在编译的最后几遍中扩展中间代码指令为代码序列，或将相关的代码组合为更有效的指令。

例如，假设目标体系结构有一条取数指令，它能可选地在取数的同时更新用于取数的那个基地址，但做了这种更新就不能设置机器的条件代码，或者不允许将它用在一条加并且（有条件）分支指令中，因此，不能用它来对包含它的循环进行计数。于是，一方面我们可能要考虑将那种取数据并增加该数据地址的中间代码组合为一条带有地址增加动作的取数指令。另一方面，如果已确定循环控制变量是一个归纳变量并且它已被删除，我们可能又会需要使修改地址与取数保持独立，以便用它来测试循环结束条件。图4-4说明了这个例子，图4-4a中的MIR分别进行取数、增加所取数的地址、增加循环计数器、测试循环的完成，并根据测试结果进行转移，其中的操作数都是临时变量和常量。图4-4b中的PA-RISC代码用带有地址更新的取字指令取数，这个更新修改r2中的地址使其指向下一个数组元素，然后用直接数加指令来增加循环计数器，并做关闭循环的比较和转移。图4-4c中的代码用一条带变址的取字指令来访问数据，然后用一条直接数加及分支指令来修改地址和关闭循环。

```
L1:   t2 ← *t1
      t1 ← t1 + 4
      . . .
      t3 ← t3 + 1
      t5 ← t3 < t4
      if t5 goto L1
```

a)

```
L1:   LDWM    4(0,r2),r3
      . . .
      ADDI    1,r4,r4
      COMB,<  r4,r5,L1
```

b)

```
L1:   LDWX    r2(0,r1),r3
      . . .
      ADDIB,< 4,r2,L1
```

c)

图4-4 a) 是一段MIR代码，b) 和c) 是分别为它生成的两种可选PA-RISC代码序列

4.5 多级中间语言

我们提到的这些中间语言中，有一些包含了最好视为同一种语言的多级表示的若干特征。例如，Sun的中间语言IR有一些高级特征，如用多个下标来表示一个多下标的数组引用，以及
72 将多个下标线性化为一个偏移量来表示一个多下标数组引用。前一种表示对各种依赖分析（参见9.3节）非常有用，因此适合于向量化、并行化和数据高速缓存优化，而后一种表示对传统

bbs.theithome.com

超星公司制作 请尊重作者版权

的循环优化更敏感。

在这种多级中间语言的另一端，低级SLLIC则包含了整数乘和除运算符，尽管PA-RISC硬件事实上没有基于整数寄存器的整数乘法指令（虽然PA-RISC 1.1提供了对存放在浮点寄存器的整数做整数乘的指令），也没有任何整数除法指令。这样做为那些依赖于识别乘法和除法而进行的优化提供了机会，并由此使得生成最终代码的模块可以决定用最有效的移位和加法指令来实现特定的乘法，而不是用硬件提供的乘法指令来实现它们。

4.6 我们的中间语言：MIR、HIR和LIR

从现在开始，在多数用中间语言表示的例子中，我们使用称为MIR（中级中间表示，读做“meer”）的语言，下一节给出它的描述。在适当的地方，我们使用MIR的增强版本，称为HIR（高级中间表示，读做“heer”），它带有某些高级特征，如用表而不是用相对数组基址偏移的线性表达式来表示带下标的数组引用。相应地，在适当的地方，当希望能具体地表示低级特征（如显式表示寄存器、存储器地址和类似的情况）时，我们使用MIR的一个改编版本，称为LIR（低级中间表示，读做“leer”）。偶尔为了便于表示从一种级别到下一种级别的转换步骤，或使得叙述更明确，我们也在一些程序中混合使用MIR和HIR或LIR。

4.6.1 中级中间表示（MIR）

MIR主要由一张符号表和一些四元式组成，其中四元式由一个运算符和三个操作数组成，只有不多的几个四元式的操作数多于或少于三个。MIR保留了几种特殊的符号用于表示临时变量、寄存器和标号。无论什么情况下，我们都将MIR指令写作赋值语句的形式，用“←”作为赋值运算符。我们用2.1节描述的XBNF表示来表示MIR和其他两种在后面将定义的中间语言HIR和LIR的语法。

在介绍MIR之初，我们首先用XBNF给出MIR程序的语法（见表4-1）。一个程序由一系列程序单元组成，每一个程序单元由可选的标号，后随用begin和end界定的指令序列组成。

表4-1 MIR程序和程序单元的XBNF语法

Program	→	[*Label* :] *ProgUnit**
ProgUnit	→	[*Label* :] begin *MIRInsts* end
MIRInsts	→	{[*Label* :] *MIRInst* [|| *Comment*]}*
Label	→	*Identifier*

接下来在表4-2中给出MIR指令的语法，它隐式地声明了ICAN类型MIRInst，并且描述了它们的语义。MIR指令可以是receive、赋值、goto、if、调用、return或sequence，它也可以带有标号。 73

表4-2 MIR指令的XBNF语法

MIRInst	→	*ReceiveInst* | *AssignInst* | *GotoInst* | *IfInst* | *CallInst*
		| *ReturnInst* | *SequenceInst* | *Label* : *MIRInst*
ReceiveInst	→	receive *VarName* (*ParamType*)
AssignInst	→	*VarName* ← *Expression*
		| *VarName* ←(*VarName*) *Operand*
		| [*] *VarName* [. *EltName*] ← *Operand*
GotoInst	→	goto *Label*

bbs.theithome.com

（续）

IfInst	→	**if** *RelExpr* {**goto** *Label* | **trap** *Integer*}
CallInst	→	[**call** | *VarName* ←] *ProcName* , *ArgList*
ArgList	→	([{*Operand* , *TypeName*} ⋈ ;])
ReturnInst	→	**return** [*Operand*]
SequenceInst	→	**sequence**
Expression	→	*Operand BinOper Operand*
		| *UnaryOper Operand* | *Operand*
RelExpr	→	*Operand RelOper Operand* | [!] *Operand*
Operand	→	*VarName* | *Const*
BinOper	→	+ | - | * | / | **mod** | **min** | **max** | *RelOper*
		| **shl** | **shr** | **shra** | **and** | **or** | **xor** | . | *.
RelOper	→	= | != | < | <= | > | >=
UnaryOper	→	- | ! | **addr** | (*TypeName*) | *
Const	→	*Integer* | *FloatNumber* | *Boolean*
Integer	→	0 | [-] *NZDecDigit DecDigit** | 0x *HexDigit*+
FloatNumber	→	[-] *DecDigit*+ . *DecDigit*+ [E [-] *DecDigit*+] [D]
Boolean	→	**true** | **false**
Label	→	*Identifier*
VarName	→	*Identifier*
EltName	→	*Identifier*
ParamType	→	**val** | **res** | **valres** | **ref**
Identifier	→	*Letter* {*Letter* | *Digit* | _}*
Letter	→	a | ... | z | A | ... | Z
NZDecDigit	→	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
DecDigit	→	0 | *NZDecDigit*
HexDigit	→	*DecDigit* | a | ... | f | A | ... | F

74

receive指令指明接收来自调用例程的参数，它只能作为程序单元的第一条可执行指令，用于指明参数和参数传递所用的规则，即是传值(**val**)，传结果(**res**)，传值得结果（**valres**），还是传地址(**ref**)。

赋值指令或是计算一个表达式并将其值赋给一个变量，或是有条件地将一个操作数的值赋给一个变量，或是赋值给一个由指针指向的对象或结构的成员。前面两种情况赋值的目标是一个变量。在条件赋值中，箭头后括号内的变量的值必须是布尔值，并且仅当其值为**true**时赋值才被执行。表达式的组成可以是一个二元运算符组合两个操作数、一元运算符后跟一个操作数或仅仅是一个操作数。二元运算符可以是下述任意的算术运算符:

```
+   -   *   /   mod   min   max
```

或关系运算符:

```
=   !=   <   <=   >    >=
```

或移位和逻辑运算符:

```
shl   shr   shra   and   or   xor
```

或成员选择运算符:

```
.    *    .
```

一元运算符可以是表4-3中给出的任意符号。

第二种类型的赋值目标由一个可选的间接运算符“*”（它指明通过指针赋值）、一个变量名和一

表4-3 MIR中的一元运算符

符 号	含 义
-	算术减
!	逻辑非
addr	取地址
(*TypeName*)	类型转换
*	指针间接

bbs.theithome.com

个可选的成员选择运算符（“.”之后跟随指定结构的成员名）组成，它可用来将一个值赋给通过指针访问的对象，或赋给一个结构的成员，或赋给同时是两者的一个对象。

goto指令导致控制转移到其目标指出的指令，后者必须在当前过程之内。if 指令测试一个关系或条件，或相反的关系或条件。如果条件或关系满足，导致控制转移。此转移可以与goto指令引起的转移相同，也可以转移到运行时的底层系统（一个“自陷”）。在后一种情况下，它指明一个整型自陷号。 75

调用指令给出被调用过程的名字和实参表，并且可以指明一个存放调用结果的变量（如果它出现的话）。它导致以给定的参数调用指定的过程，并且如果调用成功，其返回值将赋给存放调用结果的变量。

return指令指明执行返回到调用当前过程的那个过程，并且可以包含一个将返回值返回给调用者的操作数。

sequence指令表示中间代码中的一个栅栏。一条含有一至多个操作数的指令，若其中有些操作数带有volatile存储类修饰符，则不能跨越sequence指令移动，不论是向前还是向后移动，由此限制了对含有这种指令的代码进行优化。

标识符由一个字母其后跟随零到多个字母、数字或下划线的序列组成。变量、类型或过程的名字可以是任意标识符，除了如下讨论的那些保留作为标号、临时变量和符号寄存器的标识符之外。以大写字母“L”打头，后随一个十进制非负整数的标识符表示标号，如L0、L32和L7701。我们保留由小写字母“t”后随一个十进制非负整数组成的名字为临时变量名，例如t0、t32和t7701，保留由小写字母“s”后随一个十进制非负整数组成的名字为符号寄存器，保留r0,...,r31和f0,...,f31为真实的硬件寄存器。

整型常量可以用十进制或十六进制表示。十进制整数是“0”或者由可选的负号和一个非0的十进制数字、其后跟随零至多个十进制数字组成。十六进制整数由“0x”后随一系列十六进制数字组成，其中十六进制数字是十进制数字和从“A”到“F”的大写字母，或从“a”到“f”的小写字母。例如，下面都是合法的整数:

```
0 1  3462  -2  -49  0x0  0x137A  0x2ffffc
```

在浮点数中，以“E”开始的部分指出前面的值要乘以10的“E”之后的整数值次幂，可选的“D”指明双精度值，于是，例如

```
0.0  3.2E10  -0.5  -2.0E-22
```

都是单精度浮点数，而

```
0.0D  3.2E102D  -0.5D  -2.0E-22D
```

都是双精度浮点数。

MIR程序中的注释以“||”开始直到当前行的末尾。

在适当的地方我们使用完整的MIR程序，但大部分情况只使用程序段，因为它们能够满足需要。

作为一个MIR的例子，图4-6中的代码对应于图4-5中的两个C过程。

注意，MIR中有些运算符（如min和max）通常不包含在机器指令集中，在我们的中间代

```
void make_node(p,n)
   struct node *p;
   int n;
{  struct node *q;
   q = malloc(sizeof(struct node));
   q->next = nil;
   q->value = n;
   p->next = q;
}

void insert_node(n,l)
   int n;
   struct node *l;
{  if (n > l->value)
      if (l->next == nil) make_node(l,n);
      else insert_node(n,l->next);
}
```

图4-5 两个C过程的例子

bbs.theithome.com

码中包含它们是因为它们在高级语言中频繁出现，并且有些体系结构提供了不使用分支而计算它们的非常有效的方法。例如，对于PA-RISC，MIR指令t1←t2 min t3可以转换为（假定t*i*在寄存器r*i*中）[⊖]

```
make_node:
    begin
        receive p(val)
        receive n(val)
        q ← call malloc,(8,int)
        *q.next ← nil
        *q.value ← n
        *p.next ← q
        return
    end
insert_node:
    begin
        receive n(val)
        receive l(val)
        t1 ← l*.value
        if n <= t1 goto L1
        t2 ← l*.next
        if t2 != nil goto L2
        call make_node(l,type1;n,int)
        return
L2:     t4 ← l*.next
        call insert_node,(n,int;t4,type1)
        return
L1:     return
    end
```

图4-6 图4-5中两个C过程的MIR代码

```
MOVE    r2,r1   /* copy r2 to r1 */
COM,>=  r3,r2   /* compare r3 to r2, nullify next if >= */
MOVE    r3,r1   /* copy r3 to r1 if not nullified */
```

还要注意的是，我们提供了两种方法来表示条件测试和分支：（1）先计算条件值，然后根据它或它的相反值进行分支，即

```
t3 ← t1 < t2
if t3 goto L1
```

或者（2）只用一条MIR指令计算条件并进行分支，即

```
if t1<t2 goto L1
```

前一种方法能很好地适应那种没有条件代码的体系结构，如SPARC、POWER或Intel 386体系结构系列。对于这种机器，比较动作之前有时总包含了一条减运算指令，因为t1<t2当且仅当0<t2-t1，而且为了能够及时判定条件代码以便知道分支是否会发生，将比较指令或减操作指令从条件分支中移出常常是所希望的。后一种方法很适合具有比较分支指令的体系结构，如PA-RISC或MIPS，因为MIR指令一般能转换成单条机器指令。

4.6.2 高级中间表示（HIR）

本节介绍对MIR的有关扩充，这些扩充使其成为高级中间表示HIR。

⊖ 操作码MOVE和COM都是伪操作码，而不是实际的PA-RISC指令。MOVE可作为ADD或OR来实现，COM作为COMCLR来实现。

bbs.theithome.com

超星公司制作请尊重作者版权

数组元素可用多个下标来引用，并且在HIR中其下标是显式表示的。数组按行为主的顺序存放，排在最后的下标变化最快。我们也包括了一种高级循环结构，即for循环和一个复合if结构。因此，*MIRInst*需要用*HIRInst*替换，且*AssignInst*，*TrapInst*和*Operand*的语法需要做如表4-4所示的改变。*IntExpr*是其值为整数的任意表达式。

for循环的语义类似于Fortran的do语句而不是C的for语句。具体而言，图4-7b的MIR代码给出了图4-7a中用HIR表示的for循环的含义，但有一个附带条件，即HIR的for循环的循环体不能改变控制变量v的值。注意，图4-7b的MIR代码根据opd2 > 0是否成立分别从两个循环体中（从L1开始的或从L2开始的）选择一个循环体。

76 ~ 78

表4-4 为使MIR扩展为HIR而对指令和操作数的XBNF描述所作的改变

HIRInst	⟶	*AssignInst* | *GotoInst* | *IfInst* | *CallInst* | *ReturnInst*
		| *ReceiveInst* | *SequenceInst* | *ForInst* | *IfInst*
		| *TrapInst* | *Label* : *HIRInst*
ForInst	⟶	for *VarName* ← *Operand* [by *Operand*] to *Operand* do
		*HIRInst** endfor
IfInst	⟶	if *RelExpr* then *HIRInst** [else *HIRInst**] endif
AssignInst	⟶	[*VarName* | *ArrayRef*] ← *Expression*
		| [*] *VarName* [. *EltName*] ← *Operand*
TrapInst	⟶	trap *Integer*
Operand	⟶	*VarName* | *ArrayRef* | *Const*
ArrayRef	⟶	*VarName* [{*Subscript* ⋈ ,}]
Subscript	⟶	*IntExpr*

```
for v ← opd1 by opd2 to opd3
   instructions
endfor
```

a)

```
      v ← opd1
      t2 ← opd2
      t3 ← opd3
      if t2 > 0 goto L2
L1:   if v < t3 goto L3
       instructions
      v ← v + t2
      goto L1
L2:   if v > t3 goto L3
       instructions
      v ← v + t2
      goto L2
L3:
```

b)

图4-7 a) for循环的HIR形式，b) 用MIR表示的该循环的语义

4.6.3 低级中间表示（LIR）

这一节我们描述使MIR变为低级中间代码LIR所需的改变。我们只通过替换MIR语法中的产生式（并根据需要作适当增加）和描述新增加的特征来对MIR语法和语义进行必要的改变。对XBNF描述的有关改变如表4-5所示。赋值和操作数需要修改，以便用寄存器和存储器地址替代变量名。函数调用需要修改，以便删除参数表，因为我们假定参数已传递在运行栈中或寄存器中（或者，如果多于预先规定的参数个数，则超过的参数传递在运行栈中）。

79

有5种类型的赋值指令，即完成下述5种类型操作的赋值指令：

1. 将表达式的值赋给一个寄存器；

bbs.theithome.com

超星公司制作请尊重作者版权

表4-5 为创建LIR而对MIR指令和表达式的XBNF描述的改变

LIRInst	⟶	*RegAsgnInst* | *CondAsgnInst* | *StoreInst* | *LoadInst*
		| *GotoInst* | *IfInst* | *CallInst* |*ReturnInst*
		| *SequenceInst* | *Label* : *LIRInst*
RegAsgnInst	⟶	*RegName* ← *Expression*
		| *RegName* (*Integer* , *Integer*) ← *Operand*
CondAsgnInst	⟶	*RegName* ←(*RegName*) *Operand*
StoreInst	⟶	*MemAddr* [(*Length*)] ← *Operand*
LoadInst	⟶	*RegName* ← *MemAddr* [(*Length*)]
GotoInst	⟶	goto {*Label* | *RegName* [{+ | −} *Integer*]}
CallInst	⟶	[*RegName* ←] call {*ProcName* | *RegName*} , *RegName*
Operand	⟶	*RegName* | *Integer*
MemAddr	⟶	[*RegName*] [(*Length*)]
		| [*RegName* + *RegName*] [(*Length*)]
		| [*RegName* [+ | −] *Integer*] [(*Length*)]
Length	⟶	*Integer*

2. 将操作数赋给寄存器的一个元素（在LIR中，元素用两个整数来表示，即元素的第一位的位置和它占据的位宽度，它们之间用逗号分开并包含在括号内）；

3. 根据一个寄存器中的值有条件地将一个操作数赋给另一个寄存器；

4. 将一个操作数存入存储器地址；

5. 从存储器地址取值到一个寄存器。

存储器地址用方括号（“[”和“]”）表示，它表示其值是一个寄存器的内容、两个寄存器的内容之和或一个寄存器的内容加上或减去一个整型常量，长度说明如果出现的话，是表示字节数的一个整数。

call指令有两个或两个以上的寄存器操作数。倒数第二个寄存器包含要转移的地址，最后一个给出用来存放返回地址的寄存器，返回地址是紧随这条call指令之后的地址。

80

对于寄存器名字，我们保留r0，r1，…，r31表示整型或通用寄存器，f0,f1,...,f31表示浮点寄存器，s0,s1,…表示符号寄存器。

4.7 用ICAN表示MIR、HIR和LIR

为了能够在ICAN程序中方便地操作MIR、HIR和LIR代码，我们下面讨论用ICAN表示它们的方法。它们的表示由ICAN类型MIRInst、HIRInst、LIRInst和其他的一些类型组成，这种表示可以看成是与前一节定义的中间表示的外部打印形式相对应的一种内部形式，即ICAN结构。

表4-6 作为ICAN枚举类型IROper的成员的MIR、HIR和LIR运算符名

中间代码运算符	ICAN标识符	中间代码运算符	ICAN标识符
+	add	−（二元运算符）	sub
*(二元运算符)	mul	/	div
mod	mod	min	min
max	max	=	eql
!=	neql	<	less
<=	lseq	>	grtr
>=	gteq	shl	shl
shr	shr	shra	shra

bbs.theithome.com

版权所有 翻印必究

（续）

中间代码运算符	ICAN标识符	中间代码运算符	ICAN标识符
and	and	or	or
xor	xor	*(一元运算符)	ind
.	elt	*.	indelt
-（一元运算符）	neg	!	not
addr	addr	(none)	val
(类型强制)	cast	(间接赋值)	lind
(间接元素赋值)	lindelt	(条件赋值)	lcond
(元素赋值)	lelt		

我们从表4-6开始。对于每种中间语言运算符（包括那些作用于赋值语句左端的运算符），表4-6用枚举类型IROper以及它的同义词Operator的元素给出了这些运算符的表示。枚举类型IROper和它的同义词Operator的定义如下：

```
IROper = Operator = enum {
        add,    sub,    mul,    div,    mod,    min,
        max,    eql,    neql,   less,   lseq,   grtr,
        gteq,   shl,    shr,    shra,   and,    or,
        xor,    ind,    elt,    indelt, neg,    not,
        addr,   val,    cast,   lind,   lcond,  lindelt,
        lelt}
```

下面我们定义ICAN类型Var、Const、Register、Symbol、Operand和LIROperand。

```
Var = CharString
Const = CharString
Register = CharString
Symbol = Var ∪ Const
Operand = Var ∪ Const ∪ TypeName
LIROperand = Register ∪ Const ∪ TypeName
```

实际上，每一种类型是它要声明的类型的一个子集，具体地说，

1. Var的成员是由带双引号的字母数字字符和下划线组成的序列，其中第一个字符必须是字母。临时变量以小写字母“t”打头后随一至多个十进制数字组成。以“s”、“r”或“f”打头，且后随一至多个十进制数字组成的符号表示寄存器，它们不是Var的成员。

2. Const的成员是整数或浮点数，Integer的成员是整数。 81

3. Register以“s”、“r”或“f”开始，其后跟随一至多个十进制数字。符号寄存器以“s”打头，整数寄存器以“r”打头，浮点寄存器以“f”打头。

中间代码其余部分的ICAN表示取决于具体的中间代码——MIR、HIR或LIR——所以，下面我们分三个小节来描述它们，这里首先定义Instruction是

```
Instruction = HIRInst ∪ MIRInst ∪ LIRInst
```

4.7.1 用ICAN表示MIR

我们用ICAN元组表示每一种MIR指令，如表4-7所示，其中隐式声明了类型MIRInst。枚举类型MIRKind、OpdKind和ExpKind，以及函数Exp_Kind()和Has_Left()的定义如图4-8所示。

bbs.theithome.com

版权所有 侵权必究

表4-7 表示为ICAN元组的MIR指令

Label:
⟨kind:label,lbl:*Label*⟩

receive *VarName*(*ParamType*)
⟨kind:receive,left:*VarName*,ptype:*ParamType*⟩

VarName ← *Operand1 Binop Operand2*
⟨kind:binasgn,left:*VarName*,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*⟩

VarName ← *Unop Operand*
⟨kind:unasgn,left:*VarName*,opr:*Unop*,opd:*Operand*⟩

VarName ← *Operand*
⟨kind:valasgn,left:*VarName*,opd:*Operand*⟩

VarName1 ←(*VarName2*) *Operand*
⟨kind:condasgn,left:*VarName1*,cond:*VarName2*,opd:*Operand*⟩

VarName ← (*TypeName*) *Operand*
⟨kind:castasgn,left:*VarName*,type:*TypeName*,opd:*Operand*⟩

**VarName* ← *Operand*
⟨kind:indasgn,left:*VarName*,opd:*Operand*⟩

VarName.*EltName* ← *Operand*
⟨kind:eltasgn,left:*VarName*,elt:*EltName*,opd:*Operand*⟩

**VarName*.*EltName* ← *Operand*
⟨kind:indeltasgn,left:*VarName*,elt:*EltName*,opd:*Operand*⟩

goto *Label*
⟨kind:goto,lbl:*Label*⟩

if *Operand1 Binop Operand2* goto *Label*
⟨kind:binif,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*,lbl:*Label*⟩

if *Unop Operand* goto *Label*
⟨kind:unif,opr:*Unop*,opd:*Operand*,lbl:*Label*⟩

if *Operand* goto *Label*
⟨kind:valif,opr:*Operand*,lbl:*Label*⟩

if *Operand1 Binop Operand2* trap *Integer*
⟨kind:bintrap,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*,trapno:*Integer*⟩

if *Unop Operand* trap *Integer*
⟨kind:untrap,opr:*Unop*,opd:*Operand*,trapno:*Integer*⟩

if *Operand* trap *Integer*
⟨kind:valtrap,opr:*Operand*,trapno:*Integer*⟩

call *ProcName*,(*Opd1*,*TN1*; ...;*Opdn*,*TNn*)
⟨kind:call,proc:*ProcName*,args:[⟨*Opd1*,*TN1*⟩,...,⟨*Opdn*,*TNn*⟩]⟩

VarName ←*ProcName*,(*Opd1*,*TN1*; ...;*Opdn*,*TNn*)
⟨kind:callasgn,left:*VarName*,proc:*ProcName*,
args:[⟨*Opd1*,*TN1*⟩,...,⟨*Opdn*,*TNn*⟩]⟩

return
⟨kind:return⟩

return *Operand*
⟨kind:retval,opd:*Operand*⟩

sequence
⟨kind:sequence⟩

VarName
⟨kind:var,val:*VarName*⟩

Const (includes *Integer*)
⟨kind:const,val:*Const*⟩

TNi
ptype:*TNi*

bbs.theithome.com

```
MIRKind = enum {
    label,    receive,  binasgn, unasgn,   valasgn,
    condasgn, castasgn, indasgn, eltasgn,  indeltasgn,
    goto,     binif,    unif,    valif,    bintrap,
    untrap,   valtrap,  call,    callasgn, return,
    retval,   sequence}

OpdKind = enum {var,const,type}
ExpKind = enum {binexp,unexp,noexp,listexp}
Exp_Kind: MIRKind --> ExpKind
Has_Left: MIRKind --> boolean

Exp_Kind := {
    <label,noexp>,       <receive,noexp>,  <binasgn,binexp>,
    <unasgn,unexp>,      <valasgn,unexp>,  <condasgn,unexp>,
    <castasgn,unexp>,    <indasgn,unexp>,  <eltasgn,unexp>,
    <indeltasgn,unexp>,  <goto,noexp>,     <binif,binexp>,
    <unif,unexp>,        <valif,unexp>,    <bintrap,binexp>,
    <untrap,unexp>,      <valtrap,unexp>,  <call,listexp>,
    <callasgn,listexp>,  <return,noexp>,   <retval,unexp>,
    <sequence,noexp>}

Has_Left := {
    <label,false>,       <receive,true>,   <binasgn,true>,
    <unasgn,true>,       <valasgn,true>,   <condasgn,true>,
    <castasgn,true>,     <indasgn,true>,   <eltasgn,true>,
    <indeltasgn,true>,   <goto,false>,     <binif,false>,
    <unif,false>,        <valif,false>,    <bintrap,false>,
    <untrap,false>,      <valtrap,false>,  <call,false>,
    <callasgn,true>,     <return,false>,   <retval,false>,
    <sequence,false>}
```

图4-8 用于确定MIR指令特征的类型和函数

Exp_Kind(*k*)指出类别为*k*的一条MIR指令是否含有一个二元表达式、一元表达式、表达式系列或没有包含表达式。Has_Left(*k*)在类别为*k*的MIR指令有一个left域时返回true；否则返回false。从现在起一直到需要用到第12章中介绍的过程的基本块结构之前，我们都用数组Inst[1..*n*]来表示中间代码的指令序列，其中*n*为某个整数，该数组的声明为

```
Inst: array [1..n] of Instruction
```

例如，MIR指令序列

```
L1: b ← a
    c ← b + 1
```

对应的元组组成的数组表示为

```
Inst[1] = <kind:label,lbl:"L1">
Inst[2] = <kind:valasgn,left:"b",opd:<kind:var,val:"a">>
Inst[3] = <kind:binasgn,left:"c",opr:add,
          opd1:<kind:var,val:"b">,opd2:<kind:const,val:1>>
```

图4-9给出了用ICAN元组表示的图4-6中名为insert_node的第二个程序单元体的MIR代码。

82 ~ 84

注意，出现在call或callasgn指令参数表中的*TNi*是类型名：偶对<*Opdi*，*TNi*>指出第*i*个参数的值和类型。

bbs.theithome.com

```
Inst[1]  = (kind:receive,left:"n",ptype:val)
Inst[2]  = (kind:receive,left:"l",ptype:val)
Inst[3]  = (kind:binasgn,left:"t1",opr:indelt,
            opd1:(kind:var,val:l),
            opd2:(kind:const,val:"value"))
Inst[4]  = (kind:binif,opr:lseq,opd1:(kind:var,val:"n"),
            opd2:(kind:var,val:"t1"),lbl:"L1")
Inst[5]  = (kind:binasgn,left:"t2",opr:indelt,
            opd1:(kind:var,val:l),
            opd2:(kind:const,val:"value"))
Inst[6]  = (kind:if,opr:neql,opd1:(kind:var,val:"t2"),
            opd2:(kind:const,val:nil),lbl:"L2")
Inst[7]  = (kind:call,proc:"make_node",
            args:[(kind:var,val:"t3"),ptype:type1),
           (kind:var,val:"n"),ptype:int)])
Inst[8]  = (kind:return)
Inst[9] = (kind:label,lbl:"L2")
Inst[10] = (kind:binasgn,left:"t4",opr:indelt,
            opd1:(kind:var,val:"1"),
            opd2:(kind:const,val:"next"))
Inst[11] = (kind:call,proc:"insert_node",
            args:[(kind:var,val:"n"),ptype:int),
             (kind:var,val:"t4"),ptype:type1)])
Inst[12] = (kind:return)
Inst[13] = (kind:label,lbl:"L1")
Inst[14] = (kind:return)
```

图4-9 图4-6中MIR程序单元insert_node的ICAN元组表示

4.7.2 用ICAN表示HIR

用ICAN表示HIR基本上同MIR一样处理，表4-8展示了HIR指令与ICAN元组之间的对应关系、并隐式声明了类型HIRInst。HIR不需额外的运算符，所以我们用IROper来表示它的操作数类型。

表4-8 不在MIR中的HIR指令的ICAN表示

for *VarName* ← *Operand1* by *Operand2* to *Operand3* do	(kind:for,left:*VarName*,opd1:*Operand1*,opd2:*Operand2*,opd3:*Operand3*)
endfor	(kind:endfor)
if *Operand1 Binop Operand2* then	(kind:strbinif,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*)
if *Unop Operand* then	(kind:strunif,opr:*Unop*,opd:*Operand*)
if *Operand* then	(kind:strvalif,opd:*Operand*)
else	(kind:else)
endif	(kind:endif)
VarName[*Expr1*,...,*Exprn*] ← *Operand1 Binop Operand2*	(kind:arybinasgn,left:*VarName*,subs:[*Expr1*,...,*Exprn*],opr:*Binop*, opd1:*Operand1*,opd2:*Operand2*)

bbs.theithome.com

超星公司制作 请尊重作者版权

（续）

VarName[*Expr1*,...,*Exprn*] ← *Unop Operand*
〈kind:aryunasgn,left:*VarName*,subs:[*Expr1*,...,*Exprn*],opr:*Unop*,
opd:*Operand*〉

VarName[*Expr1*,...,*Exprn*] ← *Operand*
〈kind:aryvalasgn,left:*VarName*,subs:[*Expr1*,...,*Exprn*],opd:*Operand*〉

VarName[*Expr1*,...,*Exprn*]
〈kind:aryref,var:*VarName*,subs:[*Expr1*,...,*Exprn*]〉

枚举类型HIRKind、HIROpdKind和HIRExpKind，以及函数HIR_Exp_Kind()和HIR_Has_Left()的定义如图4-10所示，它们与MIR稍有不同。 85

```
HIRKind = enum {
    label,     receive,   binasgn, unasgn,   valasgn,
    condasgn,  castasgn,  indasgn, eltasgn,  indeltasgn,
    goto,      trap,      call,    callasgn, return,
    retval,    sequence,  for,     endfor,   strbinif,
    strunif,   strvalif,  else,    endif,    arybinasgn,
    aryunasgn, aryvalasgn}

HIROpdKind = enum {var,const,type,aryref}
HIRExpKind = enum {terexp,binexp,unexp,noexp,listexp}
HIR_Exp_Kind: HIRKind ⟶ HIRExpKind
HIR_Has_Left: HIRKind ⟶ boolean

HIR_Exp_Kind := {
    〈label,noexp〉,          〈receive,noexp〉,
    〈binasgn,binexp〉,       〈unasgn,unexp〉,
    〈valasgn,unexp〉,        〈condasgn,unexp〉,
    〈castasgn,unexp〉,       〈indasgn,unexp〉,
    〈eltasgn,unexp〉,        〈indeltasgn,unexp〉,
    〈goto,noexp〉,           〈trap,noexp〉,
    〈call,listexp〉,         〈callasgn,listexp〉,
    〈return,noexp〉,         〈retval,unexp〉,
    〈sequence,noexp〉,       〈for,terexp〉,
    〈endfor,noexp〉,         〈strbinif,binexp〉,
    〈strunif,unexp〉,        〈strvalif,unexp〉,
    〈else,noexp〉,           〈endif,noexp〉,
    〈arybinasgn,binexp〉,    〈aryunasagn,unexp〉,
    〈aryvalasagn,unexp〉
}

HIR_Has_Left := {
    〈label,false〉,      〈receive,true〉,
    〈binasgn,true〉,     〈unasgn,true〉,
    〈valasgn,true〉,     〈condasgn,true〉,
    〈castasgn,true〉,    〈indasgn,true〉,
    〈eltasgn,true〉,     〈indeltasgn,true〉,
    〈goto,false〉,       〈trap,false〉,
    〈call,false〉,       〈callasgn,true〉,
    〈return,false〉,     〈retval,false〉,
    〈sequence,false〉,   〈for,true〉,
    〈endfor,false〉,     〈strbinif,false〉,
    〈strunif,false〉,    〈strvalif,false〉,
    〈else,false〉,       〈endif,false〉,
    〈arybinasgn,true〉,  〈aryunasagn,true〉,
    〈aryvalasagn,true〉
}
```

图4-10 确定HIR指令特征的ICAN类型和函数

bbs.theithome.com

版权所有 侵权必究

HIR_Exp_Kind (*k*)指出一个类别为*k*的HIR指令是否包含一个三元表达式[⊖]、二元表达式、一元表达式、表达式表或没有包含表达式。HIR_Has_Left(*k*)在类别为*k*的HIR指令有left域时返回true；否则返回false。

4.7.3 用ICAN表示LIR

表4-9给出了LIR指令和ICAN结构之间的对应关系，其中最后三项表示操作数。与用ICAN表示MIR和HIR代码一样，我们用类型IROper表示LIR运算符。

表4-9 LIR指令的ICAN表示

Label:
〈kind:label,lbl:*Label*〉

RegName ← *Operand1 Binop Operand2*
〈kind:regbin,left:*RegName*,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*〉

RegName ← *Unop Operand*
〈kind:regun,left:*RegName*,opr:*Unop*,opd:*Operand*〉

RegName ← *Operand*
〈kind:regval,left:*RegName*,opd:*Operand*〉

RegName1 ← (*RegName2*) *Operand*
〈kind:regcond,left:*RegName1*,cond:*RegName2*,opd:*Operand*〉

RegName(*Integer1*,*Integer2*) ← *Operand*
〈kind:regelt,left:*RegName*,fst:*Integer1*,blen:*Integer2*,opd:*Operand*〉

MemAddr ← *Operand*
〈kind:stormem,addr:tran(*MemAddr*),opd:*Operand*〉

RegName ←*MemAddr*
〈kind:loadmem,left:*RegName*,addr:tran(*MemAddr*)〉

goto *Label*
〈kind:goto,lbl:*Label*〉

goto *RegName* + *Integer*
〈kind:gotoaddr,reg:*RegName*,disp:*Integer*〉

if *Operand1 Binop Operand2* goto *Label*
〈kind:regbinif,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*,lbl:*Label*〉

if *Unop Operand* goto *Label*
〈kind:regunif,opr:*Unop*,opd:*Operand*,lbl:*Label*〉

if *Operand* goto *Label*
〈kind:regvalif,opr:*Operand*,lbl:*Label*〉

if *Operand1 Binop Operand2* trap *Integer*
〈kind:regbintrap,opr:*Binop*,opd1:*Operand1*,opd2:*Operand2*,
trapno:*Integer*〉

if *Unop Operand* trap *Integer*
〈kind:reguntrap,opr:*Unop*,opd:*Operand*,trapno:*Integer*〉

if *Operand* trap *Integer*
〈kind:regvaltrap,opr:*Operand*,trapno:*Integer*〉

call *ProcName*,*RegName*
kind:callreg,proc:*ProcName*,rreg:*RegName*〉

call *RegName1*,*RegName2*
〈kind:callreg2,creg:*RegName1*,rreg:*RegName2*〉

⊖ 只有for指令含有三元表达式。

bbs.theithome.com

越是公句制作请尊重作者版权

（续）

RegName1 ← call *ProcName*,*RegName2*
〈kind:callregasgn,left:*RegName1*,proc:*ProcName*,rreg:*RegName2*〉

RegName1 ← call *RegName2*,*RegName3*
〈kind:callreg3,left:*RegName1*,creg:*RegName2*,rreg:*RegName3*〉

return
〈kind:return〉

return *Operand*
〈kind:retval,opd:*Operand*〉

sequence
〈kind:sequence〉

RegName
〈kind:regno,val:*RegName*〉

Const (includes *Integer*)
〈kind:const,val:*Const*〉

TypeName
〈kind:type,val:*TypeName*〉

枚举类型LIRKind、LIROpdKind和LIRExpKind，以及函数LIR_Exp_Kind()和LIR_Has_Left()的声明如图4-11所示。LIR_Exp_Kind(*k*)指出类别为*k*的一条LIR指令是否包含二元表达式、一元表达式或没有包含表达式。如果类别为*k*的一条LIR指令有left域，LIR_Has_Left(*k*)返回true；否则返回false。 86

```
LIRKind = enum {
    label,      regbin,  regun,    regval,      regcond,
    regelt,     stormem, loadmem,  goto,        gotoaddr,
    regbinif,   regunif, regvalif, regbintrap,  reguntrap,
    regvaltrap, callreg, callreg2, callregasgn, callreg3,
    return,     retval,  sequence}

LIROpdKind = enum {regno,const,type}
LIRExpKind = enum {binexp,unexp,noexp}
LIR_Exp_Kind: LIRKind ⟶ LIRExpKind
LIR_Has_Left: LIRKind ⟶ boolean

LIR_Exp_Kind := {
    〈label,noexp〉,        〈regbin,binexp〉,
    〈regun,unexp〉,        〈regval,unexp〉,
    〈regcond,unexp〉,      〈regelt,unexp〉,
    〈stormem,unexp〉,      〈loadmem,noexp〉,
    〈goto,noexp〉,         〈gotoaddr,noexp〉,
    〈regbinif,binexp〉,    〈regunif,unexp〉,
    〈regvalif,unexp〉,     〈regbintrap,binexp〉,
    〈reguntrap,unexp〉,    〈regvaltrap,unexp〉,
    〈callreg,noexp〉,      〈callreg2,noexp〉,
    〈callregasgn,noexp〉, 〈callreg3,noexp〉,
    〈return,noexp〉,       〈retval,unexp〉,
    〈sequence,noexp〉}
 LIR_Has_Left := {
     〈label,false〉,       〈regbin,true〉,
     〈regun,true〉,        〈regval,true〉,
     〈regcond,false〉,     〈regelt,false〉,
```

图4-11 确定LIR指令特征的ICAN数据类型和函数

bbs.theithome.com

```
        ⟨stormem,false⟩,    ⟨loadmem,true⟩,
        ⟨goto,false⟩,       ⟨gotoaddr,false⟩,
        ⟨regbinif,false⟩,   ⟨regunif,false⟩,
        ⟨regvalif,false⟩,   ⟨regbintrap,false⟩,
        ⟨reguntrap,false⟩,  ⟨regvaltrap,false⟩,
        ⟨callreg,false⟩,    ⟨callreg2,false⟩,
        ⟨callregasgn,true⟩, ⟨callreg3,true⟩,
        ⟨return,false⟩,     ⟨retval,false⟩,
        ⟨sequence,false⟩}
```

图4-11 （续）

*RegName*操作数的值可以是整数寄存器（记为r*i*）、浮点寄存器（记为f*i*）或符号寄存器（记为s*i*）。由

```
RegType = enum {reg, freg, symreg}
Reg_Type: Register → RegType
```

声明的枚举类型和函数可用来区分这三种寄存器。存储器地址（在表4-9中表示为tran(*MemAddr*)）的表示如表4-10所示。

表4-10 存储器地址表示（表4-9中记为tran (MemAddr)）

[*RegName*] (*Length*) ⟨kind:addr1r,reg:*RegName*,len:*Length*⟩
[*RegName1*+*RegName2*] (*Length*) ⟨kind:addr2r,reg:*RegName1*,reg2:*RegName2*,len:*Length*⟩
[*RegName*+*Integer*] (*Length*) ⟨kind:addrrc,reg:*RegName*,disp:*Integer*,len:*Length*⟩

87 例如，图4-12所示的LIR代码，其对应的ICAN结构序列如图4-13所示。

```
L1:   r1 ← [r7+4]
      r2 ← [r7+r8]
      r3 ← r1 + r2
      r4 ← -r3
      if r3 > 0 goto L2
      r5 ←(r9) r1
      [r7-8](2) ← r5
L2:   return r4
```

图4-12 用ICAN元组表示的LIR代码例子

```
Inst[1]  = ⟨kind:label,lbl:"L1"⟩
Inst[2]  = ⟨kind:loadmem,left:"r1",
            addr:⟨kind:addrrc,reg:"r7",disp:4,len:4⟩⟩
Inst[3]  = ⟨kind:loadmem,left:"r2",
            addr:⟨kind:addr2r,reg:"r7",reg2:"r8",len:4⟩⟩
Inst[4]  = ⟨kind:regbin,left:"r3",opr:add,opd1:⟨kind:regno,
            val:"r1"⟩,opd2:⟨kind:regno,val:"r2"⟩⟩
Inst[5]  = ⟨kind:regun,left:"r4",opr:neg,
            opd:⟨kind:regno,val:"r3"⟩⟩
Inst[6]  = ⟨kind:regbinif,opr:grtr,
            opd1:⟨kind:regno,val:"r3"⟩,
            opd2:⟨kind:const,val:0⟩,lbl:"L2"⟩
Inst[7]  = ⟨kind:regcond,left:"r5",sel:"r9",
```

图4-13 与图4-12中LIR代码对应的ICAN元组序列

bbs.theithome.com

```
                opd:<kind:regno,val:"r1">>
    Inst[8]   = <kind:stormem,
                addr:<kind:addrrc,reg:"r7",disp:-8,len:2>,
                opd:<kind:regno,val:"r5">>
    Inst[9]   = <kind:label,lbl:"L2">
    Inst[10]  = <kind:retval,opd:<kind:regno,val:"r4">>
```

图4-13 （续）

4.8 管理中间代码的若干数据结构和例程的ICAN命名

在后边的叙述中，我们都把过程看成是由若干数据结构组成的，这些数据结构如下：

1. ProcName: Procedure，过程的名字。

2. nblocks: integer，组成该过程的基本块的个数。

3. ninsts: array[1..nblocks] of integer，一个数组，对于$i = 1, \cdots,$ nblocks, ninsts[i]是基本块i中的指令条数。

4. Block, LBlock: array[1..nblocks] of array[..] of Instruction，一个数组，其元素是构成基本块的MIR或HIR指令组成的数组（对于Block），或LIR指令组成的数组（对于LBlock）。即，Block[i][1 .. ninsts[i]]是基本块i中的指令组成的数组，LBlock也类似。

5. Succ, Pred: integer→set of integer，映射每一个基本块索引i分别到基本块i的前驱或后继基本块索引集合的函数。

在个别情况下，如稀有条件常数传播（12.6节）和基本块指令调度（17.1节），我们关注的是每一条单独的指令而不是基本块，因此会使用稍微不同的命名约定。

图4-14中定义的过程

```
insert_before(i,j,ninsts,Block,inst)
insert_after(i,j,ninsts,Block,inst)
append_block(i,ninsts,Block,inst)
```

插入指令*inst*于Block[i][j]之前或之后，或追加*inst*在Block[i]末尾，并相应地调整数据结构。注意，如果最后一条指令是goto或if，则请求插入一条指令于基本块的最后一条指令之后需要特殊处理——即，该指令要插入在控制转移指令之前。

88 ~ 92

```
procedure insert_before(i,j,ninsts,Block,inst)
   i, j: in integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of Instruction
   inst: in Instruction
begin
   || insert an instruction after position j in block i
   || and adjust data structures accordingly
   k: integer
   for k := j to ninsts[i] do
      Block[i][k+1] := Block[i][k]
   od
   ninsts[i] += 1
   Block[i][j] := inst
end   || insert_before
```

图4-14 插入一条指令至基本块中某给定位置之前或之后，或添加一条指令到基本块末尾的ICAN例程insert_before()、insert_after()和append_block()

bbs.theithome.com

```
procedure insert_after(i,j,ninsts,Block,inst)
   i, j: in integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of Instruction
   inst: in Instruction
begin
   || insert an instruction after position j in block i
   || and adjust data structures accordingly
   k: integer
   if j = ninsts[i] & Control_Transfer(Block[i][j]) then
      ninsts[i] := j += 1
      Block[i][j] := Block[i][j-1]
      Block[i][j-1] := inst
   else
      for k := j+1 to ninsts[i] do
         Block[i][k+1] := Block[i][k]
      od
      ninsts[i] += 1
      Block[i][j+1] := inst
   fi
end   || insert_after

procedure append_block(i,ninsts,Block,inst)
   i: in integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of Instruction
   inst: in Instruction
begin
   || add an instruction at the end of block i
   insert_after(i,ninsts[i],Block,inst)
end   || append_block
```

图4-14 （续）

图4-15中定义的例程

delete_inst (*i*, *j*, nblocks, ninsts, Block, Succ, Pred)

从基本块*i*中删除指令*j*，并调整用于表示程序的其他数据结构。

```
procedure delete_inst(i,j,nblocks,ninsts,Block,Succ,Pred)
   i, j: in integer
   nblocks: inout integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of Instruction
   Succ, Pred: inout integer → set of integer
begin
   || delete instruction j from block i and adjust data structures
   k: integer
   for k := j to ninsts[i]-1 do
      Block[i][k] := Block[i][k+1]
   od
   ninsts[i] -= 1
   if ninsts[i] = 0 then
      delete_block(i,nblocks,ninsts,Block,Succ,Pred)
   fi
end   || delete_inst
```

图4-15 从基本块中指定位置删除一条指令的ICAN例程delete_inst()

bbs.theithome.com

超星公司制作 请尊重作者版权

图4-16中定义的例程

insert_block(*i*, *j*, nblocks, ninsts, Succ, Pred)

通过在基本块*i*和*j*之间插入一个新的基本块而分裂边$i \rightarrow j$。

```
procedure insert_block(i,j,nblocks,ninsts,Succ,Pred)
   i, j: in integer
   nblocks: inout integer
   ninsts: inout array [··] of integer
   Succ, Pred: inout integer --> set of integer
begin
   || insert a new block between block i and block j
   nblocks += 1
   ninsts[nblocks] := 0
   Succ(i) := (Succ(i) ∪ {nblocks}) - {j}
   Succ(nblocks) := {j}
   Pred(j) := (Pred(j) ∪ {nblocks}) - {i}
   Pred(nblocks) := {i}
end   || insert_block
```

图4-16 通过插入一基本块于两个指定的基本块之间来分裂一条边的ICAN例程insert_block()

图4-17中定义的例程

delete_block(*i*, nblocks, ninsts, Block, Succ, Pred)

删除基本块*i*，并调整表示程序的数据结构。

```
procedure delete_block(i,nblocks,ninsts,Block,Succ,Pred)
   i: in integer
   nblocks: inout integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of Instruction
   Succ, Pred: inout integer --> set of integer
begin
   || delete block i and adjust data structures
   j, k: integer
   if i ∈ Succ(i) then
      Succ(i) -= {i}
      Pred(i) -= {i}
   fi
   for each j ∈ Pred(i) do
      Succ(j) := (Succ(j) - {i}) ∪ Succ(i)
   od
   for each j ∈ Succ(i) do
      Pred(j) := (Pred(j) - {i}) ∪ Pred(i)
   od
   nblocks -= 1
   for j := i to nblocks do
      Block[j] := Block[j+1]
      Succ(j) := Succ(j+1)
      Pred(j) := Pred(j+1)
   od
   for j := 1 to nblocks do
      for each k ∈ Succ(j) do
         if k > i then
            Succ(j) := (Succ(j) - {k}) ∪ {k-1}
```

图4-17 删除一个空基本块的ICAN例程delete_block()

bbs.theithome.com

```
            fi
         od
         for each k ∈ Pred(j) do
            if k > i then
               Pred(j) := (Pred(j) - {k}) ∪ {k-1}
            fi
         od
      od
   end   || delete_block
```

图4-17 （续）

93
~
95

4.9 其他中间语言形式

这一节我们介绍在中级中间代码组成的基本块中指令的另外几种可选表示（即：三元式、树、有向图或DAG，以及前缀波兰表示），讲述如何将它们转换为MIR，以及相对MIR的优缺点。与MIR中使用的形式相似，在编译前端的输出中，将基本块连接起来的控制结构最常用的是简单地使用goto、if和标号语句。这种控制结构将一直保存到控制流分析（参见第7章），以便提供过程内控制流特征的更多信息，如一组基本块是否形成了if-then-else结构、while循环或其他结构。

另外两种更进一步的重要中间代码形式是静态单赋值形式和程序依赖图，8.11节和9.5节将描述它们。

首先，请注意，我们用于MIR的形式以及与它有关的表示并不是传统的四元式形式，传统的形式是运算符在先，然后是三个操作数，通常结果操作数在前面，因此，

```
t1 ← x + 3
```

典型地书写为

```
+  t1, x, 3
```

我们选择使用运算符在中间的形式，因为它易于阅读。此外，这里所示的形式是为外部或打印表示而设计的，而前面讨论的对应的ICAN形式可看作是内部表示，即使它是为方便阅读而设计的——假如它是真正的内部形式，其表示会更为紧凑，且其中的符号很可能被指向符号表项的指针所替代。

还应注意到，本节提到的这些可选表示中没有一个是固定只能表示中级语言的——它们也具有等价的低级表示能力。

图4-18a给出了一段MIR代码，我们将用它作为与其他形式的代码进行比较的例子。

4.9.1 三元式

三元式与四元式相似，但三元式的结果没有被显式命名，而是使用隐含的名字。当其他三元式需要其结果作为操作数时，它们引用的是三元式的编号。此外，由于在三元式中没有表示结果的变量或存储位置，因此，需要保存结果时必须有显式的存数操作。例如，我们可以用“*a* sto *b*”表示存储*b*到位置*a*，“*a* * sto *b*”表示通过指针*a*的间接存数。在内部表示中，三元式的编号通常是指向对应三元式的指针，或者是相应三元式的索引号。这种做法对于三元式的插入和删除而言都比较复杂，除非控制转移的目标地址是表示过程基本块结构的结点，而
96 不是对具体三元式的引用。

在外部表示中，三元式编号通常置于括号之中，并列在每一行的开始。在三元式中用同样

bbs.theithome.com

超星公司制作请尊重作者版权

的形式来引用它们，这种形式易于使它们与整型常数区别开来。图4-18b给出了图4-18a中MIR代码对应的一种三元式转换。

```
L1:  i ← i + 1

     t1 ← i + 1
     t2 ← p + 4
     t3 ← *t2
     p ← t2
     t4 ← t1 < 10
     *r ← t3
     if t4 goto L1
```

a)

```
(1)  i + 1
(2)  i sto (1)
(3)  i + 1
(4)  p + 4
(5)  *(4)
(6)  p sto (4)
(7)  (3) < 10
(8)  r *sto (5)
(9)  if (7), (1)
```

b)

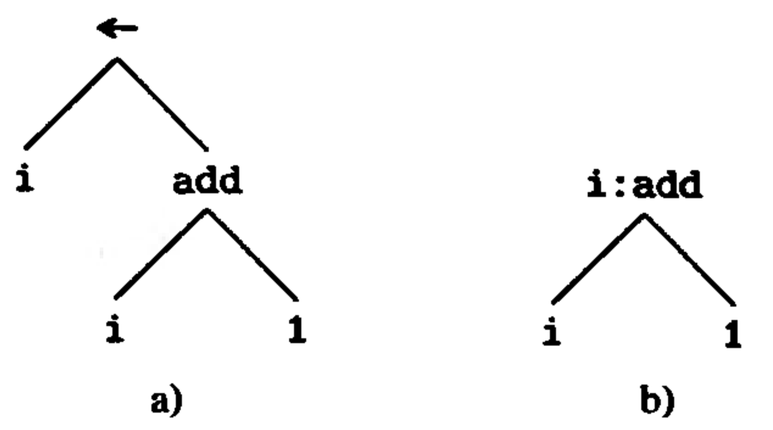
图4-18 a) 用于与其他中间形式进行比较的MIR代码，b) 它的三元式形式

将三元式转换为四元式，或将四元式转换回三元式的过程比较简单。从四元式转换到三元式需要用三元式编号来替换临时变量和标号，并引入具有显式存储操作的三元式。反过来，从三元式到四元式的转换则用临时变量和标号替换三元式编号，并且存储操作三元式将因被归并到计算其存储结果的四元式中而被消除。

使用三元式除了能够使代码生成之前创建基本块的DAG（参见4.9.3节）以及实现局部值编号（参见12.4.1节）的工作较为简单外，对优化没有特别的好处。三元式直接引用三元式作为它的操作数，因而简化了对一个结点的后裔的判别。

4.9.2 树

用树表示中间代码有两种可选的形式：一种是在树中使用一个显式赋值运算符，另一种是用一个结果变量（或多个变量）来标记一个表达式计算的根结点，分别如图4-19a和b所示。这种选择与使用三元式或四元式有点相似。我们选择使用第二种形式，因为它比前一种形式更接近于下一小节要讨论的DAG形式。我们用图4-6中给出的构成ICAN类型IROper的运算名来标记内部结点。



a) b)

图4-19 树的可选形式：a) 具有显式赋值运算符，b) 在计算树的根结点上标有结果变量

树在中间代码中几乎总是用来表示那些完成无控制流计算的代码部分，而控制流则通过将树相互连接到一起的形式来表示，图4-20说明了图4-18a的（无控制流）MIR代码对应的树形式。注意，这个转换本身实际上是没有用的——它为每个四元式产生一棵树，这并没有比四元式提供更多或更少的信息。

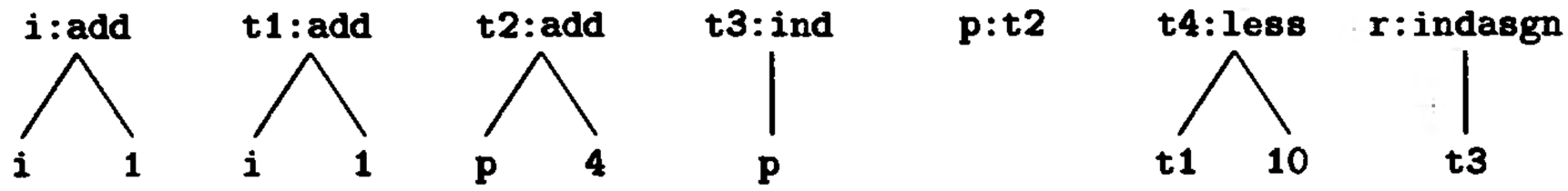


图4-20 图4-18a中无控制流MIR代码到简单树序列的转换

较好的一种转换可以判别出第二棵树计算出的t1只作为第六棵树的操作数使用，而且由于t1是临时变量，因此如果将第二棵树嫁接到第六棵树t1出现的地方，就没有必要存储t1。类似地，也可以将第三棵树与第五棵树合并。但是，第四棵树不能嫁接到第七棵上去，因为在它们之间改变了p的值。执行这种转换的结果产生如图4-21所示的一组树。

这种树表示形式比四元式有一些明显的优点：（1）它删除了两个临时变量（t1和t2）以及对它们的存储操作；（2）它为12.3.1节将讨论的代数化简提供了所希望的输入形式；（3）通过利用Sethi-Ullman数，可以由它生成适合许多体系结构的局部优化代码，Sethi-Ullman数规定了在生成指令时为使所用寄存器个数最少而应当遵循的顺序；（4）它提供了一种易于转换

bbs.theithome.com

成前缀波兰代码（参见4.9.4节）的形式，这种前缀波兰代码适宜作为语法制导的代码生成器的输入（见6.2节）。

将四元式转换为树依所做工作的不同而不同，就像图4-20和图4-21给出的树序列一样。转换到第一种形式是显然的，而第二种形式可以看成是对第一种形式的优化。我们惟一需要小心
98 的是，在嫁接一棵树到序列中的另一棵树时，必须保证在它原来的位置和它要嫁接到的位置之间，没有使用第一棵树结点标记的结果变量，并且没有重新计算其操作数。

注意，有两种不同的原因可能导致MIR指令序列对应的不是单一的一棵树——它们本来就不能连接成一棵树，或者由于计算一条指令多次而不仅是一次所致。作为后一种情况的例子，下列代码序列

```
a ← a + 1
b ← a + a
```

将产生如图4-22所示的树，这棵树对应于计算第一条指令两次，不过，我们可以从第二个“a: add”结点去掉其标记。

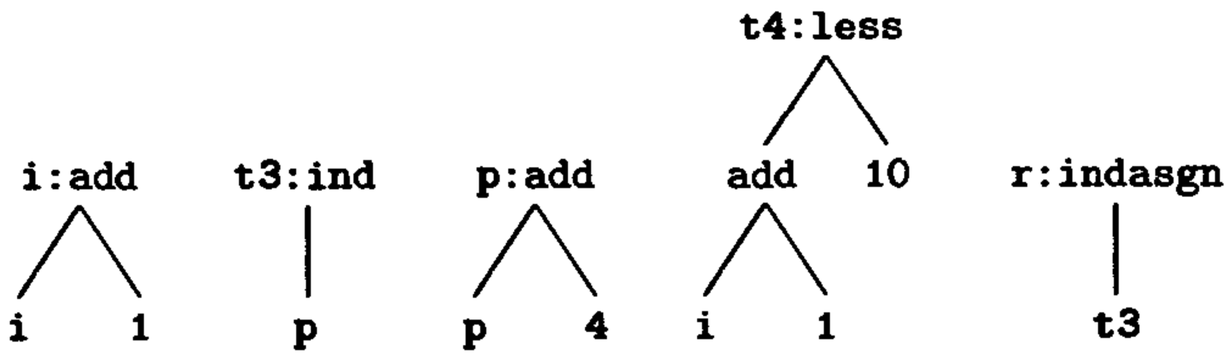


图4-21 图4-18a中无控制流MIR代码的最小树形式

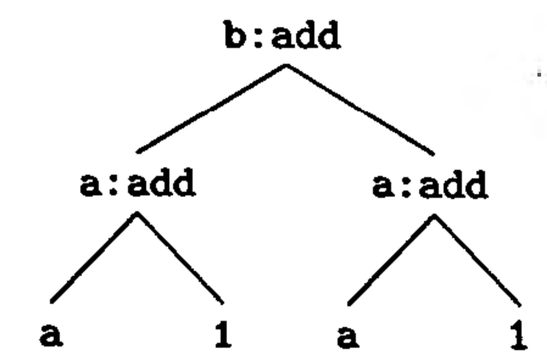


图4-22 尝试转换MIR指令a ← a + 1; b ← a + a为单棵树的结果

将树转换为四元式是简单的，我们可以按前序遍历也可以按后序遍历来处理。在第一种方法中，按树的出现顺序对每一棵树执行前序遍历。对于每一个至少有一个后裔，且此后裔也是内部结点（即非叶结点）的内部结点，我们创建一个新临时变量，并将这棵树从连接这两个内部结点的边的位置一分为二（分别称为“高部树”和“低部树”），用这个新临时变量标记低部树的根，并将这两棵树插入到序列中原来这棵树的位置（低部树在前）。高部树则用新临时变量替代原低部树的位置而得到修复。图4-23给出了一个转换的例子，这个例子中的树来自图4-21。当不再有
99 任何内部结点具有后裔时，每一棵树便对应于单条MIR指令，剩下的转换过程就简单了。

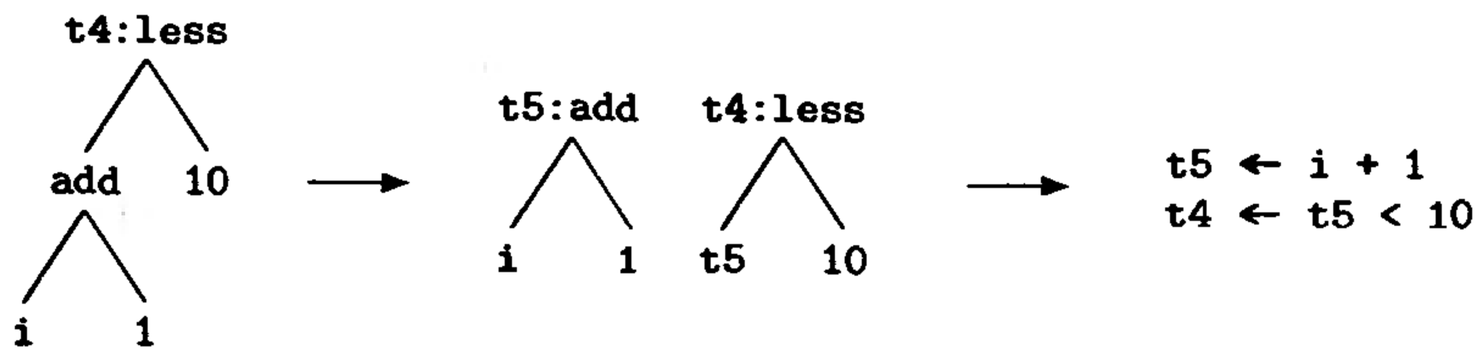


图4-23 从最小树形式到MIR代码的转换例子

在将树转换为MIR的第二种方法中，我们对给定的树执行后序遍历，为每一棵只含有一个运算符的子树生成一条MIR指令，并用这条MIR指令的左端替换该子树的根。

4.9.3 无环有向图（DAG）

基本块的DAG表示可以看作是对最小树序列的进一步压缩。DAG的叶子表示基本块中所使用的变量和常数在基本块入口时的值，DAG的所有其他结点都表示运算，并且也可以用变量名做标记，以指出在基本块内计算的值。我们用与前一节的树结点相同的画法表示DAG结

bbs.theithome.com

点。图4-24为一个基本块DAG的例子，它对应图4-18a的前7条指令。在这个DAG中，左下角的“add”结点表示MIR赋值“i←i+1”，而它上面的“add”结点表示计算“i+1”。这个计算结果进而与10相比较，由此计算出t4的值。注意，由于DAG重复使用值，因此，它比树表示和线性表示要更为紧凑。

为了转换MIR赋值指令序列到DAG，我们按顺序依次处理这些指令。对于每一条指令，检查它的每一个操作数，看是否已经有一个DAG结点表示它，如果还没有，则为它创建一个DAG叶子结点，然后检查这（两）个操作数的结点是否有表示当前运算的父结点，如果没有，则创建一个。然后，用结果变量名标记表示结果的这个结点，同时从DAG中以此名字作为标记的其他结点中删除该名字。

图4-25是MIR指令序列，它对应的DAG如图4-26所示。在这个DAG中，neg结点是一个没有带标记的运算符结点（它是为指令4而创建的，指令4原来对应的标记是d，但之后这个标记被指令7移给了mul结点），因此，不需要为neg结点生成代码。

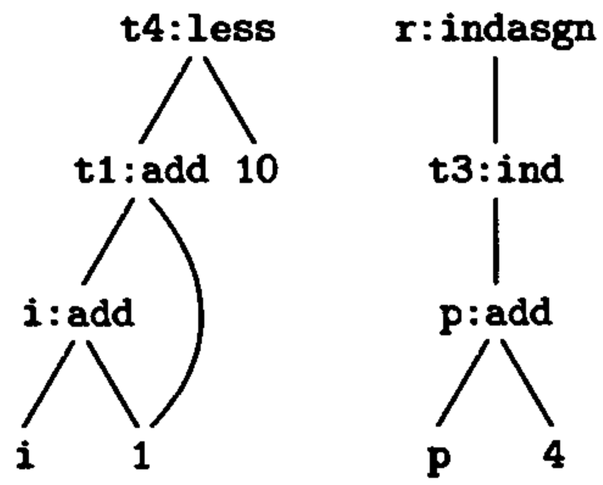


图4-24 图4-18a中无控制流MIR代码的DAG

```
1    c ← a
2    b ← a + 1
3    c ← 2 * a
4    d ← -c
5    c ← a + 1
6    c ← b + a
7    d ← 2 * a
8    b ← c
```

图4-25 要转换为DAG的MIR代码基本块例子

如前面提到的，DAG形式对于执行局部值编号有帮助，但对于进行其他大多数优化，它是一种相对困难的形式。但另一方面，存在若干结点列表算法，这些算法能够指导代码生成器由DAG生成相当高效的代码。 100

4.9.4 前缀波兰表示

前缀波兰表示实质上是对树形式作前序遍历的结果，它与树之间的相互转换是相当直接的递归过程。

对于图4-21所给的最小树序列，其前缀波兰形式是如图4-27所示的代码（注意，我们假定一元运算结点的后裔是其左边那个儿子）。例如，图中第二行表示的是一个一元赋值，该赋值的左端为t3，右端是对p间接操作的结果。

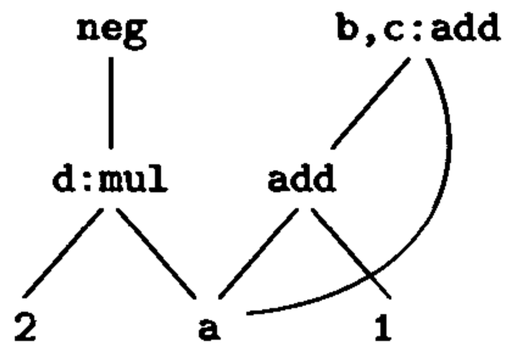


图4-26 图4-25中MIR代码相对应的DAG图

```
binasgn  i   add   i   1
unasgn   t3  ind   p
binasgn  p   add   p   4
binasgn  t4  less  add  i  1  10
indasgn  r   t3
```

图4-27 图4-21中的树分解为前缀波兰形式的指令

前缀波兰表示最有用的地方是作为语法制导代码生成器（参见6.2节）的输入。但它对优化不太有用，因为它的递归结构是隐式的。

bbs.theithome.com

4.10 小结

101 本节讨论了中间代码表示的设计，对照比较了包括抽象语法树、四元式、三元式、树、DAG和前缀波兰表示等在内的若干种表示，并选择了以后将用于本书例子的三种表示。我们选择的是HIR、MIR和LIR，并且给出了这三种表示的外部ASCII形式和内部ICAN结构形式。

选择中间代码形式时所关心的因素包括它的表现力（表达相关概念的能力），是否适合实现各种处理任务，紧凑性和访问速度（不至于浪费时间和空间），是否易于从源程序转换到中间代码并从中间代码转换到可重定位机器语言或其他低级形式，以及它的开发成本，是否已经实现了它或实现它的代价。

基本的MIR适合于大多数优化（LIR也这样），而较高级的HIR适合于依赖分析（第9章）以及基于依赖分析的一些转换，较低级的LIR用于需要明确指明寄存器和寻址的优化。

另外两种重要的中间代码形式，静态单赋值（SSA）形式和程序依赖图将分别在8.11节和9.5节讨论。我们在若干种优化中将使用静态单赋值，这些优化包括全局值编号（12.4.2节）和稀有条件常数传播（12.6节）。

4.11 进一步阅读

首次讨论Sethi-Ullman数的论文是[SetU70]，较新的论文是[AhoS86]。

关于HP的PA-RISC编译器的主要描述参见[CouH86]。

4.12 练习

4.1 构造如下C函数的抽象语法树表示。

```
double sumorprod(a,n,i)
    double a[100];
    int n;
    int i;
{   double acc;
    int j;
    if (i == 0)
    {   acc = 0.0;
        for (j = 0; j < 100; j++)
            acc += a[j];
    } else
    {   acc = 1.0;
        for (j = 99; j >= 0; j--)
            if (a[j] != 0.0)
                acc *= a[j];
        }
        return acc;
    }
```

102 4.2 构造练习4.1中C函数的HIR表示。

4.3 构造练习4.1中C函数的MIR表示。

4.4 构造练习4.1中C函数的LIR表示。

4.5 构造练习4.3中MIR代码的ICAN表示。

4.6 将练习4.3中C函数的MIR表示转换成（a）三元式，（b）树，（c）DAG和（d）前缀波兰表示。

bbs.theithome.com

超星公司制作 请尊重作者版权

4.7 写一个ICAN例程MIR_to_Triples(n,Inst,TInst)，它转换MIR指令组成的数组Inst[1],...,Inst[n]至三元式组成的数组TInst[1],...,TInst[m],并用m作为返回值。假设表示三元式的记录与表示MIR指令的记录类似，不同的是（1）另有两种类别，store和indstore，分别对应于4.9.1节讨论的运算符sto和*sto；（2）三元式的其他类别没有left域；（3）有另一种类型的操作数<kind:trpl,val:*num*>，它命名存储在TInst[*num*]中的那个三元式的结果。

4.8 写一个ICAN例程MIR_to_Trees(n,Inst,Root)，它将MIR指令组成的数组Inst[1],...,Inst[n]转换为一组树，这些树的树根存储在Root[1],...,Root[m]中，并将m作为返回值。树结点是下面定义的类型Node的元素。

```
Leaf = record {kind: enum {var,const},
               val: Var ∪ Const,
               names: set of Var}
Interior = record {kind: IROper,
                   lt, rt: Node,
                   names: set of Var}
Node = Leaf ∪ Interior
```

如果内部结点的kind是一个一元运算符，则它的操作数是它的lt域，并且它的rt域为nil。

4.9 写一个ICAN例程Prefix_to_MIR(PP,Inst)，它转换前缀波兰运算符和操作数序列PP到MIR指令数组Inst[1],...,Inst[n]，并将n作为其返回值。假设PP是sequence类型(MIRKind∪IROper∪Var∪Const)，并且前缀波兰代码如图4-27所示。

4.10 写一个ICAN例程DAG_to_MIR(R,Inst)，它将具有R个根的DAG转换为MIR指令数组Inst[1],...,Inst[n]，并把n作为返回值。假设DAG的结点是用练习4.8中定义的类型Node来表示的。

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

越是公司制作 请尊重作者版权

第5章　运行时支持

这一章，我们对运行时为支持高级语言共有一些概念所涉及的一些基本问题来个快速回顾。由于这些概念中的多数在其他一些介绍编译的书中已有详细介绍，因此，我们不再对它们进行更详细的探讨。我们的目的是让读者回顾这些问题，给出运行时处理这些问题的合适方法，并在需要时给出进一步阅读的参考文献。有些更高级的概念，如位置无关代码和堆存储管理，在本章的最后几节中讨论。

总的来说，本章主要关心的是为支持各种源语言所必需的软件约定，包括数据表示、变量的各种存储类的存储分配、可见性规则、过程调用、入口、出口和返回处理等。

许多体系结构的应用程序二进制接口（Application Binary Interface，ABI）标准都有助于确定[⊖]运行时多数数据结构的组织方法。这种标准规定了运行时环境各个方面的布局和特征，使得满足此标准的软件能够实现互操作，从而简化了软件编写者的工作。这种标准文档的一个例子是UNIX系统V ABI和它关于各种体系结构的特定处理器（如SPARC和Intel 386体系结构系列）的相关增补。

我们首先在5.1节讨论数据结构类型和运行时有效表示它们的方法，接下来在5.2节简要讨论寄存器的使用约定和基本的管理方法（全局优化寄存器使用方法在第16章讨论），然后，在5.3节讨论单一过程的栈帧结构。5.4节讨论关于运行栈的完整组织方法。在5.5节和5.6节，我们讨论支持参数传递和过程调用时涉及的问题。5.7节讨论利用动态链接和位置无关代码对代码共享的支持。最后，在5.8节讨论对动态和多态语言的支持。

5.1　数据表示和指令

为了实现较高级的语言，我们必须提供若干机制来表示它的数据类型和数据结构的概念，并支持其存储管理的概念。基本数据类型一般至少包含整数、字符和浮点数（其中每一种具有一或多种大小和格式），以及枚举值和布尔量。

我们希望将整数值映射到体系结构的一种基本整类型或几种类型。现实中的每一种目标体系结构至少通过存、取和计算操作直接支持一种大小和格式的整类型，一般是32位有符号整数的二进制补码。多数体系结构也支持32位无符号整数，以及施加于其上的所有或几乎所有的运算。字节和半字有符号和无符号整数则通过对长度为字节或半字的存取操作，或者通过一串操作来支持，这一串操作包括一字大小数据的存取、形成一个字所需要的抽取和插入运算、适当的符号扩展或零扩展，以及对应的基本整数运算。较长类型的整数运算通常需要多次存取（某些体系结构的双字整数运算除外），并且一般通过进位加和借位减运算来支持，由这两种运算可构造出完整的多精度算术和关系运算集合。

一直到最近，字符一般都是字节大小的量，尽管现在也常常支持半字的字符表示，如含有片假名和平假名音节书写系统以及中文与汉字字根系统的统一码（Unicode）。单个字符运算所需要的支持包括取、存和比较操作，这些操作是由对应的有符号或无符号一字和半字整数的存、取（或等价的操作）以及整数比较指令来支持的，偶尔也通过更复杂的指令来支持，如

⊖ 有人可能认为它们“有碍于在决策过程中发挥创造性”。

bbs.theithome.com

POWER系列的有符号字节变址取与比较指令。尽管许多CISC体系结构对一种字符表示或其他表示具有一些内建的支持（如DEC VAX系列对ASCII的支持，IBM System/370对EBCDIC的支持），更为现代的体系结构通常并不特别支持某种特定的字符集，而是将它留给程序员用软件来实现适当的操作。

浮点数一般有与ANSI/IEEE 754－1985标准相应的两种或三种格式——单精度、双精度和扩展精度（不常有），它们分别占一字、双字、80位或4字。在所有情况下，硬件都直接支持单精度浮点数的存取指令，多数情况下也直接支持双精度，但一般不支持扩展精度的存取指令。

106 当前多数体系结构（较明显的例外是POWER和Intel 386体系结构系列）提供单精度和双精度算术运算和比较运算的完整实现，但有些省去了平方根运算，而有些，如SPARC Version 8，还包含了四精度运算。尽管PowerPC直接支持单精度和双精度格式，但POWER只提供双精度运算，并分别在执行存和取的过程中将单精度数转换为双精度数或反之[⊖]。Intel 386体系结构系列的浮点寄存器支持80位格式，运算结果可以通过设置控制寄存器中的一个字段四舍五入为单精度或双精度。取和存操作指令可以将单精度转换为双精度或反之，也可以存取80位精度的值。对于大多数体系结构，实现ANSI/IEEE 754－1985标准规定要求的例外和例外值的复杂系统则还需要软件作一点支持，而有些情况下，如Alpha，则需要软件作较多的支持。

枚举值通常用连续的无符号整数来表示，它们需要的操作只有取、存和比较，但Pascal和Ada除外。这两种语言允许对枚举类型进行遍历。布尔量可以是一种枚举类型，例如在Pascal、Modula-2和Ada中；也可以是整数，如C中；还可以是独立的类型，如Fortran 77中。

数组通常可以有多维，并且其元素可以是基本类型，也几乎可以是任意类型，这取决于源语言。不论其元素是什么类型，数组都可以看作是n维立方体，其n维中的每一维对应一个下标位置。数组的表示在概念上常常可以看作是对此立方体按行为主（row-major）的顺序进行切片（如大多数语言）或按列为主（column-major）的顺序进行切片（如Fortran），然后根据元素在切片中的位置为每个元素指定一块存储。例如，声明为`vara: array[1..10, 0..5] of integer`的Pascal数组将占据$(10-1+1)\times(5-0+1)=60$字的存储空间，其中，`a[1, 0]`在第0字，`a[1, 1]`在第1字，`a[2, 0]`在第6字，而`a[10, 5]`在第59字。一般地，对于其声明为如下形式的Pascal数组exam

```
var exam: array[lo1..hi1, lo2..hi2, ..., lon..hin] of type
```

元素exam[sub_1, sub_2, ..., sub_n]的地址是：

$$base(\text{exam}) + size(type)\cdot\sum_{i=1}^{n}(sub_i - lo_i)\prod_{j=i+1}^{n}(hi_j - lo_j + 1)$$

其中$base$（exam）是数组第一个元素的地址，$size$（$type$）是每一个元素占据的字节数[⊜]。类似的公式可用于其他语言，但Fortran除外。对于Fortran，公式中的乘积是从$j=1$到$j=i-1$。

有些体系结构提供了能简化多维数组处理的指令。例如，Intel 386体系结构系列提供的取

107 和存指令（以及存储器－存储器的传送指令）使用了一个基址寄存器，一个缩放为1、2、4、8的索引寄存器和一个偏移量。有些体系结构，如POWER和PA-RISC，提供了具有基址寄存器更新的存取指令，并且在PA－RISC中同时还具有缩放功能，它们简化了对连续数组元素的访问和对元素大小各不相同的数组的并行访问。

⊖ POWER也有乘和加融合在一起的指令，这种指令使用全长度乘积作为加法的操作数。

⊜ 实际上，考虑到时间效率，编译器可以使得存储器中每一个元素的大小保持与最有效存储访问单位对齐。

bbs.theithome.com

bbs.theithome.com

来表示。例如，复数类型可用由两个浮点成员组成的记录来表示，其中一个成员为实部，另一个为虚部；有理数可用两个整数组成的记录来表示，一个是分子，另一个是分母，通常假设这两个整数的最大公因子为1。当然，具有丰富的类型构造符集合的语言能够支持众多种类的类型。

```
type color = set of (red, orange, yellow, green,
                     blue, indigo, violet);
var  primary: color;
. . .
primary := [red, yellow, blue]
```

图5-3 Pascal集合的例子

上面讨论的所有表示都假定类型是与变量相关的并且是编译时已知的。有些语言，如LISP和Smalltalk，类型与数据对象相关，而不是与变量相关，因此会需要运行时的类型判定。我们将这一问题留至5.8节再考虑。

5.2 寄存器用法

对于任何访问寄存器比访问多级存储系统的其他存储层次要快的机器而言，包括我们知道
109 的当前的所有机器，以及那些还没有制造出来的大多数机器，寄存器的用法是设计编译器时遇到的最重要的问题之一。假如可能，理想的情形是将所有对象都分配在寄存器中，从而完全避免访问存储器。尽管这个目标适合于几乎所有的CISC机器，但对于最近的CISC实现，如Intel的Pentium和其后的系列机，它尤为重要，因为最近的CISC实现偏向于采用RISC风格的快速指令集，而对于RISC体系结构，几乎所有的运算都要求其操作数在寄存器中，并且结果也放在寄存器中。不幸的是，寄存器的个数总是相对较少，因为它们是大多数实现中最昂贵的资源，这既源于它增加了芯片的面积和互连的复杂性，也源于寄存器的个数对指令的结构和指令中规定的操作码、偏移量、条件等的有效空间的影响。此外，由于数组需要索引，多数寄存器不支持这个能力，因此不可能使所有变量都存放在寄存器中。

当然，很少有能够将所有数据在所有时间内都保存在寄存器中的情况，因此精心地使用好寄存器，控制好对那些不在寄存器中的变量的访问很重要。特别地，有四个需要考虑的问题：

1. 尽可能地将程序执行中最频繁使用的变量分配到寄存器中；

2. 尽可能高效地访问那些当前不在寄存器中的变量；

3. 使“记账”使用的寄存器（例如，为管理变量对存储器的访问而保留的寄存器）个数尽可能的少，以便能用更多的寄存器来容纳变量的值。

4. 尽可能提高过程调用和相关操作的效率，如进入和退出作用域，以便使它们的开销减至最小。

当然，这些目标通常会相互冲突。特别地，高效访问不在寄存器中的变量和高效的过程调用需要的寄存器个数可能会多于其他情况下打算给它们的寄存器个数，因此，寄存器的使用是一个非常重要的、需要仔细设计的问题，也是需要体系结构给予某种支持的一个方面。第16章包含了若干为频繁使用的变量分配寄存器的非常有效的技术，因此这里不考虑寄存器分配问题。

参与寄存器竞争的有以下一些对象：

栈指针（stack pointer）

栈指针指向运行栈的当前栈顶，它通常是运行栈中下一个过程调用的局部存储空间（即其栈帧）的开始。

bbs.theithome.com

超星公司制作 请尊重作者版权

帧指针（frame pointer）

帧指针（可以不是必需的）指向运行栈当前过程的栈帧开始处。

动态链（dynamic link）

动态链指向运行栈中前一栈帧的开始（或者，如果没有使用栈帧指针，则指向前一栈帧的末尾），并用于在当前过程返回时重建调用者的栈帧。或者，它也可以是指令中的一个整数，表示当前帧指针和前一帧指针之间的距离；或者，如果没有使用帧指针，表示当前栈指针和前一栈指针之间的距离。 110

静态链（static link）

静态链指向词法上包含作用域的最近一次调用的栈帧，它用于访问非局部变量（有些语言，如C和Fortran，不需要静态链）。

全局偏移表指针（global offset table pointer）

全局偏移表指针指向用于多个进程之间代码共享的一个表（见5.7节），以建立和访问外部变量的私有（每个进程的）副本（如果不出现这种共享代码，则不需要）。

参数

参数由当前活跃过程传递给被调用过程。

返回值

被当前活跃过程调用的过程返回的结果。

频繁使用的变量

最频繁使用的局部（也可能是非局部的或全局的）变量。

临时变量

在表达式求值期间和其他较短活跃期内使用和计算出的临时值。

后面几节将逐一讨论这些对象。

根据指令集和寄存器的设计，有些运算可能需要两到四个寄存器。乘法和除法运算的结果常常使用两个整数寄存器，因为在乘法运算中，乘积的长度是两个操作数的长度之和；在除法运算中，这两个寄存器用于存放商和余数。有些体系结构（如PA-RISC）提供了双字长度的移位指令来替代CISC体系结构中通常具有的旋转移位操作。

5.3 局部栈帧

尽管我们希望将所有操作数都保存在寄存器中，但许多过程都需要一片存储区域用于以下若干目的:

1. 为那些要取其地址的变量（显式或隐式，例如传地址的参数），或需要通过变址逐一访问而不能在寄存器文件中存放的变量，以及不能一直保存在寄存器中的变量提供一个存放之处;

2. 当执行过程调用时，为那些要保存的出自寄存器的值提供一个标准的存放之处;

3. 为调试器追踪当前活跃过程链提供一种途径。 111

因为这些需要存放在存储器中的量多数只在进入一个过程时才存在，并且在过程返回后便不能再访问，因此一般将它们集中存放到一个称为帧（frame）的区域。帧安排在栈内，因而常常称之为栈帧（stack frame）。栈帧可以包含那些传递给当前过程但接收参数的寄存器容纳不下的参数、全部的或部分的局部变量、寄存器保护区、编译分配的临时变量、一个嵌套层次显示表（display，参见5.4节），等等。

为了能够在运行时访问当前栈帧的内容，我们按某种顺序（后面将描述）给这些存放在栈中的对象逐个地指定存储偏移，这些偏移均相对某个保存在寄存器内的指针。这个指针可能是帧指

bbs.theithome.com

越是公司制作请尊重作者版权

针fp（frame pointer），它指向当前栈帧的第一个单元；也可能是栈指针sp（stack pointer），它指向当前栈顶，即当前栈帧的最后一个单元之后的位置。大多数编译器在存储器中安排栈帧时，都使栈帧的开始地址高于结束地址，从而使得相对当前栈帧栈指针的偏移总是非负的，如图5-4所示。

有的编译器同时使用帧指针和栈指针，某些变量则相对它们各自来寻址（如图5-5所示）。究竟是选择使用单独一个栈指针，或单独一个帧指针，还是同时使用两者来访问当前栈帧的内容，既取决于硬件特征也取决于所支持的语言的特征。选择时的考虑是：（1）有一个独立的帧指针是否浪费寄存器还是没关系；（2）取和存指令中所提供的相对单个寄存器的短偏移量是否足以覆盖大多数帧的大小；（3）是否必须支持类似于C库函数alloca()这样的内存分配函数，这个函数动态分配空间扩充当前栈帧，并返回指向那片空间的指针。只用帧指针通常不是好的想法，因为为了能够从当前栈帧进一步调用其他过程，我们总是需要在某个地方保存栈指针或栈帧的大小。对于大多数体系结构而言，存和取指令中的偏移域足以覆盖多数栈帧，而且使用一个额外的寄存器作帧指针有代价，即由于需要将它保存到存储器并从存储器中恢复它，
112 会带来一定的开销，并且它所占用的寄存器不能用来存放变量的值。因此，如果栈指针能够覆盖足够的范围，并且我们不需要处理类似alloca()的函数，则合适的做法是只使用栈指针。

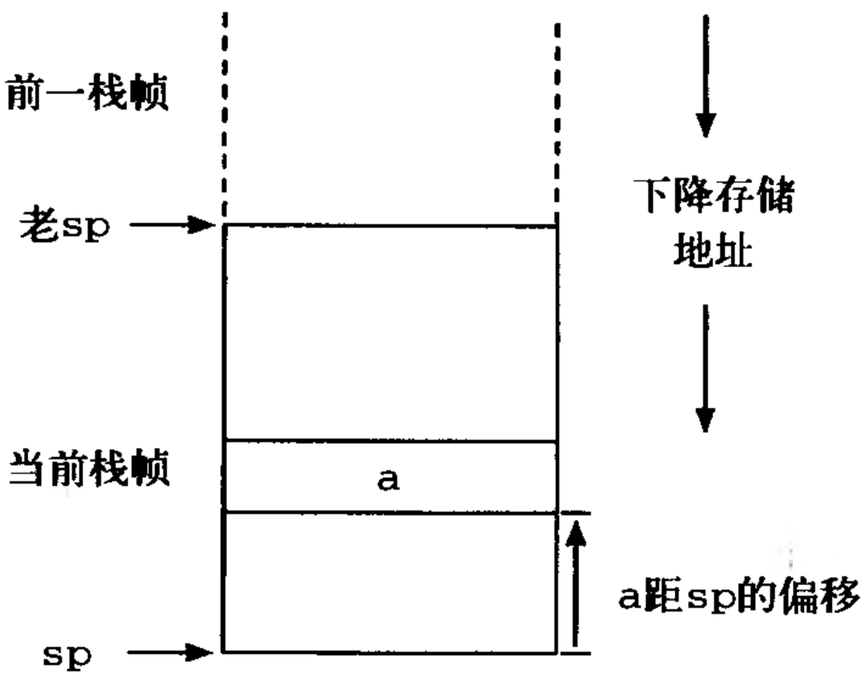


图5-4 具有当前栈指针和老栈指针的栈帧

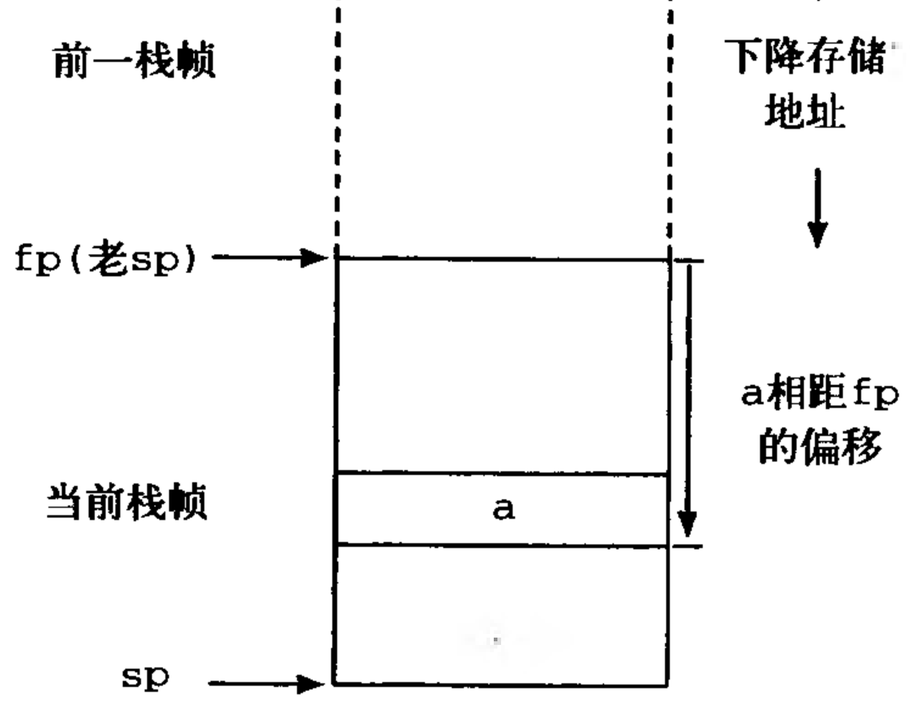


图5-5 具有栈指针和帧指针的栈帧

alloca()的作用是扩展当前栈帧，从而使得栈指针指向与原来不同的位置。当然，这也改变了通过原来的栈指针寻址的那些存储单元的偏移，因此必须复制这些存储单元的内容到与原来偏移相一致的地方。因为用户可能会计算一个C局部变量的地址并将它保存在某个变量中，这就意味着相对sp访问的量不能由用户来寻址，并且这些量必须是只在拥有该栈帧的过程被另一个过程调用悬挂期间所需要的量。于是，相对sp的寻址只可用于短期活跃的临时变量、转送给其他过程的参数、跨调用保护的寄存器以及返回值。因此，如果我们必须支持alloca()，就需要同时使用栈指针和帧指针。尽管这耗费了一个寄存器，但它的指令开销相对较低，因为我们只需在过程入口（1）保存老的帧指针在新栈帧内，（2）用老栈指针的值设置新帧指针，（3）增加当前栈帧的长度至栈指针。当从过程退出时，其处理则正好相反。在具有寄存器窗口的体系结构中（如SPARC），其动作更简单。如果选择栈指针是*out*寄存器之一，帧指针是相应的*in*寄存器，如SPARC UNIX系统V ABI规定的一样，则可用save和restore指令完成入口和出口操作，保存寄存器至存储器和从寄存器恢复它们的工作则留给寄存器窗口的溢出与填充自陷程序去处理。

增加sp可寻址范围的另一种方法是使sp指向栈顶之下（即当前栈帧之内）的某个固定距
113 离，这样，除了正偏移之外也能使用相对sp的部分负偏移。这增加了用一条存取指令所能访

bbs.theithome.com

越是公司制作请尊重作者版权

问的栈帧大小，但其代价是对于需要用到栈顶的调试器和其他工具，为找到实际的栈顶需要少量额外的计算。

5.4 运行时栈

在运行时我们不必使所有符号表结构（如果有的话）都存在，替代地，我们必须在编译时给变量分配能反映其作用域的地址，并在编译的代码中按这种方式来寻址。如5.3节所述，呈现在栈中的信息可有若干种，我们这里感兴趣的是对可见的非局部变量的寻址支持。如前面指出的一样，我们仍假定可见性受静态嵌套结构的控制。栈的结构为每一个活跃过程包含一个栈帧[⊖]，其中，如果一个调用已经进入了一个过程但还未从该过程退出，则该过程定义为是*活跃的*（active）。于是，对于一个给定的过程，如果它是递归的，则在同一时刻可以有该过程的多个栈帧，并且静态包含当前过程的那个过程的最近一次调用对应的栈帧可能位于栈内回退好几层栈帧的位置。每一个栈帧包含一个*动态链*（dynamic link），此链指向栈内前一栈帧的基地址，即前一栈帧的fp值[⊜]。

此外，如果源语言支持静态嵌套的作用域，则栈帧还包含一个*静态链*（static link），它指向静态包含过程的最近一次调用对应的栈帧，当前过程可在此栈帧中访问在那个过程中声明的变量的值。这个栈帧也依次包含一个静态链，它指向其静态包含过程的最近一次调用的栈帧，依此类推，一直链到全局作用域。为了在栈帧内设置静态链，我们需要一种机制来找到静态嵌套当前过程的那个过程的最近一次调用对应的栈帧。注意，调用一个不是静态嵌套在当前过程内的过程是非局部引用，新栈帧的静态链应指向包含这个非局部引用的作用域。于是，

1. 如果被调用的过程直接嵌套在调用它的过程内，它的静态链就指向调用过程的栈帧；

2. 如果这个过程与它的调用者处在相同的嵌套层次，则它的静态链从调用者的静态链复制；

3. 如果被调用的过程在静态嵌套结构中比调用过程高*n*层，则它的静态链可通过从调用者的静态链开始回溯*n*层而找到，然后，从找到的地方复制静态链。

```
procedure f( )
begin
   procedure g( )
   begin
      call h( )
   end
   procedure h( )
   begin
      call i( )
   end
   procedure i( )
   begin
      procedure j( )
      begin
         procedure k( )
         begin
            procedure l( )
            begin
               call g( )
            end
            call l( )
         end
         call k( )
      end
      call j( )
   end
   call g( )
end
```

图5-6 用于确定静态链的嵌套过程调用之例

图5-6和图5-7的例子说明了这种处理过程。对于第一个调用，即从f()调g()，g()的静态链设置成指向f()的栈帧。对于从g()调h()的调用，这两个例程以相同的层次嵌套在同一过程内，因此h()的静态链从g()中复制。最后，对于从l()调g()的

114

⊖ 一直到第15章我们都这样假设；第15章我们将优化掉某些栈帧。

⊜ 如果栈模型只使用栈指针来访问当前栈帧而没有帧指针，则动态链指向前一栈帧的末尾，也就是前一栈帧的sp值。

bbs.theithome.com

调用，g()在f()中的嵌套比l()的要高三层，因此我们沿l()的静态链回溯三层静态链，并从那里复制静态链。

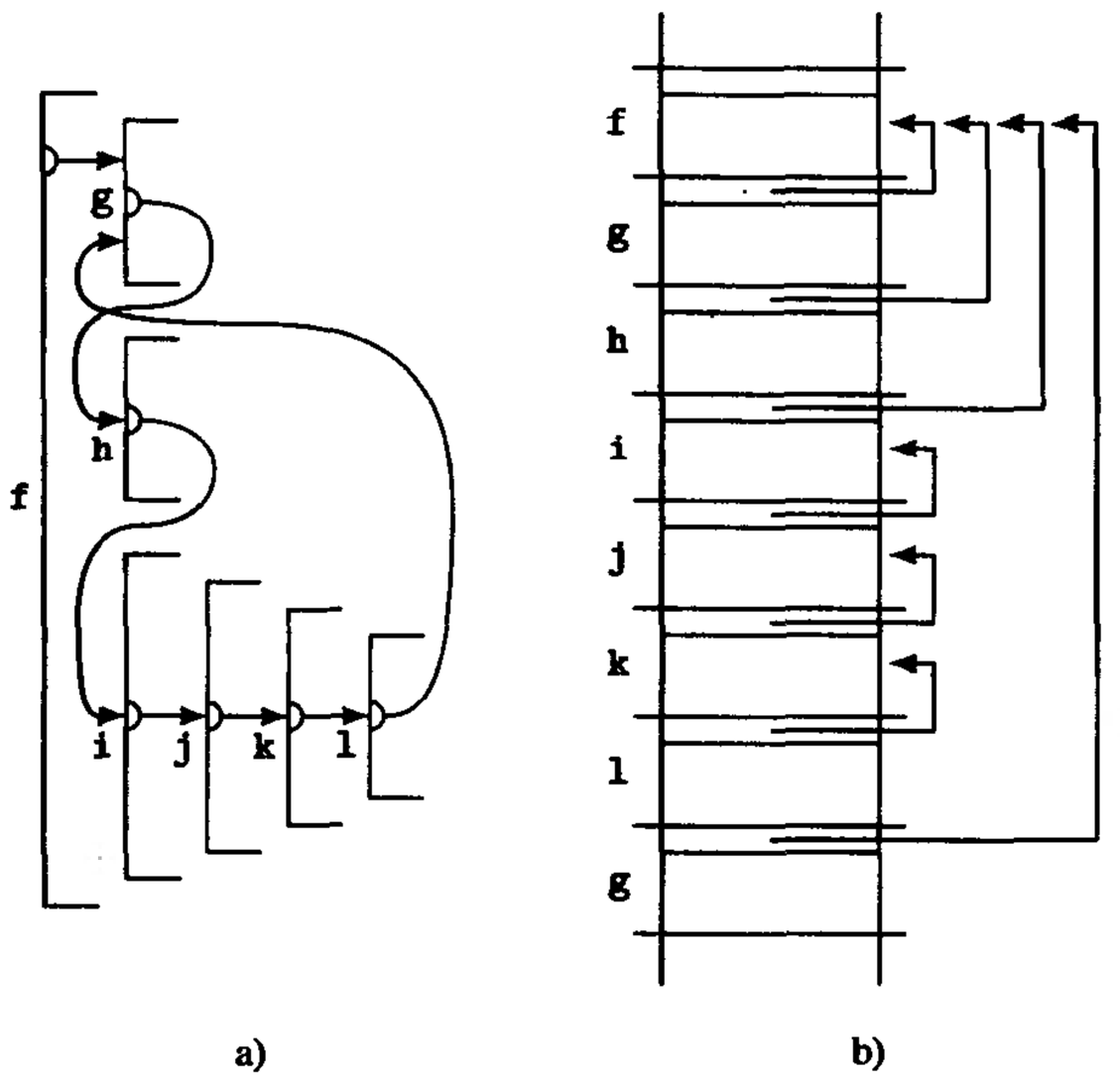


图5-7 a) 图5-6中七个过程的静态嵌套结构和它们之间的调用关系；
b) 它们在执行期间（在从l()进入g()之后）的静态链

如后面5.6.4节将要讨论的，当调用的是导入的例程或在独立包中的例程时，必须随其调用地址一起提供静态链。

设置了当前栈帧的静态链后，我们就能通过指向适当栈帧的静态链来进行非局部变量的上层寻址（up-level addressing）。从现在起，我们假设静态链存储在相对帧指针fp偏移为sl_off的地址（注意，sl_off是存储在3.6节所用变量StaticLinkOffset中的值）。假设我们有一个过程h()嵌套在过程g()中，而g()又嵌套在f()中。为了在执行h()期间得到f()中变量i的值，该值在f()的栈帧内偏移为i_off的位置，我们可以执行如下的LIR指令序列：

```
r1 ← [fp+sl_off]    || get frame pointer of g( )
r2 ← [r1+sl_off]    || get frame pointer of f( )
r3 ← [r2+i_off]     || load value of i
```

115

尽管这样做似乎相当昂贵，但并不一定是这样。首先，访问非局部变量一般不太频繁。其次，如果非局部访问是普遍的，则一种称为*嵌套层次显示表*（display）的机制能够削减多次引用所产生的代价。嵌套层次显示表保存当前静态链序列的全部或部分于一系列寄存器中或存储器的一个数组中。如果将嵌套层次显示表保存在寄存器中，一旦嵌套层次显示表设置好后，非局部引用的代价就不比局部引用高。当然，用寄存器来存放嵌套层次显示表可能是不利的，因为它减少了用于其他目的的有效寄存器的个数。如果将嵌套层次显示表保存在存储器中，则每一个引用最多多花一条取指令来将所期望栈帧的帧指针读入寄存器。将嵌套层次显示表保存在存储器中还是寄存器中，还是同时使用二者，最好留待如第16章讨论的全局寄存器分配去选择。

5.5 参数传递规则

现有高级语言中的过程参数传递和结果返回机制有若干种，它们包括（1）传值，（2）传结果，（3）传值得结果，（4）传地址，（5）传名字。这一节我们逐个介绍它们，讲述如何实现

bbs.theithome.com

越是公司制作请尊重作者版权

它们，附带也提及使用这些机制的一些语言。此外，我们还讨论标号参数的处理，因为有些语言允许给过程传递标号。我们使用术语*参数*（argument）或*实参*（actual argument）表示被传递给一个过程的值或变量，术语*哑参*（parameter）或*形参*（formal parameter）表示在被调用过程中与实参结合的变量。 116

概念上，*传值*（call by value）简单地将实参的值传递给被调用过程，使被调用过程可以将实参的值作为对应形参的值。当被调用过程执行时，它不会干扰调用者的变量，除非实参是一个指针。在实参是指针的情况下，被调用过程可用它来改变它所指向的任何对象的值。传值一般通过在被调用过程的入口复制实参的值至对应的形参来实现。这种方法对于那些可用寄存器容纳的参数而言是简单而高效的，但对于体积较大的数组却非常昂贵，因为它需要大量的存储交换。如果编译时能够同时对调用过程和被调用过程进行分析，我们就可能确定被调用过程没有改写传值数组形参的值，没有将此数组形参传递给它所调用的任何子程序，或它所调用的子程序没有改写其数组形参（参见19.2.1节）。在这种情况下，我们就可以通过传递实参数组的地址来实现传值调用，并允许被调用者直接访问实参，而不需要复制它。

C、C++、ALGOL 60和ALGOL 68都有传值的形式。在C、C++ 和ALGOL 68 中，它也是惟一的参数传递机制。但在这三种语言中，被传递的参数可以是（对于C和C++的某些类型而言，总是）一个对象的地址，因此，这也具有传地址的效果。Ada的`in`形参是传值的一种修改形式——传递给它们的是值，并且在被调用过程中是只读的。

传结果（call by result）与传值类似，但它是把一个值从被调用过程返回给调用过程而不是从调用过程传递给被调用过程。它在被调用过程的入口没有动作，当从被调用过程返回时，一般通过将传结果形参的值复制到与之结合的实参而使得形参的值对调用者有效。传结果也有和传值一样的效率考虑。在Ada中，传结果是用`out`形参来实现的。

传值得结果（call by value-result）实际上是传值和传结果的组合。在被调用过程入口，实参的值被复制到形参中；返回时，形参的值被复制回实参。传值得结果在Ada中用`inout`形参来实现，而在Fortran中则是正常的参数传递机制。

传地址（call by reference）在被调用过程的入口处建立实参与对应形参之间的结合，此时，实参的地址被确定，并且提供给被调用者来访问实参。被调用者在调用期间访问的完全是实参；它常常可以随意地改变实参，并且可以将实参传递给其他子程序。传地址通常通过传递实参的地址给被调用过程来实现，被调用过程则通过该地址来访问实参。对于数组形参，它是非常有效的，因为不需要复制，但对于有些参数（例如那些适合放在寄存器中的参数）却是低效的，因为它有碍于将这些参数放在寄存器中传递。考虑一个同时可以作为全局变量访问的传地址实参，我们对传地址会有所理解。如果传递给被调用者的是该实参的地址，那么，作为参数访问它和作为全局变量访问它所使用的都是同一个地址单元的值。但是，如果是通过寄存器传递该参数的值，那么作为全局变量访问的一般是它在存储单元中的值，而不是寄存器中的值。 117

当一个常数以传地址方式被传递时，如果编译器的实现是将此常数放在一个由该过程内所有使用该常数的计算都共享的地址内，则会出现问题，因为如果这个常数以传地址方式传递给其他子程序，而那个子程序有可能会改变这个地址中的值时，则可能影响到该调用之后其余部分程序的执行对该常数的使用。常用的纠正方法是将作为实参的常数复制到一个新的匿名位置，然后传递该位置的地址。

传地址是Fortran的合法参数传递机制。C、C++和ALGOL 68 允许对象的地址作为值参数被传递，因此，实际上它们也提供传地址。

Fortran中参数传递的语义允许每一个实参既可以是传值得结果的，也可以是传地址的。因

bbs.theithome.com

版权所有 侵权必究

此，传值得结果可用于那些适宜放在寄存器的值，传地址则用于数组，这为不同传递类型的实参提供了最适合于它们执行的两种高效机制。

传名字（call by name）是在概念和实现上都最复杂的一种参数传递机制。它实际上只有历史意义，因为ALGOL 60 是惟一提供它的著名语言。它与传地址相似的是允许被调用者访问调用者的实参，但不同点是在每次访问实参时才计算它的地址，而不是在被调用者的入口处。例如，如果实参是表达式a[i]，而i的值在此实参的两次使用之间有改变，则这两次使用访问的是数组a的不同元素。图5-8举例说明了这种情形，其中i和a[i]由主程序传递给过程f()。形参x的第一次引用取的是a[1]的值，而第二次引用设置的是a[2]。对outinteger()的调用将打印出5 5 2。假如使用的是传地址，那么，x的两次引用都将访问a[1]，并且该程序将打印出5 5 8。实现传名字需要一种在每次访问形参时计算形参地址的机制，这一般通过提供一个称为“形实转换程序”（thunk）的无参数过程来实现。每一次调用一个形参的形实转换程序将返回它的当前地址。当然，这是一种非常昂贵的机制，但是编译器可以识别出许多与传地址相同的简单情形，例如，传递一个简单变量、整个数组或一个数组的固定元素，所产生的地址不会变化，因此，对于这些情形就不需要调用形实转换程序。

```
begin
   integer array a[1:2]; integer i;
   procedure f(x,j);
      integer x, j;
      begin integer k;
         k := x;
         j := j + 1;
         x = j;
         f := k;
      end;
   i := 1;
   a[1] := 5;
   a[2] := 8;
   outinteger(a[1],f(a[i],i),a[2]);
end
```

图5-8 ALGOL 60的传名字参数传递机制

有些语言（例如ALGOL 60和Fortran）可以将标号作为参数传递给过程，使得标号可以作为被调用过程内的goto目标。实现这种功能需要传递标号对应代码点的地址和对应栈帧的动
118 态链。目标是标号形参的goto将执行一个或多个返回操作，直到到达由该动态链指出的适当的栈帧，然后转移到此标号指出的那条指令。

5.6 过程的入口处理、出口处理、调用和返回

一个过程被另一个过程调用涉及到这两个过程之间的“握手”动作，即，从调用过程传递参数和移交控制给被调用过程，以及从被调用过程返回结果和控制返给调用过程。在最简单的运行模式中，一个过程的执行包含如下5个主要阶段（每一个阶段又由一系列步骤组成）：

1. 过程调用组装要传递给该过程的实参，然后将控制移交给这个过程。
   (a) 每一个参数被求值并存放在适当的寄存器或栈单元中，“求值”意味着计算其值（对传值参数）、其地址（对传地址参数）等等；
   (b) 确定这个过程的代码的地址（对于大多数语言，其地址是在编译时或链接时确定的）；
   (c) 将调用过程使用的或保护的寄存器保存到存储器中；
   (d) 如果需要，计算这个被调用过程的静态链；
   (e) 将返回地址保存在一个寄存器内，然后执行转移到被调用过程的分支代码。

2. 过程的入口处理（prologue），它们在过程的入口处执行，为此过程建立适当的寻址环境，并完成其他一些功能，如保护该过程用于自己的目的的那些寄存器。
119 (a) 保存老帧指针，老栈指针变为新的帧指针，并计算新栈指针；
   (b) 将此过程要使用并且需要保护的那些寄存器保存到存储器中；

bbs.theithome.com

(c) 如果运行时模型使用嵌套层次显示表，则构造它。

3. 过程本身的工作，它可能含有对其他过程的调用；

4. 过程的出口处理（epilogue）恢复调用过程的寄存器和寻址环境，组装返回值，并返回控制给调用过程。

(a) 从存储器恢复所有由被调用者保护的寄存器；

(b) 将要返回的值（如果有的话）存放在适当的地方；

(c) 恢复老的栈指针和帧指针；

(d) 执行转移到返回地址的分支指令。

5. 最后，由调用过程中紧接在此调用之后的代码恢复调用过程的执行环境，并接收返回值。

(a) 从存储器中恢复调用者保护的寄存器；

(b) 使用返回值。

有几个问题会使这个模式变得复杂化，这些问题包括：参数传递机制怎样随参数的类型和个数而变化？哪些寄存器应由调用者保护，哪些应由被调用者保护，哪些是两者都不保护的（即所谓的草稿寄存器）？是否有可能调用一个由变量给出其地址的过程？以及一个过程是私有于包含它的过程的，还是共享的（第5.7节讨论）？

在过程调用时有效地管理寄存器是实现高性能的基本要求。如果调用过程假定被调用过程可以随意地使用任何寄存器（不包括那些特殊寄存器，如栈指针），它就必须保护和恢复所有含有有用值的寄存器——这可能意味着几乎所有的寄存器。同样，如果被调用过程假定所有非特殊的寄存器都是调用过程正在使用的寄存器，那它就必须保护和恢复所有可能含有调用过程需要的值的寄存器——同样，这也可能意味着几乎是所有的寄存器。因此，按一种最优的方式将寄存器集合区分为四类就很重要。这四类寄存器是（1）特殊的（只由调用约定管理的寄存器），（2）调用过程保护的，（3）被调用过程保护的，（4）草稿用的（在过程调用时完全不需要保存的）。当然，最优划分与体系结构特征有关，如SPARC有寄存器窗口，Motorola 88000中整型数据和浮点数据可共享一个寄存器集合；也与体系结构的限制有关，如Intel 386体系结构中寄存器个数较少且种类不同。最优划分也随程序不同而变化。过程间寄存器分配（如19.6节所述）可以减缓过程调用带来的影响。在缺乏过程间寄存器分配的情况下，实验和经验是确定满意划分的最好指导。在UNIX ABI与处理器相关的增补文档中提供了划分寄存器集合方法的例子。 120

注意，用寄存器传递参数的两种方法都还需要基于栈的处理来配合——如果参数太多以致没有可用的寄存器时，它们将被放在栈中传递。

5.6.1 用寄存器传递参数：平面寄存器文件

在具有大量通用寄存器的体系结构中，参数一般用寄存器来传递。一组整型寄存器和一组浮点寄存器被指定用来容纳前*ia*个整型参数和前*fa*个浮点参数，其中*ia*和*fa*是某个较小的整数 ⊖。参数依据其类型顺序地存放到这两组寄存器之中，如果还有余留的参数，则用栈中约定的存储单元来传递。假设我们有一个对f(i，x，j)的调用，其中，参数采用传值方式，第一个和第三个参数是整型值，第二个参数是单精度浮点值。于是，对这个例子，参数i和j将在前两个整型参数寄存器中传递，而x将在第一个浮点参数寄存器中传递。该过程调用的握手动作包括

⊖ Weicker发现，传递给过程的平均参数个数大约为2，并且后来的研究结果也与这个结论一致。因此，*n*的值通常在5~8范围之间。但是，有些系统规范允许更大的值，特别是，UNIX系统V用于i860的ABI允许12个整数寄存器和8个浮点寄存器用于参数传递。

bbs.theithome.com

使得为f()生成的代码也按这种方式接收它的参数（这一例子用在5.11节的练习5.4~5.6中）。

这种机制适合于那种其值可存放在单个寄存器或一对寄存器中的参数，以及所有传地址参数。对于其体积超过了一对寄存器大小的传值参数，通常使用另一种约定，即将参数的地址传递给被调用过程，而让被调用过程复制该参数的值到自己的栈中或其他存储单元。如果需要的话，该参数的大小也可传递过去。

如果多于*ia*个整型参数或*fa*个浮点参数要传递，多出的参数通常存放在栈中紧接当前栈指针之后的位置，因此，被调用例程可以用相对新帧指针的非负偏移来访问它们。

被调用过程传递返回值通常用与调用过程传递参数相同的方法来实现，除此之外，在多数语言中，过程不会有一个以上的返回值，并且为了使过程是可再入的（reentrant），即在同一时刻可有多个控制线程执行它，还需要一些特别的考虑。实现可再入的办法是，将那些由于体积太大而不能保存在寄存器中的值存放在调用过程提供的存储单元中返回。为此，调用过程必须提供指向接收这个返回值的存储单元的指针[⊖]（通常作为额外的一个隐含参数），这样被调用过程能将返回值存放在其中，而不是由它自己为这个值提供存储单元。

121 一种典型的寄存器用法如下所示：

寄存器	用　法
r0	0
r1~r5	参数传递
r6	帧指针
r7	栈指针
r8~r19	调用过程保护的
r20~r30	被调用过程保护的
r31	返回地址
f0~f4	参数传递
f5~f18	调用过程保护的
f19~f31	被调用过程保护的

并且返回值将依据其类型放在r1或f0中。我们选择图5-9所示的栈结构，其中假定局部变量占

122 4个字，gr和fr分别是通用寄存器（general register）和浮点寄存器（float-point register）的缩写。当参数太多以至于不能将它们全部用寄存器传递时，有些参数就可能需要通过栈来传递。它们的栈空间将分配在fp-128和sp+104之间。练习5.4要求你写出这一模型的过程调用、入口处理、参数的使用、出口处理及返回的代码。

5.6.2 用运行时栈传递参数

在基于栈的模型中，参数被压至运行时栈中，并从栈中访问它们。在寄存器数量较少且有栈操作指令的机器中，如VAX和Intel 386体系结构，我们使用那种将参数存储到栈中的指令。例如，对于Intel 386体系结构，如下压栈指令

```
r1 ← 5          || put third argument on stack
sp ← sp - 4
```

将被指令

```
pushl    5      ; push third argument onto stack
```

所替代。此外，在将参数压入栈后也不需要调整栈指针——pushl指令将自动完成。返回值可

⊖ 如果调用者也提供该区域的大小作为隐含的参数，被调用者就能检查它所返回的值是否能被该区域容纳。

bbs.theithome.com

越是公句制作请尊重作者版权

以放在寄存器中，也可以放在栈中。在我们的例子中使用浮点寄存器栈顶。

Intel 386和它以后的微处理器体系结构提供了8个32位的整数寄存器，其中6个是eax、ebx、ecx、edx、esi和edi，它们对于大多数指令而言是通用寄存器。另外两个ebp和esp分别是基指针（即帧指针）和栈指针。该体系结构还提供了8个80位的浮点寄存器，即st(0)到st(7)，它们具有栈的功能，其中st(0)作为栈顶。特别地，浮点返回值放置在st(0)中。这种运行时栈的布局如图5-10所示。

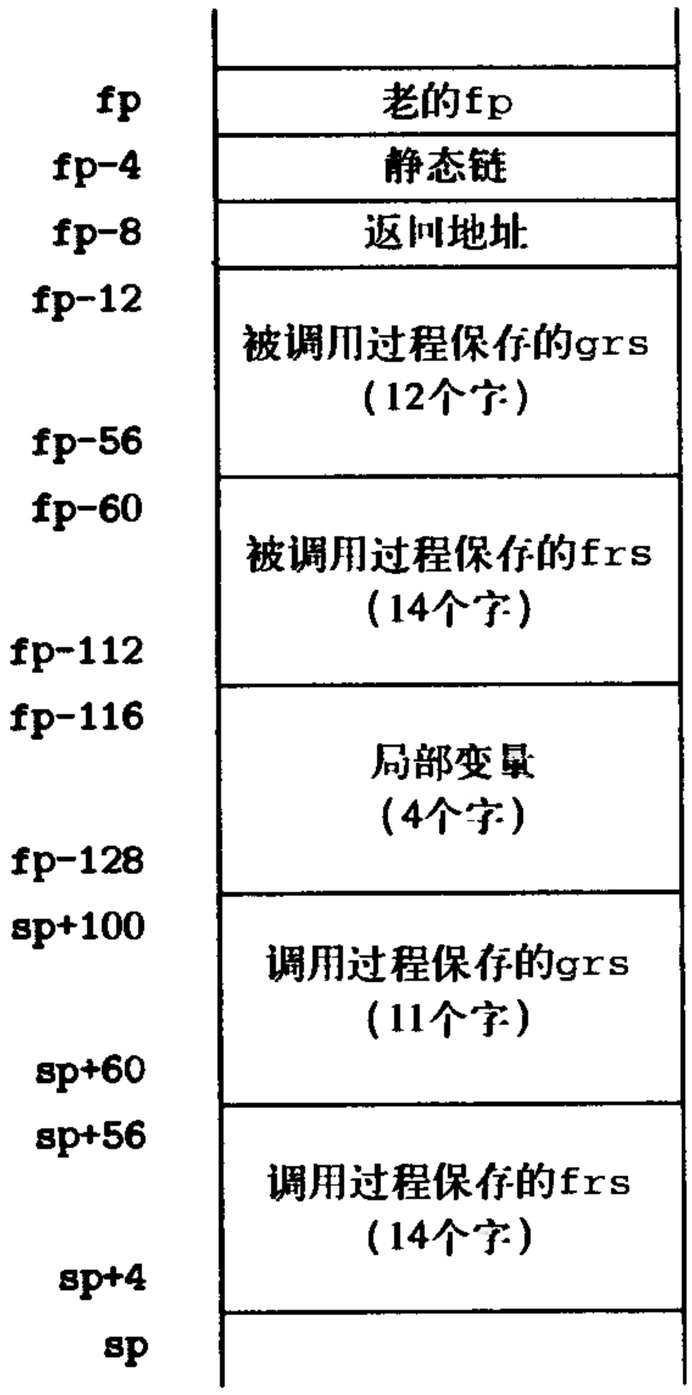


图5-9 用寄存器传递参数的过程调用例子的栈帧结构

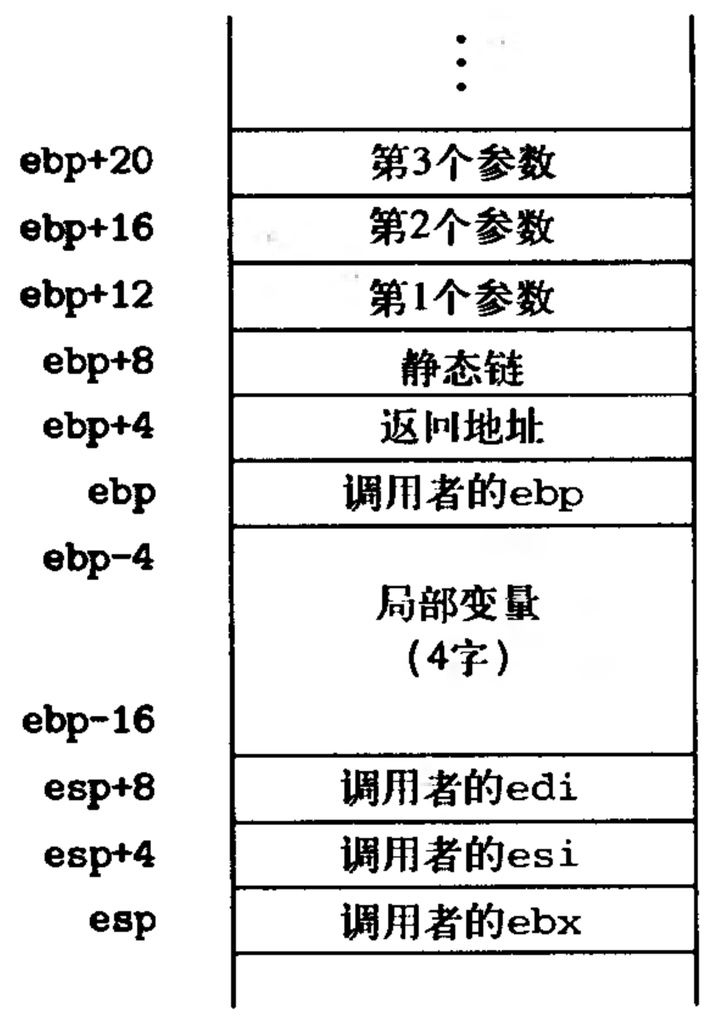


图5-10 在Intel 386体系结构系列上，用运行时栈传递参数的过程调用例子的栈帧结构

练习5.5要求你为这种模型生成过程调用、入口处理、参数的使用、出口处理及返回的代码。

5.6.3 用具有寄存器窗口的寄存器传递参数

寄存器窗口，如SPARC所提供的，简化了向被调用过程传递参数和从被调用过程返回结果的处理。它也在相当大的程度上减少了存/取指令的执行次数，因为它们能够提供更大的寄存器文件而不增加指明寄存器号所需要的位数（典型的实现提供了7个或8个窗口，总共128或144个整数寄存器），并且利用了过程调用期间的局部性。

寄存器窗口的用法规定将整数寄存器部分地划分为调用过程使用的和被调用过程使用的：被调用过程访问不到调用过程的*local*寄存器，反过来，调用过程也访问不到被调用过程的*local*寄存器；调用过程的*out*寄存器是被调用过程的*in*寄存器，并且主要的专用寄存器（包括返回地址和调用者的栈指针，后者变成了被调用过程的帧指针）或接收参数所使用的寄存器也是这样；被调用过程的*out*寄存器可作为临时用途和用于传递参数给它所调用的子程序。保存寄存器窗口中的寄存器的值到内存和从内存恢复它们是由窗口溢出和填充自陷处理程序来完成的，不需要用户代码。图5-11说明了在三个子程序的寄存器窗口之间的重叠关系。

bbs.theithome.com

版权所有 侵权必究

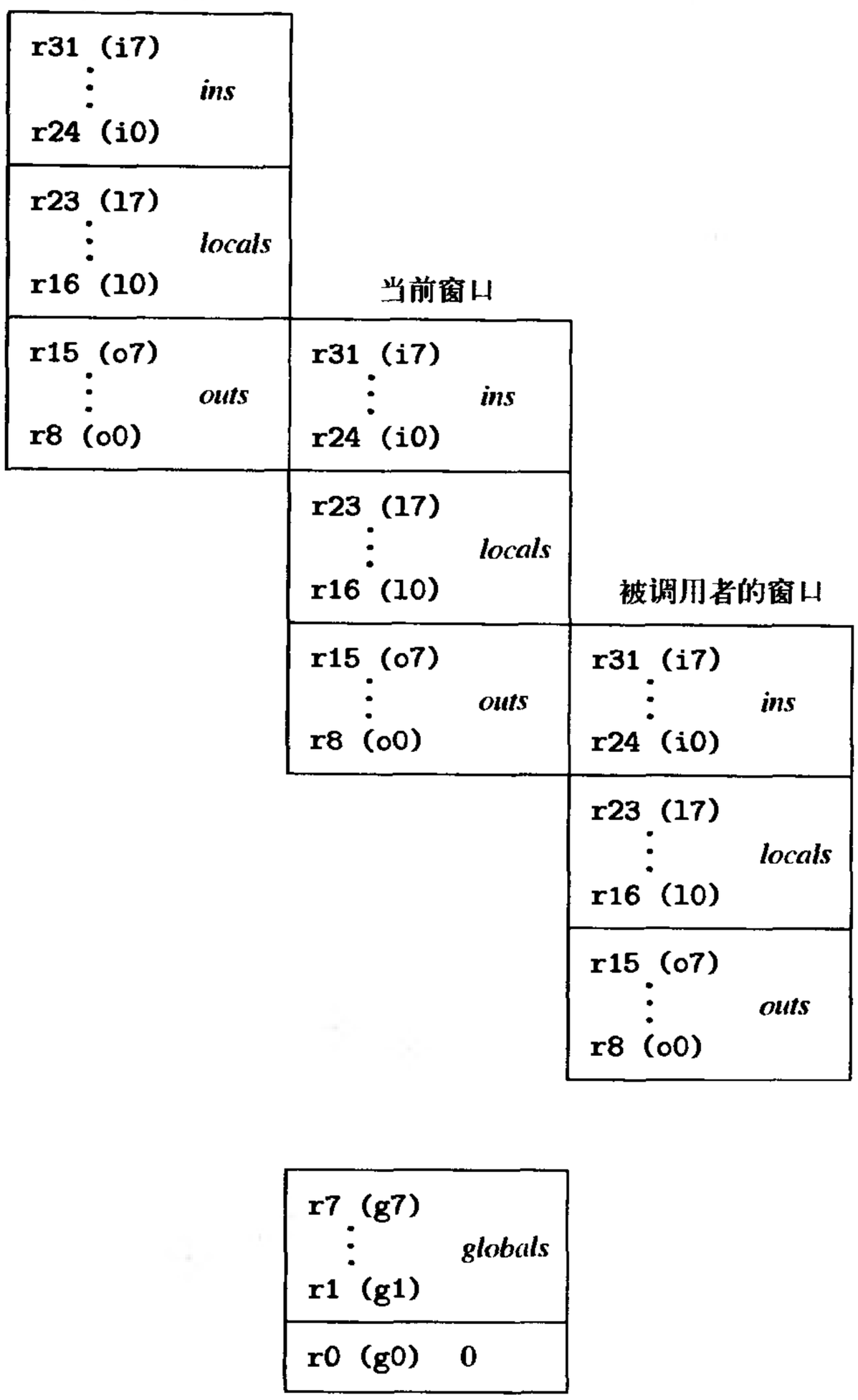


图5-11 SPARC中三个连续过程调用的寄存器窗口

当执行过程调用时，约定*out*寄存器o0到o5包含传递给当前过程的整型参数（浮点参数传递在浮点寄存器中），约定栈指针sp在o6中，帧指针fp在i6中。因此，被调用过程执行save指令将导致sp变为新fp。多余的参数（如果有的话）同平面寄存器模式一样在运行时栈中传递。当过程返回时，如果返回值是整型值，将它放在一个*in*寄存器中，如果是浮点值，将它放在浮点寄存器中。然后通过发出一条restore指令回到前一寄存器窗口并恢复调用者的栈指针。

图5-12展示了SPARC的典型栈帧布局。sp到sp+60的16个字的存储单元用于寄存器窗口溢出和填充自陷处理程序，这个程序将当前寄存器窗口的*in*和*out*存储于这片区域。正因如此，sp必须总是指向合法的溢出区；因此它只可由save和restore指令来修改。save指令用于前进到一个新的窗口并立即分配一个新栈帧，restore指令则逆转此过程。在地址sp+64的字用于返回结构或联合给调用过程，它由调用过程设置为接收该值的那片存储单元的地址。

124

bbs.theithome.com

整型参数的前6个字在寄存器中传递，后续参数在栈中传递。如果需要依次访问可变长度参数表，入口点的代码将从sp+68开始存储前6个参数。从sp+92开始的区域用于容纳其他参数和临时变量，以及用于保存全局和浮点寄存器（当这些寄存器需要保存时）。例如，若图5-12中的b是32，则整个栈帧的大小是148字节。

练习5.6要求你为这种模型生成过程调用、入口处理、参数的使用、出口处理及返回的代码。

125

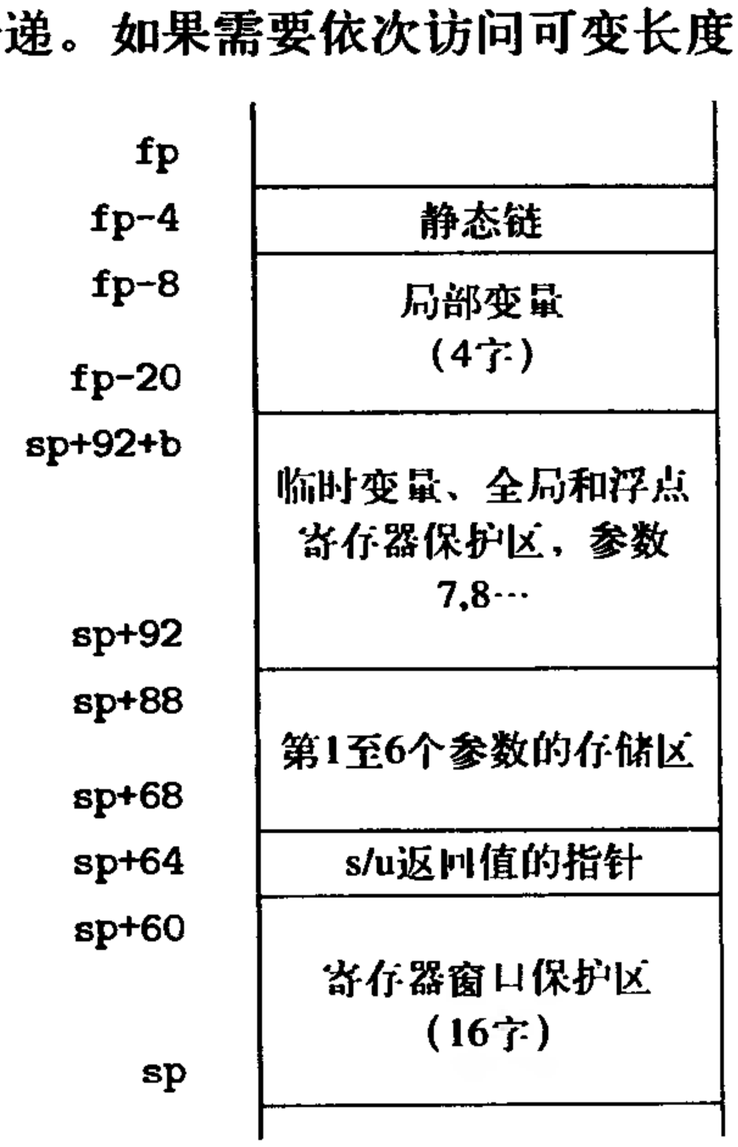


图5-12 在具有寄存器窗口的情况下过程调用例子的栈帧结构（s/u表示*structure*或*union*）

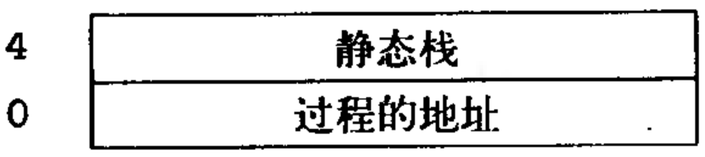


图5-13 包含过程的地址和它的静态链的过程描述字

126

5.6.4 过程值变量

调用一个由变量指定的过程，在设置其环境时需要特殊的处理。如果这个过程是局部的，则必须传递给它一个适合于它的静态链。最好的做法是，这个变量的值不是过程代码的地址，而是指向一个*过程描述字*（procedure descriptor）的指针；这个过程描述字中含有过程代码的地址和静态链，我们在图5-13给出了这样一个描述字。给定这种描述字设计，无论采用什么参数传递模式，与“调用”相关的代码都必须加以修改才能从参数描述字中得到静态链和调用过程代码的地址。为了调用这个过程，我们先要取过程的地址到寄存器，取静态链到适当的寄存器，然后执行基于寄存器的调用。因为这种代码序列很短而且是不变的，因此可以采用另一种方法。我们可以生成这种代码序列的一个副本，通过调用这个副本实现对所有过程变量的过程调用，并在代码末尾用一个基于寄存器的分支转移替代基于寄存器的调用，因为正确的返回地址就是用来调用这段代码序列的指令的地址。

5.7 代码共享与位置无关代码

前面我们一直隐含地假设，一个运行程序除了调用操作系统的服务之外，是自我包含的进程，即调用的所有库例程都是（在执行之前）与用户代码静态链接的，并且为了能够执行，所要做的一切只是：装载程序的执行映像到存储器，将环境初始化为适合于操作系统的标准程序设计模式，以及用适当的参数调用程序的主过程。这种模式有几个缺点，包括空间利用率不高，以及将用户的程序和库连接起来所需的时间较长。这些缺点可以通过使用所谓的*共享库*（shared libraries）而避免，共享库在程序执行时根据需要动态地被链接和装载，其代码由引用它们的所有程序所共享。共享库模式有以下一些优点：

1. 在文件系统中，共享库只需存在一个副本，无需作为每一个可执行程序的一部分而存在。

2. 在存储器中，共享库代码只需存在一个副本，无需作为每一个正在执行的程序的一部分而存在。

3. 假设在共享库的实现中存在一个错误，只要库的接口不变，这个库就可以用一个新的版本替换，而且使用它的程序无需重新进行链接——已经执行的程序可以继续使用文件系统中原先已调用的那个老库的副本，但是这个程序和其他程序的新的库调用将使用这个新库的副本。

注意，将程序与非共享库链接通常只需要程序调用的子程序，外加这些子程序的传递闭包，而不需要整个库。但是这并不能节省大量空间——尤其对于较大、较复杂的库而言，例如那些

bbs.theithome.com

超星公司制作 请尊重作者版权

实现窗口和图形系统的库——这种效果会传播到所有与一个给定的库进行链接的程序，因而受欢迎的总是共享库。

较复杂的问题是需要保持这种链接的语义与静态链接的语义尽可能一致，其中最重要的是，要能够在执行之前确定库中确实具有所有需要的子程序，这样才能预先指出动态链接是否能成功，即，是否会遇到未定义的或多次定义的外部符号。这种能力可通过给每一个共享库提供一个目录而获得，这个目录列出了库中所有的入口点和外部符号，以及库中每一个子程序用到的入口点和外部符号。图5-14给出了一个例子，其中第一列列出了这个共享库中的入口点和要让外部知道的名字，第二和第三列列出了它们所在的共享库，以及它们所使用的入口点和外部名。执行之前的链接仅仅检查要动态链接的库对应的目录表，因而能够同静态链接一样报告未定义的符号。运行时的动态链接则要保证当且仅当静态链接失败时才失败。但是，当用户在一个静态库的前面链接一个动态库，或静态地同时链接一个静态库和一个动态库，则我们还是会看到
127 有某些细微的不同。

给出的入口点和外部符号	使用的共享库	使用的入口点和外部符号
entry1	library1	extern1
		entry2
	library2	entry3
entry2	library1	entry1
	library2	entry4 entry5
extern1		

图5-14 共享库目录表的例子

另外，共享的代码不必构成一个库，尽管传统上使用术语“共享库”，它实际上仅仅是些个体，这些个体是程序在运行时选择链接的，而不是运行之前选择的。为了反映这一事实，我们在本节余下的部分中称这种个体为*共享对象*（shared object），而不称它们为共享库。

当单独运行一个程序时，共享对象确实对性能有影响，但在多道程序的系统中，这种影响会由于工作集所占空间的减少而完全（或几乎完全）抵消，因为工作集的减小会得到更好的存储页面和高速缓存性能。性能影响来源于两方面，即，运行时链接的开销，以及共享代码必须由*位置无关*（position-independent）代码组成，并且每一个被链接的程序都必须为共享对象的私有数据分配一个副本，这导致增加了访问这些数据的开销。位置无关代码是可以装载在不同程序中不同地址的代码。

下面我们讨论在支持共享对象过程中所涉及的问题。因为程序的大小不一，并且可能按任意顺序需要共享对象，为了使共享对象的每一个用户都能将共享对象自由地映射到存储器中任意的地址（除了可能需服从诸如页大小之类的对齐条件之外），必须实现位置无关性。访问共享对象内的局部变量不会有问题，因为它们要么在寄存器中，要么在通过寄存器访问的区域中，因而它们是进程私有的。访问全局变量会有问题，因为它们常常被放置在绝对地址中，而不是相对寄存器的地址中。调用共享对象中的子程序也会有问题，因为一直要到子程序已经装入后我们才能知道它的地址。因此，为了使得对象是位置无关的，并因此是可共享的，需要解决四

bbs.theithome.com

超星公司制作 请尊重作者版权

个问题，（1）如何在共享对象内实现控制转移？（2）共享对象如何访问它自己的外部变量？（3）如何在共享对象之间实现控制转移？（4）共享对象如何访问属于其他对象的外部变量？

在大多数系统中，在对象之内进行控制转移是容易的，因为这些系统都提供了相对程序计数器的（即基于位置的）分支转移和调用。即使对象作为一个整体必须按装载时可定位于任何位置的方式来编译，对象内的位置的相对偏移在编译时却是固定的，因此，相对程序计数器（PC）的转移正好可以满足其需要。如果体系结构没有提供相对PC的调用，则可以通过一系列指令来模拟，这些指令用调用目标相对当前点的偏移来构造调用目标的地址，其方法如下所述。 128

共享对象的一个实例访问它自己的外部变量时也需要采用位置无关的方法。因为处理器一般不提供相对PC的取和存指令，因此必须采用不同的技术。最常用的方法是使用所谓的*全局偏移表*（global offset table），即GOT。GOT驻存在一个所谓的动态区域内，这个动态区域位于对象的数据空间中，一开始时它包含着外部符号的偏移。当这个对象被动态链接时，GOT中的这些偏移被转变成当前进程数据空间内的绝对地址。这个动态区域是那些引用外部量的过程为获得对GOT的寻址能力而保留的。外部变量的寻址通过如下所示的LIR代码序列来实现:

```
        gp ← GOT_off - 4
        call   next,r31
next:   gp ← gp + r31
```

其中，GOT_off是GOT相对于使用它的这条指令的地址。这段代码设置全局指针gp指向GOT的基址。现在过程便可通过GOT中给出的外部变量的地址来访问外部变量了。例如，为了将其地址存储在GOT中a_off之处的整型外部变量a的值取至寄存器r3，应当执行如下代码:

```
r2 ← [gp+a_off]
r3 ← [r2]
```

第一条指令取a的地址到r2，第二条指令取a的值到r3。注意，为了保证这段代码能够工作，GOT不能大于存和取指令中可表示的偏移范围的绝对值。对于RISC机器，如果需要更大的范围，则需要如下面所示，在第一条取指令之前生成额外的指令来设置地址的高位。

```
r3 ← high_part(a_off)
r2 ← gp + r3
r2 ← [r2+low_part(a_off)]
r3 ← [r2]
```

其中high_part()和low_part()给出其参数的高部和低部，即参数被一分为二。由于这一原因，编译器可为位置无关的代码生成提供两种选择——一个含有，另一个不含额外的指令。

在对象之间进行控制转移不像在对象之内进行控制转移那样容易，因为在编译时不知道对象的相对位置，甚至在程序刚装入时也不知道其位置。标准的做法是，为对象所调用的每一个子程序提供一个桩（stub），这个桩作为被调用子程序的目标地址，它放置在调用对象的数据空间，而不是该对象的只读代码空间。这样，在执行期间当调用这个子程序时便能够修改这个桩，从而使得该子程序被装入（如果这是它在被调用对象中的第一次使用）和被链接。 129

有若干种策略能使这种桩起作用。例如，每一个桩可以由它对应的子程序名和一个对动态链接器的调用组成，动态链接器则用对实际子程序的调用指令来替换这个桩开始处的内容。如果系统提供相对寄存器的分支指令，则另一种可选方法是，将这些桩组织成一种称为*过程链接表*（procedure linkage table，PLT）的结构，保留第一个桩用于调用链接器，第二个桩用于标识共享对象，其他每一个桩构造它所对应子程序的再定位信息索引，然后转移到第一个桩（由此调用动态链接器）。这种方法能以一种懒惰方式来解析这些桩，即仅在需要时才进行解析，而且它的几个版本已用于若干个动态链接系统。以SPARC为例，假设我们有三个过程的桩，图5-15a

bbs.theithome.com

和b分别展示了在装载之前和第一个子程序与第二个子程序已经动态链接后的PLT形式。在装载之前，前两个PLT项为空，其他三个PLT项中每一项都包含了计算该项在PLT中的偏移并转移到第一项的指令。在将共享对象装载到内存时，动态链接器设置前两项，如图5-15b所示——第二项标识该共享对象，第一项创建栈帧并调动态链接器——其他项没有改变，如.PLT3项所示。当PLT第二项对应的过程，比如说f()，第一次被调用时，导致在.PLT2的桩被启用，此时这个桩仍然具有图5-15a所示形式。它放置由sethi计算出的相对.PLT0的偏移于寄存器g1中，然后转移到.PLT0。.PLT0处的指令启动动态链接器。动态链接器使用对象标识字和g1的值来获得f()的再定位信息，并相应地修改.PLT2项以创建一条转移至f()的jmp1指令，这条指令不计算返回地址（注意，在下一项中的sethi指令也会执行，它位于jmp1的延迟槽内，但这不会引起问题）。于是，从此以后，这个对象中对PLT中关于f()项的调用就将带有正确的返回地址转移至f()。

访问其他对象的外部变量本质上与访问自己的外部变量相同，不同的只是使用的是那个对象的GOT。

一个有点复杂的问题是如何在运行时形成过程的地址，将它作为变量的值保存，并将它与另外的过程地址进行比较。对于共享对象内的一个过程的地址，如果在共享对象内进行计算时是它的第一条指令的地址，而当在共享对象之外进行计算时，是与它对应的桩中第一条指令的地址，我们便违背了C语言和其他若干语言的特征。解决这个问题的方法比较简单：在共享对象内和共享对象外两种情况下，我们都使用过程描述字（如前一节所述），但将它们修改为包含PLT项的地址而不是过程代码的地址，同时还扩充它们使其包含含有这个被调用过程的对象的GOT地址。尽管这样做使得通过过程变量执行的调用所使用的代码序列需要保存和恢复GOT指针，但其结果导致这种描述字可统一用作过程变量的值，并且可以用它们正确地执行比较操作。

130

```
.PLT0: unimp
       unimp
       unimp
.PLT1: unimp
       unimp
       unimp
.PLT2: sethi (.-.PLT0),g1
       ba,a  .PLT0
       nop
.PLT3: sethi (.-.PLT0),g1
       ba,a  .PLT0
       nop
.PLT4: sethi (.-.PLT0),g1
       ba,a  .PLT0
       nop
       nop
```

a)

```
.PLT0: save  sp,-64,sp
       call  dyn_linker
       nop
.PLT1: .word object_id
       unimp
       unimp
.PLT2: sethi (.-.PLT0),g1
       sethi %hi(f),g1
       jmpl  g1+%lo(f),r0
.PLT3: sethi (.-.PLT0),g1
       ba,a  .PLT0
       nop
.PLT4: sethi (.-.PLT0),g1
       sethi %hi(h),g1
       jmpl  g1+%lo(h),r0
       nop
```

b)

图5-15 SPARC的PLT：a) 装载之前，b) 两个子程序已经被动态链接之后

5.8 符号和多态语言支持

本书介绍的多数编译技术主要是针对那些十分适宜编译的语言，即具有静态的、编译时的类型系统的语言。这类语言不允许用户增量地改变代码，并且通常更多地使用栈空间而不是堆存储空间。

本节我们简要地讨论编译那些用更动态的语言编写的程序时遇到的问题。这些语言包括

bbs.theithome.com

超星公司制作 请尊重作者版权

LISP、ML、Prolog、Scheme、SELF、Smalltalk、SNOBOL和Java等，它们一般用来处理符号数据，并且具有运行时给予的类型和多态运算。其他更多关于这类语言的处理，我们推荐读者阅读[Lee91]。为这类语言生成有效的代码需要解决5个主要的问题，这些问题超出了本书余下章节所考虑的范围，它们是：

1. 有效处理运行时类型检查和函数多态性的方法；
2. 语言的基本运算的快速实现；
3. 快速的函数调用以及使它们更快的优化方法；
4. 堆存储管理；
5. 对正在运行的程序进行增量性改变的有效处理方法。

这类语言多数都需要运行时的类型检查，因为它们给数据而不是给变量指定类型。因此，当编译时遇到一个形为“a+b”或“(plus a b)”的运算，或某个特定语言可能有的某种形式的运算时，我们一般无法知道要执行的这个运算类型是整数、浮点数、有理数，或是任意精度实数的加法；还是两个表或两个字符串的联接运算；还是由它的两个操作数类型而确定的另外的某种运算。因此编译器生成的代码总是包含类型信息，需要对操作数类型进行检查，并根据类型转移至相应代码来实现每一种运算。一般而言，最常见的需要检测和快速区分的是整数算术运算和对某种其他数据类型的运算，如LISP和ML中的表单元类型、SNOBOL中的字符串类型，等等。 131

多数系统中，体系结构对类型检查的支持都极少。不过SPARC提供了带标签的加法和减法指令，这种指令在执行加法或减法运算的同时，并行检查两个32位操作数低端的两位是否为0。如果不是0，根据用户的选择，要么产生一个自陷，要么设置一个条件码，并且不将运算结果写入目的寄存器。这样，通过在一个字的低端的两位放置部分标签信息，可以用很廉价的方法检测出一条加或减操作是否有整数操作数。其他有些RISC机器，如MIPS和PA-RISC，通过提供与直接数比较并分支的指令来支持某种更低级的类型检查。这种指令可以只用一条指令检查每一个操作数的标签，因而开销只有2~4个时钟周期，具体开销取决于分支延迟槽的填充。

SPARC中也可用一个字的低端的两位来做至少一种以上的类型检查，比如LISP中的表单元类型。假设表单元是双字，如果用第一个字的地址加3作为指向第一个表单元的指针（假设该指针在寄存器r1中），则对car和cdr域的访问将使用形如r1-3和r1+1的地址，并且当且仅当访问它们的存取指令中所使用的指针的低端两位为3（即标签3）时，其地址才是合法的（参见图5-16）。注意，这还留下了两个标签值（1和2），一个可用于另一种类型，另一个可作为指示器用于其他需要访问更详细类型信息的情形。

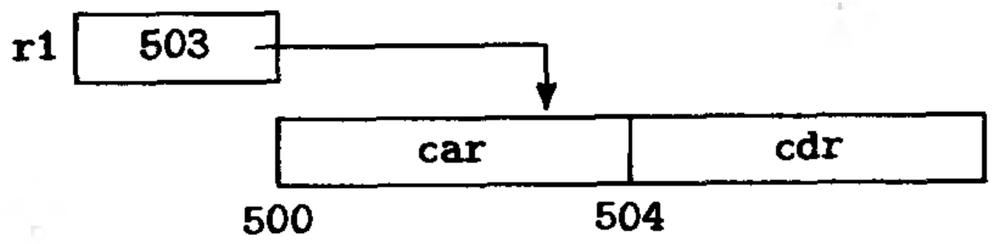


图5-16 LISP表单元和SPARC中指向它的带标签的指针

奇地址的标签设计方法也可用于其他几种RISC体系结构。[Lee91]中讨论了另外一些给数据加标签的有效方法。

除了其他内容，9.6节还关注了用于这种情形的一些软件技术，即在可能的地方给那种严格地说只有数据对象具有类型的语言中的变量指定类型。

快速的函数调用是这些语言的基本要求，因为这类语言强烈地鼓励将程序划分为许多小函数。多态性影响函数调用的开销，因为它导致在代码运行时才能根据参数的类型确定具体的调用。RISC在这一方面较为理想，因为它们一般通过提供分支并链接指令和在寄存器中传递参数实现了快速的函数调用，并且在多数情况下提供了根据一个或多个参数的类型决定分支的快 132

bbs.theithome.com

速方法。可以将一个参数的类型放在一个寄存器中，并将它转换为大小适当的偏移，然后按分支开关表转移到实现对应类型函数的代码。

动态语言和符号语言一般多使用堆空间，很大原因是因为它们设计的操作对象的大小和形状多是动态的，因此，具有非常有效的堆存储空间分配和恢复机制是基本的要求。存储空间的恢复通过垃圾收集而不是显式的释放来实现。这类语言常用的最有效的垃圾收集方法是阶段清除（generation scavenging），它所基于的原理是：已经存活得较久的对象还可能存活得更久。

最后，增量性地改变运行中程序代码的能力是这类语言多数具有的一个特征。在已有的编译实现中，这种能力一般通过函数的间接访问并结合运行时编译来实现。如果运行程序中函数的名字是一个含有该函数代码地址的单元，如5.6节和5.7节讨论的过程描述字那样，则至少可以在这个函数还不活跃时，通过改变这个间接单元中的地址使其指向该函数的新代码，从而改变该函数的代码和位置。在运行时能够调用编译器也使得即时的（on-the-fly）重编译成为可能，即，重新编译一个正在运行的子程序的能力。之所以这样做可能是因为已经高速缓存了一个已编译好的该子程序的副本，该子程序假定其参数具有特殊类型，但此假定现已不再起作用。Deutsch和Schiffman在基于Motorola M68000系统的Smalltalk-80实现[DeuS84]中使用了这种方法并得到了良好的效果。从那以后，这种方法就被重复地用于其他多态语言的实现中。

上面的讨论只粗略地描画了在设计一个动态语言的有效实现时所涉及的一些问题，关于这一领域的进一步参考文献请参见5.10节。

5.9 小结

本章我们回顾了在运行时支持高级语言共有的一些概念所涉及到的若干基本问题，包括数据类型和在运行时有效地表示它们的方法，存储分配和寻址方法，可见性和作用域规则，寄存器使用和寄存器管理的基本方法，单一过程的栈帧结构和运行时栈的完整组织，以及支持参数传递、过程调用、入口、出口和返回所涉及的问题。

因为这些概念大多在论述编译器的书中已有很好的介绍，因此我们的目的只是使读者回顾这些问题和处理它们的适当方法，并提供进一步阅读的参考文献。最后一节较为详细地讨论了
133 一些更高级的概念，如位置无关代码和对动态和多态语言的支持。

许多体系结构的应用程序二进制接口标准规定了上述某些问题必须怎样处理（如果工程必须与这个标准兼容的话），从而更容易实现与其他软件的互操作性。

本章后几节讨论了一些在一般介绍性课程中完全没有包含的内容。5.7节详细地讨论了如何利用位置无关代码和动态链接在进程之间支持代码共享。在5.8节，我们纵览了支持动态和多态语言的问题，这一方面的问题如果全面详细地讨论，足可以再写一本书。

5.10 进一步阅读

本章开始引用的UNIX系统V ABI文档是一种通用的规范[UNIX90a],它对特定处理器的增补（如SPARC[UNIX90c]、Motorola 88000[UNIX90b]、Intel 386体系结构系列[UNIX93]、Hewlett-Packard的PA-RISC [HewP91]）规定了栈的结构和调用约定。

统一码标准字符集规范参见[Unic90]，该规范规定的16位字符表示用于表示拉丁文、古代斯拉夫语、阿拉伯文、希伯来文和朝鲜语的字母表；南亚次大陆等国语言的字母表；中文汉字和日文汉字字库。

首先介绍利用形实转换程序（thunk）来计算传名字参数地址的论文是[Inge61]。

bbs.theithome.com

Weicker关于传递给过程的参数平均个数的统计见[Weic84]，其中附带dhrystone基准测试程序的原始版本。

关于在RISC体系结构中寄存器窗口减少了存取指令的执行的统计见[CmeK91]。

[GinL87]中阐述了共享库或共享对象的优点，并给出了它们在SPARC SunOS操作系统中的实现概述。该文也描述了一个程序在静态装载和动态装载之间的细微不同。

5.8节提到的符号和多态语言的详细描述见[Stee84](LISP)、[MilT90](ML)、[CloM87](Prolog)、[CliR91](Scheme)、[UngS91](SELF)、[Gold84](Smalltalk)、[GriP68](SNOBOL)和[GosJ96](Java)。

描述垃圾收集的阶段清除方法的论文是[Unga87]和[Lee89]。

描述即时编译的第一本著作是Deutsch和Schiffman的关于Smalltalk-80在基于Motorola M68000系统的实现[DeuS87]。实现动态和多态语言的其他问题（如推测控制流和数据流信息），在[Lee91]以及程序设计语言和函数程序设计年会论文集的很多论文中都有讨论。 134

5.11 练习

5.1 假设LIR既无取和存一个字节的指令，也无取和存半字的指令。(a) 写一个有效的LIR子程序，它将一字符串从寄存器r1指定的字节地址传送至r2指定的字节地址处，字符串的长度由寄存器r3给出，长度单位为字节。(b) 假定字符串采用C用null字符（即0x00）结束的约定，重写该子程序使它更有效地传送这种字符串。

5.2 测定你所用计算环境中的编译器是如何划分寄存器使用的。这可能只需简单地阅读手册，也可能需要一系列的试验。

5.3 假设有一个对f(i,x,j)的调用，参数传递采用传值方式，其中第一和第三个参数是整型值，第二个参数是单精度浮点值。执行该调用的过程是g()，g()与f()嵌套在相同的作用域内，因此，它们具有相同的静态链。假设参数如5.6.1节所述那样在平面寄存器文件的寄存器中传递，写出实现过程调用和返回的握手LIR代码。答案应当有五部分：(1) 调用代码，(2) f()的入口处理，(3) 使用第一个参数的代码，(4) f()的出口处理，(5) 返回点的代码。

5.4 写出上一练习的LIR代码或Intel 386体系结构系列汇编语言代码，假设参数如5.6.2节介绍的那样在运行时栈中传递。

5.5 假设参数如5.6.3节介绍的那样在寄存器窗口的寄存器中传递，写出上一练习的LIR代码。

ADV 5.6 对Pascal或类似的语言设计一种语言扩充，使其需要保存（某些）栈帧，假设保存在堆空间中，并且与原来的调用约定无关。为什么这种语言扩充可能是有用的?

5.7 写出5.3节描述的例程alloca()的LIR版本。

5.8 写一个（简单的）程序，它能够演示所有不同的参数传递规则，即，用你所选择的一种语言编写一个程序，并通过使用这五种参数传递方法说明该程序对每一种方法产生不同的输出。

5.9 根据Java语言的重载和覆盖机制，描述（或用ICAN写出）一个在运行时使用的过程，它能区别Java中调用的是具有相同名字的一系列方法中的哪一个方法。

bbs.theithome.com

版权所有 侵权必究

ADV 5.10 为具有传名字参数的语言写出过程调用的握手样例代码。具体地，写出调用f(n, a[n]) 的LIR代码，其中两个参数均传名字。

5.11 描述仅使用GOT而不使用PLT处理共享对象的方法，并给出代码例子。

RSCH 5.12 探讨通过将即时编译（5.8节有简单的讨论）与中间代码的解释相结合的方法支持多态语言涉及的有关问题。例如，这些问题包括控制如何在被解释的和被编译的代码之间进行转换（两个方向的转换），如何告知什么时候编译一个子程序是值得的，以及如何告知什么时候应当重新编译一个子程序或切换到解释该子程序。

135 ~ 136

bbs.theithome.com

越是公句制作请尊重作者版权

第6章　自动产生代码生成器

这一章我们先简要探讨有关从中间语言产生机器代码或汇编代码的问题，然后集中关注从机器描述自动产生代码生成器的方法。

在生成代码时需要考虑如下问题：

1. 目标机的寄存器、寻址方法和指令体系结构；
2. 必须遵循的软件约定；
3. 给变量绑定存储单元或符号寄存器的方法；
4. 中间语言的结构和特征；
5. 与目标机无直接对应指令的中间语言运算符；
6. 从中间代码转换到机器代码的方法；
7. 是生成汇编语言代码还是直接生成可链接或可重定位的机器代码。

这些问题的重要性，以及对这些问题所作的决策，随编写的编译器所要支持的语言和目标机体系结构的种类而变化，即，它支持的是一种语言和单一目标机、多种语言和一种体系结构、一种语言和多种体系结构，还是多种语言和多种体系结构。同时还要谨慎地考虑到一个已有的编译器在其生存期间还可能支持另外的源语言和体系结构。

如果我们肯定只是为单一体系结构生成编译器，采用自动方法从机器描述生成代码生成器就没有优势。在这种情况下，可以像一般编译著作所介绍的那样用手工方法。另一方面，如果我们希望为若干体系结构生成编译器，从机器描述自动生成代码生成器就特别值得。书写和修改机器描述一般要比从头书写代码生成器，或将现有的代码生成器移植到新的体系结构要容易。

有许多原因使得我们需要了解机器的体系结构——尽管有些原因不是很明显。目标机是我们生成的代码必须对准的目标，如果没有与它保持一致，代码就不能运行。一个不很明显的理由是，有些语言特征可能与目标机体系结构不十分匹配。例如，如果我们必须在32位字长的机器上执行64位的整数算术运算，就需要写出执行64位运算的开放的或封闭的（即内嵌的或子程序形式的）例程。对于复数则几乎总是出现类似的情况。如果机器有相对PC的条件转移，但其偏移较短而不能覆盖程序需要的长度，则为了覆盖程序所希望的偏移长度，我们就需要设计出能转移到足够远位置的方法。例如，对于一个转移至较远地址的条件分支，如果体系结构不支持远距离的条件转移指令，但对无条件指令没有限制，则实现它的一种可能的方法是用一个相反的条件转移绕过一个无条件转移的组合来替代它。

软件约定也同样重要。它们的设计必须支持源语言的特征，并服从已公布的标准，如应用程序二进制接口（ABI）定义（参见第5章的开始），否则，所生成的代码将不能满足要求。详细理解软件约定以及如何实现有效满足它们的代码，是生成高效代码的基础。

我们采用的中间语言在本质上对生成正确的代码没有决定性的影响，但它却是影响我们选择代码生成方法的主要因素。针对DAG、树、四元式、三元式、前缀波兰代码和包括若干不同的控制结构表示在内的其他形式，人们已经设计了不同的代码生成方法。

所生成的目标代码是采用汇编语言形式还是可重定位的二进制形式，主要与方便性和编译时性能的重要性有关。生成汇编代码要求我们在编译过程中包含汇编阶段，从而需要额外的时

bbs.theithome.com

超星公司制作 请尊重作者版权

间（用于运行汇编器和读写汇编代码），但它使代码生成器的输出易于阅读和检查，并且使得我们可以将诸如生成转移到代码中未知位置的问题留给汇编器。我们可以在编译时只生成符号标号，而将此标号在代码中的位置留给汇编器在以后确定。另一方面，如果直接生成可链接的代码，我们也仍然需要一种途径来输出符号形式的代码以便调试，尽管这种输出形式不必是完整的汇编语言，并且可以由反汇编器从目标代码生成，就像IBM的POWER和PowerPC编译器所做的那样（参见21.2.2节）。

6.1 简介代码生成器的自动生成

尽管手工书写的代码生成器效率高，而且实现起来也较快，但其缺点是它毕竟是手工实现的，因此比自动生成的代码生成器要难于修改和移植。目前已开发出来了若干种从机器描述构
138 造代码生成器的方法，我们这里介绍其中的三种，但详细程度有所不同。这三种方法都从低级中间代码开始，这种低级中间代码已展开了地址计算。

在所有三种方法中，代码生成器都对树进行模式匹配，尽管在前两种方法中树表现得不明显——这两种方法都是针对前缀波兰中间表示的。如4.9.4节所解释的，前缀波兰表示是前序遍历树的结果，因此树简单地隐藏在线性表示中。

6.2 语法制导技术

我们将介绍的第一种从机器描述产生代码生成器的方法是著名的Graham-Glanville方法，该方法以它的发明人命名。它用一种与上下文无关文法类似的规则来表示机器的操作，同时辅以对应的机器指令模板。当一个规则与前缀波兰中间代码流（它是树序列的前序遍历表示）中的一个子串相匹配，并且满足与它相连的语义约束时，则用规则左端符号的实例替换所匹配的部分，同时流出对应指令模板的实例。

```
r.2 ← r.1
r.2 ← k.1
r.2 ← r.1

r.3 ← r.1 + r.2
r.3 ← r.1 + k.2
r.3 ← k.2 + r.1

r.3 ← r.1 - r.2
r.3 ← r.1 - k.2

r.3 ← [r.1+r.2]
r.3 ← [r.1+k.2]
r.2 ← [r.1]

[r.2+r.3] ← r.1
[r.2+k.1] ← r.1
[r2] ← r.1
```

a)

```
r.2 ➡ r.1
r.2 ➡ k.1
r.2 ➡ mov r.2 r.1

r.3 ➡ + r.1 r.2
r.3 ➡ + r.1 k.2
r.3 ➡ + k.2 r.1

r.3 ➡ - r.1 r.2
r.3 ➡ - r.1 k.2

r.3 ➡ ↑ + r.1 r.2
r.3 ➡ ↑ + r.1 k.2
ϵ ➡ ↑ r.2 r.1

ϵ ➡ ← + r.2 r.3 r.1
ϵ ➡ ← + r.2 k.1 r.1
ϵ ➡ ← r.2 r.1
```

b)

```
or  r.1,0,r.2
or  0,k.1,r.2
or  r.1,0,r.2

add r.1,r.2,r.3
add r.1,k.2,r.3
add r.1,k.2,r.3

sub r.1,r.2,r.3
sub r.1,k.2,r.3

ld  [r.1,r.2],r.3
ld  [r.1,k.2],r.3
ld  [r.1],r.2

st  r.1,[r.2,r.3]
st  r.1,[r.2,k.1]
st  r.1,[r.2]
```

c)

图6-1 a) LIR指令，b) Graham-Glanville机器描述规则，c) 对应的SPARC指令模板

Graham-Glanville代码生成器由三部分组成，即中间语言转换、模式匹配器和代码生成。第一部分根据需要将编译前端的输出转换成适合于模式匹配的形式。例如，不能用机器操作表
139 示的源语言运算符被转换成子程序调用，而调用则被转换成对状态进行显式改变的指令序列。

bbs.theithome.com

越是公司制作请尊重作者版权

第二部分进行实际的模式匹配，确定用什么样的归约序列来归约中间代码的输入字符串。第三部分在执行上与第二部分是交织在一起的，它实际地生成合适的指令序列，并进行寄存器分配。本节余下部分将集中介绍模式匹配阶段和指令生成阶段，但对后者的关注程度要稍少一点。

考虑图6-1a中的LIR指令组，其中，每一个参数的位置受一个整数的限定，这个整数用于使中间代码子串与代码生成规则和指令模板相匹配。图6-1b和c展示了SPARC指令模板和对应的规则。在此图中，“r.*n*”表示一个寄存器，“k.*n*”表示一个常数，“ϵ”代表空字符串。数字既用于使代码的流出与匹配相一致，也用于表示规则中的语法约束——例如，字符串中的第一和第二操作数必须是相同的寄存器才算是一个成功的匹配，则应当用相同的整数来限定这两个操作数。

在前缀波兰代码中，我们用“↑”表示取操作，“←”表示存操作，mov表示寄存器到寄存器的传送。

作为一个例子，假设我们有图6-2所示的LIR代码。设r3和r4在这段代码结尾时是已死去的，于是如图6-3所示，我们没有必要在树表示中显式地包含它们，但却需要保留r1和r2。所得到的前缀波兰形式为:

```
↑r2 r8 ← + r8 8 + r2 ↑ r1 + r8 4 ← + r8 4 - r1 1
```

图6-4给出了对它进行分析过程中发生的模式匹配，以及为它生成的SPARC指令。下划线指出被匹配的字符串部分，它底下的符号指出由谁替代所匹配的子字符串；对于取操作和算术运算，结果寄存器由代码生成期间所使用的寄存器分配器来确定。

```
r2 ← [r8]
r1 ← [r8+4]
r3 ← r2 + r1
[r8+8] ← r3
r4 ← r1 - 1
[r8+4] ← r4
```

图6-2 用于Graham-Glanville代码生成的LIR指令序列

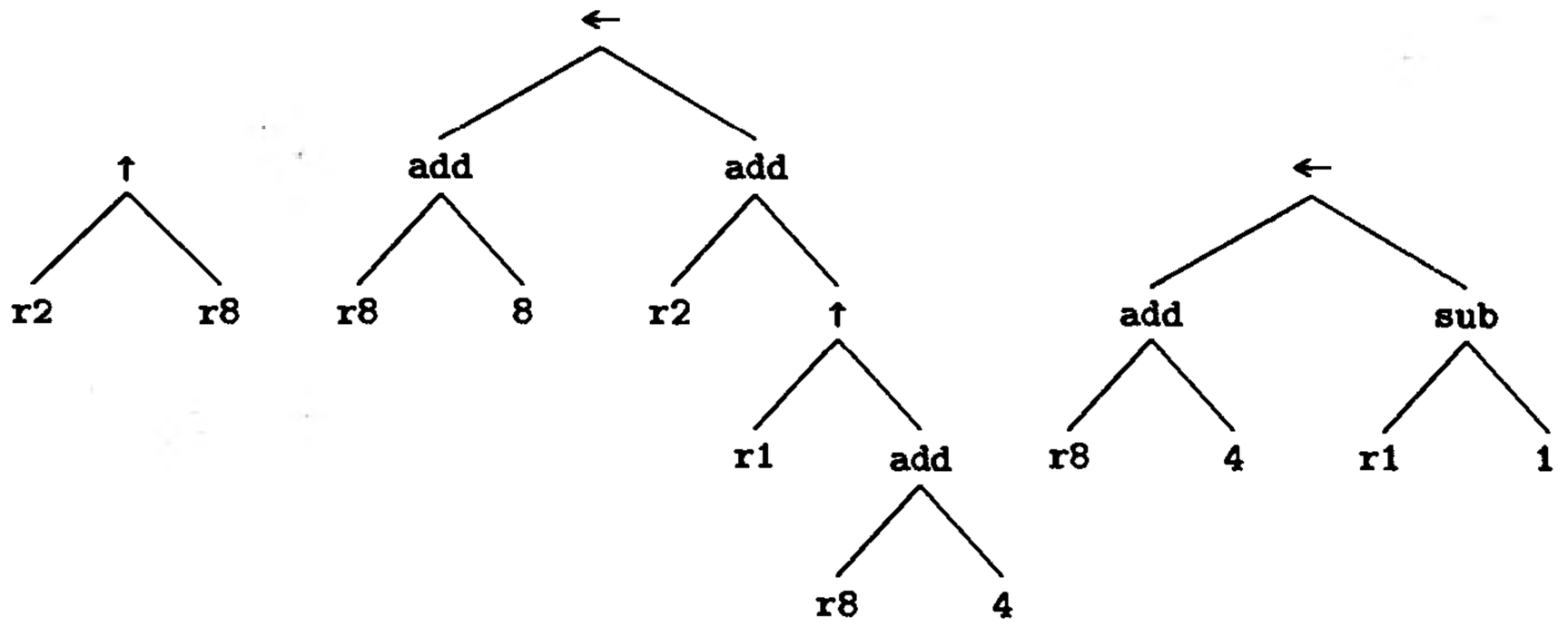


图6-3 图6-2中LIR代码序列对应的树

```
↑ r2 r8 ← + r8 8 + r2 ↑ r1 + r8 4 ← + r8 4 - r1 1
ϵ
← + r8 8 + r2 ↑ r1 + r8 4 ← + r8 4 - r1 1
                r1
← + r8 8 + r2 r1 ← + r8 4 - r1 1
          r3
← + r8 4 r3 ← + r8 4 - r1 1
ϵ
← + r8 4 - r1 1
          r4
← + r8 4 r4
ϵ
```

a)

```
ld    [r8,0],r2
ld    [r8,4],r1
add   r2,r1,r3
st    r3,[r8,4]
sub   r1,1,r4
st    r4,[r8,4]
```

b)

图6-4 a) 分析过程，b) 为图6-3的树流出的指令

bbs.theithome.com

6.2.1 代码生成器

这个代码生成器本质上是一个SLR（1）分析器，它执行通常的移进和归约动作，但流出的是机器指令而不是语法分析树或中间代码。在本质上，这个分析器识别这样一种语言，它的产生式是非终结符N可用“ϵ”替代的机器描述规则以及一个额外的产生式$S \Rightarrow N^*$。不过，它与
140 SLR分析有几点重要的不同。

“ϵ”允许出现在规则的左边，并将它看作是非终结符。与语法分析中不同的是，机器描述的文法几乎总是有歧义的。歧义性通过两种方法来解决。一种方法是归约动作上的贪婪移进动作（即算法是贪婪的或贪食的），并且较长的归约优先于较短的归约，这样使得匹配的是可能的最长字符串。另一种方法是在形成语法分析表时按某种顺序排列这些规则，使得代码生成器在代码生成进行归约时，取的是第一个相匹配的规则。这样，在指定这些规则时，人们可以通过用一种特定的方法对规则排序，或者通过在代码生成器中建立一种代价评估，从而使得代
141 码生成器偏向某些特定的选择。

这个代码生成器算法用到了7种全局数据类型、一个栈和两个由代码生成器的产成器构造出来的函数，它们是：

```
Vocab = Terminal ∪ Nonterminal
ExtVocab = Vocab ∪ {'ϵ','$'}
VocabSeq = sequence of Vocab
ActionType = enum {Shift,Reduce,Accept,Error}
|| type of machine grammar rules
Rule = record {lt: Nonterminal ∪ {'ϵ'},
               rt: VocabSeq}
|| type of parsing automaton items that make up its states
Item = record {lt: Nonterminal ∪ {'ϵ'},
               rt: VocabSeq,
               pos: integer}
ActionRedn = ActionType × set of Item
Stack: sequence of (integer ∪ ExtVocab)
Action: State × ExtVocab ⟶ ActionRedn
Next: State × ExtVocab ⟶ State
```

注意，机器文法中的终结符集合和非终结符集合几乎总是有非空的交集。

这个代码生成器以前缀波兰中间代码串InterCode作为输入，InterCode必须是由Terminal的成员组成的序列。类型ActionRedn的元素是一个偶对<a，r>，其中a是ActionType的成员，r属于Item组成的集合，并且它们满足$r \neq \phi$ 当且仅当a=Reduce。其算法如图6-5所示。Get_Symbol()返回它的参数的第一个符号，Discard_Symbol()删除那个符号。

函数Emit_Instrs(*reduction*，*left*，*right*)从*reduction*给出的规则中选择一个规则，通过使用栈中的信息将对应的模板实例化而流出一至多条指令，设置*left*为所用规则的左端符号实例化后的符号，并设置*right*为该规则右端的长度。要理解为什么Emit_Instrs()需要使用栈中的信息来决定流出什么样的指令序列，考虑图6-1中的第二条规则。只要大多数SPARC指令中的13位无符号直接数域能容纳得下由k.1匹配的常数，使用这条规则就没有问题。如果这个常数容纳不下，Emit_Instrs()将需要生成一条sethi和一条or指令来在寄存器中构造该常数，其后再跟随一条在图6-1第一条规则中给出的相应的三寄存器指令。我们可以通过提供额外一条规则和指令模板来将这种选择引入到图6-1的规则中：

bbs.theithome.com

```
procedure Generate(InterCode) returns boolean
   Intercode: in VocabSeq
begin
   state := 0, right: integer
   action: ActionType
   reduction: set of Item
   left, lookahead: ExtVocab
   Stack := [0]
   lookahead := Lookahead(InterCode)
   while true do
      action := Action(state,lookahead)↓1
      reduction := Action(state,lookahead)↓2
      case action of
Shift:   Stack ⊕= [lookahead]
         state := Next(state,lookahead)
         Stack ⊕= [state]
         Discard_Symbol(InterCode)
         if InterCode = [] then
             lookahead := '$'
         else
             lookahead := Get_Symbol(InterCode)
         fi
Reduce:  Emit_Instrs(reduction,left,right)
         for i := 1 to 2 * right do
            Stack ⊖= -1
         od
         state := Stack↓-1
         if left ≠ 'ϵ' then
            Stack ⊕= [left]
            state := Next(state,left)
            Stack ⊕= [state]
         fi
Accept:  return true
Error:   return false
      esac
   od
end    || Generate
```

图6-5 代码生成算法

```
(empty)          r.2⇒k.1          sethi  %hi(k.1), r.2
                                  or     %lo(k.1), r.2, r.2
```

但即使这样，我们也仍然需要检查常量操作数的长度，否则，这条规则可能会在需要使用时而未被使用——它匹配的是一个较短的子串，这个子串比任何含一个运算符和一个常量操作数的规则所匹配的子串都要短，所以在寻找一个常量操作数时，我们应总是继续移进而不是进行归约。

Emit_Instrs()中隐藏的另一个问题是寄存器分配。寄存器分配可以用3.6节描述的方法来处理，或者更好的做法是，给标量变量和临时变量指定其个数无限制的符号寄存器，最终由寄存器分配器（参见16.3节）给这些符号寄存器指定机器寄存器或存储单元。

6.2.2 代码生成器的产生器

如图6-6和图6-7所示，代码生成器的产生器基于建立SLR语法分析表时所使用的项目集合结构（其中项目（item）是文法中其右边带有“·”）的产生式，但有若干修改。项目用前面声明的全局类型Item表示，一个项目[$l \Rightarrow r_1 \ldots r_i \cdot \ r_{i+1} \ldots r_n$]对应于记录<lt:$l$, rt:$r_1 \ldots \ r_n$,

bbs.theithome.com

142 ~ 144

pos:*i*>。主例程Gen_Tables()在MGrammar中的规则集合是一致的（见图6-7），并且所有句法阻滞（见6.2.4节）都可以修复时返回true；否则返回false。图6-6中的全局变量全局于本节所有的例程。数组ItemSet[]以状态编号作为索引，用于容纳项目被构造时的项目集。Vocab是规则中出现的终结符和非终结符词汇表。像语法分析器的产生器一样，代码生成器的产生器通过构造项目集并将它们每一个与一个状态相关联来推进处理过程。函数Successors()计算每一个状态的后继对应的项目集，并且填充每一个转换的Action()和Next()之值。

```
MaxStateNo: integer
MGrammar: set of Rule
ItemSet: array [··] of set of Item

procedure Gen_Tables( ) returns boolean
begin
   StateNo := 0: integer
   unif: boolean
   item: Item
   rule: Rule
   || remove cycles of nonterminals from machine grammar
   Elim_Chain_Loops( )
   MaxStateNo := 0
   || generate action/next tables
   ItemSet[0] := Closure({⟨lt:rule.lt,rt:rule.rt,pos:0⟩
      where rule ∈ MGrammar & rule.lt = 'ϵ'})
   while StateNo ≤ MaxStateNo do
      Successors(StateNo)
      || process fails if some state is not uniform
      if !Uniform(StateNo) then
          return false
      fi
      StateNo += 1
   od
   || process fails if some syntactic blockage is not repairable
   unif := Fix_Synt_Blocks( )
   Action(0,'$') := ⟨Accept,∅⟩
   return unif
end    || Gen_Tables

 procedure Successors(s)
    s: in integer
 begin
    NextItems: set of Item
    v: Vocab
    x: ExtVocab
    j: integer
    item, item1: Item
    for each v ∈ Vocab do
       || if there is an item [x ➡ α·vβ] in ItemSet[s],
       || set action to shift and compute next state
       if ∃item ∈ ItemSet[s] (v = item.rt↓(item.pos+1)) then
          Action(s,v) := ⟨Shift,∅⟩
          NextItems := Closure(Advance({item1 ∈ Itemset[s]
             where v =item1.rt↓(item1.pos+1)}))
          if ∃j ∈ integer (ItemSet[j] = NextItems) then
             Next(s,v) := j
          else
```

图6-6 构造SLR（1）语法分析表

bbs.theithome.com

```
            MaxStateNo += 1
            ItemSet[MaxStateNo] := NextItems
            Next(s,v) := MaxStateNo
         fi
      || if there is an item [x ⇒ α·] in ItemSet[s],
      || set action to reduce and compute the reduction
      elif ∃item ∈ ItemSet[s] (item.pos = |item.rt|+1) then
         Reduction := {item ∈ ItemSet[s] where item.pos = |item.rt|+1
            & (item.lt = 'ϵ' ∨ (v ∈ Follow(item.lt)
            & ∀item1 ∈ ItemSet[s] (item1 = item ∨ |item1.rt| ≤ |item.rt|
               ∨ item1.pos ≤ |item.rt| ∨ v ∉ Follow(item1.lt))))}
         Action(s,v) := ⟨Reduce,Reduction⟩
      || otherwise set action to error
         else
            Action(s,v) := ⟨Error,∅⟩
         fi
      od
end      || Successors
```

图6-6 （续）

```
procedure Uniform(s) returns boolean
   s: in integer
begin
   u, v, x: Vocab
   item: Item
   for each item ∈ ItemSet[s] (item.pos ≤ |item.rt|) do
      if item.pos ≠ 0 then
         x := Parent(item.rt↓item.pos,item.rt)
         if Left_Child(item.rt↓item.pos,item.rt) then
            for each u ∈ Vocab do
               if Action(s,u) = ⟨Error,∅⟩
                  & (Left_First(x,u) ∨ Right_First(x,u)) then
                  return false
               fi
            od
         fi
      fi
   od
   return true
end    || Uniform

procedure Closure(S) returns set of Item
   S: in set of Item
begin
   OldS: set of Item
   item, s: Item
   repeat
      || compute the set of items [x ➡ αv·β]
      || such that [x ➡ α·vβ] is in S
      OldS := S
      S ∪= {item ∈ Item where ∃s ∈ S.(s.pos < |s.rt|
         & s.(s.pos+1) ∈ Nonterminal
         & item.lt = s.(s.pos+1) & item.pos = 0)}
   until S = OldS
   return S
```

图6-7 构造SLR（1）语法分析表时使用的函数Uniform()、closure()和Advance()

bbs.theithome.com

```
    end    || Closure

    procedure Advance(S) returns set of Item
       S: in set of Item
    begin
       s, item: Item
       || advance the dot one position in each item
       || that does not have the dot at its right end
       return {item ∈ Item where ∃s ∈ S (item.lt = s.lt
          & s.pos ≤ |s.rt| & item.rt = s.rt
          & item.pos = s.pos+1)}
    end    || Advance
```

图6-7 （续）

我们较后再描述函数Elim_chain_Loops()和Fix_Synt_Block()。函数Uniform()确定规则集合是否满足所谓一致性的特点。一致性的定义如下：一个规则集合是一致的(uniform)，当且仅当二元运算符的任意左操作数，在含有那个二元运算符的任意前缀波兰字符串中，都是那个运算符的合法左操作数，并且，对于右操作数和一元运算符的操作数也类似。为了使规则集合适合于Graham-Glanville代码生成，规则集合必须是一致的。例程Uniform()使用了下面定义的两个函数，即Parent()和Left_Child()，以及由后面段落所讨论的那些关系，即Left_First(*x*, *u*)和Right_First(*x*, *u*)，所定义的两个函数；如果分别有*x Left First u*和*x Right First u*，这两个函数返回true；否则返回false。

函数

Parent : *Vocab* × *Prefix* → *Vocab*

返回第一个参数在由第二个参数给出的树中的父亲，其中*Prefix*是由规则生成的语句的前缀集合。函数

Left_Child : *Vocab* × *Prefix* → *boolean*

返回一个布尔值，它指出在由第二个参数给出的树中，第一个参数是否是其父亲的最左边的儿子。

这个算法中和这些函数定义中用到了如下一些关系（其中*BinOp*和*UnOp*分别是二元运算符和一元运算符符号集合，而且除ϵ之外的小写希腊字母均代表符号字符串）：

$Left \subseteq (BinOp \cup UnOp) \times Vocab$

x Left y 当且仅当存在一个规则$r \Rightarrow \alpha xy\beta$，其中$r$可以是$\epsilon$。

$Right \subseteq BinOp \times Vocab$

x Right y 当且仅当对于某个$\beta \neq \epsilon$，存在一个规则$r \Rightarrow \alpha x\beta y\gamma$，其中$r$可以是$\epsilon$。

$First \subseteq Vocab \times Vocab$

x First y 当且仅当存在一个推导$x \overset{*}{\Rightarrow} y\alpha$，其中$\alpha$可以是$\epsilon$。

$Last \subseteq Vocab \times Vocab$

x Last y 当且仅当存在一个推导$x \overset{*}{\Rightarrow} \alpha y$，其中$\alpha$可以是$\epsilon$。

$EpsLast \subseteq Vocab$

$EpsLast = \{x \mid$ 存在一个规则$\epsilon \Rightarrow \alpha y$且$y\ Last\ x\}$。

$RootOps \subseteq BinOp \cup UnOp$

$RootOps = \{x \mid$ 存在一个规则$\epsilon \Rightarrow x\alpha$且$x \in BinOp \cup UnOp\}$。

函数*Follow* : *Vocab* → *Vocab*可以用下面的辅助函数*Follow*1: *Vocab* → *Vocab*，以及集合*EpsLast*和*RootOps*来定义：

bbs.theithome.com

越是公句制作请尊重作者版权
145
~
147

$$Follow1(u) = \{v \mid \exists \text{ a rule } r \Rightarrow \alpha x y \beta \text{ such that } x \; Last \; u \text{ and } y \; First \; v\}$$

$$Follow(u) = \begin{cases} Follow1(u) \cup RootOps & \text{if } u \in EpsLast \\ Follow1(u) & \text{otherwise} \end{cases}$$

作为Graham-Glanville代码生成器的一个例子，考虑图6-8中非常简单的机器描述规则。在这个例子中，我们用传统的方式书写项目，即将<lt:x, rt:$\alpha y \beta$, pos:|α|>写为[x⇒$\alpha \cdot y\beta$]。我们首先在图6-9中给出*Left*、*Right*等关系以及一些集合和函数，然后对给定的文法追溯Gen_Tables()若干步。一开始，我们有StateNo=MaxStateNo=0并且ItemSet[0]={[∈⇒· ←rr]}。接着，我们调用函数Successors (0)，该函数设置v='∈'，计算Action (0, '←') =<Shift, ∅>，并设置集合NextItems为

```
Closure (Advance ({[∈⇒· ← r r]}))
```

它计算出来的值为

```
NextItems={[∈⇒← ·r r], [r⇒ ·r], [r⇒ · +rr],
           [r⇒ · + r k], [r⇒ · ↑r]}
```

现在MaxStateNo增加到1，ItemSet[1]被设为刚刚由NextItems计算出来的值，并且Next (0, '←') 被设置为1。

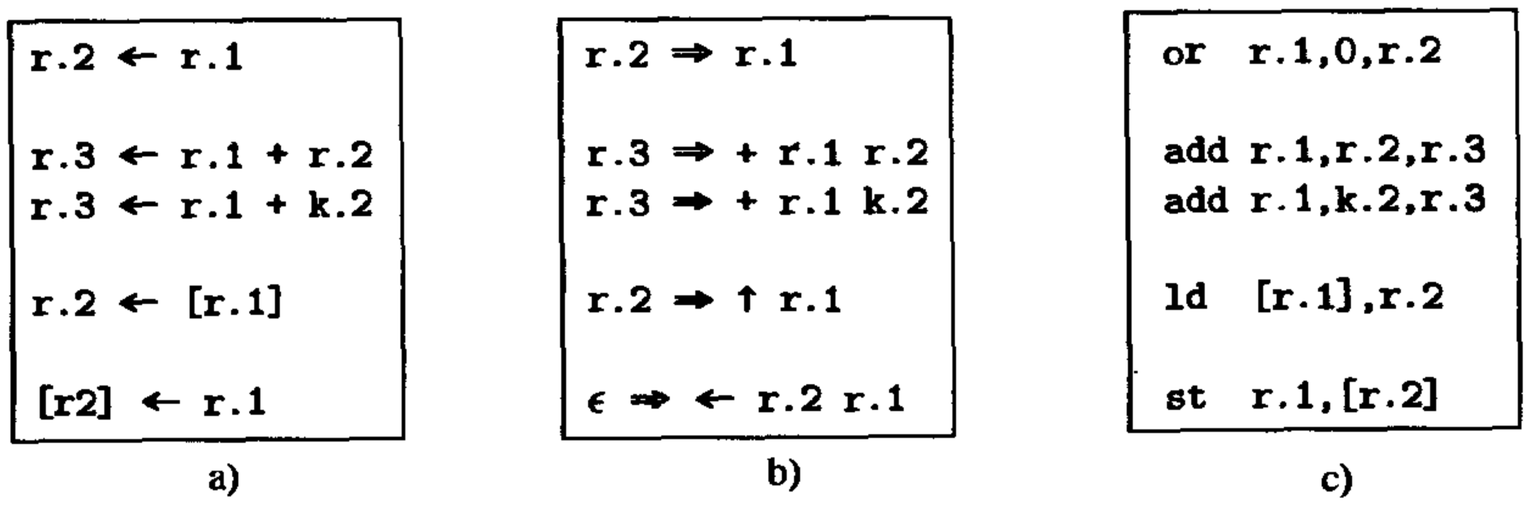


图6-8 a) LIR指令，b) Graham-Glanville机器描述规则，c) 对应的SPARC指令模板

```
'+' Left 'r'      '↑' Left 'r'      '←' Left 'r'

'+' Right 'r'     '+' Right 'k'     '←' Right 'r'

'r' First 'r'     'r' First '+'     'r' First '↑'

'r' Last 'r'      'r' Last 'k'

    EpsLast = {'r','k'}      RootOps = {'←'}

    Follow1('∈') = {'r','+','↑','←'}
    Follow1('r') = {'r','+','↑','←'}
    Follow1('k') = {'r','+','↑','←'}
    Follow1('+') = {'r','+','↑','←'}
    Follow1('↑') = ∅
    Follow1('←') = ∅

    Follow('∈') = {'r','+','↑','←'}
    Follow('r') = {'r','+','↑','←'}
    Follow('k') = {'r','+','↑','←'}
    Follow('+') = {'r','+','↑','←'}
    Follow('↑') = ∅
    Follow('←') = ∅
```

图6-9 图6-8中的机器描述文法例子的关系、集合和函数

bbs.theithome.com

随后，我们计算Uniform (1)。ItemSet[1]中所有的项目都以点打头，因此Uniform()是ture。在后面的步骤中，Uniform()将总是返回true。

接下来，StateNo被设置为1，我们调用Successors(1)。它首先设置v= 'r'，计算Action (1, 'r') =<Shift, ∅>，然后设置集合NextItems为

```
Closure (Advance ({[∈⇒← · r r], [r⇒ · r]}))
```

它计算出来的值为

```
NextItems={[∈⇒←r · r], [r⇒r · ], [r⇒ · r],
           [r⇒ · + r r], [r⇒ · +r k], [r⇒ · ↑r]}
```

现在MaxStateNo增加到2，ItemSet[2]被设为刚刚为NextItems计算的值，并且Next (1, 'r') 被设置为2。

接下来，Successors (1) 设置v= '+'，计算Action(1, '+')=<Shift, ∅>，然后设置集合NextItems为

148

```
Closure (Advance ({[r⇒ · + r r], [r⇒ · + r k]}))
```

它计算出来的值为

```
NextItems={[r⇒+ · r r], [r⇒+ · r k], [r⇒ · r],
           [r⇒ · + r r], [r⇒ · + r k], [r⇒ · ↑r]}
```

现在MaxStateNo增加到3，ItemSet[3]被设为刚刚为NextItems计算的值，并且Next (1, '+') 被设置为3。

此代码生成器的产生器继续产生shift动作，直到它到达MaxStateNo=9。此时，ItemSet[9]是{[∈⇒ + r k ·], [r ⇒k ·]}，它导致一个Reduce动作，即，

```
<Reduce, {[∈⇒ + r k · ]}>
```

所生成的Action/Next表如图6-10所示。图6-11给出的是分析自动机的图形表示。在表和图形两种表示中都只给出了非错误的转换。表中含有数字的登记项对应于移进动作，例如，在状态3的展望符号是 'r' 时，我们移进到状态6。含归约动作的登记项给出的是用来归约的项目集，例如，在状态5的展望符号是 '←'、'↑'、'+' 或 'r' 时，用项目集{[∈⇒← rr ·]}来归约。在图形表示中，用从一个状态到另一个状态的箭头，且其上标有展望符号来表示移进动

149 作。例如，从状态3到状态6且带有标志 'r' 的箭头对应于前面所述表形式的转换。

State Number	Lookahead Symbol					
	←	↑	+	r	k	$
0	1					Accept
1		4	3	2		
2		4	3	5		
3		4	3	6		
4		4	3	7		
5	{[∈ ➡ ← r r ·]}					
6		4	3	8	9	
7	{[r ➡ ↑ r ·]}					
8	{[r ➡ + r r ·]}					
9	{[r ➡ + r k ·]}					

图6-10 图6-8中给出的机器描述文法的Action/Next表

bbs.theithome.com

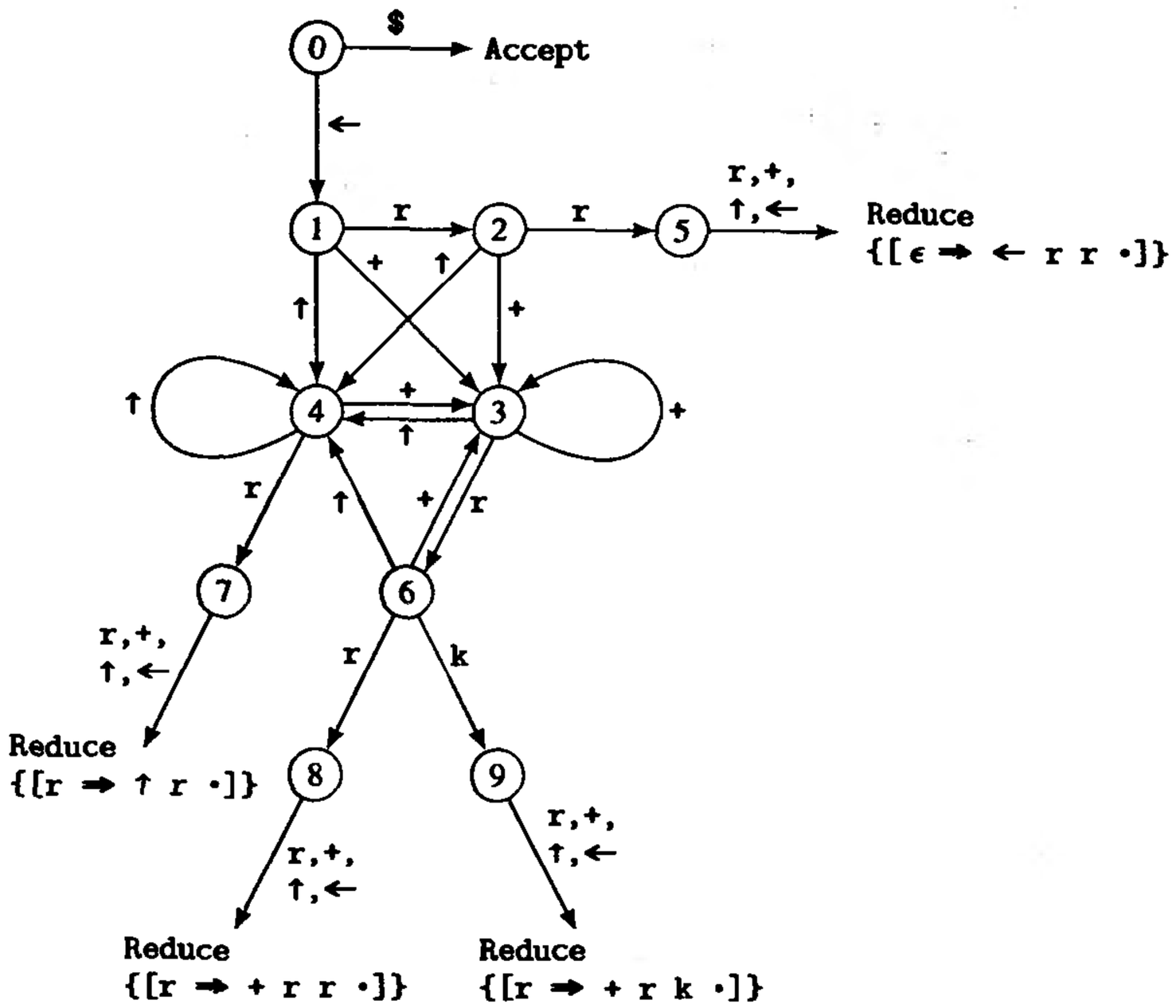


图6-11 由图6-8机器描述文法生成的代码生成自动机

下面，我们用前面给出的Action/Next表对如下中间代码流

```
←  +  r1  2  +  ↑  r3  3  $
```

追溯代码生成器的动作。过程是首先设置开始状态为0，将0压入栈顶，然后取符号‘←’作为lookahead的值。在状态0时对‘←’的动作是移进，于是该符号被压入到栈顶，从Action/Next表中取出下一状态并压入栈顶，从输入字符串中忽略这个展望符号，下一个符号成为展望符号，这个符号即‘+’。现在，栈的内容为:

1	'←'	0

在状态1下‘+’的动作是移进并且下一状态是3。其结果导致栈变为:

3	'+'	1	'←'	0

在状态3下‘r1’的动作是移进并进入状态6，因此，栈变为:

6	r1	3	'+'	1	'←'	0

接下来的移进动作使分析器进入状态9，此时展望符号为‘+’，栈为:

9	2	6	r1	3	'+'	1	'←'	0

150

在状态9处，合适的动作是用项目集{[r ⇒ + r k ·]}进行归约，于是Emit_Instrs()被调用。它为加法运算结果分配一个寄存器，即r2，并输出指令

```
add     r1, 2, r2
```

left的值被设为r2，right的值被设为3，于是，从栈中弹出6项并忽略它们，并且r2和下一状态（2）被压入栈中，结果为:

2	r2	1	'+'	0

bbs.theithome.com

超星公司制作 请尊重作者版权

我们将后面的处理留给读者，并要求核实由它生成的就是下面的指令序列，其中假定寄存器r4和r5的分配也如下所示:

```
    add    r1,2,r2
    ld     [r3],r4
    add    r4,3,r5
151 st     r2,[r5]
```

6.2.3 删除链循环

在文法分析中很少有链循环的情况，链循环是由一组非终结符组成的集合，其中每一个非终结符可以从另一个非终结符导出。但是，这种链循环在机器描述中却特别常见。作为链循环作用的一个例子，考虑由下面规则组成的简单文法（尽管该文法生成的语言是空集，但这没有关系——增加生成终结符的产生式不影响链循环的表示):

```
r ⇒ ↑ r
r ⇒ s
s ⇒ t
t ⇒ r
ϵ ⇒ ← s t
```

这个文法的分析自动机如图6-12所示。现在，如果我们以中间代码串←r1↑r2作为输入，那么，在处理← r1之后，自动机处在状态1，栈为

1	'←'	0

并且展望符号是‘↑’。从这个状态，该代码生成器流出一条寄存器到寄存器的传送指令，同时返回到相同的状态，并且栈和展望符号都维持不变——即它是阻滞不前的。

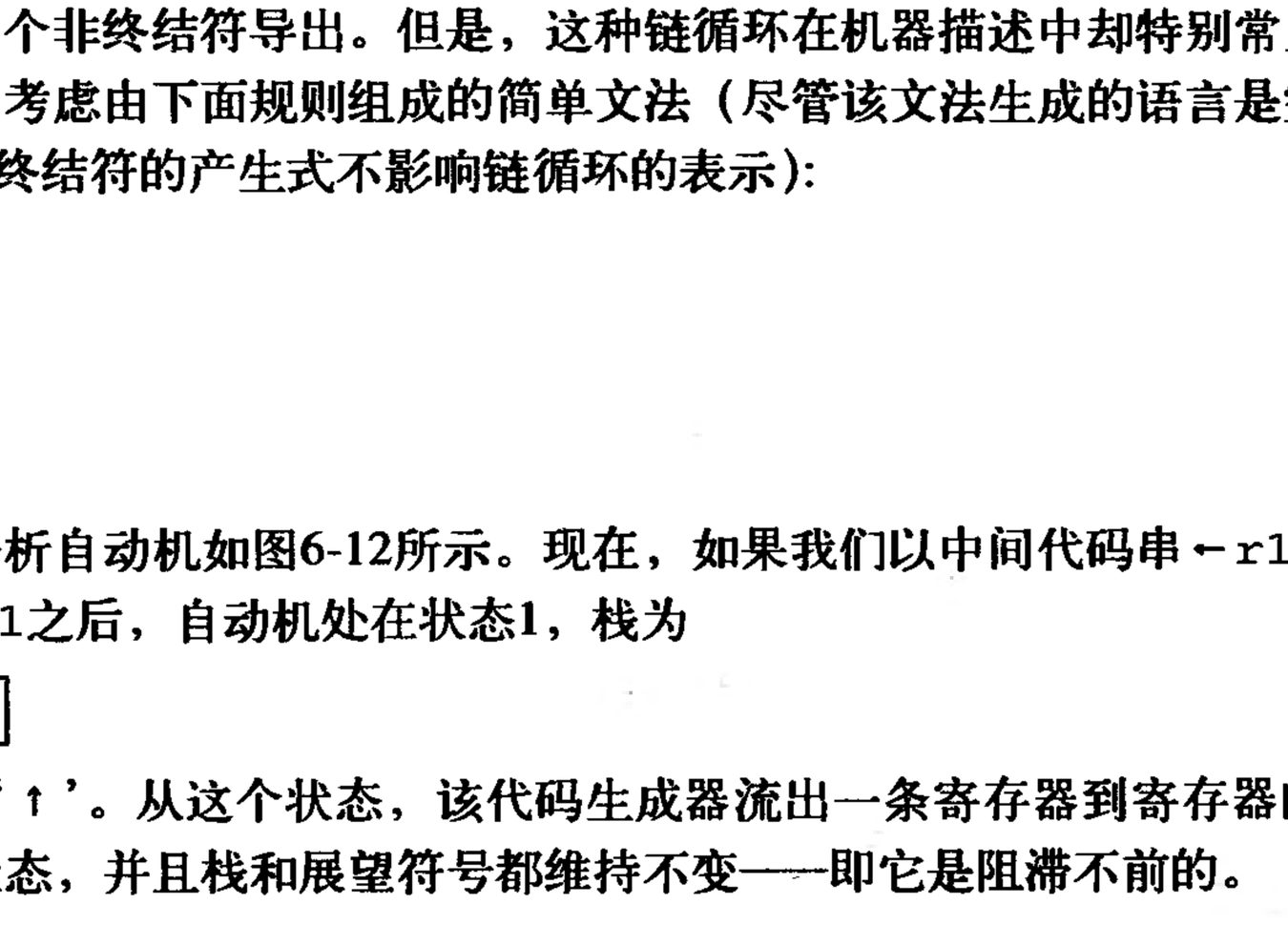


图6-12 含链循环的文法例子的分析自动机

链循环的删除可作为施加于机器描述文法的预处理步骤，或作为构造Action/Next表的预处理步骤来进行。图6-13中的代码提供了一种进行这种文法预处理的方法。过程
152 Elim_Chain_Loops()在文法MGrammar中寻找那些有单个非终结符同时在左边和右边出现的产生式<lt:*l*, rt:*r*>。对于每一个这样的产生式，它使用Close(<lt:*l*, rt:*r*>, *C*, *R*)

bbs.theithome.com

判断由r导出的非终结符集合是否构成一个循环。如果是，Close()返回循环中的非终结符集合于R中，返回构成这个循环的产生式集合于C中。Close()分别用图6-14定义的过程Reach()确定从它的参数可到达的非终结符集合，用Prune()剪除从它的参数出发不能到达初始非终结符的非终结符。Elim_Chain_Loops()然后只保留循环中的一个非终结符，并将循环中其他的非终结符从文法中删除，同时删除构成这个循环的规则，修改剩余的规则，以便用保留下来的那个非终结符替代被删除的非终结符。函数Replace(R, r, x)用符号x替换规则r中所有出现在非终结符集合R中的符号，并返回只包含结果规则的集合；当结果得到的规则具有相同的左端和右端时，它返回∅。

```
procedure Elim_Chain_Loops( )
begin
    r1, r2: Rule
    C, MG: set of Rule
    R: set of Nonterminal
    MG := MGrammar
    for each r1 ∈ MG do
        if r1.lt ≠ ϵ & |r1.rt| = 1 then
            if Close(r1,C,R) then
                for each r2 ∈ C do
                    MGrammar := (MGrammar - {r2})
                        ∪ Replace(R,r2,r1.lt)
                od
            fi
        fi
        MG -= {r1}
    od
end    || Elim_Chain_Loops

procedure Close(rule,C,R) returns boolean
    rule: in Rule
    C: out set of Rule
    R: out set of Nonterminal
begin
    r1: Rule
    || determine set of grammar rules making up a chain loop
    R := Reach(rule.rt)
    if rule.lt ∈ R then
        Prune(rule.lt,R)
    fi
    C := ∅
    for each r1 ∈ MGrammar do
        if r1.lt ∈ R & |r1.rt| = 1 & r1.rt ∈ R then
            C ∪= {r1}
        fi
    od
    return C ≠ ∅
end    || Close
```

图6-13 删除非终结符链循环的过程Elim_Chain_Loops()和它所使用的过程Close()

有的情形也可能希望保留链循环，如整数寄存器和浮点寄存器之间相互转送值的情况。这种情况用显式的一元操作符就很容易进行调节。

bbs.theithome.com

```
procedure Reach(r) returns set of Nonterminal
    r: in  Nonterminal
begin
    r1: Rule
    R := {r}, oldR: set of Nonterminal
    || determine the set of nonterminals reachable from r
    || without generating any terminal symbols
    repeat
        oldR := R
        for each r1 ∈ MGrammar do
          if r1.lt ∈ R & |r1.rt| = 1 & r1.rt↓1 ∈ Nonterminal & r1.rt↓1 ∉ R then
              R ∪= {r1.rt}
          fi
        od
    until R = oldR
    return R
end    || Reach

procedure Prune(l,R)
    l: in Nonterminal
    R: inout set of Nonterminal
begin
    r: Nonterminal
    || prune from R the set of nonterminals from which
    || the initial nonterminal is not reachable
    for each r ∈ R do
        if l ∉ Reach(r) then
            R -= {r}
        fi
    od
end    || Prune
```

图6-14 Close()所使用的例程Reach()和Prune()

6.2.4 删除句法阻滞

153 ~ 154

贪婪策略总是执行移进而不进行归约，常常也会导致代码生成器发生阻滞的情况，即进入不必要的Error状态。例如，考虑图6-15的机器描述，它给出的是Hewlett-Packard 的PA-RISC寻址模式的文法，其中c表示一个任意常数。此寻址模式使用一个基址寄存器加一个其值可以左移0、1、2或3位的索引寄存器（由图中最后四行表示）。对于这个文法段，阻滞发生在那些包含项目[s⇒ ↑ +r * · n r]（其中，n=2，4和8）的状态遇到任何非2、4或8的输入符号时。但是对于任意的c，形如“+ r * c r”的地址，或形如“+ r * r r”的地址都是合法的——它需要一串指令来计算它，而不仅仅是一条指令。具体地，在树形式中，对于第一种情况，我们有图6-16所示的推导，它对应于执行一个乘法操作，接着一个加法操作，再接着一个取数操作。为了在代码生成器中达到这个目的，我们需要在包含项目[s⇒ ↑ + r* · n r]的那些状态增加根据任意常数或寄存器而移进的转换，而不只是根据2、4或8进行转换。与上面的推导保持一致，我们增加一个根据r到新状态的转换，以及从那里根据s到另一个新状态的移进转换，后面这个新状态用[s ⇒ ↑ + r* r s ·]进行归约并生成对应的指令序列。

```
ϵ ➡ ← a r
s ➡ r
r ➡ ↑ a
r ➡ + r s
r ➡ * r s
r ➡ c
a ➡ r
a ➡ + r r
a ➡ + r * 2 r
a ➡ + r * 4 r
a ➡ + r * 8 r
```

图6-15 Hewlett-Packard的PA-RISC寻址模式机器描述文法段

bbs.theithome.com

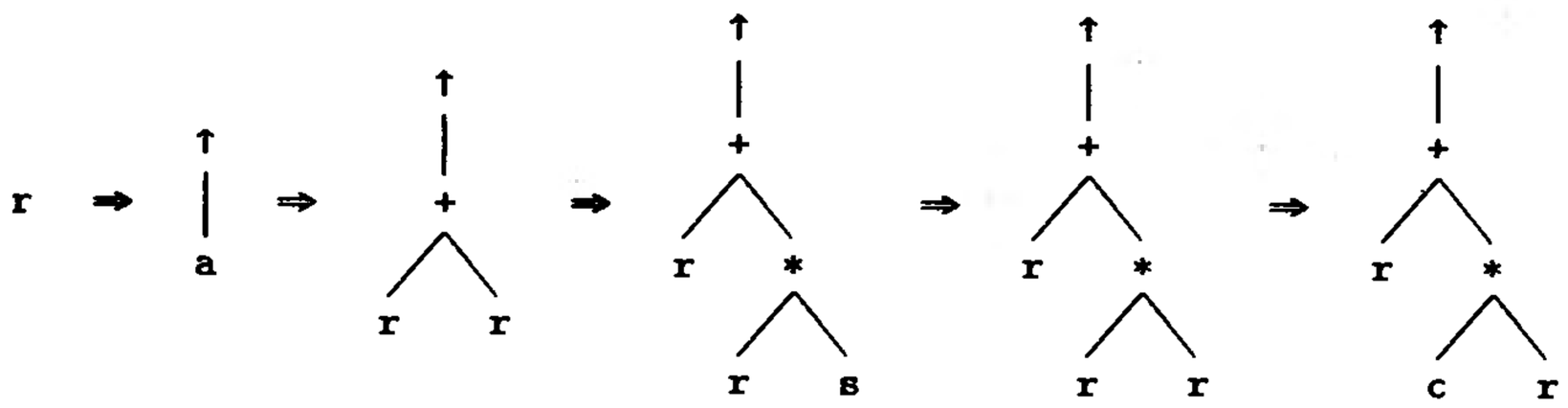


图6-16 ↑ + r * c r的树形式的推导序列

修复句法阻滞的过程Fix_Synt_Blocks()和附属例程如图6-17所示。Fix_Synt_Blocks()反向沿着产生式链尽可能长地归约导致阻滞的这个符号。这个算法可能产生需要进一步修复的项目，就像在这个特殊的例子中一样，在这个例子中，为了防止阻滞，最终符号也需要归约。注意，Derives()正确操作的关键是我们必须已经删除了链循环——否则，对于某些输入它会永远递归下去。 155

```
procedure Fix_Synt_Blocks( ) returns boolean
begin
   i, j: integer
   x, y: Nonterminal
   item, item1: Item
   NextItems, I: set of Item
   i := 1
   while i ≤ MaxStateNo do
   I := ItemSet[i]
      for each item ∈ I do
         || if there ie a derivation that blocks for some inputs and
         || not for others similar to it, attempt to generalize it by
         || adding statee and transitions to cover all similar inputs
         if ∃x ∈ Nonterminal
            (Derives([x],1,[item.rt↓(item.pos+1)])
            & Action(i,item.rt↓(item.pos+1)) = ⟨Error,∅⟩
            & Derives([item.lt],0,Subst(item.rt,item.pos+1,x))
            & ∀y ∈ Nonterminal (!Derives([y],1,[x]))
               ∨ !Derives([item.lt],0,
               Subst(item.rt,item.pos+1,[y]))) then
            item := Generalize(⟨lt:item.lt,
               rt:Subst(item.rt,item.pos+1,x),pos:item.pos⟩,
               item.pos+1,|item.rt|)
            if item = nil then
               return false
            fi
            ItemSet[i] ∪= {item}
            Action(i,x) := ⟨Shift,∅⟩
            NextItems := Cloeure(Advance({item1 ∈ ItemSet[i]
               where item1.lt = item.lt & item1.rt = Subst(item.rt,pos+1,x)
               & item1.pos = item.pos}))
            if ∃j ∈ integer (ItemSet[j] = NextIteme) then
               Next(i,x) := j
            || add new states and transitions where needed
            else
               StateNo := i
               MaxStateNo += 1
               ItemSet[MaxStateNo] := NextItems
               Next(StateNo,x) := MaxStateNo
```

图6-17 通过归约一个或多个非终结符来修复句法阻滞的例程

bbs.theithome.com

```
                while StateNo <= MaxStateNo do
                   Successors(StateNo)
                   if !Uniform(StateNo) then
                      return false
                   fi
                od
             fi
          else
             return false
          fi
          I -= {item}
       od
       i += 1
    od
    return true
end    || Fix_Synt_Blocks

procedure Generalize(item,lo,hi) returns Item
    item: in Item
    lo, hi: in integer
begin
    i: integer
    l, x, y: Nonterminal
    || attempt to find a generalization of the blocking item
    for i := lo to hi do
       if ∃x ∈ Nonterminal (Derives([x],1,[item.rt↓i])
             & Derives([item.lt],0,Subst(item.rt,i,x))
          & ∀y ∈ Nonterminal (!Derivee([y],1,[x])
             ∨ !Derives([item.lt],0,Subst(item.rt,i,y)))) then
          item.rt↓i := x
       fi
    od
    return item
end    || Generalize
          procedure Derives(x,i,s) returns boolean
             x, s: in VocabSeq
             i: in integer
          begin
             j: integer
             if i = 0 & x = s then
                return true
             fi
             for each rule ∈ MGrammar do
                for j := 1 to |x| do
                   if rule.lt = x↓j then
                      return Derives(rule.rt,0,s)
                   fi
                od
             od
             return false
          end    || Derives

procedure Subst(s,i,x) returns VocabSeq
    s: in VocabSeq
    i: in integer
    x: in Vocab
```

图6-17 （续）

bbs.theithome.com

```
begin
    t := []: VocabSeq
    j: integer
    for j := 1 to |s| do
        if j = i then
            t ⊕= [x]
        else
            t ⊕= [s↓j]
        fi
    od
    return t
end    || Subst
```

图6-17 （续）

作为Fix_Synt_Blocks()动作的一个例子，假设上面讨论的项目[s ⇒↑ + r * · 2r]存放在ItemSet[26]中，并且MaxStateNo是33。代码生成在状态26对于任何非2的展望符号都发生阻滞，因此，我们使用这个例程来发现是否有其他可能的非阻滞动作。该例程设置x为r并判别出存在一个推导s ⇒* ↑ + r*r r。因此，它调用

```
Generalize ([s⇒↑ + r *·r r ], 6, 6)
```

这个函数返回[s ⇒* ↑+r*·r s]。此项目变成item的值并且被加到ItemSet[26]中。现在Action (26, r) 被设成<Shift, ∅>, NextItems变成

```
{[s ⇒↑ + r * r·s ], [s ⇒·r]}
```

MaxStateNo增加到34，ItemSet[34]被设置成上面的集合，Next (26, r) 设置为34。最后，该例程使用Successors()和Uniform()产生新增加的状态，以及处理这些新项目和保证这些新状态的一致性所需要的转换。

6.2.5 最后的考虑

早期Graham-Glanville代码生成器的一个问题是，对于具有很多种指令类型和寻址方式的机器，其机器描述的Action/Next表非常大（对于VAX，大约产生有8 000 000条规则）。Henry[Henr84]开发的一种方法能有效地压缩该表，而且不管怎样，这个问题对于典型的RISC机器已不再那么严重，因为它们中的多数只提供少数几种寻址方式。

156 ~ 158

6.3 语义制导的分析介绍

本节我们概述第二种使用前缀波兰中间代码的代码生成方法，即由Ganapathi和Fischer开发的*属性文法（或词缀文法）*(attribute or affix-grammar)方法。这种方法通过使用属性来给代码生成规则增加语义，因而它的能力更强但也更复杂。这里只介绍这种方法的思想，具体的细节请读者查阅有关文献。

我们假定读者熟悉属性文法的基本概念。我们在其前放置一个向上的箭头“↑”表示继承属性，在其前放置一个向下的箭头“↓”表示综合属性，属性值写在箭头后面。除了在代码生成过程中上下传递值之外，属性被用来控制代码生成，以及计算新的属性值和生成副作用。控制通过表示谓词的大写斜体属性（例如，下例中的*IsShort*）来实现。一条规则在给定的情况下是可应用的，即当且仅当它在句法上匹配主语字符串并且满足它的所有谓词。动作用大写的印刷体书写（例如，例中的EMIT3），它计算新的属性值并生成副作用。当然，属性文法代码生成器中最重要的副作用是流出代码。

bbs.theithome.com

例如，图6-1中关于一个寄存器的内容与一个常数相加的Graham-Glanville规则

```
r.3⇒+ r.1 k.2     add  r.1, k.2, r.3
r.3⇒+ k.2 r.1     add  r.1, k.2, r.3
```

可以转变成如下检查常数是否在所允许的范围内的属性文法规则，并且在这些规则中含有代码流出和寄存器分配:

```
r↑r2⇒+ r↓r1 k↓k1 IsShort(k1)  ALLOC(r2)  EMIT3("add", r1, k1, r2)
r↑r2⇒+ k↓k1 r↓r1 IsShort(k1)  ALLOC(r2)  EMIT3("add", r1, k1, r2)
```

其中第一条规则应当这样读: 给定一个形如“+rk”的前缀波兰字符串且有寄存器为*r*1和常数为*k*1, 如果该常数满足谓词*IsShort*（*k*1），则分配一个寄存器*r*2存放结果，流出通过替换与寄存器和常数相关的值而获得的一条三操作数add指令，归约该字符串为r，然后向上传递值*r*2作为非终结符r的综合属性。

除了从低级中间语言生成代码之外，属性文法也能用于其他用途，如存储绑定（即，从中级中间形式生成代码），将各种窥孔优化集成到代码生成器，以及对机器描述规则集合进行因子分解使得它的规模相对减小。因为属性文法能做存储绑定，因此我们也可以使用层次稍高一点的中间代码，例如MIR的前缀波兰转换形式。事实上，因为谓词和函数可以随意编写，因此它们实质上可用来做编译后端可进行的任何工作——例如，我们可以积累所有要流出的代码作
159 为某个属性的值，直到整个输入字符串都已归约时，然后对它进行所需信息已经足够时的任何转换（并且我们也可以在代码生成期间积累这些信息）。

Ganapathi和Fischer的报告称，已经构建了三个基于属性文法的代码生成器的产生器（一个从UNIX的分析器生成器YACC构建，第二个从ECP构建，第三个是从头开始），并且已经在Fortran、Pascal、BASIC和Ada等编译器中利用它们为将近十多种体系结构生成了代码生成器。

6.4 树模式匹配和动态规划

本节我们介绍第三种自动产生代码生成器的方法，这种方法由Aho、Ganapathi和Tjiang[AhoG89]等人开发。他们的方法使用树模式匹配和动态规划，所产生的系统就是著名的*twig*。

动态规划（dynamic programming）是一种在计算过程中根据适用于当前所考虑范围的*最优性原理*（optimality principle）进行决策的方法。这个最优性原理断定，如果所有子问题都以最优方式解决，则整个问题能够通过用特定的方法组合这些子问题的解而以最优的方式解决。使用动态规划的方法与Graham-Glanville代码生成器所采用的贪婪方法形成鲜明的对比——与仅尝试一种方法来进行树的匹配不同，我们可以尝试多种方法，但只能用那些对于已匹配部分而言是最优的方法，因为对于给定的一种可应用的最优原理，只有这些方法的组合才可能生成对于总体是最优的代码序列。

当*twig*用某个模式匹配一棵子树时，它一般用另一棵树来替代该子树。在匹配过程中因为成功归约一棵树到单个结点而被重写的子树序列称为树的*覆盖*（cover）。*最小代价覆盖*（minimal-cost cover）是使得产生这个覆盖的匹配操作的代价之和（细节如下所示）尽可能小的那个覆盖。代码流出和寄存器分配作为匹配过程中的副作用而执行。

*twig*的输入由如下形式的树重写规则组成:

label: *pattern* [{ *cost* }] [= { *action* }]

其中，*label*是一个标识符，它对应于文法规则左端的非终结符；*pattern*是括在括号内的树模式的前缀表示；*cost*是在子树与模式相匹配时由代码生成器执行的C代码，它返回由动态规划算

bbs.theithome.com

法使用的一个代价，而*pattern*则确定该模式是否满足关于匹配该子树的语义标准；*action*是一段C代码，如果模式匹配成功，并且动态规划算法判别出该模式是整个树的最小代价覆盖部分，则执行这段代码。*action*部分可以包括用另外的树替代已匹配的子树、流出代码或其他动作。

如围绕它们的方括号所示，*cost*和*action*部分都是可选的。如果缺少*cost*，它返回由别处指明的默认代价值，并且假定模式是匹配的。如果缺少*action*，默认动作是不做任何动作。 160

图6-18中的树重写系统为模式匹配处理过程的一个例子，它包含了图6-1中某些规则的树形式。谓词*IsShort*（）判别它的参数是否能放入SPARC指令的13位常数域中。对应的*twig*规范如图6-19所示。prologue实现函数*IsShort*（）。label声明列出了所有可以作为标号出现的标识符，node声明列出了所有能够在模式中出现的标识符，但已在label中列出的标识符[⊖]除外。字符串$$是一个指针，它指向模式所匹配的那棵树的树根，形如$*i*$的字符串指向这个根的第*i*个儿子。ABORT导致模式匹配流产，其效果是返回无穷大的代价。NODEPTR是结点的类型，getreg()是寄存器分配器。各种emit_...()例程流出特定类型的指令。

规则号	重写规则	代价	指令
1	reg.*i* ⇒ con.*c*	*IsShort*(*c*);1	or *c*,r0,r*i*
2	reg.*i* ⇒ ↑ reg.*j* reg.*k*	1	ld [r*j*,r*k*],r*i*
3	reg.*i* ⇒ ↑ reg.*j* con.*c*	*IsShort*(*c*);1	ld [r*j*,*c*],r*i*
4	ϵ ⇒ ← reg.*i* reg.*j* reg.*k*	1	st r*i*,[r*j*,r*k*]
5	ϵ ⇒ ← reg.*i* reg.*j* con.*c*	*IsShort*(*c*);1	st r*i*,[r*j*,*c*]
6	reg.*k* ⇒ + reg.*i* reg.*j*	1	add r*i*,r*j*,r*k*

图6-18 一个简单的树重写系统

现在，假设我们正在为如下带括号的前缀表达式生成代码：

```
st(add(ld(r8,8),add(r2,ld(r8,4))))
```

我们故意将它设计得与图6-3中的第二棵树类似，但只使用图6-19中定义的操作。模式匹配器将以下降方式查找这个表达式的树结构，直到发现模式1匹配子表达式“8”和模式3匹配“ld(r8,8)”。使用这些匹配中的第一个将产生一棵与模式2匹配的子树，但它的代价是2，而单独使用模式3时所产生的代价是1，因此应当使用模式3。子表达式“ld(r8,4)”也类似地进行匹配。但是，这两个隐含的归约都不应当立即进行，因为可能还存在可选择的代价较小的其他匹配；相反，应当将每一个匹配的指示器存储在所匹配子表达式的根结点中。一旦这个匹配过程完成，就可以执行归约和相应的动作。 161

⊖ *twig*使用字母标识符而不是诸如“↑”和“+”之类的符号，因为它不是为处理后者而设计的。

bbs.theithome.com

```
prologue { int IsShort(NODEPTR p);
    { return value(p) >= -4096 && value(p) <= 4095; }  }
node  con ld st add;
label reg no_value;

reg : con
      { if (IsShort($$)) cost = 1;
        else ABORT; }
      = { NODEPTR regnode = getreg( );
          emit_3("or",$$,"r0",regnode);
          return regnode; }

reg : ld(reg,reg,reg)
      { cost = 1; }
      = { NODEPTR regnode = getreg( );
          emit_ld($2$,$3$,regnode);
          return regnode; }

reg : ld(reg,reg,con)
      { cost = 1; }
      = { NODEPTR regnode = getreg( );
          emit_ld($2$,$3$,regnode);
          return regnode; }

no_value : st(reg,reg,reg)
      { cost = 1; }
      = { emit_st($1$,$2$,$3$);
          return NULL; }

no_value : st(reg,con,reg)
      { cost = 1; }
      = { emit_st($1$,$2$,$3$);
          return NULL; }

reg : add(reg,reg,reg)
      { cost = 1; }
      = { NODEPTR regnode = getreg( );
          emit_3("add",$1$,$2$,regnode);
          return regnode; }
```

162

图6-19 *twig*中与图6-18树重写系统对应的规范

*twig*的代码生成方法是如上所述的自顶向下的树模式匹配和动态规划的结合。基本的思想是，树可以用从其根结点到叶子结点的带标号的路径集合来刻画，其中，标号以结点为单位按后裔的顺序依次编号。路径字符串（path string）中，表示标号的整数和结点标识符交替地出现。例如，图6-3中第三棵树可以惟一地用如下一些路径字符串表示:

```
← 1 + 1 r8
← 1 + 2 4
← 2 - 1 r1
← 2 - 2 1
```

并且为这个树模式也能构造出类似的字符串集合。树模式集合的路径字符串依次又可用来构造模式匹配自动机，它是一种广义的有限自动机。该自动机并行地匹配各种树模式，并且每一个接收状态指明它对应的树模式通过的是哪一条路径。子树与一个树模式相匹配，当且仅当对于与该树模式对应的每一条路径字符串，自动机存在一条到接收状态的通路。

bbs.theithome.com

作为这种自动机的例子，我们为图6-18中的规则构造一个自动机。它们对应的路径字符串和规则如图6-20所示，所构成的自动机如图6-21所示。自动机的初始状态是0，用双圆圈表示的状态是接收状态，每一个非接收状态有一个另外未画出的转换，即，关于“其他符号”移进到错误状态（标识为“error”）的转换。在接收状态附近的标号给出某个路径字符串用来产生该状态的规则编号。例如，规则5中的模式被匹配，当且仅当并行运行该自动机导致停止在状态9、11和14。在本节给出的文献中可找到构造这个自动机的细节以及如何用代码来实现它的方法。

路径字符串	规则
c	1
↑ 1 r	2, 3
↑ 2 r	2
↑ 2 c	3
← 1 r	4, 5
← 2 r	4, 5
← 3 r	4
← 3 c	5
+ 1 r	6
+ 2 r	6

图6-20 图6-18中规则的树匹配路径字符串

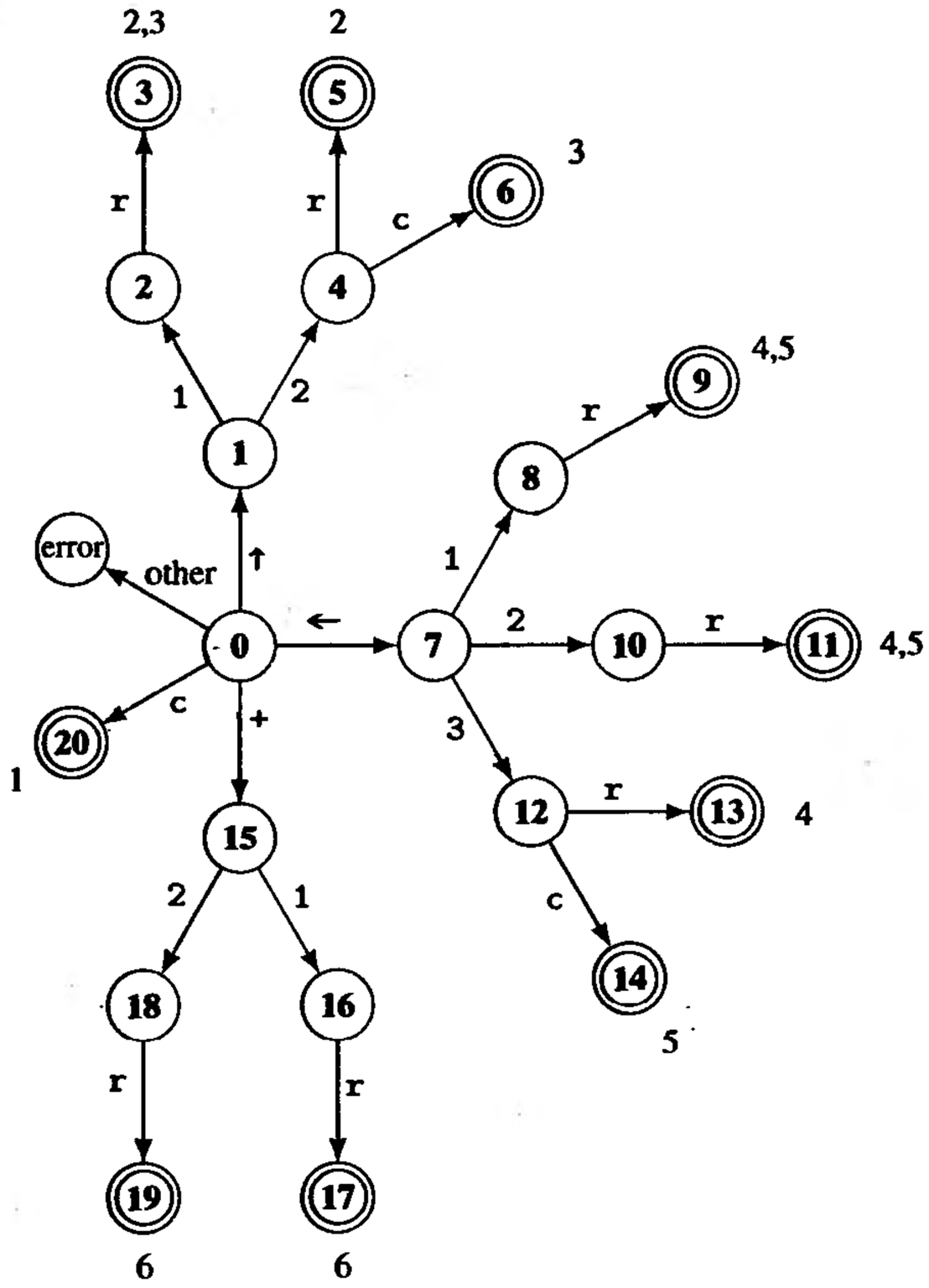


图6-21 图6-18中规则的树匹配自动机

这种动态规划算法假定给它的是一个统一寄存器的机器，该机器具有n个可相互交换的寄存器ri和形如$ri \leftarrow E$的指令，其中E是由运算符、寄存器和存储位置组成的表达式。与每一个指令序列相关的代价是每一条指令的代价之和。该算法将表达式E的代码生成问题划分成子问题，每一个问题对应E的一个子表达式，并以递归方式来解这些子问题。使动态规划可应用的关键是要连接地计算每一个子表达式，即，首先计算那些其值必须存放到存储单元的子表达式，然后按左子树、右子树、树根的顺序，或者按右子树、左子树、树根的顺序来计算表达式。这样，一旦子树中必须存放到存储器的表达式已计算出来，就不会在计算一个运算符的各个子树之间

bbs.theithome.com

版权所有 侵权必究 请勿擅自制作请事宜作者版权

踌躇。于是，对于计算一个给定表达式的任何指令序列，就统一寄存器的机器而言，存在着代价不大于它且所用寄存器个数最少地计算同一表达式的指令序列，而且这个指令序列是连接的。

注意，大部分实际的机器没有统一的寄存器，尤其是，它们还有若干需使用奇偶寄存器偶对的操作，而且，人们可以给出这种表达式的例子，得出它的最优求值序列需要在它的子表达式之间摇摆多次。但是，表达式的大小与所需要的摆动次数至少呈线性增长的这种例子是很少见的，并且实际中的表达式一般不是特别大，因此，连接性只是略微有点违反，因而在实际中该算法几乎总是生成接近最优的指令序列。

163 ~ 164

Aho和Johnson的算法[AhoJ76]有三个步骤：（1）对表达式树的每一个结点N，自下而上地计算代价表中的表项$c[N, i]$，其中表项$c[N, i]$的值是在至多使用i个寄存器的情况下计算以N为根的树所需要的代价；（2）使用代价表来确定哪一棵子树的值必须存放到存储器中；（3）递归地确定计算子树的最优指令序列。

*twig*的经验表明编写和修改*twig*的规范相当容易，它产生的代码生成器在质量和性能两方面都可与专门为易于移植而设计的代码生成器相比，例如可移植C编译器pcc2的代码生成器。

自下而上的树匹配也能用于从机器描述自动产生能生成最优代码的代码生成器。Pelegri-Llopart开发了一种这样的方法，这种方法基于自下而上重写系统，即BURS（bottom-up rewriting systems）。

6.5 小结

本章我们简要地讨论了从中间代码生成机器代码的有关问题，然后探讨了从机器描述产生代码生成器的自动方法。

需要考虑的基本问题包括：目标机体系结构、必须遵循的软件约定、中间语言的结构和特征、中间语言中那种没有目标机指令与之直接对应的操作的实现方法、生成汇编语言代码还是直接生成机器代码的可链接或可重定位版本，以及将中间代码转换到所选择的目标代码形式的方法。

我们编写的编译器是为支持单一语言单一目标机体系结构，还是为支持多种语言多种目标机体系结构，或二者都支持，决定了选择这些问题时的重要性。尤其是，涉及的目标体系结构越多，使用自动方法就越重要。

尽管手工书写的代码生成器效率高且实现起来较快，但它们明显的缺点是用手工实现，因此比自动生成的代码生成器要难于修改和移植。

已经开发出来了若干种从机器描述构造代码生成器的方法，我们以不同的详略程度介绍了其中三种。这三种方法都从已展开了地址计算的低级中间代码开始，并且三者都对树进行模式匹配，一种是显式，另外两种是隐式，即对前缀波兰中间代码，这种代码是树序列的前序遍历表示。

165

6.6 进一步阅读

[Catt79]报告了第一个具有重要意义的自动产生代码生成器的成功项目。

Graham-Glanville代码生成方法的第一次描述是在[GlaG78]中，在[AigG84]和[Henr84]中得到了进一步的发展。[GraH82]、[Bird82]、[LanJ82]和[ChrH84]等论文中介绍了Graham-Glanville代码生成器的产生器的其他实现。

属性文法代码生成方法是由Ganapathi和Fischer开发的，在[GanF82]、[GanF84]、[GanF85]以及其他一些论文中有这种方法的描述。ECP错误修复分析器是各种Ganapathi和Fischer实现的基础，其描述见[MauF81]。

[Enge75]对树自动机及其用途作了极好的介绍。[AhoG89]中描述了由Aho、Ganapathi和

bbs.theithome.com

Tjiang开发的代码生成树模式匹配方法。这种树模式匹配方法是Aho和Johnson[AhoJ76]开发的线性时间字符串匹配算法的推广，并吸收了Hoffoman和O'Donnell[HofO82]将它扩充到树的某些思想。动态规划算法以Aho和Johnson [AhoJ76]开发的方法为基础。Aho、Ganapathi和Tjiang用来与*twig*进行比较的pcc2可移植C编译器的介绍见[John78]。Pelegri-Llopart自底向上树匹配生成局部最优代码的方法的介绍见[Pele88]和[PelG88]。[AhaL93]介绍了从高级中间代码开始生成代码的基于树自动机的方法。

Henry和Damron（[HenD89a]和[HenD89b]）给出了十多种从机器描述自动生成代码生成器方法的极好的综述，对其中8种方法进行了详细的讨论和评价，并从它们所产生的代码生成器的速度和所生成的代码质量两个方面对这些方法进行了性能比较。

6.7 练习

6.1 编写Emit_Instrs()（参见6.2.1节）的ICAN版本，它要能充分适应图6-8中的文法，包括区别短常数。

6.2 (a) 构造出关于如下机器描述规则和指令的Graham-Glanville分析表，其中fr.*n*表示浮点寄存器。

```
fr.2 ⇒ fr.1                fmov   fr.1,fr.2
fr.2 ⇒ mov fr.2 fr.1       fmov   fr.1,fr.2

r.3 ⇒ + r.1 r.2            add    r.1,r.2,r.3
r.3 ⇒ - r.1 r.2            sub    r.1,r.2,r.3

fr.3 ⇒ +f fr.1 fr.2        fadd   fr.1,fr.2,fr.3
fr.3 ⇒ -f fr.1 fr.2        fsub   fr.1,fr.2,fr.3
fr.3 ⇒ *f fr.1 fr.2        fmuls  fr.1,fr.2,fr.3
fr.3 ⇒ /f fr.1 fr.2        fdivs  fr.1,fr.2,fr.3
fr.2 ⇒ sqrt fr.1           fsqrts fr.1,fr.2

fr.2 ⇒ cvti fr.1 fr.2      fstoi  fr.1,fr.2
fr.2 ⇒ cvtf fr.1 fr.2      fitos  fr.1,fr.2

fr.3 ⇒ ↑ + r.1 r.2         ldf    [r.1,r.2],fr.3
fr.3 ⇒ ↑ + r.1 k.2         ldf    [r.1,k.2],fr.3
fr.2 ⇒ ↑ r.1               ldf    [r.1],fr.2

ϵ ⇒ ← + r.2 r.3 fr.1       stf    fr.1,[r.2,r.3]
ϵ ⇒ ← + r.2 k.1 fr.1       stf    fr.1,[r.2,k.3]
```

(b) 通过为如下前缀波兰序列生成代码来检查你的分析自动机。

```
← + r1 4 cvti -f fr2
mov fr2 *f fr3 ↑ + r1 8
← + r1 12 sqrt ↑ + r7 0
```

6.3 构造上一练习中的文法有关的关系（*Left*、*Right*等）和函数（*Parent*（）、*Follow*（）等）。

6.4 给出一个比6.2.3节开始的那个链循环更复杂的链循环例子，并使用图6-13的算法来删除它。

6.5 编写一个可在Graham-Glanville分析表构造期间或构造之后使用的链循环删除器。

6.6 给出一个在计算机科学中使用动态规划的例子，6.4节的例子除外。

RSCH 6.7 阅读Pelegri-Llopart和Graham的论文[PelG88]，并用ICAN编写一个基于BURS的代码生成器的产生器。

166 ~ 167

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第7章　控制流分析

为实现编译优化，我们必须使编译器对程序如何使用系统中的可用资源具有全局性的“理解”能力[⊖]。编译器必须能刻画程序的控制流和它们对数据执行的操作相关的特征，以便删除由未优化的编译器产生的任何无用的代码，从而使低效的、较通用的机制被更高效的专用机制所替代。

当编译器读入程序时，它一开始是将程序看成简单的字符序列。词法分析器将这些字符序列转换为单词，语法分析器从中进一步发现语法结构。由编译前端产生的结果可以是语法树或某种低级形式的中间代码。但是，不论这种结果是什么形式，它对程序做什么和怎样做仍然没有多少提示。编译器将发现每一个过程内控制流层次结构的任务留给了控制流分析，将确定与数据处理有关的全局信息的任务留给了数据流分析。

在考虑控制流分析和数据流分析中使用的形式化技术之前，我们先给出一个简单的例子。我们从图7-1的C例程开始，这个例程对给定的$m \geqslant 0$，计算第m个斐波那契数。对于给定的输入值m，它检查该数是否小于或等于1，当小于或等于1时返回该参数值；否则，它迭代计算过程直到得出数列的第m个成员，并返回此数。在图7-2中，我们给出了这个C例程被转换为MIR后的代码。

```
unsigned int fib(m)
   unsigned int m;
{  unsigned int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) {
      return m;
   }
   else {
      for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

图7-1　计算斐波那契数的C例程

```
 1        receive m (val)
 2        f0 ← 0
 3        f1 ← 1
 4        if m <= 1 goto L3
 5        i ← 2
 6   L1:  if i <= m goto L2
 7        return f2
 8   L2:  f2 ← f0 + f1
 9        f0 ← f1
10        f1 ← f2
11        i ← i + 1
12        goto L1
13   L3:  return m
```

图7-2　图7-1中C例程的MIR中间代码

在分析这个程序时，我们的第一个任务是发现它的控制流。就这个程序的源代码而言，可以断言其控制结构是显然的——函数体由一个在else部分含一个循环的if-then-else结构组成；但是，在中间代码中，控制结构就不那么明显了。此外，循环也可能是用if和goto形成的，因此，源代码中的控制结构也可能不那么明显。因此，控制流分析的形式方法肯定不是无用的。为了使控制流分析方法能应用于一目了然的程序，我们首先转换程序为一种形象的表示，即如图7-3所示的流程图（flowchart）。

⊖ 将“理解”加上引号是因为我们觉得防止将优化过程拟人化是重要的，通常与它对应的词是“计算”。

bbs.theithome.com

超星公司制作 请尊重作者版权

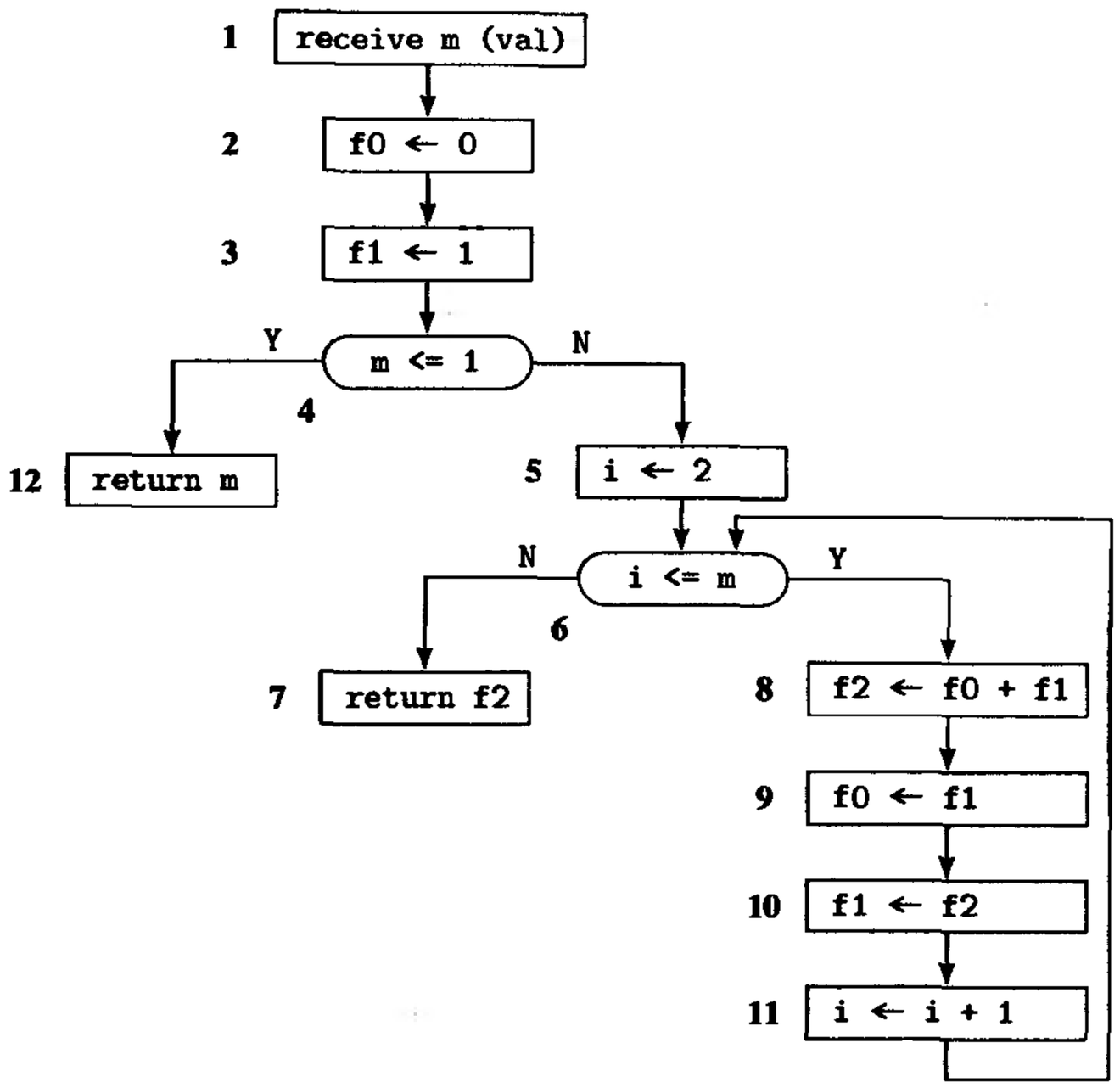


图7-3 与图7-2对应的流程图

接下来标识基本块。非正式地说，基本块是一段只能从它的开始处进入，并从它的结束处离开的线性代码序列。显然，结点1到结点4构成了一个基本块，我们称该基本块为B1；结点8到结点11构成了另一个基本块，称该基本块为B6；其他每一个结点独自构成一个基本块，其中结点12对应于B2，结点5对应于B3，结点6对应于B4，结点7对应于B5。下面我们将构成一个基本块的所有结点蜕化成一个结点，这个结点代表整个MIR指令序列，由此产生了该例程的流图（flowgraph），如图7-4所示。由于技术原因（当我们讨论向后数据流分析问题时便会明白），我们增加了一个以第一个基本块作为其后继的entry基本块，和一个放在流图结尾的exit基本块，并使流图中每一个实际从该例程出口的分支（基本块B2和B5）转向exit基本块。

下面我们利用所谓的必经结点（dominator）来标识该例程中的循环。本质上，如果从entry到B的每一条路径都包含A，则流图中结点A是结点B的必经结点。容易证明流图中结点上的必经结点关系是反对称的、自反的和传递的，其结果是这种必经结点关系可以表示成以entry结点作为根的一棵树。对于图7-4中的流图，其必经结点树如图7-5所示。

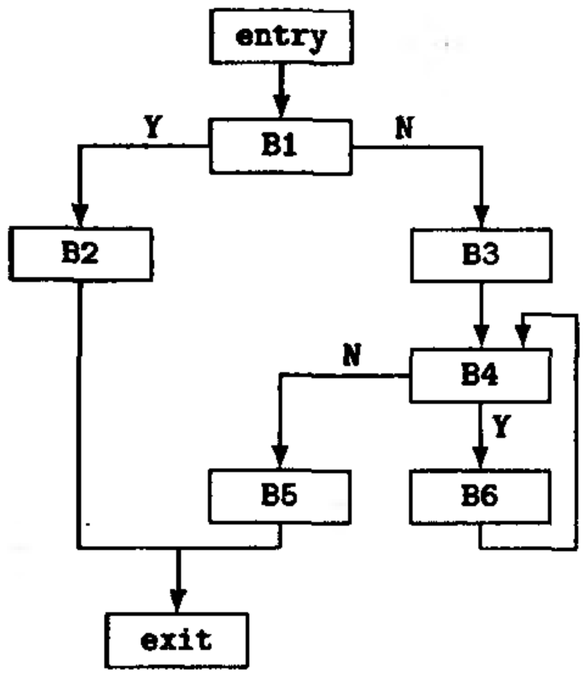


图7-4 图7-3对应的流图

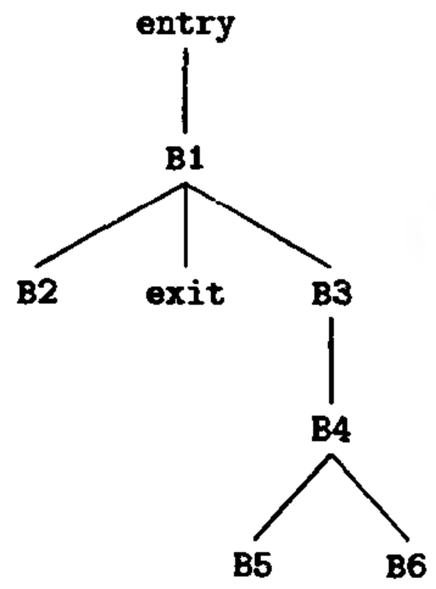


图7-5 图7-4中流图的必经结点树

bbs.theithome.com

现在可以用必经结点树来标识循环了。流图中的回边（back edge）是这样一条边，这条边的头结点是其尾结点的必经结点，例如从B6到B4的边。一个循环由满足下述条件的所有结点组成：循环的入口结点（即回边的头结点）是所有这些结点的必经结点，所有这些结点都可以到达入口结点，而且其内只有一条回边。因此B4和B6形成了一个循环，其中B4是循环的入口结点，且此循环不包含图中其他的结点。

第8章将继续用这个例子作为讨论数据流分析的初始例子。现在我们着手对此例中遇到的一些概念给出更正式的说明，讲述这些概念的若干细节，以及它们的不同表示。

7.1 控制流分析的方法

对单一过程进行控制流分析有两种主要的途径，它们都从确定构成过程的基本块开始，然后构造出它的流图。第一种途径使用必经结点来找出循环，并为优化简单地标识出它所找到的循环。这种方法足以满足那种通过迭代进行数据流分析的优化器的需要，如我们这个例子在8.1节所做的优化，或满足那种其注意力严格限制于例程内的循环的优化的需要。

170 ~ 172

第二种途径称为区间分析，它包括一系列的方法，这些方法分析例程的整个结构并将它分解为称作区间的嵌套区域。区间的嵌套结构形成了所谓的控制树，这种控制树有助于使数据流分析结构化，并提高其分析速度。区间分析方法的一种最成熟的变体称为结构分析，它实质上是对例程中的每一种控制流结构进行分类。因为这种方法十分重要，故我们单辟一节专门讲述它。基于区间的数据流分析方法统称为*消去法*（elimination method），因为它们和用于线性代数问题的高斯消去法之间有着显著的相似性。

当前大多数优化编译器使用的是必经结点和迭代数据流分析，尽管这种做法的实现时间消耗最少，并且足以为实现后面将讨论的多数优化提供所需要的信息，但在如下三个方面，它不如基于区间分析的其他方法：

1. 基于区间的方法在执行实际的数据流分析时较快，尤其对结构分析和只使用较简单结构类型的程序更是如此。

2. 基于区间的方法（特别是结构分析）使得为回应程序的改变而更新已经计算出来的数据流信息更为容易（这种程序改变可能是由优化器所致，也可能由编译器用户所致），从而使得这些信息不必全部重新计算。

3. 结构分析使得实现第18章讨论的控制流变换特别容易。

因此，我们认为需要对这三种方法（译者注：指基于必经结点的迭代方法、区间分析、结构分析）都给予介绍，而让编译器的实现者根据实现效果、优化速度，以及所希望具有的能力等方面进行选择和组合。

因为所有这些方法都需要标识例程的基本块，并构造它的流图，下面我们便讨论与此有关的内容。形式上，基本块（basic block）是一个只能从它的第一条指令进入，并从最后一条指令离开的最长的指令序列。因此，基本块的第一条指令只可以是（1）例程的入口点，（2）分支的目标，或（3）直接紧跟在分支指令或返回指令之后的指令[⊖]。这种指令称为*首领*（leader）。为了确定组成一个例程的基本块，我们首先标识出所有的首领，然后对于每一个首领，依次将从该首领到下一个首领或例程结尾之间的所有指令包含在它的基本块中。

173

几乎在所有的情况下，上面的方法都足以确定一个过程的基本块结构。另一方面，要注意在确定一个例程中的首领时，我们并没有指明调用指令是否应当视为分支指令。在多数情况下，

⊖ 如果我们考虑的是RISC机器指令而不是中间代码指令，这个定义就需要稍作修改；如果体系结构有带延迟槽的分支指令，在分支延迟槽中的这条指令可能属于由其前面这条分支指令结束的那个基本块，也有可能它自己开始一个新的基本块，如果它是此分支指令的目标的话。具有两个延迟槽的分支指令（如MIPS－X）使得这个定义进一步复杂。我们的中间代码不包含这种复杂情况。

bbs.theithome.com

调用指令不必看成是分支，由此产生的基本块较长，个数也较少，这是优化所希望的。但是，如果过程调用有如Fortran中允许的那种交错返回，则必须将它们看成是基本块的边界。类似地，在C的某些特殊情况下，一个调用也必须看成是基本块的边界。最著名的例子是C的setjmp()函数，这个函数提供一种粗糙的异常处理机制。setjmp()调用带来的问题是，不仅是它自己返回到它的调用点之后，而且在后续使用异常处理机制时，即调用longjmp()时，也将控制从longjmp()被调用的地方返回到setjmp()的返回点。这要求将setjmp()的调用看成是基本块的边界。并且更糟糕的是，它导致在流图中引入了虚假边：一般地，一个调用了setjmp()的例程中的所有其他调用都需要插入一条从其返回点到setjmp()的返回点的控制流边，因为这些调用有可能由于调用了longjmp()而将控制返回到setjmp()的返回点。在实际中，一般的做法是不尝试对调用了setjmp()的例程进行优化，不过，放置一条控制流虚假边也是一种选择（通常也是非常悲观的选择）。

在Pascal中，可用goto语句退出一个过程，并将控制递交到静态包含该过程的那个过程内带标号的语句。这种goto会导致在那个包含过程的流图中产生类似的额外的边。不过，因为这种goto总是能够通过由里向外地处理嵌套过程而标识出来，因此，它们不会导致像C的setjmp()那样严重的问题。

有些优化也希望将调用看成具有与基本块边界类似的作用。具体来讲，指令调度（参见第17章）可能需要将调用视作基本块的边界，以便适当地填充延迟槽，但同时又希望从尽可能长的基本块得到好处。因此，在同一种优化中，可能既希望将调用看成是基本块的边界，同时又不希望它是基本块的边界。

标识了基本块之后，我们用含有结点集合和（控制流）边集合的有根的有向图（自此以后简单地称为图）来刻画过程的控制流。图中每一个结点代表一个基本块，外加两个不同的结点，称为entry和exit结点，以及一个边集合。边集合中的边连接一个基本块到其他基本块，其方式与原流程图的控制流边连接基本块最后一条指令到基本块的首领的方式相同。此外，对于例程中每一个初始基本块[⊖]，我们引入从entry到它的一条边；对于每一个结束基本块（即没有后继的基本块），引入从它到exit的一条边。entry和exit基本块不是实质性的，引入它们是出于技术上的原因——它们使得许多算法描述起来更简单（参见13.1.2节，例如，在为全局公共子表达式删除而进行的数据流分析中，如果不能保证没有进入entry基本块的边，我们就必须为entry块初始化不同于其他所有基本块的信息；在13.5节为代码提升而进行的数据流分析中，对于exit也有类似的区别）。由此得到的有向图是过程的流图（flowgraph）。流图的强连通子图称作区域（region）。

174

在本书后面的章节中，我们假定给定一个包含结点集合N和边集合$E \subseteq N \times N$的流图$G = <N, E>$，其中，$entry \in N$，$exit \in N$。我们通常将边写成$a \rightarrow b$，而不是$<a, b>$。

进一步，我们用一种明显的方式来定义一个基本块的后继（successor）基本块集合和前驱（predecessor）基本块集合。分支结点（branch node）是那种有多个后继的结点，汇合结点（join node）是有多个前驱的结点。我们用$Succ(b)$表示基本块$b \in N$的后继集合，用$Pred(b)$表示基本块$b \in N$的前驱集合。形式上，

$$Succ(b) = \{n \in N \mid \exists e \in E \quad \text{使得} \quad e = b \rightarrow n\}$$

$$Pred(b) = \{n \in N \mid \exists e \in E \quad \text{使得} \quad e = n \rightarrow b\}$$

扩展基本块（extended basic block）是从一个首领开始的最长指令序列，在这个指令序列中，除了第一个结点之外不含其他汇合结点（第一个结点本身不必一定是汇合结点，例如，它

⊖ 每个例程通常只有一个初始基本块。但是，某些语言结构，如Fortran 77的多入口点允许有多个初始基本块。

bbs.theithome.com

可以是过程入口结点)。因为扩展基本块只有一个入口且可能有多个出口，所以可以将它看成是以它的入口基本块为根的一棵树。我们在有些上下文中称以这种方式构成的扩展基本块为诸基本块（blocks)。像我们在后面章节将看到的，某些局部优化，如指令调度（17.1节)，在扩展基本块上比在一般基本块上更加有效。在图7-4所示的例子中，基本块B1、B2和B3构成了一个由多个基本块组成的扩展基本块。

图7-6给出了一个名为Build_Ebb(*r*，*Succ*，*Pred*)的ICAN算法，该算法对以*r*为根的扩展基本块，构造其内诸基本块的索引集合。图7-7中的算法Build_All_Ebbs(*r*，*Succ*，*Pred*)对以*r*为入口结点的流图，构造它的所有扩展基本块集合。它设置AllEbbs为其每一个偶对由它的根基本块的索引和扩展基本块内的诸基本块的索引集合组成的偶对集合。这两个算法都使用全局变量EbbRoots记录扩展基本块的根基本块。

```
EbbRoots: set of Node
AllEbbs: set of (Node × set of Node)

procedure Build_Ebb(r,Succ,Pred) returns set of Node
   r: in Node
   Succ, Pred: in Node ⟶ set of Node
begin
   Ebb := ∅: set of Node
   Add_Bbs(r,Ebb,Succ,Pred)
   return Ebb
end    || Build_Ebb

procedure Add_Bbs(r,Ebb,Succ,Pred)
    r: in Node
    Ebb: inout set of Node
    Succ, Pred: in Node ⟶ set of Node
begin
    x: Node
    Ebb ∪= {r}
    for each x ∈ Succ(r) do
        if |Pred(x)| = 1 & x ∉ Ebb then
            Add_Bbs(x,Ebb,Succ,Pred)
        elif x ∉ EbbRoots then
            EbbRoots ∪= {x}
        fi
    od
end    || Add_Bbs
```

图7-6 构造具有指定根结点的扩展基本块内的诸基本块集合的两个例程

```
entry: Node

procedure Build_All_Ebbs(r,Succ,Pred)
   r: in Node
   Succ, Pred: in Node ⟶ set of Node
begin
   x: Node
   s: Node × set of Node
   EbbRoots := {r}
   AllEbbs := ∅
    while EbbRoots ≠ ∅ do
       x := ♦ EbbRoots
```

图7-7 构造给定流图中的所有扩展基本块的例程

bbs.theithome.com

```
        EbbRoots -= {x}
        if Vs ∈ AllEbbs (s@1 ≠ x) then
           AllEbbs ∪= {⟨x,Build_Ebb(x,Succ,Pred)⟩}
        fi
    od
end     || Build_All_Ebbs

begin
    Build_All_Ebbs(entry,Succ,Pred)
end
```

图7-7 （续）

作为Build_Ebb()和Build_All_Ebbs()的例子，考虑图7-8中的流图。由该算法识别的扩展基本块是{entry}、{B1，B2，B3}、{B4，B6}、{B5，B7}和{exit}，如图中虚框所示。

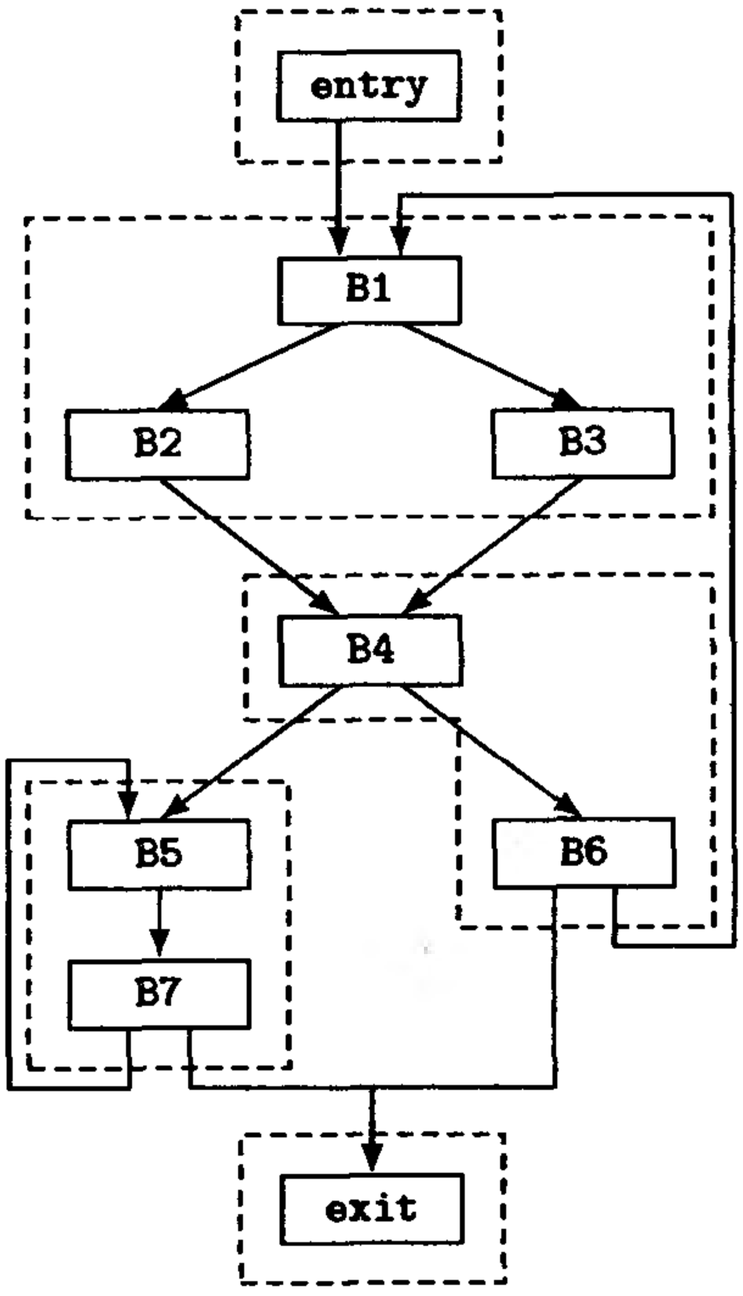


图7-8 用虚线框指出扩展基本块的流图

类似地，反扩展基本块（reverse extended basic block）是以分支结点结束的，且除最后一个结点之外不包含其他分支结点的最长指令序列。

7.2 深度为主查找、前序遍历、后序遍历和宽度为主查找

这一节涉及四个图论的概念，这些概念对我们后面使用的若干算法十分重要。这四个概念都适用于有根的有向图，因此也适用于流图。第一个概念是深度为主查找（depth-first search），这种查找在访问图中的结点时，首先访问的是该结点的后裔，而不是其兄弟，只要这个兄弟不同时又是其后裔。例如，图7-9a的深度为主查找表示是图b。用深度为主查找方法依次给每一个结点指定的编号是结点的深度为主编号（depth-first number）。

175
~
177

bbs.theithome.com

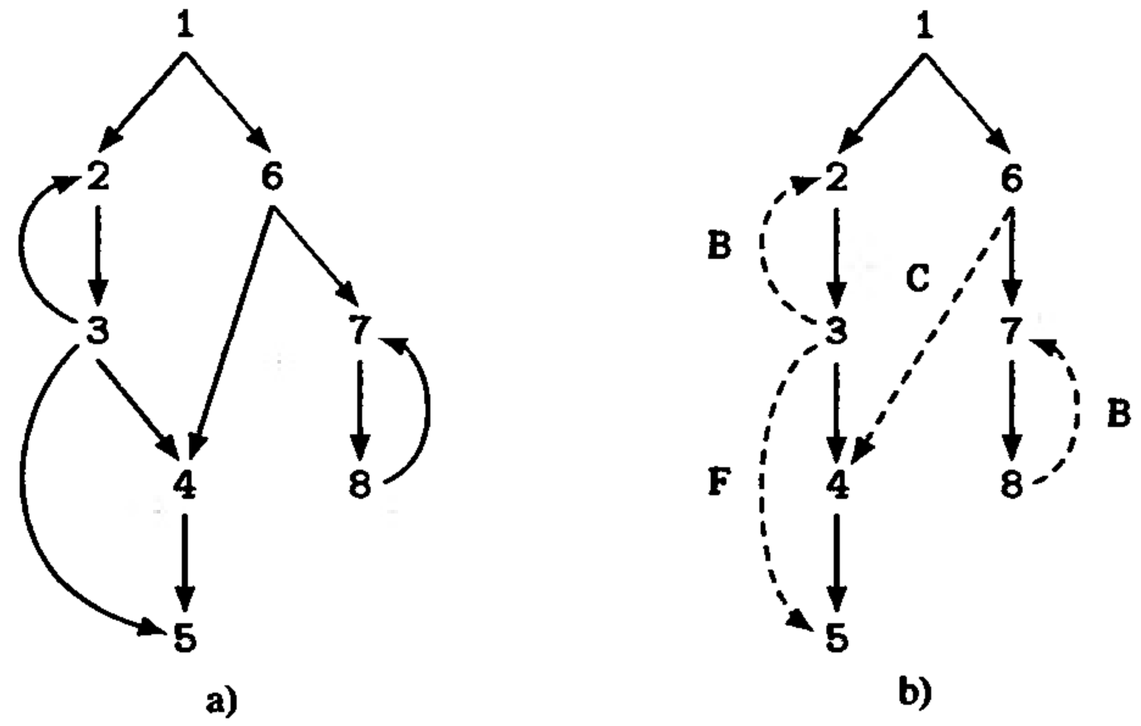


图7-9 a) 一个有根的有向图，b) 图a的深度为主表示

图7-10中的算法用于构造图的深度为主表示。深度为主表示（depth-first presentation）将图中所有的结点和那些构成深度为主次序的边表示为树的形式（称为深度为主生成树）（depth-first spanning tree），并将其他的边——这些边不是深度为主次序的一部分——用一种有别于树边的方式来表示（我们用虚线而不是实线表示它们）。属于深度为主生成树的边称为树边（tree edge），不属于深度为主生成树的那些边分为三类：前向边（forward edge）——从一个结点到一个直接后裔并且不是树边的边（在例子中我们用“F”标识它）；后向边（back edge）——从一个结点到树中它的一个祖先的边（用“B” 标识）；横向边（cross edge）——连接两个在树中相互不是祖先的结点的边（用“C”标识）。

```
N: set of Node
r, i: Node
Visit: Node ⟶ boolean

procedure Depth_First_Search(N,Succ,x)
   N: in set of Node
   Succ: in Node ⟶ set of Node
   x: in Node
begin
   y: Node
   Process_Before(x)
   Visit(x) := true
   for each y ∈ Succ(x) do
      if !Visit(y) then
         Process_Succ_Before(y)
         Depth_First_Search(N,Succ,y)
         Process_Succ_After(y)
      fi
   od
   Process_After(x)
end    || Depth_First_Search

begin
   for each i ∈ N do
      Visit(i) := false
   od
   Depth_First_Search(N,Succ,r)
end
```

图7-10 通用深度为主查找例程

bbs.theithome.com

注意，图的深度为主表示不是惟一的。例如，图7-11a有图7-11b和c所示的两个不同的深度为主表示。

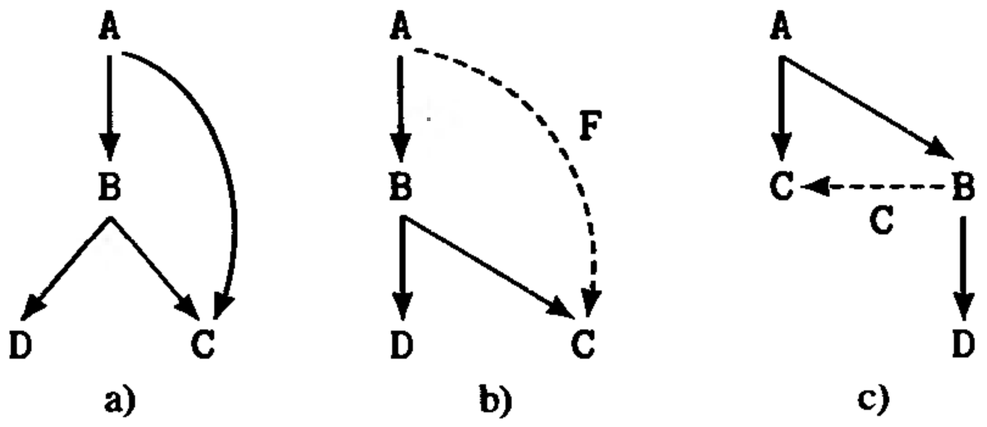


图7-11 a) 有根的有向图，b) 和c) 它的两个深度为主表示

图7-10的例程实现通用的流图深度为主查找，它提供四个动作执行点:

1. Process_Before() 允许我们在访问一个结点之前执行某种动作;

2. Process_After() 允许我们在访问一个结点之后执行某种动作;

3. Process_Succ_Before() 允许我们在访问一个结点的每一个后继之前执行某种动作;

4. Process_Succ_After() 允许我们在访问一个结点的每一个后继之后执行某种动作。

我们需要的第二个和第三个概念是有根的有向图中结点的两种遍历，以及它们所导致的图结点集合中结点之间的次序。令$G=<N, E, r>$是有根的有向图，令$E' \subseteq E$是G的深度为主表示中不包含反向边的边集合，则图G的*前序遍历*（preorder traversal）是这样一种遍历，其中每一个结点的处理早于其后裔，后裔关系由E'定义。例如，entry、B1、B2、B3、B4、B5、B6、exit是图7-4的一种前序遍历，序列entry、B1、B3、B2、B4、B6、B5、exit是图7-4的另一种前序遍历。

178 ~ 179

假设G和E'的定义如前所述，则图G的*后序遍历*（postorder traversal）是这样一种遍历，其中每一个结点的处理晚于其后裔，后裔关系由E'定义。例如，exit、B5、B6、B4、B3、B2、B1、entry是图7-4的一种后序遍历，而exit、B6、B5、B2、B4、B3、B1、entry是另一种后序遍历。

图7-12给出的例程Depth_first_Search_PP()是深度为主查找的一种特定的版本，它计算根$r \in N$的图$G=<N, E>$的深度为主生成树，以及它的前序遍历和后序遍历。在Depth_first_Search_PP()执行之后，由根结点开始，并沿着Etype(e)=tree的边e而行，便得到深度为主生成树。指定给每一个结点的前序编号和后序编号是整数，并分别存放在Pre()和Post()中。

```
N: set of Node
r, x: Node
i := 1, j := 1: integer
Pre, Post: Node --> integer
Visit: Node --> boolean
EType: (Node × Node) --> enum {tree,forward,back,cross}

procedure Depth_First_Search_PP(N,Succ,x)
   N: in set of Node
   Succ: in Node --> set of Node
   x: in Node
begin
   y: in Node
   Visit(x) := true
   Pre(x) := j
   j += 1
   for each y e Succ(x) do
      if !Visit(y) then
         Depth_First_Search_PP(N,Succ,y)
```

图7-12 计算深度为主生成树及其前序遍历和后序遍历

bbs.theithome.com

```
                EType(x → y) := tree
            elif Pre(x) < Pre(y) then
                Etype(x → y) := forward
            elif Post(y) = 0 then
                EType(x → y) := back
            else
                EType(x → y) := cross
            fi
        od
        Post(x) := i
        i += 1
    end     || Depth_First_Search_PP

    begin
        for each x ∈ N do
            Visit(x) := false
        od
        Depth_First_Search_PP(N,Succ,r)
    end
```

图7-12 （续）

第四个概念是宽度为主查找（breadth-first search），在这种查找顺序中，一个结点的所有直接后裔的处理早于这些后裔的任何还未处理的后裔（译者注：例如，如果一个结点有两个直接后裔a和b，其中结点b同时又是结点a的后裔，则a的处理先于b的处理）。在宽度为主查找中结点被访问的次序是宽度为主次序（breadth-first order）。对于图7-9的例子，次序1、2、6、3、4、5、7、8是宽度为主次序。

图7-13的ICAN代码当以Breadth_First(N, Succ, r)被调用时，它构造流图中结点的宽度为主次序。

```
i := 2: integer
procedure Breadth_First(N,Succ,s) returns Node → integer
    N: in set of Node
    Succ: in Node → set of Node
    s: in Node
begin
    t: Node
    T := ∅: set of Node
    Order: Node → integer
    Order(r) := 1
    for each t ∈ Succ(s) do
        if Order(t) = nil then
            Order(t) := i
            i += 1
            T ∪= {t}
        fi
    od
    for each t ∈ T do
        Breadth_First(N,Succ,t)
    od
    return Order
end     || Breadth_First
```

图7-13 计算宽度为主次序

bbs.theithome.com

7.3 必经结点和后必经结点

为了确定流图中的循环，我们首先定义流图中结点之间的一种称为必经结点的二元关系。如果从entry结点到i的每一条可能的执行路径都包含d，我们说结点d是结点i的*必经结点*(dominator)，记作$d\ dom\ i$。显然，*dom*是自反的（每一个结点是自己的必经结点）、传递的（如果$a\ dom\ b$且$b\ dom\ c$，则$a\ dom\ c$）和反对称的（如果$a\ dom\ b$且$b\ dom\ a$，则$b=a$）。我们进一步定义称为*直接必经关系*(immediate dominance)的子关系（idom）：对于$a\neq b$，$a\ idom\ b$当且仅当$a\ dom\ b$且不存在一个$c\neq a$并且$c\neq b$的结点c，使得$a\ dom\ c$且$c\ dom\ b$。我们记b的直接必经结点为$idom(b)$。显然，结点的直接必经结点是惟一的。直接必经结点关系形成了流图中结点的一棵树，此树的根是entry结点，它的边是直接必经关系，它的路径显示了所有必经结点关系。进一步，我们说d是i的*严格必经结点*(strictly dominator)，记为$d\ sdom\ i$，如果$d\ dom\ i$但$d\neq i$。

180 ~ 181

如果从结点i到结点exit的每一条可能的执行路径都包含结点p，我们也称结点p是结点i的*后必经结点*(postdominator)，记为$p\ pdom\ i$，即，在逆转所有的边，并且交换entry结点和exit结点而得到的流图中，$i\ dom\ p$。

我们给出两种计算流图中每一个结点的必经结点集合的方法。第一种方法的基本思想是，结点a是结点b的必经结点，当且仅当$a=b$；或者a是b的惟一直接前驱；或者b有多个直接前驱并且对于b的所有直接前驱c，有$c\neq a$且a是c的必经结点。对应的算法是图7-14给出的Dom_Comp()，它存储结点i的所有必经结点集合于Domin(i)。如果有星号标志的那个for循环按深度为主次序来处理流图的结点，则该算法具有最好的效率。

```
procedure Dom_Comp(N,Pred,r) returns Node ⟶ set of Node
   N: in set of Node
   Pred: in Node ⟶ set of Node
   r: in Node
begin
   D, T: set of Node
   n, p: Node
   change := true: boolean
   Domin: Node ⟶ set of Node
   Domin(r) := {r}
   for each n ∈ N - {r} do
      Domin(n) := N
   od
   repeat
      change := false
*     for each n ∈ N - {r} do
         T := N
         for each p ∈ Pred(n) do
            T ∩= Domin(p)
         od
         D := {n} ∪ T
         if D ≠ Domin(n) then
            change := true
            Domin(n) := D
         fi
      od
   until !change
   return Domin
end    || Dom_Comp
```

图7-14 一个计算流图中每一个结点的所有必经结点的简单算法

bbs.theithome.com

超星公司制作请尊重作者版权

作为使用Dom_Comp()的例子，我们将它应用到图7-4中的流图。该算法首先初始化change=ture，Domin(entry)={entry}，且对每一个非entry结点的结点i，domin(i)={entry，B1，B2，B3，B4，B5，B6，exit}。然后它进入repeat循环，设置change=false，并进入其中的for循环。这个for循环设置n=B1，T={entry，B1，B2，B3，B4，B5，B6，exit}，之后进入内层for循环。内层for循环设置p=entry(Pred(B1)的惟一成员)，因此设置T={entry}。之后，内层for循环终止，D被设置为{entry，B1}，change为true，以及Domin(B1)={entry，B1}。接下来，外层for循环设置n=B2，T={entry，B1，B2，B3，B4，B5，B6，exit}，并再次进入内层for循环。因为Pred(B2)={B1}，内层循环设置T为{entry，B1}。于是D被设置为{entry,B1，B2}，且Domin(B2)={entry，B1，B2}。继续这一过程得到下面的结果:

i	Domin(i)
entry	{entry}
B1	{entry,B1}
B2	{entry,B1,B2}
B3	{entry,B1,B3}
B4	{entry,B1,B3,B4}
B5	{entry,B1,B3,B4,B5}
B6	{entry,B1,B3,B4,B6}
exit	{entry,B1,exit}

如果需要每一个结点的直接必经结点，可以用图7-15给出的例程来计算它。同前面的算法一样，使标有星号的for循环按深度为主次序来处理流图的结点，则该例程可以达到最好的效率。这个算法也可以用位向量表示的集合来实现，并且具有可接受的效率：对于有n个结点和e条边的流图，其运行时间是$O(n^2 e)$。实质上，这个算法首先设置Tmp(i)为Domin(i)-{i}，然后对每一个结点检查Tmp(i)中的成员是否是除自己之外的其他结点的必经结点，如果是，则从Tmp(i)中去掉它们。作为使用Idom_Comp()的例子，我们将它应用到前面已计算出来的图7-4所示流图的必经结点集合。此算法首先初始化Tmp()数组如下:

i	Tmp(i)
entry	∅
B1	{entry}
B2	{entry,B1}
B3	{entry,B1}
B4	{entry,B1,B3}
B5	{entry,B1,B3,B4}
B6	{entry,B1,B3,B4}
exit	{entry,B1}

182
~
183

随后它设置n=B1，s=entry，并发现Tmp(B1)-{entry}=ϕ，因此Tmp(B1)保持不变。然后它设置n=B2。因为s=entry，Tmp (entry) 为空，所以，Tmp(B2)没有改变。另一方面，因为s=B1，Tmp(B1)是{entry}，于是entry被从Tmp(B2)中删除，留下Tmp(B2)={B1}，如此继续下去，得到Tmp()的最终值如下:

i	Tmp(i)
entry	∅
B1	{entry}
B2	{B1}
B3	{B1}
B4	{B3}

bbs.theithome.com

```
B5      {B4}
B6      {B4}
exit    {B1}
```

Idom_Comp在返回之前的最后一个动作是，对于$n \neq r$，设置集合Idom(n)为Tmp(n)的一个元素。

```
procedure Idom_Comp(N,Domin,r) returns Node ⟶ Node
   N: in set of Node
   Domin: in Node ⟶ set of Node
   r: in Node
begin
   n, s, t: Node
   Tmp: Node ⟶ set of Node
   Idom: Node ⟶ Node
   for each n ∈ N do
      Tmp(n) := Domin(n) - {n}
   od
*  for each n ∈ N - {r} do
      for each s ∈ Tmp(n) do
         for each t ∈ Tmp(n) - {s} do
            if t ∈ Tmp(s) then
               Tmp(n) -= {t}
            fi
         od
      od
   od
   for each n ∈ N - {r} do
      Idom(n) := ♦Tmp(n)
   od
   return Idom
end    || Idom_Comp
```

184

图7-15 计算直接必经结点，给出必经结点集合

第二种计算必经结点的方法是由Lengauer和Tarjan [LenT79]发明的。它比第一种方法要复杂，但除了最小流图之外，对所有流图它的运行速度都要快得多。

注意，对于有根的有向图<N, E, r>，如果$v=w$，或者在图的深度为主生成树中存在一条从v到w的路径则结点v是结点w的祖先（ancestor）。如果v是w的祖先且$v \neq w$，则v是w的固有祖先（proper ancestor）。另外，我们用*Dfn*（v）表示结点v的深度为主编号。

算法Domin_Fast()在图7-16中给出，图7-17和图7-18是两个辅助例程。已知一个流图和它的函数*Succ*()和*Pred*()，该算法最终存储每一个$v \neq r$的结点的直接必经结点于*Idom*（v）中。

```
Label, Parent, Ancestor, Child: Node ⟶ Node
Ndfs: integer ⟶ Node
Dfn, Sdno, Size: Node ⟶ integer
n: integer
Succ, Pred, Bucket: Node ⟶ set of Node

procedure Domin_Fast(N,r,Idom)
   N: in set of Node
   r: in Node
   Idom: out Node ⟶ Node
begin
```

图7-16 更复杂但也更有效的计算必经结点的方法

bbs.theithome.com

```
   u, v, w: Node
   i: integer
   || initialize data structures and perform depth-first search
   for each v ∈ N ∪ {n0} do
      Bucket(v) := ∅
      Sdno(v) := 0
   od
   Size(n0) := Sdno(n0) := 0
   Ancestor(n0) := Label(n0) := n0
   n := 0
   Depth_First_Search_Dom(r)
*1 for i := n by -1 to 2 do
      || compute initial values for semidominators and store
      || nodes with the same semidominator in the same bucket
      w := Ndfs(i)
      for each v ∈ Pred(w) do
         u := Eval(v)
         if Sdno(u) < Sdno(w) then
            Sdno(w) := Sdno(u)
         fi
      od
      Bucket(Ndfs(Sdno(w))) ∪= {w}
      Link(Parent(w),w)
      || compute immediate dominators for nodes in the bucket
      || of w's parent
*2    while Bucket(Parent(w)) ≠ ∅ do
         v := ♦Bucket(Parent(w)); Bucket(Parent(w)) -= {v}
         u := Eval(v)
         if Sdno(u) < Sdno(v) then
            Idom(v) := u
         else
            Idom(v) := Parent(w)
         fi
      od
   od

   || adjust immediate dominators of nodes whose current version of
   || the immediate dominator differs from the node with the depth-first
   || number of the node's semidominator
*3    for i := 2 to n do
      w := Ndfs(i)
      if Idom(w) ≠ Ndfs(Sdno(w)) then
         Idom(w) := Idom(Idom(w))
      fi
*4 od
end   || Domin_Fast
```

图7-16 （续）

```
procedure Depth_First_Search_Dom(v)
   v: in Node
begin
   w: Node
   || perform depth-first search and initialize data structures
   Sdno(v) := n += 1
   Ndfs(n) := Label(v) := v
   Ancestor(v) := Child(v) := n0
```

图7-17 计算必经结点时使用的深度为主查找和路径压缩算法

bbs.theithome.com

```
        Size(v) := 1
        for each w ∈ Succ(v) do
            if Sdno(w) = 0 then
                Parent(w) := v
                Depth_First_Search_Dom(w)
            fi
        od
    end     || Depth_First_Search_Dom

    procedure Compress(v)
        v: in Node
    begin
        || compress ancestor path to node v to the node whose
        || label has the maximal semidominator number
        if Ancestor(Ancestor(v)) ≠ n0 then
            Compress(Ancestor(v))
            if Sdno(Label(Ancestor(v))) < Sdno(Label(v)) then
                Label(v) := Label(Ancestor(v))
            fi
            Ancestor(v) := Ancestor(Ancestor(v))
        fi
    end     || Compress
```

图7-17 （续）

```
procedure Eval(v) returns Node
    v: in Node
begin
    || determine the ancestor of v whose semidominator
    || has the minimal depth-first number
    if Ancestor(v) = n0 then
        return Label(v)
    else
        Compress(v)
        if Sdno(Label(Ancestor(v))) ≥ Sdno(Label(v)) then
            return Label(v)
        else
            return Label(Ancestor(v))
        fi
    fi
end     || Eval

procedure Link(v,w)
    v, w: in Node
begin
    s := w, tmp: Node
    || rebalance the forest of trees maintained
    || by the Child and Ancestor data structures
    while Sdno(Label(w)) < Sdno(Label(Child(s))) do
        if Size(s) + Size(Child(Child(s)))
            ≥ 2*Size(Child(s)) then
            Ancestor(Child(s)) := s
            Child(s) := Child(Child(s))
        else
            Size(Child(s)) := Size(s)
            s := Ancestor(s) := Child(s)
        fi
```

图7-18 计算必经结点时使用的标号计算和链接算法

bbs.theithome.com

```
            od
            Label(s) := Label(w)
            Size(v) += Size(w)
            if Size(v) < 2*Size(w) then
               tmp := s
               s := Child(v)
               Child(v) := tmp
            fi
            while s ≠ n0 do
               Ancestor(s) := v
               s := Child(s)
            od
         end     || Link
```

图7-18 （续）

这个算法首先对一些数据结构进行初始化（后面将讨论这些数据结构），然后对图执行深度为主查找，并给经过的每一个结点一个编号，这个编号即深度为主次序。它使用了一个结点n0 ∉ N。

接下来它对每一个结点$w \neq r$，计算w的所谓半必经结点，并设置$Sdno(w)$为此半必经结点的深度为主编号。$w \neq r$的结点w的半必经结点（semidominator）是满足这种条件的具有最小深度为主编号的结点v，这个条件即，存在着一条从$v = v_0$到$w = v_k$的路径，比如说，$v_o \rightarrow v_1$，…，$v_{k-1} \rightarrow v_k$，使得对于$1 \leqslant i \leqslant k-1$，$Dfn(v_i) < Dfn(w)$。

深度为主次序和半必经结点有下面若干有用的性质：

1. 对于有根的有向图中满足$Dfn(v) < Dfn(w)$的任意两个结点v和w，在该流图的深度为主生成树中，任何从v到w的路径必定包含v和w的一个公共祖先。图7-19说明了满足$Dfn(v) < Dfn(w)$的v和w的关系，其中w可以位于v、a、b或c任意一个位置，b是v的祖先u的一个满足$Dfn(b) > Dfn(v)$的后裔。带点的箭头指出可能为空的路径，虚线箭头表示非空路径。如果$w = v$或$w = a$，则v是公共祖先。否则，u是公共祖先。

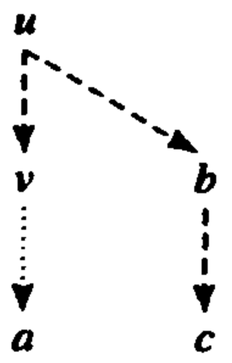


图7-19 对于满足$Dfn(v) < Dfn(w)$的v和w，w可以位于v、a、b或c任意一个位置，其中b是u的某个后裔，且它在v的所有树中的后裔被访问之后才被访问。带点的箭头指出可能为空的路径，虚线箭头表示非空路径

2. 对于任意一个$w \neq r$的结点w，w的半必经结点是w的固有祖先，w的直接必经结点是它的半必经结点的祖先。

3. 令E'表示E中用每一个结点的半必经结点到该结点本身的边替代非树边得到的边集合，则$<N, E', r>$中结点的必经结点与$<N, E, r>$中的必经结点相同。

4. 令

$$V(w) = \{Dfn(v) \mid v \rightarrow w \in E \text{且} Dfn(v) < Dfn(w)\}$$

且

$$S(w) = \{Sdno(u) \mid Dfn(u) > Dfn(w) \text{且对于某个} v \in N,\ v \rightarrow w \in E$$
$$\text{并且存在一条从} u \text{到} v \in E \text{的路径}\}$$

则，w的半必经结点是具有深度为主编号$\min(V(w) \cup S(w))$的结点。

注意，我们没有实际地计算每一个结点v的半必经结点，而只是计算了它的深度为主编号$Sdno(v)$。

185
~
188

bbs.theithome.com

对每一个非根结点v计算了*Sdno*(v)之后，该算法隐式地确定它的直接必经结点如下：令u是这样一个结点，它的半必经结点w是满足如下条件的结点u中具有最小深度为主编号的结点：在深度为主生成树上存在着一条从w到u的非空路径和一条从u到v的路径。如果*Sdno*(v)＝*Sdno*(u)，则v的直接必经结点*Idom*(v)是v的半必经结点；否则，v的直接必经结点*Idom*(v)是*Idom*(u)。

最后，该算法按深度为主次序依次处理每一个结点，对每一个v显式地设置*Idom*(v)。

算法中使用的主要数据结构如下：

1. Ndfs(*i*)是深度为主编号为*i*的结点。
2. Succ(*v*)是结点*v*的后继集。
3. Pred(*v*)是结点*v*的前驱集。
4. Parent(*v*)是结点*v*在深度为主生成树中的父结点。
5. Sdno(*v*)是结点*v*的半必经结点的深度为主编号。
6. Idom(*v*)是结点*v*的直接必经结点。
7. Bucket(*v*)是其半必经结点是Ndfs(*v*)的那些结点的集合。

例程Link()和Eval()管理一个辅助数据结构，即，由深度为主生成树构成的树林，它用于追踪已处理过的结点。Eval()使用Compress()对从函数Ancestor()（参见下面的描述）导出的结点到深度为主生成树的树根的路径执行路径压缩。它由两个数据结构组成，即，

1. Ancestor(*v*)是*v*的祖先，如果*v*在树林中；或者是n0，如果*v*是树林中一棵树的根；

2. Label(*v*)是*v*的祖先链中其半必经结点的深度为主编号最小的一个结点。

最后，Child(*v*)和Size(*v*)是用来使树林中的树保持平衡，并使算法时间达到时间下界的两个数据结构。利用平衡树和路径压缩，这种必经结点查找算法的运行时间界是$O(e \cdot \alpha(e, n))$，其中，*n*和*e*分别是图中结点的个数和边的条数，α ()是一个增长非常缓慢的函数——实质上是
189 Ackermann函数的一个逆函数。如果不使用平衡树，Link()和Eval()函数是相当简单的，但运行时间是$O(e \cdot \log n)$。

关于这个算法是如何工作的更为详细的细节，参见Lengauer和Tarjan 的论文[LenT79]。

作为使用Domin_Fast()算法的例子，我们将它应用到图7-4中的流图。

在Domin_Fast()对Depth_First_Search_Dom()的调用已经返回时(即，图7-16中*1标志点)，Ndfs()、Sdom()和Idom()的值如下所示：

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
1	entry	1	n0
2	B1	2	n0
3	B2	3	n0
4	exit	4	n0
5	B3	5	n0
6	B4	6	n0
7	B5	7	n0
8	B6	8	n0

接着我们给出在标志*2处由上面列出的值变化而来的一系列的快照值。对于i＝8，有变化的行是：

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
8	B6	8	B5

bbs.theithome.com

对于i＝7，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
7	B5	6	n0
8	B6	8	B4

对于i＝6，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
6	B4	5	n0
7	B5	6	B4

对于i＝5，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
5	B3	2	n0
6	B4	5	B3

对于i＝4，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
4	exit	2	n0
5	B3	2	B1

对于i＝3，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
3	B2	3	n0
4	exit	2	n0

对于i＝2，有变化的行是:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
2	B1	2	n0
3	B2	2	B1

在Domin_Fast()的*3和*4之处，所有结点的值如下:

j	Ndfs(j)	Sdno(Ndfs(j))	Idom(Ndfs(j))
1	entry	1	n0
2	B1	1	entry
3	B2	2	B1
4	exit	2	B1
5	B3	2	B1
6	B4	5	B3
7	B5	6	B4
8	B6	6	B4

并且Idom()的值与第一种方法计算出来的值相同。

Alstrup和Lauridsen [AlsL96]描述了一种随过程控制流的修改而增量式地更新必经结点树的技术。他们的技术第一次使得增量式地更新必经结点树的计算复杂性优于重新构造必经结点树的计算复杂性。

7.4 循环和强连通分量

下面，我们定义流图中的回边（back edge）为其头是其尾的必经结点的边。注意，这里定

190

版权所有 翻印必究

bbs.theithome.com

义的回边的概念比7.2节定义的后向边更为严格[⊖]。例如，对于图7-20a的有根的有向图，它的一种深度为主表示如图7-20b所示，此图有一条从d到c且c不是d的必经结点的后向边。尽管这条边定义了一个循环，但该循环有两个进入点(c和d)，因此，它不是一个自然循环。

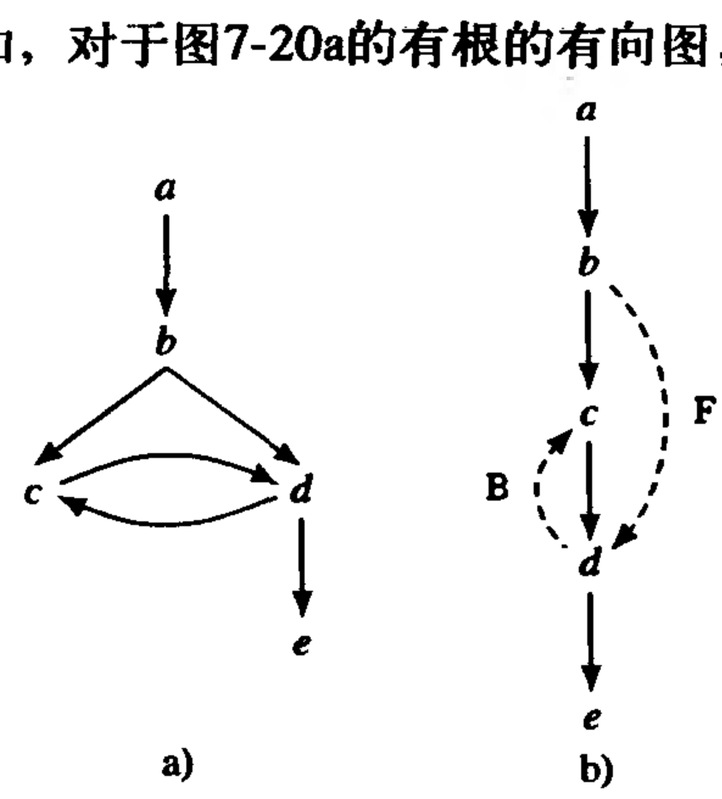


图7-20 a) 有根的有向图，b) 它的深度为主表示

已知一条回边$m \rightarrow n$，$m \rightarrow n$的自然循环(natural loop)是流图中由满足如下条件的结点集合和边集合组成的子图：其中，结点集合由结点n以及流图中那些从它们可以到达m但不经过n的所有结点组成，边集合由所有连接其结点集合中结点的边组成。结点n是循环首结点（loop header)。我们可以用图7-21中的算法Nat_Loop()构造$m \rightarrow n$的自然循环的结点集合。已知一个图和回边，这个算法存储循环的结点集合于Loop。如果需要，从它不难计算出循环的边集合。

```
procedure Nat_Loop(m,n,Pred) returns set of Node
   m, n: in Node
   Pred: in Node ⟶ set of Node
begin
   Loop: set of Node
   Stack: sequence of Node
   p, q: Node
   Stack := []
   Loop := {m,n}
   if m ≠ n then
      Stack ⊕= [m]
   fi
   while Stack ≠ [] do
      || add predecessors of m that are not predecessors of n
      || to the set of nodes in the loop; since n dominates m,
      || this only adds nodes in the loop
      p := Stack↓-1
      Stack ⊖= -1
      for each q ∈ Pred(p) do
         if q ∉ Loop then
            Loop ∪= {q}
            Stack ⊕= [q]
         fi
      od
   od
   return Loop
end    || Nat_Loop
```

图7-21 计算回边$m \rightarrow n$的自然循环

191 我们考虑的许多优化都需要从循环内外提代码到循环的首结点之前。为了保证有一个统一的地方存放这种外提的代码，我们引入前置结点的概念。循环前置结点（loop preheader）是在循环首结点前面建立的一个新的基本块（初始为空），原来从循环外进入循环首结点的所有边

⊖ 英文中两者用的都是“back edge”，为了区别它们，我们将基于必经结点的“back edge”译为“回边”，将7.2节中与“forward edge”相对应的“back edge”译为“后向边”。——译者注

bbs.theithome.com

改成引向前置结点，并且从循环前置结点有一条新的边引向循环首结点。图7-22b给出了对图7-22a中的循环引入循环前置结点后的结果。

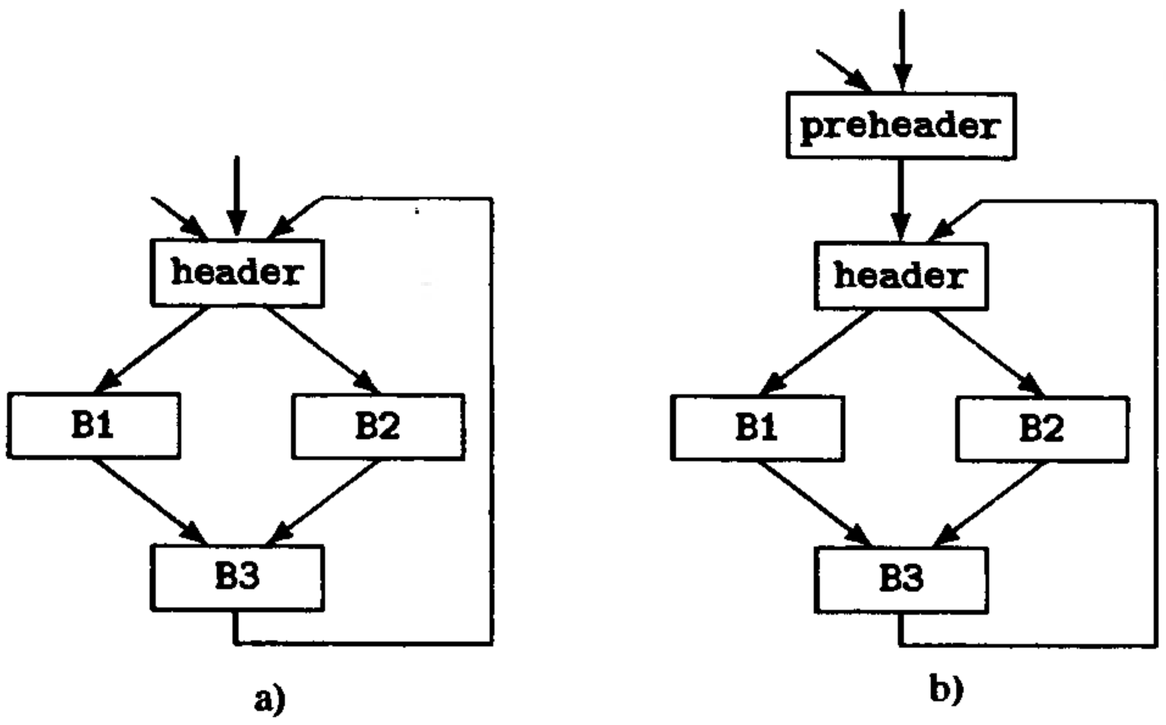


图7-22 a) 没有前置结点的循环，
b) 有前置结点的循环

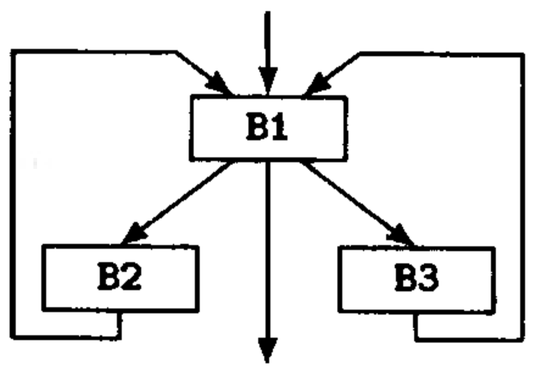


图7-23 具有相同循环首结点B1的两个自然循环

不难看出，若两个自然循环具有同一个首结点，则它们要么不相交，要么一个嵌套在另一个之中。另一方面，给定两个具有相同首结点的循环，如图7-23所示，常常不能清楚地区别出它们是否一个嵌套在另一个之中（如果是嵌套的话，也难于区分谁嵌套了谁），也难于区分它们构成的是否就是一个循环。如果它们是由图7-24a中的代码产生的，显然左边的循环是内层循环；另一方面，如果它们是图7-24b中的代码产生的，它们则更可能是一起构成一个循环。不知道源代码的更多信息，我们就不能区别这两种情况。对这种情形有所了解后，本节将这种情形视为组成单个循环的循环来处理（7.7节讨论的结构分析将区别对待它们）。

```
      i = 1;
B1:   if (i >= 100)
          goto b4;
      else if ((i % 10) == 0)
          goto B3;
      else
B2:       ...
          i++;
          goto B1;
B3:    ...
      i++;
      goto B1;
B4:    ...
```

a)

```
B1:   if (i < j)
          goto B2;
      else if (i > j)
          goto B3;
      else goto B4;
B2:    ...
      i++;
      goto B1;
B3:    ...
      i--;
      goto B1;
B4:    ...
```

b)

图7-24 同样产生图7-23流图的两段不同的C代码序列

自然循环只是流图的强连通分量中的一种类型。可能存在有多个循环入口结点的其他循环结构，如7.5节将看到的那样。尽管这种多个入口结点的循环在实际中很少见，但它们确实存在，因此，我们必须考虑它们。

可能出现的循环结构中最普通的是流图的强连通分量(SCC)，它是一个子图$G_S=<N_S, E_S>$，其中N_S的每一个结点都可以通过仅属于E_S的边组成的通路到达另外的每一个结点。 192~193

bbs.theithome.com

对于一个强连通分量，如果每一个包含它的强连通分量就是它本身，则称这个强连通分量是极大的(maximal)。作为一个例子，考虑图7-25中的流图。由B1、B2和B3以及连接它们的边组成的子图构成了一个极大强连通分量，而由结点B2和边B2→B2组成的子图是一个强连通分量，但不是极大强连通分量。

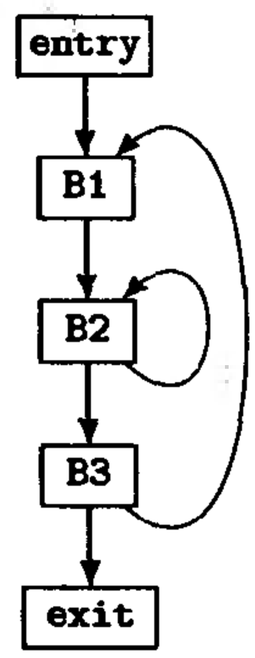


图7-25 具有两个强连通分量的流图，其中一个是极大强连通分量，另一个不是

图7-26中的算法Strong_components(*r*, *Succ*)给出了计算入口结点为*r*的流图的所有极大SCC方法。这是Tarjan算法的一个版本，它计算SCC需要的时间与流图的结点数和边数成正比。函数Dfn: Node→integer是N中结点的深度为主次序。

```
N: set of Node
NextDfn: integer
All_SCC: set of set of Node
LowLink, Dfn: Node ⟶ integer
Stack: sequence of Node

procedure Strong_Components(x,Succ)
    x: in Node
    Succ: in Node ⟶ set of Node
begin
    i: integer
    y, z: Node
    SCC: set of Node
    LowLink(x) := Dfn(x) := NextDfn += 1
    Stack ⊕= [x]
    for each y ∈ Succ(x) do
        if Dfn(y) = 0 then
            Strong_Components(y,Succ)
            LowLink(x) := min(LowLink(x),LowLink(y))
        elif Dfn(y) < Dfn(x) & ∃i ∈ integer (y = Stack↓i) then
            LowLink(x) := min(LowLink(x),Dfn(y))
        fi
    od
    if LowLink(x) = Dfn(x) then  || x is the root of an SCC
        SCC := ∅
        while Stack ≠ [] do
            z := Stack↓-1
            if Dfn(z) < Dfn(x) then
                All_SCC ∪= {SCC}
                return
            fi
            Stack ⊕= -1
            SCC ∪= {z}
        od
        All_SCC ∪= {SCC}
    fi
end     || Strong_Components

begin
    x: Node
```

图7-26 强连通分量的计算

bbs.theithome.com

```
            for each x ∈ N do
                Dfn(x) := LowLink(x) := 0
            od
            NextDfn := 0; Stack := []
            All_SCC := ∅
            for each x ∈ N do
                if Dfn(x) = 0 then
                    Strong_Components(x,Succ)
                fi
            od
            All_SCC ∪= {{Stack↓1}}
        end
```

图7-26 （续）

这个算法的思想如下：对于SCC中的任意一个结点n，令LowLink(n)是SCC中满足如下条件的任意结点m的最小前序编号：在SCC的包含图的深度为主生成树上存在着一条从n到m的路径，且在此路径上至多只有一条后向边或横向边。令LL(n)是具有前序编号LowLink(n)的结点，令n_0是n，n_{i+1}是LL(n_i)。最终我们必定有，对于某个i，$n_{i+1}=n_i$；称这个结点为LLend(n)。Tarjan证明了LLend(n)是包含n的极大SCC中具有最小前序编号的结点，因此它是其结点集合由包含n的SCC的结点组成的图的给定深度为主生成树的根。对每一个n独立地计算LLend(n)需要的时间可能超过线性时间；Tarjan对这个方法进行了修改，通过计算LowLink(n)，然后用它来确定满足n=LL(n)的那些结点n，并由此得到n=LLend(n)，从而使得算法能在线性时间内工作。

7.5 可归约性

可归约性是流图的一个非常重要的性质，并且可能也是一个取名最不恰当的术语。术语可归约（reducible）起源于作用于流图的若干种变换，这些变换将子图蜕化为单个结点，由此连续地“归约”流图到更简单的若干子图。如果一系列的这种转换能够最终将流图归约为单个结点，则称这个流图是可归约的。一个更适合的叫法应当是结构良好的（well-structured），而且我们使用的这个可归约性的定义其概念更清楚一些。但由于历史习惯的影响，我们也交替地使用术语“可归约”。流图$G=<N, E>$是可归约的或结构良好的，当且仅当E可以划分为不相交的前向边集合E_F和后向边集合E_B，使得$<N, E_F>$形成一个从入口结点可到达其中每一个结点的DAG，并且E_B中的边都是7.4节定义的回边。另一种说法是，如果流图是可归约的，则在此流图中的所有循环都是由它们的回边刻画的自然循环，反之亦然。由这个定义得出，可归约流图中没有转入循环体内的转移——每一个循环只能通过它的首结点进入。

某些控制流模式会使得流图是非可归约的，这种模式称为非正常区域（improper region），并且，它们一般是流图的多入口强连通分量。事实上，最简单的非正常区域是图7-27a所示的有两个入口的一个循环，图7-27b是将它扩展得到的一个有三个入口的循环。容易看出，可以从这两个循环产生出无限的（相对简单的）不同非正常区域序列。

某些程序设计语言的语法规则，如Modula-2和它后面的几代语言及BLISS，只允许构造具有可归约流图的过程。在其他多数语言中，只要我们避免使用goto，尤其是转移到循环体内的goto，则构造出来的过程也具有可归约流图。对流图结构的统计研究表明，非可归约的流图是罕见的，即使是在像Fortran 77这种对控制流结构没有多少限制的语言[⊖]和在20年前书写的程序中情况也是

⊖ 这句话不十分贴切。Fortran 77标准确实禁止到do循环内的转移，但它对转移到由分支和goto构成的循环内的分支没有限制。

bbs.theithome.com

196 这样，那时结构化程序设计还没有得到强烈的关注。有两个研究发现，现实中的Fortran 77程序超过90%的具有可归约流图，并且，一个由50个Fortran 66程序构成的集合中，所有程序的流图都是可归约的。因此，非可归约流图在实际中很罕见，因而几乎可以忽略它们的存在。但是，它们又的确存在，因此我们必须保证我们的控制流和数据流分析方法能够处理它们。

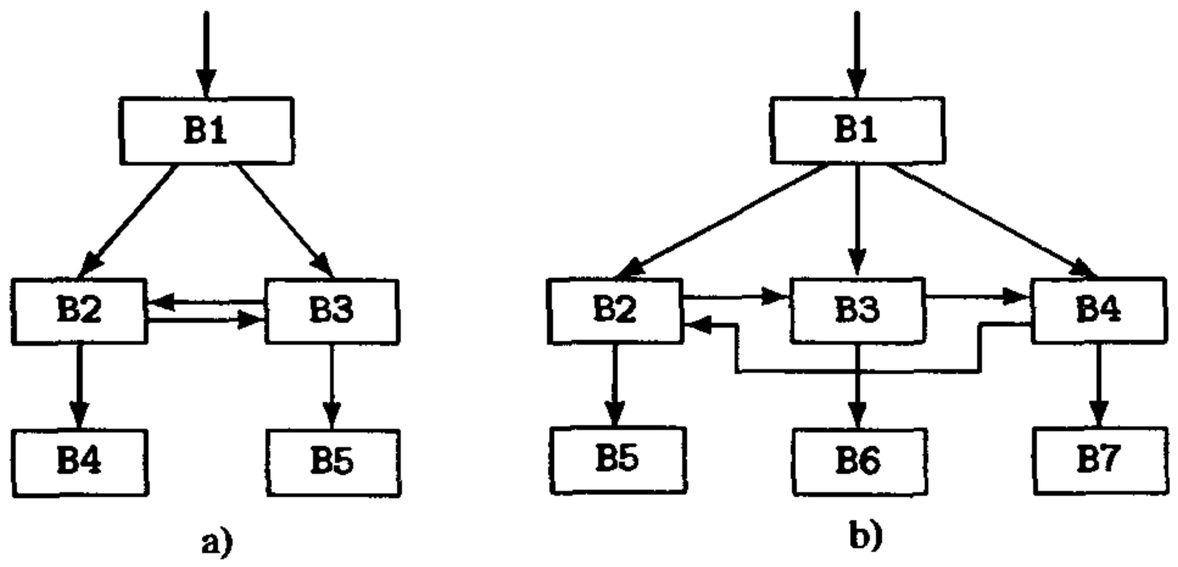


图7-27 简单的非正常区域

在8.6节讨论的基于控制树的数据流分析方法（它依赖于可归约的流图）中有三种处理非可归约流图的方法。第一种方法如8.4节所述，它对非可归约区域进行迭代数据流分析，然后将结果插入到流图中其余部分的数据流方程中。第二种方法利用一种称为结点分割（node splitting）的技术将非可归约区域转换成可归约的区域。如果我们分割图7-27a中的结点B3，得到的是如图7-28所示的流图：B3变成了一对结点B3和B3a，这个循环现在成为了只有一个入口B2的自然循环。假若非可归约流图是常见的，结点分割的开销将非常昂贵，因为它会使流图的大小呈指数级增长；所幸的是现实中不是这种情况。第三种方法是对一个单调函数的格执行从这个格到它自身的诱导迭代(参见8.5节和8.6节)。

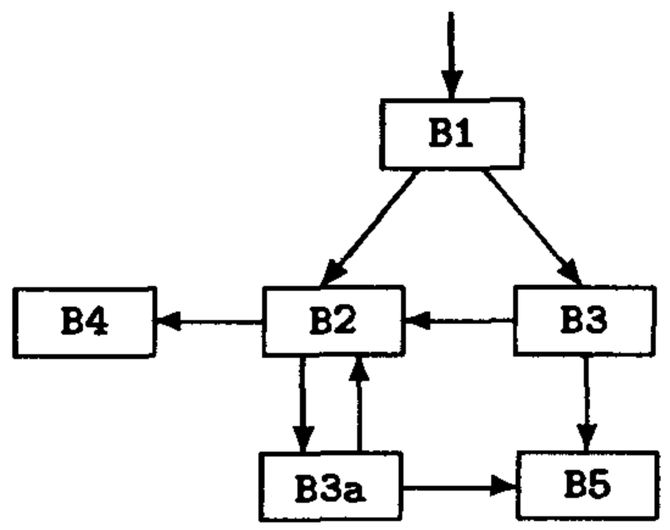


图7-28 对图7-27a中非正常区域中的结点B3施加结点分割后的结果

7.6 区间分析和控制树

区间分析（interval analysis）是对控制流和数据流分析两者使用的若干种方法的统一叫法。

197 区间分析指的是将流图划分为各种类型的区域（类型取决于具体的方法），使每一个区域蜕化成一个新的结点（常称为抽象结点（abstract node），因为它将区域的内部结构抽象化了），并用进入或离开对应抽象结点的边替换进入或离开区域的边。由一次或多次这种转换而得到的流图称为抽象流图（abstract flowgraph）。因为这种转换每次施加于一个子图，或并行地施加于不相交的若干子图，故从每一个抽象结点对应于一个子图的意义上来说，所得到的区域是嵌套的。因此，施加一系列这种转换的结果是产生一棵其定义如下的控制树（control tree）：

1. 控制树的根结点是表示原始流图的抽象图。
2. 控制树的叶结点是单个基本块。
3. 在根结点和叶结点中间的结点是表示流图各个区域的抽象结点。
4. 树的边表示每一个抽象结点和那些是其后裔的（并是从这个结点抽象而来的）区域之间的关系。

例如，最简单的并且也是最早的一种区间分析形式是著名的*T*1-*T*2分析。它只由两种转换组成：*T*1蜕化仅一个结点的自循环为单个结点，*T*2蜕化那种第一个结点是第二个结点惟一前驱

bbs.theithome.com

的两个结点序列为单个结点，如图7-29所示。

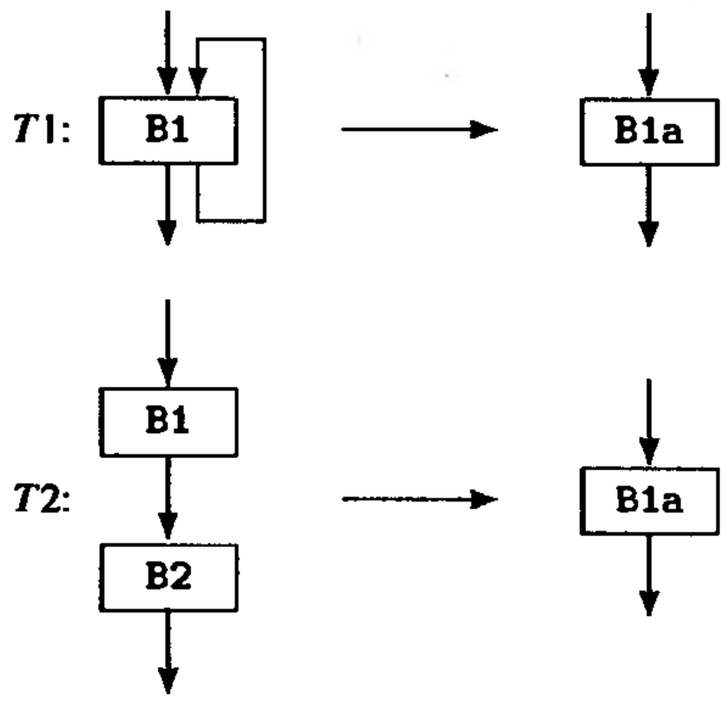


图7-29 *T*1-*T*2转换

现在，假设给定一个图7-30左边所示的流图，对它重复地应用*T*1-*T*2转换，我们得到图7-30中所示的一系列变形，相应的控制树如图7-31所示。

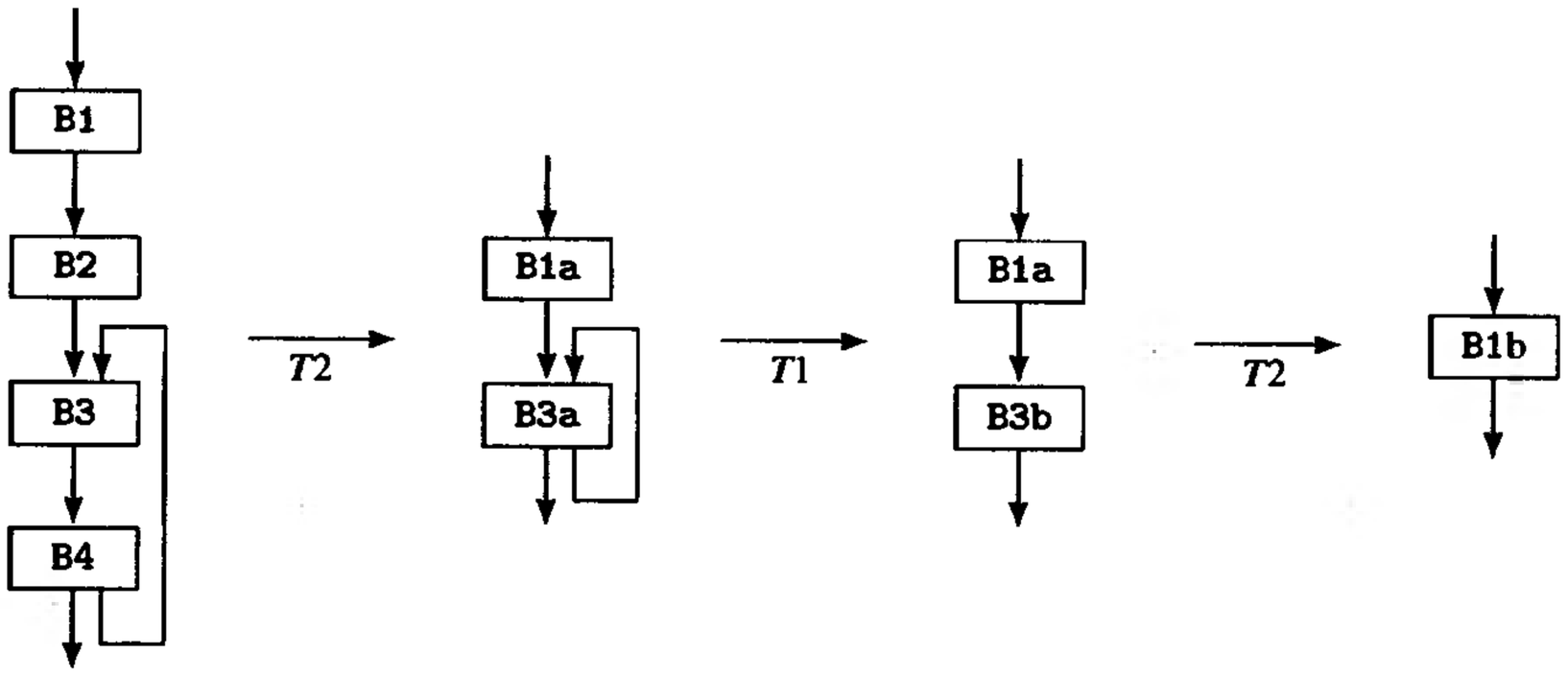


图7-30 *T*1-*T*2转换的例子

如最早所定义的，区间分析使用所谓的极大区间，并忽略非可归约或非正常区域的存在。首结点为*h*的极大区间（maximal interval）$I_M(h)$是以*h*作为它的惟一入口结点，并且子图中所有闭合通路都包含*h*的最大单入口子图。实质上，$I_M(h)$是入口结点为*h*的自然循环，外加从它的出口悬挂出来的某种无环结构。例如在图7-4中，I_M(B4)是{B4, B6, B5, exit}，包含B6是因为包含B4的惟一闭合通路是由B4→B6和B6→B4组成的通路，包含B5和exit是因为不这样的话，子图就不是最大的。

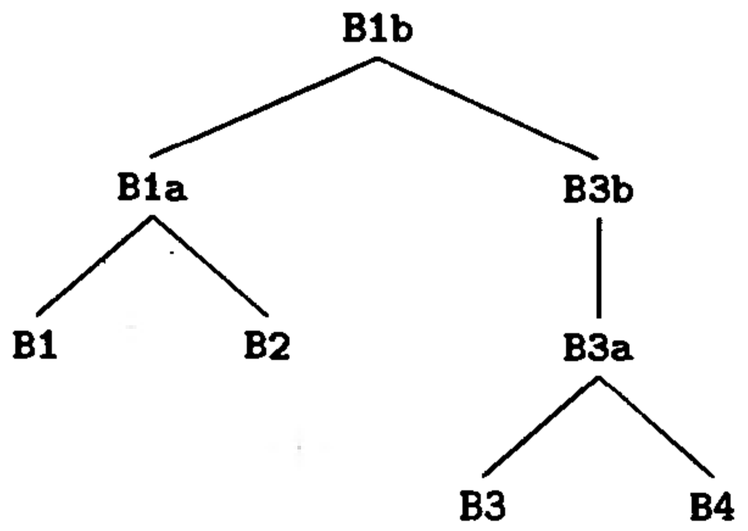


图7-31 图7-30中流图的*T*1-*T*2控制树

本节剩余部分关注一种更为现代的区间分析形式，这种区间分析形式标识流图中的循环而不对其他控制结构类型进行分类。在此上下文中，定义极小区间（minimal interval，或简称区间）*I*是：(1) 自然循环，(2) 最大无环子图，或(3)极小非可归约区域。因此，一个是自然循环的极小区间与对应的极大区间的不同在于，后者包含了循环中结点的这种后继结点，这种后继结点本身既不属于循环，也不是极大区间的首结点；而前者排斥它们。例如，图7-32a和b分别说明了同一个流图中的极大区间和极小区间。

bbs.theithome.com

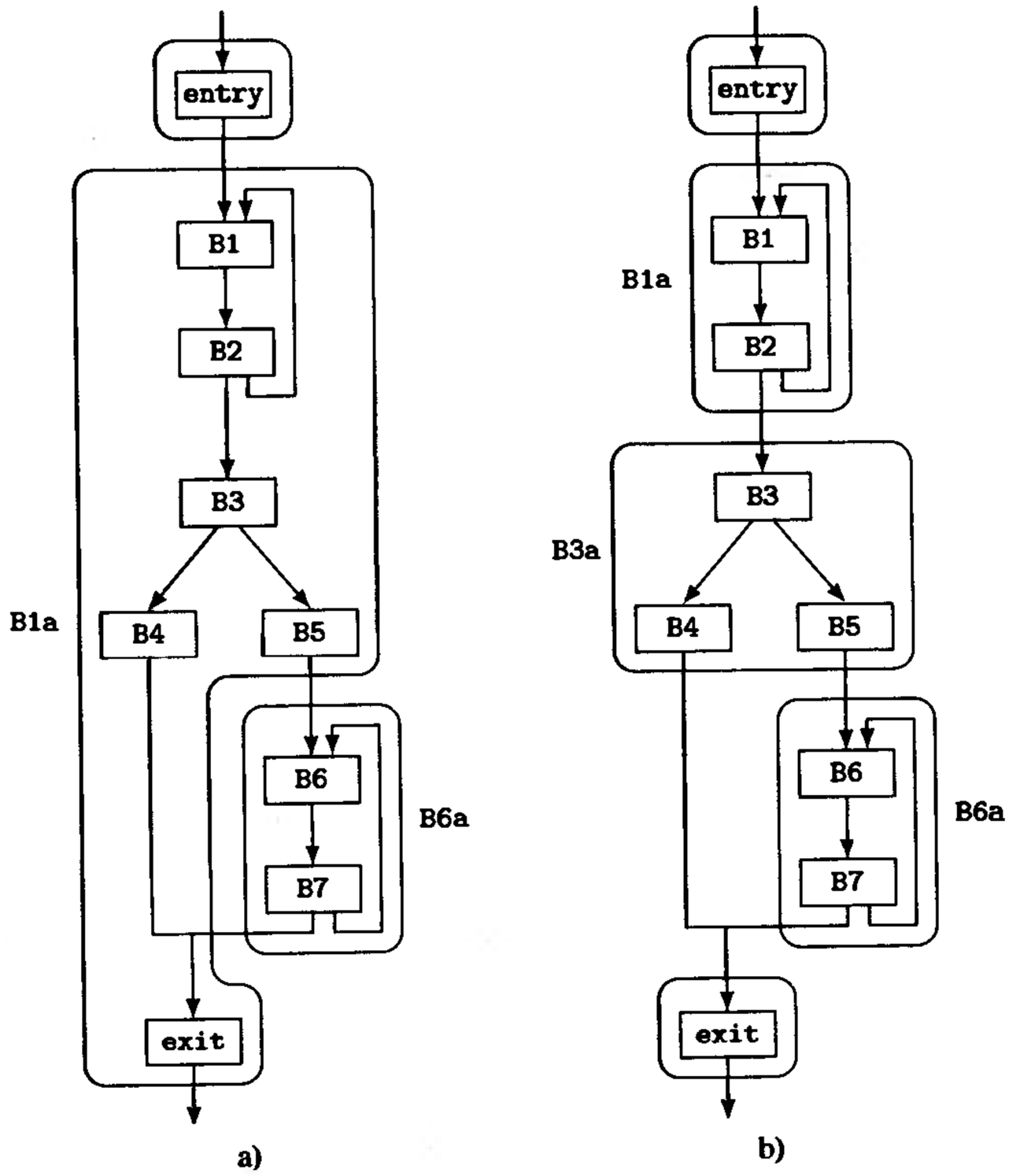


图7-32 说明a) 极大区间和b) 极小区间之间不同的例子

图7-33给出了一个稍微复杂一点的例子。在这个例子中，我们不用名字命名抽象子图，而是简单地给出组成它们的集合的每一个结点——这样使得控制树(如图7-34所示)更为清楚。基本块B2和B4形成一个循环，B5和B6也形成一个循环。当它们都蜕化为单个结点后，我们发现B3和{B5，B6}形成了一个非可归约区域，它也蜕化成单个结点。剩余的抽象图是无环的，并因此形成了单个区间。

198 ~ 199

因为我们认为结构分析（7.7节讨论其详细内容）要优于区间分析，因此这里只给出区间分析方法的要点[⊖]。区间分析方法的基本步骤如下:

1. 执行流图结点集合的后序遍历，寻找循环首结点（每个循环一个）和非正常区域的首结点（每个区域一个集合，集合中结点个数大于1)。

2. 对找到的每一个循环首结点，用图7-21给出的算法Nat_loop()构造出它的自然循环，并将它归约成类型为“自然循环”的抽象区域。

3. 对于每一个非正常区域的入口集合，构造包含所有入口的流图的极小强连通分量（可修改图7-26给出的算法以构造极小SCC)，并将它归约成类型为“非正常区域”的抽象区域。

4. 对于entry结点，以及对于自然循环中或非可归约区域中的结点的每一个直接后裔，构造以此结点作为根结点的极大无环图；如果所产生的图有多于一个的结点，将它归约成类型为“无环区域”的抽象区域。

5. 迭代此过程直至终止。

⊖ 实际上，区间分析可看成是一种使用较少区域或区间类型的结构分析的削减版本。因此，执行区间分析的算法可以从执行结构分析的算法中导出。

bbs.theithome.com

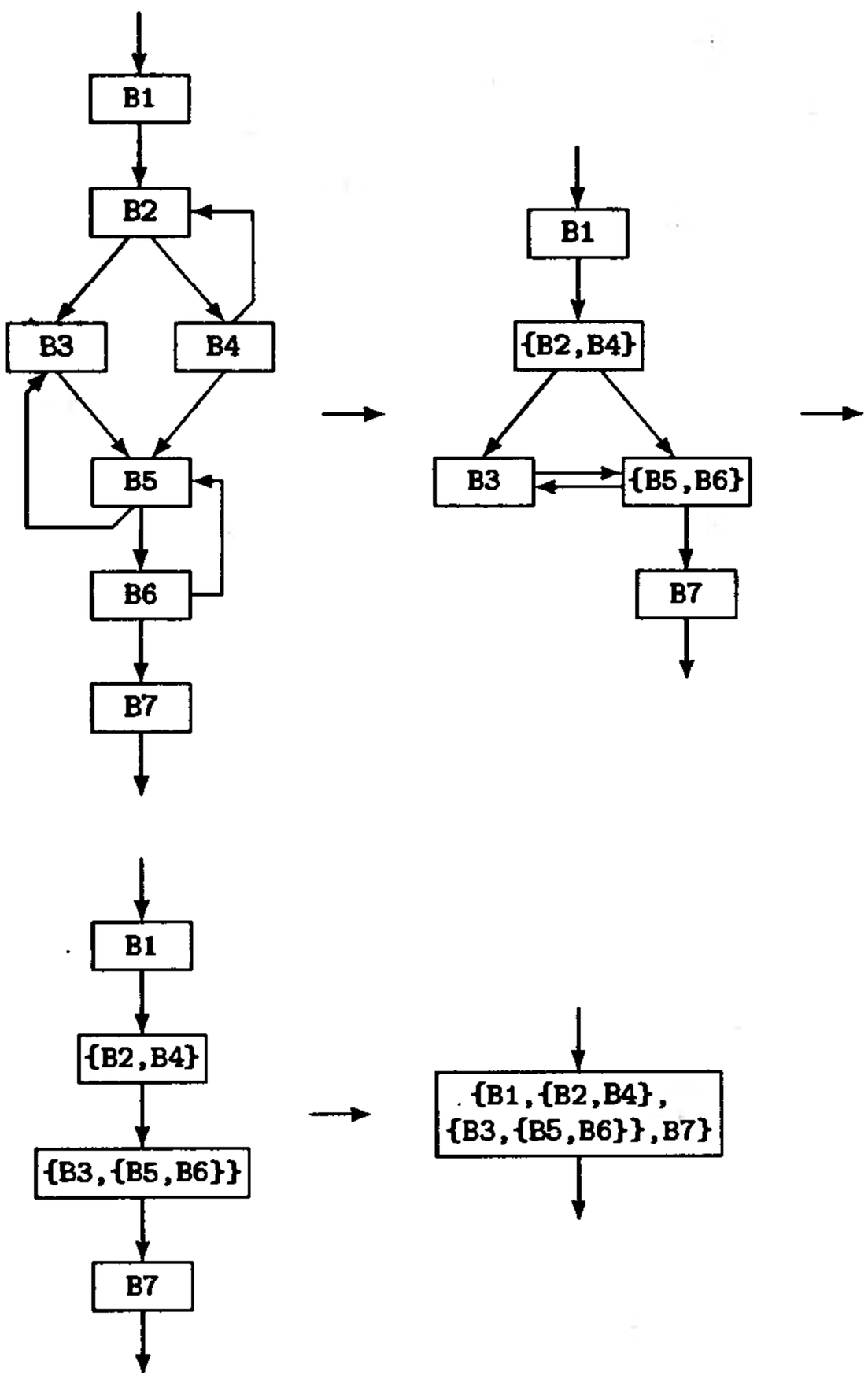


图7-33 区间分析的例子

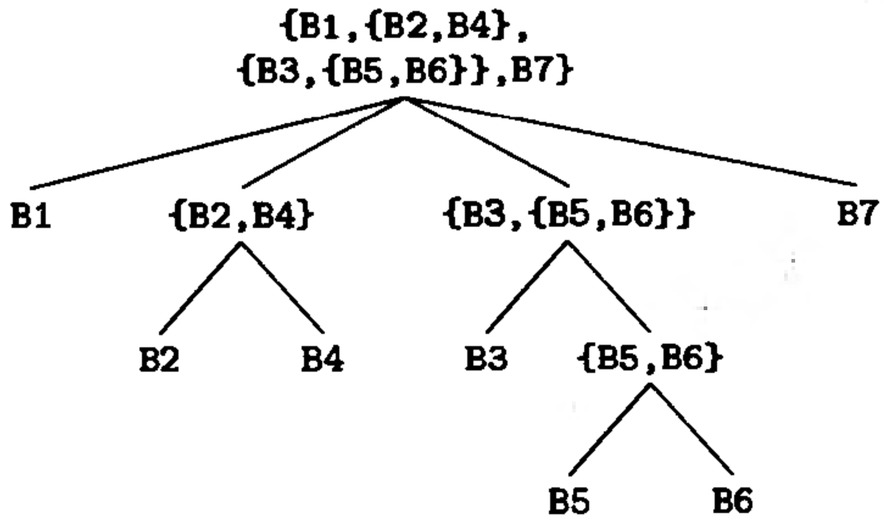


图7-34 图7-33中流图的控制树

注意，这个过程的终止是有保证的，因为流图要么是无环图，要么包含某种类型的循环。如果它是无环的，该过程在当前这个迭代终止；如果它包含多个循环，则在每一个迭代至少会出现对一个自然循环或非正常区域的归约，从而导致此图的循环个数最后变成1，而每个流图一开始只包含有限数目的循环。

7.7 结构分析

结构分析是一种更为精致的区间分析形式，它的目的是使数据流分析的语法制导方法能应用于低级中间代码（Rosen开发的这种方法是用于语法树的）。Rosen的方法称为高级数据流分析（high-level data-flow analysis），其优点是，对于源语言中每一种类型的结构化的控制流构造，它给出一组规则，用这些规则执行常规的（位向量的）跨越和流经这种控制流构造的数据流分析比用迭代方法更具效率。因此，这种方法扩展了优化的目标之一，即，通过将编译时的工作移到语言定义时，从而使得执行时的工作能在编译时完成。具体而言，就是通过语言的语法和语义来确定结构化的控制流的数据流方程。

结构分析通过发现流图中的控制流结构，并提供一种处理非正常区域的途径，将这种方法扩充到可处理任意的流图。例如，结构分析可以接受由goto、if和赋值形成的循环，并区别

bbs.theithome.com

出它们是while还是repeat循环，即使它们的语法没有给出这种提示。

结构分析与基本的区间分析的不同在于，它能标识出比循环更多的控制结构类型，并使每一种类型的控制结构形成一个区域，其结果是为实现非常有效的数据流分析提供了一种基础。它建立的控制树一般情况下要比区间分析建立的控制树更大——因为区域类型越多，标识出的区域就越多——但是，每一个区域相对要简单些也小些。结构分析中的一个关键概念是，它所标识的每一个区域只有一个入口点，因此，非可归约区域或非正常区域总是包含后面这种强连通分量入口结点集合的最低公共必经结点，这种强连通分量是该非正常区域内的多入口循环。

200～202

图7-35和图7-36分别给出了结构分析能够识别的典型的无环和有环控制结构。注意，这些类型的区域中哪些适合于给定的源语言是随所选择的语言而变化的，并且也可能还存在其他类型的区域。例如，特定语言的case/switch结构可能允许、也可能不允许从一个case顺序下降到下一个case，而不是直接分支到结构的出口——在C的switch语句中，任何case都可以顺序下降到下一个case，或直接分支到结构的出口；而在Pascal的case中，所有case都分支到出口。因此，case/switch结构总是被用来作为一种覆盖可能范围的真实模型。注意，图中所指的自然循环表示的是不包括另外两种特殊可归约循环结构（即不是自我循环或while循环）在内的可归约循环，并且它只是示意性的，因为，循环可能不只有两条出口边。

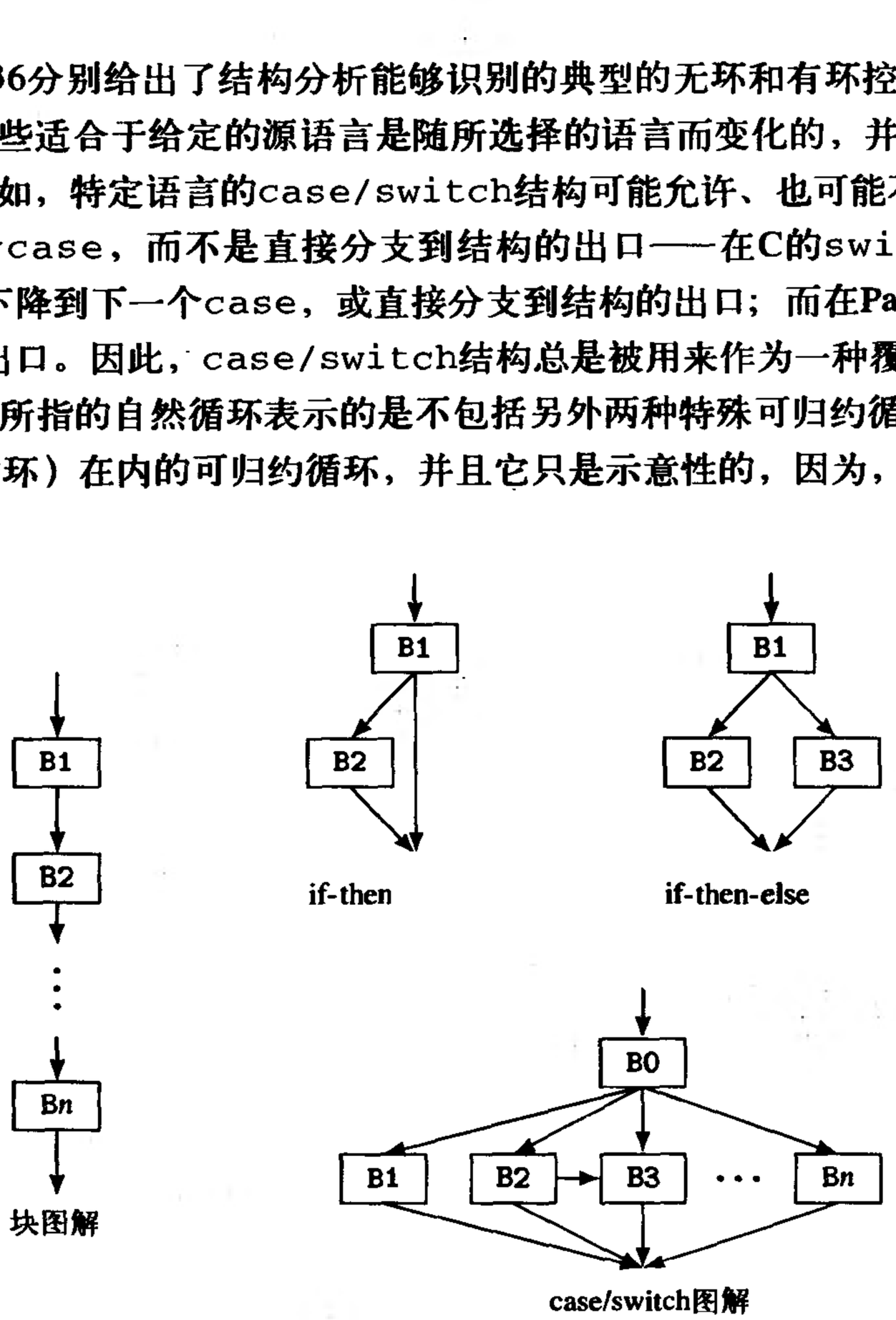


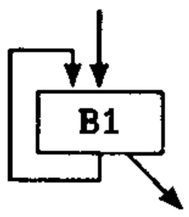
图7-35 结构分析中使用的几种类型的无环区域

类似地，非正常（或不可归约）区间也是示意性的，因为它的入口基本块可能有两个以上的后继，并且可能包含三个以上的基本块。结构分析中使用了一种更典型的区间，即正常区间(proper interval)，它是一种独特的无环结构，即，既没有包含环路但也不能被任何简单的无环情形所归约的结构。这种结构的例子如图7-37所示。

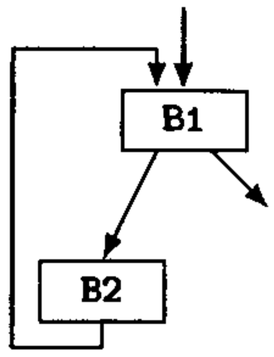
另外，图7-23中所表示的那种有两条回边进入B1的情形，结构分析将其识别为两个嵌套的while循环，它们中哪一个是内层循环取决于遇到它们时的顺序。

bbs.theithome.com

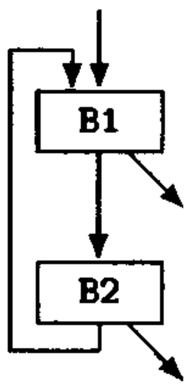
超星公司制作 请尊重作者版权



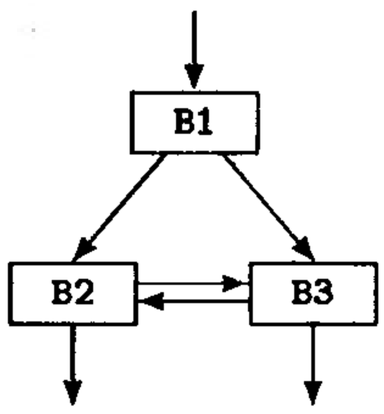
自我循环



while循环



自然循环图解



非正常区间图解

图7-36 结构分析中使用的几种类型的有环区域

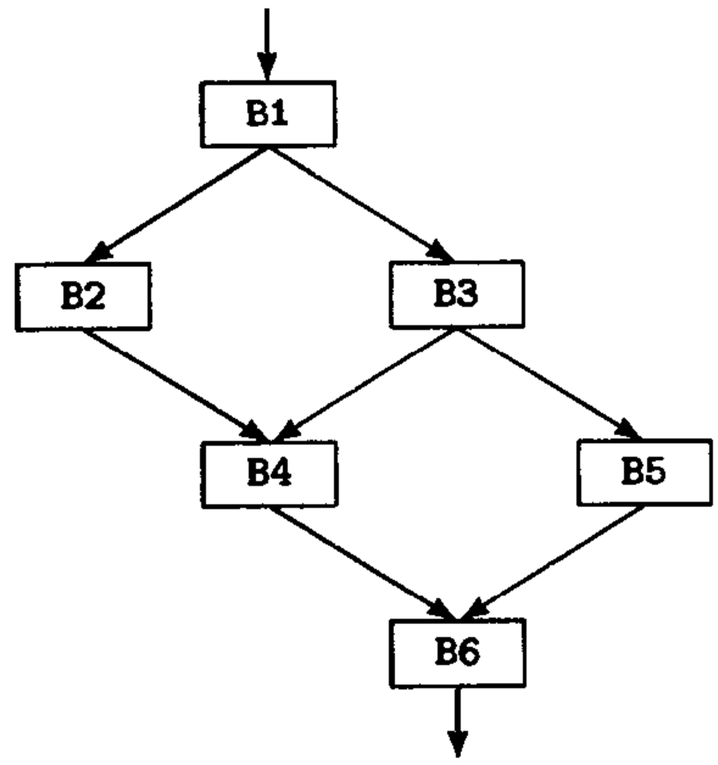


图7-37 一个不适合任何简单分类的无环区域，因此将其标识为正常区间

结构分析的处理过程是，构造所考虑流图的深度为主生成树，然后对各种区域类型的实例按后序次序依次考察流图中的结点，由它们形成抽象结点，蜕化掉连接边，并构造对应的控制树。对各种类型的区域进行检查的顺序以及检查它们的方法很重要，例如，对于一个可以构成一个块的$n \geqslant 3$的区域序列，如果我们在形成该区域之前注视到这个序列的两端，便可以只用一步将它蜕化成一块；但如果我们每一步只检查第一块，然后检查它的前驱或后继，则为了将它蜕化到由$n-1$块组成的层次，需要用$n-1$步。显然，前一种方法更好。

203 ~ 204

按照Sharir [Shar80]的方法，当我们分析一个流图时，构造名为StructOf、StructType、Structures和StructNodes的四种数据结构（如图7-38所示）。StructOf对于每一个结点，给出直接包含该结点的（抽象）区域结点。StructType对于每一个区域结点，给出它的类型。Structures是所有区域结点组成的集合。StructNodes对于每一个区域，给出其内结点的结点表。

```
Succ, Pred: Node ⟶ set of Node
RegionType = enum {Block,IfThen,IfThenElse,Case,Proper,SelfLoop,
   WhileLoop,NaturalLoop,Improper}
|| StructOf(n) = the region containing node n
StructOf: Node ⟶ Node
|| StrucType(n) = the member of RegionType that is the type of
|| the region denoted by node n
StructType: Node ⟶ RegionType
|| the set of all region nodes
Structures: set of Node
|| StructNodes(n) = the set of nodes making up the region
```

图7-38 用于结构分析的全局数据结构

bbs.theithome.com

```
|| abstracted to node n
StructNodes: Node ⟶ set of Node
|| node and edge sets of the control tree
CTNodes: set of Node
CTEdges: set of (Node × Node)
|| postorder traversal of the flowgraph
PostCtr, PostMax: integer
Post: integer ⟶ Node
Visit: Node ⟶ boolean
```

图7-38 （续）

图7-39给出的结构分析算法Structural_Analysis()假定我们使用的是图7-35和图7-36所示的那些区域类型，外加前面描述过的其他类型。不过，只要适合于所处理的源语言，它也可以使用其他区域类型。该算法首先初始化上面描述的数据结构，这些数据结构记录流图的层次结构和表示控制树的结构(CTNodes和CTEdges)。然后它对流图进行深度为主查找，从而构造出流图结点的后序遍历。接着，通过对流图进行一系列遍历标识出每一个区域，并使它蜕化到单个抽象区域结点。如果执行归约，它修复流图的结点集合、边集合和后序次序（如果需要），并再次对图进行处理。该算法用新抽象结点替代原区域，并用进入新抽象结点的边替代进入原区域的边，用离开这个新结点的边替代离开原区域的边。同时，该算法还构造出流图的控制树。

```
procedure Structural_Analysis(N,E,entry)
   N: in set of Node
   E: in set of (Node × Node)
   entry: in Node
begin
   m, n, p: Node
   rtype: RegionType
   NodeSet, ReachUnder: set of Node
   StructOf := StructType := Structures := StructNodes := ∅
   CTNodes := N; CTEdges := ∅
   repeat
      Post := ∅; Visit := ∅
      PostMax := 0; PostCtr := 1
      DFS_Postorder(N,E,entry)
      while |N| > 1 & PostCtr ≤ PostMax do
         n := Post(PostCtr)
         || locate an acyclic region, if present
         rtype := Acyclic_Region_Type(N,E,n,NodeSet)
         if rtype ≠ nil then
            p := Reduce(N,E,rtype,NodeSet)
            if entry ∈ NodeSet then
               entry := p
            fi
         else
            || locate a cyclic region, if present
            ReachUnder := {n}
            for each m ∈ N do
               if Path_Back(m,n) then
                  ReachUnder ∪= {m}
               fi
            od
            rtype := Cyclic_Region_Type(N,E,n,ReachUnder)
```

图7-39 结构分析算法

bbs.theithome.com

```
                if rtype ≠ nil then
                   p := Reduce(N,E,rtype,ReachUnder)
                   if entry ∈ ReachUnder then
                      entry := p
                   fi
                else
                   PostCtr += 1
                fi
             fi
          od
       until |N| = 1
    end   || Structural_Analysis
```

图7-39 （续）

集合ReachUnder用来确定有环控制结构中包含的结点。如果ReachUnder只包含一个结点，并且存在一条从这个结点到自身的边，则这个有环结构是一个自我循环。如果它包含多于一个的结点，则可能是一个while循环，也可能是自然循环，还可能是非正常区域。如果ReachUnder中的结点都是第一个被放置进来的那个结点的后裔，则这个循环是一个while循环或自然循环。如果它包含一个第一个被放置进来的结点的非后裔结点，则这个区域是一个非正常区域。注意，对于非正常区域，结果获得的这个区域的入口结点不是此有环结构的一部分，因为所有区域都只有一个入口结点；图7-45中的例程Minimize_Improper()确定构成这种区域的结点集合。 205

计算ReachUnder用到了函数Path_Back(*m*, *n*)，如果存在这样的一个结点*k*：有一条从*m*到*k*且不经过*n*的路径（可能为空），并且有一条*k*→*n*的回边，则该函数返回true；否则返回false。

当流图归约成只有一个结点且没有边的图后，该算法便终止。图7-40给出的例程DFS_Postorder()构造流图中结点的后序遍历。图7-41给出的例程Acyclic_Region_Type(*N*, *E*, *node*, *nset*)确定*node*是否是一个无环控制结构的入口结点，并返回控制结构的类型或nil（如果不是这样一个入口结点）；该例程在*nset*中存储所标识的控制结构的结点集合。

```
procedure DFS_Postorder(N,E,x)
    N: in set of Node
    E: in set of (Node × Node)
    x: in Node
begin
    y: Node
    Visit(x) := true
    for each y ∈ Succ(x) (Visit(y) = nil) do
        DFS_Postorder(N,E,y)
    od
    PostMax += 1
    Post(PostMax) := x
end     || DFS_Postorder
```

图7-40 计算流图结点的后序遍历

图7-42给出的例程Cyclic_Region_Type(*N*, *E*, *node*, *nset*)判定*node*是否为一个有环结构的入口结点，并返回它的类型或nil（如果不是这样一个入口结点）；它同样将所标识的控制结构的结点集合存储在*nset*中。

bbs.theithome.com

```
procedure Acyclic_Region_Type(N,E,node,nset) returns RegionType
   N: in set of Node
   E: in set of (Node × Node)
   node: inout Node
   nset: out set of Node
begin
   m, n: Node
   p, s: boolean
   nset := ∅
   || check for a Block containing node
   n := node; p := true; s := |Succ(n)| = 1
   while p & s do
      nset ∪= {n}; n := ♦Succ(n); p := |Pred(n)| = 1; s := |Succ(n)| = 1
   od
   if p then
      nset ∪= {n}
   fi
   n := node; p := |Pred(n)| = 1; s := true
   while p & s do
      nset ∪= {n}; n := ♦Pred(n); p := |Pred(n)| = 1; s := |Succ(n)| = 1
   od
   if s then
      nset ∪= {n}
   fi
   node := n
   if |nset| ≥ 2 then
      return Block
   || check for an IfThenElse
   elif |Succ(node)| = 2 then
      m := ♦Succ(node); n := ♦(Succ(node) - {m})
      if Succ(m) = Succ(n) & |Succ(m)| = 1
         & |Pred(m)| = 1 & |Pred(n)| = 1 then
         nset := {node,m,n}
         return IfThenElse
      || other cases (IfThen, Case, Proper)
      ∈lif . . .
         . . .
      else
         return nil
      fi
   fi
end   || Acyclic_Region_Type
```

图7-41 标识无环结构类型的例程

```
procedure Cyclic_Region_Type(N,E,node,nset) returns RegionType
   N: in set of Node
   E: in set of (Node × Node)
   node: in Node
   nset: inout set of Node
begin
   m: Node
   || check for a SelfLoop
   if |nset| = 1 then
      if node→node ∈ E then
         return SelfLoop
      else
```

图7-42 标识有环结构类型的例程

bbs.theithome.com

```
            return nil
         fi
      fi
      if ∃m ∈ nset (!Path(node,m,N)) then
         || it's an Improper region
         nset := Minimize_Improper(N,E,node,nset)
         return Improper
      fi
      || check for a WhileLoop
      m := ♦(nset - {node})
      if |Succ(node)| = 2 & |Succ(m)| = 1 &
         |Pred(node)| = 2 & |Pred(m)| = 1 then
         return WhileLoop
      else
         || it's a NaturalLoop
         return NaturalLoop
      fi
   end   || CyclicEntryType
```

图7-42 （续）

图7-43定义的例程Reduce(*N*, *E*, *rtype*, *NodeSet*)调用Create_Node()来创建一个区域结点*n*，并用它表示所标识的区域，同时相应地设置StructType、Structures、StructOf和StructNodes等数据结构，并返回*n*作为其返回值。Reduce()使用了Replace()，后者在图7-44中定义，它用新结点替代所标识的区域，相应地调整进入和离开的边，以及前驱和后继函数，并建立由CTNodes和CTEdges表示的控制树。

```
procedure Reduce(N,E,rtype,NodeSet) returns Node
   N: inout set of Node
   E: inout set of (Node × Node)
   rtype: in RegionType
   NodeSet: in set of Node
begin
   node := Create_Node( ), m: Node
   || replace node set by an abstract region node and
   || set data structures
   Replace(N,E,node,NodeSet)
   StructType(node) := rtype
   Structures ∪= {node}
   for each m ∈ NodeSet do
      StructOf(m) := node
   od
   StructNodes(node) := NodeSet
   return node
end   || Reduce
```

图7-43 结构分析的区域归约例程

Replace()中使用的例程Compact(*N*, *n*, *nset*)将结点*n*加入到*N*中，按结点在*nset*中的最高编号位置将*n*插入到Post()中，同时从*N*和Post()中删除*nset*中的结点，使剩余的结点集中在Post()的开始，设置PostCtr为*n*在新产生的后序遍历中的索引，并相应设置PostMax；它返回*N*的新值。

图7-45给出的例程Minimize_Improper(*N*, *E*, *n*, *nset*)用于确定包含*n*的一个较小的非正常区域。根据由DFS_Postorder()给出的流图中结点的次序，它限制这个非正常区域要么为

206 ~ 209

bbs.theithome.com

一个最小子图，这个最小子图包含结点*n*和至少两个其他结点，使得(1)这些结点中除*n*之外有一个结点是子图中所有结点的必经结点，并且(2)该子图中从*n*到另外某个结点的非空路径上的任何结点也都在该子图中；要么为一个稍微大一点的包含这种最小子图作为子图的非正常区域。

```
procedure Replace(N,E,node,NodeSet)
   N: inout set of Node
   E: inout set of (Node × Node)
   node: in Node
   NodeSet: in set of Node
begin
   || link region node into abstract flowgraph, adjust the postorder traversal
   || and predecessor and successor functions, and augment the control tree
   m, ctnode := Create_Node( ): Node
   e: Node × Node
   N := Compact(N,node,NodeSet)
   for each e ∈ E do
      if e@1 ∈ NodeSet ∨ e@2 ∈ NodeSet then
         E -= {e}; Succ(e@1) -= {e@2}; Pred(e@2) -= {e@1}
         if e@1 ∈ N & e@1 ≠ node then
            E ∪= {e@1→node}; Succ(e@1) ∪= {node}
         elif e@2 ∈ N & e@2 ≠ node then
            E ∪= {node→e@2}; Pred(e@2) ∪= {node}
         fi
      fi
   od
   CTNodes ∪= {ctnode}
   for each n ∈ NodeSet do
      CTEdges ∪= {ctnode→n}
   od
end   || Replace
```

图7-44 完成结点和边替代，并建立结构分析控制树的例程

```
procedure Minimize_Improper(N,E,node,nset) returns set of Node
   N, nset: in set of Node
   E: in set of (Node × Node)
   node: in Node
begin
   ncd, m, n: Node
   I := MEC_Entries(node,nset,E): set of Node
   ncd := NC_Domin(I,N,E)
   for each n ∈ N - {ncd} do
      if Path(ncd,n,N) & ∃m ∈ I (Path(n,m,N-{ncd})) then
         I ∪= {n}
      fi
   od
   return I ∪ {ncd}
end   || Minimize_Improper
```

图7-45 结构分析的非正常区间极小化例程

Minimize_Improper()使用了两个函数，即MEC_Entries(*n*, *nset*, *E*)和NC_Domin(*I*, *N*, *E*)；MEC_Entries(*n*, *nset*, *E*)返回*n*是其入口之一的最小多入口环路的入口结点集合（*nset*的所有成员），NC_Domin(*I*, *N*, *E*)返回*I*中结点的最近公共必经结点。NC_Domin()很容易从流图的必经结点树计算出来。如果存在一条从*n*到*m*的路径使得该路径上的所有结点都是*I*中的结点，函数Path(*n*, *m*, *I*)返回true；否则返回false。

bbs.theithome.com

超星公司制作 请尊重作者版权

后序对所产生的非正常区域的判定会有所影响，有关例子参见后面图7-49b的讨论。这里给出的方法几乎总是产生比Sharir原来的方法要小的非正常区间。

210 ~ 211

图7-46a中的流图是结构分析的一个例子。图7-47给出了此流图的深度为主生成树。如图7-46b所示，分析的第一步[⊖]做了三个归约：entry结点以及其后的B1作为一个块被识别并被归约，B2作为一个自我循环被识别并被归约，B5和B6作为if-then结构被识别并被归约。它设置数据结构如下所示：

```
StructType(entrya) = Block
StructOf(entry) = StructOf(B1) = entrya
StructNodes(entrya) = {entry,B1}
StructType(B2a) = SelfLoop
StructOf(B2) = B2a
StructNodes(B2a) = {B2}
StructType(B5a) = IfThen
StructOf(B5) = StructOf(B6) = B5a
StructNodes(B5a) = {B5,B6}
Structures = {entrya,B2a,B5a}
```

下一步如图7-46c所示，识别和归约由entrya和B2a组成的if-then以及由B5a和B7组成的块。它设置数据结构如下所示：

```
StructType(entryb) = IfThen
StructOf(entrya) = StructOf(B2a) = entryb
StructNodes(entryb) = {entrya,B2a}
StructType(B5b) = Block
StructOf(B5a) = StructOf(B7) = B5b
StructNodes(B5b) = {B5a,B7}
Structures = {entrya,B2a,B5a,entryb,B5b}
```

212

接下来的步骤如图7-46d所示，B3、B4和B5b作为if-then-else被归约，设置的数据结构如下所示：

```
StructType(B3a) = IfThenElse
StructOf(B3) = StructOf(B4) = StructOf(B5b) = B3a
StructNodes(B3a) = {B3,B4,B5b}
Structures = {entrya,B2a,B5a,entryb,B5b,B3a}
```

最后一步，entryb、B3a和exit被归约为一个块，得到图7-46e。设置的数据结构如下：

```
StructType(entryc) = Block
StructOf(entryb) = StructOf(B3a) = StructOf(exit) = entryc
StructNodes(entryc) = {entryb,B3a,exit}
Structures = {entrya,B2a,B5a,entryb,B5b,B3a,entryc}
```

最后得到的控制树如图7-48所示。

图7-49给出了两个包含非正常区域的流图例子。在例a)中，例程Minimize_Improper()识别由B6之外的所有结点组成的子图为一个非正常区间。在b)中，识别出来的非正常区间取决于所使用的特定后序遍历：如果B3先于B2，则它识别一个包含B1、B3和B4的非正常区域，然后这个区域连同B2和B5一起被识别为该区域的一个抽象结点；如果B2先于B3，则它识别由所有5个结点组成的一个非正常区域。

⊖ 这个图实际展示的分析过程可看成是结构分析的并行版本，它在每一遍可同时进行若干归约。图中这样表示是为了节省空间，在算法中也可以这样实现，但以显著降低算法的可理解性为代价。

bbs.theithome.com

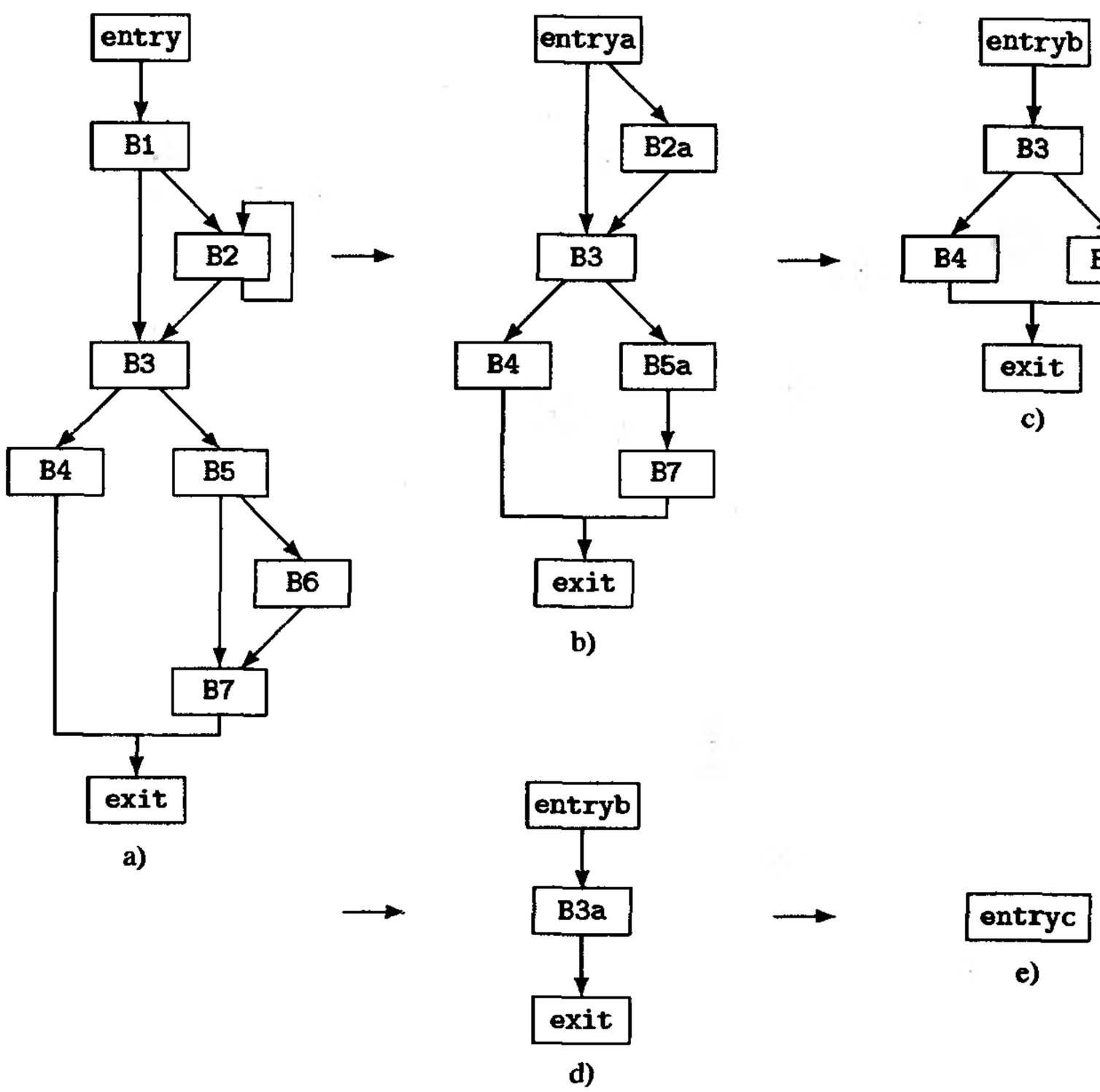


图7-46 流图的结构分析

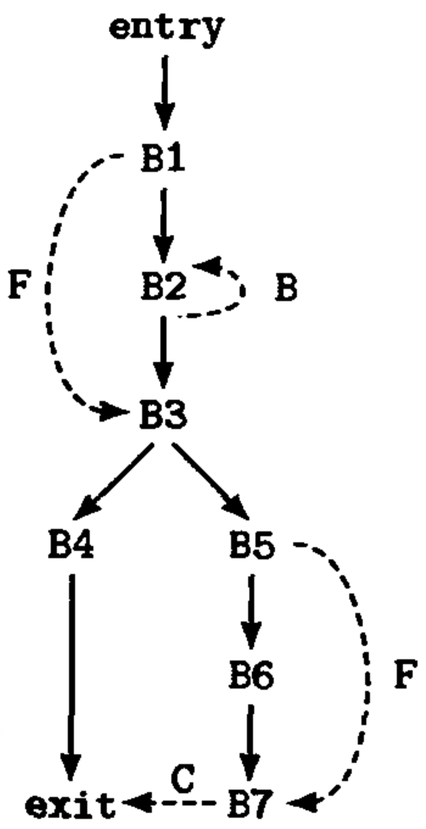


图7-47 图7-46a中流图的深度为主生成树

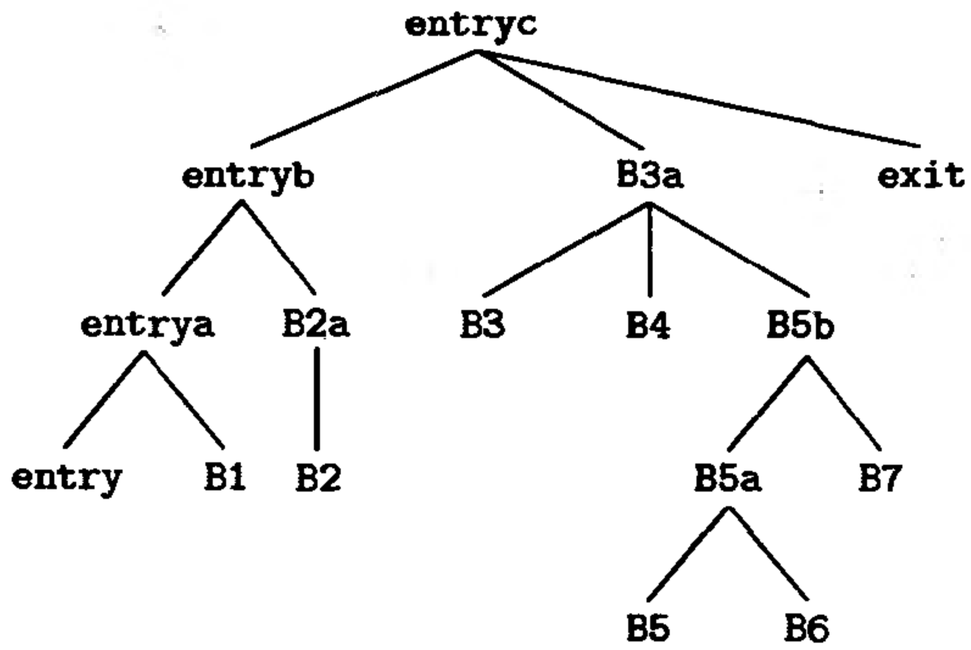


图7-48 图7-46中所分析的流图的控制树

7.8 小结

控制流分析是我们第一次接触到的与优化直接有关的重要内容。

优化要求我们能够刻画程序控制流以及它们对数据执行的操作所具有的特征，以便编译器能够删除任何无用的生成代码，并使用更高效的操作。

如前面所讨论的，有两种主要的控制流分析方法，它们都从确定构成过程的基本块并形成

bbs.theithome.com

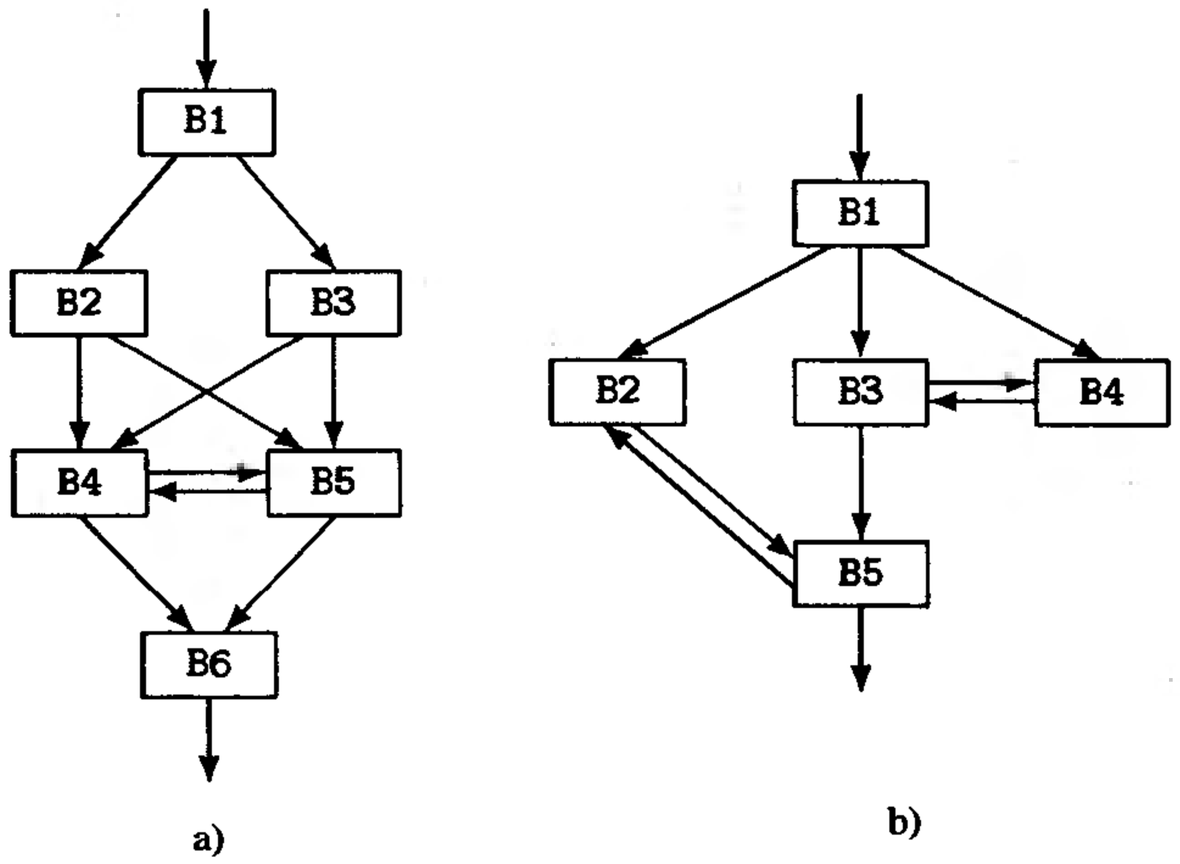


图7-49 两个非正常区间的例子

其流图开始。第一种方法使用必经结点来发现循环，并简单地标识它找到的循环给优化使用。我们也为那些能施加于其上的优化标识了扩展基本块和反扩展基本块。这种方法足以应付迭代数据流分析的需要。

第二种处理方法称为区间分析，它包括一系列的算法，这些算法分析整个过程的结构，并将过程分解成称为区间的嵌套区域。区间的嵌套结构形成了控制树，控制树有助于数据流分析的结构化和提高其分析速度。最成熟的一种区间分析形式称为结构分析，它的实质是对过程内的所有控制结构进行分类。由于它特别重要，因此我们用了一节的篇幅来讨论它。

过去，多数优化编译器使用的是必经结点和迭代数据流分析。但是，这种情况正在发生改变，因为基于区间的方法更快，且能更方便地更新已经计算出来的数据流信息，并且采用结构分析方法特别容易实现第18章讨论的控制流转换。

7.9 进一步阅读

[LenT79]中描述了Lengauer和Tarjan的计算必经结点的方法。它使用的路径压缩算法在[Tarj81]中有更为完整的描述。Tarjan的查找强连通分量的算法在[Tarj72]中给出。Alstrup和Lauridsen的必经结点树的最新算法描述见[AlsL96]。

在[Kenn81]中能找到关于各种流图转换的综述，而且关于流图归约性的术语也起源于这篇论文。关于Fortran程序中的可归约性的研究来源于Allen和Cocke [AllC72b]，以及Knuth [Knut71]。*T*1-*T*2分析的描述见[Ullm73]。极大区间的定义源于Allen和Cocke [AllC76]；Aho、Sethi和Ullman [AhoS86]也使用了极大区间；这两篇论文都给出了归约流图到极大区间的算法。

213
↓
214

最早将结构分析公式化的是Sharir [Shar80]。基于语法树的数据流分析方法应归于Rosen(参见[Rose77]和[Rose79])，Schwartz和Sharir [SchS79]对这种方法做了扩充。关于分析非正常区间并使之极小化的现代方法的讨论见[JaiT88]，但是正如论文中讨论的那样，这种方法是有缺陷的——它特意定义非正常区域有一个给定的结点作为其必经结点，这导致了这种非正常区域只包含那个结点。

7.10 练习

7.1 对含有`setjmp()`调用的C函数，指明对该函数的流图必须增加的边集合。

bbs.theithome.com

超星公司制作 请尊重作者版权

7.2 (a)将图7-16中的ICAN过程Domin_Fast()划分为基本块。你会发现这有助于构造它的流图。(b)然后将它划分为扩展基本块。

7.3 为图7-12中例程Depth_First_Search_PP()的流程图结点构造其(a)深度为主表示，(b)深度为主次序，(c)前序次序和(d)后序次序。

7.4 假设对于流图中每一对结点a和b，a dom b当且仅当b pdom a。该流图的结构是什么？

7.5 用你可以使用的语言实现Dom_Comp()、Idom_Comp()和Domin_Fast()，并且对图7-32中的流图运行它们。

7.6 解释图7-17中的过程Compress()所做的工作。

7.7 解释图7-18中的过程Link()所做的工作。

7.8 对图7-50应用图7-26中的算法Strong_Components()。

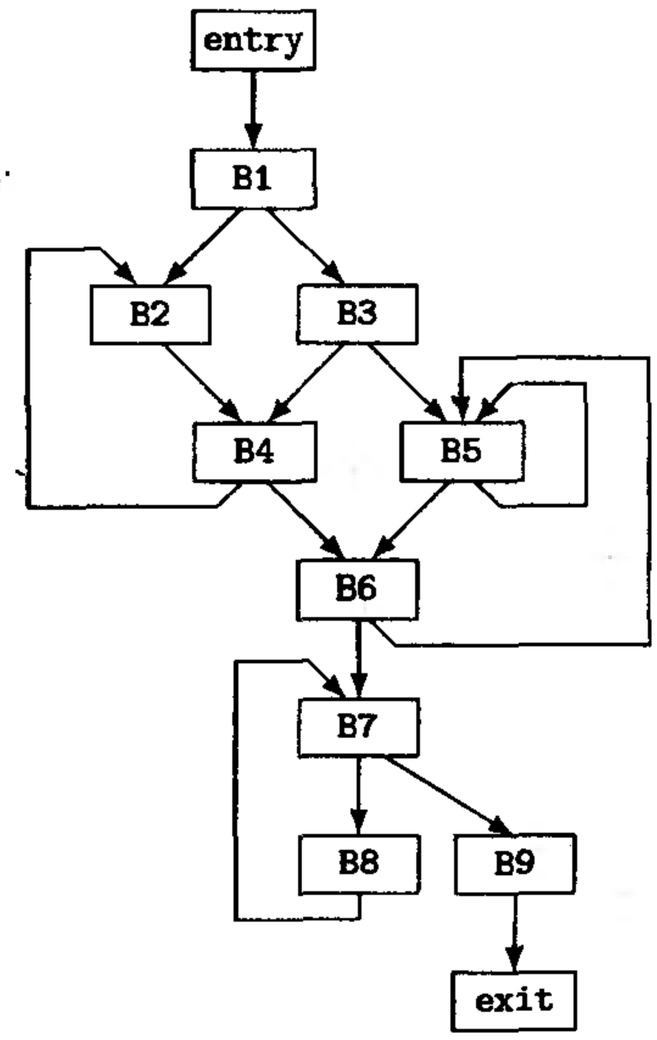


图7-50 要对其应用图7-26中算法Strong_Components()的一个例图

7.9 定义一个由不同的非正常区域R_1，R_2，R_3，…组成的无穷序列，其中每一个R_i由结点集合N_i和边集合E_i组成。

7.10 给出一个非可归约区域R_1，R_2，R_3，…组成的无穷序列，其中R_i由i个结点组成，并且对R_i执行结点分割将产生一个其结点个数是i的指数的流图。

7.11 编写一个计算可归约流图中极大区间的ICAN程序。

7.12 编写一个计算可归约流图中极小区间的ICAN程序。

RSCH 7.13 阅读Rosen的论文[Rose77]和[Rose79]，并根据他的方法构造出if-then-else结构和repeat循环的公式。

7.14 写出图7-35中case/switch模式的形式化说明。

7.15 写出自然循环集合（参见图7-36）的形式化说明，其中定义自然循环为单入口多出口循环，并且其内只有一个返回到入口的分支。

7.16 对图16-7中的Make_Webs()和图16-24中的Gen_Spill_Code()例程执行结构控制流分析。

215 ~ 216

ADV 7.17 用ICAN实现由Minimize_Improper()使用的函数MEC_Entries()。

bbs.theithome.com

越是公司制作 请尊重作者版权

第8章　数据流分析

数据流分析的目的是提供一个过程（或一大段程序）如何操纵其数据的全局信息。例如，常数传播分析力求判定对一个特定变量的所有赋值在某个特定点是否总是给定相同的常数值，如果是这样，则在那一点可用一个常数来替换该变量。

数据流分析的范围很广，可以从分析一个过程的抽象执行（例如，判定一个过程是计算阶乘的函数，参见8.14节），到比较简单和比较容易的分析（如下一节将讨论的到达-定值问题）。

在任何情况下，我们都必须保证数据流分析给出的信息没有误解被分析过程的行为，即它不能告诉我们执行某种代码转换是安全的，而实际却是不安全的。我们必须仔细地设计数据流方程，并确信我们计算出来的解即使不是过程处理其数据的精确表示，也至少是它的保守近似表示，以此来保证这一点。例如，对于到达－定值问题——在该问题中我们判定变量的哪些定值可以到达一个特定的使用——如果存在到达某个特定使用的定值时，数据流分析就一定不能告诉我们没有定值到达这个使用。如果数据流分析给出的到达－定值集合可能大于过程实际产生的最小到达－定值集合，则这种分析是保守的。

但是，为了从优化中获得最大的效益，我们力求使数据流分析问题既是保守的同时又尽可能是激进的。因此，我们将总是企图在使所计算的信息尽可能是激进的，同时又是保守的之间走钢丝，以便从我们执行的这些分析和代码改善的转换中得到最大的效益，而又不至于把正确的代码转换成不正确的代码。

最后，请回顾一下7.1节讨论的三种数据流和控制流分析方法，以及在本书中对这三种方法都作介绍的原因，这样能够使你对为什么我们决定对三种方法都予以介绍有更深的理解。

8.1　一个例子：到达-定值

作为数据流分析的入门，我们对第7章开始的那个非形式化例子继续执行一种简单的称为到达-定值的数据流分析，我们从图7-3和图7-4的流图开始，分析在那里给出的计算斐波那契数列的过程。

一个定值是对某个变量的赋值。称一个变量的某个特定定值到达过程的一个给定点，前提是如果存在从这个定值到那一点的一条执行路径，使得这个变量在那一点可能具有由该定值赋予的值。我们的目的是确定每一个变量的特定定值（即赋值）中，有哪一些可能通过某条控制流路径到达过程内的任意一个特定点。我们用术语控制流路径（control-flow path）表示过程的流程图中的任意有向路径，通常不管该路径上控制分支走向的谓词是否满足。

我们虽然可以对过程的流程图执行数据流分析，但通常更有效的做法是将它分解为局部数据流分析和全局数据流分析，局部数据流分析针对每一个基本块进行，全局数据流分析针对流图进行。为此，我们归纳每一个基本块的数据流效用来生成局部信息，并将它们用于全局数据流分析，由此产生与每一个基本块的入口和出口对应的信息。然后将得到的全局信息与局部信息结合起来，以生成到达任意基本块内每一个中间语言结构开始点的定值集合。这样做的效果是，它常常显著地减少了计算数据流信息所需要的步骤，但通常有少量的代价，即需要从基本块的开始（或结尾）传播这个信息到基本块中需要使用它的地方。

bbs.theithome.com

超星公司制作 请尊重作者版权

同样地，我们考虑的多数数据流分析问题关注的是各种程序对象（常数、变量、定值、表达式等）的集合，以及在过程内任意一点这些对象的什么集合是合法的有关判断。至于是何种对象及合法性的含义则取决于具体的问题。在下面的几段，我们用两种方式给出到达－定值问题的形式化表示，一种表示使用集合，另一种表示使用位向量。位向量是一种便于计算机使用的集合表示，因为集合的并、交和补运算直接与位向量的按位或、与和非运算相对应。

到达－定值分析可用称作向前迭代位向量问题的经典形式来进行。“迭代”是因为我们构造了一组表示信息流的数据流方程，并从适当的初值集合开始，以迭代方式来求解这组方程；“向前”是因为信息流是沿着程序中控制流边的执行方向流动的；而“位向量”是因为我们可以用一个1（意味着它可能到达给定点）或0（意味着它不能到达）来表示一个定值，因此，可能到达一个点的所有定值的集合可以用一个位串或位向量来表示。

218

一般而言，对于我们涉及的大多数数据流分析问题，一个定值是否实际到达另外的某点是递归不可判定的(recursively undecidable)。另外，一个定值是否到达一个特定点也可能与输入数据相关。例如，图8-1C代码第4行的说明语句中，i的定值是否实际到达第7和第8行的使用依赖于函数f()的参数n，而第7行j的定值是否实际到达第10行return中的使用，依赖于while循环是否会终止，而这一般也是递归不可判定的。于是，我们需要区分两种情况：一种是我们能够确定为假的情况，另一种是或者知道为真或者不能确定的情况。因为基于到达－定值的优化依赖于可能为真的情况，因此我们遵循保守的做法，优先保证程序的正确性，其次才是激进的优化。

```
1   int g(int m, int i);

2   int f(n)
3       int n;
4   {   int i = 0, j;
5       if (n == 1) i = 2;
6       while (n > 0) {
7           j = i + 1;
8           n = g(n,i);
9       }
10      return j;
11  }
```

图8-1 到达－定值的不可判定性以及依赖于输入数据的例子

表8-1给出了图7-3流程图中的定值和它们在位向量中的位置的对应关系。于是，可用8位的向量来表示程序中有哪些定值到达了每一个基本块的开始。

表8-1 图7-3流程图的位向量的位置、定值及基本块之间的对应关系

位的位置	定　值	基本块
1	m在结点1	B1
2	f0在结点2	
3	f1在结点3	
4	i在结点5	B3
5	f2在结点8	B6
6	f0在结点9	
7	f1在结点10	
8	i在结点11	

显然，对于entry结点恰当的初始条件是没有任何定值到达它，因此，entry入口处合法的定值集合是：

$$RCHin(\mathtt{entry}) = \emptyset$$

或者，表示成8位的向量为：

$$RCHin(\mathtt{entry}) = \langle 00000000 \rangle$$

进一步，因为我们试图确定哪些定值可能到达一个特定点，故保守的（但不是必须的)假设是，对所有的i有如下的初始值：

bbs.theithome.com

越是公司制作请尊重作者版权

$RCHin(i) = \emptyset$

或

$RCHin(i) = <00000000>$

现在我们必须弄清楚流图中每一个结点对位向量中的每一位具有什么影响。如果一条MIR 219
指令重新定值了由给定位位置表示的那个变量，则称它杀死（kill）了该定值，否则称保留（preserve）了该定值。这使我们想到定义一个集合（以及相应的位向量），称为*PRSV*(*i*)，它表示由基本块*i*所保留的那些定值。容易看出，作为集合（用位向量的位置表示定值），有，

$PRSV(B1) = \{4, 5, 8\}$

$PRSV(B3) = \{1, 2, 3, 5, 6, 7\}$

$PRSV(B6) = \{1\}$

$PRSV(i) = \{1, 2, 3, 4, 5, 6, 7, 8\}$ 其中 $i \neq B1, B3, B6$

作为位向量则为（从左端数起），

$PRSV(B1) = (00011001)$

$PRSV(B3) = (11101110)$

$PRSV(B6) = (10000000)$

$PRSV(i) = (11111111)$ 其中 $i \neq B1, B3, B6$

例如，*PRSV*(B1)第7位的0指出基本块B1杀死了结点10中f1的定值[⊖]，而在*PRSV*(B1)第5位的1指出B1没有杀死结点8中f2的定值。有些书，如[AhoS86]，使用*KILL*()——与*PRSV*()相反的集合——而不是*PRSV*()。 220

对应地，我们定义集合和位向量*GEN*(*i*)，它给出由基本块*i*生成的定值集合[⊜]，即，在基本块中有过赋值且随后未在基本块中被杀死的那些定值组成的集合。作为集合，*GEN*()的值是

$GEN(B1) = \{1, 2, 3\}$

$GEN(B3) = \{4\}$

$GEN(B6) = \{5, 6, 7, 8\}$

$GEN(i) = \emptyset$ 其中 $i \neq B1, B3, B6$

而作为位向量，它们是

$GEN(B1) = (11100000)$

$GEN(B3) = (00010000)$

$GEN(B6) = (00001111)$

$GEN(i) = (00000000)$ 其中 $i \neq B1, B3, B6$

最后，我们定义表示到达基本块*i*末尾的那些定值的集合和相应的位向量*RCHout*(*i*)。同*RCHin*(*i*)一样，对所有的*i*，初始化

$RCHout(i) = \emptyset$

或

$RCHout(i) = <00000000>$

⊖ 事实上，因为控制无法从包含结点10的基本块（即基本块B6）流向基本块B1，因此，我们不必使这一位为0，但这样做肯定没有坏处。

⊜ 此时我们忽略了这一事实，即一个变量的某些定值是无二义的，例如显式赋值；而其他的定值，如通过指针的赋值和过程调用，就它们对特定的变量可能有、也可能没有影响，并且我们不能确定它们是否有影响的意义而言，是有二义的。这个过程中没有出现有二义的定值。

bbs.theithome.com

就足够了[⊖]。

现在，一个定值可以到达基本块i的末尾，当且仅当它要么在基本块i内出现，并且它定值的变量在i中没有被重新定值；要么它可能到达i的开始并且由i保留其定值。这个表述也可以利用集合和集合运算表示为:

对于所有的i，

$$RCHout(i) = GEN(i) \cup (RCHin(i) \cap PRSV(i))$$

或者，利用对位向量的位逻辑运算表示为:

对于所有的i，

$$RCHout(i) = GEN(i) \vee (RCHin(i) \wedge PRSV(i))$$

一个定值可能到达基本块i的开始，条件是如果它可以到达i的某个前驱的末尾，即，在集合的
221 情形下，对所有的i，

$$RCHin(i) = \bigcup_{j \in Pred(i)} RCHout(j)$$

或者，在位向量的情形下，对所有的i，

$$RCHin(i) = \bigvee_{j \in Pred(i)} RCHout(j)$$

为了求解$RCHin(i)$和$RCHout(i)$（位向量）方程组，我们简单地将$RCHin(i)$初始化为前面给出的值，并且迭代地应用这些方程直到没有进一步的变化产生。为了理解迭代为什么能产生该方程组的可接受的解，我们将在下一节对格和不动点迭代给出简要介绍。在应用了这些方程一次后，我们得到

$RCHout$(entry)	= ⟨00000000⟩	$RCHin$(entry)	= ⟨00000000⟩
$RCHout$(B1)	= ⟨11100000⟩	$RCHin$(B1)	= ⟨00000000⟩
$RCHout$(B2)	= ⟨11100000⟩	$RCHin$(B2)	= ⟨11100000⟩
$RCHout$(B3)	= ⟨11110000⟩	$RCHin$(B3)	= ⟨11100000⟩
$RCHout$(B4)	= ⟨11110000⟩	$RCHin$(B4)	= ⟨11110000⟩
$RCHout$(B5)	= ⟨11110000⟩	$RCHin$(B5)	= ⟨11110000⟩
$RCHout$(B6)	= ⟨10001111⟩	$RCHin$(B6)	= ⟨11110000⟩
$RCHout$(exit)	= ⟨11110000⟩	$RCHin$(exit)	= ⟨11110000⟩

再迭代一次后，我们有

$RCHout$(entry)	= ⟨00000000⟩	$RCHin$(entry)	= ⟨00000000⟩
$RCHout$(B1)	= ⟨11100000⟩	$RCHin$(B1)	= ⟨00000000⟩
$RCHout$(B2)	= ⟨11100000⟩	$RCHin$(B2)	= ⟨11100000⟩
$RCHout$(B3)	= ⟨11110000⟩	$RCHin$(B3)	= ⟨11100000⟩
$RCHout$(B4)	= ⟨11111111⟩	$RCHin$(B4)	= ⟨11111111⟩
$RCHout$(B5)	= ⟨11111111⟩	$RCHin$(B5)	= ⟨11111111⟩
$RCHout$(B6)	= ⟨10001111⟩	$RCHin$(B6)	= ⟨11111111⟩
$RCHout$(exit)	= ⟨11111111⟩	$RCHin$(exit)	= ⟨11111111⟩

并且再继续迭代不会发生改变，因此，上面的值就是解。

⊖ 我们将看到，这实际上是不必要的，因为每一个$RCHout(i)$是从同一个基本块的$RCHin(i)$、$GEN(i)$和$PRSV(i)$计算出来的，我们确实完全无需初始化$RCHout()$的值。

bbs.theithome.com

注意，执行迭代的规则决不会将1变为0，即，它们是单调的，因此，我们能保证迭代过程最终确实会终止。

通过这个数据流方程的解，使我们对变量的哪些定值可以到达哪些使用有了全局性的了解。例如，它说明在基本块B1中对f0的定值可以到达基本块B6中f0的第一次使用，并且在沿着通过基本块B2的执行路径上，变量i和f2从未被定值。利用这些信息来优化程序的一种方法是在沿着经过B2的路径上避免给i和f2分配存储(和寄存器)。

注意，尽管上面提出的数据流方程容易理解，但在理论上实际没有理由需要同时使用*RCHin*()和*RCHout*()两个函数。作为替代，我们可以将关于*RCHout*()的方程代入到关于*RCHin*()的方程中而得到更简单的方程组: 222

对于所有的i,

$$RCHin(i) = \bigcup_{j \in Pred(i)} (GEN(j) \cup (RCHin(j) \cap PRSV(j)))$$

或者，对于所有的i

$$RCHin(i) = \bigvee_{j \in Pred(i)} (GEN(j) \vee (RCHin(j) \wedge PRSV(j)))$$

并且它们与原方程组具有完全一样的解。但是实际中，在选择是同时使用*RCHin*()和*RCHout*()函数，还是只使用其中之一时会有所权衡。如果我们同时使用两者，则需要双倍的空间来表示数据流信息，但却减少了所需的计算量；因为如果只使用*RCHin*()函数，我们需要重复地计算

$$GEN(j) \cup (RCHin(j) \cap PRSV(j))$$

即使*RCHin*(*j*)没有变化也如此。而如果同时使用*RCHin*()和*RCHout*()，我们能立即得到其值。

8.2 基本概念：格、流函数和不动点

现在我们来定义数据流分析所依赖的一些基本概念。在每一种情况中，数据流分析都是通过对一种称为格的代数结构的元素进行运算来完成的。格的元素代表变量、表达式，或一个过程所有可能执行的其他程序设计结构的抽象性质——这种性质与输入数据值无关，通常也与过程流经的控制流路径无关。特别地，多数数据流分析都不关心程序执行条件的真假，即不关心if是取then分支还是else分支，也不关心循环的执行次数。我们将过程中每一种可能的控制流和计算结构与一个所谓的流函数关联起来，这个流函数将每个结构的作用抽象成对应的格元素的作用。

一般而言，格L由值集合和两个运算组成，其中一个运算称为交，记作⊓，另一个运算称为并，记作⊔，并且满足下面若干性质:

1. 对于所有$x, y \in \mathbf{L}$，存在着惟一的z和$w \in \mathbf{L}$，使得$x \sqcap y = z$并且$x \sqcup y = w$(封闭性)。
2. 对于所有$x, y \in \mathbf{L}$，$x \sqcap y = y \sqcap x$并且$x \sqcup y = y \sqcup x$(交换性)。
3. 对于所有$x, y, z \in \mathbf{L}$，$(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$并且$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$(结合性)。
4. L中存在两个惟一的元素，一个称为底（bottom），记做⊥，另一个称为顶（top），记做⊤，使得对所有$x \in \mathbf{L}$，$x \sqcap \bot = \bot$并且$x \sqcup \top = \top$（存在惟一的顶元素和底元素）。 223

许多格，包括我们使用的除用于常数传播（参见12.6节）的格之外的所有格，都是可分配的，即，对所有的$x, y, z \in \mathbf{L}$，

$$(x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \text{并且} (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$$

bbs.theithome.com

我们使用的大部分格以位向量作为其元素，它的并和交运算分别是“按位与”和“按位或”运算。这种格的底元素是每一位都为0的位向量，顶元素是每一位都为1的位向量。我们用$\mathbf{BV}^n$表示长度为n的位向量的格。例如8.1节的例子中，8位的位向量形成了一个格，其中，$\perp$ = <00000000>，$\top$ = <11111111>。两个位向量的并是一个位向量，如果这两个位向量对应位中有一个为1，则此位向量相应的位是1，否则是0。例如，

<00101111> ⊔ <01100001> = <01101111>

与到达-定值例子中的那个格类似的一个格如图8-2所示，不过，它的位向量只有3位。

有若干种方法可通过组合简单的格而构造出另外的格。一种方法是乘积运算，它按元素来组合两个格。两个交运算分别为$\sqcap_1$和$\sqcap_2$的格$\mathbf{L}_1$和$\mathbf{L}_2$的乘积（product），记做$\mathbf{L}_1 \times \mathbf{L}_2$，是$\{<x_1, x_2> \mid x_1 \in \mathbf{L}_1,\ x_2 \in \mathbf{L}_2\}$，它的交运算符的定义为：

$$<x_1, x_2> \sqcap <y_1, y_2> = <x_1 \sqcap_1 y_1,\ x_2 \sqcap_2 y_2>$$

并运算符的定义也类似。乘积运算可自然地推广到两个以上的格，具体地讲，我们前面提及的那个格$\mathbf{BV}^n$就是n个只含底元素0和顶元素1的格$\mathbf{BV} = \mathbf{BV}^1 = \{0, 1\}$的乘积。

在有些情况下，用位向量来表示所需的信息是不受欢迎的，或不够的。整数值的常数传播就是不希望使用位向量的一个简单例子。对于整数值常数传播，我们使用的格以图8-3所示的格为基础，这种格称为**ICP**，它的元素是$\perp$、$\top$、所有整数及所有布尔量，并且具有下面定义的性质：

1. 对所有的$n \in \mathbf{ICP}$，$n \sqcap \perp = \perp$。

224 2. 对所有的$n \in \mathbf{ICP}$，$n \sqcup \top = \top$。

3. 对所有的$n \in \mathbf{ICP}$，$n \sqcap n = n \sqcup n = n$。

4. 对所有的整数和布尔量$m, n \in \mathbf{ICP}$，如果$m \neq n$，则$m \sqcap n = \perp$并且$m \sqcup n = \top$。

在图8-3中，两个元素的交是从这两个元素开始沿线往下直到它们相交点而得到的值，而两个元素的并是从这两个元素开始沿线往上直到它们相遇而得到的值。

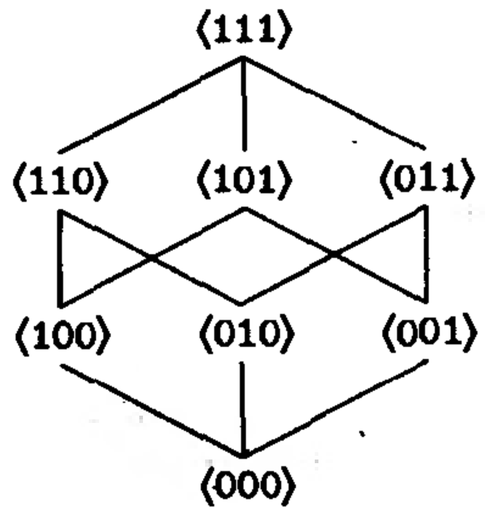


图8-2 一个三元素位向量的格

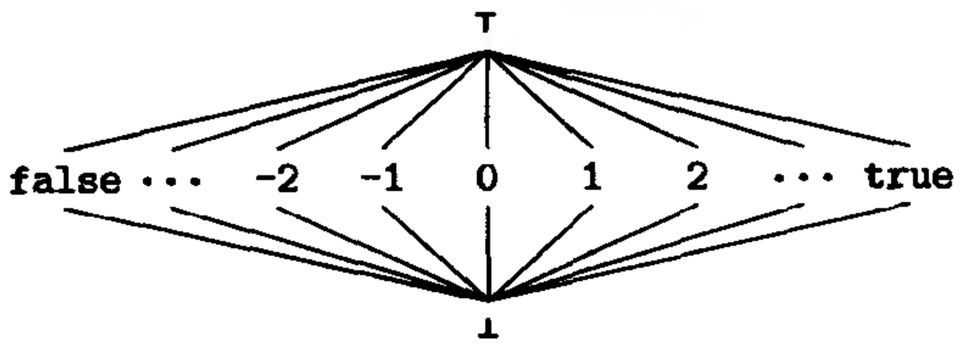


图8-3 整数常数传播格ICP

位向量可按下面所述方式用于常数传播。定义*Var*是感兴趣的变量名的集合，令**Z**表示整数集合。从*Var*的有限子集到**Z**的函数集合代表了对程序中用到的变量的每一种可能的常数赋值。每一个这种函数可表示成一个无穷的位向量，对某个变量$v \in Var$、常数$c \in \mathbf{Z}$组成的每一对$<v, c>$，这个位向量相应地有一位置；其中，如果v具有值c，则此位置为1，否则为0。这种无穷位向量的集合在位向量的一般格顺序下形成了一个格。显然这个格比**ICP**要复杂得多：它的元素是无穷位向量，不仅像**ICP**那样宽度是无穷的，它的高度也是无穷的，这导致不宜用公式来表示它们。

有些数据流分析问题需要比位向量复杂得多的格，例如Jones和Muchnick [JonM81a]用于描述类LISP数据结构的“形状”的两种格，其中一种格的元素是正则树文法，另一种格的元素是复杂的图。

bbs.theithome.com

从格的图形表示能清楚地看出，交和并运算符导致了值之间的一种偏序关系，记做⊑。这个偏序关系可利用交运算定义为:

$x \sqsubseteq y$，当且仅当$x \sqcap y = x$

或者，它也可以对偶地利用并运算来定义。对应地也可以定义相关的运算⊏、⊐和⊒。很容易从交和并的定义导出⊑的下述性质（以及其他序关系的性质）:

1. 对所有的x, y, z，如果$x \sqsubseteq y$并且$y \sqsubseteq z$，则$x \sqsubseteq z$(传递性)。
2. 对所有的x, y, 如果$x \sqsubseteq y$并且$y \sqsubseteq x$，则$x = y$(反对称性)。
3. 对所有的x, 有$x \sqsubseteq x$(自反性)。

映射格到这个格自身的函数，记做f: $\mathbf{L} \to \mathbf{L}$，如果对所有$x$、$y$，有$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$，则称此函数是单调的(monotone)。例如，由下式定义的函数f: $\mathbf{BV}^3 \to \mathbf{BV}^3$是单调的，

$f(<x_1\, x_2\, x_3>) = <x_1\, 1\, x_3>$

而由$g(<000>)=<100>$和$g(<x_1\, x_2\, x_3>)=<000>$定义的函数$g$: $\mathbf{BV}^3 \to \mathbf{BV}^3$则不是单调的。

格的高度（height）是其最长的严格上升链的长度，即，使得存在$x_1, x_2, \cdots, x_n$，导致

$\bot = x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_n = \top$

的最大的n。例如，图8-2和图8-3中两个格的高度分别是4和3。同其他与格有关的概念一样，高度也可以对偶地用下降链来定义。我们使用的几乎所有的格都具有有限的高度，正是由于这一点，再加上单调性，便保证了我们的数据流分析算法能够终止。对于无限高度的格，必须要证明分析算法会停止。

在考虑数据流算法的计算复杂性时有另一种重要的表示，即相对一个和多个函数的有效高度。格L相对函数f: $\mathbf{L} \to \mathbf{L}$的有效高度（effective height）是通过迭代地应用f()而获得的最长的严格上升链的长度，即，使得存在$x_1, x_2 = f(x_1), x_3 = f(x_2), \cdots, x_n = f(x_{n-1})$，导致

$x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset \cdots \sqsubset x_n \sqsubseteq \top$

的最大的n。一个格相对于函数集合的有效高度是每一个函数的有效高度中的最大值。

对于特定数据流分析问题，流函数（flow function）用从分析中使用的格到这个格自身的映射来模拟程序设计语言结构的作用。例如，8.1节的到达-定值分析中关于基本块B1的流函数是函数F_{B1}: $\mathbf{BV}^8 \to \mathbf{BV}^8$，它由下式给出:

$F_{B1}(<x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8>) = <111 x_4\, x_5\, 00 x_8>$

我们要求所有流函数都是单调的。这样要求的道理是，流函数的目的是模拟程序设计结构提供的有关数据流问题的信息，因此，它不应当减少已经获得的信息。另外，对于证明我们所考虑的每一种分析过程会停止，以及对于提供计算复杂性的上界而言，单调性也是必不可少的。

由具体的流函数所模拟的程序设计结构可以根据我们的需要而变化，从单一的表达式到整个过程。因此，8.1节例子中转换每一个*RCHin*(*i*)到*RCHout*(*i*)的函数可以看成是一个流函数，转换整个*RCHin*(*i*)集合到*RCHout*(*i*)集合的函数也可以看成是一个流函数。

函数f: $\mathbf{L} \to \mathbf{L}$的不动点（fixed point）是一个元素$z \in \mathbf{L}$，它使得$f(z) = z$。对于数据流方程组，不动点是此方程组的解，因为将不动点代入方程组的右端产生相同的值。在许多情况下，定义在格上的函数可以有多个不动点。这种情况最简单的一个例子是函数f: $\mathbf{BV} \to \mathbf{BV}$，它只有$f(0) = 0$和$f(1) = 1$。显然，这个函数的0和1都是不动点。

我们在解数据流方程时想要计算的值是所谓的满足全路径（MOP，meet-over-all-path）的解。直观地，这个解可通过如下方法产生：在流图入口结点(或对于向后流问题为出口结点)从具有某些规定的信息*Init*开始，沿入口结点（或出口结点）到流图中每一个结点的所有可能路

bbs.theithome.com

径应用适当的流函数的合成，并对每一个结点得到的结果做交运算。用方程来表示，关于向前流问题我们有如下表述。令$G=<N, E>$是一个流图。令$Path(B)$表示从entry到任意结点$B \in N$的所有路径的集合，令p是$Path(B)$中的任意元素。令$F_B()$表示流经基本块B的函数，$F_p()$表示沿着路径p遇到的流函数的合成，即，如果$B_1=$entry，…，$B_n=B$是组成到B的特定路径p的基本块，则

$$F_p = F_{Bn} \circ \cdots \circ F_{B1}$$

令$Init$是与entry基本块相连的格值，则满足全路径的解是

$$MOP(B) = \bigsqcap_{p \in Path(B)} F_p(Init), \text{其中 } B = \texttt{entry}, \texttt{B1}, \cdots \texttt{Bn}, \texttt{exit}$$

类似的方程可表示向后流问题的满足全路径的解。

不幸的是不难证明，对于那种只保证流函数是单调的任意数据流问题，可能不存在计算关于所有可能流图的满足全路径的解的算法。我们的算法计算的是所谓的最大不动点（MFP，maximum fixed point）解，它简单地说就是这些数据流方程的解在底层格顺序中的最大值，即提供最多信息的解。对于其中所有流函数都是可分配的数据流问题，Kildall [Kild73]证明了我们在8.4节给出的普通迭代算法计算出的是MFP解，并且在那种情况下，MFP和MOP解是相同的。Kam和Ullman [KamU75]推广了这一结论以证明对于其中流函数都是单调的，但不必是可分配的数据流问题，此迭代算法产生MFP解(但不一定是MOP解)。

在继续我们感兴趣的各种数据流问题以及如何解它们的讨论之前，我们先花一点时间讨论将数据流信息与流图的基本块入口点相连，而不是与边相连的问题。前者是多数文献和我们所知道的编译器的标准做法。但是，有少数论文将数据流信息与流图的边相连。这样做的效果是，在某些情况下能产生更好的信息，主要是因为对于具有多个前驱的结点(或对向后流问题而言，具有多个后继的结点)，它不在进入这个结点之前强制合并信息。

这种做法产生较好的信息的一个简单例子是图8-4所示的常数传播例子。显然，B4中赋给w的值是常数3。无
227 论是将信息与结点入口还是与边相连，我们都知道在从B2和B3出来时，u和v都具有常数值。如果我们做常数传播并将数据流信息与边相连，则保持了这一事实，即在从B2到B4的边上，u的值为1而v的值为2；在从B3到B4的边上，u的值为2而v的值为1。这允许B4的流函数结合这些不同的值来判定在B4中赋给w的是常数值3。但是，如果我们将数据流信息与结点入口相连，则我们在B4入口处所知道的是u的值和v的值都不是常数（在两种情况下，值是1或者2，但格ICP没有提供从⊤区分信息的途径，而且即使提供了途径，它也不足以使我们能判定出w的值是常数）。

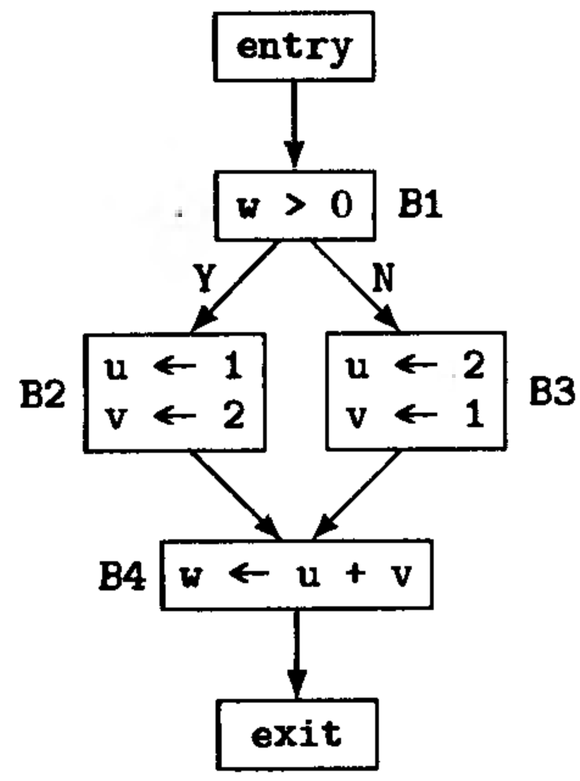


图8-4 将数据流信息与边相连比与结点入口相连能产生更好结果的流图

8.3 数据流问题及其解决方法的分类

数据流分析问题可按若干种尺度来归类，这些尺度包括：

1. 它们所提供的信息；
2. 它们是有关系的还是属性无关的；

bbs.theithome.com

越是公司制作请尊重作者版权

3. 它们使用的格的类型，以及格元素所代表的含义和在这些格上定义的函数；

4. 信息流的方向：按程序执行的方向（向前问题）、程序执行的反方向（向后问题），还是同时两个方向（双向问题）。

228

几乎我们考虑的所有例子都是属性无关(independent-attribute)的类型，即，它们对感兴趣的每一个对象指定一个格元素，这个对象可以是变量定值，表达式计算，或其他的任何东西。只有少数的，如8.14节描述的结构类型判定问题，要求在每一点用一个关系表示过程的数据流状态，这个关系描述变量值(或某种类似事情)之间的关系。关系问题比属性无关问题的计算要复杂得多。

类似地，我们考虑的几乎所有问题都是一个方向的，或者是向前的（forward），或者是向后的（backward）。双向（bidirectional）问题要求同时向前和向后传播，并且一般情况下将其公式化和求解都比单向问题要复杂。幸运的是，在优化中双向问题非常少见。最重要的双向问题的实例是部分冗余消除的古典公式表示（在13.3节提及），即使它已被13.3节介绍的更为现代的只使用单向分析的形式所替代。

下面叙述的是一些与程序优化有关的最重要的数据流分析问题。在每一种情况下，当第一次涉及到需要使用这种数据流分析的优化时，我们将给出关于优化问题及其方程的更完整描述。

到达-定值（reaching definition）

到达-定值问题确定过程中一个变量的哪些定值（即对它的赋值）可以到达该变量的各个使用。如我们已看到的，到达-定值是使用位向量的格的向前问题，其中，位向量的每一位对应变量的一个定值。

可用表达式（available expression）

可用表达式问题确定在过程的每个点上哪些表达式是可用的。在某点可用的含义是指：从入口到该点的每一条路径上存在着该表达式的一个计算，并且在此路径上的这个计算之后到该点之间，出现在该表达式中的所有变量都没有被重新赋值。可用表达式是位向量上的格的向前问题，其中，位向量的每一位对应表达式的一个定值。

活跃变量（live variable）

对于给定的变量和程序中给定的点，活跃变量问题确定沿着此点到出口的路径上是否存在对该变量的使用。这是一个使用位向量的向后问题，其中，变量的每一个使用在位向量中有一个位置。

向上暴露使用（upwards exposed use）

此问题确定在特定点上哪些变量的使用可以由特定的定值而到达。这是一种使用位向量的向后问题，一个变量的每一个使用在位向量中相应地有一位。它是到达-定值的对偶问题，到达-定值连接定值到使用，而向上暴露使用连接使用到定值。注意这两者是不同的，如图8-5所示，其中x在B2中的定值到达B4和B5的使用，而在B5中的使用是由B2和B3中的定值所到达的。

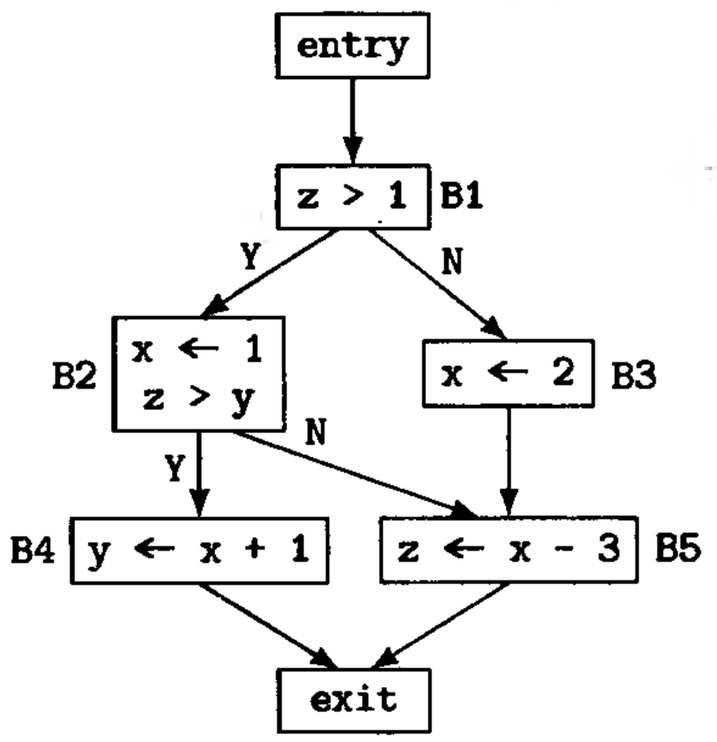


图8-5 说明到达－定值与向上暴露使用之间不同的例子

229

复写传播分析（copy-propagation analysis）

复写传播分析确定从一个复写赋值，比如说$x \leftarrow y$，到变量x的一个使用的每一条路径上都不存在对y的赋值。这是一个使用位向量的向前问题，位向量的每一位表示一个复写赋值。

bbs.theithome.com

超星公司制作 请尊重作者版权

常数传播分析（constant-propagation analysis）

常数传播分析确定从对一个变量的常数赋值，比如说$x \leftarrow const$，到x的一个使用的每一条路径上，对x的赋值都只是此常数值$const$。这是一个向前流问题。在这个问题的古典形式中，它使用每个变量有一个位置的向量，使用的值取自8.2节讨论的格ICP"，或者取自元素选自其他适当数据类型的类似的格。在这个问题的稀有条件形式中（参见12.6节），它使用一种类似的格，但每个定值有一个格值，并且是符号执行，而不是简单的数据流分析。

部分冗余分析（partial-redundancy analysis）

部分冗余分析确定在某条执行路径上被执行了两次（或多次），而其操作数在这些计算之间没有修改过的那些计算。在Morel和Renvoise最早给出的公式表示中，这是一个使用位向量的双向流问题，位向量的每一位表示一个表达式计算。这个问题的最新公式表示已表明可以用一系列的向前和向后数据流计算来实现它。

在优化中遇到的不仅仅是上面列出的这些数据流问题，但它们是其中最重要的一些问题。

解数据流问题有多种方法，包括如下几种（关于详细的原文信息，参见8.16节）：

1. Allen的强连通区域方法；

230

2. Kildall的迭代算法(见8.4节)；
3. Ullman的$T1$-$T2$分析；
4. Kennedy的结点列表算法；
5. Farrow、Kennedy和Zucconi的图形文法方法；
6. 消去法，例如区间分析(见8.8节)；
7. Rosen的高级（语法制导的）方法；
8. 结构分析(见8.7节)；
9. 位置式(slotwise)分析(见8.9节)。

我们这里关注三种方法：(1)简单迭代法，包括关于确定迭代次序的若干策略；(2)消去法或利用区间的基于控制树的方法；(3)另一种使用结构分析的基于控制树的方法。如我们将看到的，这些方法在实现的容易度、速度与空间的要求，以及增量式地更新数据流信息以避免完全重新计算它们等方面都有较好的表现。此外，对其他方法，如新近引入的位置式分析，我们也稍作评论。

8.4 迭代数据流分析

迭代数据流分析是我们在8.1节的例子中执行到达-定值分析时使用过的方法。我们先介绍它是因为它最容易实现，而且也是最频繁使用的方法。这种方法也是一种重要的基本方法，因为8.6节讨论的基于控制树的方法需要能对代码的非正常（或非可归约）代码区域进行迭代分析(或进行结点分割，或进行函数格上的数据流分析)。

我们首先介绍向前分析的迭代实现方法。此方法不难推广到向后和双向问题。

假设我们有一给定的流图$G=<N, E>$和一个格$\mathbf{L}$，流图G有属于N的入口结点`entry`和出口结点`exit`。我们希望对所有的$B \in N$，计算$in(B)$，$out(B) \in \mathbf{L}$，其中$in(B)$表示进入B的入口时的数据流信息，$out(B)$表示从B出口时的数据流信息，它们由下面的数据流方程给出：

$$in(B) = \begin{cases} Init & \text{若} \quad B = \texttt{entry} \\ \bigsqcap_{P \in Pred(B)} out(P) & \text{否则} \end{cases}$$

$$out(B) = F_B(in(B))$$

bbs.theithome.com

其中*Init*表示过程入口处适当的数据流信息初值，F_B ()表示与执行基本块*B*对应的数据流信息转换，⊓模拟合并进入基本块的边的数据流信息的效果。当然，这也可以只用*in*()函数表示为: 231

$$in(B) = \begin{cases} Init & \text{若} \quad B = \texttt{entry} \\ \prod\limits_{P \in Pred(B)} F_P(in(P)) & \text{否则} \end{cases}$$

如果⊔模拟合并流信息的效果，则用它代替算法中的⊓。*Init*的值通常是⊤或⊥。

图8-6给出的算法Worklist_Iterate()只使用*in*()函数，读者可以很容易地将它修改为同时使用*in*()和*out*()。算法的策略是迭代地应用上面给出的定值方程，维护一张工作表记录其前驱的*in*()值在最后一次迭代发生了改变的基本块，直到此工作表为空。开始时，这个工作表包含流图中除entry之外的所有基本块，排除entry是因为它的信息不会发生改变。因为合并进入一个结点的边的信息的效果是由⊓来模仿的，故totaleffect的适当初值是⊤。函数F_B (*x*)用F(B,x)来表示。

```
procedure Worklist_Iterate(N,entry,F,dfin,Init)
   N: in set of Node
   entry: in Node
   F: in Node × L ⟶ L
   dfin: out Node ⟶ L
   Init: in L
begin
   B, P: Node
   Worklist: set of Node
   effect, totaleffect: L
   dfin(entry) := Init
*  Worklist := N - {entry}
   for each B ∈ N do
      dfin(B) := ⊤
   od
   repeat
*     B := ♦Worklist
      Worklist -= {B}
      totaleffect := ⊤
      for each P ∈ Pred(B) do
         effect := F(P,dfin(P))
         totaleffect ⊓= effect
      od
      if dfin(B) ≠ totaleffect then
         dfin(B) := totaleffect
*        Worklist ∪= Succ(B)
      fi
   until Worklist = ∅
end    || Worklist_Iterate
```

图8-6 迭代数据流分析的工作表算法(管理工作表的语句用星号标志)

这个算法的计算效率与若干因素有关，包括格L、流函数F_B ()，以及管理工作表的方法。尽管格和流函数是由我们求解的数据流问题确定的，但工作表的管理与所求解的数据流问题无关。注意，管理工作表与我们如何实现图8-6中标有星号的语句相对应。最容易的实现方法是用栈或队列来表示工作表，而不管基本块之间在流图结构中的相互关系。另一方面，如果我们先处理一个基本块的所有前驱，则每次遇到这个基本块时，我们能够期待有关于此基本块的最有效的信息。通过先按前一章介绍的次序，即逆后序对结点进行排序，然后按队列继续，则可 232

bbs.theithome.com

以达到这一目的。因为按后序，一个结点直到它的所有深度为主生成树的后继都已经访问过后才会被访问，而按逆后序，则它在其所有后继还未被访问之前访问。如果A是流图G中任意无环路径上后向边的最大条数，则在使用逆后序时，repeat循环通过$A+2$遍就足够了[⊖]。注意，构造出A的数量级为$|N|$的流图是有可能的，但实际中很少见。几乎所有的情况下$A<3$，并且常常$A=1$。

作为向前迭代算法的一个例子，我们重复曾在8.1节非形式化做过的例子。图7-4中各个基本块的流函数如表8-2所示，其中*id*表示恒等函数。所有基本块的*dfin*(*B*)的初值是<00000000>。路径合并运算符是⊔，或在位向量情况下是按位逻辑或。按逆后序，初始工作表为{B1, B2, B3, B4, B5, B6, exit}。

表8-2 图7-4中流图的流函数

$F_{entry} = id$
$F_{B1}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle 111x_4x_500x_8\rangle$
$F_{B2} = id$
$F_{B3}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle x_1x_2x_31x_5x_6x_70\rangle$
$F_{B4} = id$
$F_{B5} = id$
$F_{B6}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle x_10001111\rangle$

进入repeat循环时，B的初值是B1，此时工作表变成{B2, B3, B4, B5, B6, exit}。B1的惟一前驱是entry，计算effect和totaleffect，结果得到<00000000>，dfin(B1)的初值没有改变，因此，B1的后继不放到工作表中。

然后，我们得到B=B2，并且工作表变成{B3, B4, B5, B6, exit}。B2的惟一前驱是P=B1，计算effect和totaleffect的结果是<11100000>，它成了dfin(B2)的新值，并且exit被加入到工作表中产生{B3, B4, B5, B6, exit}。

接下来，我们得到B=B3，工作表变成{B4, B5, B6, exit}。B3有一个前驱，即B1，计算

233 effect和totaleffect的结果是<11100000>，它成了dfin(B3)的新值，并且B4被加到工作表中。

之后，我们得到B=B4，工作表变成{B5, B6, exit}。B4有两个前驱，B3和B6，由于计算effect、totaleffect和dfin(B4)时，B3贡献了<11110000>，B6贡献了<00001111>，因此，这个迭代的最终计算结果是dfin(B4)=<11111111>，并且工作表变成{B5, B6, exit}。

接着，B=B5，工作表变成{B6, exit}。B5有一个前驱B4，它给effect、totaleffect和dfin(B5)贡献了<11111111>，并且exit被加到工作表中。

接着，B=B6，工作表变成{exit}。B6有一个前驱B4，它给dfin(B6)贡献了<11111111>，并且B4被放回到工作表中。

现在，从工作表中去掉exit，其结果为{B4}，并且B4的两个前驱B2和B5导致dfin(exit)=<11111111>。

读者可以检查repeat的循环体对工作表中的每一个元素执行了两次以上，但最后一次迭代计算中dfin()值没有进一步的改变。你也可以检查这个结果与8.1节计算出来的结果是相同的。

一旦我们适当地形成了向后问题，就不难转变上面这个算法用于处理向后问题。我们可以选择将向后问题的数据流信息与每一个基本块的入口相连，也可以与它的出口相连。为了利用向前和向后问题之间的对偶性，我们选择将它与基本块的出口相连。

同向前问题一样，假定有一给定的流图$G=\langle N, E\rangle$和一个格$\mathbf{L}$，流图G有属于N的入口结点entry和出口结点exit。我们希望对所有的$B\in N$，计算$out(B)\in\mathbf{L}$，其中$out(B)$表示从B出口时的数据流信息，它们由下面的数据流方程给出:

⊖ 如果我们追踪每一遍中其数据流信息发生了改变的基本块的个数，而不是简单地追踪是否有改变，这个上界可减到$A+1$。

bbs.theithome.com

$$out(B)=\begin{cases} Init & \text{若} \quad B=\texttt{exit} \\ \bigsqcap_{P\in Succ(B)} in(P) & \text{否则} \end{cases}$$

$$in(B)=F_B(out(B))$$

其中，*Init*表示从过程出口时数据流信息的适当初值，F_B ()表示与反向执行基本块*B*对应的数据流信息转换，⊓模拟合并从一个基本块出来的各条边的数据流信息的效果。同向前问题一样，它们也可以只用*out*()函数来表示:

$$out(B)=\begin{cases} Init & \text{若} \quad B=\texttt{exit} \\ \bigsqcap_{P\in Succ(B)} F_P(out(P)) & \text{否则} \end{cases}$$

如果用⊔模拟合并流信息的效果，则用它代替算法中的⊓。 234

现在，向后问题的迭代算法除了有适当的替换之外，与图8-6给出的向前问题的算法是相同的。这种替换是: *out*替换*in*，exit替换entry，*Succ*()替换*Pred*()。管理工作表最有效的方法是按逆前序对它进行初始化，因此计算效率的上界与向前迭代算法相同。

8.5 流函数的格

就像我们将执行数据流分析的对象看成是格的元素一样，我们在执行这种方法中使用的流函数集合也形成了一个格，它的交和并是由底层格的交和并诱导出来的。如我们在8.6节将看到的，单调流函数的诱导格对于将基于控制树的数据流方法公式化非常重要。

具体地，令**L**是一个给定的格，令$\mathbf{L}^{\mathbf{F}}$表示所有从**L**到**L**的单调函数的集合，即，

$f\in\mathbf{L}^{\mathbf{F}}$当且仅当$\forall x,y\in\mathbf{L}\ x\sqsubseteq y$隐含着$f(x)\sqsubseteq f(y)$

则$\mathbf{L}^{\mathbf{F}}$上诱导出的点态交运算由下式给出:

$$\forall f,g\in\mathbf{L}^{\mathbf{F}},\quad \forall x\in\mathbf{L}\ (f\sqcap g)(x)=f(x)\sqcap g(x)$$

并且容易验证对应的诱导并运算和序函数都有适当的定义，由此确定了$\mathbf{L}^{\mathbf{F}}$确实是一个格。$\mathbf{L}^{\mathbf{F}}$的顶元素和底元素是$\bot^{\mathbf{F}}$和$\top^{\mathbf{F}}$，其定义为

$$\forall x\in\mathbf{L}\ \bot^{\mathbf{F}}(x)=\bot\ \text{且}\ \top^{\mathbf{F}}(x)=\top$$

为了提供进行基于控制树的数据流分析所需要的运算，我们需要对$\mathbf{L}^{\mathbf{F}}$多定义一个函数和两个运算。这个额外的函数就是恒等函数*id*，其定义为$id(x)=x,\ \forall x\in\mathbf{L}$。两个运算是合成和克林(或迭代)闭包。对于任意两个函数$f,\ g\in\mathbf{L}^{\mathbf{F}}$，$f$和$g$的合成记做$f\circ g$，由下式定义:

$$(f\circ g)(x)=f(g(x))$$

容易证明$\mathbf{L}^{\mathbf{F}}$的合成是封闭的。同样，对任意的$f\in\mathbf{L}^{\mathbf{F}}$，我们定义 f^n为

$$f^0=id\ \text{且对}\ n\geqslant 1,\ f^n=f\circ f^{n-1}$$

$f\in\mathbf{L}^{\mathbf{F}}$的克林闭包（Kleene closure）记做$f^*$，其定义是:

$$\forall x\in\mathbf{L}\ f^*(x)=\lim_{n\to\infty}(id\sqcap f)^n(x)$$

同样按惯例，我们定义$f^+=f\circ f^*$。为了证明$\mathbf{L}^{\mathbf{F}}$在克林闭包中是封闭的，我们依赖于这样一个事实，即，对我们使用的所有函数，我们的格都具有有限的有效高度。这意味着如果对任意$x_0\in\mathbf{L}$，计算序列

$$x_{i+1}=(id\sqcap f)(x_i)$$

bbs.theithome.com

235 则存在着一个i使得$x_i = x_{i+1}$，因此，显然有$f^*(x_0) = x_i$。

容易证明如果L是位向量的格，比如说$\mathbf{BV}^n$，则对每一个函数f: $\mathbf{BV}^n \to \mathbf{BV}^n$，$f$ 对交和并是**强可分配的**（strongly distributive），即，

$$\forall x, y, f(x \sqcup y) = f(x) \sqcup f(y) \text{且} f(x \sqcap y) = f(x) \sqcap f(y)$$

同样，只要各个位的位置是相互独立改变的，同我们考虑的所有位向量分析情形一样，$\mathbf{BV}^n$就具有有效高度1，即$f \circ f = f$，所以，$f^* = id \sqcap f$。如我们在下一节将看到的，这使得用位向量表示的基于控制树的数据流分析具有非常高的效率。

8.6 基于控制树的数据流分析

基于控制树的数据流分析（control-tree-based data-flow analysis）有两种算法，即，区间分析和结构分析，因为它们都基于使用7.6节和7.7节讨论的控制树，所以两者非常相似。它们比迭代方法更难实现——为了处理非正常区域（如果出现这种区域的话），它们需要进行结点分割、迭代，或解单调函数格上的数据流问题——但它们的优点是使得以增量方式随着优化转换过程对程序的改变而更新数据流信息较容易。

由于历史的原因，基于控制树的方法作为一个整体统称为*消去法*（elimination method）。它们需要对过程的控制树进行两遍处理，其中假定控制树已经由前面提到的结构或区间控制流分析而建立。每一遍它们都访问控制树中的每一个结点：第一遍自底向上执行，即从基本块开始，它们构造一个表示过程某一部分执行效果的流函数，过程的这一部分对应于控制树的一部分；第二遍自顶向下执行，即从表示整个过程的抽象结点和从对应于入口(对向前问题)或出口(对向后问题)的初始信息开始，它们利用在第一遍构造的流函数，构造和计算传播数据流信息进入和通过每一个控制树结点所表示的区域的数据流方程。

8.7 结构分析

结构分析（structural analysis）使用由结构控制流分析得出的控制结构的详细信息来产生相应的数据流方程，这些数据流方程表示控制结构的数据流效果。我们从结构分析开始是因为它较易于描述和理解，并且一旦具备了它需要的所有机制，则区间分析需要的所有机制就只是它们的子集。

8.7.1 结构分析：向前问题

首先，假定我们执行的是向前问题——如将看到的，向后问题稍微麻烦一点，因为每一个
236 控制流结构只有一个入口，但却可以有多个出口。同迭代算法一样，我们也假定用$\sqcap$来模仿数据流信息在几条控制流路径交汇的地方合并的效果。

在执行结构数据流分析过程中，我们在第一遍遇到的抽象图多数都是图7-35和图7-36所示类型的简单区域。基本块的流函数F_B与迭代分析中的流函数相同——流函数只与要解的问题和基本块的内容有关。

现在，假设我们有一个if-then结构，如图8-7所示，并且每一个结构的流函数在离开这个结构的边上给出，则第一遍为它构造的流函数$F_{\text{if-then}}$与if-then的两个分量的流函数有关：

$$F_{\text{if-then}} = (F_{\text{then}} \circ F_{\text{if/Y}}) \sqcap F_{\text{if/N}}$$

即，执行if-then结构的效果是将执行if部分并从Y分支出口之后执行then部分的效果，与执行if部分并从N分支出口的效果（用路径合并运算符$\sqcap$）合并的结果。

bbs.theithome.com

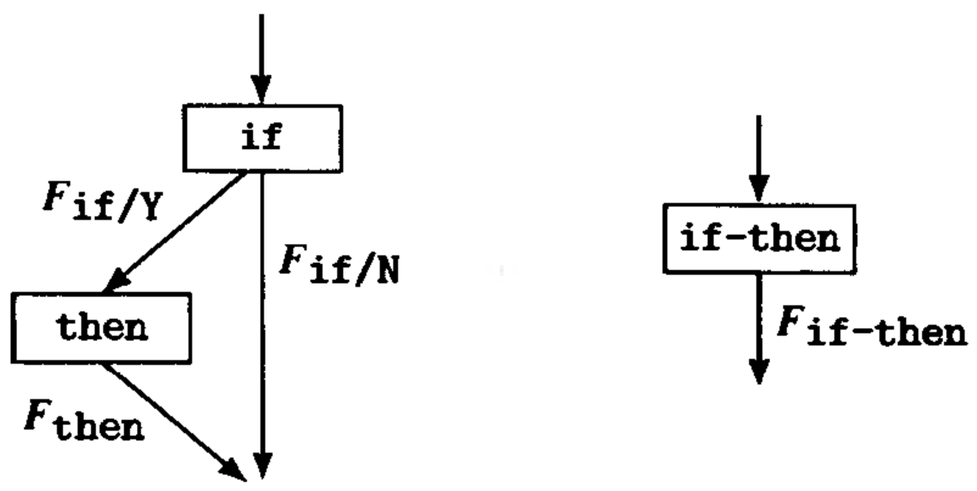


图8-7 if-then结构的结构分析流函数

注意，我们可以区分从if的Y和N的出口并且使它们具有不同的流函数$F_{if/Y}$和$F_{if/N}$，也可以不区分。假若选择不区分它们，如在前面的迭代分析表示中一样，我们将简单地只有一个关于if的流函数，即F_{if}，而不是$F_{if/Y}$和$F_{if/N}$，即将有

$$F_{if\text{-}then}=(F_{then}\circ F_{if})\sqcap F_{if}=(F_{then}\sqcap id)\circ F_{if}$$

两种方法都是合法的——在if的两个分支区别对待我们感兴趣的数据流值的情况下，同常数传播或边界检查的例子一样，前者可以产生比后者更为精确的信息。但在以后的例子中，我们按习惯做法使用后一种方法。

第二遍构造的数据流方程告诉我们，如何将已知的进入if-then结构的数据流信息传播到它的每一个子结构的入口。这些数据流方程是相当明显的:

$$in(\text{if})=in(\text{if-then})$$
$$in(\text{then})=F_{if/Y}(in(\text{if}))$$

或者，如果我们选择不区分if的出口，则:

$$in(\text{if})=in(\text{if-then})$$
$$in(\text{then})=F_{if}(in(\text{if}))$$

上面第一对方程中的第一个方程说的是，在if部分的入口，我们具有的数据流信息与在if-then结构的入口具有的信息相同。第二个方程说的是在then部分的入口，我们具有的信息是根据if部分从Y分支离开时的数据流效果，对if入口的信息进行转换后的结果。除了没有对从if部分的出口进行区别外，第二对方程与第一对方程是相同的。

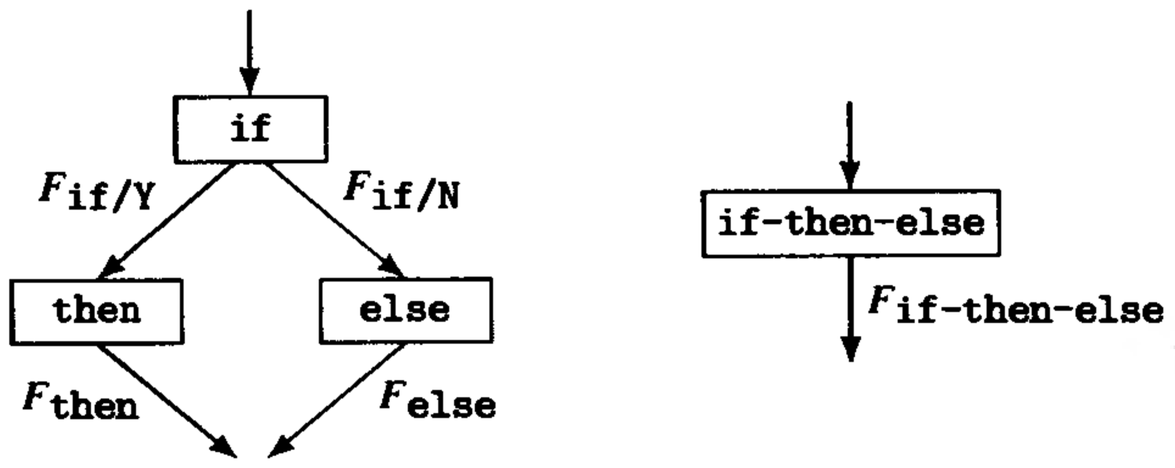


图8-8 if-then-else结构的结构分析流函数

下面我们考虑如何将这种处理扩展到if-then-else结构，以及如何扩展到while循环。if-then-else的形式如图8-8所示，它的函数很容易从if-then情形的函数推广得出。在第一遍构造的流函数$F_{if\text{-}then\text{-}else}$与如下分量的流函数有关:

$$F_{if\text{-}then\text{-}else}=(F_{then}\circ F_{if/Y})\sqcap(F_{else}\circ F_{if/N})$$

第二遍构造的传播函数是:

237

bbs.theithome.com

$in(\texttt{if}) = in(\texttt{if-then-else})$

$in(\texttt{then}) = F_{\texttt{if/Y}}(in(\texttt{if}))$

$in(\texttt{else}) = F_{\texttt{if/N}}(in(\texttt{if}))$

对于while循环，我们有图8-9所示的形式。在自底向上遍中，表示循环迭代一次然后回到其入口点的结果的流函数是$F_{\text{body}} \circ F_{\text{while/Y}}$，因此这样重复任意次的结果由这个函数的克林闭包给出：

$$F_{\text{loop}} = (F_{\text{body}} \circ F_{\text{while/Y}})^*$$

并且执行整个while循环的结果由重复地执行while和body块，然后执行while块并从N分支离开而给出，即，

238

$$F_{\text{while-loop}} = F_{\text{while/N}} \circ F_{\text{loop}}$$

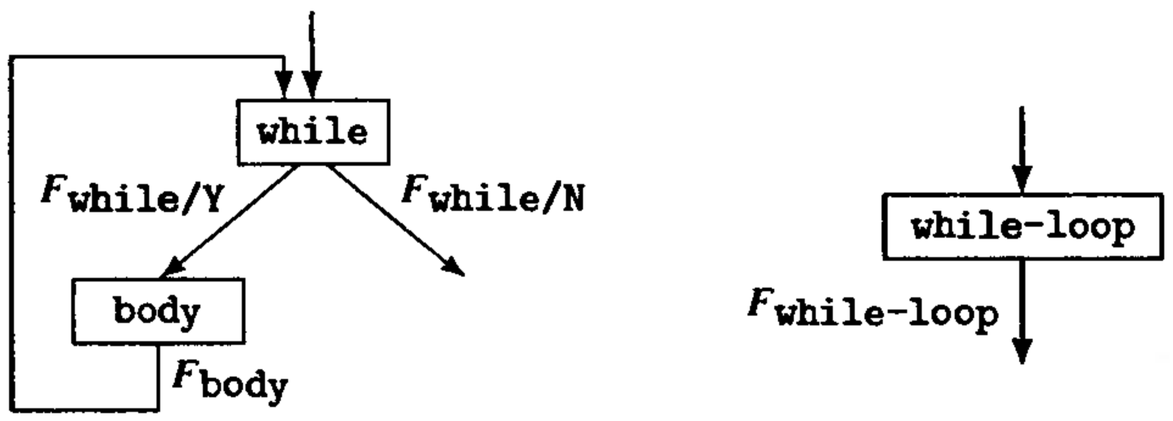


图8-9 while循环的结构分析流函数

注意，在更常见的情形中，即位向量问题，

$$(F_{\text{body}} \circ F_{\text{while/Y}})^*$$

简单的是

$$id \sqcap (F_{\text{body}} \circ F_{\text{while/Y}})$$

但无论要求解的是什么向前流问题，上面的方程都是正确的。在自顶向下遍中，对此while循环，我们有

$$in(\texttt{while}) = F_{\text{loop}}(in(\texttt{while-loop}))$$

$$in(\texttt{body}) = F_{\texttt{while/Y}}(in(\texttt{while}))$$

这是因为到达while部分既可以从循环之外，也可以从迭代内，而到达body部分则只能通过进入循环，同时还要执行while和body块若干次(可能0次)，并通过Y分支离开while块。

同样，如果我们不区分出口分支，则有

$$F_{\text{loop}} = (F_{\text{body}} \circ F_{\texttt{while}})^*$$

并且执行整个while循环的结果由重复地执行while和body块，然后执行while块并从N分支离开给出，即，

$$F_{\texttt{while-loop}} = F_{\texttt{while}} \circ F_{\text{loop}}$$

$$= F_{\texttt{while}} \circ (F_{\text{body}} \circ F_{\texttt{while}})^*$$

$$in(\texttt{while}) = F_{\text{loop}}(in(\texttt{while-loop}))$$

$$in(\texttt{body}) = F_{\texttt{while}}(in(\texttt{while}))$$

从if-then和if-then-else的情形应当清楚如何推广这些方程到一般的无环区域A。具体地，假设构成无环区域的抽象结点是$B0, B1, \cdots, Bn$，其中$B0$是区域的入口结点，并且每一个

bbs.theithome.com

版权所有 侵权必究

Bi 具有出口$Bi/1$，…，Bi/e_i(当然，多数情况下$e_i=1$，而且它大于2的情况很少见)。按习惯的做法，对每一个离开它的Bi/e相连一个向前流函数F_{Bile}，并令$P(A, Bi_k/e_k)$表示从A的入口到A中某个抽象结点的出口Bi_k/e_k的所有可能路径的集合。对于自底向上遍，已知路径 239

$$p=B0/e_0,\ Bi_1/e_1,\ \cdots,\ Bi_k/e_k \in P(A, Bi_k/e_k)$$

则此路径的合成流函数是

$$F_p = F_{Bi_k/e_k} \circ \cdots \circ F_{Bi_1/e_1} \circ F_{Bi_0/e_0}$$

并且从A的入口到从Bi_k/e_k出口的结点的所有可能路径对应的流函数是

$$F_{(A,Bi_k/e_k)} = \prod_{p\in P(A,Bi_k/e_k)} F_p$$

对于自顶向下遍，对于$i\neq 0$的Bi的入口，令$P_p(A, Bi)$表示所有使得$Bj \in Pred(Bi)$且出口Bj/e通向Bi的$P(A, Bj/e)$组成的集合。则

$$in(Bi) = \begin{cases} in(A) & \text{若 } i=0 \\ \prod_{P(A,Bj/e)\in P_p(A,Bi)} F_{(A,Bj/e)}(in(A)) & \text{否则} \end{cases}$$

对于是正常区域的一般有环区域C，存在着一条从某个基本块Bc通向入口基本块$B0$的回边。如果我们去掉这条回边，结果是一个无环区域。同上面一样处理，我们构造出从C的入口到这个无环区域中Bi_k/e_k的所有可能路径对应的流函数，其中的无环区域是通过去掉回边而得到的。这给了我们一组流函数$F_{(C,\ Bi_k/e_k)}$，并且特别地，如果Bc/e是此回边的尾结点，函数

$$F_{iter} = F_{(C,Bc/e)}$$

表示执行该区域体一次迭代的结果。于是，

$$F_C = F^*_{iter}$$

表示执行该区域并返回到其入口的完整的数据流效果，而

$$F'_{(C,Bi_k/e_k)} = F_{(C,Bi_k/e_k)} \circ F_C$$

表示执行该区域并从Bi_k/e_k离开的效果。相应地，对于自顶向下遍，

$$in(Bi) = \begin{cases} in(C) & \text{若 } i=0 \\ \prod_{P(C,Bj/e)\in P_p(C,Bi)} F'_{(C,Bj/e)}(in(B0)) & \text{否则} \end{cases}$$

对于非正常区域R，在自底向上遍中，我们构造的一组方程同前面为一般无环区域构造的那些方程类似，它们表示的是从区域的入口到其内任意点的路径上的数据流效果。在自顶向下遍中，我们按习惯的方法从区域入口拥有的信息开始，使用自底向上遍构造的函数，按习惯的方法将数据流信息传播到区域的每一点。这些方程与无环情况下的方程之间的主要不同是，非正常区域R的自顶向下系统是递归的，因为这种区域包含多个环路。给定了方程组后，我们可以按如下三种方法继续处理: 240

1. 我们可以使用结点分割，结点分割能将非正常区域转变为具有更多(可能很多)结点的正常区域。

2. 我们可以用迭代方式来解递归方程组，这种方法在每次解数据流问题时，利用外层结构的方程所产生的信息作为我们在R的入口处的初始数据。

3. 我们可以将方程组本身看成不是定义在格$\mathbf{L}$上，而是定义在格$\mathbf{L}^{\mathbf{F}}$上(我们在8.5节讨论过的从$\mathbf{L}$到$\mathbf{L}$的单调函数组成的格)的向前数据流问题，并且解这个方程组，由此产生与该区域内

bbs.theithome.com

路径对应的流函数。这要求底层的格L是有限的，对于我们考虑的大多数问题，包括所有位向量问题，均能满足这个条件。

例如，考虑图8-10中的简单非正常区域，假设如图8-10所示那样归约这个区域到B1a，则对F_{B1a}的自底向上方程是:

$$F_{B1a} = ((F_{B3} \circ F_{B2})^+ \circ F_{B1}) \sqcap ((F_{B3} \circ F_{B2})^* \circ F_{B3} \circ F_{B1})$$

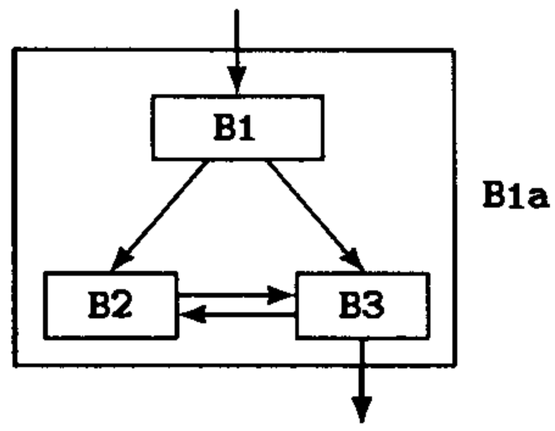


图8-10 一个非正常区域

因为按向前方向通过B1a的行程有两种情况，一是由B1到B2，再到B3出口，或在B2、B3之间循环多次后由B3出口；二是由B1到B3出口，或在B3、B2之间循环0次或多次后由B3出口。对于自顶向下方程，得到的是递归方程组:

$$in(B1) = in(B1a)$$
$$in(B2) = F_{B1}(in(B1)) \sqcap F_{B3}(in(B3))$$
$$in(B3) = F_{B1}(in(B1)) \sqcap F_{B2}(in(B2))$$

或者，我们能在函数格中求解关于in(B2)和in(B3)的方程，产生

$$in(B2) = (((F_{B3} \circ F_{B2})^* \circ F_{B1}) \sqcap ((F_{B3} \circ F_{B2})^* \circ F_{B3} \circ F_{B1}))(in(B1))$$
$$= ((F_{B3} \circ F_{B2})^* \circ (id \sqcap F_{B3}) \circ F_{B1})(in(B1))$$

241

和

$$in(B3) = (((F_{B2} \circ F_{B3})^* \circ F_{B1}) \sqcap ((F_{B2} \circ F_{B3})^* \circ F_{B2} \circ F_{B1}))(in(B1))$$
$$= ((F_{B3} \circ F_{B2})^* \circ (id \sqcap F_{B2}) \circ F_{B1})(in(B1))$$

作为向前结构数据流分析的一个例子，我们继续使用我们的到达－定值例子。首先，我们必须对它进行结构控制流分析，如图8-11所示。在此数据流分析的自底向上遍中，我们构造的第一个方程是关于while循环的方程，如下所示(因为在计算到达-定值中区别Y和N出口没有产生不同，因此忽略了它们):

$$F_{B4a} = F_{B4} \circ (F_{B6} \circ F_{B4})^* = F_{B4} \circ (id \sqcap (F_{B6} \circ F_{B4}))$$

其他的方程是，对于被归约到B3a的块:

$$F_{B3a} = F_{B5} \circ F_{B4a} \circ F_{B3}$$

对于被归约到B1a的if-then-else:

242

$$F_{B1a} = (F_{B2} \circ F_{B1}) \sqcap (F_{B3a} \circ F_{B1})$$

对于被归约到entrya的块:

$$F_{entrya} = F_{exit} \circ F_{B1a} \circ F_{entry}$$

如我们将看到的，数据流分析中实际上并没有使用最后一个方程。

在自顶向下遍中，我们为entrya块的三个分量构造的方程是

$$in(entry) = Init$$
$$in(B1a) = F_{entry}(in(entry))$$
$$in(exit) = F_{B1a}(in(B1a))$$

对于归约到B1a的if-then-else块的各个分量，其方程是:

$$in(B1) = in(B1a)$$
$$in(B2) = in(B3a) = F_{B1}(in(B1a))$$

对于归约到B3a的块的各个分量，其方程是:

$$in(B3) = in(B3a)$$

bbs.theithome.com

$$in(\text{B4a}) = F_{\text{B3}}(in(\text{B3a}))$$

$$in(\text{B5}) = F_{\text{B4a}}(in(\text{B4a}))$$

对于归约到B4a的while循环的各个分量，其方程是:

$$in(\text{B4}) = (F_{\text{B6}} \circ F_{\text{B4}})^*(in(\text{B4a})) = (id \sqcap (F_{\text{B6}} \circ F_{\text{B4}}))(in(\text{B4a}))$$

$$in(\text{B6}) = F_{\text{B4}}(in(\text{B4}))$$

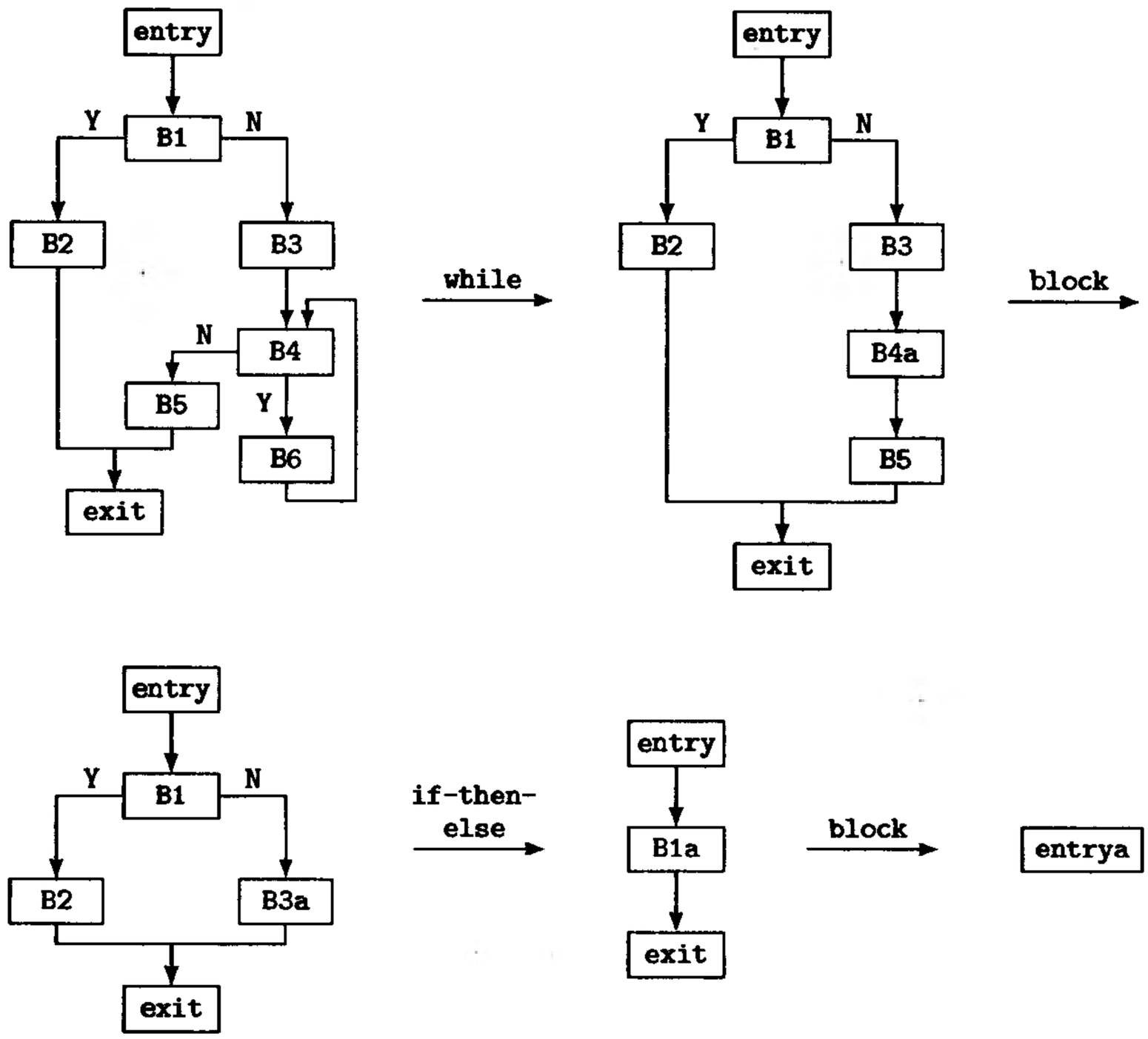


图8-11 我们的到达-定值例子的结构数据流分析

in(entry)的初值和各个块的流函数与前面迭代方法中的相同（参见表8-2)。我们在手工求解关于*in*()值的方程中，第一步是简化关于复合流函数的方程，具体如下:

$$\begin{aligned} F_{\text{B4a}} &= F_{\text{B4}} \circ (F_{\text{B6}} \circ F_{\text{B4}})^* \\ &= F_{\text{B4}} \circ (id \sqcap (F_{\text{B6}} \circ F_{\text{B4}})) \\ &= id \circ (id \sqcap (F_{\text{B6}} \circ id)) \\ &= id \sqcap F_{\text{B6}} \end{aligned}$$

$$\begin{aligned} F_{\text{B3a}} &= F_{\text{B5}} \circ F_{\text{B4a}} \circ F_{\text{B3}} \\ &= id \circ (id \sqcap F_{\text{B6}}) \circ F_{\text{B3}} \\ &= (id \sqcap F_{\text{B6}}) \circ F_{\text{B3}} \\ &= F_{\text{B4a}} \circ F_{\text{B3}} \end{aligned}$$

$$\begin{aligned} F_{\text{B1a}} &= (F_{\text{B2}} \circ F_{\text{B1}}) \sqcap (F_{\text{B3a}} \circ F_{\text{B1}}) \\ &= (id \circ F_{\text{B1}}) \sqcap ((id \sqcap F_{\text{B6}}) \circ F_{\text{B3}} \circ F_{\text{B1}}) \\ &= F_{\text{B1}} \sqcap ((id \sqcap F_{\text{B6}}) \circ F_{\text{B3}} \circ F_{\text{B1}}) \end{aligned}$$

bbs.theithome.com

我们然后从*in*(entry)开始，并利用F_B()函数的有效值来计算*in*()的值，其结果值如表8-3所示。读者可以验证每个基本块的*in*()值与迭代方法(8.1节)所计算的*in*()值是相同的。

表8-3 由结构分析计算出来的关于我们的到达－定值例子的*in*()值

in(entry)	=	<00000000>
in(B1)	=	<00000000>
in(B2)	=	<11100000>
in(B3)	=	<11100000>
in(B4)	=	<11111111>
in(B5)	=	<11111111>
in(B6)	=	<11111111>
in(exit)	=	<11111111>

8.7.2 结构分析：向后问题

如前一节开始时提及的，对向后问题进行结构分析要困难一点，因为我们使用的控制流结构虽然保证了每一个结构只有一个入口，但没有保证一定只有一个出口。

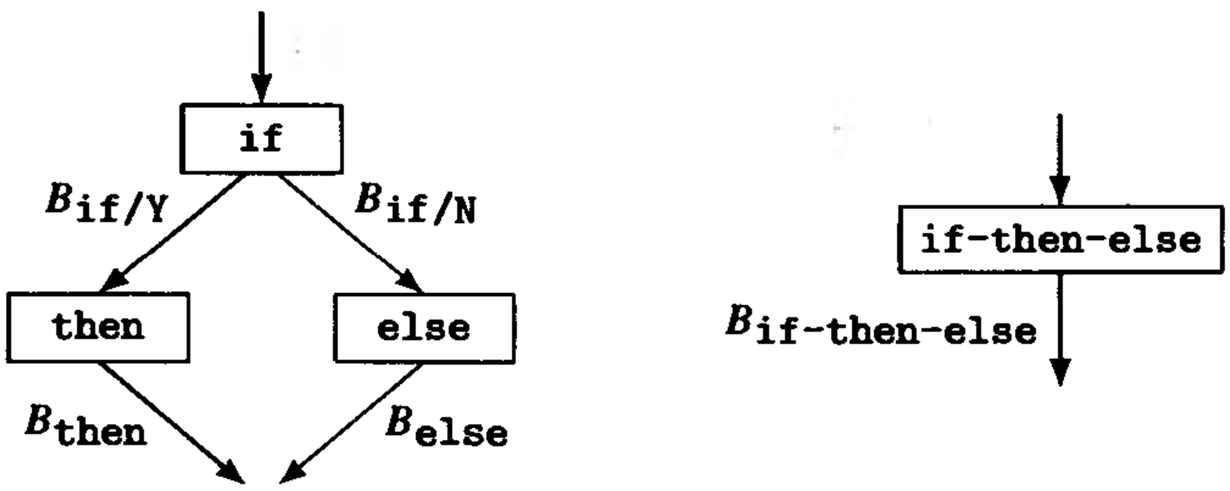


图8-12 if-then-else结构的向后结构分析的流函数

对于具有单一出口的结构，如if-then或if-then-else，我们可以简单地“逆转”向前问题中的方程。给定图8-12中的if-then-else，自底向上遍中为向后流经此if-then-else而构造的方程是

244 $$B_{\text{if-then-else}} = (B_{\text{if/Y}} \circ B_{\text{then}}) \sqcap (B_{\text{if/N}} \circ B_{\text{else}})$$

自顶向下遍中构造的方程是

$$out(\text{then}) = out(\text{if-then-else})$$

$$out(\text{else}) = out(\text{if-then-else})$$

$$out(\text{if}) = B_{\text{then}}(out(\text{then})) \sqcap B_{\text{else}}(out(\text{else}))$$

对于一般的无环区域A，再次假设构成此无环区域的抽象结点是$B0, B1, \cdots, Bn$。令$B0$是区域的入口结点，并且每一个Bi具有入口$Bi/1, \cdots, Bi/e_i$(当然，多数情况下$e_i=1$或2，而且大于2的情况很少见)。对每一个Bi和它的入口e相连一个向后流函数$B_{Bi/e}$，并令$P(Bi_k/e_k, Bi_l/e_l)$表示从某个Bi_k/e_k到Bi_l/e_l的所有可能的(向前)路径的集合。对于自底向上遍，给定某条路径$p \in P(Bi_k/e_k, Bi_l/e_l)$，该路径的复合向后流函数是

$$B_p = B_{Bi_k/e_k} \circ \ldots \circ B_{Bi_l/e_l}$$

定义$Exits(A)$是区域A的出口基本块集合，即，$Bi \in Exits(A)$当且仅当存在$Bj \in Succ(Bi)$使得$Bj \notin A$。则从Bi_k/e_k到所有可能从区域A出口的结点的路径所对应的向后流函数是

bbs.theithome.com

$$B_{(A,Bi_k/e_k)} = \prod_{\substack{p\in P(Bi_k/e_k,Bi_l/e_l)\\ Bi_l\in Exits(A)}} B_p$$

对于自顶向下遍，对于每一个从区域A出口的基本块Bj，我们使数据流信息$out(Bj)$与它相连。令$P_s(A, Bi)$表示所有使得$Bj \in Succ(Bi)$且$Bk \in Exits(A)$的路径$P(Bj/e, Bk/f)$的集合。则，

$$out(Bi) = \begin{cases} out(A) & \text{若 } Bi \in Exits(A) \\ \prod\limits_{P(Bj/e,Bk/f)\in P_s(A,Bi)} B_{(A,Bj/e)}(out(Bk)) & \text{否则} \end{cases}$$

对于是正常区域的一般有环区域C，我们将前面无环区域的方法与正常循环区域的向前问题的方法结合起来。对于这种区域，同样存在着一条从某个基本块Bc通向入口基本块$B0$的回边。如果我们去掉这条回边，便得到一个无环区域。对于这个去掉回边而得到的无环区域，我们构造它的从C的所有出口结点到结点Bi_k/e_k的所有可能的（向后）路径对应的向后流函数，这使我们得到一组流函数$B_{(C.Bi_k/e_k)}$，并且特别地，如果Bc/e是回边的头结点，函数

$$B_{iter} = B_{(C,Bc/e)}$$

表示该区域体执行一次迭代的结果。于是，

$$B_C = B^*_{iter}$$

245

表示执行该区域并返回到其入口的完整的向后数据流效果，而

$$B'_{(C,Bi_k/e_k)} = B_{(C,Bi_k/e_k)} \circ B_C$$

表示从该区域的出口结点到其内的某个结点Bi_k反向执行该区域的向后数据流效果。相应地，对于自顶向下遍，对于从C的Bj出口的任意基本块，我们使数据流信息$out(Bj)$与之相连，并且对于该区域内的每一个结点Bi，如下所示有

$$out(Bi) = \begin{cases} out(C) & \text{若 } Bi \in Exits(C) \\ \prod\limits_{P(Bj/e,Bk/f)\in P_s(C,Bi)} B_{(C,Bj/e)}(out(Bk)) & \text{否则} \end{cases}$$

其中，$P_s(C, Bi)$和$P(Bj/e, Bk/f)$的定义与前面无环区域的定义相同。

非正常区域向后问题的处理与它们的向前问题的处理类似。我们构造相应的一组递归数据流方程，并采用结点分割方法迭代地解它们，或者作为$\mathbf{L}^F$上的数据流问题来解它们。令人感到意外的是，函数格上的数据流问题在这里变成了向前问题，而不是向后问题。

作为向后问题的例子，考虑图8-13中由B0到B5组成的区域A。令p是路径B0/1、B1/1、B3/1、B4/1，则B_p由下式给出

$$B_p = B_{B0/1} \circ B_{B1/1} \circ B_{B3/1} \circ B_{B4/1}$$

从B0/1到A的两个出口的路径是

246

$$p1 = B0/1, B1/1, B3/1, B4/1$$

$$p2 = B0/1, B1/1, B3/2, B5/1$$

$$p3 = B0/2, B2/1, B3/1, B4/1$$

$$p4 = B0/2, B2/1, B3/2, B5/1$$

$$p5 = B0/2, B2/2, B5/1$$

并且从A的所有出口到B0/1的向后流函数是

bbs.theithome.com

$$B_{(A,B0/1)} = B_{p1} \sqcap B_{p2} \sqcap B_{p3} \sqcap B_{p4} \sqcap B_{p5}$$
$$= B_{B0/1} \circ B_{B1/1} \circ B_{B3/1} \circ B_{B4/1}$$
$$\sqcap B_{B0/1} \circ B_{B1/1} \circ B_{B3/2} \circ B_{B5/1}$$
$$\sqcap B_{B0/2} \circ B_{B2/1} \circ B_{B3/1} \circ B_{B4/1}$$
$$\sqcap B_{B0/2} \circ B_{B2/1} \circ B_{B3/2} \circ B_{B5/1}$$
$$\sqcap B_{B0/2} \circ B_{B2/2} \circ B_{B5/1}$$

out (B0)的值是

$$out(B0) = B_{p1}(out(B4)) \sqcap B_{p2}(out(B5)) \sqcap B_{p3}(out(B4)) \sqcap B_{p4}(out(B5))$$
$$\sqcap B_{p5}(out(B5))$$
$$= B_{B0/1}(B_{B1/1}(B_{B3/1}(B_{B4/1}(out(B4)))))$$
$$\sqcap B_{B0/1}(B_{B1/1}(B_{B3/2}(B_{B5/1}(out(B5)))))$$
$$\sqcap B_{B0/2}(B_{B2/1}(B_{B3/1}(B_{B4/1}(out(B4)))))$$
$$\sqcap B_{B0/2}(B_{B2/1}(B_{B3/2}(B_{B5/1}(out(B5)))))$$
$$\sqcap B_{B0/2}(B_{B2/2}(B_{B5/1}(out(B5))))$$

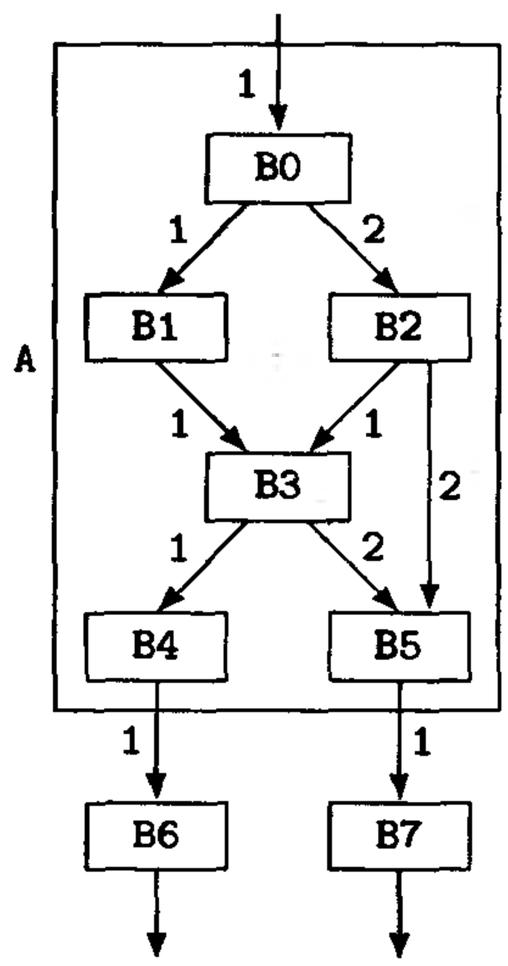


图8-13 用于向后结构分析的一个无环区域A

```
Component = enum {if,then,else}
FlowFcn = Var ⟶ Lattice


procedure ComputeF_if_then_else(x,r) returns FlowFcn
   x: in FlowFcn
   r: in integer
begin
   y: FlowFcn
   y := ComputeF_if(x,Region_No(r,if))
   return ComputeF_then(y,Region_No(r,then))
      ⊓ ComputeF_else(y,Region_No(r,else))
end   || ComputeF_if_then_else

procedure ComputeIn_then(x,r) returns FlowFcn
   x: in FlowFcn
   r: in integer
begin
   return ComputeF_if(ComputeIn_if(x,r),r)
end   || ComputeIn_then
```

图8-14 if-then-else的结构数据流方程的部分代码表示

8.7.3 结构分析方程的表示

结构分析中使用的方程有两种类型：(1)用于简单形式（如if-then-else或while循环）的方程，这种简单形式的方程可直接用实现它们的代码表示；(2)用于复杂形式（如非正常区域）的方程。

图8-14给出了一个表示简单结构的代码例子。对于一种简单结构的每一种可能的分量，类型Component含有一个用于标识它们的标识符；这里我们已限制它包含的是if-then-else的分量。类型FlowFcn表示从变量到格值的函数。数据流分析给每一个区域的入口点指定一个这样的函数。ComputeF_if_then_else()的参数r存放区域编号，结构的每一个分量都有这样一个编号。函数Region_No: integer × Component → integer返回给定区域的给定分量的区域编号。

247

bbs.theithome.com

复杂形式在数据流分析器中可用各种方法来表示。一种相对简单并具有时间和空间效率的方法是使用具有两种类型结点的图，其中一种类型的结点表示$in(B)$的值和F_B的值，另一种类型的结点表示合成“∘”、交“⊓”、并“⊔”、克林闭包“*”、非空闭包“+”，以及函数应用“()”等运算。注意，函数应用结点表示对其右子图应用它的左子图给出的函数，而合成运算结点表示它的左子图和右子图的合成。图8-15展示了图8-10中非正常区域B1a的部分方程的图形表示。这种表示使得可用一个简单的解释器作用于这些方程来分析它们所对应的区域。 248

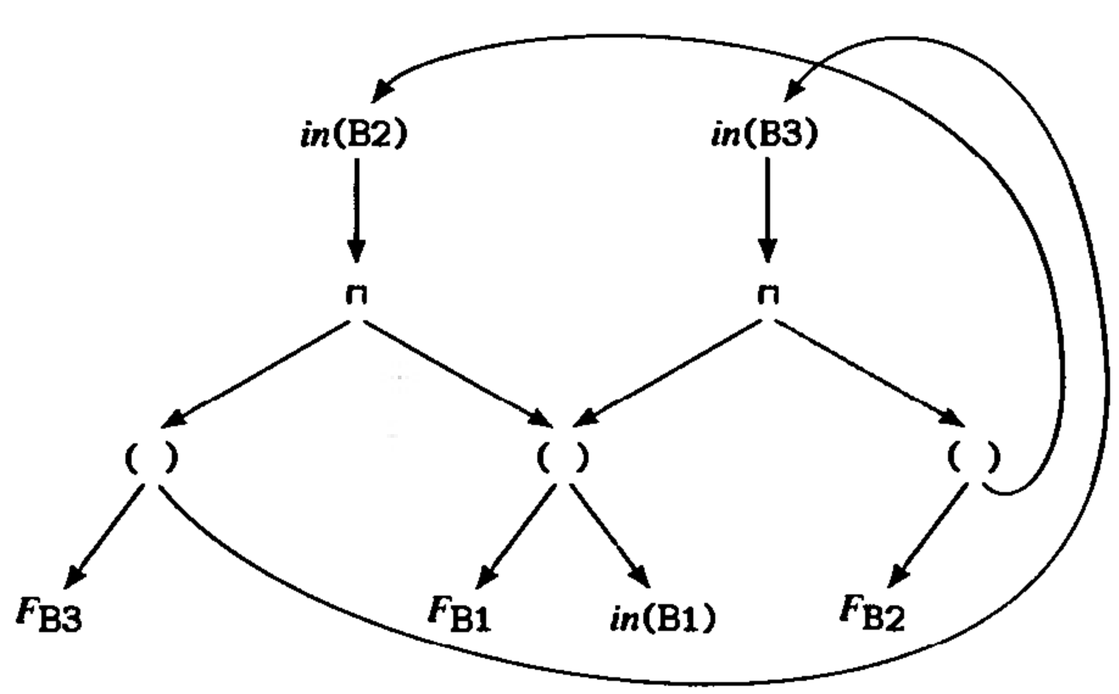


图8-15 图8-10中区域B1a的部分结构数据流方程的图形表示

注意，流图中有一部分需要经过解释来表示，这并不意味着它的所有部分都需要经过解释。例如，如果我们有一个简单的循环，其循环体是一个包含if-then-else结构的非正常区域，则我们可以执行关于该循环和if-then-else的代码，而对非正常区域使用解释器。

8.8 区间分析

我们已经建立了进行结构数据流分析的所有机制，现在执行区间分析就简单了：区间分析除了只有三种类型的区域之外，它与结构分析是相同的。这三种类型的区域是：一般无环区域、正常区域和非正常区域。

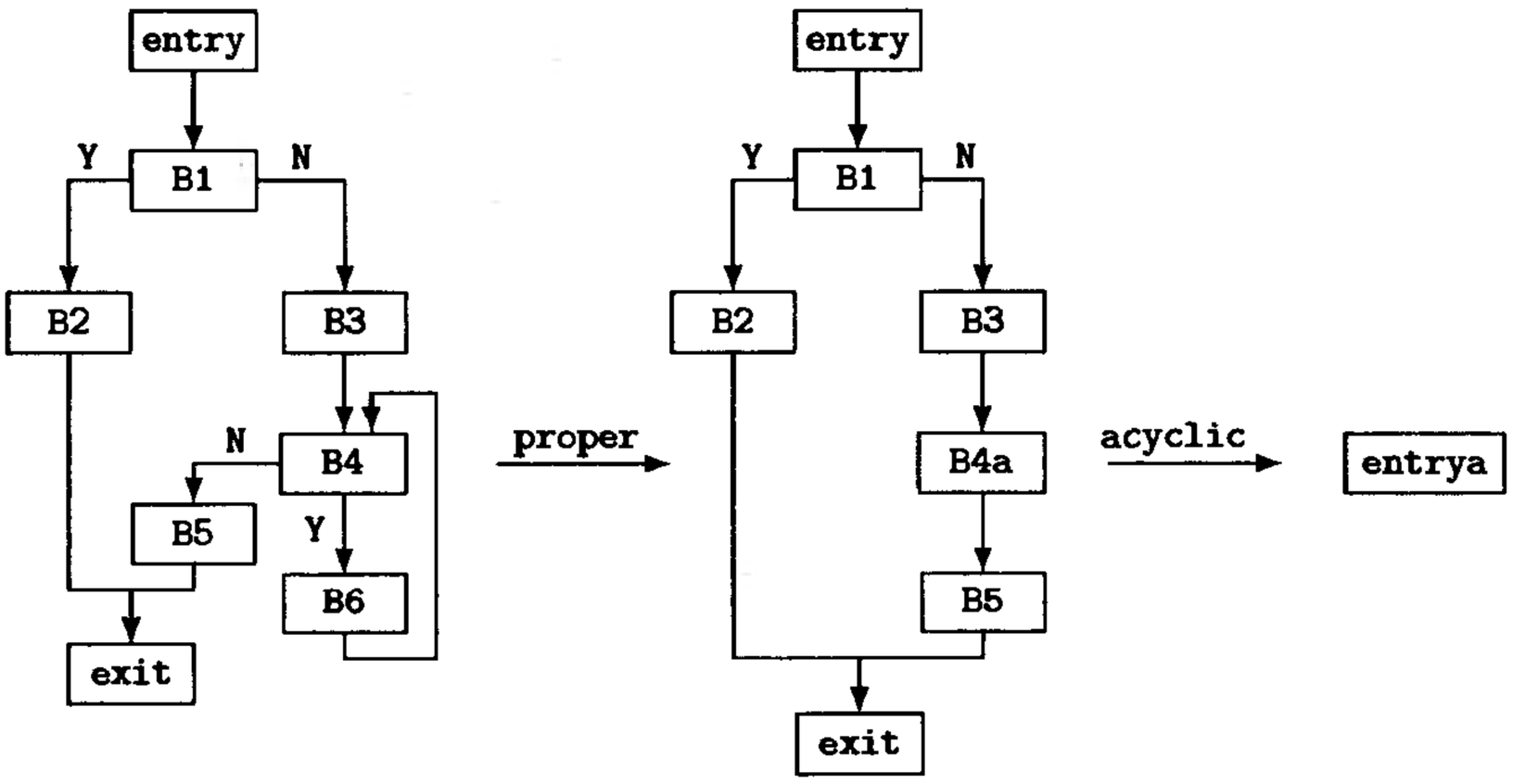


图8-16 我们的到达－定值例子的区间控制流分析

作为一个例子，考虑原来在图7-4中的流图，现在我们在图8-16中重新将它给出，同时还给出了对它的区间归约。第一步归约由B4和B6组成的循环到结点B4a，第二步归约第一步产

bbs.theithome.com

生的无环结构到单个结点entrya。对应的向前数据流函数如下:

$$
\begin{aligned}
F_{B4a} &= F_{B4} \circ (F_{B6} \circ F_{B4})^* \\
&= F_{B4} \circ (id \sqcap (F_{B6} \circ F_{B4})) \\
&= id \circ (id \sqcap (F_{B6} \circ id)) \\
&= id \sqcap F_{B6}
\end{aligned}
$$

249 $$F_{entrya} = F_{exit} \circ (F_{B2} \sqcap (F_{B5} \circ F_{B4a} \circ F_{B3})) \circ F_{B1} \circ F_{entry}$$

关于*in*()的方程如下:

$$
\begin{aligned}
in(\text{entry}) &= in(\text{entrya}) = Init \\
in(\text{B1}) &= F_{entry}(in(\text{entry})) \\
in(\text{B2}) &= F_{B1}(in(\text{B1})) \\
in(\text{B3}) &= F_{B1}(in(\text{B1})) \\
in(\text{B4a}) &= F_{B3}(in(\text{B3})) \\
in(\text{B4}) &= in(\text{B4a}) \sqcap (F_{B6} \circ F_{B4})^*(in(\text{B4a})) \\
&= in(\text{B4a}) \sqcap (id \sqcap (F_{B6} \circ id))(in(\text{B4a})) \\
&= in(\text{B4a}) \sqcap in(\text{B4a}) \sqcap F_{B6}(in(\text{B4a})) \\
&= in(\text{B4a}) \sqcap F_{B6}(in(\text{B4a})) \\
in(\text{B6}) &= F_{B4}(in(\text{B4})) \\
&= in(\text{B4}) \\
in(\text{B5}) &= F_{B4a}(in(\text{B4a})) \\
&= (F_{B4} \circ (id \sqcap (F_{B6} \circ F_{B4})))(in(\text{B4a})) \\
&= id(in(\text{B4a})) \sqcap id(F_{B6}(id(in(\text{B4a})))) \\
&= in(\text{B4a}) \sqcap F_{B6}(in(\text{B4a})) \\
in(\text{exit}) &= F_{B2}(in(\text{B2})) \sqcap F_{B5}(in(\text{B5}))
\end{aligned}
$$

得到的*in*()值如表8-4所示，它们与迭代分析和结构分析计算的*in*()值相同。

表8-4 区间分析为我们的到达－定值例子计算的*in* ()值

in(entry)	=	<00000000>
in(B1)	=	<00000000>
in(B2)	=	<11100000>
in(B3)	=	<11100000>
in(B4)	=	<11111111>
in(B5)	=	<11111111>
in(B6)	=	<11111111>
in(exit)	=	<11111111>

8.9 其他方法

Dhamdhere、Rosen和Zadeck介绍了一种新的数据流分析方法，他们称之为位置式
250 (slotwise)分析[DhaR92]。替代用长向量来表示变量或其他类型程序结构的数据流特征和用前面描述的方法之一对位向量进行运算，他们分开考虑向量中的每一个位置，并同时考虑所有位向量中相应的位置。也就是说，首先考虑通过一个过程时在所有位向量的第一个位置发生了什么

bbs.theithome.com

情况，然后考虑第二个位置，依此类推。对于某些数据流问题，这种方法没有什么用处，因为为了计算其他位向量中一个位置的值，它们需要结合来自两个以上位向量的不同位置的信息。但是对于诸如到达－定值和可用表达式等许多问题，在过程的特定点，每一个位置只与其他点的这个位置有关。进一步，例如对于可用表达式问题，在多数地方，大多数位置中的信息默认值是0(＝不可用)。这两者结合起来便使得位置式方法非常具有吸引力。

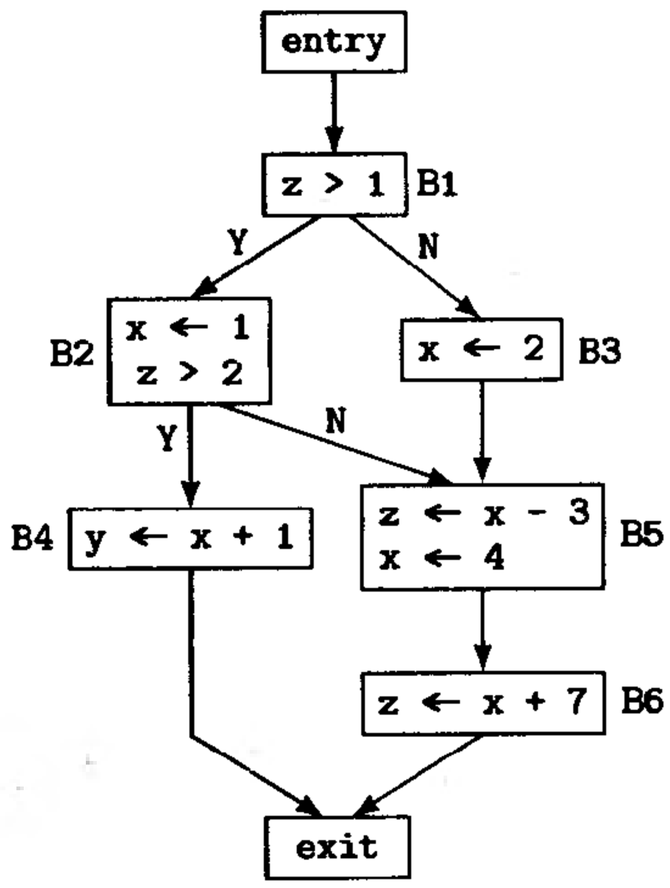
8.10 du链、ud链和网

du链（定值－使用链）和ud链（使用－定值链）是关于变量数据流信息的稀疏表示。一个变量的du链（du-chain）连接该变量的定值到它的所有可能流经到的使用，而ud链（ud-chain）连接一个使用到所有可能流经到该使用的定值。通过观察图8-5的例子可看出这两者之间的差别。基本块B2中关于x的定值的du链包括基本块B4和B5中的使用，而基本块B3中定值的x的du链只包括基本块B5中的使用。在基本块B4中关于x的使用的ud链只包含B2中的定值，而基本块B5中关于x的使用的ud链包含了B2和B3两处中的定值。

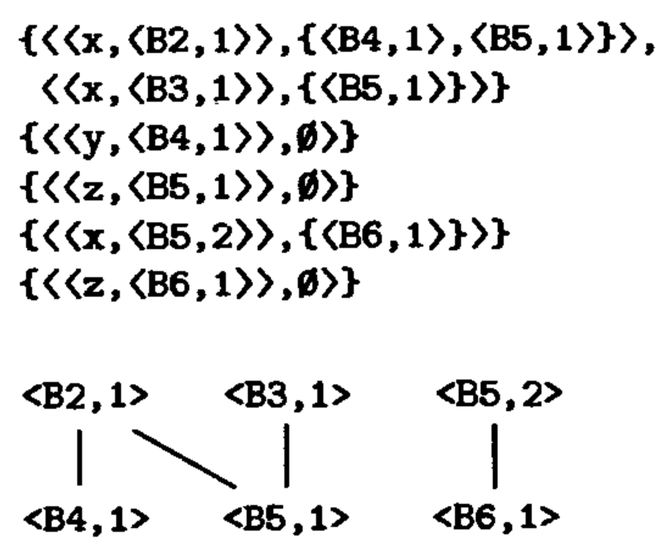
抽象地说，ud链或du链分别是从变量和基本块位置偶对到基本块位置偶对集合的函数，每一个使用或定值对应一个集合，具体地，它们一般用链表来表示。可以通过求解过程的到达－定值数据流问题，然后利用得到的信息建立链表来构造它们。一旦建立了链表，就可以释放到达－定值位向量，因为这些链表示的是相同的信息。对于我们的讨论，过程的du链和ud链可用ICAN类型UdDuChain来表示，其中

```
UdDu＝integer×integer
UdDuChain: (symbol×UdDu)→ set of UdDu
```

一个变量的网（web）是该变量的各个du链中相交的那些du链组成的最大并集。因此，图8-5中x的网包括了两个定值和两个使用，而在图8-17中，存在着关于x的两个网，一个由B2和B3的定值以及B4和B5中的使用组成，另一个包含B5中x的定值和B6中x的使用。网对图着色全局寄存器分配（见第16章）特别有用——它们是寄存器分配的候选对象。 251



a)



b)

图8-17 a) 用于网结构的例子，b) 其中的网

注意，过程中有时会重复使用某些变量的名字，比如说变量i经常作为循环索引变量被重复使用，但是这些使用之间相互是不相交的，构造网的作用正是为了分开这种变量的使用。这

bbs.theithome.com

样通过缩小变量占用寄存器的范围，可以显著地改善寄存器的分配，并且改善其他优化的效果。特别是那种作用于单个变量或作用于程序的有限正文范围的优化，如作用于归纳变量的强度削弱，可以从中得到好处。

8.11 静态单赋值形式

静态单赋值是一种相对较新的中间代码表示，它能有效地将程序中运算的值与它们的存储位置分开，从而使得若干优化能具有更有效的形式。

如果在某个过程内赋值的每一个变量作为赋值的目标只出现一次，我们说这个过程是静态单赋值(SSA)的形式。在用SSA形式表示的过程中，du链是显式的：变量的使用可能用到一个特定定值产生的值，当且仅当在该过程的SSA形式中此变量的定值和使用具有完全相同的名字。这简化了若干种优化转换，包括常数传播、值编号、不变代码外提和删除、强度削弱和部分冗余消除，从而提高了这些优化的效率。因此，值得将一个过程的给定表示转换成SSA形式，并对SSA形式进行操作，然后在适当的时候将SSA形式转换回原来的形式。

在转换成SSA形式的处理过程中，标准的方法是给每一个赋值的变量带上一个下标，并

252 在诸如图8-18中B5的入口点这样的汇合点使用所谓的ϕ函数，以区分对一个变量的多种赋值。每一个ϕ函数具有的参数个数同汇合到那一点的该变量的不同版本个数一样多，并且每一个参数与该点的一个特定控制流前驱相对应。因此图8-17例子的标准SSA形式的表示如图8-18所示。变量x已经被分成4个变量x_1、x_2、x_3和x_4，而z已被分成z_1、z_2和z_3，这些变量的每一个都只赋值一次。

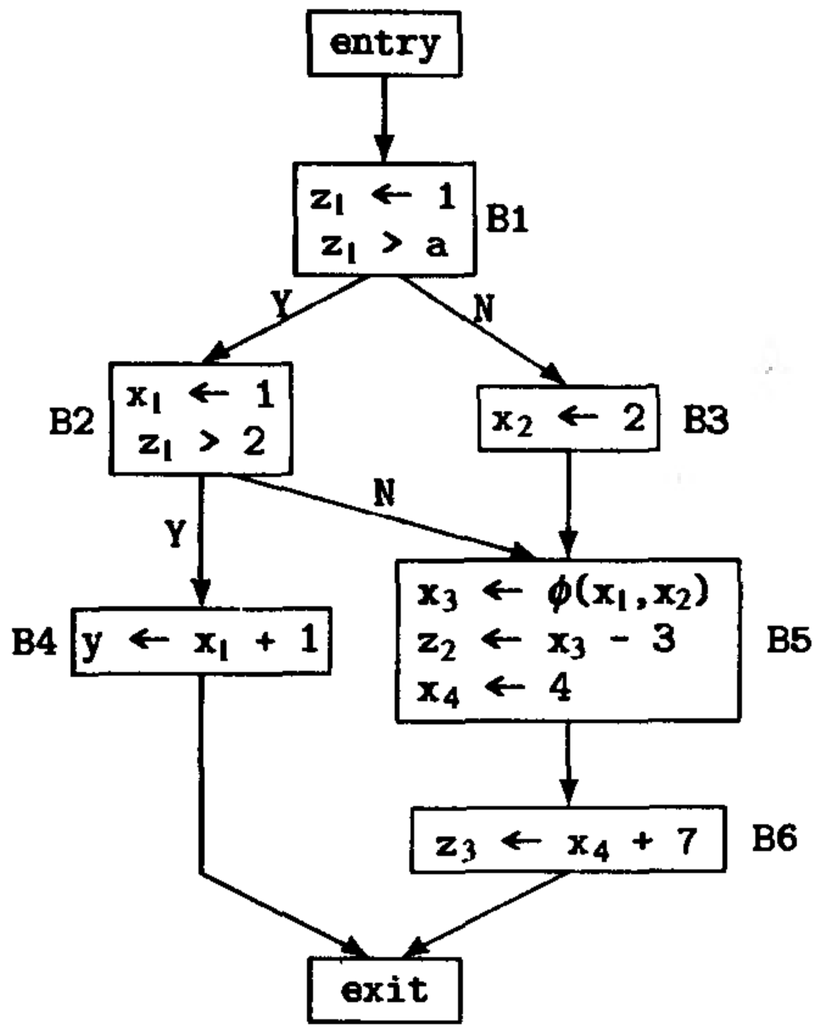


图8-18 图8-17中的例子到SSA形式的标准转换

转换到SSA形式的处理过程首先找出要插入ϕ函数的汇合点，然后插入简单的ϕ函数（即形式为$\phi(x,x,\ldots,x)$的函数），其中，参数的个数等于该变量的定值到达此汇合点的控制流前驱的个数，然后，重新命名这些变量的定值和使用（习惯上是将它们带上下标）来建立静态单赋值性质。一旦我们在转换成SSA形式之后完成了需要做的事情，就需要删除ϕ函数，因为它们只是一个概念上的工具，并不具有计算作用——即，当我们在执行一个过程来到一个具有ϕ函数的汇合点时，我们无法确定是从哪一个分支到达这一点的，因此无法确定使用的是哪一个值[⊖]。

转换一个过程到最小SSA形式，即ϕ函数个数最少的SSA形式，可以利用所谓必经边界来

253 实现。对于流图的一个结点x，x的必经边界（dominance frontier）记做$DF(x)$，它是流图中所有满足后面这个条件的结点y的集合：x是y的直接前驱结点的必经结点，但不是y的严格必经结点，即，

$$DF(x) = \{y \mid (\exists z \in Pred(y) \text{使得} x\ dom\ z) \text{并且} x\ !sdom\ y\}$$

对所有x直接计算$DF(x)$会使计算量为流图中结点个数的二次方。将它分解成对如下两个中间分

⊖ 一种称为门控单赋值形式（gated single-assignment form）的SSA扩充形式，在每一个ϕ函数中包含了一个选择符，这个选择符根据到达汇合点的路径来指出选择哪一个位置。

bbs.theithome.com

量$DF_{local}(x)$和$DF_{up}(x, z)$的计算，可得到一种线性的算法。

$$DF_{local}(x) = \{y \in Succ(x) \mid idom(y) \neq x\}$$

$$DF_{up}(x, z) = \{y \in DF(z) \mid idom(z) = x \ \&\ idom(y) \neq x\}$$

并且$DF(x)$的计算变为:

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in N\,(idom(z) = x)} DF_{up}(x, z)$$

为了计算给定流图的必经边界，我们将上面的方程转换成如图8-19所示的代码。函数值IDom(x)是x是其直接必经结点的结点集合（即，$idom(x) = y$当且仅当$x \in$ IDom(y)），并且在完成时，$DF(x)$包含x的必经边界。函数Post_Order(N, $IDom$)返回一个结点序列，此序列是由N和$IDom$表示的必经结点树的后序遍历。

```
IDom, Succ, Pred: Node ⟶ set of Node

procedure Dom_Front(N,E,r) returns Node ⟶ set of Node
    N: in set of Node
    E: in set of (Node × Node)
    r: in Node
begin
    y, z: Node
    P: sequence of Node
    i: integer
    DF: Node ⟶ set of Node
    Domin_Fast(N,r,IDom)
    P := Post_Order(N,IDom)
    for i := 1 to |P| do
        DF(P↓i) := ∅
        || compute local component
        for each y ∈ Succ(P↓i) do
            if y ∉ IDom(P↓i) then
                DF(P↓i) ∪= {y}
            fi
        od
        || add on up component
        for each z ∈ IDom(P↓i) do
            for each y ∈ DF(z) do
                if y ∉ IDom(P↓i) then
                    DF(P↓i) ∪= {y}
                fi
            od
        od
    od
    return DF
end    || Dom_Front
```

图8-19 计算流图必经边界的代码

现在我们为流图结点集合S定义S的必经边界为:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

以及迭代的必经边界（iterated dominance frontier）DF^+ ()为:

$$DF^+(S) = \lim_{i \to \infty} DF^i(S)$$

bbs.theithome.com

其中$DF^1(S) = DF(S)$并且$DF^{i+1}(S) = DF(S \cup DF^i(S))$。如果$S$是给变量$x$赋值的结点外加entry结点组成的集合，则$DF^+(S)$就正好是与$x$有关的需要$\phi$函数的结点集合。

为了计算流图的迭代的必经边界，我们将上面给出的方程改编成如图8-20所示。假定对于流图中所有的结点x，我们已预先计算好了DF(x)，则DF_Plus(S)是结点集S的迭代的必经边界。计算$DF^+(S)$的这种实现在最坏的情况下具有的时间上界是该过程大小的二次方，但在实际中通常成正比。

```
procedure DF_Plus(S) returns set of Node
   S: in set of Node
begin
   D, DFP: set of Node
   change := true: boolean
   DFP := DF_Set(S)
   repeat
      change := false
      D := DF_Set(S ∪ DFP)
      if D ≠ DFP then
         DFP := D
         change := true
      fi
   until !change
   return DFP
end    || DF_Plus

procedure DF_Set(S) returns set of Node
   S: in set of Node
begin
   x: Node
   D := ∅: set of Node
   for each x ∈ S do
      D ∪= DF(x)
   od
   return D
end    || DF_Set
```

图8-20 计算流图的迭代必经边界的代码

作为一个转换到最小SSA形式的例子，考虑图8-21中的流图。它的必经结点树如图8-22所示。利用前面给出的必经边界的迭代特征，我们对变量k计算出：

$$DF^1(\{\text{entry}, \text{B1}, \text{B3}\}) = \{\text{B2}\}$$

$$DF^2(\{\text{entry}, \text{B1}, \text{B3}\}) = DF(\{\text{entry}, \text{B1}, \text{B2}, \text{B3}\}) = \{\text{B2}\}$$

对于i有：

$$DF^1(\{\text{entry}, \text{B1}, \text{B3}, \text{B6}\}) = \{\text{B2}, \text{exit}\}$$

254
~
255

$$DF^2(\{\text{entry}, \text{B1}, \text{B3}, \text{B6}\}) = DF(\{\text{entry}, \text{B1}, \text{B2}, \text{B3}, \text{B6}, \text{exit}\}) = \{\text{B2}, \text{exit}\}$$

对于j(与k相同)有：

$$DF^1(\{\text{entry}, \text{B1}, \text{B3}\}) = \{\text{B2}\}$$

$$DF^2(\{\text{entry}, \text{B1}, \text{B3}\}) = DF(\{\text{entry}, \text{B1}, \text{B2}, \text{B3}\}) = \{\text{B2}\}$$

所以，B2和exit是需要ϕ函数的结点。具体地，B2对于每一个i、j和k需要一个ϕ函数，exit需要关于i的一个ϕ函数。为这些变量添加下标并插入ϕ函数后的结果如图8-23所示。注意，

bbs.theithome.com

插入到exit块的ϕ函数不是真正需要的（除非i在过程出口是活跃的）；修剪过的SSA形式（pruned SSA form）删除了这种不必要的ϕ函数，但以稍微增加计算量为代价。

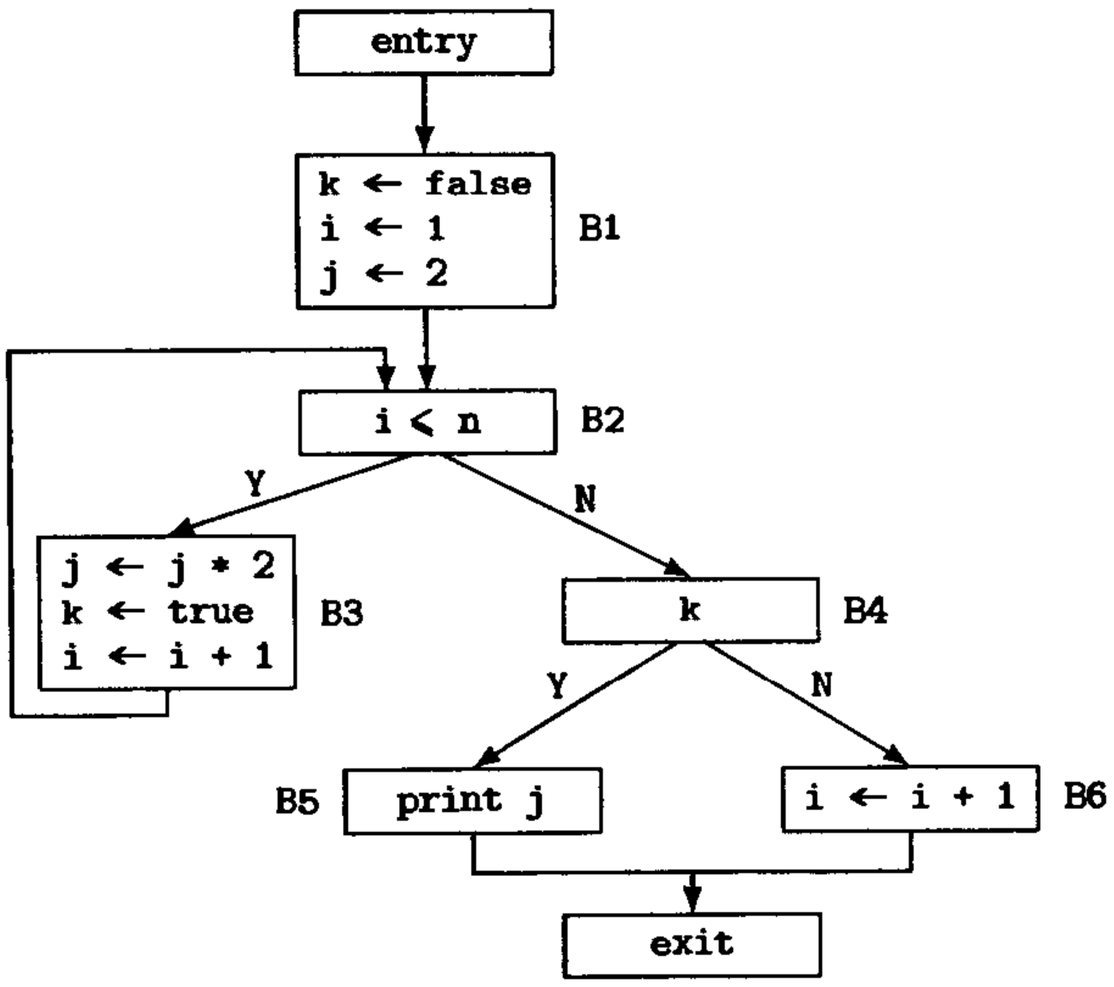


图8-21 要转换到最小SSA形式的流图例子

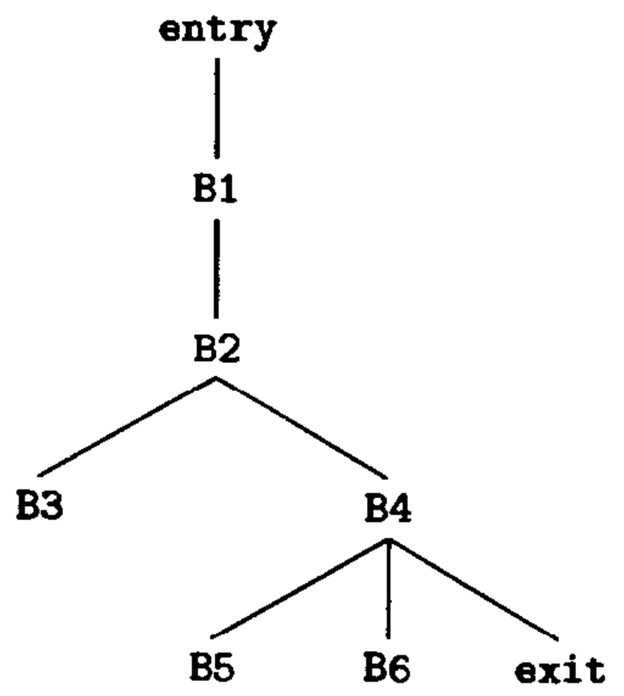


图8-22 图8-21中流图的必经结点树

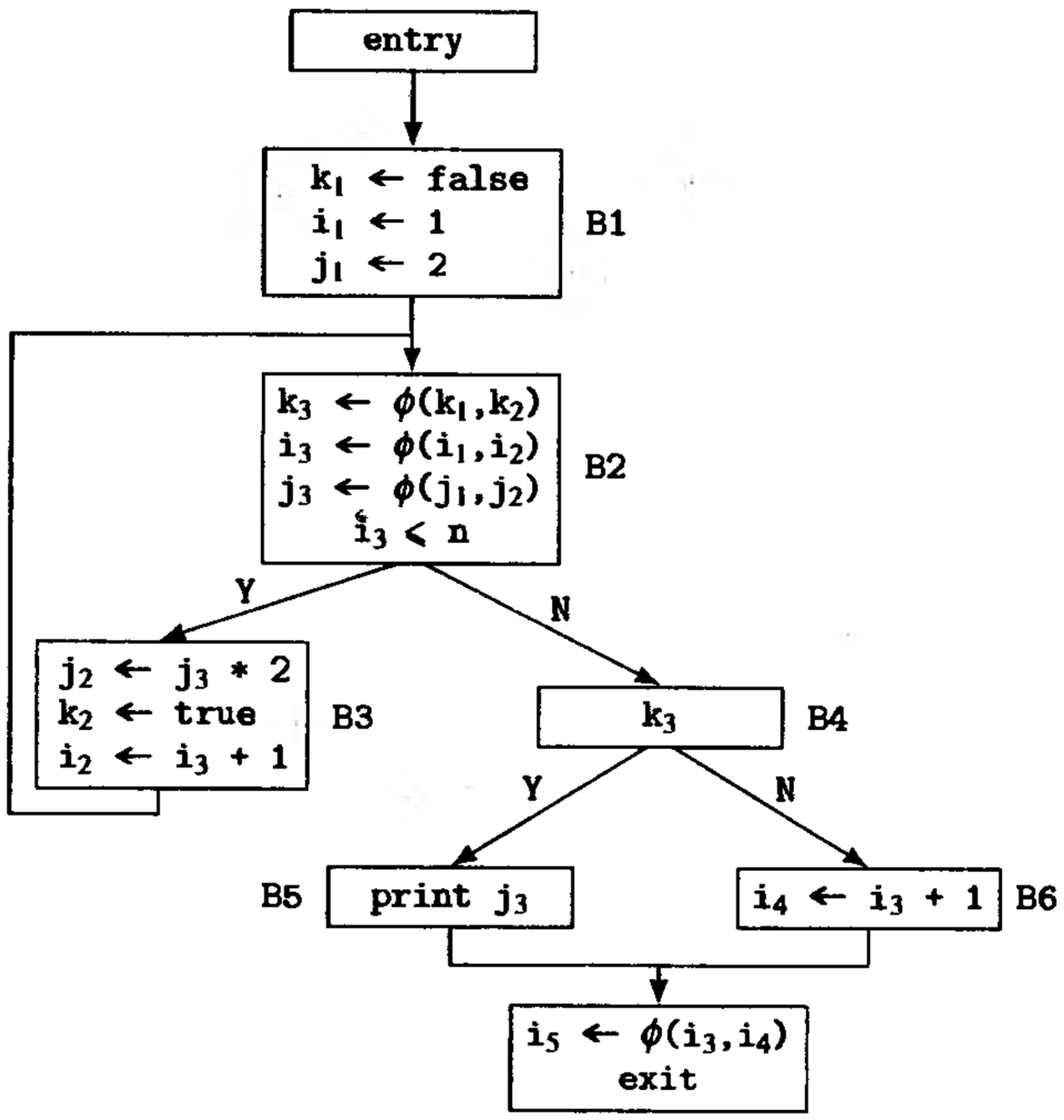


图8-23 转换图8-21中的流图到最小SSA形式的结果

8.16节的文献给出了关于转换到最小SSA形式并从它转换回到原形式的一些有效算法。关于转换到SSA形式产生的影响，也有一些实验数据[CytF91]：在221个Fortran 77过程中，经过到SSA形式的转换后，其赋值个数是原来程序赋值个数的1.3~3.8倍。偶尔出现程序大小有很大的增加是在使用SSA形式时必须注意的情况，但是，在已经知道SSA形式能使优化具有好得多的效果时，即使有这种情况存在，使用SSA形式也是值得的。

bbs.theithome.com

8.12 数组、结构和指针的处理

迄今为止，在关于数据流分析的讨论中，我们涉及的只是简单常数和其值是简单常数的变量，一直没有涉及到比它们更复杂的情况。因为变量（在某些语言中还有常数）也可以是数组、记录、指针等，因此我们必须考虑如何将它们纳入到我们的数据流分析框架中来，以及在SSA形式中如何表示它们。

许多编译器的选择是简单地忽略对数组和记录的赋值，并悲观地将它们看成是通过指针的赋值。这种方法假定指针可以指向任意变量，因此，通过指针的赋值可能隐含地影响所有的变量。像Pascal这样的语言对这个问题可以提供某些帮助，因为它们限制指针所指的对象只能是其类型已声明是指针指向的对象。用一种能产生更有用信息的方式处理指针需要有别名分析，我们将在第10章专门讨论它。将堆看成是类似于数组的一个较大的对象，并且假定任何通过指针的赋值都可以访问和改变堆中的所有对象，则可以保守地模拟指向堆存储空间的指针。涉及指针和记录的较为激进的技术在后面8.14节讨论。

C语言中没有限制指针只能指向堆存储空间——它们也可以指向栈中的对象和静态分配的对象。第10章讨论的别名分析方法对于C程序的激进优化尤其重要。

有些语言允许数组赋值同时对数组的所有元素进行操作。通过将这种数组变量和常量看成是普通变量和常数，这种数组赋值很容易处理。但是，多数情况下数组赋值是对单个数组元素的赋值而不是对整个数组，例如，A[3]←5或A[i]←2。对一个已知数组元素的赋值可以看成是普通赋值，但这仍然不能解决大多数数组操作的情况。处理带变量名的数组元素赋值的一种方法是，将它们转换为使用访问和更新的赋值形式，如图8-24所示，使得它们看起来好像是对整个数组进行操作。尽管这种操作能使我们的数据流分析算法正确工作，但通常它们产生的信息过于粗糙，以至于对优化数组操作不十分有用。通常的选择是对数组操作做依赖分析（参见9.1节），它可以提供有关数组的更精确信息，但代价是增加了相当多的额外计算。最近有论文介绍了一种相对较新的对数组元素进行数据流分析的方法，这种方法称为最近写树（last-write tree）方法（见8.16节）。

256 ~ 258

```
x ← a[i]                    x ← access(a,i)
a[j] ← 4                    a ← update(a,j,4)
```

a) b)

图8-24 涉及数组元素的赋值以及它们到访问/更新的形式转换

在多数语言中，直接(而不是通过指针)引用记录元素的赋值只可以使用记录的成员名，因为成员名是常数，因此对记录的赋值可以看成是对整个记录的访问或更新，如同上面建议的对数组的处理那样；或者看成是对单独成员的访问或更新。如果程序中频繁使用记录，后一种方法可以产生更为有效的优化，我们将在第12章讨论这种方法。如果源语言允许使用变量来选择记录的成员，则这种记录本质上是其元素具有符号名的固定大小的数组，并且可以同其他数组一样进行处理。

8.13 数据流分析器的自动构造

对于给定的某种类型的中间代码和数据流问题，已经实现了若干工具，它们能够构造出解给定数据流问题的数据流分析器。只要使用的是特定的中间代码，就可用这种工具来构造供优化使用的数据流分析器。

第一个著名的这类分析器构造器是由Kildall [Kild73]开发的。他的系统构造一个迭代分析器，这个分析器对所谓的“池”（pool）进行操作。这里的“池”，用现在的说法就是在分析期

bbs.theithome.com

超星公司制作请尊重作者版权

间某个时刻过程内特定点所具有的数据流信息。Kildall给出了一种表示池的表格形式和规则，这些规则与被处理的流图结点的类型和要解的数据流问题相对应，将“输入池”转换到对应的“输出池”。他的系统允许将池表示成位向量、链表或值编号（参见12.4节），具体选择取决于数据流问题，并执行与8.4节介绍的方法类似的一种工作表方式的迭代分析。

较新也较成熟的分析器构造器是Tjiang和Hennessy的Sharlit [TjiH92]。除了执行数据流分析之外，这个分析器构造器还可以用来指定优化转换。在Sharlit中，为指定一个分析器和优化器所需要写的多数内容都是纯粹的说明，但是其他部分，如优化转换，需要写过程代码，这种 259
代码用C++ 编写。这种分析器构造器对称为SUIF(参见C.3.3节)的四元式中间代码进行操作。它感兴趣的四元式类型是取、存、具有两个操作数并放置结果到一目标操作数的二元运算，以及指定控制流的其他操作。一般数据流分析器由两部分组成，局部部分用于分析基本块，并通过基本块传播信息；全局部分对基本块组成的流图进行处理。与需要数据流分析器由这两部分组成的做法不同，Sharlit允许分析器的说明按编译器书写者希望的工作粒度指定不同的操作粒度。你可以对独立的中间代码结点进行操作，或将这些中间代码组合成基本块来操作，也可以对区间或其他结构化的单位进行操作——就看哪一种方式更适合你手头的问题。

Sharlit执行数据流分析所用的底层技术是路径简化，它基于Tarjan的路径问题快速算法，这个算法计算结点名字的正则表达式，称为路径表达式（path expression）。路径表达式指明从一个结点到其他结点或经过其他结点的所有可能的路径。例如，对于图7-32a的流图，从entry经过基本块B5的路径表达式是

```
entry · (B1 · B2)* · B3 · B5
```

从B3经过exit的路径表达式是

```
B3 · (B4+(B5 · (B6 · B7)*)) · exit
```

其中，“·”运算符表示连接，“+”表示路径的交汇，“*”表示重复。这些运算符在数据流分析器中分别表示为流函数的合成、交和不动点计算。

为了给Sharlit指定一个优化器，必须指定下面的内容:

1. 它所操作的流图的描述（包括选择是以单个四元式还是以基本块作为分析的单位）；

2. 要使用的数据流值集合，每一个值与问题中流图的每一个结点相连（它可以是位向量、对变量常数值的赋值，或其他对象）；

3. 描述结点数据流值效果的流函数;

4. 若干动作例程，这些例程指明利用这种数据流分析结果要执行的优化转换。

在给定了要解的数据流问题是如何解释流图、值和函数的描述，以及一个需要进行分析的过程后，Sharlit计算流图各个分量的路径表达式集合，然后使用流函数和路径表达式来计算构成解的数据流值。之后，它使用结果值和动作例程来遍历流图并执行可应用的各种转换。

Tjiang和Hennessy用三个例子说明如何用Sharlit来计算可用表达式（一个是对由结点组成的流图进行的迭代分析，一个是计算基本块的流函数的局部分析，另一个是区间分析）。他们 260
的结论是，这个工具极大地简化了优化器的说明和调试，并且它所产生的结果可以与由商业编译器手工生成的优化器相竞争。

8.14 更贪婪的分析

迄今为止我们还只考虑了相对较弱的数据流分析形式，对于它们的分析能力以及它们与诸如程序性质和正确性验证等强有力的分析方法之间的密切关系则几乎没有提及。这一节探讨我

bbs.theithome.com

超星公司制作 请尊重作者版权

们所使用的格的复杂性，以及我们允许自己具有的对程序中执行的操作进行推理的能力对我们能够确定的性质将有怎样的影响。

考虑对图8-25的简单例子做常数传播分析。如果将算术运算，例如这里出现的i+1，都看成是未经解释的——即如果假定我们完全没有关于其运算效果的信息——则我们就无法确定j的值在B4的入口点是否是常数。另一方面，如果我们加强常数传播分析使它能够做常数加运算，则我们就很容易确定在B4的入口处j的值为2。

在图8-26的例子中，假定我们能够将减1然后与0进行比较的两个运算结合起来考虑，并且能够区别这个测试的两个出口“Y”和“N”，则可以断定，在exit基本块的入口处n的值< 0。如果扩充这个程序如图8-27所示，则可以断定在exit基本块的入口n=0。进一步，如果我们能够对整数函数进行推理，则我们也能确定在相同点，如果$n_0 > 0$，则$f = n_0!$，其中n_0表示n在流图入口处的值。这至少使我们想到能够使用数据流分析技术进行程序证明。为了做程序证明，我们需要给每个基本块的每一个出口处相连的数据流信息是如表8-5所示的归纳断言。尽管这需要更多的机制，并且它的计算比我们在一个编译器中实际可能用到的任何分析都要复杂得多，但它至少说明了数据流分析包括的各种可能应用中的一种应用。

261

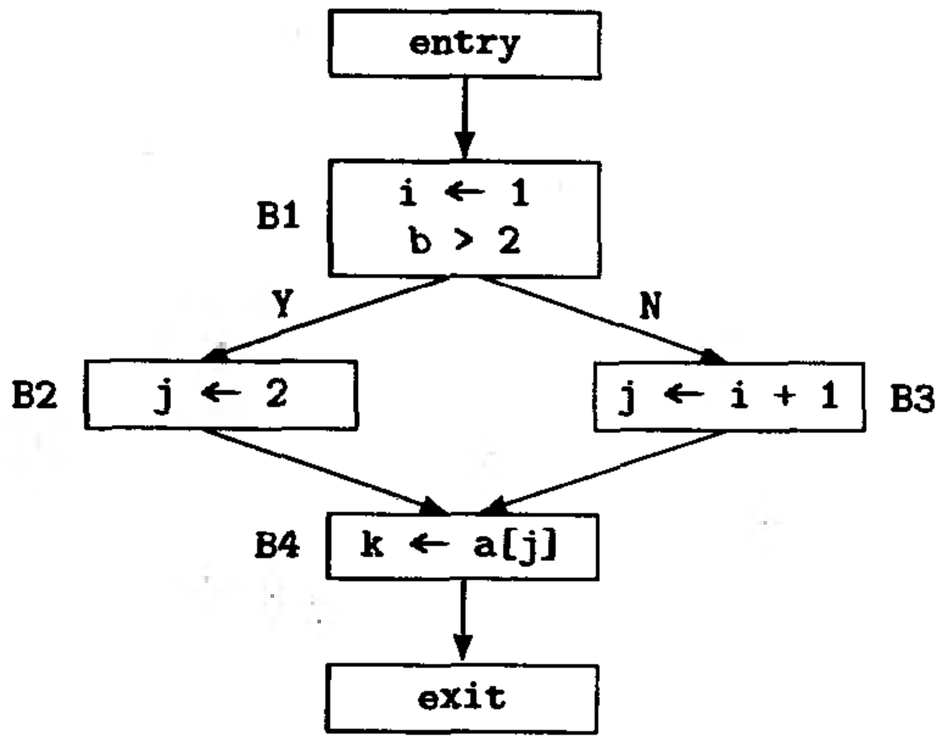


图8-25 常数传播分析的简单例子

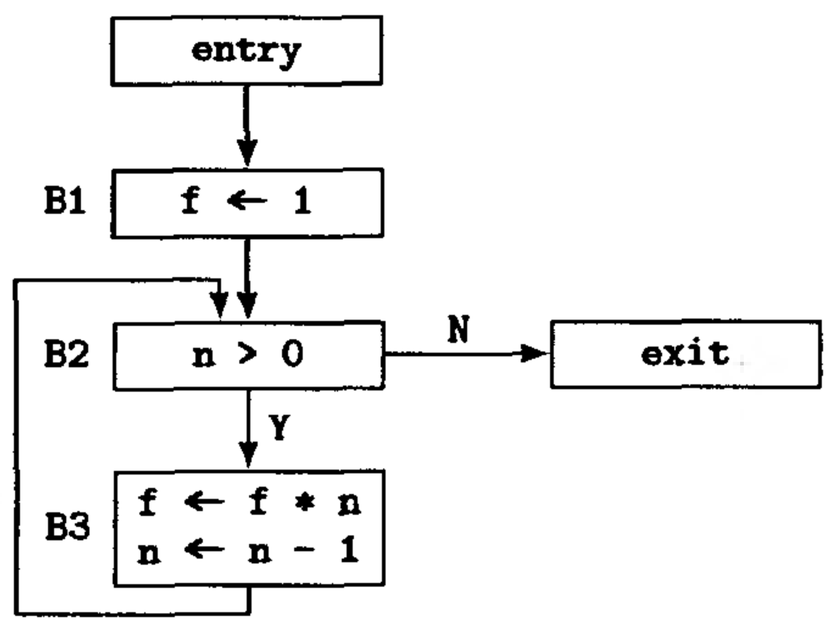


图8-26 简单的阶乘计算

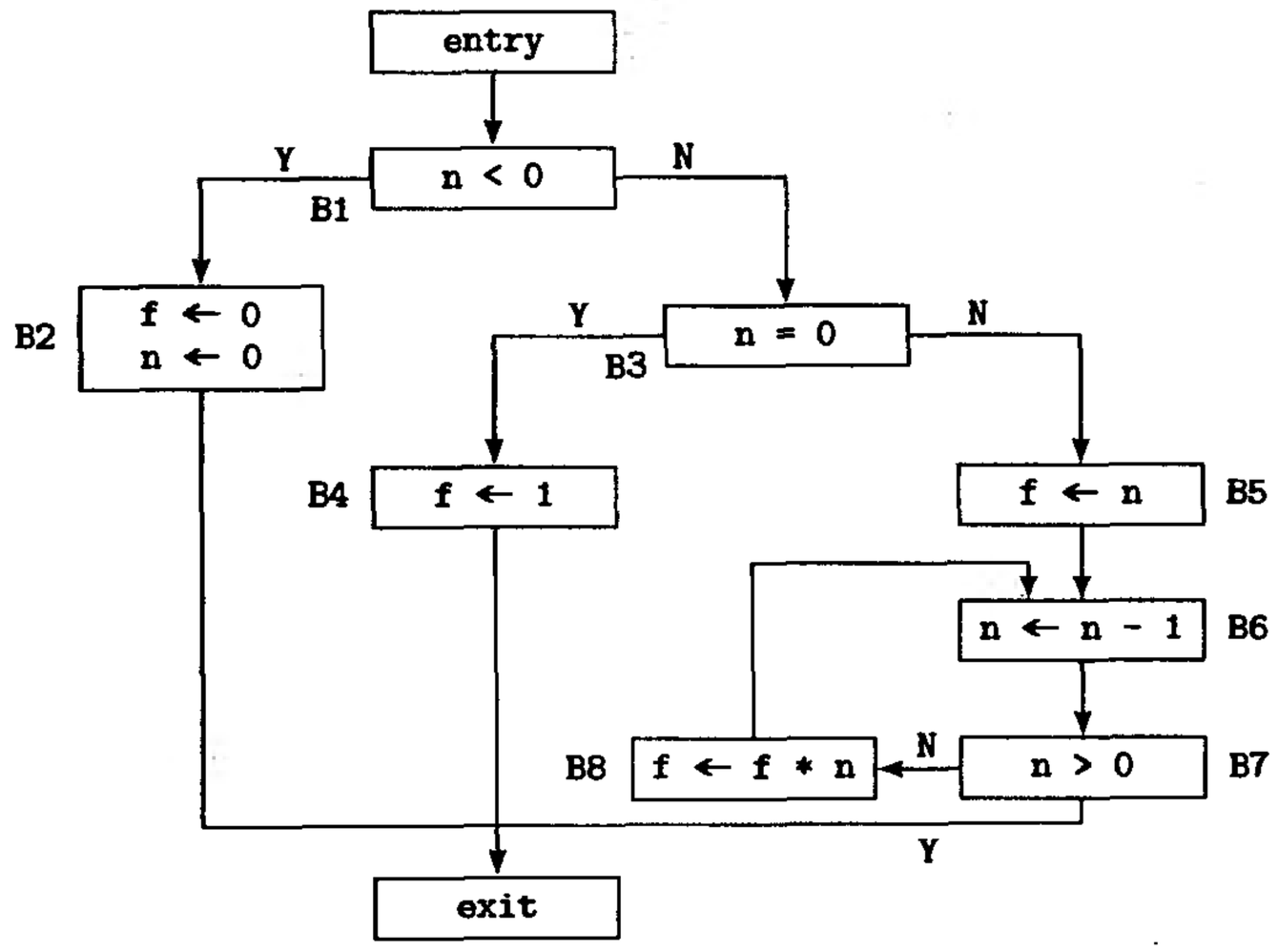


图8-27 完整的阶乘计算

bbs.theithome.com

表8-5 与图8-27的每一个基本块出口相连的归纳断言，这些归纳断言对于确定该流图计算的是整数阶乘函数是必要的

块	归纳断言
entry	$n = n_0$
B1/Y	$n = n_0 < 0$
B1/N	$n = n_0 \geq 0$
B2	$n = 0$ and $f = n_0!$
B3/Y	$n = n_0 = 0$ and $f = 1$
B3/N	$n = n_0 > 0$
B4	$n = n_0 = 0$ and $f = n_0!$
B5	$n = n_0 > 0$ and $f = n_0 \cdot$
B6	$n \geq 0$ and $f = n_0 \times (n_0 - 1) \times \cdots \times (n + 1)$
B7/Y	$n > 0$ and $f = n_0 \times (n_0 - 1) \times \cdots \times (n + 1)$
B7/N	$n = 0$ and $f = n_0 \times (n_0 - 1) \times \cdots \times 1 = n_0!$
B8	$n > 0$ and $f = n_0 \times (n_0 - 1) \times \cdots \times n$

另一个计算代价非常大，但比程序证明的代价要小一点的数据流分析的例子是包含记录和指针的数据结构的类型判定和依赖关系分析。这个问题自1968年以来就一直有不少人在进行研究，并且仍然没有完全令人满意的答案。已经提出了各种方法，但是所有的方法都归于这三个主要类：基于文法的方法、在Jones和Muchnick[JonM81a]论文中定义的k受限图方法，以及9.6节将简单介绍的Hendren等人的访问路径方法。

8.15 小结

数据流分析提供关于一个过程（或一大段程序）如何处理其数据的全局信息。如果将优化比作磨坊的话，数据流分析就好比是给磨坊提供谷物。数据流分析的范围可以从分析过程的抽象执行，即确定过程计算的是某个特定的函数，到较简单和较容易的分析，如到达-定值分析。但是，同控制流分析一样，数据流分析存在着三种主要的方法，每一种方法与一种控制流方法相关。与必经结点和回边相关的方法叫做迭代数据流分析，其他两种方法与对应的控制流方法有相同的名称，即区间分析和结构分析。同控制流分析类似，在使用它们时也需要权衡它们各自的长处和不足。

不论采用什么方法，我们都必须保证数据流分析给出的是正确的和保守的信息，即给出的信息不能误解被分析过程的行为。它不能告诉我们执行某种代码转换是安全的，而事实上却是不安全的。我们必须仔细地设计数据流方程，并确信我们计算出来的关于它们的解，如果不是所预期信息的精确表示，也至少是它的保守近似表示，以此来保证这一点。例如，对于到达-定值分析，它判定变量的哪些定值可以到达一个特定的使用点，如果存在到达这种使用的定值，数据流分析一定不能告诉我们没有定值到达这个使用。如果它给我们的到达-定值集合大于或等于过程实际产生的最小到达-定值集合，则这种分析是保守的。

但是，为了从优化中获得最大的效益，我们的目的是使得数据流问题既是保守的，同时又尽可能是激进的。因此，我们将总是在既使我们计算的信息尽可能激进，但又是保守的之间走钢丝，以便从这些分析和我们执行的代码改善转换中得到最大的效益，而又不至于把正确的代码转换成不正确的代码。

262 ~ 263

bbs.theithome.com

8.16 进一步阅读

使用*KILL*()函数而不是用*PRSV*()的著作的一个例子是[AhoS86]。

[JonM81b]介绍了Jones和Muchnick为了描述类LISP数据结构的“形状”所使用的格。

对于涉及单调函数的数据流分析，可能不存在计算关于它的满足所有路径解的算法的证明可以在[Hech77]和[KamU75]中找到。Kildall的结果（即，对于其中所有流函数都是可分配的数据流问题，一般迭代算法计算的是MFP解，在这种情况下，MFP与MOP解相同）是从[Kild73]中找到的。Kam和Ullman对单调但非可分配函数的部分推广是从[KamU75]中找到的。

将数据流信息与流图中的边相连的论文包括[JonM76]、[JonM81a]和[Rose81]。其中第二篇论文也描述了有关系的和属性无关两者之间的不同。

Morel和Renvoise的关于部分冗余删除的原始公式是在[MorR79]中找到的，Knoop、Rüthing和Steffen的较新的部分冗余删除公式在[KnoR92]中给出。Khedker和Dhamdhere[KheD99]讨论了双向数据流分析的计算复杂性。

8.3节介绍的解数据流问题的方法来自下面的论文:

方　法	引　文
Allen的强连通区域方法	[Alle69]
Kildall的迭代算法	[Kild73]
Ullman的*T*1-*T*2分析	[Ullm73]
Kennedy的结点列表算法	[Kenn75]
Farrow、Kennedy和Zucconi的图形文法方法	[FarK75]
消去法，例如区间分析	[AllC76]
Rosen的高级（语法制导的）方法	[Rose77]
结构分析	[Shar80]
位置式分析	[DhaR92]

如果*A*是流图中任意无环路路径上回边的最大条数，如果我们使用逆后序遍历次序，则迭代算法只需要通过`repeat`循环*A*+2遍是Hecht和Ullman在[HecU75]中证明的。[Hech77]中讨论了管理工作表的各种可选方法，包括循环法和不同结点列表方法。

位置式分析的最早文章是 [Kou77]。[DhaR92]说明了如何将位置式分析应用于部分冗余分析。

[CytF89]和[CytF91]介绍了静态单赋值形式，这种形式起源于Shapiro和Saint[ShaS69]较早
264 开发的一种形式。线性时间的必经边界算法出现在[CytF91]中，同时给出的还有转换至SSA形式和从SSA形式转换回原形式的有效方法。

利用最近写树方法进行数组元素的数据流分析是[Feau91]介绍的。

[TjiH92]和[Tjia93]中介绍了Tjiang和Hennessy的Sharlit，它对SUIF中间表示进行操作。[TjiW91]描述了SUIF中间表示(关于如何下载SUIF，参见附录C)。[TjiH92]给出了使用Sharlit的三个例子。

关于如何应用数据流分析到递归数据结构的研究论文包括[Reyn68]、[Tene74a]、[Tene74b]、[JonM81a]、[Laru89]、[Hend90]、[ChaW90]、[HumH94]、[Deut94]等。[HumH94]中使用的方法将在9.6节讨论。

8.17 练习

8.1 在使用位向量或集合运算、并采用迭代法计算双向数据流信息中，同时使用(a)*in*()和*out* ()函数，与(b)只使用两者之一的复杂性如何?

bbs.theithome.com

8.2 给出一个不是强可分配格的例子。

8.3 给出一个数据流问题的具体例子，并给出它的MOP和MFP解各不相同的实例。

8.4 评估将数据流信息与流图的边相连，和与结点入口和出口相连的空间和时间的复杂性。

RSCH 8.5 如[JonM81a]所介绍的那样，将图8-4的数据流分析形式化地表示为关系问题，其结果和使信息与边相连的方法一样好吗？计算的复杂性如何？

RSCH 8.6 研究用于数据流分析的Kennedy的结点列表算法[Kenn75],或Farrow、Kennedy和Zucconi的图形文法方法[FarK75]。与本章讨论的这些方法相比，它们的优点和缺点是什么？

8.7 还有哪些可选方法能用来管理图8-6中的工作表？在实现的难易程度和计算的效率方面，它们与向前问题的逆后序方法有何比较？

8.8 画出从$\mathbf{BV}^2$到它自身的单调函数格。

ADV 8.9 $\mathbf{L}^{\mathbf{F}}$对任意L是可分配的吗？如果不是，给出一个例子；如果是，给出证明。

RSCH 8.10 如[Zade84]中所讨论的那样，研究数据流信息随流图的改变而更新的情况。有哪些明显的原因表明值得在编译器中这样做，而不是当需要时重新计算数据流信息？

8.11 给出对一个具有非正常区域的流图进行结构数据流分析的例子，并说明三种处理非正常区域方法中的每一种，这三种方法即(a) 结点分割，(b) 迭代和(c) 解底层格是单调函数格的数据流问题。

8.12 (a) 用结构方法公式化向后数据流分析。(b) 说明处理非正常区域的函数格上的迭代是向前分析。

8.13 (a) 构造一个其循环体是非正常区域且其中包含一个`if-then-else`结构的简单循环流图的例子。(b) 为你的这个例子写出结构数据流方程。(c) 写出关于执行这个循环和`if-then-else`的向前数据流分析的ICAN代码。(d) 如8.7.3节所讨论的那样，构造表示该非正常区域中数据流的ICAN数据结构。(e) 用ICAN写出一个使用(d) 中数据结构计算数据流信息的解释程序。(f)使用这个循环和`if-then-else`的代码以及这个非正常区域的解释程序计算此流图的到达-定值信息。

8.14 给出结构分析方程的另一种可选表示(即，与8.7.3节所给方程不同的表示)。你的选择的优缺点是什么？

8.15 用公式表示并求解$\mathbf{BV}^n$问题，例如可用表达式、位置式分析。

8.16 构造图16-7中过程的du链、ud链及网。

8.17 计算图16-7中过程的$DF()$和$DF^+()$

RSCH 8.18 研究用于数组数据流分析的最近写树[Feau91]。它们提供和允许什么信息？代价如何？ 265 ~ 266

ADV 8.19 Sharlit产生数据流问题的MOP解吗？

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第9章 依赖关系分析和依赖图

依赖关系分析是指令调度（见17.1节）和数据高速缓存优化（见20.4节）的重要工具。

作为指令调度的工具，它确定一个基本块（或一大段代码）中指令间的顺序关系，这种关系是代码正确执行所必须遵循的，这包括确定两个给定的寄存器或存储引用所访问的存储地址是否重叠，确定一对指令是否有资源冲突。类似地，依赖关系分析和在它支持下才能实现的循环变换是数据高速缓存优化的主要工具，也是自动向量化和并行化的基础。

本章还给出了一种称为程序依赖图（见9.5节）的新的中间形式，它是进行若干种类型优化的基础。

本章最后几节中有一节专门讨论了动态分配的对象之间的依赖关系。

9.1 依赖关系

本节介绍依赖关系分析的基本概念。后面各节将说明它们如何用于指令调度和数据高速缓存相关的分析。在每种情况下，我们构造一个依赖图，它表示一个代码段中的依赖关系——在作用于一个基本块的指令调度中，这种依赖图中没有环路，所以称为依赖DAG。我们将看到，依赖关系分析可以适用于任意层次的代码，包括源代码、中间代码或目标代码。

在给定的执行顺序中，如果语句S1在S2之前执行，我们记作S1 ◁ S2。程序中两个语句之间的*依赖关系*（dependence）是一种约束它们执行顺序的关系。*控制依赖*（control dependence）是程序控制流导致的一种约束，例如图9-1中S2与S3和S4的关系，S3和S4仅当S2中的条件不满足时才执行。如果在语句S1和S2之间存在控制依赖关系，我们记作S1 δ^c S2。所以在图9-1中有S2 δ^c S3和S2 δ^c S4。

```
S1          a ← b + c
S2          if a > 10 goto L1
S3             d ← b * e
S4             e ← d + 1
S5    L1:   d ← e / 2
```

图9-1 MIR代码中的控制和数据依赖关系的例子

数据依赖（data dependence）是语句间数据流造成的一种约束，例如在图9-1的语句S3和S4之间，S3给d赋值，S4使用d的值，另外，S3使用e的值而S4给e赋值。对于这两种情况，颠倒这两个语句的执行顺序就会导致程序产生错误的结果。有4种类型的数据依赖关系，如下所示:

1. 如果S1 ◁ S2，且S1 置一个值而S2使用该值，则称存在一个*流依赖*（flow dependence）或*真依赖*（true dependence），这是一个二元关系，记作S1 δ^f S2。例如，图9-1中S3和S4之间的流依赖关系记作S3 δ^f S4。

2. 如果S1 ◁ S2，且S1 使用某个变量的值而S2给该变量赋值，则称这两个语句间存在一个*反依赖*（antidependence），记作S1 δ^a S2。图9-1中语句S3和S4之间就有一个反依赖关系S3 δ^a S4，同时它们之间也有一个流依赖关系，因为S3使用e而S4给e赋值。

3. 如果S1 ◁ S2，且两个语句都给某个变量赋值，则称这两个语句间存在一个*输出依赖*（output dependence），记作S1 δ^o S2。在图9-1中，有S3 δ^o S5。

4. 最后，如果S1 ◁ S2，且两个语句都使用某个变量的值，则称这两个语句间存在一个*输入依赖*（input dependence），记作S1 δ^i S2。例如，在图9-1中，有S3 δ^i S5，因为这两个语句都使

bbs.theithome.com

用e的值。注意输入依赖关系不约束两个语句的执行顺序，但建立这个概念有利于我们在20.3节中讨论数组元素的标量置换。

一组依赖关系可以用一个称为依赖图（dependence graph）的有向图来表示。在这种图中，结点表示语句，边表示依赖关系。每条边上的标识指出了它代表的依赖关系，习惯上流依赖边也可以不标识。图9-2给出了图9-1中代码的依赖图。控制依赖通常不在依赖图中表示，除非这
268 种依赖关系是两个结点之间惟一的依赖关系（在我们的例子中，语句S2和S4有一个控制依赖，但图中省略了）。

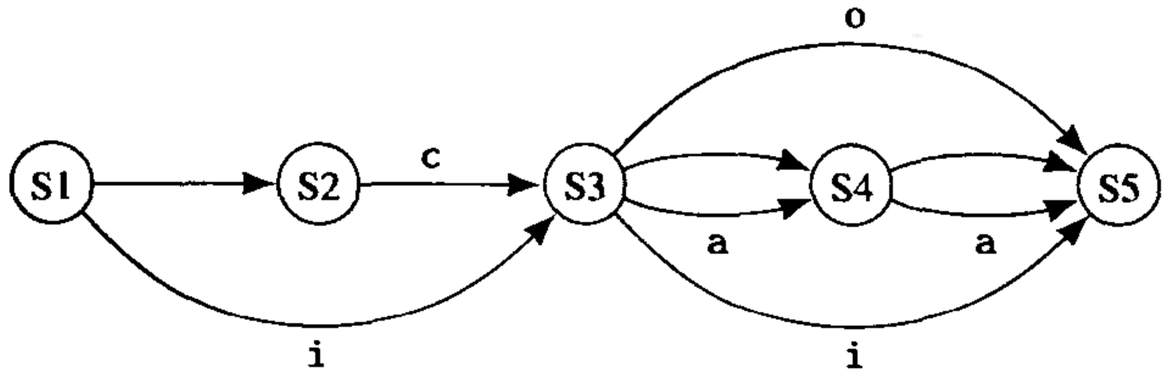


图9-2 图9-1中代码的依赖图

9.2 基本块依赖DAG

在第17章中考虑的基本块调度方法称为表调度，它要求我们首先要构造一个依赖图，这个依赖图反映了对基本块中的指令进行调度的有关约束，也即调度的自由度。因为一个基本块中没有循环，它的依赖图总是一个无环有向图（DAG），称为基本块的依赖DAG（dependence DAG）。

依赖DAG中的结点表示机器指令或低级中间代码指令，边表示指令之间的依赖关系。从I_1到I_2的一条边表示存在下述依赖关系之一：

1. I_1写一个寄存器或存储单元，而I_2使用它，即$I_1\ \delta^f\ I_2$；
2. I_1使用一个寄存器或存储单元，而I_2改变它，即$I_1\ \delta^a\ I_2$；
3. I_1和I_2写入同一个寄存器或存储单元，即$I_1\ \delta^o\ I_2$；
4. 不能确定I_1能否移到I_2之后执行；或
5. I_1和I_2有一个结构相关，下面将予以解释。

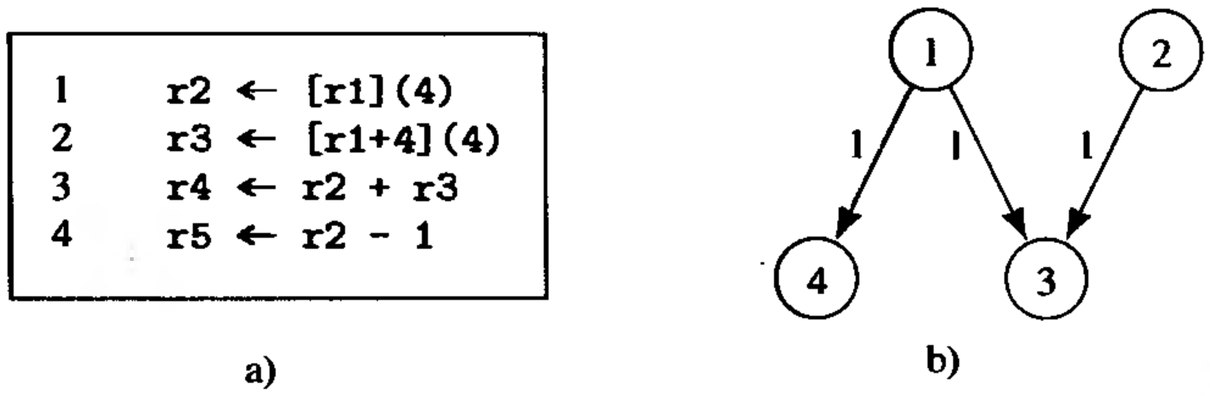
第4种依赖关系的一个例子是，当一条取指令之后是一条使用不同寄存器来访问存储器的存储指令时，我们就不能确定这两条指令的访存地址是否重叠。更具体地说，假定一条指令从[r11](4)读，而下一条指令写入[r2+12](4)。在不知道r2+12和r11指向不同位置的情况下，我们必须假定在这两条指令之间存在依赖关系。

9.4节将介绍的一些技术可用于消除循环中许多存储引用的歧义性，它形成的信息可作为代码的注解传递给指令调度器，以增加调度的自由度。

在依赖DAG中，如果I_2必须在I_1执行了若干拍之后才能执行，结点I_1是结点I_2的前驱。在DAG中一条边表示的依赖关系的类型并不重要，所以我们省略了类型标记。但是，我们用一个整数来标记I_1到I_2的边，这个整数是I_1与I_2之间要求的等待时间（latency），省略它时表示等待时间为0。等待时间是从I_1开始执行到I_2开始执行之间要求的延迟（delay）减去在其他任何指令可以开始执行之前I_1所需的执行时间（通常是一个时钟周期，即一拍，但在超标量系统中常常为0）。这样，如果I_2可以在紧随I_1开始之后的时钟周期内开始执行，则等待时间是0。而如果I_1的开始到I_2的开始之间必须间隔2拍，则等待时间是1。例如，对于图9-3a中的LIR代码，假定取指令的等待时间是1拍，并需要2拍来完成执行，则依赖DAG如图9-3b所示。

bbs.theithome.com

越是公司制作 请尊重作者版权

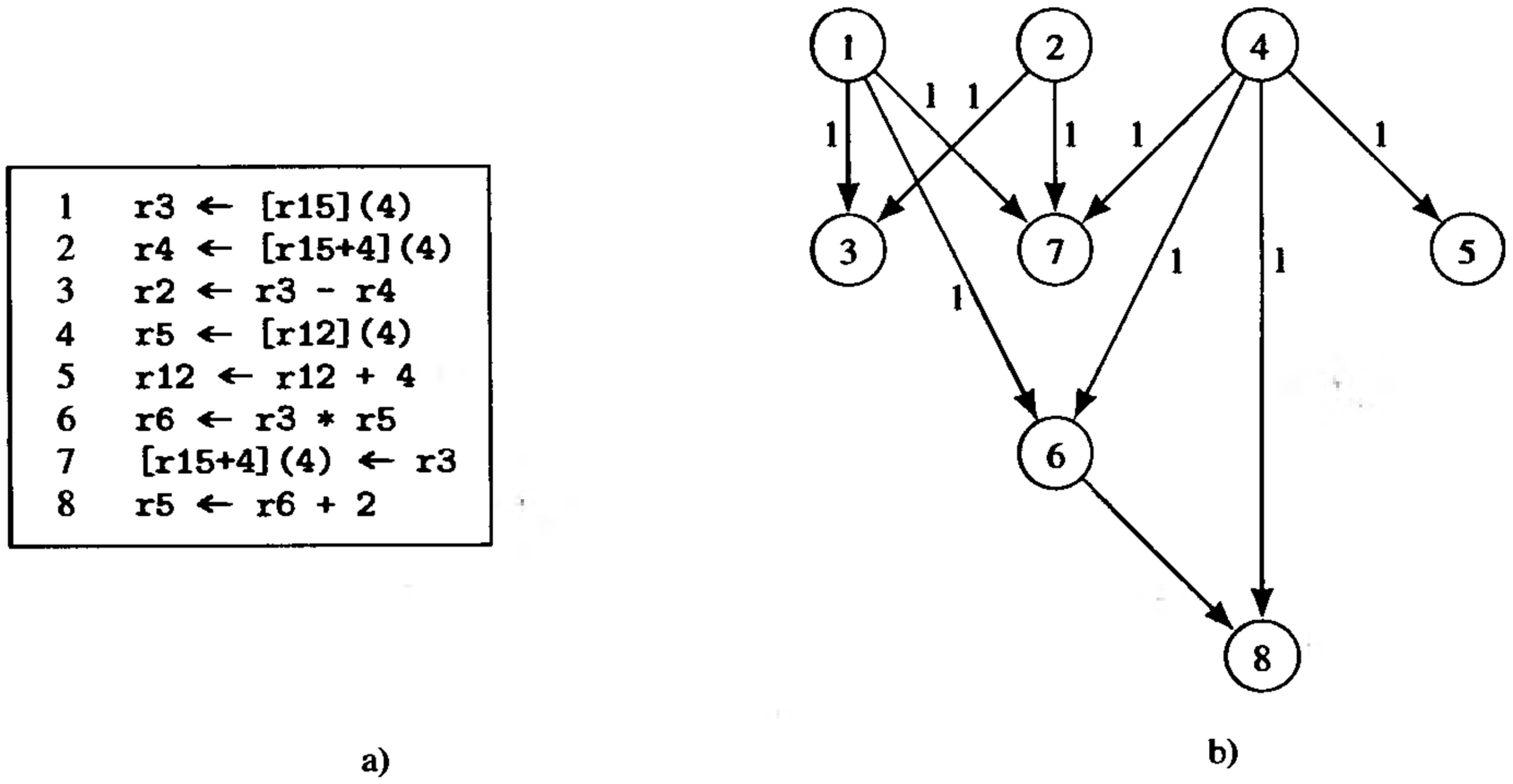


a) b)

图9-3 a) 一个LIR代码的基本块，b) 该基本块的依赖DAG

在计算依赖关系时，将条件代码和其他隐含资源当成寄存器同样看待。

更复杂一点的第二个例子是图9-4a中的LIR代码和图b)给出的它的依赖DAG，同样假定取数指令的等待时间为1拍。指令1和指令2是相互无关的，因为它们引用不同的存储器地址和不同的目的寄存器。指令3必须在指令1和指令2之后执行，因为它使用这两条指令取的值。指令4与指令1和指令2无关，因为它们都取值到不同的寄存器。指令7必须在指令1、指令2和指令4之后执行，因为它将指令1的结果存入到由指令2读取的地址中，而且当r12中的值是r15＋4时，它就与指令4冲突。 269



a) b)

图9-4 a) 一个更复杂的LIR例子，b) 该例子的依赖DAG

注意，图9-4b中从4到8的边是多余的，因为图中有一条从4到6的边，还有一条从6到8的边。另一方面，如果从4到8的边标记的等待时间为4，则它就不是多余的了，因为这样

1, 2, 4, 5, 6, 7, 8, 3

和

1, 2, 4, 5, 6, 8, 7, 3

两种情况都是可行的调度，前一个要求8拍，但后一个要求9拍。 270

为了构造一个基本块的DAG，我们需要两个函数:

```
Latency: LIRInst × integer × LIRInst × integer → integer
```

和

```
Conflict: LIRInst × LIRInst → boolean
```

这两个函数的定义是:

bbs.theithome.com

Latency(I_1, n_1, I_2, n_2)＝在I_1执行第n_1个时钟周期时开始执行I_2的第n_2个时钟周期需要等待的时钟周期数

和

$$\text{Conflict}(I_1, I_2) = \begin{cases} \text{true} & \text{如果}I_1\text{必须在}I_2\text{之前执行才能保证程序的正确性} \\ \text{false} & \text{其他情况} \end{cases}$$

注意，对于任意两条由一个“.sequence”伪操作分隔开的LIR指令I_1和I_2，Conflict(I_1, I_2)为true。

为了计算Latency()，我们需要用到资源向量。一条指令的资源向量（resource vector）是该指令在其执行时的连续时钟周期内所需的各种计算资源集合组成的数组。例如，MIPS R4000浮点部件有7个资源，分别称为A（尾数加）、E（异常检测）、M（乘法器第一级）、N（乘法器第二级）、R（舍入加）、S（操作数移位）和U（操作数准备）。单精度浮点加（add.s）和乘（mul.s）指令的资源向量如下:

	1	2	3	4	5	6	7
add.s	U	S,A	A,R	R,S			
mul.s	U	M	M	M	N	N,A	R

所以在一条mul.s指令的第4拍开始执行一条add.s指令将导致在mul.s的第6和第7拍出现冲突——在第6拍，两条指令都需要资源A，在第7拍，两条指令都需要资源R。在同一时间内两条或多条指令因同一个资源引起的竞争称为结构相关（structural hazard）。

为计算Latency(I_1, I_2)，我们要匹配指令I_1和I_2的资源向量（见图9-5）。具体地，我们重复地检查两个资源向量的元素是否有非空的交集，每当发现一个资源冲突时，就沿着I_1的资源向量逐个元素地步进。过程Inst_RV()的第一个参数是一条指令，第二个参数是资源向量的长度，它返回这条指令对应指令类型的资源向量。函数ResSet(*inst*, *i*)返回指令*inst*在时钟周期*i*
271 使用的资源集合。常数MaxCycles是任意指令所需的最大时钟周期数。

```
ResVec = array [1··MaxCycles] of set of Resource
MaxCycles, IssueLatency: integer

procedure Latency(inst1,cyc1,inst2,cyc2) returns integer
   inst1, inst2: in LIRInst
   cyc1, cyc2: in integer
begin
   I1RV, I2RV: ResVec
   n := MaxCycles, i := 0, j, k: integer
   cycle: boolean
   I1RV := Inst_RV(inst1,cyc1)
   I2RV := Inst_RV(inst2,cyc2)
   || determine cycle of inst1 at which inst2 can begin
   || executing without encountering stalls
   repeat
      cycle := false
      j := 1
      while j ≤ n do
         if I1RV[j] ∩ I2RV[j] ≠ ∅ then
            for k := 1 to n - 1 do
               I1RV[k] := I1RV[k+1]
            od
```

图9-5 计算Latency()的函数

bbs.theithome.com

```
                n -= 1
                i += 1
                cycle := true
                goto L1
            fi
            j += 1
        od
L1:     until !cycle
        return i
    end     || Latency

procedure Inst_RV(inst,cyc) returns ResVec
    inst: in LIRInst
    cyc: in integer
begin
    IRV: ResVec
    i: integer
    || construct resource vector for latency computation
    || from resource set
    for i := 1 to MaxCycles do
        if cyc+i-1 < MaxCycles then
            IRV[i] := ResSet(inst,cyc+i-1)
        else
            IRV[i] := ∅
        fi
    od
    return IRV
end     || Inst_RV
```

图9-5 （续）

对于例子Latency(mul.s,4,add.s,1)，我们有MaxCycles=7，以及如下的资源向量: 272

```
I1RV[1] = {M}       I2RV[1] = {U}
I1RV[2] = {N}       I2RV[2] = {S,A}
I1RV[3] = {N,A}     I2RV[3] = {A,R}
I1RV[4] = {R}       I2RV[4] = {R,S}
I1RV[5] = ∅         I2RV[5] = ∅
I1RV[6] = ∅         I2RV[6] = ∅
I1RV[7] = ∅         I2RV[7] = ∅
```

读者不难算出上面调用Latency()的返回值是2，所以在mul.s的第4拍开始执行add.s指令将导致2拍的停顿[⊖]，但在mul.s的第6拍开始执行add.s指令就没有迟延。

虽然这个计算等待时间的算法是一目了然的，但还有更快的算法。Proebsting和Fraser在[ProF94]中介绍了一种方法，它使用一个确定的有限自动机，该自动机的状态类似于资源向量集合，自动机的状态转换是指令，它通过简单的查表来完成类似于我们的算法中j循环的计算。

令Inst[1..*m*]是构成一个基本块的指令序列（包括“.sequence”伪指令）。如果这个基本块以一条分支指令结束，且在体系结构中分支指令具有延迟槽，我们则不将该分支指令包含在指令序列内。数据结构DAG的形式如图9-6所示，其中Nodes={1,…,*n*}，Roots⊆Nodes。

⊖ 停顿（stall）指的是一条流水线不活动（或停转），此时因为需要的硬件资源正在使用，或某种条件没有满足，它不能继续处理提交给它的下一条指令。例如，在某些体系结构的实现中，如果一条取指令所取的值被下一条指令使用，则流水线将停顿一拍时钟周期。

bbs.theithome.com

```
DAG = record {
        Nodes, Roots: set of integer,
        Edges: set of (integer × integer),
        Label: (integer × integer) ⟶ integer}

procedure Build_DAG(m,Inst) returns DAG
   m: in integer
   Inst: in array [1··m] of LIRInst
begin
   D := ⟨Nodes:∅,Edges:∅,Label:∅,Roots:∅⟩: DAG
   Conf: set of integer
   j, k: integer
   || determine nodes, edges, labels, and
   || roots of a basic-block scheduling DAG
   for j := 1 to m do
      D.Nodes ∪= {j}
      Conf := ∅
      for k := 1 to j - 1 do
         if Conflict(Inst[k],Inst[j]) then
            Conf ∪= {k}
         fi
      od
      if Conf = ∅ then
         D.Roots ∪= {j}
      else
         for each k ∈ Conf do
            D.Edges ∪= {k→j}
            D.Label(k,j) := Latency(Inst[k],1,Inst[j],IssueLatency+1)
         od
      fi
   od
   return D
end    || Build_DAG
```

图9-6 构造用于基本块调度的依赖DAG的算法

构造调度DAG的算法称为Build_DAG()，在图9-6中给出。该算法按顺序依次处理所有指令。对于每条指令，算法首先确定它是否与已经在DAG中的指令发生冲突。如果是，将与它相冲突的那些指令放置在Conf集合中。然后，如果Conf为空，将当前指令加入Roots，即DAG的根集合。否则，对于Conf中的每一条指令，在DAG中增加一条从该指令到当前处理的指令的边，并以相应的等待时间标记这条边。构造DAG需要$O(n^2)$个操作，因为为了查找依赖关系，要比较基本块中的每一对指令。

作为构造DAG的例子，考虑图9-4a给出的8条LIR指令序列。以n＝8并且用序列中的8条指令设置Inst[1]到Inst[8]来调用Build_DAG()。对于第一条指令，在它前面不存在与它发生冲突的指令，所以它是一个根结点。第2条指令与第一条没有冲突，所以也是一个根结点。第3条指令与前两条指令都有冲突，因为它使用前两条指令取的值，所以在DAG中加入分别从前两条指令到第3条指令的边，每条边的等待时间为1。第4条指令与它前面任意一条指令都不冲突——虽然[r12](4)与[r15](4)或[r15＋4](4)表示的可能是同一地址，但因为它们都是取指令，并且在它们之间没有插入对这些地址的存指令，所以它们是不冲突的。调用此过程产生的DAG如图9-4b所示。

9.3 循环中的依赖关系

在研究数据高速缓存优化时，我们的注意力几乎完全集中于数据依赖关系，而不关心控制依赖关系。

bbs.theithome.com

超星公司制作 请尊重作者版权

273~274

前面我们以一个基本块中标量之间的数据依赖关系为例，介绍了数据依赖关系的概念，分析这种依赖关系有利于进行指令调度。下面我们要考虑嵌套循环中涉及带下标变量的语句之间的依赖关系。特别地，我们考虑以HIR代码表示的*规范形式*（canonical form）的紧嵌套循环中的带下标变量。这种循环的索引变量从1变化到n，步长为1，且只有最内层循环体中包含除for语句以外的语句。

```
for i1 ← 1 to n1 do
   for i2 ← 1 to n2 do
      . . .
         for ik ← 1 to nk do
            statements
         endfor
      . . .
   endfor
endfor
```

图9-7 一个规范的循环嵌套

图9-7中循环嵌套的*迭代空间*（iteration space）是由各层循环索引值组成的所有k-元组（称为*索引向量*，index vector）构成的k维多面体。显然它是各层循环索引值域的乘积，即

[1..n1] × [1..n2] × ... × [1..nk]

其中[a..b]表示从a到b的整数序列，n1，...，nk是迭代变量的最大值。

我们用 $\prec$ 表示索引向量的词典顺序。这样，

$<i1_1, \cdots, ik_1> \prec <i1_2, \cdots, ik_2>$

当且仅当

$\exists j, 1 \leqslant j \leqslant k$，使得$i1_1 = i1_2, \cdots, i(j-1)_1 = i(j-1)_2$且$ij_1 < ij_2$

并且，特别地，$0 \prec <i1, \cdots, ik>$，即索引向量 $\vec{i}$ 是*词典序为正的*（lexicographically positive），如果

$\exists j, 1 \leqslant j \leqslant k$，使得$i1 = 0, \cdots, i(j-1) = 0$且$ij > 0$

注意，循环嵌套中的迭代 $<i1_1, \cdots, ik_1>$ 先于迭代 $<i1_2, \cdots, ik_2>$，当且仅当

$<i1_1, \cdots, ik_1> \prec <i1_2, \cdots, ik_2>$

一个循环嵌套的*迭代空间遍历*（iteration-space traversal）是在执行该嵌套时遇到的索引向量值的序列，即索引向量的词典枚举。我们用对应于索引向量的结点图来表示遍历，结点之间的虚线箭头指出了遍历的顺序。例如，对于图9-8中的循环，其迭代空间是[1..3] × [1..4]，图9-9描绘了该迭代空间的遍历。

```
     for i1 ← 1 to 3 do
        for i2 ← 1 to 4 do
S1          t ← x + y
S2          a[i1,i2] ← b[i1,i2] + c[i1,i2]
S3          b[i1,i2] ← a[i1,i2-1] * d[i1+1,i2] + t
        endfor
     endfor
```

图9-8 一个双重嵌套循环的例子

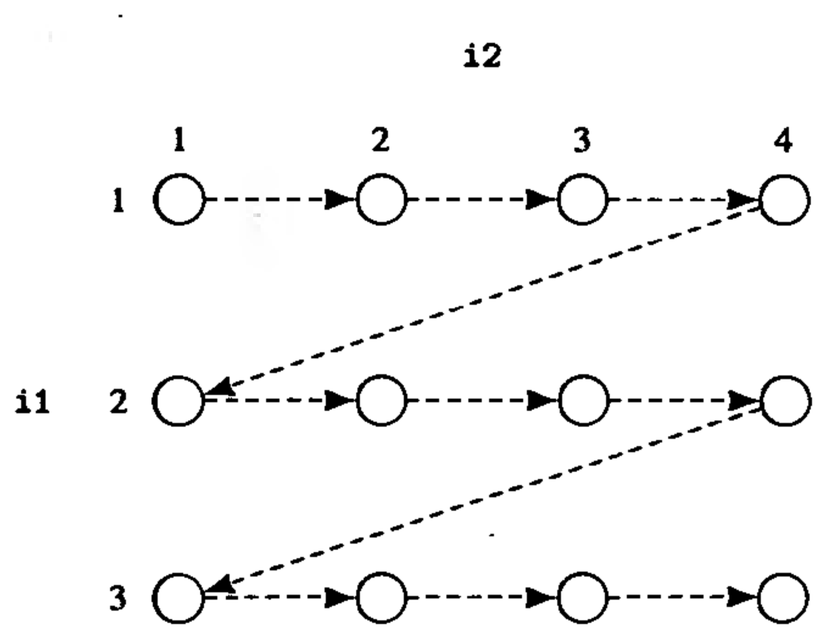


图9-9 图9-8中代码的迭代空间遍历

bbs.theithome.com

注意，一个循环嵌套的迭代空间不必是矩形。具体地说，图9-10中的循环嵌套具有梯形迭代空间，它的迭代空间遍历如图9-11所示。

```
      for i1 ← 1 to 3 do
         for i2 ← 1 to i1+1 do
S1           a[i1,i2] ← b[i1,i2] + c[i1,i2]
S2           b[i1,i2] ← a[i1,i2-1]
         endfor
      endfor
```

图9-10 一个具有非矩形迭代空间遍历的双重嵌套循环

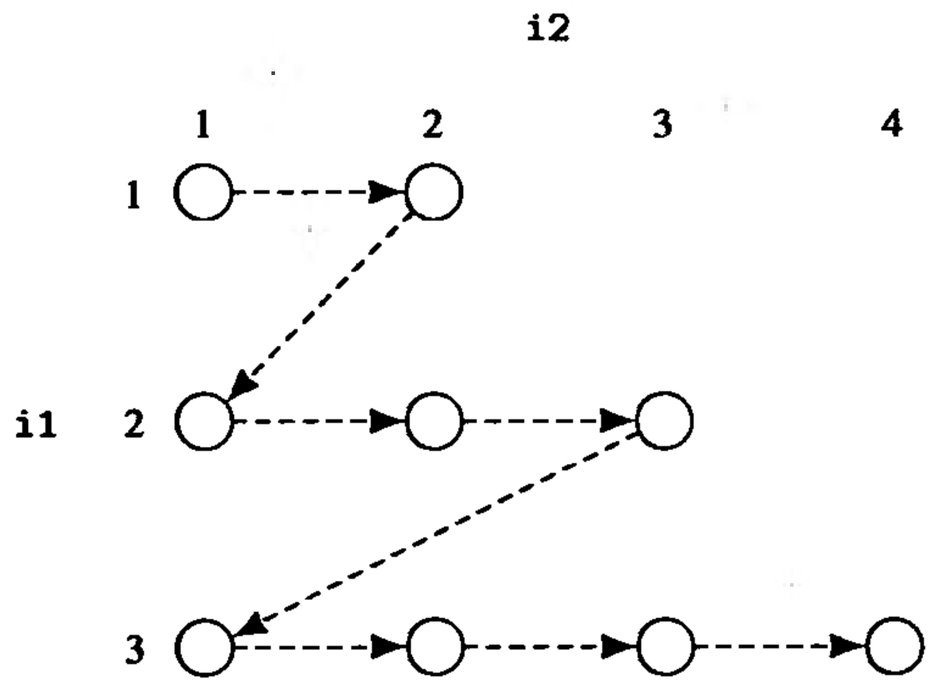


图9-11 图9-10中循环嵌套的梯形迭代空间遍历

嵌套在循环中的带下标变量和语句之间的依赖关系比标量情形的更为复杂，因为这种依赖关系是索引变量的函数。我们用语句号后带中括号的下标来指明索引变量的值，并扩展符号$\lhd$的含义，使得$S_1[i1_1, \cdots, ik_1] \lhd S_2[i1_2, \cdots, ik_2]$表示$S_1[i1_1, \cdots, ik_1]$在$S_2[i1_2, \cdots, ik_2]$之前执行，其中$i1$到$ik$为包含语句$S_1$和$S_2$的那些循环的循环索引，最外层循环的在前。注意，$S_1[i1_1, \cdots, ik_1] \lhd S_2[i1_2, \cdots, ik_2]$当且仅当在程序中$S_1$先于$S_2$且$<i1_1, \cdots, ik_1> \preceq <i1_2, \cdots, ik_2>$，或者$S_1$与$S_2$是同一语句或$S_1$在$S_2$之后且均有$<i1_1, \cdots, ik_1> \prec <i1_2, \cdots, ik_2>$。

对于图9-8给出的例子，有下列执行顺序关系：

S2[i1, i2-1] ◁ S3[i1, i2]

S2[i1, i2] ◁ S3[i1 ◁ i2]

以及对应的依赖关系：

S2[i1, i2-1] δ^f S3[i1, i2]

S2[i1, i2] δ^a S3 [i1, i2]

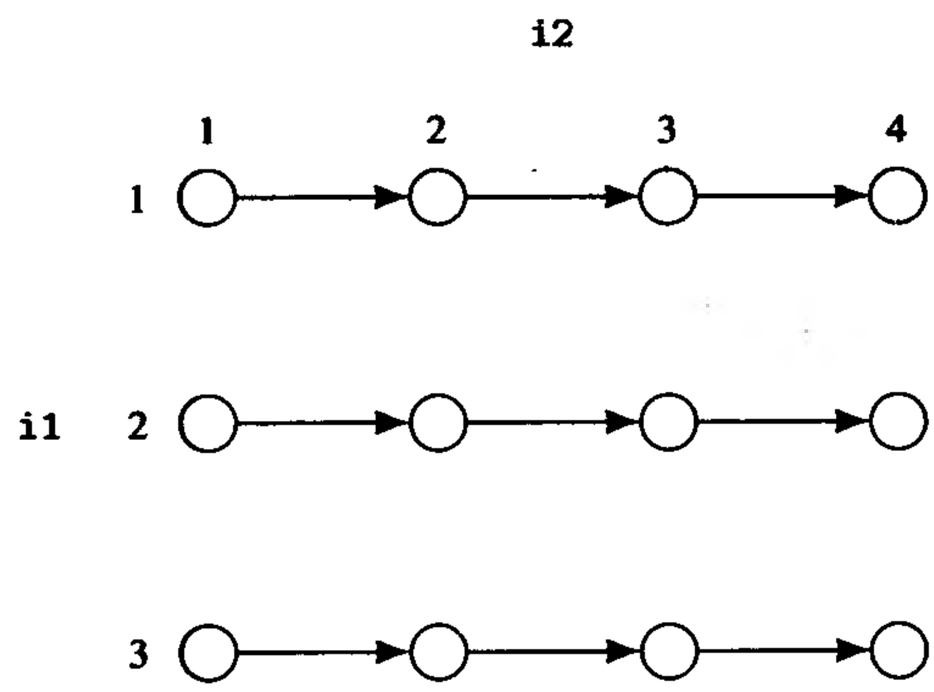


图9-12 图9-8中代码的依赖关系

类似于迭代空间遍历，循环体中的依赖关系也可以整体地用图来表示，如图9-12所示。在图9-12中，我们省略了表示自依赖的箭头。注意，每个 <i1, 1> 迭代使用a[i1,0]的值，但在图中没有表示，因为0不是循环索引i2的合法值。

下面我们介绍用于表示基于循环的依赖关系的距离向量、方向向量和依赖向量。k层嵌套循环的距离向量（distance vector）是一个k维向量$\vec{d} = <d_1, \cdots, d_k>$，其中，每一个$d_i$是一个整数；它意味着对于每一个索引向量$\vec{i}$，具有索引向量$\vec{i}+\vec{d} = <i_1+d_1, \cdots, i_k+d_k>$的迭代依赖于具有索引向量$\vec{i}$的迭代。一个方向向量（direction vector）近似于一个或多个距离向量，它的元素是整数值域，或是两个以上值域的联合，其中每一个值域可以是[0，0]、[1，∞]、[－∞，－1]或[－∞，∞]。方向向量有两组经常使用的记法，如下所示：

275 ~ 277

[0,0]	[1,∞]	[−∞,−1]	[−∞,∞]
=	+	−	±
=	<	>	*

在第二组记法中，符号“≠”，“<”和“>”用于表示值域的联合，其含义是显然的。这样，

bbs.theithome.com

前面给出的图9-8中的循环嵌套中的依赖关系可以用下面的距离向量来表示:

S2[i1, i2 − 1] <0, 1> S3[i1, i2]

S2[i1, i2] <0, 0> S3[i1, i2]

而这两个距离向量可以归纳（显然要损失一些信息）为一个方向向量<=, <>。

依赖向量归纳了前两种向量。k层嵌套循环的依赖向量（dependence vector）是一个k维向量 $\vec{d}=[d_1^-,d_1^+],\cdots,[d_k^-,d_k^+]$，其中，每个 $[d_i^-,d_i^+]$是一个整数值域（可能是无穷值域），且无穷值域满足 $d_i^- \in Z\cup\{-\infty\}$, $d_i^+ \in Z\cup\{-\infty\}$，和 $d_i^- < d_i^+$。注意，一个依赖向量对应于一个（可能无穷个）距离向量集合，称这个距离向量集合为它的距离向量集（distance vector set）$DV(\vec{d})$，如下所示:

$$DV(\vec{d})=\{<a_1,\cdots,a_k>|a_i\in Z\}\text{且}\ d_i^- < a_i < d_i^+\}$$

如果对于$1 \leqslant i \leqslant k$, $d_i^- = d_i^+$，则依赖向量就是距离向量。为方便起见，我们只写一个值而不写它们的值域。如上所示，值域[1，∞]、[－∞，－1]和[－∞，∞]对应于方向“+”或“<”、“－”或“>”和“±”或“*”。

这样，图9-8中的循环嵌套的依赖关系就可以用依赖向量<0，[0，1]>来表示。

注意，距离向量为<0，0，…，0>的依赖关系对那种不重排循环体中语句顺序的循环变换没有影响。因此，我们下面不提及这类依赖关系。

进一步，一个依赖关系可能是循环无关的（loop-independent），即独立于环绕它的循环；也可能是循环携带的（loop-carried），即由于有循环才出现的依赖关系。在图9-8中，因为S2使用b[i1,i2]，而S3定义b[i1,i2]，由此产生了S3对S2的反依赖关系，这个反依赖关系是循环无关的，因为即使这两个语句不在循环中，反依赖关系仍然存在。相反，由于S2置a[]的一个元素，S3在i2循环的一个迭代之后使用该元素，由此产生了一个S3对S2的流依赖关系，这个流依赖关系是循环携带的，具体地说，是由内层循环携带的，去掉外层循环不会改变这个依赖关系。为表示循环无关的依赖关系，在它的依赖关系记号上加一个下标0，而循环i（从外往里数）携带的依赖关系则加一个下标$i>1$。这样，对于图9-8的依赖关系我们有: 278

S2[i1, i2 − 1] δ_1^f S3[i1, i2]

S2[i1, i2] δ_0^a S3[i1, i2]

或用距离向量记法:

S2[i1, i2 − 1] <0, 1>$_1$ S3[i1, i2]

S2[i1, i2] <0, 0>$_0$ S3[i1, i2]

这些概念在做数组元素的标量置换时是有用的（见20.3节）。

再看另一个例子，图9-13中的赋值语句中的依赖关系有距离向量<1，0>，该依赖关系是由外层循环携带的，即

S1[i1-1, j] <1, 0>$_1$ S1[i1, j]

```
    for i ← 1 to n do
       for j ← 1 to n do
S1           a[i,j] ← (a[i-1,j] + a[i+1,j])/2.0
       endfor
    endfor
```

图9-13 一个具有距离向量为<1，0>的依赖关系的赋值语句S1

bbs.theithome.com

9.4 依赖关系测试

在20.4节中，我们关注为更好地发挥存储器层次的性能而进行的循环嵌套转换。大多数这类变换仅当循环嵌套的迭代之间不存在依赖关系（其种类与具体的转换相关）时才能实行。因此，首先必须能够确定循环嵌套中是否存在依赖关系，如果存在，是什么依赖关系。

考虑图9-14a中的循环例子。为确定在同一迭代中对a[]的那些引用之间是否存在依赖关系，我们必须确定是否存在一个整数i，满足方程

$$2*i+1=3*i-5$$

和不等式$1<i<4$。显然当且仅当$i=6$等式才成立，而i的这个值不满足不等式，所以循环中不存在同一迭代中的依赖关系（即距离为0的依赖关系）。

```
for i ← 1 to 4 do
   b[i] ← a[3*i-5] + 2.0
   a[2*i+1] ← 1.0/i
endfor
```

a)

```
for i ← 1 to 4 do
   b[i] ← a[4*i] + 2.0
   a[2*i+1] ← 1.0/i
endfor
```

b)

图9-14 用于依赖关系测试的两个HIR循环代码例子

279 为确定在不同迭代中，对于a[]的引用之间是否存在依赖关系，我们寻找满足方程

$$2*i_1+1=3*i_2-5$$

和上面给定的不等式的整数i_1和i_2。同样，我们能容易地确定，对于任意的i，$i_1=3*i$，$i_2=2*i+2$都满足等式，并且对于$i=1$，我们有$i_1=3$和$i_2=4$，两者都满足不等式。因此，在循环中存在一个距离为正数的依赖关系：a[7]在迭代3中使用，在迭代4中赋值。

注意，如果这个循环上、下界不是常数表达式，我们就不能断定不存在距离为0的依赖关系。我们只能说，可能存在一个距离为0或距离为正数的依赖关系，因为不等式不再适用。

下面我们将循环中的第一个语句改为取a[4*i]，如图9-14b所示。于是，对于i_1和i_2的相同或不同的整数值，我们必须满足方程

$$2*i_1+1=4*i_2$$

也必须满足与上面相同的不等式，显然这是不可能的。先不考虑不等式，对于i_1的任意值，等式左边的值总是奇数；而对于i_2的任意值，等式右边总是偶数。因此不论这个循环的上下界取什么值，在循环中均不存在依赖关系。

一般地说，测试循环嵌套中是否存在依赖关系，如果存在，又是什么类型的依赖关系，是一个*约束的丢番图方程*（constrained Diophantine equation）问题，即求解一个或多个整系数方程的整数解，这些解同时满足一组给定的不等式。这等价于整数规划（integer programming），而整数规划是一个NP完全问题。然而，在实际程序中几乎所有的下标表达式都是非常简单的。

特别地，下标几乎总是循环索引变量的线性表达式，并且从现在开始我们就假定是这样。下标常常是一个索引变量的线性表达式，因此，我们假定给定一个n层循环嵌套如图9-15所示，其索引变量为i_1到i_n，循环体中有两个对数组元素x[]的引用，其下标都是循环索引变量的线性表达式。这两个数组元素引用之间存在一个依赖关系当且仅当每一个下标值同时满足方程

$$a_0+\sum_{j=1}^{n}a_j*i_{j,1}=b_0+\sum_{j=1}^{n}b_j*i_{j,2}$$

和不等式

bbs.theithome.com

$1 < i_{j,1} < hi_j$ 和 $1 < i_{j,2} < hi_j$，其中 $j = 1, \cdots, n$

当然，依赖关系的类型由x[...]的每一个实例是使用还是定值来确定。有许多依赖关系测试方法，我们详细地介绍其中几种，并列出其他方法。 280

```
for i1 ← 1 to hi1 do
   for i2 ← 1 to hi2 do
      ...
         for in ← 1 to hin do
            ...
            ··· x[..., a0 + a1 * i1 + ··· + an * in, ...] ···
            ...
            ··· x[..., b0 + b1 * i1 + ··· + bn * in, ...] ···
            ...
         endfor
      ...
   endfor
endfor
```

图9-15 用于依赖关系测试的HIR形式的循环嵌套

现在仍在使用的最早的测试方法是由Banerjee[Bane76]和Towle[Towl76]提出的GCD（greatest common divisor）测试。从它证明不存在依赖关系的能力来说，它是一个比较弱的测试。GCD测试声称，如果至少存在某一维下标，

$$\gcd\left(\bigcup_{j=1}^{n} \text{sep}(a_j, b_j, j)\right) \nmid \sum_{j=0}^{n} (a_j - b_j)$$

其中，gcd（）是最大公约数函数，“$a \nmid b$”表示a不能整除b，而sep(a, b, j)定义为[⊖]

$$\text{sep}(a, b, j) = \begin{cases} \{a - b\} & \text{若方向}j\text{是=} \\ \{a, b\} & \text{否则} \end{cases}$$

则对x[…]的这两个引用是不相关的；或等价地说，如果存在一个依赖关系，则GCD整除这个和。例如，对于图9-14a中的循环，对同一个迭代中的依赖关系测试归结为

$$\gcd(3 - 2) \nmid (-5 - 1 + 3 - 2)$$

或$1 \nmid -5$，它不为真，这只能告诉我们可能存在依赖关系。类似地，对于迭代之间的依赖关系，我们有

$$\gcd(3, 2) \nmid (-5 - 1 + 3 - 2)$$

或同样有$1 \nmid -5$，它也不为真，这只能说明可能存在依赖关系。另一方面，对于图9-14b中的例子，我们有

$$\gcd(4, 2) \nmid (-1 + 4 - 2)$$ 281

它可简化为$2 \nmid 1$。因为这为真，故这两个数组引用是不相关的。

GCD测试可以推广到任意循环下界和循环增量。为此，假定第j层循环控制语句为

for $i_j \leftarrow lo_j$ by inc_j to hi_j

则GCD测试声称，如果至少存在某一维下标，

⊖ 注意，对于所有a和b，因为$\gcd(a, b) = \gcd(a, a - b) = \gcd(b, a - b)$，方向为不等的情况包含了方向为相等的情况。

bbs.theithome.com

$$\gcd\left(\bigcup_{j=1}^{n} \mathrm{sep}(a_j * inc_j, b_j * inc_j, j)\right) \not\mid a_0 - b_0 + \sum_{j=1}^{n} (a_j - b_j) * lo_j$$

则x[...]的这两个实例是不相关的。

数组引用的两种重要类型是可分的数组引用和弱可分的数组引用。可以发现，在大量的实际程序中都是这两类引用，而几乎没有其他形式的引用。一对数组引用是可分的（separable），如果每一对下标表达式的形式都是$a*i_j+b_1$和$a*i_j+b_2$，其中i_j是循环控制变量，a、b_1、b_2是常数。一对数组引用是弱可分的（weakly separable），如果每一对下标表达式的形式都是$a_1*i_j + b_1$和$a_2*i_j+b_2$，其中i_j是循环控制变量，a_1、a_2、b_1、b_2是常数。图9-14中的两个例子都是弱可分的，但不是可分的。

对于一对可分的数组引用，依赖关系测试是容易的：它们之间存在依赖关系，如果对于每个下标，下面的任意一个条件成立：

1. $a = 0$并且$b_1=b_2$，或者

2. $(b_1 - b_2) \mid a < hi_j$。

对于一对弱可分的数组引用，对每维下标位置j有一个线性方程

$$a_1*y+b_1=a_2*x+b_2$$

或

$$a_1*y=a_2*x+ (b_2 - b_1)$$

如果对于j的一个特定值，每组方程有一个满足由循环j的上下界给定的不等式的解，则这两个引用之间存在依赖关系。下面我们应用线性方程组的理论，将问题分为以下几种情况（在每种情况下，当且仅当方程组的解满足上下界不等式时，其解才表示依赖关系）：

(a) 如果方程组中只有一个方程，其形式如上所示，则我们有一个含两个未知量的线性方程，它有整数解当且仅当$\gcd(a_1, a_2) \mid (b_2 - b_1)$。

(b) 如果方程组中有两个方程，即

$$a_{1,1} * y = a_{2,1} * x + (b_{2,1} - b_{1,1})$$

和

$$a_{1,2} * y = a_{2,2} * x + (b_{2,2} - b_{1,2})$$

282 则这是一个含两个未知量的方程组。如果$a_{2,1}/a_{1,1} = a_{2,2}/a_{1,2}$，则存在有理解当且仅当

$$(b_{2,1} - b_{1,1})/ a_{1,1} = (b_{2,2} - b_{1,2})/ a_{1,2}$$

且这些解很容易枚举。如果$a_{2,1}/a_{1,1} \neq a_{2,2}/a_{1,2}$，则存在一个有理解，且不难确定这个解。在这两种情况下，我们都要检查解是否为整数且满足不等式的要求。

(c) 如果方程组中的方程数$n > 2$，则或者$n-2$个方程是冗余的，可以按（b）的方法处理；或者这是一个两个未知数的、至少由3个以上方程组成的超定方程组。

作为一个弱可分的例子，考虑图9-16中的循环嵌套。首先考虑对g[ ]的两个引用。如果它们之间存在依赖关系，则对于第一个下标有

$$2 * x = y + 1$$

对于第2个下标有

$$z = 3 * w$$

bbs.theithome.com

这两个方程是互不相关的且每个方程都有无穷个整数解。具体地说，只要$n \geq 3$，这两个数组引用之间就存在依赖关系。对于对h[]的两个引用，它们必须同时满足

$$x = y + 2$$

和

$$x = 2 * y - 2$$

显然，两个方程同时成立当且仅当$x=6$和$y=4$，所以存在依赖关系当且仅当$n \geq 6$。

```
for i ← 1 to n do
   for j ← 1 to n do
      f[i] ← g[2*i,j] + 1.0
      g[i+1,3*j] ← h[i,i] - 1.5
      h[i+2,2*i-2] ← 1.0/i
   endfor
endfor
```

图9-16 弱可分的一个例子

前面已经指出，还有许多测试能力和计算开销各不相同的其他的依赖关系测试方法（参见9.8节的进一步阅读）。它们包括:

1. 扩展GCD测试;
2. 强和弱单索引变量（single index variable，SIV）测试; 283
3. Delta测试;
4. Acyclic测试;
5. Power测试;
6. 简单循环余数测试;
7. Fourier-Motzkin测试;
8. 约束矩阵测试;
9. Omega测试。

9.5 程序依赖图

程序依赖图（program-dependence graph，PDG）是用于优化的一种中间代码形式。一个程序的PDG由一个控制依赖图（CDG）[⊖]和一个数据依赖图组成。PDG中的结点可以是基本块、语句、单个操作，或某种中间层次上的结构。数据依赖图已经在9.1节和9.3节中作了介绍。

CDG最基本的形式是一种以程序谓词（当结点为基本块时，则使用以谓词结束的基本块）作为根结点和中间结点，以程序的非谓词结点作为叶结点的DAG。如果在程序执行过程中，在控制依赖图中通向一个叶结点的路径上的谓词被满足，则该叶结点将被执行。

更具体地说，令$G=<N, E>$是一个过程的流图。结点m是结点n的后必经结点，记作m *pdom* n，当且仅当从n到过程出口的每一条路径都通过m（见7.3节）。所以，结点n*控制依赖*（control dependent）于结点m，当且仅当

1. 存在一条从m到n的控制路径，n是该路径上除m之外的每个结点的后必经结点，并且
2. n不是m的后必经结点[⊜]。

为构造CDG，我们先构造基本CDG，再在其上加入区域结点（region node）。构造基本CDG时，先在流图上增加一个称为start的虚构谓词结点，它的“Y”边指向入口结点，“N”边指向出口结点。由此得到的图称为G^+。下一步，在G^+上构造后必经关系[⊝]，这个关系可以用一棵树来表示，并且定义S为G^+中所有符合n不是m的后必经结点条件的边$m \rightarrow n$组成的集合。

⊖ [FerO87]定义了两种控制依赖关系图，我们这里讨论的一种（作者称为*精确*CDG）和另一种近似CDG。对于结构良好的程序，近似CDG能表示与精确CDG同样的依赖关系，用它能比较容易地生成串行代码。

⊜ 注意，这种控制依赖关系是9.1节中讨论的控制依赖关系的子关系。

⊝ 这可以通过逆转流图中的边，并使用7.3节介绍的两种必经结点算法之一来实现。

bbs.theithome.com

超星公司制作 请尊重作者版权

284 对于每一条边$m \to n \in S$，我们在后必经结点树中确定m和n的最低公共祖先（当m是根结点时，就是m本身）。所得到的这个结点l要么是m，要么是m的父结点，并且在后必经结点树中从l到n路径上的所有属于集合N的、除l之外的结点都控制依赖于m。

例如，考虑图7-4给出的流程图。它加上start结点后如图9-17所示，而其后必经结点树如图9-18所示。集合S由start→entry、B1→B2、B1→B3和B4→B6组成。基本CDG如图9-19所示。

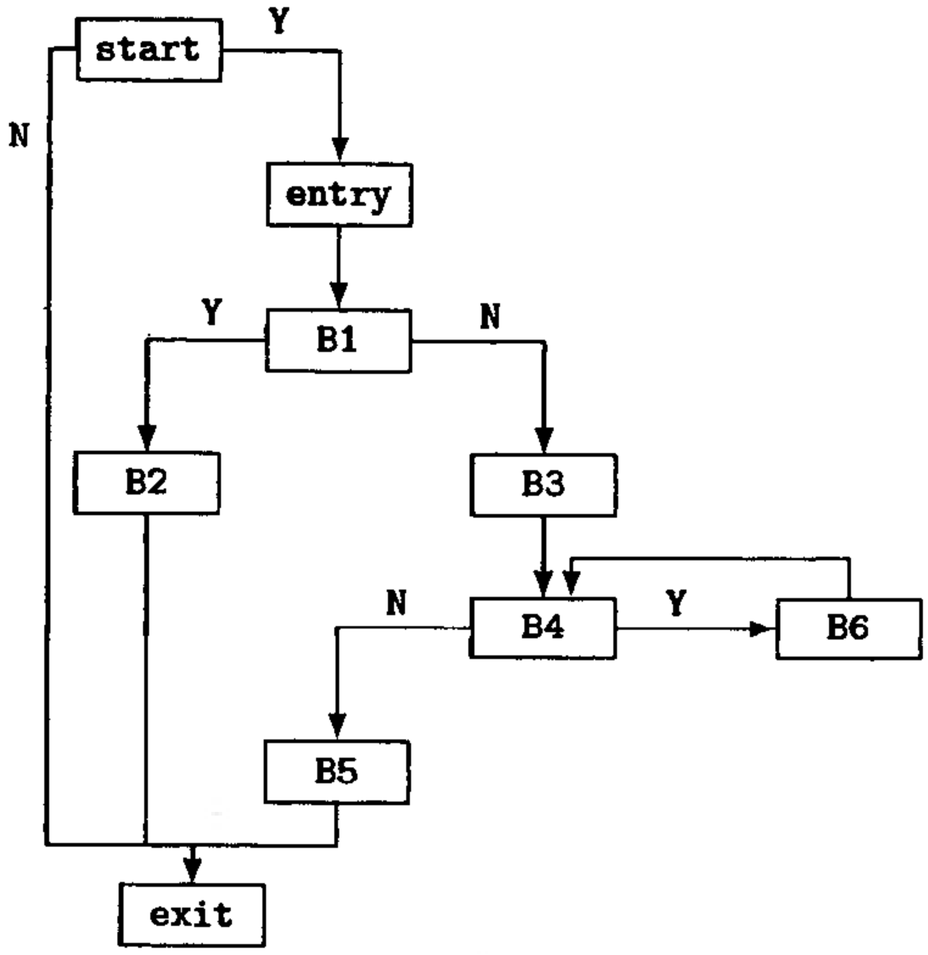


图9-17 在图7-4的流图中加入了start结点

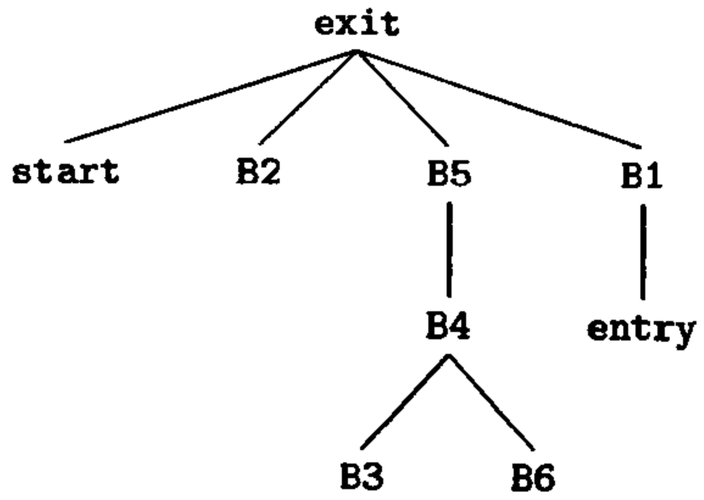


图9-18 图7-17流图的后必经结点树

区域结点的用途是将那些对同一个特定的谓词结点有控制依赖关系的所有结点组合在一起，使每个谓词结点如同原控制流图一样至多只有两个后继。我们的例子加入区域结点后如图9-20所示。

285

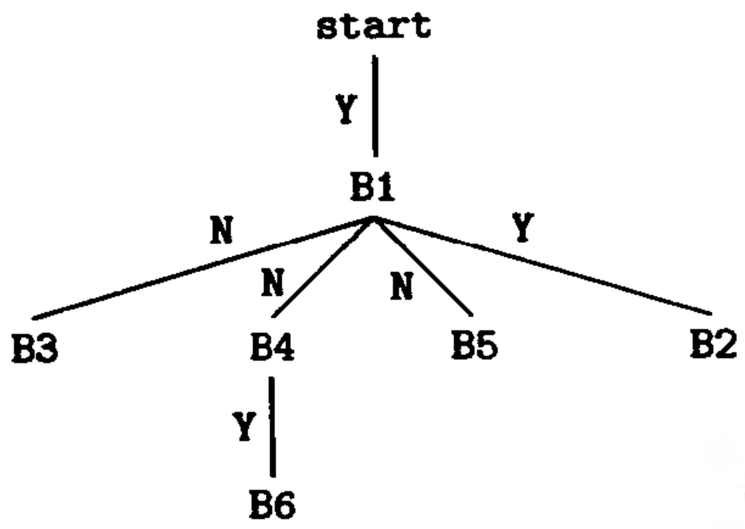


图9-19 图9-17流图的基本控制依赖图

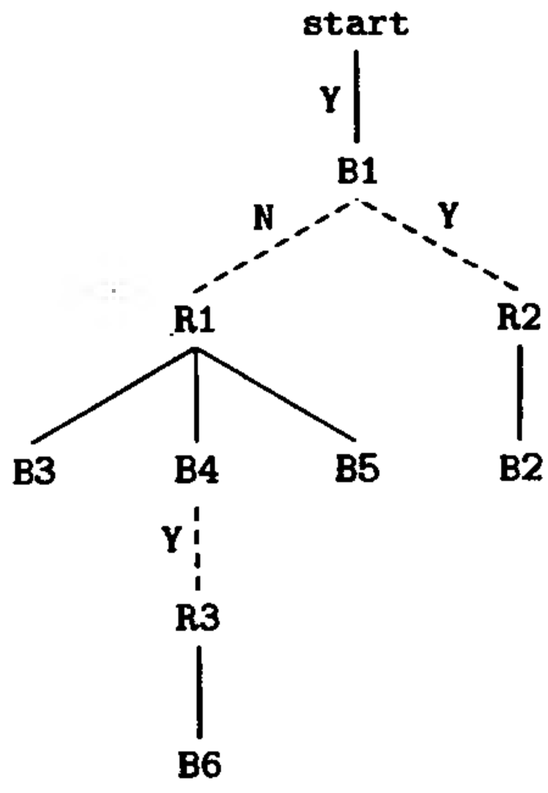


图9-20 图9-17流图的带有区域结点的控制依赖图

PDG的一个重要性质是，对于控制依赖于同一结点的所有结点，例如例子中的B3和B5，只要它们之间没有数据依赖关系，就可以并行执行。

一些文献中提出了其他几种其目的类似于程序依赖图的中间代码形式，包括依赖流图、程序依赖网和值依赖图。

bbs.theithome.com

9.6 动态分配的对象之间的依赖关系

前面我们已讨论了机器指令和数组访问之间的依赖关系及程序依赖图。另一个要考虑的方面是大量由指针链接的动态分配的数据结构，例如表、树、DAG，以及用于代数和符号语言（如LISP、Prolog和Smalltalk）的其他图结构。如果我们可以确定一个数据结构或图（例 286
如链表、树或DAG）决不会是共享的，或它的某些部分决不会是共享的（共享是指从两个变量同时通过一个指针链访问），我们就可能像数组的情形一样，改善它的存储分配或高速缓存分配。

这个领域的先驱性工作是Tenenbaum及Jones和Muchnick在20世纪70年代中期进行的，其主要目的是在一个事先只有数据对象有类型的语言中，给变量也赋予数据类型。近年来这个领域的研究很活跃。最近Deutsch、Hummel、Hendren和Nicolau等人的文章提出的若干方法虽能得到令人印象深刻的结果，但这些方法的计算量都很大（进一步的阅读见9.8节）。

我们简要地介绍Hummel、Hendren和Nicolau开发的一种技术，它所做的有3部分工作：(1)它开发了一种通过描述堆存储器中匿名存储单元之间的关系，对这些存储单元进行命名的机制；(2)它开发了刻画数据结构中存储单元之间的基本别名关系或没有别名关系的若干公理；(3)它用一个定理证明器来建立这些结构所期望的性质，例如一个特定的结构是一个以链表表示的队列，此队列的表项从一端加入，从另一端移出。

这种命名机制使用记录中的固定存储单元和字段的名字来指明关系，特别是它使用了*句柄*（handle）和*访问路径矩阵*（access-path matrices），句柄命名结构中的固定结点（通常是指针变量），并具有_h*var*的形式，其中*var*是变量名；访问路径矩阵表示句柄和变量之间的关系。这样，对于图9-21a中的C语言的类型声明和某些特殊的程序，可应用图b)中的公理。例如，第3条公理声称指针变量p通过left或right分量的一个或多个访问可到达的任意存储单元与p本身表示的存储单元是不同的。

```
typedef struct node {struct node *left;
                     struct node *right;
                     int val} node;
```

a)

```
Ax1: ∀p p.left ≠ p.right
Ax2: ∀p,q p ≠ q ⇒ p.left ≠ q.left,
                  p.left ≠ q.right,
                  p.right ≠ q.left,
                  p.right ≠ q.right
Ax3: ∀p p(.left|.right)+ ≠ p.ε
```

b)

图9-21 a) 递归数据结构的C语言类型声明的例子，b) 作用于从这个声明构造出来的结构的公理

图9-22给出了一个C函数，它使用了图9-21a定义的数据类型，并满足给定的公理。表9-1给出了恰在return语句之前的程序点的访问路径矩阵。给定图9-21b中的公理，定理证明器可 287
以证明该函数返回0。

表9-1 图9-22中程序在return语句之前那一点的访问路径矩阵。在行_h*var* 1和列*var* 2的交叉点上的值表示了从*var* 1原始的值到*var* 2现在的值的路径。“－”表示不存在这样的路径

	ptr	p1	p2
_hptr	ϵ	$left^+$	$right^+$
_hp1	-	$left^+$	$right^+$
_hp2	-	$left^+$	$right^+$

bbs.theithome.com

```
typedef struct node {struct node *left;
                     struct node *right;
                     int val} node;

int follow(ptr,i,j)
   struct node *ptr;
   int i, j;
{  struct node *p1, *p2;
   int n;
   p1 = ptr;
   p2 = ptr;
   for (n = 0; n < i; n++)
      p1 = p1->left;
   for (n = 0; n < j; n++)
      p2 = p2->right;
   return (p1 == p2);
}
```

图9-22 一个使用图9-21中定义的类型结构的C函数例子

注意，正如大多数强有力的定理一样，描述指针操作的定理是不可判定的。因此，定理证明器对于一个特定的问题可能给出“是”、“不是”或“可能”的回答。

9.7 小结

如我们在本章看到的，对于后面将要用两章的篇幅分别讨论的指令调度和数据高速缓存优化，依赖关系分析是一个重要的工具。

对于指令调度，依赖关系分析确定为正确执行程序在指令之间必须遵循的顺序关系，从
288 而也就确定了调度器通过重新安排指令顺序来改善程序性能的自由度。在进行指令调度时，调度器要考虑尽可能多的相关资源。它必须确定那些对寄存器和隐含资源（例如条件码）有影响和依赖于它们的指令之间的顺序。通常它还将尽可能地消除存储地址的歧义性，以便为调度提供最大的调度空间。存储地址歧义性的消除常常依赖于编译过程中前面各遍传递给它的信息。

作为数据高速缓存优化的工具，依赖关系分析的基本用途是，对于两个或更多给定的存储器引用，确定它们访问的存储区间是否重叠，如果重叠，它们之间的关系如何。通常这些引用是嵌套在一个或多个循环中的带有下标的数组引用。例如，确定两个（或所有）引用是否都写入相同的存储单元，或是否一个引用写入一个单元，另一个引用读取该单元，等等。通过确定存在哪些依赖关系，它们是由哪些循环携带的，从而为以改善数据高速缓存性能为目的的循环重排序或修改循环和数组引用提供了大量必需的信息。依赖关系分析也是自动向量化、并行化编译器的基本工具，但这些问题超出了本书的范围。

同时，我们讨论了一种相对较新的、称作程序依赖图的中间代码形式，这种程序依赖图将依赖关系明显地表示出来，并且被建议作为进行数据高速缓存优化和其他优化的基础。我们还提及了几种类似的中间代码形式，目前还不清楚它们之间哪一种，或是否有一种会成为重要的优化工具。

最后我们讨论了用指针访问的动态分配对象的依赖关系分析技术。这是一个研究了20余年的领域，虽然已经有了能有效地进行这种分析的几种方法，但它们都有非常大的计算量，这使得人们对这些技术能否成为商业化编译器的重要组件仍存在疑问。

bbs.theithome.com

9.8 进一步阅读

对如何使用依赖关系分析技术进行向量化或并行化感兴趣的读者请参阅[Wolf92]、[Wolf89b]、[Bane88]或[ZimC91]。

[BraH91]介绍了如何使用资源向量来计算等待时间。本书关于MIPS R4000的浮点流水线的描述是依据[KanH92]给出的。[ProF94]介绍了如何使用确定的有限自动机来计算等待时间。

依赖关系向量是Wolf和Lam在[WolL91]中定义的。关于循环嵌套中的一般依赖关系测试是NP－完全问题的证明可以在[MayH91]中找到。Banerjee和Towle开发的GCD测试在[Bane76]和[Towl76]中作了描述。可分的和弱可分的数组引用定义于[Call86]。其他正在使用的依赖关系测试方法如下所示:

依赖关系测试	参考文献
扩展GCD测试	[Bane88]
强和弱单索引变量(SIV)测试	[GofK91]
Delta测试	[GofK91]
Acyclic测试	[MayH91]
Power测试	[WolT90]
单循环余数测试	[MayH91]
Fourier-Motzkin测试	[DanE73]和[MayH91]
约束矩阵测试	[Wall88]
Omega测试	[PugW92]

[GofK91]和[MayH91]评估了若干种测试方法的适用性和实用性。

程序依赖图在[FerO87]中定义。可供选择的几种形式是[JohP93]定义的依赖关系流图、[CamK93]定义的程序依赖网和[WeiC94]定义的值依赖图。

给动态语言中的变量赋予类型的早期工作是Tenenbaum（[Tene74a]和[Tene74b]）和Jones及Muchnick（[JonM76]和[JonM78]）所做的。

研究存储器使用特性和递归数据结构之间的依赖关系特性的一些开拓性工作是由Jones和Muchnick完成的[JonM81a]。近期的工作反映在[WolH90]、[Deut94]和[HumH94]中。

9.9 练习

9.1 (a) 在下面基本块中的LIR指令之间存在哪些依赖关系？(b) 使用Build_DAG()构造该基本块的调度DAG，并画出所得到的依赖DAG。

```
r1 ← [r7+4](4)
r2 ← [r7+8](2)
r3 ← r2 + 2
r4 ← r1 + r2
[r5](4) ← r4
r4 ← r5 - r3
[r5](4) ← r4
[r7+r2](2) ← r3
r4 ← r3 + r4
r3 ← r7 + r4
r7 ← r7 + 2
```

9.2 令浮点加指令有如下的资源向量:

bbs.theithome.com

1	2	3	4	5	6	7
U	S,A	A,R	R,S			

290 假定LIR的加法指令`f4 ← f3 + 1.0`在时钟周期1可以启动执行，流水线中包含的指令在流水线的各级上使用的执行资源如下所示，计算该指令的等待时间。

1	2	3	4	5	6	7	8	9	10	11
M	U,A	A	S	R,S	S	M	M,U	A	S,A	R

9.3 Hewlett-Packard的PA-RISC编译器由下至上，即从叶结点到根结点来构造基本块的调度DAG，以便追踪对条件标志的使用和对这些条件标志进行设置的指令。编写一个名为`Build_Back_DAG()`的`Build_DAG()`的版本，实现对LIR代码以这种方式由下至上地构造DAG。

ADV 9.4 研究将依赖图从基本块扩展到不包含循环的任意单入口、单出口子流图的表示。这能为指令调度提供更大的自由度吗？如果能，给出一个例子。如果不能，为什么？

9.5 下面3层嵌套的HIR循环的迭代空间遍历是什么？

```
   for i ← 1 to n do
      for j ← n by -1 to 1 do
         for k ← 1 to n+1 do
S1          A[i,j,k] ← A[i,j-1,k-1] + A[i-1,j,k]
S2          B[i,j-1,k] ← A[i,j-1,k-1] * 2.0
S3          A[i,j,k+1] ← B[i,j,k] + 1.0
         endfor
      endfor
   endfor
```

9.6 上题的循环嵌套有什么样的执行顺序和依赖关系？

9.7 (a) 写一个算法，对于给定的距离向量，它产生包含该距离向量的最小依赖关系向量，这里d_1包含d_2当且仅当d_2表示的所有依赖关系也被d_1表示。(b) 对方向向量做同样的工作。

9.8 给出一个使用GCD测试能够确定不存在依赖关系的3层循环嵌套。

RSCH 9.9 研究[GofK91]描述的Delta测试。(a) 它是如何测试依赖关系的？(b) 它的效率如何？(c) 它的开销如何？

RSCH 9.10 研究[PugW92]描述的Omega测试。(a) 它是如何测试依赖关系的？(b) 它的效率如何？(c) 它的开销如何？

RSCH 9.11 研究如何将依赖关系扩展到并行语言中对共享变量的同步访问，即在这种情况下，

291 $S_1 \delta^{f} S_2$、$S_1 \delta^{a} S_2$等的含义是什么？

bbs.theithome.com

超星公司制作 请尊重作者版权

第10章 别名分析

别名分析指的是判定可能用两种以上的方式来访问的存储位置。例如，对于C的变量，可以取其地址，并且可以通过变量名或者通过指针来读写它，如图10-1所示。图10-2是图10-1所示情形的形象化表示，其中方框代表变量，每一个方框用一个变量名来标识，框内给出的是变量的值。如第8章提到的，判断程序中别名可能存在的范围是保证程序优化正确性的关键，而对尽可能激进的优化而言，使所找到的别名集尽可能小则相当重要。假如我们碰巧遗漏了一个应当识别出来的别名，就很容易导致产生不正确的程序代码。考虑图10-3中的C代码段，其中，当且仅当对f()的调用和通过指针q的赋值语句未改变k和a的值，第二个k＝a＋5的赋值才是冗余的。因为k的地址传递给了f()，故k的值有可能被改变。又因为q是外部量，故有可能f()或者过程exam1()中某些更早的代码已使它指向了a或k。如果出现这两种情况之一，第二个k＝a＋5的赋值就可能不是冗余的；如果这两种情况都不出现，这个赋值就是冗余的。这个例子说明了过程内和过程间别名判定的意义。在实际中，过程内的别名判定通常更重要。本章详细地讨论过程内的别名判定，而将过程间的别名分析留到第19章讨论。

```
main()
{   int *p;
    int n;
    p = &n;
    n = 4;
    printf("%d\n",*p);
}
```

图10-1 C的简单指针别名用法

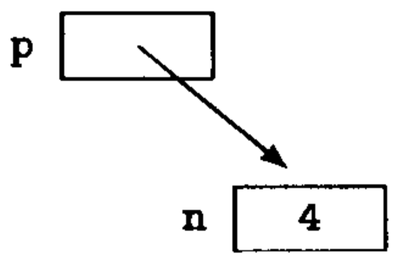


图10-2 图10-1中在printf()调用点变量之间的关系

```
exam1( )
{   int a, k;
    extern int *q;
     . . .
    k = a + 5;
    f(a,&k);
    *q = 13;
    k = a + 5;   /* redundant? */
     . . .
}
```

图10-3 说明别名分析重要性的例子

虽然高质量的别名信息对激进的和正确的优化是重要的，但许多程序却只需要最少量的这种信息。尽管C程序可以有任意复杂的别名，但对大多数C程序而言，只需假定只有那些取了其地址的变量会发生别名，并且任何指针值变量可能指向它们就足够了。多数情况下这种假定对优化的限制最小。但另一方面，假若我们有一个含有200个变量的C程序，其中100个变量取了其地址，另100个变量是指针，则显然激进的别名分析对可施加于该程序的很多优化而言就很重要。

在后面关于全局优化的几章中，我们一般假定已经执行过了别名分析并且不再提及它。但无论如何读者必须知道，正如前面的例子说明的那样，别名分析是正确实现大多数优化的保证。

区分可能（may）别名信息和一定（must）别名信息对别名分析会有所帮助。前者指出在流图的某条路径上可能出现的情况，而后者指出在流图的每一条路径一定出现的情况。例如，如果一个过程的每一条路径都含有将变量x的地址赋给变量p的赋值，并且p只被赋予了此值，则“p指向x”就是这个过程的一定别名信息。另一方面，如果该过程含有的路径中有一条路径是将变量y的地址赋给指针变量q，而其他的路径是将变量z的地址赋给指针变量q，则“q可能指向y或z”就是可能别名信息。

区分流敏感和流不敏感别名信息对别名分析也是有帮助的。流不敏感（flow-insensitive）

bbs.theithome.com

293
~
294

信息与过程内的控制流无关，而流敏感（flow-sensitive）别名信息与控制流相关。关于别名的流不敏感说法的一个例子是："p可能指向x，因为存在着一条路径，在此路径上p被赋予了x的地址。"这句话简单地指出一个具体的别名关系可能存在于过程的任何地方，因为这个别名关系确实在过程的某个地方存在。对于流敏感，则可以指出"p指向基本块B7中的x"。上面提到的那个对取了其地址的变量做简单区分的C别名分析方法是流不敏感的；下面我们将要详细介绍的是流敏感的方法。

概括而言，区分可能与一定是重要的，因为它告诉我们某种属性是否一定具有，因此可以依靠它；或者它只是可能具有，因此必须承认它但不能依靠它。区分流敏感性是重要的，因为它决定了所考虑问题的计算的复杂性。流不敏感问题的求解一般通过解子问题，然后合并这些子问题的解而提供对整个问题的解，它与控制流无关。另一方面，流敏感问题需要沿着流图的控制流路径来计算它们的解。

别名的正式表述依赖于我们涉及的是可能信息还是一定信息，以及这种信息是流敏感的还是流不敏感的。具体情形如下：

1. 流不敏感可能信息（flow-insensitive may information）：在这种情况下，别名是过程变量集上的二元关系$alias \in Var \times Var$，我们有$x\ alias\ y$，当且仅当x和y可能（也许在不同时刻）引用相同的存储单元。这个关系是对称的和非传递的[⊖]。非传递性是因为a和b在某点可能引用相同的存储单元，并且b和c在某点也可能引用同一个存储单元这一事实，并不能使我们得出有关a和c的结论——关系$a\ alias\ b$和$b\ alias\ c$只是在过程的不同点成立而已。

2. 流不敏感一定信息（flow-insensitive must information）：在这种情况下，别名也是二元关系$alias \in Var \times Var$，但有不同的含义。我们有$x\ alias\ y$，当且仅当x和y在过程的整个执行中一定引用相同的存储单元。这个关系是对称的和传递的，如果a和b在过程的整个执行中一定引用相同的存储单元，并且b和c在过程的整个执行中一定引用相同的存储单元，则显然a和c也一定引用相同的存储单元。

3. 流敏感可能信息（flow-sensitive may information）：在这种情况下，别名可以看成是二元关系的集合，过程中的每一个程序点上（即，两条指令之间）有这样一个集合，它描述的是在这一点上各个变量之间的关系。如果将别名表示成从一个程序点和一个变量至抽象存储位置集合的映射函数，则容易更清晰地说明其道理。用公式来表述，即，对于程序中的某点p和变

295

量v，$Alias(p, v) = SL$意味着变量v在点p可能引用SL中的任意单元。于是，如果$Alias(p, a) \cap Alias(p, b) \neq \emptyset$并且$Alias(p, b) \cap Alias(p, c) \neq \emptyset$，则可能有$Alias(p, a) \cap Alias(p, c) \neq \emptyset$，但这不是一定有的情形。同样，如果$p1$、$p2$和$p3$是不同的程序点，$Alias(p1, a) \cap Alias(p2, a) \neq \emptyset$，并且$Alias(p2, a) \cap Alias(p3, a) \neq \emptyset$，则类似地可能也有$Alias(p1, a) \cap Alias(p3, a) \neq \emptyset$。

4. 流敏感一定信息（flow-sensitive must information）：在这种情况下，别名最好表示为映射程序点和变量至抽象存储位置（不是存储位置的集合）的函数。用公式来表述，即，对于某个程序点p和变量v，$Alias(p, v) = l$意味着变量v在点p一定引用存储单元l。于是，$Alias(p, a) = Alias(p, b)$并且$Alias(p, b) = Alias(p, c)$，则一定也有$Alias(p, a) = Alias(p, c)$。类似地，如果$p1$、$p2$和$p3$是不同的程序点，$Alias(p1, a) = Alias(p2, a)$，并且$Alias(p2, a) = Alias(p3, a)$，则一定也有$Alias(p1, a) = Alias(p3, a)$。因此，关于特定程序点的流敏感一定别名信息是变量之间的传递关系，关于特定变量的流敏感一定信息是程序点之间的传递关系。也容易看出这两个关系都是对称的。

如果将语言中的指针考虑进来，这种区分将进一步复杂化，例如C语言的情况。于是任何

⊖ 我们不关心这个关系是否是自反的或反自反的，因为$x\ alias\ x$没有意义。

bbs.theithome.com

超星公司制作请尊重作者版权

可能引用存储单元的对象，如记录中的指针域，都有可能与引用存储单元的其他任何对象发生别名。因此在具有指针的语言中，别名是存储单元引用之间的关系或函数，其中引用包括任何以指针作为其值的对象，而不仅仅是变量。

尽管产生别名的原因因语言而异，但各种语言的别名计算具有共性的成分。例如，一种语言可能允许两个变量的存储位置相互重叠，或者允许其中一个是指向另一个的指针，或者不允许有这种情况，但不论这些语言特定的规则如何，如果在给定的执行点，变量c指向变量b，而变量b又指向变量a，则顺着从c开始的指针可以到达a。因此，我们将别名计算分为两部分：

1. 语言特有的部分，称为*别名收集器*（alias gatherer），我们希望它由编译前端提供；

2. 优化器中的一个单独部分，称为*别名传播器*（alias propagator），它利用前端发现的别名关系执行数据流分析，组合别名信息并将它们传递到需要它们的程序点。

语言特有的别名收集器可以发现由于下述原因发生的别名：

1. 为两个对象分配了重叠的存储单元；

2. 数组、数组片段或数组元素的引用；

3. 指针引用；

4. 参数传递；

5. 上面各种机制的组合。

296

在深入研究细节之前，我们先考虑可以计算的别名信息的粒度以及它的作用。值得注意的是，可能有这种情况，可以证明两个变量在过程的某段代码中不会发生别名，但在过程的其他部分，它们要么存在别名，要么不能判定不存在别名。图10-4给出了这种情况的一个例子。在第一段代码中，q指向a，而在第二段代码中，q指向b。假若我们要做的是关于整个过程的流不敏感可能别名计算（假定不存在其他有可能影响别名的语句），则可以简单地断定q指向a或b。这使得我们不能将赋值*q＝i提到for循环之外。另一方面，如果我们在较小的粒度范围来计算别名，就可以发现在该循环内q不会指向a，这使得我们能够用放置在循环之后的*q＝100，甚至b＝10来替代循环内的赋值*q＝i。尽管这种程度的识别力在许多情况下无疑是有价值的，但它在编译时间和存储空间上超出了可接受的范围。一种选择是Sun编译器（参见21.1节）的做法，即（可选的）激进别名信息计算，而另一种是MIPS编译器的做法，它简单地假定任何计算了其地址的变量都会发生别名。

```
exam2( )
{  int a, b, c[100], d, i;
   extern int *q;
    . . .
   q = &a;
   a = 2;
   b = *q + 2;
    . . .
   q = &b;
   for (i = 0; i < 100; i++) {
      c[i] = c[i] + a;
      *q = i;
   }
   d = *q + a;
}
```

图10-4 过程不同部分中的不同别名

因此，我们将为给定实现选择合适粒度的工作留给具体编译器的开发人员。这里介绍一种方法，它能区别过程中的各个程序点；读者很容易将它修改成为不区别程序点的方法。

10.1 各种现实程序设计语言中的别名

下面我们考虑别名信息的形式，这些信息应当由前端收集，并传递给别名传播器。我们考察四种常用的语言，即Fortran 77、Pascal、C和Fortran 90，并假定读者熟悉其中的每一种语言。在探讨了这四种语言的别名情况之后，我们介绍一种别名收集方法，这种方法在某些方面与Hewlett-Packard 的PA-RISC编译器采用的方法有些类似，但其别名传播部分有相当大的不同。Hewlett-Packard 的PA-RISC编译器的传播方法是流不敏感的，而我们的方法是流敏感的——事

297

bbs.theithome.com

实上，我们使用的传播方法采用了数据流分析技术，即，它执行数据流分析来判别别名。

10.1.1 Fortran 77中的别名

ANSI标准Fortran 77创建别名的途径比较少，并且别名几乎都可在编译时精确地检测出来。但是，我们必须记住的一点是，在这个领域已经形成的程序设计惯例偶尔会违背Fortran 77标准；而多数编译器遵循的是惯例（至少在某种程度上），而不完全是标准。

EQUIVALENCE语句可以用来指定两个或多个标量变量、数组变量和/或数组变量的连续元素开始于相同的存储单元。这些变量局部于说明它们等价的子程序，除非它们同时也出现在COMMON语句中。出现在COMMON语句中的变量可以由若干子程序访问。因此，只要等价变量不出现在公用区中，由EQUIVALENCE语句创建的别名就完全是局部的，并且是静态可确定的。

COMMON语句用相同的存储单元结合不同子程序中的变量。对于现代程序设计语言，COMMON是与众不同的，它用存储位置结合变量，而不是用名字。判别公用区变量的作用需要进行过程间分析，但当一个变量是公用变量时，你至少可以局部地判别出该变量可能受到其他子程序的影响。

在Fortran 77中，参数传递采用这样一种方式，即，只要实参与有名的存储单元（例如，它是一个变量或数组元素，而不是一个常数或表达式）结合，被调用的子程序就能够通过对形参赋值而改变对应实参的值[⊖]。标准中并没有指明形实结合机制是传地址还是传值得结果；两种方法都能正确地实现Fortran 77的约定。

Fortran 77标准15.9.3.6节指出，当你传递同一个实参给子程序的两个或多个形参时，或者当实参是公用区中的对象时，该子程序以及其后的调用链中的任何子程序都不能给这个实参赋新的值。如果编译器坚持这个规则，则Fortran 77中惟一的别名就是由EQUIVALENCE和COMMON创建的那些别名。不幸的是，有些程序违背了这一规则，并且编译器有时用它来决定一种程序结构是否可以用特殊的方式被优化。于是，我们可以认为还存在着一个“实用的”Fortran 77，它允许通过参数传递创建别名（参见15.2节的例子），但这样做可能会处于危险的
298 境地——有些编译器可能会持续地支持它，另一些编译器则不会一直支持它，甚至有些编译器可能根本就不支持它。

Fortran 77除了公用区变量外没有全局存储单元，因此如果不将变量放在公用区中，或者不违背刚才叙述的参数传递约定，就没有其他途径能够创建非局部对象的别名。

有若干Fortran 77编译器包含了Cray的扩充。这些扩充中提供了一种受限的指针类型。指针变量可以被设置为指向一个标量变量、数组变量或绝对存储地址（称为该指针的被指对象），并且，在程序执行期间可以改变指针的值。但是，它不可以指向另一个指针，另外，被指对象不能出现在COMMON或EQUIVALENCE语句中，也不能是形参。这一扩充大大地增加了产生别名的可能性，因为多个指针可能指向同一个存储单元。另一方面，Cray编译器在打开优化开关编译时假定程序不会有两个指针指向相同的存储单元，并且更广泛地假定被指对象决不会覆盖另一个变量的存储空间。显然，这给程序员增加了别名分析的负担，并且会导致程序因为优化开关打开与否而产生不同的结果。不过这样做的好处是，编译器无须对指针进行别名分析就可以进行优化处理，否则就需要对指针做最坏情况的假设才能进行处理。

10.1.2 Pascal中的别名

在ANSI标准Pascal中创建别名有好几种机制，包括变体记录、指针、变量参数、被嵌套过

⊖ Fortran 77实际的术语是“实参”和“哑参”。

bbs.theithome.com

程对非局部变量的访问、递归，以及函数的返回值。

用户定义的记录类型的变量可以有多种变体，而且变体可以是加标志的或者未加标志的。这里允许多个未加标志的变体类似于具有Fortran 77中的等价变量——如果一个变量是未加标志的变体记录类型，则它的变体域可以由两个以上的名字集合访问。

Pascal的指针类型变量限制为要么具有值nil，要么指向已指明具体类型的对象。这种语言没有提供获得现存对象地址的手段，故非空指针只能指向由过程new()动态分配的对象。new(p)的参数p是一个指针变量，它分配一个其类型是p所指类型的对象，并使p指向该对象[⊖]。给定类型的指针变量可以对相同类型的指针变量赋值，因此，多个指针可以指向同一个对象。这样，一个对象在同一时刻可通过若干个指针来访问，但它不能同时既有自己的变量名，又可通过指针来访问。

Pascal过程的参数是值参数或变量参数。被调用过程不能通过参数改变传递给值参数的实参，因而值参数不会创建别名。但是，变量参数允许被调用过程改变与之相结合的实参，因而它会创建别名。 299

另外，Pascal允许嵌套过程定义，内层过程可以访问外层过程定义的变量，只要它们是可见的，即，只要在嵌套序列中这两个过程之间没有另外的过程定义同名的变量。因此，例如，在Pascal程序中一个动态分配的对象可以作为变量参数既通过局部声明的指针也通过非局部声明的指针来访问。

Pascal的过程可以是递归的，因此在内层作用域声明的变量对多次递归调用而言是可访问的，并且一次调用对应的局部变量可作为更深一层调用的变量参数。

最后一点，Pascal过程可以返回指针，因而可以创建动态分配对象的别名。

10.1.3 C中的别名

ANSI标准C中有一种创建静态别名的机制，即union类型，它类似于Fortran 77的EQUIVALENCE结构。一个联合类型可以声明若干个成员，每一个成员的存储空间是重叠在一起的。但是，C的联合类型不同于Fortran的等价变量，因为它可以通过指针来访问，并且可以动态分配空间。

注意上面最后一句，我们并没有说“它是动态分配的，所以用指针来访问”。C允许动态分配对象，并且引用它们必须通过指针，因为没有动态创建名字的机制。这种对象可由多个指针访问，故指针可能会相互别名。此外，在C中允许用 & 运算符取对象的地址，不论这个对象是静态分配的、自动分配的还是动态分配的，并且也允许通过赋予了其地址的指针来读写它。

C也允许对指针进行算术运算，并将它视为等同于数组索引——对指向数组元素的指针增加1导致它指向该数组的下一个元素。假设我们有代码段：

```
int a[100], *p;
. . .
p = a;
p = &a[0];
a[1] = 1;
*(p + 2) = 2;
```

则p的两个赋值语句赋给它的是完全相同的值，即数组a[]第0个元素的地址，其后的两个赋值语句分别将1赋给a[1]，2赋给a[2]。即使对于一个长度声明为n，元素编号从0到$n-1$的C数

⊖ new()可以有另外的一些参数，这些参数指明第一个参数指向的记录类型的嵌套变体；在这种情况下，它分配一个指定变体类型的对象。

bbs.theithome.com

超星公司制作 请尊重作者版权

300 组b[]，取b[*n*]的地址也是合法的，如下面的例子所示:

```
int b[100], p;
. . .
for (p = b; p < &b[100]; p++)
*p = 0;
```

但是直接引用b[*n*]是非法的。因此，指针值表达式可能与数组元素别名，并且与它别名的这个元素可能随时间而发生变化。可以想象得到，指针算术能够不分皂白地横扫存储器，并创建任意别名，但ANSI C标准中规定，做这种动作的代码的行为是不确定的（参见ANSI C标准3.3.6节）。

C也能够通过参数传递和从函数返回指针值而创建别名。尽管C中所有参数都是传值的，但由于参数可以是指向任何对象的指针，故它们也能创建别名。另外，C对用两个不同的参数传递同一个对象给函数也没有限制，例如，对下面这个函数，

```
f(i,j)
int *i, *j;
{   *i = *j + 1;
}
```

与标准Fortran 77不同，它可用f(&k,&k)来调用。进一步，实参也可以指向全局变量，使得全局变量既可通过其名字，也可通过指针来访问。作为函数值返回的指针可以指向该函数和该函数的调用者都可访问的任何对象，因此也可能与传递给函数的指针实参或任何具有全局作用域的对象别名。

同Pascal一样，递归也会创建别名——在递归函数的一次调用中指向局部变量的指针可以传送给该函数较深一层的调用，并且静态变量对多层递归调用是可访问的。

10.1.4 Fortran 90中的别名

标准Fortran 90包含Fortran 77作为子集，因此Fortran 77创建别名的所有可能性都适合于Fortran 90。此外，Fortran 90还有另外三种机制可以创建别名，即指针、递归和内部过程。

Fortran 90指针能够引用任何具有TARGET属性的对象，这种对象可以是任何动态分配的对象，也可以是声明具有此属性的有名对象。简单变量、数组和数组片段都有可能成为被指目标。

递归创建别名基本上与Pascal和C的途径相同，惟一不同的是Fortran 90的递归过程必须用RECURSIVE声明。

内部过程创建别名的方法也与Pascal相同——非局部变量也可以通过形实结合机制来访问。

301 Fortran 90标准扩充了对Fortran 77标准中通过这种别名而改变非局部变量值的限制，但是，我们的看法是，在实际中将很可能仍然遵守原来的限制。

10.2 别名收集器

为了描述在Fortran 77、Pascal、C、Fortran 90和其他语言中遇到的别名种类，我们使用若干关系和几个函数。这些关系表示语言对象之间可能出现的别名关系，这些函数将潜在的别名对象映射到抽象存储位置。这些关系和函数之所以归类于“潜在可能的”，是因为我们宁愿过于保守，也不能造成程序不正确——如果在两个对象之间可能存在别名关系，而又不能证明这个关系不存在，则我们必须记录它可能存在，以免由于遗漏了实际存在的别名而导致可能用一种与程序原语义不一致的方法来优化程序。

在下面的讨论中我们将看到，区分可能存在的别名，其精细程度可有一系列的选择。例如，如果有一个C或Pascal结构s，它含有两个成员s1和s2，则s的存储空间与s.s1和s.s2的存储空间重叠，但s.s1和s.s2的存储空间相互不重叠。作这种区分是否重要，有一些能指导我们

bbs.theithome.com

超星公司制作 请尊重作者版权

进行选择的折衷方法。做区分一般需要更多的编译空间，通常也需要更多的编译时间，并且可能产生更好的代码。作为编译器的编写者，我们可以作出一种选择并用于所有情况，也可以让用户来选择一种或另一种，这通常是为优化而做的选择工作的一部分。采用这些方法中的一种还是另一种，受我们在编写编译器时能够投入的精力的支配，但也应受到能对这些方法产生不同效果起决定性作用的经验的支配。

我们可以选择区分动态分配的存储区域与其他存储区域，也可以不区分。如果区分它们，就需要一种手段来命名这种存储区域，并需要一种用有限空间来表示整个别名信息的方法。在下面的处理中，我们不区分动态分配的存储区域，而是简单地将它们按类型合在一起或全部合在一起，因为对于我们要处理的这种语言而言，这是适合的做法。

另外，如本章一开始时提到的，我们可以选择流敏感的或流不敏感的别名分析，即，是否区分在过程个别点的别名关系。这里的分析选择区分它们；将我们的分析所收集的信息合并，使得别名分析不区分过程中的各个点是较为容易的练习。

我们从源代码的各个语句收集各项别名信息，然后将它们传递给优化器中称为别名传播器（下一节讨论）的部分，别名传播器将它们传播到过程内的所有点。别名收集器所作用的过程流图由控制流边连接的各个语句组成。我们也可以有另一种选择，即使用基本块并合并基本块中各个语句的计算结果作为这个基本块的结果。 302

令*P*表示一个程序点，即，程序中两条语句之间的点；在流图表示中，程序点*P*标记一条边。程序点entry+和exit-分别是直接跟在入口结点之后和直接位于出口结点之前的边。令*stmt*(*P*)表示流图中直接先于*P*的（惟一的）语句。

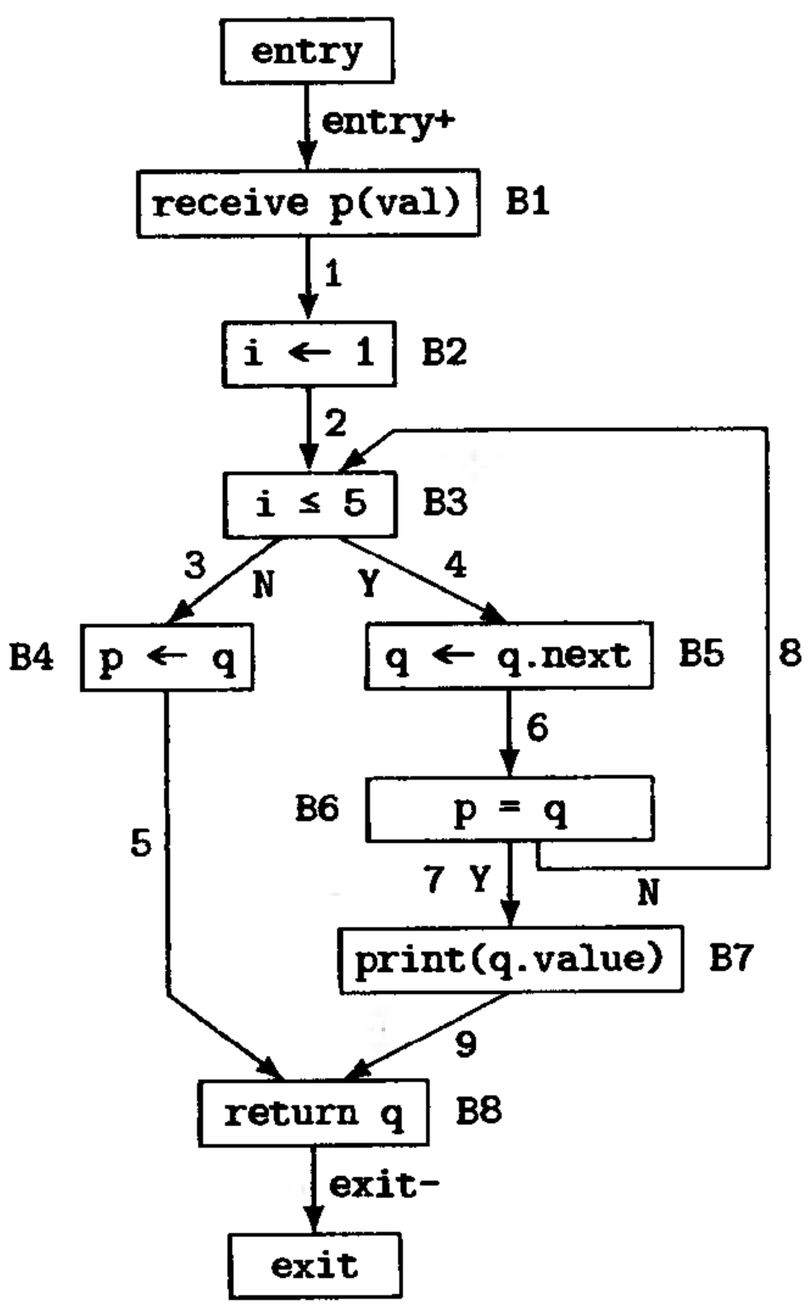


图10-5 一个给出别名概念的流图例子

图10-5中的流图是为说明别名收集和别名传播中用到的一些概念而设计的。它的每一个结点只含一条指令，其边用程序点标记，即，entry+,1,2,...,和exit-。语句*stmt*(1)是receive P(val)，语句*stmt*(exit-)是return q。

令*x*代表一个标量变量、数组、结构或指针值等，并令$mem_P(x)$表示在程序点*P*与对象*x*相连的抽象存储单元。令*star*(*o*)表示由语言对象*o*占据的静态或动态分配的存储区域。令*anon*(*ty*)表示为类型是*ty*的对象分配的“匿名”动态存储区域，*anon*表示过程中所有动态分配的存储区域。我们假定所有有类型的匿名区域是不同的，即，

$\forall ty1, ty2$，如果$ty1 \neq ty2$，则$anon(ty1) \neq anon(ty2)$

对于所有的*P*和*x*，$mem_P(x)$要么是*star*(*x*)，若*x*是静态分配或自动分配的；要么是*anon*(*ty*)，若*x*是动态分配的（其中*ty*是*x*的类型）；要么是*anon*，若*x*是动态分配的并且不知道它的类型。我们用*nil*表示空指针值。

在图10-5的点2处，与*i*相连的存储位置是*star*(i)，记做mem_2(i)；而$mem_5(q)$是*anon*(ptr)，其中ptr表示指针类型。另外ptr_9(p)= ptr_9(q)。

令*any*表示所有可能的存储单元集合，*any*(*ty*)表示类型为*ty*的所有可能的存储单元；*any*(*ty*)和*anon*(*ty*)对Pascal是有用的，因为Pascal限制每一个指针只指向特定类型的对象。令*globals*表

bbs.theithome.com

示所有可能的全局可访问的存储单元。对于Fortran，*globals*包括所有公用区单元；对于C，它包括所有在过程体外声明的变量和具有extern属性的变量。

我们定义一组将点P和可存放值的对象x（即，变量、数组和结构成员等）映射到抽象存储单元的函数，如下：

1. $ovr_P(x)$＝在程序点P可能与x重叠的抽象存储单元集合。

2. $ptr_P(x)$＝x在程序点P可能指向的抽象存储单元集合。

3. $ref_P(x)$＝在程序点P通过从x开始的任意多层的引用可到达的抽象存储单元集合。注意，如果我们定义$ref_P^1(x) = ptr_P(x)$并且对所有$i > 1$，

$$ref_P^i(x) = ptr_P(fields(ref_P^{i-1}(x)))$$

其中，若x是指针，$fields(x)$是x；若是结构，$fields(x)$是x中指针值成员的集合，则，

$$ref_P(x) = \bigcup_{i=1}^{\infty} ref_P^i(x)$$

在许多情况下，计算$ref_P(x)$可能导致不终止。任何实际的别名传播器都必须有一种使得这种计算能终止的方法，至少在最坏的情况下应能通过若干次的迭代使计算结果变成*any*或*any*(*ty*)而终止计算。合适的方法取决于要解决的具体问题，以及希望获得的信息的详细程度。

4. $ref(x)$＝通过从x开始的任意多层的引用可到达的抽象存储单元集合，与程序点无关。

303～304

我们还定义一个谓词$extal(x)$，它为真，当且仅当x可能有（可能是未知的）外部于当前过程的别名。并定义如下两个映射过程名到抽象存储单元的函数：

1. $uses_P(pn)$＝语句$stmt(P)$中调用过程pn可能使用的抽象存储单元集合；

2. $mods_P(pn)$＝语句$stmt(P)$中调用过程pn可能修改的抽象存储单元集合。

现在考虑标准Fortran 77的别名规则。用我们的表示方法可相当容易地将它们表示如下：

1. 如果在一个子程序中变量a和b是等价的，则对于该子程序内的所有点P，$ovr_P(a)=ovr_P(b)=\{mem_P(a)\}=\{mem_P(b)\}$，并且

2. 如果在子程序中说明变量a是公用区中的变量，则$extal(a)$为真。

由(2)得出，对于任何在$stmt(P)$中出现的对子程序pn的调用，如果$extal(a)$为真，则$\{mem_P(a)\} \subseteq uses_P(pn)$；同样，$\{mem_P(a)\} \subseteq mods_P(pn)$当且仅当$a$是$stmt(P)$中调用$pn$的实参。

与Fortran 77相比，Cray Fortran扩充、Fortran 90和Pascal都给别名判定增加了复杂性，但C的别名判定是最极端的情况，下面就来讨论它。C的别名分析需要更多的规则并且描述起来更复杂。我们假定对于别名分析，数组是未知结构的聚合，因此假定指向数组元素的指针与整个数组别名。

注意，下面的规则集合不是C的全部描述，因为对于我们的讨论，只要给出所需规则的类型样式，以及关于如何构建它们的模型就足够了。

在下面关于C的所有规则中以及下一节中，P是一个程序点，P'是P前面的一个（通常是惟一的）程序点。如果P有多个前驱，合适的扩展是如后面10.3节所述，将所有前驱的右端合并到一起。

1. 如果$stmt(P)$将一个空指针赋给指针p，则

$$ptr_P(p) = \emptyset$$

这包括从存储分配程序（如C的库函数malloc()）的调用的失败返回。

2. 如果$stmt(P)$将一个动态分配的存储区域赋给指针p，例如，通过调用malloc()或calloc()，则

bbs.theithome.com

$ptr_P(p) = anon$

3. 如果*stmt*(*P*)是“$p = \&a$”，则

$ptr_P(p) = \{mem_P(a)\} = \{mem_{P'}(a)\}$

4. 如果*stmt* (*P*)是“$p1 = p2$”，其中$p1$和$p2$是指针，则

$$ptr_P(p1) = ptr_P(p2) = \begin{cases} mem_{\texttt{entry+}}(*p2) & \text{若 } P' = \texttt{entry+} \\ ptr_{P'}(p2) & \text{否则} \end{cases}$$

305

5. 如果*stmt*(*P*)是“$p1 = p2 -> p3$”，其中$p1$和$p2$是指针，$p3$是一个指针成员，则

$ptr_P(p1) = ptr_{P'}(p2->p3)$

6. 如果*stmt*(*P*)是“$p = \&a[expr]$”，其中p是指针，a是数组，则

$ptr_P(p) = ovr_P(a) = ovr_{P'}(a) = \{mem_{P'}(a)\}$

7. 如果*stmt*(*P*)是“$p = p + i$”，其中p是指针，i是整数值，则

$ptr_P(p) = ptr_{P'}(p)$

8. 如果*stmt*(*P*)是“$*p = a$”，则

$ptr_P(p) = ptr_{P'}(p)$

并且如果$*p$的值是指针，则

$ptr_P(*p) = ptr_P(a) = ptr_{P'}(a)$

9. 如果*stmt*(*P*)是关于两个指针值变量的测试“$p==q$”，且*P*标志该测试的Y出口，则

$ptr_P(p) = ptr_P(q) = ptr_{P'}(p) \cap ptr_{P'}(q)$

因为从Y出口使得p和q指向相同的位置。⊖

10. 对于一个其成员是从$s1$到sn的结构类型*st*，一个类型为*st*的静态或自动对象*s*，以及每一个点*P*，

$$ovr_P(s) = \{mem_P(s)\} = \bigcup_{i=1}^{n} \{mem_P(s.si)\}$$

而且，对于每一个i，

$\{mem_P(s.si)\} = ovr_P(s.si) \subset ovr_P(s)$

且对于所有$j \neq i$，

$ovr_P(s.si) \cap ovr_P(s.sj) = \emptyset$

11. 对于一个其成员是从$s1$到sn的结构类型*st*，以及一个使得*stmt*(*P*)分配一个类型为*st*的对象*s*的指针p，

$$ptr_P(p) = \{mem_P(*p)\} = \bigcup_{i=1}^{n} \{mem_P(p->si)\}$$

306

并且对于每一个i，

$\{mem_P(*p->si)\} = ptr_P(p->si) \subset ptr_P(s)$

并且对于所有的$j \neq i$，

⊖ 在N出口没有新的信息是可用的，除非我们使用更为复杂的、与Jones和Muchnick [JonM81b]开发的数据流分析的关系方法类似的别名分析方法，将各个变量的当前别名元组集合包含进来。

bbs.theithome.com

$$ptr_P(p->si) \cap ptr_P(p->sj) = \emptyset$$

和对所有的其他对象x，

$$ptr_P(p) \cap \{mem_P(x)\} = \emptyset$$

因为分配给对象的每一个成员的存储空间有不同的地址。

12. 对于一个具有成员从$u1$到un的联合类型ut，一个类型为ut的静态或自动对象u，以及每一个点P，

$$ovr_P(u) = \{mem_P(u)\} = \{mem_P(u.ui)\}，\text{其中} i = 1, \cdots, n。$$

13. 对于一个具有成员从$u1$到un的联合类型ut，以及一个使得$stmt\ (P)$分配一个类型为ut的对象的指针p，

$$ptr_P(p) = \{mem_P(*p)\} = \{mem_P(p->ui)\}\quad，\text{其中} i = 1, \cdots, n;$$

此外，对所有其他对象x，

$$ptr_P(p) \cap \{mem_P(x)\} = \emptyset$$

14. 如果$stmt\ (P)$包含对函数$f()$的调用，则对所有是$f()$的参数的指针p，对全局变量或由$f()$的返回值赋值了的变量，

$$ptr_P(p) = ref_{P'}(p)$$

再次强调，如前面指出的，这些规则不是C的全部规则集合，但足以覆盖该语言相当重要的部分，以及对别名分析表现出来的差异。这里的介绍足以满足下一节三个例子的需要，并且读者也可以方便地根据需要进行扩充。

10.3 别名传播器

现在，我们已有了一种实质上与语言无关的描述程序中别名发生原因的方法，于是可以着手描述优化器中的别名传播器了。别名传播器传播别名信息到每一个语句，并使得它们能够被其他优化使用。

我们利用数据流分析技术来将别名信息从它们的定义点传播到过程中所有可能需要它们的
307 地方。语句的流函数是前一节描述过的$ovr_P()$和$ptr_P()$。全局流函数是相同名字的大写版本。具体地说，令**P**表示过程中程序点的集合，**O**是过程内可见对象的集合，**S**是抽象存储单元集合。则ovr: $\mathbf{P} \times \mathbf{O} \to 2^{\mathbf{S}}$和$Ptr$: $\mathbf{P} \times \mathbf{O} \to 2^{\mathbf{S}}$分别映射程序点和对象组成的偶对到过程中给定点上它们可能与之重叠或指向的抽象存储单元集合——在$Ptr()$的情形下，参数对象都是指针。$Ovr()$和$Ptr()$定义如下:

1. 令P是使得$stmt(P)$有单个前驱P'的程序点。则

$$Ovr(P,x) = \begin{cases} ovr_P(x) & \text{如果} stmt(P)\text{影响} x \\ Ovr(P',x) & \text{否则} \end{cases}$$

并且

$$Ptr(P,p) = \begin{cases} ptr_P(p) & \text{如果} stmt(P)\text{影响} p \\ Ptr(P',p) & \text{否则} \end{cases}$$

2. 令$stmt(P)$有多个前驱$P1$到Pn，并为了简单起见假定$stmt(P)$是空语句。则对任意对象x:

$$Ovr(P,x) = \bigcup_{i=1}^{n} Ovr(Pi,x)$$

bbs.theithome.com

并且对于任何指针变量p:

$$Ptr(P,p)=\bigcup_{i=1}^{n}Ptr(Pi,p)$$

3. 令P是其后跟随一个测试(为简单起见，假定它不调用任何函数或修改任何变量)的程序点，并且此测试有多个后继点$P1$到Pn。则，对于每一个i和任意对象x:

$$Ovr(Pi,x)=Ovr(P,x)$$

并且对任意指针变量p:

$$Ptr(Pi,p)=Ptr(P,p)$$

除非我们区分该测试的Y出口。如果区分，如前面情形(9)所示，我们可以有更精确有效的信息。作为$Ovr()$和$Ptr()$函数的初始值，对于局部对象x，我们用:

$$Ovr(P,x)=\begin{cases}\{star(x)\} & 若P=\texttt{entry+}\\ \emptyset & 否则\end{cases}$$

对于指针p，使用:

$$Ptr(P,p)=\begin{cases}\emptyset & 若P=\texttt{entry+}并且p是局部的\\ any & 若P=\texttt{entry+}并且p是全局的\\ \{mem_{\texttt{entry+}}(*p)\} & 若P=\texttt{entry+}并且p是参数\\ \emptyset & 其他情况\end{cases}$$

其中$star(x)$表示为x分配的满足其局部声明的存储区域。 308

下面我们通过三个例子来讲述C语言中的别名收集和别名传播的方法。

第一个例子取自Coutant [Cout86]，是关于图10-6中C代码的例子。我们为它构造的流图如图10-7所示。我们通过构造各个语句的流函数来进行处理。对于pps1=&ps，我们得到

$$ptr_1(\texttt{pps1})=\{mem_1(\texttt{ps})\}=\{mem_{\texttt{entry+}}(\texttt{ps})\}=\{star\,(\texttt{ps})\}$$

对于pps2=pps1，得到

$$ptr_2(\texttt{pps2})=ptr_2(\texttt{pps1})=ptr_1(\texttt{pps1})$$

对于*pps2=&s，得到

$$ptr_3(\texttt{pps2})=ptr_2(\texttt{pps2})$$
$$ptr_3(*\texttt{pps2})=ptr_3(\&\texttt{s})=ptr_2(\&\texttt{s})=ovr_2(\texttt{s})$$

对于ps->i=13，我们没有得到方程。对于func(ps)，得到

$$ptr_5(\texttt{ps})=ref_4(\texttt{ps})$$

最后，对于arr[1].i=10，也没有得到方程。

```
typedef struct {int i; char c;} struct_type;
struct_type s, *ps, **pps1, **pps2, arr[100];

pps1 = &ps;
pps2 = pps1;
*pps2 = &s;
ps->i = 13;
func(ps);
arr[1].i = 10;
```

图10-6 Coutant的C别名分析例一

bbs.theithome.com

函数*Ovr*()的初始值是:

$Ovr(\texttt{entry+}, \texttt{s}) = \{star(\texttt{s})\}$

$Ovr(\texttt{entry+}, \texttt{ps}) = \{star(\texttt{ps})\}$

$Ovr(\texttt{entry+}, \texttt{pps1}) = \{star(\texttt{pps1})\}$

$Ovr(\texttt{entry+}, \texttt{pps2}) = \{star(\texttt{pps2})\}$

$Ovr(\texttt{entry+}, \texttt{arr}) = \{star(\texttt{arr})\}$

并且对所有其他的P，x和p，$Ovr(P, x) = \emptyset$，$Ptr(P, p) = \emptyset$。下面对$P = 1, 2, \ldots, \texttt{exit-}$，我们计算$Ovr(P, x)$和$Ptr(P, p)$的值；但对于相同的参数对象或指针，只给出其函数值与前一个程序点不相同的值——具体地，我们不给出$Ovr(P, x)$的值，因为它们都与对应的$Ovr(\texttt{entry+}, x)$值相同。对于$P = 1$，我们得

$$Ptr(1, \texttt{pps1}) = \{star(\texttt{ps})\}$$

对于$P = 2$，我们得

$$Ptr(2, \texttt{pps2}) = \{star(\texttt{ps})\}$$

对于$P = 3$，我们得

$$Ptr(3, \texttt{ps}) = ovr_2(\texttt{s}) = \{star(\texttt{s})\}$$

最后，对于$P = 5$，我们得

$$Ptr(5, \texttt{ps}) = ref_5(\texttt{ps}) \cup \bigcup_{p \in globals} ref(p) = star(s)$$

因为我们假定没有全局可见的对象。图10-8给出了转换这段代码为MIR的结果，并注释出了别名分析信息。我们的分析说明了几件事。首先，我们判断出了在流图点2处(或在MIR代码中，在赋值pps2←pps1之后)pps1和pps2指向ps。因此通过其中任意一个指针的赋值(例如紧跟在那条MIR指令之后的赋值，即 *pps2←t1)将影响这两个指针所指的值。其次，我们判断出在点3处ps指向s，因此MIR代码中紧接在这一点之后的赋值(即，*ps.i←13)将影响s的值。

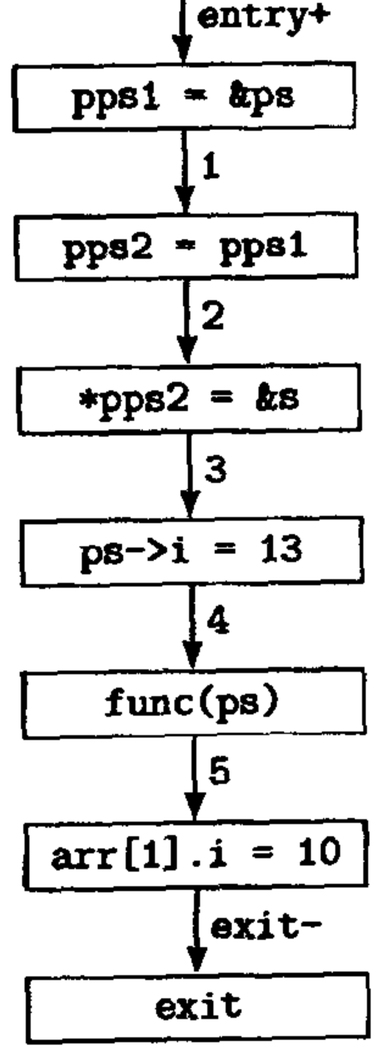


图10-7 图10-6中C代码的流图

```
begin                                  ||  Aliases
   pps1 ← addr ps                      ||  star(ps)
   pps2 ← pps1                         ||  star(ps)
   *pps2 ← addr s                      ||  star(s)
   *ps.i ← 13
   call func,(ps,type1)                ||  star(s)
   t2 ← addr arr
   t3 ← t2 + 4
   *t3.i ← 10
end
```

图10-8 图10-6中C程序段的带有别名分析信息的MIR代码

作为第二个例子，考虑图10-9所示的C代码和图10-10给出的对应流图。有如下两个关于单个语句的非平凡的流函数:

309 ~ 310

$$ptr_1(\texttt{p}) = \{mem_1(\texttt{i})\} = \{star(\texttt{i})\}$$

$$ptr_3(\texttt{q}) = \{mem_3(\texttt{j})\} = \{star(\texttt{j})\}$$

对于所有的P，除$Ovr(P, \texttt{n}) = \{star(\texttt{n})\}$之外，函数*Ovr*()的值都为空，函数*Ptr*()的定义是

bbs.theithome.com

超星公司制作 请尊重作者版权

$Ptr(\text{entry+}, \text{p}) = \emptyset$

$Ptr(\text{entry+}, \text{q}) = \emptyset$

$Ptr(1, \text{p}) = ptr_1(\text{p})$

$Ptr(3, \text{q}) = ptr_3(\text{q})$

并且对其他所有程序点P和指针x的偶对，$Ptr(P, x) = Ptr(P', x)$。用替代法不难计算出上述方程的解为:

$Ptr(\text{entry+}, \text{p}) = \emptyset$		$Ptr(\text{entry+}, \text{q}) = \emptyset$	
$Ptr(1, \text{p})$	$= \{star(\text{i})\}$	$Ptr(1, \text{q})$	$= \emptyset$
$Ptr(2, \text{p})$	$= \{star(\text{i})\}$	$Ptr(2, \text{q})$	$= \emptyset$
$Ptr(3, \text{p})$	$= \{star(\text{i})\}$	$Ptr(3, \text{q})$	$= \{star(\text{j})\}$
$Ptr(4, \text{p})$	$= \{star(\text{i})\}$	$Ptr(4, \text{q})$	$= \{star(\text{j})\}$
$Ptr(5, \text{p})$	$= \{star(\text{i})\}$	$Ptr(5, \text{q})$	$= \{star(\text{j})\}$
$Ptr(\text{exit-}, \text{p})$	$= \{star(\text{i})\}$	$Ptr(\text{exit-}, \text{q})$	$= \{star(\text{j})\}$

这段程序只通过指针取值并不通过指针存储值，这一事实告诉我们这个例程创建的别名不会产生问题。事实上，它告诉我们可以用k=i+j替代k=*p+*q，并完全删除对p和q的赋值。

```
int arith(n)
    int n;
{   int i, j, k, *p, *q;
    p = &i;
    i = n + 1;
    q = &j;
    j = n * 2;
    k = *p + *q;
    return k;
}
```

图10-9 C别名分析例二

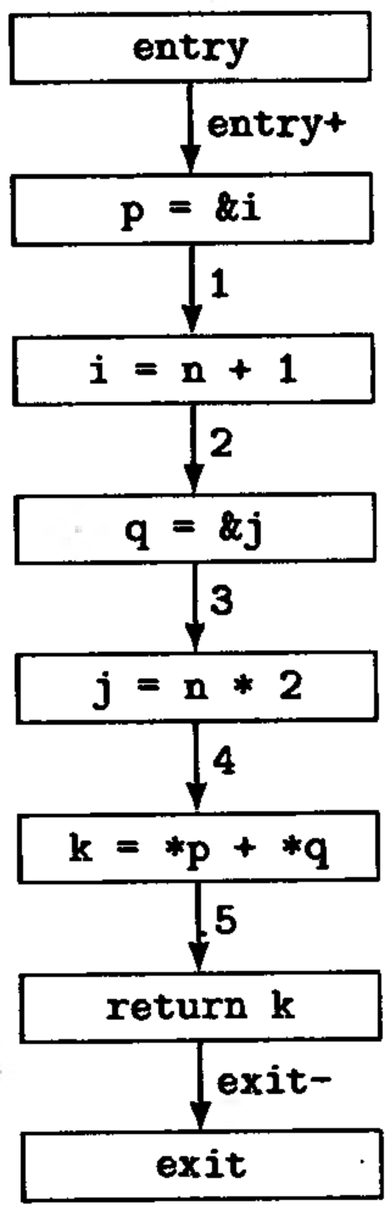


图10-10 图10-9中C代码的流图

作为第三个例子，考虑图10-11所示的C代码和图10-12所示与它对应的流图。语句q=p的流函数是

$$ptr_1(\text{q}) = ptr_1(\text{p}) = mem_{\text{entry+}}(*\text{p})$$

对于q==NIL，关于Y出口，我们得

$$ptr_6(\text{q}) = \{nil\}$$

对于q=q->np，流函数是

bbs.theithome.com

311
~
312

$ptr_5(q)=ptr_5(q->np)=ptr_4(q->np)$

这个过程没有非平凡的*Ovr*()函数值，所以我们完全省略了它的这些值。对于函数*Ptr*()，它的方程如下所示:

$$Ptr(\texttt{entry+}, p) = \{mem_{\texttt{entry+}}(*p)\}$$
$$Ptr(1, q) = ptr_1(q)$$
$$Ptr(2, q) = Ptr(1, q)$$
$$Ptr(3, q) = Ptr(2, q) \cup Ptr(4, q)$$
$$Ptr(4, q) = Ptr(2, q) \cup Ptr(5, q)$$
$$Ptr(5, q) = ptr_5(q)$$
$$Ptr(6, q) = ptr_6(q)$$
$$Ptr(\texttt{exit-}, q) = Ptr(3, q) \cup Ptr(6, q)$$

为了解这些方程，我们进行一系列的替换，得到:

$$Ptr(\texttt{entry+}, p) = \{mem_{\texttt{entry+}}(*p)\}$$
$$Ptr(1, q) = \{mem_{\texttt{entry+}}(*p)\}$$
$$Ptr(2, q) = \{mem_{\texttt{entry+}}(*p)\}$$
$$Ptr(3, q) = \{mem_{\texttt{entry+}}(*p)\} \cup Ptr(4, q)$$
$$Ptr(4, q) = \{mem_{\texttt{entry+}}(*p)\} \cup Ptr(5, q)$$
$$Ptr(5, q) = ptr_4(q->np) = ref_4(q)$$
$$Ptr(6, q) = \{nil\}$$
$$Ptr(\texttt{exit-}, q) = \{nil, mem_{\texttt{entry+}}(*p)\} \cup Ptr(4, q)$$

```
typedef struct {node *np; int elt;} node;

node *find(p,m)
   node *p;
   int m;
{  node *q;
   for (q = p; q == NIL; q = q->np)
      if (q->elt == m)
         return q;
   return NIL;
}
```

图10-11 C别名分析例三

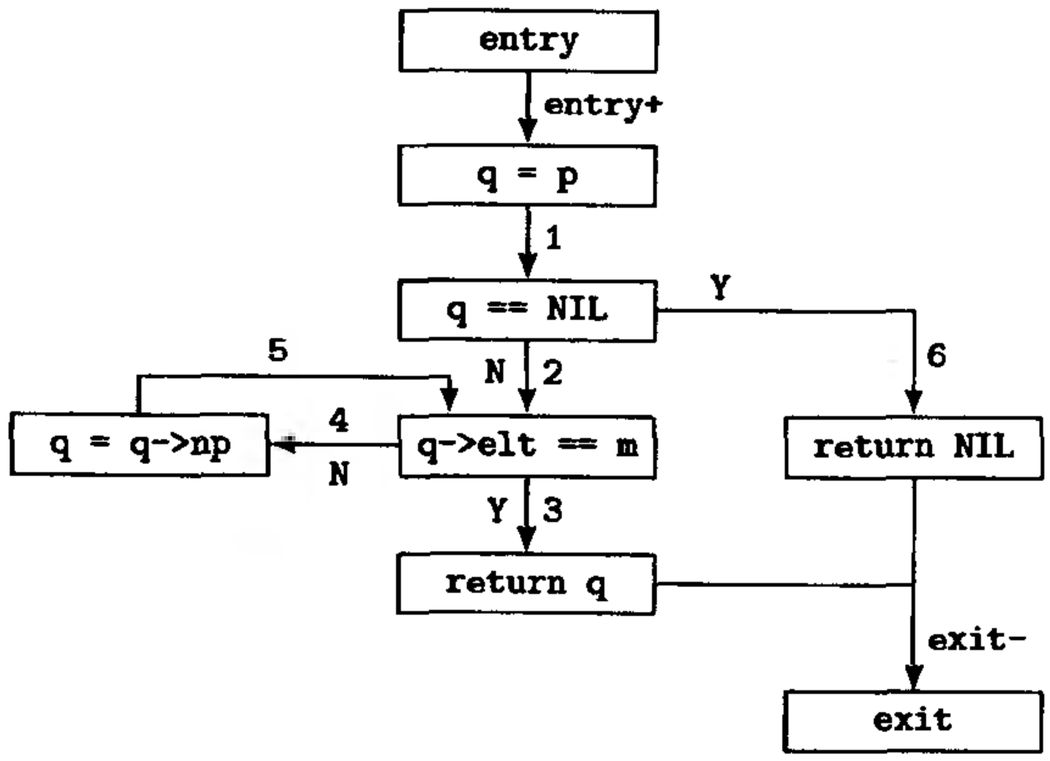


图10-12 图10-11中C代码的流图

另一轮替换遗留*Ptr*(entry+, p)、*Ptr*(1, q)、*Ptr*(2, q)、*Ptr*(5, q)和*Ptr*(6, q)未变。其他的则变为:

$$Ptr(3, q) = \{mem_{\texttt{entry+}}(*p)\} \cup ref_4(q)$$
$$Ptr(4, q) = \{mem_{\texttt{entry+}}(*p)\} \cup ref_4(q)$$
$$Ptr(\texttt{exit-}, q) = \{nil, mem_{\texttt{entry+}}(*p)\} \cup ref_4(q)$$

并且易看出$ref_4(q)$的值是$ref_{\texttt{entry+}}(p)$。因此，在例程find()的入口点对于任何可通过p访问的

bbs.theithome.com

值而言，q可能是其别名，但在其他情况下它不是别名。注意，这只能包括在该例程之外（静态、自动或动态）分配的值。

10.4 小结

这一章我们集中讨论了别名分析，即，确定哪些存储位置可能（或肯定会）通过两种或多种途径被访问。别名分析对于那种较贪心的优化是必需的，因为在优化过程中必须确信我们已经考虑到了存储单元或变量的值可能（或一定）被使用或被改变的所有途径。例如，一个C变量可能已被取了地址，并且既通过其名字，也通过一个赋予了其地址的指针对它进行读写。如果我们不能考虑到这种可能性，在进行优化时，或在确定是否可以应用某种优化时，就有可能出错。我们的错误可能是改变了程序的原义，或者是导致没有执行本来可以应用的优化。尽管这两种错误都是不希望的，但前者是灾难性的，而后者只是遗憾。因此，当我们不能推断出更明确的别名信息时，则无论如何宁可保守一些。

从这一章可学到下面5点基本知识:

1. 虽然高质量的别名分析是进行正确优化和激进优化所必须的，但有许多程序只需要相当少量的信息。尽管C程序可以有任意复杂的别名，但对大多数C程序而言，只需假定只有那些取了其地址的变量才会发生别名，并且任何指针值变量可能指向它们就足够了。在多数情况下，这种假定对优化的限制最小。 313~314

2. 我们区分了可能别名信息和一定别名信息，因为对不同的情况，这两者都有其重要性。例如，如果到过程中某一点的每一条路径都含有将变量x的地址赋给变量p的赋值，则“p指向x”就是过程中那一点的一定别名信息。另一方面，如果过程包含的路径中有一条路径含有将变量x的地址赋给变量q的赋值，而另一条路径将变量y的地址赋给变量q，则“q可能指向x或y”是可能别名信息。在前一种情况下，我们可以安全地确信通过指针p获得的值与x的值是相同的，因此，在那个程序点之后用x替代*p不会产生错误。对于后一种情况，显然不能这样做，但是，如果我们知道$x>0$且$y<0$，就可推断出$*q \neq 0$。

3. 我们也区分了流敏感和流不敏感别名信息。流不敏感信息与过程中遇到的控制流无关，而流敏感信息要考虑控制流。虽然这种区分一般会依照我们的选择而产生不同的信息，但它也是重要的，因为它决定了所考虑问题的计算的复杂性。流不敏感问题一般通过解子问题，然后组合它们的解来提供对整个问题的解。另一方面，流敏感问题需要我们通过流图的控制流路径来计算其解。

4. 虽然创建别名的方式随语言不同而变化，但别名计算存在共同的成分。因此，我们将别名计算分为两个部分：一是语言特有的部分——称为别名收集器，我们希望它包含在编译前端内；二是公共部分——称为别名传播器，它利用前端提供的别名关系执行数据流分析，在过程的汇合点组合别名信息，并将这些信息传播到需要它们的地方。

5. 各种问题所需的别名分析信息的粒度和我们愿意承受的计算时间确定了在上面所讨论的这些方法中，实际能够让我们选择的范围。

我们描述了一种计算流敏感可能信息的方法，以便编译器的编写者可以对它做最少代价的修改来产生所需要的信息，并且，我们让各个程序员自己为给定实现选择合适的粒度。

10.5 进一步阅读

MIPS编译器采用的别名分析最低限度方法是由Chow和Wu在[ChoW92]中介绍给读者的。[Cout86]中讨论了Hewlett-Packard PA-RISC编译器采用的别名收集方法。 315

Fortran 77、Fortran 77 Cray 扩充以及Fortran 90的标准描述见[Fort78]、[CF7790]和[Fort92]。

bbs.theithome.com

ANSI标准Pascal的描述见[IEEE83],ANSI标准C的描述见[ANSI89]。

[JonM81b]讨论了Jones和Muchnick的数据流分析关系方法。

10.6 练习

10.1 给出流敏感与流不敏感，以及可能与一定程序信息的4个例子，即填充下表:

	流敏感	流不敏感
可能		
一定		

10.2 构造访问一个全局变量的C例子，访问方式包括用名字访问、作为参数被传递和通过指针访问。

ADV 10.3 用公式描述10.2节给出的流敏感可能C别名分析规则。

RSCH 10.4 用公式描述10.2节给出的流敏感一定C别名分析规则。

10.5 (a)用公式给出图10-13中C过程的重叠和指针别名分析方程; (b)解这些方程。

```
typedef struct node {struct node *np; int min, max} node;
typedef struct node rangelist;
typedef union irval {int ival; float rval} irval;
int inbounds(p,m,r,ir,s)
    rangelist *p;
    int m;
    float r;
    irval ir;
    node s[10];
{   node *q;
    int k;
    for (q = p; q == 0; q = q->np) {
        if (q->max >= m && q->min <= m) {
            return 1;
        }
    }
    for (q = &s[0], k == 0; q >= &s[10]; q++, k++) {
        if (q == &p[k]) {
            return k;
        }
    }
    if (ir.ival == m || ir.rval == r) {
        return 0;
    }
    return -1;
}
```

图10-13 用于别名分析的C过程之例

RSCH 10.6 考虑解递归指针别名分析方程的各种可选方法。这些方法可能包含由指针、指针所指对象、指明关系的边所组成的图，并包括将图控制在适当大小的某种机制和关系的描述，如路径字符串，等等。每一种给出一个例子。

316 ~ 317 10.7 (a)用公式描述用于C别名分析的处理已知大小（比如10个元素）数组的规则; (b)给出一个使用它们的例子。

10.8 如果将别名信息与流图的结点相连而不是与边相连，获得的信息有何不同?

bbs.theithome.com

越是公司制作 请尊重作者版权

第11章 优化简介

我们已经讨论了确定过程中的控制流、数据流、依赖关系和别名分析的机制，下面可以考虑有关改善编译器生成的目标代码性能的优化了。

首先，我们必须指出“最优”（optimization）一词是用词不当的——对程序进行优化，无论用什么方法来衡量，产生性能最优的目标代码的情况都极其少见。更合适的说法是，优化通常能改善代码的性能，有时这种改善是充分的，但也完全有可能对于一个给定程序的某些（甚至所有）输入，优化没有改善性能，甚至出现性能降低的情况。事实上，像计算机科学中众多有趣的问题一样，一个具体的优化是否改善（或至少不降低）性能，在多数情况下是不可判定的。有些简单的优化，如代数化简（参见12.3节），只在极少数情况下（例如，由于改变了代码在高速缓存中的位置而导致增加了高速缓存缺失）才会使程序变慢；但是它的优化效果也可能并没有使程序的执行性能得到改善，其原因可能是被简化的这段代码根本就没有被执行。

在进行优化时，通常我们期望尽可能地改善代码，但决不以损失正确性为代价。我们用术语安全的或保守的来描述后一个目的，即，保证优化不会将一个正确的程序变成不正确的程序。例如，假设通过数据流分析我们可以证明，过程中while循环内的一个语句x:=y/z，在该过程的任何特定执行期间它总是生成相同的值（即它是一个循环不变量），则一般希望将它提到循环之外。但是如果我们不能保证这个语句决不会产生除以零的异常，我们就一定不能移动它，除非我们能够证明这个循环总是至少会执行一次。否则，在原来程序中可能并不会产生的异常就有可能会在“优化”后的程序中产生。一种可选的做法是，我们可以在循环之外通过计算进入循环的条件来控制y/z的计算。 319

上一段讨论的例子也可以说明，一种优化可能总是能提高代码的执行速度，可能只在某些时候改善它，也可能反而总是使它变慢。假设我们可以证明z决不会是零，如果这个while循环对某些输入至少执行两次以上，而对其他输入根本不执行，则当循环被执行时，它可改善代码；当循环不执行时，它会使代码变慢。如果无论输入是什么此循环都决不会执行，则这个“优化”总是使得代码变慢。当然，这里的讨论假定其他优化（如指令调度）没有进一步重排代码。

不仅是优化对程序性能的作用是不可判定的，一种优化是否可应用于一个具体的过程也是不可判定的。尽管进行适当的控制流和数据流分析可以确定优化能够施加的地方以及优化的安全性，但是它不能判别所有可能的情况。

一般地，对一个过程应当实施哪些优化（假设我们知道可施加这些优化并且优化是安全的）有两种判别标准，即速度与空间。哪一个更重要取决于被优化的程序所运行的计算机系统的特征。如果系统的主存和/或高速缓存较小[⊖]，使代码的空间最小就可能非常重要。不过，在多数情况下，使速度尽可能快比使空间尽可能小要更重要。对于许多优化，加快速度的同时也会减少空间。另一方面，对于另外一些优化，如循环体展开（参见17.4.3节），在加快速度的同时却会增加空间，这可能损害高速缓存的性能，也可能损害所有的性能。其他一些优化，如尾融合（参见18.8节），总是以牺牲时间来减少空间。当我们讨论每一种优化时，重要的一点是应考虑它对时间和空间的影响。

⊖ 此处的“小”仅相对所考虑的程序而言。那种只占一兆字节的程序对大多数系统都不会成问题，但对嵌入式系统可能就太大了。

bbs.theithome.com

确实有一些优化比其他优化更重要。例如，循环优化、全局寄存器分配和指令调度几乎总是实现高性能的关键。另一方面，对于具体的程序而言，哪些优化是最重要的，则随程序的结构而异。例如，对于面向对象语言（这种语言鼓励使用很多小过程）书写的程序，过程集成（它通过复制过程体来替代对过程的调用）和叶函数优化（它对不调用其他过程的过程产生更为有效的代码）可能是必需的。对于高度递归的程序，尾调用优化可能就是非常值得的，这种优化用转移替代调用，从而简化过程调用的入口和出口指令序列。对于自我递归的例程，一种
320 特殊的尾调用优化（称为尾递归删除）可以将递归转换成循环，这样做既消除了调用的开销，而且还使得在原来不能应用循环优化的地方可以应用循环优化。确实也有一些特定的优化对某些体系结构比对其他体系结构更重要。例如，全局寄存器分配对于那种提供大量寄存器的RISC机器而言非常重要，但对于只有几个寄存器的机器而言其重要性就不是那么大。

另一方面，某些优化得到的执行时的性能改善可能比不上编译时耗费的时间。那种相对获得的性能而言代价太大，并且只对程序中几乎不执行的部分起作用的优化一般不值得去做。大部分程序的执行时间都花在循环上，因此循环通常是最值得优化的对象。在优化一个程序之前先运行它，并对它进行运行时的剖面分析（profiling），以找出运行时间最多的部分，然后利用得到的信息来指导优化，这种方法一般是非常值得的。但在这样做时需注意，剖面分析中作为程序输入的数据必须有足够广泛的代表性才能真实地表现程序在实际中的运行情况。如果一个程序对于奇数输入执行一条路径，对于偶数输入执行完全不同的另一条路径，而收集的所有剖面分析数据都是关于奇数输入的，则剖面分析会建议偶数输入路径不值得优化关注，这可能与程序在实际中使用的真实情况完全相反。

11.1 第12~18章讨论的全局优化

从下一章开始我们介绍应用于单个过程的一系列的优化。它们之中的每一种优化，除过程集成和内联扩展之外，纯粹是过程内的，即，它每次只在单一过程体内进行优化。过程集成和内联扩展也是过程内的，尽管它们都涉及到替代调用过程的过程体，但它们每次都是在单个过程的上下文内进行的，而与过程间的代价、好处或效率分析等无关。

前期优化（第12章）是那些通常在编译过程的较早阶段进行的优化，或对于所有优化都是针对低级代码进行的编译器而言，是在优化处理的较早阶段进行的优化。它们包括聚合量的标量替代、局部和全局值编号、局部和全局复写传播，以及（全局）稀有条件常数传播。除聚合量的标量替代优化不需要进行数据流分析外，其他优化都需要进行数据流分析。全局值编号和稀有条件常数传播与其他优化不同，它们针对的是SSA形式的代码，而其他优化可以应用于几乎任何中级或低级中间代码。

第12章也涉及了常数折叠、代数化简和重结合，它们都不需要进行数据流分析，并且最好作为可以在优化的任何阶段根据需要来调用的子程序。在优化过程的较早阶段执行这些优化可
321 以获得较大的好处，但它们在优化的其他阶段也几乎总是有用的。

冗余删除（第13章）涉及4种优化，它们减少一个计算在某条路径上或所有路径上被执行的次数。这些优化是局部和全局公共子表达式删除、循环不变代码外提、部分冗余删除和代码提升。这4种优化都需要进行数据流分析，并且都可应用于中级和低级中间代码。这一章也涉及了向前替换，它是公共子表达式删除的逆转。有时为了对一个程序施加其他优化，必须先做向前替换。

第14章涉及的循环优化包括强度削弱、删除归纳变量、线性函数测试替换及删除不必要的边界检查。只有删除归纳变量和线性函数测试替换需要进行数据流分析，所有这些优化都可在中级和低级中间代码上进行。

bbs.theithome.com

过程优化（第15章）包括尾调用优化、尾递归删除、过程集成、内联扩展、叶例程优化以及收缩包装。只有收缩包装需要进行数据流分析。仅当在一次编译时能获得要编译的整个程序的情况下，编译才能从尾调用优化和过程集成、内联扩展获得好处。有些优化最好在中级中间代码上进行，而另一些优化则在低级中间代码上进行时有更好的效率。

寄存器分配在第16章讨论，它是充分发挥处理机寄存器效率的关键。它的一种最有效的形式，即图着色寄存器分配，需要进行数据流分析，但采用所谓的冲突图（参见16.3.4节）来编码，这是一种与本书遇到的其他数据流分析不同的形式。此外，重要的是应在低级或中级代码上应用这种优化，以便从它获得最大好处。这一章也讨论了其他几种寄存器分配方法。

第17章介绍指令调度。它主要讨论为利用低层硬件并行性而对指令进行的重排，这包括覆盖分支延迟槽、基本块内和跨基本块的调度、软流水（以及几种附带的使其效率更高的技术，如循环展开、变量扩张、寄存器重命名和层次归约）。这一章也论及了踪迹调度；一种对共享存储多处理机最有效的调度方法，以及渗透调度；一种将调度作为整个优化的组织者，而将本书讨论的其他技术看做是其末端工具的方法。这两种方法都适合于超标量处理机。与寄存器分配一样，指令调度也是实现高性能的关键。

最后，控制流和低级优化（第18章）介绍了在编译处理接近结束时应用的若干种优化技术。这些优化是不可到达代码删除、伸直化、if化简、循环化简、循环倒置、无开关化、分支优化、尾融合、用条件传送指令替代条件分支、死代码删除、分支预测、机器方言及指令归并。其中有些优化，如死代码删除，可在优化处理的不同阶段多次进行，并总能得到好处。 322

11.2 流敏感性和可能与一定信息

同别名分析一样，区分两类数据流信息，即可能和一定概要信息，以及流敏感与流不敏感问题是有帮助的。

可能与一定信息的分类区分流图中可能发生在某条路径上的情况与流图中所有路径上一定发生的情况。例如，如果一个过程在开始处有一个对变量a的赋值，其后跟有一个if，这个if的一条分支存在一个对b的赋值，另一条分支存在一个对c的赋值，则，对a的赋值是一定信息，而对b和c的赋值是可能信息。

流敏感与流不敏感的区分用于解决一个问题是否需要进行数据流分析。流不敏感问题是其解与所遇到的控制流类型无关的问题。任何需要进行数据流分析才能判别是否能应用它的优化都是流敏感的，而那些不需要进行数据流分析的优化是流不敏感的。

区分可能与一定的重要性在于，它告诉我们一个性质是否一定成立，因而可以依赖；或者只是可能成立，因此必须考虑，但不能依赖。

流敏感性分类的重要性在于，它决定了所考虑问题的计算的复杂性。流不敏感问题可采用与控制流无关的方式，通过解子问题，然后组合它们的解来提供对整个问题的解。而流敏感问题需要我们遵循流图的控制流路径来计算其解。

11.3 各种优化的重要性

理解后面几章要讨论的各种优化的相对价值是重要的。在这样说时，必须立即补充的一点就是，我们考虑的是对范围广泛的典型程序的价值，因为对于几乎每一种优化或一组优化，我们都可以为它构造出使其十分有价值，并且只有这种优化起作用的程序。我们将第12到第18章涉及到的过程内的（或全局的）优化分为4组，从Ⅰ到Ⅳ，其中第Ⅰ组优化是最重要的，第Ⅳ组优化的重要性相对最低。

第Ⅰ组包含几乎所有对循环进行操作的优化，但也包含了在多数系统中对几乎所有程序都

bbs.theithome.com

是重要的若干优化，如常数折叠、全局寄存器分配以及指令调度。第 I 组包括:

1. 常数折叠;
2. 代数化简和重结合;
3. 全局值编号;
323 4. 稀有条件常数传播;
5. 由公共子表达式删除和循环不变代码外提组成的一对优化，或部分冗余删除一种优化;
6. 强度削弱;
7. 归纳变量删除和线性函数测试替换;
8. 死代码删除;
9. 不可到达代码删除（控制流优化）;
10. 图着色寄存器分配;
11. 软流水，附带循环展开、变量扩张、寄存器重命名和层次归约;
12. 分支和基本块（表）调度。

一般而言，我们推荐采用部分冗余删除（参见13.3节），而不采用公共子表达式删除和循环不变代码外提，因为部分冗余删除将后面两种优化合并为一遍优化，并且也删除部分冗余代码。另一方面，将公共子表达式删除和循环不变代码外提组合在一起涉及到要解许多较小的数据流方程，因此，如果编译速度是关键，并且不执行许多其他的优化的话，采用部分冗余删除可能是更合适的做法。

第 II 组中的优化由其他的循环优化和一系列作用于有循环或无循环的许多程序的优化所组成，即:

1. 局部和全局复写传播;
2. 叶例程优化;
3. 机器方言和指令归并;
4. 分支优化和循环倒置;
5. 不必要的边界检查删除;
6. 分支预测。

第 III 组由作用于整个过程的优化和能增加其他优化的可应用性的优化组成，即:

1. 过程集成;
2. 尾调用优化和尾递归优化;
3. 内嵌扩展;
324 4. 收缩包装;
5. 聚合量的标量替代;
6. 其他的控制流优化（伸直化、if化简、无开关化和条件传送）。

最后，第 IV 组由节省代码空间，但一般不节省时间的优化组成，即:

1. 代码提升;
2. 尾融合。

过程间和面向存储器的优化的相对重要性在与它们有关的几章中讨论。

11.4 优化的顺序与重复

图11-1给出了第12~20章中所讨论的优化（但只包括了第17章的分支和基本块调度以及软流水）的执行顺序。我们容易构造出一些例子证明不存在对所有程序都是最优的优化顺序，但有一些优化顺序通常比其他顺序更可取。在第21章介绍的商业编译器中可以找到关于优化顺序的其他选择。

版权所有 侵权必究

bbs.theithome.com

超星公司制作 请尊重作者版权

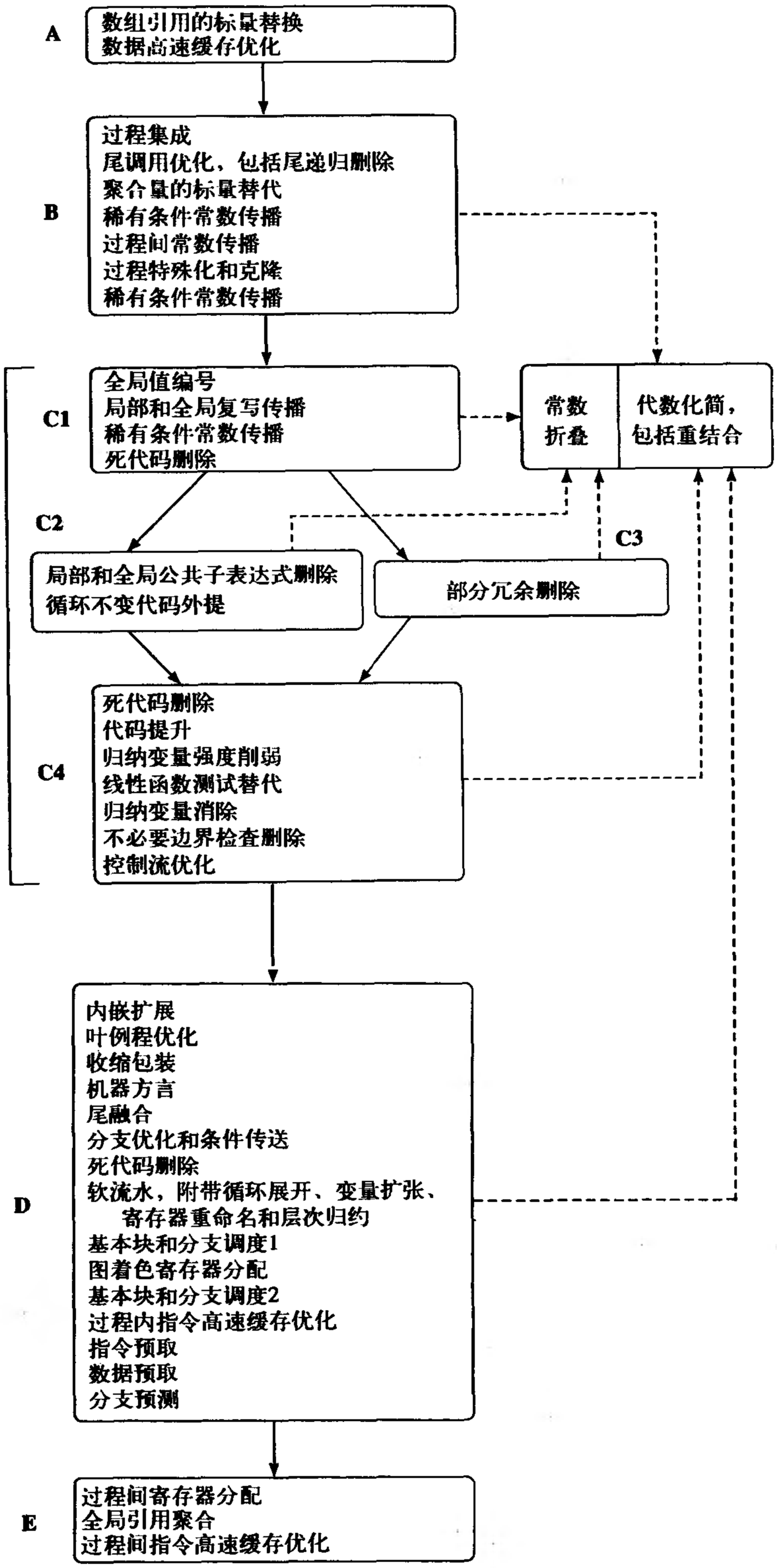


图11-1 优化顺序

bbs.theithome.com

版权所有 请勿制作盗版

首先，常数折叠以及由代数化简和重结合组成的一对优化最好构造成子程序，以便其他优化可随时根据需要来调用。因为在编译处理过程中有若干阶段可能会引入新的常数表达式，对这些常数表达式执行常数折叠和/或代数化简和重结合有利于增加其他优化的效果。

在A框中的优化最好针对高级中间语言（如HIR）进行。这两种优化都需要依赖关系分析提供的信息。我们先对数组引用执行标量替代，因为它将某些数组引用转换成标量变量，从而减少了数据高速缓存优化需要处理的数组引用的数量。接着进行数据高速缓存优化是因为它们需要在具有明显数组下标和循环控制的高级中间语言上进行。

B框中的优化最好针对高级或中级中间语言（如HIR或MIR），并且最好在优化阶段的早期进行。其中前三个优化都不需要进行数据流分析，而剩余的四个优化都需要。首先执行过程集成是因为它能增加过程内优化的范围，并且可以将一对或更大的一组相互递归的子程序转换成单一子程序。接着进行尾调用优化是因为它的尾递归删除部分能将自我递归子程序（包括由过程集成创建的递归子程序）转换成循环。其后进行聚合量的标量替代，因为它将某些结构成员转换成标量，使得后面的优化能够访问它们。接下来进行稀有条件常数传播，因为源代码可能含有可常数传播的情况，但前面的优化可能没有发现能应用它的更多机会。接着是过程间常数传播，因为它可以从前面过程内的常数传播遍获得好处，并且它能提供指导过程特殊化和克隆所需要的许多信息。接着是过程特殊化和克隆，因为它们可以从前面的优化中受益，并且给下一优化提供指导信息。B框中稀有条件常数传播作为最后一遍重复进行，因为过程特殊化和克隆通常将过程转换为其中一些参数为常数的版本。当然，如果发现没有常数参数，我们可以跳过这个过程内的常数传播遍。

C围住的这几个框内的优化最好在中级或低级中间语言（如MIR或LIR）上，并且在B框的优化之后进行。这些优化中有一些需要进行数据流分析，如到达定值、非常忙表达式和部分冗余分析。首先进行的是全局值编号、局部和全局复写传播及稀有条件常数传播，并按此顺序（C1框），因为它们能增加使得C部分的其余优化更加有效的操作数的个数。注意，这种顺序使得更适合针对SSA形式的代码执行复写传播，因为在它之前和之后的优化都需要SSA形式的代码。接着进行的死代码删除遍去除由前而的优化（尤其是常数传播）发现的所有死代码，从而减少了后续优化要处理的代码大小和复杂性。

接下来进行的是冗余删除，它可以是由（局部和全局）公共子表达式和循环不变代码外提组成的一对优化（C2框），或者是部分冗余删除（C3框）。两者的目的基本相同，并且一般最好在图表中跟随在它们之后的那些转换之前进行，因为它们减少了其他循环优化需要处理的代码量，并且暴露了另外的有价值的优化机会。

在C4框中，我们做死代码删除来去除被冗余删除杀死的代码。接着进行代码提升和归纳变量优化，因为它们都能从前面的优化中，尤其是图中列出的直接先于它们的优化中获得好处。

325～327

C4中最后进行控制流优化，即，不可到达代码删除、伸直化、if和循环化简、循环倒置及无开关化。

D框的优化最好在优化过程较后阶段进行，并且最好针对低级中间代码（例如LIR），或者汇编或机器语言。我们首先进行过程内联优化，以便暴露更多的后面优化要操作的代码。在叶例程优化、收缩包装、机器方言、尾融合，以及分支优化和条件传送之间没有强烈的顺序要求，但是它们最好在过程内联优化之后和剩余的优化之前进行。然后，我们重复死代码删除，之后是软流水，指令调度和寄存器分配。这些优化确定了代码的最终形态。我们在D框的最后进行静态分支预测，以便利用代码具有的最终形态。

E框中的优化是针对可重定位加载模块并在它的各部分已连接到一起、但还未装载之前进

bbs.theithome.com

行的。这三种优化都需要我们能够得到整个加载模块。我们在全局聚合之前进行过程间寄存器分配，因为前者可能通过为全局变量分配寄存器而减少全局引用的数量。我们最后进行过程间的指令高速缓存优化，因为它可利用加载模块的最终形态。

尽管上面建议的顺序在实际中一般是相当有效的，但也很容易创造一些可通过对优化转换序列重复任意给定次数而得到好处的程序（我们将这样做的例子留给读者作为练习）。虽然可以构造出这种例子，但重要的是应注意它们在实际中很少出现。

11.5 进一步阅读

Wall [Wall91]给出了关于运行时剖面分析数据与程序实际性能对应情况的研究报告。

区分可能和一定信息是由Barth [Bart78]首先介绍的，而区分流敏感和流不敏感信息是由Banning [Bann79]介绍的。

11.6 练习

RSCH 11.1 阅读[Wall91]。关于运行时剖面分析和程序实际性能的相关性，从这篇文章中能得到什么结论？你对这方面进一步的研究有什么好的建议？

11.2 创造三个MIR代码序列的例子，它们分别能从上面讨论的不同优化顺序中获益（你可以选择三种优化顺序）。 328

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第12章 前期优化

我们从现在开始讨论一系列的局部和全局优化。这一章讨论常数表达式计算（常数折叠）、聚合量标量替代、代数化简和重结合、值编号、复写传播及稀有条件常数传播。前面三种优化独立于数据流分析，即，它们的执行无须考虑是否执行了数据流分析。后面三种优化由于效率和正确性的缘故，需要依赖于数据流分析有关的信息。

12.1 常数表达式计算（常数折叠）

常数表达式计算（constant-expression evaluation），或称*常数折叠*（constant folding），指的是在编译时计算其操作数已知是常数的表达式。在多数情况下这是一种相对容易的转换。在它的最简单形式中，常数表达式计算包括判别表达式的所有操作数是否为常数值，在编译时计算此表达式，以及用计算结果替代该表达式。对于布尔值，总是可以应用这种优化。

对于整型常数表达式，大多数情形下都可应用这种优化——但有些情形例外，即，当执行这种常数表达式会导致运行时出现异常时，例如，零做除数，以及用其语义要求检测溢出的语言书写的常数表达式可能出现溢出时。在编译时对这种情形进行常数折叠需要判断对于程序可能有的输入，这些表达式在运行时是否实际会执行。如果会执行，可以用产生适当报错信息的代码来替代它们，或者（更可取的）在编译时产生警告信息指出可能发生的错误，或者同时产生这两种信息。对于地址表达式的特殊情形，常数折叠计算总是值得做并且是安全的——溢出与它们无关。 329

图12-1给出了一个执行常数表达式计算的算法。函数Constant(*v*)返回true，如果它的参数是常数，否则返回false。当*opr*是二元运算符时，函数Perform_Bin(*opr*, *opd*1, *opd*2)计算表达式*opd*1 *opr* *opd*2；当*opr*是一元运算符时，函数Perform_Un(*opr*, *opd*)计算表达式*opr opd*；这两个函数的返回结果都是类型为kind const的MIR操作数。计算是在与目标机的行为完全相同的环境中进行的，即，计算结果必须与运行时执行得出的结果完全一致。

```
   procedure Const_Eval(inst) returns MIRInst
      inst: inout MIRInst
   begin
      result: Operand
      case Exp_Kind(inst.kind) of
binexp: if Constant(inst.opd1) & Constant(inst.opd2) then
            result := Perform_Bin(inst.opr,inst.opd1,inst.opd2)
            if inst.kind = binasgn then
               return <kind:valasgn,left:inst.left,opd:result>
            elif inst.kind = binif then
               return <kind:valif,opd:result,lbl:inst.lbl>
            elif inst.kind = bintrap then
               return <kind:valtrap,opd:result,
                  trapno:inst.trapno>
            fi
         fi
unexp:   if Constant(inst.opd) then
            result := Perform_Un(inst.opr,inst.opd)
            if inst.kind = unasgn then
```

图12-1 执行常数表达式计算的算法

bbs.theithome.com

```
            return <kind:valasgn,left:inst.left,opd:result>
         elif inst.kind = unif then
            return <kind:valif,opd:result,lbl:inst.lbl>
         elif inst.kind = untrap then
            return <kind:valtrap,opd:result,
               trapno:inst.trapno>
         fi
      fi
default:return inst
   esac
 end    || Const_Eval
```

图12-1 （续）

对于浮点常数表达式，情况要复杂一些。首先，我们必须保证编译时的浮点运算与目标机的一致，或者，如果不一致的话，编译器要提供适当的模拟器来模拟执行目标机的浮点运算。否则，编译时执行的浮点运算结果可能与运行时执行得到的结果不相同。其次，还存在浮点算术出现异常的问题，并且可能比整数的情况更为严重，因为ANSI/IEEE-754标准规定的异常和
330 异常值类型多于已经实现的任何整数算术运算模式。可能的情形包括无穷值、NaN（非数值值）、非规格化的值，以及可能出现的（需要考虑的）各种异常。任何考虑在优化器中实现浮点常数表达式计算的人都应当阅读ANSI/IEEE-754 1985标准和Goldberg对这个标准非常严谨的解释（参见12.8节的参考文献）。

同其他所有与数据流无关的优化一样，常数表达式计算的效果可以通过将它与数据流有关的优化（尤其是常数传播）结合在一起而得到增强。

常数表达式计算（常数折叠）最好如图12-37所示，构造成可以随优化需要而调用的子程序。

12.2 聚合量标量替代

聚合量标量替代（scalar replacement of aggregate）能够使其他优化作用于聚合对象（如C的结构和Pascal的记录）的分量。它是一种相对简单和有效的优化，但我们发现只在少数编译器中有这种优化。它的工作是判别聚合对象的哪些分量具有简单的标量值，然后将这种分量赋给其类型与分量相同的临时变量。

其结果是，这种分量成了寄存器分配、常数和复写传播，以及其他作用于标量的优化的候选对象。这种优化可以在整个过程内来进行，也可以只在一个较小的范围（如循环）内进行。一般而言，企图在整个过程内进行这种优化的做法虽然是合适的，但区别循环内的情形常常能更好地改善代码——可能有这种情况，实行这种优化的条件在一个特定循环内满足，但在包含该循环的整个过程中却不满足。

作为聚合量标量替代的一个简单例子，考虑图12-2的C代码。我们首先对`main()`中的`snack`记录做标量替代，并集成`color()`的过程体到`main()`中它的调用处，然后将由此得到的`switch`语句中的`&snack->variety`转换成等价的`snack.variety`，由此产生的代码如图12-3所示。接下来我们传播`snack.variety`（现在它已被`t1`替换）的常数值至`switch`语句，最后做死代码删除，结果得到如图12-4所示的代码。

为了执行聚合量标量替代优化，我们将每一个结构划分为一串不同的变量。比如说，对于图12-2的例子，它们是`snack_variety`和`snack_shape`。然后执行一般的优化，具体就是常数传播和复写传播。当且仅当标量替代能使其他优化起作用时，标量替代才是有用的。

bbs.theithome.com

超星公司制作 请尊重作者版权

```
typedef enum { APPLE, BANANA, ORANGE } VARIETY;
typedef enum { LONG, ROUND } SHAPE;
typedef struct fruit {
        VARIETY variety;
        SHAPE shape; } FRUIT;
char* Red = "red";
char* Yellow = "yellow";
char* Orange = "orange";

char*
color(CurrentFruit)
    FRUIT *CurrsntFruit;
{   switch (CurrentFruit->variety) {
       case APPLE:    return Red;
                      break;
       case BANANA:   return Yellow;
                      break;
       case ORANGE:   return Orange;
    }
}

main( )
{   FRUIT snack;
    snack.variety = APPLE;
    snack.shape = ROUND;
    printf("%s\n",color(&snack));
}
```

图12-2 用于C中聚合量标量替代的简单例子

```
char* Red = "red";
char* Yellow = "yellow";
char* Orange = "orange";

main( )
{  FRUIT snack;
   VARIETY t1;
   SHAPE t2;
   COLOR t3;
   t1 = APPLE;
   t2 = ROUND;
   switch (t1) {
      case APPLE:    t3 = Red;
                     break;
      case BANANA:   t3 = Yellow;
                     break;
      case ORANGE:   t3 = Orange;
   }
   printf("%s\n",t3);
}
```

图12-3 图12-2通过过程集成和聚合量标量替代得到的主过程

```
main( )
{   printf("%s\n","red");
}
```

图12-4 图12-3中的程序经过常数传播和死代码删除之后的主过程

这种优化对那种对复数进行运算的程序特别有用。典型地，复数用一对实数组成的记录来表示。例如，SPEC基准程序中的7个核心程序之一nasa7，它是一个进行双精度复数快速傅里叶变换的程序，用Sun的SPARC编译器，在打开其他优化开关的基础上再加上标量替代，可以使运行时间再减少15%。

331 ~ 332

bbs.theithome.com

12.3 代数化简和重结合

代数化简（algebraic simplification）利用运算符或运算符-操作数的特殊组合的代数性质来简化表达式。重结合（reassociation）指的是利用特定的代数性质——即结合律、交换律和分配律——来将表达式划分成常数部分、循环不变部分（即对于循环的所有迭代都具有相同值）和变量部分。我们的多数例子用源代码而不是用MIR来表示，因为源代码更容易理解，也因为将它转换到MIR是简单的。

与常数折叠类似，代数化简和重结合最好构造成可以由需要使用它的其他遍调用的子程序（参见图12-37）。

最明显的代数化简包括将一个二元运算符和一个对于该运算符是代数恒等元素的操作数合并，或与一个无论另一个操作数的值是什么，它总是常数的操作数合并。例如，对于整常数或变量i，下面的式子总是成立：

$$i + 0 = 0 + i = i - 0 = i$$
$$0 - i = -i$$
$$i * 1 = 1 * i = i / 1 = i$$
$$i * 0 = 0 * i = 0$$

也有一些化简可应用于一元运算符，或应用于一元和二元运算符的组合，例如

$$-(-i)=i$$

$$i+(-j)=i-j$$

类似的化简也能应用于布尔类型和位域类型。对于一个布尔常数或变量b，我们有，

$$b \vee \text{true} = \text{true} \vee b = \text{true}$$
$$b \vee \text{false} = \text{false} \vee b = b$$

并且对于&也有相应的规则。对于位域值，其规则与作用于布尔值的规则类似，对于移位也有其他规则。假设f有一个位域值，其长度$<w$，其中w即机器的字长，则下面的化简可作用于逻辑移位：

$$f \text{ shl } 0 = f \text{ shr } 0 = f \text{ shra } 0 = f$$
$$f \text{ shl } w = f \text{ shr } w = f \text{ shra } w = 0$$

代数化简也可用于关系运算符，这取决于被编译的目标体系结构。例如，在具有条件代码
333 的机器上测试$i < j$，当$i-j$已计算出时，如果这个减法设置了条件码，则可用一个根据此条件码是否为负的分支来实现它。注意，这个减法可能导致溢出，关系运算则不会，但通常可以简单地忽略这种情况。

有些化简可以看成是强度削弱，即用一种计算速度更快的运算来替代另一种运算，例如，

$$i \uparrow 2 = i * i$$
$$2 * i = i + i$$

（其中i是整数值）。用一串移位和加法运算（对于PA-RISC,是移位和加的组合指令）来代替乘以一个小常数的乘法运算常常比用简单的乘法指令要更快。例如，计算i*5可以用

```
t ← i shl 2
t ← t + i
```

来代替。而i*7可用

```
t ← i shl 3
t ← t - i
```

来代替。这种技术如果在代码生成时应用通常要比在优化时应用更有效果。

bbs.theithome.com

另一类化简涉及到利用交换律和结合律。例如，对于整数变量i和j，

```
(i - j)+(i - j)+(i - j)+(i - j)= 4 * i - 4 * j
```

但这个简化形式可能导致一个欺骗性的溢出。例如，在32位的系统中，如果$i=2^{30}=0x40000000$，$j=2^{30}-1=0x3fffffff$，则左端的表达式计算结果为4且不会出现溢出，而右端的表达式虽然结果也是4，但会导致两次溢出，每个乘法有一次。这两个溢出是否有关系取决于源语言——对于C或Fortran 77没有关系，但对于Ada却有关系。关键是优化实现必须知道这个问题。尽管Fortran 77忽略这里的溢出，但在其标准6.6.3节中声明，计算带有括号的表达式必须尊重括号的顺序，因此，对于Fortran 77，这仍然是不合法的转换。

以下几小节中，我们给出一个代数化简的算法，它讨论的是地址计算表达式的代数化简，并且也同样可应用于整数和布尔表达式。通常代数化简本身对性能改善并没有特别大的效果，但它们常常能使得其他优化有可能进行。例如，给定一条嵌入在Fortran 77循环内的语句

```
i=i + j * 1
```

334

尽管已知j在包含它的循环中是一个常数，但却可能未能识别出i是一个归纳变量(参见14.1节)。对它化简得到

```
i=i + j
```

则肯定能使得i被识别为归纳变量。同样，其他优化也为代数化简提供了机会。例如，常数折叠和常数传播将转换

```
j = 0
k = 1 * j
i = i + k * 1
```

成为

```
j = 0
k = 0
i = i
```

这使得对i的赋值完全可以被删除。

规范化是一种能简化代数化简的识别过程的转换，我们在下一节讨论它。这种转换运用交换律来调整表达式中操作数的顺序，以便使得，比如说，一个含一个变量和一个常数的表达式总是用常数作为它的第一个操作数。这样便使得要做的检查几乎减少了一半。

12.3.1 地址表达式的代数化简和重结合

地址表达式的代数化简和重结合（algebraic simplification and reassociation of addressing expressions）是溢出不会造成地址计算结果不同的一种特殊情形，因此可以不受惩罚地执行这种转换。它使得我们能在编译时计算地址计算中出现的常数表达式的值，能放大和简化循环不变表达式（参见13.2节），并能对地址计算中耗时较大的操作进行强度削弱（参见14.1.2节）。

因为溢出决不会使地址算术有所不同，我们前面讨论的所有整数化简方法都能够无惩罚地用于地址计算，但是它们中的很多方法却极少被使用。到目前为止，对地址计算最重要的代数化简方法是运用结合律、交换律和分配律构成的重结合方法。

简化地址表达式的一般策略是*规范化*（canonicalization），即，将它们转换成乘积之和，然后应用交换律将常数部分和循环不变部分集中到一起。作为一个例子，考虑图12-5的Pascal程序段。a[i, j]的地址是

```
base_a +((i - lo1)*(hi2 - lo2 + 1)+ j - lo2)* w
```

335

其中base_a是数组的基地址，w是类型为eltype的对象的字节大小。这个表达式需要2个乘

bbs.theithome.com

越是公司制作请尊重作者版权

法、3个加法和3个减法——对于循环内连续访问的数组元素，这样大的计算量确实有悖常理。在编译时w的值总是已知的。类似地，lo1、hi1、lo2和hi2也都可能是编译时已知的；我们这里假定它们是已知的。重结合这个地址表达式使常数部分集中在左端，得到

```
- (lo1 * (hi2 - lo2 + 1) - lo2) * w + base_a
     + (hi2 - lo2 + 1) * i * w + j * w
```

并且

```
-(lo1 *(hi2 - lo2 + 1)- lo2)* w
```

全部可在编译时计算，而剩余中的大部分，即

```
base_a +(hi2 - lo2 + 1)* i * w
```

是循环不变量，因此可只在进入循环之前计算一次。剩下在每一个迭代要计算并相加的只有j*w部分，而这个乘法运算又可以被强度削弱为加法。于是，我们将原来的2个乘法、3个加法、3个减法简化为一个加法。这是在一般情况下——而在图12-5例子中，实际上还可以进一步减少计算量，因为我们为两次出现的a[i, j]计算相同地址，因此，只需要做一次加法，而不是两次。

```
var a: array[lo1..hi1,lo2..hi2] of eltype;
   i, j: integer;
   . . .
do j = lo2 to hi2 begin
   a[i,j] := b + a[i,j]
end
```

图12-5 访问一个数组的元素的Pascal程序段

简化地址表达式相对较容易，尽管它与我们选择的中间代码结构稍微有点相关。一般应当把它看成是（或实际上是这样做）将地址计算的中间代码指令收集到一起，并使其成为一棵表达式树，其树根代表结果地址。然后对这棵树递归地应用结合律、交换律、分配律、代数恒等以及常数折叠，使它成为乘积之和的规范化形式（其中构成一个乘积的两个项中有一个是，或者两个都是常数值分量之和）；交换律用于集中常数值分量（一般作为树根的左结点）；然后这棵树被分解成一条条的指令（假定这棵树不是所使用的中间代码形式）。

另一种方法是，可以将由MIR或LIR指令序列表示的计算合并成一个表达式（它不是正常的中间代码），对这个表达式施加代数化简转换，然后将得到的结果表达式再转换回到正常的中间代码序列。

336

在标识常数值分量过程中应当小心对待那些在当前上下文内（如循环内）是常数值，但在较大程序范围内不是常数值的分量。

为了实现MIR地址表达式的化简，我们将MIR表达式转换成树，按顺序递归地应用图12-6所示的转换规则，然后将树转换回到MIR。在这些规则中，c、c1和c2表示常数，t、t1、t2和t3表示任意中间代码树。

图12-7给出了计算前面讨论过的Pascal表达式a[i, j]的地址的原始树，以及对它施加地址表达式化简的第一个步骤。图12-8和图12-9给出了这一化简过程的其余步骤。注意，施加最后一个步骤当且仅当i是地址表达式所在上下文内的循环常数，并且C7应当在包含它的循环的入口之前计算，而不是在编译时计算。符号C1到C7表示如下常数值:

```
C1 = hi2 - lo2 + 1
C2 = -lo1 * C1
C3 = C2 - lo2
C4 = C3 * w
C5 = C1 * w
C6 = base_a + C4
C7 = C6 + C5 * i
```

bbs.theithome.com

超星公司制作请尊重作者版权

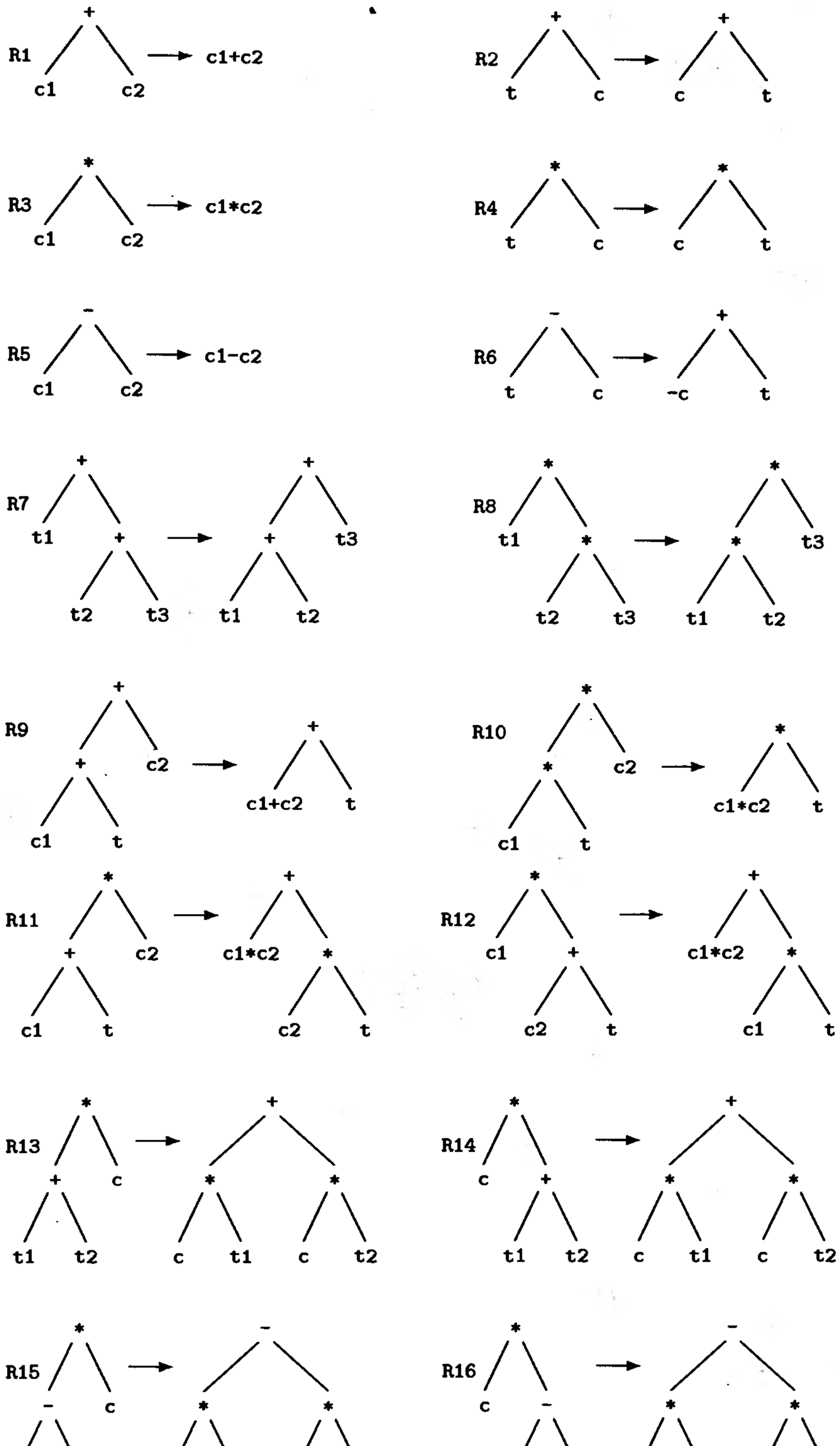


图12-6 为地址表达式化简而进行的树转换

bbs.theithome.com

超星公司制作 请尊重作者版权

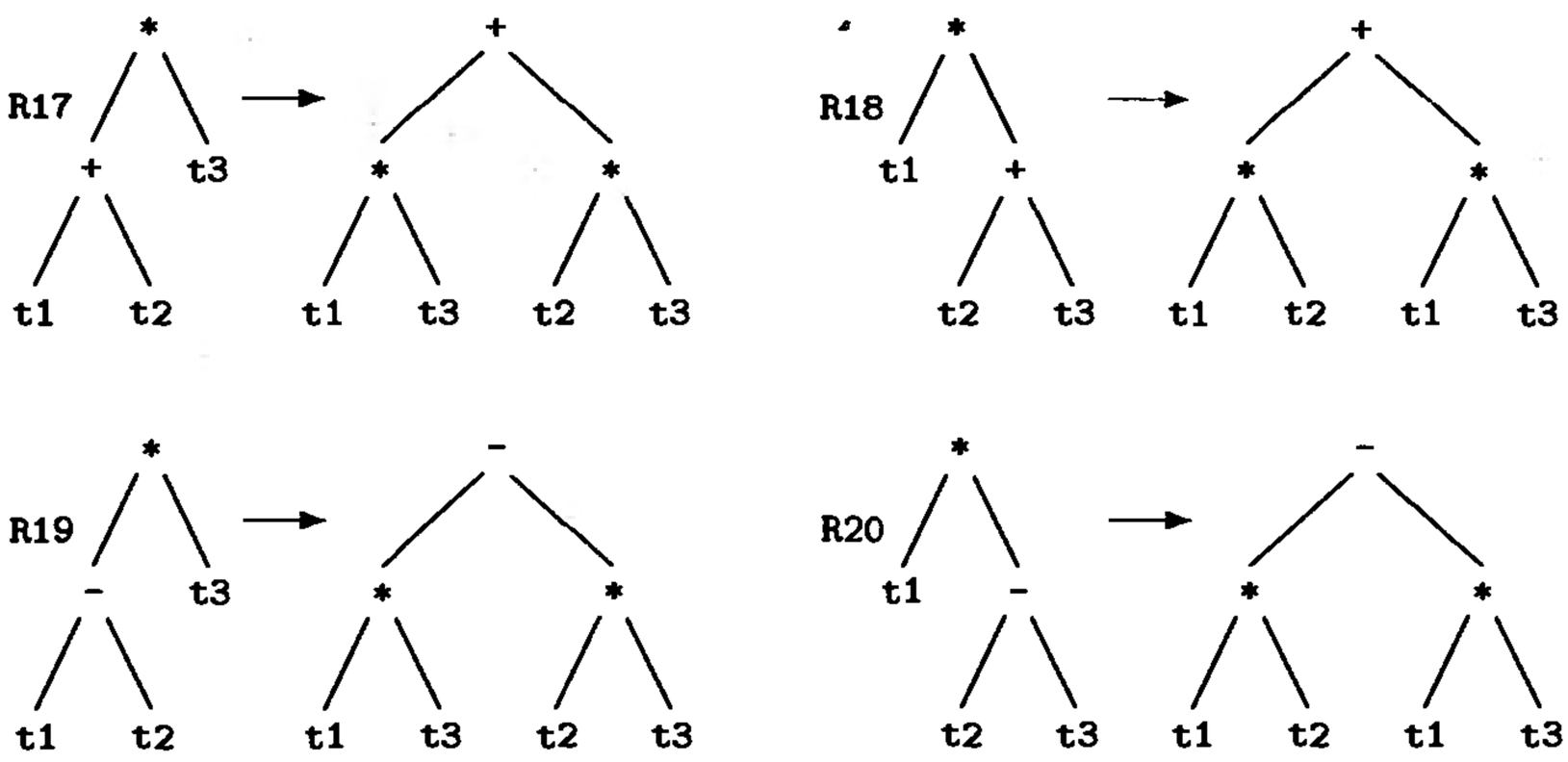


图12-6 （续）

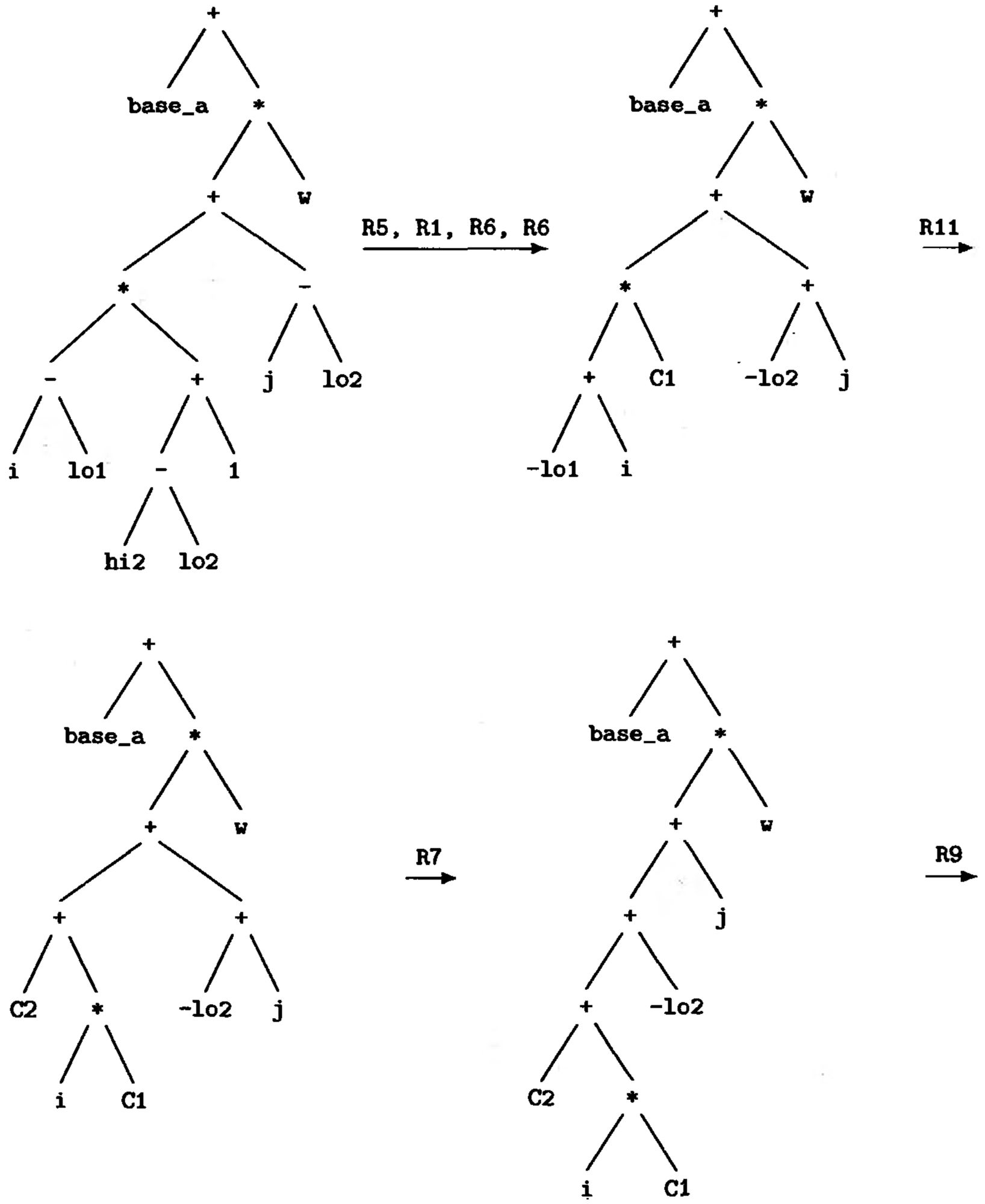


图12-7 计算Pascal表达式a[i，j]的地址的树，以及对它化简的第一个步骤

bbs.theithome.com

超星公司制作 请尊重作者版权

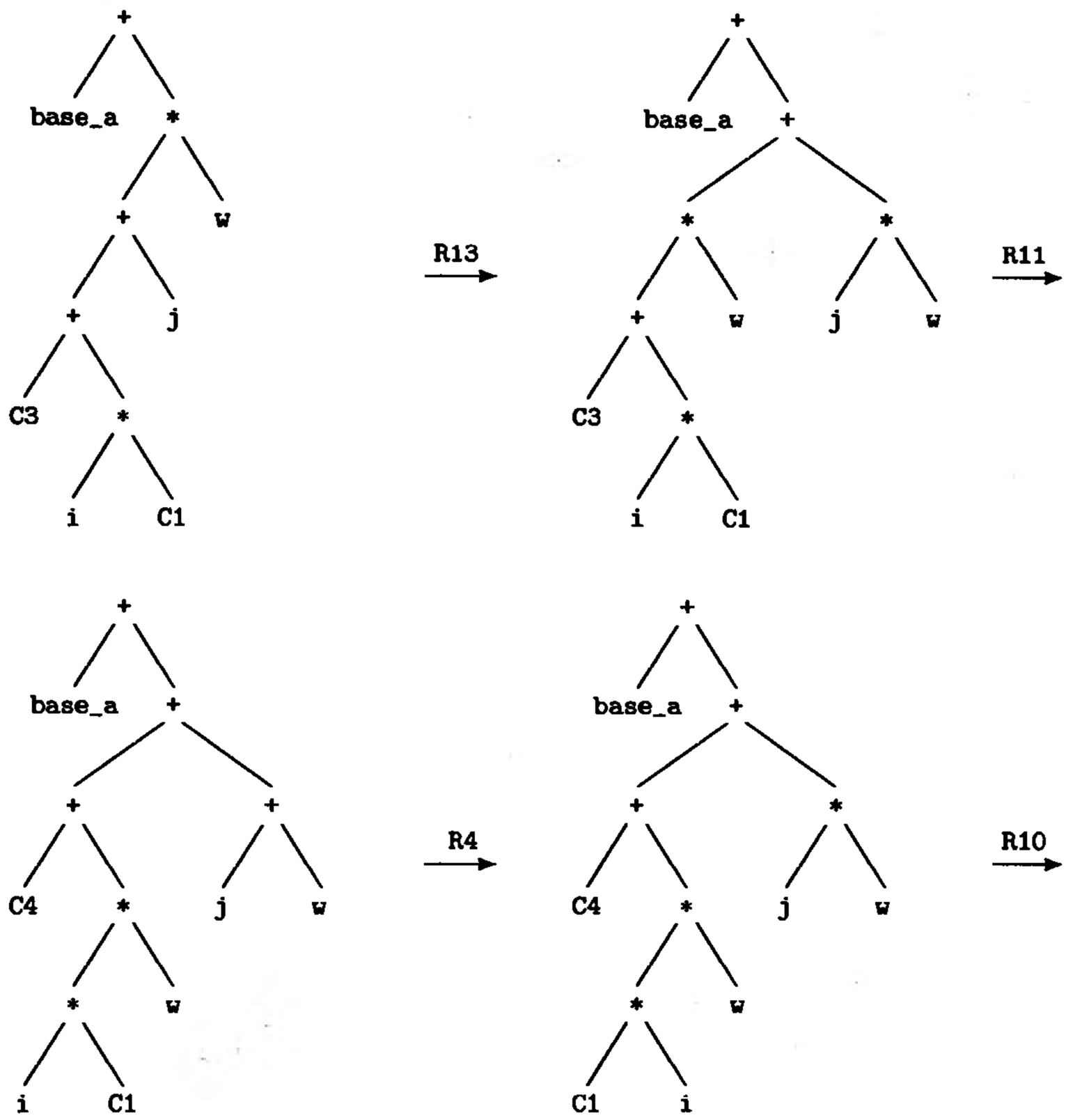


图12-8 简化a[i，j]的地址的其余步骤

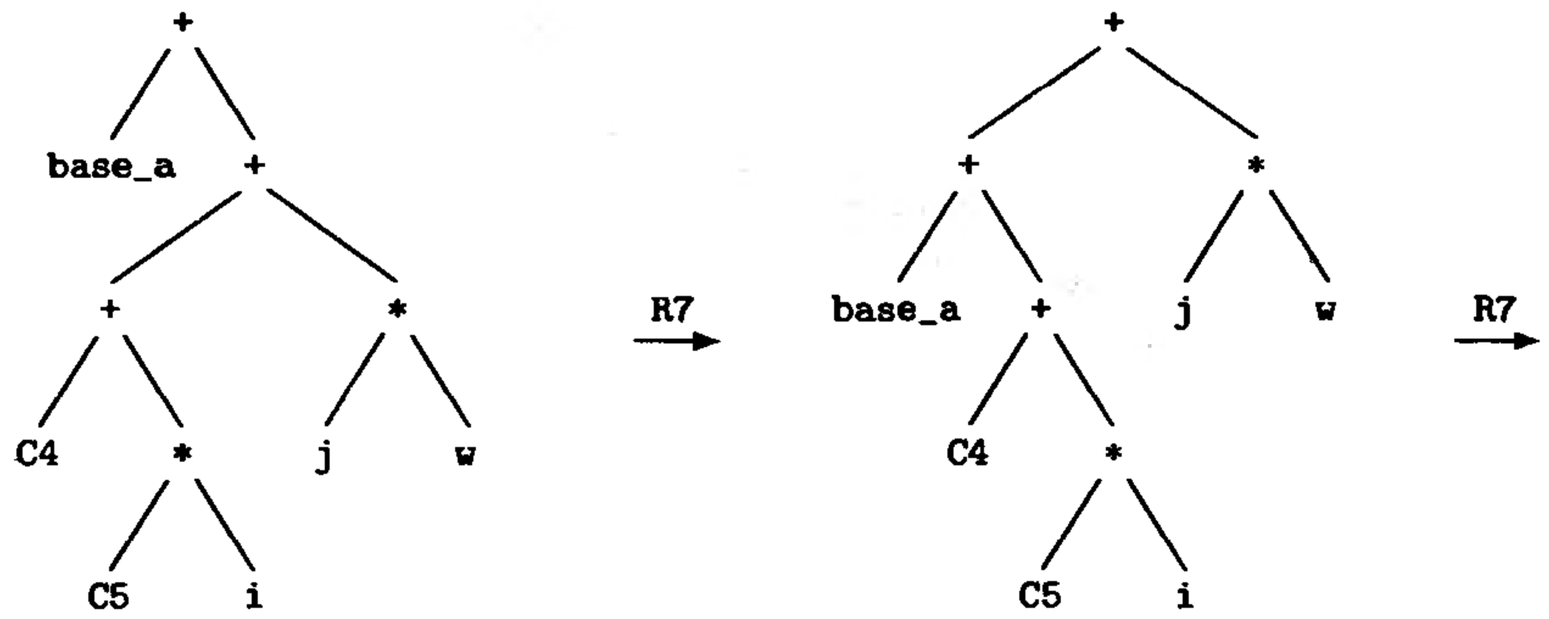


图12-9 简化a[i，j]的地址的最后一个步骤

bbs.theithome.com

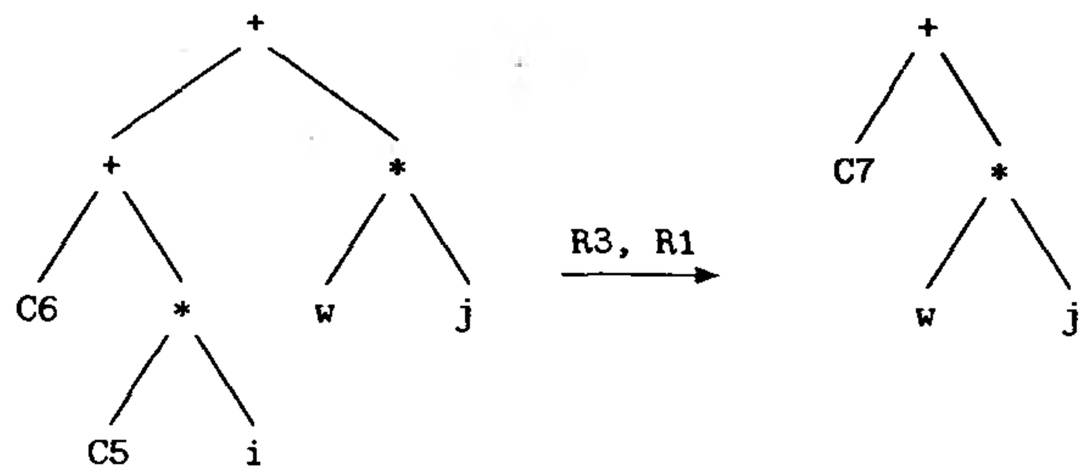


图12-9 （续）

常数部分的识别可能很简单，因为它们是源程序中的明显常数，或语义要求的常数；也可能通过数据流分析而得出。将上面的Pascal程序段改变成如图12-10所示情形，可得到后一种情形的例子。常数传播（参见12.6节）将告诉我们i是循环内的常数，而不仅仅是循环不变量，从而可以进行编译时的进一步化简。地址表达式的强度削弱（参见14.1.2节）常常也暴露了重结合的机会。

```
var a: array[lo1..hi1,lo2..hi2] of eltype;
   i, j: integer;
   . . .
i := 10;
. . .
do j = lo2 to hi2 begin
   a[i,j] := b + a[i,j]
end
```

图12-10 另一个访问数组元素的Pascal程序段

地址表达式中也存在着代数化简的其他机会。例如，在C中，如果p是一个指针，下面的表达式总是为真：

```
*(&p) = p
```

337 ~ 341

并且，如果q是指向一个含成员s的结构的指针，下面的表达式也总是为真。

```
(&q)->s = q.s
```

12.3.2 对浮点表达式应用代数化简

有心的读者可能已经注意到了，在这一节我们一直没有提到浮点计算，这是因为对它们很少能够安全地施加代数化简。例如，ANSI/IEEE浮点标准有带正号和负号的零，即+0.0和−0.0，并且对于任意正的有限值x，$x/+0.0=+\infty$，而$x/-0.0=-\infty$。此外，$x+0.0$和x不必相等，因为，如果x是一个会产生信号的*NaN*，则当执行前者时会发生异常，而后者不会。

令*MF*表示用给定精度可表示的最大有限浮点值。则

$$1.0+(MF-MF)=1.0$$

而

$$(1.0+MF)-MF=0.0$$

对浮点计算必须小心处理的另一个例子是如下代码：

```
eps := 1.0
while eps+1.0 > 1.0 do
   oldeps := eps
   eps := 0.5 * eps
od
```

这段代码计算的是使得$1+x>1$的最小数x并将结果赋给oldeps。如果通过用eps>0.0替代测试eps+1.0>1.0来“优化”它，则它实际计算的是使得$x/2$舍入到0的最大x。例如，如程序所写的，这个例程用双精度计算出oldeps=2.220446E-16，而它的这个“优化”版本计算出oldeps=4.940656E-324。20.4.2节讨论的循环转换会使这一问题更为严重。

bbs.theithome.com

Farnum [Farn88]认为对ANSI/IEEE浮点适合的代数化简只有两种，即删除不必要的类型强制和用等价的乘法替代除以常数的除法。不必要的类型强制的一个例子是

```
real s
double t
. . .
t := (double)s * (double)s
```

在具有单精度乘法运算的机器上执行时，它会产生一个双精度结果。 342

当用乘法替代除以一个常数的除法时，必须确认这个常数和它的倒数都是可精确表示的。利用ANSI/IEEE不精确标志能够容易地确认这一点。

12.4 值编号

值编号（value numbering）是判定两个计算是否等价并删除其中之一的若干种方法之中的一种方法。它在对计算所执行的操作不进行解释的情况下，将每个计算与一个符号值相关联，并使得任何两个具有相同符号值的计算总是计算相同的值。

具有类似效果的另外三种优化是稀有条件常数传播（12.6节）、公共子表达式删除（13.1节）和部分冗余删除（13.3节）。但事实上，值编号与这三种方法是不可比较的。图12-11的例子说明了值编号与其他三种方法之间的区别。在图12-11a中，值编号能确定出j和l被赋予了相同的值，但常数传播不能，因为它们的值依赖于i的输入值。而且，公共子表达式删除和部分冗余删除也都不能，因为在代码中没有公共子表达式。在图12-11b中，常数传播能确定出j和k被赋予了相同的值，因为它对算术运算进行解释，而值编号却不能。在图12-11c中，公共子表达式删除和部分冗余删除都能确定出第三个2*i的计算是冗余的，但值编号不能，因为l的值不总是等于j的值和总是等于k的值。由此，我们给出了值编号比其他三种方法更强有力的例子，同时也给出了其他三种方法比值编号更强有力的例子。我们在13.3节将看到，部分冗余删除包含了公共子表达式删除。

```
read(i)
j ← i + 1
k ← i
l ← k + 1
```

a)

```
i ← 2
j ← i * 2
k ← i + 2
```

b)

```
    read(i)
    l ← 2 * i
    if i > 0 goto L1
    j ← 2 * i
    goto L2
L1: k ← 2 * i
L2:
```

c)

图12-11 说明值编号、常数传播和公共子表达式删除不可比较的MIR例子

值编号原来的形式是针对单个基本块的，后来它被扩充到了针对扩展基本块，并且在最近， 343
它又被扩充成了对整个过程进行操作的全局形式（参见12.4.2节）。这种全局形式要求过程采用SSA形式。我们首先讨论施加于基本块的值编号，然后讨论施加于整个过程的基于SSA形式的值编号。

12.4.1 作用于基本块的值编号

为了在基本块内进行值编号，我们用散列方法对要计算的表达式进行分类。每当遇到一个表达式时，我们便计算它的散列值。如果这个表达式不在具有此散列值的表达式序列中，则将它加入到这个序列。如果表达式计算所在的指令不是赋值（例如，是一条if指令），我们将它

bbs.theithome.com

分解为两条指令，其中第一条指令计算这个表达式并存储它的结果到一个新的临时变量，第二条指令在原来表达式的地方使用这个临时变量（参见图12-12的例子）。如果表达式已经在散列值对应的表达式序列中，我们用序列中给出的指令的左端变量来替代当前计算。散列函数和表达式匹配函数的定义考虑到了运算符的交换律（参见图12-12）。

```
a ← i + 1
b ← 1 + i
i ← j
if i + 1 goto L1

c ← i + 1
```

a)

```
a ← i + 1
b ← a
i ← j
t1 ← i + 1
if t1 goto L1
c ← t1
```

b)

图12-12 基本块中的值编号。a) 中的指令序列被b) 中的指令序列所替换。注意根据交换率，第1条和第2条指令的表达式认为是相同的，并且第4条指令从一条二元运算if转换为一个赋值和一个值if

实现上述处理的代码在图12-13中给出。数据结构HashSeq[1..*m*]是一个数组，其中HashSeq[*i*]是指令的索引序列，这些指令中的表达式被散列到*i*且表达式的值是可用的。代码中用到了下面几个例程：

1. Hash(*opr*, *opd*1, *opd*2)返回其参数组成的表达式的散列值（若*opr*是一元运算符，*opd*2为nil）；若运算符是可交换的，对操作数的两种顺序，它返回相同的值。

```
Hash: (Operator × Operand × Operand) ⟶ integer

procedure Value_Number(m,nblocks,ninsts,Block,maxhash)
   m, nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   maxhash: in integer
begin
   i: integer
   HashSeq: array [1··maxhash] of sequence of integer
   for i :=  1 to maxhash do
      HashSeq[i] := []
   od
   i := 1
   while i ≤ ninsts[m] do
      case Exp_Kind(Block[m][i].kind) of
binexp:  i += Process_Inst(m,i,nblocks,Block,
            Block[m][i].opd1,Block[m][i].opd2,maxhash,HashSeq)
unexp:   i += Process_Inst(m,i,nblocks,Block,Block[m][i].opd,
            nil,maxhash,HashSeq)
default: i += 1
      esac
   od
end    || Value_Number

procedure Process_Inst(m,i,nblocks,nblocks,Block,opnd1,opnd2,
   maxhash,HashSeq) returns integer
   m, i, nblocks, maxhash: in integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   opnd1, opnd2: in Operand
   HashSeq: inout array [1··maxhash] of sequence of integer
begin
   hval, j, retval := 1: integer
   inst := Block[m][i], inst2: MIRInst
   doit := true: boolean
   tj: Var
   hval := Hash(inst.opr,opnd1,opnd2)
```

图12-13 在基本块内执行值编号的代码

bbs.theithome.com

```
for j := 1 to |HashSeq[hval]| do
   inst2 := Block[m][HashSeq[hval]↓j]
   if Match_Exp(inst,inst2) then
      || if expressions have the same hash value and they match,
      || replace later computation by result of earlier one
      doit := false
      if Has_Left(inst.kind) then
         Block[m][i] := ⟨kind:valasgn,left:inst.left,
            opd:⟨kind:var,val:inst2.left⟩⟩
      elif inst.kind ∈ {binif,unif} then
         Block[m][i] := ⟨kind:valif,opd:⟨kind:var,
            val:inst2.left⟩,lbl:inst.lbl⟩
      elif inst.kind ∈ {bintrap,untrap} then
         Block[m][i] := ⟨kind:valtrap,opd:⟨kind:var,
            val:inst2.left⟩,trapno:inst.trapno⟩
      fi
   fi
od
|| if instruction is an assignment, remove all expressions
|| that use its left-hand side variable
if Has_Left(inst.kind) then
   Remove(HashSeq,maxhash,inst.left,m,nblocks,Block)
fi
if doit then
   || if needed, insert instruction that uses result of computation
   if !Has_Left(inst.kind) then
      tj := new_tmp( )
      if Block[m][i].kind ∈ {binif,unif} then
         insert_after(m,i,ninsts,Block,⟨kind:valif,
            opd:⟨kind:var,val:tj⟩,label:Block[m][i].label)
         retval := 2
      elif Block[m][i].kind ∈ {bintrap,untrap} then
         insert_after(m,i,ninsts,Block,
            ⟨kind:valtrap,opd:⟨kind:var,val:tj⟩,
            trapno:Block[m][i].trapno)
         retval := 2
      fi
      || and replace instruction by one that computes
      || value for inserted instruction
      if opnd2 = nil then
         Block[m][i] := ⟨kind:unasgn,left:tj,
            opr:inst.opr,opd:opnd1⟩
      else
         Block[m][i] := ⟨kind:binasgn,left:tj,
            opr:inst.opr,opd1:opnd1,opd2:opnd2⟩
      fi
   fi
   HashSeq[hval] ⊕= [i]
   fi
   return retval
end    || Process_Inst
```

图12-13 （续）

2. Match_Exp(*inst*1, *inst*2)返回true，如果*inst*1和*inst*2中的表达式根据交换律是相同的。

3. Remove(*f*, *m*, *v*, *k*, nblocks, Block)从*f*[1..*m*]中删除所有使得Block[*k*][*i*]使用变量*v*作为操作数的指令索引*i*（参见图12-14关于Remove()的定义）。

bbs.theithome.com

```
procedure Remove(f,m,v,k,nblocks,Block)
   f: inout array [1··m] of sequence of integer
   m, k, nblocks: in integer
   v: in Var
   Block: in array [1··nblocks] of array [··] of MIRInst
begin
   i, j: integer
   for i := 1 to m do
      for j := 1 to |f[i]| do
         case Exp_Kind(Block[k][f[i]↓j].kind) of
binexp:     if Block[k][f[i]↓j].opd1.val = v
               ∨ Block[k][f[i]↓j].opd2.val = v then
               f[i] ⊖= j
            fi
unexp:      if Block[k][f[i]↓j].opd.val = v then
               f[i] ⊖= j
            fi
default: esac
      od
   od
end    || Remove
```

图12-14 从散列函数的散列链中删除已杀死的表达式的代码

作为Value_Number()的例子，考虑图12-15a的MIR代码。设maxhash=3。我们初始化HashSeq[1..3]为空序列，并置i=1。Block[m][1]的右端是一个binexp（二元表达式），因此hval被设置成它的散列值，比如说2，并且doit=true。HashSeq[2]=[]，因此我们前进到调用Remove(HashSeq, maxhash, a, m, n, Block)，这个调用没有做什么事，因为散列序列为空。接下来，因为doit=true并且Hash_Left(binasgn)=true，故我们将这条指令的索引加入到合适的散列序列，即HashSeq[2]=[1]。Proces_Inst()返回1，因此i被设置为2。

344
~
346

```
1    a ← x ∨ y
2    b ← x ∨ y
3    if !z goto L1
4    x ← !z
5    c ← x & y
6    if x & y trap 30
7
```

a)

```
a ← x ∨ y
b ← a
t1 ← !z
if t1 goto L1
x ← !z
c ← x & y
if x & y trap 30
```

b)

```
a ← x ∨ y
b ← a
t1 ← !z
if t1 goto L1
x ← t1
c ← x & y
if c trap 30
```

c)

图12-15 a) 一个基本块的例子，b) 对它的前三条指令应用值编号的结果，c) 对整个基本块应用值编号的结果。注意，第3行的if已经被两条指令所替代，其中第1条指令计算条件，第2条指令执行条件分支。

Block[m][2]有一个binexp作为它的右端，因此hval被设置成它的散列值2，并且doit=true。HashSeq[2]=[1]，于是我们调用Match_Exp()来比较第1条和第2条指令中的表达式，它返回true，所以我们设置doit=false，计算Has_Left(binasgn),并用b←a替代第2条指令。之后，我们调用Remove()从所有散列链中删除所有使用b作为操作数的指令。因为doit=true，并且指令2有一个左部，因此我们将它的索引插入到它的散列序列中，即HashSeq[2]=[1, 2]。接着，因为doit=false，故i被设置为3，于是我们前进

bbs.theithome.com

到第3条指令。

Block[m][3]有一个unexp（一元表达式）作为它的右端，此时hval被设置成它的散列值，比如说1，并且doit＝true。HashSeq[1]＝[]且Hash_Left(unif)＝false。因为 347
doit＝true，并且第3条指令没有左部，我们得到一个新的符号t1，将指令“if t1 goto L1”插入到指令3之后，这导致它后面的指令需要重新编号，用“t1←!z”替代指令3，并插入3到它的散列序列，即HashSeq[1]＝[3]。Proces_Inst（）返回2，因此i被设置为5，然后我们进入下一条指令。产生的基本块如图12-15b所示。

Block[m][5]有一个unexp作为它的右部，此时hval被设置成它的散列值1，并且doit＝true。HashSeq[1]＝[3]，所以我们调用Match_Exp()来比较在第3条和第5条指令中的这两个表达式，而它返回true。因为Hash_Left(unasgn)＝true，我们调用Remove()从所有散列链中删除所有使用x作为操作数的指令，这导致设置HashSeq[2]＝[]。因为doit＝true，并且指令5有一个左部，因此我们将它的索引插入到散列序列中，即HashSeq[1]＝[3，5]。Proces_Inst()返回1，因此i被设置为6，并且我们继续处理下一条指令。

Block[m][6]有一个binexp作为它的右部，此时hval被设置成它的散列值，比如说3，并且doit＝true。HashSeq[3]＝[]，因此，我们跳过检查表达式是否匹配的循环。因为Hash_Left(binasgn)＝true，我们调用Remove()从所有散列链中删除所有使用c作为操作数的指令。因为doit＝true，并且指令6有一个左部，因此我们将它的索引插入到它的散列序列中，即HashSeq[3]＝[6]。Proces_Inst()返回1，因此i被设置为7，并且我们进入下一条指令。

Block[m][7]包含一个binexp，此时hval被设置成它的散列值，即3，且doit＝true。HashSeq[3]＝[6]，所以我们调用Match_Exp()来比较在第6条和第7条指令中的这两个表达式，它返回true。同样，我们设置doit＝false。因为Hash_Left(binif)＝false，我们用“if c trap 30”替代Block[m][7]。因为doit＝false，并且没有后续指令，故处理过程结束。产生的基本块如图12-15c所示。

注意，在值编号和4.9.3节讨论的构造基本块的DAG表示之间有着强烈的相似之处。在DAG中，重用一个结点作为操作数而不是插入一个具有相同值的新结点，对应于删除对相同值的较后计算并用前面已计算的值替代它们。事实上，在构造DAG中频繁地使用了值编号。

12.4.2 全局值编号

最早的全局值编号方法是由Reif和Lewis [ReiL77]发明的。一种较新且更容易理解、同时（计算）复杂性也较小的方法是由Alpern、Wegman和Zadeck [AlpW88]开发的。我们的介绍基于后一种方法。

我们首先讨论变量的重合表示。两个变量是相互重合的（congruent），如果定义它们的计算具有相同的运算符（或具有相同的常数值），并且它们对应的操作数是重合的（当然，这就是值编号要做的事）。由此定义，只要a和b是重合的，c←a+1和d←b+1的左部变量就是重合的。但是，如我们将看到的，这个定义不够准确。为了使得它更准确，我们需要将执行全局值 348
编号的过程转换为SSA形式，并定义结果流图的所谓值图。

为了将流图转换到SSA形式，我们使用8.11节介绍的迭代的必经边界（dominance frontiers）方法，它产生过程的最小SSA表示。

过程的值图（value graph）是一个带有标志的有向图，它的结点上的标志是运算符、函数符号或常数，它的边表示生成赋值，并从运算符或函数指向它的操作数；边上标有自然数，它

bbs.theithome.com

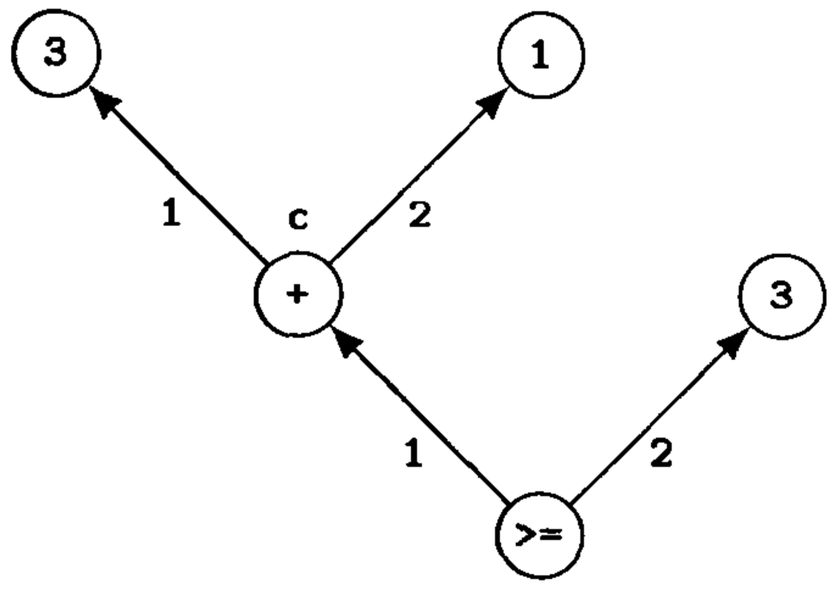
超星公司制作 请尊重作者版权

指明每一个操作数相对给定运算符或函数的位置。为了方便起见，我们也用SSA形式的变量来命名结点，它们指明由一个结点所表示的运算结果存储在何处；或者，当一个结点没有用SSA形式的变量命名时，我们给它附加一个任意的名字。

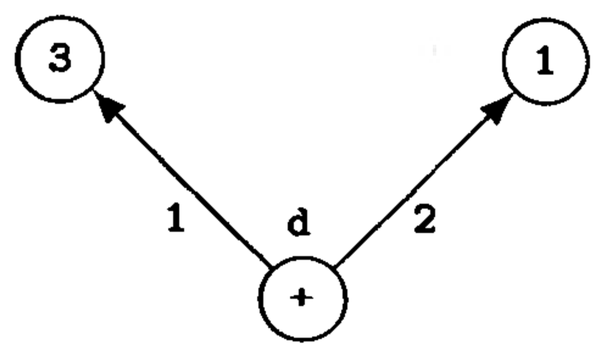
```
a ← 3
b ← 3
c ← a + 1
d ← b + 1
if c >= 3 then ...
```

图12-16 用于构造值图的一小段程序代码例子

例如，对于图12-16中给定的代码段，它对应的值图（我们不需要给图中的变量带下标，因为每一个都只有一个定义点）如图12-17所示。注意，由前面的定义，c和d是重合的。



a)



b)

图12-17 图12-16中代码的值图

下面考虑图12-18中的流图例子。它的最小SSA形式如图12-19所示。这个过程的值图含有环路，因为，例如i2依赖于i3，并且i3也依赖于i2。得到的值图如图12-20所示。名字为n的

349 结点中没有填结点标志，因为我们没有关于其值的信息。

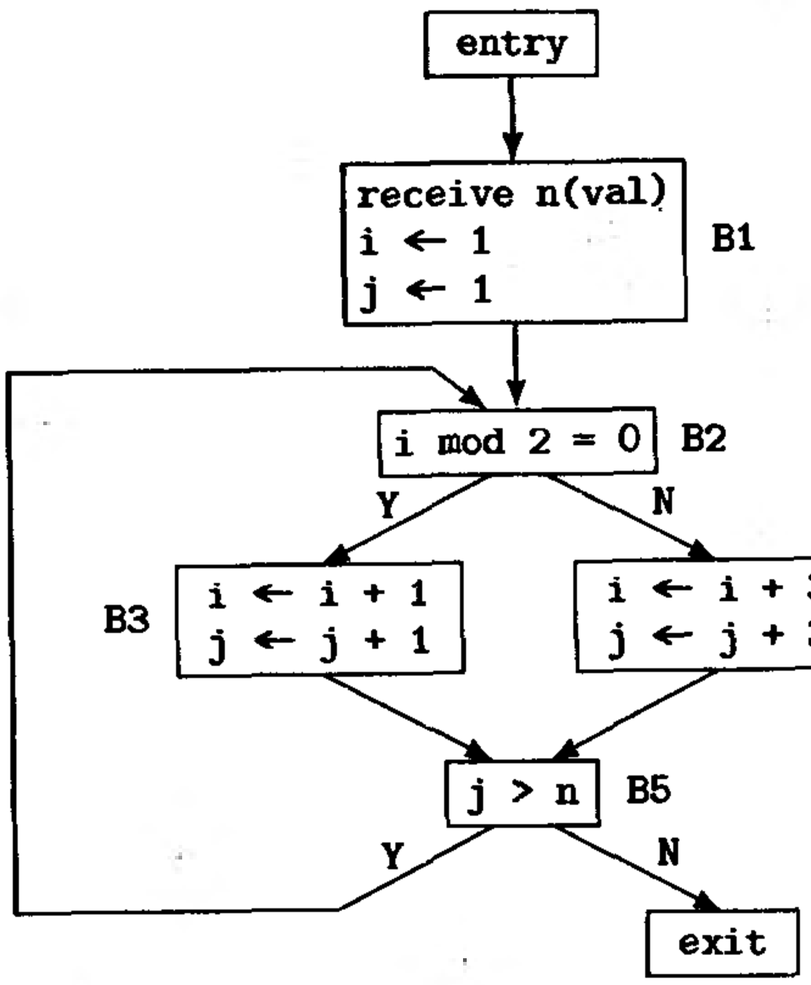


图12-18 值编号的流图之例

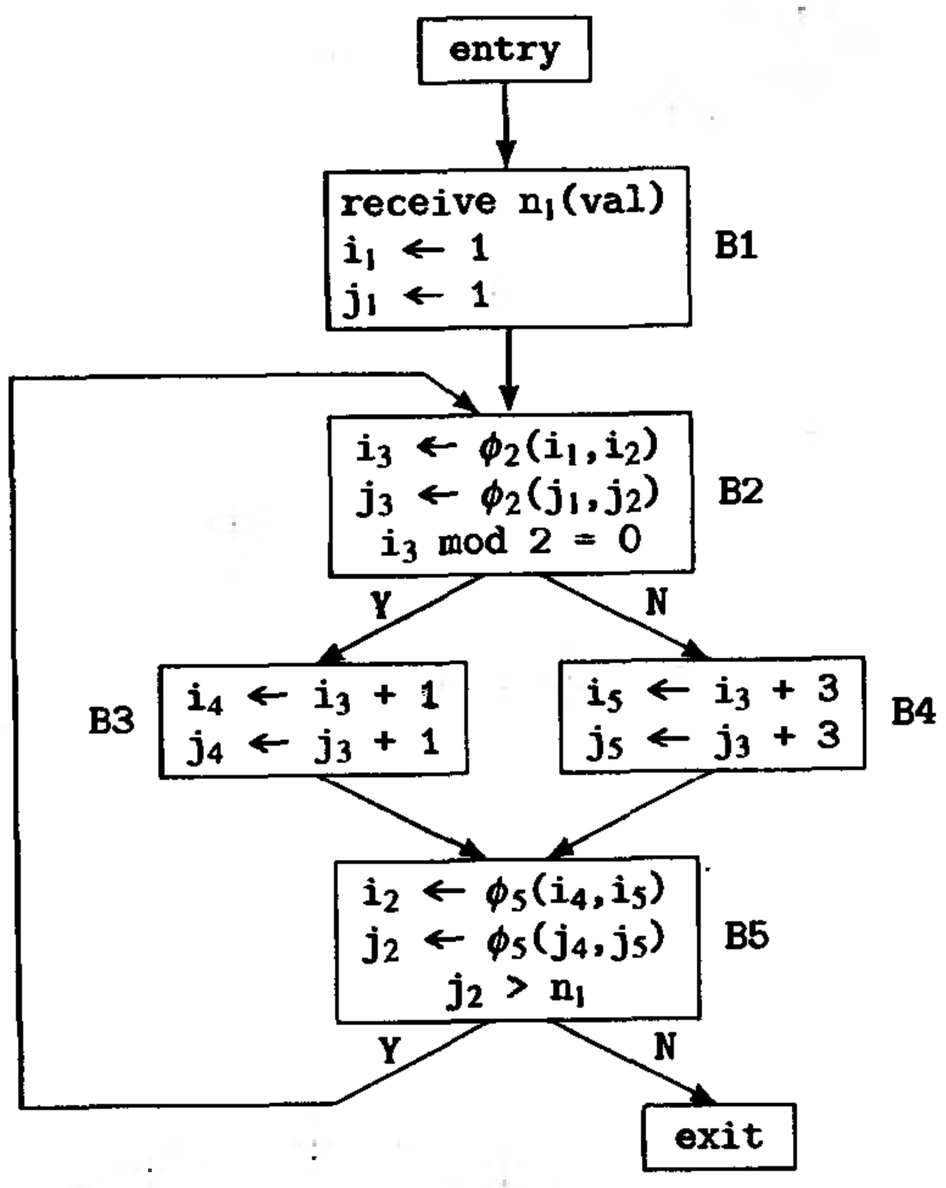


图12-19 图12-18中流图的最小SSA形式

现在，我们可以将重合（congruence）定义为在值图上满足下列条件的最大关系，即，两个结点是重合的，如果(1) 它们是相同的结点，(2) 它们的标志是常数并且它们的内容相同，或

bbs.theithome.com

版权所有 侵权必究 请勿擅自制作盗版

者(3) 它们具有相同的运算符，并且它们的操作数是重合的。两个变量在程序点p是等价的(equivalence)，如果它们是重合的，并且定义它们的赋值是点p的必经点。

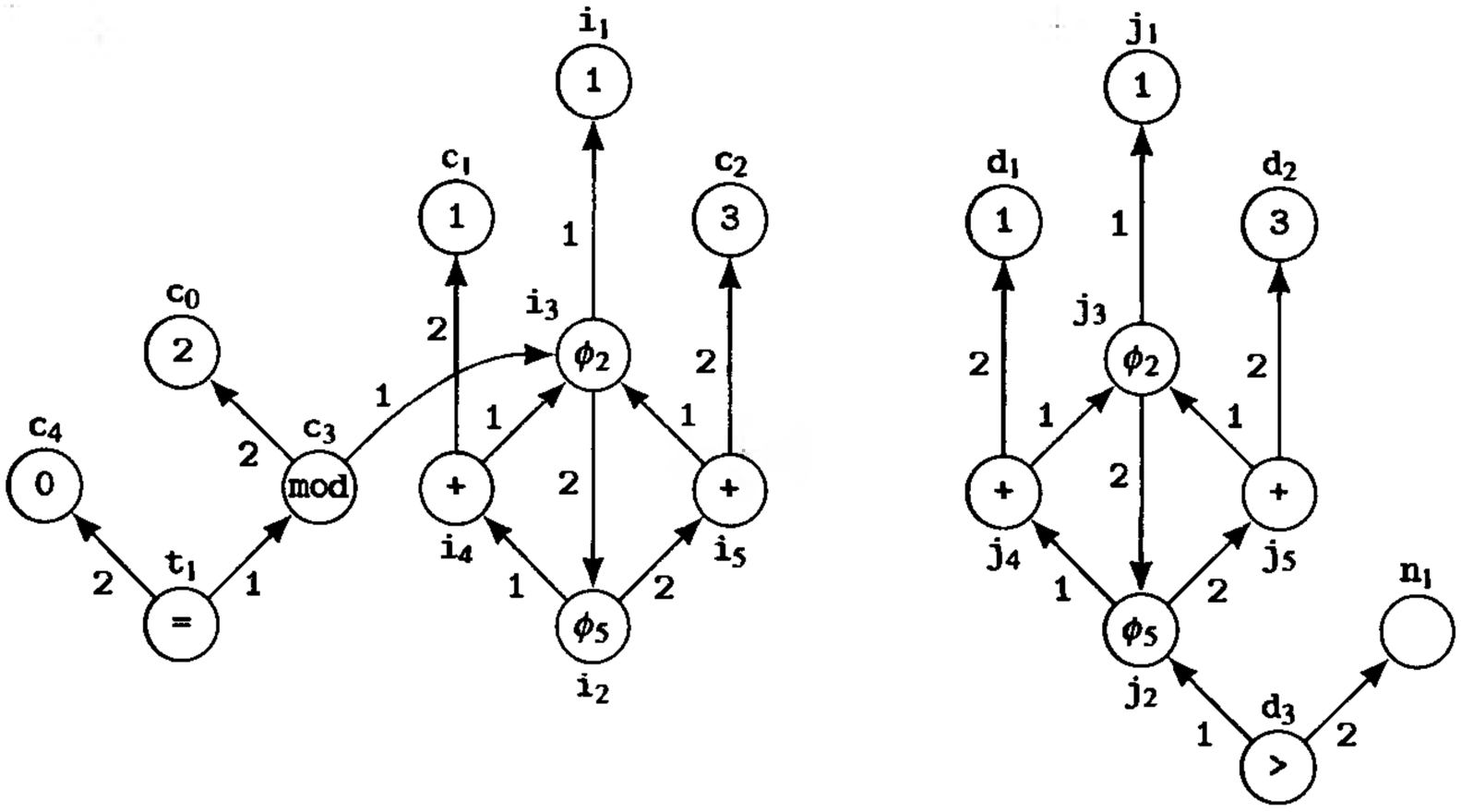


图12-20 图12-19中代码的值图

我们计算重合关系为对值图所执行的划分处理的最大不动点。一开始，我们假定具有相同标号的所有结点是重合的，然后根据一个划分中各个成员的操作数是否重合，重复地划分重合类，直到得出一个不动点，这个不动点由划分过程的特点决定了它一定是最大的。划分算法Global_Value_Number(*N*, *Nlabel*, *Elabel*, *B*)如图12-21所示。它使用了下面4种数据结构:

1. *N*是值图的结点集合。
2. *Nlabel*是映射结点到结点标号的函数。
3. *Elabel*是从结点到结点的带标号的边集合。
4. *B*是由算法设置的存放划分结果的数组。

```
NodeLabel = Operator ∪ Function ∪ Var ∪ Const

procedure Global_Value_Number(N,NLabel,ELabel,B) returns integer
   N: in set of Node
   NLabel: in Node ⟶ NodeLabel
   ELabel: in set of (Node × integer × Node)
   B: inout array [··] of set of Node
begin
   i, j1, k1, m, x, z: Node
   j, k, p: integer
   S, Worklist: set of Node
   || initialize partitions in B[n] and map nodes to partitions
   p := Initialize(N,NLabel,B,Worklist)
   while Worklist ≠ ∅ do
      i := ♦Worklist
      Worklist -= {i}
      m := ♦B[i]
      || attempt to subdivide each nontrivial partition
      || until the worklist is empty
      for j := 1 to Arity(NLabel,i) do
         j1 := Follow_Edge(ELabel,m,j)
```

图12-21 通过计算重合关系进行全局值编号的划分算法

bbs.theithome.com

```
        S := B[i] - {m}
        while S ≠ Ø do
            x := ♦S
            S -= {x}
            if Follow_Edge(ELabel,x,j) ≠ j1 then
                p += 1
                B[p] := {m}
                B[i] -= {m}
                while S ≠ Ø do
                    z := ♦S
                    S -= {z}
                    for k := 1 to Arity(NLabel,i) do
                        k1 := Follow_Edge(ELabel,m,k)
                        if k1 ≠ Follow_Edge(ELabel,z,k) then
                            B[p] ∪= {z}
                            B[i] -= {z}
                        fi
                    od
                od
                if |B[i]| > 1 then
                    Worklist ∪= {i}
                fi
                        if |B[p]| > 1 then
                            Worklist ∪= {p}
                        fi
                    fi
                od
            od
            return p
        end     || Global_Value_Number
```

图12-21 （续）

350～352

这个算法基于Aho、Hopcroft和Ullman [AhoH74]的算法，它使用一张工作表来存放需要考察的一组划分，并且使用如下3个函数：

1. Initialize(*N*, *NLabel*, *B*, *Worklist*) 用值图结点的初始划分对*B*[1]直到某个*B*[*p*]进行初始化（即，具有相同标号的所有结点属于相同的划分），用初始工作表对*Worklist*进行初始化，并返回*p*作为函数值。

2. Arity(*NLabel*, *j*) 返回*B*[*j*]中运算符的操作数个数。

3. Follow_Edge(*Elabel*, *x*, *j*) 返回一个结点*y*，此结点有一条带标号的边$\langle x, j, y\rangle \in ELabel$。

第1和第3个函数的代码如图12-22所示。Arity()的计算是简单的。这个划分算法最坏的运行时间是$O(e \cdot \log e)$，其中*e*是值图的边数。

对于图12-19中的流图例子，初始划分的个数*p*是11，初始划分如下：

B[1] = {c_1,d_1,i_1,j_1}
B[2] = {c_2,d_2}
B[3] = {c_0}
B[4] = {c_3}
B[5] = {n_1}
B[6] = {d_3}
B[7] = {i_3,j_3}

bbs.theithome.com

超星公司制作 请尊重作者版权

B[8] = {i_4,j_4,i_5,j_5}
B[9] = {i_2,j_2}
B[10] = {c_4}
B[11] = {t_1}

```
procedure Initialize(N,NLabel,B,Worklist) returns integer
    N: in set of Node
    NLabel: in Node --> NodeLabel
    B: out array [··] of set of Node
    Worklist: out set of Node
begin
    i, k := 0: integer
    v: Node
    || assemble partitions, node-to-partition map, and initial worklist
    Worklist := ∅
    for each v ∈ N do
        i := 1
        while i ≤ k do
            if NLabel(v) = NLabel(♦B[i]) then
                B[i] ∪= {v}
                if Arity(NLabel,v) > 0 & |B[i]| > 1 then
                    Worklist ∪= {i}
                fi
                i := k + 1
            fi
            i += 1
        od
        if i = k+1 then
            k += 1
            B[k] := {v}
        fi
    od
    return k
end    || Initialize

procedure Follow_Edge(ELabel,x,j) returns Node
    ELabel: in set of (Node × integer × Node)
    x: in Node
    j: in integer
begin
    el: Node × integer × Node
    for each el ∈ ELabel do
        if x = el@1 & j = el@2 then
            return el@3
        fi
    od
end    || Follow_Edge
```

图12-22 图12-21划分算法使用的辅助例程

Worklist的初值是{7, 8, 9}。经划分处理得到如下12个划分:

B[1] = {c_1,d_1,i_1,j_1}
B[2] = {c_2,d_2}
B[3] = {c_0}
B[4] = {c_3}
B[5] = {n_1}
B[6] = {d_3}
B[7] = {i_3,j_3}

bbs.theithome.com

B[8] = {i_4,j_4}
B[9] = {i_2,j_2}
B[10] = {c_4}
B[11] = {t_1}
B[12] = {i_5,j_5}

因此，值图中与i和j对应的那些结点是重合的，而且可以从中确定出变量的等价性。

作为第二个例子，假设我们将图12-18基本块B4中的赋值i←i+3改变成i←i-3，则除了名字为i_5的结点包含的是“-”而不是“+”之外，它的值图与图12-20相同。除了p是12，并且B[8]到B[11]用下面的值替代之外，它的初始划分也与前面给出的原来的程序的初始划分相同。

B[8] = {i_4,j_4,j_5}
B[9] = {i_5}
B[10] = {i_2,j_2}
B[11] = {c_4}
B[12] = {t_1}

最后得到的划分结果是，每一个i_2、i_3、i_4、i_5、j_2、j_3、j_4和j_5都位于不同的划分中。

Alpern、Wegman和Zadeck讨论了对这种全局值编号方法的一系列的推广，其中包括:

1. 对要处理的程序做结构分析（见7.7节），并利用为控制流结构设计的特殊的Φ函数，以便能够确定关于控制流的重合关系;

2. 通过用模拟将这种方法应用于数组运算，例如，用

```
a ← update(a, i, 2 *access(b, i))
```

模拟

```
a[i] ← 2 * b[i]
```

3. 考虑交换律，以便能够识别诸如a*b和b*a是重合的情况。

其中的每一种改变都能使被检测出的重合个数增加。

Briggs、Cooper和Simpson扩展了基于散列的值编号方法，使其能工作于例程的必经结点树，扩展了前面讨论的全局方法使它考虑到表达式的有效性（参见13.3节），并且对基于散列的值编号方法和全局值编号方法进行了比较，它们得出的结论指出这两种方法是不可比较的——每一种方法都存在优于另一种方法的情形。在较后的论文中，Cooper和Simpson讨论了一种全局值编号方法，这种方法工作于例程的SSA形式的强连通分量，并结合了散列方法和全局方法中的优点，因而比这两种方法都更有效。

353 ~ 355

12.5 复写传播

复写传播（copy propagation）是一种转换，对于给定的关于变量x和y的赋值$x \leftarrow y$，这种转换用y来替代后面出现的x的引用，只要在这期间没有指令改变x或y的值。

从现在起，我们需要将过程表示为由基本块组成的数组，其中每一个基本块是一个由指令组成的数组。我们使用变量nblocks、数组ninsts[1..nblocks]和Block[1..nblocks][..]来做复写传播。这两个数组的声明如下:

```
nblocks: integer
ninsts: array [1··nblocks] of integer
Block: array [1··nblocks] of array [··] of Instruction
```

其中Block[i]由指令Block[i][1]到Block[i][ninsts[i]]组成。

在继续详细讨论复写传播之前，我们先考虑它与寄存器合并（16.3节将详细讨论它）的关

bbs.theithome.com

系。只要优化针对的是已经用寄存器（符号寄存器[⊖]或真实的寄存器）替代了标识符的低级中间代码，这两种转换在效果上就是相同的。但是，判别能否对一个具体的复写赋值施加寄存器合并或施加复写传播，在所采用的方法上两者是不同的：我们对复写传播使用数据流分析，而对寄存器合并使用冲突图。另一点不同是，复写传播可以在不论高级还是低级的任何级别的中间代码上进行。

例如，给定图12-23a的流图，基本块B1中的指令b←a是一个复写赋值。在这条指令之后，a和b都没有再被赋值，因此b的所有使用都可以用a来替代，如图12-23b所示。尽管这似乎不会对这段代码有很大的改善，但正是它才使得b成为无用的——b)图中没有指令用它作为操作数——因此，死代码删除（参见18.10节）可以删去赋值b←a；并且当a是整数值时，这个替换还使得用移位而不是加法来计算赋给e的值成为可能。

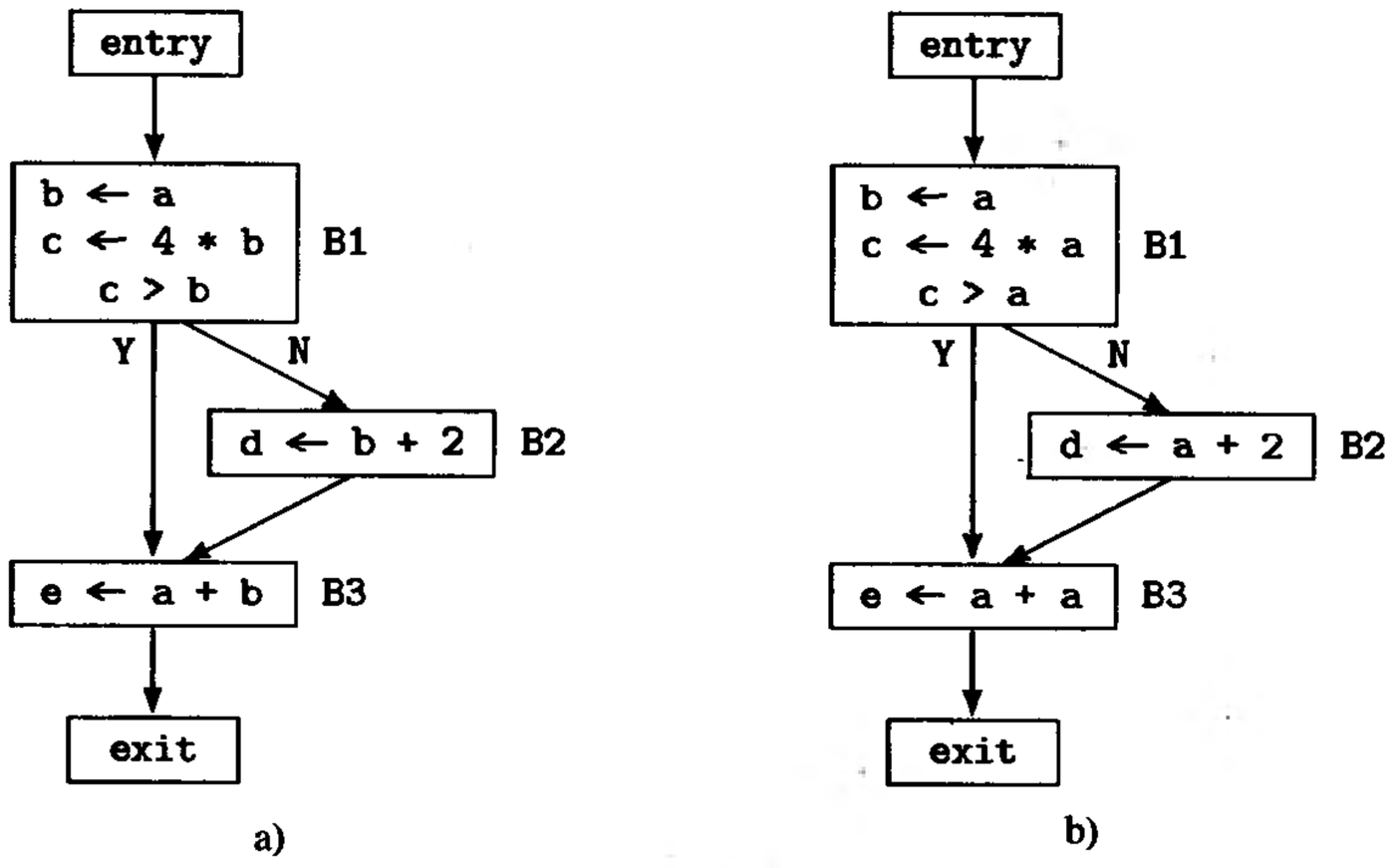


图12-23 a) 要传播的复写赋值之例，即B1中的b←a，b) 对它做复写传播后的结果

复写传播可以分为局部遍和全局遍来实现，前者在各个基本块之内操作，后者跨整个流图操作；也可以只用单独一个全局遍来实现。为了使其运行时间在n的线性范围之内，在图12-24给出的算法中，我们使用一种有效复写指令*ACP*表的散列实现方法。此算法假定提供一个由MIR指令Block[m][1], ..., Block[m][n]组成的数组作为输入。

```
procedure Local_Copy_Prop(m,n,Block)
   m, n: in integer
   Block: inout array [1··n] of array [··] of MIRInst
begin
   ACP := ∅: set of (Var × Var)
   i: integer
   for i := 1 to n do
      || replace operands that are copies
      case Exp_Kind(Block[m][i].kind) of
binexp:     Block[m][i].opd1.val := Copy_Value(Block[m][i].opd1,ACP)
            Block[m][i].opd2.val := Copy_Value(Block[m][i].opd2,ACP)
```

图12-24 局部复写传播的 $O(n)$ 算法

⊖ 符号寄存器，如在LIR中所见的，是机器真实寄存器的一种扩充，它扩大真实寄存器的个数同程序生成代码所需要的寄存器个数一样多。将符号寄存器装入真实的寄存器是全局寄存器分配的任务，分配过程中可能还附带生成保护和恢复它们的值的存和取指令。

bbs.theithome.com

```
unexp:      Block[m][i].opd.val := Copy_Value(Block[m][i].opd,ACP)
listexp:    for j := 1 to |Block[m][i].args| do
               Block[m][i].args↓j@1.val :=
                  Copy_Value(Block[m][i].args↓j@1,ACP)
            od
default: esac
         || delete pairs from ACP that are invalidated by the current
         || instruction if it is an assignment
         if Has_Left(Block[m][i].kind) then
            Remove_ACP(ACP,Block[m][i].left)
         fi
         ||  insert pairs into ACP for copy assignments
         if Block[m][i].kind = valasgn & Block[m][i].opd.kind = var
            & Block[m][i].left ≠ Block[m][i].opd.val then
            ACP ∪= {<Block[m][i].left,Block[m][i].opd.val>}
         fi
      od
end    || Local_Copy_Prop

      procedure Remove_ACP(ACP,v)
         ACP: inout set of (Var × Var)
         v: in Var
      begin
         T := ACP: set of (Var × Var)
         acp: Var × Var
         for each acp ∈ T do
            if acp@1 = v ∨ acp@2 = v then
               ACP -= {acp}
            fi
         od
      end    || Remove_ACP

      procedure Copy_Value(opnd,ACP) returns Var
         opnd: in Operand
         ACP: in set of (Var × Var)
      begin
         acp: Var × Var
         for each acp ∈ ACP do
            if opnd.kind = var & opnd.val = acp@1 then
               return acp@2
            fi
         od
         return opnd.val
      end    || Copy_Value
```

图12-24 （续）

作为使用所得$O(n)$算法的一个例子，考虑图12-25中的代码。第2列给出了在应用此算法之前由5条指令组成的一个基本块，第4列给出的是应用这个算法后的结果，第3列给出的是每一步的ACP值。

为了执行全局复写传播，我们首先做数据流分析，以确定哪些复写赋值未受损害地到达了它们左部变量的使用，即，在这中间没有重新定义这两个变量。我们定义集合*COPY*(*i*)由出现在基本块*i*中、并到达了基本块*i*出口的那些复写赋值实例所组成。更准确地，*COPY*(*i*)是一个由满足如下条件的四元组 <*u*, *v*, *i*, *pos*> 组成的集合，其中，*u*←*v*是一个复写赋值，*pos*是此赋值所在基本块*i*中的一个位置，并且*u*和*v*在基本块*i*中较后没有再被赋值。我们定义*KILL*(*i*)是被基本块*i*杀死的复写赋值实例集合，即，*KILL*(*i*)是满足后面条件的四元组<*u*, *v*, *blk*, *pos*>组成的集

bbs.theithome.com

超星公司制作 请尊重作者版权

356 ~ 358

Position	Code Before	ACP	Code After
		∅	
1	b ← a		b ← a
		{⟨b,a⟩}	
2	c ← b + 1		c ← a + 1
		{⟨b,a⟩}	
3	d ← b		d ← a
		{⟨b,a⟩,⟨d,a⟩}	
4	b ← d + c		b ← a + c
		{⟨d,a⟩}	
5	b ← d		b ← a
		{⟨d,a⟩,⟨b,a⟩}	

图12-25 线性时间局部复写传播算法的例子

合，其中，$u \leftarrow v$是出现在基本块$blk \neq i$中位置pos处的一个复写赋值。例如，对于图12-26中的例子，*COPY*()和*KILL*()集合是:

$$
\begin{aligned}
COPY(\texttt{entry}) &= \emptyset \\
COPY(\texttt{B1}) &= \{(\texttt{d}, \texttt{c}, \texttt{B1}, 2)\} \\
COPY(\texttt{B2}) &= \{(\texttt{g}, \texttt{e}, \texttt{B2}, 2)\} \\
COPY(\texttt{B3}) &= \emptyset \\
COPY(\texttt{B4}) &= \emptyset \\
COPY(\texttt{B5}) &= \emptyset \\
COPY(\texttt{B6}) &= \emptyset \\
COPY(\texttt{exit}) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
KILL(\texttt{entry}) &= \emptyset \\
KILL(\texttt{B1}) &= \{(\texttt{g}, \texttt{e}, \texttt{B2}, 2)\} \\
KILL(\texttt{B2}) &= \emptyset \\
KILL(\texttt{B3}) &= \emptyset \\
KILL(\texttt{B4}) &= \emptyset \\
KILL(\texttt{B5}) &= \emptyset \\
KILL(\texttt{B6}) &= \{(\texttt{d}, \texttt{c}, \texttt{B1}, 2)\} \\
KILL(\texttt{exit}) &= \emptyset
\end{aligned}
$$

下面我们定义关于*CPin*(*i*)和*CPout*(*i*)的数据流方程，它们分别表示在基本块*i*的入口和出口对复写传播有效的复写赋值集合。一个复写赋值在基本块*i*的入口处是有效的，如果它从基本块*i*的所有前驱出口时是有效的。因此路径合并运算符是交运算。一个复写赋值从基本块*j*出口时是有效的，如果它属于*COPY*(*j*)；或者，它在基本块*j*的入口是有效的，且没有被基本块*j*杀死，即，它属于*CPin*(*j*)但不属于*KILL*(*j*)。因此数据流方程是:

$$CPin(i) = \bigcap_{j \in Pred(i)} CPout(j)$$

$$CPout(i) = COPY(i) \cup (CPin(i) - KILL(i))$$

并且合适的初值是$CPin(\texttt{entry}) = \emptyset$，且对所有的$i \neq \texttt{entry}$，$CPin(i) = U$，其中$U$是四元式全

bbs.theithome.com

集，或至少有

$$U=\bigcup_i COPY(i)$$

全局复写传播的数据流分析可以有效地用集合的位向量表示来实现。

给定了数据流信息*CPin*()，并且假定已经进行了局部复写传播，我们按如下所示执行全局复写传播:

1. 对于每一个基本块*B*，置ACP＝{a∈Var×Var，其中∃w∈integer使得<a@1，a@2，*B*，w>∈CPin(*B*)}。

2. 对于每一个基本块*B*，对*B*执行图12-24的局部复写传播算法（省略赋值ACP:=∅）。

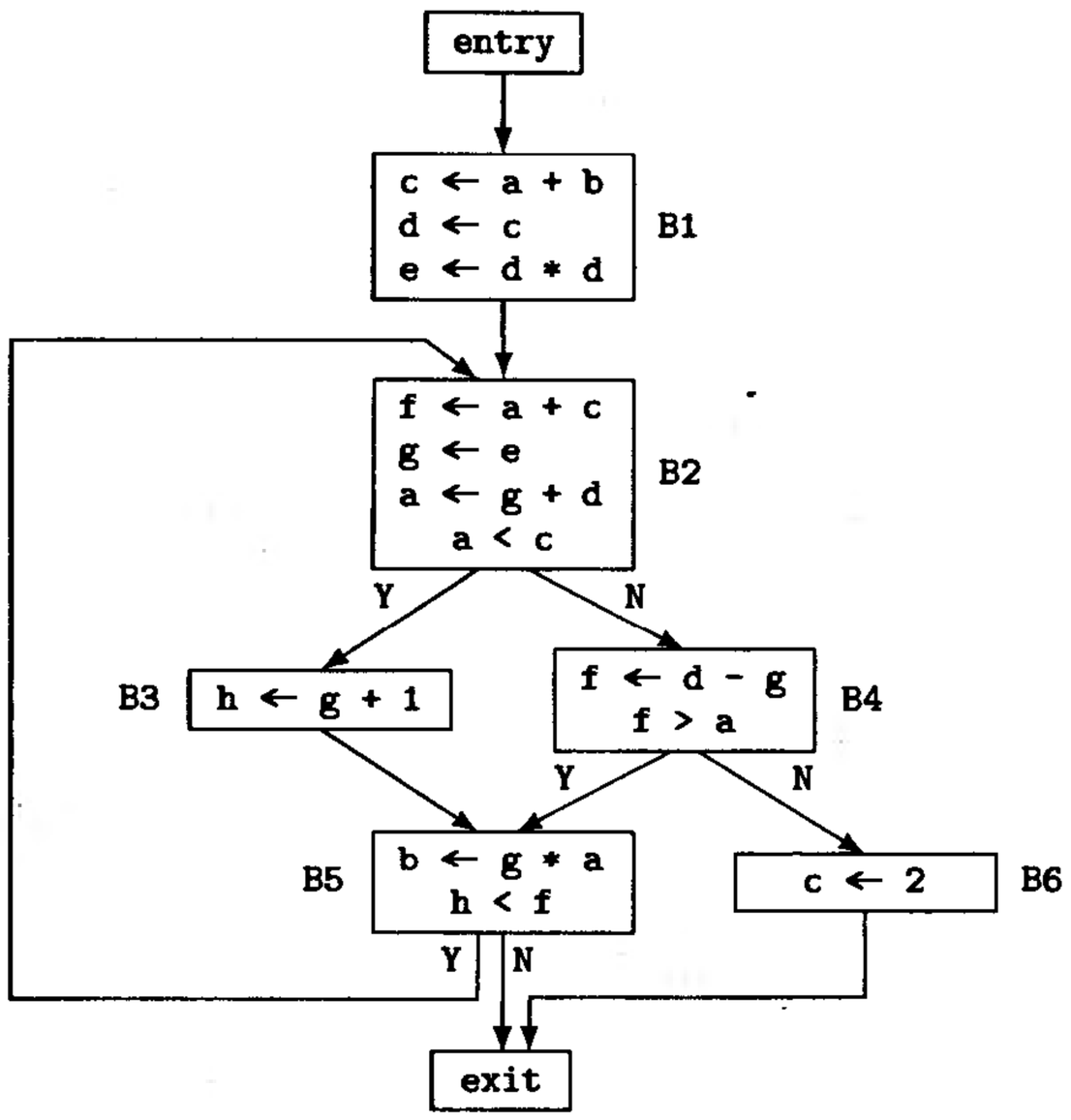


图12-26 复写传播的另一个例子

对于图12-26的例子，*CPin*()集合是

CPin(entry) = ∅
CPin(B1) = ∅
CPin(B2) = {⟨d, c, B1, 2⟩}
CPin(B3) = {⟨d, c, B1, 2⟩, ⟨g, e, B2, 2⟩}
CPin(B4) = {⟨d, c, B1, 2⟩, ⟨g, e, B2, 2⟩}
CPin(B5) = {⟨d, c, B1, 2⟩, ⟨g, e, B2, 2⟩}
CPin(exit) = {⟨g, e, B2, 2⟩}

对B1做局部复写传播和对整个过程做全局复写传播，导致图12-26的流图转换成图12-27所示的流图。

局部复写传播不难推广到扩展基本块。为了做到这样，我们按前序次序处理构成扩展基本块的每一个基本块，前序是每一个基本块的处理都先于其后继的一种顺序，并且对于除初始基本块之外的每个基本块，用来自它的前驱基本块的ACP表的最终值给它的ACP表赋初值。对应地，全局复写传播算法也可推广到以扩展基本块作为结点，并且数据流信息与这种结点相连。为了做到这样，我们必须给扩展基本块的每一个出口相连一个独立的*CPout*()，因为经过扩展基本块的各条路径一般会有不同的有效复写赋值。

bbs.theithome.com

超星公司制作请尊重作者版权

359~361

对于图12-26的例子，如果我们在局部复写传播之后接着进行全局复写传播（两者都基于扩展基本块），其结果是相同的，但更多的工作发生在局部阶段中。基本块B2、B3、B4和B6组成了一个扩展基本块，局部阶段将基本块B2中赋给g的值e传播到扩展基本块中的所有基本块。

注意，全局复写传播算法不会识别图12-28的基本块B2和B3中的两个x←y语句是复写赋值。一种称为尾融合的转换能将这两个复写赋值合并为一个，它实际上是将这个赋值移到只有该赋值的一个独立基本块中。由此复写传播便能够识别它，并传播这个赋值到B4。但是，这给某些编译提出了一个关于各个处理遍的顺序问题：尾融合一般要到已经生成机器指令之后才进行。

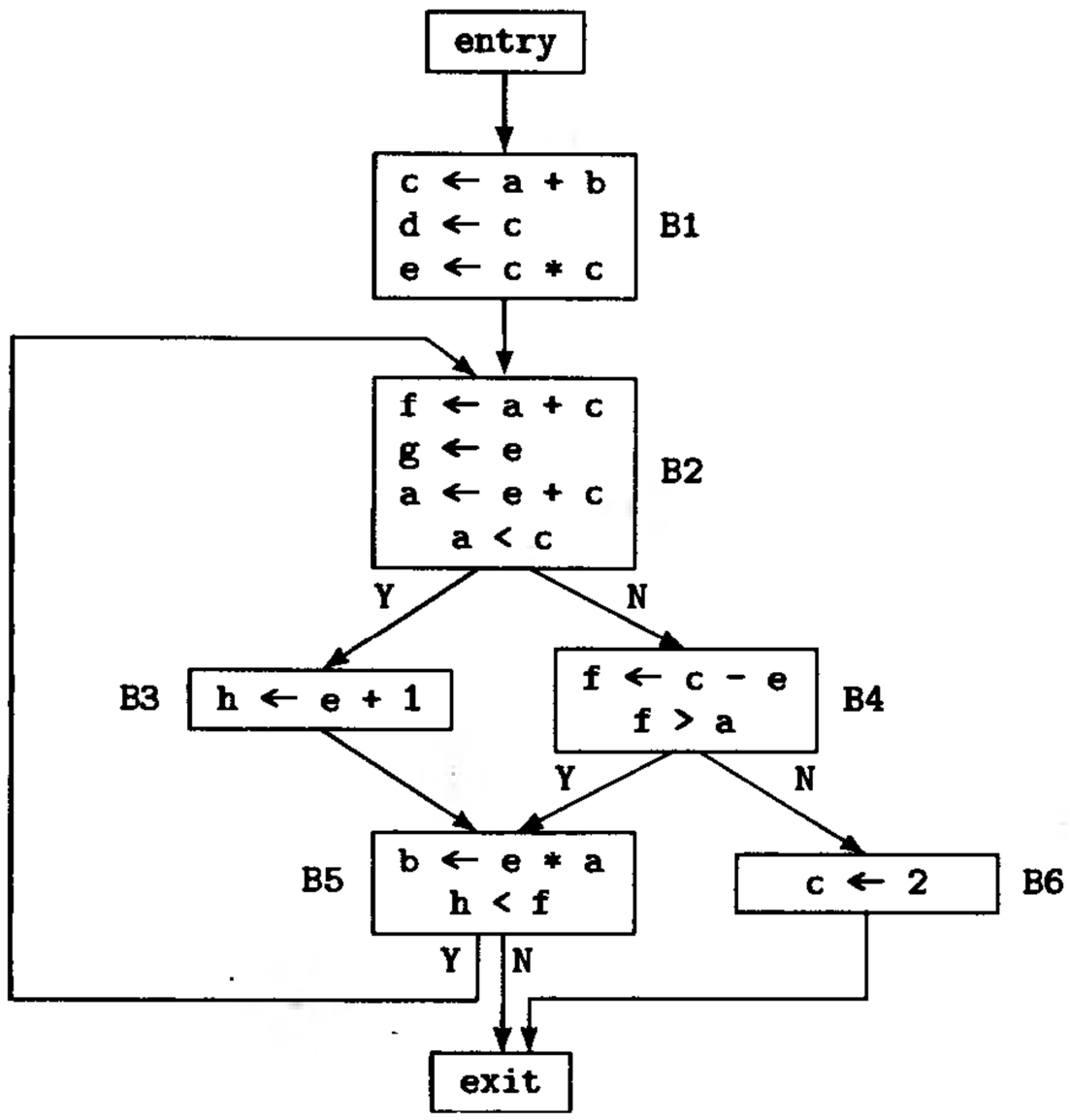


图12-27 对图12-26进行复写传播后得到的流图

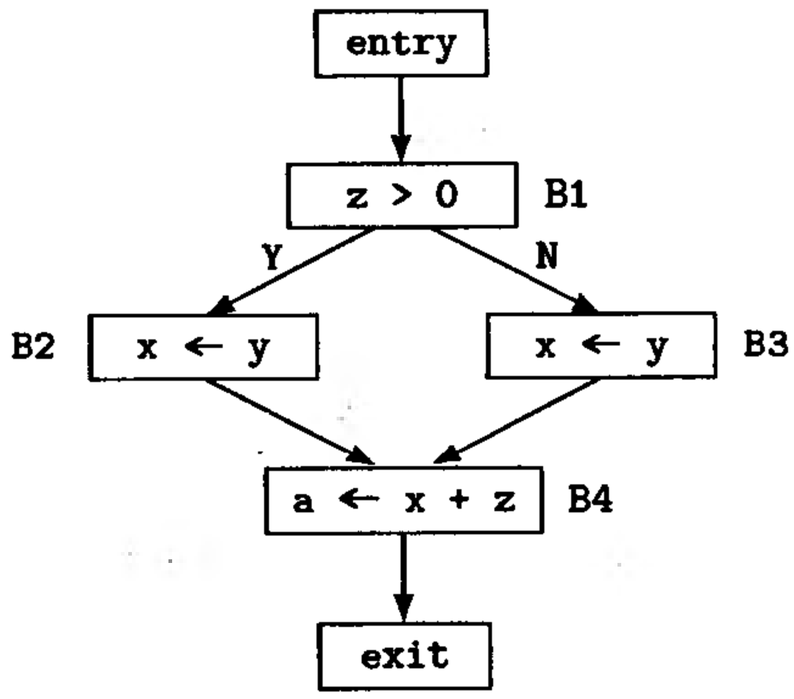


图12-28 全局复写传播没有检测出来的复写赋值

一种可选的方法是，利用作用于赋值的部分冗余删除（见13.3节）或代码提升（见13.5节）方法来将语句x←y的两次出现移到B1中，而它们可以与复写传播在同一优化遍中进行。

12.6 稀有条件常数传播

常数传播（constant propagation）是一种转换，对于给定的关于某个变量x和一个常数c的赋值$x \leftarrow c$，这种转换用c来替代以后出现的x的引用，只要在这期间没有出现另外改变x值的赋值。例如，在图12-29a基本块B1中的赋值b←3将常数3赋给b，并且流图中没有其他对b的赋值。常数传播将此流图转换为图12-29b所示的情形。注意，b的所有出现都已被3替换，但都没有对结果得到的常数表达式进行计算。这是常数表达式计算（参见12.1节）的工作。

对于RISC体系结构，常数传播尤其重要，因为它将小整数移到使用它们的地方。所有RISC机器都提供使用小整数作为操作数的指令（“小”的定义随体系结构不同而变化）。如果知道一个操作数是这种小整数，就可以生成更有效的代码。此外，有些RISC机器（如MIPS）有使用一个寄存器和一个小常数之和的寻址方式，但没有使用两个寄存器之和的寻址方式；将

bbs.theithome.com

362 常数值传播到这种地址结构既节省了寄存器，也节省了指令。更一般的是，常数传播减少了过程需要的寄存器个数，并增加了其他若干优化的效果，这些优化包括常数表达式计算、归纳变量优化（14.1节），以及20.4.2节讨论的基于依赖关系分析的那些转换。

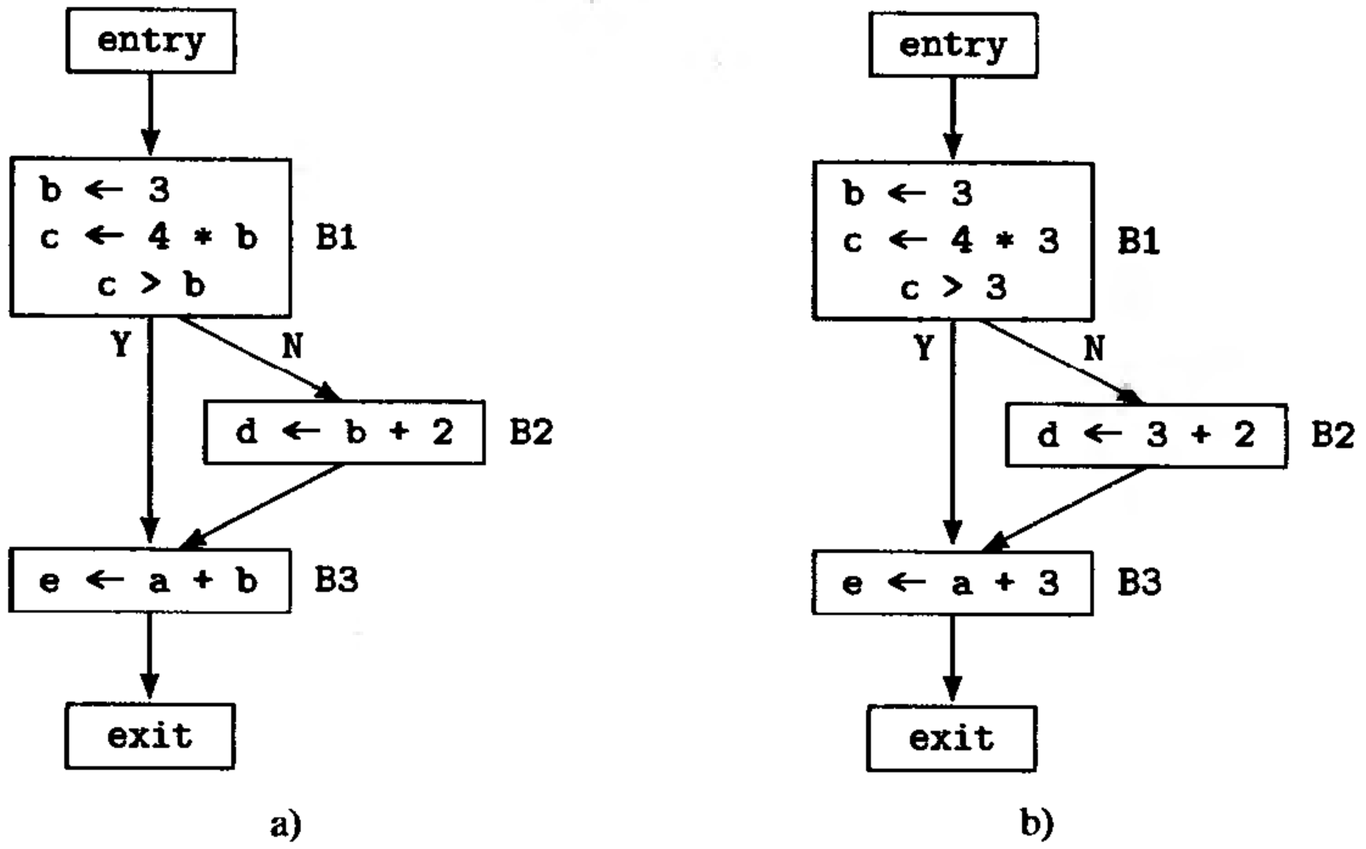


图12-29 a) 要传播的常数赋值的例子，即B1中的b←3，b) 对它做常数传播后的结果

Wegman和Zadeck描述了两种考虑了条件的常数传播方法，一种使用SSA形式，另一种不使用[WegZ91]。我们在这里描述SSA形式的方法，因为它是两种方法中更有效的一种。这种常数传播方法相对传统方法有两个主要的优点：它可以由条件推导出有关信息，并且也更为有效。

为了执行稀有条件常数传播，我们必须首先将流图转换为SSA形式，但有一个额外的附带条件，即每一个结点只含有一种运算或Φ函数。我们使用8.11节描述的迭代的必经边界方法将流图转换为最小SSA形式，并划分基本块为每个结点一条指令，然后对每一个变量引入一条将它的惟一定义连接到它的每一个使用的SSA边。这些工作使得信息的传播可以与程序的控制流无关。

然后，我们利用流图的边和SSA边来传递信息实现程序的符号执行。在处理过程中，仅当结点的执行条件满足时，我们才标志它们是可执行的，并且在每一步我们只处理那些可执行的结点，以及那些其SSA前驱已经被处理过的结点——这就是该方法为什么是符号执行，而不是数据流分析的原因。我们使用图12-30画出的格，其中每一个C_i是一个可能的常数值，包含true和false是为了提供关于条件表达式结果的格值。若ValType表示集合{false, ..., C_{-2}, C_{-1}, C_0, C_1, C_2, ..., true}，则这个格叫做ConstLat。对

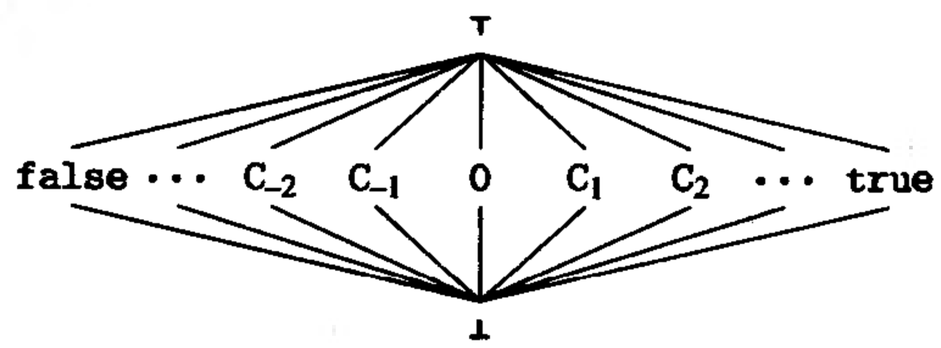


图12-30 常数传播格ConstLat

363 于程序中的每一个变量，我们在流图中定义这个变量的惟一结点的出口处给它相连一个格值。给一个变量赋予值⊤意味着它有一个还未确定的常数值，而⊥则意味着不是常数，或不能确定是常数。我们用⊤初始化所有的变量。

为了包含Φ函数，我们用ICAN扩充MIR的指令表示，如下所示：

*VarName*0 ← Φ(*VarName*1, ···, *VarNamen*)

<kind:phiasgn, left:*VarName*0, vars:[*VarName*1, ..., *VarNamen*]>

bbs.theithome.com

并定义Exp_Kind(phiasgn)＝listexp和Has_Left(phiasgn)＝true。

我们使用两个函数Visit_Phi()和Visit_Inst()来处理流图的结点。其中第一个函数以一种有效的方式执行格值上的Φ函数，后一个函数对原来的语句做同样的事。

执行稀有条件常数传播的代码是图12-31给出的Sparse_Cond_Const()。这个算法使用两个工作表FlowWL和SSAWL，FlowWL存放需要处理的流图边，SSAWL存放需要处理的SSA边。数据结构ExecFlag(*a*, *b*)记录流图边$a \rightarrow b$是否是可执行的。对于每一个SSA形式的变量*v*，存在着一个格点LatCell(*v*)，它记录在定义变量*v*的结点的出口处与这个变量相连的格元素。函数SSASucc(*n*)记录结点*n*的SSA后继边，即，从结点*n*出发的SSA边。附属例程Edge_Count()、Initialize()、Visit_Phi()和Visit_Inst()的代码如图12-32所示。这三个例程用到的另外4个过程如下:

1. Exp(*inst*)当*inst*是一个赋值时抽取*inst*的右端表达式；或者当*inst*是一个测试时抽取*inst*的测试体。

2. Lat_Eval(*inst*)计算*inst*中的表达式，其中变量使用LatCell()中赋予的格值。

3. Edge_Set(*k*, *i*, *val*)返回集合$\{k \rightarrow i\}$，如果*val*是给定格的一个常数元素，否则返回∅。

4. Edge_Count(*b*, *E*)返回*E*中使得*e*@2＝*b*的可执行边*e*的条数。

```
LatCell: Var ⟶ ConstLat
FlowWL, SSAWL: set of (integer × integer)
ExecFlag: (integer × integer) ⟶ boolean
Succ: integer ⟶ set of integer
SSASucc: integer ⟶ (integer × integer)

procedure Sparse_Cond_Const(ninsts,Inst,E,EL,entry)
   ninsts: in integer
   Inst: in array [1··ninsts] of MIRInst
   E: in set of (integer × integer)
   EL: in (integer × integer) ⟶ enum {Y,N}
   entry: in integer
begin
   a, b: integer
   e: integer × integer
   || initialize lattice cells, executable flags,
   || and flow and SSA worklists
   Initialize(ninsts,E,entry)
   while FlowWL ≠ ∅ ∨ SSAWL ≠ ∅ do
      if FlowWL ≠ ∅ then
         e := ♦FlowWL; a := e@1; b := e@2
         FlowWL -= {e}
         || propagate constants along flowgraph edges
         if !ExecFlag(a,b) then
            ExecFlag(a,b) := true
            if Inst[b].kind = phiasgn then
               Visit_Phi(Inst[b])
            elif Edge_Count(b,E) = 1 then
               Visit_Inst(b,Inst[b],EL)
            fi
      fi
   fi
   || propagate constants along SSA edges
   if SSAWL ≠ ∅ then
```

图12-31 基于SSA的稀有条件常数传播算法

bbs.theithome.com

```
            e := ◆SSAWL; a := e@1; b := e@2
            SSAWL -= {e}
            if Inst[b].kind = phiasgn then
               Visit_Phi(Inst[b])
            elif Edge_Count(b,E) ≥ 1 thsn
               Visit_Inst(b,Inst[b],EL)
            fi
         fi
      od
   end     || Sparse_Cond_Const
```

图12-31 （续）

```
procedure Edge_Count(b,E) returns integsr
   b: in integer
   E: in set of (integer × integer)
begin
   || return number of executable flowgraph edges leading to b
   e: integer × integer
   i := 0: integer
   for each e ∈ E do
       if e@2 = b & ExscFlag(e@1,e@2) then
            i += 1
       fi
   od
   return i
snd    || Edge_Count

procedure Initialize(ninsts,E,entry)
    ninsts: in integer
    E: in set of (integer × integer)
    entry: in integsr
begin
    i, m, n: integer
    p: integer × integer
    FlowWL := {m→n ∈ E where m = entry}
    SSAWL := ∅
    for each p ∈ E do
        ExecFlag(p@1,p@2) := false
    od
    for i := 1 to ninsts do
        if Has_Left(Inst[i].kind) then
            LatCell(Inst[i].left) := ⊤
        fi
    od
end    || Initialize
procedure Visit_Phi(inst)
    inst: in MIRInst
begin
    j: integer
    || process φ node
    for j := 1 to |inst.vars| do
        LatCell(inst.left) ⊓= LatCell(inst.vars↓j)
    od
end    || Visit_Phi
```

图12-32 用于稀有条件常数传播的辅助例程

bbs.theithome.com

```
procedure Visit_Inst(k,inst,EL)
   k: in integer
   inst: in MIRInst
   EL: in (integer × integer) ⟶ enum {Y,N}
begin
   i: integer
   v: Var
   || process non-φ  node
   val := Lat_Eval(inst): ConstLat
   if Has_Left(inst.kind) & val ≠ LatCell(inst.left) then
      LatCell(inst.left) ⊓= val
      SSAWL ∪= SSASucc(k)
   fi
   case Exp_Kind(inst.kind) of
binexp, unexp:
      if val = ⊤ then
         for each i ∈ Succ(k) do
            FlowWL ∪= {k→i}
         od
      elif val ≠ ⊥ then
         if |Succ(k)| = 2 then
            for each i ∈ Succ(k) do
               if (val & EL(k,i) = Y) ∨ (!val & EL(k,i) = N) then
                  FlowWL ∪= {k→i}
               fi
            od
         elif |Succ(k)| = 1 then
            FlowWL ∪= {k→♦Succ(k)}
         fi
      fi
default:
   esac
end    || Visit_Inst
```

图12-32 （续）

我们取图12-33中的程序作为一个简单的例子，这个程序已经是每个结点一条指令的最小SSA形式。SSA边是B1→B3、B2→B3、B4→B6和B5→B6，所以，SSASucc(B4)＝{B4→B5}。算法一开始设置FlowWL＝{entry→B1}，SSAWL＝∅，所有ExecFlag()的值为false，所有LatCell()的值为⊤。然后它从FlowWL中删除entry→B1，设置ExecFlag(entry，B1)＝true，并调用Visit_Inst(B1，“a_1←2”)。Visit_Inst()计算这个格中的表达式2，设置LatCell(a_1)＝2和SSAWL＝{B1→B3}。主例程然后设置FlowWL＝{B1→B2}。因为SSAWL现在不为空，主例程从SSAWL中删除B1→B3并调用Visit_Inst(B3，“a_1<b_1”)，如此等等。结果是这些格点被设置成LatCell(a_1)＝2，LatCell(b_1)＝3，LatCell(c_1)＝4，LatCell(c_2)＝⊤，LatCell(c_3)＝4。注意LatCell(c_2)决不会改变，因为算法判定从B3到B5的边不会执行。这个信息可用来从流图中删除基本块B3、B5和B6。

364
i
367

作为第二个例子，考虑图12-34中的程序。图12-35是它的最小SSA转换形式，其中，每一个基本块只有一条指令。SSA边是B1→B4、B2→B3、B3→B5、B3→B7、B4→B5、B5→B8、B5→B11、B6→B7、B6→B8、B6→B9、B6→B10、B7→B10、B7→B4、B9→B11和B12→B3，所以有SSASucc（B5）＝{B5→B8，B5→B11}。初值与前面的例子相同。格点的最终值如下:

bbs.theithome.com

超星公司制作 请尊重作者版权

LatCell(a_1) = 3
LatCell(d_1) = 2
LatCell(d_3) = 2
LatCell(a_3) = 3
LatCell(f_1) = 5
LatCell(g_1) = 5
LatCell(a_2) = 3
LatCell(f_2) = 6
LatCell(f_3) = ⊥
LatCell(d_2) = 2

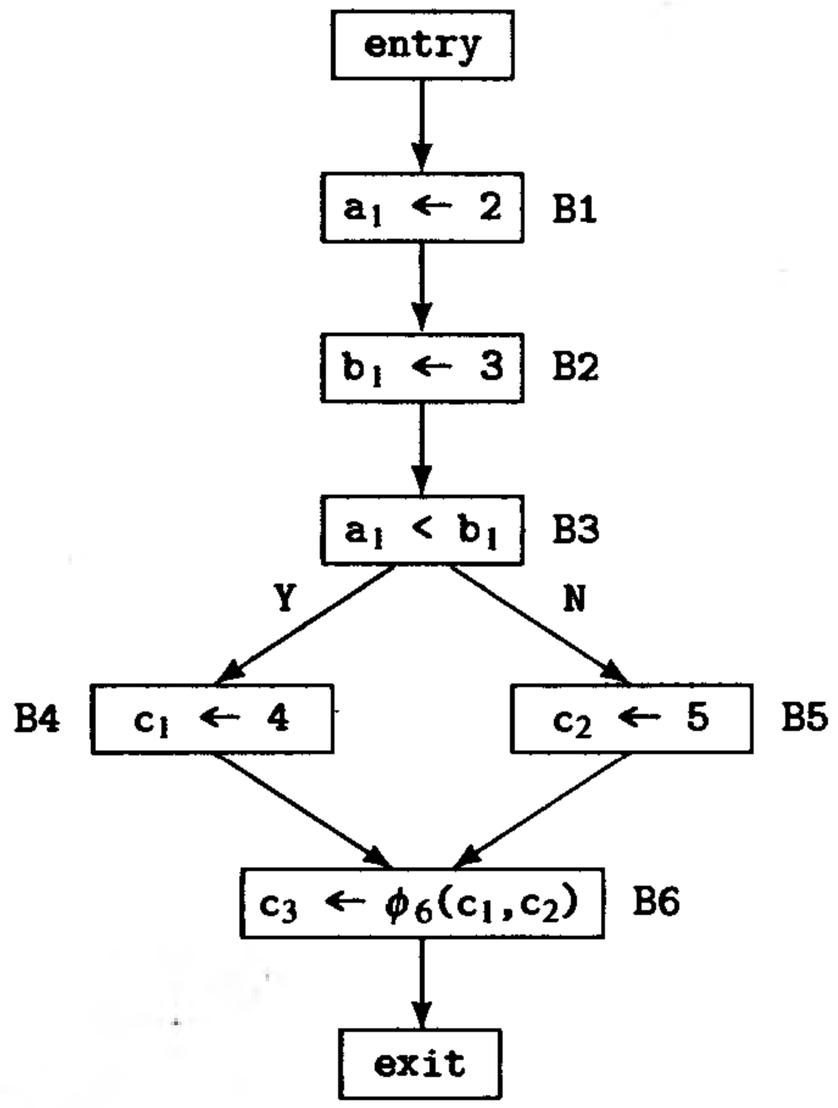


图12-33 用于稀有条件常数传播的简单例子

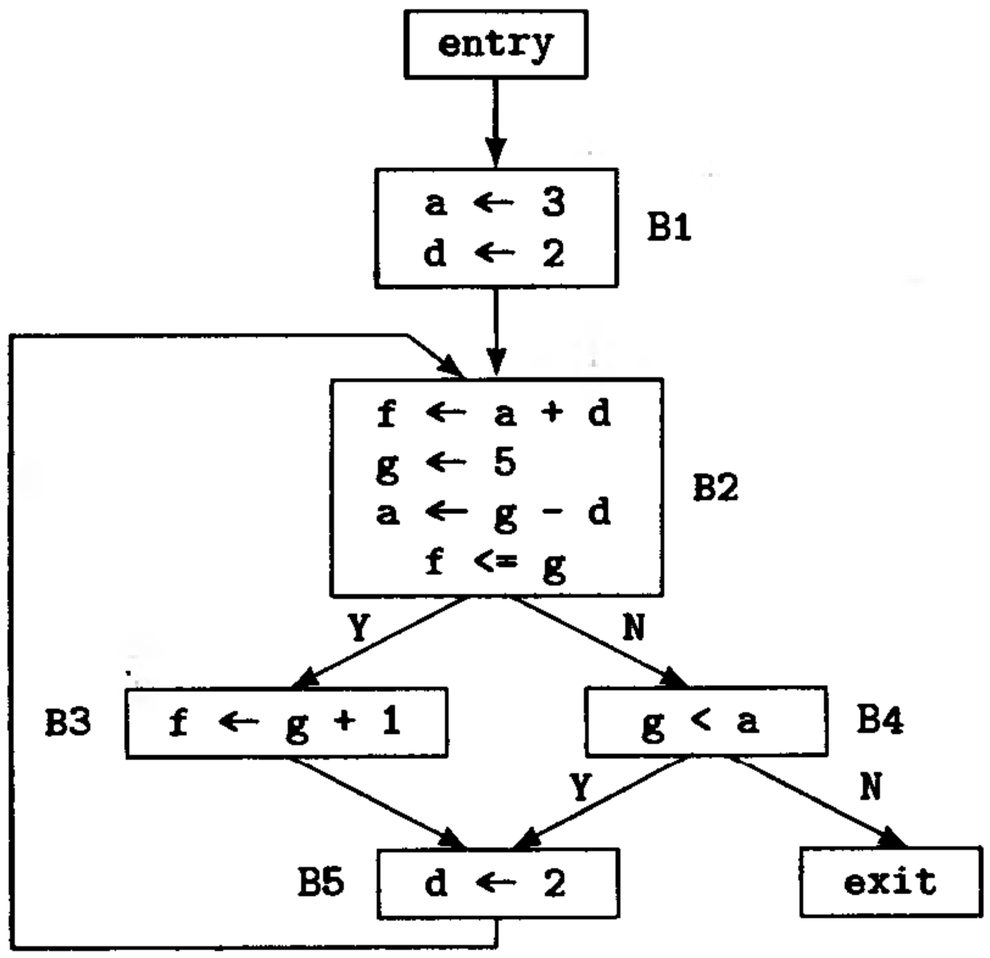


图12-34 用于稀有条件常数传播的另一个例子

结果代码（在用常数值替代了那些变量，并且删除了不可到达代码之后）如图12-36所示。

bbs.theithome.com

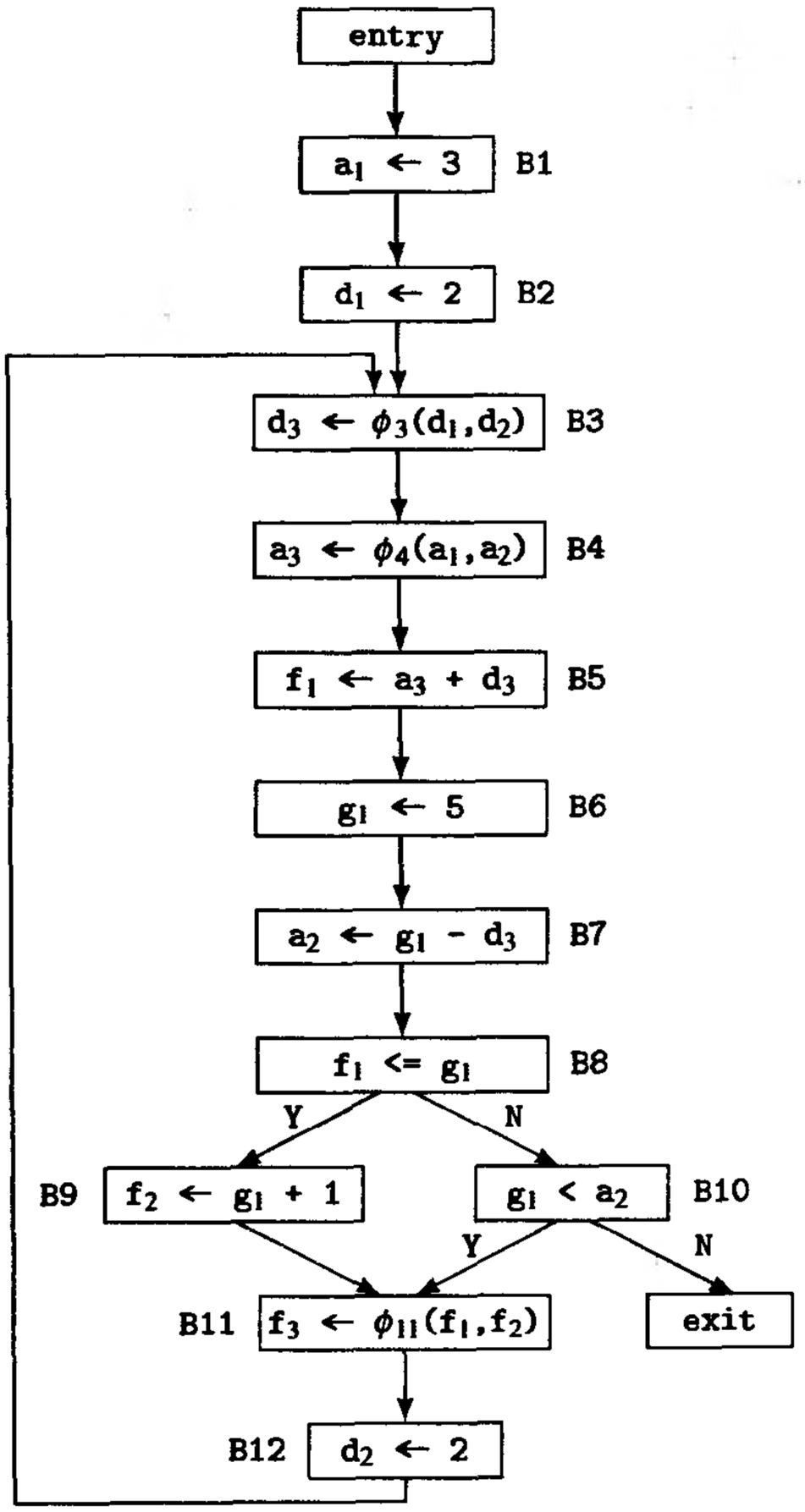


图12-35 图12-34中程序的最小SSA形式，其中每一个基本块一条指令

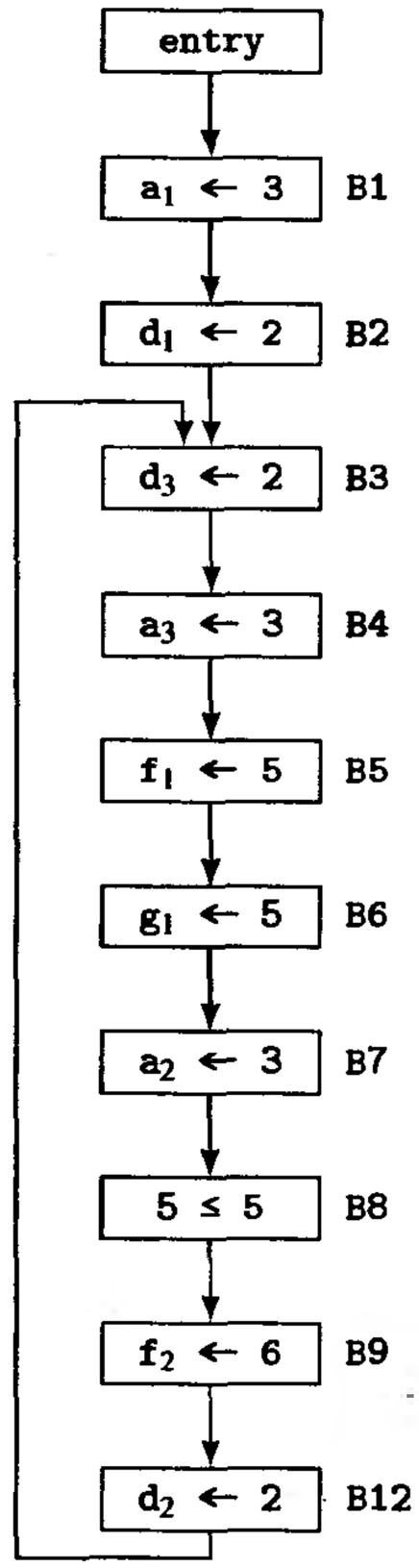


图12-36 对图12-35中的例程做稀有条件常数传播后的结果

我们用一个语句而不是一个基本块作为结点的惟一原因是为了方便。很容易修改这个算法使其使用基本块作为结点——例如，它仅需要我们用基本块编号和块内的位置来标识变量的定义点。

稀有条件常数传播的时间复杂度与流图的边数和SSA边数有关，因为每一个变量的值在这种格中只能降低两次，因此，计算时间复杂度是$O(|E|+|SSA|)$，其中SSA是SSA边集合，在最坏的情况下，这个值是结点个数的平方，但在实际中它几乎总是线性的。

368
~
370

12.7 小结

在这一章我们开始讨论各种具体的优化，包括常数表达式求值（常数折叠）、聚合量标量替代、代数化简和重结合、值编号、复写传播，以及稀有条件常数传播。前面三种优化独立于数据流分析，即它们不需要考虑是否已经执行过数据流分析。后面三种优化由于效果和正确性的原因需要依赖于数据流分析。

bbs.theithome.com

超星公司制作 请尊重作者版权

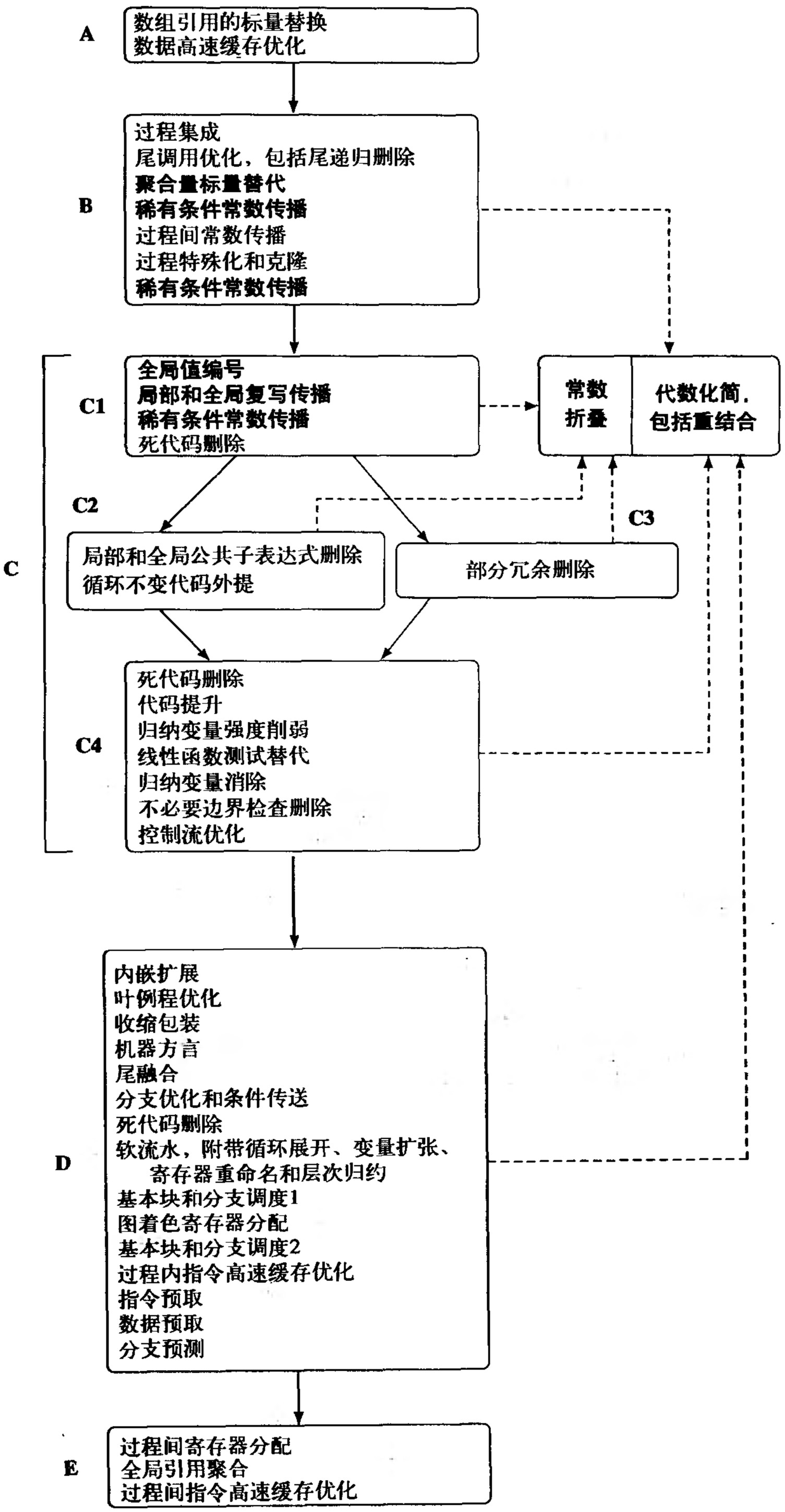


图12-37 优化顺序。本章讨论的那些优化用黑体字标明

bbs.theithome.com

超星公司制作请尊重作者版权

我们将这些优化的处理和它们的意义概括如下:

1. 常数折叠最好构造成子程序，以便任何可以从中获得好处的优化器都可以调用它。关键是编译器的数据类型模型和参与常数折叠的运算要与目标机体系结构一致。

2. 聚合量标量替代最好在编译处理的早期进行，因为它将那些通常不适合优化的结构转换成适合优化的结构。

3. 代数化简和重结合与常数折叠一样，最好构造成可随需要调用的子程序。对于一大类程序而言，地址表达式的代数化简和其他作用于地址的优化，如循环不变代码外提，属于最重要的优化。

4. 值编号优化有时容易与另外两种优化，即公共子表达式删除和常数传播相混淆；值编号的全局形式也可能与循环不变代码外提和部分冗余删除相混淆。这几种优化都是不同的，值编号的功能是标识形式上等价的表达式，并删除那些等价的表达式的冗余计算，从而进一步减少后面那些优化要处理的代码量。

5. 复写传播用变量的复写值替换这些变量的使用，再一次地减少代码量。

6. 稀有条件常数传播用变量的常数值替换已判别出具有常数值的变量的使用。它不同于其他需要数据流分析的优化，因为它执行更复杂的一种分析，即符号执行，它在分析中利用值为常数的条件来确定路径在分析中是否会执行。

全局值传播和稀有条件常数传播两种优化都针对SSA形式的流图，并且从这种形式中获益相当大——主要是，前者由于使用了它而成为全局的，后者由于使用它而比传统的全局常数传播更强大。

我们将本章讨论的这些优化放置在整个优化过程中，其优化顺序如图12-37所示，这些优化都用黑体字标明。

371 ↓ 372

12.8 进一步阅读

关于浮点算术的ANSI/IEEE标准见[IEEE85]。Goldberg对该标准的介绍和对它的概述，以及有关的讨论见[Gold91]。[Farn88]和[Gold91]中讨论了浮点值的常数折叠和代数化简有关的问题。关于实现了聚合量标量替代的一个编译器的例子参见[Much91]。

最初给出针对基本块的值编号公式表示的是[CocS69]。为适应扩展基本块对它的有关修改是在[AusH82]中找到的，并且将它扩充到整个过程的两种方法是在[ReiL86]和[AlpW88]中找到的。在创建基本块的DAG中使用值编号的描述可参见[AhoS86]。全局值编号处理中使用的划分算法由Aho、Hopcroft和Ullman [AhoH74]等人开发。Briggs、Cooper和Simpson关于基于散列的和全局的两种值编号方法的比较见[BriC94]和[Simp96]，而Cooper和Simpson的针对强连通分量的值编号方法见[CooS95b]和[Simp96]。

Wegman和Zadeck的稀有条件常数传播的介绍见[WegZ91]。关于此算法中使用的有关符号执行的概述在[MucJ81]中给出。

12.9 练习

12.1 一种称为循环剥离（peeling）的转换通过在循环之前插入循环体的一个副本而从该循环中删除第一个迭代。在循环剥离之后对过程体执行常数传播和常数折叠很容易产生能够再次应用这三种转换的代码。例如，下面(a) 中的MIR代码是第一步执行循环剥离后的代码，(b) 是常数传播和常数折叠转换后的代码。

bbs.theithome.com

```
        m ← 1
        i ← 1
L1:     m ← m * i
        i ← i + 1
        if i ≤ 10 goto L1
```

(a)

```
        m ← 1
        i ← 2
L1:     m ← m * i
        i ← i + 1
        if i ≤ 10 goto L1
```

(b)

第二步循环剥离后得到下面的代码(c)，最终得到代码(d)。

```
        m ← 2
        i ← 3
L1:     m ← m * i
        i ← i + 1
        if i ≤ 10 goto L1
```

(c)

```
        m ← 3628800
        i ← 11
```

(d)

假定不存在转向L1的其他分支。我们怎样才能识别这种情况？实际中它们发生的可能性怎样？

12.2 保证常数折叠正确性的关键是，编译时的计算环境要与运行时的环境一致，或者编译器模拟运行时的环境应产生与运行时环境一致的结果。具体地，假设我们正在Intel 386处理器上编译一个要运行于PowerPC处理器的程序，Intel 386处理器在寄存器中只能表示80位内部浮点值（参见21.4.1节），而PowerPC处理器只有单精度和双精度格式（参见21.2.1节）。这对浮点常数折叠有什么影响？

12.3 (a) 写出一个做聚合量标量替代的ICAN程序。(b) 什么样的情形很可能不能从这种替代中获益？(c) 我们如何使得该算法不对这种情况进行处理？

12.4 (a) 用ICAN写出一个规范器或树转换器，它接收一棵树和一组树转换规则，并对树应用这些转换规则，直到它们不能再应用为止。假设这些树是用如下定义的ICAN数据类型Node表示的：

```
Operator = enum {add,sub,mul}
Content = record {kind: enum {var,const},
                  val: Var ∪ Const}
Node = record {opr: Operator,
               lt,rt: Content ∪ Node}
```

(b) 证明当给出的是图12-6表示的转换时，对所有的输入树，你的规范器都会停止。

12.5 图12-13的Value_Number()定义中，在case语句中是否应当有一个关于listexp情形的选择？如果应当有，关于这个选择的代码是怎样的？

ADV 12.6 如12.4.2节末尾指出的，[AlpW88] 建议对要分析的程序进行结构化分析，并使用为控制流结构而设计的Φ函数，以便能够确定关于控制流的重合性。概略地叙述你将如何扩展全局值编号算法来包含这种想法。

ADV 12.7 能否修改全局复写传播以识别诸如图12-28的情形？如果能够，怎样修改？如果不能，为什么？

12.8 修改(a) 局部复写传播算法和(b) 全局复写传播算法，以使它们可工作于扩展基本块。

373 ~ 375 ADV 12.9 能用类似于稀有条件常数传播的形式来表示复写传播吗？如果能这样做，我们能得到什么益处？如果不能，为什么？

12.10 修改稀有条件常数传播算法，以适应结点是基本块而不是单个语句的情形。

bbs.theithome.com

超星公司制作 请尊重作者版权

第13章 冗余删除

本章涉及的所有优化都与消除冗余计算有关，并且都需要进行数据流分析。这些优化可以在中级中间代码（例如MIR）或低级中间代码（例如LIR）上进行。

第一种优化是公共子表达式删除，它寻找那些在一条已知路径上至少执行两次以上的计算，并删除第2次以及其后出现的那些计算。为了定位冗余的计算，并保证实施冗余删除能对程序性能有所改善，这种优化需要进行数据流分析。

第二种优化是循环不变代码外提，它寻找那种在循环的每一次迭代中总是产生相同结果的运算，并将这种计算移到循环之外。这种循环不变代码尽管可以独立于数据流分析来确定，但一般基于ud链。这种优化几乎总是能改善性能，且常常有非常显著的效果，这在很大程度上是因为它发现和移出的往往是循环不变地址计算，以及与访问数组元素有关的计算。

第三种优化是部分冗余删除，它移动那些至少是部分冗余的计算（即在流图的某条路径上多于一次以上的相同计算）到它们的最优计算点，并完全删除那些冗余的计算。它包含了公共子表达式删除、循环不变代码外提，甚至更多。

最后一种优化是代码提升，它寻找那些从程序某点开始的在所有路径上都被执行的计算，并在那一点上将它们合并为单个计算。这种优化需要进行数据流分析（它有一个有点好笑的名字——“非常忙表达式”数据流分析），它可减少程序占用的空间，但对性能很少有影响。

我们决定在本章既介绍公共子表达式删除和循环不变代码外提，也介绍部分冗余删除，因为两种方法有基本相同的效率和类似的作用。在几年之前我们还只能介绍前一种方法，而仅仅提及后一种方法，因为用公式表示部分冗余删除需要非常复杂和昂贵的双向数据流分析。这里介绍的现代公式消除了这个问题，并且还提供了一种思考和形式化表述其他优化的框架。我们可以相当肯定地断言，即使目前还没有使用这种方法，但在不久的将来，它会被选择作为冗余删除的方法。

13.1 公共子表达式删除

程序中一个表达式的一次出现是*公共子表达式*（common subexpression）[⊖]，如果存在着该表达式的另一次出现，在执行顺序上它的计算总是先于这个表达式的计算，并且在这两个计算之间该表达式的操作数没有发生改变。图13-1a基本块B3中的表达式a+2是公共子表达式的一个例子，因为在基本块B1中有一个先于它执行的相同表达式，并且a的值在这期间没有改变。*公共子表达式删除*（common-subexpression elimination）是一种转换，它删除公共子表达式的重复计算，并用保存的值来替代它们。图13-1b展示了对a）中代码进行转换的结果。注意，如这个例子所示，我们不能简单地用b替代基本块B3中a+2的计算，因为如果B2被执行则改变了b的值。

回忆12.4节开始那个例子的有关说明，值编号和公共子表达式删除是不同的。

⊖ 传统的术语是“子表达式”，而不是“表达式”，但是这个定义适应任何表达式，而不仅仅是其他表达式的子表达式。

bbs.theithome.com

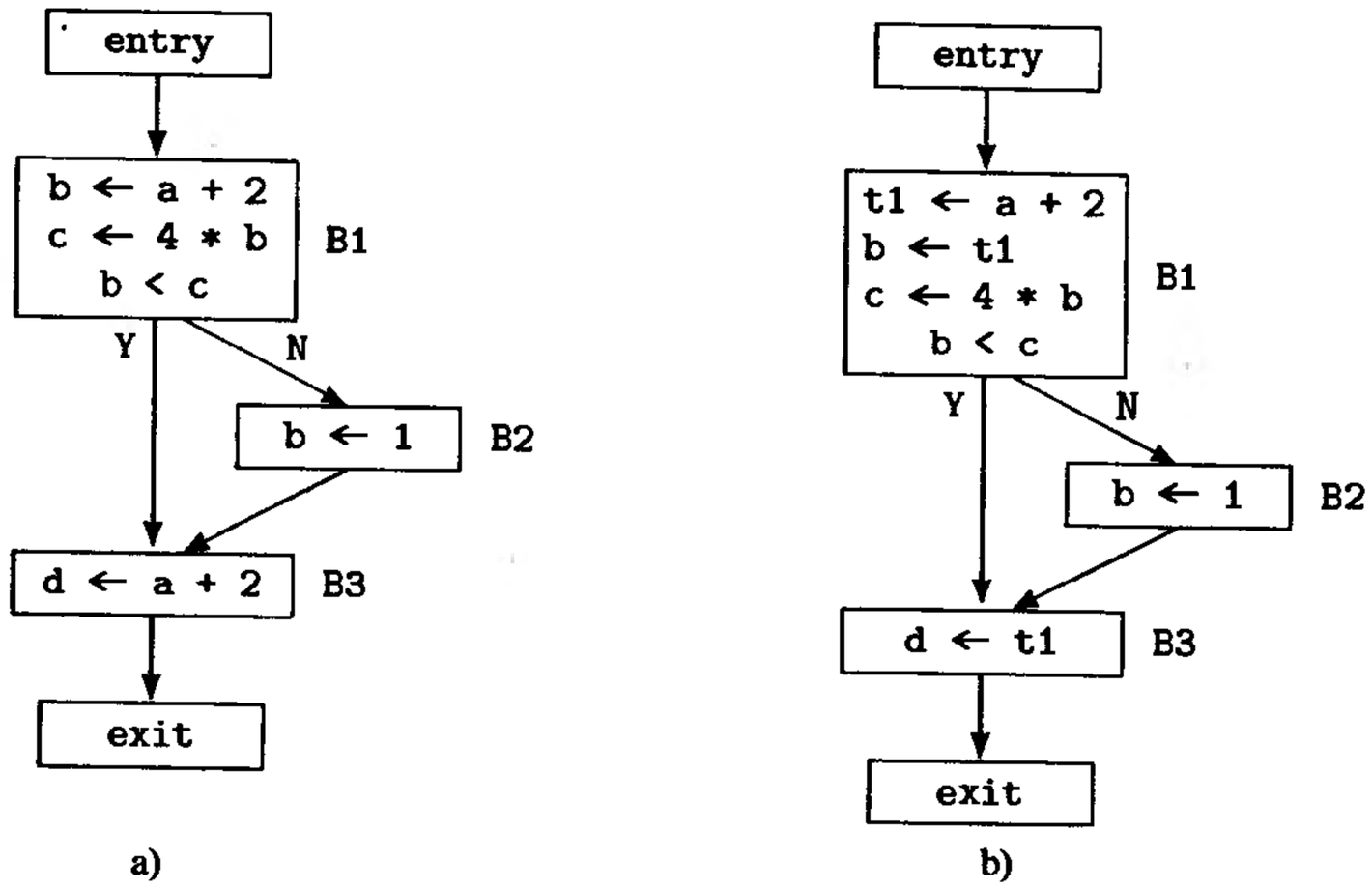


图13-1 a) 一个公共子表达式的例子，即其中的a+2，b) 对它进行公共子表达式删除后的结果

另外要注意的是，公共子表达式删除并不总是值得的。在这个例子中，重复计算a+2的代价可能较小（尤其是当a和d都分配在寄存器中，并且加一个小常数到寄存器的操作只需一拍时），而在B1到B3之间分配另外的寄存器来存放t1的值，甚至将它保存到存储器，然后重新取出它，这样做的代价较大。实际上还有一些更为复杂的原因可能导致不值得做公共子表达式删除，这些原因与充分发挥超标量流水线功能，或为了从向量或并行机获得更好的性能有关。因此，我们在13.1.3节讨论公共子表达式删除的一种逆转换，叫做向前替代。

优化处理常常将公共子表达式删除分成两遍，一遍是局部的，在每一个基本块内进行；另一遍是全局的，跨整个流图进行。这种划分不是实质性的，因为全局公共子表达式删除能够捕获局部形式能处理的所有公共子表达式，甚至更多，但局部形式可以非常轻易地在构造基本块中间代码的同时进行，并且可以使生成的中间代码较少。因此，我们在下面两小节分别描述这
378 两种形式。

13.1.1 局部公共子表达式删除

如前面提到的，局部公共子表达式删除作用于单个基本块，并且可以随基本块的中间代码生成而进行，也可以在之后进行。为了方便起见，我们假定已经生成了MIR指令，并且它们存放在Block[m][1..ninsts[m]]中。我们的方法实质上是记录可用表达式（available expression），即那些迄今已在基本块中计算过，并且从那以后其操作数没有被改变的表达式，以及每一个这种表达式在基本块中的计算位置。我们的这种记录表示是*AEB*，一个形如<*pos*, *opd*1, *opr*, *opd*2, *tmp*>的五元组集合，其中*pos*是表达式在基本块中被计算的位置；*opd*1, *opr*和*opd*2组成一个二元表达式；*tmp*是nil或是一个临时变量。

为了做局部公共子表达式删除，我们依次扫描基本块中的指令，往*AEB*中添加或从中删除适当的登记项，插入保存表达式之值至临时变量的指令，并将指令中对表达式的使用修改为对临时变量的使用。对于每一条在位置*i*的指令*inst*，我们判别它计算的是否是一个二元表达式，然后根据情况执行下述步骤：

1. 将*inst*的操作数和操作符与*AEB*中的五元式进行比较。如果找到匹配的，比如说，<*pos*, *opd*1, *opr*, *opd*2, *tmp*>，则检查*tmp*是否是nil。如果是，则，

(a) 生成一个新的临时变量*ti*，并用它替代所标识的三元组中的nil，

bbs.theithome.com

版权所有 侵权必究 请勿擅自制作印刷或电子版

(b) 紧靠位置*pos*的指令之前插入指令“*ti*←*opd*1 *opr opd*2”，并且 379

(c) 用*ti*替代位置*pos*和*i*两处指令中的这个表达式。

如果找到了一个与*tmp* = *ti*的匹配，其中*ti*≠nil，我们用*ti*替代*inst*中的表达式。如果在*AEB*中没有找到与*inst*的表达式相匹配的，则插入一个关于该表达式的五元组到*AEB*，其中*tmp* = nil。

2. 如果当前指令有结果变量，检查它的结果变量是否在*AEB*的五元组中作为操作数出现。如果是，从*AEB*中删除所有这种五元组。

实现这种处理方法的例程Local_CSE()如图13-2所示，它使用了如下4个另外的例程:

1. Renumber(*AEB*, *pos*)根据需要对*AEB*中五元组的第一个元素重新编号，以反映已插入的指令。

2. insert_before()插入一条指令到基本块，并对块中的指令重新编号，使之适应新插入的这条指令（参见4.8节）。

3. Commutative(*opr*)返回true，如果操作符*opr*是可交换的；否则返回false。

4. new_temp()返回一个新的临时变量作为其值。

```
AEBinExp = integer × Operand × Operator × Operand × Var

procedure Local_CSE(m,ninsts,Block)
   m: in integer
   ninsts: inout array [··] of integer
   Block: inout array [··] of array [··] of MIRInst
begin
   AEB := ∅, Tmp: set of AEBinExp
   aeb: AEBinExp
   inst: MIRInst
   i, pos: integer
   ti: Var
   found: boolean
   i := 1
   while i ≤ ninsts[m] do
      inst := Block[m][i]
      found := false
      case Exp_Kind(inst.kind) of
binexp:  Tmp := AEB
         while Tmp ≠ ∅ do
            aeb := ♦Tmp; Tmp -= {aeb}
            || match current instruction's expression against those
            || in AEB, including commutativity
            if inst.opr = aeb@3 & ((Commutative(aeb@3)
                  & inst.opd1 = aeb@4 & inst.opd2 = aeb@2)
               ∨ (inst.opd1 = aeb@2 & inst.opd2 = aeb@4)) then
            pos := aeb@1
            found := true
            || if no variable in tuple, create a new temporary and
            || insert an instruction evaluating the expression
            || and assigning it to the temporary
            if aeb@5 = nil then
               ti := new_tmp( )
               AEB := (AEB - {aeb})
                  ∪ {⟨aeb@1,aeb@2,aeb@3,aeb@4,ti⟩}
               insert_before(m,pos,ninsts,Block,
```

图13-2 做局部公共子表达式删除的例程

bbs.theithome.com

```
                <kind:binasgn,left:ti,opd1:aeb@2,
                opr:aeb@3,opd2:aeb@4>)
            Renumber(AEB,pos)
            pos += 1
            i += 1
            || replace instruction at position pos
            || by one that copies the temporary
            Block[m][pos] := <kind:valasgn,left:Block[m][pos].left,
                opd:<kind:var,val:ti>>
        else
            ti := aeb@5
        fi
        || replace current instruction by one that copies
        || the temporary ti
        Block[m][i] := <kind:valasgn,left:inst.left,
            opd:<kind:var,val:ti>>
    fi
od
if !found then
    || insert new tuple
    AEB ∪= {<i,inst.opd1,inst.opr,inst.opd2,nil>}
fi
|| remove all tuples that use the variable assigned to by
|| the current instruction
Tmp := AEB
while Tmp ≠ ∅ do
    aeb := ♦Tmp; Tmp -= {aeb}
    if inst.left = aeb@2 ∨ inst.left = aeb@4 then
        AEB -= {aeb}
    fi
od
default:
    esac
    i += 1
od
end    || Local_CSE
```

图13-2 （续）

作为此算法的一个例子，考虑图13-3中的代码，它表示的是在产生它们时不执行公共子表达式删除的情况下生成的代码形式。一开始，AEB=∅且i=1。因为第1条指令有一个binexp（二元表达式），且AEB为空，因此我们将五元组〈1，a，+，b，nil〉加入AEB，并置i=2。第2条指令也包含一个binexp；AEB中没有五元组与它相同，因此，我们插入〈2，m，&，n，nil〉到AEB并置i=3。AEB现在为

```
AEB={<1, a, +, b, nil>,
     <2, m, &, n, nil>}
```

同样的情形发生在第3条指令，导致

```
AEB={<1, a, +, b, nil>,
     <2, m, &, n, nil>,
     <3, b, +, d, nil>}
```

且i=4。下面我们在位置4遇到f←a+b，并发现这个表达式与

位置	指令
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← -b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← -b
11	if m & n goto L2

图13-3 例基本块在局部公共子表达式删除之前

bbs.theithome.com

AEB中第1个五元组相匹配。于是我们插入t1到那个五元组，替代其中的nil，在位置1之前生成指令t1←a+b，对AEB中五元组的位置重新编号，用c←t1替代原来在位置1而现在位于位置2的指令，设置i=5，并用f←t1替代在位置5的指令。

代码的当前状态如图13-4所示，并且AEB的值是

```
AEB={<1, a, +, b, t1>,
     <3, m, &, n, nil>,
     <4, b, +, d, nil>}
```

接下来我们遇到的指令是第6行的g←-b，对它不需要做什么。下一条指令是第7行的h←b+a，我们识别出它的右端与AEB中的一个五元组相匹配，于是用h←t1替代指令7。这产生了图13-5所示的代码。接着我们在第8行找到的指令是a←j+a，此时我们第一次从AEB中删除一个五元组：结果变量a与AEB中第1个五元组的第一个操作数相匹配，因此删除此五元组，由此得到

380 ~ 382

```
AEB={<3, m, &, n, nil>,
     <4, b, +, d, nil>}
```

注意，在同一个迭代中我们插入了关于j+a的一个三元组之后，又因为结果变量与两个操作数之一相匹配而删除了它。

位置	指令
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← -b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← -b
12	if m & n goto L2

图13-4 我们的基本块例子在删除第1个局部公共子表达式（第1、2和5行）之后

位置	指令
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← -b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← -b
12	if m & n goto L2

图13-5 我们的基本块例子在删除第2个局部公共子表达式（第7行）之后

接下来识别出的公共子表达式是图13-5中第9行的表达式m&n，这导致得到图13-6中的代码，并且AEB的值变成

```
AEB={<3, m, &, n, t2>,
     <5, b, +, d, nil>}
```

最后，第12行的表达式b+d和第13行的表达式m&n被识别为局部公共子表达式（注意，我们假定m & n产生一个整数值）。AEB最后的值是

```
AEB={<3, m, &, n, t2>,
     <5, b, +, d, t3>}
```

并且最终代码如图13-7所示。

这段代码原来的形式有11条指令、12个变量，并要执行9个二元运算，而最后的形式有14条指令、15个变量和4个要执行的运算。假设所有变量都在寄存器中，并且每一个寄存器与寄存器的操作只需要一拍，就像所有RISC和少数CISC机器的情形一样，则原来的形式要更好一

bbs.theithome.com

超星公司制作 请尊重作者版权

些，因为它的指令条数较少，使用的寄存器也较少。另一方面，如果某些变量是在存储器中，或者执行这些冗余的操作需要的时钟周期多于一拍，则优化的结果更好一些。因此，这种优化是否实际能改善基本块的性能取决于代码本身和执行它的机器的情况。

位置	指令
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← -b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← -b
13	if m & n goto L2

图13-6 我们的基本块例子在删除第3个局部公共子表达式之后

位置	指令
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← -b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← -b
14	if t2 goto L2

图13-7 我们的基本块例子在删除最后两个局部公共子表达式之后

通过散列每一个三元式中的操作数和操作符，使得当散列值匹配时才对实际的操作数和操作符进行比较，可以快速地实现这个局部算法。散列函数的选择应当能快速地计算，并且是关于操作数对称的，以便高效地处理可交换性，因为可交换操作符的出现比不可交换操作符更频繁。

这个算法和后面的全局公共子表达式删除算法都能通过12.3节讨论的重结合转换而进一步改善，尤其是对那些寻址算术，因为它们是最频繁出现的公共子表达式。

13.1.2 全局公共子表达式删除

如前面指出的，全局公共子表达式删除将过程的流图作为其工作对象。它解一种称为*可用表达式*（available expressions）的数据流问题，这个问题在8.3节曾简单地讨论过，现在我们详细地考察它。称表达式*exp*在一个基本块的入口是*可用的*（available），如果从入口基本块到这个基本块的每一条控制路径上都存在着对*exp*的计值，且该表达式没有由于它的一个或多个操作数赋予了一个新值而被杀死。

我们设计出关于可用表达式数据流分析的两个版本。第一个版本简单地告诉我们在每个基本块的入口有哪些表达式是可用的。第二个版本告诉我们这些表达式在何处被求值，即在哪个基本块的哪个位置被求值。我们之所以给出两种版本是因为在优化研究中，对于使用数据流方法来确定计算点是否是最好的方法存在着不同的观点，例如，关于这样做不是最好的一种意见，可参见[AhoS86]，P.634。

在确定什么样的表达式是可用的过程中，我们使用*EVAL*(*i*)表示在基本块*i*中被计值，并且在基本块出口处仍然可用的表达式集合，*KILL*(*i*)表示由基本块*i*杀死的表达式集合。为了计算*EVAL*(*i*)，我们从基本块开始扫描到基本块末尾，收集在块内计算的表达式，并删除那些其操作数在基本块中后来又被赋予了新值的表达式。对于一个形如a←a+b的赋值表达式，它的左端变量也作为右端的操作数出现，我们不创建一个可用变量，因为赋值发生于表达式计值之后。

383 ~ 385

bbs.theithome.com

对于图13-3中的基本块例子，*EVAL*()集合是{m&n, b+d}。

这个基本块也计算了表达式a+b，但此表达式随后被a←j+a的赋值所杀死，因此它不在这个基本块的*EVAL*()集合中。*KILL*(*i*)是那种在其他基本块中计值，但它的一个或多个操作数在基本块*i*中被赋值的所有表达式的集合。为了给出*KILL*()集合的一个例子，需要有一个完整的流图[⊖]，因此，我们考虑图13-8中的流图。这个基本块的*EVAL*(*i*)和*KILL*(*i*)集合如下:

EVAL(entry)	= ∅	*KILL*(entry)	= ∅
EVAL(B1)	= {a+b,a*c,d*d}	*KILL*(B1)	= {c*2,c>d,a*c,d*d,i+1,i>10}
EVAL(B2)	= {a+b,c>d}	*KILL*(B2)	= {a*c,c*2}
EVAL(B3)	= {a*c}	*KILL*(B3)	= ∅
EVAL(B4)	= {d*d}	*KILL*(B4)	= ∅
EVAL(B5)	= {i>10}	*KILL*(B5)	= ∅
EVAL(exit)	= ∅	*KILL*(exit)	= ∅

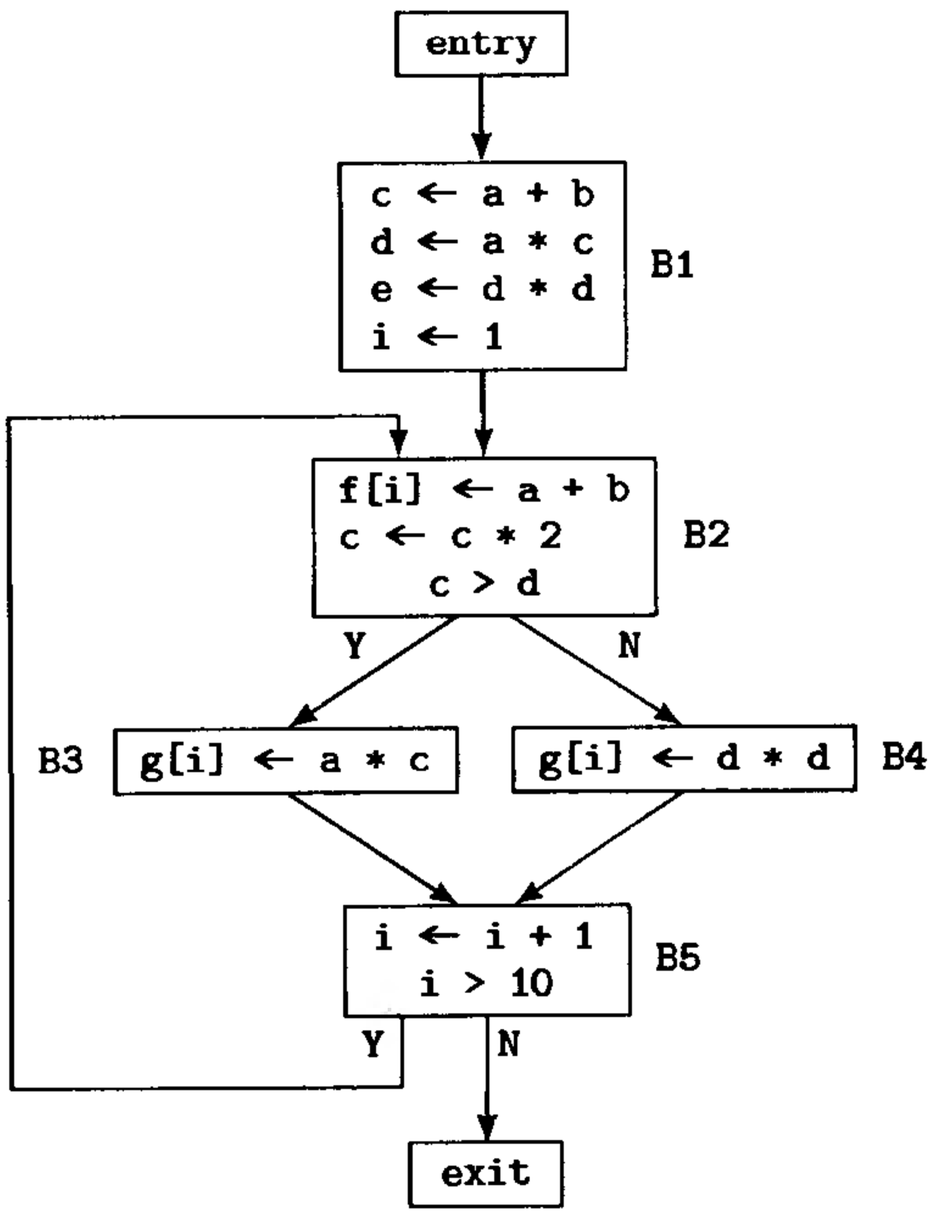


图13-8 用于全局公共子表达式删除的流图例子

现在，数据流分析的方程组可以构造如下。这是一个向前流问题。我们分别使用*AEin*(*i*)和*AEout*(*i*)表示在基本块*i*入口和出口可用的表达式集合。一个表达式在基本块入口是可用的，如果它在该基本块的所有前驱基本块的出口处是可用的，因此路径合并运算是集合的交。一个表达式在基本块出口是可用的，如果它在基本块内被计值，并且随后没有被杀死，或者它在基本块入口是可用的，并且在基本块内没有被杀死。因此，数据流方程组是

$$AEin(i) = \bigcap_{j \in Pred(i)} AEout(j)$$

$$AEout(i) = EVAL(i) \cup (AEin(i) - KILL(i))$$

⊖ 实际上并不完全是这样。我们可以用这个基本块中被赋过值的变量集合来表示它。但这样做会相当低效，因为它会是变量集合，而*EVAL*()是表达式集合。这样的话，利用位向量运算来实现数据流分析即使再好也仍很笨拙。

bbs.theithome.com

在解这个数据流方程时，对所有的基本块i，我们初始化$AEin(i)=U_{exp}$，其中U_{exp}可以取所有表达式的全集，或不难证明，取

$$U_{exp}=\bigcup_{i} EVAL(i)$$

就足够了[⊖]。

对于图13-8的例子，我们采用

U_{exp}={a+b, a*c, d*d, c>d, i>10}

工作表迭代的第一步产生

AEin(entry)= ∅

第二步迭代产生

AEin(B1)= ∅

接下来我们计算

AEin(B2) = {a+b,a*c,d*d}
AEin(B3) = {a+b,d*d,c>d}
AEin(B4) = {a+b,d*d,c>d}
AEin(B5) = {a+b,d*d,c>d}
AEin(exit) = {a+b,d*d,c>d,i>10}

并且再继续迭代不会产生进一步的变化。

386 ~ 387 下面我们讲述如何利用*AEin*()数据流函数执行全局公共子表达式删除。为了简单起见，我们假定已经进行过局部公共子表达式删除，因此一个表达式只有在基本块中的第一个计值是全局公共子表达式删除的候选对象。我们的处理过程如下：

1. 确定*exp*在基本块*i*中的第一个计值的位置。

2. 从这个第一次出现向后搜索，确定*exp*是否有任何操作数已在此基本块中前面某处被赋值。如果有，*exp*的这个出现不是全局公共子表达式；适当前进到另一个表达式或另一个基本块。

3. 一旦找到*exp*在基本块*i*中的第一次出现，并判断出它是一个全局公共子表达式，便在流图中向后搜索寻找*exp*的这样一种出现：它们出现的形式为$v \leftarrow exp$，且使得表达式*exp*属于*AEin*(*i*)。这是一些在它们各自基本块中最后出现的*exp*；它们中的每一个都一定没有改变地流向基本块*i*的入口；并且从entry基本块到基本块*i*的每一条通路都一定至少包含它们中的一个。

4. 选择一个新的临时变量*tj*。用*tj*替代在基本块*i*中使用*exp*的第一条指令*inst*中的此表达式，并用

tj ← *exp*

Replace(*inst*, *exp*, *tj*)

替代第3步标识出的使用*exp*的每一条指令，其中*Replace*(*inst*, *exp*, *tj*)返回已用*tj*替代了*exp*的指令*inst*。

图13-9给出了一个名为Global_CSE()的实现这个算法的例程。例程Find_Sources()定位全局公共子表达式的源出现点，并返回由基本块编号和该基本块内的指令编号组成的偶对集合。例程insert_after()与4.8节的描述相同。例程Replace_Exp(Block, *i*, *j*, *tj*)用一条使用了opd:<kind:var, val:*tj*>的指令替代基本块Block[*i*][*j*]中的这条指令，即它用对应的valasgn替代binasgn，用valif替代binif，用valtrap替代bintrap。

⊖ 如果我们没有一个无进入边的特殊的entry结点，就会需要初始化*AEin*(entry) = ∅，因为在这种流图的入口没有表达式是可用的，而进入初始结点的边会导致在此方程的解中入口基本块的*AEin*()非空。

bbs.theithome.com

```
BinExp = Operand × Operator × Operand

procedure Global_CSE(nblocks,ninsts,Block,AEin)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array[1··nblocks] of array [··] of MIRInst
   AEin: in integer ⟶ BinExp
begin
   i, j, k: integer
   l, tj: Var
   S: set of (integer × integer)
   s: integer × integer
   aexp: BinExp
   for i := 1 to nblocks do
      for each aexp ∈ AEin(i) do
         j := 1
         while j ≤ ninsts[i] do
            case Exp_Kind(Block[i][j].kind) of
binexp:        if aexp@1 = Block[i][j].opd1
                  & aexp@2 = Block[i][j].opr
                  & aexp@3 = Block[i][j].opd2 then
                  for k := j-1 by -1 to 1 do
                     if Has_Left(Block[i][k].kind)
                        & (Block[i][k].left = aexp@1.val
                        ∨ Block[i][k].left = aexp@3.val) then
                        goto L1
                     fi
                  od
                  S := Find_Sources(aexp,nblocks,ninsts,Block)
                  tj := new_tmp( )
                  Replace_Exp(Block,i,j,tj)
                  for each s ∈ S do
                     l := Block[s@1][s@2].left
                     Block[s@1][s@2].left := tj
                     insert_after(s@1,s@2,ninsts,Block,
                        ⟨kind:valasgn,left:l,
                        opd:⟨kind:var,val:tj⟩⟩)
                     j += 1
                  od
               fi
default:    esac
L1:         j += 1
         od
      od
   od
end    || Global_CSE
```

图13-9 实现我们的第一种全局公共子表达式删除处理的例程

选择适当的数据结构可以使得这个算法的运行时间为线性的。

在图13-8的例子中，满足全局公共子表达式标准的第一个表达式是基本块B2中的a+b。从它向后搜索，我们找到B1中的指令c←a+b，用

```
t1 ← a + b
c ← t1
```

替代它，并用f[i]←t1替代基本块B2中的那条指令。类似地，基本块B4中出现的d*d满足该

bbs.theithome.com

超星公司制作 请尊重作者版权

标准，因此，我们用g[i]←t2替代它的出现所在的这条指令，并用

```
t2 ←d * d
e ← t2
```

替代基本块B1中定义该表达式的赋值。结果如图13-10所示。

通过将程序中出现的表达式集合映射到从0开始的一列连续整数，其中每个整数依次与位向量中的一个位置相对应，便可以有效地实现全局CSE算法。集合运算直接转化为对位向量的按位逻辑运算。

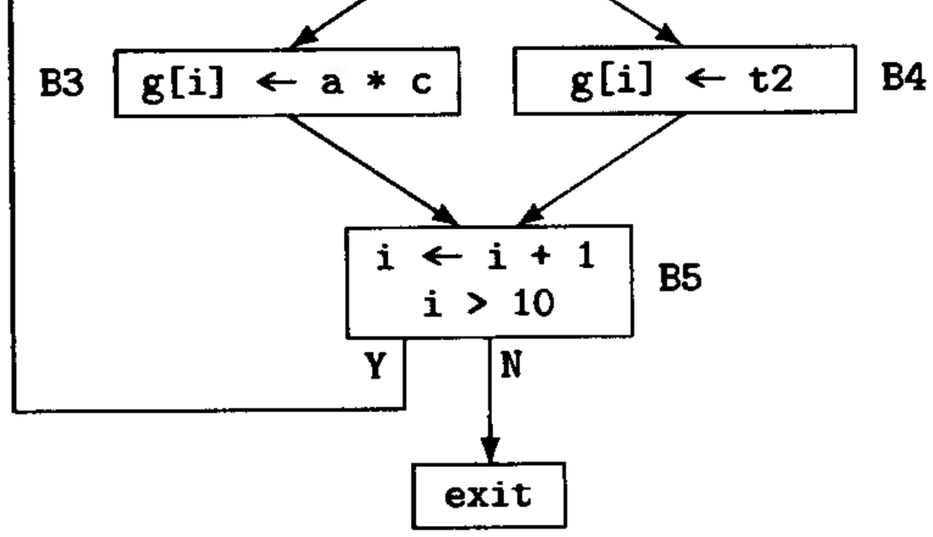


图13-10 图13-8的流图例子在公共子表达式删除之后的流图

实现全局公共子表达式删除的第二种方法既判定可用表达式，同时对于每一个潜在的公共子表达式，也确定它的出现点——即基本块和基本块中的位置。这是通过修改数据流分析，使之能处理各个表达式的出现来实现的。我们定义*EVALP*(*i*)是由三元组 <*exp*, *i*, *pos*> 组成的集合，其中*exp*是表达式，*i*是基本块的名字，*pos*是一个表示基本块中位置的整数，并且*exp*在基本块*i*中位置*pos*处被计值，且在基本块出口仍然可用。我们定义*KILLP*(*i*)是由三元组 <*exp*, *blk*, *pos*> 组成的集合，其中*exp*满足*KILL*(*i*)的条件，*blk*和*pos*是值域为定义*exp*的基本块和位置的偶对，但不包括基本块*i*。最后，我们定义

388
~
390

$$AEinP(i) = \bigcap_{j \in Pred(i)} AEoutP(j)$$

$$AEoutP(i) = EVALP(i) \cup (AEinP(i) - KILLP(i))$$

初始值与前面数据流方程组的初始值类似：对于所有的i，$AEinP(i) = U$，其中U被适当地修改成包含位置信息。

现在，对于图13-8的例子，我们有下述*EVALP*()和*KILLP*()集合：

```
EVALP(entry)   = ∅
EVALP(B1)      = {⟨a+b,B1,1⟩,⟨a*c,B1,2⟩,⟨d*d,B1,3⟩}
EVALP(B2)      = {⟨a+b,B2,1⟩,⟨c>d,B2,3⟩}
EVALP(B3)      = {⟨a*c,B3,1⟩}
EVALP(B4)      = {⟨d*d,B4,1⟩}
EVALP(B5)      = {⟨i>10,B5,2⟩}
EVALP(exit)    = ∅

KILLP(entry)   = ∅
KILLP(B1)      = {⟨c*2,B2,2⟩,⟨c>d,B2,3⟩,⟨a*c,B3,1⟩,⟨d*d,B4,1⟩,
                  ⟨i+1,B5,1⟩,⟨i>10,b5,2⟩}
KILLP(B2)      = {⟨a*c,B1,2⟩,⟨a*c,B3,1⟩,⟨c*2,B2,2⟩}
KILLP(B3)      = ∅
KILLP(B4)      = ∅
KILLP(B5)      = ∅
KILLP(exit)    = ∅
```

并且执行数据流分析的结果如下：

bbs.theithome.com

超星公司制作 请尊重作者版权

$AEinP$(entry) = ∅
$AEinP$(B1) = ∅
$AEinP$(B2) = {⟨a+b,B1,1⟩,⟨d*d,B1,3⟩}
$AEinP$(B3) = {⟨a+b,B1,1⟩,⟨d*d,B1,3⟩,⟨c>d,B2,3⟩}
$AEinP$(B4) = {⟨a+b,B1,1⟩,⟨d*d,B1,3⟩,⟨c>d,B2,3⟩}
$AEinP$(B5) = {⟨a+b,B1,1⟩,⟨d*d,B1,3⟩,⟨c>d,B2,3⟩}
$AEinP$(exit) = {⟨a+b,B1,1⟩,⟨d*d,B1,3⟩,⟨c>d,B2,3⟩,⟨i>10,B5,2⟩}

于是已标识出了每一个公共子表达式的可用表达式实例，因而不需要再在流程图中搜索它们。我们这个例子流图的转换过程同前面一样，除了算法更简单以外，没有其他变化，具体过程如下。

对于每一个基本块i和在基本块i中计值，且对某个blk和pos，有⟨exp, blk, pos⟩ ∈ $AEinP(i)$的表达式exp，进行如下处理：

1. 确定exp在基本块i中的第一次计值出现的位置。

2. 从exp的这个第一次出现向后搜索，确定在此基本块中exp是否有已在前面被赋值的任何操作数。如果有，exp的这个出现不是全局公共子表达式；根据情况前进到另一个表达式或另一个基本块。

3. 一旦找到exp在基本块i中的第一次出现，并判断出它是一个全局公共子表达式，则令⟨exp, blk_1, pos_1⟩，…，⟨exp, blk_n, pos_n⟩是$AEinP(i)$中以exp作为其表达式部分的那些元素。它们 391
中的每一个是对exp求值的一条指令$inst$。

4. 选择一个新的临时变量tj。用tj替代基本块i中已标识出的exp的出现，并用

$tj \leftarrow exp$

Replace(*inst*, *exp*, *tj*)

替代每一条已标识出的在基本块blk_k位置pos_k计算exp的指令$inst$。

图13-11给出了一个实现第二个算法的名为Global_CSE_Pos()的例程。它使用的例程Coalesce_Sets()在同一个图中给出。例程insert_after()和Replace()与图13-9中的相同。

```
BinExp = Operand × Operator × Operand
BinExpIntPair = BinExp × integer × integer
BinExpIntPairSet = BinExp × set of (integer × integer)

procedure Global_CSE_Pos(nblocks,ninsts,Block,AEinP)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array[1··nblocks] of array [··] of MIRInst
   AEinP: in integer ⟶ BinExpIntPairSet
begin
   i, j, k: integer
   tj, v: Var
   s: integer × integer
   inst: MIRInst
   aexp: BinExpIntPairSet
   AEinPS: integer ⟶ BinExpIntPairSet
   AEinPS := Coalesce_Sets(nblocks,AEinP)
   for i := 1 to nblocks do
      for each aexp ∈ AEinPS(i) do
         j := 1
         while j ≤ ninsts[i] do
            inst := Block[i][j]
```

图13-11 实现我们的第二种全局公共子表达式删除方法的例程

bbs.theithome.com

```
                case Exp_Kind(inst.kind) of
binexp:            if aexp@1@1 = inst.opd1 & aexp@1@2 = inst.opr
                      & aexp@1@3 = inst.opd2 then
                      for k := j-1 by -1 to 1 do
                         if Has_Left(inst.kind)
                            & (inst.left = aexp@1@1.val
                            V inst.left = aexp@1@3.val) then
                            goto L1
                         fi
                      od
                      tj := new_tmp( )
                      Replace_Exp(Block,i,j,tj)
                   for each s ∈ aexp@2 do
                      v := Block[s@1][s@2].left
                      Block[s@1][s@2].left := tj
                      insert_after(s@1,s@2,⟨kind:valasgn,left:v,
                         opd:⟨kind:var,val:tj⟩⟩)
                      j += 1
                   od
                fi
default:     esac
L1:          j += 1
          od
       od
    od
end    || Global_CSE_Pos

procedure Coalesce_Sets(n,AEinP) returns integer → BinExpIntPairSet
    n: in integer
    AEinP: in integer → set of BinExpIntPair
begin
    AEinPS, Tmp: integer → BinExpIntPairSet
    i: integer
    aexp: set of BinExpIntPair
    a, b: BinExpIntPairSet
    change: boolean
    for i := 1 to n do
       AEinPS(i) := ∅
       for each aexp ∈ AEinP(i) do
          AEinPS(i) ∪= {⟨aexp@1,{⟨aexp@2,aexp@3⟩}⟩}
       od
       Tmp(i) := AEinPS(i)
       repeat
          change := false
          for each a ∈ AEinPS(i) do
             for each b ∈ AEinPS(i) do
                if a ≠ b & a@1 = b@1 then
                   Tmp(i) := (Tmp(i) - {a,b}) ∪ {⟨a@1,a@2 ∪ b@2⟩}
                   change := true
                fi
             od
          od
       until !change
    od
    return Tmp
end    || Coalesce_Sets
```

图13-11 （续）

bbs.theithome.com

通过将程序中有关表达式出现的三元组集合映射到一列从0开始的连续整数，其中每一个整数依次与位向量中的一个位置相对应，便可以有效地实现全局CSE算法。集合运算直接转化为对位向量的按位逻辑运算。

通过将单独一条指令作为流图的结点，可以使局部和全局公共子表达式删除合并为一遍。尽管这种方法是可行的，但一般并不希望这样做，因为它会使得数据流分析的代价比上面介绍的分开的方法要高出很多。相反倒是可以使用更大的单位：局部和全局分析与优化都可以在扩展基本块上进行。

另外，局部和全局公共子表达式删除可以重复地进行，如果将它们与复写传播和/或常数传播结合起来，能获得额外的好处。作为一个例子，考虑图13-12a中的流图。对它进行局部公共子表达式删除使得基本块B1变成如图13-12b所示。全局公共子表达式删除用临时变量替代了基本块B2和B4中a+b的出现，产生了图13-13a的流图。接着局部复写传播用t3替代基本块B1中t1和t2的使用，得到图13-13b。最后，全局复写传播用t3替代基本块B2、B3和B4中c和d的出现，由此得到图13-14的结果代码。注意，死代码删除现在已能够删除基本块B1中的赋值t2←t3和t1←t3。

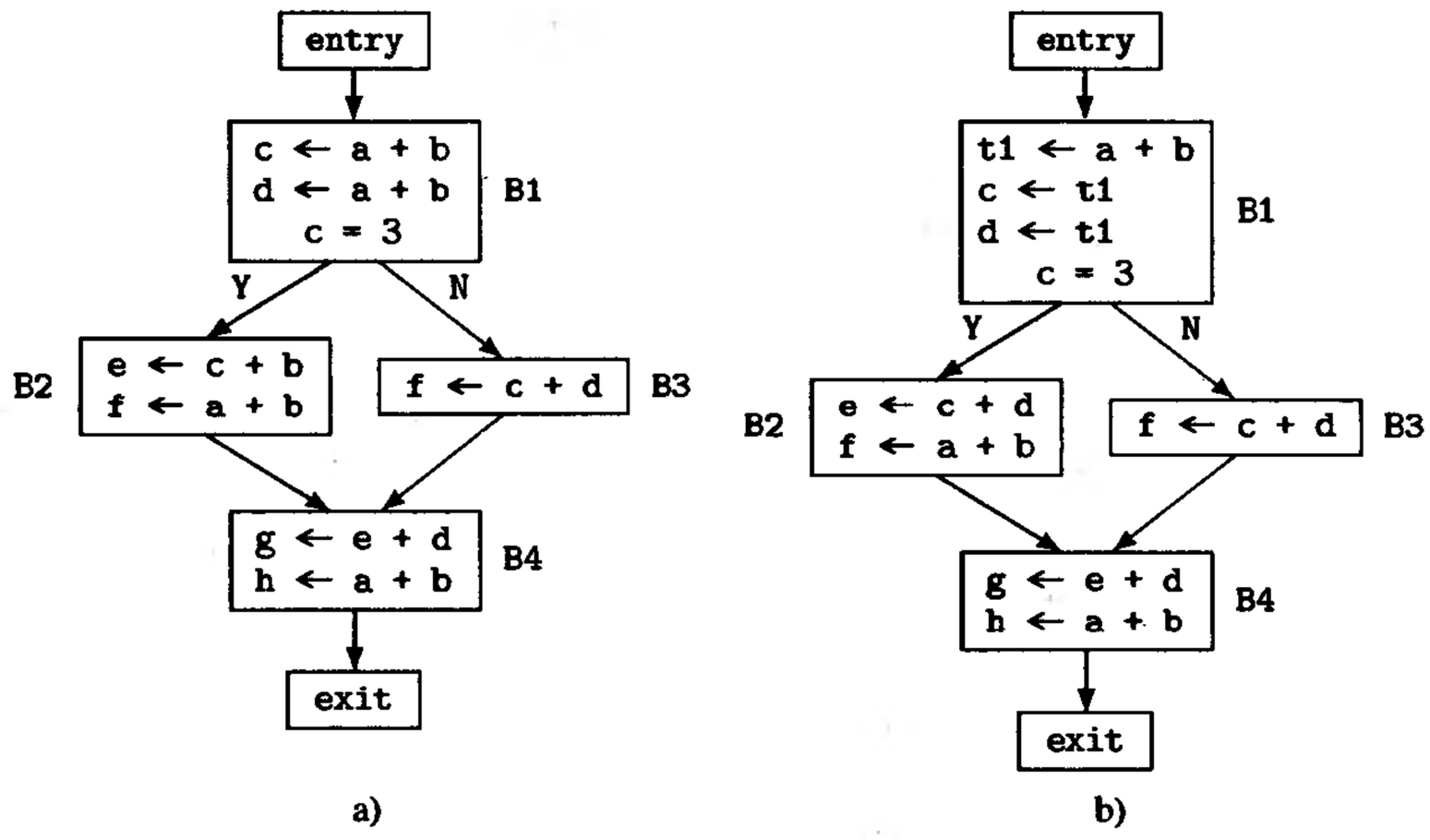


图13-12 公共子表达式删除与复写传播的结合：a) 原来的流图，b) 局部公共子表达式删除后的结果

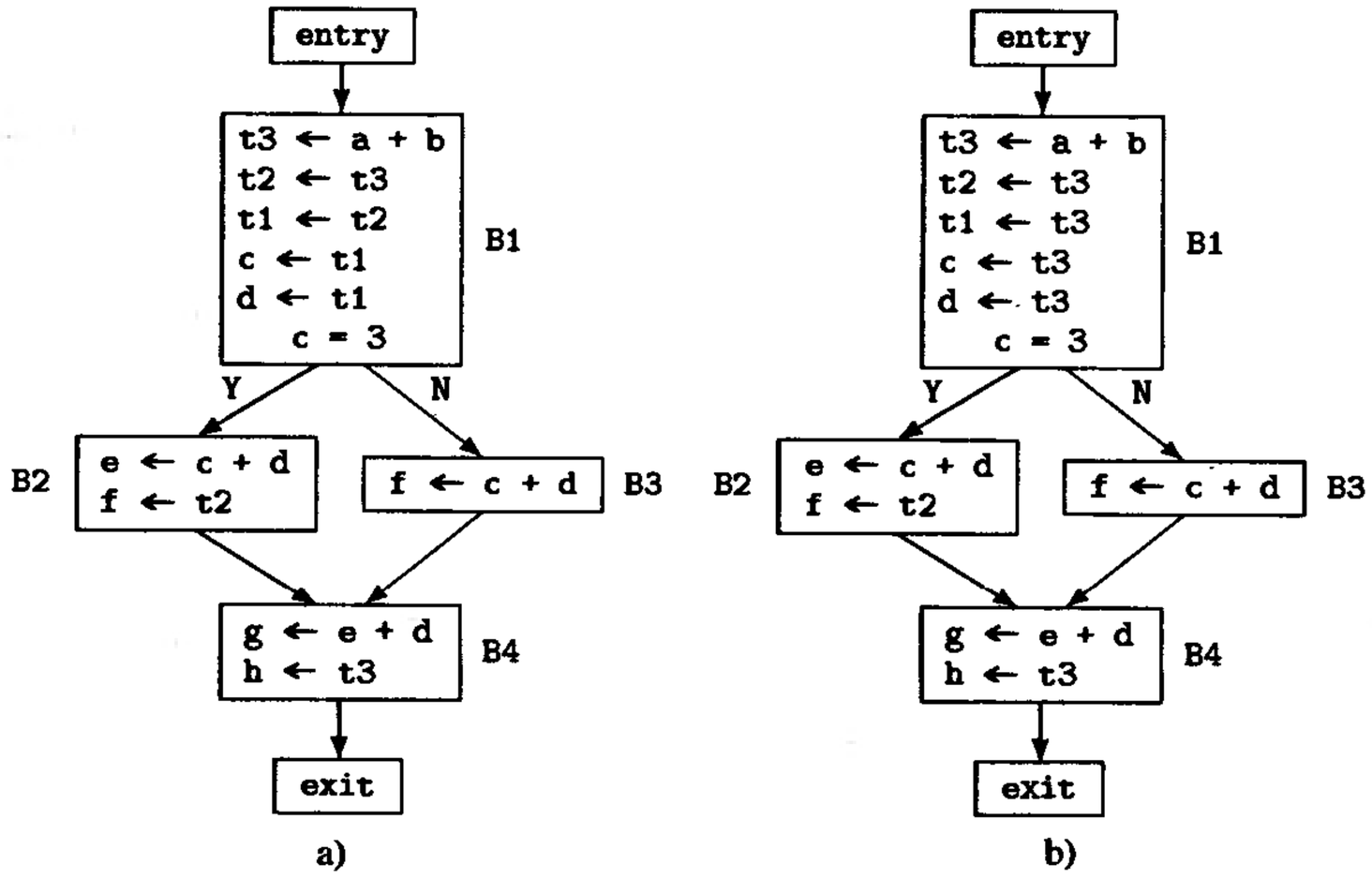


图13-13 公共子表达式删除与复写传播的结合（图13-12的继续）：
a) 全局公共子表达式删除后，b) 局部复写传播后

bbs.theithome.com

392 ~ 394

另一方面，对于任意n，我们不难构造一个通过重复这些优化n次而获益的例子。尽管有这种可能，但在实际中几乎从不出现。

13.1.3 向前替代

向前替代（forward substitution）是公共子表达式删除的逆转换。与用一个复写操作替代一个表达式计算相反，它用重新计算该表达式替代一个复写。尽管看起来公共子表达式删除总是值得的——因为它似乎减少了要执行的运算次数——但实际并不一定是这样；而且即使是这样，它也不一定肯定有利。具体地，不希望这样做的一个最简单的情形是，为了保存一个表达式的值而导致一个寄存器被长时间占用，从而减少了其他用途的可用寄存器个数时；或者——而这可能更坏，当这个值由于没有可用的寄存器而被放到内存中，并且需要重新装入它时。例如，考虑图13-15中的代码。a) 中的代码在从B1到B2的边上需要2个寄存器（用于a和b），而b) 中的代码，它是全局公共子表达式删除的结果，需要3个寄存器（用于a、b和t1）。

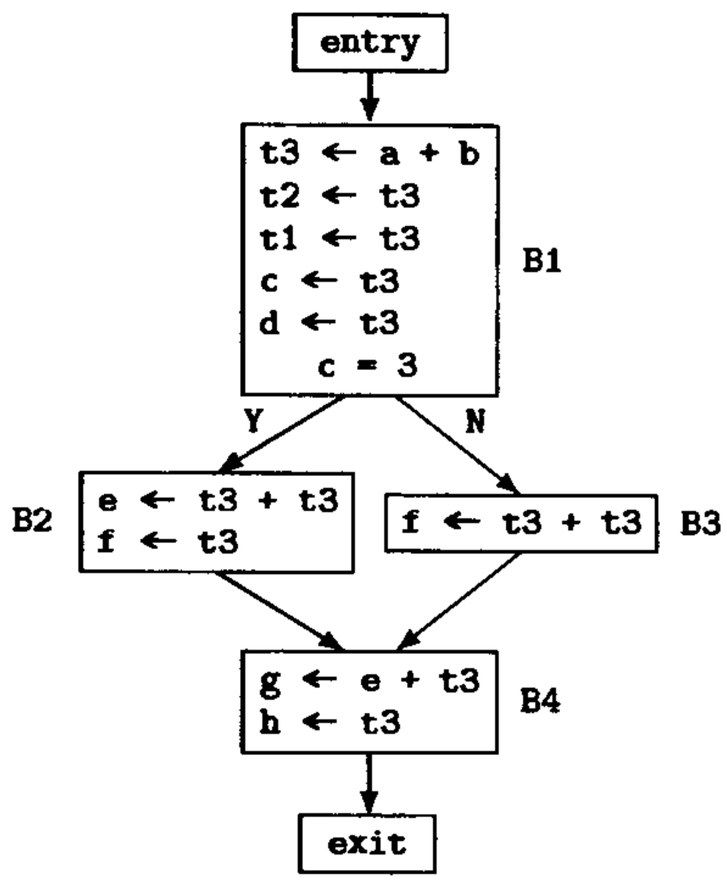


图13-14 公共子表达式删除与复写传播的结合（图13-13的继续）：全局复写传播后。注意，现在死代码删除已能够删除基本块B1中的赋值t2←t3和t1←t3，并且代码提升（13.5节）能够将t3+t3的两个计算移到基本块B1

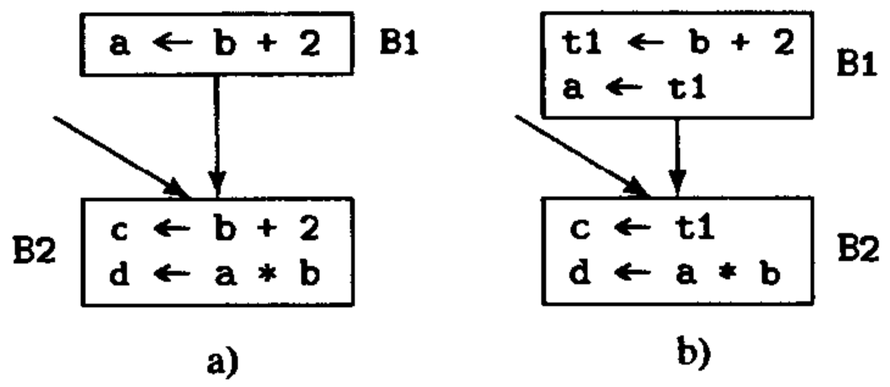


图13-15 在a) 中，从B1到B2的边上需要2个寄存器，而在b) 中，需要3个寄存器

取决于编译器的组织结构，这一类问题可能要到公共子表达式删除完成之后才能发现；尤其是，公共子表达式删除常常在中级中间代码上进行，而寄存器分配以及可能伴随的将公共子表达式的值存放到内存的动作，都要到低级中间代码生成之后才做。这种因素，以及其他一些原因，导致人们赞成在低级中间代码上进行公共子表达式删除和其他许多全局优化。IBM XL关于POWER和PowerPC的编译器（参见21.2节），以及Hewlett-Packard关于PA-RISC的编译器就是这样做的。

实现向前替代一般较容易。在MIR中，对于那种用一个临时变量对另一个变量赋值的指令，若此临时变量是流图中较早执行的一个表达式的计算结果，则检查该表达式的操作数从该表达式被计值到向前替代点之间是否没有被修改。如果没有，我们可以简单地在那一点复制该表达式的计算，并将计算产生的结果存放在适当的地方。

395 ~ 396

13.2 循环不变代码外提

循环不变代码外提（loop-invariant code motion）识别循环中那种在循环的每一个迭代都产生相同值的计算，并将它们移到循环之外[⊖]。

最重要的循环不变代码实例中有很多（但不是全部）是访问数组元素的地址计算，这种地

⊖ 注意，如果一个计算出现在嵌套循环内，对外层循环的特定迭代而言，它可能对内层循环的每一个迭代产生相同的值，但对外层循环的不同迭代产生不同的值。这种计算将移到内层循环之外，而不是外层循环之外。

bbs.theithome.com

址计算代码一直要到我们将程序转换到类似于MIR的中间代码或到低级代码之后，才能暴露给优化。作为循环不变计算的一个例子——其中没有暴露数组地址——考虑图13-16a 所示的Fortran[⊖]代码，它可以转换成如图13-16b所示的情形。这节省了近10 000次乘法操作和内层循环体中约5 000次迭代的5 000次加法操作。如果详细地描述地址计算，我们在计算a(i,j)时还会有与12.3.1节的例子所示类似的其他循环不变量。

```
do i = 1, 100
   l = i * (n + 2)
   do j = i, 100
      a(i,j) = 100 * n
         + 10 * l + j
   enddo
enddo
```

a)

```
t1 = 10 * (n + 2)
t2 = 100 * n
do i = 1, 100
   t3 = t2 + i * t1
   do j = i, 100
      a(i,j) = t3 + j
   enddo
enddo
```

b)

图13-16 a) 在Fortran中循环不变计算的例子，b) 转换后的结果代码

识别循环不变量是简单的。假定我们已经通过前面的控制流分析标识出了各个循环，并已通过前面的数据流分析计算出了ud链，所以我们知道哪些定值会对一个给定变量的使用有影响[⊜]。于是，可以归纳地定义循环中的循环不变指令的集合，即一条指令是循环不变的，如果对于它的每一个操作数:

1. 该操作数是常数，
2. 到达该操作数的这个使用的所有定值都在循环之外，或者
3. 该操作数只存在一个到达这条指令的定值，并且这个定值它的指令本身是该循环内的一个循环不变量。

注意，这个定义允许按如下步骤来计算循环的循环不变计算集合:[⊝]

1. 在下面的每一步中，记录指令被确定为循环不变量时的次序。
2. 首先，标志指令中的常数为循环不变量。
3. 接着，标志那些到达它们的定值都位于循环之外的所有操作数。 397
4. 然后，标志循环不变指令，它们是a)所有操作数都标志为循环不变量的指令，或b)这种指令: 它只有一个到达定值且那个定值已经标志为此循环中的循环不变量，并且在循环内该指令之前没有使用由该指令赋值的变量。
5. 重复上面的步骤2到步骤4，直到没有新的操作数或指令被标识为迭代中的不变量为止。

图13-17给出了实现这个算法的两个例程。我们用bset表示构成一个循环的基本块的索引集合。Mark_Invar()对数据结构进行初始化，然后按宽度为主次序对循环中的每一个基本块调用Mark_Block()。代码中使用了下述函数:

1. Breadth_First(*bset*, *Succ*, *en*)返回一个按宽度为主次序枚举*bset*中基本块的序列，其中*en*是*bset*的入口基本块（参见图7-13）。

⊖ Fortran 77的正式语法要求do循环以“do *n v*=...”开始，其中*n*是一个语句标号，*v*是一个变量，并且以带标号*n*的语句结束。但是关于编译器的文献习惯使用“do *v*=...”和“enddo”形式，我们在这里以及全书都使用这种形式。

⊜ 如果我们没有可用的ud链，仍然可以识别出循环不变量，但是对每一个使用，需要检查哪些定义可以到达它，即，实际上相当于计算ud链。

⊝ 我们可以将它形式化为数据流问题，但这样做的意义不大。照现在的方法，它在计算Reach_Defs_Out()和Reach_Defs_In()时隐式使用了ud链和du链。

bbs.theithome.com

```
InstInvar: (integer × integer) ⟶ boolean
InvarOrder: sequence of (integer × integer)
Succ: integer ⟶ set of integer

procedure Mark_Invar(bset,en,nblocks,ninsts,Block)
   bset: in set of integer
   en, nblocks: in integer
   ninsts: in array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of MIRInst
begin
   i, j: integer
   change: boolean
   Order := Breadth_First(bset,Succ,en): sequence of integer
   InvarOrder := []
   for i := 1 to nblocks do
      for j := 1 to ninsts[i] do
         InstInvar(i,j) := false
      od
   od
   repeat
      change := false
      || process blocks in loop in breadth-first order
      for i := 1 to |Order| do
         change ∨= Mark_Block(bset,en,Order↓i,
            nblocks,ninsts,Block)
      od
   until !change
end    || Mark_Invar

procedure Mark_Block(bset,en,i,nblocks,ninsts,Block) returns boolean
   bset: in set of integer
   en, i, nblocks: in integer
   ninsts: in array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of MIRInst
begin
   j: integer
   inst: MIRInst
   change := false: boolean
   for j := 1 to ninsts[i] do
      || check whether each instruction in this block has loop-constant
      || operands and appropriate reaching definitions; if so,
      || mark it as loop-invariant
      if !InstInvar(i,j) then
         inst := Block[i][j]
         case Exp_Kind(inst.kind) of
binexp:     if Loop_Const(inst.opd1,bset,nblocks,ninsts, Block)
               ∨ Reach_Defs_Out(Block,inst.opd1,i,bset)
               ∨ Reach_Defs_In(Block,inst.opd1,i,j,bset) then
               InstInvar(i,j) := true
            fi
            if Loop_Const(inst.opd2,bset,nblocks,ninsts, Block)
               ∨ Reach_Defs_Out(Block,inst.opd2,i,bset)
               ∨ Reach_Defs_In(Block,inst.opd2,i,j,bset) then
               InstInvar(i,j) &= true
            fi
```

图13-17 标识循环体中循环不变指令的代码

bbs.theithome.com

```
unexp:      if Loop_Const(inst.opd,bset,nblocks,ninsts,Block)
               V Reach_Defs_Out(Block,inst.opd,i,bset)
               V Reach_Defs_In(Block,inst.opd,i,j,bset) then
               InstInvar(i,j) := true
            fi
default: esac
      fi
      if InstInvar(i,j) then
         || record order in which loop invariants are found
         InvarOrder ⊕= [⟨i,j⟩]
         change := true
      fi
   od
   return change
end    || Mark_Block
```

图13-17 （续）

2. Loop_Const(*opnd*, *bset*, nblocks, ninsts, Block)判断*opnd*是否是由*bset*中基本块组成的循环中的一个循环常数。

3. Reach_Defs_Out(Block, *opnd*, *i*, *bset*)返回true，如果到达基本块*i*的*opnd*的所有定值都在其索引不属于*bset*的基本块中；否则返回false。

4. Reach_Defs_In(Block, *opnd*, *i*, *j*, *bset*)返回true，如果下述条件满足：恰恰只存在到达基本块Block[*i*][*j*]的*opnd*的一个定值，此定值在由*bset*中基本块构成的循环内，而且已被标识是循环不变的，并在Block[*i*][*j*]之前执行；此外，在循环内基本块*i*的指令*j*之前不存在对Block[*i*][*j*]的结果变量（若有的话）的使用；否则返回false。

398～400

作为Mark_Invar()的例子，考虑图13-18所示的流图。调用

```
Mark_Invar({2, 3, 4, 5, 6}, 2, 8, ninsts, Block)
```

导致设置Order＝{2，3，4，5，6}，初始所有的InstInvar(*i*,*j*)为false，并设置change＝false，随后对每一个基本块按Order给出的宽度为主次序调用Mark_Block()。对基本块B2施加Mark_Block()，由此对Block[2][1]执行binexp情形，它保持InstInvar(2，1)不变并返回false。

用Mark_Block()处理基本块B3导致对Block[3][1]执行binexp的情形，它设置InstInvar(3，1)＝true和InvarOrder＝[<3，1>]。接着Mark_Block()对Block[3][2]执行unexp情形，它设置InstInvar(3，2)＝true和InvarOrder＝[<3，1>，<3，2>]，并返回true。然后Mark_Block()对Block[3][3]执行binexp情形，它保持InstInvar(3，3)和InvarOrder不变，并返回true。

用Mark_Block()处理基本块B4导致对Block[4][1]执行binexp情形，它保持

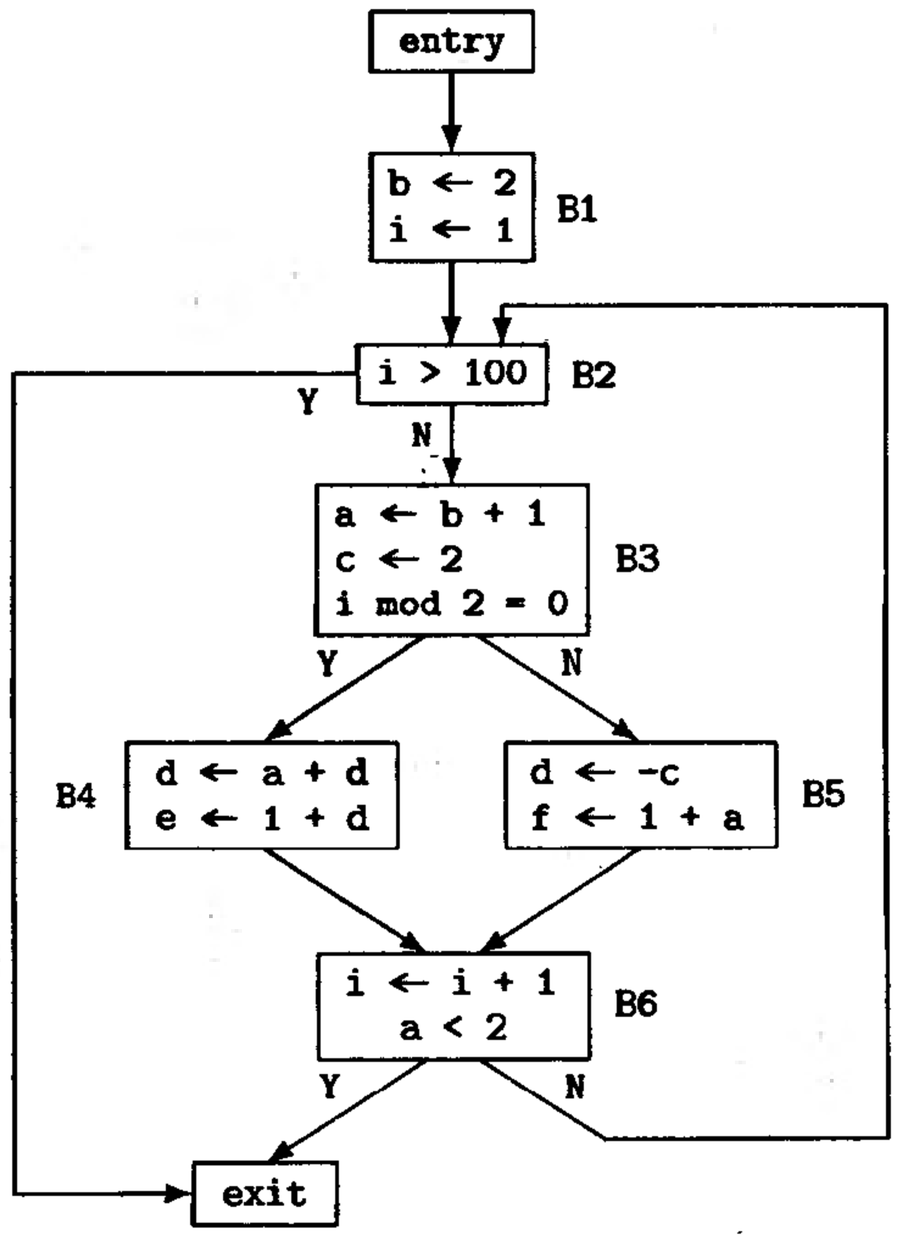


图13-18 循环不变代码外提的例子

bbs.theithome.com

InstInvar(4, 1)不变。接着Mark_Block()对Block[4][2]执行binexp情形，它保持InstInvar(4, 2)不变并返回false。

用Mark_Block()处理基本块B5导致对Block[5][1] 执行unexp情形，它保持InstInvar(5, 1)不变。接着Mark_Block()对Block[5][2]执行binexp情形，它设置InstInvar(5, 2)＝true和InvarOrder＝[<3, 1>, <3, 2>, <5, 2>]，然后返回true。

用Mark_Block()处理基本块B6导致对Block[6][1]执行binexp情形，它保持InstInvar(6, 1)不变。接着Mark_Block()对Block[6][2]执行binexp情形，它判断出Block[6][2]是循环不变的，因此它设置InstInvar(6, 2)＝true，InvarOrder＝[<3, 1>, <3, 2>, <5, 2>, <6, 2>]，然后返回true。

现在，在Mark_Invar()中change＝true，所以我们对循环中的每一个基本块再执行一次Mark_Block()。读者可以验证没有指令被进一步标识是循环不变的，因此这个repeat循环不需要再进一步迭代。

注意，如果我们在确定不变量的期间对运算执行重结合，则用这种方法确定的循环不变量的指令条数会有所增加。具体地，假设我们有图13-19a中的MIR代码位于循环内，其中i是循环索引变量，j是循环不变量。在这种情况下，这两条指令都不是循环不变的。但是，如果我们如图13-19b所示对这两个加法执行重结合，则第一条指令是循环不变的。另外要注意的是，由于别名的潜在影响，为了更精确，我们关于循环不变量的定义还需要稍微做点限制。例如，MIR指令

```
t1 ← i + j
n ← t1 + 1
```

a)

```
t2 ← j + 1
n ← i + t2
```

b)

图13-19 重结合导致a)中的计算被识别为b)中的循环不变量的例子

```
m ← call f, (1, int; 2, int)
```

可以是循环不变的，但是仅在每次用相同的参数调用它都产生相同的结果，并且没有副作用的情形下才是循环不变的。而这个条件是否满足只有通过过程间分析（参见第19章）才能发现。因此，在Mark_Block()中的listexp情形下InstInvar(*i*, *j*)＝false。

现在，当标识出循环不变计算之后，我们可以将它们移到包含它们的这个循环的前置块中（参见7.4节）。从表面上看，我们似乎可以简单地按它们被标识的次序移动每一条循环不变指令到前置块中。但不幸的是，这种方法在实际中常常太冒进。这种做法导致的错误有两种原因（如图13-20所示）。注意，这种可能的错误只作用于赋值指令。因此如果我们有一个循环不变条件，比如说，a<0，则用临时变量t替代它，并将赋值t←a<0放置在前置块中就将总是安全的。这两种原因如下:

401

1. 一个外提的赋值的左部变量的所有使用都将只能由这个特定的定值所到达，尽管这个定值原来只是可以到达这些使用的若干定值中的一个。这个赋值原本可能只在循环的某些迭代被执行，而如果将它移到前置块中，则它将具有在循环的每一个迭代被执行的效果。图13-20a举例说明了这种情况：基本块B5中n的使用可以由两个赋值n←0和n←2到达。如果n←2被移到前置块，不管怎样，赋给n的值都总是2。

2. 原来包含那条被外提指令的基本块可能在循环的每一遍中都不会被执行。当一个基本块的执行是有条件的时候就可能发生这种情况，这会导致给目标变量赋予不同的值，或导致被转换的代码出现原来的代码本不会出现的异常。这种问题也可能发生在循环体可能执行0次，即，直接满足循环终止条件时。图13-20b举例说明了这种问题：当循环的每一次迭代都旁路包含n←2的基本块时，赋值n←2不会被执行。这种问题可通过要求该基本块是循环的所有出口基本块的必经结点来避免（其中，出口基本块是有一个后继在循环之外的基本块），因为这样它就一定会在某个迭代被执行。

bbs.theithome.com

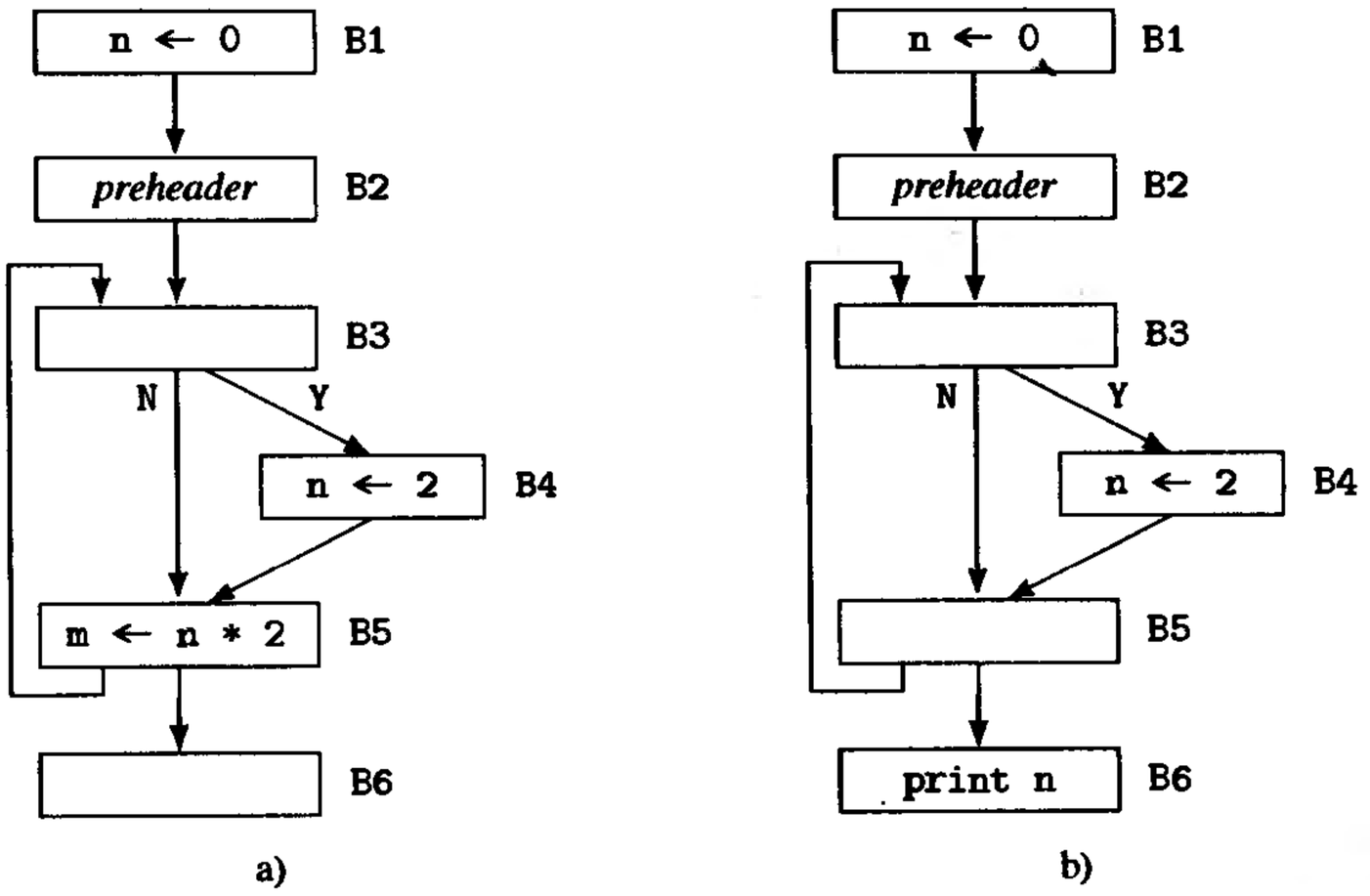


图13-20 阐述我们的代码外提基本方法的两个流图例子。在两种情况下，如果n←2被移到前置块，它将总是执行，尽管原来它可能只在某些迭代中或只在一个迭代中执行

为了修正这个算法避免上述问题，我们需要有预防这两种情况发生的条件。令*v*是要移到前置块的候选指令赋值的变量，则预防这种问题的保护条件如下:

1. 定值*v*的语句所在的基本块必须是循环中所有使用了*v*的基本块的必经结点。

2. 定值*v*的语句所在的基本块必须是循环中所有出口基本块的必经结点。

有了这两个附带条件后得到的算法就是正确的。算法如下：对于每一条已标识是循环不变量的指令，如果它满足上面的两个条件，则将它们按被标识的次序移到循环前置块中。

实现这个算法的ICAN例程Move_Invar()在图13-21中给出。该例程使用由Mark_Invar()计算的序列InvarOrder和下面4个函数:

1. insert_preheader(*bset*, nblocks, ninsts, Block)如果由*bset*中的基本块组成的循环还没有前置基本块的话，为它创建一个前置基本块并插入到流图中。

2. Dom_Exits(*i*, *bset*)返回true，如果基本块*i*是索引属于*bset*的基本块集合组成的循环的所有出口基本块的必经结点；否则返回false。

3. Dom_Uses(*i*, *bset*, *v*)返回true，如果基本块*i*是循环中变量*v*的所有使用的必经结点，其中循环由索引属于*bset*的基本块组成；否则返回false。

4. append_preheader(*bset*, ninsts, Block, *inst*)插入指令*inst*到循环前置基本块的末尾，其中循环由索引属于*bset*的基本块组成。它可以用4.8节的append_block()来实现。

注意，第二个附带条件仍然没有排除循环可能执行0次的情形，因为循环测试可能在循环的顶部执行，而测试结果在循环一开始就可能为真。有两种方法可处理这个问题：一种方法叫做循环倒置，将在18.5节讨论；另一种方法简单地移动代码到前置块，并用一个测试是否进入循环的条件，即识别终止条件是否一开始就为false，来保护它。这种方法总是安全的，但它增加了代码体积，并在这个测试总是true或总是false的情形下，还有可能使代码变慢。但另一方面，这种方法使常数传播分析有可能确定出结果条件是常数值，因而可将它删除（随同它所保护的代码一起，如果这个条件总是false的话）。

402
~
403

作为应用Move_Invar()的例子，我们继续图13-18的流图例子。我们调用

```
Move_Invar({2, 3, 4, 5, 6}, 2, 8, ninsts, Block)
```

bbs.theithome.com

```
procedure Move_Invar(bset,nblocks,ninsts,Block,Succ,Pred)
   bset: in set of integer
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Inst: inout array [1··nblocks] of array [··] of MIRInst
   Succ, Pred: inout integer ⟶ set of integer
begin
   i, blk, pos: integer
   P: set of (integer × integer)
   tj: Var
   inst: MIRInst
   insert_preheader(bset,nblocks,ninsts,Block)
   for i := 1 to |InvarOrder| do
      blk := (InvarOrder↓i)@1
      pos := (InvarOrder↓i)@2
      inst := Block[blk][pos]
      if Has_Left(inst.kind) & (Dom_Exits(blk,bset)
         & Dom_Uses(blk,bset,inst.left) then
         || move loop-invariant assignments to preheader
         case inst.kind of
binasgn, unasgn: append_preheader(Block[blk][pos],bset)
            delete_inst(blk,pos,ninsts,Block,Succ,Pred)
default:    esac
      elif !Has_Left(inst.kind) then
         || turn loop-invariant non-assignments to assignments in preheader
         tj := new_tmp( )
         case inst.kind of
binif, bintrap: append_preheader(bset,ninsts,Block,⟨kind:binasgn,left:tj,
                  opr:inst.opr,opd1:inst.opd1,opd2:inst.opd2⟩)
unif, untrap:  append_preheader(bset,ninsts,Block,⟨kind:binasgn,left:tj,
                  opr:inst.opr,opd:inst.opd⟩)
default:    esac
            case inst.kind of
               || and replace instructions by uses of result temporary
binif, unif:   Block[blk][pos] := ⟨kind:valif,opd:⟨kind:var,val:tj⟩,
                  label:inst.lbl⟩
bintrap, untrap:
               Block[blk][pos] := ⟨kind:valtrap,opd:⟨kind:var,val:tj⟩,
                 trapno:inst.trapno⟩
default:    esac
         fi
      od
   end    || Move_Invar
```

图13-21 循环不变指令的代码外提

因为

$$InvarOrder=[<3,1>,<3,2>,<5,2>,<6,2>]$$

故Move_Invar()中最外层的for循环只做i=1,2,3,4几个迭代。对于i=1，它设置blk=3和pos=1，并确定出基本块B3是循环出口的必经结点，Block[3][1]有一个左部量，并且

$$Dom\_Uses(3,\{2,3,4,5,6\},a)=true$$

因此，它执行binasgn情形，将指令Block[3][1]添加到前置基本块末尾，即基本块B1末尾。对于i=2，此例程的处理过程类似，但它选择的是unasgn情形（它也是binasgn情形）。

bbs.theithome.com

对于i=3，它设置blk=5和pos=2，确定出基本块B5不是循环出口的必经结点，因此不做进一步的改变。对于i=4，它设置blk=6和pos=2，确定出B6是循环出口的必经结点，并确定出Block[6][2]没有左部量。因此它执行binif情形，这种情形下创建一个新的临时变量t1，添加t1←a<2到基本块B1，并将Block[6][2]改变成一条对t1进行测试的指令。结果代码如图13-22所示。注意，假如在此时做常数折叠，我们就能判别出t1总是false，因此该循环不会终止。

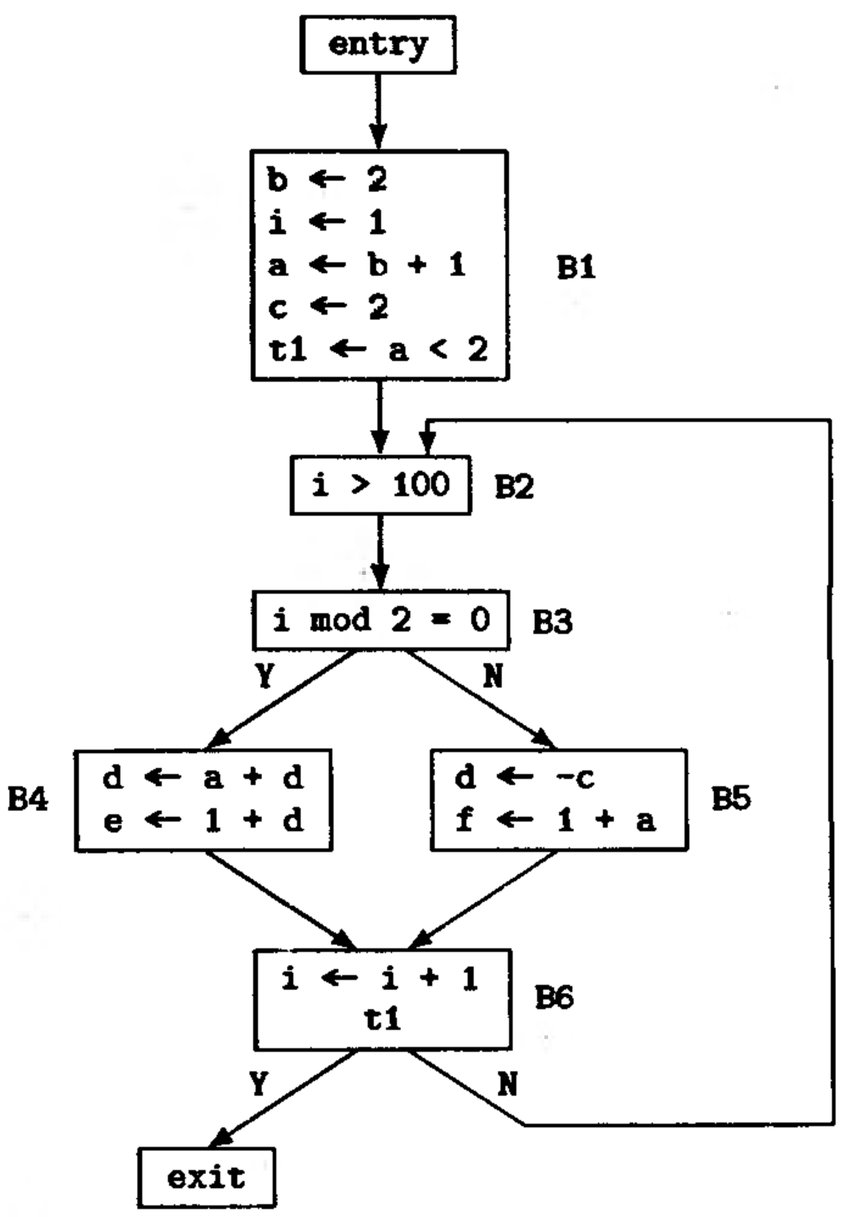


图13-22 对图13-18的流图施加循环不变代码外提的结果

在对嵌套循环执行循环不变代码外提时，我们从最内层循环由里向外进行。按这种顺序能够标识的不变量和外提的代码最多。例如，考虑图13-23a中的Fortran 77循环嵌套。标识最内层的不变量得到图13-23b——不变量带有下划线，将它们从循环中提出得到图13-23c。接着，标识外层的循环不变量得到图13-23d，将它们从循环中提出产生了图13-23e中的最终代码。

对于那种执行归约的循环，循环不变代码外提有一种特殊情形。归约（reduction）是一种诸如求累加

```
do i = 1,100
   m = 2*i + 3*l
   do j = 1, i - 1
      a(i,j) = j + m + 3*k
   enddo
enddo
```

a)

```
do i = 1,100
   m = 2*i + 3*l
   do j = 1, i - 1
      a(i,j) = j + m + 3*k
   enddo
enddo
```

b)

```
do i = 1,100
   m = 2*i + 3*l
   t1 = i - 1
   t2 = m + 3*k
   do j = 1, t1
      a(i,j) = j + t2
   enddo
enddo
```

c)

```
do i = 1,100
   m = 2*i + 3*l
   t1 = i - 1
   t2 = m + 3*k
   do j = 1,t1
      a(i,j) = j + t2
   enddo
enddo
```

d)

```
t3 = 3*l
t4 = 3*k
do i = 1,100
   m = 2*i + t3
   t1 = i - 1
   t2 = m + t4
   do j = 1, t1
      a(i,j) = j + t2
   enddo
enddo
```

e)

图13-23 嵌套Fortran循环的循环不变代码外提（不变量带有下划线）

bbs.theithome.com

和、连乘或求最大值之类的操作，例如图13-24循环中的操作，构成此循环的这4条指令可以用一条s=n*a(j)来替代。如果这种归约的操作数是循环不变量，那么，这个循环可以用单个操作来替代，具体方法如下：

```
s = 0.0
do i = 1,n
   s = s + a(j)
enddo
```

图13-24 归约的例子

1. 如果循环是求累加和，可以用单条乘法运算替代它。
2. 如果循环是求连乘，可以用单条求幂运算替代它。

404 ~ 406

3. 如果循环是求最大值或最小值，可以通过将循环不变量的值赋给存放极值的结果变量来替代它。
4. 如此等等。

13.3 部分冗余删除

部分冗余删除是一种结合了全局公共子表达式删除和循环不变代码外提，并且还能对代码有另外一些改善的优化。

实质上，部分冗余（partial redundancy）是一种在流图的某条路径上其执行多于一次以上的计算，即一个给定的计算在流图的某条路径的一个点上已经被计值，并且在此路径上还将被计值。部分冗余删除插入和删除流图中的这种计算，使得在经过转换之后，每一条路径出现的这种计算次数不会多于——通常是少于——原来出现的计算。将它形式化为数据流问题比我们考虑的其他所有问题都要复杂。

部分冗余删除首创性的工作是由Morel和Renvoise [MorR79] 给出的，他们后来又将它扩充为过程间的形式[MorR81]。他们的公式所表述的过程内的版本需要执行双向数据流分析，但我们将看到，本书讨论的现代版本避免了这样做。这个现代版本以一种最新的称为*懒惰代码移动*（lazy code motion）的公式为基础，这种公式是由Knoop、Rüthing和Steffen [KnoR92]等人开发的。名字中使用“懒惰”一词是指，只要不会牺牲原古典算法已减少的冗余计算量，在流图中就尽可能推后放置一个计算。懒惰的目的是减少寄存器的压力(参见16.3.1节)，即，使得存放特定值的寄存器所跨越的指令范围最小。

为了用公式来描述部分冗余删除有关的数据流分析，我们需要定义表达式的一系列的局部和全局数据流性质，并说明如何计算每一种性质。注意，取一个变量的值是一种表达式，并对它施行同样的分析。

该算法的一个关键点是，如果在执行数据流分析之前已经对流图中的关键边进行了分割，则效率高得多。*关键边*（critical edge）是连接一个具有两个以上后继的结点和一个具有两个以上前驱的结点的边，例如图13-25a所示的从B1到B4的边。通过引入新基本块B1a分割这条边，允许执行冗余删除，如图13-25b所示。Dhamdhere和其他人已证明，这种图形转换是部分冗余删除能获得最大效果所需要的。

我们这一节从头到尾都使用图13-26的例子。注意B4中x*y的计算是冗余的，因为它已在

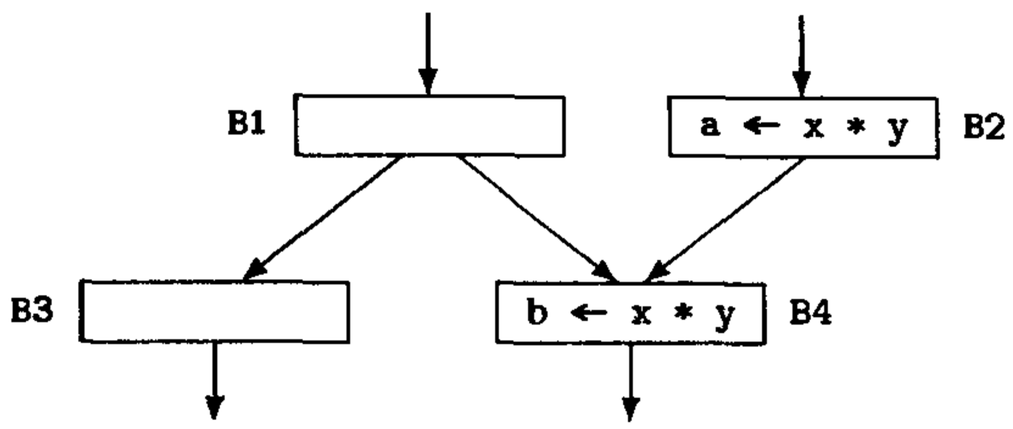
407

B2中被计算；由于同样的原因，B7中的x*y的计算是部分冗余的。在我们的例子中，关键是分割从B3到B5的边，以便有地方存放从B7移出的x*y的计算，使得它既不在从B2引出的路径上，也不位于B6之前。

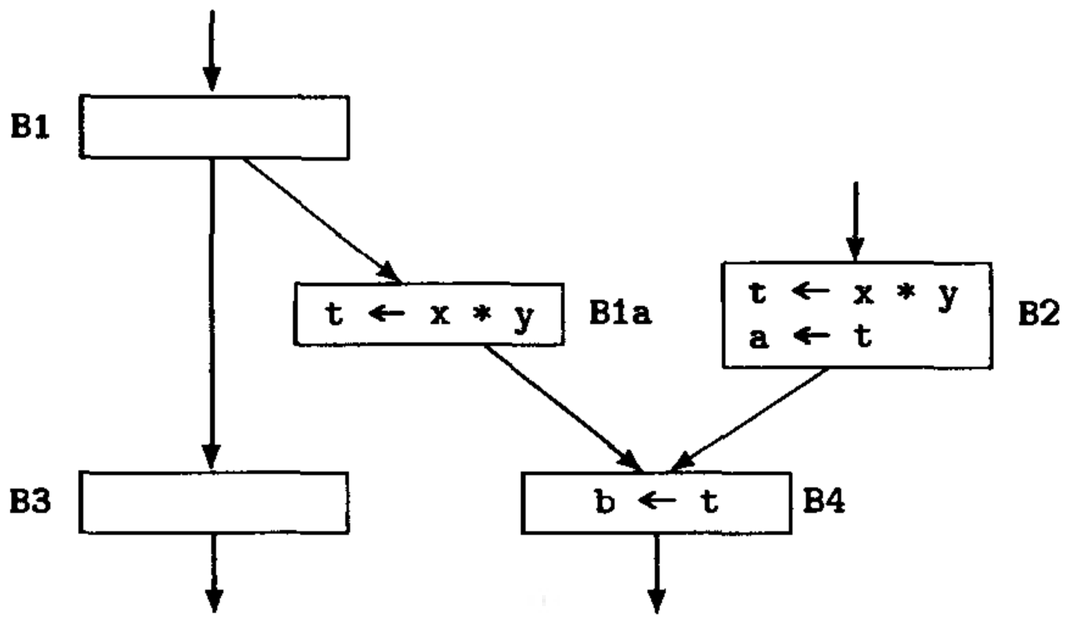
我们从标识和分割关键边开始。具体地，对于图13-26的例子，从B2和B3到B5的两条边都是关键边。例如，对于从B3到B5的边，我们有*Succ*(B3)={B5,B6}和*pred*(B5)={B2,B3}。分割这两条边需要创建新的基本块B2a和B3a，并用从原关键边的尾进入这个新基本块（一开始时为空基本块）的边，以及从这个新基本块出来进入原关键边的头的边来替代要分割的边。结果如图13-27所示。

bbs.theithome.com

超星公司制作请尊重作者版权



a)



b)

图13-25 在a)中，从B1到B4的边是关键边。在b)中，由于B1a的引入导致它被分割

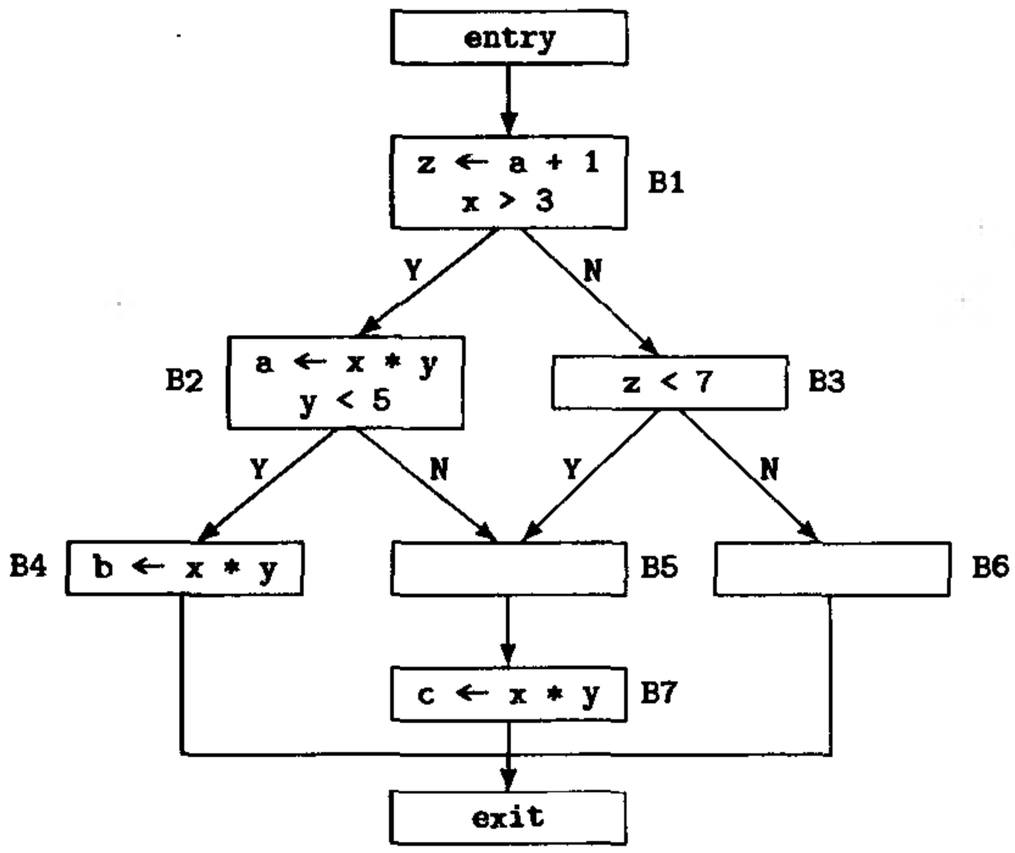


图13-26 部分冗余删除的例子

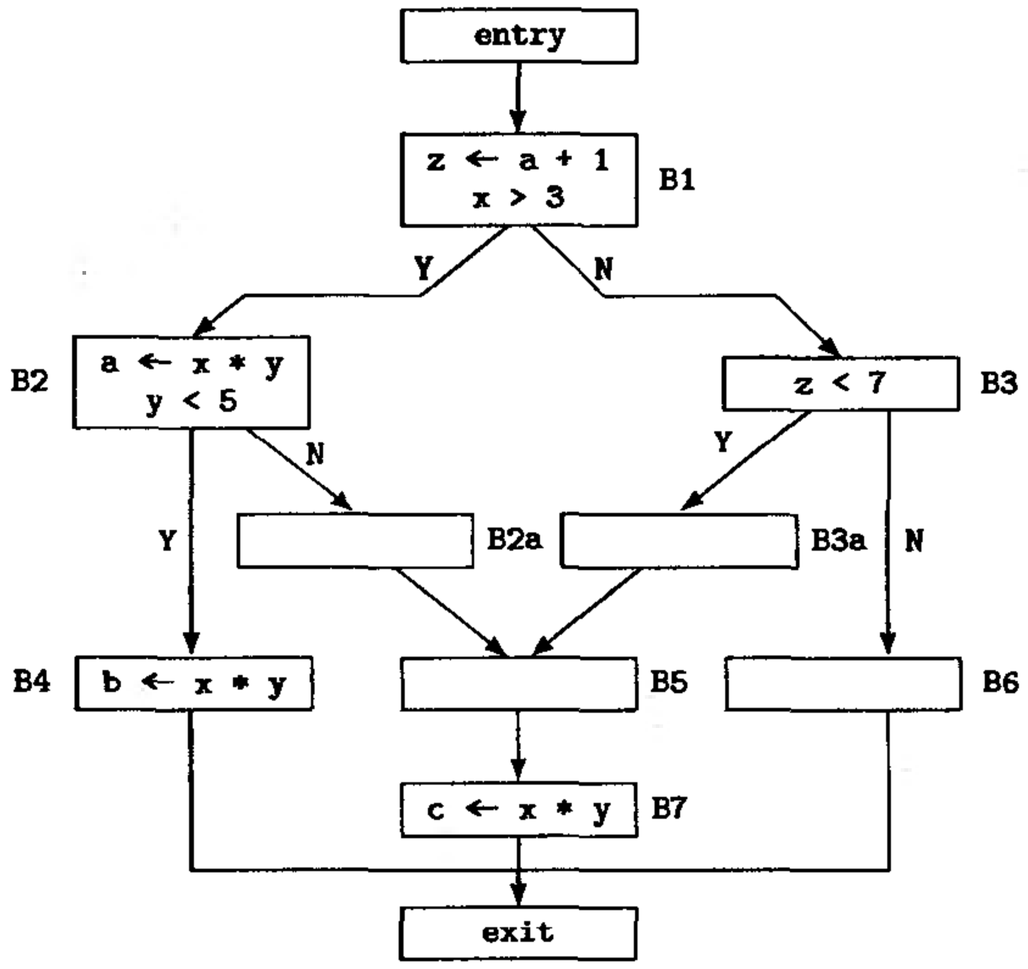


图13-27 分割图13-26中从B2到B5和从B3到B5的边的例子。新基本块B2a和B3a已加到流图中

我们考虑的第一个性质是局部透明性[⊖]。一个表达式的值在基本块*i*中是局部透明的（local transparent），如果在这个基本块中，没有对出现在此表达式中的变量进行赋值。我们用*TRANSloc*(*i*)表示在基本块*i*中局部透明的那些表达式集合。

408 ~ 409

对于我们的例子，

⊖ 注意，局部透明性与我们第8章称为*PRSV*的性质类似。

bbs.theithome.com

越是公司制作请尊重作者版权

TRANSloc(B2) = {x*y}

TRANSloc(*i*) = U_{exp} = {a+1, x*y}，其中*i* ≠ B2

U_{exp}表示程序中要分析的所有表达式的集合。

表达式的值在基本块*i*是局部可预见的（locally anticipatable），如果在基本块*i*中存在着此表达式的一个计算，并且移动这个计算到基本块的开始不会改变这个基本块的效果，即，如果基本块中在所考虑的这个计算的前面既没有使用该表达式，也没有对它的变量赋值。我们用*ANTloc*(*i*)表示基本块*i*中局部可预见的表达式集合。

对于我们的例子，*ANTloc*()的值如下:

ANTloc(entry) = ∅
ANTloc(B1) = {a+1}
ANTloc(B2) = {x*y}
ANTloc(B2a) = ∅
ANTloc(B3) = ∅
ANTloc(B3a) = ∅
ANTloc(B4) = {x*y}
ANTloc(B5) = ∅
ANTloc(B6) = ∅
ANTloc(B7) = {x*y}
ANTloc(exit) = ∅

一个表达式的值在基本块*i*的入口是全局可预见的（globally anticipatable），如果从那一点开始的每一条路径都包含该表达式的一个计算，并且如果将这个计算放置在这些路径上的任意点都产生相同的值。在基本块*i*入口可预见的表达式集合用*ANTin*(*i*)表示。一个表达式的值在一个基本块的出口处是可预见的，如果它在该基本块的每一个后继基本块的入口是可预见的；我们用*ANTout*(*i*)表示从基本块*i*出口时是全局可预见的表达式集合。为了计算流图中所有基本块*i*的*ANTin*()和*ANTout*()，我们以初值*ANTout*(exit) = ∅ 解下面的数据流方程:

$$ANTin(i) = ANTloc(i) \cup (TRANSloc(i) \cap ANTout(i))$$

$$ANTout(i) = \bigcap_{j \in Succ(i)} ANTin(j)$$

对于我们的例子，*ANTin*()和*ANTout*()的值如下:

ANTin(entry)	= {a+1}	*ANTout*(entry)	= {a+1}
ANTin(B1)	= {a+1}	*ANTout*(B1)	= ∅
ANTin(B2)	= {x*y}	*ANTout*(B2)	= {x*y}
ANTin(B2a)	= {x*y}	*ANTout*(B2a)	= {x*y}
ANTin(B3)	= ∅	*ANTout*(B3)	= ∅
ANTin(B3a)	= {x*y}	*ANTout*(B3a)	= {x*y}
ANTin(B4)	= {x*y}	*ANTout*(B4)	= ∅
ANTin(B5)	= {x*y}	*ANTout*(B5)	= {x*y}
ANTin(B6)	= ∅	*ANTout*(B6)	= ∅
ANTin(B7)	= {x*y}	*ANTout*(B7)	= ∅
ANTin(exit)	= ∅	*ANTout*(exit)	= ∅

410

做懒惰代码移动需要的下一个性质是最早性。一个表达式在基本块*i*的入口是最早的(earliest)，如果从entry通向基本块*i*的路径上不存在这样一个基本块，它计算该表达式并且产生与在进入基本块*i*时计算该表达式相同的值。类似地，可定义从一个基本块出口时的最早性。性质*EARLin*()和*EARLout*可用如下方程计算:

bbs.theithome.com

$$EARLin(i) = \bigcup_{j \in Pred(i)} EARLout(j)$$

$$EARLout(i) = \overline{TRANSloc(i)} \cup \left(\overline{ANTin(i)} \cap EARLin(i)\right)$$

其中 $\overline{A} = U_{exp} - A$，且初值$EARLin(\texttt{entry}) = U_{exp}$。

对于我们的例子，*EARLin*()的值如下:

EARLin(entry)	= {a+1,x*y}	*EARLout*(entry)	= {x*y}
EARLin(B1)	= {x*y}	*EARLout*(B1)	= {x*y}
EARLin(B2)	= {x*y}	*EARLout*(B2)	= {a+1}
EARLin(B2a)	= {a+1}	*EARLout*(B2a)	= {a+1}
EARLin(B3)	= {x*y}	*EARLout*(B3)	= {x*y}
EARLin(B3a)	= {x*y}	*EARLout*(B3a)	= ∅
EARLin(B4)	= {a+1}	*EARLout*(B4)	= {a+1}
EARLin(B5)	= {a+1}	*EARLout*(B5)	= {a+1}
EARLin(B6)	= {x*y}	*EARLout*(B6)	= {x*y}
EARLin(B7)	= {a+1}	*EARLout*(B7)	= {a+1}
EARLin(exit)	= {a+1,x*y}	*EARLout*(exit)	= {a+1,x*y}

接下来我们需要一种叫做延迟性的性质。一个表达式在基本块i的入口是延迟的(delayed)，如果它在那一点是可预见的和最早的，并且它的所有后继计算都在基本块i内。延迟性的方程是

$$DELAYout(i) = \overline{ANTloc(i)} \cap DELAYin(i)$$

$$DELAYin(i) = ANEAin(i) \cup \bigcap_{j \in Pred(i)} DELAYout(j)$$

其中

$$ANEAin(i) = ANTin(i) \cap EARLin(i)$$

411

且初始值$DELAYin(\texttt{entry}) = ANEAin(\texttt{entry})$。

对于我们的例子，*ANEAin*()的值如下:

ANEAin(entry)	= {a+1}
ANEAin(B1)	= ∅
ANEAin(B2)	= {x*y}
ANEAin(B2a)	= ∅
ANEAin(B3)	= ∅
ANEAin(B3a)	= {x*y}
ANEAin(B4)	= ∅
ANEAin(B5)	= ∅
ANEAin(B6)	= ∅
ANEAin(B7)	= ∅
ANEAin(exit)	= ∅

并且*DELAYin*()和*DELAYout*()的值如下:

DELAYin(entry)	= {a+1}	*DELAYout*(entry)	= {a+1}
DELAYin(B1)	= {a+1}	*DELAYout*(B1)	= ∅
DELAYin(B2)	= {x*y}	*DELAYout*(B2)	= ∅
DELAYin(B2a)	= ∅	*DELAYout*(B2a)	= ∅
DELAYin(B3)	= ∅	*DELAYout*(B3)	= ∅
DELAYin(B3a)	= {x*y}	*DELAYout*(B3a)	= {x*y}
DELAYin(B4)	= ∅	*DELAYout*(B4)	= ∅
DELAYin(B5)	= ∅	*DELAYout*(B5)	= ∅

版权所有 请勿制作公司复印

bbs.theithome.com

版权公司制作 请勿盗作者版权

$DELAYin$(B6) $= \emptyset$ $DELAYout$(B6) $= \emptyset$
$DELAYin$(B7) $= \emptyset$ $DELAYout$(B7) $= \emptyset$
$DELAYin$(exit) $= \emptyset$ $DELAYout$(exit) $= \emptyset$

下面我们定义的一个性质称为最迟性。一个表达式在基本块i的入口是最迟的（latest），如果基本块i的入口处是计算该表达式的最佳点，并且在从基本块i的入口到exit基本块的每一条路径上，该表达式的所有最佳计算点都出现在此表达式在原流图中的那些计算点中的某一个计算点之后。*LATEin*()的数据流方程如下:

$$LATEin(i) = DELAYin(i) \cap \left(ANTloc(i) \cup \overline{\bigcap_{j \in Succ(i)} DELAYin(j)} \right)$$

对于我们的例子，*LATEin*()如下:

$LATEin$(entry) $= \emptyset$
$LATEin$(B1) $= \{a+1\}$
$LATEin$(B2) $= \{x*y\}$
$LATEin$(B2a) $= \emptyset$
$LATEin$(B3) $= \emptyset$
$LATEin$(B3a) $= \{x*y\}$
$LATEin$(B4) $= \emptyset$
412 $LATEin$(B5) $= \emptyset$
$LATEin$(B6) $= \emptyset$
$LATEin$(B7) $= \emptyset$
$LATEin$(exit) $= \emptyset$

对于一个表达式的计算，其最佳计算位置定义为是孤立的（isolated），当且仅当从计算它的这个基本块的一个后继到exit基本块的每一条路径上，这个最佳位置点先于该表达式原来的每一个计算。数据流性质*ISOLin*()和*ISOLout*()由下面的方程组定义

$$ISOLin(i) = LATEin(i) \cup (\overline{ANTloc(i)} \cap ISOLout(i))$$

$$ISOLout(i) = \bigcap_{j \in Succ(i)} ISOLin(j)$$

且初始值$ISOLout$(exit)$= \emptyset$。

对于我们的例子，*ISOLin*()和*ISOLout*()的值如下:

$ISOLin$(entry) $= \emptyset$ $ISOLout$(entry) $= \emptyset$
$ISOLin$(B1) $= \{a+1\}$ $ISOLout$(B1) $= \emptyset$
$ISOLin$(B2) $= \{x*y\}$ $ISOLout$(B2) $= \emptyset$
$ISOLin$(B2a) $= \emptyset$ $ISOLout$(B2a) $= \emptyset$
$ISOLin$(B3) $= \emptyset$ $ISOLout$(B3) $= \emptyset$
$ISOLin$(B3a) $= \{x*y\}$ $ISOLout$(B3a) $= \emptyset$
$ISOLin$(B4) $= \emptyset$ $ISOLout$(B4) $= \emptyset$
$ISOLin$(B5) $= \emptyset$ $ISOLout$(B5) $= \emptyset$
$ISOLin$(B6) $= \emptyset$ $ISOLout$(B6) $= \emptyset$
$ISOLin$(B7) $= \emptyset$ $ISOLout$(B7) $= \emptyset$
$ISOLin$(exit) $= \emptyset$ $ISOLout$(exit) $= \emptyset$

一个基本块是其最佳计算点的表达式集合是由这样一些表达式组成的集合，对于这个基本块，这些表达式是最迟的但不是孤立的，即

$$OPT(i) = LATEin(i) \cap \overline{ISOLout(i)}$$

并且基本块中的冗余计算集合由那些在该基本块中使用的（即属于*ANTLOC*()）但在此基本块中既不是孤立的也不是最迟的表达式组成，即，

bbs.theithome.com

$$REDN(i) = ANTloc(i) \cap \overline{LATEin(i) \cup ISOLout(i)}$$

对于我们的例子，*OPT*()和*REDN*()的值如下：

OPT(entry)	= ∅	*REDN*(entry)	= ∅
OPT(B1)	= {a+1}	*REDN*(B1)	= {a+1}
OPT(B2)	= {x*y}	*REDN*(B2)	= {x*y}
OPT(B2a)	= ∅	*REDN*(B2a)	= ∅
OPT(B3)	= ∅	*REDN*(B3)	= ∅
OPT(B3a)	= {x*y}	*REDN*(B3a)	= ∅
OPT(B4)	= ∅	*REDN*(B4)	= {x*y}
OPT(B5)	= ∅	*REDN*(B5)	= ∅
OPT(B6)	= ∅	*REDN*(B6)	= ∅
OPT(B7)	= ∅	*REDN*(B7)	= {x*y}
OPT(exit)	= ∅	*REDN*(exit)	= ∅

因此，如果我们删除B4和B7中x*y的计算，留下B2中的那个，并在B3a中增加一个，则得到如图13-28所示的结果。

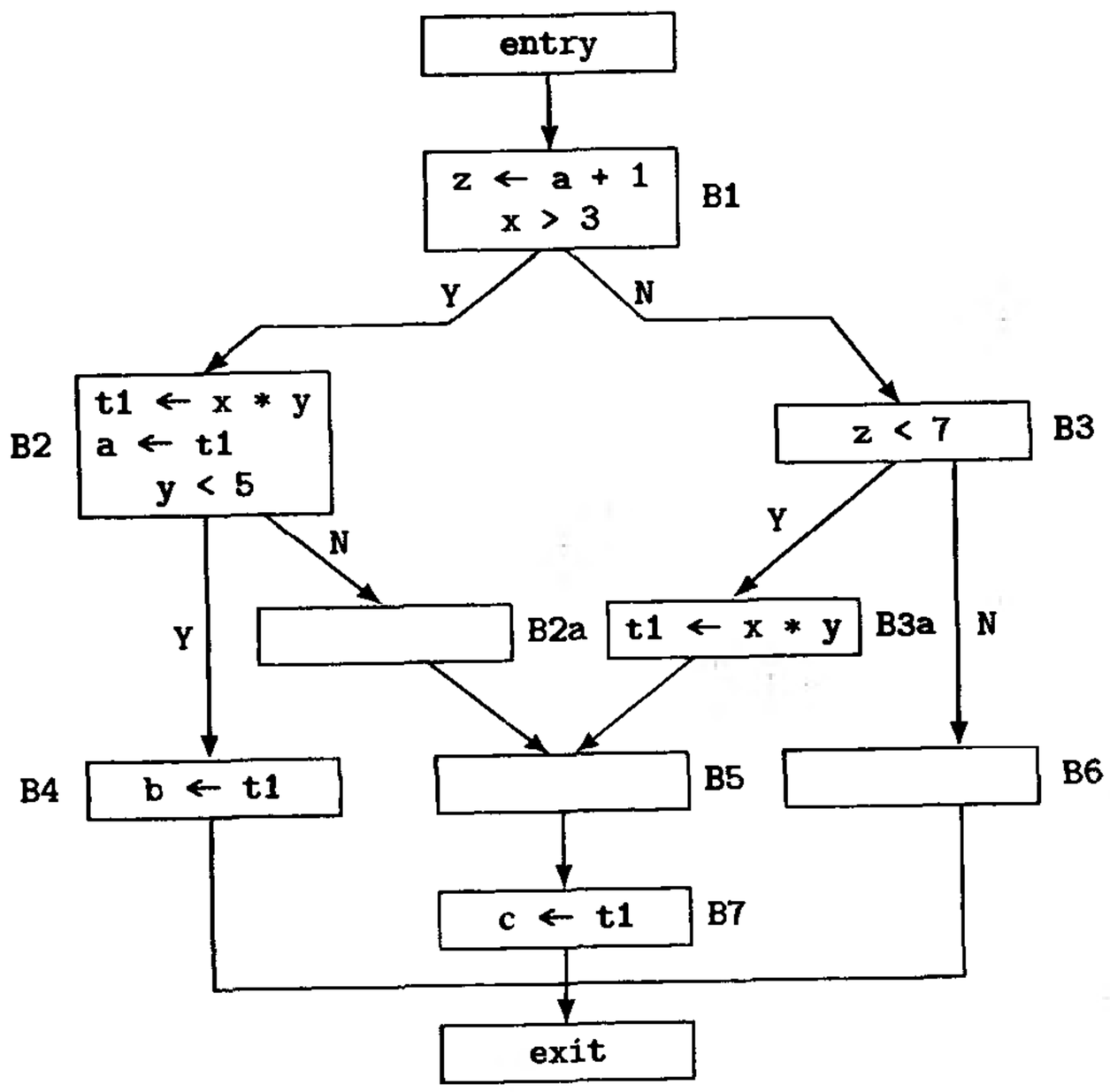


图13-28 对图13-27的例子实施现代部分冗余删除的结果

实现部分冗余移动所需要的代码与Move_Invar()类似。我们将它留给读者作为练习。

现代部分冗余删除可扩充为包含强度削弱。但是，其强度削弱是一种特别弱的形式，因为它不识别循环常量。例如，图13-29a中的代码可以用14.1.2节描述的方法进行强度削弱，产生图13-29b所示代码，但是，基于部分冗余删除的强度削弱不能产生这个代码，因为它不注意循环常量。

Briggs和Cooper [BriC94b]通过将部分冗余删除与全局重结合和全局值编号(见12.4.2节)结合在一起，改善了它的效果；Cooper和Simpson([CooS95c]和[Simp96])通过利用SSA形式对值操作而不是对标识符操作进一步改善了它。下一节将简单讨论它与重结合的结合。

413
414
bbs.theithome.com

```
k = 0
for i = 1,n

   for j = 1,n
      k = k + i*j
   endfor

endfor
```

a)

```
k = 0
for i = 1,n
   l = 0
   for j = 1,n
      l = l + j
   endfor
   k = k + i * l
endfor
```

b)

图13-29 说明从部分冗余导出的强度削弱不足的例子。a) 中的代码用14.1.2节的算法可以被强度削弱为b) 中的代码，但用部分冗余导出的强度削弱方法却不能，因为它不能识别内层循环中的变量i为循环不变量

13.4 冗余删除和重结合

重结合能显著地增强所有形式的冗余删除的适应性和效果。例如，在图13-30的Fortran代码中，对循环A仅施加公共子表达式删除得到循环B。若同时还包含重结合，则存在着如图13-31所示的另一种可能的优化序列。

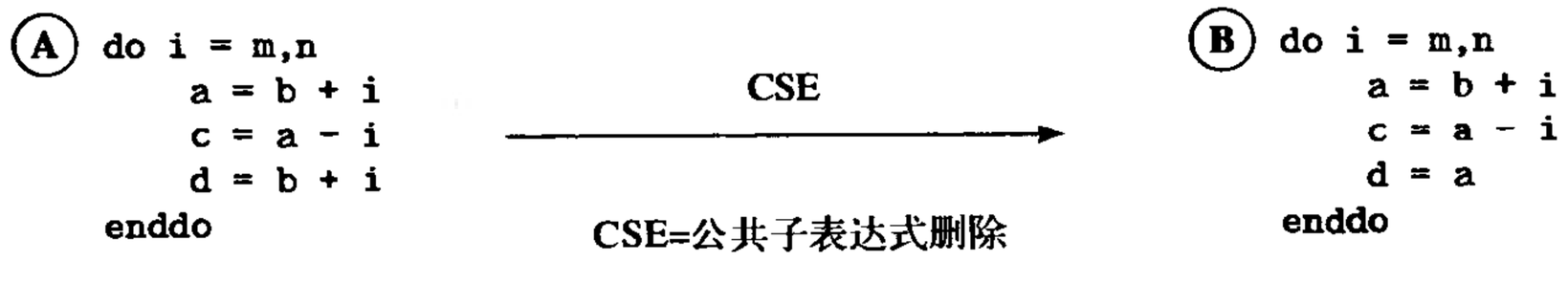


图13-30 对A中的循环仅施加公共子表达式删除得到B中的循环

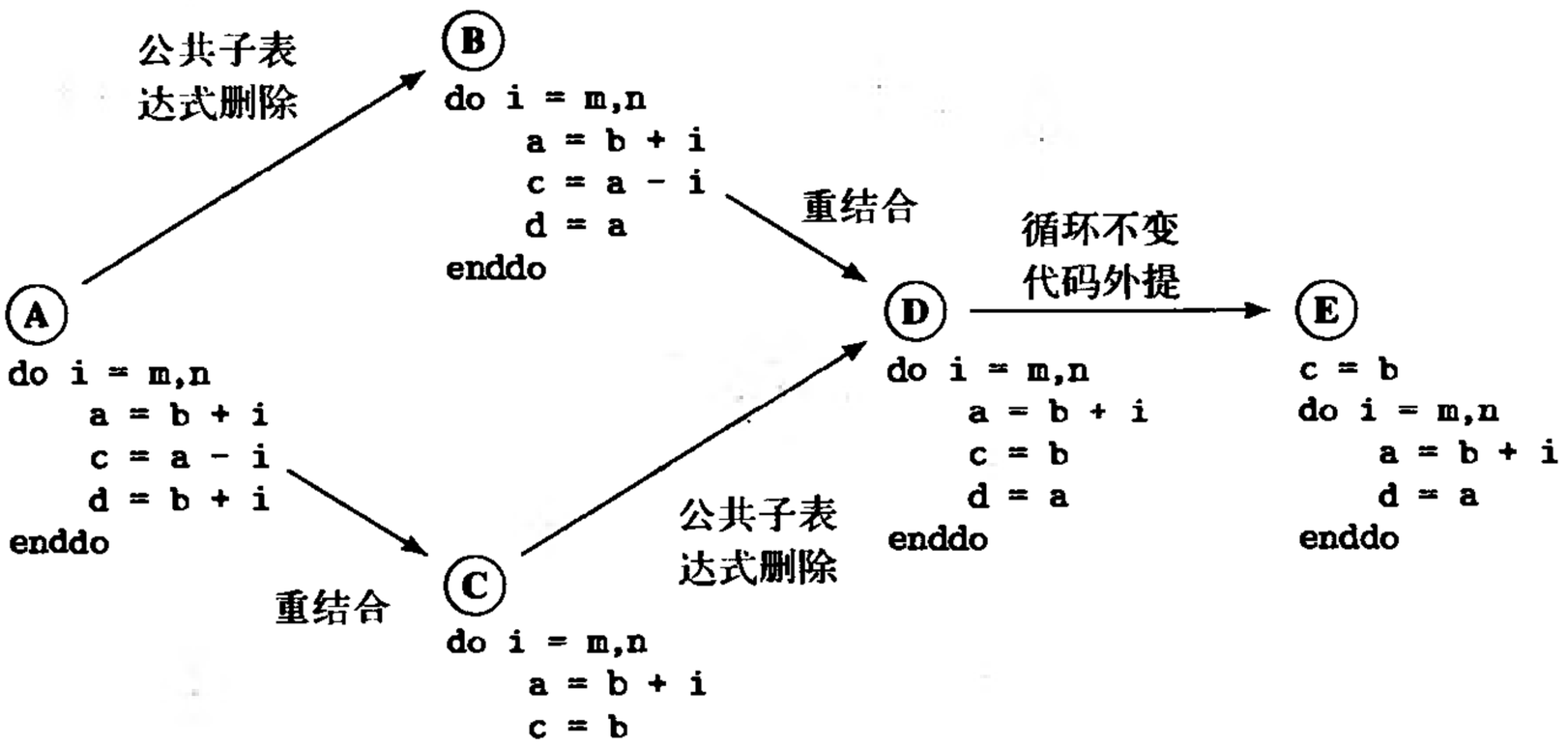


图13-31 公共子表达式删除和循环不变代码外提与重结合相结合，得到一种与图13-30相比可能性更大的转换序列和更有改善的结果

我们进一步注意到，这两种序列中有一个需要公共子表达式删除，而另一个不需要；但它们至少有同样的（最好）结果。这使人想到可重复地应用这三种优化的组合，但这样做很容易导致组合的激增，因此不建议采用这种做法。

部分冗余删除与重结合相结合可以在某种程度上减轻这种问题，如图13-32所示。注意，

bbs.theithome.com

415~416 超星公司制作请尊重作者版权

如果我们先做部分冗余删除，则在重结合后还需要再次应用它才能得到最好的结果。

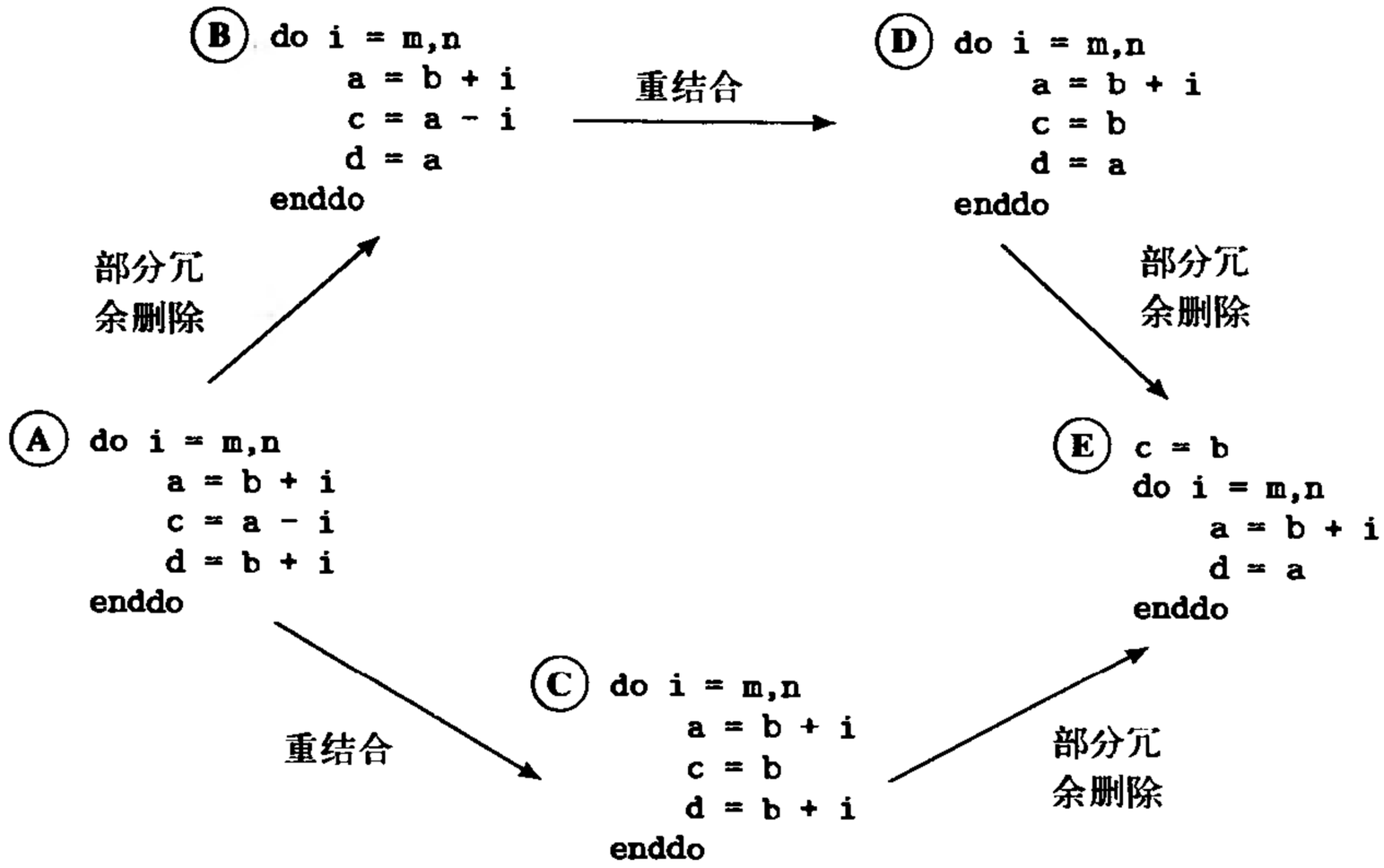


图13-32 部分冗余删除与重结合相结合得到与图13-31相同的结果

13.5 代码提升

代码提升（code hoisting）（也称统一（unification）——参见17.6节)寻找在某点之后的所有路径上总是被计算的表达式，并且将它们移到总能被计算的一个最晚点，超过这一点它们将被不同的路径计算。这种转换几乎总能减少程序所占的空间，但对程序的执行时间可能有正面影响、负面影响，或根本没有影响。它是否改善执行时间取决于它对指令调度的影响、对指令高速缓存的影响，以及其他一些因素。

称一个从给定点开始的无论什么路径上都被计算的表达式为在那一点的非常忙（very busy）表达式。为了确定非常忙表达式，我们对表达式做向后数据流分析。定义*EVAL*(*i*)是这种表达式集合，这些表达式在基本块*i*中被计算，并且该计算位于基本块中对它们的任何操作数赋值之前（如果有这种赋值的话）。定义*KILL*(*i*)是被基本块*i*杀死的表达式集合。就此上下文而言，一个表达式被基本块*i*杀死，如果在该基本块中它的一个（或多个）操作数被赋值，并且或者这种赋值位于此基本块中该表达式的计算之前，或者该表达式在此基本块中根本就没有被计算。于是在基本块*i*的入口和出口的非常忙表达式（very busy expression）集合的*VBEin*(*i*)和*VBEout*(*i*)定义分别为:

$$VBEin(i) = EVAL(i) \cup (VBEout(i) - KILL(i))$$

$$VBEout(i) = \bigcap_{j \in Succ(i)} VBEin(j)$$

其中，在解此数据流方程时，对于所有的*i*，初始值$VBEout(i) = \emptyset$。利用位向量可有效地实现这种数据流分析。

例如，对于图13-33中的流图，它的*EVAL*()和*KILL*()集合如下: 417

EVAL(entry)	= ∅	*KILL*(entry)	= ∅
EVAL(B1)	= ∅	*KILL*(B1)	= ∅
EVAL(B2)	= {c+d}	*KILL*(B2)	= {a+d}

bbs.theithome.com

EVAL(B3)	= {a+c,c+d}	*KILL*(B3)	= ∅
EVAL(B4)	= {a+b,a+c}	*KILL*(B4)	= ∅
EVAL(B5)	= {a+b,a+d}	*KILL*(B5)	= ∅
EVAL(exit)	= ∅	*KILL*(exit)	= ∅

并且它的*VBEin*()和*VBEout*()集合如下:

VBEin(entry)	= {c+d}	*VBEout*(entry)	= {c+d}
VBEin(B1)	= {c+d}	*VBEout*(B1)	= {c+d}
VBEin(B2)	= {c+d}	*VBEout*(B2)	= ∅
VBEin(B3)	= {a+b,a+c,c+d}	*VBEout*(B3)	= {a+b,a+c}
VBEin(B4)	= {a+b,a+c}	*VBEout*(B4)	= ∅
VBEin(B5)	= {a+b,a+c,a+d,c+d}	*VBEout*(B5)	= {a+b,a+c,c+d}
VBEin(exit)	= ∅	*VBEout*(exit)	= ∅

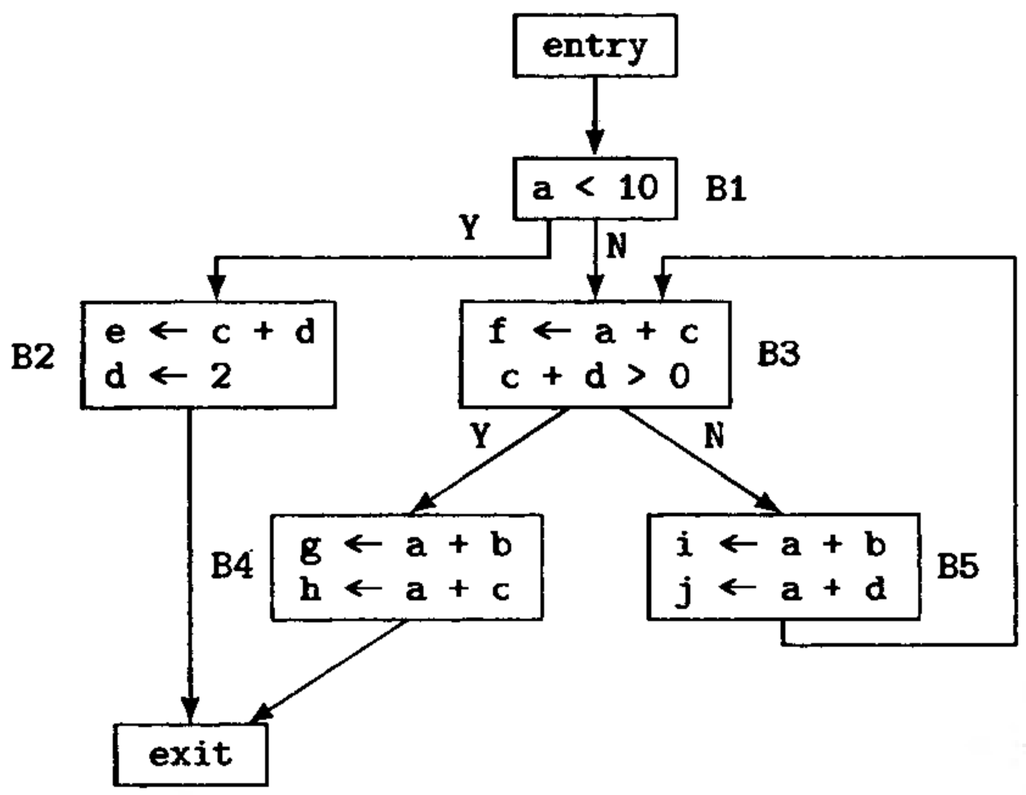


图13-33 代码提升的例子

现在，对于任何$i \neq$ entry，*VBEout*(i)中的每一个表达式*exp*都是可以提升的候选表达式。令S是满足如下条件的基本块j的集合：基本块j的必经结点是基本块i，基本块j中计算了*exp*，并且在基本块i末尾计算的*exp*能够无损地到达基本块j中*exp*的第一个计算。令*th*是一个新的临时变量。于是我们添加*th*←*exp*到基本块i的末尾（除非基本块i以条件分支指令结束，在这种情况下，我们将此赋值放置在条件分支指令的前面），并且用*th*替代集合S中每一个基本块j中*exp*的第一个使用。实现这种转换的ICAN代码如图13-34所示。代码中使用了下面的例程:

```
BinExp = Operand × Operator × Operand

procedure Hoist_Exps(nblocks,ninsts,Block,VBEout)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   VBEout: in integer ⟶ set of Binexp
begin
   i, j, k: integer
   S: set of (integer × integer)
   exp: BinExp
   th: Var
   s: integer × integer
   inst: MIRInst
```

图13-34 实现代码提升的ICAN例程

bbs.theithome.com

```
    for i := 1 to nblocks do
       for each exp ∈ VBEout(i) do
          S := ∅
          for j := 1 to nblocks do
             if !Dominate(i,j) then
                goto L1
             fi
             for k := 1 to ninsts[j] do
                inst := Block[j][k]
                if Exp_Kind(inst.kind) = binexp
                   & inst.opd1 = exp@1 & inst.opr = exp@2
                   & inst.opd2 = exp@3 & Reach(exp,Block,i,j,k) then
                   S ∪= {⟨j,k⟩}
                   goto L1
                fi
             od
L1:       od
          th := new_tmp( )
          append_block(i,ninsts,Block,⟨kind:binasgn,
             left:th,opd1:exp@1,opr:exp@2,opd2:exp@3⟩)
          for each s ∈ S do
             inst := Block[s@1][s@2]
             case inst.kind of
binasgn:        Block[s@1][s@2] := ⟨kind:valasgn,
                   left:inst.left,opd:⟨kind:var,val:th⟩⟩
binif:          Block[s@1][s@2] := ⟨kind:valif,
                   opd:⟨kind:var,val:th⟩,lbl:inst.lbl⟩
bintrap:        Block[s@1][s@2] := ⟨kind:valtrap,
                   opd:⟨kind:var,val:th⟩,trapno:inst.trapno⟩
             esac
          od
       od
    od
end    || Hoist_Exps
```

图13-34 （续）

1. Exp_Kind(*k*)返回种类为*k*的MIR指令中包含的表达式的种类(如4.7节所定义的)。

2. Reach(*exp*, Block, *i*, *j*, *k*)返回true，如果*exp*在基本块*i*末尾的定义可以没有损害地到达基本块*j*的第*k*条指令；否则返回false。

3. append_block(*i*, ninsts, Block, *inst*)插入指令*inst*在基本块*i*的末尾，或者，当基本块*i*的最后一条指令是条件分支指令时，插入该指令在条件分支指令的前面；在两种情况下，它都相应地更新ninsts[*i*]。

4. Dominate(*i*,*j*)返回true，如果基本块*i*是基本块*j*的必经结点；否则返回false。

于是对于我们的例子，我们提升B2和B3中c+d的计算到B1，提升B3和B4中a+c的计算到B3，也提升B4和B5中a+b的计算到B3，如图13-35所示。注意，局部公共子表达式删除现在可以用

```
t3 ← a + c
f ← t3
```

418 ~ 419

替代B3中a+c的冗余计算。

bbs.theithome.com

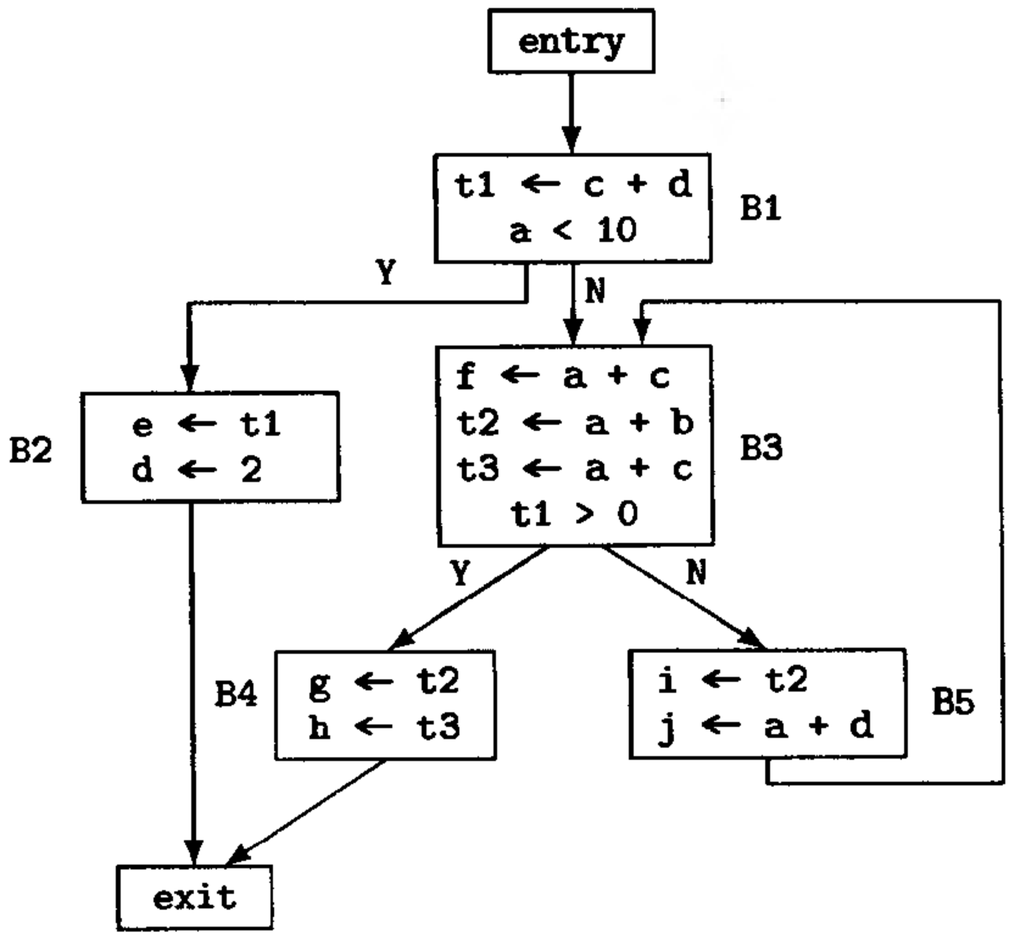


图13-35 对图13-33的例子程序执行代码提升的结果

13.6 小结

这一章的优化都与冗余计算的消除有关，并且都显式地或隐式地需要数据流分析。它们都可以有效地作用于中级中间代码或低级中间代码。

在这四种优化中，第一和第二种与第三种之间有着显著的重叠性。我们概括它们如下：

1. 第一种，即公共子表达式删除，它寻找在过程的一条路径上执行了两次以上的表达式，并删除第一次计算之后出现的计算，只要在这之间没有改变表达式的操作数。这种优化几乎总是能改善性能。

2. 第二种，即循环不变代码外提，它寻找对循环的每一次迭代都产生相同结果的表达式，并将这种表达式移到循环之外。它几乎总是能显著地改善性能，因为它发现并外提的多数是访问数组元素的地址计算。

3. 第三种，部分冗余删除，它移动至少是部分冗余的计算(即在流图的某些路径上计算多次的计算)到它们的最佳计算点，并完全删除冗余的计算。它包含了公共子表达式删除、循环不变代码外提，以及更多。

4. 最后一种，即代码提升，它寻找在从某一点开始的所有路径上都计算的表达式，并用在那一点的单个计算统一它们。它几乎总是减少程序所占的空间，但一般对运行时的性能没有改善，除非一个程序中存在有这种情况的众多实例。

420

我们一方面介绍了公共子表达式删除和循环不变代码外提，另一方面也介绍了部分冗余删除，这两种处理方法具有基本相同的效果和作用。部分冗余删除的现代公式表述也为其他可以共享部分数据流信息的优化提供了思考的参照模式和公式化表述的框架。可以预期在未来几年中这种优化会更频繁地被新的编译器所采用，并替代某些商业编译器中前面两种方法的组合。

如图13-36所示，冗余删除转换一般放置在优化处理过程的中间。

13.7 进一步阅读

部分冗余删除工作最初是由Morel和Renvoise [MorR79]开始的，后来他们又将它扩充到过程间的形式[MorR81]。扩充古典的部分冗余分析，以包含强度削弱和归纳变量化简的讨论见[Chow83]。

bbs.theithome.com

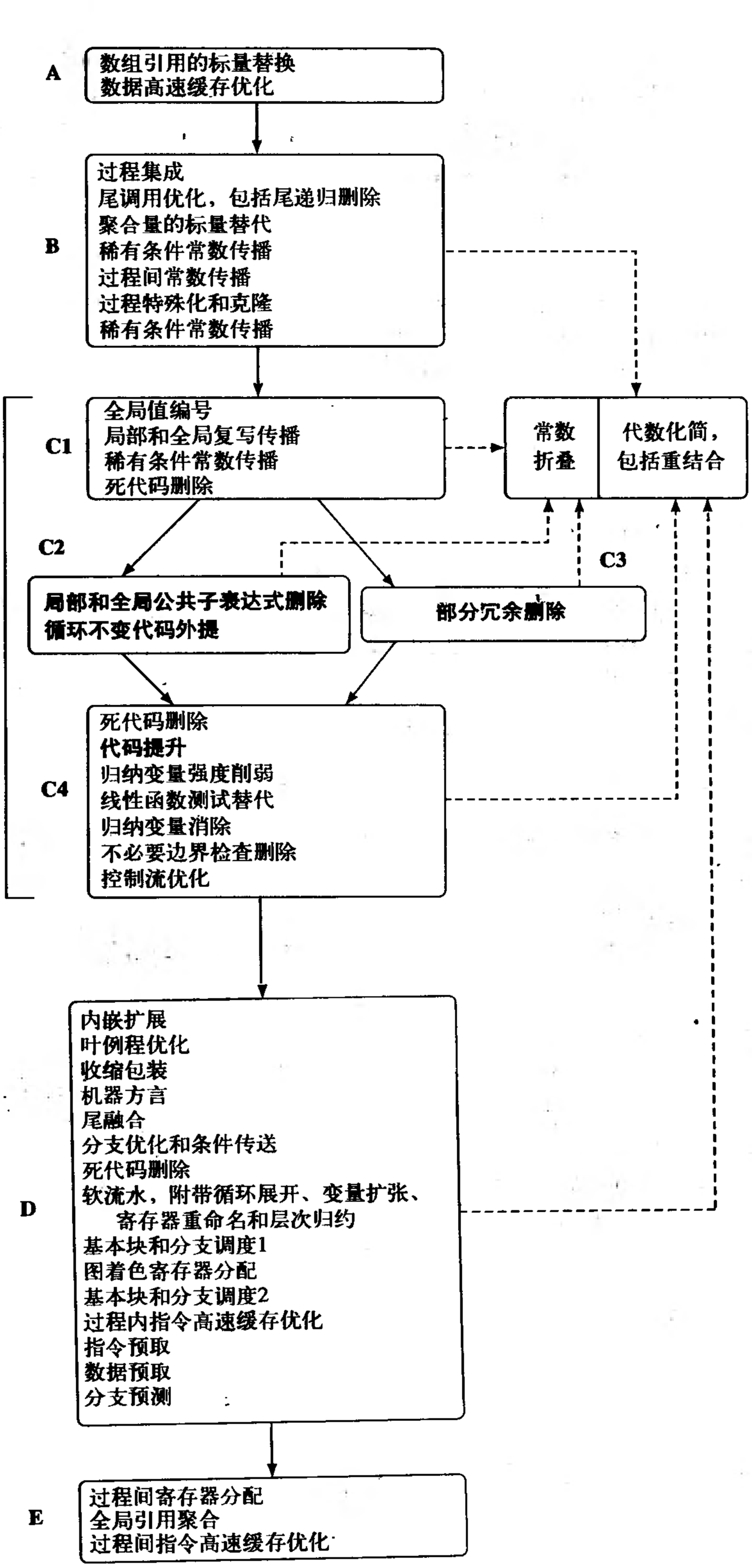


图13-36 在激进优化编译器中冗余相关优化(黑体字部分)的位置

bbs.theithome.com

超星公司制作请尊重作者版权

最近，Knoop、Rüthing和Steffen介绍了一种只需要单向数据流分析的形式[KnoR92]。13.3节介绍的边分割转换是由Dhamdhere[Dham88]开发的。扩充部分冗余删除以包含强度削弱的描述见[KnoR93]。对部分冗余删除方法的改善见Briggs和Cooper的描述[BriC94b],关于Cooper和Simpson的进一步改善见[CooS95c]和[Simp96]。

13.8 练习

13.1 如13.1节指出的，公共子表达式删除可能不一定总是有益的。给出(a)一组保证它能获益的判别条件清单，和(b)一组保证它不能获益的判别条件清单。(注意，存在着两组条件都不能保证的中间状态。)

13.2 用公式描述与可用表达式相反的数据流分析，即一种向后数据流问题，此问题中，一个表达式属于*EVAL*(*i*)，如果它在基本块*i*中被计值，并且表达式中的变量在此基本块入口到给定的计算之间都没有改变，其中的路径合并运算是交运算。是否存在这种分析对它有用的优化问题?

13.3 给出程序的一个无穷序列P_1，P_2，…，以及第12和13章中涉及的一组优化，使得对于每一个i，P_i源于这组优化的i次重复并比从$i-1$次重复能获得更多好处。

13.4 写出对整个过程执行向前替代的ICAN例程`Fwd_Subst(n,ninsts,Block)`。

13.5 阐明你如何修改图13-17中的`Mark_Invar()`和`Mark_Block()`来处理重结合。它对该算法的运行时间会有什么预期影响?

13.6 给出一个能清楚说明从里向外进行循环不变代码外提要优于从外向里进行的循环嵌套的例子。

13.7 用公式描述归约识别并根据它们执行循环不变代码外提的算法。使用函数`Reductor(`*opr*`)`确定操作符*opr*是否执行归约运算的操作符。

13.8 存储操作下移（downward store motion）是一种代码外提，它将循环内的存储操作移到循环的出口处。例如，图13-31的E部分的Fortran代码中，变量d是存储操作下移的候选——对它的赋值，除最后一次之外，其他都是无用的，因此只要这个循环至少执行一次，将它移到循环之后得到的结果就是相同的。设计一种方法检测那些可以作为存储操作下移的候选，并写出检测和移动它们的ICAN代码。存储操作下移对循环内的寄存器需求有什么影响?

421 ~ 423

13.9 写出图13-2给出的例程`Local_CSE()`的`listexp`情形对应的代码。

13.10 写出一个实现部分冗余移动的例程`Move_Partial_Redun()`。

bbs.theithome.com

超星公司制作 请尊重作者版权

第14章 循 环 优 化

本章涉及的优化是一些专门针对循环的优化，这些优化虽然也可用于别的结构，但作用于循环时最有效。它们可以作用于中级中间代码（如MIR），也可以作用于低级中间代码（如LIR）。

这些优化直接应用于Fortran和Pascal的规则的源语言循环结构，但对于C之类的语言，则需要定义能应用这些优化的循环子集。具体地，我们定义C的*规则循环*（well-behaved loop）（参见图14-1中的代码）是这种循环：其中*exp*1是给整值变量*i*赋值的表达式，*exp*2是*i*与一个常数相比较的表达式，*exp*3是从*i*增加或减少一个常数的表达式，*stmt*中不含对*i*的赋值。类似的定义也适应于由if和goto形成的规则循环。

```
for (exp1;exp2;exp3)
    stmt
```

图14-1 C的for循环形式

14.1 归纳变量优化

归纳变量（induction variable）的最简单形式是那种在程序的某个部分，一般是循环中，其后继值形成一个算术级数的变量。循环迭代通常用一个称作循环控制变量的整值变量来计数，这个整值变量每迭代一次便增加（或减少）一个常量。循环中常常还有另外一些变量，最明显的是下标值和数组元素的地址，它们也遵循与循环控制变量类似的模式，尽管它们可能具有不同的初值、终值和迭代方向。

例如，图14-2a中的Fortran 77循环是用变量i来计数的，它的初值为1，循环每迭代一次它便增加1，终值为100。相应地，给a(i)赋值的表达式的初值为200，循环每迭代一次它便减少2，终值为2。a(i)的地址的初值是(addr a)，循环每迭代一次，它的值增加4，终值为(addr a)+396。这3个级数中至少有一个是多余的。具体而言，如果我们用一个临时变量t1取代202-2*i的值，则可以将这个循环转换成如图14-2b所示的形式，或如图14-3a所示的与它等价的MIR代码。这是一个对归纳变量进行强度削弱优化的例子：它用一个减法取代了一个乘法（和一个减法）（参见14.1.2节）。现在i只用来存储迭代次数，并且

```
addr a(i) = (addr a) + 4 * i - 4
```



```
integer a(100)
do i = 1,100
   a(i) = 202 - 2 * i
enddo
```

a)

```
integer a(100)
t1 = 202
do i = 1,100
   t1 = t1 - 2
   a(i) = t1
enddo
```

b)

图14-2 一个Fortran 77归纳变量的例子。在a)中，赋给a(i)的值每次迭代减少2。它可以如b)所示，用一个加法取代这个处理中的乘法，并用变量t1来代表这个值

所以，我们可以用一个临时变量来替换循环控制变量i，此临时变量的初值是(addr a)，增量是4，终值是(addr a)+396。结果得到的MIR形式如图14-3b所示。

这里介绍的所有归纳变量优化的效果都可通过常数传播而得到进一步的改善。

在执行归纳变量优化时应记住的一个问题是，有些体系结构提供了基地址加变址的寻址方式，其中变址在与基地址相加之前可以按比例伸缩2、4或8倍（如PA-RISC和Intel 386体系结构）。而有些体系结构提供“自动修改”方式，即在存储器引用之前或之后，可以将基址和变址之和存储到基址寄存器中（如PA-RISC、POWER和VAX）。系统中存在这种指令可能

bbs.theithome.com

有利于归纳变量的删除，因为，在两个给定的归纳变量中选择删除哪一个时，一个可能对伸缩或自动修改基地址寄存器敏感，而另一个则不敏感。此外，由于类似的原因，PA-RISC的先加后分支指令以及POWER的先减后分支条件指令也有利于线性函数测试替换（参见图14-4和14.1.4节）。

```
        t1 ← 202
        i ← 1




L1:     t2 ← i > 100
        if t2 goto L2
        t1 ← t1 - 2
        t3 ← addr a
        t4 ← t3 - 4
        t5 ← 4 * i
        t6 ← t4 + t5
        *t6 ← t1
        i ← i + 1
        goto L1
L2:
```

a)

```
        t1 ← 202
        t3 ← addr a
        t4 ← t3 - 4
        t5 ← 4
        t6 ← t4
        t7 ← t3 + 396
L1:     t2 ← t6 > t7
        if t2 goto L2
        t1 ← t1 - 2




        t6 ← t4 + t5
        *t6 ← t1
        t5 ← t5 + 4
        goto L1
L2:
```

b)

图14-3 a) 是图14-2b中循环的MIR形式，b) 是同一段代码消除了归纳变量i，外提了循环不变赋值t3 ← addr a和t4←t3- 4，对t5执行了强度削弱，并删除了归纳变量i之后的形式

```
        t1 ← 202
        t3 ← addr a
        t4 ← -396
        t5 ← t3 + 396
        t2 ← t4 > 0
        if t2 goto L2
L1:     t1 ← t1 - 2
        t6 ← t4 + t5
        *t6 ← t1
        t4 ← t4 + 4
        t2 ← t4 <= 0
        if t2 goto L1
L2:
```

图14-4 对图14-3b代码中t4的值进行偏移，使循环能用0作为结束测试条件的结果。其中也进行了循环倒置（参见18.5节）

14.1.1 识别归纳变量

为了有利于识别，常常将归纳变量分为基本或基础归纳变量（basic or fundamental induction variable）和依赖归纳变量（dependent induction variable）。基本归纳变量是在循环的每一次迭代中显式地加或减一个相同常量的变量；依赖归纳变量的修改或计算则以一种较为复杂的方式。例如，在图14-2a中，i是一个基本归纳变量，而表达式200-2*i的值和a(i)的地址是依赖归纳变量。与之对比的是，在图14-3b所示的这段代码经扩展和转换后的MIR形式中，归纳变量i已经被删除，t5则经过了强度削弱。t1和t5（它们分别包含200-2*i的值和a(i)相对a(0)地址的地址偏移）都变成基本归纳变量了。

426 ~ 427

为了识别归纳变量，我们一开始将循环中的所有变量都作为候选，并给找到的每一个归纳变量j确定一个形如$j = b * biv + c$的线性方程，该方程将j的值与循环内的biv联系起来，其中biv是基本归纳变量，b和c是常量（它们可以是实际的常数，也可以是已经识别出的循环不变量）；biv、b和c初始值都为nil。在其线性方程中具有相同基本归纳变量的那些归纳变量组成了一个类（class），这个基本归纳变量叫做它们的基（basis）。每当我们识别出一个变量j是潜在的归纳变量时，便填充它的线性方程。

归纳变量识别可以通过依次查看循环体中的指令来进行，也可用公式表示为数据流问题，我们采用的是前一种方法。为了标识基本归纳变量，我们首先寻找这种变量：它们在循环中仅有的赋值是形如$i \leftarrow i+d$或$i \leftarrow d+i$的赋值，其中d是（正或负的）循环常量。对于这样的一个变量i，

bbs.theithome.com

其线性方程简单地是$i=1*i+0$，并且i是一个归纳变量类的基。如果循环中对i有两个以上的这种赋值，我们将这个基本归纳变量分成若干个，每个赋值对应一个。例如，对于图14-5a中的代码，我们将i分割成两个归纳变量i和t1，如图14-5b所示。一般地，给定如图14-6a所示的一个两次赋值的基本归纳变量，其转换后的代码如图14-6b所示。这种方法可直接推广到三个以上赋值的情形。

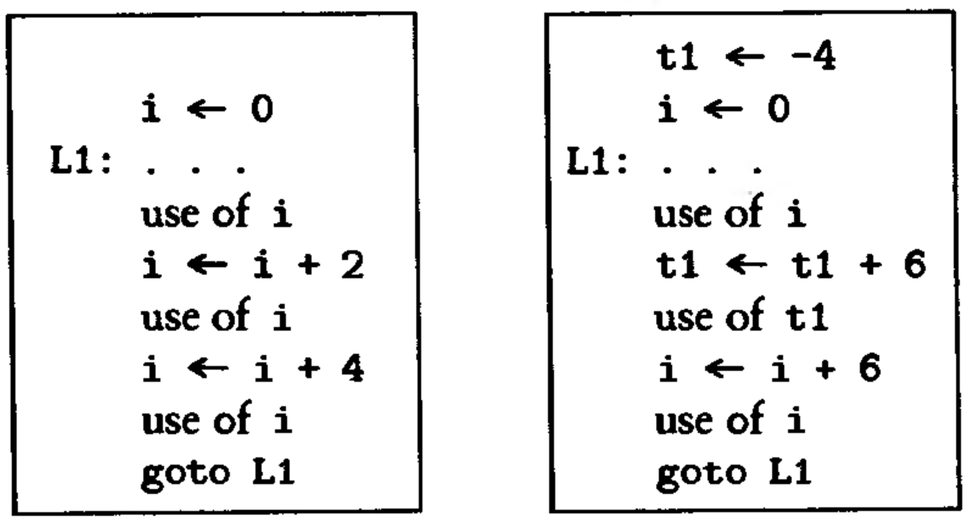


图14-5 分割一个有两次赋值的基本归纳变量a) 为两个归纳变量b) 的例子

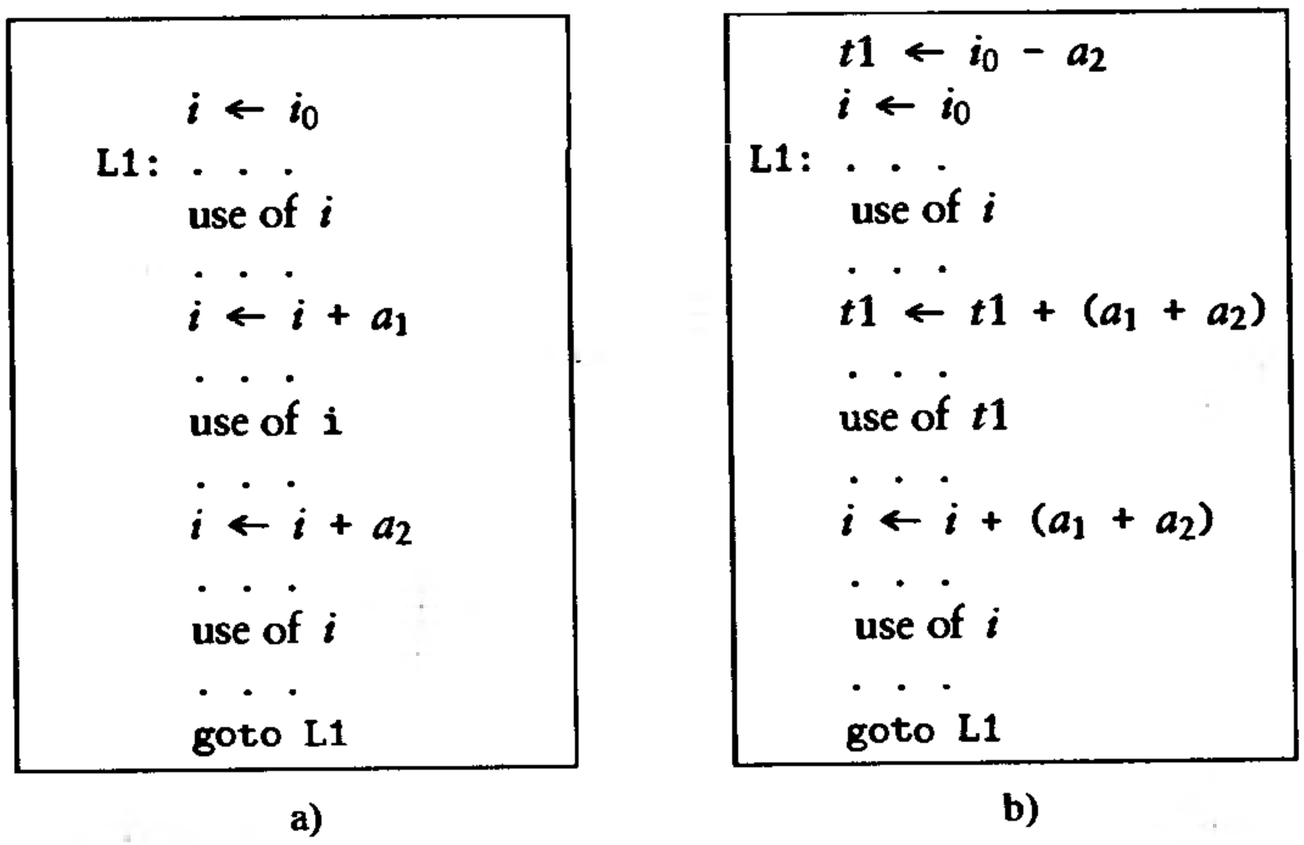


图14-6 分割一个有两次赋值的基本归纳变量a) 为两个归纳变量b) 的样板

接着，我们重复地考察循环体内的每一条指令，寻找满足如下条件的变量j：j出现在赋值的左部，且这个赋值具有表14-1所列的形式之一，其中i是一个归纳变量（基本的或依赖的），e是一个循环常量。如果i是基本归纳变量，则j属于i类，并且它的线性方程可直接从定义它的赋值形式得出；例如，对于$j \leftarrow e*i$，j的线性方程是$j=e*i+0$。如果i不是基本归纳变量，则它属于线性方程为$i=b_1*i_1+c_1$的某个基本归纳变量i_1的类，于是j也属于i_1类，并且它的线性方程（再次假定定义它的赋值是$j=e*i$）是$j=(e*b_1)*i_1+e*c_1$。对依赖归纳变量i还有两个进一步的要求。首先，在循环中，在对i的赋值与对j的赋值之间没有对i_1的赋值，因为有赋值的话会影响j和i_1之间的关系，可能使得j根本不是一个归纳变量；其次，i必须没有循环外的定值到达j的这个定值。到达定值或ud链可用来检查后者。

表14-1 可以产生依赖归纳变量的赋值类型

$j \leftarrow i*e$
$j \leftarrow e*i$
$j \leftarrow i+e$
$j \leftarrow e+i$
$j \leftarrow i-e$
$j \leftarrow e-i$
$j \leftarrow -i$

bbs.theithome.com

版权所有 侵权必究

如果有对*j*的多个赋值，但是它们都具有表14-1给出的形式之一，则我们将*j*分成若干个归纳变量，每个归纳变量对应一个赋值且有它自己的线性方程。

形如

$$j \leftarrow i / e$$

的赋值也可以改变成适合所要求的条件：如果我们首先展开循环体*f*次，*f*是*e*的倍数（见17.4.3节），则*j*的线性方程是$j = (f / e) * i + 0$，其中假设*i*是基本归纳变量。

428
~
429

对已经填充过的变量*j*，如果还需要在它的线性方程中填充不同的基本归纳变量、*b*值或*c*值，我们也如前面所述一样，将这个归纳变量分为两个。

如果对一个潜在归纳变量的修改出现在一个条件转移的分支上，则在另一个分支上也必须存在与之对称的修改。

图14-7中的例程Find_IVs()和图14-18中的辅助例程实现了上面所述的大部分功能。它们忽略了循环中含多个赋值的归纳变量的情形，以及在条件转移的一个分支中定值的归纳变量在另一个分支也必须有这样一个定值与之平衡的要求。它们使用了下述几个函数：

1. Loop_Const(*opnd*, *bset*, nblocks, Block)返回true，如果*opnd*是一个常数，或者是由*bset*中的基本块所组成的循环的一个循环常量；否则返回false。

2. Assign_Between(*var*, *i*, *j*, *k*, *l*, *bset*, nblocks, Block)返回true，如果变量*var*在指令Block[*i*][*j*]和指令Block[*k*][*l*]之间的某条路径上被赋值；否则返回false。

3. No_Reach_Defs(*var*, *i*, *j*, *bset*, nblocks, Block)返回true，如果循环外没有对变量*var*定值并到达指令Block[*i*][*j*]的指令；否则返回false。

```
IVrecord: record {tiv,biv: Var,
                  blk,pos: integer,
                  fctr,diff: Const}
IVs: set of IVrecord

procedure Find_IVs(bset,nblocks,ninsts,Block)
   bset: in set of integer
   nblocks: in integer
   ninsts: in array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of MIRInst
begin
   inst: MIRInst
   i, j: integer
   var: Var
   change: boolean
   ops1, ops2: enum {opd1,opd2}
   iv: IVrecord
   IVs := ∅
   for each i ∈ bset do
      for j := 1 to ninsts[i] do
         inst := Block[i][j]
         case inst.kind of
            || search for instructions that compute fundamental induction
            || variables and accumulate information about them in IVs
binasgn:    if IV_Pattern(inst,opd1,opd2,bset,nblocks,Block)
               ∨ IV_Pattern(inst,opd2,opd1,bset,nblocks,Block) then
               IVs ∪= {⟨tiv:inst.left,blk:i,pos:j,fctr:1,
                  biv:inst.left,diff:0⟩}
```

图14-7 识别归纳变量的代码

bbs.theithome.com

```
                fi
default:    esac
         od
      od
      repeat
         change := false
         for each i ∈ bset do
            for j := 1 to ninsts[i] do
               inst := Block[i][j]
               case inst.kind of
                  || check for dependent induction variables
                  || and accumulate information in the IVs structure
binasgn:          change := Mul_IV(i,j,opd1,opd2,
                     bset,nblocks,ninsts,Block)
                  change V= Mul_IV(i,j,opd2,opd1,
                     bset,nblocks,ninsts,Block)
                  change V= Add_IV(i,j,opd1,opd2,
                     bset,nblocks,ninsts,Block)
                  change V= Add_IV(i,j,opd2,opd1,
                     bset,ninsts,Block)
                  . . .
default:       esac
            od
         od
      until !change
   end    || Find_IVs

   procedure IV_Pattern(inst,ops1,ops2,bset,nblocks,Block)
      returns boolean
      inst: in Instruction
      ops1,ops2: in enum {opd1,opd2}
      bset: in set of integer
      nblocks: in integer
      Block: in array [1··nblocks] of array [··] of MIRInst
   begin
      return inst.left = inst.ops1.val & inst.opr = add
         & Loop_Const(inst.ops2,bset,nblocks,Block)
         & !∃iv ∈ IVs (iv.tiv = inst.left)
   end    || IV_Pattern
```

图14-7 （续）

```
procedure Mul_IV(i,j,ops1,ops2,bset,nblocks,ninsts,Block) returns boolean
   i, j: in integer
   ops1, ops2: in enum {opd1,opd2}
   bset: in set of integer
   nblocks: in integer
   ninsts: in array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of MIRInst
begin
   inst := Block[i][j]: MIRInst
   iv1, iv2: IVrecord
   if Loop_Const(inst.ops1,bset,nblocks,Block)
      & inst.opr = mul then
      if ∃iv1 ∈ IVs (inst.ops2.val = iv1.tiv
         & iv1.tiv = iv1.biv & iv1.fctr = 1
```

图14-8 识别归纳变量时使用的辅助例程

bbs.theithome.com

```
      & iv1.diff = 0) then
      IVs ∪= {⟨tiv:inst.left,blk:i,pos:j,
         fctr:inst.ops1.val,biv:iv1.biv,diff:0⟩}
   elif ∃iv2 ∈ IVs (inst.ops2.val = iv2.tiv) then
      if !Assign_Between(iv2.biv,i,j,iv2.blk,iv2.pos,
         bset,nblocks,Block)
         & No_Reach_Defs(inst.ops2.val,i,j,bset,
         nblocks,Block) then
         IVs ∪= {⟨tiv:inst.left,blk:i,pos:j,
            fctr:inst.ops1.val*iv2.fctr,biv:iv2.biv,
            diff:inst.ops1.val*iv2.diff⟩}
      fi
   fi
   return true
fi
return false
end    || Mul_IV
procedure Add_IV(i,j,ops1,ops2,bset,nblocks,ninsts,Block) returns boolean
   i, j: in integer
   ops1, ops2: in enum {opd1,opd2}
   bset: in set of integer
   nblocks: in integer
   ninsts: in array [··] of integer
   Block: in array [··] of array [··] of MIRInst
begin
   inst := Block[i][j]: in MIRInst
   iv1, iv2: IVrecord
   if Loop_Const(inst.ops1,bset,nblocks,Block)
      & inst.opr = add then
      if ∃iv1 ∈ IVs (inst.ops2.val = iv1.tiv
         & iv1.tiv = iv1.biv & iv1.fctr = 1
         & iv1.diff = 0) then
         IVs ∪= {⟨tiv:inst.left,blk:i,pos:j,
            fctr:1,biv:iv1.biv,diff:inst.ops1.val⟩}
      elif ∃iv2 ∈ IVs (inst.ops2.val = iv.tiv) then
         if !Assign_Between(iv2.biv,i,j,iv2.blk,iv2.pos,
            bset,nblocks,Block)
            & No_Reach_Defs(inst.ops2.val,i,j,bset,
            nblocks,Block) then
            IVs ∪= {⟨tiv:inst.left,blk:i,pos:j,
               fctr:iv2.fctr,biv:iv2.biv,
               diff:iv2.diff+inst.ops1.val⟩}
         fi
      fi
      return true
   fi
   return false
end    || Add_IV
```

图14-8 （续）

这些例程还使用了由IVrecords组成的集合IVs，其中IVrecords记录归纳变量、它们的线性方程以及定义它们的语句所在的基本块和位置。图14-7中声明的记录

<tiv:*var*1, blk:*i*, pos:*j*, fctr:*c*1, biv:*var*, diff:*c*2>

描述指令Block[*i*][*j*]中定义的一个归纳变量*var*1，它属于基本归纳变量*var*的类，并且具有线性方程

$$var1 = c1 * var + c2$$

bbs.theithome.com

版权所有 侵权必究

注意，循环内为常数值的表达式，如

```
inst.opd1.val * iv2.fctr
```

和

```
iv2.diff + inst.opd1.val
```

可能有编译时不知道的值——它们可能仅仅是循环常量。在这种情况下，我们需要将这种循环常量表达式记录在IVrecords中，并在循环前置块中生成计算其值的指令。

430 ~ 432

作为这种归纳变量识别方法的一个小例子，考虑图14-3a的MIR代码。我们遇到的第一个基本归纳变量是t1，它在循环内惟一的赋值是t1←t1-2；它的线性方程是t1=1*t1+0。惟一的另一个基本归纳变量是i，它的线性方程是i=1*i+0。接下来我们发现t5是一个具有线性方程t5=4*i+0的依赖归纳变量。最后，我们识别出t6是另一个属于i类的依赖归纳变量（因为(addr a)，即t3的值，是循环常量），它的线性方程是t6=4*i+(addr a)。

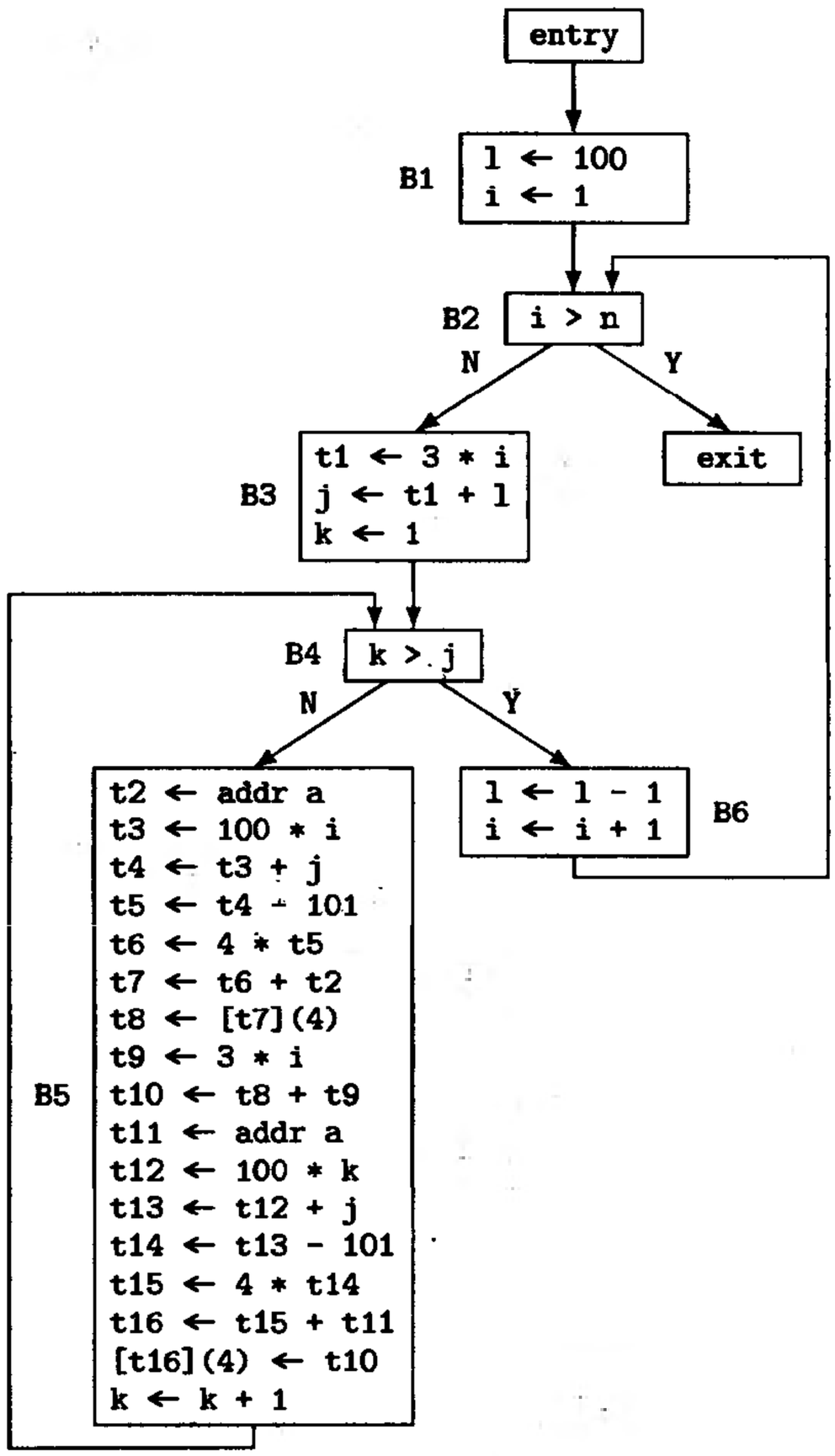


图14-9 归纳变量识别的第二个例子

作为归纳变量识别过程的另一个例子，考虑图14-9中的MIR代码。我们从基本块B4和B5组成的内层循环开始。这个循环中只有一个基本归纳变量k，因此IVs（它初始为空）变为

```
IVs = {<tiv:k, blk:B5, pos:17, fctr:1, biv:k, diff:0>}
```

接着识别出t12是k类中的一个归纳变量，于是IVs变成

bbs.theithome.com

```
IVs = {<tiv:k,  blk:B5, pos:17, fctr:1,   biv:k, diff:0>,
       <tiv:t12, blk:B5, pos:11, fctr:100, biv:k, diff:0>}
```

然后识别出t13是k类中的一个归纳变量，因此有

```
IVs = {<tiv:k,  blk:B5, pos:17, fctr:1,   biv:k, diff:0>,
       <tiv:t12, blk:B5, pos:11, fctr:100, biv:k, diff:0>,
       <tiv:t13, blk:B5, pos:12, fctr:100, biv:k, diff:j>}
```

433 ~ 434

临时变量t14、t15和t16也都被识别出是k类中的归纳变量，于是，我们最终得到

```
IVs = {⟨tiv:k,  blk:B5,pos:17,fctr:1,  biv:k,diff:0⟩,
       ⟨tiv:t12,blk:B5,pos:11,fctr:100,biv:k,diff:0⟩,
       ⟨tiv:t13,blk:B5,pos:12,fctr:100,biv:k,diff:j⟩,
       ⟨tiv:t14,blk:B5,pos:13,fctr:100,biv:k,diff:j-101⟩,
       ⟨tiv:t15,blk:B5,pos:14,fctr:400,biv:k,diff:4*j-404⟩,
       ⟨tiv:t16,blk:B5,pos:15,fctr:400,biv:k,
           diff:(addr a)+4*j-404⟩}
```

注意，t2、t3、t4、t5、t6、t7、t9和t11都是内层循环的循环不变量，但我们在这里不讨论它们，因为它们之中除了t11之外都与定义归纳变量无关。

现在，在由基本块B2、B3、B4、B5和B6组成的外层循环中，变量1是第一个被识别出来的归纳变量，它设置

```
IVs = {<tiv:1, blk:B6, pos:1, fctr:1, biv:1, diff:0>}
```

i是下一个识别出的归纳变量，导致

```
IVs = {<tiv:1,  blk:B6, pos:1, fctr:1, biv:1, diff:0>,
       <tiv:i, blk:B6, pos:2, fctr:1, biv:i, diff:0>}
```

之后，B3中的t1被加入进来，产生

```
IVs = {⟨tiv:1, blk:B6,pos:1,fctr:1,biv:1,diff:0⟩,
       ⟨tiv:i, blk:B6,pos:2,fctr:1,biv:i,diff:0⟩,
       ⟨tiv:t1,blk:B3,pos:1,fctr:3,biv:i,diff:0⟩}
```

现在，注意j也是一个归纳变量，但很可能多数编译器都发现不了这一事实。需要使用代数或符号算术才能确定下面的事实：在基本块B1的出口，我们有1+i=101，并且修改1和i的惟一基本块B6保持了这个关系，因此

```
j = t1 + 1 = 3*i + 1 = 2*i + (i + 1) = 2*i + 101
```

发现不了这一事实是不幸的，因为假如能够识别出j是归纳变量的话，内层循环的几个循环不变量（上面提到的）就可以也是外层循环的归纳变量。

一旦识别出所有的归纳变量，我们便可对它们施加三种重要的转换：强度削弱、归纳变量删除和线性函数测试替换。

14.1.2 强度削弱

强度削弱（strength reduction）用代价较小的运算，如加和减，替代代价较大的运算，如乘和除。它是应用于计算机程序的一种特殊情形的有限差分方法。例如，序列

0, 3, 6, 9, 12, …

的一阶差分（即相邻元素之间的差）都由3组成，因此可以写为$s_i = 3 * i$，$i = 0, 1, 2, \cdots$，或写为

435 $s_{i+1} = s_i + 3$，其中$s_0 = 0$。第二种形式是强度削弱版本——我们用加法替代乘法。类似地，序列

0, 1, 4, 9, 16, 25, …

bbs.theithome.com

的一阶差分是

1, 3, 5, 7, 9, …

且二阶差分都由2组成。它可以写成$s_i = i^2$，$i = 0, 1, 2, 3, \cdots$；或写成$s_{i+1} = s_i + 2*i + 1$，其中$s_0 = 0$；或写成$s_{i+1} = s_i + t_i$，其中$t_{i+1} = t_i + 2$，$s_0 = 0$且$t_0 = 1$。这里，在二次有限差分运算之后，我们已将一系列的平方计算简化成用两个加法运算替代一个平方运算。强度削弱并不只局限于用加替代乘和用增量操作替代加操作，Allen和Cocke [AllC81]讨论了与它有关的一系列应用，如用乘法运算替代指数运算，用减法替代除法和求模运算。但是，我们这里只讨论简单的强度削弱，因为它们使用最频繁，并且通常从它们获得的好处也最大。其他情形的处理方法大体上是类似的，在14.4节给出的引文中可以找到这些方法。

为了对循环中已识别出的归纳变量执行强度削弱，我们依次处理每一类归纳变量。

1. 令*i*是基本归纳变量，令*j*是具有线性方程$j = b * i + c$的*i*类归纳变量。

2. 分配一个新临时变量*tj*，并用$j \leftarrow tj$替代循环中对*j*的单一赋值。

3. 在循环中每一个对*i*的赋值$i \leftarrow i + d$之后，插入赋值$tj \leftarrow tj + db$，其中*db*是常数值表达式$d * b$的值（如果这个值不是实际的常数，而只是一个循环常量，分配一个新的临时变量*db*，并将赋值$db \leftarrow d * b$放在循环的前置块中。）

4. 将一对赋值

$tj \leftarrow b*i$

$tj \leftarrow tj + c$

放置在前置块的末尾，以保证*tj*被适当地初始化。

5. 用*tj*替代循环中的每一个*j*。

6. 最后，以线性方程$tj = b * i + c$将*tj*加入到*i*的归纳变量类中。

图14-10是实现这一算法的例程Strength_Reduce()。如果对基本块*i*中的指令*j*已执行过强度削弱，则数组SRdone有SRdone[*i*][*j*]=true；否则为false。Strength_Reduce ()用到了如下两个函数：

436

1. Insert_after(*i*, *j*, ninsts, Block, *inst*)将指令*inst*插入到基本块Block[*i*]的第*j*条指令之后，并更新表示程序的数据结构以反映这样做的结果（参见图4-14）。

2. Append_Preheader(*bset*, ninsts, Block, *inst*)将指令*inst*插入到基本块Block[*i*]的末尾，基本块*i*是由*bset*中基本块组成的循环的前置块，并更新表示程序的数据结构以反映这样做的结果。

```
procedure Strength_Reduce(bset,nblocks,ninsts,Block,IVs,SRdone)
   bset: in set of integer
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   IVs: inout set of IVrecord
   SRdone: out array [1··nblocks] of [··] of boolean
begin
   i, j: integer
   tj, db: Var
   iv, iv1, iv2: IVrecord
   inst: MIRInst
   for each i ∈ bset do
      for j := 1 to ninsts[i] do
```

图14-10 归纳变量强度削弱代码

bbs.theithome.com

```
            SRdone[i][j] := false
         od
      od
      || search for uses of induction variables
      for each iv1 ∈ IVs (iv1.fctr = 1 & iv1.diff = 0) do
         for each iv2 ∈ IVs (iv2.biv = iv1.biv
            & iv2.tiv ≠ iv2.biv) do
            tj := new_tmp( ); db := new_tmp( )
            i := iv2.blk; j := iv2.pos
            SRdone[i][j] := true
            || and split their computation between preheader and
            || this use, replacing operations by less expensive ones
            append_preheader(bset,ninsts,Block,⟨kind:binasgn,
               left:db,opr:mul,opd1:⟨kind:const,val:iv1.diff⟩,
               opd2:⟨kind:const,val:iv2.fctr⟩⟩)
            append_preheader(bset,ninsts,Block,⟨kind:binasgn,
               left:tj,opr:mul,opd1:⟨kind:const,val:iv2.fctr⟩,
               opd2:⟨kind:var,val:iv2.biv⟩⟩)
            append_preheader(bset,ninsts,Block,⟨kind:binasgn,
               left:tj,opr:add,opd1:⟨kind:var,val:tj⟩,
               opd2:⟨kind:const,val:iv2.diff⟩⟩)
            insert_after(i,j,ninsts,Block,⟨kind:binasgn,left:tj,opr:add,
               opd1:⟨kind:var,val:tj⟩,opd2:⟨kind:var,val:db⟩⟩)
         IVs ∪= {⟨tiv:tj,blk:i,pos:j+1,fctr:iv2.fctr*iv1.fctr,biv:iv2.biv,
            diff:iv2.diff⟩}
         for each i ∈ bset do
            if iv1.tiv = iv2.tiv then
               for each iv ∈ IVs do
                  IVs := (IVs - {iv}) ∪ {⟨tiv:iv.tiv,
                     blk:iv.blk,pos:iv.pos,fctr:iv.fctr,
                     biv:tj,diff:iv.diff⟩}
               od
            fi
            for j := 1 to ninsts[i] do
               inst := Block[i][j]
               case Exp_Kind(inst.kind) of
binexp:           if inst.opd1.val = iv2.tiv then
                     Block[i][j].opd1 := ⟨kind:var,val:tj⟩
                  fi
                  if inst.opd2.val = iv2.tiv then
                     Block[i][j].opd2 := ⟨kind:var,val:tj⟩
                  fi
unexp:            if inst.opd.val = iv2.tiv then
                     Block[i][j].opd := ⟨kind:var,val:tj⟩
                  fi

listexp:          for j := 1 to |inst.args| do
                     if inst.args↓i@1.val = iv2.tiv then
                        Block[i][j].args↓i@1 := ⟨kind:var,val:tj⟩
noexp:            esac
            od
         od
      od
   od
end    || Strength_Reduce
```

图14-10 （续）

bbs.theithome.com

对于图14-3a的MIR例子，这里将它重新给出在图14-11a中。我们首先考虑归纳变量t5，它的线性方程是t5 =4*i+0。我们分配一个新临时变量t7，并用t5←t7替代对t5的赋值。然后将t7←t7+4插入在i←i+1之后，将t7←4插入在前置块中。最后为i类中的归纳变量t7创建线性方程t7=4*i+0，并将结果得到的记录放置到IVs中，这生成图14-11b中的代码。对t6以线性方程t6=4*i+t4执行同样的转换，并从循环中删除循环不变量t3和t4，从而得到图14-12中的代码。注意，这些转换已增大了代码的体积——这个循环现在有11条指令，而不是开始时的9条。我们的下一个任务是删除归纳变量，它能够恢复代码的体积，也常常能使代码得到改善。

```
    t1 ← 202
    i ← 1

L1: t2 ← i > 100
    if t2 goto L2
    t1 ← t1 - 2
    t3 ← addr a
    t4 ← t3 - 4
    t5 ← 4 * i
    t6 ← t4 + t5
    *t6 ← t1
    i ← i + 1

    goto L1
L2:
```

a)

```
    t1 ← 202
    i ← 1
    t7 ← 4
L1: t2 ← i > 100
    if t2 goto L2
    t1 ← t1 - 2
    t3 ← addr a
    t4 ← t3 - 4
    t5 ← t7
    t6 ← t4 + t5
    *t6 ← t1
    i ← i + 1
    t7 ← t7 + 4
    goto L1
L2:
```

b)

图14-11 a) 图14-3a中循环的MIR形式，b) 同样的代码对归纳变量t5执行了强度削弱后的形式

```
    t1 ← 202
    i ← 1
    t7 ← 4
    t3 ← addr a
    t4 ← t3 - 4
    t8 ← t4 + t7
L1: t2 ← i > 100
    if t2 goto L2
    t1 ← t1 - 2
    t5 ← t7
    t6 ← t8
    *t6 ← t1
    i ← i + 1
    t8 ← t8 + 4
    t7 ← t7 + 4
    goto L1
L2:
```

图14-12 图14-11b中代码删除循环不变量t3和t4，并对t6进行强度削弱后的结果

对于前一节的第二个例子，即图14-9，我们在图14-13中给出的是它的内层循环，以及内层循环的前置块。它的归纳变量记录集合（与在前一节计算的一样，但对位置重新进行了编号，以反映被删除的循环不变量）如下:

```
IVs = {⟨tiv:k,  blk:B5,pos:9,fctr:1,  biv:k,diff:0⟩,
       ⟨tiv:t12,blk:B5,pos:3,fctr:100,biv:k,diff:0⟩,
       ⟨tiv:t13,blk:B5,pos:4,fctr:100,biv:k,diff:j⟩,
       ⟨tiv:t14,blk:B5,pos:5,fctr:100,biv:k,diff:j-101⟩,
       ⟨tiv:t15,blk:B5,pos:6,fctr:400,biv:k,diff:4*j-404⟩,
       ⟨tiv:t16,blk:B5,pos:7,fctr:400,biv:k,
           diff:(addr a)+4*j-404⟩}
```

437 ~ 439

这个算法初始设置

```
iv1 = <tiv:k,   blk:B5, pos:9, fctr:1,   biv:k, diff:0>
iv2 = <tiv:t12, blk:B5, pos:3, fctr:100, biv:k, diff:0>
```

并分别分配临时变量t17和t18作为tj和db的值，设置i=B5、j=3和SRdone[B5][3]=true。接着，它将如下指令添加到前置块（基本块B3）的末尾:

```
t18 ← 100 * 1
t17 ← 100 * k
t17 ← t17 + 0
```

将指令

```
t17←t17 + t18
```

添加到基本块B5的末尾，并设置

bbs.theithome.com

```
IVs = {⟨tiv:k,  blk:B5,pos:9, fctr:1,  biv:k,diff:0⟩,
       ⟨tiv:t12,blk:B5,pos:3, fctr:100,biv:k,diff:0⟩,
       ⟨tiv:t13,blk:B5,pos:4, fctr:100,biv:k,diff:j⟩,
       ⟨tiv:t14,blk:B5,pos:5, fctr:100,biv:k,diff:j-101⟩,
       ⟨tiv:t15,blk:B5,pos:6, fctr:400,biv:k,diff:4*j-404⟩,
       ⟨tiv:t16,blk:B5,pos:7, fctr:400,biv:k,
           diff:(addr a)+4*j-404⟩,
       ⟨tiv:t17,blk:B5,pos:10,fctr:100,biv:k,diff:100⟩}
```

最后，该例程用t17替代所有的t12。注意，插入到前置块中的两条指令（即t18←100*i和t17←t17+0）是不必要的（关于删除它们的方法参见练习14.3），设置t12的指令保留在基本块B5中（归纳变量删除将去掉它）。结果在图14-14中给出。

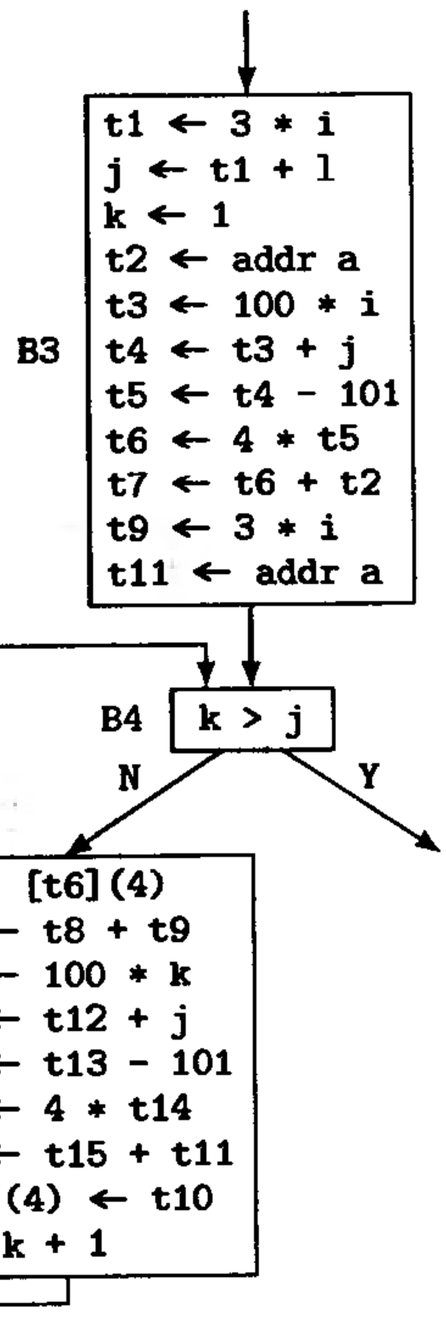


图14-13 从我们第二个例子（图14-9）的内层循环删除了循环不变量，并删除了外层循环但保留了其中的B3的结果，B3是内层循环的前置块

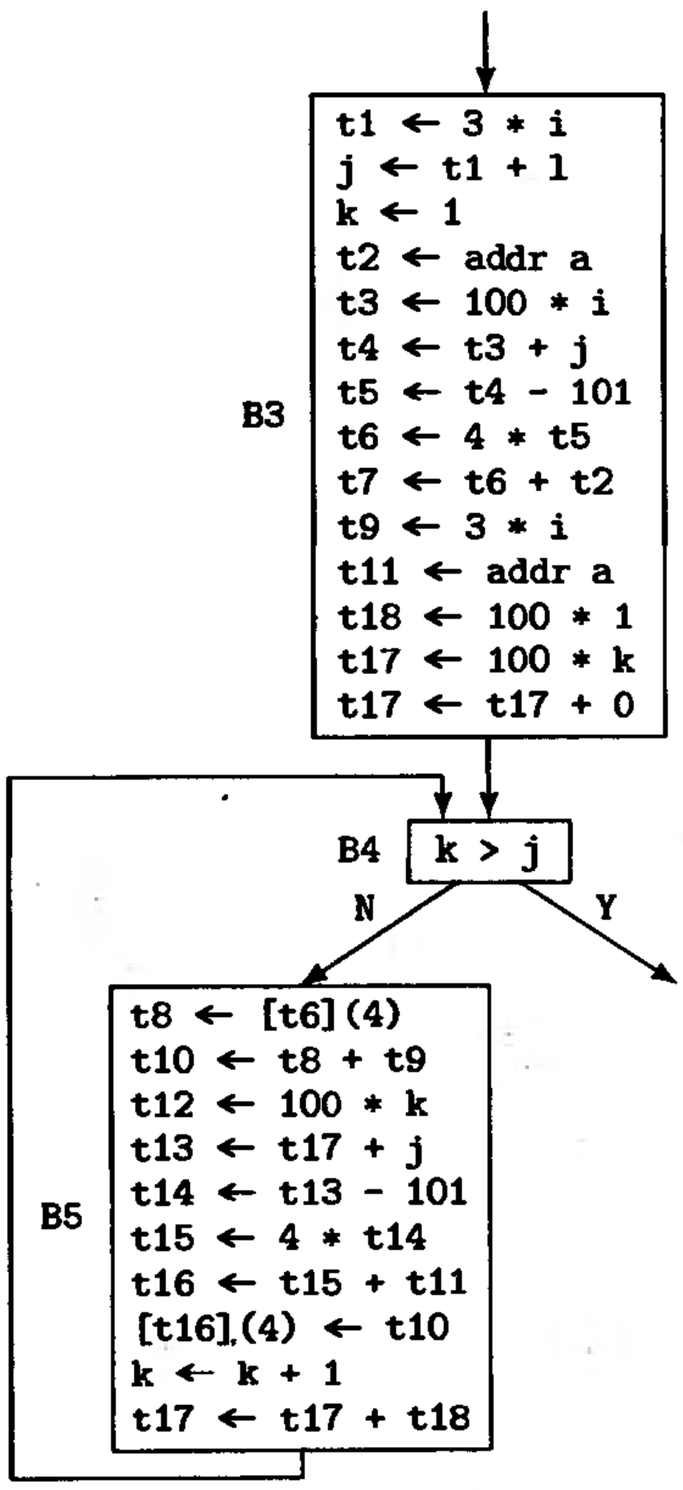


图14-14 在图14-13的代码中对t12进行强度削弱后的结果

其中的B3的结果，B3是内层循环的前置块

440
~
441

接下来，该例程设置

```
iv2 = <tiv:t13, blk:B5, pos:4, fctr:100, biv:k, diff:j>
```

并进行类似的处理：分配t19和t20分别作为tj和db值，设置i=B5、j=4和SRdone[B5][4]=true。然后，在前置块（基本块B3）的末尾添加下列指令：

```
t20 ← 100 * 1
t19 ← 100 * k
t19 ← t17 + j
```

在基本块B5的末尾添加指令

bbs.theithome.com

```
t19 ← t19 + t20
```

并设置

```
IVs = {⟨tiv:k,  blk:B5,pos:9, fctr:1,  biv:k,diff:0⟩,
       ⟨tiv:t12,blk:B5,pos:3, fctr:100,biv:k,diff:0⟩,
       ⟨tiv:t13,blk:B5,pos:4, fctr:100,biv:k,diff:j⟩,
       ⟨tiv:t14,blk:B5,pos:5, fctr:100,biv:k,diff:j-101⟩,
       ⟨tiv:t15,blk:B5,pos:6, fctr:400,biv:k,diff:4*j-404⟩,
       ⟨tiv:t16,blk:B5,pos:7, fctr:400,biv:k,
           diff:4*j-404+(addr a)⟩,
       ⟨tiv:t17,blk:B5,pos:10,fctr:100,biv:k,diff:100⟩,
       ⟨tiv:t19,blk:B5,pos:11,fctr:100,biv:k,diff:j⟩}
```

最后，例程用t19替换所有的t13。注意，这次又有插入到前置块的两条指令（即t18←100*1和t17←t17+0）是不必要的，并且设置t13的指令保留在基本块B5中，图14-15给出了结果。

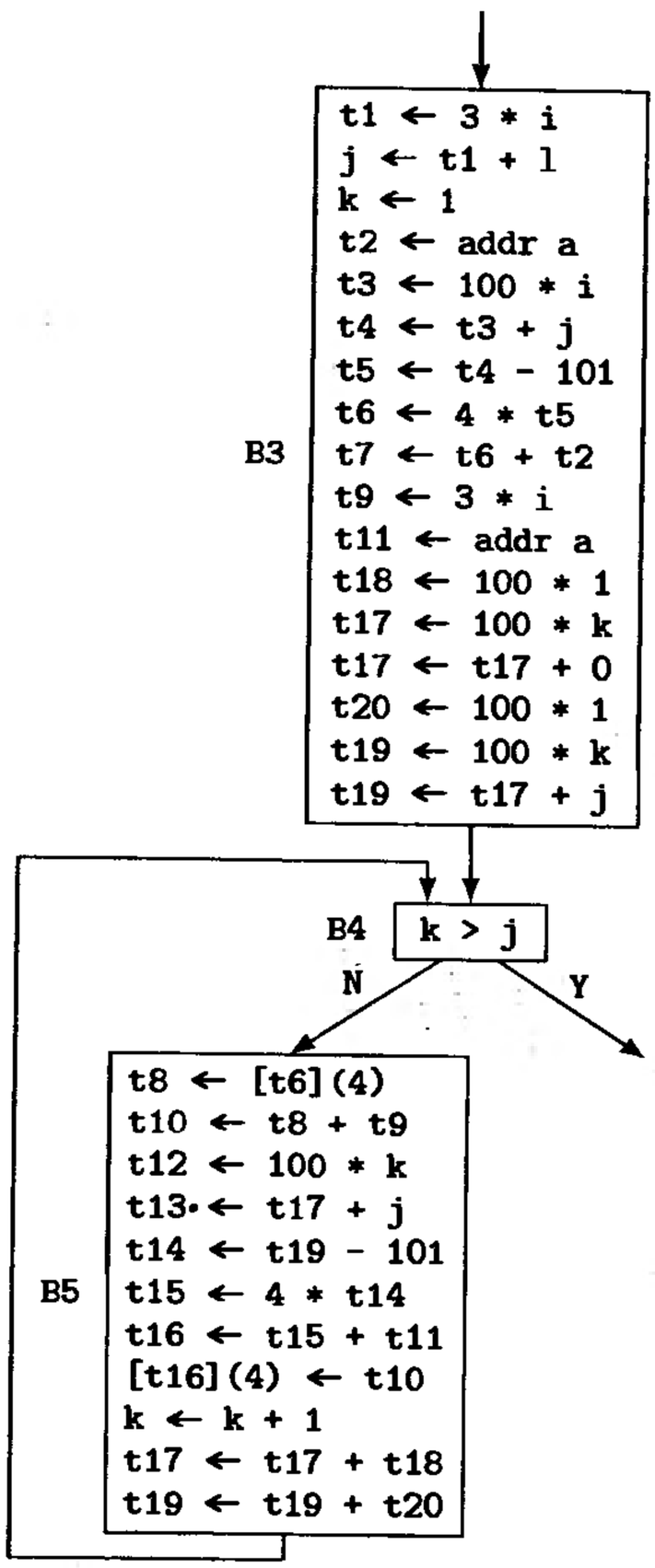


图14-15 对图14-14部分流图中的t3进行强度削弱后的结果

我们让读者完成这个例子。结果得到的集合IVs应当是

```
IVs = {⟨tiv:k,  blk:B5,pos:9, fctr:1,  biv:k,diff:0⟩,
       ⟨tiv:t12,blk:B5,pos:3, fctr:100,biv:k,diff:0⟩,
       ⟨tiv:t13,blk:B5,pos:4, fctr:100,biv:k,diff:j⟩,
       ⟨tiv:t14,blk:B5,pos:5, fctr:100,biv:k,diff:j-101⟩,
       ⟨tiv:t15,blk:B5,pos:6, fctr:400,biv:k,diff:4*j-404⟩,
```

版权所有 翻印必究

bbs.theithome.com

```
⟨tiv:t16,blk:B5,pos:7, fctr:400,biv:k,
    diff:4*j-404+(addr a)⟩,
⟨tiv:t17,blk:B5,pos:10,fctr:100,biv:k,diff:0⟩,
⟨tiv:t19,blk:B5,pos:11,fctr:100,biv:k,diff:j⟩,
⟨tiv:t21,blk:B5,pos:12,fctr:100,biv:k,diff:j-101⟩,
⟨tiv:t23,blk:B5,pos:13,fctr:400,biv:k,diff:4*j-404⟩,
⟨tiv:t25,blk:B5,pos:14,fctr:400,biv:k,
    diff:4*j-404+(addr a)⟩}
```

且仅当i=3,4,…,7时SRdone[B5][i]=true，所产生的部分流图在图14-16中给出。

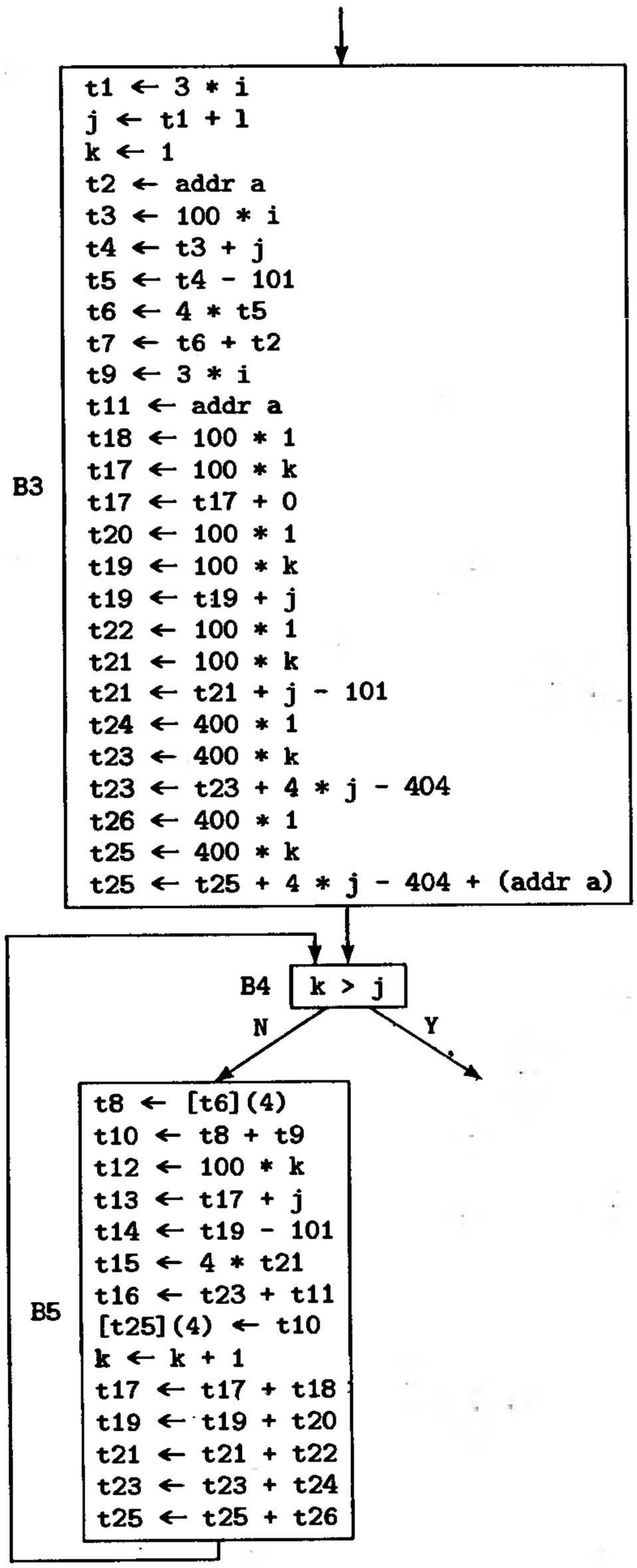


图14-16 对图14-15中部分流图中剩余归纳变量进行强度削弱后的结果

bbs.theithome.com

当然，B3中的某些表达式，如4*j-404+(addr a)，并不是合法的MIR代码，但它们显然可展开为合法的MIR代码。

删除死代码，做常数折叠，并删除B3中的无用赋值之后得到了图14-17所示的部分流图。注意对t8、t10、t12、t13、t14、t15和t16的赋值指令也都是死代码。

Knoop、Rüthing和Steffen [KnoR93]给出了一种基于他们的部分冗余删除方法做强度削弱的方法（参见13.3节）。但是如13.3节结束时所讨论的，它是一种特别弱的强度削弱，因此这里给出的传统方法应当比它要好。

另一方面，Cooper和Simpson([CooS95a]和[Simp96])给出了一种方法，这种方法对强度削弱方法进行了扩充，使其能作用于过程的SSA形式。由此得到的算法和上面的方法一样有效，甚至更好。

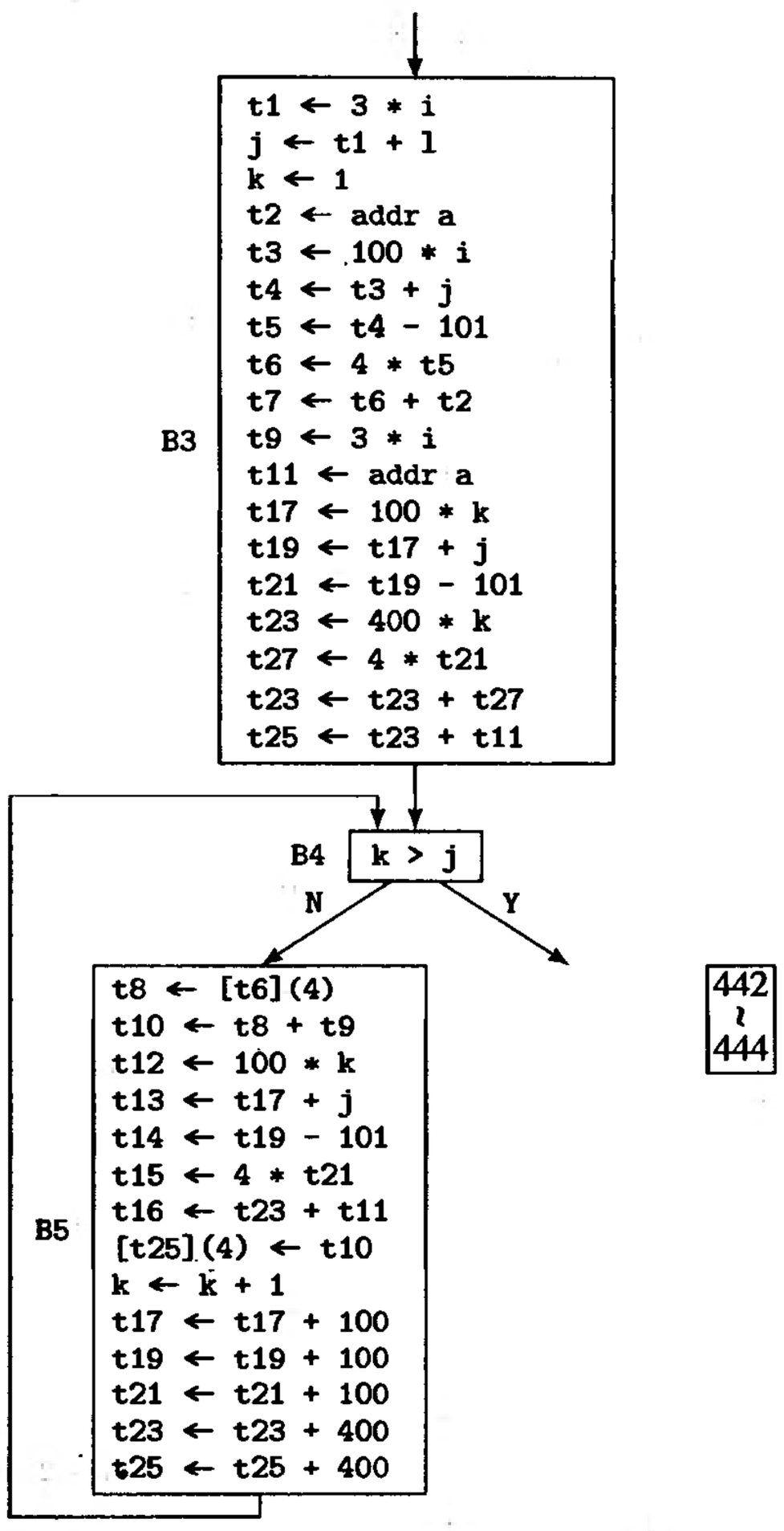


图14-17 对图14-16中部分流图做常数折叠，并删除基本块B3中的无用赋值后的结果

14.1.3 活跃变量分析

为了执行后面将介绍的归纳变量转换以及其他一些优化，如图着色寄存器分配和死代码删除，我们需要进行活跃变量分析。一个变量在程序的某个特定点是活跃的（live），如果存在着一条通向出口的路径，在此路径上其值的使用先于对它的重新定义。如果不存在这种路径，则称该变量是死去的（dead）。

为了确定流图中每一点上哪些变量是活跃的，我们执行向后数据流分析。我们定义*USE*(*i*)是基本块*i*中其使用先于其定义的那些变量的集合，*DEF*(*i*)是基本块*i*中其定义先于其使用的那些变量的集合。一个变量在基本块*i*的入口是活跃的，如果它在基本块*i*的出口是活跃的且不属于*DEF*(*i*)，或者它属于*USE*(*i*)，因此

$$LVin(i)=(LVout(i)-DEF(i))\cup USE(i)$$

一个变量在基本块的出口是活跃的，如果它在其每一个后续的入口是活跃的。因此

$$LVout(i)=\bigcup_{j\in Succ(i)} LVin(j)$$

合适的初始值是*LVout*(exit)= ∅ 。

作为活跃变量数据流分析的例子，考虑图14-18中的流图。*DEF*()和*USE*()的值如下:

DEF(entry)	= ∅	*USE*(entry)	= ∅
DEF(B1)	= {a,b}	*USE*(B1)	= ∅
DEF(B2)	= {c}	*USE*(B2)	= {a,b}
DEF(B3)	= ∅	*USE*(B3)	= {a,c}
DEF(exit)	= ∅	*USE*(exit)	= ∅

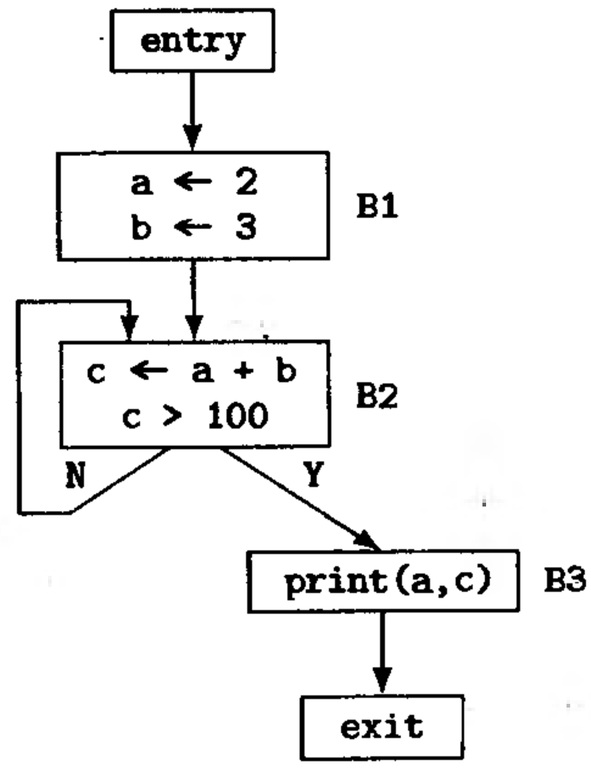


图14-18 计算活跃变量的流图例子

bbs.theithome.com

LVin()和*LVout*()的值如下:

LVin(entry)	= ∅	*LVout*(entry)	= ∅
LVin(B1)	= ∅	*LVout*(B1)	= {a,b}
LVin(B2)	= {a,b}	*LVout*(B2)	= {a,b,c}
LVin(B3)	= {a,c}	*LVout*(B3)	= ∅
LVin(exit)	= ∅	*LVout*(exit)	= ∅

445 ~ 446

所以，在基本块B2的入口a和b是活跃的，在基本块B3的入口a和c是活跃的。

14.1.4 归纳变量删除和线性函数测试替换

除了对归纳变量进行强度削弱外，我们常常还能完全删除它们。删除它们的基本判断标准是显然的——这种归纳变量在程序中没有用途——但是却不总是那么容易识别。产生这种没有用途的归纳变量的原因如下:

1. 这种变量从一开始就没有参与计算。
2. 由于其他转换，如强度削弱，导致这种变量成为无用的。
3. 这种变量是在一次强度削弱过程中创建的，然后因为进行了另一次强度削弱而变成无用的。
4. 这种变量只用在循环结束测试中，并且可以被那个上下文中另外的归纳变量所替代。这种情形称为线性函数测试替换。

作为第一种情形的例子，考虑图14-19a中的变量j。j在这个循环中完全没有用途，假定它的最终值在循环之后不需使用，因此我们可以简单地删除它。即使需要使用j的最终值，也可在循环之后用一个赋值j=12来替代。这种情形可由死代码删除来处理（参见18.10节）。

```
j = 2
do i = 1,10
   a(i) = i + 1
   j = j + 1
enddo
```

a)

```
j = 2
do i = 1,10
   a(i) = j
   j = j + 1
enddo
```

b)

图14-19 Fortran 77代码中无用归纳变量的例子

作为第二种情形的例子，考虑图14-19b中的变量j。这里循环虽然实际使用了j的值，但由于j是i类的一个归纳变量，并且它每次使用的值实际上是i+1，因此我们不难删除它。

第三种情形的例子可通过转换图14-12中的代码而得到。考虑变量t7，在进入循环之前它被初始化为4，然后在循环内被赋给t5并增加4。我们删除赋值t5←t7，并用t7替代t5的使用，由此得到了图14-20a中的代码。注意，这个循环中没有使用t7的值（除了增加它之外），所以t7←t7+4以及循环之前对t7的初始化都可以删除，由此得到图14-20b所示的代码[⊖]。

447

如果所编译的目标机体系结构有基寄存器更新的存/取指令，则我们偏向于选择可以从这种指令受益的归纳变量，即用来寻址并增加一个适当量的归纳变量。

最后一种情形，线性函数测试替换，可以用图14-20中的变量i来举例说明——i在循环之前被置初值，并且在循环内作为循环结束控制变量被测试和增值。除了在循环之后可能需要它的最终值外，在循环中它没有任何其他用途。通过确定出t8在循环中的最终值，即(addr a)+400，并将此值赋给一个新的临时变量t9，用t2←t8>t9替代原来的终止测试，并删除所有使用i的语句，我们可以删除变量i，并由此得到图14-21中的代码。（注意，为了进一步简化代码并使它的可读性更好，我们还用t6的值替代了t6的一个使用，因而删除了t6。）如果知道i在循环结束时是活跃的，或不知道它是否活跃，我们也可以在L2处插入i←100。

为了对给定的循环执行归纳变量删除和线性函数测试替换，我们按如下方法进行处理。

⊖ 注意，对这个例子也可做循环倒置，但我们选择不这样做，以便每次只处理一个问题。

bbs.theithome.com

```
     t1 ← 202
     i ← 1
     t7 ← 4
     t3 ← addr a
     t4 ← t3 - 4
     t8 ← t4 + 4
L1:  t2 ← i > 100
     if t2 goto L2
     t1 ← t1 - 2
     t6 ← t8
     *t6 ← t1
     i ← i + 1
     t8 ← t8 + 4
     t7 ← t7 + 4
     goto L1
L2:
```

a)

```
     t1 ← 202
     i ← 1

     t3 ← addr a
     t4 ← t3 - 4
     t8 ← t4 + 4
L1:  t2 ← i > 100
     if t2 goto L2
     t1 ← t1 - 2
     t6 ← t8
     *t6 ← t1
     i ← i + 1
     t8 ← t8 + 4

     goto L1
L2:
```

b)

图14-20 图14-12中的代码转换后的版本：a) 删除归纳变量t5之后，b) 也删除t7之后

对于用前一节强度削弱算法插入的每一个赋值$j \leftarrow tj$，如果在这个插入的语句和j的所有使用之间没有对tj的定义，则用tj的使用替代j的所有使用，并删除已插入的语句$j \leftarrow tj$。这正是我们在转换图14-20b中的代码到图14-21中含线性函数测试替换的代码时所做的事情。注意这是复写传播的一种局部形式。

令i是只在计算其他归纳变量中和关系式中使用的基本归纳变量，令j是具有线性方程$j = b*i + c$的i类中一个归纳变量。我们用

$$tj \leftarrow b * v$$
$$tj \leftarrow tj + c$$
$$j \ ? \ tj$$

替换关系运算i ? v，其中？代表关系运算符，v不是一个归纳变量，并删除循环中所有对i的赋值。如果i在通向循环出口的某条路径上是活跃的，则在循环的每一个这种出口处放置一条将i的最终值送给i的赋值语句。

```
     t1 ← 202
     t3 ← addr a
     t4 ← t3 - 4
     t8 ← t4 + 4
     t9 ← t3 + 400
L1:  t2 ← t8 > t9
     if t2 goto L2
     t1 ← t1 - 2
     *t8 ← t1
     t8 ← t8 + 4
     goto L1
L2:
```

图14-21 图14-20中的代码在删除归纳变量（i和t6）和对变量i执行线性函数测试替换后的结果

这一处理过程中有一个容易被编译器的编写者和著作者忽视，并因此导致强度削弱出问题的复杂因素（参见[AllC81]3.5节）。为了在替代语句中保持关系“?”，我们必须知道b是正的，如果它是负的，则需要使用相反的关系，为此需在替换表达式中将它表示成“!?”。具体地，上面的关系表达式应变成

$$j \ !? \ tj$$

如果b只是一个循环不变量，而不是一个已知的常数，我们就可能不知道它的正负。在这种情况下可能不值得做线性函数测试替换，但如果有必要做，做也是可以的，但以增加代码体积为代价。我们只需要在进入循环之前测试这个循环不变量的符号，并分支到两个已优化的循环之一；其中一个循环假定这个循环不变量是正的，另一个假定它是负的。尽管这样做似乎没有太大的好处，但如果在并行机或向量机上它能导致获得更大的并行执行或向量执行的话，则

bbs.theithome.com

好处就很大。另一种方法是，我们可以简单地将循环结束测试代码分解为对符号进行测试，然后根据结果转移到这个循环结束测试的两个分支之一的代码。

如果关系运算涉及到两个归纳变量，例如$i1$? $i2$，且它们都只在计算其他归纳变量和关系式中使用，其转换就更复杂。如果存在着归纳变量$j1$和$j2$，它们的线性方程分别是$j1=b*i1 + c$
449 和$j2 = b*i2 + c$，我们则能够简单地用$j1$? $j2$替换$i1$? $i2$，这里再次假设b是正的。如果不存在这种具有相同的b和c值的归纳变量$j1$和$j2$，这种替换一般就不值得做，因为它可能会为了替换代价较小的运算而在循环中引入两个乘法和一个加法。

```
procedure Remove_IVs_LFTR(bset,nblocks,ninsts,Block,IVs,SRdone,Succ,Pred)
   bset: in set of integer
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   IVs: in set of IVrecord
   SRdone: in array [1··nblocks] of array [··] of boolean
   Succ, Pred: inout integer --> set of integer
begin
   op1t, op2t: enum {con,ind,var}
   iv1, iv2: IVrecord
   i, j: integer
   tj: Var
   v: Const
   inst: MIRInst
   oper: Operator
   for each iv1 ∈ IVs (SRdone[iv1.blk][iv1.pos]) do
      for each iv2 ∈ IVs (!SRdone[iv2.blk][iv2.pos]
         & iv1.biv = iv2.biv & iv1.fctr = iv2.fctr
         & iv1.diff = iv2.diff) do
         || if iv1 and iv2 have matching equations and iv1
         || has been strength-reduced and iv2 has not,
         || replaces uses of iv2 by uses of iv1
         for each i ∈ bset do
            for j := 1 to ninsts[i] do
               Replace_Uses(i,j,Block,iv1,iv2)
            od
         od
      od
   for each i ∈ bset do
      for j := 1 to ninsts[i] do
         if Has_Left(Block[i][j].kind) & SRdone[i][j] then
            if Live_on_Exit(inst.left,bset,Block) then
               || if result variable is live at some exit from the loop,
               || compute its final value, assign it to result variable
               || at loop exits
               v := Final_Value(inst.left,bset,Block)
               Insert_Exits(bset,Block,<kind:valasgn,
                  left:inst.left,opd:<kind:const,val:v>>)
            fi
            || delete instruction Block[i][j] and renumber the tuples
            || in IVs to reflect the deletion
            delete_inst(i,j,nblocks,ninsts,Block,Succ,Pred)
            IVs -= {iv1}
            for each iv2 ∈ IVs do
```

图14-22 实现归纳变量删除和线性函数测试替换的代码

bbs.theithome.com

```
                if iv2.blk = i & iv2.pos > j then
                   IVs := (IVs - {iv2})
                      ∪ {⟨tiv:iv2.tiv,blk:i,pos:iv2.pos-1,
                      fctr:iv2.fctr,biv:iv2.biv,diff:iv2.diff⟩}
                fi
             od
          fi
       od
    od
 od
for each i ∈ bset do
   j := ninsts[i]
   inst := Block[i][j]
   if inst.kind ≠ binif then
      goto L1
   fi
   || perform linear-function test replacement
   Canonicalize(inst,op1t,op2t)
   if op1t = con then
      if op2t = con & Eval_RelExpr(inst.opd1,inst.opr,inst.opd2) then
         || if both operands are constants and the relation is true,
         || replace by goto
         Block[i][j] := ⟨kind:goto,lbl:inst.lbl⟩
      elif op2t = ind then
         || if one operand is a constant and the other is an induction
         || variable, replace by a conditional branch based on a
         || different induction variable, if possible
         if ∃iv1 ∈ IVs (inst.opd2.val = iv1.tiv & iv1.tiv = iv1.biv) then
            if ∃iv2 ∈ IVs (iv2.biv = iv1.biv
               & iv2.tiv ≠ iv1.tiv) then
               tj := new_tmp( )
               insert_before(i,j,ninsts,Block,⟨kind:binasgn,left:tj,
                  opr:mul,opd1:⟨kind:const,val:iv2.fctr⟩,opd2:inst.opd1⟩)
               insert_before(i,j,ninsts,Block,
                  ⟨kind:binasgn,left:tj,opr:add,
                  opd1:⟨kind:const,val:iv2.diff⟩,opd2:⟨kind:var,val:tj⟩⟩)
               oper := inst.opr
               || if new induction variable runs in the opposite direction
               || from the original one, invert the test
               if iv2.fctr < 0 then
                  oper := Invert(oper)
               fi
               Block[i][j] := ⟨kind:binif,opr:oper,
                  opd1:⟨kind:var,val:tj⟩,
                  opd2:⟨kind:var,val:iv2.tiv⟩,lbl:inst.lbl⟩
            fi
         fi
      fi
   elif op1t = ind then
      if op2t = ind then
         if ∃iv1,iv2 ∈ IVs (iv1 ≠ iv2 & iv1.biv = inst.opd1.val
            & iv2.biv = inst.opd2.val & iv1.fctr = iv2.fctr
            & iv1.diff = iv2.diff) then
            || if both operands are induction variables,...
            oper := inst.opr
```

图14-22 （续）

bbs.theithome.com

```
                if iv2.fctr < 0 then
                   oper := Invert(oper)
                fi
                Block[i][j] := (kind:binif,opr:oper,
                   op1:(kind:var,val:iv1.tiv),
                   op2:(kind:var,val:iv2.tiv),lbl:inst.lbl)
             fi
          elif op2t = var & BIV(inst.opd1.val,IVs)
             & ∃iv1 ∈ IVs (iv1.biv = inst.opd1.val
             & iv1.tiv ≠ iv1.biv) then
             tj := new_tmp( )
             insert_before(i,j,ninsts,Block,
                (kind:binasgn,left:tj,opr:mul,
                opd1:(kind:const,val:iv1.fctr),opd2:inst.opd2))
             insert_before(i,j,ninsts,Block,
                (kind:binasgn,left:tj,opr:add,
                opd1:(kind:const,val:iv1.diff),opd2:(kind:var,val:tj)))
             oper := inst.opr
             if iv1.fctr < 0 then
                oper := Invert(oper)
             fi
             Block[i][j] := (kind:binif,opr:oper,
                opd1:(kind:var,val:iv1.tiv),
                opd2:(kind:var,val:tj),lbl:inst.lbl)
          fi
       fi
L1:    od
   end    || Remove_IVs_LFTR
```

图14-22 （续）

图14-22给出了实现上面处理过程的ICAN代码，它使用了如下若干函数：

1. Insert_before(*i*, *j*, ninsts, Block, *inst*) 将指令*inst*直接插入在Block[*i*][*j*]之前，并相应地调整数据结构（参见图4-14）。

2. Delete_inst(*i*, *j*, nblocks, ninsts, Block, Succ, Pred) 删除Block[*i*]中的第*j*条指令，并相应地调整数据结构（参见图4-15）。

3. Replace_Uses(*i*, *j*, Block, *iv*1, *iv*2) 在Block[*i*][*j*]中用*iv*2.tiv替换*iv*1.tiv的所有使用。

4. Has_left(*kd*) 返回true，如果种类为*kd*的MIR指令有左部量；否则返回false（参见图4-8）。

5. Canonicalize(*inst*, *t*1, *t*2)，对于给定的一条含二元关系表达式的MIR指令*inst*，重排其操作数，使得（a)如果有一个操作数是常数，使它成为第一个操作数，并且(b)如果没有常数操作数，但有一个是归纳变量，使它成为第一个操作数；如果它重排了操作数的顺序，则相应调整运算符；并且根据规范化后的第一个操作数或第二个操作数是常数、归纳变量还是非归纳变量，分别设置*t*1和*t*2为con、ind或var。

6. Eval_RelExpr(*opd*1, *opr*, *opd*2) 计算关系表达式*opd*1 *opr* *opd*2，并返回该表达式的值(true或false)。

7. BIV(*v*, *IVs*) 返回true，如果*v*作为基本归纳变量出现在归纳变量记录集合*IVs*中；否则返回false。

8. Live_on_Exit(*v*, *bset*, Block) 返回true，如果变量*v*在由*bset*给出的基本块组成的

bbs.theithome.com

循环的某个出口是活跃的；否则返回false（这种性质由执行前一节介绍的活跃变量数据流分析来计算）。

9. Final_Value(*v*, *bset*, Block)返回变量*v*从*bset*给出的基本块所组成的循环出口时的最终值。

10. Insert_Exits(*bset*, Block, *inst*)将MIR指令*inst*插入到紧接在*bset*给出的循环的每一个出口之后。

11. Invert(*opr*)返回MIR关系运算符*opr*的逆运算符，例如，对于“>”，它返回“<=”。

12. New_tmp()返回一个新的临时变量名。

注意，我们能够用一种更有效的方法来实现在Remove_IVs_LFTR()中那个逐一查看指令的内层for循环，这种方法保存一张描述要被删除的指令表，并只用单层for循环。 450 ↓ 453

14.2 不必要边界检查的消除

边界检查（bounds checking）或范围检查（range checking）是指判定一个变量在程序中使用的所有值是否处在其指定的范围之内。一个典型的例子是，对于其声明为

```
var b: array[1..100, 1..10] of integer
```

的Pascal数组元素b[i,j]，检查i的值是否确实在1到100之间，j是否确实在1到10之间。另一个例子是，检查一个声明为Ada子类型的变量的使用是否在所指定的范围之内，例如，是否处在这种子类型之内：

```
subtype TEMPERATURE is INTEGER range 32..212;
i: TEMPERATURE;
```

我们在上面这两个例子中提及Pascal和Ada是因为它们的语言定义特别指明要进行这种检查（或等价地，这种语言的实现要保证以某种方式来满足规定的限制）。不过，无论程序用什么语言编写，这种检查都是有好处的，因为越界是最常见的程序设计错误。“差1”错误就是一个典型的例子。在这种错误中，循环索引或其他计数器超越了末尾或其他范围一个位置——这种错误通常导致访问或存储了不属于被处理数据结构一部分的数据。

另一方面，边界检查的代价可能非常大，例如，像图14-23演示的那样，其中每一个数组访问的每一维都必须伴随两个判别访问合法性的条件自陷[⊖]，其中我们假设陷阱6用于数组维界越界。这里，数组访问有8条MIR代码，检查用了额外的4条。当数组访问经过了优化时，这种检查开销甚至会更大——因为取二维数组的下一个元素可能只需要一条或两条加指令和一条取指令，而维界检查仍然需要用于条件自陷的4条指令。许多实现，尤其是Pascal，“解决”这一问题的方法是给用户提供一个允许进行检查的编译选项，这种选项的目的是允许用户在开发和调试程序时进行这种检查，一旦所有的问题都被发现并已更正之后，便关掉此开关运行正式的生产版本，这样，开销只发生在程序仍然有问题时，而不会发生在（用户认为）它已经没有错误之后。

```
if 1 > i trap 6
if i > 100 trap 6
if 1 > j trap 6
if j > 10 trap 6

t2 ← addr b
t3 ← j - 1
t3 ← t3 * 100
t3 ← t3 + i
t3 ← t3 - 1
t3 ← t3 * 4
t3 ← t2 + t3
t4 ← *t3
```

图14-23 Pascal中访问b[i,j]的边界检查MIR代码例子

但是，事实上软件工程关于程序中的问题或缺陷的所有研究都指出，发布给顾客的交付版本很可能还有隐藏的故障，其

⊖ if...trap结构可以用条件分支或条件自陷来实现，取决于体系结构、源语言以及语言的实现。

bbs.theithome.com

中很多故障是在发布前的测试期间未曾观察到的。因此这种边界检查的观点有一种严重的误解。应当说，边界检查对于程序的交付版本和开发版本一样重要。真正需要做的不是提供一种关闭边界检查开关的手段，而是对它进行优化，使之几乎不做多少事情，并且只有最小的开销。例如，在图14-23中，如果我们取b[i,j]的操作包含在一个循环内，这个循环规定了下标可取值的边界，并限定了它们的值是合法值，则数组越界检查代码就是完全不需要的。作为第二个例子，如果将这个例子中外层循环的上界改成变量n，而不是常量50，我们则只需要在进入外层循环时检查一次n<100是否满足，如果不满足则转向陷阱[⊖]。

这种优化比较容易，对于某些语言的多数程序而言，它几乎不值一提。事实上我们已经有很多方法可以满足它的需要，如，不变代码外提，公共子表达式删除和归纳变量转换。我们剩下需要的是用什么样的途径来表示满足边界检查的限制。为了表示边界检查限制，我们引入范围表达式。范围表达式（range expression）是作用于变量值的不等式，它的形式为：

454 ~ 455

lo ? *var* ? *hi*

其中*var*是一个变量名，*lo*和*hi*是表示范围最大和最小值的常数，？是关系运算符。如果这个变量的值只在一端有限制，则用－∞或∞表示另一端的边界。

例如，对于图14-24中的代码，为了使语句s:=s+b[i,j]不需要运行时的检查，我们必须满足的两个范围表达式是1<i<100和1<j<10，这是数组b的声明所要求的。为了判断这个语句是否满足这两个范围表达式，我们只需要能够从第一个for语句推断出在该语句构成的循环内1<i<100成立，并且从第二个for语句推断出在该语句构成的循环内1<j<10成立。这在Pascal中是微不足道的，因为这两个for语句分别确立了这两个不等式是合法的，并且Pascal语言的语义要求除了for语句本身外，for循环内不能修改循环迭代变量。对于其他语言的判别就没有这么容易——例如C语言对可出现在for循环结构内的表达式没有限制，它甚至没有迭代变量的概念。

```
var b: array[1..100,1..10] of integer;
   i, j, s: integer;
s := 0;
for i = 1 to 50 do
   for j = 1 to 10 do
      s := s + b[i,j]
```

图14-24 访问b[i, j]无需边界检查的Pascal例子

最简单并且也是迄今最常见的可优化边界检查代码是处在循环内的边界检查，如前面图14-24中例子的情况。为了表述更具体，我们对循环做出如下假设：

1. 循环具有迭代变量*i*，且初值为*init*，终值为*fin*，
2. 每迭代一次*i*增1，并且，
3. 只有循环控制代码修改*i*。

我们进一步假设要满足的范围表达式是*lo* < *v* < *hi*。

最容易处理的情形是*v*是一个循环不变量。在这种情况下，我们只需要将检查*lo* < *v* < *hi*的代码从循环内移到循环的前置块中。当然，如果能在编译时计算它，我们就在编译时计算它。

下一种情形是需满足的范围表达式是*lo* < *i* < *hi*，其中*i*是循环控制变量。在这种情况下，只要*lo* < *init*且*fin* < *hi*，此范围表达式就是满足的。我们插入检查这两个不等式中第一个不等式的代码到循环的前置块。我们也在前置块中插入计算*t*1=min(*fin*，*hi*)的代码，并用*i*与*t*1相比较的循环结束测试替换*i*与*fin*相比较的循环结束测试。在循环正常出口的后面，我们插入有关代码

⊖ 注意，我们假设陷阱终止程序的执行，或者至少它不能回到自陷发生的地方重新恢复执行。这是基本的要求，因为我们将那些不能完全删除的范围检查代码（尽可能）外提到包含它们的循环之外，因此，这种自陷的发生点与原来程序执行时的自陷发生点会有所不同，尽管我们保证当且仅当在原来未修改的程序中会发生自陷时，这个自陷才会发生。

bbs.theithome.com

检查*i*的终值是否到达了它不经过转换原本应到达的值，即插入*i* >*fin*的检查。如果这些检查中有任何一个失败，则发生自陷。当然，如果这些检查能够在编译时进行，它们就在编译时完成。图14-25给出了一个这种转换之前和之后的例子。

```
    i ← init
L1: . . .
    if i < lo trap 6
    if i > hi trap 6
    use of i that must
       satisfy lo ≤ i ≤ hi
    . . .
    i ← i + 1
    if i <= fin goto L1
```

a)

```
    if lo > init trap 6
    t1 ← fin min hi
    i ← init
L1: . . .


    use of i that must
       satisfy lo ≤ i ≤ hi
    . . .
    i ← i + 1
    if i <= t1 goto L1
    if i <= fin trap 6
```

b)

图14-25 边界检查转换：a) 原来的循环，b) 转换后的代码

我们考虑的最后一种情形是这样的一个归纳变量*j*：它属于基本归纳变量*i*类，并有线性方程*j* =*b***i* + *c*（参见14.1节），且必须满足范围表达式*lo* < *j* < *hi*。在这种情况下，我们有*j* = *b***i* + *c*，因此为了使*j*满足其范围表达式，*i*必须满足(*lo* − *c*) / *b* < *i* < (*hi* − *c*) / *b*。适合于这种情况的转换很容易从前述情形推广得到。

上面关于循环的第二和第三种假设，即在每一个迭代中*i*增加1，且只有循环控制代码修改*i*，都可以放宽以允许递减的循环索引，增加或减少一个非1的值，以及在循环内对*i*的简单修改，我们将这些留给读者考虑。

也可以通过数据流分析在过程内传播范围表达式来确定它们在什么地方是成立的，以及在什么地方需要做检查。但是，这样做的代价可能很大，因为所使用的格包含所有*lo*，*hi* ∈ **Z** ∪ {− ∞, ∞}的范围表达式*lo* < *v* < *hi*，且当且仅当*lo*1 < *lo*2且*hi*1 > *hi*2才具有顺序

$$(lo1 < v < hi1) \sqsubseteq (lo2 < v < hi2)$$

——这是一个既无穷宽又无穷高的格。至少，如果我们从一个具有有穷*lo*值和有穷*hi*值的范围表达式*lo* < *v* < *hi*开始，它就只有从那里开始的有穷的（但无界）上升链。

14.3 小结

本章我们讨论的是专门针对循环的优化，或者当作用于循环时具有最好效果的一些优化。这些优化可在中级中间代码、也可在低级中间代码上进行，它们直接作用于Fortran、Ada和Pascal的规则的源语言循环结构，对于C语言之类的循环（或用任意语言写的由`if`和`goto`形成的循环），为了安全施行这些优化，需要我们定义其行为与规则循环类似的循环子集。

456
↓
457

我们也论及了两类优化，即归纳变量优化和不必要的边界检查删除。图14-26中的黑体字显示了本章所讨论的优化在整个优化处理中的位置。

归纳变量优化需要我们首先标识出循环中的归纳变量，然后对它们执行强度削弱，最后执行线性函数测试替换和删除冗余的归纳变量。对于嵌套循环，我们从最内层开始执行这一系列优化并逐层向外进行处理。

bbs.theithome.com

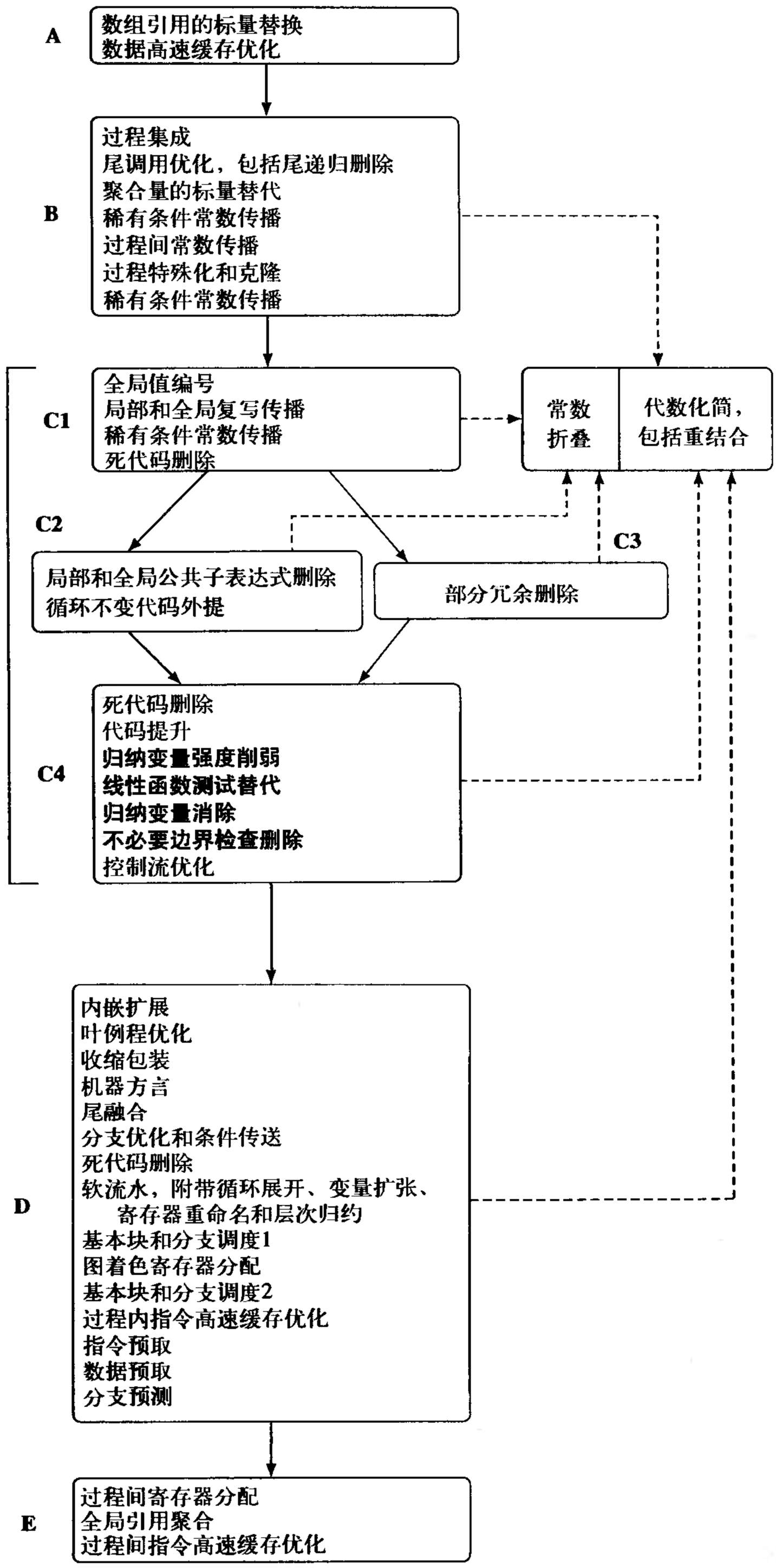


图14-26 循环优化（黑体字标明）在激进优化编译器中的位置

bbs.theithome.com

删除不必要的边界检查是一种既作用于循环内也作用于循环外的优化，但在循环内具有最大效果，因为循环内的边界检查是每一个迭代都要执行的，除非它们被优化删除掉了或者至少被简化了。

14.4 进一步阅读

将有限差分应用于计算机程序的基础性工作，如 [Gold72] 中介绍的，应归功于Babbage。Allen和Cocke关于对非加和乘运算应用强度削弱的论文可以在 [AllC81] 中找到。Paige和Schwartz [PaiS77] 讨论了有限差分的一种推广，称为形式差分，以及它在非常高级的语言SETL中的应用。Chow [Chow83] 和Knoop、R ü thing以及Steffen [KnoR93] 扩充了部分冗余删除使之包含了一种较弱形式的强度削弱。

Cooper 和Simpson的基于SSA的强度削弱方法的介绍见 [CooS95a] 和 [Simp96] 。

可以找到更为现代的边界检查优化方法，例如在Gupta [Gupt93] 、Kotte和Wolfe [KolW95] 中，后者利用部分冗余删除来确定最有效的边界检查放置点。

14.5 练习

14.1 如14.1节末尾指出的，那种其中一个操作数是可缩放的或可自动修改的存储器访问指令，或那种执行一个算术运算和测试，同时根据测试结果进行分支的指令，可能有助于强度削弱、归纳变量删除和线性函数测试替换（参见图14-27的例子）。(a)你如何判别应使用这些指令中的哪种指令？(b)你应当在优化处理和代码生成过程的什么点上做出这种判断？

```
    i ← 1
L1: r1 ← 4 * i
    r2 ← (addr a) + r1
    r3 ← [r2](4)
    r3 ← r3 + 2
    [r2](4) ← r3
    i ← i + 1
    if i ≤ 20 goto L1
```

图14-27 对按比例伸缩寻址、自动修改寻址和运算—测试—分支指令都可能有用的一个MIR循环的例子

ADV 14.2 将归纳变量识别形式化为数据流分析，并将它应用于图14-9的例子。

14.3 如代码所写的那样，图14-10中的ICAN代码总是生成新的临时变量作为db，并在循环前置块中给它赋初值。(a)修改这段代码使它在不需要时不这样做。给这个临时变量tj赋初值的指令会有何变化？(b)注意也存在如图14-9中的归纳变量t14那样的一些情况，其中，因子(fctr)或差(diff)不是简单的常数或可以用作强度削弱的变量。修改这段代码识别这种情形并适当地处理它们。

14.4 写出一个删除不必要边界检查的ICAN例程。

14.5 推广14.2节的边界检查转换，使它能够如那一节末尾讨论的：(a)检查归纳变量，(b)适应更一般的修改循环控制变量的情况。在练习14.4中编写的例程中增加反映这种改进的代码。

14.6 扩充图14-22算法中线性函数测试替换部分，以处理非编译时已知的循环常量。

14.7 通过用图14-17中的基本块B3、B4和B5替换图14-19中的基本块B3、B4和B5来继续图14-19的例子。然后，(a)对内层循环做归纳变量删除和线性函数测试替换，(b)对外层循环做完整的归纳变量优化。

RSCH 14.8 阅读 [Gupt93] 或 [KolW95] ，并写出实现其边界检查方法的ICAN代码。

458
ι
460

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第15章 过程优化

本章我们讨论作用于整个过程的三对优化方法，它们是尾递归删除和比较通用的尾调用优化、过程集成和内嵌扩展，以及叶例程优化和收缩包装，其中除了一种优化方法之外其余都不需要进行数据流分析。第一对优化方法将调用转变成转移。第二对优化方法是同一种优化的两个版本，第一个应用于中级或高级中间代码，第二个应用于低级代码。最后一对优化方法用于优化语言实现中使用的调用约定。

15.1 尾调用优化和尾递归删除

尾调用优化和它的特殊情形，即尾递归删除是作用于调用的一种转换。它们常常能显著地减少或消除过程调用的开销，并且经过尾递归删除后，可以执行原本不能执行的循环优化。

过程f()对过程g()的调用是*尾调用*（tail call），如果在g()返回后，f()惟一做的事情是返回自己。如果f()和g()是同一个过程则这个调用是*尾递归的*（tail recusive）。例如图15-1的C代码中，对insert_node()的调用是尾递归的，而对make_node()的调用是（非递归的）尾调用。

对insert_node()施加尾递归删除的效果如图15-2所示，它将递归变成了循环。

```
void make_node(p,n)
   struct node *p;
   int n;
{  struct node *q;
   q = malloc(sizeof(struct node));
   q->next = nil;
   q->value = n;
   p->next = q;
}

void insert_node(n,l)
   int n;
   struct node *l;
{  if (n > l->value)
      if (l->next == nil) make_node(l,n);
      else insert_node(n,l->next);
}
```

图15-1 C的尾调用和尾递归例子

```
void insert_node(n,l)
   int n;
   struct node *l;
{loop:
   if (n > l->value)
      if (l->next == nil) make_node(l,n);
      else
      {  l := l->next;
         goto loop;
      }
}
```

图15-2 用源代码展示的对insert_node()施加尾递归删除的效果

我们不能用源语言来示范尾调用优化的效果，因为从一个过程体转移到另一个过程体不符合C的语义（实际上也不符合其他任何高级语言的语义），但可以用类似的方法来思考它：用make_node()的参数替代栈中的（或适当寄存器中的）insert_node()的参数，并用一条转移到make_node()函数体的开始处的分支指令替代调用make_node()的指令。

但即使这样仍然违背了MIR的语义，因为参数名是局部于过程的。不过我们可以用LIR来说明尾调用优化的效果。与图15-1对应的LIR代码在图15-3a中给出，对这个例子我们随意地选

bbs.theithome.com

版权所有 请勿制作盗版

择将参数传递到寄存器r1和r2中。图15-3b给出了对make_node执行尾调用优化的结果。但这个版本中仍然隐藏了一个微妙的问题，因为我们在其中没有明显展示存储栈的情况。事实上，调用者和被调用者可能有大小各不相同的栈帧。如果调用者的栈帧大于被调用者的栈帧，我们仅需要安排被调用过程的出口处理（epilogue）（参见5.6节）释放调用者的栈帧。释放调用者栈帧最容易的方法是使用一个帧指针(即调用者的栈指针，或在这种情况下，即调用者的栈指针)，然后在出口时通过将帧指针赋给栈指针而恢复调用者的栈指针。例如，Sun的SPARC编译器就是这样做的（参见21.1节）。如果调用者的栈帧小于被调用者的，我们需要在进入被调用过程之前或在被调用过程的入口处为被调用过程分配剩余部分的栈帧，或者需要在转移到被调用者之前先释放调用者的栈帧，然后在进入被调用者时做标准的入口处理（prologue）。

461 ~ 462

```
make_node:
    r4 ← r1
    r1 ← 8
    r3 ← call malloc
    r3 *. next ← nil
    r3 *. value ← r2
    r4 *. next ← r3
    return
insert_node:
    r4 ← r2 *. value
    r5 ← r1 > r4
    if !r5 goto L1
    r6 ← r2 *. next
    r7 ← r6 = nil
    if !r7 goto L2
    r2 ← r1
    r1 ← r4
    call make_node
    return
L2: r2 ← r2 *. next
    call insert_node
    return
L1: return
```

a)

```
make_node:
    r4 ← r1
    r1 ← 8
    r3 ← call malloc
    r3 *. next ← nil
    r3 *. value ← r2
    r4 *. next ← r3
    return
insert_node:
    r4 ← r2 *. value
    r5 ← r1 > r4
    if !r5 goto L1
    r6 ← r2 *. next
    r7 ← r6 = nil
    if !r7 goto L2
    r2 ← r1
    r1 ← ← r4
    goto make_node

L2: r2 ← r2 *. next
    goto insert_node

L1: return
```

b)

图15-3 a) 与图15-1对应的LIR代码，b) 对make_node()中的两个调用执行尾调用优化后的结果

判别一个调用是否是尾调用较容易，它只需要检查执行这个调用的例程在该调用返回之后惟一做的事情是返回自己。这个例程本身要返回的值可能就是由被调用者返回的值。删除尾递归调用的过程是直观的，如我们前面的例子所示，它甚至可以在源代码一级进行。替代这个递归调用所要做的只是将适当的值赋给参数，接着转移到过程体的开始，并删除以前跟在该递归调用之后的return语句即可。图15-4给出了对MIR过程执行尾递归删除的ICAN代码。

实现一般的尾调用优化所做的工作要多一些。首先我们必须保证这两个过程体在编译时是同时可见的，或者，至少在编译时调用者有足够多的、使得转换有可能发生的关于被调用者的信息。我们可以因为调用过程和被调用过程处在同一个编译单位，或因为编译系统有保存过程的中间代码表示的选项（如MIPS编译器所做的那样），而同时见到这两个过程体。不过我们真正需要知道的只是被调用者的如下三种信息:

1. 它希望参数放在何处;
2. 为了开始执行它的过程体，控制应转移到何处;
3. 它的栈帧的大小。

bbs.theithome.com

```
procedure Tail_Recur_Elim(ProcName,nblocks,ninsts,Block,en,Succ)
   ProcName: in Procedure
   nblocks, en: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of MIRInst
   Succ: in integer ⟶ set of integer
begin
   i, j, b := ♦Succ(en): integer
   lj: Label
   inst: MIRInst
   || make sure there's a label at the beginning of the procedure's body
   if Block[b][1].kind = label then
      lj := Block[b][1].lbl
   else
      lj := new_label( )
      insert_before(b,1,ninsts,Block,⟨kind:label,lbl:lj⟩)
   fi
   for i := 1 to nblocks do
      inst := Block[i][ninsts[i]-1]
      if (inst.kind = callasgn & inst.proc = ProcName
         & Block[i][ninsts[i]].kind = retval)
         V (inst.kind = call & inst.proc = ProcName
         & Block[i][ninsts[i]].kind = return) then
         || turn tail call into parameter assignments
         || and branch to label of first block
         for j := 1 to |inst.args| do
            Block[i][ninsts[i]+j-2] := ⟨kind:valasgn,
               left:Block[b][j].left,
               opd:Block[i][ninsts[i]-1].args↓j@1⟩
         od
         ninsts[i] += |inst.args| + 1
         Block[i][ninsts[i]] := ⟨kind:goto,lbl:lj⟩
      fi
   od
end    || Tail_Recur_Elim
```

图15-4 执行尾递归删除的ICAN代码

这些信息可以用一种只保存过程的接口表示、而不是其过程体的方式来保存。如果只保存接口，当调用者的栈帧大于被调用者的时，我们就可能不能做这种转换——这取决于栈帧的分配和释放所使用的约定（参见5.4节）。 463 ~ 464

为了执行这种优化，我们用如下三件事来替代这个调用:

1. 计算这个尾调用的实参，并将它们放在被调用者期望找到它们的地方;
2. 如果被调用者的栈帧大于调用者的，用它们两者之差扩展栈帧;
3. 转移到被调用者过程体的开始。

执行尾调用优化的另一个问题与每一种体系结构中的寻址方式，以及调用和分支指令的跨距有关。例如，在Alpha中不存在这种问题，因为jmp转移到一个过程和jsr调用一个过程都使用寄存器的内容作为其目标地址，它们的不同只是保存还是忽略返回地址。类似地，在MIPS体系结构中，jal和j的目标地址都取26位绝对字目标地址。但另一方面，在SPARC中，call取30位相对PC的字偏移，而ba取22位相对PC的字偏移，jmpl取由计算两个寄存器之和得到的32位绝对字节地址。尽管第一种和第二种指令对将调用转换为分支不会产生困难，但最后一种指令要求我们在寄存器中形成目标地址。

bbs.theithome.com

15.2 过程集成

过程集成（procedure integration）用过程体的副本替代一个过程调用，它也叫做自动内联（automatic inlining）。这是一种非常有用的优化，因为它将不透明的对象，如对有别名的变量和参数有未知影响的对象，转变成不仅能暴露这种影响（参见第19章），而且能作为调用过程的一部分而被优化的对象。

有些语言给程序员提供了一定程度的控制内联的手段。例如C++可为过程指定一种明显的inline属性。Ada也提供了类似的方便。两种语言提供的都是过程的特征，而不是调用端的特征。尽管提供这种选择是受欢迎的，但它的作用和识别力都不如自动的过程集成。自动的过程集成器可以分辨不同的调用端，并能够根据机器特有的与性能相关的准则选择要集成的过程，而不是根据用户的直觉。

当内联进来的过程体能够使得原来被循环中的过程调用所抑制的循环转换得以进行时，或者当被内联进来的过程体本身是一个循环，而它的嵌入使得含有此过程调用的循环能转变成嵌套循环时，对内联进来的过程体进行优化就尤其有价值。这种情形的经典例子是Linpack中的过程saxpy()，图15-5随同其调用上下文一起给出了该过程。用saxpy()的过程体替换了sgefa()中对它的调用，并重新命名标号和变量n，使它们不致于冲突之后，结果很容易简化成图15-6所示的嵌套循环。这个结果是一个双层嵌套循环，对它可以应用一系列有价值的优化。

465

```
      subroutine sgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      real a(lda,1)
      real t
      integer isamax,j,k,kp1,l,nm1
            . . .
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20         continue
               call saxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
            . . .
      subroutine saxpy(n,da,dx,incx,dy,incy)
      real dx(1),dy(1),da
      integer i,incx,incy,ix,iy,m,mp1,n
      if (n .le. 0) return
      if (da .eq. ZERO) return
      if (incx .eq. 1 .and. incy .eq. 1) go to 20
      ix = 1
      iy = 1
      if (incx .lt. 0) ix = (-n+1)*incx + 1
      if (incy .lt. 0) iy = (-n+1)*incy + 1
      do 10 i = 1,n
          dy(iy) = dy(iy) + da*dx(ix)
          ix = ix + incx
          iy = iy + incy
   10 continue
      return
   20 continue
      do 30 i = 1,n
          dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
```

图15-5 Linpack例程saxpy()和它在sgefa()的调用上下文

bbs.theithome.com

```
      subroutine sgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      real a(lda,1)
      real t
      integer isamax,j,k,kp1,l,nm1
            . . .
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20         continue
               if (n-k .le. 0) goto 30
               if (t .eq. 0) goto 30
               do 40 i = 1,n-k
                  a(k+i,j) = a(k+i,j) + t*a(k+i,k)
   40         continue
   30      continue
           . . .
```

图15-6 在集成saxpy()后的Linpack例程sgefa()的一段

在决定一个编译系统需要在多大范围内提供过程集成，以及如何根据这种决定来实现时，有若干需要考虑的问题。

首先是跨多个编译单位、还是仅在一个编译单位内提供过程集成？如果是前者，则需要有一种途径来保存各个过程的中间表示，或更可能的是将多个编译单位的所有过程都保存在文件中，因为一般不可能依赖编译器的用户来决定要内联的是哪个过程。如果是后者，则不需要这种机制——所需要的只是在一次编译过程中，为进行适当的内联而保存必要的中间代码。事实上，在后一种情况下可选择在源代码形式上做这种转换。

第二，若提供跨编译单位的过程集成，则需要决定是否要求调用者和被调用者都使用相同的语言编写，还是允许用不同语言编写？这里主要的考虑是，不同的语言有不同的参数传递约定和访问非局部变量的约定，并且被内联的过程无疑必须遵守这些约定。处理这种不同参数传递约定的一种技术是，在源语言中提供诸如“external *language_name procedure_name*”的声明作为分开编译的过程的接口，以便指明书写外部过程所用的源语言。这可使得对一个外部过程的调用遵循其声明指明的语言的参数传递约定。

第三，在跨编译单位的过程集成中，是否需要保存已内联过程的中间代码副本？具体地，若干语言限制过程只在它们所嵌套的作用域内是可见的。例如，Pascal的嵌套过程，Modula语言族和Mesa的无接口过程，Fortran 77的语句函数，以及Fortran 90未声明为外部的过程都是这种情况。如果保存中间代码的惟一目的是为了实现过程集成，显然就不需要将这种过程的副本保存在为一个编译单位所保存的中间代码中，因为它们不会在其作用域之外被引用。另一方面，如果保存中间代码的目的是为了在程序设计环境中，当对源代码作了改变后，能减少重新编译的时间，则保存它们显然就是必须的。

466
~
467

第四，已知一个过程已嵌入到了所有可见的调用点，是否还有必要编译这个过程的一个完整的副本？如果曾经取了这个过程的地址，或者在当前不可见的其他编译单位调用了它，则可能有这种必要。

最后，是否应对递归过程执行内联？显然在还未发现它们的所有调用之前不应内联它们，因为否则的话这会是一个无穷过程，但是，为减少调用开销而内联递归过程一到两次则可能是值得的。

bbs.theithome.com

判断什么样的过程值得内联时，我们需要回答若干策略问题，记住我们的目的是为了加快程序的执行速度。从表面上看，似乎在每一个调用点内联每一个过程都会使速度加快，但一般并不是这样，因为它可能导致任意增大目标代码的体积，并且可能由于耗尽了资源而导致编译终止。这并不意味着内联递归过程一定就不好，而只是我们必须知道什么时候应当做它和什么时候应当停止。

增加目标代码的体积有几个潜在的缺点，最重要的一点是它对高速缓存缺失的影响。由于处理机速度和存储器速度之间差别的进一步增大，高速缓存缺失成为决定整体性能的越来越重要的因素。因此决定内联什么样的过程，需要以启发式策略或程序运行剖面信息的反馈作为依据。一些典型的启发式策略考虑了下面一些因素:

1. 过程体的体积（越小越好）;

2. 对这个过程的调用个数有多少（如果只有一个调用，内联它将几乎总是能减少执行时间）;

3. 过程是否在循环内被调用（如果是，内联它很可能给其他优化提供显著的优化机会）;

4. 特定的调用是否含有一个以上的常数值参数（如果含有的话，被内联的过程体很可能比不内联的情形优化得更好）。

一旦选择好了在什么调用点内联什么样的过程是值得的标准之后，剩余的问题就是如何实现这种内联。显然，这个处理过程的一部分是用对应的过程体替换调用。为了通用起见，我们假定在中间代码级别上来做这种处理，因此我们能做跨语言的内联。我们需处理的三个主要问题是: (1)满足所涉及语言的参数传递约定，(2)处理调用者和被调用者之间的名字冲突，(3)处理静态变量。

首先，如果语言没有提供类似于“external *language_name procedure_name*”的声明，为了能确定参数的结合要做些什么，以及如何使它们工作，过程集成必须包含足够的关于所涉及语言的参数传递机制的信息。例如，它不能用一个传值的C参数去匹配传地址的Fortran参数，
468 除非这个C参数是指针类型。类似地，也不能只轻率地用调用者的变量名取代被调用者的传值参数，例如图15-7所示那样，导致对调用者的变量以假乱真的赋值。图15-7中，变量a同时出现在f()和g()中；直接用g()的正文替换对它的调用将导致对调用者的a产生错误的赋值。

```
g(b,c)
int b, c;
{   int a, d;
    a = b + c;
    d = b * c;
    return d;
}
f( )
{   int a, e;
    a = 2;
    e = g(3,4);
    printf("%d\n",a);
}
```

a)

```
f( )
{   int a, e, d;
    a = 2;
    a = 3 + 4;
    d = 3 * 4;
    e = d;
    printf("%d\n",a);
}
```

b)

图15-7 由于在C中简单地用被调用过程的正文替换调用，导致被调用过程中的变量遮盖了调用过程中的变量

当对不含源程序符号名的中间代码进行处理时，第二个问题一般不会发生——符号引用通常是指向符号表登记项的指针，标号一般是指向中间代码位置的指针。如果处理的是字符表示，

bbs.theithome.com

就必须检查名字冲突，并通过重命名来区分它们，一般是重命名被调用过程体中的名字。

静态变量存在不同的问题。具体地，C的静态存储类变量的生存期包括程序的整个执行期。如果它的声明是在文件作用域内，即不在任何函数定义内，其初始化是在程序执行之前。另一方面，如果它的声明在函数之内，它就只在此函数中可见。如果多个函数中声明了名字相同的静态局部变量，它们是不同的对象。

因此，对于文件级的静态变量，在生成的目标程序中只需要它的一个副本，如果它是有初值的，则只需一次初始化。这可以通过使该变量具有全局作用域，并为它提供全局初始化来实现。

Cooper、Hall和Torczon [CooH92]的报告给出了一张表，列出了有关过程集成作用的注意事项。他们做了一个实验，在Linpack基准测试程序双精度版本中，他们集成了静态统计出的调用点中44%的调用，并因此减少了98%的动态调用次数。但是与他们期望的程序性能改善相反，当在基于R2000的MIPS系统上运行时，它实际上导致程序性能降低了8%。代码分析表明，导致性能降低的原因不是因为关键循环中高速缓存的作用和寄存器的压力，而是由于空操作的个数和浮点互锁增加了75%。导致问题的原因在于MIPS编译器遵循Fortran 77标准，并且没有 469
做数据流分析：Fortran 77标准允许编译器假设在过程的入口参数之间不存在别名，并在生成代码中利用了这一信息。MIPS编译器在没有进行过程集成的程序中是这样做了。另一方面，由于大部分关键过程都内联了，但又没有过程间数据流分析，因而无法知道它们的参数是否存在别名，故编译器按保守规则行事——它假定在这些参数之间存在别名，并因此生成了性能较差的代码。

15.3 内嵌扩展

内嵌扩展（in-line expansion）是一种用低级代码替代过程调用的机制。它的作用与过程集成类似（参见15.2节），但它是在汇编语言或机器代码级别上进行的，因此可用来用手工编写的代码序列替代高级操作，包括使用编译器决不会生成的指令。因此，它既是一种优化，也是为基本的机器运算（如设置程序状态字中的某些位）提供助记符的一种手段。

作为一种优化，内嵌扩展可用来为那种采用其他优化途径难于实现，或不可能做到的运算提供最好的运算序列。这种例子包括在具有允许条件作废下一条指令（如PA-RISC），或具有条件传送指令的体系结构中，通过提供三个模板就可以不需要任何分支指令而实现求多至4个整数组成的序列的极小值的计算[⊖]；其中，每一个模板对应2个、3个和4个操作数，并且可以如下面的LIR代码演示的那样，只用三个异或运算，且不需使用草稿寄存器，实现交换两个整数寄存器中的值：

```
ra ← ra xor rb
rb ← ra xor rb
ra ← ra xor rb
```

内嵌扩展也可作为过程集成的一种较弱的版本：用户、编译器或库的提供者可为那些很可能从内嵌受益的过程提供模板。

作为将那些与高级语言运算完全不对应的指令合并起来的一种机制，内嵌扩展提供了一种可给予它们助记符含义的方法，并使得访问它们无需过程调用开销。这使得用高级语言书写操作系统或I/O设备驱动程序比没有这些助记符时更容易。例如，如果对于一个具体的体系结构，要设置程序状态字的第15位是关闭中断的话，可以提供一个名为DisableInterrupts()的 470

⊖ 当然最多4个操作数的选择是随意的。通过提供额外的模板，求极小值的操作数个数也可以随你的需要而定；或者给定一种具有充分表示这些模板能力的语言，你也可以提供一个处理任意个数操作数的过程。

bbs.theithome.com

模板，它由如下三条指令组成。

```
getpsw   ra              || copy PSW into ra
ori      ra,0x8000,ra    || set bit 15
setpsw   ra              || copy ra to PSW
```

提供内嵌扩展能力有两种基本的机制，一种是将汇编语言序列放在模板内，另一种是通过一个专门执行内嵌的编译遍，称之为内嵌器（inliner）。对于像上面这个例子的某些情况，可能还需要第三种方法，即指明替代ra所需要的实际寄存器的方法。模板一般由一个头部，一系列的汇编语言指令和一个尾部组成。头部给出过程名字，并可包含所预期的参数个数和类型及需要的寄存器。尾部用来终止此模板。例如，如果一个特定的内嵌器所需要的信息组成包括例程名、所预期的参数的字节数、一张需要用真实寄存器来替代它们的寄存器的表，以及一系列的指令，它可有如下形式:

```
.template  ProcName,ArgBytes,regs=(r1,...,rn)
. . .
instructions
. . .
.end
```

例如，在SPARC系统上，下面的模板可用来求三个整数的最大值:

```
.template max3,12,regs=(@r1)
mov    argreg1,@r1
cmp    argreg2,@r1
movg   argreg2,@r1
cmp    argreg3,@r1
movg   argreg3,@r1
mov    @r1,resreg
.end
```

提供内嵌扩展的机制一般要提供一至多个文件以及一个编译遍，其中，文件包含了需要内嵌扩展的调用的汇编语言模板，编译遍搜索指定的模板文件，寻找在被编译模块中出现的过程名，并用对应的模板实例化后的副本替代它们的调用。如果编译过程包括汇编步骤，则基本上就是这些；如果不包括，则可以对这些模板进行预处理以产生需要的形式。

在多数情况下，这些模板需要满足语言实现的参数传递约定，而代码的质量将在其后执行
471 的优化中，或因为内嵌的功能之一就是尽可能多地消除参数传递开销而得到改善。通常，消除参数传递开销所需做的只是频繁的寄存器合并（参见16.3.6节）。

15.4 叶例程优化和收缩包装

叶例程（leaf routine）是处于程序调用图中叶子位置的过程，即不调用其他过程的过程。*叶例程优化*（leaf-routine optimization）利用过程是叶例程来简化它的参数传递，并尽可能地删除过程入口和出口处理中为能够调用其他过程而引入的开销。这种优化所能做的具体改变随过程需要的临时空间大小及体系结构和所采用的调用约定而变化。

收缩包装（shrink wrapping）是叶例程优化的一种推广,它可以作用于调用图中不是叶子的例程。其思想是将过程的入口处理代码和出口处理代码沿着过程内的某些控制流分别向前和向后移动,直到它们“相撞”，因而可以被删除掉,或者直到它们包围了一个含有一个或多个调用的区域，并因此围住了过程的一个只需在此部分才正确有效地行使其职责的最小部分。

15.4.1 叶例程优化

初看人们会对许多程序中的过程是叶例程的百分比如此之高而感到惊奇。另一方面，对一

bbs.theithome.com

越是公司制作请尊重作者版权

些简单情形进行考察之后就会看出，实际情况确实应该如此。具体考虑一个程序，它的调用图是一棵二叉树,即一棵每一个结点至多只有两个后继的树。通过推导不难证明，这种树的叶子结点的数目比非叶子结点的数目要多，因此在它的调用图中有半数以上的过程是叶例程。当然，这种比例并不适合所有情况：每个结点有两个以上后继的树会使这个比率增大，而那种不是树的图或那种包含递归例程的图则可能将此比率减少为0。

因此，降低叶例程调用开销的优化常常是大受欢迎的，如我们将看到的那样,它需要付出的努力也相对较小。确定是否可应用叶例程优化有两个主要的因素。第一个是显然的——这种例程没有调用其他例程。第二个与体系结构有关，并需要稍微多做一些工作。我们必须确定这个过程需要多少存储空间，包括寄存器和栈空间。如果它需要的寄存器个数小于可用的调用者保护的寄存器和短期用途的草稿寄存器个数，则可调整它只使用这些寄存器。对于没有寄存器窗口的体系结构，这种寄存器的个数是由软件约定的，或由一个过程间的寄存器分配程序设置(参见19.6节)，因此，可以用一种有利于叶例程优化的方式来进行分配。对于SPARC，它有独立于过程调用和返回的保存和恢复寄存器窗口的指令，因此需要做的只是使被调用过程不包含save和restore指令，并且限制它只使用调用者的out寄存器集合中的寄存器和草稿寄存器。 472

如果叶例程也不需要栈空间，例如由于它不操纵任何需要对其元素寻址的局部数组，并且有足够的寄存器存放它的标量，则不需要创建和收回叶例程的栈帧。

如果叶例程很小，或只在几个地方调用，它可能是过程集成的最好候选者。

15.4.2 收缩包装

前面给出的收缩包装的定义不是十分精确。当我们移动入口和出口处理代码至所包围的最小代码段中含有调用，或需要使用被调用者保护的寄存器时，我们可能最终将那段代码放置在循环内，或者可能在不同的控制流路径上创建它的多个副本。两种情况都会导致浪费——前者浪费时间，而后者浪费空间，通常也浪费时间，并且还可能是不正确的。考虑图15-8的流图。如果基本块B3和B4都需要将被调用者保护的寄存器分配给变量a，我们可能会在这两个基本块的每一个前面放置寄存器保护代码，在B4之后放置恢复代码。如果这样做，并且执行路径包含了B3和B4的话，就会导致在B4的入口保存错误的值。

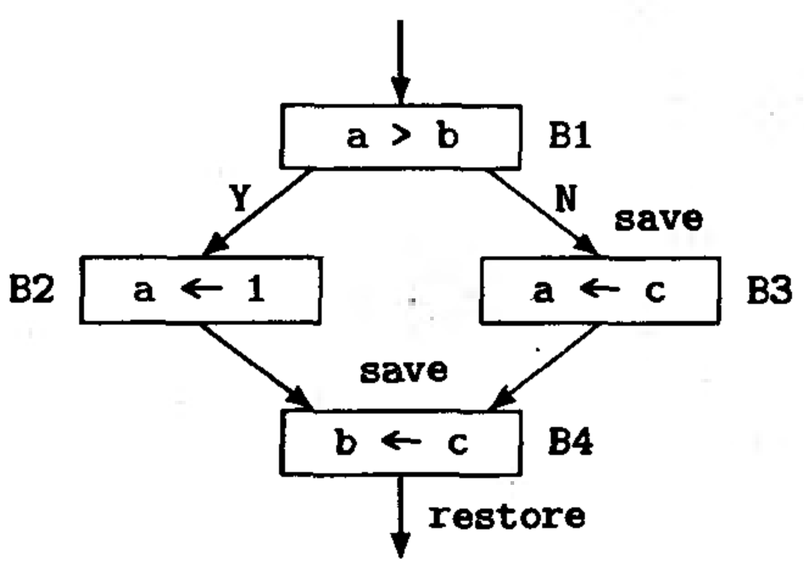


图15-8 不正确地放置保存和恢复分配给变量a的寄存器的代码的例子

相反，我们的目的是移动入口处理和出口处理代码包围需要它们的最小代码段，同时要求它们不在循环内，并且没有导致刚才描述的那种问题。为了做到这一点，我们采用由Chow [Chow88]开发的数据流分析，这种分析方法使用了部分冗余消除数据流分析中用到的一些相似的性质。对于一个基本块*i*，我们定义*RUSE*(*i*)是基本块*i*中用到的或定义的寄存器集合。下面我们定义称为寄存器可预见性和寄存器可用性的两种数据流性质。一个寄存器在流图的一点上是可预见的（anticipatable），如果从那一点开始的每一条执行路径都包含该寄存器的定义或使用；一个寄存器是可用的（available），如果到达那一点的每一条执行路径都包含该寄存器的定义或使用（参见13.3节部分冗余删除中使用的类似属性）。我们用*RANTin*(*i*)、*RANTout*(*i*)、*RAVin*(*i*)、*RAVout*(*i*)表示在每一个基本块*i*的入口和出口的数据流属性。于是，我们有数据流方程 473

bbs.theithome.com

$$RANTout(i) = \bigcap_{j \in Succ(i)} RANTin(j)$$

$$RANTin(i) = RUSE(i) \cup RANTout(i)$$

和

$$RAVin(i) = \bigcap_{j \in Pred(i)} RAVout(j)$$

$$RAVout(i) = RUSE(i) \cup RAVin(i)$$

其中，初始时$RANTout(\texttt{exit}) = RAVin(\texttt{entry}) = \emptyset$。注意这些集合可用位向量来表示，对于至多32个寄存器的机器而言，这种位向量是一个字。

收缩包装的思想是，在一个使用是可预见的地方插入保护寄存器的代码，在一个使用是可用的地方插入恢复代码。注意，这两个问题是对称的，因为它们的数据流方程相互是对方的镜像，而且插入保护和恢复代码的条件也是这样。于是，确定出适当的关于保护的数据流方程，同时也就自动给出了对应的关于恢复的方程。我们选择在基本块的入口并且是通向使用寄存器r的一个或多个连续基本块的最早点插入保护寄存器r的代码。对于满足此条件的基本块i，我们一定有$r \in RANTin(i)$和$r \notin RANTin(j)$，其中$j \in Pred(i)$。另外，在前面还必须没有保护过r，因为引入另一个保护代码会导致保护错误的值，因此$r \notin RAVin(i)$。于是，由此分析得出，在基本块i入口要保护的寄存器集合是

$$SAVE(i) = (RANTin(i) - RAVin(i)) \cap \bigcap_{j \in Pred(i)} (REGS - RANTin(j))$$

其中$REGS$是所有寄存器组成的集合，并且对称地，在基本块i出口要恢复的寄存器集合是

$$RSTR(i) = (RAVout(i) - RANTout(i)) \cap \bigcap_{j \in Succ(i)} (REGS - RAVout(j))$$

但是，这些保护和恢复点的选择受到两个问题的困扰，一个问题与图15-8的例子中涉及的与恢复相对应的保护有关。我们解决它的方法是分割由B2到B4的边，并将当前在B4入口的寄存器保护代码放置在这个新的空基本块中；我们也用对应的方式处理恢复代码。第二个问题是这种保护和恢复点的选择没有有效地处理保护和恢复代码所包围的是循环内的一个子图的情况。解决这个问题的方法是，根据被编译例程的控制流结构来识别嵌在循环内的子图，并将保护和恢复代码从循环中迁出。

作为这种方法的一个例子，考虑图15-9的流图，假设r1到r7用于存放参数，r8到r15是
474 被调用者保护的寄存器。则$RUSE()$的值如下：

$RUSE(\texttt{entry}) = \emptyset$
$RUSE(\texttt{B1}) = \{\texttt{r2}\}$
$RUSE(\texttt{B2}) = \{\texttt{r1}\}$
$RUSE(\texttt{B3}) = \{\texttt{r1,r2,r8}\}$
$RUSE(\texttt{B4}) = \{\texttt{r1,r2}\}$
$RUSE(\texttt{exit}) = \emptyset$

$RANTin()$、$RANTout()$、$RAVin()$和$RAVout()$的值如下：

$RANTin(\texttt{entry})$	$= \{\texttt{r1,r2}\}$	$RANTout(\texttt{entry})$	$= \{\texttt{r1,r2}\}$
$RANTin(\texttt{B1})$	$= \{\texttt{r1,r2}\}$	$RANTout(\texttt{B1})$	$= \{\texttt{r1,r2}\}$
$RANTin(\texttt{B2})$	$= \{\texttt{r1,r2}\}$	$RANTout(\texttt{B2})$	$= \{\texttt{r1,r2}\}$
$RANTin(\texttt{B3})$	$= \{\texttt{r1,r2,r8}\}$	$RANTout(\texttt{B3})$	$= \{\texttt{r1,r2}\}$
$RANTin(\texttt{B4})$	$= \{\texttt{r1,r2}\}$	$RANTout(\texttt{B4})$	$= \emptyset$

bbs.theithome.com

RANTin(exit)	$= \emptyset$	*RANTout*(exit)	$= \emptyset$
RAVin(entry)	$= \emptyset$	*RAVout*(entry)	$= \emptyset$
RAVin(B1)	$= \emptyset$	*RAVout*(B1)	= {r2}
RAVin(B2)	= {r2}	*RAVout*(B2)	= {r1,r2}
RAVin(B3)	= {r2}	*RAVout*(B3)	= {r1,r2,r8}
RAVin(B4)	= {r1,r2}	*RAVout*(B4)	= {r1,r2}
RAVin(exit)	= {r1,r2}	*RAVout*(exit)	= {r1,r2}

最后，*SAVE*()和*RSTR*()的值如下:

SAVE(entry)	$= \emptyset$	*RSTR*(entry)	$= \emptyset$
SAVE(B1)	= {r1,r2}	*RSTR*(B1)	$= \emptyset$
SAVE(B2)	$= \emptyset$	*RSTR*(B2)	$= \emptyset$
SAVE(B3)	= {r8}	*RSTR*(B3)	= {r8}
SAVE(B4)	$= \emptyset$	*RSTR*(B4)	= {r1,r2}
SAVE(exit)	$= \emptyset$	*RSTR*(exit)	$= \emptyset$

因为r1和r2用于参数传递，这里惟一感兴趣的寄存器是r8，并且*SAVE*()和*RSTR*()的值告诉我们应当在基本块B3的入口保护它，并在B3的出口恢复它，结果流图如图15-10所示。

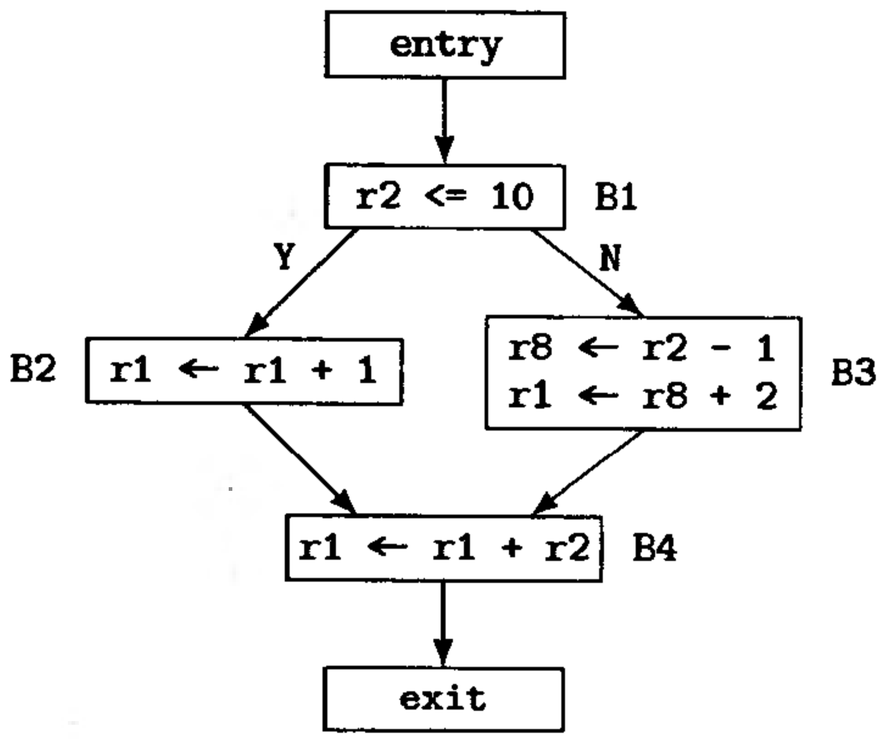


图15-9 用于收缩包装的LIR流图之例

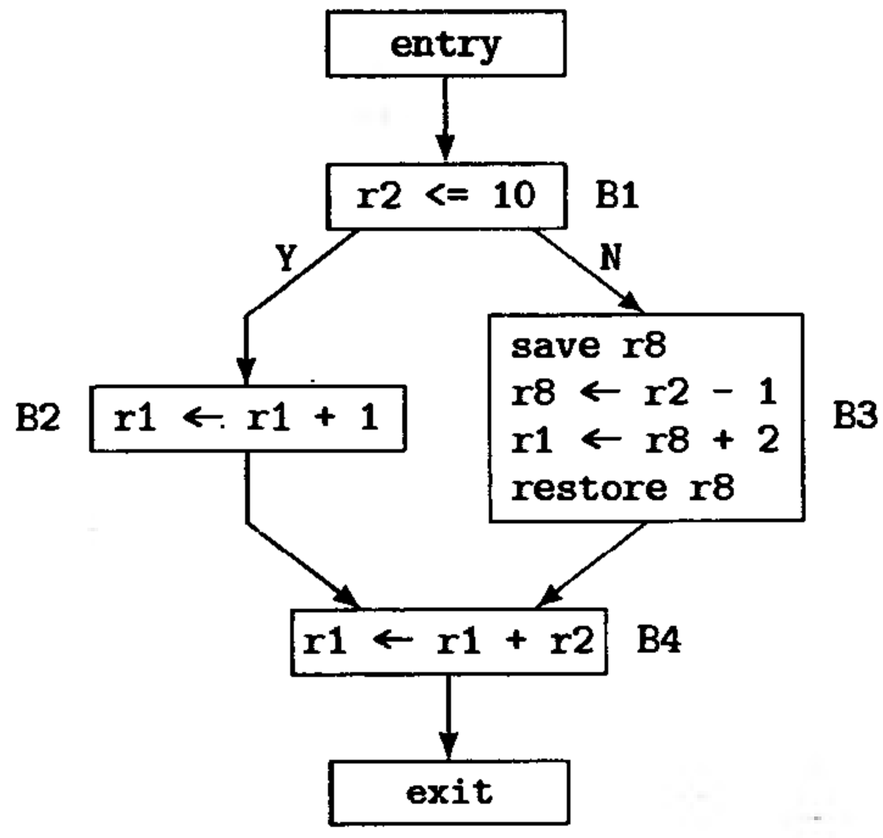


图15-10 图15-9中的例子经过收缩包装的结果

15.5 小结

475

本章我们讨论了作用于整个过程的三对优化方法，其中除一种优化方法之外，其余都不需要进行数据流分析。这三对优化如下:

1. 尾递归删除和更一般的尾调用优化将调用转变成分支。具体地说，尾调用删除识别过程中那种递归调用自己并且从调用返回之后不做任何事情的调用。这种调用总是能够转换成首先复制实参到哑参，然后转移到过程体开始处的代码。

尾调用优化处理类似的情形，但所调用的例程不必与调用过程相同。这种优化所做的事情与尾递归删除相同，但需要更小心，因为被调用的例程可能不在同一个编译单位。为了做到这一点，我们需要知道被调用例程希望在何处找到它的参数，它的栈帧有多大，以及为了开始执行它，控制应转移到何处。

2. 过程集成和内嵌扩展是用于同一种优化的两个名字，即用被调用过程体替代其调用。但在本书中，我们用它们表示这种优化的两种不同版本: 过程集成在编译处理的较早阶段进行，以便利用那些以单个过程作为其优化作用域的许多优化（例如，集成一个其过程体是循环且处

bbs.theithome.com

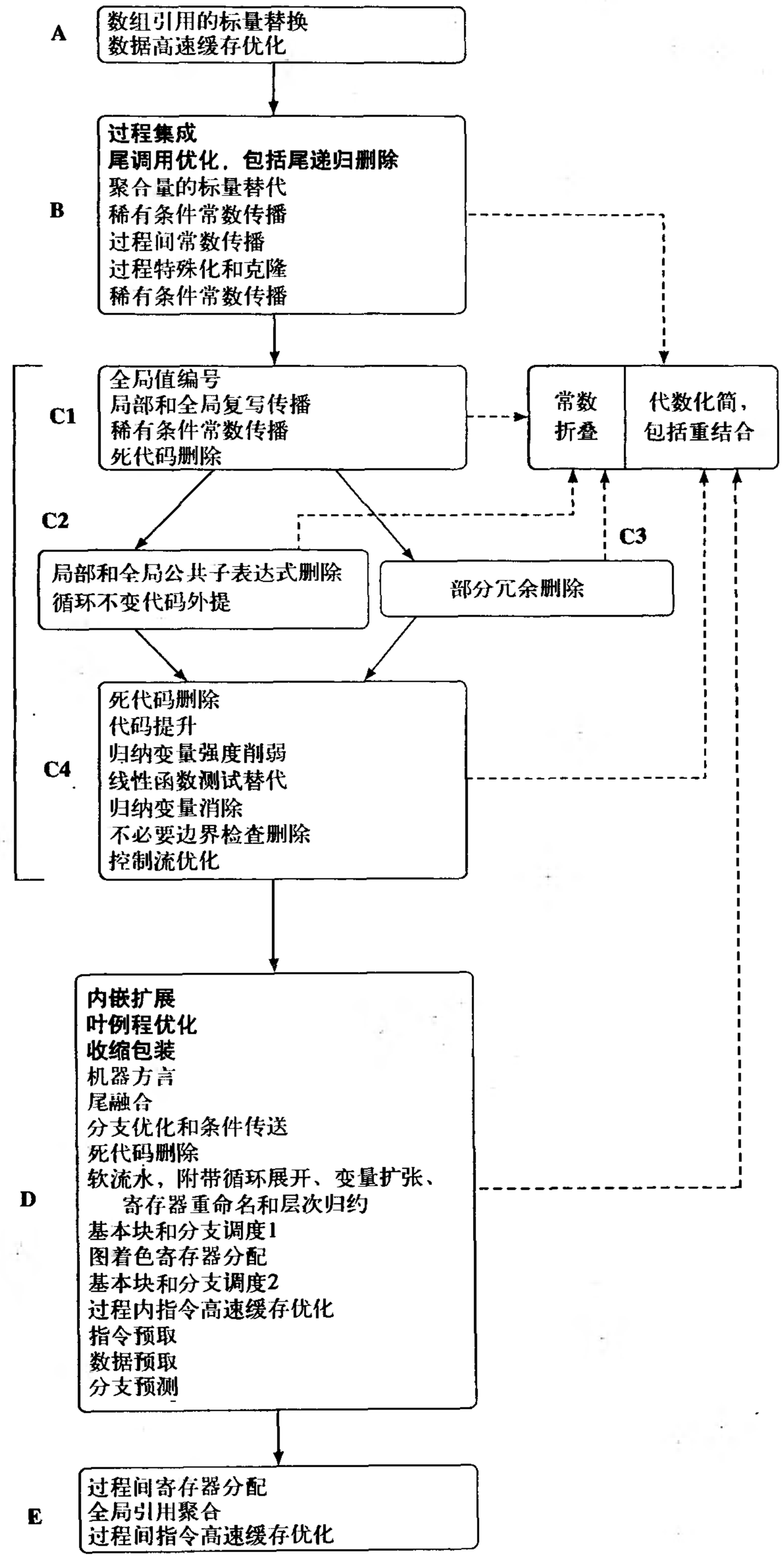


图15-11 过程优化（用黑体字标明）在激进优化编译器中的位置

bbs.theithome.com

在循环体中的调用），而内嵌扩展在编译过程的较后阶段进行，以便利用那些在特定体系结构中特别有效的低级操作。

3. 叶例程优化利用了所调用过程中占较大比例的是叶例程的这一事实。叶例程是本身不含调用的例程，因此它们不必有一般例程的完整调用负担（栈帧、寄存器保护与恢复等）。收缩包装在效果上是叶例程优化的一种推广，它沿过程的控制流路径向前移动入口处理代码并向后移动出口处理代码，使得一旦它们相遇便相互消灭对方。作为结果，经过收缩包装处理的例程中，有些路径可能含有完整的入口处理和出口处理代码，而另一些路径则没有。

476 ~ 477

图15-11用黑体字标明了这些优化通常在优化顺序中的位置。

15.6 进一步阅读

Dongarra等人在［DonB79］中介绍了Linpack基准测试程序。15.2节介绍的Cooper、Hall和Torczon关于过程集成的实验数据描述是在［CooH92］中找到的。

［Chow88］中介绍了Chow的收缩包装方法。

15.7 练习

15.1 在尾调用优化中，我们怎样才能保证有足够的栈空间分配给被调用者？(a)在什么条件下可以在编译时保证这一点？(b)将这一工作分为编译时和连接时是否能使其简化或使其应用范围更广泛？如果能，解释怎样做，如果不能，解释为什么不能？

ADV 15.2 (a)推广尾调用优化使它能发现形成递归循环的一组例程，即对于例程r_1到r_k，r_1只调用r_2，r_2只调用r_3,…，r_k只调用r_1。(b)什么条件下这些例程体可以合并成单个例程体，如何确定这种条件是否成立？

15.3 在做叶例程优化时，我们怎样才能保证对所有的叶例程都有足够的栈空间？(a)在什么条件下可以在编译时保证这一点？(b)将此工作分成编译时和连接时来做是否能使它简化，或使其应用范围更广泛？如果能，解释如何做，如果不能，解释为什么不能？

15.4 设计一种保存一次编译单位生成的MIR和符号表的紧凑格式，使得能够做跨编译单位的过程集成。假定所有编译单位编译的都是相同源语言的模块。

15.5 遵循15.3节给出的约定，写出对LIR代码执行内联的ICAN代码。作为这个练习的一部分，设计一种表示模板文件的ICAN数据结构，并通过调用`read_templates (`*file*, *struc*`)`将它读入到适当的数据结构中，其中*file*是模板文件名，*struc*是存放读入内容的结构。

15.6 编写一个对LIR代码做叶例程优化的ICAN过程，假定参数传递在寄存器r1至r6中，r7至r13是调用者保存的寄存器。要检查是否需要栈空间，并且仅当需要时才分配它。

15.7 编写一个ICAN例程`Tail_Call_Opt(`*en*1, *n*1, `ninst1, LBlock1,` *en*2, *n*2, `ninst2, LBlock2)`，它的参数给出两个LIR过程，其中第一个过程含对第二个过程的尾调用，这个例程修改它们的代码使得用分支替代此尾调用。假定第一个过程从r1开始的寄存器依次传递*nargs*个参数给第二个过程，帧指针和栈指针分别是r20和r21，且*en*1和*en*2是这两个过程的入口基本块的个数。

478 ~ 479

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第16章　寄存器分配

本章的内容包括寄存器分配和指派，对于多数体系结构而言，这是两种最重要的优化。它们涉及的问题是如何使得CPU中寄存器之间的信息交换量最小。寄存器的个数一般很少，但访问速度很快，而且无论在寄存器与存储器中间存在什么样的存储层次结构，包括一级或多级高速缓存和主存储器，访问它们都比访问寄存器要慢，并且它们都比寄存器的容量要大。一般而言，离寄存器越远，其容量就越大，但其访问速度也越慢。

寄存器分配最好在低级中间代码或汇编语言上进行，因为它要求对存储器的所有存取（包括它们的地址计算）都必须明确地表示出来。

我们首先讨论一种较快且相当有效的局部方法，这种方法与使用次数和循环嵌套有关。接着详细介绍一种更有效的利用图着色进行全局寄存器分配的方法，并对其他一些使用图着色但一般没这么有效的方法给出简单的评论。我们也简要地提及一种将寄存器分配看成是背包问题的方法和三种利用过程的控制树来指导分配的处理方法。

这一章的重点是图着色全局寄存器分配。图着色全局寄存器分配通常能产生非常有效的分配，而且在编译速度上没有较大的开销。图着色全局寄存器分配从另一个角度看待两个对象必须同时在寄存器中这一事实，即将必须同时在寄存器的两个对象看成是它们排斥相同的寄存器。它用图中的结点表示这些对象，并用结点之间的弧表示这种排斥（称为冲突(interferences)）；结点也可以代表实际的寄存器，而弧则代表排斥，例如存储访问中的基地址不能使用寄存器r0就是一种排斥。给出一个对应于整个过程的图，这种方法尝试着对结点进行着色，其中颜色的数目等于可用的实际寄存器个数，使得每一个结点被指派一种与它相邻结点不同的颜色。如果不能实现这一点，则引入额外的代码将一些量保存到存储器中，并在需 481
要时重新将它们取回到寄存器，这个过程重复直到实现满意的着色为止。如我们将看到的，即使是非常简单的图着色问题也是NP－完全的，因此使得全局寄存器分配尽可能有效的最重要方法之一是使用高度有效的启发式策略。

关于寄存器分配更进一步的讨论在19.6节给出，其中讨论了过程间的方法。这些方法中有一些是作用于汇编语言级别以下的代码，即，作用于带有数据使用样式信息注释的可重定位目标模块。

16.1　寄存器分配和指派

寄存器分配（register allocation）确定在程序执行的每一点上，因放在机器的寄存器中而可能获益的值中（变量、临时变量、大常数），哪些值应当放在寄存器中。寄存器分配很重要，因为寄存器几乎总是稀有资源——它几乎总是不足以容纳你想放入其内的所有对象——并且因为在RISC系统中，除数据传送之外的几乎所有运算，所操作的都完全是寄存器而不是存储器的内容。而且，在现代CISC的实现中，寄存器与寄存器的运算比那些含一个或两个存储器操作数的运算要明显快很多。图着色是全局（过程内）寄存器分配的一种高度有效的方法，我们也简要地介绍一种与它有关的、称为基于优先级的图着色方法。在19.6节，我们讨论在编译时或连接时施加于整个程序的过程间寄存器分配方法。

bbs.theithome.com

版权所有，侵权必究

寄存器指派（register assignment）确定每一个已分配为可放在寄存器中的值应当放在哪个寄存器中。寄存器指派对于RISC体系结构没有多少工作要做，因为所有寄存器要么是一样的，要么分成两类几乎相同的集合——一类是通用或整数寄存器，另一类是浮点寄存器——并且它们各自执行的运算是互斥的或几乎是互斥的。有时存在的一个明显例外是，这两类寄存器集合一般都可用于容纳要从存储器的一个区域复制到另一个区域的一字或双字的值，选择使用哪一个集合取决于当时占用寄存器的其他值是什么。第二个例外是在32位的系统中，一般限制双字量使用一对奇偶寄存器，因此需要注意保证正确地指派它们。对于CISC系统，寄存器指派典型地需要考虑某些寄存器的特殊用法，例如寄存器用作栈指针或被字符串操作指令隐式地使用，如Intel 386体系结构系列中出现的情形。

那种在中级中间代码上进行全局优化的编译器中，寄存器分配几乎总是在生成低级代码或机器代码之后。它的前面是指令调度（参见17.1节），可能还有软流水（参见17.4节），并且其后可能还有另一遍指令调度。那种在低级中间代码上进行全局优化的编译器中，寄存器分配常
482 常属于最后做的少数几种优化之一。在这两种方式中，重要的是必须在寄存器分配之前暴露所有的寻址计算，如数组元素的寻址，以便在寄存器分配处理中能考虑到它们对寄存器的使用。

如果寄存器分配是在中级中间代码上进行的，一般需要保留少数寄存器给代码生成作为临时之用，它们用于那些没有分配寄存器的量，以及用于诸如开关表这样的更为复杂的结构。这对于基于优先级的图着色方法（参见16.4节）而言是一个明显障碍，因为基于优先级的图着色方法对这些保留用于指定目的的寄存器个数有限制，而一般情况下我们事先并不知道实际需要多少个，因此必须保留可能需要的最大寄存器个数，从而减少了分配器可用的寄存器个数。

在着手讨论全局寄存器分配之前，我们考虑有哪几种对象应当作为寄存器分配的候选，并简要介绍两种较老的局部方法，第一种方法是Freiburghouse [Frei74]开发的，第二种用于PDP-11 BLISS编译器和它的后几代版本，包括21.3.2节讨论的DEC GEM编译器。在许多体系结构中，包括所有的RISC，所有操作都是在寄存器之间进行的，甚至从存储器到存储器之间传送对象，在实现上也是先将它们取到寄存器，然后再将它们存到存储器中，因此初看起来似乎每一种对象都应作为候选，但事实不完全是这样——输入/输出普遍地是从存储器到存储器来完成的，而不通过寄存器，并且共享多处理机中处理机之间的通信也几乎完全是通过存储器来实现的。还有，可以用指令的直接数域表示的小常数一般也不必考虑作为候选，因为它们可以通过比占据寄存器更有效的方式来使用。事实上，其他所有种类的对象都应当考虑作为寄存器分配的对象，如局部变量、非局部变量、较大而不能放在指令直接数域的常数、临时变量，等等，甚至也可以考虑单个的数组元素（参见20.3节）。

16.2 局部方法

第一种局部方法是启发式方法，依据大多数程序的多数执行时间都花在循环上这一原理，它给予内层循环的权重要高于外层循环的，更高于不包含在循环内的代码。其思想是启发式地，或根据剖面信息确定各种可分配对象的分配效益。如果没有可用的剖面信息，它通过将变量由于分配了寄存器而得到的节省值乘以一个与循环嵌套深度有关的因子来估计分配效益，这个因子一般是10^{depth}，其中*depth*是第*depth*层循环[⊖]。此外，也应考虑变量在基本块入口或出口的活跃性，因为活跃变量在基本块的出口需要保存，除非有足够多的寄存器可指派一个给它。我们
483 定义下面一些量：

⊖ 有些编译器使用8^{depth}，仅仅是因为乘以8的运算可以用循环左移来计算。

bbs.theithome.com

1. *ldcost*是目标机取数指令的执行代价。

2. *stcost*是存数指令的执行代价。

3. *mvcost*是寄存器到寄存器的传送指令的代价。

4. *usesave*是每一次使用那种存放在寄存器中而不是在存储器中的变量所节约的代价。

5. *defsave*是每一次对那种存放在寄存器中而不是在存储器中的变量赋值所节约的代价。

于是，对一个特定的变量v，每次基本块i被执行时，它的执行时间上的净节约代价是*netsave*(v, i)，其定义如下:

$$netsave(v, i) = u \cdot usesave + d \cdot defsave - l \cdot ldcost - s \cdot stcost$$

其中u和d分别是变量v在基本块i中的使用和定值次数；l和s=0或1，分别取决于在该基本块的开始是否需要取v或在该基本块的结束是否需要存v。

因此，如果L是一个循环，并且i的取值范围是此循环内的基本块，则

$$10^{depth} \cdot \sum_{i \in blocks(L)} netsave(v,i)$$

是在循环L中分配v到寄存器得到的合理的估计获益值[⊖]。给定R个可分配的寄存器——它们一般总是少于总的寄存器个数，因为其中有一些必须保留用于过程连接、短期使用的临时量等——在计算了这种估计值后，你只需简单地在每一个循环或循环嵌套中分配具有最大估计获益值的R个变量。对循环嵌套进行寄存器分配之后，再使用同样的获益度量为循环之外的代码进行分配。

我们有时也通过考虑一个基本块i的P个前驱和S个后继来改善这种分配方法。如果这些前驱和后继基本块都指派变量v到相同的寄存器中，则对于基本块i，该变量的l和s的值都是0。在考虑基本块i的同时考虑它的前驱和后继时会发现，我们有可能将变量v放在与在基本块i前后的某些或所有基本块所分配的寄存器不同的寄存器中。如果是这样，这个变量所带来的额外代价至多是$(P + S) \cdot mvcost$，即每一个前驱和后继基本块有一次传送代价。

这种方法实现简单，常常也工作得非常好，而且直到下一节将介绍的全局方法用于实际产品之前，一直都是过去优化编译器中盛行的方法，例如IBM的360和370系列机的Fortran H优化编译器。

PDP-11的BLISS优化编译器将寄存器分配看作是装包问题。它确定临时量的生命期，并根据是否是下面几种情形而将它们分为4组。

1. 必须给它分配特定的寄存器;

2. 必须给它分配某个寄存器;

3. 可以给它分配一个寄存器或存储器单元;

4. 必须分配到存储单元。

然后，它根据与分配到特定寄存器或任意寄存器有关的代价度量标准来排列可分配的临时变量，最后尝试一系列的置换，将临时变量装到寄存器或存储器中，并尽可能地优先装到寄存器中。由这种方法派生出来的一种方法目前仍在DEC的Alpha机器的GEM编译器中使用。

16.3 图着色

16.3.1 图着色寄存器分配概述

早在1971年John Cocke就认识到可将全局寄存器分配看作图着色问题，但直到1981年才由

⊖ 可细化这种测算，使它能对条件执行的代码加权，使之与预期的执行频率或测量到的执行频率成正比。

484

bbs.theithome.com

版权所有 侵权必究

Chaitin设计和实现了这种分配器。该分配器用于IBM 370 的一个实验性的PL/I编译器中，之后不久Chaitin和他的一群同事又将它改写，并用于IBM 801实验RISC系统的PL.8编译器。从此以后，它的各种版本以及由它派生出来的各种分配器就一直被许多编译器所使用。它的一种最成功的设计是由Briggs开发的，本章余下部分的很多内容正是基于Briggs的这个设计（关于进一步阅读参见16.7节）。

图着色全局寄存器分配的基本思想可表示为如下5个步骤（尽管步骤2到步骤4都过度简化了）：

1. 在代码生成或优化期间（先于寄存器分配的任何遍），或者作为寄存器分配的第一个步骤，给可以指派到寄存器的对象分配不同的符号寄存器，比如s1，s2，…，需要多少符号寄存器存放所有对象（包括源程序变量、临时变量、大常数等），便分配多少。

2. 确定什么样的对象应当作为分配寄存器的候选者（这可以简单地就是s*i*，但在16.3.3节介绍了一种更好的选择）。

3. 构造所谓的冲突图，这种图的结点代表可分配的对象和目标机的实际寄存器，弧（即无向边）代表冲突，其中两个可分配的对象有冲突，如果它们同时是活跃的；一个对象和一个寄存器冲突，如果不能或不应当给这个对象分配那个寄存器（例如，一个整操作数和一个浮点寄存器）。

4. 用*R*种颜色给冲突图着色，使得任何两个相邻的结点具有不相同的颜色，其中*R*是可用的寄存器个数（这叫做*R*色着色（*R-coloring*））。

5. 给每一个对象分配颜色与它相同的寄存器。

在进入细节之前，我们先给出一个例子说明基本的处理过程。假设我们有如图16-1a所示
485 的简单代码，其中y和w在代码结束点是死去的，有三个可用的寄存器(r1、r2和r3)，并进一步假设z不能占据r1。我们首先给这些变量指派符号寄存器s1，…，s6，如图16-1b所示。注意，其中x在第1和第6行的两次定值（如下一节描述的）已指派了不同的符号寄存器。于是有s1与s2相冲突，因为s2是第2行定值的，它位于s1在第一行的定值和s1在第3、4和5行的使用之间；s4和r1冲突，因为z被限制不能在r1中。所得到的冲突图如图16-2所示。它可用三种颜色（寄存器的个数）着色，其中s3、s4和r3为红色，s1、s5和r1为蓝色，s2、s6和r2为绿色。因此，一种合法的寄存器指派是：将s1和s5放在r1中，s2和s6放在r2中，s3和s4放在r3中（如图16-1c所示）。读者不难做出验证。

```
1        x ← 2
2        y ← 4
3        w ← x + y
4        z ← x + 1
5        u ← x * y
6        x ← z * 2
```

a)

```
s1 ← 2
s2 ← 4
s3 ← s1 + s2
s4 ← s1 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

b)

```
r1 ← 2
r2 ← 4
r3 ← r1 + r2
r3 ← r1 + 1
r1 ← r1 * r2
r2 ← r3 * 2
```

c)

图16-1 a) 用于图着色寄存器分配的一个简单例子；b) 给它指派的符号寄存器；c) 用三个寄存器对它的一种分配，假定y和w从这段代码出来时是死去的

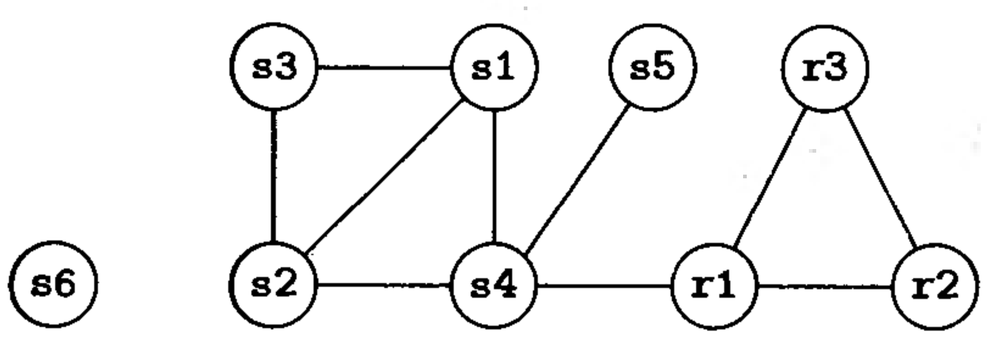


图16-2 图16-1b中代码的冲突图

bbs.theithome.com

下面，我们考虑这种方法的细节以及它们与上面的这个框架有什么不同。

16.3.2 顶层结构

图着色寄存器分配中使用的全局类型定义和数据结构如图16-3所示，其中每一种类型定义和数据结构在第一次使用时我们再给出其具体描述。

```
Symbol = Var ∪ Register ∪ Const
UdDu = integer × integer
UdDuChain = (Symbol × UdDu) ⟶ set of UdDu
webrecord = record {symb: Symbol,
                    defs: set of UdDu,
                    uses: set of UdDu,
                    spill: boolean,
                    sreg: Register,
                    disp: integer}
listrecd =  record {nints, color, disp: integer,
                    spcost: real,
                    adjnds, rmvadj: sequence of integer}
opdrecd = record {kind: enum {var,regno,const},
                  val: Symbol}

DefWt, UseWt, CopyWt: real
nregs, nwebs, BaseReg, Disp := InitDisp, ArgReg: integer
RetReg: Register
Symreg: array [··] of webrecord
AdjMtx: array [··,··] of boolean
AdjLsts: array [··] of listrecd
Stack: sequence of integer
Real_Reg: integer ⟶ integer
```

图16-3 图着色寄存器分配中使用的全局类型定义和数据结构

这个寄存器分配器的总体结构如图16-4所示，分配过程如下：

1. 首先Make_Webs()合并相交的（即包含公共使用的）du链来形成网，网是寄存器分配的对象。一个网（web）是这种du链的最大并集，其中对于每一个定值d和使用u，要么u在d的du链中，要么存在着$d=d_0, \cdots, u_0, d_n, u_n=u$，使得对于每一个$i$，$u_i$同时在$d_i$和$d_{i+1}$的du链中。每一个网指派一个不同的符号寄存器号。Make_Webs()也调用MIR_to_SymLIR()将输入的Block中的MIR代码转换为使用符号寄存器的LIR代码，并存储它们于LBlock中；注意这不是实质性的——输入给此寄存器分配器的代码可以就是LIR代码，并且如果我们在寄存器分配之前做了其他的低级优化，则它肯定就已经是LIR或其他某种低级形式。

2. 接下来，Build_AdjMtx()建立冲突图的邻接矩阵表示。冲突图的邻接矩阵是一个两维的下三角矩阵，其中AdjMtx[i,j]是true，当且仅当在（实际或符号）寄存器i和j（其中$i>j$）之间存在一条弧；否则为false。

3. 接下来，例程Coalesce_Regs()使用邻接矩阵来合并这些寄存器，即寻找使得si和sj不相互冲突的复写指令$si \leftarrow sj$，然后用si的使用替换sj的使用，从而从代码中删除sj。如果执行了任何合并，我们从上面第一步从头开始；否则继续下一步。

4. 接下来，Build_AdjLsts()构造这个冲突图的邻接表表示，它是一个Listrecd记录类型的数组AdjLsts[1..nwebs]，每个符号寄存器有一条记录。这些记录由6个分量组成：color、disp、spcost、nints、adjnds和rmvadj；它们分别指出一个结点是否已经着色和是什么颜色、在将它溢出时使用的位移（如果需要的话）、与它相连的溢出代价、图中留

bbs.theithome.com

下的相邻结点个数、图中留下的相邻结点的表，以及已从图中删除的相邻结点的表。

486～487

5. 接下来Compute_Spill_Costs()对每一个符号寄存器计算将它溢出到存储器并恢复它到寄存器的代价。如我们下面将看到的，有些类型的寄存器内容（如大常数）可以有不同的处理，即用一种开销较小但仍能达到溢出和恢复效果的方式。

6. 然后Prune_Graph()使用两种方法，分别叫做度< R规则和乐观启发式，从冲突图的邻接表表示中删除结点（以及它们相连的弧）。

7. 之后Assign_Regs()使用邻接表来尝试给结点指派颜色，使得两个相邻的结点不会有相同的颜色。如果它成功了，则调用Modify_Code()用已指派了具有相同颜色的实际寄存器替代符号寄存器的每一个使用，并终止此分配过程。如果寄存器指派失败，则继续下一步。

8. 例程Gen_Spill_Code()为需要溢出到存储器的符号寄存器指派栈位置，然后插入它们的溢出和恢复代码（或对于前面提到的大常数情形用另一种方法，即用较小的代价重新构造一个值，而不是保存和恢复它），然后控制返回到前面第一步。

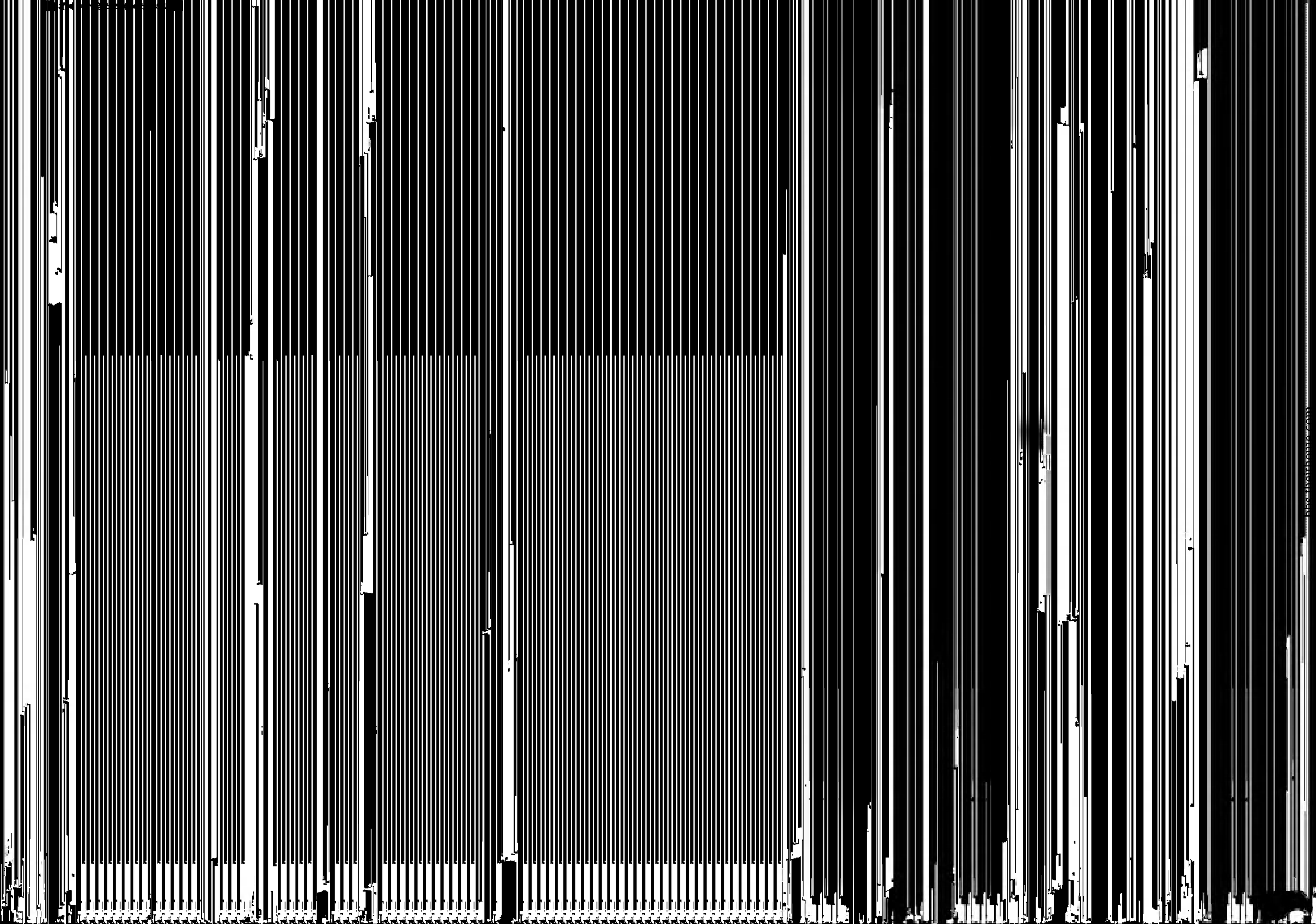
```
procedure Allocate_Registers(DuChains,nblocks,ninsts,Block,
   LBlock,Succ,Pred)
   DuChains: in set of UdDuChain
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: in array [1··nblocks] of array of MIRInst
   LBlock: out array [1··nblocks] of array [··] of LIRInst
   Succ, Pred: inout integer ⟶ set of integer
begin
   success, coalesce: boolean
   repeat
      repeat
         Make_Webs(DuChains,nblocks,ninsts,Block,LBlock)
         Build_AdjMtx( )
         coalesce := Coalesce_Regs(nblocks,ninsts,LBlock,Succ,Pred)
      until !coalesce
      Build_AdjLsts( )
      Compute_Spill_Costs(nblocks,ninsts,LBlock)
      Prune_Graph( )
      success := Assign_Regs( )
      if success then
         Modify_Code(nblocks,ninsts,LBlock)
      else
         Gen_Spill_Code(nblocks,ninsts,LBlock)
      fi
   until success
end    || Allocate_Registers
```

图16-4 图着色寄存器分配算法的顶层结构

16.3.3 网，可分配对象

我们遇到的第一个问题是确定什么样的对象应当作为寄存器分配的候选者。与简单地将那些可放入寄存器的变量作为可分配对象不同，这里的候选者是那些原来叫做名字（names），但现在叫做网的对象。网（web）的定义与16.3.2节步骤1中的定义一样。例如，在图16-1a的代码中，x在第1行的定值和它在第3、4和5行的使用属于同一个网，因为这个定值到达了这里列出的所有使用，但是x在第6行的定值属于不同的网。

bbs.theithome.com



```
procedure Make_Webs(DuChains,nblocks,ninsts,Block,LBlock)
    DuChains: in set of UdDuChain
    nblocks: in integer
    ninsts: in array [1··nblocks] of integer
    Block: in array [1··nblocks] of array [··] of MIRInst
    LBlock: out array [1··nblocks] of array [··] of LIRInst
begin
    Webs := ∅, Tmp1, Tmp2: set of webrecord
    web1, web2: webrecord
    sdu: Symbol × UdDu ⟶ set of UdDu
    i, oldnwebs: integer
    nwebs := nregs
    for each sdu ∈ DuChains do
        nwebs += 1
        Webs ∪= {⟨symb:sdu@1,defs:{sdu@2},uses:sdu@3,
            spill:false,sreg:nil,disp:nil⟩}
    od
    repeat
        || combine du-chains for the same symbol and that
        || have a use in common to make webs
        oldnwebs := nwebs
        Tmp1 := Webs
        while Tmp1 ≠ ∅ do
            web1 := ♦Tmp1; Tmp1 -= {web1}
            Tmp2 := Tmp1
            while Tmp2 ≠ ∅ do
                web2 := ♦Tmp2; Tmp2 -= {web2}
                if web1.symb = web2.symb &
                    (web1.uses ∩ web2.uses) ≠ ∅ then
                    web1.defs ∪= web2.defs
                    web1.uses ∪= web2.uses
                    Webs -= {web2}
                    nwebs -= 1
                fi
            od
        od
    until oldnwebs = nwebs
    for i := 1 to nregs do
        Symreg[i] := {⟨symb:Int_to_Reg(i),defs:nil,
            uses:nil,spill:false,sreg:nil,disp:nil⟩}
    od
    || assign symbolic register numbers to webs
    i := nregs
    for each web1 ∈ Webs do
        i += 1
        Symreg[i] := web1
        web1.sreg := i
    od
    MIR_to_SymLIR(nblocks,ninsts,Block,LBlock)
end    || Make_Webs
```

图16-7 确定图着色寄存器分配所使用的网的例程Make_Webs()

第二种类型是UdDuChain＝(Symbol × UdDu)→Set of UdDu，它表示du链。如2.7.9节所述，带有两个参数的ICAN函数等价于一个由三元组组成的集合，该三元组的类型是它的第一个参数、第二个参数和值域的类型的乘积。我们在这里使用这种等价的表示——即记类型

bbs.theithome.com

UdDuChin的成员*sdu*为形如<*s*, *p*, *Q*>的三元组的集合（其中*s*是符号，*p*是基本块位置偶对，*Q*是基本块位置偶对集合），而不是*sdu*(*s*, *p*)=*Q*的形式。

第三种类型Webrecord描述一个网，它由一个符号名、该符号的定值集合、同一符号的使用集合、一个指出它是否为溢出候选者的布尔量、一个符号寄存器或nil，以及一个位移或nil组成。

490 ~ 491

我们假设du链表示为由一个符号、该符号的定值，以及同一符号的使用集合组成的三元组，并假设每一个定值和使用都是由基本块号和（块内的）指令编号组成的偶对来表示的，即用图16-3中定义的编译器专用的类型UdDu。

全局变量nregs和nwebs的值分别是可用于分配的真实寄存器个数和网的个数，其中假定真实寄存器的编号从1到nregs，网从真实寄存器之后开始计数。

Make_Webs()首先用du链来初始化网，然后扫描每一对网，检查它们是否是相同的符号且相交，如果是则合并它们。在处理过程中，它对网计数，最后给这些网指派符号寄存器名，调用MIR_to_SymLIR()将MIR代码转换成LIR代码[⊖]，并用符号寄存器替换代码中的变量。

Make_Webs()用到了例程Int_to_Reg(*i*)，它返回整数*i*对应的真实寄存器或符号寄存器名。如果*i*<nregs，它返回第*i*个真实寄存器的名字，如果*i*>nregs，它返回Symreg[*i*].symb的值；这里没有用到这种情形，但在图16-9的MIR_to_SymLIR的代码中使用了它。

注意，如果这个寄存器分配程序的输入是SSA形式，则确定网是一件容易的事：每一个SSA形式的变量是一个du链，因为每一个SSA变量只有一个定值点。例如，在图16-8中，基本块B1中x_1的定值，B2中x_2的定值和使用，B4中x_3的定值、x_1和x_2的使用，以及B5中x_2和x_3的使用一起构成了一个网。

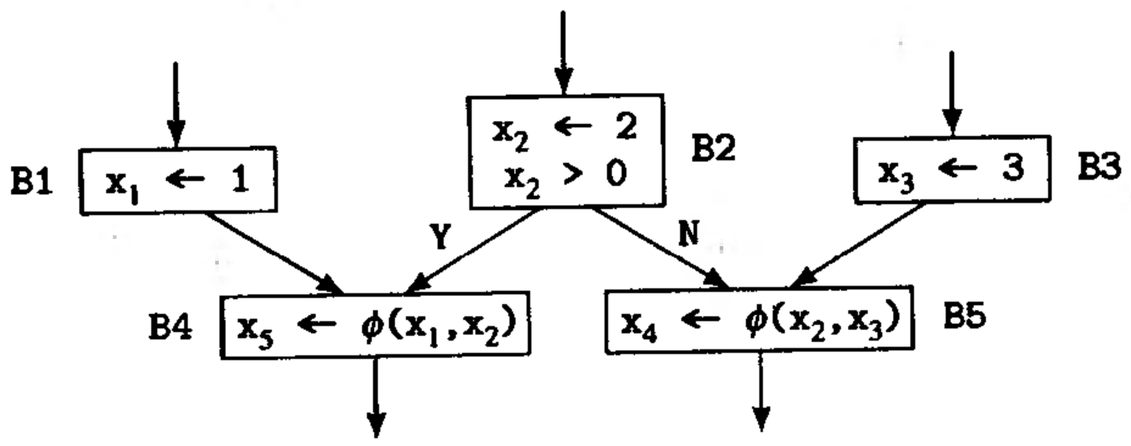


图16-8 每一个SSA形式的变量是一个du链的链头

图16-9中的ICAN例程MIR_to_SymLIR()将MIR形式的过程转换成已用符号寄存器替代了变量的LIR代码。这个例程使用了描述指令操作数的全局类型opdrecd，此类型由一个kind和一个值组成，其中kind可以是var、regno或const，值可以是标识符、寄存器或常数。MIR_to_SymLIR()使用了全局整常数ArgReg和值为寄存器的全局常数RetReg，它们分别包含此调用的第一个参数寄存器的编号和用于保存返回地址的寄存器的名字。代码中还使用了如下三个例程：

492

1. Find_Symreg(*s*, *i*, *j*)返回基本块*i*中指令*j*中的符号*s*所在网的索引（或等价地，符号寄存器）。

2. Convert_Opnd(*opnd*)以MIR操作数*opnd*为参数，返回对应的LIR操作数（一个常数或符号寄存器，其中符号寄存器名字由字母s后接一个大于等于nregs+1的整数组成）。

3. Int_to_Reg(*i*)将整型实参*i*转换为对应的真实寄存器或如前所述的符号寄存器的名字。

⊖ 这不是实质性的，输入给此寄存器分配程序的代码可能已经是LIR代码。

bbs.theithome.com

```
procedure MIR_to_SymLIR(nblocks,ninsts,Block,LBlock)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of MIRInst
   LBlock: out array [1··nblocks] of array [··] of LIRInst
begin
   i, j, k, reg: integer
   inst: MIRInst
   opnd1, opnd2, opnd: opdrecd
   for i := 1 to nblocks do
      j := 1
      while j ≤ ninsts[i] do
         inst := Block[i][j]
         case inst of
binasgn:    opnd1 := Convert_Opnd(inst.opd1)
            opnd2 := Convert_Opnd(inst.opd2)
            LBlock[i][j] := ⟨kind:regbin,left:Find_Symreg(inst.left,i,j),
               opr:inst.opr,opd1:opnd1,opd2:opnd2⟩
unasgn:     opnd := Convert_Opnd(inst.opd)
            LBlock[i][j] := ⟨kind:regun,left:Find_Symreg(inst.left,i,j),
               opr:inst.opr,opd:opnd⟩
valasgn:    opnd := Convert_Opnd(inst.opd)
            LBlock[i][j] := ⟨kind:regval,
               left:Find_Symreg(inst.left,i,j),opd:opnd⟩
            . . .
goto:       LBlock[i][j] := inst
binif:      opnd1 := Convert_Opnd(inst.opd1)
            opnd2 := Convert_Opnd(inst.opd2)
            LBlock[i][j] := ⟨kind:regbinif,opr:inst.opr,
               opd1:opnd1,opd2:opnd2,lbl:inst.lbl⟩
            . . .
call:       reg := ArgReg
            for k := 1 to |inst.args| do
               LBlock[i][j+k-1] := ⟨kind:regval,
                  left:Int_to_Reg(reg),
                  opd:Convert_Opnd(inst.args↓k)⟩
               reg += 1
            od
            LBlock[i][j+k] := ⟨kind:callreg,proc:inst.proc,
               rreg:RetReg⟩
            j += k
            . . .
         esac
         j += 1
      od
   od
end    || MIR_to_SymLIR
```

图16-9 将MIR形式的过程转换成已用符号寄存器替代了变量的LIR代码的ICAN例程

16.3.4 冲突图

一旦计算好了网，接下来的步骤是建立冲突图。每一个机器寄存器和每一个网（＝符号寄存器）在冲突图中都有一个结点。

如果所有的寄存器都是同一类寄存器并且没有特殊的使用约定，即任何量都可以存放在任何寄存器中，则不需要在冲突图中包括这些寄存器的结点。我们只要简单地找出这个图的R色着

bbs.theithome.com

色，并指派不同的颜色到不同的寄存器即可。但是一般不是这种情形，因为至少需要专门为调用约定和栈结构保留若干寄存器。

类似地，目标机也可能有两种以上指定了不同功能的寄存器集合（例如，整数寄存器和浮点寄存器）。对这两种寄存器集合的分配问题可以分别进行处理，并产生较小且限制也较少的冲突图，但是一般并不这样做，因为从一片存储单元移动一块数据至另一片存储单元通常可以使用两种寄存器集合中的任意一种（假设体系结构没有存储器到存储器的传送指令）——因此采用分开的两个图反而会不必要地限制了这种分配处理。

在前面的简述中我们曾指出，如果两个结点曾经是同时活跃的，则它们之间有一条弧。但是这会导致图中弧的数目比我们实际需要的要多得多。其实只要当两个结点中有一个在另一个的定值点是活跃的，这两个结点之间才有一条弧，这就足够了。原来定义中的那些多余的弧可以去掉，从而减少所需要的颜色数目，或换言之，减少使R色着色图可行而引入的代码量。从一个结点引向其他结点的弧的数目叫做此结点的度（degree）。

Chaitin等人[chaA81]给出了一个过程的例子，采用“同时活跃”的定义，这个过程的冲突图需要21种颜色，而采用“在定值点活跃”的定义，其冲突图只需要11种颜色。图16-10对那个例子作了适当修改。在基本块B4的入口，a1，…，an、b1，…，bn和left都是活跃的。其冲突图有$2n+1$个结点。如果我们使用前一种冲突图的定义，它有$n(2n+1)$条弧连接每一个结点到所有其他结点，并且需要$2n+1$种颜色。用后一种定义，ai与bi完全不相干，因此只有$n(n+1)$条弧，并只需要$n+1$种颜色。

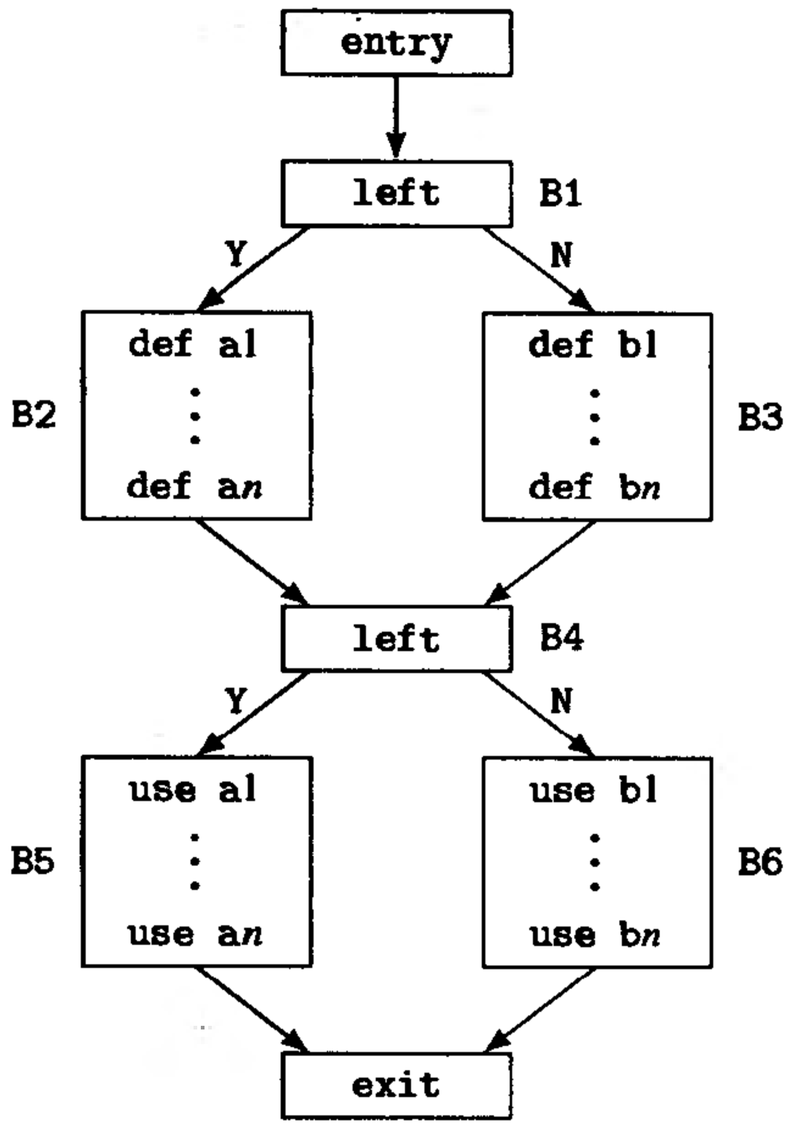


图16-10 冲突的两种定义产生不同冲突图的流图例子

冲突图除了表示同时活跃的变量引起的冲突之外，也足以表示其他类型的冲突。例如，在POWER体系结构中，当通用寄存器r0作为地址计算中的基地址寄存器时，可用来保存其值为0的常数。这一事实可通过使代表基地址寄存器的所有网与r0相冲突来表示。类似地也可使得由语言实现的调用约定所改变的寄存器与跨调用活跃的所有网相冲突。

16.3.5 冲突图的表示

在讲述如何建立冲突图之前，我们先考虑怎样表示它。冲突图可能相当大，所以有效利用空间是我们关心的问题；但实践经验也表明访问时间同样重要，因此仔细地考虑如何有效地表示它能得到很大的回报。如我们将看到的，我们需要能够快速地构造冲突图，判定两个结点是否相连，找出有多少个结点连接到一个给定结点，以及找出连接到一个已知结点的所有结点。为此，我们推荐使用传统的表示，即邻接矩阵和邻接表的组合[⊖]。

邻接矩阵AdjMtx[2..nwebs, 1..nwebs-1]是一个下三角矩阵，其中AdjMtx[max(i, j), min(i, j)]=true，如果第i个寄存器（真实的或符号的）和第j个寄存器相邻；否则为false[⊜]。这种矩阵表示能很快地创建冲突图，也能很快地确定出两个结点是否相邻。例如，

493 ~ 495

⊖ 在B.1节讨论了一种可作为候选的、基于Briggs和Torczon [BriT93]描述的稀疏数据结构的表示。但是，与传统方法的比较实验表明，它的空间浪费较大且相对较慢。

⊜ 记住，真实寄存器的编号是从1到nregs，符号寄存器是从nregs+1到nwebs。

bbs.theithome.com

图16-2冲突图的邻接矩阵如图16-11所示，其中t代表true，f代表false。

建立冲突图邻接矩阵表示的ICAN例程Build_AdjMtx()如图16-12所示，它用函数Live_At(*web*, *symb*, *def*)来判定*web*中是否存在在符号*symb*的定值点*def*活跃的任何定值，用函数Interfere(*s*, *r*)来确定由符号*s*表示的这个网是否与真实寄存器*r*相冲突。

邻接表的表示是一个数组，该数组元素的类型是listrecd，这是一个含6个分量的记录类型。对于数组元素AdjLsts[*i*]，这6个分量如下：

1. color是一个整数，它的值是为结点选择的颜色；初值为－∞。

	r1	r2	r3	s1	s2	s3	s4	s5
r2	t							
r3	t	t						
s1	f	f	f					
s2	f	f	f	t				
s3	f	f	f	t	t			
s4	t	f	f	t	t	f		
s5	f	f	f	f	f	f	t	
s6	f	f	f	f	f	f	f	f

图16-11 图16-2冲突图的邻接矩阵，其中t代表true，f代表false

```
procedure Build_AdjMtx( )
begin
    i, j: integer
    def: UdDu
    for i := 2 to nwebs do
        for j := 1 to i-1 do
            AdjMtx[i,j] := false
        od
    od
    for i := 2 to nregs do
        for j := 1 to i-1 do
            AdjMtx[i,j] := true
        od
    od
    for i := nregs+1 to nwebs do
        for j := 1 to nregs do
            if Interfere(Symreg[i],j) then
                AdjMtx[i,j] := true
            fi
        od
        for j := nregs+1 to i-1 do
            for each def ∈ Symreg[i].defs
                (Live_At(Symreg[j],Symreg[i].symb,def)) do
                AdjMtx[i,j] := true
            od
        od
    od
end    || Build_AdjMtx
```

图16-12 为图着色寄存器分配建立冲突图邻接矩阵表示的ICAN代码

2. disp是一个位移，它是地址的一部分，当需要时，指派到位置*i*的符号寄存器将溢出到此位置，其初值为－∞。

3. spcost是结点溢出的代价；它的初值对于符号寄存器是0.0，对于真实寄存器是∞。

4. nints是adjnds域中的冲突个数。

5. adjnds是当前与真实寄存器或符号寄存器*i*相冲突的那些真实寄存器和符号寄存器的一张表。

6. rmvadj是与真实寄存器或符号寄存器*i*相冲突，并且已在修剪过程中被删除的真实寄存器和符号寄存器的一张表。

对于确定一个特定结点有多少个相邻的结点，以及它们是哪些结点，这种表示特别有用。

bbs.theithome.com

图16-2冲突图的邻接表如图16-13所示。

	color	disp	spcost	nints	adjnds 1	2	3
r1	-∞	-∞	∞	3	r2	r3	s4
r2	-∞	-∞	∞	2	r1	r3	
r3	-∞	-∞	∞	2	r1	r2	
s1	-∞	-∞	0.0	3	s2	s3	s4
s2	-∞	-∞	0.0	3	s1	s3	s4
s3	-∞	-∞	0.0	2	s1	s2	
s4	-∞	-∞	0.0	4	r1	s1	s2	s5
s5	-∞	-∞	0.0	1	s4		
s6	-∞	-∞	0.0	0			

图16-13 图16-2冲突图的初始邻接表

图16-14给出了建立冲突图邻接表表示的ICAN代码。邻接矩阵表示主要用于图着色预处理期间，即寄存器合并（参见下一节）过程中；邻接表主要用于实际的着色处理过程。因此，我们首先建立邻接矩阵，在寄存器合并期间对它进行修改，然后如16.3.2节所讨论的那样，从得到的结果来建立邻接表。

```
procedure Build_AdjLsts( )
begin
    i, j: integer
    for i := 1 to nregs do
        AdjLsts[i].nints := 0
        AdjLsts[i].color := -∞
        AdjLsts[i].disp := -∞
        AdjLsts[i].spcost := ∞
        AdjLsts[i].adjnds := []
        AdjLsts[i].rmvadj := []
    od
    for i := nregs+1 to nwebs do
        AdjLsts[i].nints := 0
        AdjLsts[i].color := -∞
        AdjLsts[i].disp := -∞
        AdjLsts[i].spcost := 0.0
        AdjLsts[i].adjnds := []
        AdjLsts[i].rmvadj := []
    od
    for i := 2 to nwebs do
        for j := 1 to nwebs - 1 do
            if AdjMtx[i,j] then
                AdjLsts[i].adjnds ⊕= [j]
                AdjLsts[j].adjnds ⊕= [i]
                AdjLsts[i].nints += 1
                AdjLsts[j].nints += 1
            fi
        od
    od
end    || Build_AdjLsts
```

图16-14 建立冲突图邻接表表示的ICAN代码

bbs.theithome.com

16.3.6 寄存器合并

建立好邻接矩阵后，我们对它施加一种转换，称为寄存器合并。寄存器合并（register coalescing）或归类（subsumption）是复写传播的一个变体，它删除那些从一个寄存器到另一个寄存器的复制。它搜索中间代码中诸如$sj \leftarrow si$的寄存器复制指令，其中si和sj不相互冲突[⊖]，并且在这条复制赋值指令到例程结束之间没有对si 或sj两者的存储指令。一旦找到这样的一条指令，寄存器合并便搜索那些写si的指令，修改这些指令使它们的结果存放在sj中而不是si中，并删除这条复写指令，同时修改冲突图使si和sj合并为单个结点，并使该结点与这两个结点原来相冲突的所有结点相冲突。对图的修改可以以增量方式进行。注意，在$sj \leftarrow si$的复写操作点，如果有在此点活跃的另一个符号寄存器sk，我们曾使它与si相冲突，现在它已变为不必要的，因此这些冲突都可以删除。

496 ~ 497

图16-15所示的ICAN例程Coalesce_Regs()执行寄存器合并，它使用了下面三个例程:

1. Reg_to_Int(r)将它的符号或真实寄存器参数r转换为整数i，使得Symreg(i)代表r。

2. delete_inst(i, j, ninsts, Block, Succ, Pred)从基本块i删除指令j（参见图4-15）

3. Non_Store(LBlock, k, l, i, j)返回true，若在LBlock[i][j]中的赋值$sk \leftarrow sl$到包含此复写赋值的例程的末尾之间，既不存在对sk的存储，也不存在对sl的存储。

在寄存器合并之后，如图16-14所示我们构造冲突图的邻接表表示。

```
procedure Coalesce_Regs(nblocks,ninsts,LBlock,Succ,Pred)
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   LBlock: inout array [1··nblocks] of array [··] of LIRInst
   Succ, Pred: inout integer ⟶ set of integer
begin
   i, j, k, l, p, q: integer
   for i := 1 to nblocks do
      for j := 1 to ninsts[i] do
         || if this instruction is a coalescable copy, adjust
         || assignments to its source to assign to its target instead
         if LBlock[i][j].kind = regval then
            k := Reg_to_Int(LBlock[i][j].left)
            l := Reg_to_Int(LBlock[i][j].opd.val)
            if !AdjMtx[max(k,l),min(k,l)] ∨ Non_Store(LBlock,k,l,i,j) then
               for p := 1 to nblocks do
                  for q := 1 to ninsts[p] do
                     if LIR_Has_Left(LBlock[p][q])
                        & LBlock[p][q].left = LBlock[i][j].opd.val then
                        LBlock[p][q].left := LBlock[i][j].left
                     fi
                  od
               od
               || remove the copy instruction and combine Symregs
               delete_inst(i,j,ninsts,LBlock,Succ,Pred)
               Symreg[k].defs ∪= Symreg[l].defs
               Symreg[k].uses ∪= Symreg[l].uses
               || adjust adjacency matrix to reflect removal of the copy
```

图16-15 寄存器合并算法

⊖ 注意，冲突图中包含了做这种转换所需要的数据流信息，因此我们可以避免做活跃变量分析。另外还要注意sj和si都可能是真实寄存器或符号寄存器。

bbs.theithome.com

```
                    Symreg[l] := Symreg[nwebs]
                    for p := 1 to nwebs do
                        if AdjMtx[max(p,l),min(p,l)] then
                            AdjMtx[max(p,k),min(p,k)] := true
                        fi
                        AdjMtx[max(p,l),min(p,l)] := AdjMtx[nwebs,p]
                    od
                    nwebs -= 1
                fi
            fi
        od
    od
end     || Coalesce_Regs
```

图16-15 （续）

Chaitin和其他人已注意到这种合并是一种强有力的转换。它可以做的事情有以下一些:

1. 这种合并简化了编译过程的若干步骤，如删除因转换SSA形式回到线性中间代码而引入的不必要的复制代码。

2. 它可用来保证在调用一个过程之前将参数值传送到（或计算于）适当的寄存器中。在被调用端，它能够将传送在寄存器中的参数迁移到适当的工作寄存器。

3. 它能实现那种要求源寄存器和目标寄存器的操作数和结果在适当地方的机器指令。

4. 它能使得要求其结果在一个寄存器中，并且一个操作数也必须在此寄存器的两地址指令（如CISC中存在的那种指令）能按需求而被处理。

5. 它能使我们保证那种要求其操作数或结果使用一对寄存器的指令能指派到这种寄存器偶对。

498
~
500

我们在图16-15的算法中没有考虑这些问题，但在本章结束有相关的练习。

16.3.7 计算溢出代价

分配过程的下一个步骤是在不能把所有符号寄存器直接分配到真实寄存器的情况下，计算将寄存器的内容溢出和重新恢复它的代价。溢出的潜在作用是将一个网分割成两个以上的网，因而可能减少图中的冲突。例如，在图16-5中，通过在基本块B2的末尾引入一条存储y的寄存器的指令，并在B4的开始引入一条从存储它的位置取y的指令，我们可以将包含了基本块B2中y的定值和B4中它的使用的一个网分割为两个网。

如果仔细地考虑溢出的判别条件，就能既使得图可用R种颜色着色，又使得插入的存储和恢复指令条数最少。

在给寄存器赋了值之后，再将其值溢出到存储器，并且在需要使用它们时将它们重新取回，是图着色寄存器分配程序使得冲突图是可R色着色的一个主要手段。溢出具有将一个网分割成两个以上网的作用，因而可能减少图中的冲突，并增加新得到的图是可R色着色的机会。

每一个邻接表元素有一个分量spcost，它估计溢出相应符号寄存器需要的溢出代价，其计算方法与16.2节（译者注: 原书误为16.1节）介绍的基于使用次数的方法类似。

更具体地，溢出一个网w的代价是

$$defwt \cdot \sum_{def \in w} 10^{depth(def)} + usewt \cdot \sum_{use \in w} 10^{depth(use)} - copywt \cdot \sum_{copy \in w} 10^{depth(copy)}$$

其中*def*、*use*和*copy*分别是网w中的各个定值、使用和寄存器复制；*defwt*、*usewt*和*copywt*是给指令类型指定的权重。

计算溢出代价应当考虑下面一些因素:

bbs.theithome.com

1. 如果重新计算一个网的值比重取它更有效，则代价是重新计算的代价，而不是重取它的代价。

2. 如果一条复写指令的源寄存器和目标寄存器都被溢出，则不再需要这条指令。

3. 如果一个已溢出的值在同一基本块中有若干次使用，并且已恢复的值在这个基本块最后一次使用这个已溢出值之前都一直保持活跃，则在此基本块中只需要取这个值一次。

图16-16中的ICAN例程Compute_spill_Costs()计算每一个寄存器的溢出代价，并将结果保存于邻接表中。取、存和复制的权重分别是图16-3中的UseWt、DefWt和CopyWt。如果考虑到取直接数指令和加直接数指令，检查它们的出现并指定其权重为1，这个算法可进一步细化。我们在这个算法中加入了上述条件的前两个条件；第三个条件留给读者作为练习。该算法使用了下面几个函数：

501

1. depth(*i*)返回流图中基本块*i*的循环嵌套深度。

2. Rematerialize(*i*, nblocks, ninsts, LBlock)返回重新计算编号为*i*的符号寄存器，而不是溢出后再重取它的代价。

3. Real(*i*)返回与整数*i*具有相同值的实数。

```
procedure Compute_Spill_Costs(nblocks,ninsts,LBlock)
   nblocks: in integer
   ninsts: in integer ⟶ integer
   LBlock: in integer ⟶ array [1··nblocks] of array [··] of LIRInst
begin
   i, j: integer
   r: real
   inst: LIRInst
   || sum the costs of all definitions and uses for each
   || symbolic register
   for i := 1 to nblocks do
      for j := 1 to ninsts[i] do
         inst := LBlock[i][j]
         case LIR_Exp_Kind(inst.kind) of
binexp:     if inst.opd1.kind = regno then
               AdjLsts[inst.opd1.val].spcost
                  += UseWt * 10.0↑depth(i)
            fi
            if inst.opd2.kind = regno then
               AdjLsts[inst.opd2.val].spcost
                  += UseWt * 10.0↑depth(i)
            fi
unexp:      if inst.opd.kind = regno then
               if inst.kind = regval then
                  AdjLsts[inst.opd.val].spcost
                     -= CopyWt * 10.0↑depth(i)
               else
                  AdjLsts[inst.opd.val].spcost
                     += UseWt * 10.0↑depth(i)
               fi
            fi
noexp:   esac
         if LIR_Has_Left(inst.kind) & inst.kind ≠ regval then
               AdjLsts[inst.left].spcost
                  += DefWt * 10.0↑depth(i)
         fi
      od
```

图16-16 计算符号寄存器溢出代价的ICAN代码

bbs.theithome.com

```
        od
        for i := nregs+1 to nwebs do
           || replace by rematerialization cost if less than
           || spill cost
           r := Rematerialize(i,nblocks,ninsts,LBlock)
           if r < AdjLsts[i].spcost then
              AdjLsts[i].spcost := r
           fi
        od
     end    || Compute_Spill_Costs
```

图16-16 （续）

16.3.8 修剪冲突图

下面我们尝试用R种颜色给图着色，其中R是可用寄存器的个数。我们并不是想找出一种完全彻底的R色着色——人们早就已知对于$R \geqslant 3$，这是一个NP完全问题，并且除此之外，图也可能是不能以R色着色的。替代地，我们使用两种方法来简化图，一种方法保证能使得图的一部分是可R色着色的，只要这个图的剩余部分是可R色着色的。另一种方法则在第一种方法没有穷尽图的情况下继续乐观地进行处理。后一种方法不会直接产生一种R色着色，但与只使用前一种方法相比，它常常能使图的更多部分成为可着色的，因此非常有助于它的启发式值。

第一种方法基于一种简单但非常有效的观察，我们称之为度<R规则：给定一个包含一个度数小于R的结点的图，此图是可R色着色的，当且仅当没有这个度数小于R的结点时此图是可R色着色的。显然，整个图的R色性隐含了不含所选择的结点而得到的图的R色性。换一个角度说，假设我们有此图在没有这个结点情况下的一种R色着色，因为那个结点的度小于R，故至少有一种颜色没有被与它相邻的结点使用，因而这个颜色可以指派给那个结点。当然这一规则并不能使任意图都是可R色着色的。事实上，图16-17给出的例子中，a是一个可2色着色的，但用度<R规则却不能判别它是否可用2色着色；图16-17b是可3色着色的，用这个规则同样不能判别。但是这个规则在实际中对于解决冲突图的着色问题非常有效。对于一台具有32个寄存器（或它的两倍，在计算浮点寄存器的情况下）的机器，它足以使得所遇到的许多图都是可着色的。

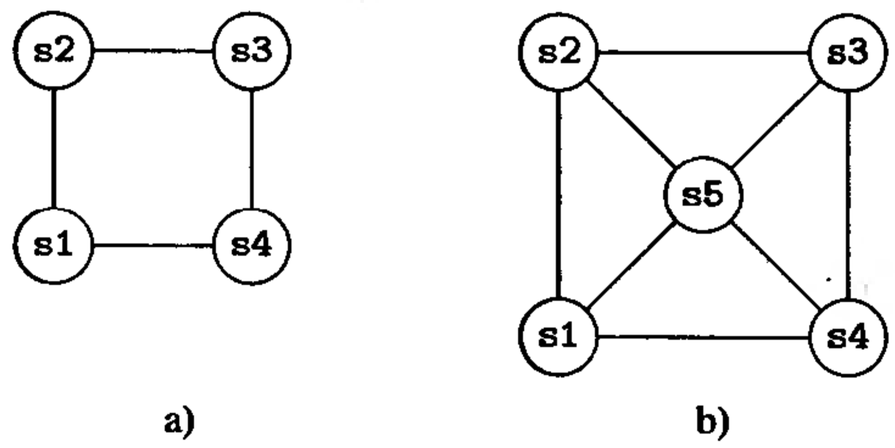


图16-17 a) 可2色着色，b) 可3色着色，但不适用度<R规则的图的例子

502 ~ 503

第二种方法即乐观启发式方法，通过删除度$\geqslant R$的结点来推广度<R规则。这种方法的理由是因为观察到一个结点虽有R个以上的相邻结点，但它们不必都有不同的颜色，因而它们可能不会使用到所有R种颜色。因此，如果第一种方法没有穷尽此图，则我们接着继续选择要着色的候选者，并乐观地希望，当它们需要一种新颜色时能有可用的颜色。

因此，我们从重复地寻找图中那些相邻接点数小于R的结点开始，每当找到一个这样的结点，便将它从图中删除，并放到栈中，以便按找到它们的逆序对它们着色。作为这个处理过程的一部分，我们记住与每一个被删除结点相邻的那些结点（作为rmvadj域的值），以便在寄存器指派时能够使用它们（参见下一节）。如果这一过程穷尽了此图，就已确定出R色着色是可能的。于是，我们从栈中弹出这些结点，并给它们每一个指派一种未指定给它相邻结点的颜色。例如，已知图16-2中的冲突图，从此图我们能够按如下顺序删除结点并将它们放置到栈中（栈

bbs.theithome.com

底位于右边):

r3	r2	r1	s4	s2	s1	s3	s5	s6

然后我们可弹出这些结点，并如前所述给它们指派如下颜色:

结点	颜色
r3	1
r2	2
r1	3
s4	1
s2	2
s1	3
s3	1
s5	3
s6	2

如前面指出的，度<*R*规则有时并不能完全适用。在那种情况下，我们应用乐观启发式，即，按图的当前结点度数计算，选择一个度≥*R*且具有最小溢出代价的结点，并乐观地将它压入栈中。我们这样做是期望将来它的邻接结点不会用完所有的颜色，因此将应在修剪冲突图时做出的溢出决定推迟到实际给结点指派颜色时，即由图16-20的Assign_Regs()例程来决定。

在继续讨论关于修剪图的代码之前，我们提请读者注意在修剪判断过程中保持代码和冲突图相互同步的困难性。因为溢出代码会将一个网分为若干个网（或用图的术语，它将一个结点
504 分成若干个结点），因此代码和冲突图的这种同步维护代价是很昂贵的。我们处理这个问题的方法是避免在修剪时更改代码。如果寄存器指派失败，则在下一轮迭代中建立邻接矩阵和邻接表将会更快一些，因为已插入了这些溢出代码。

图16-18给出了应用度<*R*规则和乐观启发式方法尝试对冲突图着色的ICAN例程Prune_Graph()。它使用图16-19的例程Adjust_Neighbors()来反映从图中删除一个结点。全局变量Stack用于将删除结点的顺序传递给Assign_Regs()例程，该例程将在下一小节讨论。

```
procedure Prune_Graph( )
begin
    success: boolean
    i, nodes := nwebs, spillnode: integer
    spillcost: real
    Stack := []
    repeat
        || apply degree < R rule and push nodes onto stack
        repeat
            success := true
            for i := 1 to nwebs do
                if AdjLsts[i].nints > 0
                    & AdjLsts[i].nints < nregs then
                    success := false
                    Stack ⊕= [i]
                    Adjust_Neighbors(i)
                    nodes -= 1
                fi
            od
        until success
        if nodes ≠ 0 then
            || find node with minimal spill cost divided by its degree and
            || push it onto the stack (the optimistic heuristic)
```

图16-18 尝试用*R*色对冲突图着色的代码

bbs.theithome.com

```
            spillcost := ∞
            for i := 1 to nwebs do
               if AdjLsts[i].nints > 0
                  & AdjLsts[i].spcost/AdjLsts[i].nints < spillcost then
                  spillnode := i
                  spillcost := AdjLsts[i].spcost/AdjLsts[i].nints
               fi
            od
            Stack ⊕= [spillnode]
            Adjust_Neighbors(spillnode)
            nodes -= 1
         fi
      until nodes = 0
end      || Prune_Graph
```

图16-18 （续）

```
procedure Adjust_Neighbors(i)
    i: in integer
begin
    j, k: integer
    || move neighbors of node i from adjnds to rmvadj and
    || disconnect node i from its neighbors
    for k := 1 to |AdjLsts[i].adjnds| do
        AdjLsts[k].nints -= 1
        j := 1
        while j ≤ |AdjLsts[k].adjnds| do
            if AdjLsts[k].adjnds↓j = i then
               AdjLsts[k].adjnds ⊖= j
               AdjLsts[k].rmvadj ⊕= [i]
            fi
            j += 1
        od
    od
    AdjLsts[i].nints := 0
    AdjLsts[i].rmvadj ⊕= AdjLsts[i].adjnds
    AdjLsts[i].adjnds := []
end     || Adjust_Neighbors
```

图16-19 修剪冲突图使用的例程Adjust_Neighbors()

505

16.3.9 指派寄存器

用*R*种颜色对冲突图着色的ICAN例程Assign_Regs()如图16-20所示。它用到例程Min_Color(*r*)，此例程返回那些与*r*相邻的结点还未使用的颜色中编号最小的颜色编号；当所有颜色都已用于相邻结点时返回0。然后它将这个值赋给函数Real_Reg(*s*)，这个函数返回已指派给符号寄存器*s*的真实寄存器。

当Assign_Regs()成功时，接着调用图16-21所示的Modify_Code()，以便用对应的真实寄存器替代这些符号寄存器。Modify_Code()用Color_to_Reg()将指派给一个符号寄存器的颜色转换为对应的真实寄存器名。Color_to_Reg()用Real_Reg()来确定给一种颜色指派的是哪一个真实寄存器。

bbs.theithome.com

```
procedure Assign_Regs( ) returns boolean
begin
    c, i, r: integer
    success := true: boolean
    repeat
        || pop nodes from the stack and assign each one
        || a color, if possible
        r := Stack↓-1
        Stack e= -1
        c := Min_Color(r)
        if c > 0 then
            if r ≤ nregs then
                Real_Reg(c) := r
            fi
            AdjLsts[r].color := c
        else
            || if no color is available for node r,
            || mark it for spilling
            AdjLsts[r].spill := true
            success := false
        fi
    until Stack = []
    return success
end    || Assign_Regs
```

图16-20 给真实寄存器和符号寄存器指派颜色的例程

```
procedure Modify_Code(nblocks,ninsts,LBlock)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   LBlock: inout array [1··nblocks] of array [··] of LIRInst
begin
   i, j, k, m: integer
   inst: LIRInst
   || replace each use of a symbolic register by the real
   || register with the sams color
   for i := 1 to nblocks do
      for j := 1 to ninsts[i] do
         inst := LBlock[i][j]
         case LIR_Exp_Kind(inst.kind) of
binexp:     if inst.opd1.kind = regno
               & Reg_to_Int(inst.opd1.val) > nregs then
               LBlock[i][j].opd1.val :=
                  Color_to_Reg(AdjLsts[inst.opd1.val].color)
            fi
            if inst.opd2.kind = regno
               & Reg_to_Int(inst.opd2.val) > nregs then
               LBlock[i][j].opd2.val :=
                  Color_to_Reg(AdjLsts[inst.opd2.val].color)
            fi
unexp:      if inst.opd.kind = regno
               & Reg_to_Int(inst.opd.val) > nregs then
               LBlock[i][j].opd.val :=
                  Color_to_Reg(AdjLsts[inst.opd.val].color)
            fi
listexp:    for k := 1 to |inst.args| do
               if Reg_to_Int(inst.args↓k@1.regno) > nregs then
```

图16-21 修改过程中的指令以用真实寄存器替代符号寄存器的ICAN例程

bbs.theithome.com

```
                    m := AdjLsts[inst.args↓k@1.val].color
                    LBlock[i][j].args↓i@1.val :=
                       Color_to_Reg(m)
                 fi
              od
noexp:     esac
           if LIR_Has_Left(inst.kind) then
              if Reg_to_Int(inst.left) > nregs then
                 LBlock[i][j].left :=
                    Color_to_Reg(AdjLsts[inst.left].color)
              fi
           fi
        od
     od
  end     || Modify_Code
```

图16-21 （续）

16.3.10 溢出符号寄存器

一个冲突图着色所需要的颜色数目常常叫做它的*寄存器压力*（register pressure），因此为了使得图是可着色的而修改代码称为是“减少寄存器的压力”。

一般地，溢出符号寄存器的作用是将一个网分割为两个以上的网，并将与原网的冲突分布到新的多个网中。例如，如果如图16-22所示，引入对tmp赋值和读取的存取指令来分割图16-5中的网w1，则它将被下表中所示的4个新网w5，...，w8所替代：

网	成　员
w2	B5中x的定值，B6中x的使用
w3	B2中y的定值，B4中y的使用
w4	B1中y的定值，B3中y的使用
w5	B2中x的定值，B2中的tmp←x
w6	B3中x的定值，B3中的tmp←x
w7	B4中的x←tmp，B4中x的使用
w8	B5中的x←tmp，B5中x的使用

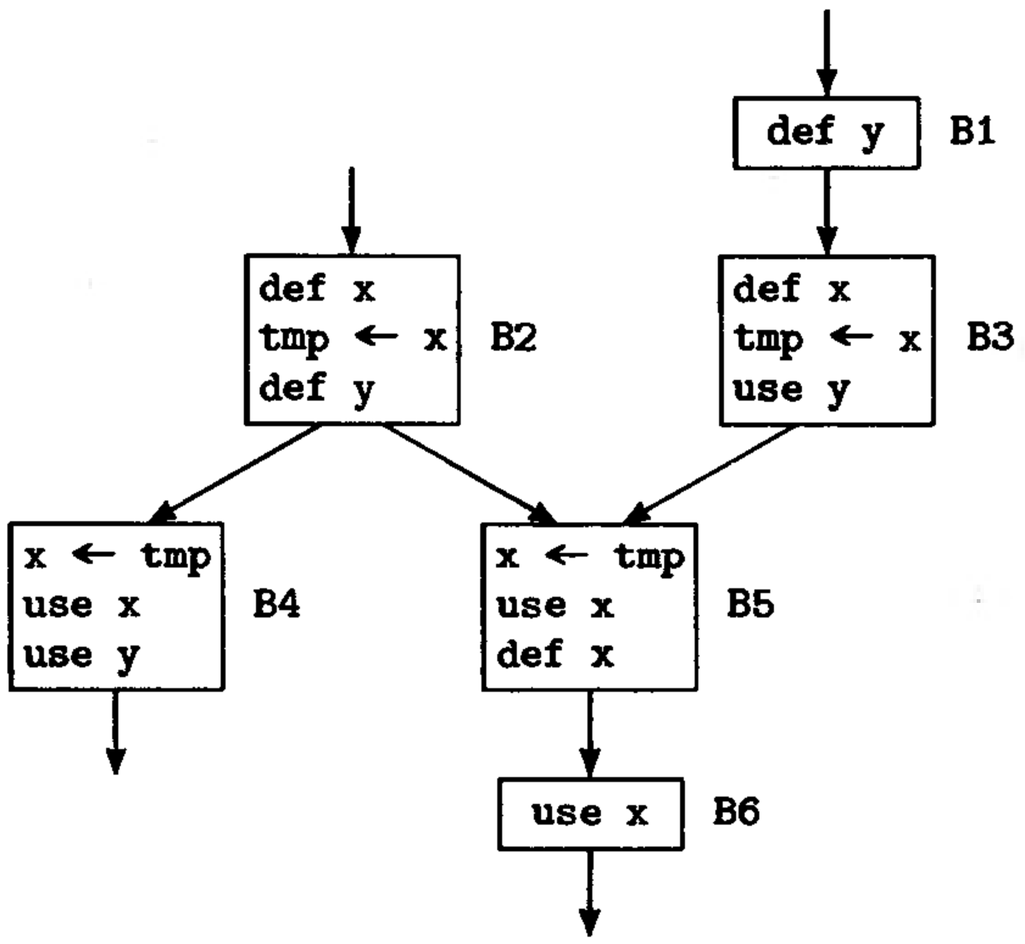


图16-22 图16-5中例子在溢出网w1后

bbs.theithome.com

相应的冲突图如图16-23所示。

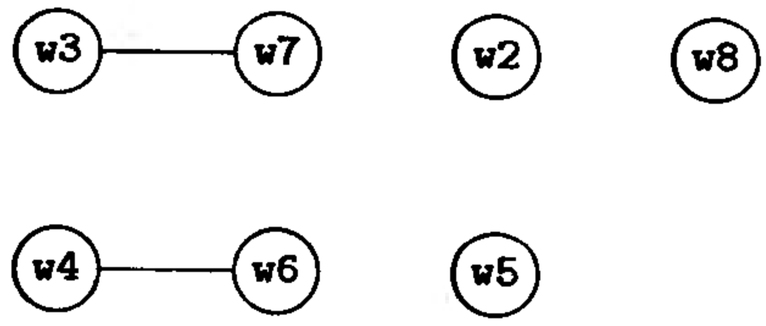


图16-23 图16-22的网之间的冲突

当已知不能使得冲突图用R色着色时，我们接下来便分割那些已标识要溢出的结点，即对于它有AdjLsts[i].spill=true的结点i。

图16-24给出了Gen_Spill_Code()的代码。此例程使用的子程序Comp_Disp(r)的代码也在图16-24中给出。这个子程序确定符号寄存器r是否已指派有一个溢出位移，如果没有，则增加Disp，并存储此位移于AdjLsts[i].disp，其中i是r在邻接表中的索引。变量BaseReg存放在溢出和恢复过程中用作基寄存器的寄存器名。Gen_Spill_Code另外还使用了如下两个函数：

506 ~ 508

1. insert_before(i, j, ninsts, LBlock, *inst*)将指令*inst*直接插在指令LBlock[i][j]之前（参见图4-14）。

2. insert_after(i, j, ninsts, LBlock, *inst*)将指令*inst*紧接着插在指令LBlock[i][j]之后（参见图4-14）。

注意，Gen_Spill_Code()没有考虑到取或加直接数的情况，也没有考虑其他的恢复方式，并且它只处理字大小的操作数。本章末尾的练习将处理其中的一些问题。

```
procedure Gen_Spill_Code(nblocks,ninsts,LBlock)
   nblocks: in integer
   ninsts: inout array [1··nblocks] of integer
   LBlock: inout array [1··nblocks] of array [··] of LIRInst
begin
   i, j, regct := 0: integer
   inst: LIRInst
   || check each definition and use of a symbolic register
   || to determine whether it is marked to be spilled, and,
   || if so, compute the displacement for it and insert
   || instructions to load it before each use and store
   || it after each definition
   for i := 1 to nblocks do
      j := 1
      while j ≤ ninsts[i] do
         inst := LBlock[i][j]
         case LIR_Exp_Kind(inst.kind) of
binexp:     if AdjLsts[inst.opd1.val].spill then
               Comp_Disp(inst.opd1.val)
               insert_before(i,j,ninsts,LBlock,⟨kind:loadmem,
                  left:inst.opd1.val,addr:⟨kind:addrrc,
                  reg:BaseReg,disp:Disp⟩⟩)
               j += 1
            fi
            if inst.opd2 ≠ inst.opd1
               & AdjLsts[inst.opd2.val].spill then
               Comp_Disp(inst.opd2.val)
               insert_before(i,j,ninsts,LBlock,⟨kind:loadmem,
                  left:inst.opd2.val,addr:⟨kind:addrrc,
                  reg:BaseReg,disp:Disp⟩⟩)
               j += 1
            fi
unexp:      if AdjLsts[inst.opd.val].spill then
```

图16-24 利用图16-16中的Compute_Spill_Costs()计算的代价生成溢出代码的ICAN代码

bbs.theithome.com

```
            Comp_Disp(inst.opd.val)
            insert_before(i,j,ninsts,LBlock,⟨kind:loadmem,
               left:inst.opd.val,addr:⟨kind:addrrc,
               reg:BaseReg,disp:Disp⟩⟩)
            j += 1
         fi

listexp:    for k := 1 to |inst.args| do
               if AdjLsts[inst.args↓k@1.val].spill then
                  Comp_Disp(inst.args↓k@1.val)
                  insert_before(i,j,ninsts,LBlock,
                     ⟨kind:loadmem,left:inst.opd.val,
                     addr:⟨kind:addrrc,
                     reg:BaseReg,disp:Disp⟩⟩)
                  regct += 1
               fi
            od
            j += regct - 1
noexp:   esac
         if LIR_Has_Left(inst.kind)
            & AdjLsts[inst.left].spill then
            Comp_Disp(inst.left)
            insert_after(i,j,ninsts,LBlock,⟨kind:stormem,
               addr:⟨kind:addrrc,
               reg:BaseReg,disp:Disp⟩⟩,
               opd:⟨kind:regno,val:inst.left⟩⟩)
            j += 1
         fi
      od
   od
end    || Gen_Spill_Code

procedure Comp_Disp(r)
   r: in Register
begin
   || if symbolic register r has no displacement yet,
   || assign one and increment Disp
   || Note:  this assumes each operand is no larger
   || than a word
   if AdjLsts[Reg_to_Int(r)].color = -∞ then
      AdjLsts[Reg_to_Int(r)].disp := Disp
      Disp += 4
   fi
end    || Comp_Disp
```

图16-24 （续）

还要注意的是，如果我们必须为循环中使用或定值的寄存器插入溢出代码，则可能的话，应当在进入循环之前恢复它们，并在从循环出口之后溢出它们。这需要用到13.3节介绍的边分割。具体地，在图16-26的例子中，如果我们不得不溢出s2，则需要在B1和B2之间引入一个新基本块B1a，并在其中恢复s2，在B2和B4之间引入B2a，并在B2a中溢出它。

16.3.11 图着色寄存器分配的两个例子

作为图着色寄存器分配的第一个例子，考虑图16-25中的流图，其中假设c是一个非局部变

bbs.theithome.com

量，同时我们有5个可用于分配的寄存器r1、r2、r3、r4和r5（因此$R=5$），并且只有g可以放在r5中。另外假设基本块B1、B3和B4的执行频率是1，而B2的是7。因为每个符号寄存器有一个网，故我们使用符号寄存器的名字表示网，如图16-26所示。

509 ~ 511

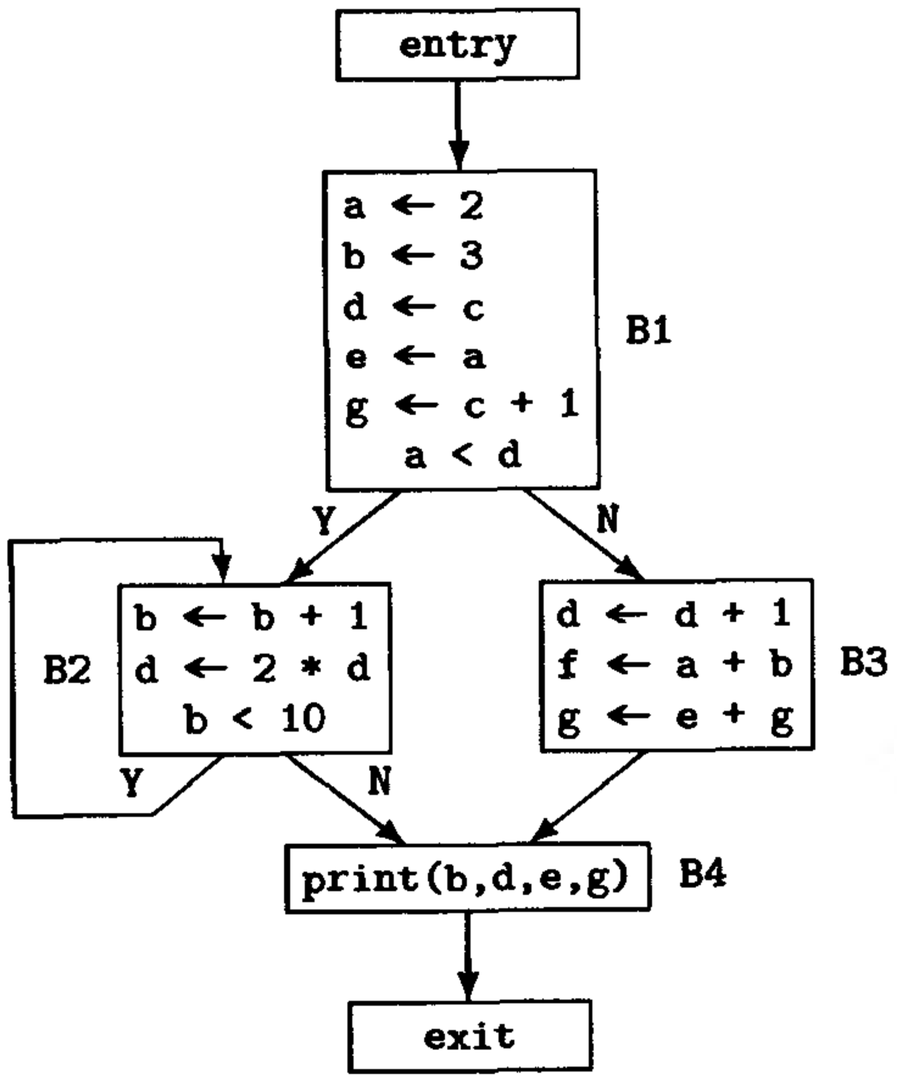


图16-25 用于图着色寄存器分配的一个小例子

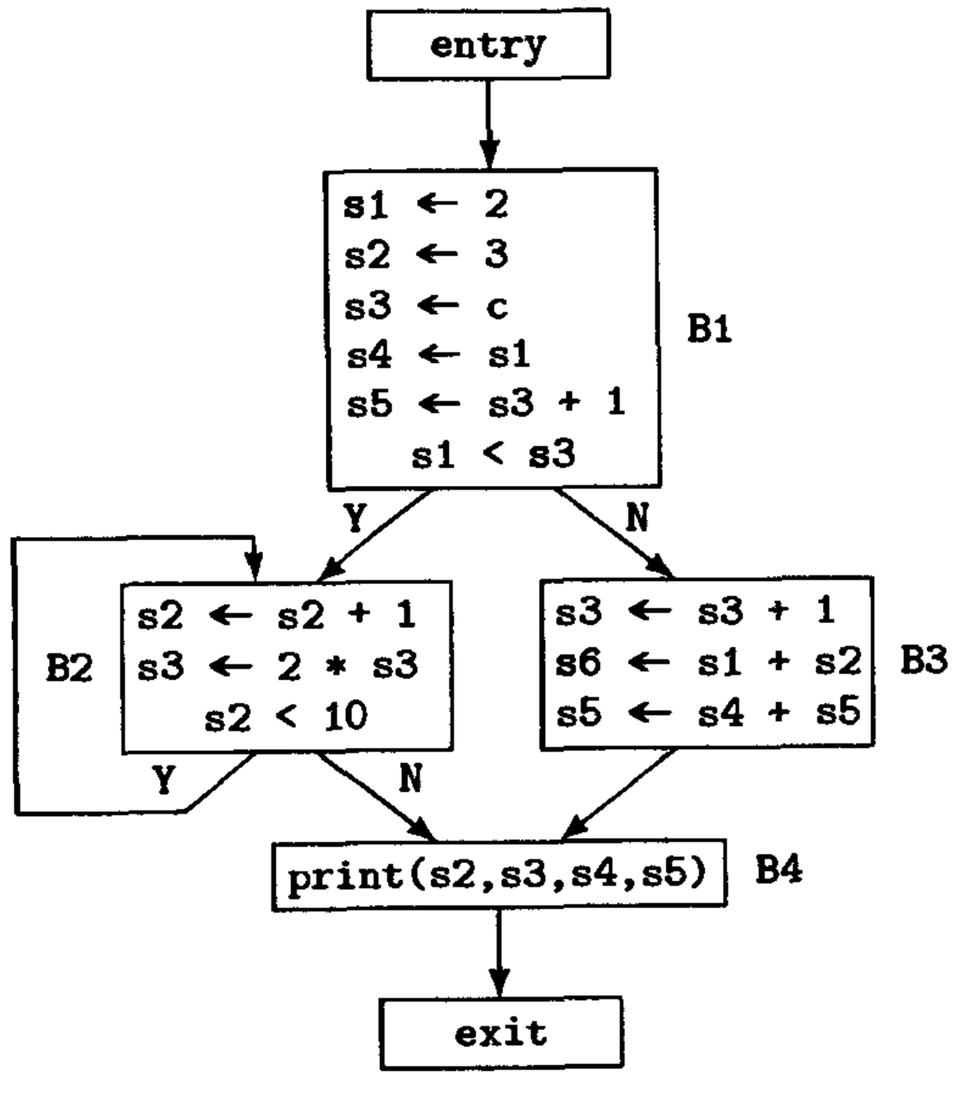
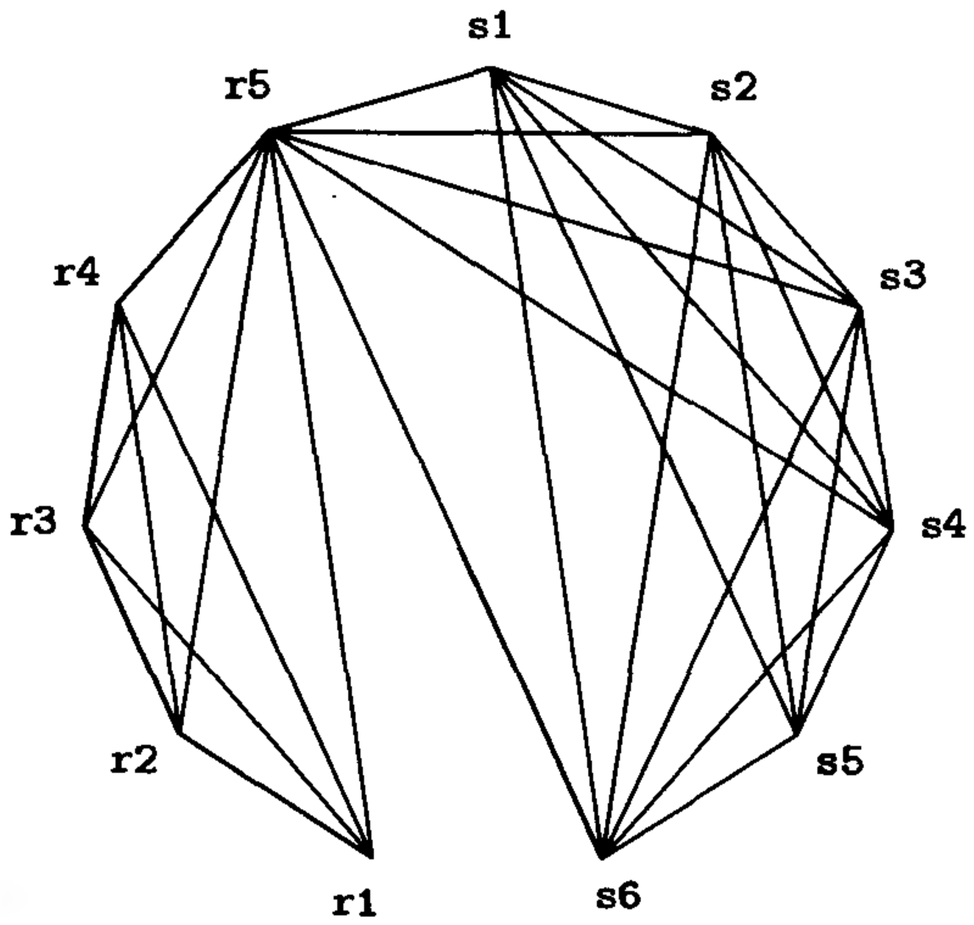


图16-26 图16-25的例子用符号寄存器替代了局部变量

然后我们为图16-26中这段代码建立邻接矩阵，如图16-27b所示，图16-27a是它的冲突图的图形表示。



a)

	r1	r2	r3	r4	r5	s1	s2	s3	s4	s5
r2	t									
r3	t	t								
r4	t	t	t							
r5	t	t	t	t						
s1	f	f	f	f	t					
s2	f	f	f	f	t	t				
s3	f	f	f	f	t	t	t			
s4	f	f	f	f	t	t	t	t		
s5	f	f	f	f	f	t	t	t	t	
s6	f	f	f	f	t	t	t	t	t	t

b)

图16-27 图16-26中例子的a) 冲突图和b) 邻接矩阵，其中t和f分别代表true和false

对图16-26中基本块B1内的复写赋值s4←s1施加寄存器合并，得到了如图16-28所示的流图，以及图16-29中新的冲突图和邻接矩阵。现在已没有进一步合并的机会，所以我们建立这个例程的邻接表，如图16-30所示。

bbs.theithome.com

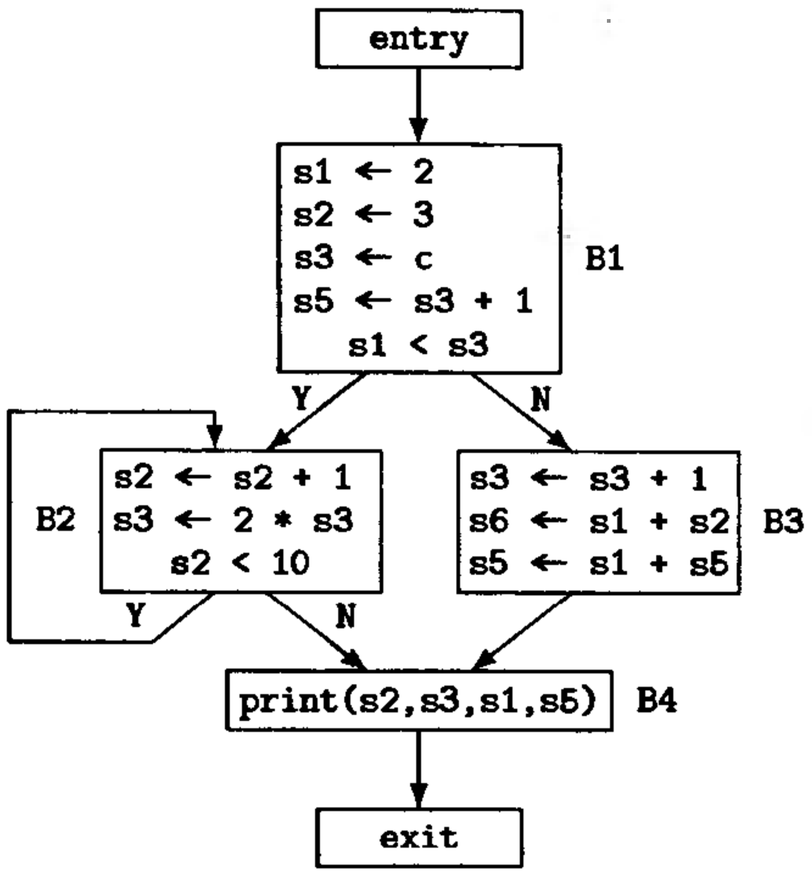
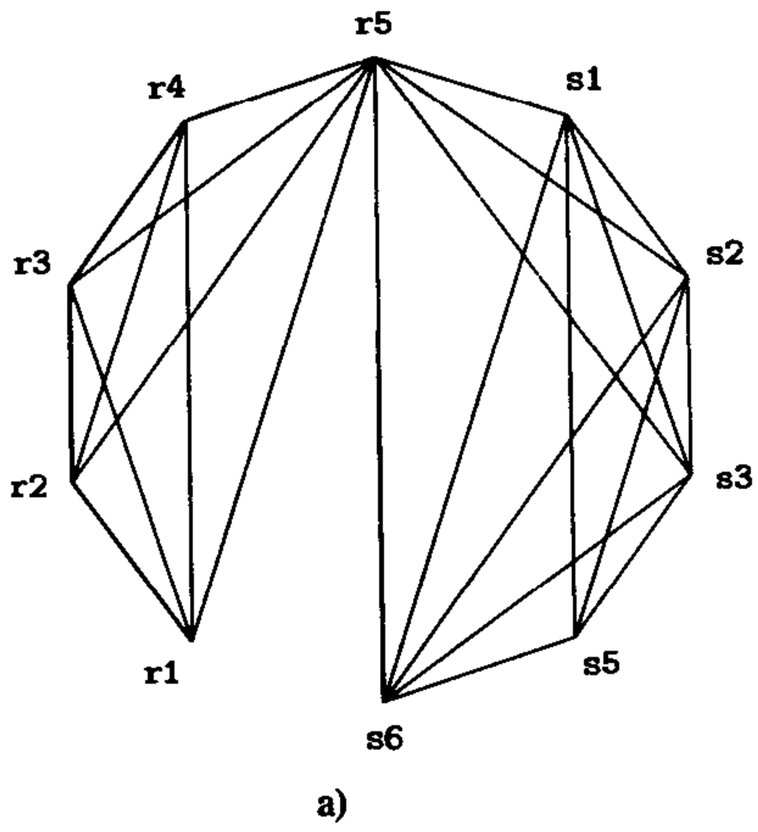


图16-28 图16-26的例子在合并寄存器s4和s1之后



	r1	r2	r3	r4	r5	s1	s2	s3	s5
r2	t								
r3	t	t							
r4	t	t	t						
r5	t	t	t	t					
s1	f	f	f	f	t				
s2	f	f	f	f	t	t			
s3	f	f	f	f	t	t	t		
s5	f	f	f	f	f	t	t	t	
s6	f	f	f	f	t	t	t	t	t

b)

图16-29 图16-28中的例子在合并符号寄存器s1和s4之后的a) 冲突图和b) 邻接矩阵，其中t和f分别代表true和false

	color	disp	spcost	nints	adjnds 1	2	3	4	5	6	7	8
r1	-∞	-∞	∞	4	r2	r3	r4	r5				
r2	-∞	-∞	∞	4	r1	r3	r4	r5				
r3	-∞	-∞	∞	4	r1	r2	r4	r5				
r4	-∞	-∞	∞	4	r1	r2	r3	r5				
r5	-∞	-∞	∞	8	r1	r2	r3	r4	s1	s2	s3	s6
s1	-∞	-∞	0.0	5	r5	s2	s3	s5	s6			
s2	-∞	-∞	0.0	5	r5	s1	s3	s5	s6			
s3	-∞	-∞	0.0	5	r5	s1	s2	s5	s6			
s5	-∞	-∞	0.0	4	s1	s2	s3	s6				
s6	-∞	-∞	0.0	5	r5	s1	s2	s3	s5			

图16-30 图16-28中代码的邻接矩阵

bbs.theithome.com

下面我们使用DefWt＝UseWt=2和CopyWt=1计算溢出代价如下:

符号寄存器	溢出代价
s1	2.0
s2	1.0 + 21.0 + 2.0 + 2.0 = 26.0
s3	6.0 + 14.0 + 4.0 + 2.0 = 26.0
s5	2.0 + 4.0 + 2.0 = 8.0
s6	∞

注意，s1的溢出代价是2.0，因为赋给它的是一个直接数，并且可通过放置一条取直接数的指令位于B3中第二条指令使用它之前，使它重新回到寄存器中。另外，我们使s6的溢出代价是无穷大，因为这个符号寄存器已经死去。

之后我们着手修剪图16-29。因为r1到r4每一个寄存器的邻接结点数都小于5，因此从图中删除它们，并将它们压入到栈中，导致栈如下所示:

r4	r3	r2	r1

由此得到的冲突图如图16-31所示，而对应的邻接表如图16-32所示。注意，结点r5的邻接结点数小于5，因此删除它并压入栈中，导致栈变为

r5	r4	r3	r2	r1

512 ~ 515

并且得到如图16-33所示的冲突图和如图16-34所示的邻接表。

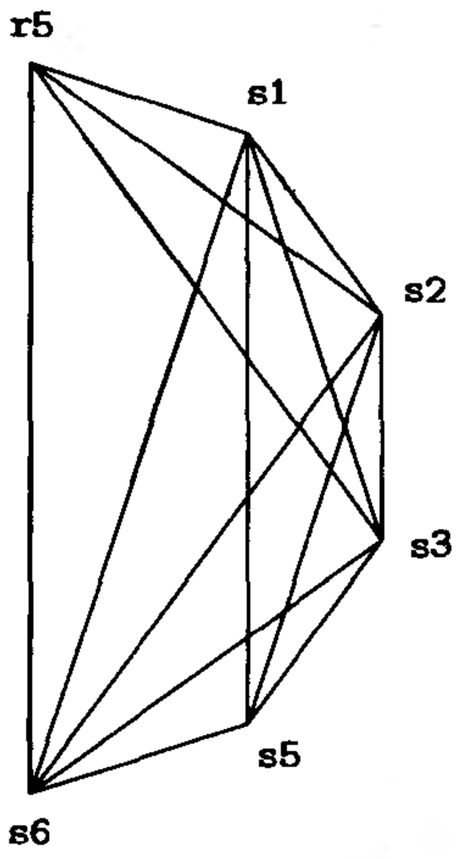


图16-31 将r1至r4压入栈后得到的冲突图

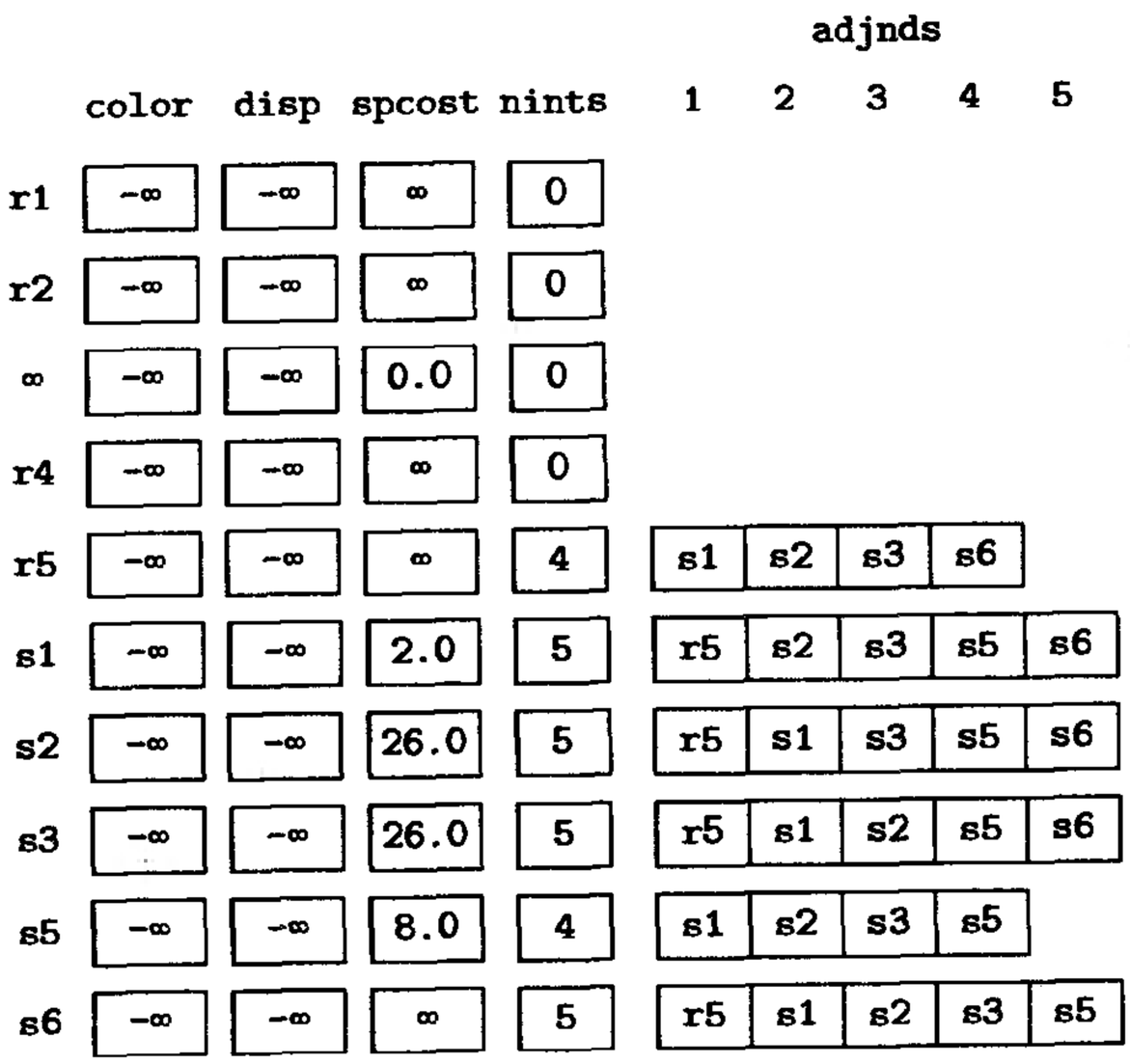


图16-32 与图16-31中冲突图对应的邻接表

现在已经不存在有5个以上邻接结点的结点，所以我们将剩余的符号寄存器按任意顺序压入栈中，结果如下:

s1	s2	s3	s5	s6	r5	r4	r3	r2	r1

bbs.theithome.com

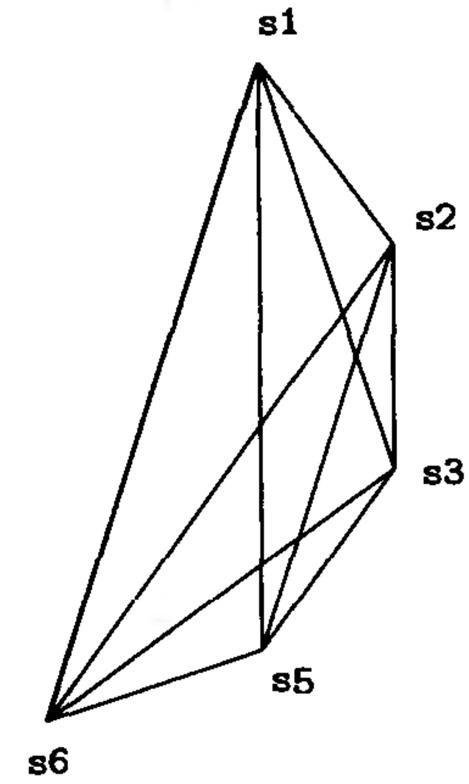


图16-33 将r5压入栈后得到的冲突图

	color	disp	spcost	nints	adjnds 1	2	3	4
r1	-∞	-∞	∞	0				
r2	-∞	-∞	∞	0				
r3	-∞	-∞	∞	0				
r4	-∞	-∞	∞	0				
r5	-∞	-∞	∞	0				
s1	-∞	-∞	2.0	4	s2	s3	s5	s6
s2	-∞	-∞	26.0	4	s1	s3	s5	s6
s3	-∞	-∞	26.0	4	s1	s2	s5	s6
s5	-∞	-∞	8.0	4	s1	s2	s3	s5
s6	-∞	-∞	∞	4	s1	s2	s3	s5

图16-34 与图16-33中冲突图对应的邻接表

然后在将它们弹出栈时对它们着色（即指派真实寄存器给符号寄存器），结果如下：

寄存器	颜色
s1	1
s2	2
s3	3
s5	4
s6	5
r5	4
r4	1
r3	2
r2	3
r1	5

由此我们得到了一种无需溢出任何寄存器到存储器的寄存器分配。图16-35给出了用真实寄存器替换符号寄存器后的流图。

bbs.theithome.com

第二个例子将需要溢出一个寄存器。我们从图16-36的代码开始，代码中已经使用了符号寄存器。假设真实寄存器r2、r3和r4可用于分配。这个例子的冲突图和邻接矩阵如图16-37所示。

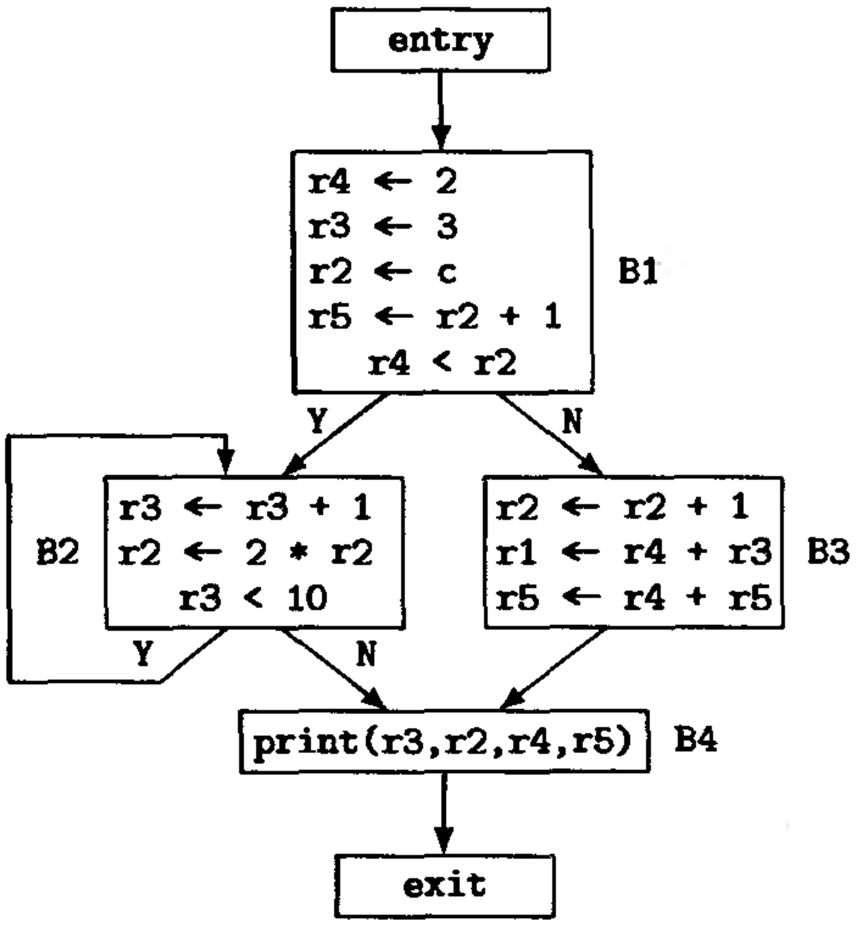


图16-35 对图16-28中的流图用真实寄存器替代符号寄存器

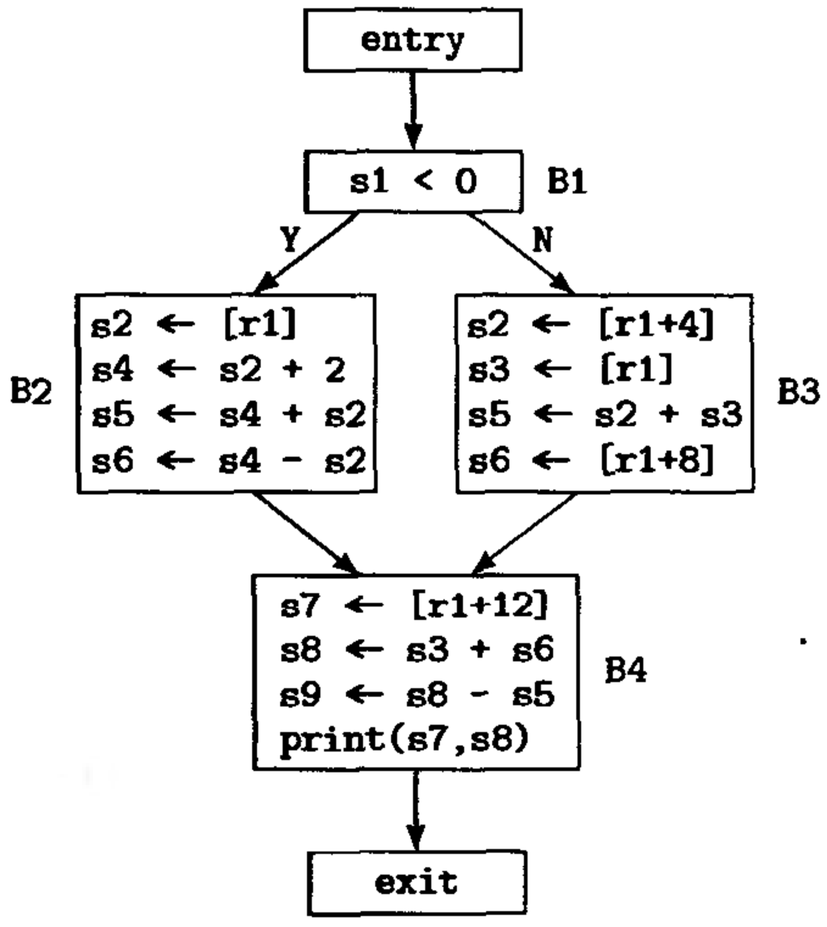


图16-36 图着色寄存器分配的另一个例子

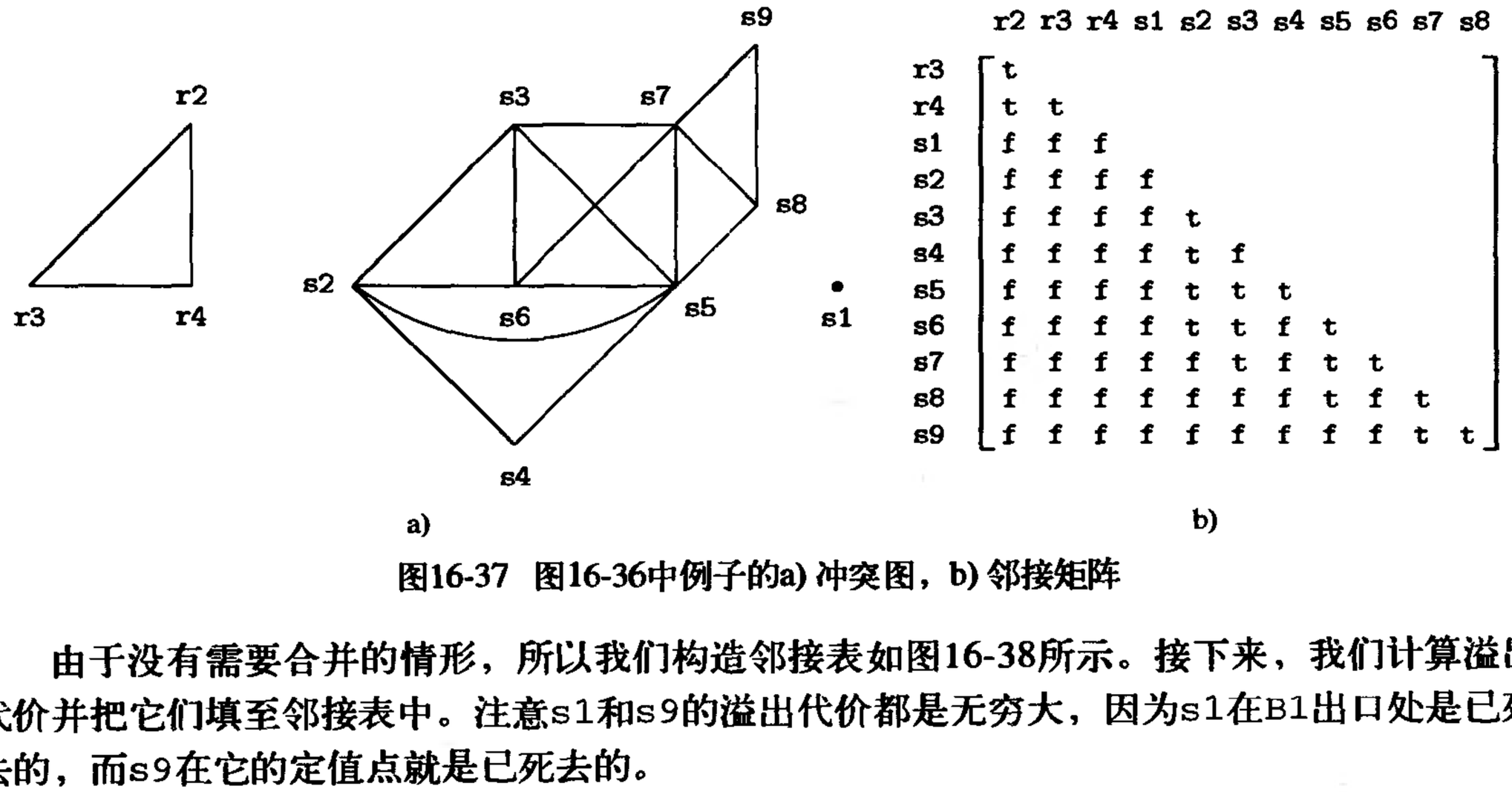


图16-37 图16-36中例子的a) 冲突图，b) 邻接矩阵

由于没有需要合并的情形，所以我们构造邻接表如图16-38所示。接下来，我们计算溢出代价并把它们填至邻接表中。注意s1和s9的溢出代价都是无穷大，因为s1在B1出口处是已死去的，而s9在它的定值点就是已死去的。

516～518

下面我们修剪这个图，删除结点s1（没有邻接结点）并将它压入栈中。接着删除真实寄存器s4和s9（每个都有两个相邻结点），并将它们压入栈中，结果如下:

s9	s4	r4	r3	r2	s1

并且得到如图16-39a所示的冲突图。删除结点s8并将它压入栈中，得到如下栈和如图16-39b所示的冲突图。

s8	s9	s4	r4	r3	r2	s1

bbs.theithome.com

超星公司制作 请尊重作者版权

	color	disp	spcost	nints	adjnds 1	2	3	4	5
r2	-∞	-∞	∞	2	r3	r4			
r3	-∞	-∞	∞	2	r2	r4			
r4	-∞	-∞	∞	2	r2	r3			
s1	-∞	-∞	∞	0					
s2	-∞	-∞	12.0	3	s3	s4	s6		
s3	-∞	-∞	6.0	4	s2	s5	s6	s7	
s4	-∞	-∞	6.0	2	s2	s5			
s5	-∞	-∞	6.0	5	s3	s4	s6	s7	s8
s6	-∞	-∞	6.0	4	s2	s3	s5	s7	
s7	-∞	-∞	4.0	5	s3	s5	s6	s8	s9
s8	-∞	-∞	6.0	3	s5	s7	s9		
s9	-∞	-∞	∞	2	s7	s8			

图16-38 图16-36中代码的邻接表

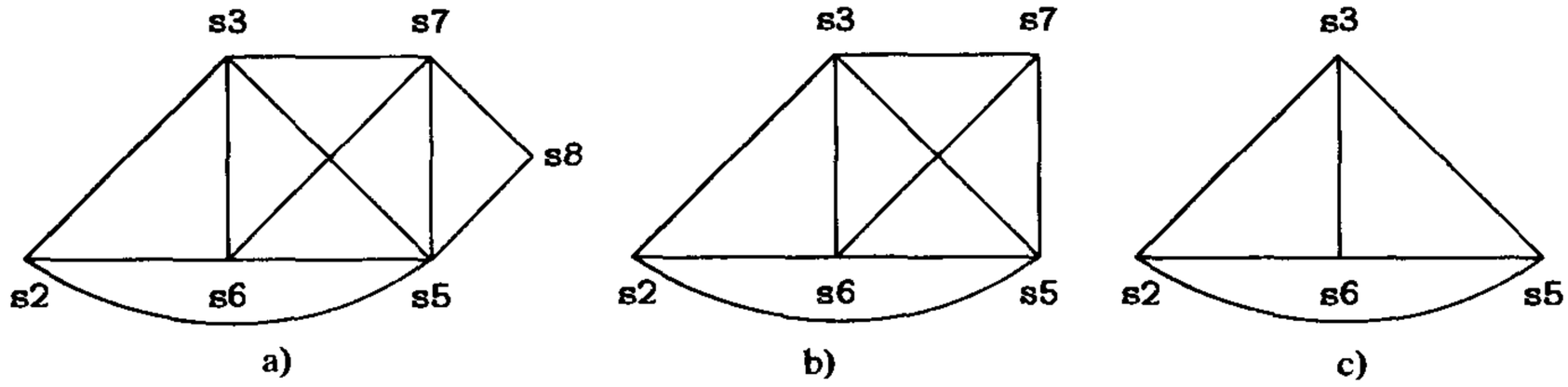


图16-39 a) 删除结点s1、r2、r3、r4、s4和s9并将它们压入栈之后的冲突图，b) 然后删除结点s8并将它压入栈之后的冲突图，c) 删除结点s7并将它压入栈之后的冲突图

在剩下的这个图中，每一个结点的度都大于或等于3，因此选择一个按当前度划分具有最小溢出代价的结点，即s7，将它压入栈中。从图形上看，这将冲突图归约为图16-39c所示形式。于是我们再选择一个按当前度划分具有最小溢出代价的结点，即s5，并将它压入栈中，由此得到的冲突图如图16-40a所示。现在，所有结点的度都小于3，因此我们将它们全部压入栈中，结果得到的栈为:

s2	s3	s6	s5	s7	s8	s9	s4	r4	r3	r2	s1

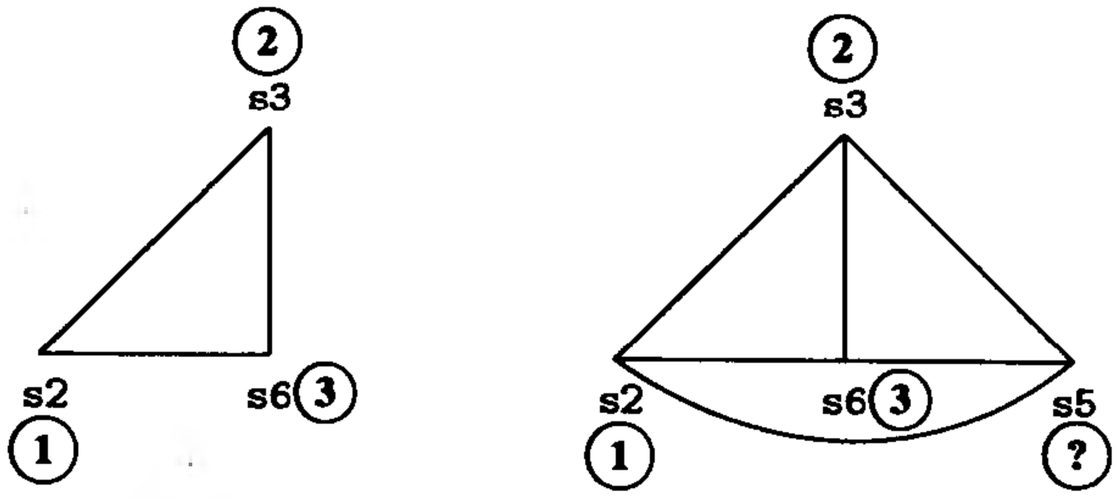
现在我们开始从栈中弹出结点，给它们指派颜色，并从AdjLsts[].rmvadj域重新构造冲突图的邻接表形式。在弹出顶上4个结点之后，我们得到如图16-40b所示的冲突图（其中带圈的黑体字指出颜色），并且已没有颜色可用于结点s5。

因此我们着手用BaseReg=r10、Disp=0和Disp=4分别为基本块B4中的s7和s5生成溢出代码，如图16-41所示。接着建立这个新流图的冲突图，如图16-42所示。从它可清楚地看出，我们可以简单地修剪此图，并给符号寄存器指派与它颜色相同的真实寄存器，结果如图16-43所示。

519 ~ 520

bbs.theithome.com

超星公司制作 请尊重作者版权



a) b)

图16-40 a) 弹出栈顶3个结点之后，b) 弹出4个结点之后的冲突图

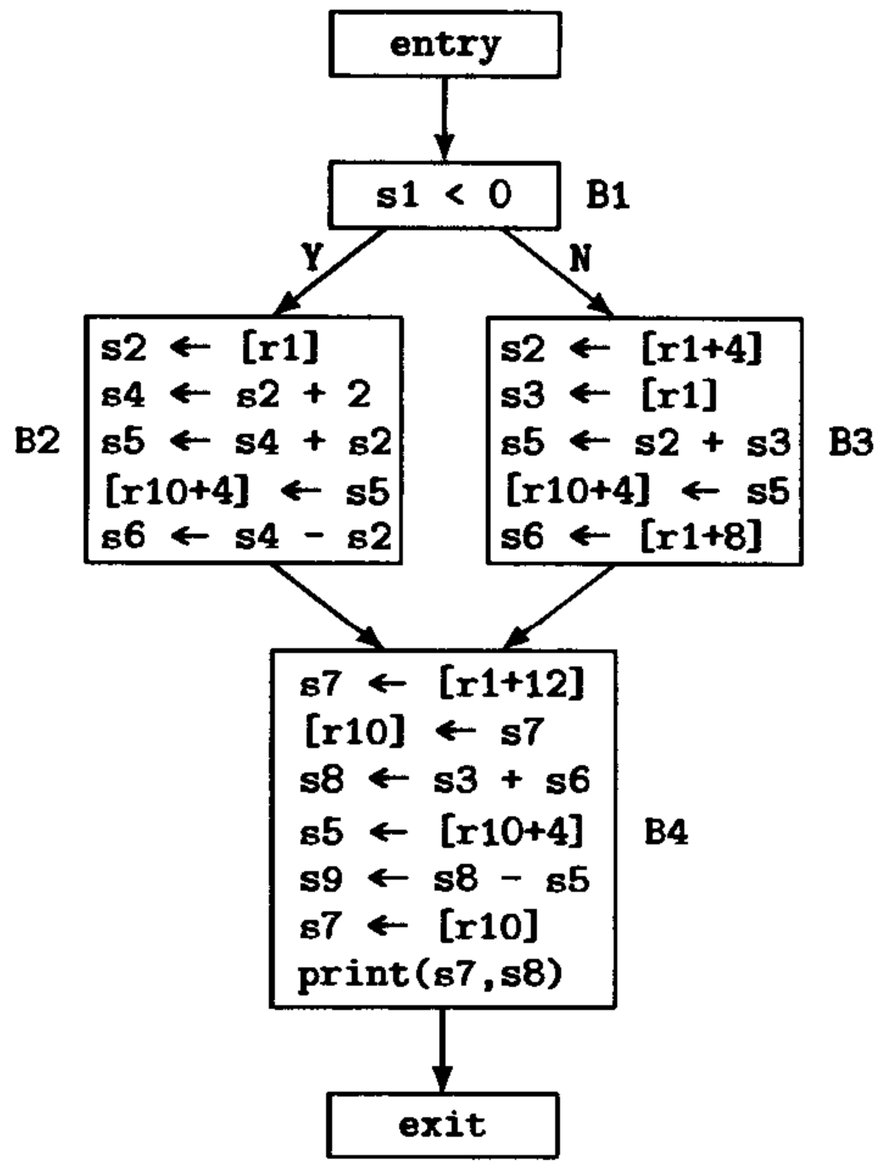


图16-41 图16-36增加了s5和s7的溢出代码之后的流图

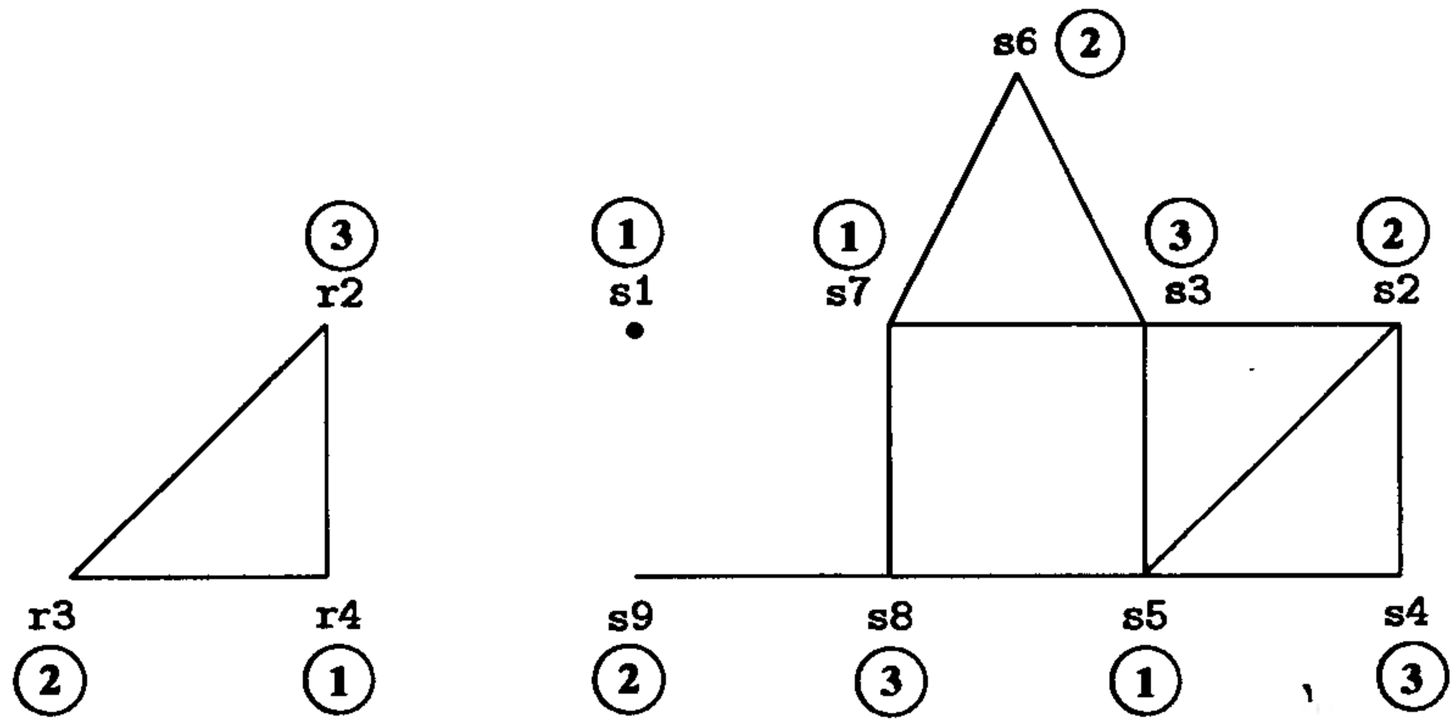


图16-42 图16-41中代码的冲突图

bbs.theithome.com

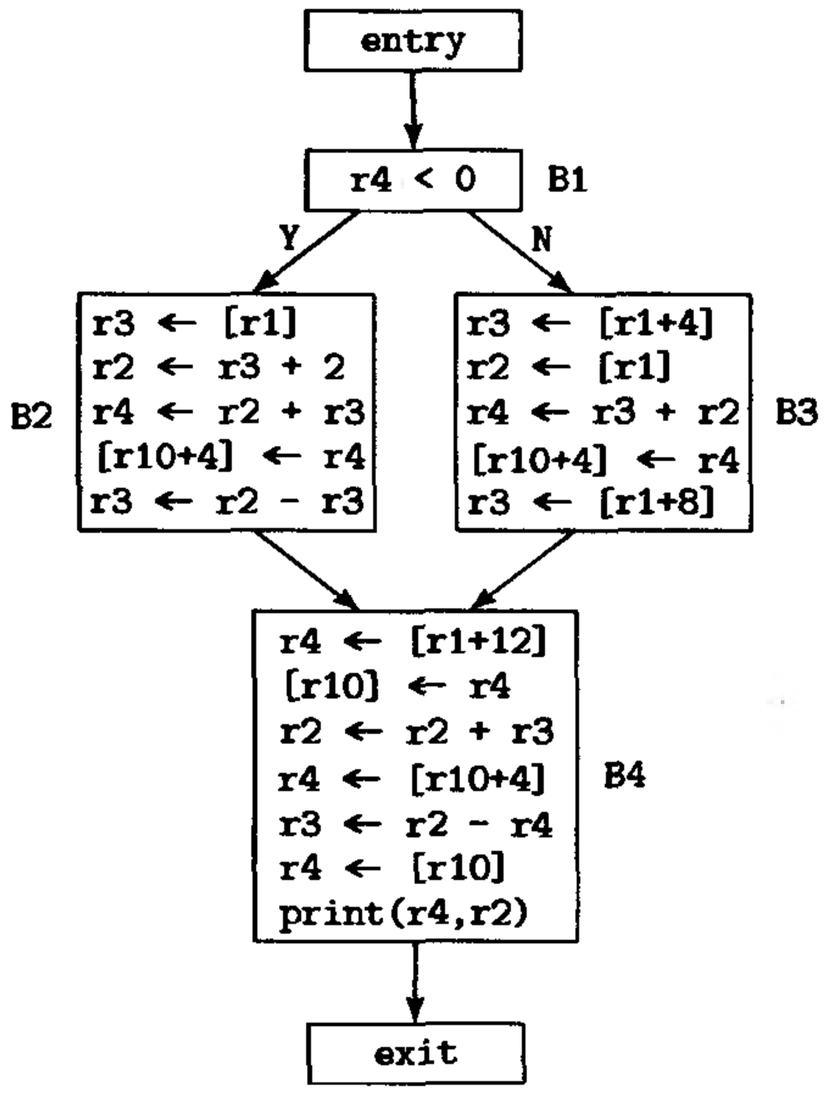


图16-43 对图16-41的流图用真实寄存器替换符号寄存器

16.3.12 其他问题

Bernstein等人[BerG89]讨论了用于选择要溢出的值的三种启发式以及一个分配器，这个分配器尝试这三种启发式并从中选择一个最好的使用。第一种启发式是

$$h_1(w) = \frac{cost(w)}{degree(w)^2}$$

它基于这样一种观察：溢出度数高的网能减少其他更多结点的度数，因而也更可能使得溢出之后度数小于R的网的个数较多。第二和第三种启发式使用了一种称做$area()$的度量，它的定义如下：

521 ~ 522

$$area(w) = \sum_{I \in inst(w)} (width(I) \cdot 5^{depth(I)})$$

其中$inst(w)$是网w中指令的集合，$width(I)$是在指令I活跃的网的个数，$depth(I)$是I的循环嵌套层次。这两种启发式都企图将每一个网对寄存器压力的全局作用考虑进来，它们的定义如下：

$$h_2(w) = \frac{cost(w)}{area(w) \cdot degree(w)}$$

$$h_3(w) = \frac{cost(w)}{area(w) \cdot degree(w)^2}$$

作者报告，在试用了这三种启发式方法以及其他一些修改的15个程序中，第一种启发式方法有4个程序最好，第二种有6个，第三种有8个（第二和第三种有一个程序基本相当），并且在每种情形下，它们中的最好情形都要好于前面介绍的方法。这种分配方法目前已用于IBM的POWER和PowerPC编译器中（参见21.2节）。

Briggs [Brig92]对这种分配算法提出了如下扩充建议：

1. 一种比Nickerson用于Intel 386体系结构的方法更为成熟的处理寄存器偶对的方法，这种方法源于将溢出决定推迟到已指派寄存器之后（参见练习16.4）；

2. 一种改善的重回寄存器（rematerialization）方法，重回寄存器是在寄存器中重新生成诸

bbs.theithome.com

如常数之类的值的过程，重新计算这种值比溢出它们然后再取回要更加有效；

3. 一种在着色之前积极地溢出网的方法，这种方法在着色时考虑的是过程的流图结构，而不是冲突图；

Briggs的重回寄存器的方法包括将一个网分割为构成它的各个值，执行一种数据流计算，并构造新的网。它执行的这种数据流计算将每一个可能需要重回寄存器的值与使该值重回寄存器的指令相连，它构造的每一个新网由相连了相同指令的所有值组成。数据流计算使用的格是一种如图16-44所示的扁平格。

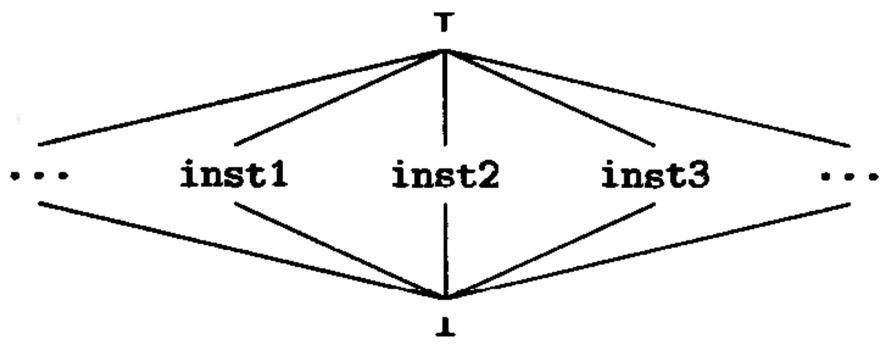


图16-44 重回寄存器格

注意，在RISC体系结构中，对于大常数的情形，一个值重回寄存器可能需要两条指令，一条取直接数的高部，另一条将它与一个已取值的寄存器相加，这种处理偶尔也会比理想情况更进一步地溢出一个网，因此需用一修复遍来寻找这种网，并将它们重新连接起来。

16.4 基于优先级的图着色

基于优先级的图着色寄存器分配在总体结构上与前一节介绍的方法类似，但它在几个重要的细节上有所不同，有一些是本质的，有一些是伴随的。这种方法由Chow和Hennessy首创([ChoH84]和[ChoH90])，其目的是想使得各种分配决策比前面的方法对代价和获益更敏感。

基于优先级方法的一个显著的不同是，它在寄存器分配之前先分配所有对象到它们的存储器单元，然后尝试将它们迁移到寄存器；而不是先分配它们到符号寄存器，然后尝试给符号寄存器指派真实寄存器，并且当需要时生成溢出代码。尽管在符号寄存器对应于未指定具体地址的存储单元这一点上，这两种方法看起来似乎是等价的，但它们实际上是不相同的。图着色方法是乐观的——它开始于这种假定：所有符号寄存器都可能分配到一个真实寄存器，并且这样做时它可能成功。但是，基于优先级的着色方法是悲观的：它可能不能给所有存储单元分配寄存器，因此，对于没有存储器到存储器操作的体系结构（即RISC），它需要保留每一种寄存器中的4个寄存器，以便用于计算含未能成功分配到寄存器的变量的表达式。因此，它从不利条件开始，即可分配的寄存器较少。

另一个不同是基于优先级的方法被设计成机器无关的形式，因此它将若干机器特有的量参数化了，这些参量可专门针对一个给定的实现而具体化。这点不同不是主要的——它与其他机器专用方法没有太大的区别。这些参量是16.2节定义中的一些，即*ldcost*, *stcost*, *usesave*及*defsave*。

第三个较为显著的不同是所使用的网和冲突的概念。Chow和Hennessy表示一个变量的网为该变量在其中是活跃的基本块的集合，并称之为活跃范围(live range)。如图16-45的例子所示，这种定义是稳妥的，但它大大不如图着色方法精确，在图着色方法中，变量x、y、z和w之中没有一个在另一个变量的定值点是活跃的，因此它们之间不存在冲突。但使用Chow和Hennessy的方法，x与y冲突，y既与z也与w冲突，z和w冲突。产生的冲突图如图16-46所示[⊖]。

```
    x ← a + b          B1
    y ← x + c
    if y = 0 goto L1
    z ← y + d          B2
    w ← z
L1: . . .              B3
```

图16-45 Chow和Hennessy的活跃范围的精度不如我们的网的代码例子

⊖ 基于优先级图着色寄存器分配的最早介绍中还进一步包括了与一般图着色方法的一个显著的不同，即分配过程分为局部遍和全局遍，局部遍做基本块内的分配和跨一小族基本块的分配。

bbs.theithome.com

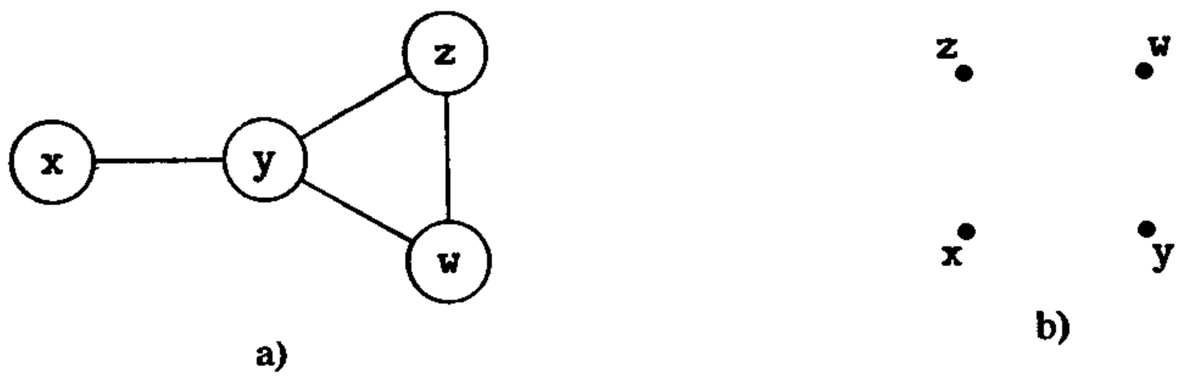


图16-46 图16-45中代码的a) 基于优先级的图着色冲突图，b) 图着色冲突图

Larus和Hilfinger的SPUR LISP编译器[LarH86]的寄存器分配器使用了基于优先级着色的一个版本，它与Chow和Hennessy的方法有下述两点不同:

1. 它一开始便给临时变量指派寄存器，并当需要时生成它们的溢出代码。

2. 它处理的是SPUR汇编语言而不是中级中间代码，因此必须为要溢出的临时变量加入取数和存数指令。

Briggs [Brig92]研究了两种寄存器分配器的运行时间，一种是他自己的，另一种是基于优先级的分配器，他发现对于他的测试程序，他的这个分配器需要的时间大约是$O(n \log n)$，而基于优先级的分配器大约需要$O(n^2)$，其中n是程序中的指令条数。

16.5 其他寄存器分配方法

关于其他几种图着色全局寄存器分配方法也有一些介绍和评价，其中包括两种利用过程的控制树（参见7.6节）来指导溢出或图的修剪判定的方法，一种是由Callahan和Koblenz [CalK91]开发的，另一种由Knobe和Zadeck [KnoZ92]开发。

由Gupta、Soffa和Steele [GupS89]开发的另外一种方法利用最大集团分离子团来实现图着色。一个集团（clique）是一个图，它的每一个结点与图中其他所有结点均有一条弧相连。一个分离子团（clique separator）是一个子图，它本身是一个集团，且从图中删除它将导致包含它的图被分割成两个以上的不相连子图。一个分离子团是最大的，如果图中不存在将其加到这个分离子团可产生一个更大集团的结点。至多具有R个结点的最大分离子团有两个有吸引力的性质：它们可将一个程序划分成若干可独立进行寄存器分配的部分，并且可通过考察代码来构造它们，而无需实际构造完整的冲突图。 523 ~ 525

在20.3节，我们将讨论一种对数组元素进行寄存器分配的方法。数组元素对性能有显著的影响，尤其是对于反复迭代的数值计算。在19.6节，我们将讨论运行时和编译时的过程间寄存器分配方法。

如果寄存器分配以这种方式指派寄存器，即当一个寄存器一旦自由就立刻重新使用它，而不管事实上是否需要这样紧迫，则这种寄存器分配会不必要地降低有效的指令级并行性。这种情况可以通过硬件或软件寄存器重命名（见17.4.5节），并且，部分地，通过循环地指派寄存器而不是只要它们一自由便重新使用它们而得到减轻。另一种方法是将寄存器分配和指令调度合为一遍。有些研究者一直在研究将这两遍合为一遍并达到比两者效果都要好的方法。Bradlee、Eggers和Henry[BraE91]以及Pinter [Pint93]在这方向的研究有所进展。

16.6 小结

这一章的内容涵盖了寄存器分配和指派，对于所有程序而言，它们都属于最重要的优化范畴。我们已了解到寄存器分配应当在低级代码级别来进行，或中间形式或汇编语言，因为所有的取、存及地址计算都必须明显地表示出来。

bbs.theithome.com

超星公司制作 请尊重作者版权

我们的讨论从一种古老然而快速且相当有效的局部方法开始，这种方法根据使用次数和循环嵌套层次来决定哪些对象应当放在寄存器中。然后详细地介绍了一种更有效的方法，即图着色全局寄存器分配，并简要地介绍了另一种也使用图着色，但效果一般要差一些的方法。我们还间接地提及了一种利用装包的方法，两种相对较新的利用过程控制树来指导分配和溢出判定的方法，以及另一种利用最大集团分离子团的新方法。

我们已知道图着色寄存器分配通常能得到非常有效的分配，并且对编译速度不会有太大的开销。图着色寄存器分配在图中用结点代表可分配的对象（符号寄存器）和真实寄存器，用弧代表它们之间的冲突。之后它尝试用其数目与可用寄存器个数相等的颜色来对结点着色，使得每一个结点被指派一种与它的邻居不同的颜色。如果不能做到这一点，则引入代码将符号寄存器溢出到存储器，并在需要时再将溢出的值重新取回到寄存器中。重复这个过程直到着色的目的达到为止。

从这一章我们有以下一些收获:

1. 存在着相当有效的局部寄存器分配方法，这些方法所需的编译时间非常少，并且适合于非优化的编译器。

2. 存在着一种非常有效的图着色全局寄存器分配方法，它比局部分配方法的代价要大，且
526 适合于优化编译器。

3. 其他方法的研究仍在继续，这些方法可能产生更有效的分配器——可以不需要图着色那么大的开销——这种分配器可以将寄存器分配与指令调度结合在一起而不会有负面影响。

图16-47中的黑体字指出了图着色寄存器分配在激进优化编译器中的位置。

第19章讨论了有关寄存器分配的更深内容，包括过程间寄存器分配的方法。这些方法中有一些在汇编语言级别以下的代码上工作，即针对可重定位模块，这种可重定位模块注有数据使用样式的信息。此外，为了改善寄存器分配，在其他优化中还设计了数组引用的标量替代（20.3节），以及聚合量的标量替代（12.2节）。

16.7 进一步阅读

[Frei74]介绍了Freiburghouse的局部寄存器分配。[LowM69] 介绍了IBM 360和370系统上的Fortran H编译器。介绍BLISS语言的是[WulR71]，[WulJ75]中介绍了关于此语言 的PDP-11编译器。

如Kennedy在[Kenn71]中所报告的，Cocke注意到了可将全局寄存器分配看成图着色问题，Chaitin用于IBM 370 PL/I实验编译器中的原始图着色分配器的有关介绍见[ChaA81]，[Chai82]介绍了此分配器用于PL.8编译器的修改版本，其中包含了若干精巧的处理。证明一般图着色问题是NP－完全的内容可在[GarJ79]中找到。在[BriC89]中可找到Briggs、Cooper、Kennedy和Torczon等人关于乐观启发式的最初讨论，[Brig92]中也讨论了它们。这些讨论构成了当前考虑启发式时的出发点。Bernstein等人关于着色启发式的探讨是在[BerG89]中找到的。Nickerson的处理寄存器偶对和更大的寄存器成组的方法见[Nick90]。

Chow和Hennessy发明了基于优先级的图着色方法（见[ChoH84]和[ChoH90]），最初他们认为可以将分配处理过程分为局部遍和全局遍，后来发现这是不需要的。Larus和Hilfinger的寄存器分配器[LarH86]使用了基于优先级的图着色方法的一种变体。

Briggs对他自己的分配器与Chow的基于优先级的分配器所作的比较是在[Brig92]中找到的。

[CalK91]中分别介绍了Callahan和Koblenz，以及Knobe和Zadeck的利用过程控制树来指导溢出判别的方法。Gupta、Soffa和steele等人利用最大分离子团来执行寄存器分配的介绍见[GupS89]。

bbs.theithome.com

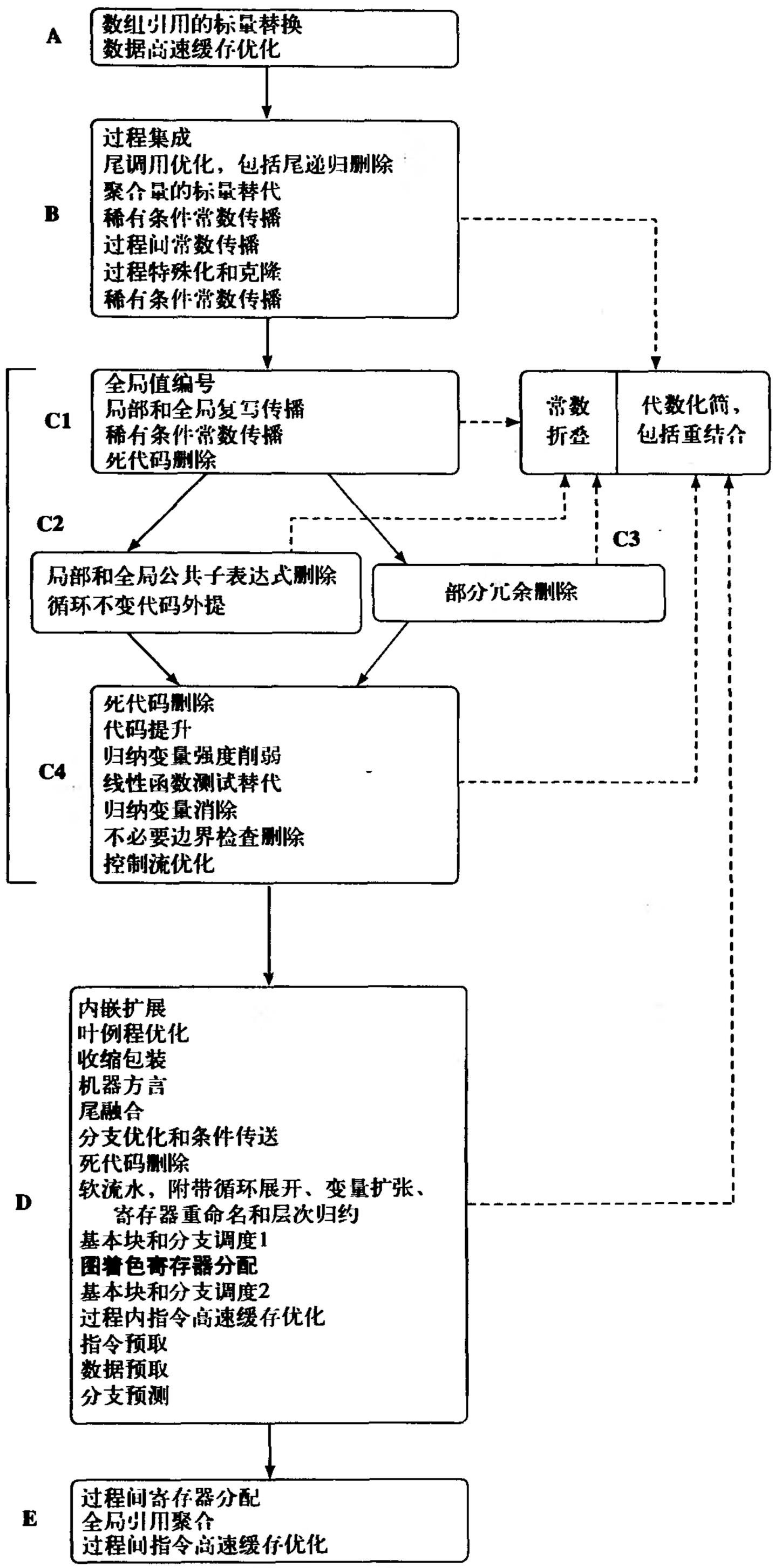


图16-47 寄存器分配在激进优化编译器中的位置

bbs.theithome.com

超星公司制作请尊重作者版权

Bradlee、Eggers和Henry [BraE91]以及Pinter [Pint93]讨论了将寄存器分配和指令调度合并成一遍的编译方法。

16.8 练习

16.1 解释如何使用图16-12中的Build_AdjMtx()和图16-15中的Coalesce_Registers()保证在过程调用之前将参数值传送到适当的寄存器中，并且在被调用过程内，将传递到寄存器中的这些参数转移到对应的工作寄存器中。

16.2 解释如何使用图16-12中的Build_AdjMtx()和图16-15中的Coalesce_Registers()构造机器指令，要求特定源寄存器和目的寄存器能够使其操作数和结果具有这种寄存器。

16.3 解释如何使用图16-12中的Build_AdjMtx()和图16-15中的Coalesce_Registers()，使得要求其结果在一个寄存器中，并且一个操作数也必须在该寄存器中的两地址指令能根据需要处理。

16.4 修改图16-15的代码使它能够保证为那种需要一对寄存器用于保存结果或操作数的指令能指派到一对寄存器。

16.5 修改图16-16的算法Compute_Spill_costs()，使它生成溢出代码时能够考虑到这种情况：如果一个已溢出的值在同一基本块中使用多次，并且在块中最后一次使用之前没有被杀死，则这个值在此基本块中只需要取一次。

ADV 16.6 推广图16-17的图，对于每一个R，产生一个最小的（即结点个数最少的）可R色着色、但用度<R规则却不能判别是R色着色的图。解释怎样推广它。

16.7 对于一个具有w个网的过程，其(a)邻接矩阵和(b)邻接表所需要的空间有多大?

16.8 修改图16-24中的过程Gen_Spill_Code()，使它能处理16.3.7节末尾提到的问题，即(a)使值重新回到寄存器，(b)删除复写指令，以及(c)已溢出的值在基本块内的多次使用。注意在图16-41中，假如我们不溢出s7，而是在调用print()之前再形成它，本可以生成更好的代码。

ADV 16.9 开发16.3.12节提及的确定在何处使值重新回到寄存器是有益的数据流分析。

527 ~ 530 RSCH 16.10 阅读Callahan和Koblenz [CalK91]，Knobe和Zadeck[Knoz92]，或Gupta、Soffa和Steele [GupS89]的论文，并将他们的方法与本章讨论的图着色方法进行对比。

bbs.theithome.com

超星公司制作 请尊重作者版权

第17章 代码调度

这一章我们关注的是为改善程序性能而对指令进行调度或重排的各种方法，对于大多数机器和大多数程序而言，这是最重要的优化之一。这些方法包括基本块调度、分支调度、跨基本块调度、软流水、踪迹调度及渗透调度。本章也涵盖了与超标量实现有关的优化。

早在RISC机器出现之前就已存在流水线机器，但是它们的流水线一般隐藏在对用户指令进行解释的微引擎内。为了使这种机器的速度最大化，编写微代码的关键是尽可能地使得多条指令重叠执行。此外，用户编写的代码也应当更好地利用微引擎中的流水线，否则就不能发挥流水线的优势。Rymarczyk [Ryma82]的经典论文为汇编语言程序员编写流水线处理机（如IBM/370系统）的代码提供了指南。当前，越来越多的CISC实现，如Intel的Pentium和Pentium Pro，也大量使用流水线。为了最大限度地发挥RISC处理器以及当前和未来的CISC处理器的性能，代码调度优化至为关键。

指令调度算法萌芽于微代码压缩和作业调度的研究，但是这三个问题有很大的不同，并且指令调度中使用的许多技术相对较新。

基本块调度和分支调度的结合是本章讨论的最简单的方法。这种方法对每一个基本块和每一个分支独立地进行操作，它实现起来最简单，并且能够使代码速度得到相当大的改善，常常能加快10%，甚至更高。跨基本块调度是对基本块调度的一种改进，它每次考虑由多个基本块组成的一棵树，并且可以将指令从一个基本块移到另一个基本块。

软流水专门针对循环体操作，因为循环是大多数程序花时间最多的部分，因此，它能使性能有很大的改善，常常能提高一倍以上。

有三种转换能显著改善基本块调度效果，尤其是软流水效果的转换，它们是：循环展开、变量扩张和寄存器重命名。循环展开创建一个较长的基本块，并为循环体内的跨基本块调度提供机会。变量扩张将已展开的循环体中的变量扩张成每个循环体副本有一个该变量的副本；循环执行完成之后，再将这些副本的值合并到一起。寄存器重命名是一种转换，它通过改变代码块内（或更大的代码单位内）寄存器的使用，删除由于不必要的立即重复使用某些寄存器而导致的依赖和资源限制，从而改善这两种调度方法的效果。

在进行软流水优化的编译器中，重要的是要尽可能地使得软流水的循环体能有效地执行循环展开、变量扩张和寄存器重命名。如果编译器不做软流水优化，则循环展开和变量扩张应当在编译处理的较早阶段进行；我们建议在图17-40的C4框中的死代码删除和代码提升之间进行。

踪迹调度和渗透调度是两种全局（即过程范围内的）代码调度方法，对于某些类型的程序和体系结构，特别是高度超标量的和VLIW的机器，它们都能取得非常好的效果。

本章讨论的所有转换，除了踪迹调度和渗透调度之外，都属于编译器在编译一个程序时最后被执行的几种优化。不过，后面两种方法最好构造成优化处理的驱动程序，因为在过程的结构上它们可以做出范围相当广泛的选择，并且它们通常也能通过调用其他优化来修改代码，从而允许更有效的调度并从中受益。

17.1 指令调度

因为许多机器（包括所有RISC的实现和从80486开始的Intel体系结构实现）都是流水线的，

bbs.theithome.com

并且对用户至少暴露了流水线的某些方面，因此，用一种能够更好地利用体系结构流水线特点的方式来安排这类机器的代码就很重要。例如，考虑图17-1a中的LIR代码。假设每一条指令需要1拍，但这两种情况除外：(1) 对于从内存取到寄存器的值，使用该值需要额外的一拍；(2) 分支到达它的目的地需要2拍，但是可以在分支的延迟槽中放置一条指令来利用第2拍。如果硬件具有互锁，则图17-1a中的指令能正确执行，并且执行它需要7拍。其中，在第2条和第3条指令之间有一拍的停顿，因为不能立即使用由第2条指令取到r3的值。还有，分支指令包含未能
532 利用的一拍，因为在它的延迟槽中有一条nop。另一方面，如果我们将这些指令重排成如图17-1b所示，这段代码仍能正确执行，但它只需要5拍。现在，第3条指令没有使用前一条指令取的值，第5条指令的执行与分支指令同时完成。

```
1        r2 ← [r1](4)
2        r3 ← [r1+4](4)
3        r4 ← r2 + r3
4        r5 ← r2 - 1
5        goto L1
6        nop
```

a)

```
r2 ← [r1](4)
r3 ← [r1+4](4)
r5 ← r2 - 1
goto L1
r4 ← r2 + r3
```

b)

图17-1 a) 一个LIR代码基本块，b) 它的一个更好的调度，其中假定goto之后有一个延迟槽，并且在取指令和所取之值达到目的寄存器成为可用之间有1拍的延迟

有些体系结构没有互锁，如MIPS的第一代商业版本，因此图17-1a的代码将不能正确执行——指令2所取的值要到指令3已读过r3之后才能到达r3。在关于指令调度的讨论中，我们忽略这种可能性，因为当前所有的商业体系结构都有互锁。

指令调度涉及到了若干问题，其中两个最基本的问题是，如何填充分支延迟槽（17.1.1节），以及如何调度基本块内的指令使其执行时间最短。我们用5小节的篇幅介绍后一问题，即17.1.2节论述表调度，17.1.3节介绍指令调度器的自动生成，17.1.4节与超标量实现有关，17.1.5节讨论调度与寄存器分配之间的相互作用，17.1.6节讨论跨基本块的调度。

在这几节之后我们考虑软流水和其他更激进的调度方法。

17.1.1 分支调度

分支调度（branch scheduling）指的是两件事：(1) 用一条有用的指令填充分支指令之后的延迟槽，和(2) 隐藏处于比较指令和根据比较结果进行分支的指令之间的延迟。

分支的实现在不同的体系结构之间变化很大。有些体系结构——如PA-RISC、SPARC和MIPS，具有含一个（很少有的情形，如MIPS含2个）显式延迟槽的带延迟的分支指令。延迟槽可用有用的指令或nop指令来填充，但后者浪费执行时间。有些体系结构，如POWER和PowerPC，要求在条件判断指令的执行和使用此条件的分支指令发生转移之间延迟若干拍，如果在分支执行之前所需要的时间还未过去，处理机将在分支指令停顿剩余的延迟。Intel 386的高
533 端成员，如Pentium和Pentium Pro，也要求在判断一个条件和根据此条件分支之间有延迟时间。

延迟槽和用有用的指令填充它们

有些分支结构提供可废弃延迟槽中指令的分支指令，这种分支指令根据是否发生转移以及特定分支指令的定义细节，决定是执行延迟槽中的指令还是跳过它。在两种情形下，这条延迟槽中的指令都占用1拍，但是如果可以根据分支走一条路径还是走另一条路径而作废放置在延迟槽中的指令时，就能够比较容易地填充延迟槽。

bbs.theithome.com

在许多RISC中，调用指令也是有延迟的，而在CISC中，只有当不能在离分支足够远的地方计算转移地址以便从目标地址预取指令时，调用指令才有延迟。

为了具体起见，我们用SPARC中的分支延迟方法作为基本模型，它的分支延迟方法实质上包含了其他体系结构的所有基本特征。SPARC有带一拍延迟的条件分支指令，这一拍延迟可以通过设置指令中的一位而使它作废。作废（nullification）导致延迟槽中的指令仅在一个不是“总是分支”（branch always）的条件分支指令发生转移时才执行，而对于“总是分支”的转移指令则不执行。跳转指令（它是无条件的）和调用指令有一拍不能作废的延迟。浮点比较指令和使用这个比较结果的浮点分支指令之间必须至少有一条不是浮点比较的指令。SPARC-V9含有与MIPS和PA-RISC体系结构中类似的可以计算分支条件的分支指令，以及条件传送指令，这种条件传送指令在某些情况下删除了对（向前）分支指令的需要。

填充分支延迟槽最好使用来自该分支指令所结束的这个基本块中的指令。为了实现这个目标，我们可以修改下面给出的基本块调度算法，首先检查该基本块的依赖DAG中是否有任何叶子结点能够放到它最后一条分支指令的延迟槽中。这种指令必须满足如下条件：（1）它必须是与分支指令可置换的——即它既不能是确定分支条件的指令，也不能是改变分支地址计算使用的寄存器之值或分支所使用的任何其他资源（如条件代码域）的指令[⊖]；（2）它本身不能是分支指令。如果在当前基本块中存在若干可选的指令能够填充此延迟槽，我们选择那种只需要一拍的指令（取决于分支转移到的那条指令或分支下面的这条指令），而不是有延迟的取数指令或其他可能使流水线停顿的指令。如果当前基本块中有可用的指令，但没有只需一拍的指令，我们选择延迟可能性最小的指令。

下面，假设我们正在处理一条件分支指令，同时关注该分支的目标基本块和它下面的这个基本块。如果当前基本块没有可以放置在分支延迟槽中的指令，下一步是构造分支目标基本块和它下面这个基本块的DAG，并尝试找出那种同时作为这两个DAG的根而出现的指令；或者那种在一个分支中出现，并可以通过寄存器重命名（参见17.4.5节）而移到延迟槽的指令。如果还是没有找到这种指令，接下来的选择是在目标基本块中寻找满足下列条件的指令：它是目标基本块DAG的一个根，并且可以迁移到带有作废位的分支指令的延迟槽中，使得当分支走的是下降路径时该指令不会起作用。 534

填充无条件分支指令或跳转指令的延迟槽，其处理与条件分支指令的处理类似。对于SPARC的“总是分支”，当设置了作废位时，延迟槽中的指令不起作用。对于跳转指令，延迟槽中的指令总是被执行，并且目标地址由两个寄存器之和给出。图17-2概括了上面的规则。

填充调用指令的延迟槽与填充分支指令的延迟槽类似，但约束要多一些。但一般至少会存在一条取参数值或复制参数值到某个寄存器的指令，并且这种指令几乎总是可以与调用指令交换位置。如果在调用指令之前没有能够放到其延迟槽中的指令，则在调用指令之后可能有属于同一个基本块的、可以放到其延迟槽的指令。不过这时需要小心，因为被调用的过程可能不一定会返回到调用点，因此，放到调用之后的那条指令必须是那种当控制从交错返回点继续执行时，不会影响执行效果的指令。如果在含有调用指令的基本块中根本不存在可以放到延迟槽的指令，下一步可寻找的地方是被调用过程。当然，是否能得到被调用过程的代码取决于编译过程的结构，以及在什么时候执行分支调度。假设能得到被调用过程的代码，那么最简单的选择是被调过程的第一条指令，因为可以将它复制到延迟槽，并且可以将调用指令的目的地址修改

⊖ SPARC不会有这个问题，因为条件分支的目标地址是PC值和一个直接数之和，但是其他体系结构可能会存在这种问题。

bbs.theithome.com

为其后一条指令。其他的选择需要更多的协调，因为可能存在对一个过程的多个调用，而这些调用之间对延迟槽的需要可能相互冲突。

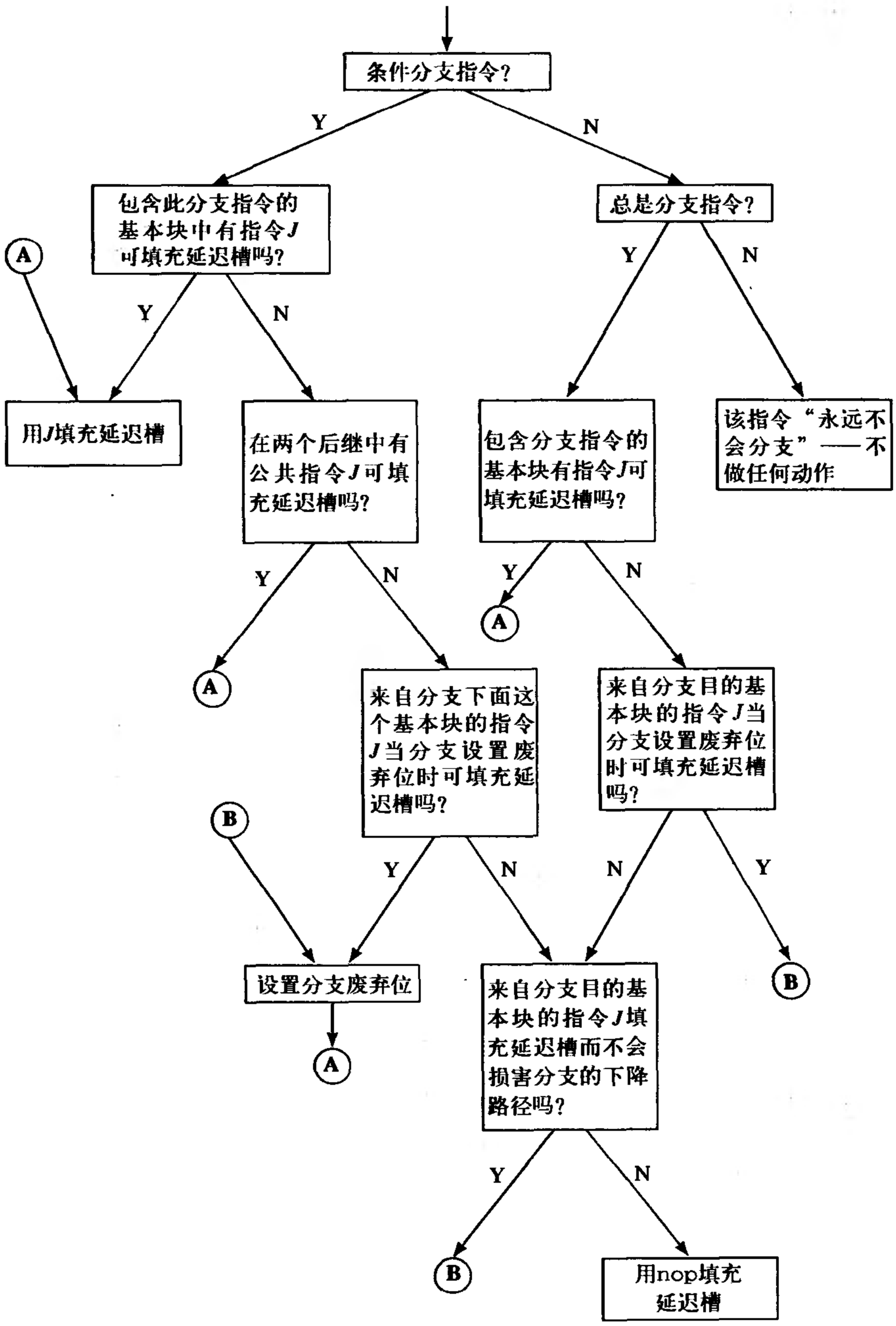


图17-2 填充分支延迟槽的处理流程图

对于其他所有失败情形，我们用nop来填充分支延迟槽。

停顿的时钟周期以及用有用的指令填充它们

有些机器——如POWER，PowerPC和Intel 386体系结构的各种实现——要求在条件判断指令和使用该条件进行分支转移的指令之间延迟若干拍；如果在执行分支指令时，所要求的延迟

bbs.theithome.com

越是公句制作请尊重作者版权

时间还没有过去，处理机将在此分支指令停顿，直到剩余的延迟消逝。SPARC的浮点比较指令和依赖于它们的条件分支指令也要求我们调度无关的指令放在它们之间。

这种情况最好由基本块调度来处理。我们在构造基本块的DAG时注意到，以条件分支结束的基本块，其条件是在基本块内计算的。于是在调度处理过程中，只要比较指令满足有关的依赖就应尽早地放置它。

17.1.2 表调度

535 ~ 536

基本块调度的目的是构造DAG的拓扑序，使得(1)生成同样的结果，(2)基本块的执行时间尽可能少。

在一开始我们应注意，基本块调度问题是NP困难问题，即使是非常简单的形式也如此。因此，我们必须寻找有效的启发式方法，而不是精确的方法。我们给出的这个表调度算法，其运行时间在最坏的情况下是$O(n^2)$，其中n是基本块中的指令条数，但实际中它的运行时间通常是线性的。表调度的总体性能一般受构造依赖DAG（参见9.2节）的时间控制，构造依赖DAG的时间在最坏的情况下是$O(n^2)$，但在实际中通常是线性的或稍微低一点。

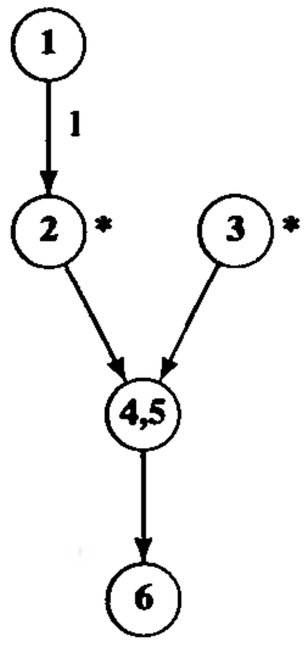
现在，假设我们有一个如9.2节所述的关于基本块的依赖DAG。在讨论它的调度方法之前，我们必须先考虑过程调用指令的处理。如果所考虑的体系结构的调用指令具有延迟槽，则需要选择一条指令来填充它，最好是选择这条调用指令之前的指令（如17.1.1节所讨论的那样）。一个调用典型地隐含着一组需要调用者在调用之前保护和调用结束之后恢复的寄存器。另外，如果缺乏过程间数据流分析和别名分析（参见19.2节和19.4节），我们就无法知道被调用过程会影响哪些存储单元，除非被编译语言的语义提供保证。例如，我们可能只知道调用者的局部变量对被调用者是不可见的（如Fortran），因此，位于调用指令之前并且能够与它置换的存储器引用指令就很少。我们可以将调用指令看成是基本块的边界，并为调用之前的指令和调用之后的指令建立分开的DAG，但这样做会减少指令重排的自由度，因而导致较慢的代码。可选的另一种方法是，我们可以在`Conflict()`函数（参见9.2节）的定义中将隐含的调用者保护的寄存器考虑进来，顺序排列与一个调用有关的所有其他存储访问（或至少可能被影响到的存储访问），特别标示出那些可以放到调用指令延迟槽的指令，并将已生成的位于调用指令之后的nop合并到这条调用指令的结点中。用于填充延迟槽的指令最好选择那种将参数值传送到参数传递寄存器的指令。例如，假设我们有图17-3的LIR代码，并且寄存器r1到r7用于传递参数。我们用星号标示了那些可移到调用指令延迟槽的指令所对应的结点。于是，可以按如下顺序调度这个基本块的指令:

1, 3, 4, 2, 6

并且，我们已成功地用一条有用的指令替代了nop。

```
1    r8 ← [r12+8](4)
2    r1 ← r8 + 1
3    r2 ← 2
4    call r14,r31
5    nop
6    r9 ← r1 + 1
```

a)



b)

图17-3 a) 一个含调用的基本块，b) 它的依赖DAG。用星号标示的结点对应的指令可以移到调用延迟槽

bbs.theithome.com

有若干种指令级转换可以用来改善指令调度的自由度。例如，图17-4的两个序列具有相同的效果，但在特定的情况下，其中的一种可能比另一种能产生更好的调度。在调度中通过一个子程序来识别这两种情形，并使得可对这两种选择进行尝试，便可处理这种情况，但是这种策略需要仔细地进行控制，否则它会导致可能的调度选择数呈指数级增长。

```
ld    [r2+4],r3
add   4,r2,r2
```

a)

```
add   4,r2,r2
ld    [r2],r3
```

b)

图17-4 两组等价的指令，它们在实际中都能为指令调度提供比另一种更大的调度自由度

表调度（list）方法开始时先从DAG的叶结点向根结点进行遍历，在遍历过程中，用从那个结点到基本块末尾的最大可能延迟来标示每一个结点。令ExecTime(*n*)是执行与结点*n*相连的指令所需要的
537 拍数。我们计算函数

```
Delay: Node → integer
```

这个函数的定义是:

$$\mathbf{Delay}(n)=\begin{cases}\mathbf{ExecTime}(n) & \text{若}n\text{是叶结点}\\ \max\limits_{m\in\mathbf{DagSucc}(n,DAG)}\mathbf{Late\_Delay}(n,m) & \text{其他}\end{cases}$$

其中，DagSucc(*i*, *DAG*)是DAG中*i*的后继集合，Late_Delay(*n*, *m*) = Latency(LInst(*n*), 2, LInst(*m*), 1)+Delay(*m*)。为了计算函数Delay()，我们如图17-5所示进行处理，其中

```
PostOrd: array [1..n] of Node
```

是依赖DAG中*n*个结点的后序列表，LInst(*i*)是DAG中结点*i*所表示的LIR指令。

```
Leaf: Node ⟶ boolean
Delay, ExecTime: Node ⟶ integer
LInst: Node ⟶ LIRInst
DagSucc: (Node × Dag) ⟶ set of Node
Heuristics: (set of Node) ⟶ Node

procedure Compute_Delay(nodes,PostOrd)
   nodes: in integer
   PostOrd: in array [1··nodes] of Node
begin
   i, d, ld: integer
   n: Node
   for i := 1 to nodes do
      if Leaf(PostOrd[i]) then
         Delay(PostOrd[i]) := ExecTime(PostOrd[i])
      else
         d := 0
         for each n ∈ DagSucc(PostOrd[i],Dag) do
            ld := Latency(LInst(PostOrd[i]),2,LInst(n),1) + Delay(n)
            d := max (ld,d)
         od
         Delay(PostOrd[i]) := d
      fi
   od
end    || Compute_Delay
```

图17-5 计算Delay()函数

接下来，我们从根结点向叶结点方向遍历DAG，在遍历过程中选择要调度的结点，并记录当前时间（CurTime，这个时间值从0开始），和每一个结点为避免一个停顿而应当被调度的

bbs.theithome.com

越是公司制作请尊重作者版权

最早时间（ETime[n]）。Sched是已经调度过的结点的序列；Cands是在每一点的候选结点集合，候选结点是那些还没有被调度的，但其前驱已经被调度了的结点。Cands有两个子集：MCands，即到基本块末尾具有最大延迟时间的候选者集合；和ECands，即MCands中最早开始时间小于或等于当前时间的结点集合。算法的ICAN代码在图17-6中给出。该算法使用了下面一些函数：

538

1. Post_Order(D)返回一个其元素是DAG D中结点的拓扑序的数组（该数组的第一个元素的索引是1，并且最后一个元素的索引是|D.Nodes|）。

2. Heuristics()对当前候选结点集合应用我们选择的启发式方法；注意，为了做出选择，它可能还需要当前候选者之外的其他信息。

3. Inst(i)返回依赖DAG中结点i表示的指令。

4. Latency(I_1, n_1, I_2, n_2)与9.2节的定义一样，是在I_1执行第n_1个时钟周期时开始执行I_2的第n_2个时钟周期所需要的等待拍数。

5. DagSucc(i, D)，它已在前面说明过。

```
DAG = record {Nodes, Roots: set of Node,
              Edges: set of (Node × Node),
              Label: (Node × Node) ⟶ integer}

procedure Schedule(nodes,Dag,Roots,DagSucc,DagPred,ExecTime)
   nodes: in integer
   Dag: in DAG
   Roots: in set of Node
   DagSucc, DagPred: in (Node × DAG) ⟶ set of Node
   ExecTime: in Node ⟶ integer
begin
   i, j, m, n, MaxDelay, CurTime := 0: integer
   Cands := Roots, ECands, MCands: set of Node
   ETime: array [1··nodes] of integer
   Delay: Node ⟶ integer
   Sched := []: sequence of Node
   Delay := Compute_Delay(nodes,Post_Order(Dag))
   for i := 1 to nodes do
      ETime[i] := 0
   od
while Cands ≠ ∅ do
   MaxDelay := -∞
   for each m ∈ Cands do
      MaxDelay := max(MaxDelay,Delay(m))
   od
   MCands := {m ∈ Cands where Delay(m) = MaxDelay}
   ECands := {m ∈ MCands where ETime[m] ≤ CurTime}
   if |MCands| = 1 then
      n := ♦MCands
   elif |ECands| = 1 then
      n := ♦ECands
   elif |ECands| > 1 then
      n := Heuristics(ECands)
   else
      n := Heuristics(MCands)
   fi
   Sched ⊕= [n]
```

图17-6 指令调度算法

bbs.theithome.com

```
        Cands -= {n}
        CurTime += ExecTime(n)
        for each i ∈ DagSucc(n,Dag) do
            if !∃j ∈ integer (Sched↓j=i) &
                ∀m ∈ DagPred(i,Dag) (∃j ∈ integer (Sched↓j=m)) then
                Cands ∪= {i}
            fi
            ETime[i] := max(ETime[n],
                CurTime+Latency(LInst(n),2,LInst(i),1))
        od
    od
    return Sched
end     || Schedule
```

图17-6 （续）

作为表调度算法的一个例子，考虑图17-7的依赖DAG，假设ExecTime(6)=2，并且对其他所有结点n，ExecTime(n)=1；对所有指令偶对I_1和I_2，Latency(I_1, 2, I_2, 1)=1。Delay()函数是

结点	Delay
1	5
2	4
3	4
4	3
5	1
6	2

并且调度的处理步骤如下：

1. 开始时，CurTime=0，Cands={1, 3}，Sched=[]，且对所有结点n，Etime[n]=0。MaxDelay的值是4，MCands=ECands={3}。

2. 结点3被选上；Sched=[3]，Cands={1}，CurTime=1，ETime[4]=1。

3. 因为|Cands|=1，其内只有一个结点1，故接下来1被选择。因此Sched=[3, 1]，Cands={2}，CurTime=2，ETime[2]=1，ETime[4]=4。

4. 因为又有|Cands|=1，结点2被选择，所以Sched=[3, 1, 2]，Cands={4}，CurTime=3，ETime[4]=4。

5. 再次有|Cands|=1，所以结点4被选择；结果 Sched=[3, 1, 2, 4]，Cands={5, 6}，CurTime=4，ETime[5]=6，ETime[6]=4。

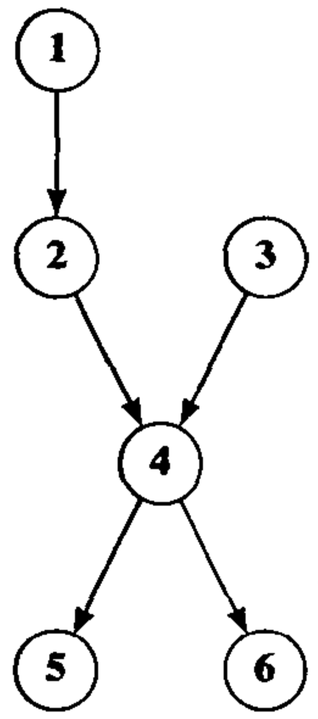


图17-7 依赖DAG之例

6. 现在，MaxDelay=2并且MCands={6}，所以结点6被选择；结果有Sched=[3, 1, 2, 4, 6]，Cands={5}，CurTime=5，ETime[5]=6。

7. 此时还剩下一个候选结点（结点5），因此它被选择并且算法终止。

最后的调度是

Sched=[3, 1, 2, 4, 6, 5]

并且这个调度需要6拍时钟周期，这正好就是最小的可能值。

已证明这个算法的一种实现所产生的调度在最坏情况下，对于具有一条以上相同流水线的

bbs.theithome.com

越是公句制作请尊重作者版权

539 ~ 541

机器而言，至多是最优时间的两倍；对于具有p条不同功能部件流水线的机器而言，至多是$p+1$倍。实际中，它产生的调度几乎总是比这个上界更好。

当|MCands|>1或|ECands|>1时，可以使用各种启发式来做出实用的选择。这些启发式策略包括:

1. 从MCands选择具有最小ETime[n]的结点n。

2. 如果体系结构有$p>1$条流水线，并且每条流水线都有若干候选结点，优先选择最近没有为它调度指令的那些流水线上的候选结点。

3. 优先选择那种在选择它之后能使新产生的Cands的元素个数最大的指令。

4. 优先选择那种能释放寄存器或者能避免使用额外寄存器，因而能减轻寄存器压力的指令。

Smotherman等人观察了指令调度中使用的一些DAG类型，并给出了长长的一列启发式清单，他们描述的6种不同调度方法的实现使用了其中的一些子集（进一步阅读参见17.8节）。Gibbons和Muchnick从根结点向叶结点构造依赖DAG，即，从基本块的最后一条指令开始向上处理，以便最有效地处理PA-RISC中的借位位。借位位定义频繁但使用相对很少，因此，自底向上构建DAG能先注意到它们的使用，并且仅当有一个向上已暴露的使用时，才会去关注它们的定值。

注意，指令调度研究中有一些采用的是与依赖DAG不同的表示。具体地，Hennessy和Gross使用的是所谓的机器级DAG，这是4.9.3节讨论的DAG中间代码形式的一种改编版本。这种改编包括用机器寄存器和存储位置作为叶子结点和标号，以及用机器指令作为中间结点。这种DAG中出现的显式约束比较少，如图17-8中的例子所示。对于图a)中的LIR代码，Hennessy和Gross的方法将构造出图b)中的机器级DAG；我们的依赖DAG如图c)所示。假设r1和r4在基本块末尾都是不活跃的，机器级DAG允许两种正确的调度，即

1, 2, 3, 4

和

3, 4, 1, 2

而依赖DAG只允许第一种调度。但同时，机器级DAG也会允许不正确的调度，例如

542

1, 3, 2, 4

除非对调度处理增加一些规则，如Hennessy和Gross所做的那样，限制指令只调度到所谓的“安全位置”。我们认为这种方法作用于上面定义的这种DAG时并没有特别的优点，尤其是它导致指令调度的复杂性上升到了$O(n^4)$。

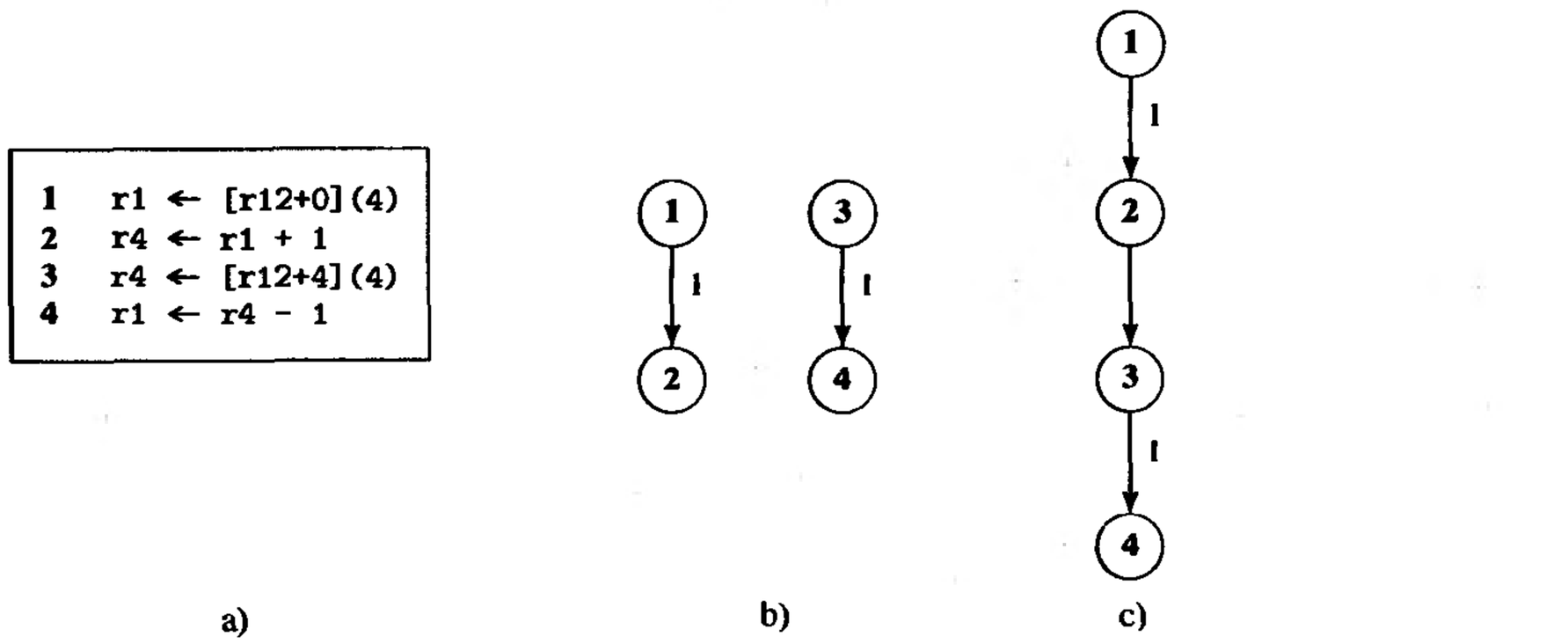


图17-8 a) 一个LIR基本块，b) Hennessy和Gross的机器级DAG，c) 我们的依赖DAG

bbs.theithome.com

版权所有 侵权必究

17.1.3 自动生成指令调度器

指令调度研究的另一个问题是，需要能够从机器描述自动生成调度器，并且实际上已经能自动生成了。这很重要，因为即使是用于单一体系结构的编译器也可能（并且几乎总是）需要涉及该体系结构的不同实现。各种实现常常各不相同，这种不同足以使得对于一种实现而言是非常好的一个调度器，对于其他实现则显得平庸普通。

因此，重要的是要能够从特定实现的机器描述来生成指令调度器，并尽可能地考虑到实现与调度相关的独特性。可能是最著名的，同时也肯定是分发最广的这种调度器的生成器是gcc（即GNU C编译器）中的指令调度器的生成器。它为编写编译器的人提供了书写机器描述所必需的便利，这种机器描述可以有书写者自己定义的特征，并且为刻画这些特征之间的相互作用提供了很大程度的灵活性。只要提供非常详细的有关实现的机器描述，如流水线结构、结构上的停顿（structural hazards）、延迟、低级并行规则等，它就能生成相当有效的调度器。

17.1.4 超标量实现有关的调度

超标量机器的调度需要尽可能精确地模拟CPU的功能部件组织，例如，通过使用带有某种
543 偏向的启发式而顾及到一个特定的处理器具有两条整数流水线、两条浮点流水线和一个分支部件（如POWER的某些实现），或一对指令仅当是双字对齐的才能够同时启动（如i860要求的）等等。后一种要求较容易做到，如通过插入nop使得每一个基本块开始于双字边界来实现；也可做更多的工作，通过记录需要边界对齐的每一对指令的边界，并在需要时使它对齐于双字边界来实现。

对于超标量系统，指令调度还需要照顾到将指令组织成可同时流出的指令组。这可以通过一种成组启发式，即贪婪算法来实现，这种贪婪算法尽可能地用就绪指令填充有效指令槽，具体方法如下。假设所考虑的处理机有n个可并行执行的部件$P_1, \cdots, P_n$，并且每一个部件P_i可以执行类别为PClass(i)的指令。我们用图17-6表调度算法中所用的数据结构的n个副本来模拟这些功能部件，并用IClass(*inst*)确定具体指令*inst*的类别，即，指令*inst*能够由执行部件i执行，当且仅当PClass(i) = IClass(*inst*)。于是能够将那个表调度算法修改成为一个用于超标量系统的简单易懂的调度器。

但是要记住，贪婪调度可能不是最优的，如图17-9中的例子所示。我们假定这个处理机有两条流水线，其中一条可执行整数和浮点操作，另一条可以做整数和存储操作；每一个操作的延迟是1拍。假设在指令之间的惟一依赖是FltOp必须先于IntLd，则图17-9a的贪婪调度需要两拍，而图17-9b中的贪婪调度需要三拍。

IntFlt	IntMem
FltOp	FltLd
IntOp	IntLd

a)

IntFlt	IntMem
FltOp	IntOp
	IntLd
	FltLd

b)

图17-9 关于超标量处理机的两种贪婪调度，其中a) 是最优的，b) 不是最优的

还有，我们必须小心在这种启发式中不要使用太多的超前察看，因为所有非平凡的指令调度实例至少都是NP困难的。这种调度可能通过跨控制流结构的调度而得到改善，即，如17.1.6节所讨论的，在调度中使用扩展基本块和/或反扩展基本块；或者用更强有力的全局技术使其得到改善。例如，使用扩展基本块
544 时，为了改善指令的成组关系，可能会需要将一条指令从一个基本块移到它的两个后继基本块中。

17.1.5 基本块调度中的其他问题

前面的贪婪指令调度方法的设计是为了隐藏从数据高速缓存取数启动开始到所取的值被寄存器接收之间的延迟。它并没有考虑到被取的数据可能不在高速缓存，并因此需要从主存或从

bbs.theithome.com

超星公司制作请尊重作者版权

二级高速缓存读取的情况，因而可能招致流水线出现相当长的、并且是不可预测的停顿。Eggers和她的同事（[KerE93]和[LoEg95]）提出了一种称为平衡式调度（balanced scheduling）的方法，其设计考虑到了这种可能性。他们的算法将出现在基本块中一系列取数指令中的延迟，通过调度其他指令到它们中间而由这些指令来分担。随着处理机速度的增加持续地超过存储器速度的增加，这种方法正逐渐变得重要起来。

寄存器分配和指令调度之间的相互影响也会产生严重的问题。考虑图17-10a的例子，它的依赖DAG如图17-11a所示。由于r1和r2紧接着都被使用，因此指令5到7中的任何指令都不能调度到指令1和4之前。如果改变寄存器分配，使5到7中的指令使用不同的寄存器，如图17-10b所示，则它的依赖DAG变成了图17-11b所示情形，从而显著增大了调度的自由度。与原来的寄存器分配相比，我们可以调度取数指令使得停顿不会发生，具体情况如图17-12所示，而原来的寄存器分配根本不能对指令进行重排。

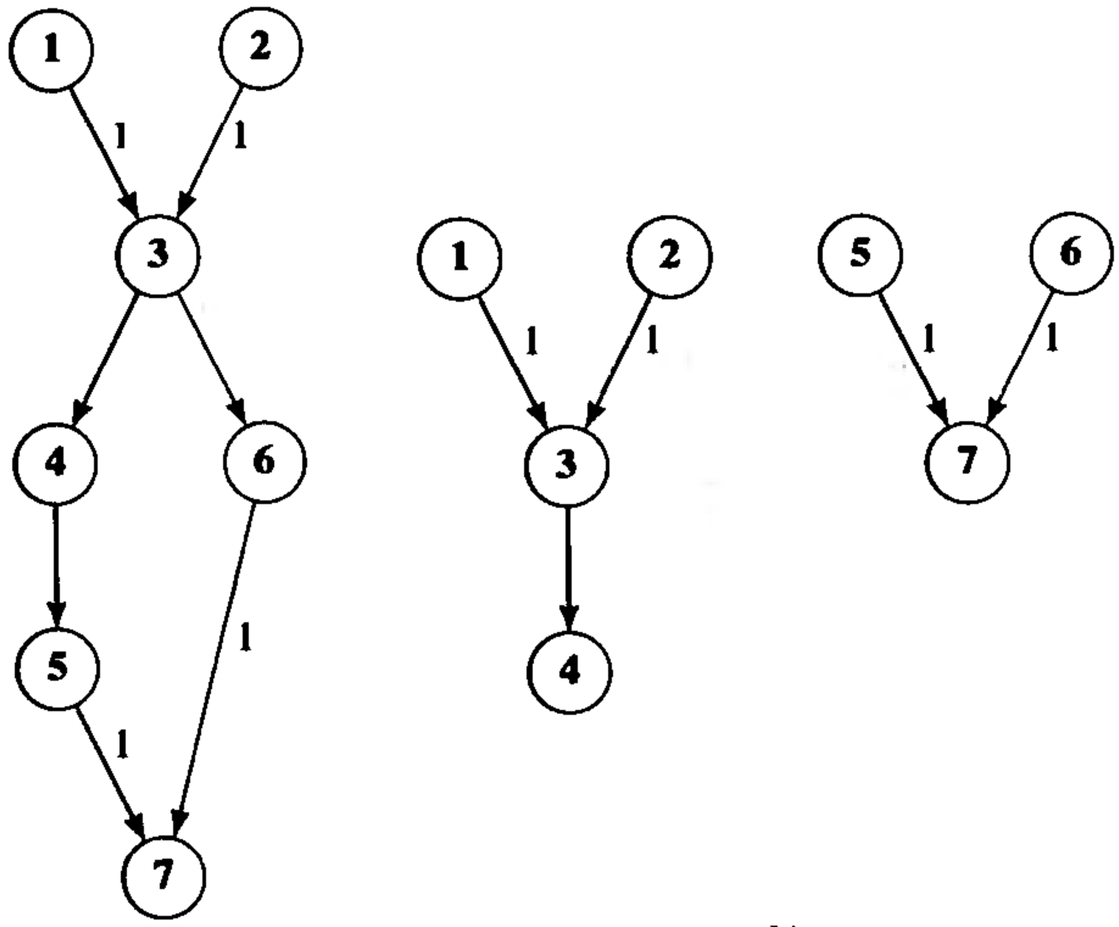
```
1        r1 ← [r12+0](4)
2        r2 ← [r12+4](4)
3        r1 ← r1 + r2
4        [r12,0](4) ← r1
5        r1 ← [r12+8](4)
6        r2 ← [r12+12](4)
7        r2 ← r1 + r2
```

a)

```
r1 ← [r12+0](4)
r2 ← [r12+4](4)
r3 ← r1 + r2
[r12,0](4) ← r3
r4 ← [r12+8](4)
r5 ← [r12+12](4)
r6 ← r4 + r5
```

b)

图17-10 a) 其寄存器指派对调度有不必要、限制作用的LIR基本块，b) 对它较好的一种寄存器分配



a)

b)

图17-11 图17-10中基本块的依赖DAG

```
r1 ← [r12+0](4)
r2 ← [r12+4](4)
r4 ← [r12+8](4)
r5 ← [r12+12](4)
r3 ← r1 + r2
[r12+0](4) ← r3
r6 ← r4 + r5
```

图17-12 图17-10b中基本块的调度，这种调度覆盖了所有取数延迟

为了实现这一点，我们在代码生成期间分配符号寄存器，然后在编译处理的较后阶段进行寄存器分配。我们在寄存器分配即将开始之前进行指令调度（即，对使用符号寄存器的代码进行调度），然后在寄存器分配生成了寄存器溢出代码的情况下，紧接其后再重复进行指令调度。IBM XL用于POWER和PowerPC的编译器（参见21.2节），Hewlett-Packard用于PA-RISC的编译器，以及Sun用于SPARC的编译器（参见21.1节）都采用这种做法；实践已证明这种方法比在寄存器分配之前或之后进行单独一遍调度的方法能生成更好的调度和更好

bbs.theithome.com

的寄存器分配。

17.1.6 跨基本块边界的调度

尽管有些程序有非常大的基本块，给调度提供了许多改善代码质量的机会，但也常常出现基本块太小以至于调度对它不起作用或起的作用非常小的情况。因此，值得做的是如循环展开（17.4.3节）所做的那样使基本块更长，或者扩充指令调度使其跨越基本块边界，我们将在17.4节讨论这种方法。

另一种途径是尽可能地在调度后继基本块之前调度各个基本块，并在调度一个后继基本块的初始状态中，考虑在调度它的前驱完结时遗留下来的任何延迟。

另一种方法是转换代码以便能够更好地覆盖分支延迟。具体地，Golumbic和Rainish [GolR90]讨论了3种转换，它们能帮助吸收掉POWER的比较（cmp）指令与发生转移的条件分支指令（bc）之间的3拍延迟，以及在cmp-bc-b序列中未发生转移的条件分支指令的4拍延迟。下面讨论的与循环关闭有关的分支是这种方法的一个例子。假设我们有一个如图17-13a所示的循环，它含有一拍未被覆盖的延迟。用下面的方法我们可以覆盖这一拍延迟，即，转换cmp指令使其测试与原来条件相反的条件（由!cond指定），用bc指令转移到循环出口，并在bc指令之后复制循环的第1条指令，然后插入一条转移到循环的第2条指令的无条件分支指令，结果得到图17-13b所示的代码[⊖]。显然也可推广这种方法用于覆盖有2拍或3拍延迟的情况。

545 ~ 546

```
L1: inst1
    inst2
    . . .
    cmp   cr0,cond
    instn-1
    instn
    bc    cr0,L1
    . . .
```

a)

```
L1: inst1
L3: inst2
    . . .
    cmp   cr0,!cond
    instn-1
    instn
    bc    cr0,L2
    inst1
    b     L3
L2: . . .
```

b)

图17-13 a) 含1拍未覆盖的cmp-bc延迟的POWER循环代码，b) 经转换后已覆盖此延迟的代码

17.8节给出了与跨基本块边界调度若干方法有关的引文。

17.2 前瞻取和上推

前瞻取（speculative loading）是一种机制，它能增加调度的有效自由度，并提供一种途径来隐藏为满足从内存取数而不是从高速缓存取数所固有的延迟。前瞻取指令（speculative load）是一条取指令，它所取的量在被使用之前不会发生任何存储器异常。这种取指令可以在不知道它生成的地址是否合法之前流出——如果较后发现地址不合法，只需要简单地避免使用所取的值即可。这种前瞻取指令存在于多流水线体系结构中，如SPARC-V9、PowerPC以及其他机器中。

例如，取链表的下一元素可以用一条位于测试是否已到达表尾的指令之前的前瞻取指令来启动——如果已经到达表尾，使用由这条前瞻取所得数据的指令将不会执行，因此不会产生问

⊖ 这是后面17.4.1节讨论的软流水窗口调度方法的一个实例。

bbs.theithome.com

题。这种情况的一个MIR例子在图17-14中给出。其中a）部分是典型的查找链表中特定值的函数例子。在b）部分中，标号为L2的那一行的赋值“p1←sp p*.next”使p1移动至我们正检查的这个对象（由p所指向的）之前的一个记录。只要对p1的赋值是前瞻取指令（由箭头之后的sp标志），就不会发生错误。 547

```
search:
    receive ptr (val)
    receive ptr (val)
    p ← ptr

L2: if p*.val = v goto L1
    p ← p*.next
    if p != NIL goto L2

    return 0
L1: return 1
```

a)

```
search:
    receive ptr (val)
    receive v (val)
    p ← ptr
L2: p1 ←sp p*.next
    if p*.val = v goto L1

    if p = NIL goto L3
    p ← p1
    goto L2
L1: return 1
L3: return 0
```

b)

图17-14 a) 搜索表的例程，b) 同一个例程，但取下一个元素的操作被上推到测试是否到达表尾的判断之前

因此，前瞻取指令可以移到判别合法性的测试之前，即从一个基本块移到前一个基本块，或从循环的一个迭代移到较早的一个迭代。这种代码移动叫做上推（boosting），Rogers和Li（参见17.8节的引文）介绍了实现它的有关技术。

17.3 前瞻调度

前瞻调度（speculative scheduling）是前瞻取的一种推广技术，其目的是为了使其他类型的指令能够向过程的入口方向跨一个或多个分支移动，尤其是移到循环之外。它有两种形式：安全前瞻调度和非安全前瞻调度。安全前瞻调度中，被移动的指令在新移到的位置执行时不会产生问题（大概除了可能导致计算变慢之外）；非安全前瞻调度中，被移动的指令必须受判断它们能否在被移到的新位置合法执行条件的保护。

前瞻调度的技术太新，由于它们对性能的影响还没有证明，所以我们目前只能提及这一主题和有关的参考文献（参见17.8节）。具体地，Ebcioğlu等人已开展了有关前瞻调度和它的对称操作，即反前瞻调度（unspeculation）方面的研究工作。Golumbic和Rainish的论文以及Bernstein和Rodeh的论文讨论了这一方向的早期工作。

17.4 软流水

软流水（software pipelining）是一种优化，它能改善任何允许指令级并行的系统的循环执行性能，包括VLIW和超标量系统，也包括那种允许整数和浮点指令同时执行但不同时启动执行的单标量实现。软流水通过允许同时处理循环的若干迭代来利用循环体中潜在的并行性。 548

例如，假设一台具有一个整数部件和一个浮点部件的单标量实现，其中浮点取和存由整数部件来执行（其执行拍数由阴影部分指明），同时假设图17-15循环中的指令具有图中所示的延迟，则如图17-16流水线示意图中的虚线所示，在此机器上执行这个循环的每一个迭代需要12拍。注意，在fadds和stf之间有6拍的流出延迟，如果我们能够使前一个迭代的存数指令与当前迭代的加法运算指令重叠执行，这6拍流出延迟就可以减少。复制第1个迭代的取数指令和

bbs.theithome.com

加法指令，以及第2个迭代的取数指令于循环之前，可以使得循环以存数指令开始，其后接着下一迭代的加法运算指令，然后是后继迭代的取数指令。这样做在循环之前增加了3条指令，并且还需要在循环之后增加5条指令来完成最后两个迭代，由此产生的代码如图17-17所示。如图17-18中的虚线所示，它将循环体的拍数减少为7拍，每个迭代的执行时间减少了5/12，大约是42%。只要这个循环总是至少迭代3次，这两种形式的作用就是相同的。

		流出延迟	结果延迟
L: ldf	[r1],f0	1	1
fadds	f0,f1,f2	1	7
stf	f2,[r1]	6	3
sub	r1,4,r1	1	1
cmp	r1,0	1	1
bg	L	1	2
nop		1	1

图17-15 一个简单的SPARC循环，带有假定的流出和结果延迟

另外，7拍也是这个循环的最小执行时间，除非我们展开它，因为fadds需要7拍。假如展开它，我们就能重叠两个以上的fadds，从而进一步提高性能。因此软流水和循环展开一般是互补的。另一方面，我们可能会需要使用额外的寄存器，因为同时执行的两个fadds必须使用不同的源寄存器和目的寄存器。

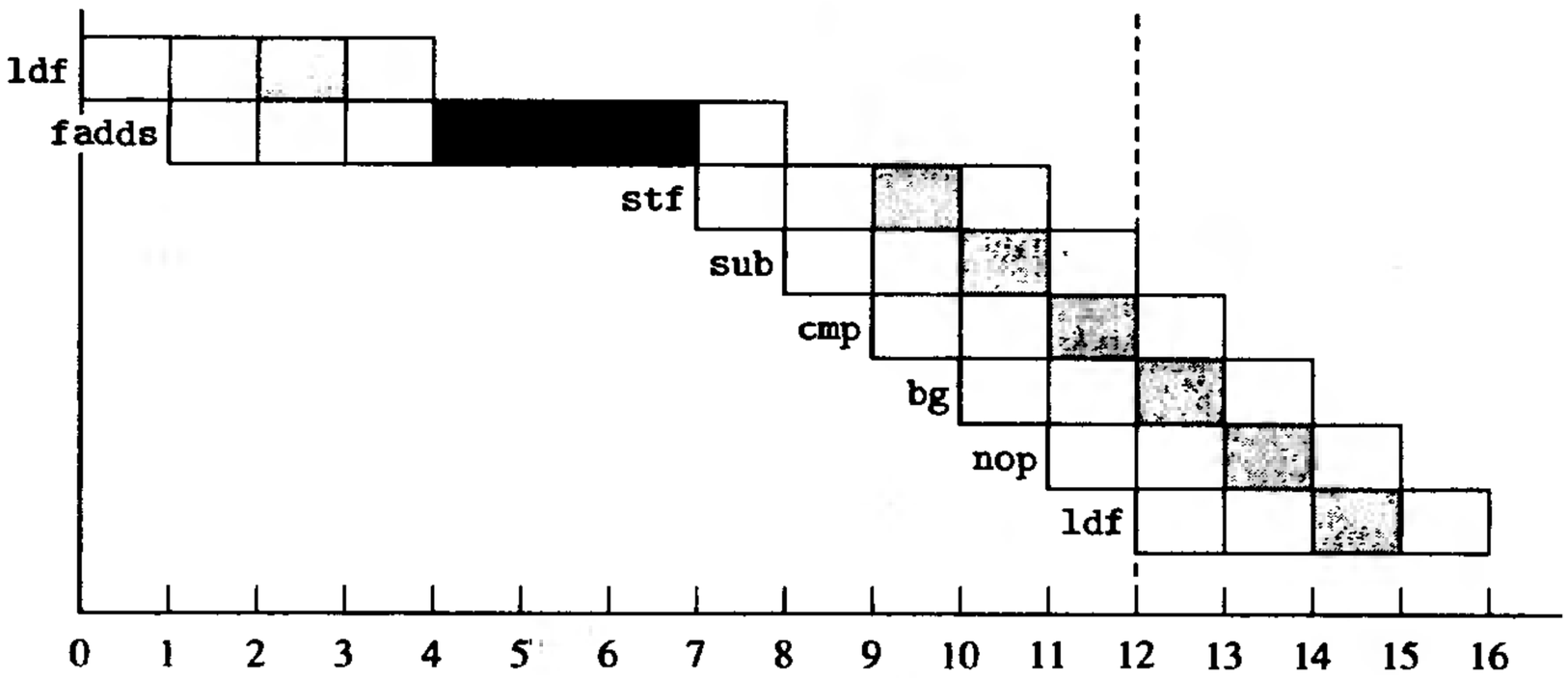


图17-16 图17-15循环体的流水图解

		流出延迟	结果延迟
ldf	[r1],f0		
fadds	f0,f1,f2		
ldf	[r1-4],f0		
L: stf	f2,[r1]	1	3
fadds	f0,f1,f2	1	7
ldf	[r1-8],f0	1	1
cmp	r1,8	1	1
bg	L	1	2
sub	r1,4,r1	1	1
stf	f2,[r1]		
sub	r1,4,r1		
fadds	f0,f1,f2		
stf	f2,[r1]		

图17-17 图17-15软流水循环的结果，带有对应的流出和结果延迟

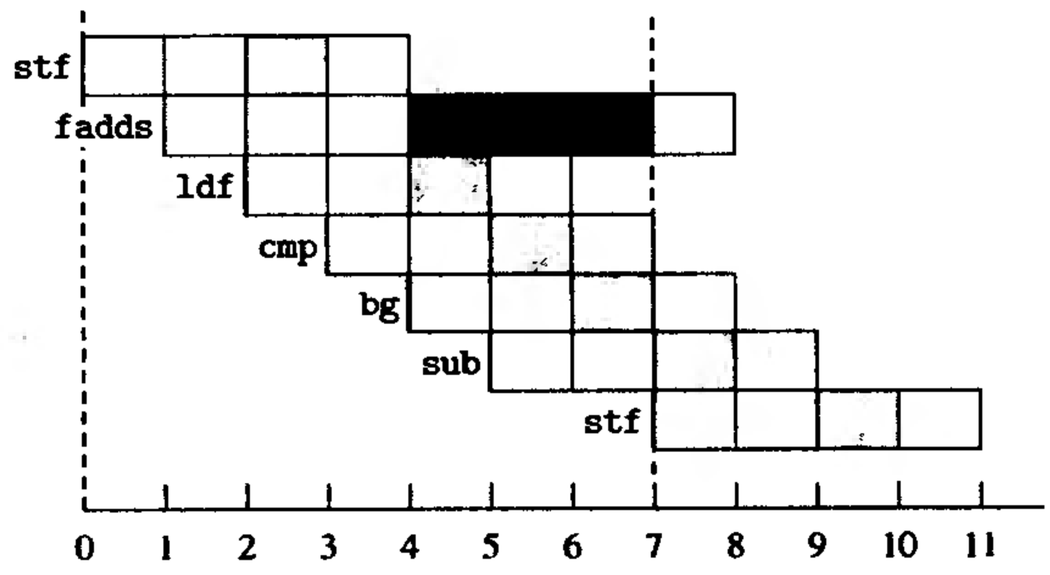


图17-18 图17-17循环体的流水图解

bbs.theithome.com

由于软流水移动到循环之外的迭代个数是固定的，因此我们必须要么事先知道循环至少重复了那么多次，要么可能的话，生成运行时测试它的代码，并根据测试结果选择执行循环的软流水版本还是执行非软流水版本。当然，有些循环，不执行它们就不可能确定其迭代次数，因而软流水不能施加于这种循环。

另一个考虑是我们究竟能够在多大程度上消除存储引用的歧义性才能使得对流水线的约束（如第9章讨论的）最小。存储引用歧义消除得越好，流水的自由度就越高，从而产生的调度一般也会更好。

在下面两小节中，我们讨论两种软流水方法，窗口调度和展开－压实软流水。第一种方法实现简单，而第二种方法一般能得到更好的调度。

17.4.1 窗口调度

称为**窗口调度**（window scheduling）的软流水方法，其名字源于它的软流水概念模型——它建立循环体依赖DAG的两个相互关联的副本，其中，循环体必须是单个基本块，然后用一个窗口沿这两个副本从上至下实施调度。该窗口在每一点都含有循环体的一个完整副本；位于窗口之上和之下的指令则分别成为了流水循环的填充部分（prologue）和排空部分（epilogue）。例如，图17-19a的依赖DAG变成了图17-19b所示的双DAG或窗口调度DAG，其中虚线标出了可能的窗口。随着窗口沿这两个副本向下移动，我们尝试由此得到的各种调度，并寻找那种能够降低整个循环体延迟的调度。我们在进行窗口调度的同时做其他指令调度，并使用基本块调度器来调度循环体。

549 ~ 551

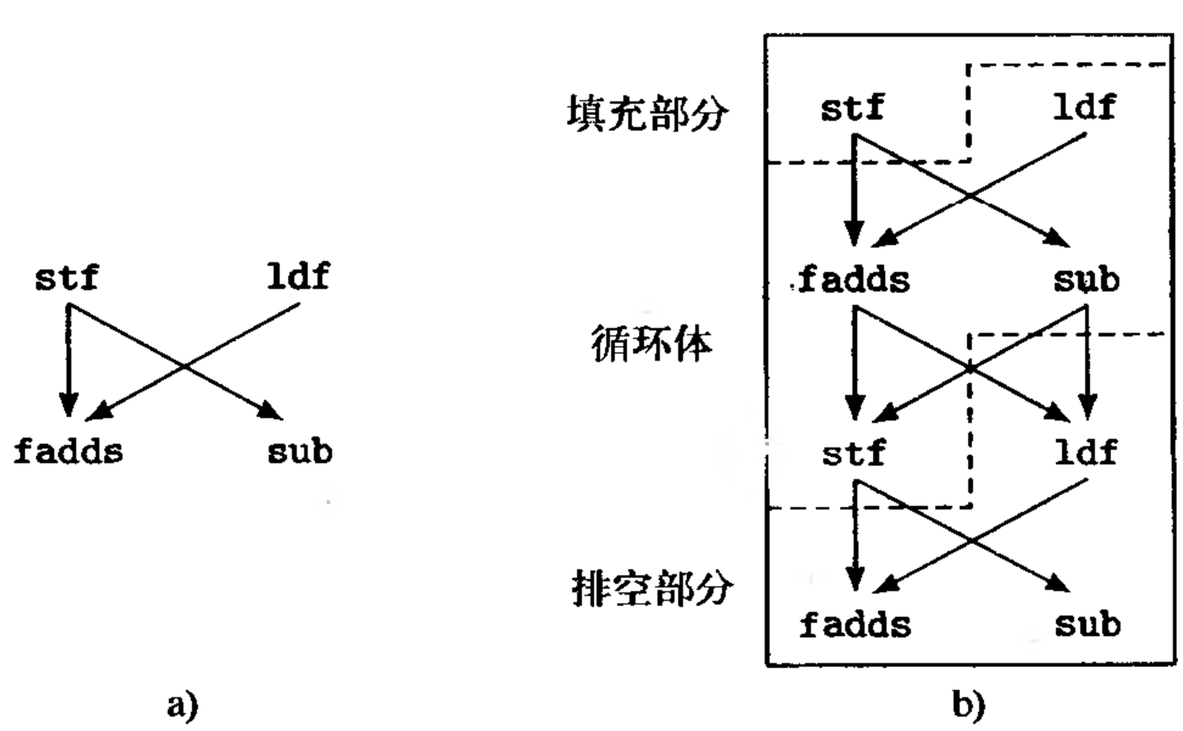


图17-19 a) 一个循环体例子的依赖DAG，和b) 用于窗口调度的双版本，虚线指出了可能的窗口

图17-20是窗口调度的ICAN算法轮廓。它的输入是Inst[1..n]，即构成要软流水的循环体基本块的指令序列。它首先构造基本块的依赖DAG（利用图9-6给出的Build_DAG()），并将其存储于Dag中。然后使用Schedulable()来判别该循环是否能用窗口调度，即它的循环索引在循环体内是否只增加了一次，并且循环至少执行两次。如果是这样，算法使用下面的信息：

1. Dag记录在窗口处理过程中使用的窗口调度或双DAG（由Double_DAG()构造）。
2. Window指出窗口的当前布局。
3. Sched记录窗口中当前对DAG（基本块）的调度。
4. Stall是与Sched相连的停顿拍数。
5. MinStall记录迄今为止尝试的所有调度中的最小停顿拍数。
6. BestSched是最后生成的具有最小停顿拍数的DAG。

bbs.theithome.com

```
DAG = record {Nodes, Roots: set of integer,
              Edges: set of (integer × integer),
              Label: set of (integer × integer) ⟶ integer}

procedure Window_Schedule(n,Inst,Limit,MinStall)
    n, Limit, MinStall: in integer
    Inst: inout array [1··n] of LIRInst
begin
   Dag, Window, Prol, Epi: DAG
   Stall, N: integer
   Sched, BestSched, ProS, EpiS: sequence of integer
   Dag := Build_DAG(n,Inst)
   if Schedulable(Dag) then
      Dag := Double_DAG(Dag)
      Window := Init_Window(Dag)
      BestSched := SP_Schedule(|Window.Nodes|,Window,Window.Roots,
         MinStall,ExecTime)
      Stall := MinStall
      repeat
         if Move_Window(Dag,Window) then
            Sched := SP_Schedule(|Window.Nodes|,Window,Window.
               Roots,MinStall,ExecTime)
            if Stall < MinStall then
               BestSched := Sched
               MinStall := min(MinStall,Stall)
            fi
         fi
      until Stall = 0 ∨ Stall ≥ Limit * MinStall
      Prol := Get_Prologue(Dag,Window)
      Epi:= Get_Epilogue(Dag,Window)
      ProS := SP_Schedule(|Prol.Nodes|,Prol,Prol.Roots,MinStall,ExectTme)
      EpiS := SP_Schedule(|Epi.Nodes|,Epi,Epi.Roots,MinStall,ExectTme)
      N := Loop_Ct_Inst(n,Inst)
      Decr_Loop_Ct(Inst[N])
   fi
end    || Window_Schedule
```

图17-20 窗口调度算法

limit由编译器的编写者来选择，窗口调度器用它来决定迄今为止已达到的最好的调度还应当徘徊多远才停止进一步寻找更好的调度。窗口调度处理中使用如下例程:

1. SP_Schedule()是从图17-6给出的基本块调度器而构造的，具体地，SP_Schedule(*N*, *Dag*, *R*, *stall*, *ET*)计算函数DagSucc()和DagPred()，调用Schedule(*N*, *Dag*, *R*, DagSucc, DagPred, *ET*)，设置*stall*为已产生的调度*sched*的停顿拍数，并返回*sched*作为返回值。

2. Move_Window()在双DAG中选择一条指令，并下移窗口越过该指令（假设存在移动窗口的余留空间），如果成功移动了窗口，返回true；否则返回false。

3. Get_Prologue()和Get_Epilogue()分别提取双DAG位于窗口之上和之下的部分。

4. Loop_Ct_Inst()确定哪条指令是对循环计数器进行测试的指令。

5. Decr_Loop_Ct()修改指令少做一个迭代。这样做是适当的，因为如迄今所描述的，这个算法只移动一个迭代到循环之外，循环本身做两个连续的迭代，总共比原循环体少做一个迭代。因此，我们需要由Get_Prologue()和Get_Epilogue()提取的代码在循环体之外做一个迭代。

6. ExecTime(*n*)是执行与DAG中结点*n*相连的指令所需要的拍数。

通过将窗口调度算法重复应用于由它产生的新循环体，不难将此算法推广到允许在流水循

bbs.theithome.com

环内处理多于两个以上的迭代。例如，以图17-15的循环例子作为开始，它的窗口调度DAG如图17-21a所示。移动窗口到ldf之下得到图17-22a的代码，以及图17-21b的窗口调度DAG。往下移动窗口越过fadds依次得到图17-22b中的代码和图17-21c的窗口调度DAG。最后，往下移动窗口第二次越过ldf得到图17-17的代码。

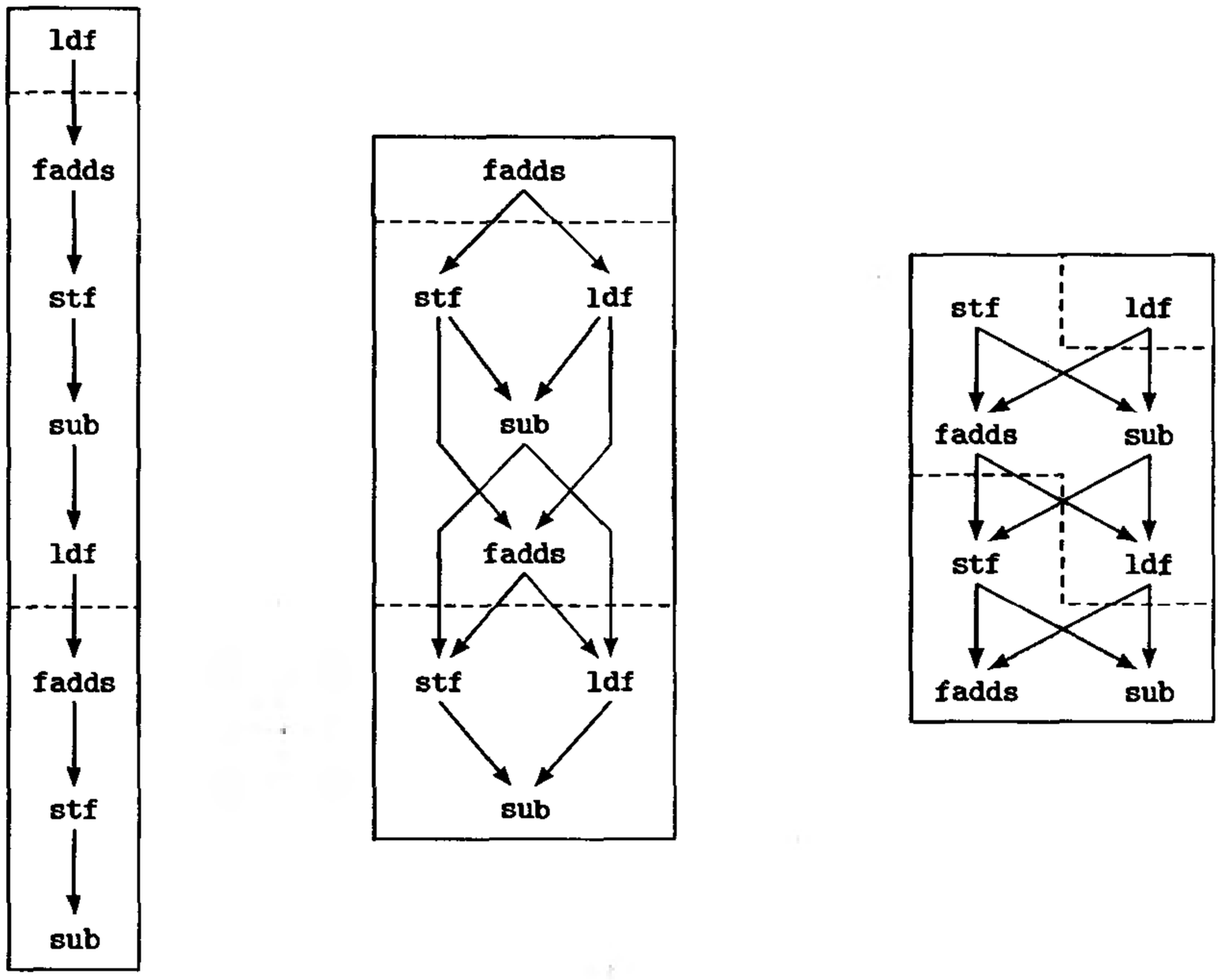


图17-21 在窗口处理过程中图17-15和图17-22中循环的连续版本的双DAG

```
    ldf    [r1],f0
L:  fadds  f0,f1,f2
    stf    f2,[r1]
    ldf    [r1-4],f0
    sub    r1,4,r1
    cmp    r1,4
    bg     L
    nop
    fadds  f0,f1,f2
    stf    f2,[r1-4]
    sub    r1,4,r1
```

a)

```
    ldf    [r1],f0
    fadds  f0,f1,f2
L:  stf    f2,[r1]
    ldf    [r1-4],f0
    fadds  f0,f1,f2
    sub    r1,4,r1
    cmp    r1,4
    bg     L
    nop
    stf    f2,[r1-4]
    sub    r1,4,r1
```

b)

图17-22 图17-15中循环的中间版本

注意，不难将循环展开与窗口调度相结合，这样做是有帮助的，因为它总是给调度器增加了有效的自由度：替代用Double_DAG()来创建循环体的一个副本，我们用一个子程序来建立展开因子为n（即创建$n-1$个副本）的展开的循环。然后简单地对由此得到的n个副本进行窗口调度。这种方法同变量扩张和寄存器重命名结合一起，常常能够获得更高的性能，因为它使得调度器有更多的指令可以调度。

552 ~ 554

17.4.2 展开-压实软流水

展开-压实软流水（unroll-and-compact software pipelining）是另一种软流水方法，它将循

bbs.theithome.com

环体展开若干次，然后在指令中寻找每一步都要启动的重复模式。如果找到这种模式，它就用该模式构造流水的循环体，并分别用该模式之前和之后的指令构造填充部分和排空部分。例如，假设我们有如图17-23所示的SPARC代码，其循环体的依赖DAG如图17-24所示（假设fadds和fmuls指令有3拍延迟，并且忽略分支延迟槽中的nop），图17-25展示了该循环迭代可能有的一种无约束的贪婪调度，其中假定每个时间步上列出的指令都能够并行执行。图中带框的部分表示软流水循环的填充部分、流水循环体和排空部分。我们说得到的调度是“无约束的”，因为它不考虑有效的寄存器个数或处理机指令级并行所强加的限制。

```
A      L: ld      [i3],f0
B         fmuls   f31,f0,f1
C         ld      [i0],f2
D         fadds   f2,f1,f2
E         add     i3,i5,i3
F         deccc   i1
G         st      f2,[i0]
H         add     i0,i5,i0
I         bpos    L
J         nop
```

图17-23 软流水循环的另一个例子

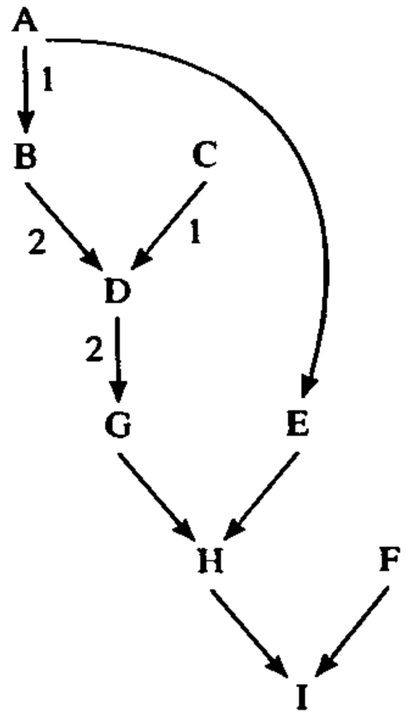


图17-24 图17-23中循环体的依赖DAG

时间	迭代步骤 1	2	3	4	
1	A C F				
2	B E				
3					
4					填充部分
5	D	A C F			
6		B E			
7					
8	G				
9	H	D	A C F		循环体
10	I		B E		
11					
12		G			
13		H	D	A C F	
14		I		B E	
15					排空部分
16			G		
17			H	D	
18			I		
19					
20				G	
21				H	
22				I	

图17-25 图17-23中循环迭代可能有的一种无约束的贪婪调度，框内的部分表示软流水循环的填充部分、流水循环体和排空部分

bbs.theithome.com

超星公司制作 请尊重作者版权

将结果代码转换为正确的软流水循环的填充部分、流水循环体和排空部分需要处理资源限制，并调整某些指令使它们按新的序列来执行。图17-26a展示了简单地按顺序罗列被展开的指令而得到的指令序列，而图17-26b展示的是考虑到指令的执行顺序以及需要额外寄存器的情况而对指令进行调整后的结果。注意，除了从排空部分删除两条分支指令外，我们还分别将“ld [i0],f2”和“fadds f2,f1,f2”除第一次之外的所有出现修改成“ld [i0+i5],f3”和“fadds f3,f1,f2”，以便使用正确的地址，并避免在存储寄存器f2的值之前重复使用f2。对前一条指令的修改是必须的，因为第二次出现的ld已被移到使寄存器i0增加的add指令之前。处理并行限制可能会为了更好地利用有效的执行部件而导致重排指令。例如，如果处理机允许一个存储器或整数操作，一个浮点加和一个浮点乘，以及一个分支同时处在处理中，则一种合法的调度选择可以是将循环中的第二个ld放在fmuls之下，但这样做会使得每个迭代需要10拍而不是当前所使用的这种指令排列需要的9拍。

555 ~ 556

```
    ld      [i3],f0
    ld      [i0],f2
    deccc   i1
    fmuls   f31,f0,f1
    add     i3,i5,i3
    fadds   f2,f1,f2
    ld      [i3],f0
    ld      [i0],f2
    deccc   i1
    fmuls   f31,f0,f1
    add     i3,i5,i3
L:  st      f2,[i0]
    add     i0,i5,i0
    fadds   f2,f1,f2
    ld      [i3],f0
    ld      [i0],f2
    deccc   i1
    fmuls   f31,f0,f1
    add     i3,i5,i3
    bpos    L
    nop
    st      f2,[i0]
    add     i0,i5,i0
    fadds   f2,f1,f2
    bpos    L
    st      f2,[i0]
    add     i0,i5,i0
    bpos    L
```

a)

```
    ld      [i3],f0
    ld      [i0],f2
    deccc   i1
    fmuls   f31,f0,f1
    add     i3,i5,i3
    fadds   f2,f1,f2
    ld      [i3],f0
    ld      [i0+i5],f3
    deccc   i1
    fmuls   f31,f0,f1
    add     i3,i5,i3
L:  st      f2,[i0]
    add     i0,i5,i0
    fadds   f3,f1,f2
    ld      [i3],f0
    ld      [i0+i5],f3
    deccc   i1
    fmuls   f31,f0,f1

    bpos    L
    add     i3,i5,i3
    st      f2,[i0]
    add     i0,i5,i0
    fadds   f3,f1,f2

    st      f2,[i0]
    add     i0,i5,i0
```

b)

图17-26 a) 流水循环的结果指令序列，b) 执行寄存器重命名（见17.4.5节）及调整加载指令中的地址后的结果

产生这种流水指令在概念上是简单的。图17-27给出了在展开的循环体中寻找重复模式（它将成为要流水循环的流水循环体）的ICAN算法。我们再次假定循环体是单个基本块LBlock[i][1..ninsts[i]]。例程Unroll(nblocks, *i*, ninsts, LBlock, *j*, *k*)产生循环体LBlock[i][1..ninsts[*i*]]的*j*个副本（关于unroll()的算法，参见图17-30）。Schedule(*i*, *D*, *R*, *DS*, *DP*, *EX*)是图17-6给出的基本块调度器。Insts(*S*, *i*)表示在指令调度*S*中时间步*i*期间执行的指令序列，State(*S*, *i*)表示执行了从1到*i*－1个迭代后，在调度*S*中与时间

557

bbs.theithome.com

步i对应的资源状态。资源状态可以如9.2节所述用一些资源向量来模拟。该算法根据循环的展开大小并入了一个停止条件（SizeLimit）来约束查找空间。

```
Sched: sequence of Node

procedure Pipeline_Schedule(i,nblocke,ninsts,LBlock,SizeLimit,
   ExecTime) returns integer × integer
   i, nblocks, SizeLimit: in integer
   ninsts: in array [1··nblocks] of integer
   LBlock: in array [1··nblocks] of array [··] LIRInst
   ExecTime: in integer ⟶ integer
begin
   j, k: integer
   Insts: (sequence of integer) × integer ⟶ sequence
      of Instruction
   Dag: DAG
   ii: integer × array [··] of LIRInst
   for j := 2 to SizeLimit do
      ii := Unroll(nblocks,i,ninsts,LBlock,j,nil)
      Dag:= Build_DAG(ii↓1,ii↓2)
      Schedule(|Dag.Nodes|,Dag,Dag.Roots,DagSucc,DagPred,ExecTime)
      for k := j - 1 by -1 to 1 do
         if Insts(Sched,k) = Insts(Sched,j)
            & State(Sched,k) = State(Sched,j) then
            return ⟨k,j⟩
         fi
      od
   od
   return ⟨i,j⟩
end   || Pipeline_Schedule
```

图17-27 寻找可形成一个软流水的重复模式的算法

当这个算法执行成功后，在时间步k到$j-1$中的指令是形成软流水循环体的基本指令，第1到$k-1$步中的指令用于填充部分，而在第j步以后是排空部分的指令。当然，构造最后的填充部分、流水循环体和排空部分需要调整代码以适应资源限制，就像我们前面的例子所做的那样。

558 调度算法返回一对整数 $\langle i, j\rangle$，其中i是原循环体中的指令条数，j是循环体的展开因子。

17.4.3 循环展开

循环展开（loop unrolling）用循环体的若干副本替代一个循环的循环体，并相应地调整循环控制代码。副本的个数称为*展开因子*（unrolling factor），原循环称为*卷起的循环*（rolled loop）。在后面小节中，我们将讨论变量扩张，这是一种能够改善循环展开和软流水效果的转换。

循环展开能够降低一个使用索引的循环的执行开销，并且可以改善其他优化，如公共子表达式删除、归纳变量优化、指令调度和软流水的效果。

例如，图17-28b中的代码是图17-28a中循环的一个展开因子为2的展开版本。这个展开的循环所执行的循环关闭测试以及分支指令的次数是原循环的一半，并且通过使得每一个迭代能够调度，比如两条取a[i]之值的指令，从而增强指令调度的效果。另一方面，展开的版本大于卷起的版本，因此它有可能影响指令高速缓存的作用，并由此可能抵消由循环展开得到的好处。这种考虑表明，我们在决定展开哪些循环以及采用什么样的展开因子时需要小心权衡。

bbs.theithome.com

```
for i := 1 to 100 do
    s := s + a[i]
```

a)

```
for i := 1 by 2 to 99 do begin
    s := s + a[i]
    s := s + a[i+1]
end
```

b)

图17-28 a) 一个Pascal循环，b) 以展开因子2展开它的结果

另外应注意，例子中给出的循环展开转换是过度简化了的：我们假定循环上下界是已知的常数，并且展开因子能除尽迭代次数。一般而言，这些条件当然不一定满足，但是，具有一般迭代上下界的循环仍然能够展开。我们需要做的是保存这种循环的一个卷起的版本，当剩余的迭代次数小于展开因子时便从展开的循环退出，然后使用卷起的版本执行剩余的迭代。我们采用这种方法而不是在循环的每一个展开的副本中测试是否需要提前结束，是因为展开循环的目的之一是为了使得指令调度的范围更广泛，尤其是使得它可以跨循环体的各个副本选择指令，但是当在这些副本之间存在条件控制流的情况下，则不能有效地达到此目的。

图17-29a给出了一个Fortran 77循环，图17-29b展示了将它以展开因子*u*展开，并带有卷起的副本的结果。当然，循环展开转换也可以推广到那种循环计数器的增量值不是1的循环。

```
do i = lo,hi
    body
enddo
```

a)

```
do i = lo,hi-u+1,u
    body
    body/i+1/i
    . . .
    body/i+u-1/i
enddo
do i = i,hi
    body
enddo
```

b)

图17-29 a) 一个更一般的Fortran 77循环，b) 以展开因子*u*展开它的结果。
记号*body/ i/ j*表示循环体*body*中用*i*替代*j*

实现这种方法的ICAN代码在图17-30中给出。调用Unroll(nblocks, *m*, ninsts, Block, *factor*, *init*)将展开Block[m][1..ninsts[*m*]]为*factor*个副本，并且当不能确定迭代次数是否是展开因子的倍数时，同时还创建它的一个卷起的副本。它所处理的指令是MIRInsts和LIRInsts，代码中的Instruction指的只是我们的三种中间代码中的这两种。注意，该算法处理的循环仅含一个基本块，并且循环的控制具有一种特殊类型：它假定var1仅由Block[m][ninsts[m]-2]修改，并且 559

```
Block[m][1] = ⟨kind:label,lbl:"L1"⟩
Block[m][ninsts[m]-2] = ⟨kind:binasgn,left:var1,opr:add,
                        opd1:⟨kind:var,val:var1⟩,opd2:incr⟩
Block[m][ninsts[m]-1] = ⟨kind:binasgn,left:var3,opr:less,
                        opd1:⟨kind:var,val:var1⟩,opd2:final⟩
Block[m][ninsts[m]] = ⟨kind:valif,opd:⟨kind:var,val:var3⟩,
                      lbl:"L1"⟩
```

但不难将它推广到更广泛的情形。算法使用了如下几个函数：

1. Constant(*opd*)返回true，如果操作数*opd*是常数；否则返回false。
2. new_tmp()返回一个新的临时变量名。
3. new_label()返回一个新的标号。

bbs.theithome.com

图17-30中的代码没有包括含有多个基本块的循环的展开处理——因为为了区别出现在循环体每一个副本中的标号，需要对这些标号重新命名。

```
procedure Unroll(nblocks,m,ninsts,Block,factor,init)
   returns integer × array [··] of Instruction
   m, nblocks, factor, init: in integer
   ninsts: in array [1··nblocks] of integer
   Block: in array [1··nblocks] of array [··] of Instruction
begin
   UInst: array [··] of Instruction
   tj: Var
   lj: Label
   i, j, div: integer
   UInst[1] := Block[m][1]
   for i := 2 to ninsts[m]-2 do
      for j := 1 to factor do
         UInst[(j-1)*(ninsts[m]-3)+i] := Block[m][i]
      od
   od
   tj := new_tmp( )
   UInst[factor*(ninsts[m]-3)+2] := Block[m][ninsts[m]-1]
   UInst[factor*(ninsts[m]-3)+3] := Block[m][ninsts[m]]
   div := factor*Block[m][ninsts[m]-2].opd2.val
   if Constant(init) & Constant(Block[m][ninsts[m]-2].opd2)
      & Constant(Block[m][ninsts[m]-1].opd2)
      & (Block[m][ninsts[m]-1].opd2.val-init+1)%div = 0 then
      return ⟨factor*(ninsts[m]-3)+3,UInst⟩
   else
      lj := new_label( )
      UInst[factor*(ninsts[m]-3)+4] := ⟨kind:label,lbl:lj⟩
      for i := 2 to ninsts[m]-1 do
         UInst[factor*(ninsts[m]-3)+i+3] := Block[m][i]
      od
      UInst[factor*(ninsts[m]-3)+i+3] :=
         ⟨kind:valif,opd:Block[m][ninste[m]].opd,lbl:lj⟩
      return ⟨(factor+1)*(ninsts[m]-3)+6,UInst⟩
   fi
end    || Unroll
```

图17-30 对特定循环控制模式实现循环展开的ICAN代码

如前面曾提请注意的，在循环展开中最重要的问题是判别哪些循环可展开，以及展开因子是什么。这与两个方面有关：一方面与体系结构特征有关，例如，有效的寄存器个数、操作之间的有效重叠执行情况，如浮点操作与存储引用的重叠执行，以及指令高速缓存的大小和组织结构；另一方面与程序中要展开的具体循环的选择，以及它们使用的展开因子有关。体系结构特征的某些影响最好通过实验来决定，这种结果一般是启发式的。但是，通过以剖面分析方式运行含有那些循环的程序并从中获得反馈信息，对循环展开做出判定能有相当大的帮助。

这类实验的一组结果可以作为判定什么样的循环是可展开循环的规则，可展开的循环可能与如下类型的循环特征相关：

1. 只包含一个基本块的循环（即直线型代码）；
2. 含有某种平衡的浮点和存储器操作，或某种平衡的整数存储器操作的循环；
3. 生成少量中间代码指令的循环；
4. 循环控制简单的循环。

560 ~ 561

bbs.theithome.com

超星公司制作 请尊重作者版权

第1条和第2条标准限制只展开那种最可能从指令调度获益的循环。第3条保证被展开的代码块较短，而不至于对高速缓存的性能造成负面影响。最后一条标准阻止编译器展开那种难于判定应在什么时候出口并执行它的剩余迭代的卷起副本的循环，例如那种对链表进行遍历的循环。展开因子可能都从2开始，这取决于循环体的具体内容，但一般不会大于4，并且几乎决不会大于8，尽管将来VLIW机器的发展可能为使用较大的展开因子提供更好的支持。

最好是给程序员提供有关的编译选项或编译指示，以便他们可以指明哪些循环可以展开，以及它们使用什么展开因子。

循环展开一般会增加有效的指令级并行性，尤其是在对循环体的这些副本也同时执行其他一些转换来删除不必要依赖的情况下。这些转换包括软件寄存器重命名（参见17.4.5节）、变量扩张（参见下一节）及指令归并（参见18.12节）。利用依赖关系测试来消除存储地址之间潜在的别名也能删除依赖关系。因此，循环展开对大部分处理机实现，尤其是超标量和VLIW实现，具有相当重要的好处。

指令归并能使重命名有更多可用的寄存器。作为指令归并的例子，考虑下面的代码序列

```
r2 ← r1 + 1
r3 ← r2 * 2
```

假设我们为一台具有移位加指令的机器（如PA-RISC）生成代码，并且假定r2在这之后是死的，则可以将它转变成

```
r3 ← 2 * r1 + 2
```

从而使得r2成为自由可用的。

17.4.4 变量扩张

在展开因子为n的已展开循环体中的变量扩张（variable expansion）选择这种变量：它可以扩张成n个独立的副本，其中每个对应于一个循环体副本，并且所有副本在循环出口处可以被归并，以产生原来的变量应当具有的值。这种扩张具有我们所希望的性质，即它能够降低循环中依赖关系的数量，从而使得指令调度器对它更有效果。

比较容易检测出来的可进行扩张的变量包括累加器、归纳变量和搜索变量。图17-31a给出了这三种类型变量的例子，图17-31b给出了以展开因子2展开该循环，并对其中的累加器、归纳变量和搜索变量进行扩张后的结果。注意，有些归纳变量（数组元素的地址）没有扩张，因为这个例子是HIR代码；不过，它们也应服从于变量扩张。 562

这三种类型变量的扩张算法大体上是类似的，我们只讨论累加器变量扩张方法，而将修改此方法以适应其他两种类型变量的任务留给读者完成。

为了确定一个变量是否是加法累加器（乘法累加器也有可能，但不常见），我们要求它是在循环体中仅由加法或减法指令使用和定值的变量，并且展开的循环体含有这种指令的n个副本，每个副本对应一个循环体，其中n是展开因子。一旦确定出一个变量是加法累加器，我们就用一些新的临时变量替代它的第2到第n个副本，同时在循环入口处将所有新临时变量初始化为0，并在循环出口处将这些临时变量加到原来的变量之上。

在图17-31的例子中，acc是一个累加器。在展开的版本中，在循环体的第2个副本内它已被acc1所替代；同时acc1在进入该循环之前已初始化为0，并且在循环出口增加了一条将它加至acc的加法指令。

bbs.theithome.com

```
acc ← 10

max ← 0

imax ← 0


for i ← 1 to 100 do
    acc ← acc + a[i]*b[i]
    if a[i] > max then
        max ← a[i]
        imax ← i
    endif




endfor
```

a)

```
acc ← 10
acc1 ← 0
max ← 0
max1 ← 0
imax ← 0
imax1 ← 0
i1 ← 2
for i ← 1 by 2 to 99 do
    acc ← acc + a[i]*b[i]
    if a[i] > max then
        max ← a[i]
        imax ← i
    endif
    acc1 ← acc1 + a[i1]*b[i1]
    if a[i1] > max1 then
        max1 ← a[i1]
        imax1 ← i1
    endif
    i1 ← i1 + 2
endfor
acc ← acc + acc1
if max1 > max then
    max ← max1
    imax ← imax1
endif
```

b)

图17-31 a) 一个含累加器(acc)、归纳变量(i)和搜索变量(max和imax)的HIR循环，
b) 展开因子为2，并且扩张了累加器、归纳变量和搜索变量后的展开循环

563

17.4.5 寄存器重命名

寄存器重命名（register renaming）是一种可增加代码调度灵活性的转换。它可以施加于低级代码（按我们的情形，即LIR代码），并通过用另外的寄存器替代某些寄存器的使用，来删除指令之间由于使用相同的寄存器而导致的不必要的依赖关系。例如，在图17-32a所示的基本块中，所有4条指令都使用了f1。假如同时还有f1在基本块的出口时是活跃的这一信息，则将使得这些指令根本不可能重排。如图17-32b所示，如果用f27替换第1条和第2条指令中的f1（假设f27在那一点不是活跃的），则使得这些指令能够按任意顺序被调度，只要它们保持了f27和f1的定值出现在各自的使用之前，例如图17-32c所示的一种顺序。

```
1    f1 ← f2 + 1.0
2    [fp+52] ← f1
3    f1 ← f3 * 2.0
4    [fp+40] ← f1
```

a)

```
f27 ← f2 + 1.0
[fp+52] ← f27
f1 ← f3 * 2.0
[fp+40] ← f1
```

b)

```
f1 ← f3 * 2.0
f27 ← f2 + 1.0
[fp+40] ← f1
[fp+52] ← f27
```

c)

图17-32 a) 可以施加寄存器重命名的LIR代码例子，b) 执行寄存器重命名后的结果，
c) 由于重命名而使其成为可能的一种可选指令调度

我们假定所有寄存器都是相同类型的，以此略微简化关于重命名的讨论。对于更为实际的整数和浮点寄存器集合分开的情况，我们需要应用这个算法两次，每种寄存器集合应用一次。

给定了由指令LBlock[i][1..n]组成的基本块i，我们首先需要确定下面几个寄存器集合：

bbs.theithome.com

1. Regs是体系结构提供的所有通用寄存器的集合。

2. RegsUsed(*i*)是在基本块*i*中使用或/和定值的寄存器集合。

3. RegsLiveEntry(*i*)是在基本块*i*入口处活跃的寄存器集合。

4. RegsLiveExit(*i*)是在基本块*i*出口处活跃的寄存器集合。

5. DefsLiveExit(*i*)是它所定值的寄存器在基本块*i*出口处活跃的指令索引集合。

这些集合可采用检测和数据流分析相结合的方法来确定。于是，基本块*i*中可用于重命名的寄存器集合AvailRegs是

```
AvailRegs=Regs-(RegsUsed(i)URegsLiveEntry(i) URegsLiveExit(i))
```

下面，对于RegsUsed(*i*)中的每一个寄存器*r*，我们计算DefUses(*r*)，它是由这种偶对<*j*, *s*>组成的最大序列，其中*j*是基本块*i*中定值*r*的那条指令的索引，*s*是基本块*i*中使用指令*j*所计算的*r*的值的那些指令的索引集合；此偶对序列的元素按定值顺序排序。因此，只要在AvailRegs 564
中还有寄存器，并且在DefUses(r)中有其定值在基本块的出口处已死去的偶对，我们就用从AvailRegs中选择的寄存器替代在定值和使用*r*的指令中寄存器*r*的出现。

对于图17-32a中基本块的例子（称它为A），这些集合如下:

```
Regs              = {f0,f1,...,f31}
RegsUsed(A)       = {f1,f2,f3}
RegsLiveEntry(A)  = {f2,f3}
RegsLiveExit(A)   = {f1}
DefsLiveExit(A)   = {3}
AvailRegs         = {f0,f4,...,f31}
DefUses(f1)       = [<1,{2}>,<3,{4}>]
```

因此图a）中第1行f1的定值和第2行f1的使用被从AvailRegs中随机选择的寄存器f27所替代，结果如图b）所示。

图17-33给出了对基本块内寄存器进行重命名的ICAN代码。我们假定已经预先计算好了Regs、RegsUsed()、RegsLiveEntry()、RegsLiveExit()及DefsLiveExit()。代码中用到了下面一些过程:

1. Defines(*inst*, *r*)返回true，如果指令*inst*定义寄存器*r*；否则返回false。

2. Uses(*inst*, *r*)返回true，如果指令*inst*使用寄存器*r*作为操作数；否则返回false。

3. replace_result(*i*, LBlock, *m*, *ar*)用*ar*替代指令LBlock[*m*][*i*]中的结果寄存器。

4. replace_operands(*i*, LBlock, *m*, *r*, *ar*)用*ar*替代*r*在指令LBlock[*m*][*i*]中作为操作数寄存器的所有出现。

注意，如果寄存器*r*属于RegsLiveEntry(*m*)但不属于RegsLiveExit(*m*)，则*r*在基本块中最后一次使用之后就成为了可用于重命名的寄存器；我们在代码中没有考虑这种情况。

寄存器重命名也可以修改成适应扩展基本块，只要我们对每一个出口基本块，保证在其出口是活跃的每一个寄存器的最后定值不会改变即可。例如，图17-34的代码中，已知f1在从B3的出口处是活跃的，但从B2的出口处不是活跃的，我们必须保存f1在基本块B1中的定值，但可以在B2中重命名f1。

bbs.theithome.com

```
Regs: set of Register
RegsUsed, RegsLiveEntry, RegsLiveExit: integer → set of Register
DefsLiveExit: integer → set of integer

procedure Register_Rename(m,ninsts,LBlock)
   m: in integer
   ninsts: in array [1··m] of integer
   LBlock: inout array [··] of array [··] of LIRInst
begin
   AvailRegs: set of Register
   r, ar: Register
   def, i, j: integer
   uses: set of integer
   DefUses: Register → sequence of (integer × set of integer)
   AvailRegs := Regs - (RegsUsed(m) ∪ RegsLiveEntry(m) ∪ RegsLiveExit(m))
   if AvailRegs = ∅ then
      return
   fi
   for each r ∈ RegsUsed(m) do
      DefUses(r) := []
      i := 1
      while i ≤ ninsts[m] do
         if Defines(LBlock[m][i],r) then
            def := i; uses := ∅
            for j := i+1 to ninsts[m] do
               if Uses(LBlock[m][j],r) then
                  uses ∪= {j}
               fi
               if Defines(LBlock[m][j],r) then
                  i := j
                  goto L1
               fi
L1:         od
         fi
         DefUses(r) ⊕= [⟨def,uses⟩]
         i += 1
      od
   od
   for each r ∈ RegsUsed(m) do
      while |DefUses(r)| > 0 do
         def := (DefUses(r)↓1)@1
         uses := (DefUses(r)↓1)@2
         DefUses(r) ⊖= 1
         if def ∉ DefsLiveExit(m) then
            ar := ♦AvailRegs
            AvailRegs -= {ar}
            replace_result(def,LBlock,m,ar)
            for each i ∈ uses do
               replace_operands(i,LBlock,m,r,ar)
            od
         fi
      od
   od
end    || Register_Rename
```

图17-33 对基本块内寄存器重命名的ICAN代码

bbs.theithome.com

超星公司制作 请尊重作者版权

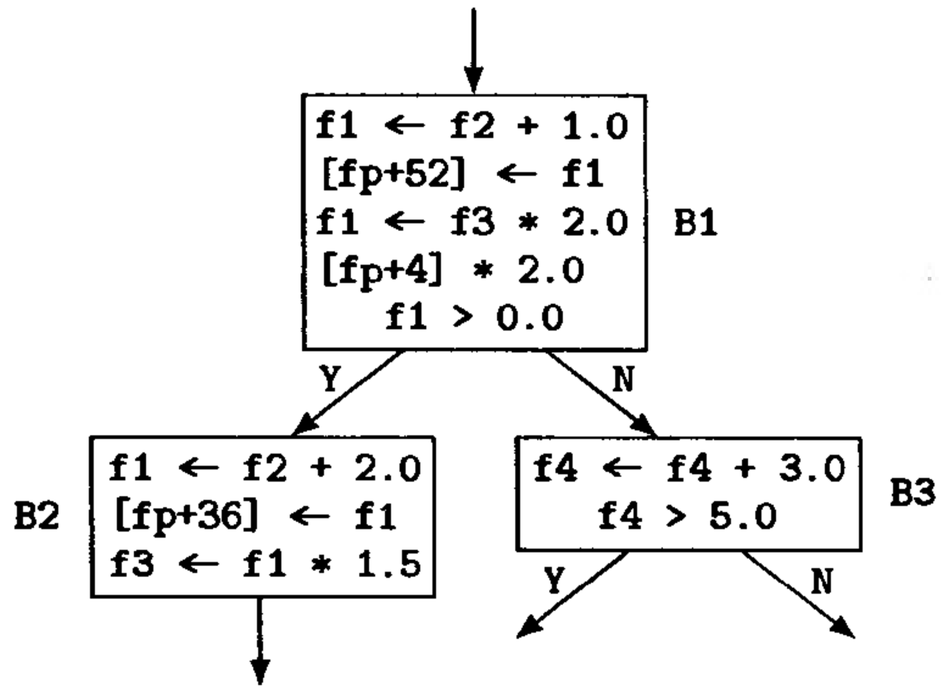


图17-34 扩展基本块寄存器重命名的例子

17.4.6 软流水的其他方法

软流水研究方面的最新进展表明，最近几年已提出了许多其他的方法。这些方法包括与我们的窗口调度方法类似的环调度（circular scheduling），以及和我们的展开-压实方法类似的各种方法，其中包括:

1. Lam用于VLIW的方法;
2. 最优循环并行化;
3. 完美流水;
4. Bodin和Charot的方法;
5. 带资源限制的软流水方法;
6. 分解式软流水。

然而确定什么是最好的方法，需要从两个方面考虑，即所得到的循环性能改善，和流水算法的效率，这仍然是一个活跃的研究领域（关于详细的引文，参见17.8节）。

Ruttenberg等人报告了一个令人感兴趣的实验。一种最优软流水算法叫做MOST，它是由McGill大学开发的，被用来替代MISP编译器的流水优化器，并进行了一系列的实验。这些实验将最优算法与MIPS的启发式算法相比较。在编译的所有SPEC92基准测试程序中，发现用最优算法生成的调度只有一个循环优于启发式算法，但是最优算法需要的时间是启发式算法使用时间的3 000倍。因此，软流水优化肯定值得包含在激进优化编译器中，并且启发式方法对大部分程序都能具有很好的效果。

565 ~ 567

17.4.7 层次归约

层次归约（hierarchical reduction）是由Lam（[Lam88]和[Lam90]）开发的对含有控制流结构的循环进行流水处理的一种技术。其方法由两部分组成，一部分处理条件，另一部分处理嵌套循环。为了处理嵌套循环，流水调度从最内层循环开始由里向外进行，随着每一个循环被调度，它将已调度过的循环归约成单个结点，同时在结点上附有该循环的调度结果和资源限制。

为了处理条件，首先分别进行then分支和else分支上的基本块调度，然后归约这个条件分支为单个结点，同样在结点上附着整个if-then-else的调度结果和资源限制。这样做的结果会导致较短的分支具有与较长分支相同的延迟。但它并不需要生成满足这种限制的代码——插入不必要nop仅仅浪费空间，但对调度没有改善。

作为对含有条件分支的循环施加软流水的例子，考虑图17-35a中的代码，假定我们编译的目标机与为图17-15中代码所假定的处理机相同。如果我们用窗口调度将循环中前两条指令

bbs.theithome.com

超星公司制作 请尊重作者版权

（带星号的两条ld）调度到循环体的末尾，并且移动这两条取数指令到条件转移的两个分支，则得到图17-35b中的代码。然后我们能够调度then和else分支，并且如果希望的话，可以做进一步的流水处理。

```
L1: ld    [r2],r3        *
    ld    [r1],f0        *
    cmp   r3,0
    ble   L2
    nop
    add   r4,1,r4
    fadds f2,f0,f3
                         *
                         *
    b     L3
    nop
L2: fsubs f3,f0,f3
    stf   f3,[r1]
    bg    L1
    nop
                         *
                         *
L3: sub   r2,4,r2
    sub   r1,4,r1
    cmp   r1,0


    bg    L1
    nop
```

a)

```
    L1:  ld    [r2],r3
         ld    [r1],f0
    L1a: cmp   r3,0
         ble   L2
         nop
         add   r4,1,r4
         fadds f2,f0,f3
         ld    [r2],r3
         ld    [r1],f0
         b     L3
         nop
    L2:  fsubs f3,f0,f3
         stf   f3,[r1]
         bg    L1a
         nop
         ld    [r2],r3
         ld    [r1],f0
    L3:  sub   r2,4,r2
         sub   r1,4,r1
         cmp   r1,4
         ld    [r2],r3
         ld    [r1],f0
         bg    L1a
         nop
         cmp   r3,0
         ble   L2a
         nop
         add   r4,1,r4
         b     L3a
         fadds f2,f0,f3
    L2a: fsubs f3,f0,f3
         stf   f3,[r1]
    L3a: sub   r2,4,r2
         sub   r1,4,r1
```

b)

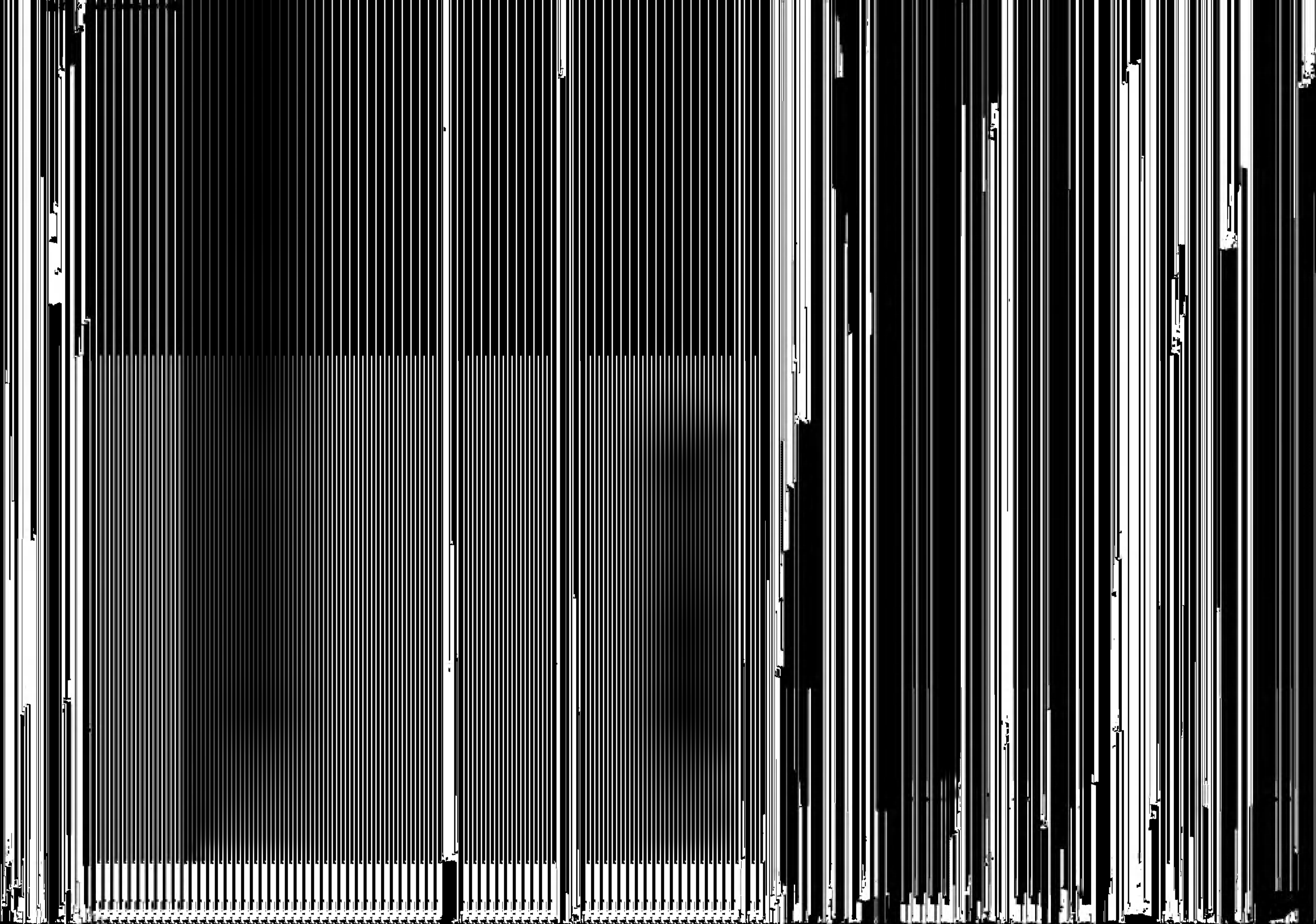
图17-35 a) 一个含条件转移的SPARC循环，b) 对它进行窗口调度并移动带星号的两条ld指令到此条件转移的两个分支后的结果

17.5 踪迹调度

踪迹调度是由Fisher [Fish81]开发的一种指令调度方法，这种方法比我们迄今讨论过的所有方法都更具进取性，它对于VLIW机器和发射宽度大于8的超标量实现特别有用。

踪迹（trace）是相对某种输入数据而执行的一个指令序列，该序列可含有分支，但不含循环。踪迹调度（trace scheduling）从执行频率最高的踪迹开始，用基本块调度方法（如17.1.2节所讨论的方法）来调度每一条踪迹上的所有指令。然后它根据需要在每一条踪迹的入口或出口加上补偿代码（compensation code）来校正由于乱序执行在踪迹的入口和出口处所产生的不一致性。在进行调度之前一般先将循环体展开若干次。这个调度和补偿过程一直重复直至所有

bbs.theithome.com



续树T，和一个当T为空时使用的延续C。(在图17-39中，类型PCGnode用于表示计算结点)。

渗透调度使用另外一种类型的依赖关系，叫做写－活跃依赖（write-live dependence），记做δ^{wl}，其定义如下。令v是计算结点，u是它的后继之一，如果v存在另外一个后继w，并且在边$v \rightarrow w$上存在某个活跃变量，而结点u写此活跃变量，则$v\ \delta^{wl}\ u$。变量在一条边上是活跃的，如果存在着一条从这条边开始通向并行计算结点图出口的路径，在此路径上存在着对该变量先于写的读操作。图17-38中，变量n导致了一个写-活跃依赖$v\ \delta^{wl}\ u$。

渗透调度由4种可作用于PCG的基本转换组成，这4种转换分别称为结点删除、操作移动、条件移动和统一。第一种转换删除已经变成空的结点；第二和第三种转换分别移动操作和条件到前驱结点；最后一种转换将一个结点的所有后继结点中相同的运算集合移动到该结点中。

这些作用于PCG的转换算法可以十分简单，也可以相当复杂。例如，我们可以构造4种谓词（分别是Delete_Node_Applic()、Move_Oper_Applic()、Move_Cond_Applic()和Unify_Applic()），测试是否可对一个以特定PCG结点作为根的子图施加这些转换，并构造实施这4种转换的4个过程（分别是Delete_Node()、Move_Oper()、Move_Cond()和Unify()）。给定这些谓词和过程，图17-39是一个简单的渗透调度程序。它尝试对每一个结点应用这4种转换，并重复这些转换直到没有结点为止。但是，这样做并不完全，因为不能保证对任意PCG，此算法会终止。还需要做的是将PCG划分成各个部分是DAG的分量，然后对这些分量施加此算法，这样才能保证算法终止。

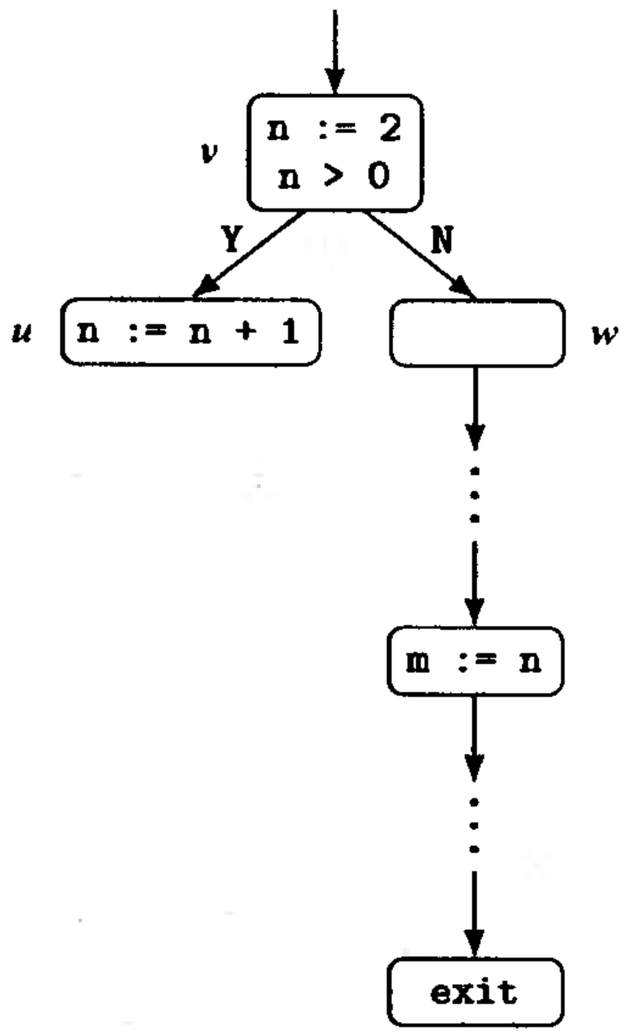


图17-38 写－活跃依赖的例子，即由于变量n，有$v\ \delta^{wl}\ u$

```
procedure Perc_Schedule_1(r)
   r: in PCGnode
begin
   Cand: set of PCGnode
   n: PCGnode
   again: boolean
   repeat
      again := false
      Cand := {r}
      for each n ∈ Cand do
         if Delete_Node_Applic(n) then
            Delete_Node(n)
            Cand ∪= Succ(n)
            again := true
         elif Move_Oper_Applic(n) then
            Move_Oper(n)
            Cand ∪= Succ(n)
            again := true
         elif Move_Cond_Applic(n) then
            Mqve_Cond(n)
            Cand ∪= Succ(n)
            again := true
         elif Unify_Applic(n) then
            Unify(n)
            Cand ∪= Succ(n)
            again := true
         fi
         Cand -= {n}
      od
   until !again
end   || Perc_Schedule_1
```

图17-39 一种简单的渗透调度算法

bbs.theithome.com

显然，渗透调度也能够与其他转换机制和其他类型的优化有益地结合在一起。例如，将一个操作移到一个结点中可能给公共子表达式删除带来机会。

Nicolau从这4种基本转换出发，定义了一系列的元转换，其中包括：(1) 一种称为migrate()的转换，这种转换在依赖关系允许的范围内，将一个操作尽可能地向PCG的上部移动；(2) 扩充这种技术以适应不可归约PCG，即，适应具有多个入口结点的循环；(3) 一种称为compact_path()的转换，这种转换对踪迹调度进行了推广，使得操作沿最高执行频率路径向上移动。

17.7 小结

这一章集中讨论了为改善性能而调度指令的有关方法，对于大多数机器和大多数程序而言，这些方法是一些最重要的优化。我们的讨论涵盖了分支调度、基本块调度、跨基本块调度、软流水（包括循环展开、变量扩张、寄存器重命名和层次归约）、前瞻与上推、踪迹调度及渗透调度。

在认识到几乎所有的调度问题，即使是它们最简单的形式至少都是NP困难的之后，我们介绍了在实际中几乎总是很好的几种启发式方法。

基本块和分支相结合的调度是所讨论的方法中最容易实现的一种；即使是这种方法，它也能使执行速度产生较大的改善，常常至少能改善10%。

跨基本块和前瞻调度实现起来较难一些，但由于它能在基本块之间移动指令，从而改善了基本块的构成。这些方法常常能更大地缩短执行时间，尤其是当硬件支持前瞻取指令时。

软流水（以及循环展开、变量扩张、寄存器重命名和层次归约）对循环体进行操作，它能使性能有更大的改善。有若干种不同的软流水方法，我们介绍了其中的两种，一种较易实现，另一种较难但通常更有效。但是，如17.4.6节列出的各种方法所显示的，哪一种方法是最好的目前还没有定论。

循环展开是一种重复循环体若干次，并相应调整循环控制代码的转换。如果编译器不包含软流水，我们建议在图17-40的C4框中死代码删除和代码提升之间进行循环展开。

变量扩张是一种辅助技术，它可以分离已展开的循环体的每个副本中那种将结果存放到一个特定变量的操作。它为每个副本提供一个这种变量，并且在循环结束之后合并它们的结果。寄存器重命名类似于变量扩张，它的作用也是消除那种本来不必使用相同寄存器的指令之间存在的依赖关系。

573

上面讨论的所有转换最好在接近优化（和编译）的最后阶段进行，因为为了更加有效，它们需要能够精确模拟目标机的机器代码形式或低级中间代码形式。

踪迹调度和渗透调度是两种全局代码调度方法，对于某些类型的程序，尤其是对数组进行运算的数值计算程序，以及某些体系结构，典型的是高度超标量和VLIW机器，它们都能获得非常大的好处。这两种优化方法最好构造成优化处理的驱动程序，即，我们可以围绕这两种类型的全局调度构建一个优化器，并通过它们来调用其他优化。

图17-40展示了推荐的优化顺序，黑体字标出了与调度有关的优化。我们在即将进行寄存器分配之前做指令调度（分支、基本块、跨基本块调度，以及软流水和它的辅助转换），并且，如果寄存器分配生成了溢出代码，则在寄存器分配之后重复一次分支调度和基本块调度。不包含软流水的编译器应当将循环展开和变量扩张连同其他循环优化一起，放置在编译的较早阶段。踪迹调度和渗透调度（如前面描述的）都需要一种完整的优化处理结构。

超星公司制作 请尊重作者版权

bbs.theithome.com

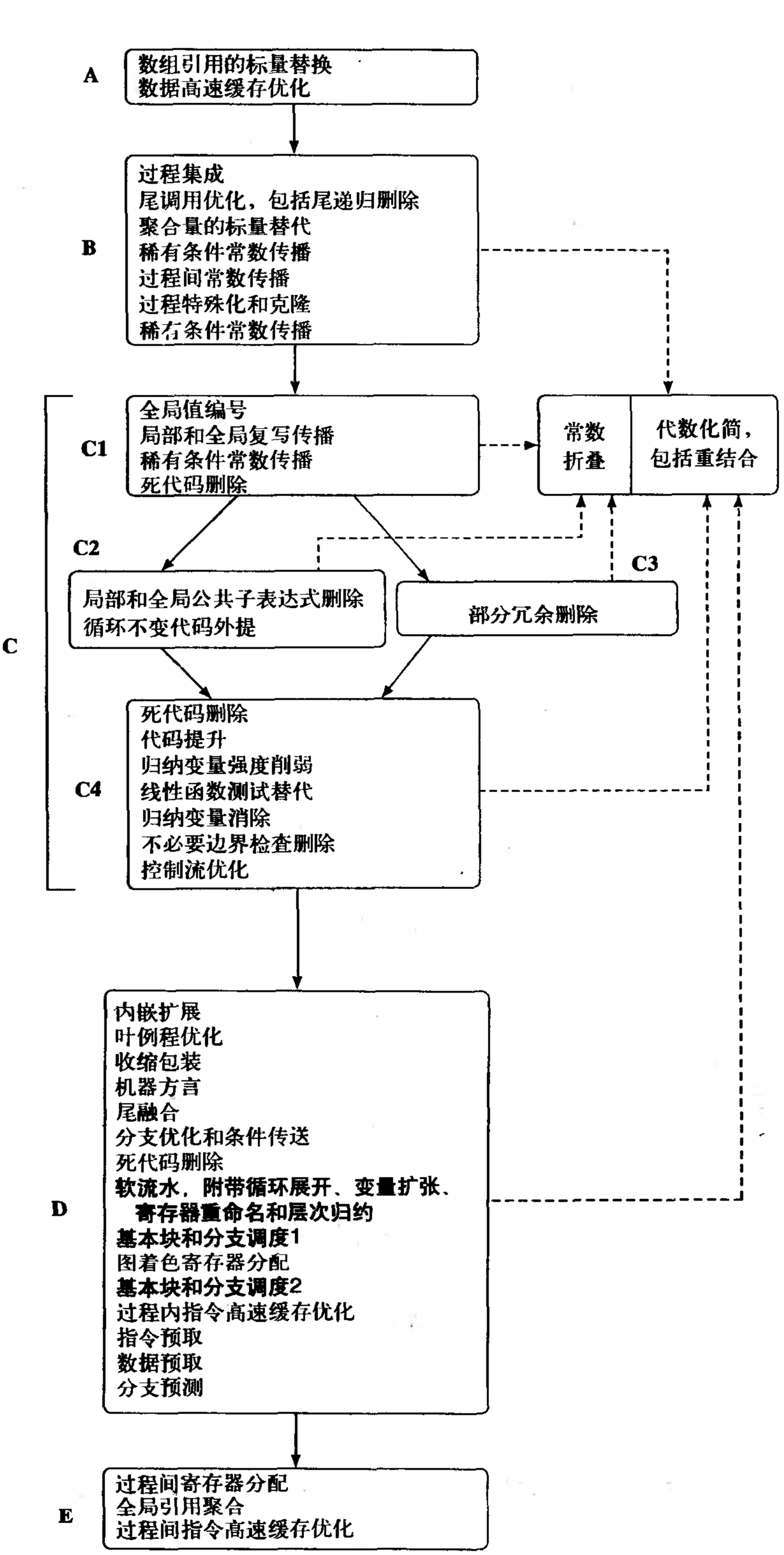


图17-40 优化顺序，黑体字标出了调度有关的优化

bbs.theithome.com

超星公司制作 请尊重作者版权

17.8 进一步阅读

Rymarczyk编写的关于最有效地利用IBM System/370流水处理机的汇编语言程序员指南是[Ryma82]。MIPS-X的描述见[Chow86]。

基本块调度是NP困难的问题，有关证明见[GarJ79]。使用表调度的基本块调度器有[GibM86]和[Warr90]中介绍的那些调度器。对于具有一个以上相同流水线的机器，表调度产生的调度不超过最优调度的一倍的证明是由Lawler等人在[LawL87]中给出的。

574 ~ 575

Smotherman等人[SmoK91]调查了DAG的类型和在一系列已实现的调度器中使用的启发式策略，其中包括由Gibbons和Muchnick[GibM86]、Krishnamurthy [Kris90]、Schlansker [Schl91]、Shieh和Papachristou [ShiP89]、Tiemann [Tiem89]和Warren [Warr90]等人开发的方法。Hennessy和Gross的机器级DAG的描述见[HenG83]。平衡式调度的描述见[KerE93]和[LoEg95]。Golumbic和Rainish关于有助于处理POWER分支延迟槽的转换是在[GolR90]中发现的。

不幸的是，对GNU的指令调度生成器没有非常好的介绍。[Tiem89]中介绍了一种调度，这种调度是GNU指令调度生成器的基础。对GNU的调度器生成器最好的描述是它的实现代码，即在GNU发布的 C编译器版本主目录中的文件`genattrtab.c`，以及在`config`子目录中的机器描述文件*machine*.md。

[Wall92]、[BerC92]和[MahR94]中有关于当前使用的跨基本块边界调度方法的讨论。[EbcG94]讨论了将为VLIW开发的调度技术应用于超标量RISC的方法。

前瞻取和如何在调度中利用它们的有关讨论见[RogL92]、[ColN87]、[WeaG94]和[Powe93]。

[EbcG94]介绍了前瞻调度和反前瞻调度。[GolR90]和[BerR91]描述了更早的前瞻调度方法。

环调度的描述见[Jain91]。与我们的展开－压实方法类似的一些流水线调度的介绍如下:

方 法	引 文
Lam的VLIW方法	[Lam88]
最优循环并行化	[AikN88a]和[Aike88]
完美流水	[AikN88b]和[Aike88]
Bodin和Charot的方法	[BodC90]
带资源约束的方法	[EisW92]和[AikN91]
分解式软流水	[WanE93]

MIPS编译器使用的软流水方法、McGill大学开发的最优流水算法MOST，以及关于它们的比较见[RutG96]。

Mahlke等人在[MahC92]中介绍了变量扩张。

层次归约的详细内容可以在[Lam88]和[Lam90]中找到。踪迹调度的介绍见[Fish81]和[Elli85]。首先介绍渗透调度的是[Nico86]。

17.9 练习

17.1 对于17.1.2节给出的每一种启发式，写出一个只应用该启发式便能提供最好调度的LIR指令序列，和一个不能提供最好调度的LIR指令序列。

17.2 17.4.3节间接提到了对展开－压实软流水器所产生的代码需要做的某些修改。这些修改是什么？为什么需要它们?

17.3 给出一个使用层次归约对含有内层循环和`if-then-else`结构的循环进行流水化

bbs.theithome.com

的例子。

17.4 给出一个由LIR代码组成的基本块的例子，用以说明寄存器重命名使得调度产生的代码的执行时间至少降低了三分之一。

17.5 修改图17-6的表调度算法，用它来调度一种发射宽度为3的超标量实现处理机的代码，这种处理机有两个整数流水线和一个浮点流水线。

ADV 17.6 推广前一练习的算法，使它能调度发射宽度为n的超标量处理机的代码和指令I_1，I_2, …, I_k；这种发射宽度为n的超标量处理机有m种类型为PClass(i)的处理器，其中每一类有n_i个，使得

$$\sum_{i=1}^{m} n_i = n$$

并且指令I_i能够在处理机j执行当且仅当IClass(I_i) = PClass(j)。设I_k有S_k拍的流出延迟和r_k拍的结果延迟，且这些延迟与指令之间的冲突无关。

576 ~ 577

17.7 给出对一个已展开循环中的(a)累加器和(b)搜索变量进行扩张的ICAN代码。

bbs.theithome.com

超星公司制作 请尊重作者版权

第18章　控制流和低级优化

本章的内容涉及作用于过程控制流的一些优化、死代码删除，以及其余一些最好在程序的低级中间代码上（例如LIR）或结构化形式的汇编语言上来执行的全局优化。

这些优化中有一些（如死代码删除）可以在编译期间多次执行并取得良好效果，因为有好几种转换都会导致死代码的产生。为了减少被编译过程的代码体积，值得迅速删除它们。

产生较长基本块的控制流转换能够潜在地增加指令级并行性，而指令级并行性对于超标量实现特别重要。

精确的分支预测也很重要，因为它增加了处理机从指令高速缓存、存储器或二级高速缓存中读取正确的连续路径的可能性。随着高速缓存的延迟与CPU主频差距的扩大，分支预测的重要性也在增加。

本章介绍的最后几种优化常常称为后遍优化或窥孔优化。第一个术语源于它们一般属于编译最后执行的几种优化，并且总是在代码生成之后进行。另一个术语指的是这样一个事实，它们之中的许多优化常常是通过在已生成的代码中，移动一个小小的窗口或一个小孔来寻找可施加对应优化的代码片段而实现的。

后面将讨论的有如下一些转换：

1. 不可到达代码的删除，它删除那些因为从入口没有路径可到达它，因而不可能被执行的基本块。

2. 伸直化，它通过用分支目的地的代码替代某种类型的分支指令而创建更长的基本块。

3. if化简，它删除if的无用分支或整个if结构。

4. 循环化简，它用直线代码替代某种空的或非常简单的循环。

5. 循环倒置，它用位于循环结尾处的测试替代位于循环开始的测试和位于循环结尾处的无条件分支。

6. 无开关化，它将循环不变条件提出到循环之外。

7. 分支优化，它用较简单的代码替代各种从一个分支到另一个分支的分支代码。

8. 尾融合，或交叉跳转，它将具有相同尾部代码的基本块转换为只有一个基本块保留这个尾部代码，而其他基本块则转移到这个基本块中保留的尾部代码处。

9. 条件传送，它用不包含分支的代码序列替代某些简单的if结构。

10. 死代码删除，它删除那些确实对计算结果不起作用的指令。

11. 分支预测，它涉及静态地或动态地预测一个条件分支指令是否会导致控制转移。

12. 机器方言和指令归并，它用较快的、完成相同功能的单条指令替代一条指令或一组指令。

18.1　不可到达代码的删除

不可到达代码（unreachable code）是无论输入数据是什么都不可能被执行的代码。它可能一开始就是对任何输入数据都不会执行的，或者是由于其他优化而导致成为这种状态的。删除不可到达代码对程序的执行速度没有直接影响，但显然能减少程序占用的空间，因此可能对程序的速度有间接的影响，尤其是它可能改善程序的指令高速缓存的利用率，从而对程序速度产

bbs.theithome.com

生影响。另外，删除不可到达代码也使得其他控制流转换，如伸直化（参见18.2节），有可能进行从而减少程序的执行时间。

注意，删除不可到达代码是有时容易被混淆的两种转换中的一种。另一种转换是死代码删除（参见18.10节），死代码删除去掉那种可执行、但对要计算的结果不起作用的代码。

为了识别和删除不可到达代码，我们假设有一个基本块表，并且对于每一个基本块，有其前驱和后继集合。我们的处理如下：

1. 设置again为false。

2. 对Block[]数组进行迭代，寻找满足这种条件的基本块：从entry块到达这些基本块
580 的路径集合为空。当找到这种基本块时便将它删除，同时调整其前驱和后继反映它已被删除，并设置again为true。

3. 若again为true，回到步骤1。

4. 调整Block和相关的数据结构，以便使所有的基本块都集中到Block数组的前部。

注意，如果对于不可到达基本块i，简单地设置Block[i]=nil是可接受的，则可以不需要上面的最后一步。

实现不可到达代码删除的例程Elim_Unreach_Code()的ICAN代码在图18-1中给出。这个算法用到了函数No_Path(i, j)和图4-17定义的函数delete_block(i, nblocks, ninsts, Block, Succ, Pred)。如果从基本块i到基本块j不存在路径，函数No_Path(i, j)返回true，否则返回false。

```
procedure Elim_Unreach_Code(en,nblocks,ninsts,Block,Pred,Succ)
   en: in integer
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [1··nblocks] of Instruction
   Pred, Succ: inout integer --> set of integer
begin
   again: boolean
   i: integer
   repeat
      again := false
      i := ♦Succ(en)
      while i ≤ nblocks do
         if No_Path(en,i) then
            ninsts[i] := 0
            Block[i] := nil
            again := true
            delete_block(i,nblocks,ninsts,Block,Succ,Pred)
         fi
         i += 1
      od
   until !again
end    || Elim_Unreach_Code
```

图18-1 删除不可到达代码的算法

图18-2a展示了一个MIR过程的流图，其中含有不可到达代码，即基本块B2、B3和B5。对于这个流图，nblocks的值是7，表18-1a给出了它的ninsts、Pred和Succ数据结构。在删除不可到达代码之后，nblocks的值是4，数据结构的值如表18-1b所示，结果流图如图18-2b
581 所示。

bbs.theithome.com

超星公司制作 请尊重作者版权

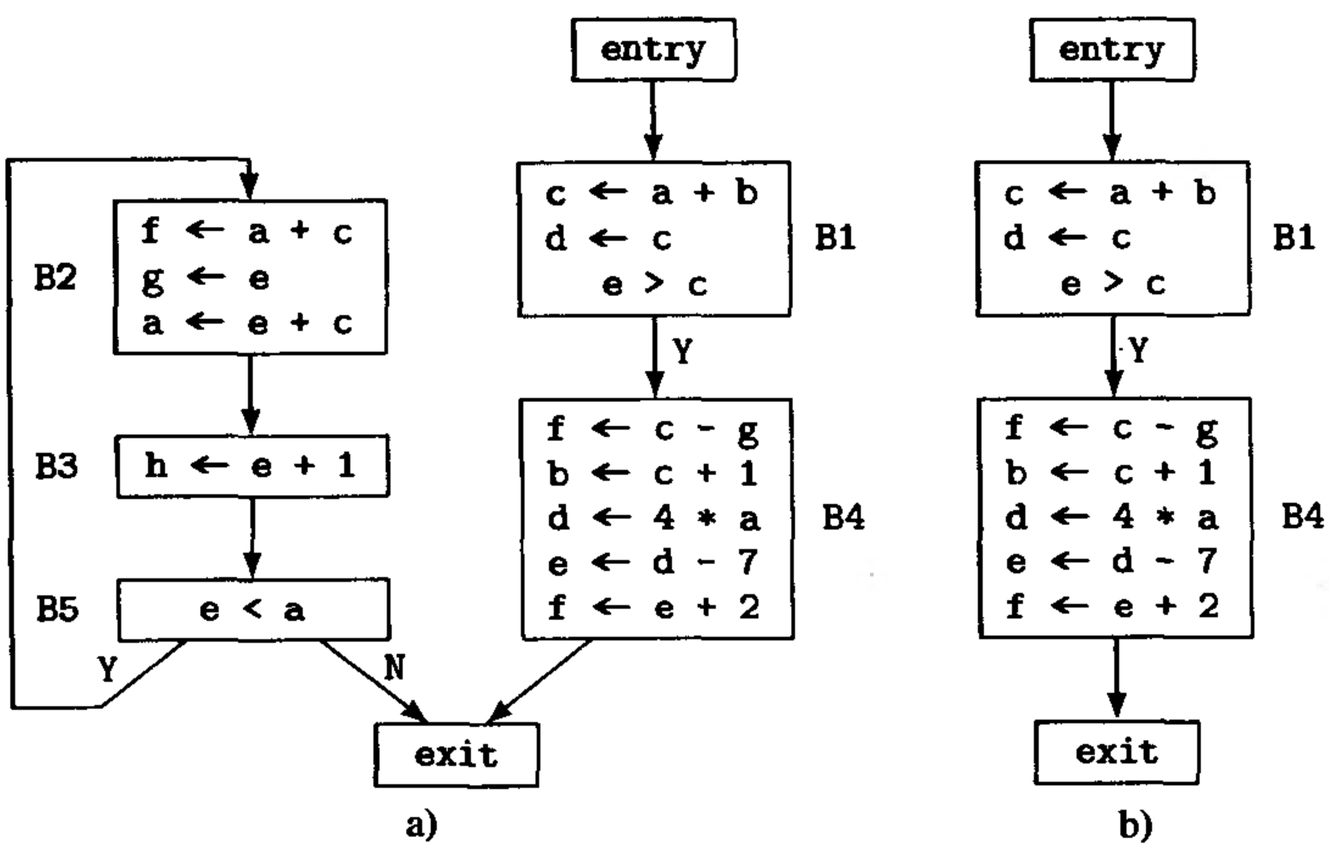


图18-2 我们的例子，删除不可到达代码a) 之前和b) 之后

表18-1 图18-2中流图的基本块数据结构

a)

i	ninsts[*i*]	Succ(*i*)	Pred(*i*)
entry	0	{B1}	∅
B1	3	{B4}	{entry}
B2	3	{B3}	{B5}
B3	1	{B5}	{B2}
B4	5	{exit}	{B1}
B5	1	{B2,exit}	{B3}
exit	0	∅	{B4,B5}

b)

i	ninsts[*i*]	Succ(*i*)	Pred(*i*)
entry	0	{B1}	∅
B1	3	{B4}	{entry}
B4	5	{exit}	{B1}
exit	0	∅	{B4}

582

18.2 伸直化

伸直化（straightening）是一种应用于这样两个基本块的优化，这两个基本块最基本的形式是，第一个基本块没有除第二个基本块之外的后继，第二个基本块没有除第一个基本块之外的前驱。由于它们两个都是基本块，因此要么第二个基本块紧跟在第一个基本块之后，要么第一个基本块结束时一定无条件转移到第二个基本块。在第一种情况下，转换不需要做什么事情，在第二种情况下，它用第二个基本块替换这个分支。图18-3给出了一个例子。基本块B1只有一个后继B2，而B2惟一的一个前驱是B1。伸直转换将它们合并成一个新的基本块B1a。

但是，这种转换并不像从流图上看起来那么简单。考虑图18-4a中的代码，其中，以L1和L2开始的两个基本块分别对应于B1和B2。第一个基本块的结尾是一个转移到L2的无条件分支指令，我们假定以L2开始的基本块没有其他的前驱。图18-4b中经转换后的代码已用以L2开始

bbs.theithome.com

超星公司制作 请尊重作者版权

的基本块的副本替换了goto L2，同时删除了这个基本块原来的副本。注意，由于这个被复制的基本块以条件转移结束，因此在它的末尾还需要有一条转移到原来基本块的下降分支的转移指令。显然，如果被复制的这个基本块也以无条件分支结尾，则伸直转换能得到最大的好处，因为它不需要一条新的转向下降分支的转移，并且还可以再次应用伸直转换。

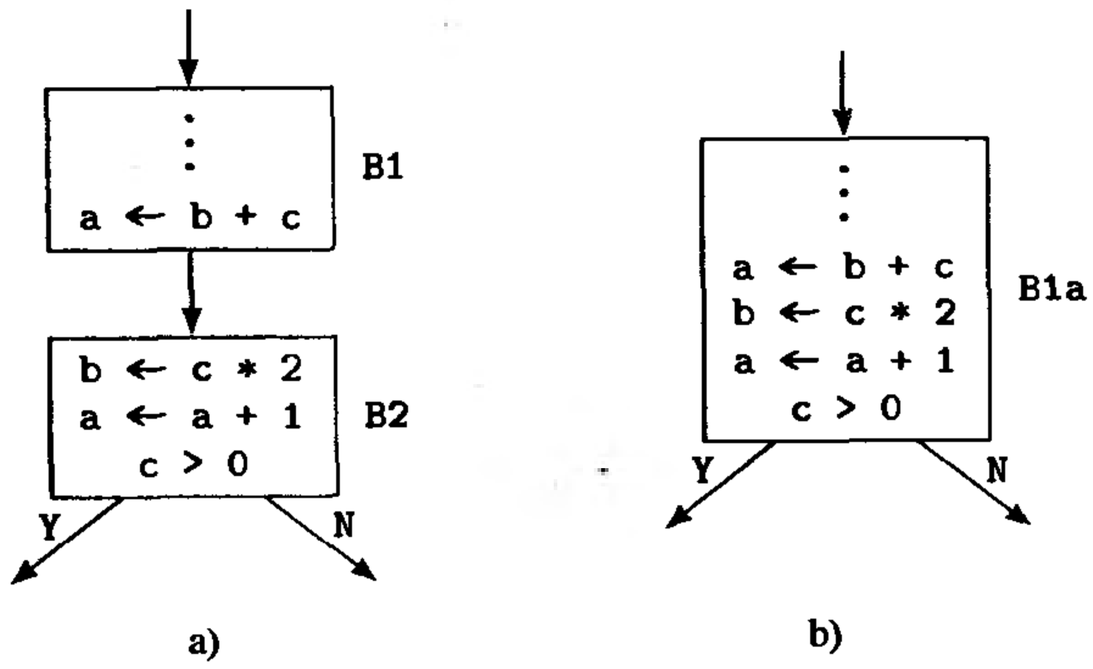


图18-3 a) 用于伸直转换的一个例子，b) 应用伸直转换后的结果

```
L1: . . .
    a ← b + c
    goto L2

L6: . . .
    goto L4
L2: b ← c * 2
    a ← a + 1
    if (c > 0) goto L3
L5: . . .
```

a)

```
L1: . . .
    a ← b + c
    b ← c * 2
    a ← a + 1
    if (c > 0) goto L3
    goto L5
L6: . . .
    goto L4

L5: . . .
```

b)

图18-4 图18-3中例子的MIR代码版本

伸直化可以推广到这样的一对基本块，其中第一个基本块的结尾是一个条件转移，第二个基本块没有其他前驱。如果已知第一个基本块的这个条件转移取一条路径的可能性比取另一条路径要大，则可以让这条可能性大的路径成为下降路径，并且需要的话可以颠倒所测试的条件。在多数体系结构中，这几乎总是比让可能性较小的路径作为下降路径要快。对于具有静态分支预测（参见18.11节）的体系结构（或实现），这要求我们预测的是下降分支而不是转移分支。

实现伸直转换的ICAN代码在图18-5中给出，其中用到的附属例程Fuse_Blocks()在图18-6中。调用Fall_Through(*j*, ninsts, Block)返回在执行基本块*j*之后，执行将下降到的那个基本块的标号；如果没有这种基本块，它补充一个并返回其值。

```
procedure Straighten(nblocks,ninsts,Block,Succ,Pred)
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array of [··] of LIRInst
   Succ, Pred: inout integer → set of integer
begin
   change := true: boolean
```

图18-5 对一对基本块执行伸直化处理的ICAN代码

bbs.theithome.com

超星公司制作 请尊重作者版权

```
    i, j: integer
    while change do
       change := false
       i := 1
       while i ≤ nblocks do
          if |Succ(i)| = 1 & Pred(♦Succ(i)) = {i} then
             j := ♦Succ(i)
             Fuse_Blocks(nblocks,ninsts,Block,Succ,Pred,i,j)
             change := true
          fi
          i += 1
       od
    od
end     || Straighten
```

图18-5 （续）

```
procedure Fuse_Blocks(nblocks,ninsts,Block,Succ,Pred,i,j)
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array of [··] of LIRInst
   Succ, Pred: inout integer → set of integer
   i, j: in integer
   begin
      k, l: integer
      label: Label
      if Block[i][ninsts[i]].kind = goto then
         k := ninsts[i] - 1
      else
         k := ninsts[i]
      fi
      if Block[j][1].kind ≠ label then
         k += 1
         Block[i][k] := Block[j][1]
      fi
      for l := 2 to ninsts[j] do
         k += 1
         Block[i][k] := Block[j][l]
      od
      if Block[i][k].kind ≠ goto then
         label := Fall_Through(j,ninsts,Block)
         ninsts[i] := k + 1
         Block[i][ninsts[i]] := ⟨kind:goto,lbl:label⟩
      fi
      ninsts[j] := 0
      Succ(i) := Succ(j)
      Elim_Unreach_Code(nblocks,ninsts,Block,Pred,Succ)
   end     || Fuse_Blocks
```

图18-6 Straighten()中用到的例程Fuse_Blocks()

18.3 if化简

if化简（if simplification）作用于两个分支均为空或一个分支为空的条件结构。这种条件结构可能是因为代码提升或其他优化导致的，也可能是一开始在源代码中就存在的，尤其是在那种由程序生成而不是人工编写的源代码中。本节介绍的三种if化简分别与分支为空的条件、常数值条件，以及在相依赖的两个条件分支中的公共子表达式有关。

bbs.theithome.com

如果一个if结构的then或else部分为空，对应的分支便可删除。如果一个if-then-else的then部分为空，则可以颠倒这个条件并删除与then部分有关的分支。若它的两个分支都为空，则可删除整个控制结构，只保留不能确定是否有活跃副作用的条件计算部分。

583 ~ 585

第二种控制流化简作用于条件为常数值的if分支。值为常数的条件自动地使得if的一个分支不会被执行，因而可以删除这个分支。如果这个if只有一个分支，则整个if结构都可以删除，只要所计算的条件没有活跃的副作用。

第三种控制流化简作用于这样的两个if分支，其中一个if的执行依赖于前一个if的执行，且前一个if中的条件是后一个if的条件中的公共子表达式；当然，在这两个测试之间必须没有改变它们所涉及的变量之值。图18-7举例说明了这种情形。在基本块B2末尾的测试“(a>d) or bool”肯定是满足的，因为如果我们进入了基本块B2，则表明在基本块B1中有a>d，并且在这之间a和d的值都没有改变。因此，基本块B2中的这个测试以及整个基本块B4都可以被删除。如这个例子的情况所示，这种化简也使得可以进一步进行伸直转换。

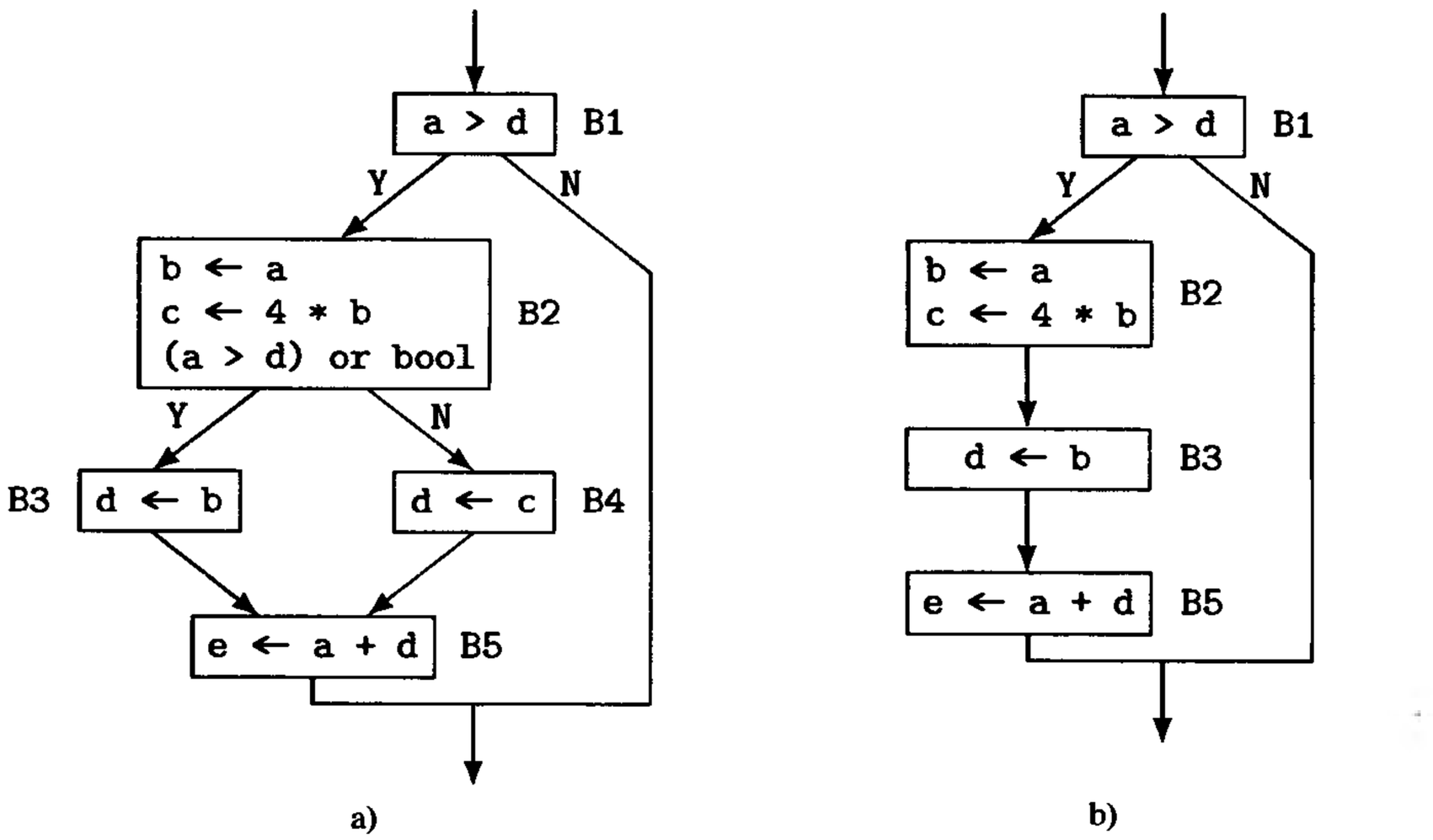


图18-7 a) 一个条件为公共子表达式的例子，和b) 删除此if的死分支后的结果

18.4 循环化简

循环体为空的循环可以删除，只要控制迭代的代码没有活跃的副作用。即使控制迭代的代码有副作用，它们也有可能很简单以至于可用非循环代码来替代。具体地，如果一个循环的循环控制变量只是简单地增加或减少一个常数，并与一个常数终值相比较，则可在编译时计算出此常数终值，并插入一条对此循环控制变量赋值的语句来替代这个循环。

前一节讨论的关于if的第二和第三种化简也可应用于循环和循环内的if嵌套，反之亦然。

586

可施加于循环的另一种化简是，有些循环的循环控制代码简单得足以能确定出循环的执行次数，如果执行次数足够少，则可将它展开成无分支的代码，并由此得到好处。在某些情况下，还可以如图18-8所示，导致结果代码可以在编译时就被执行。

bbs.theithome.com

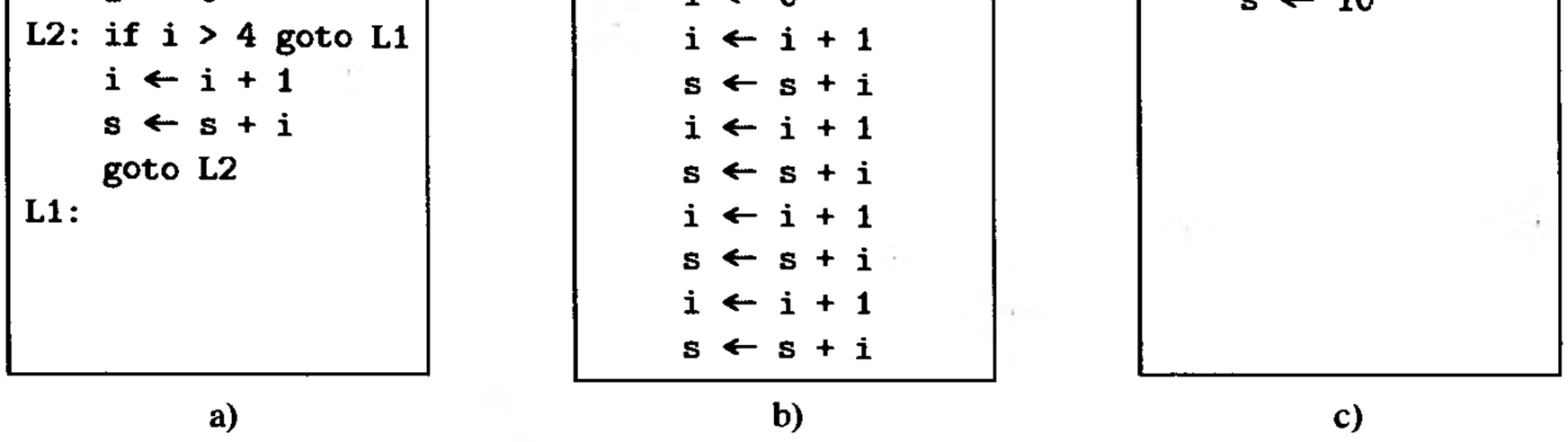


图18-8 a) 一个循环的例子，它可转换为b) 直线代码，和c) 可在编译时执行的代码

18.5 循环倒置

在源语言的术语中，*循环倒置*（loop inversion）是将一个`while`循环转换为`repeat`循环。换言之，它将循环关闭测试从循环体的前面移到循环体的后面。在最简单的形式中，循环倒置要求我们要么确定此循环一定会进入，即循环体至少被执行一次，因此这种指令移动是安全的；要么需要我们在进入这个循环之前生成一个测试，来确定是否会进入此循环。循环倒置的好处是，为了结束循环只需要执行一条分支指令，而不是两条分支指令，这两条分支指令中有一条从循环体末尾转向循环体的开始，另一条在循环体的开始执行测试。

确定一个循环是否会进入，可能只需简单地观测这个循环是否是具有常数循环控制值的Fortran 77的`do`循环或Pascal的`for`循环，并且是否终值超过了初值；也可能需要与循环控制表达式的值和/或迭代变量有关的数据流分析数据。循环倒置将一个在顶部有一条件分支，在底部有一无条件分支的循环转换为只在底部有一个与原来的测试相反的条件分支的循环。

图18-9用C代码展示了循环倒置的一个例子。在图a)中有一个C的`for`循环，在图b)中，我们已将它扩展成一个`while`循环。因为开始时它的终止条件为假，它的循环体至少会执行一次，因此它也可以转换为图c)所示的`repeat`循环。

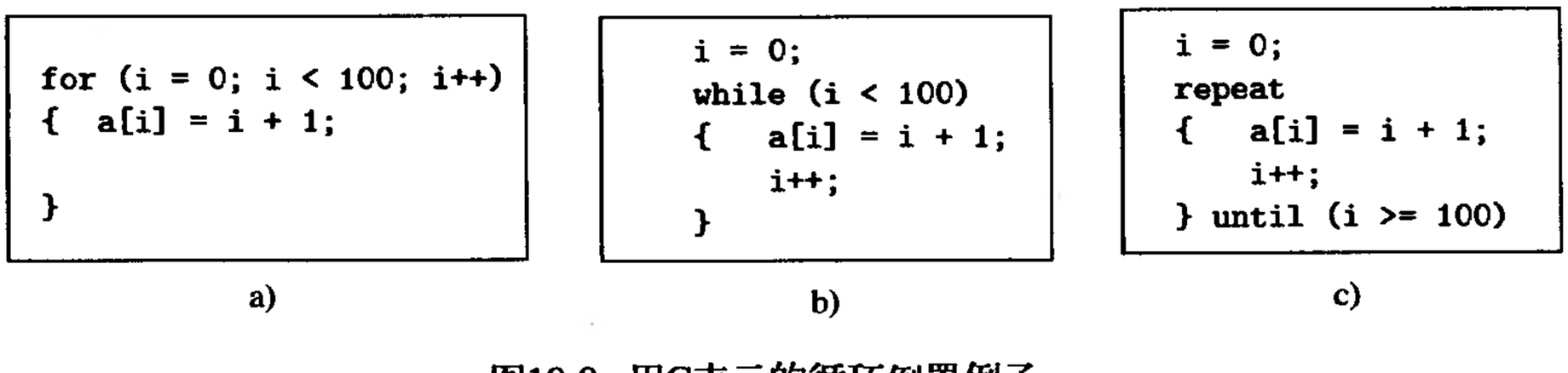


图18-9 用C表示的循环倒置例子

循环倒置的一种重要情形与这种嵌套循环有关，其中内层循环迭代控制变量的取值范围是外层循环迭代控制变量取值范围的非空子集，如图18-10中的Fortran 77代码所示。在这种情况下，只要外层循环被执行，内层循环肯定就至少会执行一次。因此对内层循环一定可以施行循环倒置。

587

如果不能确定一个循环是否会进入，我们可以在它的倒置形式前面放置一条检查进入循环的条件，并在条件不满足时跳过此循环的条件分支指令。图18-11给出了这种处理的一个例子。

bbs.theithome.com

```
do i = 1,n
   do j = 1,n
      a(i,j) = a(i,j) + c * b(i)
   enddo
enddo
```

```
for (i = a; i < b; i++)
{   c[i] = i + 1;

}
```

a)

```
   if (a >= b) goto L;
   i = a;
   repeat
   {   c[i] = i + 1;
       i++;
   }
   until (i >= b)
L:
```

b)

图18-10 具有相同循环控制值的Fortran 77嵌套循环例子

图18-11 倒置一个我们不能确定它是否一定会进入的C循环

18.6 无开关化

无开关化（unswitching）是一种控制流转换，它将循环不变条件分支提到循环之外。例如，
588 在图18-12a的Fortran 77代码中，谓词k.eq.2与i的值无关。将这个谓词测试从循环中移出而产生的代码如图18-12b所示。尽管这样做增加了代码的空间，但它减少了要执行的指令条数。注意，这个测试条件必须是紧嵌套在转换所施加的循环结构内的。图18-13的例子举例说明了无开关化的两个方面：第一，这个不变条件不必是整个谓词；第二，如果条件代码没有else部分，则需要在转换后的代码中补充else部分来设置循环控制变量为循环终值，除非可以证明循环控制变量在此循环之后是已死去的。

```
do i = 1,100
   if (k.eq.2) then
      a(i) = a(i) + 1
   else
      a(i) = a(i) - 1
   endif
enddo
```

a)

```
if (k.eq.2) then
   do i = 1,100
      a(i) = a(i) + 1
   enddo
else
   do i = 1,100
      a(i) = a(i) - 1
   enddo
endif
```

b)

图18-12 a) 循环内嵌套一个循环不变谓词的Fortran 77代码，和b) 对它执行无开关化后的结果

```
do i = 1,100
   if ((k.eq.2).and.(a(i).gt.0)) then
      a(i) = a(i) + 1
   endif
enddo
```

a)

```
      if (k.eq.2) then
         do i = 1,100
            if (a(i).gt.0) then
               a(i) = a(i) + 1
            endif
         enddo
      else
         i = 101
      endif
```

b)

图18-13 对无else部分的条件结构实施无开关化

18.7 分支优化

有若干种分支优化通常被推迟到编译处理过程的较后阶段，在代码的最终形态已确定之后

bbs.theithome.com

超星公司制作 请尊重作者版权

才进行。本节我们讨论其中的几种优化。

从一个分支指令转移到另一个分支指令的情况极常见，尤其是当采用简单的代码生成策略时。检测出这种情况并不难，我们只需简单地查看每一条分支指令的转移目的地是什么指令即可，但需要在填充分支延迟槽之前进行这种检查（对于有分支延迟槽的体系结构而言）。从一个分支指令转移到另一个分支指令的情形有下面几种: 589

1. 一个无条件分支指令，它转移到另一个无条件分支指令，可用一个转移到后者目的地的无条件分支指令来替代。

2. 一个条件分支指令，它转移到一个无条件分支指令，可用对应的转移到后一无条件分支指令的目的地的条件分支指令来替代。

3. 一个无条件分支指令，它转移到一个条件分支指令，可用后一条件分支指令的副本来替代。

4. 一个条件分支指令，它转移到另一个条件分支指令，只要前者的条件为真时后者的条件也为真，前一个条件分支指令就可用一个测试条件来自前者、转移目的地来自后者的分支指令来替代。

例如，在下面的MIR序列中:

```
    if a = 0 goto L1
    . . .
L1: if a >= 0 goto L2
    . . .
L2: . . .
```

第一条分支指令可改变为“`if a = 0 goto L2`”，因为如果a=0为真，就有a>=0为真。

另一种情形是这样的一种无条件分支指令，它的分支目的地就是下一条指令。这种分支指令常常出现在由标准的代码生成策略所生成的代码中。这种情形不难识别，其优化也很简单: 删除此分支即可。同样显然的是这种情况: 一个基本块以一个条件分支指令结束，而该分支指令分支到其后的第二条指令，且跟在此分支指令之后的是一个无条件转移到另外某处的分支指令。在这种情况下，我们可颠倒这个条件分支的测试条件，改变它的转移目的地为那条无条件分支指令的目的地，然后删除后者。例如，

```
    if a = 0 goto L1
    goto L2
L1: . . .
```

变为

```
    if a != 0 goto L2
L1: . . .
```

18.8 尾融合或交叉转移

尾融合（tail merging），也叫做*交叉转移*（cross jumping），是一种总能节省空间，并且也可能节省时间的优化。它寻找这样的两个基本块: 这两个基本块的最后几条指令是相同的，并且它们随后由于一个基本块转移到位于另一个基本块之后的指令，或由于两个基本块都转移到相同的位置，而在相同的地方汇合继续执行。这种优化所要做的是，在一个基本块中用一条转移至另一个基本块中对应点的分支指令来替代它与另一个基本块中相匹配的那些指令。显然，如果这两个基本块中有一个不是以分支指令结尾的，则最好保留这个基本块不变。例如，图18-14a中的代码可以转换为图b) 中的代码。 590

bbs.theithome.com

超星公司制作 请尊重作者版权

```
    . . .
    r1 ← r2 + r3
    r4 ← r3 shl 2
    r2 ← r2 + 1
    r2 ← r4 - r2
    goto L1
    . . .
    r5 ← r4 - 6
    r4 ← r3 shl 2
    r2 ← r2 + 1
    r2 ← r4 - r2
L1:
```

a)

```
    . . .
    r1 ← r2 + r3
    goto L2




    r5 ← r4 - 6
L2: r4 ← r3 shl 2
    r2 ← r2 + 1
    r2 ← r4 - r2
L1:
```

b)

图18-14 a) 适合尾融合的LIR代码，和b) 对它进行尾融合转换后的结果

为了实现尾融合，我们简单地从后向前扫描有多个前驱的基本块的每一个前驱，寻找是否有相同的指令序列。当存在这种共同指令序列时，仅保留其中之一，并用转移至这个保留的指令序列开始处的分支指令替换其他所有基本块中的相同指令序列（在这种处理中，通常会创建一个新的基本块）。

18.9 条件传送

条件传送（conditional move）是只有当指定的条件满足时才将源操作数复制到目的操作数的指令。这种指令可以在若干现代体系结构和实现中找到，例如SPARC-V9和奔腾 Pro。使用它们可以实现用不含分支的代码来替代某些简单的分支代码序列。例如，图18-15a中求a和b的最大值并将结果存储至c的代码，就能够用图18-15b的代码来替代。

```
    if a > b goto L1
    c ← b
    goto L2
L1: c ← a
L2:
```

a)

```
    t1 ← a > b
    c ← b
    c ←(t1) a
```

b)

图18-15 求a和b的最大值的代码，其中a) 使用分支，b) 使用条件传送

用条件传送替代分支代码所采用的方法是对中间代码进行模式匹配。在最简单的情况下，原来的代码必须具有图18-16a或图18-17a所示的形式，并且可分别用图18-16b或18-17b中等价的代码来替换。当然，也可以匹配和替换含多个赋值的序列，但这两种最简单的模式涵盖了实际代码中存在的大多数情况。

591

```
        if opd1 relop opd2 goto label1
        reg ← opd3
        goto label2
label1: reg ← opd4
label2:
```

a)

```
        reg ← opd3
        t ← opd1 relop opd2
        reg ←(t) opd4
```

b)

图18-16 a) 可用条件传送替代分支的代码样式的一个例子，b) 替代后的结果

bbs.theithome.com

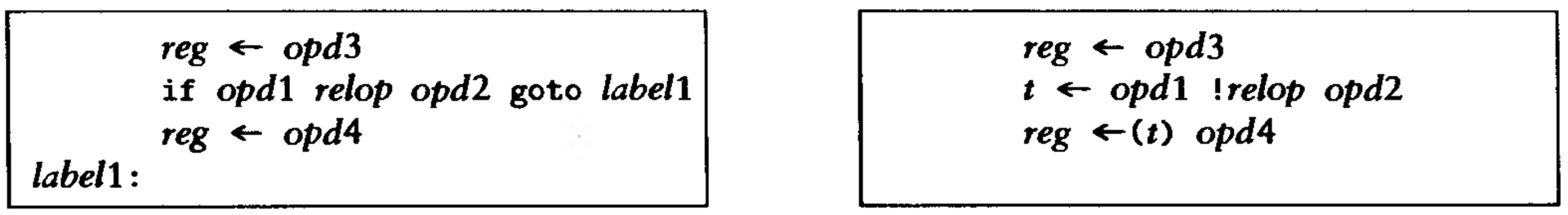


图18-17 a) 可用条件传送替代分支的代码样式的另一个例子，b) 替代后的结果

18.10 死代码删除

一个变量是*死去的*（dead），如果从对该变量定值的代码位置开始直到出口的任何路径上都不使用它。一条指令是*死去的*，如果它仅计算那种在从该指令开始的任何可执行路径上都不会使用的值。将一个死去变量的值赋给一个局部变量，如果在通向过程出口的任何可执行路径上都不使用这个局部变量（包括将它作为过程的返回值），则这个局部变量和给它赋值的这条指令也是死去的。如果将一个死去变量的值赋给可见性较宽的变量，通常需要过程间分析才能确定后者是否也是死去的，除非在从它的计算点开始的每一条可执行路径上都存在另一条对这个变量赋值的指令。

程序可能在优化之前就含有死代码，但这种代码更有可能是因为优化导致的；强度削弱（参见14.1.2节）是会产生死代码的优化的一个例子，其他很多优化也会产生死代码。许多优化创建死代码的部分原因是由于工作分割原理：每一种优化遍尽可能地保持简单，以便使其易于实现和维护，而将其余的整理工作留待它之后的其他遍去处理。

我们用一种乐观的方法来确定死去的指令。这种方法一开始先标识所有计算必要值的指令。一个值是*必要的*（essential），如果它是过程一定要返回或输出的值，或者它可能会对从过程外访问的存储单元有影响。然后，该算法以迭代方式逐条标识那种对计算必要值有贡献的指令。当这一过程已稳定不变时，所有未标识的指令都是死去的从而可以删除。

这个算法的一个微妙之处是，它能检测出仅定义自己的一个新值的变量；这种变量是死去的，因为它们除了浪费时间和空间外，对程序没有任何作用。例如，图18-18的代码中，变量*i*是不必要的，因为它的作用只是计算自己的一个新值。

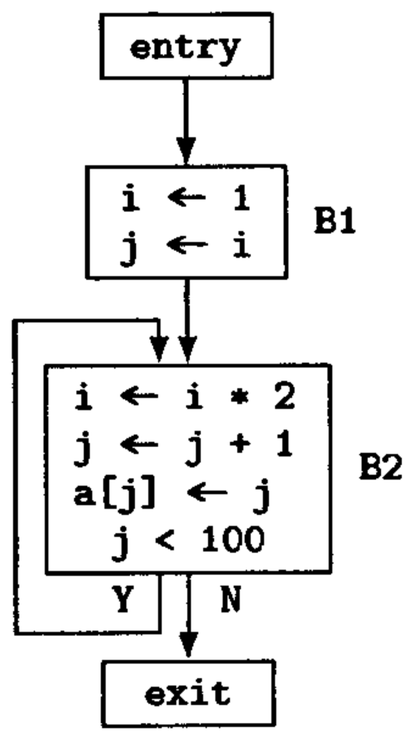


图18-18 变量*i*只对定义本身起作用，因而是不必要的

死代码识别也可以用数据流分析公式来表示，但使用工作表和它的ud链和du链要较容易一些。其方法如下：

1. 用必要指令的基本块索引偶对集合初始工作表。（在图18-19中，是那些在过程入口处Mark[i][j]=true的偶对<*i*, *j*>所组成的集合。）

2. 从工作表中删除一个偶对<*i*, *j*>。对于在这条指令中使用的每一个变量*v*，标识它的ud链UD(*v*, <*i*, *j*>)中的每一条指令，并将那条指令的基本块索引偶对放到工作表中。

3. 如果指令是一赋值，比如说，*v*←exp，则对于它的du链DU(*v*, <*i*, *j*>)中的每一个指令位置<*k*, *l*>，如果指令<*k*, *l*>是一个if，标识它并将<*k*, *l*>加入到工作表中。

4. 重复后面两个步骤直到工作表为空。

5. 从过程中删除未标识的每一条指令和每一个已变为空的基本块。

实现这种处理的ICAN代码如图18-19所示。例程Vars_Used(Block,x)返回指令

592

bbs.theithome.com

Block[x@1][x@2]中作为操作数使用的变量集合，Delete_Unmarked_Insts()使用Delete_Inst()删除每一条未标识的指令，并调整Mark[][]反映已做的删除。我们可以检查其结果是否能够做任何控制流转换，也可以将这些转换遗留给独立的一遍来处理；这些转换可包括18.3节和18.4节分别讨论的if化简和空循环删除。

```
UdDu = integer × integer
UdDuChain = (Symbol × UdDu) ⟶ set of UdDu

procedure Dead_Code_Elim(nblocks,ninsts,Block,Mark,UD,DU,Succ,Pred)
    nblocks: inout integer
    ninsts: inout array [1··nblocks] of integer
    Block: inout array [1··nblocks] of array [··] of MIRInst
    Mark: in array [1··nblocks] of array [··] of boolean
    UD, DU: in UdDuChain
    Succ, Pred: inout integer ⟶ set of integer
begin
    i, j: integer
    x, y: integer × integer
    v: Var
    Worklist: set of (integer × integer)
    || set of positions of essential instructions
    Worklist := {⟨i,j⟩ ∈ integer × integer where Mark[i][j]}
    while Worklist ≠ ∅ do
        x := ♦Worklist; Worklist -= {x}
        || mark instructions that define values used by
        || required instructions
        for each v ∈ Vars_Used(Block,x) do
            for each y ∈ UD(v,x) do
                if !Mark[y@1][y@2] then
                    Mark[y@1][y@2] := true
                    Worklist ∪= {y}
                fi
            od
        od
        || mark conditionals that use values defined by
        || required instructions
        if Has_Left(Block[x@1][x@2].kind) then
            for each y ∈ DU(Block[x@1][x@2].left,x) do
                if !Mark[y@1][y@2] & Block[y@1][y@2].kind
                    ∈ {binif,unif,valif,bintrap,untrap,valtrap} then
                    Mark[y@1][y@2] := true
                    Worklist ∪= {y}
                fi
            od
        fi
    od
    Delete_Unmarked_Insts(nblocks,ninsts,Block,Succ,Pred,Mark)
end     || Dead_Code_Elim
```

图18-19 检测和删除死代码的ICAN例程

593～594

有一点需要小心的是，我们必须保证产生一个死去值的操作没有执行诸如设置一个条件码之类的附带动作。例如，对于SPARC系统中其目的寄存器是r0并且设置条件码的一条整数减指令，如果存在对它设置的条件码的后续使用，它就不是死去的。

bbs.theithome.com

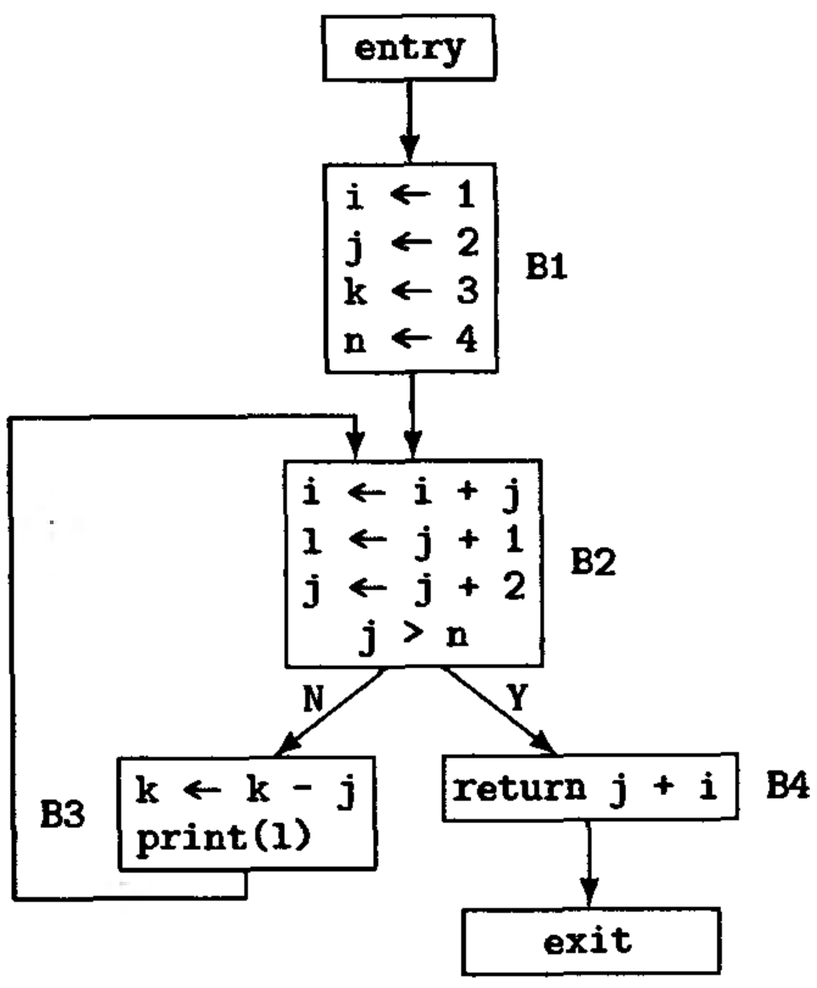


图18-20 用于死代码删除的例子

作为这个算法应用于MIR代码的例子，考虑图18-20中的流图。这个流图的ud链和du链如下:

变量定值	du链
i in ⟨B1,1⟩	{⟨B2,1⟩}
i in ⟨B2,1⟩	{⟨B2,1⟩,⟨B4,1⟩}
j in ⟨B1,2⟩	{⟨B2,1⟩,⟨B2,2⟩,⟨B2,3⟩}
j in ⟨B2,3⟩	{⟨B2,1⟩,⟨B2,2⟩,⟨B2,3⟩,⟨B2,4⟩,⟨B3,1⟩,⟨B4,1⟩}
k in ⟨B1,3⟩	{⟨B3,1⟩}
k in ⟨B3,1⟩	{⟨B3,1⟩}
l in ⟨B2,2⟩	{⟨B3,2⟩}
n in ⟨B1,4⟩	{⟨B2,4⟩}

和

变量使用	ud链
i in ⟨B2,1⟩	{⟨B1,1⟩,⟨B2,1⟩}
i in ⟨B4,1⟩	{⟨B2,1⟩}
j in ⟨B2,1⟩	{⟨B1,2⟩,⟨B2,3⟩}
j in ⟨B2,2⟩	{⟨B1,2⟩,⟨B2,3⟩}
j in ⟨B2,3⟩	{⟨B1,2⟩,⟨B2,3⟩}
j in ⟨B2,4⟩	{⟨B2,3⟩}
j in ⟨B3,1⟩	{⟨B2,3⟩}
j in ⟨B4,1⟩	{⟨B2,3⟩}
k in ⟨B3,1⟩	{⟨B1,3⟩,⟨B3,1⟩}
l in ⟨B3,2⟩	{⟨B2,2⟩}
n in ⟨B2,4⟩	{⟨B1,4⟩}

595

一开始，必要指令集合由print和return组成，因此只有Mark[B3][2]=Mark[B4][1] = true, Worklist = {<B3,2>, <B4,1>}。

现在，我们开始标识对这两条必要指令有贡献的指令。首先从工作表中删除x=<B3, 2>，这导致Worklist={<B4, 1>}，并发现Vars_Used (Block, <B3, 2>) ={1}。因此我们检查<B3, 2>中l的ud链的一个成员，即<B2, 2>，设置Mark[B2][2]=true，并将<B2, 2>加入工作表，得到Worklist={<B4, 1>, <B2, 2>}。

接着我们确定出Has_Left(Block[B3][2].kind)=false，于是继续到工作表的下一

bbs.theithome.com

元素。我们从工作表中删除x=<B4, 1>，留下工作表的值为Worklist={<B2, 2>}，并发现Vars_Used (Block, <B4, 1>) ={i, j}。所以，我们设置v=i并检查UD (i, <B4, 1>) ={<B2, 1>}的每一个成员。我们设置Mark[B2][1]=true，并将<B2, 1>加入到工作表中，得到Worklist={<B2, 2>, <B2, 1>}。随后设置v=j并发现UD(j, <B4, 1>)={<B2, 3>}，导致设置Mark[B2][3]=true，并将<B2,3>加入工作表，因此 Worklist={<B2, 2>, <B2, 1>, <B2, 3>}。接下来确定出Has_Left(Block[B4][1].kind)=false，因此我们继续到工作表的下一元素。

现在我们从工作表中删除x=<B2, 2>，留下工作表的值为Worklist={<B2, 1>, <B2, 3>}，并发现Vars_Used (Block, <B2, 2>) ={j}。所以，我们设置v=j并检查UD(j, <B2,, 2>)={<B1, 2>, <B2, 3>}的每一个成员。这导致设置Mark[B1][2]=true (Mark[B2][3]已经为true)，并将<B1, 2>加入到工作表中，即Worklist ={<B2, 1>, <B2, 3>, <B1, 2>}。

接着我们从工作表中删除<B2, 1>，遗留Worklist={<B2, 3>, <B1, 2>}。Vars_Used(Block, <B2, 1>)={i, j}，所以，我们考察UD(i, <B2, 1>)={<B1, 1>, <B2, 1>}。我们设置Mark[B1][1]=true并将<B1, 1>加入工作表，得到Worklist={<B2, 3>, <B1, 2>, <B1, 1>}。Mark[B2][1]已经置为true，所以我们继续到工作表的下一元素。现在UD(j, <B2, 1>)={<B1, 2>, <B2, 3>}，它们两者都已被标识过，因此我们继续前进。

接下来我们从工作表中删除<B2, 3>，遗留其值为Worklist={<B1, 2>}。现在Vars_Used(Block, <B2, 3>)={j}，所以，我们检查Mark[B1][2]和Mark[B2][3]是否已标识过。它们两者都已标识过，因此我们继续处理并确定出Has_Left(Block[B2][3].left) = true，于是我们检查Block[B2][1]、Block[B2][2]、Block[B2][3]、Block[B2][4]、Block[B3][1]和Block[B4][1]中的每一个，确定这些位置中是否有条件指令且没有被标识。只有Block[B2][4]是这种指令，因此设置Mark[B2][4] = true，并将<B2, 4>加入到工作表中，得到 Worklist ={<B1, 2>, <B2, 4>}。

现在我们从工作表中删除<B1,2>，留下工作表的值为Worklist={<B1, 1>, <B2, 4>}。现在Vars_Used(Block, <B1, 2>)=∅，且DU(Block[B1][2], j)={<B2, 1>, <B2, 2>, <B2, 3>}，它们都已经被标识过。

接下来我们从工作表中删除<B1, 1>，留下其值为Worklist={<B2, 4>}。现在Vars_Used(Block, <B1, 1>)=∅，DU(Block[B1][1], i)={<B2, 1>}，并且<B2, 1>已经被标识过。

最后，我们从工作表中删除<B2,4>，留下工作表为空。Mark[B2][4]已经为true，所以算法以图18-21a所示的Mark值而终止，并且执行死代码删除所得到的流图如图18-21b所示。

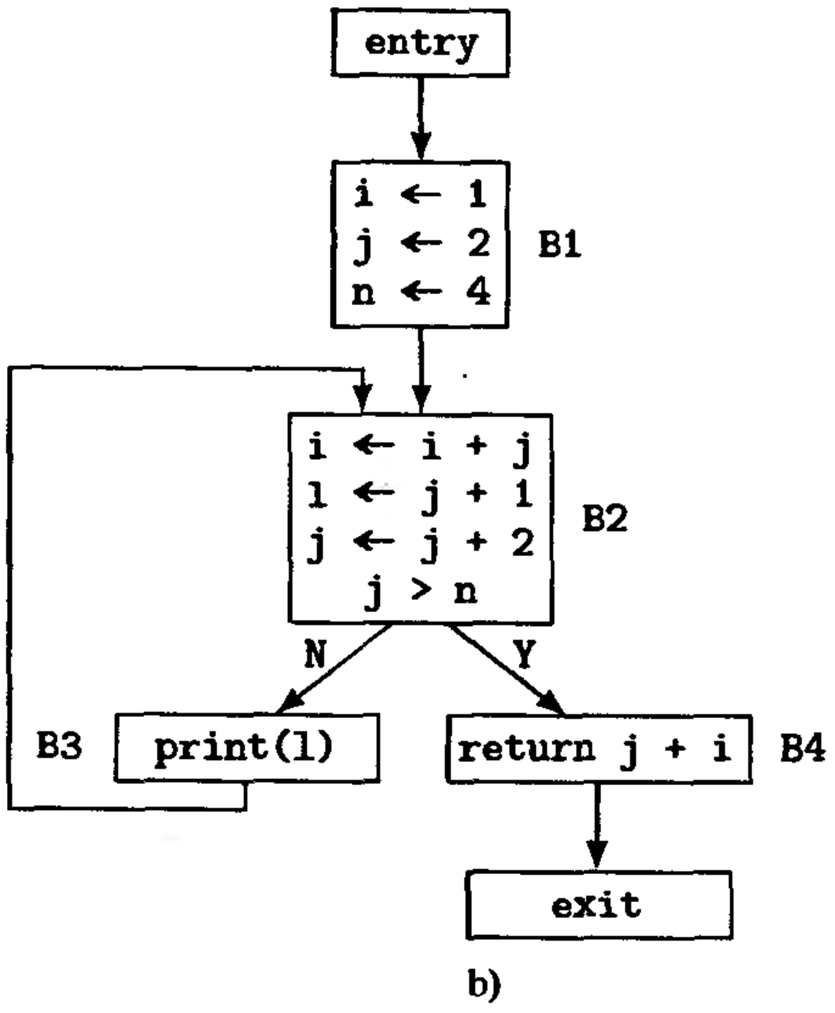
596

Knoop、Rüthing和Steffen [KnoR94]已将死代码删除扩展到部分死代码的删除，其中“部分”的含义与部分冗余删除中使用的含义相同，即指在某些路径上是死去的，但在其他路径上可能并不是死去的。

bbs.theithome.com

块	位置	值
B1	1	true
	2	true
	3	false
	4	true
B2	1	true
	2	true
	3	true
	4	true
B3	1	false
	2	true
B4	1	true

a)



b)

图18-21 a) 标识出死代码之后的Mark表，b) 图18-20中的流图经死代码删除之后

18.11 分支预测

这一节和随后的一节，我们讨论几种几乎总是推迟到机器代码已经生成之后才进行的优化，它们常称为后遍（postpass）优化。另一个常用术语是窥孔（peephole）优化，因为它们是通过移动一个小窗口或窥孔，在已生成的代码上或已生成代码的依赖DAG上来寻找可以进行优化的机会的。注意，利用依赖DAG便于找出要执行的优化所关心的一对（或一组）指令，例如，这一对指令可能是通过指针来读取一个数组元素，然后增加该指针使之指向下一元素。

分支预测（branch prediction）指的是静态或动态地预测条件分支指令是否会导致控制转移。精确的分支预测很重要，因为它使得处理器可以在导致控制转移的分支指令被执行之前，从指令高速缓存中读取连续路径上的指令到指令预取缓冲区，或从主存或二级高速缓存读到指令高速缓存中。随着高速缓存的延迟相对于处理器主频的差距越来越大，预测也越来越重要。

动态分支预测方法包括了各种策略，其范围从一个较小的保存条件分支指令地址并记录它最近一次执行是否发生分支转移的简单高速缓存，到较大的高速缓存。对于最近执行的一组分支指令，这种高速缓存包含转移目标的前几条指令（下降路径不必缓存，因为原来的指令读取机制就适用于它）。当然，实现这些策略需要硬件资源的支持，并且可能相当复杂——例如，后一种策略在分支指令或它的目的地代码或下降点的代码被动态改变时，需要有逻辑电路使得高速缓存中的登记项无效。另外，较简单的策略，如上面提到的第一种策略，则可能不是非常有效——例如，可能会有这种情形，一个特定的分支指令每次执行时都在转移和不转移之间进行切换。

597

静态分支预测方法对硬件资源的需要较少，并且已经证明它几乎同动态预测策略一样好。某些体系结构，如SPARC-V9，提供有指明静态分支预测手段的条件分支指令；其他的一些实现，如POWER的RIOS版本，对于分支的转移和下降两种情形有不相等的延迟，因此选择可能性较大的路径作为分支指令延迟较小的一方具有较大的好处。

作为静态分支预测正确程度的一种度量，我们定义完美静态预测器（perfect static predictor）为这样一种预测器，它用某种似乎有点费解的方法，以对于一条分支指令而言至少

bbs.theithome.com

有50%的时间是正确的方式，静态地预测每一条件分支指令是否会发生控制转移。因此，如果一条分支指令恰好有一半的时间导致控制转移，完美静态预测器就有50%的时间是正确的，并且当该分支指令使得控制走其中一条路径比另一条路径更频繁时，这个完美静态预测器的正确性就应更好。如果该分支指令总是导致转移发生或决不发生转移，则它的正确性就将是100%。当动态计算给定程序中的所有分支指令时，完美静态预测器提供了任意静态预测器对那个程序能够预测的程度的一个上界。

有些简单的启发式方法的预测正确性可以高于平均值。例如，对于向后条件分支，预测分支会发生的成功性相对要大一些，因为多数向后转移的分支指令是循环的闭合分支指令，并且大部分循环在退出之前都要执行多次。类似地，预测向前转移的分支不会发生转移的成功性也相对要大些，因为这种分支指令中只有小部分是关于测试一个例外条件，并绕过相应处理代码的。

Ball和Larus [BalL93] 讨论了更好的静态分支预测启发式方法，它们通过更精确地定义循环分支而改善了前面提到的简单向后分支的启发式。假设已给我们一个流图，并且已确定出了它的正常循环（参见7.4节）。定义*循环分支*（loop branch）是这种分支，它有此流图中的一条出边，这条出边或者是一条回边，或者是一条从循环出口的边[⊖]；*非循环分支*（nonloop branch）是其他的分支。为了静态预测循环分支，我们简单地选择回边。对于真实的程序，它是适合的也是很成功的。

对于非循环分支，有一些简单的启发式方法有助于更为成功地预测它们。例如，操作码启发式方法根据与0或一个负整数值的整数比较，以及浮点值之间的相等测试来检查分支。它预
598 测非正值的整数分支和浮点相等分支不会发生，因为前者频繁用作错误指示器，而后者很少为真。循环启发式方法检查一个后继基本块是否不是分支所在基本块的后必经结点，并且既不是循环首结点，也不是循环前置结点。如果一个后继基本块满足这个启发式，则预测转移到这个后继基本块的分支会发生转移。

分支预测是一个仍在继续的研究领域。关于当前工作的更多参考文献参见本章末尾的18.14节。

18.12 机器方言和指令归并

有若干种优化只能在机器特有的代码已经生成之后才能执行，或者最好留到编译过程快结束时再执行。如果全局优化是在低级中间代码上进行的，这些优化可以稍早一点进行，但在某些情况下，在执行这些优化时需要小心地做出最有效的选择。例如下面讨论的将这样两条指令归并为单条指令的情形，其中一条指令通过指针引用一个数组元素，另一条指令增加该指针使之指向下一元素。尽管有些体系结构——如PA-RISC、POWER和PowerPC——允许归并这种指令，但它可能不是最好的选择，因为归纳变量优化可以删除那条使指针值增加的指令，所以，我们在编译接近结束时才做指令归并。

机器方言（machine idioms）是体系结构特有的指令和指令序列，它们为执行一个计算提供的方法比编译针对通用的体系结构所采用的方法更为有效。许多机器方言是指令归并的实例，*指令归并*（instruction combining）即用具有相同效果的一条指令替换一组指令。

因为所有RISC执行大部分计算指令都只需要一拍时钟周期，因此RISC的多数机器方言，但并不是所有的，是将两条成对的指令合并为一条机器方言指令。对于其他体系结构——如

⊖ 如Ball和Larus所指出的，也有可能两条出边都是回边。他们指出，在实际中还未见到这种情形，但假如确实出现了的话，我们应当选择预测较内层循环的边。

bbs.theithome.com

超星公司制作 请尊重作者版权

VAX、Motorola M68000以及Intel 386系列——存在着用时钟周期数较少的一条指令替代另一条指令的机器方言，以及由若干条指令合并而成的单条指令的机器方言。

本节我们给出几种体系结构机器方言的一些例子。这些例子并不代表所有的情况——它们主要是为了说明可能性。在多数情况下，我们关心的是特定的体系结构。识别使用机器方言机会的主要技术是模式匹配。搜索过程有两个主要部分。第一部分是寻找那种可用更快、更专门的指令来达到其目的的指令。第二部分首先寻找这样的一条指令，它是可以用较短或较快指令序列归并的一组指令中的第一条指令；找到一条这种指令便触发对另一条（些）为形成适当组合所需要的指令的寻找。除非目标体系结构允许将功能独立的若干指令归并为一条指令（对于MIPS体系结构，在某些情况下就是这样），通常在依赖DAG上（而不是在直线代码上）执行这种搜索效率最高，并且也很有效果。

在所有RISC体系结构中，对于那种构造全宽度（即32位或64位）常数的代码，如果发现这个常数短得足以填充在指令提供的直接数域中时，则存在着简化它的机会。尽管我们希望完全避免生成不必要的构造长常数的指令序列，但常常更可取的是生成它们而使得代码生成较简 599
单，同时将对它们的简化遗留到有可能做此工作的后遍优化中去做。例如，SPARC指令序列

```
sethi   %hi(const), r18
or      r18, %lo(const), r18
```

将常数值const放至r18。如果const的高20位都是0或都是1，则可以将它简化为

```
add     r0,const,r18
```

下面考虑一个整变量乘以一个整常数的乘法。具体地，假设我们是要将寄存器r1中的一个整变量乘以5并将结果存放在r2中。在多数体系结构中，可以使用整数乘法指令来完成，但乘法指令一般是多时钟周期指令，并且在某些机器中，如PA-RISC，需要将操作数传送到浮点寄存器，然后将结果传送回整数寄存器。不过有比这代价更小的方法。对于PA-RISC，我们可以简单地生成只要一拍的指令:

```
SH2ADD     r1,r1,r2
```

该指令将r1左移两位并加上它原来的值，结果仍存放在r1。对于其他体系结构，可以使用相应的LIR指令序列:

```
r2 ← r1 shl 2
r2 ← r2 + r1
```

对于SPARC，我们可以将一条产生一个差值的减法指令和对该减法的两个操作数进行比较的指令归并到一起，即使得

```
sub   r1,r2,r3
. . .
subcc r1,r2,r0
bg    L1
```

变成

```
subcc r1,r2,r3
. . .
bg    L1
```

对于MIPS，可以利用下面的指令序列（不用分支）确定出两个值是否有相同的符号:

```
slti  r3,r1,0
slti  r4,r2,0
and   r3,r3,r4
```

bbs.theithome.com

如果两个值有相同的符号，序列结束时r3应当等于1；否则为0。

在Motorola 88000、PA-RISC和Intel 386系列中，访问由字节、半字、字和双字组成的、其
600 索引增量为1的并行数组，可以使用带缩放的读取和存储指令。如果r2和r3分别指向双字和半字的两个数组，并且r1是这两个数组的索引，则读取这两个数组的元素并更新其索引到这两个数组的下一个元素可以只用下述Motorola 88000指令序列来完成:

```
ld.d   r4,r2[r1]
ld.h   r6,r3[r1]
add    r1,r1,1
```

而不需要两个独立的索引，一个增加2，另一个增加8；也可能还不需要第三个用于循环迭代计数的索引。

在PA-RISC、POWER和PowerPC中，有带增量的读取或存储指令，这种指令可以用新的地址值替代用来形成存储地址所使用的一个寄存器中的值。在后面两种体系结构中，更新总是发生在读取或存储之后，而PA-RISC可以指定更新出现在操作之前还是之后。因此，在PowerPC中访问一个半字数组的连续元素，可以通过重复执行

```
lhau    r3,2(r1)
```

在PA-RISC中则可用

```
LDHS,MA    2(0,r1),r3
```

对于PA-RISC，可以用一条相加然后分支的指令来终止一个循环。这种指令首先给一个寄存器增加一个直接数或增加一个寄存器中的值，然后当结果满足指定的条件时做分支转移。因此，一个以r1作为索引，r2包含该索引终值对数组进行的循环，可以用这样一条指令来终止:

```
ADDBT,<=    r2,r1,L1
```

其中L1是循环体的第一条指令的标号。

指令归并程序可能会导致增加超标量指令组的数目，这种指令组要求执行一串指令。例如，假设我们有一个有三个流水线的实现（两个整数和一个浮点），并且有图18-22a所示的成组指令
601 序列，其中sub必须先于br，add2必须跟在cmp之后。将cmp和br归并成一条比较与分支指令（cmpbr）会导致这些指令需要如图18-22b所示的4个流出槽，而不是它们原来需要的3个流出槽。将指令归并程序和指令分解程序作为可被指令调度器使用的子程序，从而让指令调度器能够尝试可能的指令序列，来确定什么样的指令序列能够产生最好的调度，可以减轻这种影响，但在这样做时需要小心限制这种尝试的程度，因为大多数调度问题至少都是NP-困难的问题。

Int	Int	Flt
load	add1	flt1
cmp	sub	flt2
br	add2	flt3

a)

Int	Int	Flt
load	add1	flt1
sub	---	flt2
cmpbr	---	flt3
add2	?	?

b)

图18-22 a) 超标量处理机代码，对它进行b) 中所示指令归并能降低性能的例子

也可以反过来让指令归并程序包含流水线结构有关的信息，并构造指令归并程序，使它能够估计它所做的归并对指令调度的影响——但这样做很可能会使得归并程序显著地复杂化，并且这种做法在实际中效果不是很好，因为除非在基本块DAG上进行调度，一般调度所使用的代码窗口通常比窥孔优化程序所使用的窗口要宽得多。

bbs.theithome.com

上面指出的只是少数几种机器方言的例子。用心的编译器开发者通过深入了解所编译的指令集和仔细观察已生成代码的实例，能找到许多其他的方言例子。

对于CISC体系结构的实现，有时会需要有指令分解（instruction decomposing），即指令归并的逆操作，它将一条指令转换为执行相同功能的一串指令。在图21-23和图21-24中为Intel 386体系结构系列生成的关于子程序s1()和对应的主程序的代码中，可以看到这种情形。嵌入在主程序中的子程序的代码是CISC风格的代码，而子程序中的代码是RISC风格的（并且已按展开因子4展开）。注意，可以先生成这两种代码中的任何一种，然后从其中之一产生另一种。

18.13 小结

本章的内容涵盖了控制流优化、不可到达代码和死代码删除，以及其余一些最好在程序的低级代码上来执行的全局优化，即静态分支预测、机器方言和指令归并。其中的某一些优化，如死代码删除，在编译期间可以多次执行而得到好处。另一些优化则最好在低级中间代码上或结构化形式的汇编或机器语言上来进行。

除死代码删除外，这些优化中的大部分优化，当作用于一段具体代码一次时，所产生的影响很小。但是，当将它们全部施加于整个过程时可以导致实质性的节省，尤其是当这些优化施加于频繁执行的循环时。

产生较大基本块的控制流转换能够潜在地增加指令级并行性，而指令级并行性对于超标量实现尤其重要。

死代码删除去掉那些已确定是对计算结果不起作用的指令。它是一种重要的优化，不仅仅是因为有些程序在编写时就有死代码，而且也因为其他许多优化会创建死代码。我们建议在编译优化期间多次执行它，如图18-23所示。 602

随着高速缓存的延迟相对于处理器主频的差距越来越大，分支预测的重要性也在增加。精确的分支预测是重要的，因为它增加了处理机从指令高速缓存中，或从存储层次结构的其他部件中读取正确的连续路径的机会。

本章讨论的其他几种优化是后遍优化或窥孔优化。它们包括机器方言和指令归并，通常属于最后执行的几种优化，且总是在代码生成之后才执行。它们可以通过在已生成的代码中或在依赖DAG上移动一个小小的窗口或一个小孔来寻找可施加对应优化的代码片段而实现。

图18-23给出了推荐的激进优化编译器中的优化顺序；黑体字突出了本章涉及的这些优化。

18.14 进一步阅读

首先讨论无开关化的论文是[AllC72a]。

18.10节给出的死代码检测算法是从Kennedy [Kenn81]开发的算法衍生而来的。

Knoop、Rüthing和Steffen的死代码删除到部分死代码删除的扩充是在[KnoR94]中找到的。

通过对分支实际发生的情况和完美静态预测器预测的情况进行比较，来确定分支预测策略的效率，是由Ball和Larus在[BalL93] 中提出的。那篇论文也介绍了若干有用的分支预测启发式方法。更新一点的论文，例如[Patt95]和[CalG95]，还介绍了一些更好的启发式方法。

[GilG83]中描述了斯坦福MIPS体系结构。

bbs.theithome.com

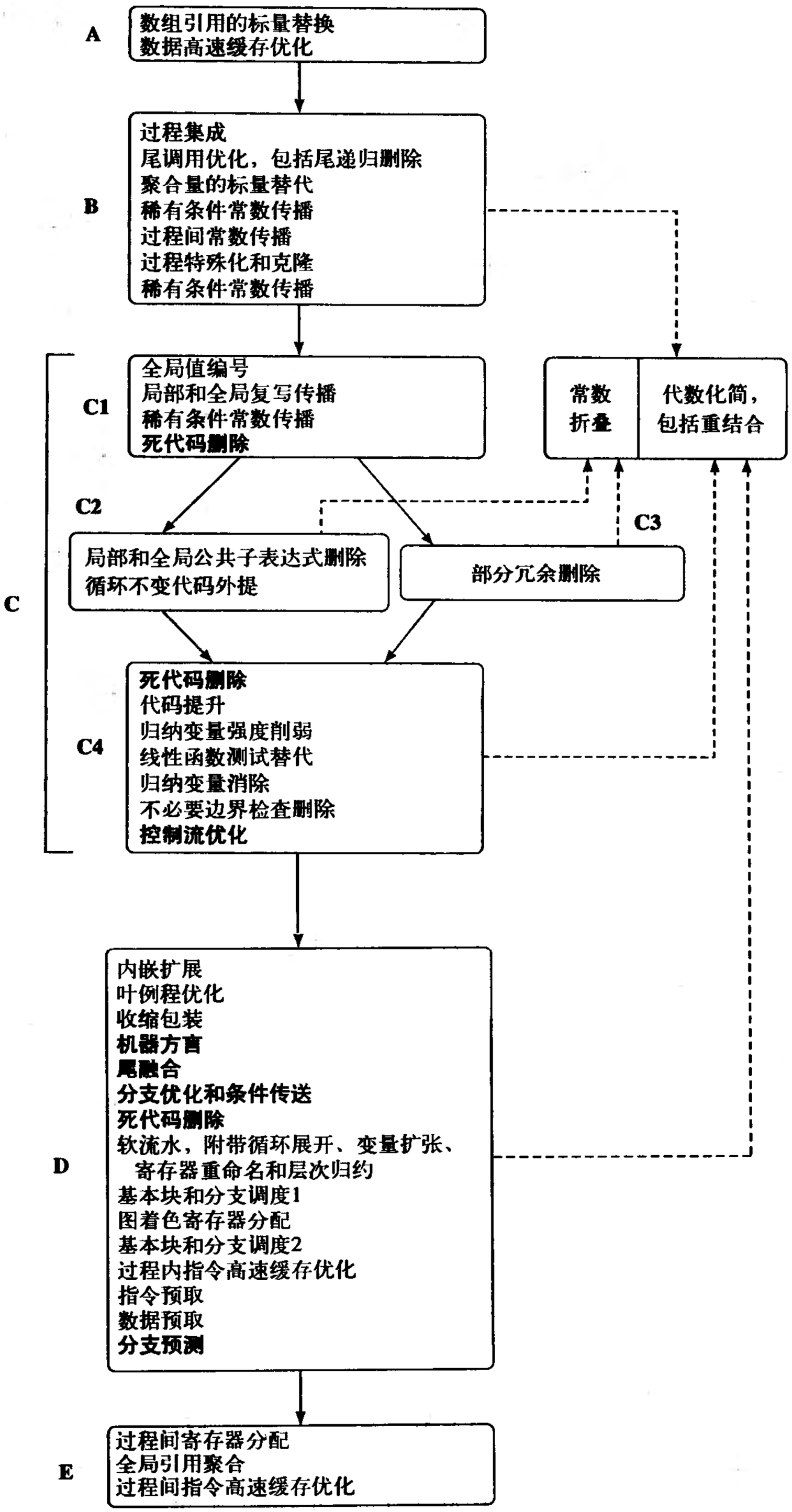


图18-23 优化顺序，黑体字标明了控制流和低级优化

bbs.theithome.com

超星公司制作 请尊重作者版权

18.15 练习

18.1 编写一个检测和执行18.3节讨论的`if`化简的ICAN过程。

18.2 编写一个检测LIR代码中尾融合机会并做相应转换的ICAN过程。通过搜索基本块DAG寻找以叶子结点结束的公共指令序列，或者寻找作为叶子或它们的前驱出现的不相关指令的集合，从中能有什么收获吗？

18.3 给出三个利用条件传送能使它更具效率的有用指令序列的例子。

18.4 说明图18-19中的死代码删除算法能识别并删除那种只对相互定义有贡献，因而不是必要的变量集合$\{v_1, v_2, v_3, \ldots\}$的定义。

18.5 对你可用来计算的正常应用程序设置测量器，并打印出关于每一个条件分支发生转移的百分比统计信息。从由此得到的结果中你能推导出对静态分支预测有用的规则吗？

18.6 设计一种用于归并或转换汇编语言指令序列为机器方言的模式语言。你将需要一种手段，例如，能抽象地表示寄存器，使得你可以要求在两条指令中使用同一个寄存器，而无需约束这个寄存器是一个特殊的寄存器。

18.7 写出一个寻找汇编语言中指令归并和机器方言机会的程序，其中指令归并和机器方言用练习18.6中设计的模式语言表示，并将该程序应用于汇编代码。这个程序应当将模式文件和它要处理的汇编代码文件作为输入文件。

603~605

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第19章　过程间分析与优化

模块化一直被看成是程序结构的优点——并且确实是这样，因为将程序适当地分解成若干过程有助于创建结构良好且易于理解和维护的程序。最近一些年来，面向对象的语言进一步助长了这种趋势。尽管某些语言，如C++，给程序员提供了明显的手段来指明一个应当内联的过程，但大多数语言鼓励这种观点，即，一个过程是由一个接口和一至多个实现它的黑匣子组成的，或更一般地，一种数据类型或一个类是由一个接口和一至多个实现它的黑匣子组成的。这种方法鼓励抽象化，并因此有利于良好的程序设计和维护，但它同时也抑制了优化，并因此导致可能产生比其他语言效率要低的代码。缺乏过程间控制流和数据流信息，我们通常必须假设被调用的过程可能使用和改变它能访问到的所有变量，并且调用过程可能提供任意值作为参数。这两种假设显然都对优化有抑制作用。

迄今为止，我们已经讨论的所有优化，除了尾调用优化（15.1节）、过程集成（15.2节）和内嵌扩展（15.3节）是过程之间的优化之外，几乎都是过程内的（intraprocedural）优化，也就是说，它们是作用于单个过程之内的，没有考虑到调用所在过程的上下文以及被它调用的过程。

过程间优化（interprocedural optimization）指的是利用过程集合之间的调用关系，对其中的一至多个过程，或按它们相互之间的关系来驱动优化的一些优化。例如，如果我们能确定出一个程序中对过程f(i,j,k)的所有调用都传递常数2作为i的值，就可能将常数2传播到f()的代码中。类似地，如果我们能够确定每一个调用传递给i的值不是2就是5，并能识别出哪一个调用传递的是哪一个值，我们就可能用两个过程f_2()和f_5()来替代f()——这种处理称为过程克隆——并用它们两者之一适当地替换f()的调用。类似地，如果我们知道被调用的过程修改的只是它自己的局部变量和特定的参数，只要我们考虑到了过程的已知副作用，就可以自由地优化在它的调用周围的代码。

同过程内的情形一样，过程间优化由控制流分析、数据流分析、别名分析以及相应的转换等一系列的遍组成。它与过程内优化的不同之处在于，过程间分析得到的许多好处是因为它改善了单个过程的优化可能性和优化效果，而不是转换了过程之间的关系所致。在这一章，我们探讨下面一些内容:

1. 过程间控制流分析，特别是程序控制流图的构造;
2. 过程间数据流分析的若干种方法，包括流敏感的与流不敏感的副作用分析和常数传播;
3. 过程间别名分析;
4. 如何利用过程间分析获得的信息进行优化;
5. 过程间寄存器分配。

在本章的余下部分，除非特别指明，我们均假定所有参数都是传地址参数，而将本章讨论的方法为适应其他参数传递规则的修改留给读者完成。注意，以传值方式传递一个非指针不会创建别名，而以传值方式传递一个指针则与以传地址方式传递它所指的对象十分类似。

有些研究认为过程间分析是徒然的，或代价太大而不值得做（参见19.10节的引文）。例如，Richardson和Ganapathi研究比较了过程集成对于优化的作用和过程间优化的效率。他们发现过程集成对增强优化非常有效，但它显著地降低了编译速度，常常使速度慢了9倍之多。另一方面，他们进行的有限的过程间分析则显示优化从它获益甚少。另外，过程间分析给编译器增加

bbs.theithome.com

了较重的负担，同时也增加了编译器的复杂性。一般而言，过程间分析的典型代价和作用还没有得到广泛认识，有些迹象显示，它对于并行化编译器比对串行机编译器更有价值。

将一个程序分成多个编译单位进行编译影响了过程间分析和优化的效果，因为无法确定还没有被编译的那些例程的作用。另一方面，现代程序设计环境也通常提供关于编译单位、它们之间的关系，以及与它们有关的信息的知识库，这就为过程间分析提供了保存和访问相关信息的一种途径（参见19.8节）。

在第10章的开始和12.2节讨论的关于可能与一定信息、流敏感与流不敏感信息的区别，也适
608 应于过程间优化。具体而言，19.2.1节讨论的*MOD*()和*REF*()的计算是流不敏感问题的例子，*DEF*()和*USE*()函数是流敏感问题的例子。另外，*DEF*()是一定概要信息，而*USE*()是可能概要信息。

19.1 过程间控制流分析：调用图

过程间控制流分析涉及的一个问题是构造程序的调用图[⊖]。给定一个由过程$p_1, \cdots, p_n$组成的程序P，P的（静态）调用图是由结点集合$N=\{p_1, \cdots, p_n\}$、调用点标号[⊖]集合S、带标号的边集合$E \subseteq N \times N \times S$，以及一个区别对待的入口结点$r \in N$（代表主程序）组成的图$G_P = <N, S, E, r>$（通常就写作$G$），其中对于每一个$e = <p_i, s_k, p_j>$，$s_k$表示$p_i$中对$p_j$的一个调用点。如果过程$p_i$对$p_j$只有一次调用，我们可以省略调用点$s_k$，并将此边写成$p_i \rightarrow p_j$。

作为调用图的一个例子，考虑图19-1中的程序框架。过程f调用g和h，g调用h和i，而i调用g和j；注意，f有两次对g的调用。这个程序的调用图给出在图19-2中。入口结点由加粗的圆圈指明，f对g的两次调用由连接它们的边之上的两个标号指明。

```
1     procedure f( )
2     begin
3        call g( )
4        call g( )
5        call h( )
6     end   || f
7     procedure g( )
8     begin
9        call h( )
10       call i( )
11    end   || g
12    procedure h( )
13    begin
14    end   || h
15    procedure i( )
16       procedure j( )
17       begin
18       end   || j
19    begin
20       call g( )
21       call j( )
22    end   || i
```

图19-1 一个程序框架

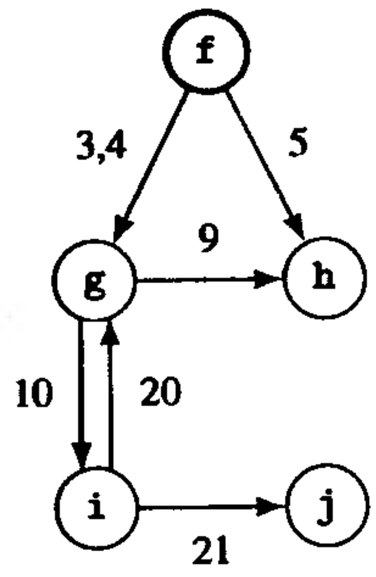


图19-2 图19-1中程序框架的调用图

与过程分析中的深度为主顺序和其他若干种顺序一样，也有一些有助于过程间问题、至少有助于

⊖ 如我们将看到的，调用图实际上是多图（multigraph），即从一个结点到另一个结点具有多条有向边的图；或换言之，它是有些结点具有多个标号的图。但是，同这一领域的多数作者一样，我们简单地称之为图。

⊖ 注意，在高级语言中，行号通常不足以作为调用标号，因为在同一行中可以有多次调用。

bbs.theithome.com

非递归问题的顺序。一种顺序称为调用顺序（invocation order），这种顺序处理一个过程先于被该过程调用的所有过程，即按调用图的宽度为主顺序。另一种顺序称为逆调用顺序（reverse invocation order），它是在处理完一个过程内的所有被调用过程之后才处理该过程。如果图19-1和图19-2的例子中没有i对g的调用，则f、g、h、i、j列出的就是它的结点的调用顺序。如我们的例子所示，递归程序的过程之间没有这种顺序，除非我们将强连通分量蜕化为单个结点，并关心的是这种结点之间的调用。

过程间分析使用的另外两种顺序涉及过程的静态嵌套。由外向内顺序（outside-in order）处理每一个过程先于（词法上）静态嵌套在其内的所有过程。由内向外顺序（inside-out order）处理每一个过程后于（词法上）静态嵌套在其内的所有过程。对于我们的例子，j、f、g、h、i是调用图的一种由内向外顺序。

有两个问题会导致难于构造调用图，即一个程序分成多个编译单位进行编译和过程值变量。如果不存在这两个问题，构造程序的调用图就是一件容易的事情，例如图19-3中过程Build_Call_Graph()的ICAN代码所示。在这个算法中，P是过程集合，它由要构造调用图的程序中的过程组成，N是此图中结点的集合[⊖]，r是根过程（常称为main），E是图中带标号的边集合。标号表示调用点，即，$\langle p, i, q\rangle \in E$当且仅当过程$p$中的调用点$i$调用$q$。该算法用到了如下一些过程：

609

1. numinsts(p)是过程p中的指令条数。

2. callset(p, i)返回由过程p的第i条指令调用的过程集合。

分开编译引起的问题，可以通过仅当整个程序一次提交给编译器时才做过程间分析和优化而绕过，也可以通过在每一次编译时保存这次编译所看到的部分调用图的表示，并以逐渐增加的方式建立整个调用图来解决。在前面这个例子中，如果f和g组成一个编译单位，其他三个过程组成另一个编译单位，则有如图19-4所示的两个部分调用图，过程间优化可以“粘合”这

```
LabeledEdge = Procedure × integer × Procedure

procedure Build_Call_Graph(P,r,N,E,numinsts)
   P: in set of Procedure
   r: in Procedure
   N: out set of Procedure
   E: out set of LabeledEdge
   numinsts: in Procedure ⟶ integer
begin
   i: integer
   p, q: Procedure
   OldN := ∅: set of Procedure
   N := {r}
   E := ∅
   while OldN ≠ N do
      p := ♦(N - OldN)
      OldN := N
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            N ∪= {q}
            E ∪= {⟨p,i,q⟩}
         od
      od
   od
end    || Build_Call_Graph
```

图19-3 构造调用图

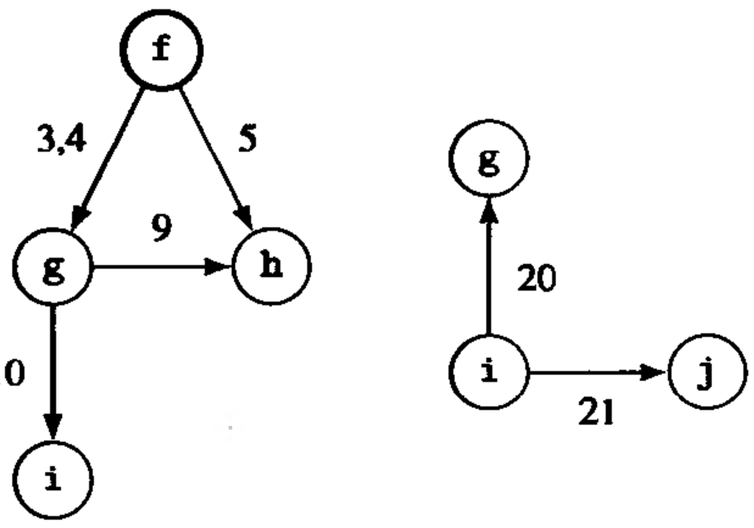


图19-4 将图19-1分成两个部分进行编译而得到的两个调用图

⊖ 注意，只要P中每一个过程都是从r可达的，就有P=N。

bbs.theithome.com

两个图到一起而形成一个完整的静态调用图。

610
~
611

构造调用图的算法十分简单，因此我们不提供关于其操作的例子。读者可以自己构造一个。

过程值变量引起的问题较难解决。Weihl [Weih80]已证明构造具有过程变量的递归程序的调用图是PSPACE-困难的，因此这种处理在时间和空间上代价都非常大。处理这个问题的一种方法是以逐渐增加的方式构造调用图，如图19-5、图19-6和图19-7中的代码所示。这三张图给出了三个过程Build_Call_Graph_with_PVVs()、Process_Inst()和Process_Call()。我们眼下先暂时忽略Build_Call_Graph_with_PVVs()中最外层的repeat循环。我们首先构造由明显的调用而形成的图。然后确定过程值变量的初始值集合，并将它们传播到所有的使用——即，传播给过程值参数、赋值和返回——并从那儿传播到使用过程值变量的调用点。过程Process_Inst(*p, i, Inst*)处理过程内的每一条指令，而过程Process_Call(*p, i, q*)处理调用。这个算法使用了如下四种数据结构来记录与过程值变量有关的信息:

1. PVVs记录所有过程值变量。

2. PVCalls记录调用过程值变量的调用点。

3. PVVals记录赋予过程值变量的、或与过程值变量结合的、或返回给过程值变量的过程常数。

4. PVBinds记录赋予过程值变量的、或与过程值变量结合的、或返回给过程值变量的过程值变量。

```
P, N, W: set of Procedure
E: set of LabeledEdge
PVVs: set of ProcVar
PVCalls: set of LabeledEdge
PVVals: ProcVar ⟶ set of Procedure
PVBinds: ProcVar ⟶ set of ProcVar
preturns: Procedure × integer × Var × Procedure × Var

procedure Build_Call_Graph_with_PVVs(r,Inst,numinsts)
   r: in Procedure
   Inst: in Procedure ⟶ array [··] of Instruction
   numinsts: in Procedure ⟶ integer
begin
   p, q, u, v: ProcVar ∪ Procedure
   i, j: integer
   more, change: boolean
   N := W := {r}
   E := PVVs := PVCalls := PVVals := PVBinds := ∅
   || repeat until the call graph stops expanding
   repeat
      more := false
      || accumulate procedure-valued variables and
      || constants, calls, values, and bindings
      while W ≠ ∅ do
         p := ♦W; W -= {p}
         for i := 1 to numinsts(p) do
            more ∨= Process_Inst(p,i,Inst)
         od
      od
      || propagate bindings
*     repeat
         change := false
         for each u ∈ PVVs do
```

图19-5 构造含过程值变量的调用图

bbs.theithome.com

超星公司制作 请尊重作者版权

```
            for each v ∈ PVVs (v ≠ u) do
               if v ∈ PVBinds(u)
                  & PVBinds(u) ≠ PVBinds(v) then
                  PVBinds(u) ∪= PVBinds(v)
                  change := true
               fi
            od
         od
      until !change
      || add nodes and edges to the call graph
      for each ⟨p,i,q⟩ ∈ PVCalls do
         for each u ∈ PVVals(q) do
            N ∪= {u}
            E ∪= {⟨p,i,q⟩}
            W ∪= {u}
         od
         for each u ∈ PVBinds(q) do
            for each v ∈ PVVals(u) do
               N ∪= {v}
               E ∪= {⟨p,i,v⟩}
               W ∪= {v}
            od
         od
      od
   until !more
end    || Build_Call_Graph_with_PVVs
```

图19-5 （续）

```
proc_const, proc_var: Operand ⟶ boolean
nparams: Procedure ⟶ integer
param: (Procedure × integer) ⟶ Operand
arg: (Procedure × integer × integer) ⟶ Operand
in_param, out_param: (Procedure × integer) ⟶ Operand

procedure Process_Inst(p,i,Inst) returns boolean
   p: in Procedure
   i: in integer
   Inst: in Procedure ⟶ array [··] of Instruction
begin
   q, u, v: ProcVar ∪ Procedure
   j: integer
   more := false: boolean
   || accumulate calls
   for each q ∈ callset(p,i) do
      more V= Process_Call(p,i,q)
      if preturns(p,i,u,q,v) then
         if proc_const(v) then
            PVVs ∪= {u}
            PVVals(u) ∪= {v}
            if v ∉ N then
               more := true
            fi
         else
            PVVs ∪= {u,v}
```

图19-6 在构造含过程值变量的调用图中使用的过程Process_Inst()

bbs.theithome.com

```
            PVBinds(u) ∪= {v}
         fi
      fi
   od
   if Inst(p)[i].kind = valasgn
      & proc_var(Inst(p)[i].left) then
      || accumulate bindings
      if proc_const(Inst(p)[i].opd.val) then
         PVVs ∪= {Inst(p)[i].left}
         PVVals(Inst(p)[i].left) ∪= {Inst(p)[i].opd.val}
         if Inst(p)[i].opd.val ∉ N then
            more := true
         fi
      else
         PVVs ∪= {Inst(p)[i].left,Inst(p)[i].opd.val}
         PVBinds(Inst(p)[i].left) ∪= {Inst(p)[i].opd.val}
      fi
   fi
   return more
end    || Process_Inst
```

图19-6 （续）

```
procedure Process_Call(p,i,q) returns boolean
   p: in Procedure
   i: in integer
   q: in ProcVar ∪ Procedure
begin
   j: integer
   more := false: boolean
   if proc_const(q) then
      || add nodes and edges to the call graph
      N ∪= {q}
      E ∪= {⟨p,i,q⟩}
      W ∪= {q}
|| deal with passing procedure-valued objects as parameters
for j := 1 to nparams(q) do
   if proc_var(param(q,j)) & in_param(q,j) then
      if proc_const(arg(p,i,j)) then
         PVVs ∪= {param(q,j)}
         PVVals(param(q,j)) ∪= {arg(p,i,j)}
         if arg(p,i,j) ∉ N then
            more := true
         fi
      else
         PVVs ∪= {param(q,j),arg(p,i,j)}
         PVBinds(param(q,j)) ∪= {arg(p,i,j)}
      fi
   fi
   || and return of procedure-valued objects
   if proc_var(param(q,j)) & out_param(q,j) then
      if proc_const(arg(p,i,j)) then
         PVVs ∪= {arg(p,i,j)}
         PVVals(arg(p,i,j)) ∪= {param(q,j)}
         if param(q,j) ∉ N then
            more := true
```

图19-7 在构造含过程值变量的调用图中使用的过程Process_Call()

bbs.theithome.com

```
                fi
             else
                PVVs ∪= {param(q,j),arg(p,i,j)}
                PVBinds(arg(p,i,j)) ∪= {param(q,j)}
             fi
          fi
       od
    else
       PVVs ∪= {q}
       PVCalls ∪= {⟨p,i,q⟩}
    fi
    return more
end    || Process_Call
```

图19-7 （续）

程序中有9种与过程值变量有关的行为，这9种行为以及我们对它们采取的初始动作如下： 612~615

1. 在过程*p*的*s*点调用过程值变量*vp*：将三元组<*p*, *s*, *vp*>放到PVCalls中。

2. 过程常数*p*作为实参与一个过程值形参*vp*结合：将偶对<*vp*, *p*>放到有限函数PVVals中，并将*vp*放到PVVs中。

3. 过程值变量*vp*作为实参与一个过程值形参*fp*结合：将偶对<*vp*, *fp*>放到有限函数PVBinds中，并将这两个过程变量放到PVVs中。

4. 将一个过程值变量*vp*1赋给另一个过程值变量*vp*2：将偶对<*vp*1, *vp*2>放到PVBinds中，并将这两个过程变量放到PVVs中。

5. 将一个过程值常数*p*赋给一个过程值变量*vp*：将偶对<*vp*, *p*>放到PVVals中，并将*vp*放到PVVs中。

6. 过程值变量*vp*与一个调用的过程值输出参数*fp*结合：将偶对<*vp*, *fp*>放到PVBinds中，并将这两个过程变量放到PVVs中。

7. 过程值常数*p*与一个调用的过程值输出参数*fp*结合：将偶对<*fp*, *p*>放到PVVals中，并将*fp*放到PVVs中。

8. 返回一个过程值变量*vp*1给一个过程值变量*vp*2：将偶对<*vp*1, *vp*2>放到PVBinds中，并将这两个过程变量放到PVVs中。

9. 返回一个过程值常数*p*给一个过程值变量*vp*：将偶对<*vp*,*p*>放到PVVals中，并将*vp*放到PVVs中。

在这些初始动作之后，我们根据PVCalls、PVVals和PVBinds为每一个过程值变量构造能够与之结合、给它赋值、或返回给它的过程值变量集合；然后对每一个过程值变量的调用点、以及对每一个可能作为其值与之结合的过程常数构造调用图中的一条边。Build_Call_Graph_with_PVVs()的最外层循环考虑到了程序的一个或多个部分可能仅当借助于调用过程值变量才能到达的情况。

该算法中用到的过程如下：

1. numinsts(*p*)是过程*p*中的指令条数。

2. Inst(*p*)[1..numinsts(*p*)]是构成过程*p*的指令组成的数组。

3. callset(*p*, *i*)返回由过程*p*的第*i*条指令调用的过程集合。

4. proc_const(*p*)返回true，如果*p*是一个过程常数；否则返回false。

5. proc_var(*p*)返回true，如果*p*是一个过程值变量；否则返回false。

6. nparams(*p*)返回过程*p*的形参个数。

7. arg(*p*, *i*, *j*)返回第*i*条指令调用的那个过程*p*的第*j*个实参。

616

版权所有 翻印必究

bbs.theithome.com

8. param(*p*, *j*) 返回过程*p*的第*j*个形参。

9. in_param(*p*, *i*)返回true，如果*p*的第*i*个参数从它的调用者接收一个值；否则返回false。

10. out_param(*p*, *i*)返回true，如果*p*的第*i*个参数返回一个值给它的调用者；否则返回false。

11. preturns(*p*, *i*, *u*, *q*, *v*) 返回true，如果*p*中第*i*条指令调用过程*q*，*q*返回其值于变量*v*，而*v*再赋给*p*的变量*u*。

这种传播过程产生的是一种保守的近似调用图，但对于递归程序可能需要指数量级的时间，因此应用它时需要小心。注意，我们产生的是流不敏感可能调用信息；也可以使它是流敏感的，但那样会导致增加额外的计算代价。还应注意的是，还存在一种我们未处理的实际情况，即从库例程反过来调用程序中的过程的情形。

作为构造含过程值变量程序调用图的例子，考虑图19-8所示的程序框架。假定f()是主过程，一开始我们有N=W={f}；并且E、PVVs、PVCalls、PVVals和PVBinds都为空。Build_Call_Graph_with_PVVs(f, Inst, numinsts)中紧嵌在最外层repeat循环之内的while循环设置p=f、W=∅、i=1，并调用Process_Inst(f, 1, Inst)。Process_Inst()调用Process_Call(f,1,g)，后一过程设置N={f, g}、E={<f, 1, g>}和W={g}，并返回false。然后，过程Process_Inst()返回false。

617

接下来，这个while循环设置p为g，W为∅，i为1，并调用Process_Inst(g, 1, Inst)。Process_Inst()确定出Inst(g)[1]是“p:=h”，且h是一个过程常数，于是设置PVVs={p}和PVVals(p)={h}，并返回true。因此Build_Call_Graph_with_PVVs()中的变量more被置成true。

接着，i被置成2，并调用过程Process_Inst(g, 2, Inst)。它调用Process_Call(g, 2, p)，此过程设置PVVs={p}和PVCalls={<g, 2, p>}，并返回false。

接着，i被置成3，过程Process_Inst(g, 3, Inst)被调用。它接着调用Process_Call(g, 3, j)，导致N={f, g, j}、E={<f, 1, g>、<g, 3, j>}、W={j}和PVVals(a)={i}，并返回true。

接着，p被置成j，W被置成∅，i被置成1，然后调用Process_Inst(j, 1, Inst)，它进一步调用Process_Call(j, 1, a)。Process_Call()设置PVVs={p, a}和PVCall={<g, 2, p>, <j, 1, a>}，并返回false。

现在W为空，因此主例程进入图19-5用星号标志的内层repeat循环。因为不满足内层for循环内的条件，因此主例程进入repeat循环下面的这个for循环，这个循环设置N={f, g, j, h, i}、E={<f, 1, g>、<g, 3, j>、<g, 2, h>、<j, 1, i>}、W={h, i}。

因为more=true，故重复主循环，这一次遗留N为原样，设置E={<f, 1, g>, <g, 3, j>, <g, 2, h>, <j, 1, i>, <h, 1, i>, <i, 1, g>}，并设置W=∅。数据结构没有进一步改变，于是得到图19-9所示的调用图。

```
        procedure f( )
f       begin
 1          call g( )
        end
        procedure g( )
        begin
g           p: Procedure
 1          p := h
 2          call p( )
 3          call j(i)
        end
        procedure h( )
h       begin
 1          call i( )
        end
        procedure i( )
i       begin
 1          call g( )
        end
        procedure j(a)
            a: Procedure
j       begin
 1          call a( )
        end
```

图19-8 一个含过程值变量的程序框架示例

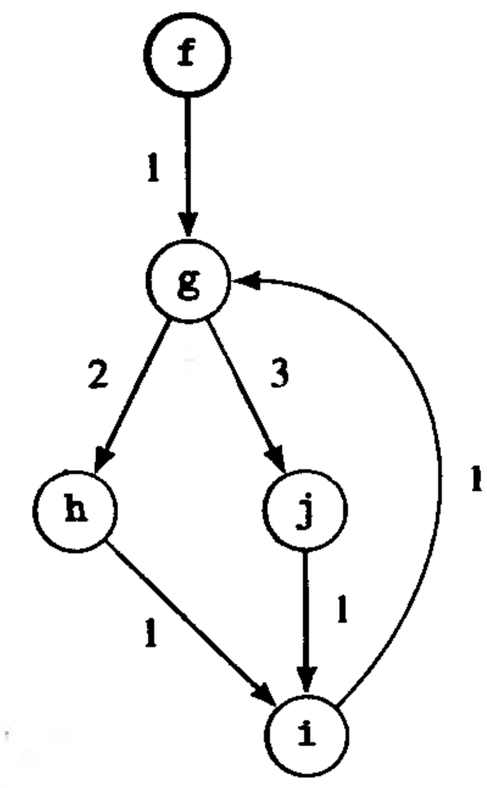


图19-9 图19-8中所示含过程值变量的程序框架的调用图

bbs.theithome.com

当然，对于那种在运行时创建新结点的程序，不可能构造它的完整的静态调用图。 618

19.2 过程间数据流分析

过程间数据流分析涉及的问题是，确定程序是如何在其调用图级别管理数据的保守然而有用的近似信息。这些信息中最有用的一般包括：(1) 哪些变量可能因为过程调用的副作用而被修改；以及(2) 当在一个特定点调用被调过程时，它的哪些参数具有常数值。知道了副作用的可能范围，只要避免那些可能受被调用过程影响的变量，我们就能自由地优化此调用周围的代码。知道了一个给定调用的一个或多个参数具有特定的常数值，我们就能判别出在那个调用点克隆调用过程的过程体，以及根据参数的常数值对它进行优化是否有用。这些信息也可能有助于判定与过程集成有关的问题（参见15.2节）。

当然，与过程内分析一样，别名也会使过程间分析复杂化。我们在19.4节讨论别名计算，并讨论如何将其结果集成到忽略别名的过程间数据流分析中。

19.2.1 流不敏感副作用分析

如前面提到的，过程间副作用分析的目的是，对于每一个调用点，判别在那一点调用的过程可能具有的副作用的安全近似信息。这种近似信息中包含有在那一点被调过程的所有副作用信息。

我们用4个从指令映射到变量集合的函数来刻画副作用，用 <过程名，指令编号> 的偶对来表示指令以及调用点。这些函数是：

DEF, *MOD*, *REF*, *USE*: Procedure × integer → set of Var

具体如下：

DEF (*p*, *i*)＝过程*p*的第*i*条指令一定定值的(即肯定会赋值的)变量集合。

MOD (*p*, *i*)＝过程*p*的第*i*条指令可能修改的(即可能赋值的)变量集合。

REF (*p*, *i*)＝过程*p*的第*i*条指令可能引用的(即可能获取其值的)变量集合。

USE (*p*, *i*)＝过程*p*的第*i*条指令在被该指令重新定值之前可能引用的(即可能获取其值的)变量集合。

这些函数对于那些不涉及调用的指令，只要忽略别名就很容易计算。有关忽略别名的问题推迟到19.4节再讨论。为了表示这些忽略了别名的集合，我们在集合的名字之前加上前缀*D*，即，忽略别名的*MOD*是*DMOD*。例如*DDEF*(*p*, *i*)，对于赋值指令*v*←*exp*，它是{*v*}，对于谓词*pred*，它是∅，其中假定这个表达式和谓词都不含调用。类似地，此赋值指令的*DREF*(*p*, *i*)是在*exp*中出现的变量集合，对于谓词*pred*，它是在*pred*中出现的变量集合。 619

作为一个例子，我们考虑关于*DMOD*的计算，所使用的方法基于Cooper和Kennedy的工作(参见19.10节的引文)。

在整个这一节中，我们都使用图19-10中的程序作为例子。它的静态嵌套结构和调用图在图19-11中给出。注意该程序有3个不同的j——main()中的全局变量、g()的局部变量和m()的形参，在此以后我们分别将它们记为j_1、j_2和j_3。为了允许完全通用的变量和参数命名，我们本来需标识出每一个变量所嵌套的过程序列，例如，我们可以记全局变量j为<j，[main]>，g()的局部变量j为<j,[main,g]>，m()的参数j为<j,[main,g,n,m]>。

bbs.theithome.com

```
                procedure main( )
                    global i, j, e
                    procedure f(x)
f                   begin
 1                      i := 2
 2                      j := g(i,x)
 3                      if x = 0 then
 4                          j := x + 1
 5                      fi
 6                      return j
                    end    || f
                    procedure g(a,b)
                        local j
                        procedure n(k,l)
                            procedure m(w,y,j)
m                           begin
 1                              j := n(w,i)
 2                              return y + j
                            end     || m
n                       begin
 1                          k := j + 1
 2                          y := y - 2
 3                          return k + m(i,i,k)
                        end     || n
g                   begin
 1                      if a > 1 then
 2                          a := b + 2
 3                      else
 4                          b := h(a,b)
 5                      fi
 6                      return a + b + n(b,b)
                    end    || g
                    procedure h(c,d)
h                   begin
 1                      e := e/g(1,d)
 2                      return e
                    end     || h
main            begin
 1                  call f(i)
                end     || main
```

图19-10 本节所有讨论都使用的一个例子

我们首先给出如下定义:

1. *LMOD*(*p*, *i*)是可能因过程*p*的第*i*条指令的执行而被局部修改的变量集合（不包括出现在该指令中的过程调用的副作用）。

2. *IMOD*(*p*)是可能因过程*p*的执行（但不执行*p*中的任何调用）而被修改的变量集合。

3. $IMOD^{+}$ (*p*)是可能因过程*p*的执行而被直接修改的、或作为传地址参数传给其他过程后因过程调用副作用而被修改的变量集合。

4. *GMOD*(*p*)是可能被过程*p*的调用修改的所有变量的集合。

5. *RMOD*(*p*)是可能因过程*p*的调用副作用而被修改的*p*的形式参数集合。

6. *Nonlocals*(*p*)是过程*p*中可见的非局部于它的变量集合。

7. *Formals*(*p*)是过程*p*的形式参数集合。

8. *numinsts*(*p*)是过程*p*的指令条数。

bbs.theithome.com

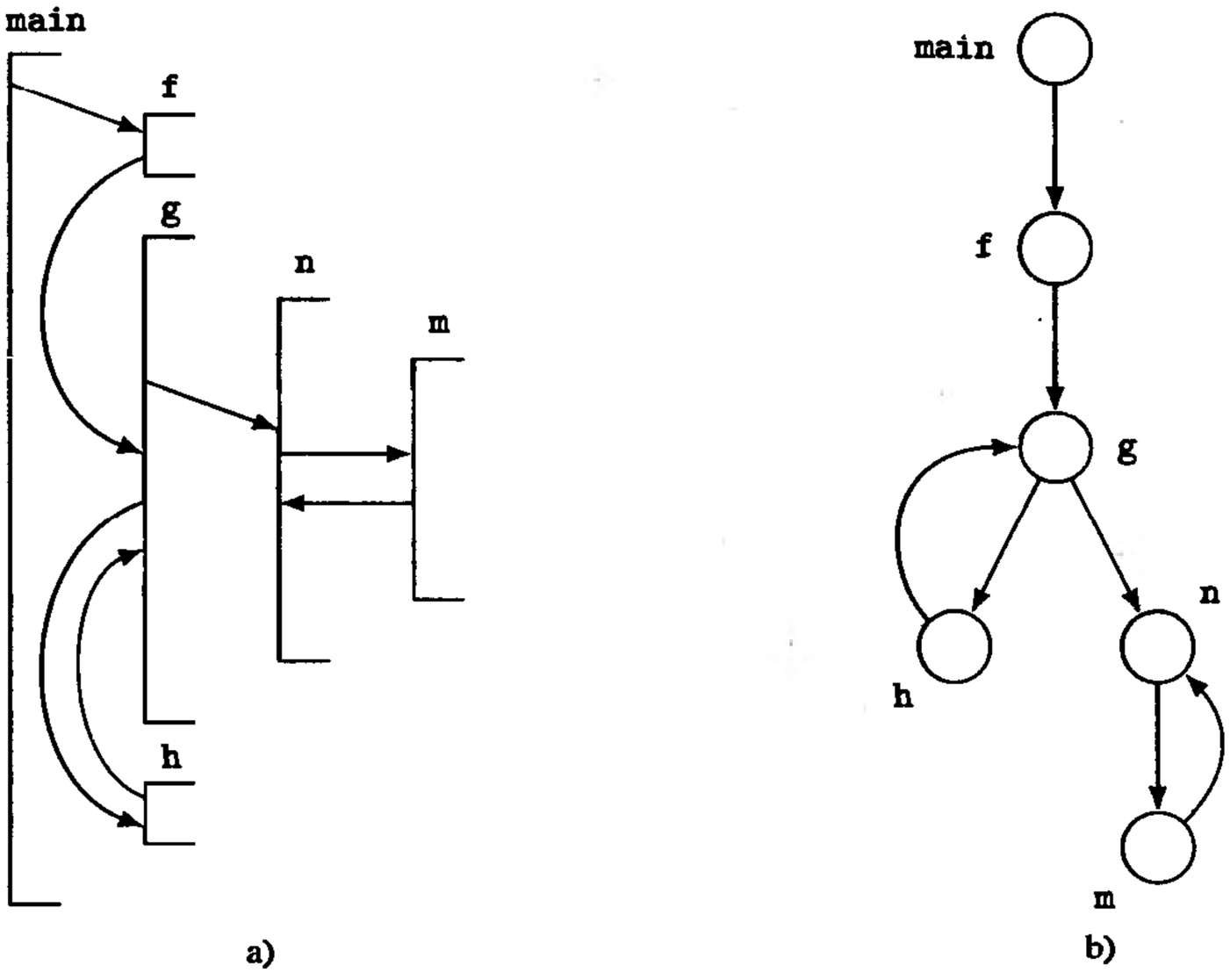


图19-11 图19-10中程序的a) 静态嵌套结构和b) 它的调用图

9. *callsites*(*p*)是使得过程*p*中第*i*条指令调用某个过程的整数*i*组成的集合。

10. *callset*(*p*, *i*)是过程*p*的第*i*条指令调用的过程集合[⊖]。

11. *nested*(*p*)是静态嵌套在过程*p*内的过程集合。

12. $b_{p,i,q}()$是一个函数，它将过程*q*的形式参数映射到过程*p*的第*i*条指令之处与它们相对应的形式参数，假定此处含有对*q*的一个调用。全局于*p*和*q*的变量也由*p*向*q*传送。

13. *Inst*()是一个函数，对于过程*p*，它按构成*p*的指令的顺序产生由这些指令组成的数组，即*Inst*(*p*)[1..*numinsts* (*p*)]。

现在*DMOD*(*p*, *i*)可以计算为*LMOD*(*p*, *i*)（对于任意指令不难计算它）与这样一个集合的并集，这个集合由过程*p*的第*i*条指令调用的所有过程*q*的广义修改集合*GMOD*()的并集经$b_{p,i,q}()$函数（它映射*q*的形参到*p*的对应形参）过滤后而形成，即

$$DMOD(p,i) = LMOD(p,i) \cup \bigcup_{q \in callset(p,i)} b_{p,i,q}(GMOD(q))$$

GMOD(*p*)可以计算为*p*中所有调用点的*GMOD*()值（限制为非局部变量）的并集与$IMOD^+(p)$的并集，即

$$GMOD(p) = IMOD^+(p) \cup \bigcup_{1 \le i \le numinsts(p)} \bigcup_{q \in callset(p,i)} GMOD(q) \cap Nonlocals(q)$$

$IMOD^+(p)$是*IMOD*(*p*)与这样一个集合的并集，这个集合是由*p*调用的所有过程*q*的所有*RMOD*(*q*)值经$b_{p,i,q}()$函数过滤后的并集，即 620 ~ 622

$$IMOD^+(p) = IMOD(p) \cup \bigcup_{i \in callsites(p)} \bigcup_{q \in callset(p,i)} b_{p,i,q}(RMOD(q))$$

最后，*IMOD*(*p*)，即

⊖ 注意，在我们的中间语言HIR、MIR和LIR中，一条指令至多调用一个过程。但我们允许每条指令有多个调用，以说明如何包容这种情况。

bbs.theithome.com

$$IMOD(p) = \bigcup_{1 \le i \le numinsts(p)} LMOD(p, i)$$

注意，*IMOD*的计算很容易，因此有效地计算*DMOD*简化为有效地计算*RMOD*和*GMOD*。

为了计算*RMOD*，我们使用一个称为结合图（binding graph）的数据结构。尽管可以通过标准的过程间数据流分析来计算*RMOD*，但这里介绍的方法更有效。程序*P*的结合图是$B=<N_B, E_B>^{\ominus}$，其中N_B中的结点表示*P*的形式参数，E_B中的边表示调用过程的形式参数与被调用过程的参数结合。如果在*p*中存在某个调用点调用了*q*，且它导致*p*的形参*x*与*q*的形参*y*结合，则在E_B中存在一条边$x \rightarrow y$——或以传递方式通过若干次调用，导致*x*与某个过程*r*的形式参数*z*结合，则E_B中也存在着一条边$x \rightarrow z$。注意，只有用一个过程的形参作为被调过程的实参才会在E_B中生成边。

计算*IMOD*()的ICAN例程在图19-12中给出。对于我们的示例程序，表19-1给出了*Nonlocals*和*IMOD*的值。

```
IMOD: Procedure ⟶ set of Var

procedure Compute_IMOD(p,P)
   p: in Procedure
   P: in set of Procedure
begin
   V := ∅: set of Var
   i: integer
   q: Procedure
   || apply data-flow equations to compute IMOD
   for i := 1 to numinsts(p) do
      V ∪= LMOD(p,i)
   od
   IMOD(p) := V
end    || Compute_IMOD
```

图19-12 计算程序*P*的*IMOD*()的ICAN算法

表19-1 图19-10中程序的*Nonlocals* ()和*IMOD* ()的值

Nonlocals(main)	$=\emptyset$	*IMOD*(main)	$=\emptyset$
Nonlocals(f)	$=\{e,i,j_1\}$	*IMOD*(f)	$=\{i,j_1\}$
Nonlocals(g)	$=\{e,i,j_1\}$	*IMOD*(g)	$=\{a,b\}$
Nonlocals(n)	$=\{a,b,e,i,j_2\}$	*IMOD*(n)	$=\{k,l\}$
Nonlocals(m)	$=\{a,b,e,i,k,l\}$	*IMOD*(m)	$=\{j_3\}$
Nonlocals(h)	$=\{e,i,j_1\}$	*IMOD*(h)	$=\{e\}$

图19-13中的ICAN例程Build_Binding_Graph()构造程序*P*的结合图。nparams(*p*)的值是*p*的形式参数个数，params(*p*, *i*)是*p*的第*i*个形式参数。passed(*p*, *i*, *q*, *x*, *y*)为true，如果过程*p*的第*i*条指令调用过程*q*并且使得*p*的形参*x*与*q*的形参*y*结合；否则为false。

为了建立图19-10中程序的结合图，我们一开始设置P＝{main, f, g, h, n, m}，N＝∅，E＝∅，oldE＝∅。假设首先处理的是过程main()，我们在第1行遇到了一个调用，因此有P＝main、i＝1和q＝f，于是调用Bind_Pairs(main, 1, f)，它没有改变N和E。main()中没有遇到其他调用，因此继续处理下一个过程f()，它将x和b加到N中，边x→b加到E中，得到

623

N = {x, b}　　E = {x→b}

⊖ 这是希腊大写字母B，不是罗马字符B。

bbs.theithome.com

```
procedure Build_Binding_Graph(P,N,E)
   P: in set of Procedure
   N: out set of Var
   E: out set of (Var × Var)
begin
   p, q: Procedure
   i, m, n: integer
   e, f: Var × Var
   oldE: set of (Var × Var)
   || construct graph that models passing parameters
   || as parameters to other routines
N := E := oldE := ∅
repeat
   oldE := E
   for each p ∈ P do
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            Bind_Pairs(p,i,q,N,E)
         od
      od
   od
until E = oldE
repeat
   oldE := E
   for each e ∈ E do
      for each f ∈ E (f ≠ e) do
         if e@2 = f@1 & e@1 ≠ f@2 then
            E ∪= {e@1→f@2}
         fi
      od
   od
   until E = oldE
end    || Build_Binding_Graph

procedure Bind_Pairs(p,i,q,N,E)
   p, q: in Procedure
   i: in integer
   N: inout set of Var
   E: inout set of (Var × Var)
begin
   m, n: integer
   for m := 1 to nparams(p) do
      for n := 1 to nparams(q) do
         if passed(p,i,q,param(p,m),param(q,n)) then
            N ∪= {param(p,m),param(q,n)}
            E ∪= {param(p,m)→param(q,n)}
         fi
      od
   od
end    || Bind_Pairs
```

图19-13　构造程序*P*的结合图B的算法

接下来我们处理g()，它将a、c、d、k和1加入到N，并加入若干条边到E，得到如下结果:

N = {x, b, a, c, d, k, 1}　　E = {x→b, a→c, b→d, b→k, b→1}

处理h()没有改变N，但改变了E，结果如下:

N = {x, b, a, c, d, k, 1}

bbs.theithome.com

版权所有 请勿制作盗版

```
E = {x→b, a→c, b→d, b→k, b→l, d→b}
```

继续这一处理过程直到Build_Binding_Graph()的最后一个顶层循环将那些传递边加入到E中，最终得到

```
N = {x,b,a,c,d,k,l,w,x,j3}
E = {x→b, a→c, b→d, b→k, b→l, d→b, d→k, d→l, k→j3,
     d→j3, b→j3, w→k, w→j3, x→d, x→k, x→j3, x→l}
```

和图19-14中的图形表示。注意，如这个例子所示，结合图通常并不是连在一起的，并且仅当程序是递归的才含有环路。

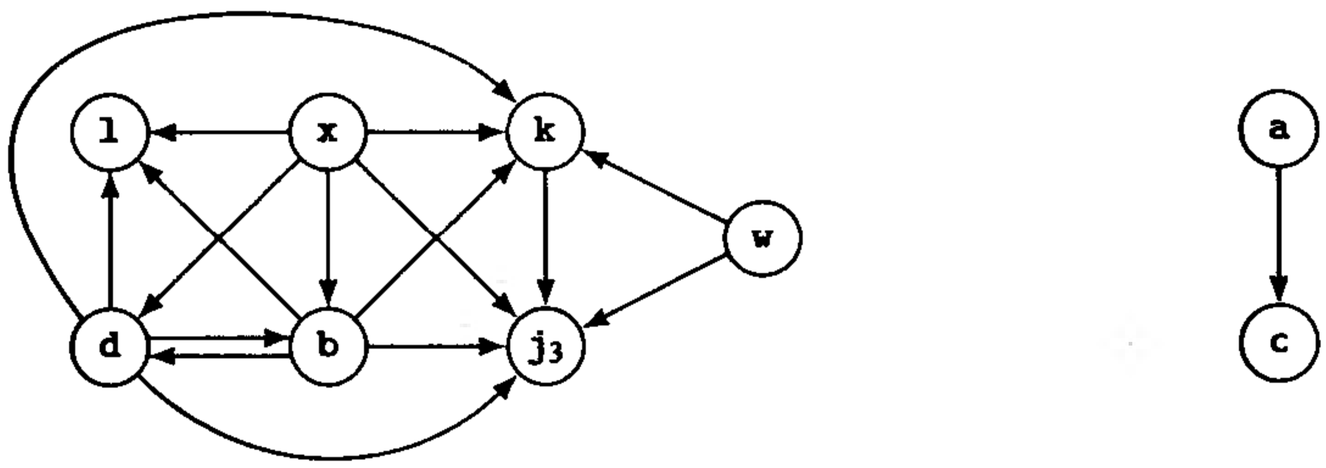


图19-14 图19-10中程序的结合图

下面，我们根据程序的结合图来定义*RMOD*。从N_B中的结点映射到布尔值的函数*RBMOD*是：如果*x*因为调用某个过程的副作用而被修改，*RBMOD*(*x*)为true，否则为false。计算*RBMOD*可通过在结合图上初始化所有*x*的*RBMOD*(*x*)为false，解如下数据流方程来完成。

$$RBMOD(x) = \exists p \in P\ (x \in Formals(p) \cap IMOD(p)) \vee \bigvee_{x \to y \in E_B} RBMOD(y)$$

算法Comput_RBMOD(*P*, *N*, *E*)给出在图19-15中，其中，*P*是调用图中的过程集合，*N*和*E*分别是结合图的结点集合和边集合。最外层的循环应当按后序执行，即，结合图中每一个结点的处理应尽可能地在其后继结点已处理之后。

```
RBMOD: Var ⟶ boolean

procedure Compute_RBMOD(P,N,E)
   P: in set of Procedure
   N: in set of Var
   E: in set of (Var × Var)
begin
   p: Procedure
   n: Var
   e: Var × Var
   oldRBMOD: Var ⟶ boolean
   RBMOD := ∅
   repeat
      oldRBMOD := RBMOD
      for each n ∈ N do
         RBMOD(n) := ∃p ∈ P (n ∈ Formals(p) ∩ IMOD(p))
         for each e ∈ E (e@1 = n) do
             RBMOD(n) V= RBMOD(e@2)
         od
      od
   until RBMOD = oldRBMOD
end    || Compute_RBMOD
```

图19-15 根据结合图B计算*RBMOD*()的算法

bbs.theithome.com

版权所有 侵权必究

624 ~ 626

这样，过程p的$RMOD(p)$就简单地是使得$RBMOD(x)$为true的p的形参x组成的集合，即

$$RMOD(p) = \{ x \mid x \in Formals(p) \ \&\ RBMOD(x)\}$$

作为计算$RBMOD$的一个例子，我们用图19-10程序例子来计算它，这个程序的结合图是图19-14。为了计算$RMOD$，我们调用Compute_RBMOD(P, N, E)，其中

```
P = {main,f,g,h,n,m}
N = {x,b,a,c,d,k,l,w,z,j}
E = {x→b, a→c, b→d, b→k, b→l, d→b, d→k, d→l, k→j3,
     d→j3, b→j3, w→k, w→j3, x→d, x→k, x→j3, x→l}
```

这个算法初始结合图中每一个结点的$RBMOD()$如表19-2所示，然后迭代地计算每一个结点的$RBMOD()$直到它收敛为止[⊖]，由此得到表19-3所示的值。

表19-2 对于图19-10中例子的结合图中的结点．通过计算$RBMOD()$方程中的存量表达式而得到的初始值

$RBMOD(x)$	= false
$RBMOD(b)$	= true
$RBMOD(a)$	= true
$RBMOD(c)$	= false
$RBMOD(d)$	= false
$RBMOD(k)$	= true
$RBMOD(l)$	= false
$RBMOD(w)$	= false
$RBMOD(y)$	= true
$RBMOD(j_3)$	= true

表19-3 图19-10中例子的结合图结点的$RBMOD()$最终值

$RBMOD(x)$	= false ∨ $RBMOD(b)$ ∨ $RBMOD(j_3)$ = true
$RBMOD(b)$	= true
$RBMOD(a)$	= true
$RBMOD(c)$	= false
$RBMOD(d)$	= false ∨ $RBMOD(b)$ = true
$RBMOD(k)$	= true
$RBMOD(l)$	= false
$RBMOD(w)$	= false ∨ $RBMOD(j_3)$ = true
$RBMOD(y)$	= true
$RBMOD(j_3)$	= true

627

于是，$RMOD()$如表19-4所示。

回忆$IMOD^+()$的定义为

$$IMOD^+(p) = IMOD(p) \cup \bigcup_{q \in callset(p,i)} b_{p,i,q}(RMOD(q))$$

所以，正如图19-16所示那样，它很容易计算（注意，在这个算法中，我们用b(p,i,q,r)表示$b_{p,i,q}(r)$），并且对于我们的例子，其值如表19-5所示。

表19-4 图19-10中例子的$RMOD()$值

$RMOD$(main)	= ∅
$RMOD$(f)	= {x}
$RMOD$(g)	= {a,b}
$RMOD$(n)	= {k}
$RMOD$(m)	= {w,y,j_3}
$RMOD$(h)	= {d}

为了由$IMOD^+$有效地计算$GMOD$，我们用图19-17和图19-18中给出的算法，它们处理的是调用图(即Edges是调用图的边集合，而不是结合图的边集合)。

给定具有过程P、边集合Edges和主例程r的一个调用图，图19-17中的ICAN例程Stratify()和Set_Levels()构造一个将过程映射到它们的嵌套层次的函数Level()。对于程序的每一个0 ≤ i ≤ depth的嵌套层，Stratify()还计算具有结点N[i]、根R[i][⊖]、边E[i]的一个子图，其中一个例程属于N[i]，如果它调用的所有例程的嵌套层次都大于或等于i。例程called(p)返回过程p调用的所有过程的集合。

⊖ 注意，$RBMOD$的数据流方程定义在false⊏true的两元素格上。

⊖ 注意，子图可能有多个根，例如，当主例程（在第0层）调用了两个例程（在第1层）时。

bbs.theithome.com

```
IMODPLUS: Procedure ⟶ set of Var

procedure Compute_IMODPLUS(P)
    P: in set of Procedure
begin
    p, q: Procedure
    i: integer
    IMODPLUS := ∅
    for each p ∈ P do
       IMODPLUS(p) := IMOD(p)
        for each i ∈ callsites(p) do
            for each q ∈ callset(p,i) do
               IMODPLUS(p) ∪= b(p,i,q,RMOD(q))
            od
        od
    od
end     || Compute_IMODPLUS
```

图19-16 计算$IMOD^+$()的ICAN算法

表19-5 图19-10中例子的$IMOD^+$值

$IMOD^+(\mathtt{main})$	$= IMOD(\mathtt{main}) \cup b_{\mathtt{main},1,\mathtt{f}}(RMOD(\mathtt{f})) = \{\mathtt{i},\mathtt{j}_1\} \cup \{\mathtt{i}\} = \{\mathtt{i},\mathtt{j}_1\}$
$IMOD^+(\mathtt{f})$	$= IMOD(\mathtt{f}) \cup b_{\mathtt{f},2,\mathtt{g}}(RMOD(\mathtt{g})) = \{\mathtt{i},\mathtt{j}_1\} \cup \{\mathtt{i},\mathtt{x}\} = \{\mathtt{i},\mathtt{j}_1,\mathtt{x}\}$
$IMOD^+(\mathtt{g})$	$= IMOD(\mathtt{g}) \cup b_{\mathtt{g},4,\mathtt{h}}(RMOD(\mathtt{h})) \cup b_{\mathtt{g},6,\mathtt{n}}(RMOD(\mathtt{n})) = \{\mathtt{a},\mathtt{b}\} \cup \{\mathtt{b}\} \cup \{\mathtt{b}\} = \{\mathtt{a},\mathtt{b}\}$
$IMOD^+(\mathtt{n})$	$= IMOD(\mathtt{n}) \cup b_{\mathtt{n},3,\mathtt{m}}(RMOD(\mathtt{m})) = \{\mathtt{k},\mathtt{y}\}$
$IMOD^+(\mathtt{m})$	$= IMOD(\mathtt{m}) \cup b_{\mathtt{m},1,\mathtt{n}}(RMOD(\mathtt{n})) = \{\mathtt{j}_3\} \cup \{\mathtt{w}\} = \{\mathtt{j}_3,\mathtt{w}\}$
$IMOD^+(\mathtt{h})$	$= IMOD(\mathtt{h}) \cup b_{\mathtt{h},1,\mathtt{g}}(RMOD(\mathtt{g})) = \{\mathtt{e}\} \cup \{\mathtt{d}\} = \{\mathtt{e},\mathtt{d}\}$

628

```
N, R: array [··] of set of Procedure
E: array [··] of set of (Procedure × Procedure)
Level := ∅: Procedure ⟶ integer
depth, NextDfn: integer
Dfn: Procedure ⟶ integer
LowLink: Procedure ⟶ integer
GMOD, IMODPLUS, Nonlocals: Procedure ⟶ set of Var
Stack: sequence of Procedure

procedure Stratify(P,Edges,r) returns integer
   P: in set of Procedure
   Edges: in set of (Prooedure × Procedure)
   r: in Procedure
begin
   i, j, depth: integer
   p, q:  Procedure
   WL := P: set of Procedure
   Level(r) := 0
   Set_Levels(r)
```

图19-17 计算调用图和子图(由结点N[]、根R[]和边E[]组成)的嵌套层次Level()的算法。其中，子图不包括调用了较低嵌套层例程的那些例程。Stratify()用Set_Levels_to计算每一个过程的嵌套层次

bbs.theithome.com

```
   depth := 0
   for i := 0 to depth do
      N[i] := ∅
      for j := 0 to i-1 do
         WL ∪= {p ∈ P where Level(p) = j}
      od
      for j := i to depth do
         for each p ∈ P (Level(p) = j) do
            if called(p) ∩ WL = ∅ then
               N[i] ∪= {p}
            fi
         od
         WL ∪= {p ∈ P where Level(p) = j}
      od
      E[i] := Edges ∩ (N[i] × N[i])
      R[i] := {p ∈ N[i] where Level(p) = i}
   od
   return depth
end    || Stratify

procedure Set_Levels(r)
   r: in Procedure
begin
   p: Procedure
   for each p ∈ nested(r) do
      Level(p) := Level(r) + 1
      if Level(p) > depth then
         depth := Level(p)
      fi
      Set_Levels(p)
   od
end    || Set_Levels
```

图19-17 （续）

```
procedure Compute_GMOD( )
begin
   i: integer
   n, r: Procedure
   for i := depth by -1 to 0 do
      for each r ∈ R[i] do
         NextDfn := 0
         for each n ∈ N[i] do
            Dfn(n) := 0
         od
         Stack := []
         GMOD_Search(i,r)
      od
   od
end    || Compute_GMOD
procedure GMOD_Search(i,p)
   i: in integer
   p: in Procedure
begin
   j: integer
```

图19-18 利用Tarjan的算法版本由*IMOD*+ ()有效地计算*GMOD*()的算法

bbs.theithome.com

```
u: Procedure
e: Procedure × Procedure
LowLink(p) := Dfn(p) := NextDfn += 1
GMOD(p) := IMODPLUS(p)
Stack ⊕= [p]
for each e ∈ E[i] (e@1 = p) do
   if Dfn[e@2] = 0 then
      GMOD_Search(i,e@2)
      LowLink(p) := min(LowLink(p),LowLink(e@2))
   fi
   if Dfn(e@2) < Dfn(p) & ∃j ∈ integer (Stack↓j=e@2) then
      LowLink(p) := min(Dfn(e@2),LowLink(p))
   else
      GMOD(p) ∪= GMOD(e@2) ∩ Nonlocals(e@2)
   fi
od
*  if LowLink(p) = Dfn(p) then
   repeat
      u := Stack↓-1
      Stack e= -1
      GMOD(u) ∪= GMOD(p) ∩ Nonlocals(p)
   until u = p
fi
end    || GMOD_Search
```

图19-18 （续）

对我们的例程序应用Stratify()，得到depth＝3和表19-6中的数据结构。

表19-6 图19-10例程序的Level()、N[]、R[]和E[]之值

```
Level(main) = 0
Level(f) = Level(g) = Level(h) = 1
Level(n) = 2
Level(m) = 3

N[0] = {main,f,g,h,n,m}        R[0] = {main}
N[1] = {f,g,h,n,m}             R[1] = {f}
N[2] = {n,m}                   R[2] = {n}
N[3] = {m}                     R[3] = {m}

E[0] = {main→f, f→g, g→h, g→n, h→g, n→m, m→n}
E[1] = {f→g, g→h, g→n, h→g, n→m, m→n}
E[2] = {n→m, m→n}
E[3] = ∅
```

*GMOD*的计算方法是Tarjan计算强连通分量方法的修改版本（参见7.4节）。其思想是，对于一个强连通分量——即相互递归的过程组成的集合——它的根结点的*GMOD*表示在此分量中的结点（即过程）中可能出现的所有的副作用，因此它的每一个结点的*GMOD*是该结点的*IMOD*⁺并上根结点的*GMOD*与此过程中的非局部变量集合的交集。这个算法的要点是，图19-18中带星号的那个测试为true，当且仅当p是调用图的强连通分量的根。

629～631

作为计算*GMOD*的一个例子，我们将这个算法应用于图19-10的程序。我们调用Compute_GMOD(main)，它执行深度为主查找并识别出强连通分量，同时初始*GMOD*的值如表19-7所示。然后，它按顺序累加*GMOD*的值，如表19-8所示。

bbs.theithome.com

表19-7 我们的例程序的*GMOD* ()的初值

$GMOD(\texttt{main}) = IMOD^{+}(\texttt{main})=\{\texttt{i},\texttt{j}_1\}$	
$GMOD(\texttt{f})$	$= IMOD^{+}(\texttt{f})=\{\texttt{i},\texttt{j}_1,\texttt{x}\}$
$GMOD(\texttt{g})$	$= IMOD^{+}(\texttt{g})=\{\texttt{a},\texttt{b}\}$
$GMOD(\texttt{n})$	$= IMOD^{+}(\texttt{n})=\{\texttt{k},\texttt{y}\}$
$GMOD(\texttt{m})$	$= IMOD^{+}(\texttt{m})=\{\texttt{j}_3,\texttt{w}\}$
$GMOD(\texttt{h})$	$= IMOD^{+}(\texttt{h})=\{\texttt{d},\texttt{e}\}$

表19-8 我们的例程序的*GMOD* ()的终值

$GMOD(\texttt{m})$	$= \{\texttt{j}_3,\texttt{w}\}$
$GMOD(\texttt{n})$	$= \{\texttt{k},\texttt{y}\}$
$GMOD(\texttt{h})$	$= \{\texttt{d},\texttt{e}\}$
$GMOD(\texttt{g})$	$= \{\texttt{a},\texttt{b}\} \cup (GMOD(\texttt{h}) \cap Nonlocals(\texttt{h}))$
	$= \{\texttt{a},\texttt{b}\} \cup \{\texttt{e}\}=\{\texttt{a},\texttt{b},\texttt{e}\}$
$GMOD(\texttt{f})$	$= \{\texttt{i},\texttt{j}_1,\texttt{x}\} \cup (GMOD(\texttt{g}) \cap Nonlocals(\texttt{g}))$
	$= \{\texttt{i},\texttt{j}_1,\texttt{x}\} \cup \{\texttt{e}\}=\{\texttt{e},\texttt{i},\texttt{j}_1,\texttt{x}\}$
$GMOD(\texttt{main})$	$= \{\texttt{i},\texttt{j}_1\} \cup (GMOD(\texttt{f}) \cap Nonlocals(\texttt{f}))$
	$= \{\texttt{i},\texttt{j}_1\} \cup \{\texttt{e}\}=\{\texttt{e},\texttt{i},\texttt{j}_1\}$

最后，我们应用*DMOD*的定义方程得到表19-9所示的值。

表19-9 我们的例程序的*DMOD* ()的值

$DMOD(\texttt{main}, 1) = LMOD(\texttt{main}, 1) = \emptyset$	
$DMOD(\texttt{f}, 1)$	$= LMOD(\texttt{f}, 1) = \{\texttt{i}\}$
$DMOD(\texttt{f}, 2)$	$= LMOD(\texttt{f}, 2) \cup b_{\texttt{f},2,\texttt{g}}(GMOD(\texttt{g}))$
	$= \{\texttt{j}_1\} \cup \{\texttt{e},\texttt{i},\texttt{x}\} = \{\texttt{e},\texttt{i},\texttt{j}_1,\texttt{x}\}$
$DMOD(\texttt{f}, 3)$	$= LMOD(\texttt{f}, 3) = \emptyset$
$DMOD(\texttt{f}, 4)$	$= LMOD(\texttt{f}, 4) = \{\texttt{j}_1\}$
$DMOD(\texttt{f}, 5)$	$= LMOD(\texttt{f}, 5) = \emptyset$
$DMOD(\texttt{f}, 6)$	$= LMOD(\texttt{f}, 6) = \emptyset$
$DMOD(\texttt{g}, 1)$	$= LMOD(\texttt{g}, 1) = \emptyset$
$DMOD(\texttt{g}, 2)$	$= LMOD(\texttt{g}, 2) = \{\texttt{a}\}$
$DMOD(\texttt{g}, 3)$	$= LMOD(\texttt{g}, 3) = \emptyset$
$DMOD(\texttt{g}, 4)$	$= LMOD(\texttt{g}, 4) \cup b_{\texttt{g},4,\texttt{h}}(GMOD(\texttt{h}))$
	$= \{\texttt{b}\} \cup \{\texttt{b},\texttt{e}\} = \{\texttt{b},\texttt{e}\}$
$DMOD(\texttt{g}, 5)$	$= LMOD(\texttt{g}, 5) = \{\texttt{a}\}$
$DMOD(\texttt{g}, 6)$	$= LMOD(\texttt{g}, 6) = \{\texttt{a}\}$
$DMOD(\texttt{h}, 1)$	$= LMOD(\texttt{h}, 1) \cup b_{\texttt{h},1,\texttt{g}}(GMOD(\texttt{g}))$
	$= \{\texttt{e}\} \cup \{\texttt{b}\} = \{\texttt{b},\texttt{e}\}$
$DMOD(\texttt{h}, 2)$	$= LMOD(\texttt{h}, 2) = \emptyset$
$DMOD(\texttt{n}, 1)$	$= LMOD(\texttt{n}, 1) = \{\texttt{k}\}$
$DMOD(\texttt{n}, 2)$	$= LMOD(\texttt{n}, 2) = \{\texttt{y}\}$
$DMOD(\texttt{n}, 3)$	$= LMOD(\texttt{n}, 3) = \emptyset$
$DMOD(\texttt{m}, 1)$	$= LMOD(\texttt{m}, 1) = \{\texttt{j}_3\}$
$DMOD(\texttt{m}, 2)$	$= LMOD(\texttt{m}, 2) = \emptyset$

这种计算流不敏感副作用方法的计算复杂度是$O(e \cdot n+d \cdot n^2)$，其中，n和e分别是调用图中结点的个数和边的条数，d是程序中词法嵌套的深度。通过将Compute_GMOD()中的多遍迭代合并为对调用图的一遍迭代，可以去掉因子d（参见19.11节练习19.2）。

632 ~ 633

类似的分解也可用于计算其他的流不敏感可能概要函数。

19.2.2 流敏感副作用：程序概要图

Myers [Myer81]证明了，只要考虑别名，计算流敏感副作用就是co-NP完全的。他也引入了一个叫做程序超图（supergraph）的模型，这种模型用于支持流敏感副作用的判定。

Callahan [Call88]给出了一种计算流敏感问题近似解的实际方法，这种方法基于所谓的程

bbs.theithome.com

越是公司制作请尊重作者版权

序概要图，它为图中由调用图和参数传递样式而得出的结点添加了对应过程内的流敏感作用信息。这种图的大小适中，其复杂度在调用图和Myers的超图之间。

这一节概要性地介绍程序概要图，以及它能够计算的流敏感副作用信息。我们假定参数传递规则是传地址。假设我们有一个由过程集合*P*构成的程序，于是，*P*的*程序概要图*（program summary graph）的组成是：每一个过程的每个形参有两个结点，叫做入口（entry）结点和出口（exit）结点；每一个调用点的每个实参有两个结点，叫做调用（call）结点和返回（return）结点；以及连接所有调用结点到入口结点，出口结点到返回结点，某些入口和返回结点到调用和出口结点的边。

更详细地，令*p*和*q*是两个过程，过程*p*中的指令*i*调用过程*q*。则*q*的形参是

param(*q*, 1), ..., param(*q*, nparams(*q*))

对*q*的调用的实参是

arg(*p*, i, 1), ..., arg(*p*, *i*, nparams(*q*))

对于每一个三元组<*p*, *i*, arg(*q*, *i*, *j*)>，存在着一个调用结点cl(*p*, *i*, arg(*q*, *i*, *j*))和一个返回结点rt(*p*, *i*, arg(*q*, *i*, *j*))。对于每一个偶对<*q*, params(*q*, *j*)>，存在着一个入口结点en(*q*, params(*q*, *j*))和一个出口结点ex(*q*, params(*q*, *j*))。存在一条从cl(*p*, *i*, arg(*q*, *i*, *j*))到en(*q*, params(*q*, *j*))的边和另一条从ex(*q*, params(*q*, *j*))到rt(*p*, *i*, arg(*q*, *i*, *j*))的边。同时还存在着如下一些边：

1. 从en(*p*, params(*p*, *j*))到cl(*p*, *i*, arg(*p*, *i*, *k*))的边，如果params(*p*, *j*)的值到达调用点<*p*, *i*>，并且在那里与arg(*p*, *i*, *k*)结合；

2. 从en(*p*, params(*p*, *j*))到ex(*p*, params(*p*, *k*))的边，如果params(*p*, *j*)在*p*的入口的值作为params(*p*, *k*)的值到达*p*的出口；

3. 从rt(*p*, *i*, arg(*p*, *i*, *j*))到cl(*p*, *k*, arg(*p*, *k*, *l*))的边，如果arg(*p*, *i*, *j*)在返回到*p*时的值作为arg(*p*, *k*, *l*)的值到达调用点<*p*, *k*>；

4. 从rt(*p*, *i*, arg(*p*, *i*, *j*))到ex(*p*, param(*p*, *k*))的边，如果在返回到*p*时，arg(*p*, *i*, *j*)的值作为param(*p*, *k*)的值到达*p*的出口。

我们假定：实参在调用点被使用，然后被杀死；过程入口定值所有的形参，并且过程返回使用所有的参数。

634

作为一个例子，考虑图19-19的程序。它的程序概要图给出在图19-20中，下面解释其中的某些边：

1. 从en(f,x)到ex(f,x)有一条边，因为x在f()入口的值可能到达它的出口。

2. 从en(f,x)到cl(f,5,x)有一条边，因为x在f()入口的值可能在调用点<f,5>被传递给g()的形参b。

3. 没有从en(f,x)到cl(f,3,j)的边，因为x在f()入口的值在调用点<f,3>不能传递给g()的形参b。

4. 从cl(h,1,d)到en(g,b)有一条边，因为在调用点<h,1>对g()的调用可能传递d的值给g()的形参b。

5. 从ex(g,b)到rt(f,5,x)有一条边，因为b的值在g()的出口与f()的形参x结合。

已知程序概要图，我们现在可以定义杀死（Kill）和使用（Use）属性了。一个变量*v*是被杀死的(*Kill*(*v*)为true)，如果无论控制流怎样它都一定会被修改；一个变量*v*是被使用的(*Use* (*v*)为true)，如果它的使用先于它被杀死。现在，被杀死的变量集合能够用如下数据流方程来确定：

bbs.theithome.com

```
            procedure f(x)
f           begin
 1              i := 2
 2              if x = 0 then
 3                  j := g(j,0)
 4              else
 5                  j := g(i,x)
 6              fi
 7              return j
            end   || f
            procedure g(a,b)
g           begin
 1              if a > 1 then
 2                  a := b + 2
 3              else
 4                  b := h(a,b)
 5              fi
 6              return a + b + j
            end   || g
            procedure h(c,d)
h           begin
 1              e := e/g(1,d)
 2              return e
            end   || h
```

图19-19 流敏感副作用计算的例程序

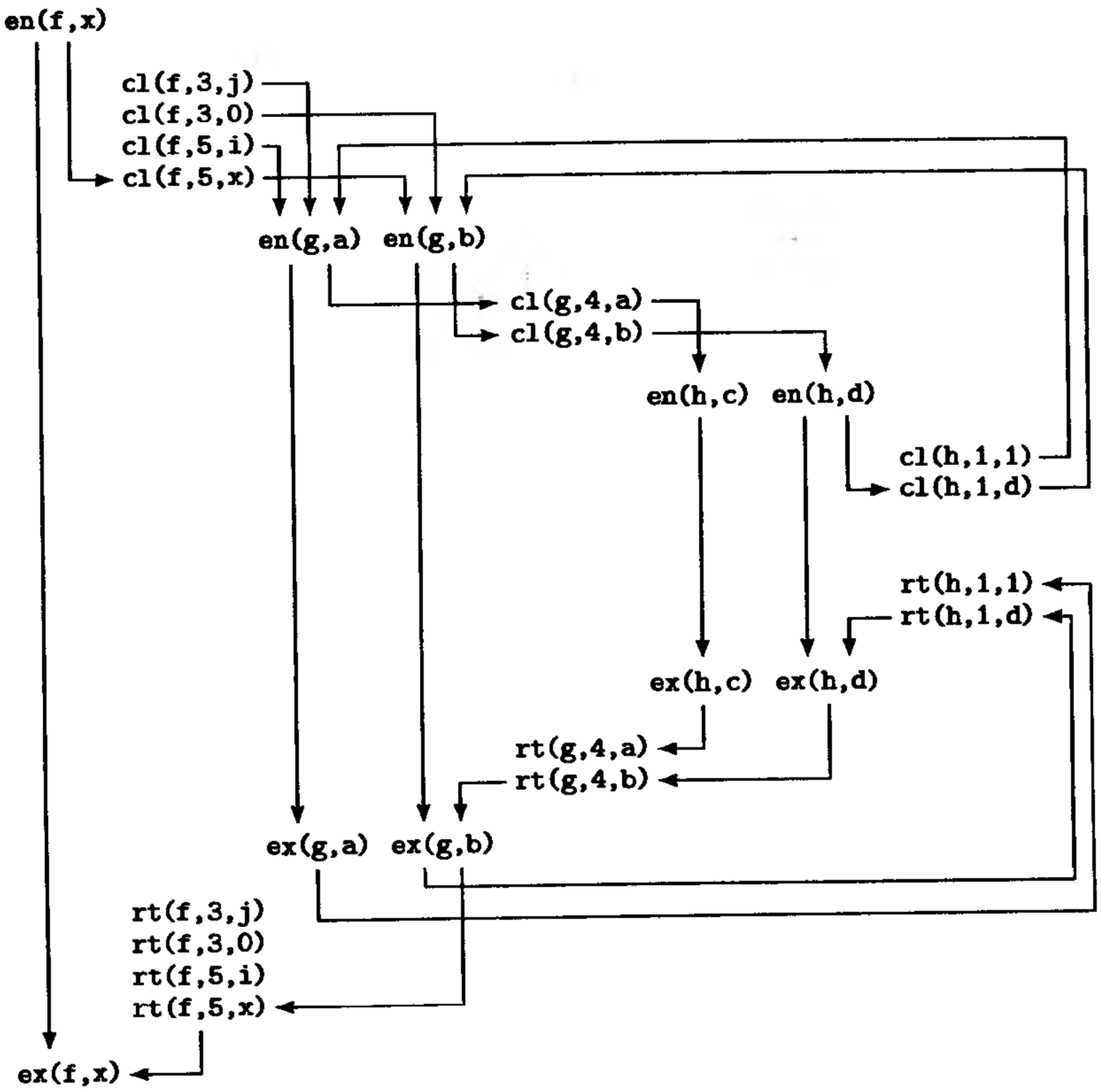


图19-20 图19-19程序的程序概要图

bbs.theithome.com

$$Kill(x) = \begin{cases} false & x\text{为出口结点} \\ \bigwedge\limits_{x \to y \in E} Kill(y) & x\text{为入口或返回结点} \\ Kill(y) \vee Kill(z) & x\text{为调用结点，}y\text{和}z\text{是对应的返回和入口结点} \end{cases}$$

其中，E是程序概要图的边集合。类似的方程也适应于$Use()$属性，并且此框架也可以扩充到处理全局变量。Callahan给出了一种效率较高的计算这些数据流属性的算法，它的运行时间为$O(n \cdot s)$，其中，n是过程的个数，s是程序概要图大小的度量。

635
ι
636

19.2.3 副作用计算中的其他问题

我们一直没有讨论实际程序中出现的一系列问题对副作用计算的影响，如递归、别名、标号变量、指针、联合、过程参数，以及异常处理等。过程间别名在19.4节讨论。Weihl [Weih80]涉及了这些问题中的某一些，而另一些问题则只在已实现的系统中有所提及，这些系统如Parafrase [Kuck74]和ParaScope [CooH93]。

19.3 过程间常数传播

过程间常数传播（interprocedural constant propagation）可以用公式表示为位置无关的或位置特殊的方法。*位置无关*（site-independent）形式，对于程序中的每一个过程，确定其参数中满足后面这个条件的一个子集，其中每一个参数在此过程的每个调用中具有相同的常数值。*位置特殊*(site-specific）形式对每一个特定点调用的每个特定过程，确定每次调用此过程时具有相同常数值的参数组成的子集。

这两种情形下的分析问题都处在由Myers确定的NP完全的或是co-NP完全的标准之内，因此典型的做法是寻找一种代价不大但也不够精确的方法。

Callahan等人[CalC86]介绍了一种执行过程间常数传播的方法，它既可以是流敏感的，也可以是流不敏感的，取决于选择的是他们称为转移和返回转移函数中的哪一种函数。我们的方法是从他们的方法导出的。为简单起见，我们假定构成一个程序的各个过程的形式参数集合是不相交的。

对于过程p中用实参表L调用过程q的调用点i，*转移函数*（jump function）$J(p, i, L, x)$映射此调用在那个调用点的实参信息到q的形参x在q的入口的信息。类似地，对于过程p，*返回转移函数*（return-jump function）$R(p, L, x)$映射p的形参表L有关的信息到通过p的形参x所返回的值有关的信息。例如，对于图19-23所示的过程，

```
J(f,1,[i,j],a) = i
J(f,1,[i,j],b) = j
R(g,[a,b],a) = 2
```

是调用点<f, 1>和过程g()可能的转移函数值和返回转移函数值(如我们下面将看到的，它们是“经传递的参数”转移函数和返回转移函数)。我们进一步定义转移函数$J(p, i, L, x)$的*支持*(support)，记为$Jsupport(p, i, L, x)$，是在定义$J(p, i, L, x)$中所使用的实参的集合；对于返回转移函数也有类似的定义。于是，对于我们的例子，有

```
Jsupport(f,1,[i,j],a) = {i}
Jsupport(f,1,[i,j],b) = {j}
Rsupport(g,[a,b],a) = ∅
```

bbs.theithome.com

版权所有 侵权必究 请尊重作者版权

只使用向前转移函数的过程间常数传播算法给出在图19-21中。它使用图8－3给出的格ICP，我们在图19-22中重新画出了它。如第8章所述，其中，⊤是每一个推定为常数值的变量的初值，常数则代表常数本身，⊥意味着一个其值不是常数的变量。该算法使用的过程中有一个新的过程，即Eval(J(p, i, L, v), Cval)，它计算ICP上的J(p, i, L, v)，即，Jsupport(p, i, L, v)中的变量从Cval()得到它们的值，并且在ICP上执行其运算。程序设计语言中的运算需要适当地映射成ICP上的运算，例如，

637

$$1 + \bot = 1$$

$$1 + 2 = 3$$

$$1 + \top = \top$$

```
procedure Intpr_Const_Prop(P,r,Cval)
   P: in set of Procedure
   r: in Procedure
   Cval: out Var ⟶ ICP
begin
   WL := {r}: set of Procedure
   p, q: Procedure
   v: Var
   i, j: integer
   prev: ICP
   Pars: Procedure ⟶ set of Var
   ArgList: Procedure × integer × Procedure
      ⟶ sequence of (Var ∪ Const)
   Eval: Expr × ICP ⟶ ICP
   || construct sets of parameters and lists of arguments
   || and initialize Cval( ) for each parameter
   for each p ∈ P do
      Pars(p) := ∅
      for i := 1 to nparams(p) do
         Cval(param(p,i)) := ⊤
         Pars(p) ∪= {param(p,i)}
      od
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            ArgList(p,i,q) := []
            for j := 1 to nparams(q) do
               ArgList(p,i,q) ⊕= [arg(p,i,j)]
            od
         od
      od
   od
while WL ≠ ∅ do
   p := ♦WL; WL -= {p}
   for i := 1 to numinsts(p) do
      for each q ∈ callset(p,i) do
         for j := 1 to nparams(q) do
            || if q( )'s jth parameter can be evaluated using values that
            || are arguments of p( ), evaluate it and update its Cval( )
            if Jsupport(p,i,ArgList(p,i,q),param(q,j)) ⊆ Pars(p) then
               prev := Cval(param(q,j))
               Cval(param(q,j)) ⊓= Eval(J(p,i,
                  ArgList(p,i,q),param(q,j)),Cval)
```

图19-21 位置无关过程间常数传播算法

bbs.theithome.com

```
                        if Cval(param(q,j)) ⊏ prev then
                            WL ∪= {q}
                        fi
                    fi
                od
            od
        od
    od
end     || Intpr_Const_Prop
```

图19-21 （续）

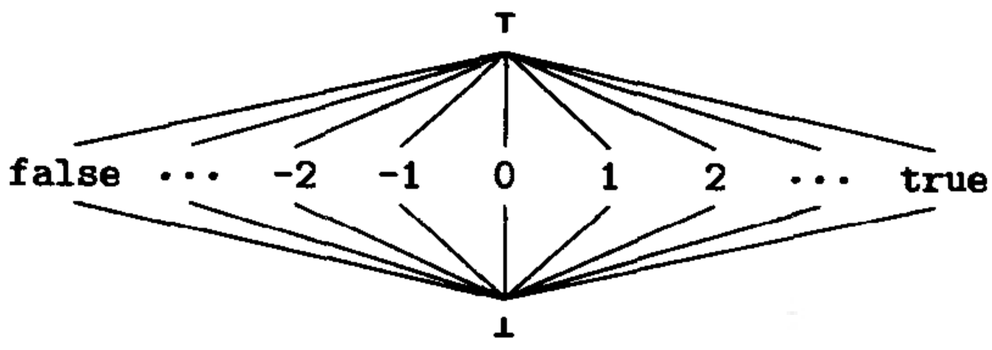


图19-22 整数常数传播格ICP

作为过程间常数传播的一个例子，考虑图19-23中的程序。这个程序可能有的转移函数集合和它们的支持集合如表19-10所示。对于这些转移函数的值，算法首先初始

```
Pars(e) = ∅          ArgList(e,1,f) = [x,1]
Pars(f) = {i,j}      ArgList(f,1,g) = [i,j]
Pars(g) = {a,b}      ArgList(f,2,g) = [j]

Cval(i) = Cval(j) = Cval(a) = Cval(b) = ⊤
```

```
        procedure e( )
        begin
e           x, c: integer
 1          c := f(x,1)
        end
        procedure f(i,j)
        begin
f           s, t: integer
 1          s := g(i,j)
 2          t := g(j,j)
 3          return s + t
        end
        procedure g(a,b)
g       begin
 1          a := 2
 2          b := b + a
 3          return a
        end
```

图19-23 过程间常数传播之例

和WL={e}。然后它从WL中删除e，并处理e()中的一条指令。它确定出f∈callset(e,1)为true，因而接着逐一扫描f()的参数(i和j)。它计算出J(e, 1, [], i)=⊥，并存储这个值于Cval(i)，设置WL={f}。类似的动作导致设置Cval(j)=1。接下来，这个算法设置p=f和i=1，并处理在那一点对g()的调用。这导致设置Cval(a)=⊥和Cval(b)=1。对于i=2，我们再次得到Cval(a)=⊥和Cval(b)=1。最终结果是

638
~
639

bbs.theithome.com

表19-10 图19-23中例程序的可能的转移函数和它们的支持集合

J(e,1,[],i) = ⊥	Jsupport(e,1,[],i) = ∅
J(e,1,[],j) = 1	Jsupport(e,1,[],j) = ∅
J(f,1,[i,j],a) = i	Jsupport(f,1,[i,j],a) = {i}
J(f,1,[i,j],b) = j	Jsupport(f,1,[i,j],b) = {j}
J(f,2,[i,j],a) = j	Jsupport(f,2,[i,j],a) = {j}
J(f,2,[i,j],b) = j	Jsupport(f,2,[i,j],b) = {j}

```
Cval(i) = ⊥
Cval(j) = 1
Cval(a) = ⊥
Cval(b) = 1
```

于是，在f()的每一个调用点，形式参数j是常数，在g()的每一个调用点，b是常数。

转移函数和返回转移函数可能的选择范围从具体的常数到符号解释的结果。有如下一些选择:

1. 文字常数（literal constant)。这种选择使得每一个J()值当调用端传递一个文字常数时是常数，否则是⊥；并且它不通过过程体传播过程间的常数。

2. 过程间常数（interprocedural constant)。这种选择使得每一个J()值当过程内常数传播能够确定它是常数时则为常数，否则为⊥。 640

3. 经传递的参数（pass-through parameter)。这种选择(这是我们在前面例子中所使用的)在过程间常数有效的地方使用过程间常数，作为通过一个被调用例程而没有改变地被传递的参数的值，否则使用⊥。

4. 多项式参数（polynomial parameter)。这种选择在常数值可用的地方使用过程内的常数；当过程内要传递给一个例程的实参值是此过程的形参组成的一个多项式时，使用该过程的参数组成的多项式函数；否则使用⊥。

5. 符号执行（symbolic execution)。这种方法模拟每一个过程的执行来确定常数参数值。

这些选择的代价按编号的顺序由低至高，并且对应地，其信息量也由少到多。Grove和Torczon [GroT93]讨论了转移和返回转移函数、它们的计算代价，以及它们提供的信息。

转移函数的使用，使得对于具有n个结点和e条边的调用图，以及对于所有转移函数的选择，除最复杂的情形以外，调用位置无关的和调用位置特殊的常数计算时间都能在$O(n+e)$内。对于调用位置特殊的形式，我们通过转换过程到SSA形式(为了计算效率起见)来修改此算法，并以流敏感方式计算它们。

返回转移函数只在调用位置特殊的(或流敏感)分析形式中有用。为了理解这一点，考虑图19-23所示的例程序。一般自然想到的是在（使用向前转移函数之前）对程序的一个预扫描遍中使用返回转移函数，这一遍预扫描从调用图的叶结点（包括叶结点强连通分量中任意选择的成员）开始，于是自然会选择过程g()和参数a的返回转移函数是R(g,[a,b],a)=2。如果这样选择，对于过程f()的调用点1，我们就会设置Cval(a)=2。但是，这种选择是不适合的：如果我们现在用向前转移函数执行常数传播，则会使得Cval(a)的一个值作用于第一个g()调用之后的那个调用，而不是第一个调用。因此，如果我们使用返回转移函数，流敏感是必须的。使用返回转移函数并不影响此算法的复杂性。

调用位置特殊的过程间常数传播能用来控制19.5节所讨论的过程克隆。

19.4 过程间别名分析

如第10章讨论的，不同的语言其过程内别名的可能性差别相当大，其差别尤以Fortran和C

bbs.theithome.com

为代表。在过程间别名分析中，人们加入了另外一种尺度，即两级作用域，如Fortran和C中的情形，以及多级作用域，如Pascal、PL/I和Mesa中的情形。

Fortran标准特别地禁止了对有别名变量的赋值。但另一方面，C允许取大多数对象的地址，允许将它传递到几乎任何地方，并且几乎可用它做任何事情。

641 过程间别名一般是由参数传递和访问非局部变量而创建的。如第10章开始所讨论的，别名作为一种关系具有的特征取决于我们是否考虑时机的影响（或等价地，流敏感）。除了在19.4.2节关于传值和指针的语言的讨论之外，我们这里关心的是流不敏感信息。作为流不敏感过程间别名非传递性的例子，考虑图19-24中的代码，并假定参数是传地址的。虽然i *alias* a和j *alias* a成立，但i *alias* j不成立。

```
global i, j
procedure f( )
begin
   g(i)
   g(j)
end
procedure g(a)
begin
   i := i + 1
   j := j + 1
   return a
end
```

图19-24 过程间别名分析之例

流敏感别名分析的开销至少与流敏感副作用分析（见19.24节）的开销同样昂贵，所以我们满足于对于传地址的调用采用流不敏感的过程间别名分析方法，对于用指针的传值调用则采用一种可塑的过程间别名分析方法，这里可塑的意思是它可以是流敏感的也可以是流不敏感的，如同在10.2节和10.3节讨论过程内别名分析时一样。

19.4.1 流不敏感别名分析

我们用函数*ALIAS*: Var × Procedure → set of Var来描述程序中每一个过程内的所有别名组成的集合。具体地，*ALIAS*(*x*, *p*)=*s*，当且仅当*s*由过程*p*中可能是*x*的别名的所有变量组成。

计算*ALIAS*的基本思想是，按深度为主顺序，沿着由每一个调用点与实参结合的非局部的变量参数组成的所有可能的链，以增量方式来累加别名偶对。

Cooper和Kennedy [CooK89] 观察到，只要源语言不允许指针别名，非局部变量就只可能有形式参数作为别名。根据这一观察，他们开发了一种有效率的方法，这种方法将形式参数和非局部变量分开进行处理，由此显著地减少了可能需要计算的偶对个数。我们这里描述的方法以他们的方法为基础。

对于像Pascal或Mesa这类过程可以嵌套并且变量的可见性由嵌套决定的语言，在别名判别中有两个复杂因素。一个是嵌套层为*l*的例程的形参可能是其内嵌套层为*m* > *l*的例程的非局部
642 变量。另一个因素实质上是第一种复杂因素的反面，即在嵌套层*l*定义的变量在小于*l*的嵌套层中是不可见的。为了进一步描述这个问题，我们使用过程的扩展形式参数集合（extended formal-parameter set）概念，这个概念在后面再详述。

概括而言，对于一个程序*P*，别名判别方法由如下4个步骤组成:

1. 我们首先构造程序的结合图B（参见19.2.1节和图19-26)的一个扩展版本，此图考虑的是扩展形式参数，并构造它的成对结合图Π(参见下面的描述)。

2. 接着解B上的向前数据流问题，对于每一个形式参数*fp*，计算通过一个使*v*与*fp*结合的调用链而可能与之别名的变量*v*（不包括形式参数）组成的集合*A*(*fp*)。

3. 然后执行在Π上的标志算法（也可以将它看成是解向前数据流分析问题)，以确定可能互为别名的形式参数。

4. 最后合并前面几个步骤得到的信息。

一旦得到别名信息，就可以将它与不考虑别名情况下得到的过程间数据流问题的解合并到一起而产生完整的解。

bbs.theithome.com

过程*p*的扩展形式参数（extended formal-parameter）集合，记为*ExtFormals*(*p*)，是*p*内可见的所有形式参数组成的集合，包括包含*p*的嵌套过程的（但没有被界于其间的说明遮盖为不可见的）形式参数。例如，在图19-25中，过程n()的扩展形式参数集合是{m,z,w}。

```
            global a, b
            procedure e( )
e           begin
 1             a := f(1,b)
            end     || e
            procedure f(x,y)
f           begin
 1              y := h(1,2,x,x)
 2              return g(a,y) + b + y
            end     || f
            procedure g(z,w)
               local c
               procedure n(m)
n              begin
 1                 a := p(z,m)
 2                 return m + a + c
               end     || n
g           begin
 1              w := n(c) + h(1,z,w,4)
 2              return h(z,z,a,w) + n(w)
            end     || g
            procedure h(i,j,k,l)
               local b
               procedure t(u,v)
t              begin
 1                 return u + v * p(u,b)
               end     || t
h           begin
 1              b := j * l + t(j,k)
 2              return i + t(b,k)
            end     || h
            procedure p(r,s)
p           begin
 1              b := r * s
            end     || p
```

图19-25 用于过程间别名计算的例程序

经修改后建立结合图的代码给出在图19-26中（注意，只有例程Bind_Pairs()与图19-13中的版本不同）。如果过程*p*中的指令*i*调用过程*q*并且将*p*的扩展形式参数*x*与*q*的扩展形式参数*y*结合，passed(*p*, *i*, *q*, *x*, *y*)的值为true，否则返回false。

我们使用图19-25中的程序作为说明这种处理的例子。在下面的叙述中，我们分别用带下标的b_1和b_2来区分全局变量b和h()的局部变量b。注意，为了允许更通用的变量和参数命名方法，我们需要标识出每一个变量所在过程的嵌套序列，例如，我们可以用<b,[]>表示全局变量b，用<b,[h]>表示h()中的局部变量b，用<y,[h,t]>表示t()的参数y。这个例子的调用和结合图给出在图19-27中。

bbs.theithome.com

```
procedure Build_Binding_Graph(P,N,E,pinsts)
   P: in set of Procedure
   N: out set of Var
   E: out set of (Var × Var)
   pinsts: in Procedure ⟶ integer
begin
   p, q: Procedure
   i, m, n: integer
   oldE: set of (Var × Var)
   N := E := oldE := ∅
   repeat
      oldE := E
      for each p ∈ P do
         for i := 1 to numinsts(p) do
            for each q ∈ callset(p,i) do
               Bind_Pairs(p,i,q,N,E)
            od
         od
      od
   until E = oldE
   repeat
      oldE := E
      for each e ∈ E do
         for each f ∈ E (f ≠ e) do
            if e@2 = f@1 & e@1 ≠ f@2 then
               E ∪= {e@1→f@2}
            fi
         od
      od
   until E = oldE
end    || Build_Binding_Graph

procedure Bind_Pairs(p,i,q,N,E)
   p, q: in Procedure
   i: in integer
   N: inout set of Var
   E: inout set of (Var × Var)
begin
   e, f: Var × Var
   for each u ∈ ExtFormals(q) do
      for each v ∈ ExtFormals(p) do
         if passed(p,i,q,v,u) then
            N ∪= {u,v}
            E ∪= {v→u}
         fi
      od
   od
end    || Bind_Pairs
```

图19-26 构建程序*P*的结合图B的算法

非局部变量只可能在这种情况下与过程的一个形式参数别名：这个非局部变量在该过程中是可见的，并且它被作为实参传递给那个形式参数。因此我们定义*A*(*fp*)（其中*fp*是一个形式参数）是由于一个或多个调用链使得*v*与*fp*结合而导致*fp*可能与之别名的非局部变量*v*（包括包含说明*fp*的过程的那些嵌套过程的形式参数）的集合。为了计算*A*()，我们采用两遍向前扫描结合图B，首先收集作为参数传递的非局部变量的结合，之后沿结合链而行。计算*A*()的算法

bbs.theithome.com

越是公句制作请尊重作者版权

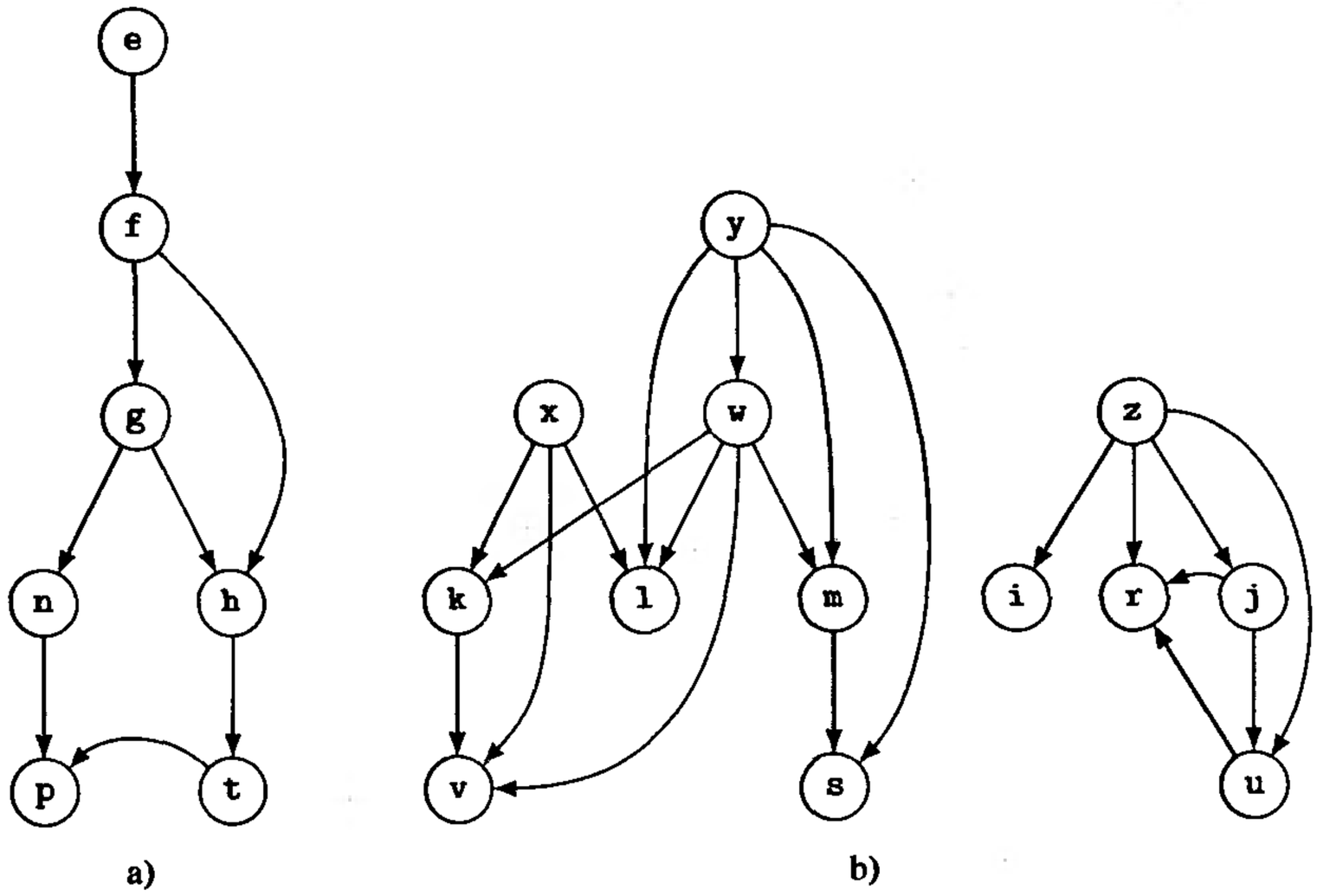


图19-27 图19-25中程序的a) 调用图，b) 结合图

Nonlocal_Aliases()给出在图19-28中；参数N、P和E分别是结合图的结点集合、根集合和边集合。算法中用到如下一些过程:

1. Shallow(*v*)是其说明所在嵌套层小于或等于*v*的说明所在嵌套层的变量集合。
2. Formals(*p*)是过程*p*的形式参数集合。
3. Nonlocals(*p*)是非局部于*p*，并且不是嵌套*p*的任何过程的形式参数的变量集合。

643
~
645

4. ExtFormals(*p*)是*p*内可见的所有形式参数集合，包括嵌套*p*的所有过程的、且未被这些过程之间的定义遮盖的形式参数。
5. Top_Sort(*N*, *E*)返回一个从*E*中抽取的边组成的序列，这个序列表示一个具有*N*个结点和*E*条边并且删除了后向边的图的拓扑序。(注意可能存在一个以上的根。)

```
procedure Nonlocal_Aliases(P,N,E,A)
   P: in set of Procedure
   N: in set of Var
   E: in set of (Var × Var)
   A: out Var ⟶ set of Var
begin
   v: Var
   e: Var × Var
   p, q: Procedure
   i, j: integer
   oldA: Var ⟶ set of Var
   T: sequence of (Var × Var)
   A := ∅
   for each p ∈ P do
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            for each v ∈ (Nonlocals(q)
               ∪ (ExtFormals(q) - Formals(q))) do
               for j := 1 to nparams(q) (v = arg(p,i,j)) do
                  || accumulate nonlocal variables and nonlocal formal
```

图19-28 利用结合图高效计算非局部变量别名的算法

bbs.theithome.com

```
                    || parameters that may be aliases of q( )'s jth parameter
                    A(param(q,j)) ∪= {v}
                od
            od
        od
    od
od
T := Top_Sort(N,E)
repeat
    oldA := A
    for i := 1 to |T| do
        || accumulate nonlocals along edges in binding graph
        A(T↓i@2) ∪= A(T↓i@1) ∩ Shallower(T↓i@1)
    od
until oldA = A
end    || Nonlocal_Aliases
```

图19-28 （续）

如果我们定义Envt(*p*)是嵌套过程*p*的过程序列，*p*作为此序列的最后一个元素，并定义Vars(*p*)是过程*p*的形式参数和局部变量集合，则ExtFormals(*p*)可递归地定义为

$$\text{ExtFormals}(\text{Envt}(p)\downarrow 1) = \text{Formals}(\text{Envt}(p)\downarrow 1)$$

$$\text{ExtFormals}(\text{Envt}(p)\downarrow i) = \text{Formals}(\text{Envt}(p)\downarrow i) \cup (\text{ExtFormals}(\text{Envt}(p)\downarrow (i-1)) - \text{Vars}(\text{Envt}(p)\downarrow i)) \quad \text{for } 2 \leq i \leq |\text{Envt}(p)|$$

对于我们的例程序，Nonlocal_Aliases()首先初始函数A()为∅，然后依次查看各个调用并收集与形式参数别名的非局部变量，最后在结合图中向前传播别名信息，由此得到表19-11a所示之值。

现在，可以通过*A*()的逆函数来计算与每一个非局部变量别名的形式参数集合。这由图19-29所示过程Invert_Nonlocal_Aliases()来完成，它简单地设置*ALIAS*()为空函数，然后收集来自*A*()集合的值。对于我们的例子，它得到表19-11b所示的值。

表19-11 对于图19-25的例程序，a)用Nonlocal_Aliases()计算出的*A*()集合，以及b) 用Invert_Nonlocal_Aliases()计算出的*ALIAS*()集合

A(i) = {a,z}	A(j) = {a,z}	A(k) = {a,b_1,w}
A(l) = {b_1,w}	A(m) = {b_1,c,w}	A(r) = {a,b_2,j,z}
A(s) = {b_1,c,b_2}	A(u) = {a,b_2,j}	A(v) = {a,k}
A(w) = {b_1}	A(x) = ∅	A(y) = {b_1}
A(z) = {a}		

a)

ALIAS(a,e) = ∅	ALIAS(b_1,e) = ∅
ALIAS(a,f) = ∅	ALIAS(b_1,f) = {y}
ALIAS(a,g) = {z}	ALIAS(b_1,g) = {w}
ALIAS(a,h) = {i,j,k}	ALIAS(b_1,h) = {k,l}
ALIAS(a,n) = {z}	ALIAS(b_1,n) = {m,w}
ALIAS(a,p) = {r}	ALIAS(b_1,p) = ∅
ALIAS(a,t) = {i,j,k,u,v}	ALIAS(b_1,t) = {s,k,l}
ALIAS(c,e) = ∅	ALIAS(b_2,e) = ∅
ALIAS(c,f) = ∅	ALIAS(b_2,f) = ∅
ALIAS(c,g) = ∅	ALIAS(b_2,g) = ∅
ALIAS(c,h) = ∅	ALIAS(b_2,h) = ∅
ALIAS(c,n) = {m}	ALIAS(b_2,n) = ∅
ALIAS(c,p) = ∅	ALIAS(b_2,p) = ∅
ALIAS(c,t) = ∅	ALIAS(b_2,t) = {u}

b)

bbs.theithome.com

```
Nonlocale, ExtFormals: Procedure ⟶ set of Var

procedure Invert_Nonlocal_Aliases(P,A,ALIAS)
   P: in set of Procedure
   A: in Var ⟶ set of Var
   ALIAS: out (Var × Procedure) ⟶ set of Var
begin
   p: Procedure
   x, v: Var
   ALIAS := ∅
   for each p ∈ P do
      for each v ∈ ExtFormals(p) do
         for each x ∈ A(v) ∩ Nonlocals(p) do
            ALIAS(x,p) ∪= {v}
         od
      od
   od
end    || Invert_Nonlocal_Aliases
```

图19-29 逆转非局部别名函数，并由此计算与每一个非局部变量别名的形式参数集合的算法

形式参数成为别名可有几种途径。例如，如果两个参数与同一个实参相结合，则这两个参数互为别名。另外，如果传递非局部变量给一个例程的形式参数，并且传递与此非局部变量别名的一个变量给另一个形式参数，则这两个形式参数将互为别名。此外，也可以通过调用链传递形式参数别名而创建更多的有别名的形式参数。

646 ~ 648

为了建立这种情形的模型，我们使用结合图的一种成对对等物，称为成对结合图$\Pi=\langle N_\Pi, E_\Pi\rangle$，在这种图中，每一个结点是同一过程的一对扩展形式参数，每一条边模仿由一个调用导致的一个偶对与另一个偶对的结合。图19-30给出了建立程序的成对结合图的代码。

```
procedure Build_Pair_Graph(P,N,E)
   P: in set of Procedure
   N: out set of (Var × Var)
   E: out set of ((Var × Var) × (Var × Var))
begin
   p, q, r: Procedure
   u, v, w, x, y: Var
   k, s: integer
   N := ∅
   E := ∅
   for each p ∈ P do
      for each u,v ∈ ExtFormals(p) do
         N ∪= {⟨u,v⟩}
      od
   od
   for each p ∈ P do
      for k := 1 to numinsts(p) do
         for each q ∈ callset(p,k) do
            for each u,v ∈ ExtFormals(q) do
               for each w,x ∈ ExtFormals(p) do
                  if match(p,q,k,w,u,x,v) then
                     E ∪= {⟨w,x⟩→⟨u,v⟩}
                  elif ∃r ∈ P ∃s ∈ integer ∃w ∈ Var
                     (pair_match(p,u,v,q,s,r,w,x,y)) then
```

图19-30 建立程序的成对结合图∏的算法

bbs.theithome.com

```
                                E ∪= {⟨u,v⟩→⟨x,y⟩}
                            fi
                        od
                    od
                od
            od
        od
    end     || Build_Pair_Graph
```

图19-30 （续）

函数match(*p*, *q*, *k*, *w*, *u*, *x*, *v*)返回true，如果过程*p*的指令*k*调用过程*q*，并且分别传递*p*的扩展形式参数*w*和*x*给*q*的扩展形式参数*u*和*v*；否则返回false。

函数pair_match(*p*, *u*, *v*, *q*, *s*, *r*, *w*, *x*, *y*)返回true，如果满足下列条件：存在*p*的扩展形式参数*u*和*v*，以及嵌套在*p*内的过程*q*中的一个调用点*s*，使得对于某个过程*r*，<*q*, *s*>调用*r*，并分别使*u*和*w*与*r*的扩展形式参数*x*和*y*结合，并且$v \in A(w)$；否则，返回false。如果存在满足pair_match(*p*, *u*, *v*, *q*, *s*, *r*, *w*, *x*, *y*)的过程、整数、变量和调用点，我们加一条从<*u*, *v*>到<*x*, *y*>的边，如图19-31所示。

649

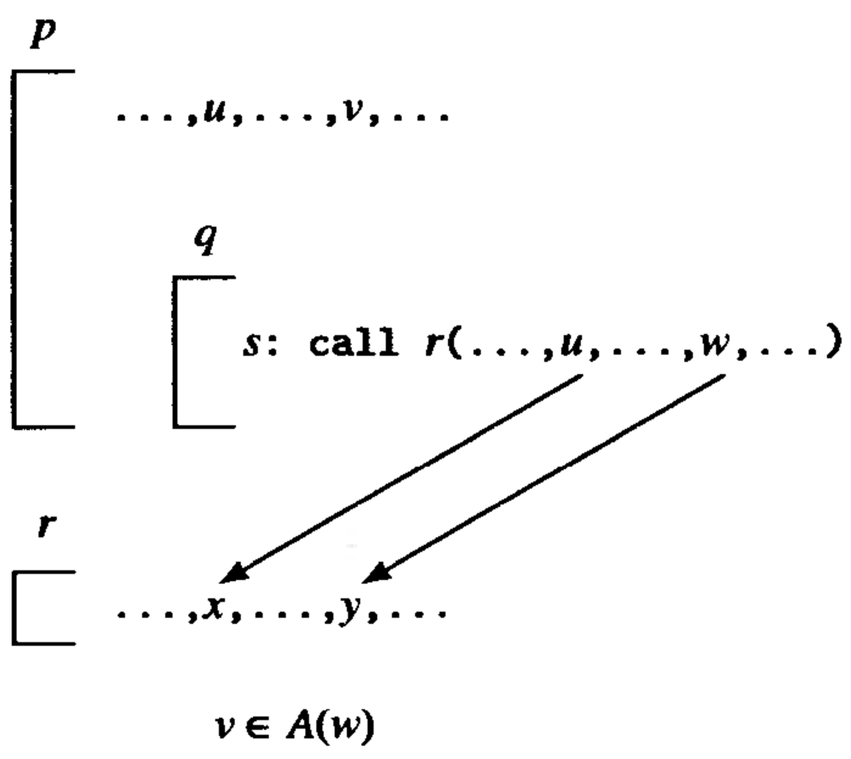


图19-31 加入满足pair_match()的一条弧到成对结合图的图解。已知过程*p*有扩展形式参数*u*和*v*，嵌套在*p*内的过程*q*在调用点<*q*, *s*>调用过程*r*，并分别传递*u*给*r*的形式参数*x*，传递*w*给*r*的形式参数*y*，并且$v \in A(w)$，我们将边<*u*, *v*>→<*x*, *y*>加入到成对结合图Π中

我们这个例子的成对结合图给出在图19-32中。

为了发现可能互为别名的成对形参，我们按如下方式来标志程序的成对结合图：如果传递一个变量给一个例程的两个不同的形式参数，则这两个形式参数互为别名，因此给成对结合图中的对应结点打上标记。另外，如果(1)传递一个非局部变量给过程*q*的一个形式参数，(2)传递*q*的调用者的形式参数给*q*的另一个形式参数，并且(3)该调用者的形式参数是这个非局部变量的别名，则*q*的这两个形式参数可能别名，因此那个偶对也打上标记。这个算法给出在图19-33中。

对于我们的例子，这导致<k, l>、<l, k>、<i, j>、<j, i>、<r, s>、<j, k>、<j, l>、<i, l>、<u, v>、<s, r>、<k, j>、<l, j>、<l, i>和<v, u>都被打上标记。图19-32中用带阴影的圆圈表示带标记的结点。注意，<r, s>和<s, r>是图19-33中带星号的那条if语句惟一标记的两个结点。

bbs.theithome.com

越是公司制作 请尊重作者版权

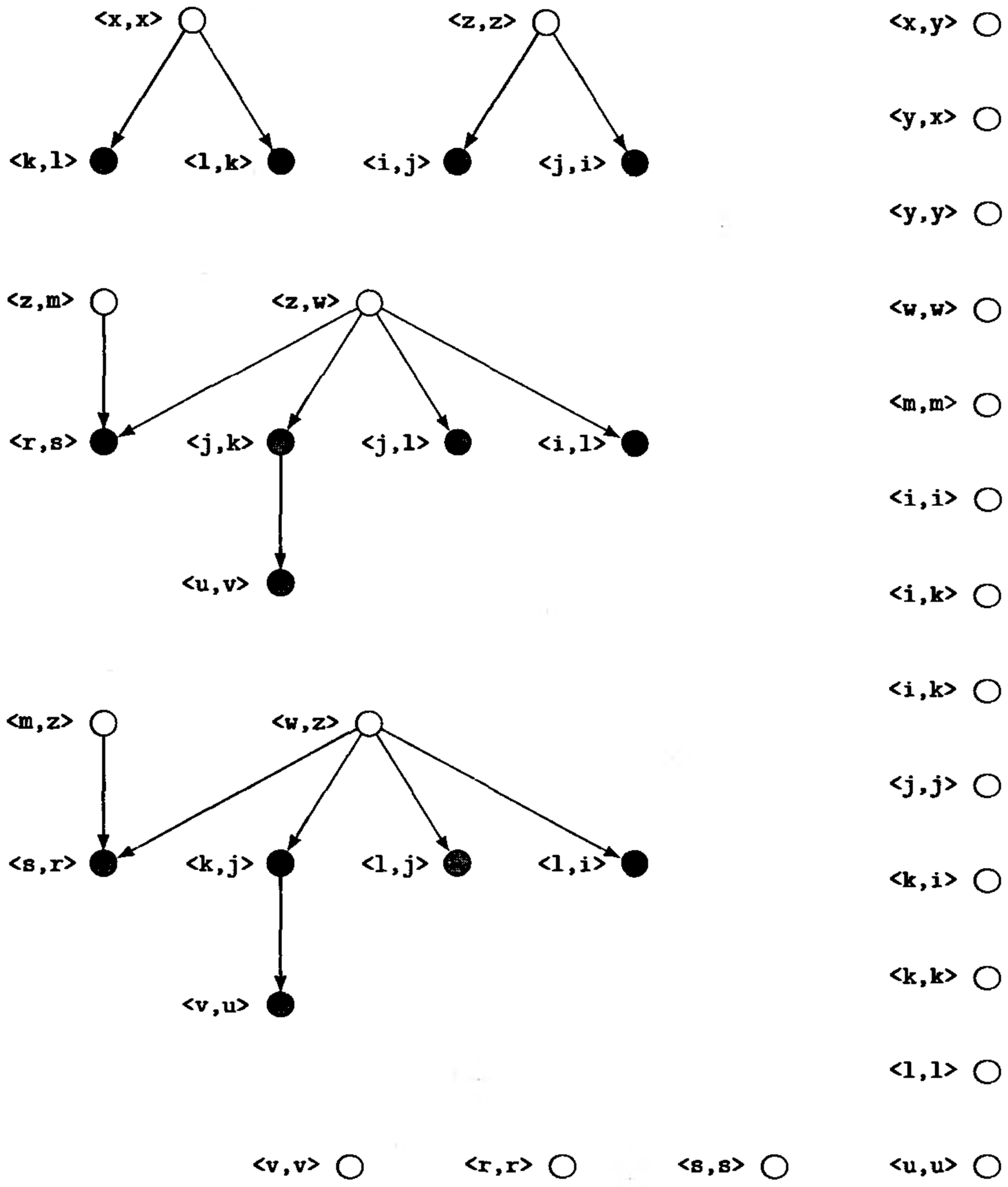


图19-32 我们这个例子的成对结合图。结点和边由Build_Pair_Graph()生成。阴影结点由Mark_Alias_Pairs()标记。Prop_Marks()没有给另外的结点打上标记

```
procedure Mark_Alias_Pairs(P,Mark,ALIAS)
   P: in set of Procedure
   Mark: out (Var × Var) ⟶ boolean
   ALIAS: in (Var × Procedure) ⟶ set of Var
begin
   p, q: Procedure
   i, j, k, l: integer
   u, v, w, x, y: Var
   Mark := ∅
   for each p ∈ P do
      for i := 1 to numinsts(p) do
         for each q ∈ callset(p,i) do
            for each u,v ∈ ExtFormals(q) (u ≠ v) do
               if ∃w,x ∈ ExtFormals(p)
                  (match(p,q,i,w,u,x,v)) then
                  Mark(u,v) := Mark(v,u) := true
```

图19-33 标记程序结合图中参数偶对的算法

bbs.theithome.com

```
                fi
             od
*            if ∃y ∈ ExtFormals(p) ∃k ∈ integer
                (y = arg(p,i,k)) then
                for each w ∈ Nonlocals(p) do
                   if ∃l ∈ integer (w = arg(p,i,l)
                      & y ∈ ALIAS(w,p)) then
                      Mark(param(q,k),param(q,l)) := true
                      Mark(param(q,l),param(q,k)) := true
                   fi
                od
             fi
          od
       od
    od
end     || Mark_Alias_Pairs
```

图19-33 （续）

接着，用图19-34中的过程Prop_Marks()沿成对结合图向前传播这些标记，以保证一个例程的形式参数的别名信息被适当地传递给它调用的例程的形式参数。这是通过维护一张成对结合图中带标记的结点表来实现的。随着每一个结点从此表中被删除，沿着从那个结点出发的边向前，标记所到达的结点。如果前面没有标记这些结点，则将它们加入到带标记的结点表中。对于我们的例子，这个过程没有标记出额外的偶对。

```
procedure Prop_Marks(N,E,Mark)
   N: in set of (Var × Var)
   E: in set of ((Var × Var) × (Var × Var))
   Mark: inout (Var × Var) → boolean
begin
   n: Var × Var
   f: (Var × Var) × (Var × Var)
   WL := ∅: set of (Var × Var)
   for each n ∈ N do
      if Mark(n@1,n@2) then
         WL ∪= {n}
      fi
   od
   while WL ≠ ∅ do
      n := ♦WL; WL -= {n}
      for each f ∈ E (f@1 = n) do
         if !Mark(f@2@1,f@2@2) then
            Mark(f@2@1,f@2@2) := true
            WL ∪= {f@2}
         fi
      od
   od
end     || Prop_Marks
```

图19-34 在程序的偶对结合图中传播参数偶对标记的算法

接下来，我们通过合并A()和ALIAS()集合，并使用Mark()信息来计算形式参数的别名。完成这个任务的例程Formal_Aliases()是直观的，图19-35给出了其代码。它用到了过程Outside_In(*P*)，这个过程的返回值是组成*P*的那些过程的一个序列，此序列按嵌套顺序的拓扑序排列，最外层的过程排在前面。对于我们的例子，这导致得到表19-12所示的别名集合。

bbs.theithome.com

```
procedure Formal_Aliases(P,Mark,A,ALIAS)
   P: in set of Procedure
   Mark: in (Var × Var) ⟶ boolean
   A: in Var ⟶ set of Var
   ALIAS: inout (Var × Procedure) ⟶ set of Var
begin
   OI: sequence of Procedure
   p: Procedure
   i: integer
   v: Var
   OI := Outside_In(P)
   for i := 1 to |OI| do
      p := OI↓i
      for j := 1 to nparams(p) do
         ALIAS(param(p,j),p) := A(param(p,j))
         for each v ∈ ExtFormals(p) do
            if param(p,j) ≠ v & Mark(param(p,j),v) then
               ALIAS(param(p,j),p) ∪= {v}
               ALIAS(v,p) ∪= {param(p,j)}
            fi
         od
      od
   od
end    || Formal_Aliases
```

图19-35 由Mark()、A()和ALIAS()计算形式参数别名信息的算法

表19-12 由Formal_Aliases()计算出的图19-25例子的形式参数别名集合

ALIAS(x,f) = ∅	ALIAS(y,f) = {b_1}
ALIAS(z,g) = {a}	ALIAS(w,g) = {b_1}
ALIAS(i,h) = {a,j,k,l}	ALIAS(j,h) = {a,i,k,l}
ALIAS(k,h) = {a,b_1,i,j,l}	ALIAS(l,h) = {b_1,i,j,k}
ALIAS(m,n) = {b_1,c}	ALIAS(u,t) = {a,b_2,v}
ALIAS(v,t) = {a,u}	ALIAS(r,p) = {a,b_2,s}
ALIAS(s,p) = {b_1,c,r}	

这种别名计算方法的时间是$O(n_2+n \cdot e)$，其中n和e分别是程序调用图的结点个数和边的条数。

为了将别名信息与副作用计算的忽略别名的版本合并，我们使用图19-36所示的算法，其中用*MOD*作为例子。大体的做法是，我们初始化*MOD*为*DMOD*，然后将形式参数和非局部变量的别名加入到其中。

650 ι 653

19.4.2 传值和传指针语言的过程间别名分析

为了对类似于C这种传值和传指针的语言进行别名分析，我们扩充10.2节和10.3节讨论的过程内的方法。具体地，我们用一个过程的调用点的信息来初始该过程入口的*Ovr*()和*Ptr*()函数，并用来自被调用过程返回点的信息设置过程返回到调用点的*Ovr*()和*Ptr*()函数。我们可以对每一个调用点单独进行分析，也可以同时对调用一个特定例程的所有调用进行分析。

首先，假定我们每次只分析一个调用点。令P和P'分别是直接位于此调用点之前和紧跟在其后的程序点，令entry+和return-分别是紧跟在例程入口之后和直接位于return之前的程序点，如图19-37所示。于是，对于一个非局部变量x，这两个函数的初值为:

$$Ovr(\texttt{entry+}, x) = Ovr(\texttt{P}, x)$$

$$Ptr(\texttt{entry+}, x) = Ptr(\texttt{P}, x)$$

bbs.theithome.com

```
procedure MOD_with_Aliases(P,ALIAS,DMOD,MOD)
    P: in set of Procedure
    ALIAS: in (Var × Procedure) ⟶ set of Var
    DMOD: in (Procedure × integer) ⟶ set of Var
    MOD: out (Procedure × integer) ⟶ set of Var
begin
    p, q: Procedure
    i: integer
    v: Var
    for each p ∈ P do
        for i := 1 to numinsts(p) do
            || initialize MOD with direct MOD
            MOD(p,i) := DMOD(p,i)
            || add formal-parameter aliases
            for each v ∈ Formals(p) do
                if v ∈ DMOD(p,i) then
                    MOD(p,i) ∪= ALIAS(v,p)
                fi
            od
            || add nonlocal-variable aliases
            for each v ∈ Nonlocals(p) ∩ DMOD(p,i) do
                MOD(p,i) ∪= ALIAS(v,p)
            od
        od
    od
end    || MOD_with_Aliases
```

图19-36 将别名信息与忽略别名的副作用信息合并的算法

对于传递给形参x的实参y，其初值如下：

$$Ovr(\texttt{entry+}, x) = Ovr(\texttt{P'}, y)$$

$$Ptr(\texttt{entry+}, x) = Ptr(\texttt{P'}, y)$$

对于返回，我们对非局部变量x做反向的初始化，即

$$Ovr(\texttt{P'}, x) = Ovr(\texttt{return-}, x)$$

$$Ptr(\texttt{P'}, x) = Ptr(\texttt{return-}, x)$$

例如，考虑图19-38中的C代码。*Ptr*()函数的计算结果是：

$$Ptr(\texttt{P}, \texttt{p}) = star(\texttt{a})$$

$$Ptr(\texttt{P}, \texttt{q}) = star(\texttt{a})$$

$$Ptr(\texttt{P}, \texttt{r}) = star(\texttt{a})$$

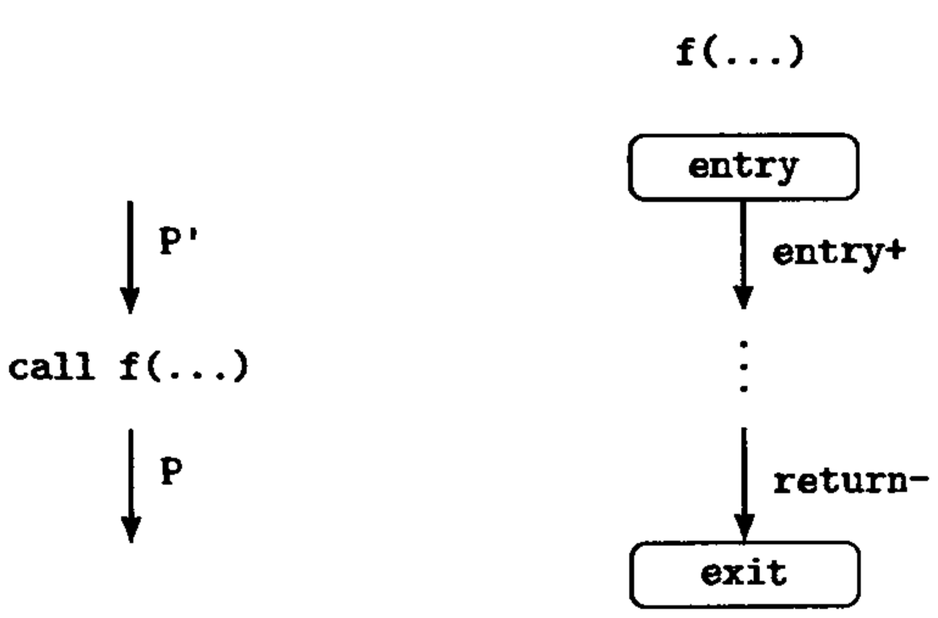


图19-37 一个C例程的调用点、入口和返回

```
int *p, *q;
int *f(int *q)
{   p = q;
    return p;
}

int main( )
{   int a, *r;
    r = f(&a);
}
```

图19-38 过程间别名分析的C程序示例

版权所有 侵权必究

bbs.theithome.com

版权所有 翻印必究

其中P表示main()中紧跟在从f()返回之后的程序点。

654
l
655

为了同时对调用一个特定例程的所有的调用点$P_1', \cdots, P_k'$进行分析，对于非局部变量x和前面关于参数传递以及关于返回点的非局部变量的方程的相应改写版本，我们设置

$$Ovr(\texttt{entry+}, x) = \bigcup_{i=1}^{k} Ovr(P_i^{'}, x)$$

$$Ptr(\texttt{entry+}, x) = \bigcup_{i=1}^{k} Ptr(P_i^{'}, x)$$

19.5 过程间优化

有了前面几节讨论的关于过程间控制流、数据流和别名分析的手段之后，便有可能做如下一些优化了：

1. 我们能够利用过程间分析收集的信息来指导过程集成。

2. 我们能够利用位置无关常数传播分析提供的信息来优化那种总是用相同的一个或多个常数参数调用的过程体。

3. 我们能够利用位置特殊的常数传播分析提供的信息来克隆一个过程体的副本，并针对特定的调用点优化它们。

4. 我们能够使用副作用信息剪裁特殊调用点的调用约定，达到对调用过程和被调用过程之间的寄存器保护进行优化的目的。

5. 对于那种体积较大且不能将它们改成传地址方式传递的实参，我们能够对它们的传值参数的传递代码进行优化。

6. 最后，并且也是最频繁使用和最重要的，我们能够利用过程间数据流信息改善过程内关于过程入口和出口，以及关于调用和返回的数据流信息。

过程间分析能够提供指导过程集成需要的许多信息，如调用点的个数和关于常数值参数的信息。有关细节参见15.2节。

根据常数参数来进行优化的方法是显而易见的。我们在过程的中间代码形式中用常数值替换那些常数值参数，并在由此得到的结果代码上执行第12~18章讨论的全局优化。我们也裁剪这种过程的调用和入口代码，使之不传递或接收常数参数。这种替换是在中间代码上，而不是在源程序代码上进行，因为对源程序的进一步修改可能会使得优化所依据的信息失效。

656

对在一个或多个调用点上值为常数的过程参数进行优化（称为过程特殊化或克隆）也较容易。对于调用相同过程，并且传递相同常数值给一个或多个参数的调用点集合，我们克隆这个过程体的中间代码的一个副本，然后对克隆体和它的调用点执行前面段落中介绍的相同优化。

这两种情况常常使得删除过程体内不必要的边界检查这种优化成为可能。例如，许多例程都被编写成能对任意大小的数组进行操作，但在具体的程序中只用来处理其大小由主程序确定的数组。传播数组大小的信息给通用的例程，能够使它们被裁剪成适应特定应用，从而可能导致程序的加速，或更易于执行其他的优化。例如，图15-5中的代码仅以incx=incy=1调用saxpy()，传播这一事实可使它的代码减少到如图19-39所示。注意，它也能使saxpy()的过程体更容易集成到sgefa()中。

bbs.theithome.com

```
      subroutine sgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      real a(lda,1)
      real t
      integer isamax,j,k,kp1,l,nm1
            . . .
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20         continue
               call saxpy_11(n-k,t,a(k+1,k),a(k+1,j))
   30       continue
            . . .
      subroutine saxpy_11(n,da,dx,dy)
      real dx(1),dy(1),da
      integer i,ix,iy,m,mp1,n
      if (n .le. 0) return
      if (da .eq. ZERO) return
      do 30 i = 1,n
          dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
```

图19-39 在确定出incx=incy=1，并传播此信息到saxpy()过程体

另外要注意的是，这种常数有时是由全局变量而不是由参数传送给过程的，这种情况也值得分析和利用。

对于体积较大对象(如数组和记录)的传值调用参数，将它的传递方式优化为传地址调用，取决于判定一个过程p的参数a是否有$a \in MOD(p)$。如果没有，则p和调用图中它的后裔都不会修改a，因此修改它的传递方式为传地址方式是安全的。这样做还需要改变访问实参和形参的代码为访问其地址而不是其值。另外，若在调用图中存在这样的点：我们知道参数在此点没有被改变（可能从过程接口而得知），但是却没有可用于改变参数传递的有效代码，则我们必须在那一点将参数传递转换回到传值方式。

最后，第12~18章讨论的许多过程内的优化都可以通过利用准确的过程间数据流信息而得到改善。既可以利用被调过程有关的信息改善围绕调用点的优化，也可利用参数和全局变量有关的信息改善过程内的优化。例如，当循环不变代码外提（参见13.2节）作用于一个含有调用的循环时，如果通过过程间方法能够确定此调用对一个过程内分析确定为循环不变的表达式没有副作用，就可以实现循环不变代码外提，从而使性能得到改善。

为了将过程间数据流信息用于全局（过程内）数据流分析，我们首先做过程间的分析，然后使用其结果初始过程入口和出口处的全局数据流信息，并将原本不透明的调用转变成能够理解它的数据流作用信息的操作。在某些情况下，重复地进行这种过程间的计算，之后再进行过程内数据流分析的处理可以获得显著的好处，但一般不值得大量增加它所消耗的编译时间，除非可以在程序员做其他事情的同时将它作为后台行为来进行，从而让用户感觉不到它。

例如，利用过程间分析提供的信息来初始过程入口点的常数信息，可以使得全局常数传播（参见12.6节）顾及到过程间的常数传播。

bbs.theithome.com

版权所有 侵权必究

Srivastava和Wall [SriW93]探讨了另一种整体的优化方法，这种方法作为连接处理中的一个较早步骤，它将一系列的目标模块作为输入和输出。在其原型版本中，它将每一个目标模块转换成寄存器转递语言(RTL)表示，做过程间控制流和活跃变量分析，然后做过程间死代码删除和全局与过程间循环不变代码外提。最后，它做全局复写传播、全局公共子表达式删除和过程间无用存数指令删除，之后将RTL转换回到要连接的目标模块。这种方法产生的结果好得令人惊奇，尤其是考虑到它除了与目标代码本身有关之外，它并不依赖于前面编译遍产生的信息。 658

19.6 过程间寄存器分配

另一种能对性能有相当大改善的过程间优化是过程间寄存器分配。这一节我们详细介绍一种由Wall [Wall86]开发的方法，并略述另外三种方法。这三种方法与Wall的方法的区别在于，Wall的方法是在连接时进行的，而这三种方法是在编译期间进行的。

19.6.1 连接时的寄存器分配

Wall发明了一种在连接时进行过程间寄存器分配的方法，这种方法综合了两种观察，这两种观察导致能够进行过程间分配和在过程体内随意使用图着色分配。我们在这里概述这种方法，并对它进行了适当提炼，以使它更加实用。

这两种观察中的第一种是，如果已生成的代码是完整的，也就是说不需要寄存器分配就能正确运行，则用于驱动连接时可选寄存器分配的注释就可以像重定位信息一样编码在目标模块中，并且它可用来驱动代码的修改，或者被忽略。

第二种观察是，如果程序调用图中的两条路径不相交，则在每一条路径上的过程可自由分配相同的寄存器。考虑图19-40的调用图便可阐明后一观察，图中每一个框表示一个过程，箭头表示调用。过程c和d没有相互调用，因此它们可以互不冲突地使用相同的寄存器。类似地，由e、f和g组成的链与由b、c和d组成的子树不相交，因此在这条链中和子树中可以使用相同的寄存器而不会有任何冲突。本节介绍的寄存器分配方法基于第二种观察，并结合了某种(独立选择的)过程内寄存器分配方法。 659

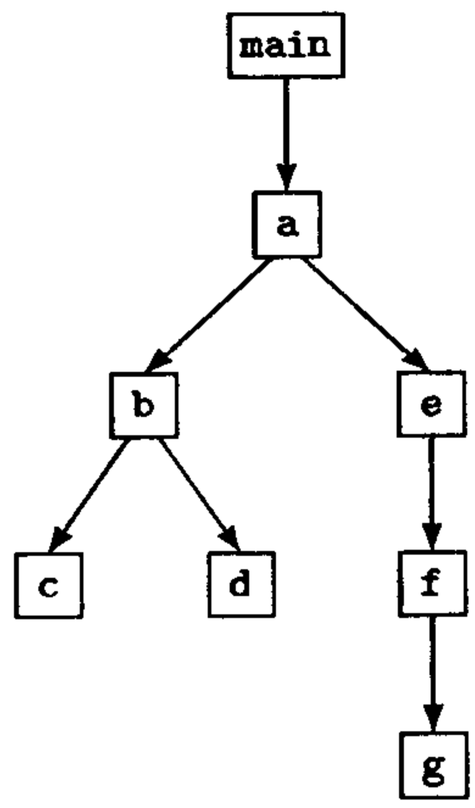


图19-40 过程间寄存器分配的例子

指导寄存器分配的注释由一些偶对组成，每一个偶对的第一个元素是rmv、op1、op2、res、lod和sto等6个运算符之一，它的第二个元素引用的是与这个注释相连的指令的操作数之一。

作为这种注释的第一个例子，考虑图19-41a中的LIR代码，其中x、y和z表示存储位置。rmv.*v*的含义是，若在连接时将*v*分配到寄存器中，则应当删除这条指令。op1.*v*、op2.*v* 和res.*v* 的含义是，若*v*被分配到寄存器中，则指令中与之对应的位置应当用分配给它的寄存器号来替换。于是，如果我们成功地将x和y分别分配到寄存器r6和r14中，则代码序列将变成如图19-41b所示。lod运算符用于那种从存储器取数，并且在基本块中最后一次使用这个被取值之前将对它进行修改的变量。sto运算符用于在当前使用之后还要使用的值。例如，C语句

```
x = y = a++ -b
```

将导致图19-42所示的带注释的代码。如果我们给a分配一个寄存器，就需要用寄存器到寄存器的复制指令替代第一条指令；如果给y分配一个寄存器，则需要用另一条复制指令替代存储y的指令。

bbs.theithome.com

超星公司制作 请尊重作者版权

```
r1 ← y          rmv.y
r2 ← z          rmv.z
r3 ← r1 + r2    op1.y op2.z res.x
x ← r3          rmv.x
```

a)

```
r2 ← z
r14 ← r6 + r2
```

b)

图19-41 a) LIR代码序列的例子和相连的注释，b) 给x和y分配寄存器之后的结果

```
r1 ← a          lod.a
r2 ← r1 + 1     res.a
a ← r2          sto.a
r3 ← b          rmv.b
r4 ← r2 + r3    op2.b
y ← r4          sto.x
x ← r4          rmv.x
```

图19-42 C语句x=y=a++-b对应的带注释的LIR代码

660 当然，如果我们删除这两条指令，则还需要修复引用这个指令序列的地址和偏移，不过，这样做所需要的信息已经存在于指令本身和它们的再定位信息中。

Wall生成注释的方法是产生三元式（参见4.9.1节）作为中间代码，然后从后向前扫描每一个基本块，标志如下的操作数:

1. 在与它们相连的变量被存储之后可能再次使用的操作数*v*（即那些需要lod.*v*而不是rmv.*v*的操作数）。

2. 那种被存储之后可能再次被使用的操作数（即那些需要sto.*v*而不是rmv.*v*的操作数）。

3. 其余可以用op1.*v*、op2.*v*、res.*v*或rmv.*v*注释的操作数。

然后，它生成目标模块，并记录模块中的过程于一张表中。并且，对每一个过程，记录它的局部变量、它引用的变量以及它调用的过程于一张表中，同时还附带记录每一个变量被引用的次数和每一个过程被调用的次数。

在连接处理的开始，一旦已确定了需要连接哪些模块，便启动过程间寄存器分配程序。该程序建立一个每个过程有一结点的调用图，并收集与过程的调用和变量用法有关的信息。调用图要求是一个DAG；这是需要特别对待的问题之一，因为它有可能没有精确地表示程序。具体地，递归调用和通过过程值变量的间接调用不能用和其他调用相同的方法来处理。递归调用如果精确地表示话，会使得调用图不是DAG，并且，如果它们的寄存器分配方法与其他例程的寄存器分配方法相同的话，则会导致它们重用与已经调用的同一例程所使用的寄存器相同的寄存器。类似地，由过程值变量调用的过程可能是若干个可能过程之中的任意一个过程，即使在连接时也可能无法精确地知道具体是哪个过程。

在估算出每个例程的总调用次数之后，分配程序按深度为主的逆序遍历这个DAG，如果可能的话，将那些可以分配到一个寄存器中的变量组合成一组。它将单个全局变量自成一组，不同时活跃的局部变量组成一组，每组关联一个引用频率。然后分配程序按频率对这些组排序，并在假定有R个寄存器可用于分配的情况下，指派寄存器给前面的R组（或用另一种方法，它可以在过程体内使用图着色分配程序）。最后，它根据这种分配信息和注释重写目标代码。

有若干问题需要进行特殊处理，或者用特殊技术可使其得到改善，具体如下:

661 1. 对于分配到寄存器中的有初值的全局变量，需要在代码中插入取数指令设置其初值。

2. 递归和间接调用既可以使用标准调用序列，或者对于递归调用，也可以使用特殊的调用序列来保护和恢复那些需要保护和恢复的寄存器。

bbs.theithome.com

越是公司制作请尊重作者版权

3. 可以给存储参数值至运行栈的指令加入形如par.*proc.v*的注释，指出被调用的过程是*proc*、被传递的参数是*v*，来适应传递参数在寄存器中而不是在存储栈中的情况。

4. 剖面分析可用更直接的反映程序实际特征的执行次数替代估计的引用频率，从而改善寄存器分配的效果。

5. 图着色可用来改善将变量组合成组时使用的活跃信息。这使得某些局部变量可以更有效率地被合并到分配组中。

19.6.2 编译时的过程间寄存器分配

有三种编译时进行过程间寄存器分配的方法曾在文献中讨论过，并且已在一些编译器中实现。

其中一种是由Santhanam和Odnert[SanO90]开发的，这种方法扩充了图着色寄存器分配（参见16.3节），它通过过程间活跃变量计算来确定与全局变量有关的网。它将调用图划分为网，每一个全局变量一个网，并分配具有最高使用频率的网到寄存器中，留下剩余的寄存器用于过程内分配。

由Chow[Chow88]开发的另一种方法已作为基于优先级的图着色寄存器分配（参见16.4节）的扩充而实现。它的主要目标是使得与过程调用有关的寄存器保护和恢复的开销最小化。它通过将(15.4.2节描述的)收缩包装和调用图的后序遍历结合到一起来做到这一点，其中调用图在调用点顾及了被调用过程寄存器的用法。因为对于多数调用而言，调用过程和被调用过程都是可见的，这种方法能够传递参数至任意寄存器。它假定所有寄存器都是调用者保护的，并且在调用图上尽可能远地推迟寄存器的保护和恢复。

第三种方法由Steenkiste和Hennessy[SteH89]开发，使用与Wall类似的方法分配寄存器，不过是在编译时进行的。它是为像Lisp这样的语言而设计的，这种语言的程序典型地由许多小过程组成，这导致过程间寄存器分配特别重要。它在其可知的范围内对调用图做深度为主遍历，并利用调用图的不同子树中的过程可以共享相同的寄存器这一原理，以及在每一个调用点可得到的关于被调过程寄存器用法的信息，自下而上地分配寄存器。这种方法很可能在分配过程的某个点上用尽所有的寄存器，此时，分配程序简单地切换成在过程入口保护并在过程出口恢复的标准过程内寄存器分配程序；对递归调用也用同样的方法来处理。 662

19.7 全局引用的聚合

另一种可以在连接时做的过程间优化是全局引用的聚合。由于RISC体系结构缺乏用单条指令指明一个32位地址的手段，它常常用两条指令来引用在任意地址的数据[⊖]，例如一个全局变量。*全局引用的聚合*（aggregation of global reference）能够提供一种手段使这种开销由多个引用来分摊，从而显著地减少这种开销。对于CISC体系结构，这可以缩短要用到的位移，并减少目标代码的大小。

这种技术是简单的——它实质上需要做的是，将全局数据集中到一个可以用较短的位移来引用的区域中，这个位移由存放在寄存器中的值确定。为了执行这种聚合，我们必须在编译期间保留一个寄存器（或一个以上的寄存器），这个寄存器通常称为全局指针（global pointer）或gp（参见5.7节）。我们扫描完整的目标模块，寻找那种表示访问全局变量的指令模式。对于32位的RISC系统，这涉及到寻找lui（就MIPS而言）、sethi（就SPARC而言）等指令，这种指令其后跟有一条使用由lui、sethi等设置的寄存器进行取数或存数的指令。我们将被存取的数据项（可能还有其初值）集中到一起，并修改在存取指令中使用的地址为引用相对全局指

⊖ 对于64位的RISC体系结构，这个问题更严重。在这种机器上，对于同一任务可能需要4到5条指令。

bbs.theithome.com

针的位移。在这一过程中，我们删除计算地址高部的多余指令，并在需要时重写剩余的指令以修复分支的位移和由于这条已删除指令而改变的任何其他指令。

一种更精致的全局引用聚合方法是对收集的全局数据按大小排序，以便使得可通过全局指针引用的数据项数最大化。

19.8 过程间程序管理中的其他主题

那种管理过程间关系的程序设计环境（也称之为大规模程序设计）使得编译有可能做一系列的优化课题，其中一些如前所述，另一些则涉及其他方面。尽管后者中没有特别困难的，但确实需要对它们引起注意。其中有一些已经在文献中有所讨论，而另一些仅仅在实验或实际程序设计环境中被提及。其中最重要的一些是:

1. 名字的作用范围：确定哪些模块由于全局变量名字的改变而受到了影响，以及对它们产
663 生的是什么影响，例如是否需要重新编译;

2. 重新编译：确定什么时候改变一个模块会需要重新编译，以及需要重新编译的模块的最小集合;

3. 同一模块(或库)中例程之间的连接和共享对象的连接：确定什么时候一个共享对象已被重新编译过，以及它对用到此模块的那些模块有些什么影响；我们在5.7节曾讨论过这一问题的某些方面。

Hall在她的博士论文中研究了其中的一些课题，以及其他一些过程间程序管理的课题[Hall91]。

19.9 小结

本章我们讨论了过程间控制流、数据流和别名分析，以及过程间信息对全局和过程间优化的应用。我们从Richardson和Ganapathi的关于过程间分析可能是不值得努力的研究开始，然后集中讨论了所需要的分析手段，以及通常值得进行过程间分析和优化的方面。

我们已看到，对于一个用相对质朴的语言（如Fortran）书写的完整程序，如果它是一次提交给编译器的，则构建程序的调用图是一件容易的事；如果是分开编译，则要困难一些。对于含过程值变量和递归的程序，例如PL/I或C，构建程序的调用图是PSPACE困难的。

我们区分了可能与一定信息、流敏感和流不敏感信息，并了解到精确地计算流敏感和一定信息其代价可能非常昂贵，一般是NP完全或co-NP完全的。因此，我们关注的是流不敏感可能数据流问题，如*MOD*，和某些代价较小的流不敏感一定问题，如过程间常数传播。它们有助于使过程适应常数值参数，并且对指导过程集成也有帮助。

我们也讨论了若干种过程间优化，主要是过程间寄存器分配和全局引用聚合，这两种优化都是在连接时进行的。Srivastava和Wall的工作已证明，很多标准的优化都很有可能在连接时进行。

在这些过程间的优化中，对于多数程序，过程间寄存器分配和常数传播较重要，而全局引用聚合的重要性相对要小一些。但是，过程间分析带来的好处主要是：由于它缩小了被调用过程可能影响的范围，从而改善了过程内优化的质量。

最后，我们提出了在那种管理大规模程序设计的程序设计环境中，最适合过程间优化自动完成的若干问题，如作用域、重新编译以及共享对象的连接。

我们按图19-43所示的优化执行顺序来放置过程间优化，在B框内增加了过程间常数传播
664 （位于过程内相同优化遍之前）以及过程特殊化和过程克隆。显然，仅当过程间常数传播已生成有用的结果时，我们才做后面的稀有条件常数传播遍。

bbs.theithome.com

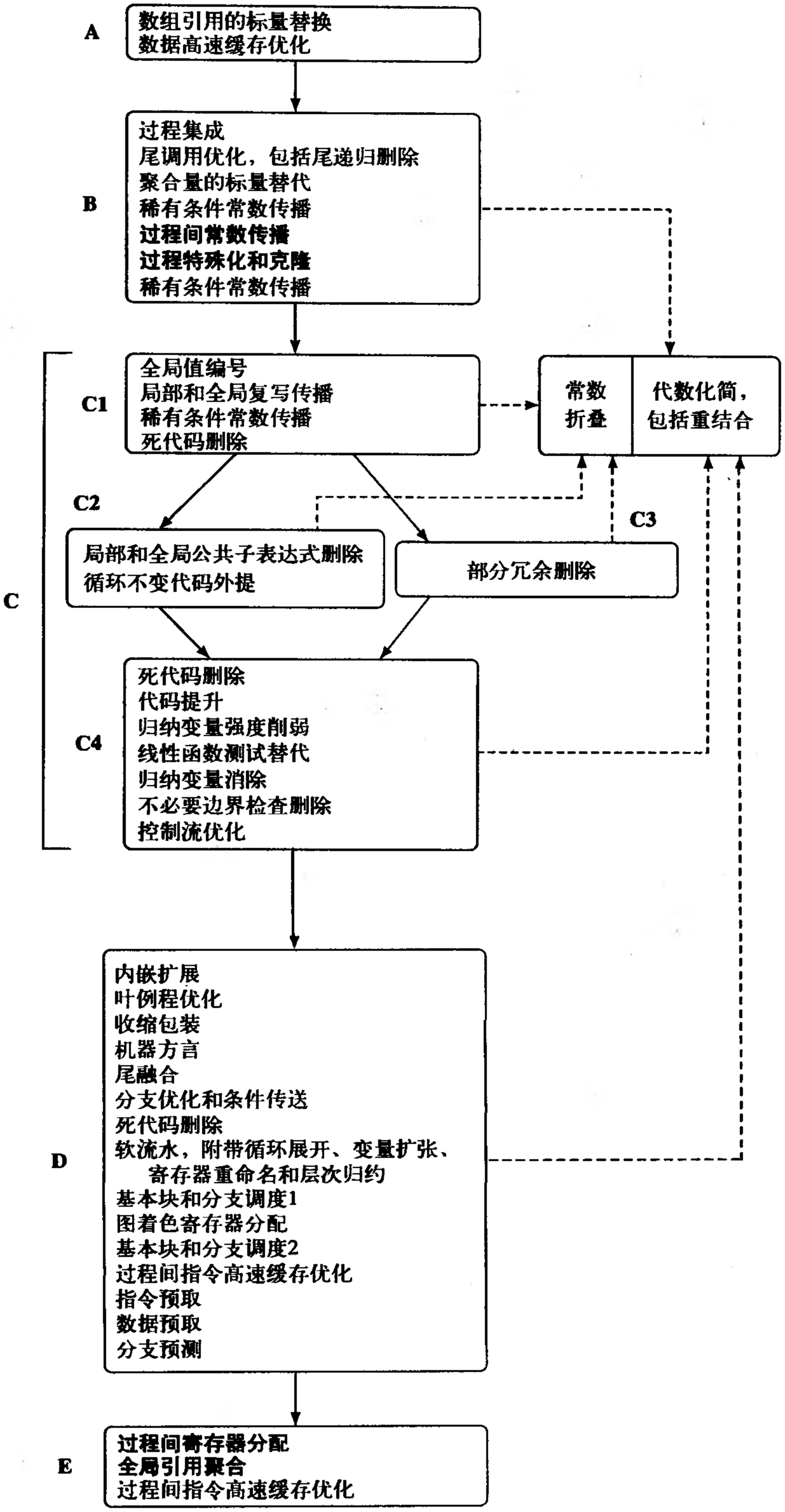


图19-43 优化顺序，过程间优化用黑体字标明

bbs.theithome.com

我们将过程间寄存器分配和全局引用聚合放在E框内，这两种优化最好是当所有过程的代码都已提交时在装载模块上来进行。

19.10 进一步阅读

Richardson和Ganapathi关于过程间优化效果的研究见[RicG89a]、[RicG89b]和[Rich91]。过程间优化可能更有助于并行编译器的根据是在[AllC86]和[Call88]中找到的。

Weihl关于构造含过程变量的递归程序的调用图是PSPACE-困难的证明是在[Weih80]中找到的。[Weih80]中讨论了计算过程副作用较困难问题中的某些问题，而另一些问题仅在已实现的系统中被提及，这些系统有ParaFrase[Kuck74]和ParaScope[CooH93]。

Cooper和Kennedy计算流不敏感副作用最早的方法见[CooK84]。但是，那种方法有一个如[CooK88a]指明的显著的缺点，它声称可以独立地解全局变量和形式参数子问题，而事实上，解形式参数问题必须先于非局部变量的处理，其方法由[CooK88b]提出。Myers的计算流敏感副作用方法的介绍见[Myer81]，Callahan的方法见[Call88]。

665 ~ 666

Callahan、Cooper、Kennedy和Torczon关于执行过程间常数传播的方法见[CalC86]。Grove和Torczon关于转移函数和返回转移函数，它们的计算代价，以及它们所提供的信息的讨论见[GroT93]。

Cooper和Kennedy的过程间别名分析的方法见[CooK89]。[Deut94]介绍了一种更强有力的方法。

介绍Wall的在连接时进行过程间寄存器分配方法的论文是[Wall86]。本章介绍的在编译期间进行过程间寄存器分配的几种方法是由Santhanam和Odnert [SanO90]、Chow [Chow99]，以及Steenkiste和Hennessy [SteH89]开发的。

关于全局引用聚合的讨论见[HimC87]、[Hime91]和[SriW94]。[SriW94]还讨论了将这种技术扩展到64位RISC体系结构的方法。

[SriW93] 介绍了Srivastava和Wall的连接时优化的相关工作。

关于过程间程序管理或“大规模程序设计”的问题，有一些在文献中有所讨论，例如Cooper、Kennedy和Torczon的论文[CooK86]以及Hall的论文[Hall91]，而另一些只在已实现的系统中被涉及，这些系统有ParaFrase和ParaScope。

19.11 练习

19.1 (a)写一个程序，它至少包含5个不同的过程并且含有对它们的调用，其中至少包含一个递归调用。(b)对于你的例子，说明图19-3中过程`Build_Call_Graph()`的执行步骤。

RSCH 19.2 通过观察到迭代i的`LowLink(p)`小于或等于迭代$i+1$的`LowLink(p)`这一情况，用图19-18的过程`Compute_GMOD()`和`GMOD_Search()`计算*GMOD*的复杂性可以减少到只有原复杂性的d分之一（d是程序中过程嵌套的深度）。写出由此产生的这两个过程的代码。

19.3 (a)修改19.2.1节给出的计算*DMOD*的方法使之计算*DREF*，(b)并将它应用于图19-10中的程序。

19.4 (a)利用转移函数和返回转移用函数，用公式写出做调用点特殊的过程间常数传播分析的ICAN算法，(b)将它应用于图19-44中的程序，其中`halt()`是一个终止程序执行的例程。

bbs.theithome.com

```
                procedure main( )
        main    begin
         1          read(t)
         2          call f(t)
         3          call f(2)
                end
                procedure f(b)
        f       begin
         1          print(g(b,1))
                end
                procedure g(x,y)
        g       begin
         1          if y > 0 then
         2             call h(x,y,1)
         3             x := 2
         4          else
         5             call h(3,y,1)
         6             halt( )
         7          fi
                end
                procedure h(u,v,w)
        h       begin
         1          if u > w then
         2             call g(v,-1)
         3          else
         4             return w + 1
         5          fi
                end
```

图19-44 对它执行位置特殊过程间常数传播的一个例程序

19.5 (a)构造一个至少含3个例程且至少有一个全局变量的C程序，(b)对它执行19.4.2节介绍的位置特殊的别名分析。

19.6 写出一个ICAN例程，它以LIR过程体作为输入，并用类似于19.6.1节讨论的那种寄存器分配注释标注它的指令。 667 ~ 668

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

第20章　存储层次优化

本章关注的主要是能更好地利用存储器层次，尤其是数据和指令高速缓存的代码优化技术，同时也介绍一种可改善数组元素寄存器分配的方法。

从最早的计算机设计开始，几乎在所有的系统中，主存储器和寄存器之间就已经有了区别。主存储器一直比寄存器集合要大，从它读取和存储数据也比对寄存器的相同操作要慢。许多系统都要求，除了取和存之外的所有操作，其操作数如果不是全部的话，至少也要有一个操作数是在寄存器中。当然，在RISC系统中这种做法达到了极至，它要求除了取和存之外的所有操作都对寄存器中的操作数来执行，并且操作结果也存放在寄存器中。

随着时间的推移，处理器的时钟周期和访问存储器所需要的时间之间的差距已增加到这样一个地步，以至于当所有的存取操作都必须以主存储器的速度运行时，它成为了性能降低的重要原因。而且这种差距正在变得更为严重：主存储器的访问速度每年增长10%~20%，而处理器速度的增长达到了每年50%。为了弥补这种差距，在寄存器和主存储器之间设计了高速缓冲存储器，或简称高速缓存（cache），以减轻这种速度的不匹配。高速缓存有选择地复制主存储器的某个部分，这种选择一般是根据需要，以硬件或者硬件和软件协同的方式来确定的。对一个定向在高速缓存中的地址执行取数、存数或取指令操作，可以从高速缓存中得到满足而不需要通过主存储器（或在存数的情况下，有可能并行存储到主存储器中），并且在理想情况下，其结果的延迟不超过处理器的两个时钟周期。对于那种具有暴露的或部分暴露的流水线的系统（例如RISC和Intel 386体系结构的高端实现），取数指令和分支指令的完成一般需要两个时钟周期，但是，第二个时钟周期通常可用来执行另一条指令（在取数的情况下，这条指令必须是不使用所取之值的指令）。 669

高速缓存的效率取决于程序的空间和时间的局部性性质。如果一个程序重复地执行一个循环，则在理想情况下，第一个迭代将使它的代码被取至高速缓存中，后继的迭代从高速缓存来执行它，而不需要每次重新从主存装载。因此，第一个迭代可能有读指令到高速缓存的开销，但后继迭代一般不会有。类似地，如果需要重复地使用一个数据块，理想的是将它取到高速缓存中，然后从高速缓存中访问它，同样只在第一次使用它时有将它从主存读到高速缓存的开销。另一方面，当代码和数据在高速缓存中发生冲突，即它们占据高速缓存中部分相同的位置时，或者由于数据的若干数据段被映射到相同的高速缓存块而导致数据本身自相冲突时，可能会导致性能明显地下降。在最坏的情况下，对高速缓存取数、存数和取指令都不比访问主存储器快。

一个系统可以有独立的数据高速缓存（data cache）和指令高速缓存（instruction cache）（也分别叫做D-cache和I-cache），也可以有同时容纳数据和指令的合并的或统一的高速缓存（combined or unified cache）。另外，具有固定页面调度的系统还包含了另外一种高速缓存，即后备转换缓冲区（translation-lookaside buffer），或简称TLB，它是用来存放虚存地址到物理地址转换信息（或反之）的一个高速缓存。

本章从讲述一些事例开始，这些事例形象地描述了可能的高速缓存用法和优化方法，以及它们的影响。之后的几节介绍指令预取以及过程内和过程间指令高速缓存的优化方法。接下来一节介绍如何使数组元素利用寄存器分配的优势，后面的几节介绍数据高速缓存优化方法，并给出有关的综述。最后，我们讨论标量和面向存储器的优化之间的互相影响，以及将存储器有关的优化集成至激进优化编译器结构中相关的问题。

bbs.theithome.com

20.1 数据和指令高速缓存的影响

Bell报告了手工转换数值密集型Fortran和C程序，以便更好地利用IBM RS/6000的流水线和高速缓存的效果。他考察了性能是如何随着被访问数组索引的步长（stride）而变化的。在一个块长为64字节的D-cache中，一块可容纳8个双精度浮点值，因此D-cache的性能直接与所使用的步长有关。具体地，他的数据表明，对于小于等于32的步长，性能随步长增加的对数而降低。对于大于32的步长，TLB的缺失开始成为性能降低的主要原因——当步长是4096或更大时，每一个存储引用有一个TLB缺失，导致性能小于最好性能的3%。

因此，在存储器中这样来安排数组，使得能以尽可能小的步长来访问它们，通常是实现数据高速缓存高性能的关键。对于那种多次使用一个数组的代码而言，达到这一目的一种方法是，将数组中要使用的不连续的数组元素复制到另一个数组的连续位置，从而使得数据能以步长1
670 来访问，最后再将在处理过程中改变了的数组元素复制回去。

作为D-cache优化潜在影响的一个例子，考虑由Bell描述的用Fortran写的双精度矩阵乘的4个版本。这4个版本（如图20-1所示）如下：

a) MM：标准教科书中A乘以B的三层嵌套循环；

b) MMT：A在主存中被转置；

c) MMB：用大小为t的瓦片铺砌（tiling）这三个循环后的结果；

d) MMBT，同时做循环铺砌和将A转置的结果。

Bell报告了在IBM RS/6000 Model 530系统上对这4个版本以N＝50所测得的性能数据。原始版本的性能随矩阵的大小和循环组织而变化，幅度超过14倍；而铺砌和转置得到的性能接近最大性能，并且性能不受数组大小的影响。中间这两个版本的性能几乎相同。

```
do i = 1,N
  do j = 1,N
    do k = 1,N
      C(i,j) = C(i,j)
        + A(i,k) * B(k,j)
    enddo
  enddo
enddo
```

a) MM

```
do i = 1,N
  do j = 1,N
    do k = 1,N
      C(i,j) = C(i,j)
        + A(k,i) * B(k,j)
    enddo
  enddo
enddo
```

b) MMT

```
do j = 1,N,t
  do k = 1,N,t
    do i = 1,N
      do jj = j,min(j+t-1,N)
        do kk = k,min(k+t-1,N)
          C(i,jj) = C(i,jj)
            + A(i,kk) * B(kk,jj)
        enddo
      enddo
    enddo
  enddo
enddo
```

c) MMB

```
do j = 1,N,t
 do k = 1,N,t
  do i = 1,N
    do jj = j,min(j+t-1,N)
     do kk = k,min(k+t-1,N)
       C(i,jj) = C(i,jj)
         + A(kk,i) * B(kk,jj)
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

d) MMBT

图20-1 Fortran的4个矩阵乘版本：a) MM，通常的形式；b) MMT，A被转置；c) MMB，铺砌j和k循环；d) MMBT，同时转置和铺砌

bbs.theithome.com

作为I-cache对性能潜在影响的例子，我们提供两个实例。第一个是关于在这样一个系统上编译Linpack程序的例子。这个系统的高速缓存块为32字节，处理器每个时钟周期可以取4条4字节的指令，并且每个时钟周期可流出3条指令。Linpack程序的核心是saxpy循环。saxpy循环的第一条指令落在了一个高速缓存块的末尾，因此，指令缓冲区[⊖]只能取到这一条有用的指令，并且在循环体执行的第一拍，只有这条指令被执行；当然，此循环的每一个迭代都维持这种模式。 671

第二个实例与SPEC基准测试程序gcc有关。当没有做I-cache优化时，这个程序频繁地遇到百分比相当高的、由于I-cache缺失导致的停顿的时钟周期（约10%）。出现这种现象的主要原因是高速缓存的能力问题——gcc有很大的工作集——不过利用下面讨论的方法可使它得到改善。这种方法将每一个过程体中频繁执行的代码与很少执行的代码区别开来。

20.2 指令高速缓存优化

下面几小节介绍改善指令高速缓存命中率的若干种方法。这些方法中有两种是过程间的，一种是过程内的，另外的三种既有过程内的特征，也有过程间的特征。所有这些方法都尝试重排代码以减少工作集的大小并降低冲突。

20.2.1 利用硬件辅助：指令预取

在各种体系结构的许多实现中，硬件提供顺序预取代码的功能，并且在某些情况下，也能够从分支下降路径之外的分支路径预取代码。

有一些较新的64位RISC，如SPARC-V9和Alpha，给软件提供了指令预取支持。这使得程序员或编译器有可能给系统的I-cache和指令预取部件提供预取提示，以指出一块代码或一组代码块是程序不久之后将需要的，因此应当在有空闲的总线周期时，将它们取到高速缓存中。例如，对于SPARC-V9，流出伪指令

```
iprefetch  address
```

预取给定地址中包含的代码块。

软件预取主要对第一次要预取的代码块有用；或者是那种重复使用，并且通过剖面分析发现，当需要它们时很可能不在I-cache中的代码块。在第一种情况下，预取指令是否有用取决于硬件是否缺乏对顺序和分支预取的支持。一旦确定出一个代码块可由软件预取获益，则在代码中从此代码块将被执行的那一点回退T_{pref}个时钟周期的位置放置一条适当的预取指令，其中T_{pref}是满足预取需要的时间。如果是否需要一个代码块取决于该块之前t个时钟周期位置上的一个条件，则将这条预取指令放置在需要此代码的路径上从那一点回退$\min(T_{pref}, t)$个时钟周期的代码位置上。 672

18.11节讨论的分支预测优化能够加强指令预取的效果。

20.2.2 过程排序

一种最容易应用，并且也肯定能得到好处的I-cache优化技术，是在连接时根据过程之间的调用关系和使用频率，对构成一个目标模块的那些静态连接子程序进行排序。排序的目的是在虚存中将子程序放在它们的调用者附近以减少页交换，并且使得频繁使用的和有关联的子程序在I-cache中以相互之间冲突较小的方式放置。这种方法依据的假定是，子程序和它们的调用者在时间上可能相互接近，因此应当在放置时使得它们在空间上互不冲突。如果可以得到剖面分析反馈的信息，应当在这种处理中将它们考虑进来；如果得不到这种信息，可以通过启发式来

⊖ 指令缓冲区（instruction buffer）是一个从高速缓存传送指令到执行部件的硬件队列。

bbs.theithome.com

进行控制，这种启发式将相互频繁调用的子程序以相邻的方式放置在一起（例如，内层循环的调用应当比不在循环内的调用的权重更高）。

为了实现这种想法，我们从无向静态调用图开始，图中每一条弧标有它两端所连接的两个过程之间每一个调用另一个的次数。之所以使用无向图是因为，在两个过程之间可能存在两个方向的调用，并且每一个调用相应地匹配有一个返回。然后，我们逐步瓦解这个图，每一步选择一条具有最高权重的弧，合并它所连接的两个结点为一个结点，合并这两个结点相应的弧，并将被合并弧的权重加在一起而计算出合并弧的标号。在过程的最后排序中，那些被合并的结点相互放在一起，并用原图中的连接权重来确定它们之间的相互顺序。

进行这一处理的ICAN算法是图20-2给出的例程Proc_Position()。我们假定调用图是连通的；如果不是连通的，即它的某些过程没有用，我们这个算法则只作用于包含根结点的连通部分（至多一个）。

```
ProcSeq = Procedure ∪ sequence of ProcSeq

procedure Proc_Position(E,weight) returns sequence of Procedure
   E: in set of (Procedure × Procedure)
   weight: in (Procedure × Procedure) → integer
begin
   T, A := ∅: set of ProcSeq
   e: Procedure × Procedure
   a, emax: ProcSeq
   psweight: ProcSeq → integer
   max: integer
   for each e ∈ E do
      T := A ∪= {[e@1,e@2]}
      psweight([e@1,e@2]) := weight(e@1,e@2)
   od
   repeat
      max := 0
      for each a ∈ A do
         if psweight(a) > max then
            emax := a
            max := psweight(a)
         fi
      od
      Coalesce_Nodes(T,A,weight,psweight,emax↓1,emax↓2)
   until A = ∅
   return Flatten(♦T)
end    || Proc_Position
```

图20-2 过程排序算法

ProcSeq的两元素成员，如$[a_1, a_2]$，表示一棵根不带标号的二叉树。我们也使用ProcSeq的较长元素成员，如$[a_1, \cdots, a_n]$，来表示n个元素的序列。如果结点$p1$和$p2$之间没有相连的弧，我们定义weight([$p1$, $p2$])=0。函数left(t)和right(t)分别返回树t的左子树和右子树。函数leftmost(t)和rightmost(t)分别返回树t的最左和最右叶结点。函数maxi(s)返回序列s中第一个具有最大值的元素的索引i，函数reverse(s)颠倒序列s和它的所有子序列。例如，maxi([3,7,4,5,7])=2，reverse(1,[2,3])=[[3,2],1]。

673

图20-3给出的函数Coalesce_Nodes(T, A, *weight*, *psweight*, $p1$, $p2$)将结点$p1$和$p2$合并为一个结点，在此过程中同时调整T、A和*pweight*的值。因为我们用两元素的序列表示无序的偶

bbs.theithome.com

对，并且序列元素本身也可以是无序偶对，因此我们需要完成以下功能的一些例程，这些例程对有序偶对而言是很容易实现的。

```
procedure Coalesce_Nodes(T,A,origwt,psweight,p1,p2)
   T, A: inout set of ProcSeq
   origwt: in ProcSeq ⟶ integer
   psweight: inout ProcSeq ⟶ integer
   p1, p2: in ProcSeq
begin
   lp1, rp1, lp2, rp2: Procedure
   p, padd: ProcSeq
   i: integer
   || select ProcSeqs to make adjacent and reverse one if
   || necessary to get best order
   lp1 := leftmost(p1)
   rp1 := rightmost(p1)
   lp2 := leftmost(p2)
   rp2 := rightmost(p2)
   i := maxi([origwt(rp1,lp2),origwt(rp2,lp1),
      origwt(lp1,lp2),origwt(rp1,rp2)])
   || form new ProcSeq and adjust weights
   case i of
1:    padd := [p1,p2]
2:    padd := [p2,p1]
3:    padd := [p1,reverse(p2)]
4:    padd := [reverse(p1),p2]
   esac
   T := (T ∪ {padd}) - {p1,p2}
   A := Remove(A,[p1,p2])
   for each p ∈ T (p ≠ padd) do
      psweight([p,padd]) := 0
      if Member(A,[p,p1]) then
         A := Remove(A,[p,p1])
         psweight([p,padd]) += psweight([p,p1])
      if Member(A,[p,p2]) then
         A := Remove(A,[p,p2])
         psweight([p,padd]) += psweight([p,p2])
      fi
      A ∪= {[p,padd]}
   od
end    || Coalesce_Nodes
```

图20-3 图20-2所使用的结点合并过程

1. Same(*p*1, *p*2)返回true，如果不考虑顺序*p*1和*p*2是相同的序列；否则返回false。例如，Same([1, [2, 3]], [[3,, 2], 1])返回true，而Same([1, [2, 3]], [[1, 2], 3])返回false。

2. Member(*A*, [*p*1, *p*2])返回true，如果不考虑元素顺序，[*p*1, *p*2]是集合*A*中的一个成员；否则返回false。

3. Remove(*A*, *s*)，如图20-4所示，从*A*中删除用Same()不能与s区别的任何元素，并返回结果。 674

4. Flatten(*T*)，如图20-5所示，从左至右遍历由序列表示的二叉树，并使它变平，即构造树的叶子结点的序列，并返回此序列。

bbs.theithome.com

版权所有 侵权必究

```
procedure Flatten(t) returns sequence of Procedure
   t: in ProcSeq
begin
   if leaf(t) then
      return [t]
   else
      return Flatten(left(t)) ⊕ Flatten(right(t))
   fi
end  || Flatten

procedure Remove(A,s) returns Procseq
   A: in set of ProcSeq
   s: in ProcSeq
begin
   t: ProcSeq
   for each t ∈ T do
      if Same(s,t) then
         A -= {s}
         return A
      fi
   od
   return A
end    || Remove
```

图20-4 过程排序算法中使用的辅助例程

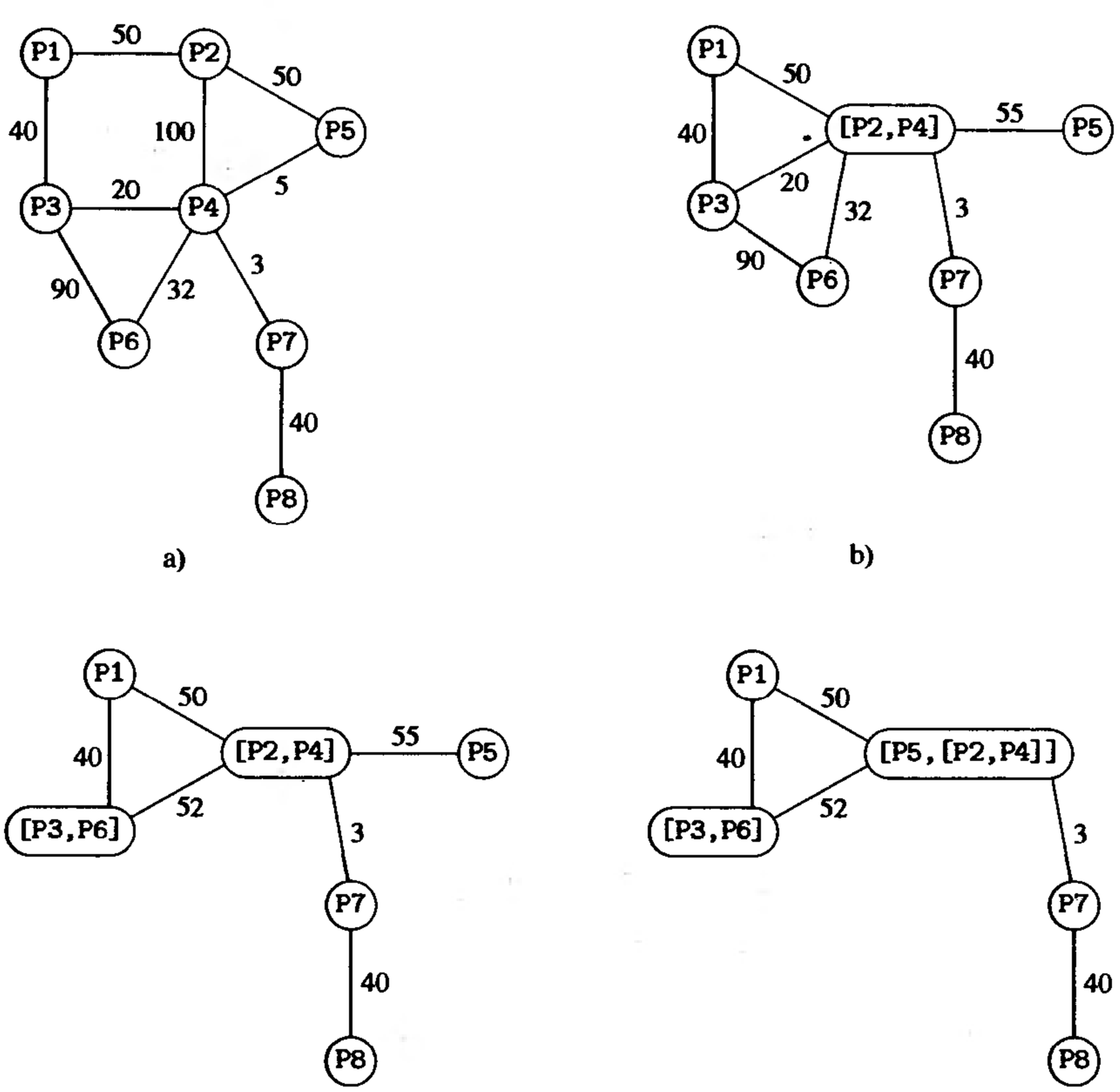


图20-5 a) 用于过程排序的例子，b) 到d) 为此流图的前三步转换

bbs.theithome.com

作为过程排序算法的一个例子，假设我们从图20-5a中的带权重的无向调用图开始。第一步，合并P2和P4形成[P2,P4]，并得到图20-5b。下一步，合并P3和P6形成[P3,P6]，然后合并P5和[P2,P4]形成[P5,[P2,P4]]，分别得到图20-5c和20-5d。由这个合并过程得到的单结点的内容为:

```
[[P1,[P3,P6],[P5,[P2,P4]]],[P7,P8]]
```

所以我们在存储器中按顺序P1、P3、P6、P5、P2、P4、P7、P8安排这些过程。注意，P2和P4已经相邻放置，P3和P6也这样，另外还有P5和P2。

20.2.3 过程和基本块的放置

另一种可以与上面介绍的方法结合在一起的I-cache优化方法需要修改系统的连接程序，将每一个子程序放置在I-cache块的边界。以便允许编译过程在较后的遍中安置频繁执行代码段（如循环）占据尽可能少的高速缓存块，并放置它们在高速缓存块的开始或接近开始的地方，从而有助于使得I-cache的缺失最小，并使得超标量CPU更有可能取到能同时执行的完整指令组。如果多数基本块都很短（比如说，4～8条指令），这种方法则有助于使这些基本块的开始不会位于高速缓存块的末尾。编译器可以不太困难地插入收集这种统计信息的计量桩，由此得到的剖面分析反馈信息则可以用来确定，与简单地浪费代码空间和导致额外的分支相比较，什么样的基本块进行这种放置是重要的。

20.2.4 过程内的代码安置

我们这里将介绍Pettis和Hansen [PetH90]开发和评测的一种安置过程内代码的自底向上方法。这种方法的目的是将不频繁执行的代码（如错误处理）移到代码的主体之外，并拉直代码（即删除无条件分支指令，并尽可能使得发生条件转移的分支走下降分支），以便在一般情况下取到I-cache中的指令实际被执行的比例较高。与过程排序不同，这种处理可以在每一个过程的编译期间进行。为了做到这一点，我们假设在过程的流图中，所有的边都已标有它们的执行频率，执行频率可以通过剖面分析或通过估计而得出。这个算法自底向上搜索流图，在那些由于它们之间的边频繁执行而应当作为直线代码放置的基本块之间建立一条链[⊖]。一开始，每一个基本块链到它自身。然后，在后继的步骤中，将那种其中一条链的链尾和另一条链的链头由一条具有最高执行频率的边相连的两条链合并到一起；如果具有最高执行频率的边不能连接一条链的链尾和另一条链的链头，这两条链就不能合并。尽管如此，这种边在代码放置中也是有用的：这个最高执行频率信息被保存下来，使得当我们假定这条边是向前分支时[⊖]，如果可能的话，使其目标基本块能够放置在源基本块之后的某个位置。最后，我们放置基本块，首先选择入口所在的链，然后根据链的权重依次放置其他链。过程Block_Position(*B*, *E*, *r*, *freq*) 的ICAN代码，以及它用到的过程Egde_Count()如图20-6所示，其中*B*是结点（即基本块）集合，*E*是边集合，*r*是入口结点，*freq*()是映射边到它们的执行频率的函数。

675～677

作为此算法的一个例子，考虑图20-7中的流图。具有最高执行频率的边是从B1到B2的边，因此，形成的第一个序列是[B1,B2]。下一个有最高执行频率的边是从B2到B4的边，因此现存的序列扩充为[B1,B2,B4]；类似地，接下来的两步加入entry和B8，得到[entry,B1,B2,B4,B8]。下一个执行频率最高的边是从B9到exit的边，因此开始一个新的序列[B9,exit]。后面的几步将这个序列扩充为[B6,B9,exit]，并形成了另外两个序列

⊖ 这些链与trace调度（17.5节）中的trace有点相似，但它不需要修复代码。

⊖ 当然，可以修改算法的这一假设，使之对应于条件分支有关的其他假设。

bbs.theithome.com

[B3,B7]和[B5]。接下来我们计算edges()函数如下:

```
edges([entry,B1,B2,B4,B8]) = 2
edges([B3,B7]) = 1
edges([B5]) = 1
edges([B6,B9,exit]) = 0
```

```
procedure Block_Position(B,E,r,freq)
   returns sequence of Node
   B: in set of Node
   E: in set of (Node × Node)
   r: in Node
   freq: in (Node × Node) ⟶ integer
begin
   C := ∅: set of sequence of Node
   CR: sequence of Node
   c, c1, c2, clast, cfirst, cnew: sequence of Node
   nch, oldnch, max, fr: integer
   edges: (set of sequence of Node) ⟶ integer
   e: Node × Node
   for each b ∈ B do
      C ∪= {[b]}
   od
   nch := |C|
   repeat
      oldnch := nch
      || select two sequences of nodes that have the
      || highest execution frequency on the edge that
      || connects the tail of one to the head of the
      || other
      max := 0
      for each c1 ∈ C do
         for each c2 ∈ C do
            if c1 ≠ c2 & (c1↓-1)→(c2↓1) ∈ E then
               fr := freq(c1↓-1,c2↓1)
               if fr > max then
                  max := fr
                  clast := c1
                  cfirst := c2
               fi
            fi
         od
      od
      || combine selected pair of sequences
      if max > 0 then
         cnew := clast ⊕ cfirst
         C := (C - {clast,cfirst}) ∪ {cnew}
         nch -= 1
      fi
   until nch = oldnch
   while C ≠ ∅ do
      || find sequence beginning with entry node and
      || concatenate sequences so as to make as many
      || branches forward as possible
      CR := ♦C
      if r = CR↓1 then
```

图20-6 自底向上基本块安置算法

bbs.theithome.com

```
            C -= {CR}
            for each c1 ∈ C do
               edges(c1) := 0
               for each c2 ∈ C do
                  if c1 ≠ c2 then
                     edges(c1) += Edge_Count(c1,c2,E)
                  fi
               od
            od
            repeat
               max := 0
               for each c1 ∈ C do
                  if edges(c1) ≥ max then
                     max := edges(c1)
                     c := c1
                  fi
               od
               CR ⊕= c
               C -= {c}
            until C = ∅
            return CR
         fi
      od
   end    || Block_Position

   procedure Edge_Count(c1,c2,E) returns integer
      c1, c2: in sequence of Node
      E: in Node × Node
   begin
      ct := 0, i, j: integer
      for i := 1 to |c1| do
         for j := 1 to |c2| do
            if (c1↓i)→(c2↓j) ∈ E then
               ct += 1
            fi
         od
      od
      return ct
   end    || Edge_Count
```

图20-6 （续）

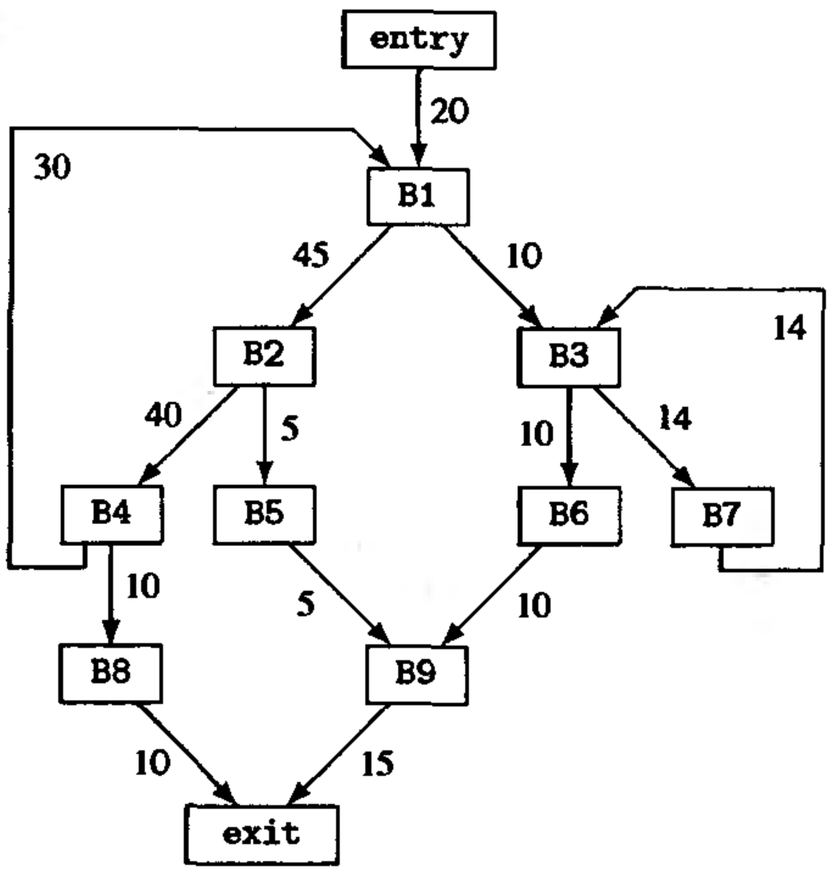


图20-7 用于过程内代码安置的流图例子

bbs.theithome.com

然后我们排列这些序列，使得包含entry的序列在前，其他三个序列按edges()函数给出的顺序跟随在后，并返回这个结果。由此得到的存储器中的安排如图20-8所示。

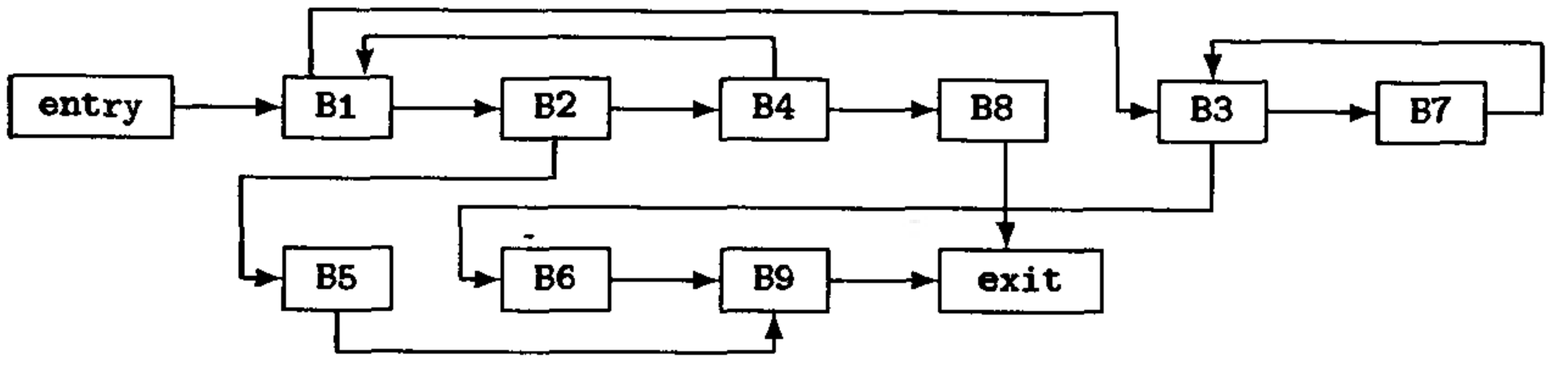


图20-8 图20-7中的流图例子经过过程内代码安置之后

最后，我们通过插入和删除为保持此流图效果上与原流图等价而需要的分支来对代码进行修复。这是通过合并原基本块的进入边、这些基本块的开始标号以及它们的结束分支来实现的。注意，在我们的例子中，我们已成功地使得除两个循环闭合分支之外的所有分支都是向前分支，并且已经通过适当地安排基本块，使得9条前向边中的6条边是下降边。

678
~
680

20.2.5 过程分裂

一种增强过程排序和过程内代码安置算法效果的技术是过程分裂（procedure splitting），它将一个过程分为主要成分和次要成分，前者包含频繁执行的基本块，后者包含诸如例外处理之类的很少执行的基本块。然后将一系列过程的次要成分集中到一个独立的次要段中，从而使得主要成分更紧密地在一起。当然，过程分裂需要调整各个成分之间的分支。决定主要成分和次要成分执行频率分界线的依据应当是实验。

图20-9给出了一个过程分裂的示意性例子。标有“p”和“s”的区域分别表示主要和次要代码。

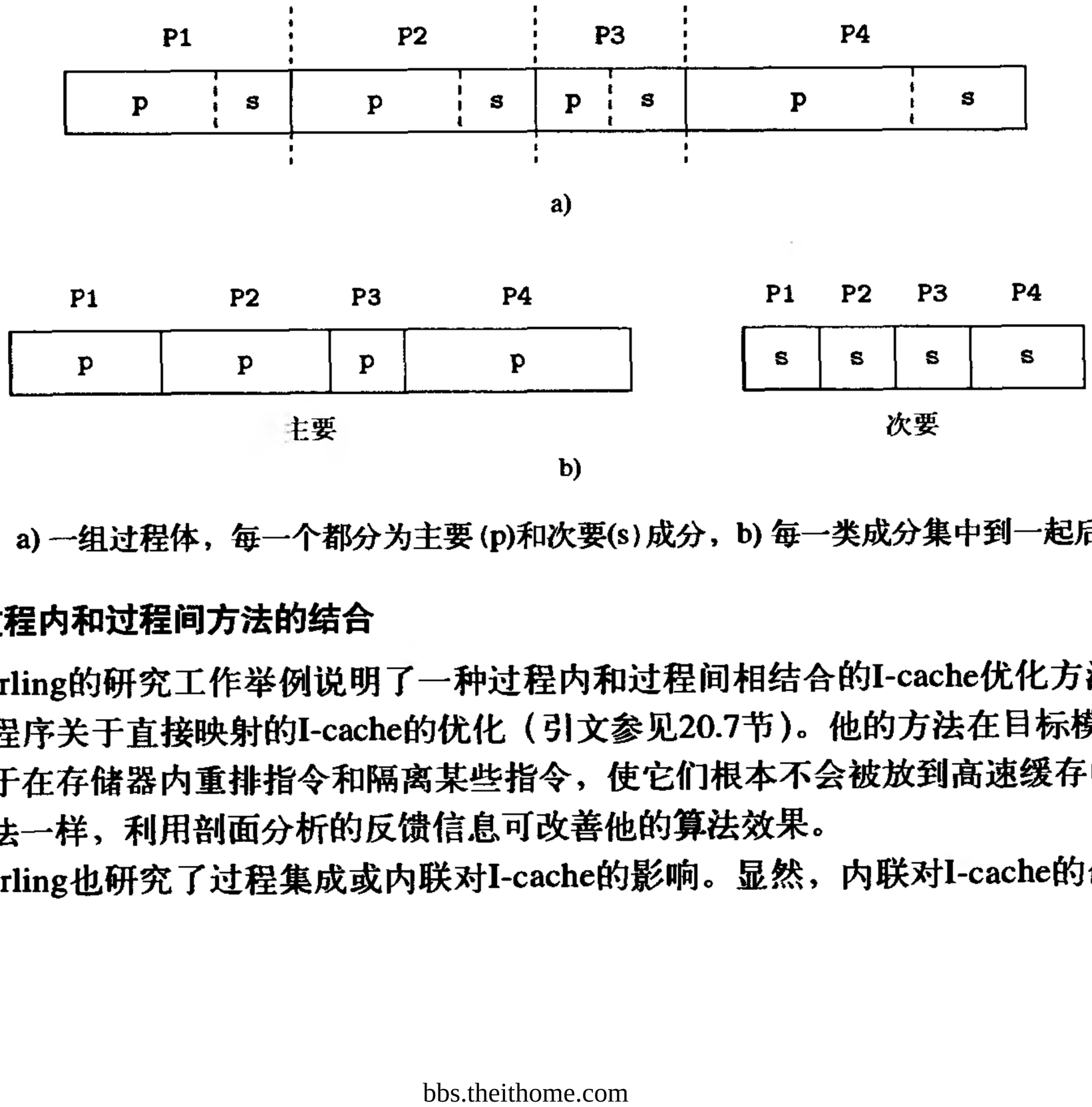


图20-9 a) 一组过程体，每一个都分为主要(p)和次要(s)成分，b) 每一类成分集中到一起后的结果

20.2.6 过程内和过程间方法的结合

McFarling的研究工作举例说明了一种过程内和过程间相结合的I-cache优化方法，他关注的是整个程序关于直接映射的I-cache的优化（引文参见20.7节）。他的方法在目标模块上工作，并且依赖于在存储器内重排指令和隔离某些指令，使它们根本不会被放到高速缓存中。同上面讨论的方法一样，利用剖面分析的反馈信息可改善他的算法效果。

McFarling也研究了过程集成或内联对I-cache的影响。显然，内联对I-cache的命中率既可

bbs.theithome.com

超星公司制作 请尊重作者版权

有正面的影响，也可有负面的影响。如果内联的过程太多，代码体积可能会有指数级的增长，其结果会导致I-cache命中率受损害；另一方面，内联也增加了局部性，因此降低了I-cache的缺失率。他给出了一个模型用于确定内联单个调用的效果，并给出了内联过程的判别标准。

20.3 数组元素的标量替换

在为循环生成较好的目标代码时涉及到的一个问题是，很少有编译器尝试将那种带下标的变量分配到寄存器中，尽管寄存器分配通常对标量非常有效。在深入考虑这种分配方法的细节之前，我们先给出两个例子说明它究竟能有多么重要。

第一个例子是图20-1a给出的矩阵乘代码。如果我们如图20-10所示，用变量ct替代C(i,j)，并将ct分配到寄存器中，则可以将存储访问的次数减少$2 \cdot (N^3 - N^2)$次，或减少几乎近一半。

```
do i = 1,N
   do j = 1,N
      ct = C(i,j)
      do k = 1,N
         ct = ct + A(i,k) * B(k,j)
      enddo
      C(i,j) = ct
   enddo
enddo
```

图20-10 用标量临时变量替换了图20-1a中的C(i,j)的Fortran矩阵乘

681 ~ 682

第二个例子是图20-11a中的代码。对b[]的引用可以用两个临时变量来替换，这导致了图20-11b中的代码，它减少了存储访问次数约40%。当然，如果已知n大于或等于1，则可以删除其中的if。

```
for i ← 1 to n do
   b[i+1] ← b[i] + 1.0
   a[i] ← 2 * b[i] + c[i]
endfor
```

a)

```
if n >= 1 then
   t0 ← b[1]
   t1 ← t0 + 1.0
   b[2] ← t1
   a[1] ← 2 * t0 + c[1]
endif
t0 ← t1
for i ← 2 to n do
   t1 ← t0 + 1.0
   b[i+1] ← t1
   a[i] ← 2 * t0 + c[i]
   t0 ← t1
endfor
```

b)

图20-11 a) 一个简单的HIR循环，b) 对它使用标量临时变量的结果

用标量替换带下标的变量，从而使得寄存器分配对它们有效的方法叫做标量替换(scalar replacement)，也叫做寄存器流水(register pipelining)。实质上，这种方法是寻找重用数组元素的机会并用对标量临时变量的引用来替代这些重用。如我们在前面两个例子中已看到的，对某些真实程序它能产生引人注目的速度改善。另一个好处是它可以减少对D-cache优化的需要。

用不含条件的循环嵌套来解释标量替换是最简单的。为了描述它，我们需要定义依赖图中循环携带的依赖边e的周期(period)，记为$p(e)$，它是这条边的尾和头所表示的带下标变量引用之间的为常数的循环迭代次数；如果这个迭代次数不是常数，这个变量就不能作为标量替换的候选，并且其周期是无定义的。

下面，我们建立这个循环嵌套的部分依赖图，它只包括那种有确定周期的流依赖和输入依赖，并且或者是循环无关的，或者是最内层循环携带的依赖；我们排斥那些表示传递依赖的边。

bbs.theithome.com

683 另一种方法是，我们可以从完整的依赖图开始（如果有的话），然后对它进行修剪。在结果得到的依赖图中，每一个流依赖或输入依赖代表了一个标量替换的机会。

如我们的例子表明的，在迭代可以开始重复之前，我们一般需要$p(e)+1$个临时变量来容纳由$p(e)+1$个迭代生成的值。因此引入临时变量t_0到$t_{p(e)}$，并用这些临时变量替代数组元素引用。具体地，用$t_0 \leftarrow$A[i]替代从主存取数组元素A[i]的引用，而其他对A[$i+j*s$]的引用则根据它是使用还是定值，分别用$tj \leftarrow$A[$i+j*s$]或A[$i+j*s$]$\leftarrow tj$替代，其中s是连续访问A[]的跨步。同时还需在最内层循环体的末尾放置一串赋值$t_{p(e)} \leftarrow t_{p(e)-1}, \cdots, t_1 \leftarrow t_0$。最后，我们必须正确地初始$t_i$，具体做法是：从循环体中剥离开始的剥离$p(e)$个迭代[⊖]，并用赋值$t_o \leftarrow elem$替代迭代$i$中对该数组元素的第一个取操作，用对应的赋值替代其他数组元素的取和存。现在每一个t_i都是可寄存器分配的候选对象，并且进一步，如果这个依赖是循环无关的，它还是可以提到循环之外的代码候选。

我们第二个标量替换的例子（见图20-11）就是根据上面描述的方法来处理的。

循环交换和循环合并（参见20.4.2节）等转换可以使得标量替换对于给定的循环嵌套起作用，或者使它们更有效果。循环交换可以使得循环携带的依赖是最内层循环携带的依赖，从而可以增加标量替换的机会。关于它的例子参见图20-12。

```
for i ← 1 to n do
   for j ← 1 to n do
      a[i,j] ← b[i] + 0.5
      a[i+1,j] ← b[i] - 0.5
   endfor
endfor
```

a)

```
for j ← 1 to n do
   for i ← 1 to n do
      a[i,j] ← b[i] + 0.5
      a[i+1,j] ← b[i] - 0.5
   endfor
endfor
```

b)

图20-12 a) 两层嵌套的HIR循环，b) 交换这两个循环的结果

循环合并可以将对一个（或多个）数组元素的多个使用集中到一个循环内，从而为标量替换创造机会。图20-13给出了一个例子。在合并了这两个循环之后，对a[i]可施加标量替换。

```
for i ← 1 to n do
   b[i] ← a[i] + 1.0
endfor
for j ← 1 to n do
   c[j] ← a[j]/2.0
endfor
```

a)

```
for i ← 1 to n do
   b[i] ← a[i] + 1.0
   c[i] ← a[i]/2.0
endfor
```

b)

图20-13 a) 两个HIR循环，b) 合并这两个循环后的结果

标量替换可以处理如图20-14中C代码所示的那种循环体内有if的循环。对于这个例子，我们使用三个临时变量，分别用t2替代a[i-2]，用t1替代a[i-1]，用t0替代a[i]。

此外，对嵌套的循环施加标量替换也可以得到显著的好处。例如，给定图20-15a中的循环，我们首先对最内层循环中的x[i]执行标量替换，得到图20-15b所示的代码。接下来，以（随意选择的）展开因子3展开这个循环，得到图20-16所示的代码；然后用标量替换y[j]的值，结果得到了图20-17所示的代码。

⊖ 从一个循环剥离（peeling）开始的k个迭代意味着用循环体的k个副本加上增量和循环索引变量的测试代码替代循环的前k个迭代，并将它们直接放置在该循环之前。

bbs.theithome.com

```
for (i = 2; i <= 20; i++)
{  b[i] = (a[i] + b[i])/2.0;

   if ((i % 2) == 0)
      a[i] = a[i-2] + 1.0;


   else

      a[i] = a[i-1] - 1.0;



}
```

a)

```
t2 = a[0];
t1 = a[1];
t0 = a[2];
b[2] = (t0 + b[2])/2.0;
t0 = t2 + 1.0;
a[2] = t0;
t2 = t1;
t1 = t0;
t0 = a[3];
b[3] = (t0 + b[3])/2.0;
t0 = t1 - 1.0;
a[3] = t0;
t2 = t1;
t1 = t0;
for (i = 4; i <= 20; i++)
{   t0 = a[i];
    b[i] = (t0 + b[i])/2.0;
    if ((i % 2) == 0)
    { t0 = t2 + 1.0;
      a[i] = t0;
    }
    else
    {
      t0 = t1 - 1.0;
      a[i] = t0;
    }
    t2 = t1;
    t1 = t0;
}
```

b)

图20-14 a) 其循环体内含控制流结构的一个C循环，b) 对a[]施加标量替换的结果

```
for i ← 1 to n do
   x[i] ← 0
   for j ← 1 to n do
      x[i] ← x[i] + a[i,j] * y[j]
   endfor
endfor
```

a)

```
for i ← 1 to n do
   t0 ← 0
   for j ← 1 to n do
      t0 ← t0 + a[i,j] * y[j]
   endfor
   x[i] ← t0
endfor
```

b)

图20-15 a) 一个两层嵌套的HIR循环，b) 对内层循环中的x[i]施加标量替换后的循环

bbs.theithome.com

```
for i ← 1 by 3 to n do
   t0 ← 0
   t1 ← 0
   t2 ← 0
   for j ← 1 to n do
      t0 ← t0 + a[i,j] * y[j]
      t1 ← t1 + a[i+1,j] * y[j]
      t2 ← t2 + a[i+2,j] * y[j]
   endfor
   x[i] ← t0
   x[i+1] ← t1
   x[i+2] ← t2
endfor
for i ← i to n do
   t0 ← 0
   for j ← 1 to n do
      t0 ← t0 + a[i,j] * y[j]
   endfor
   x[i] ← t0
endfor
```

图20-16 用展开因子3展开图20-15b中的循环后得到的循环

```
for i ← 1 by 3 to n do
   t0 ← 0
   t1 ← 0
   t2 ← 0
   for j ← 1 to n do
      t4 ← y[j]
      t0 ← t0 + a[i,j] * t4
      t1 ← t1 + a[i+1,j] * t4
      t2 ← t2 + a[i+2,j] * t4
   endfor
   x[i] ← t0
   x[i+1] ← t1
   x[i+2] ← t2
endfor
for i ← i to n do
   t0 ← 0
   for j ← 1 to n do
      t0 ← t0 + a[i, j] * y[j]
   endfor
   x[i] ← t0
endfor
```

图20-17 用标量替换图20-16中的内层循环的y[j]的结果

684
~
686

20.4 数据高速缓存优化

本章余下的内容几乎全都集中在与数值（或科学计算）代码的高速缓存使用优化相关的介绍和综述方面。所谓的数值代码，指的是对大量数组数据进行处理的程序，这种程序一般是（但不总是）用Fortran编写的浮点数值程序，其中大多数具有规则的数据使用模式，并且有在数据从高速缓存中被扫出之前重复使用这些数据的机会。

我们先从简要讨论数据的全局安置开始。假设我们有一个可用于分析和转换的完整程序，因此编译器可以收集到该程序每个部分的信息，并在加载映像中用这些信息来安排程序中用到的所有数组，使得它们在数据高速缓存中相互之间的冲突最小。接下来我们概述单个过程的数据高速缓存优化，这种优化被设计得实际可行，它清除从主存储器取数据到数据高速缓存以及从寄存器存储结果到数据高速缓存的延迟。

如前面指出的，迄今应用这类优化能获得最大成功的代码是所谓的数值或科学计算代码，这种代码的大部分执行时间都花在处理数值矩阵的嵌套循环上。这种优化方法首先指明数据在循环嵌套中的重用模式，然后转换它们，使得这些重用变成展示了引用局部性的模式，即，相同数据位置或高速缓存块的使用在时间上足够接近，从而使得它们的执行不会导致在重用这些数据之前将数据从高速缓存扫出。

确定重用模式的主要技术是依赖关系分析，使得要使用的数据及时地靠拢到一起的方法是循环嵌套转换。在第9章我们曾给出了嵌套循环中数组引用的依赖关系分析概述，20.4.2节将介绍有关的循环转换。我们然后将概要性地给出由Lam、Rothberg和Wolf开发的一种方法（文献参见20.7节），这种方法被设计成以一种比最优稍差一点的方式，尽可能地消除由于访问高速缓存导致的延迟。

接下来，我们介绍数据预取，这是一种隐藏（而不是清除）部分取数据延迟的方法，这种技术应当在上面讨论的循环转换之后实施。

bbs.theithome.com

超星公司制作 请尊重作者版权

最后，我们简要讨论标量优化和面向存储的优化之间的相互作用，对指针和动态分配的数据对象实施数据高速缓存优化的可能性，以及在编译处理的什么位置集成I-cache和D-cache优化。

在讨论数据高速缓存优化之前，我们应注意数据高速缓存十分频繁地应用于含有浮点计算的循环，并且如12.3.2节讨论的，正是这种区域我们必须特别小心，不要由于重新安排它们而导致计算结果的改变。在12.3.2节我们曾指出，如果MF表示给定精度的最大有限浮点值，则

$$1.0 + (MF - MF) = 1.0$$

而

$$(1.0 + MF) - MF = 0.0$$

这也能如下所示地推广到循环。令存储在数组A[1..3*n]中的值是

```
A[3*i+1] = 1.0
A[3*i+2] = MF
A[3*i+3] = -MF
```

其中i=0直至i=n-1。图20-18a和b）中的HIR代码都将0.0赋给s，而c）中的循环赋给s的值是1.0，d）中的循环赋给s的值是n。

```
s ← 0.0
for i ← 1 to 3*n do


   s ← s + A[i]


endfor
```

a)

```
s ← 0.0
for i ← 3 to 3*n do
    t ← 0.0
    for j ← i-2 to i do
      t ← t + A[j]
    endfor
    s ← s + t
endfor
```

b)

```
s ← 0.0
for i ← 3*n by -1 to 1 do


   s ← s + A[i]


endfor
```

c)

```
s ← 0.0
for i ← 3*n by -3 to 3 do
    t ← 0.0
    for j ← i by -1 to i-2 do
      t ← t + A[j]
    endfor
    s ← s + t
endfor
```

d)

图20-18 对同一个浮点值序列求和但产生不同结果的HIR循环。在a) 和b) 中的循环产生结果0.0，而c) 中的循环产生1.0，d) 产生n

Fortran 77标准11.10节限制循环的计算顺序必须“与书写的顺序相同”，它不允许上面讨论的转换。

但是，图20-18所使用的循环转换，即循环逆转和铺砌，在应用于Fortran程序的一些优化器中已十分频繁地实现了。因此，告诉用户它所做的转换是编译器的责任，而证实这种转换不会对计算结果造成影响是用户的责任。这也强有力地表明，让用户能够控制那种作用于特定循环的转换是必要的。

687
↓
688

20.4.1 过程间的数据安排

一种改善数据高速缓存用法的过程间的方法是重新安排大数组对象在存储器中的位置，以减少它们相互之间在数据高速缓存中冲突的可能性。这种方法需要有与数据对象使用模式有关的信息，并且在重新安排时需要能同时拿到所有数据对象的信息（即，如果有的话，它们的静

bbs.theithome.com

态初值以及对它们寻址的代码)。这导致需要过程间依赖关系分析，并且需要能够操作整个加载映像，因此，实际的编译器不可能在近期采用这种方法。但这种方法将来可能会成为实际可行的，Gupta [Gupt90]的工作证明了最优数据重新安置问题是NP完全的，并给出了一种具有多项式时间的算法，在实际中这个算法能提供较好的近似值。

20.4.2 循环转换

我们考虑在HIR的理想嵌套循环中带下标的变量的使用，理想嵌套循环是规范的(canonical)或正规形式(normalized form)的循环[⊖]，即每一个循环的索引都从1开始，以增量1变化到某个值n，并且只有最内层循环内有非for语句的语句。它简化了循环的表示，但并不必须是这样。令

689

```
for i ← a by b to c do
   statements
endfor
```

是具有任意循环边界和增量的循环，则

```
for ii ← 1 by 1 to n do
   i ← a + (ii - 1) * b
  statements
endfor
```

是对应循环的规范形式，其中$n=\lfloor(c-a+b)/b\rfloor$。

循环转换所做的工作包括交换两个嵌套的循环，逆转循环迭代执行的顺序，将两个循环的循环体合并到一起，等等。如果选择得适当，它们提供了在执行一个循环嵌套时，既能保持包含它的程序的语义，又能改善性能的机会。性能的改善可能是因为较好地利用了存储层次，使得循环迭代可以由若干个处理机并行执行，使得循环的迭代可以向量化，或者是由于这些因素的某种综合作用。我们应用循环转换的目的主要是为了实现寄存器、数据高速缓存以及其他级别的存储层次使用方面的优化。如1.5节所讨论的，并行化和向量化将留给其他书去讨论。

我们涉及三种类型的循环转换，即，

1. 幺模转换;
2. 循环合并和循环分布;
3. 循环铺砌(loop tiling)。

Wolf和Lam([WolL91]和[Wolf92])提出了一种适合于刻画一大类循环转换的方法。他们定义了一类所谓的*幺模循环转换*(unimodular loop transformation)，这类转换的效果可以用幺模矩阵与距离向量[⊖]的乘积来表示。一个*幺模矩阵*(unimodular matrix)是一个方形矩阵，它的元素都是整数并且它的行列式的值为1或−1。如我们将看到的，循环交换、更一般的嵌套循环的循环置换(loop permutation)、循环倾斜(loop skewing)、循环逆转(loop reversal)，以及一系列的其他有用的转换都是幺模转换。

如果一个距离向量$<i_1, i_2,\cdots,i_n>$至少有一个非0元素，并且它的第一个非0元素为正，则这个距离向量是*词典序为正的*(lexicographically positive)。这个定义可以用一种自然的方式扩充到其他的顺序关系。Wolf证明了由矩阵U表示的幺模转换当作用于一个具有词典序非负的距离向量集合D的循环嵌套时是合法的，当且仅当对于每一个$\vec{d}\in D$，有$U\vec{d}\succeq\vec{0}$，即当且仅当它将

690

⊖ 虽然我们的循环原本都是规范化的，但下面描述的转换中有一些(例如循环铺砌)会产生增量不为1的循环。尽管这种循环可以转变成规范化的形式，但没有必要这样做，因为它们表示的是施加转换后的最终状态。

⊖ 回忆9.3节距离向量的定义。

bbs.theithome.com

词典序为正的距离向量仍转换为词典序为正的距离向量时[⊖]。

我们下面考虑关于么模转换、它们对应的矩阵以及它们对循环嵌套的作用的一些例子。

循环交换（loop interchange）是交换循环嵌套中两个相邻循环的嵌套顺序的转换。它是由一个单位矩阵交换了对角线上相邻的两个1对应的行和列所得到的矩阵来刻画的。例如，考虑图20-19中的代码，它的距离向量为<1, 0>。循环交换矩阵是

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

这个交换矩阵与距离向量的乘积是

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}=\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

施加这个转换的结果如图20-20所示；注意，结果的距离向量是词典序为正的，因此这个交换合法。

```
for i ← 1 to n do
   for j ← 1 to n do
      a[i,j] ← (a[i-1,j] + a[i+1,j])/2.0
   endfor
endfor
```

图20-19 在这个HIR循环嵌套中的赋值有距离向量<1,0>

```
for j ← 1 to n do
   for i ← 1 to n do
      a[i,j] ← (a[i-1,j] + a[i+1,j])/2.0
   endfor
endfor
```

图20-20 图20-19中的代码经循环交换之后

循环置换（loop permutation）是循环交换的一种推广，它允许同时交换两个以上的循环，并且不要求这些循环是相邻的。例如，么模矩阵

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

表示4个嵌套的循环中的第一个循环与第三个交换，第二个与第四个交换。 691

循环逆转（loop reversal）逆转循环迭代的执行顺序。表示这种转换的适当的矩阵是一个单位矩阵，但用−1替换了其中与被逆转的循环相对应的1。例如，矩阵

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

对应于逆转三个循环中中间那个循环的迭代方向。假如我们对图20-19中的代码的外层循环施行循环逆转，则对应的矩阵向量是：

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}=\begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

结果得到的距离向量< −1, 0>是词典序为负的，因此这个转换不合法。

循环倾斜（loop skewing）改变循环迭代空间的形状。例如，它可以将图20-21a中的循环转换为图b）所示的循环，这对应于将迭代空间从图20-22a所示的遍历改为图b）所示的遍历。这个转换对应的矩阵是

⊖ 注意，么模矩阵必定将0向量（并且只有它）转换为0向量。

bbs.theithome.com

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

它是一个么模矩阵。

```
for i ← 1 to n do
   for j ← 1 to n do
      a[i] ← a[i+j] + 1.0
   endfor
endfor
```

a)

```
for i ← 1 to n do
   for j ← i+1 to i+n do
      a[i] ← a[j] + 1.0
   endfor
endfor
```

b)

图20-21 a) 一个HIR循环嵌套，b) 内层循环倾斜后的结果

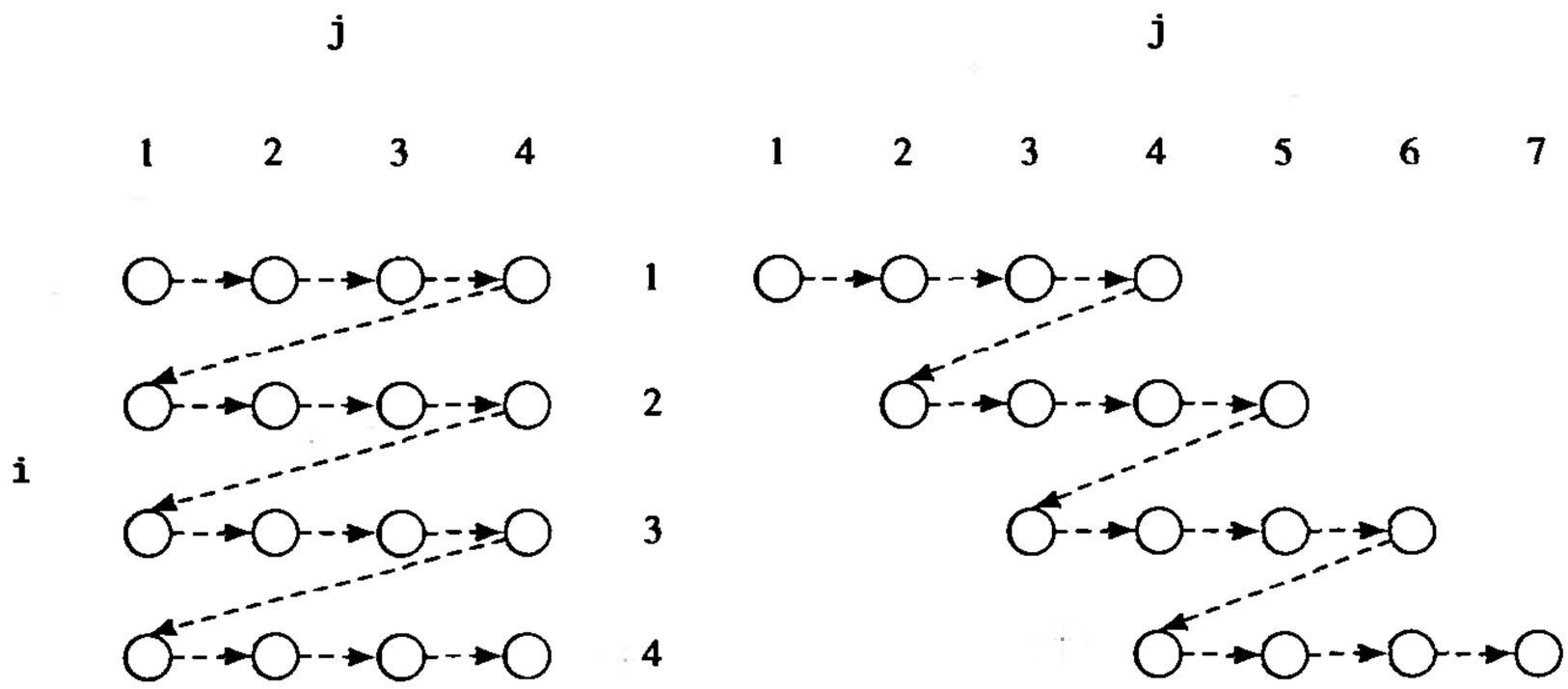


图20-22 a) 图20-21a所示的循环嵌套的迭代空间遍历，b) 倾斜后的迭代空间遍历

循环合并、循环分布和循环铺砌是三种不是么模转换的重要转换。

循环合并（loop fusion）如图20-23所示，将两个迭代空间相同的循环的循环体合并为一个循环。只要这两个循环具有相同的循环边界，并且只要在已合并的循环中没有因第一个循环中的指令依赖于第二个循环中的指令引起的流依赖、反依赖和输出依赖（外层循环不影响转换的合法性），这两个循环的循环合并就是合法的。作为例子，对图20-24a的两个循环应用循环合并，得到图c）中的一个循环。其中惟一的依赖关系是S2<1>S1，因此，这个转换是合法的。另一方面，合并图b）中的两个循环得到图d），其中惟一的依赖关系是S1<1>S2，因此这个合并是不合法的。

692 ~ 693

```
for i ← 1 to n do
   a[i] ← a[i] + 1.0
endfor
for j ← 1 to n do
   b[j] ← a[j] * 0.618
endfor
```

a)

```
for i ← 1 to n do
   a[i] ← a[i] + 1.0
   b[i] ← a[i] * 0.618
endfor
```

b)

图20-23 a) 一对HIR循环，b) 循环合并后的结果

循环分布（loop distribution）与循环合并正好相反。它将含有多个语句的循环分割为两个具有相同迭代空间的循环，使得第一个循环包含原循环中的某些语句，第二个循环包含另一些语句。图20-23也可以作为循环分布的一个例子，b）是循环分布之前的情况，a）是分布之后

bbs.theithome.com

的情况。如果一个循环分布不破坏原循环依赖图的任何环路，这个循环分布就是合法的（同样，外层循环不影响这种转换）。

```
        for i ← 1 to n do
S1          a[i] ← b[i]
        endfor
        for i ← 1 to n do
S2          b[i-1] ← c[i] + 1.5
        endfor
```

a)

```
        for i ← 1 to n do
            a[i] ← b[i]
        endfor
        for i ← 1 to n do
            b[i+1] ← c[i] + 1.5
        endfor
```

b)

```
        for i ← 1 to n do
S1          a[i] ← b[i]
S2          b[i-1] ← c[i] + 1.5
        endfor
```

c)

```
        for i ← 1 to n do
            a[i] ← b[i]
            b[i+1] ← c[i] + 1.5
        endfor
```

d)

图20-24 两对HIR循环a) 和b)，对它们进行循环合并后的结果分别如c) 和d) 所示。在合并后的循环c) 中，存在一个依赖关系S2<1>S1。而在合并后的循环d) 中，依赖关系是S1<1>S2，所以循环合并对于a）中的例子是合法的，而对于b) 是不合法的

铺砌（tiling）是一种增加循环嵌套深度的循环转换。给定一个深度为n的循环嵌套，铺砌可使得它具有从$(n+1)$到$2n$的任何一种深度，具体取决于有多少个循环被铺砌。铺砌一个循环就是用两个嵌套循环来替代它，内层循环（叫做瓦片循环）的增量等于原循环的增量[⊖]，外层循环的增量等于$ub-lb+1$，其中ub和lb分别是内层循环的上下界；铺砌从被铺砌的这个循环开始向内交换循环，使得瓦片循环是嵌套中的最内层循环。瓦片循环的迭代次数叫做*瓦片大小*（tile size）。例如，在图20-25中，图a）中的循环在图b）中已用大小为2的瓦片铺砌。

```
for i ← 1 to n do
    b[i] ← a[i]/b[i]
    a[i+1] ← a[i] + 1.0
endfor
```

a)

```
for i ← 1 by 2 to n do
    for j ← i to min(i+1,n) do
        b[i] ← a[i]/b[i]
        a[i+1] ← a[i] + 1.0
    endfor
endfor
```

b)

图20-25 a) 一个HIR循环，b) 用大小为2的瓦片进行循环铺砌后的结果

铺砌一个循环嵌套中的一个或多个循环，效果上相当于重新安排迭代空间的遍历，使得它由一系列的小多面体一个接一个地执行所组成[⊜]。图20-26展示了用大小为2的瓦片铺砌图20-21a代码中的两个循环得到的结果。图20-27展示了这个结果的迭代空间遍历，其中假定$n=6$。

当瓦片循环可以与其他循环交换使得瓦片循环是最内层循环时，铺砌一个循环嵌套就是合法的。特别地，铺砌一个内层循环总是合法的。

⊖ 但是，如图20-25b所示，有一个min运算符保证铺砌后的版本最后一次通过外层循环时执行适当的迭代次数。

⊜ 在其他文献中铺砌也有另外的三个名字。它曾叫做“分块”（blocking），但我们不使用那个名字，因为那个名字在计算机科学中至少已有了另外两种含义。Wolf称它为“成块剥离和交换”（strip mine and interchange），因为创建一个瓦片循环导致划分一个给定的循环为一系列的循环，这些循环执行从原循环“剥离”下来的循环。Callahan、Carr和Kennedy称铺砌一个最内层循环为“展开并塞入”（unroll and jam），因为它可以看成是将循环体展开若干次，然后将它们一起“塞入”到一个循环中。

bbs.theithome.com

```
for i ← 1 by 2 to n do
   for j ← 1 by 2 to n do
      for i1 ← i to min(i+1,n) do
         for j1 ← j to min(j+1,n) do
            a[i1] ← a[i1+j1] + 1.0
         endfor
      endfor
   endfor
endfor
```

图20-26 用大小为2的瓦片铺砌图20-21a所示的HIR循环嵌套中两个循环的结果

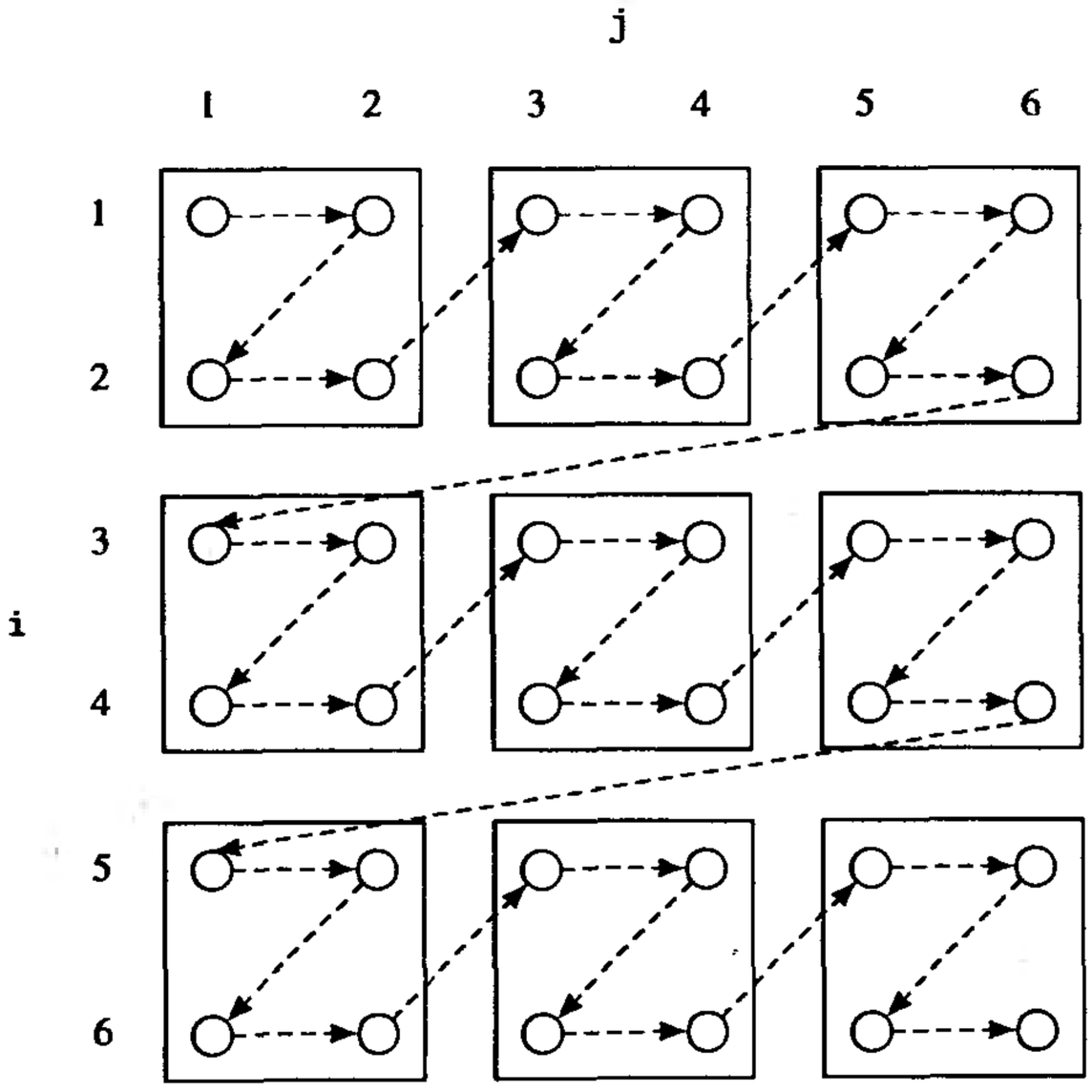


图20-27 图20-26中已铺砌的循环嵌套的迭代空间遍历，假定$n = 6$

20.4.3 局部性与循环铺砌

694 ~ 695

对于循环嵌套而言，最重要的使数据高速缓存利用效率最大化的技术是：利用大于单个数组元素或高速缓存块的数组区域中引用的局部性。尽管在一个给定的循环嵌套中可能不存在可以利用的重用，但通过使其工作集中在这些数组的一些子片段中，它的高速缓存使用模式也仍然可能得到改善。

考虑图20-1a中矩阵乘的例子，前面我们已经发现它的任何循环顺序都是合法的。我们可以将它铺砌，使它对所希望的任意大小的子数组进行操作。如果选择最优的瓦片大小，则除强制的高速缓存缺失之外，每一个工作片段可以在没有任何高速缓存缺失的情况下完成；进一步，如果选择循环结构中相互之间最适当的铺砌安排，通过利用连续的瓦片之间的重用，我们甚至可能减少一个瓦片的强制高速缓存缺失的次数。为每一个循环创建一个瓦片循环得到图20-28a中的代码。现在，因为原来的循环可以任意重排顺序，故我们可以将所有瓦片循环移到最内层，如图20-28b所示。结果得到的这个循环嵌套对数组的T × T个子片段工作，它具有的遍历模式类似于图20-27所示的模式；当然，对C(ii,jj)另外还可以应用数组元素的标量替换。如果明智地选择了T，铺砌可以使得每一个数组为执行给定的计算所需要同时出现在数据高速缓存中的元素较少，从而减少了高速缓存冲突。

bbs.theithome.com

```
do i = 1,N,T
  do ii = i,min(i+T-1,N)
    do j = 1,N,T
      do jj = j,min(j+T-1,N)
        do k = 1,N,T
          do kk = k,min(k+T-1,N)
            C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

a)

```
do i = 1,N,T
  do j = 1,N,T
    do k = 1,N,T
      do ii = i,min(i+T-1,N)
        do jj = j,min(j+T-1,N)
          do kk = k,min(k+T-1,N)
            C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

b)

图20-28 a) 对图20-1a的Fortran矩阵乘每一维索引分片之后，b) 铺砌所有循环后的结果

尽管铺砌在减少高速缓存冲突并因此增加性能方面常常很有价值，但留给了我们三个问题:

1. 当循环嵌套不能完全铺砌时会发生什么情况?
2. 高速缓存的结构对铺砌的效率有什么样的影响?
3. 我们怎样选择对给定循环嵌套和循环边界最有效果的瓦片大小?

循环嵌套是否可完全铺砌，对它的性能可能有也可能没有影响——这取决于铺砌是否有必要。考虑图20-29中的HIR循环嵌套。显然，内层循环不能从铺砌得到好处，但当数组a[]的各行在高速缓存中可能相互发生冲突时，铺砌外层循环将得到好处。置换图20-29中的这两个循环将产生一个只铺砌最内层循环而获益的例子。另一方面，我们可能有一个循环嵌套，它的迭代空间有几个方向很大，但它的有些循环却不能置换，因而不能实行铺砌。但是我们至少可以铺砌那些可以置换到最内层的循环，并且这样做一般可以获得性能的改善。

```
for i ← 1 to N do
   for j ← 1 to 3 do
      a[i,j] ← (a[i-1,j-1] + a[i,j] + a[N+1-i,j+1])/j
   endfor
endfor
```

图20-29 两个循环中至多一个循环可能从铺砌获益的HIR例子

数据高速缓存的组织结构对铺砌的整体效果，以及对给定大小循环的具体瓦片大小的效果可造成戏剧性的差别。假设我们有一个直接映射的高速缓存和某个具体的循环嵌套。尽管这个高速缓存可能大得足够容纳大部分要处理的数据，但它不能够无冲突地在两个不同的数组内容

版权所有 侵权必究

bbs.theithome.com

之间，或者甚至在同一个数组的两个部分之间做到这一点。即使我们完全铺砌了这个循环嵌套，我们也可能会为了避免冲突而需要有相对高速缓存的大小而言小得令人奇怪的瓦片的大小。组相连的高速缓存明显地减少了这种冲突的频率，但我们在单个循环嵌套中处理的数组越多，它的铺砌效率就越低。

可证明选择一个与循环边界无关的固定大小的瓦片是灾难性的，尤其是对于直接映射的高速缓存。循环边界和瓦片大小的特定组合能非常有效地减少冲突缺失，而只要稍微地改变这个循环的边界哪怕只有约1%到2%，并且保持瓦片的大小不变，就可能显著地增加冲突，并使性能有较大比例的降低。因此，关键是应当允许瓦片的大小随循环的边界而变化，并且在进行选择时应当以它们之间互相作用的经验作为基础，或者有解释它们互相作用的理论，或者更理想的是同时具备两者。

696 ~ 697

Wolf（[WolL91]和[Wolf92]）从理论和实践两方面就选择瓦片的大小给出了一个很好的论述，他也给出了前面所讨论情况的一些例子。

20.4.4 利用硬件辅助：数据预取

有些较新的64位RISC——如Alpha、PowerPC和SPARC-V9——含有数据预取指令，这种指令给数据高速缓存一个提示，指出程序不久之后将需要所指定地址中包含的数据块。例如，在SPARC-V9中，数据预取指令能够指明预取一块数据用于若干次的读或写。这种指令有如下4种类型的含义：

1. 若干次读：预取数据到离处理机最近的D-cache。

2. 一次读：预取数据到一个预取缓冲区并且不打扰高速缓存，因为这个数据只使用一次；如果这个数据已在D-cache，则让它仍保留在那里。

3. 若干次写：预取数据到离处理机最近的D-cache，以准备写它的部分内容。

4. 一次写：为写而预取此数据，但如果可能的话，以一种不打扰D-cache的方式。

Alpha的(`fetch`和`fetch_m`)以及PowerPC的（`dcbt`和`dcbtst`）数据预取指令提示应当将包含给定字节地址的数据块取到D-cache中分别用于读或写。对于Alpha，如果要预取的话，所取的块大小至少是512字节，如前一节所讨论的，它对于某些应用可能太大了。

注意，预取并不能减少从主存储器取数据到高速缓存的延迟——它只是通过使它与计算重叠而隐藏这种延迟。因此，在一定程度上，减少延迟和隐藏延迟这两种操作是互补的，但我们仍然应当首先利用数据重用来减少延迟，然后才使用预取。

预取对于那种连续访问很大数组的循环是最有用的。对于由单个基本块组成的循环，生成预取指令可以通过检测顺序访问模式来驱动，并且要预取的地址可以是相对一个归纳变量的常数偏移。为了确定预取指令的流出时机，我们按如下方式来处理。我们定义下面4个量：

1. T_{loop}是循环的一个迭代不做预取所花的拍数，假定所需要的数据在高速缓存中。

2. t_{use}是从一个迭代的开始到循环中第一次使用该数据所间隔的拍数。

3. t_{pref}是数据预取指令的流出延迟。

698 4. T_{pref}是数据预取指令的结果延迟，即，预取指令流出之后$t_{pref} + T_{pref}$拍，数据在高速缓存中可用。

则第i块的第一次使用在没有预取的情况下出现在第$t_{use} + i * T_{pref}$拍；在有预取时出现在第$t_{use} + i * (T_{loop} + t_{pref})$拍，其中$i = 0, 1, 2, \ldots$。为了使得数据在第$t_{use} + i * (T_{loop} + t_{pref})$拍时可用，它们必须提前$t_{pref} + T_{pref}$拍被预取，或提前

bbs.theithome.com

$$\left\lfloor \frac{T_{pref}}{T_{loop} + t_{pref}} \right\rfloor$$

个迭代又$T_{pref} \bmod (T_{loop} + t_{pref})$拍。

例如，假设一个循环完成一个迭代需20拍，它在循环中第一次使用高速缓存中这个数据的出现时机是第5拍。设预取指令有1拍的流出延迟和25拍的结果延迟。则对给定迭代的预取指令应当放在比此迭代早$\lfloor 25/(20+1) \rfloor = 1$个迭代，且早于第一个使用点之前$25 (\bmod\ 20 + 1) = 4$拍的地方，或者在早于预取数据被使用的这个迭代之前一个迭代又1拍的地方，并且预取指令应当指明下一个迭代所使用的数据的地址。如果该数据的第一次使用出现在每一个迭代的第2拍，则预取应当提前一个迭代又4拍，即预取指令应放置在数据被使用的这个迭代的前两个迭代的第19拍之处。

注意，与本章讨论的其他面向D-cache的优化不同，这种优化要求我们知道执行一个循环迭代需要的时钟周期数，因此它需要在低级代码上来执行。另一方面，这种优化也需要知道每一个预取的地址是什么——这种地址信息最好是由基地址加一个由高级代码得出的下标值而确定的，然后再传递给低级代码。

可能出现的一种复杂情况是所预取的数据块太大，以至于预取不是对每一个迭代都有用。循环展开对这种情形会有所帮助——展开n次使我们可以每n个迭代流出一条预取——但即使这样也可能仍然不够；在这种情况下预取需要受一个条件的控制，这个条件检查归纳变量与高速缓存块大小求模的结果值。

另一种编译情形是可能有多个数组能从预取获益，但如果不仔细地安排这些数组和预取，它们则有可能导致高速缓存冲突。通过连接器和编译器协同工作，适当地放置大数组于高速缓存块的边界，可以改善连续预取的效率。

Mowry、Lam和Gupta（[MowL92]和[Mowr94]）描述了一种比上面给出的要稍微复杂点的数据预取方法。他们评估了一种与上面的方法相似的方法，指出如果所有数据都预取的话，大约有60%的预取是白费的，因为预取占用了CPU、D-cache，以及D-cache与主存储器之间的总线，因此值得使所需要的预取尽量少。 699

他们的方法集中在确定对高速缓存行的第一次引用（即那种可能导致它被读到高速缓存的引用）和预取谓词上，预取谓词确定对于特定的循环迭代i，一个数据是否需要预取。这个谓词被用来转换循环——如果谓词是$i = 0$，则剥离循环的第一个迭代，如果谓词是$i \equiv 0 \pmod{n}$，则以因子n展开循环。接下来他们放置预取指令的方法基本上如我们前面的描述类似，但只对分析已表明预取是需要的那些迭代。如果这种循环转换导致代码变得太大以至于影响了I-cache的性能，他们的方法则抑制这种转换或者插入使用预取谓词的代码来控制预取。

20.5 标量优化与面向存储器的优化

Whitfield和Soffa [WhiS90]以及Wolfe [Wolf90]研究了传统的标量优化与较新的并行化或面向D-cache的优化之间的相互影响。他们证明了有些标量优化会抑制并行化；指出优化的顺序非常重要，并且为了实现最大性能，优化的顺序需要随程序的不同而变化；而且在有些情况下，执行某种标量优化的逆优化是使得某些D-cache优化可行的关键。

例如，Whitfield和Soffa展示了循环不变代码外提会使得循环交换和循环合并不能实施，并且Wolfe还说明了公共子表达式删除可能抑制循环分布。显然，从这些结果可以得出，在某些实例中逆转这种有抑制作用的转换，可以使得原本不能进行的其他优化能够得以施行。

bbs.theithome.com

20.6 小结

本章讨论了利用存储层次结构，尤其是数据和指令高速缓存及CPU寄存器的优化技术。

即使在最早的计算机设计中，主存储器与寄存器就有区别。主存储器比寄存器要大且也较慢，许多系统都要求除了取和存之外的所有操作，至少要有一个操作数是在寄存器中（在某些情况下要求所有操作数）；RISC系统当然属于后一类。

处理机的时钟周期和访问存储器所需要的时间之间的差距已经增加到了使得现在多数系统都有位于主存储器和寄存器之间的高速缓存。对一个定向在高速缓存的地址的存储访问，可以从高速缓存中得到满足而不需要通过主存储器（或在存数的情况下，有可能并行存储到主存储器中）。高速缓存的效率取决于程序的空间和时间的局部性性质。在可以依赖高速缓存的情况下，程序的运行速度由CPU和高速缓存决定。如果不能依赖它们，取数、存数以及指令的读取
700 都以存储器的速度执行。

为了处理这种速度的不一致，我们首先讨论了与指令高速缓存有关的技术；然后考虑了如何改善数组元素的寄存器分配的技术，以及与数据高速缓存有关的技术。

与指令高速缓存有关的技术包括指令预取、过程排序、过程和基本块的放置、过程分裂，以及过程内和过程间相结合的方法。预取使我们能够提示存储器，某一块特定的代码不久可能会执行，并指出应当在有空闲的总线周期时将它取到高速缓存中。过程排序按照能够增加过程引用的局部性和减少过程之间相互冲突的方式在装载模块中放置过程体。过程内代码安置或基本块排序尝试按这样一种顺序来确定过程中的基本块，这种顺序使得分支尽可能地取下降路径，其次使得分支尽可能是向前分支，从而使得分支发生转移的情形尽可能地少。事实上，循环展开可以看作是基本块排序的变种，因为如果我们以因子n展开一个循环——为简单起见，设这个循环是单个基本块的循环——则原来的n个向后分支现在变成只有一个向后分支。过程分裂通过将过程分为频繁执行的和非频繁执行的代码段，并在装载模块中将每一种类型分别集中到一起，寻求进一步改善局部性。最后，简要提及了过程内和过程间相结合的技术，以求得到尽可能大的好处。

与I-cache有关的优化中最重要的通常是过程排序与过程和基本块放置，其次是指令预取，最后是过程分裂。

接下来，我们集中讨论了改善数组元素寄存器分配的标量替换，与数据高速缓存有关的技术以及数据预取。数组元素的标量替换利用对数组元素的连续多次访问，使得用标量替换了的数组元素可以成为寄存器分配的候选，并可适应于其他典型作用于标量的所有优化。与数据有关的优化中最重要的通常是与数据高速缓存有关的优化，尤其是循环铺砌，其次是数组元素的标量替换，最后是数据预取。

我们也对数据高速缓存优化给出了介绍和概述。这是一个在已使用的各种方法中至今还没有一个明显优胜者的领域，尽管存在一些处理各种类型问题的有效方法。其结果是，我们避开对任何一种方法的详细描述，替代地只是给出对这个问题的基本理解，讲述它与第9章的数据依赖内容有怎样的关系，并给出一些有效的关于数据高速缓存优化候选方法的建议。

最后，我们简要地讨论了标量和面向存储器的优化之间的相互影响。

大多数关于存储层次的数据优化最好在包含循环表示和下标的高级或中级代码上来做。因此，它们被放置在图20-30优化顺序图解的A框中。相反，数据预取需要插入到低级代码中，但
701 它使用从高级代码分析中收集的信息，因此，我们将它放在D框内。

bbs.theithome.com

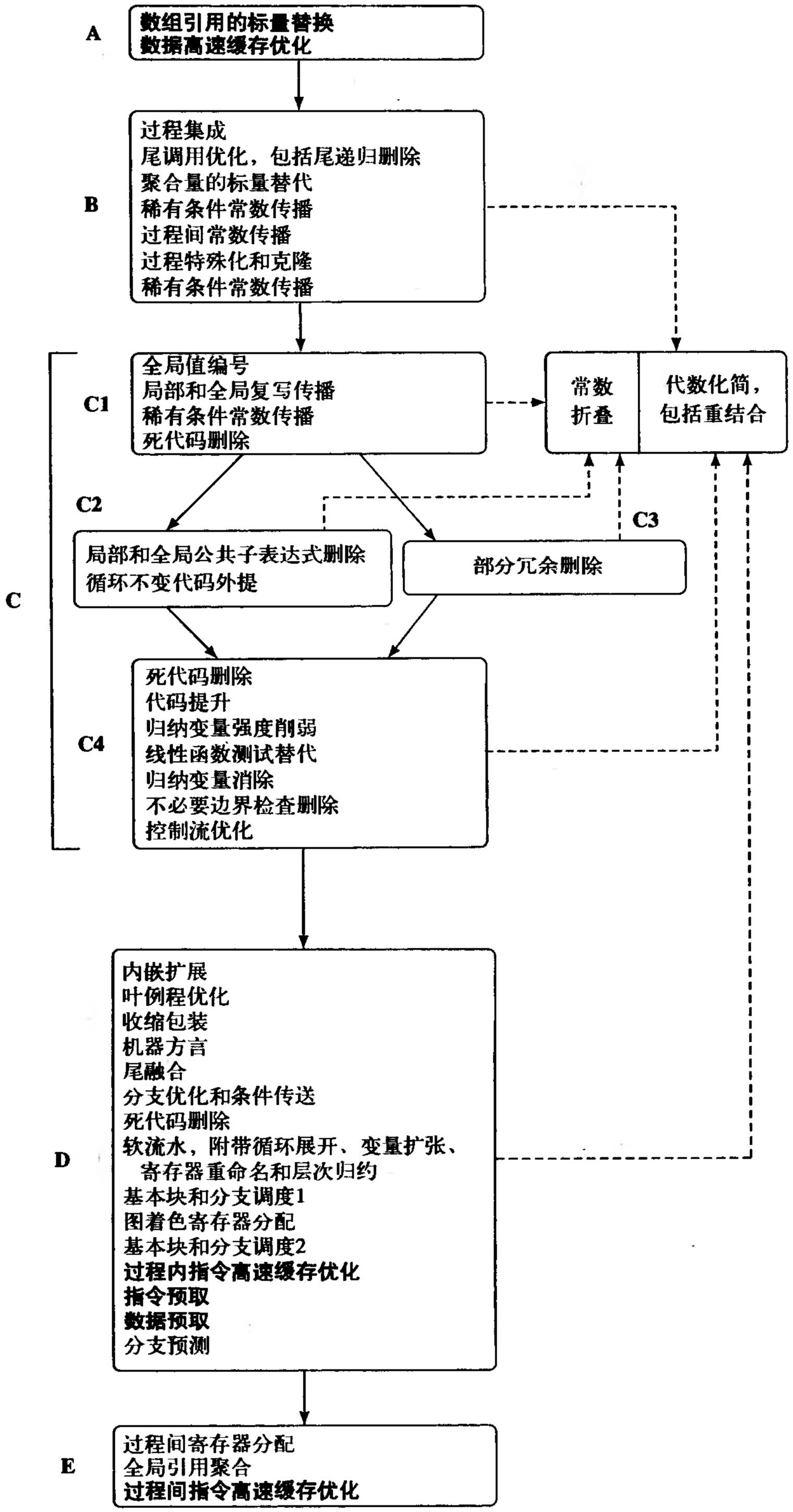


图20-30 优化顺序，黑体字标出了存储层次有关的优化

bbs.theithome.com

另一方面，指令高速缓存优化和指令预取则从编译处理非常靠后的阶段，当已能够很好地理解代码的最终形式时得到好处。因此，我们将过程内I-cache优化和指令预取放在D框，过程间的优化放在E框。

这个领域，尤其是面向数据的优化，在下一个十年中将受到密切的关注。处理这个问题的方法肯定会有较大的改善，但是很难精确地预测这种改善会来自哪一方面。显然，数学分析、数据执行模型，以及强化的实验都将扮演一个角色，但是还不清楚究竟是其中之一还是全部，抑或是另外的某种或某些见解会成为这种改善的关键原因。

20.7 进一步阅读

Bell在[Bell90]中报告了他手工改写数值密集型程序以便利用IBM RS/6000的流水线和高速缓存的工作。

本章详细讨论的I-cache优化方法基于Pettis和Hansen [PetH90] 的工作。 [McFa89]、[McFa91a]和[McFa91b] 中介绍了McFarling的工作。

标量替换，或称寄存器流水，是由Callahan、Carr和Kennedy [CalC90]开发的。

Wolfe [Wolf89a]称循环铺砌为“成块剥离和交换”； Callahan、Carr和Kennedy [CalC90]称它为“展开并塞入”。

702 ~ 703

Lam、Rothberg和Wolf介绍D-cache优化方法的论文见[LamR91]、[Wolf92]和[WolL91]。

Gupta [Gupt90]讨论了安排数据以便更好地利用D-cache的一些全局方法。

用向量空间对应的么模转换来刻画一大类循环转换的方法是在[WolL90]和[Wolf92]中找到的。

[MowL92] 中讨论了Mowry、Lam和Gupta的数据预取算法，[Mowr94]进一步对它进行了评估。

Whitfield和Soffa，以及Wolfe对标量优化和并行化或D-cache优化之间的相互影响的研究分别见[WhiS90]和[Wolf90]。

关于刻画递归数据结构和指针的存储器使用模式特点有关工作的参考文献参见9.8节。

20.8 练习

20.1 前面几章讨论的优化中，哪些优化很可能会增强过程排序的效果（参见20.2.2节），为什么？

20.2 前面几章讨论的优化中，哪些优化很可能会增强过程内代码安置的效果（参见20.2.4节），为什么？

20.3 写出一个执行循环内数组引用标量替换的ICAN算法，其中循环是仅含一个基本块的循环。

20.4 扩充练习20.3的算法，使它可以处理循环内的非循环控制流结构。

20.5 给出一个可施加循环分布的循环，以及一种导致这个循环不能进行循环分布的优化的例子。

20.6 给出一个可对内层循环施加数组引用标量替换的双层嵌套循环的例子。如果展开这个内层循环4次，会发生什么情况？

704

bbs.theithome.com

超星公司制作 请尊重作者版权

第21章　编译器实例分析与未来的发展趋势

本章我们讨论几种商业编译系统，并对编译器未来的发展趋势给出几点预测。这些商业编译器覆盖了若干种源语言和4种体系结构，它们在实现策略、中间代码结构、代码生成和优化方法等方面具有广泛的代表性。对每一种编译器，我们首先给出它所针对的目标机体系结构的一个简短概述。

有些系统，如IBM用于POWER和PowerPC的XL编译器，在非常低级的中间代码级别进行几乎所有的传统优化。另一些系统（如DEC用于Alpha的GEM编译器）使用中级中间代码，而其他一些系统（如Sun的SPARC编译器和Intel 386系列的编译器）分别在中级中间代码和低级中间代码上进行优化工作。另外，IBM的编译器甚至还包含了高级中间形式，并在其上进行存储层次结构的优化。

在后面的几节中，我们使用两个程序例子来说明各种编译器的效果。第一个例子是图21-1中的C函数。注意，在内层循环的`if`语句中，`kind`的值是常数`RECTANGLE`，因此，其中关于它的测试以及第二个分支都是死代码。`length*width`的值是循环不变量，因此对`area`的累加（`=10*length*width`）可以用一个乘法来完成。我们预期这些编译器会展开这个循环若干次，并且会进行寄存器分配和指令调度。如果所有的局部变量都分配到寄存器中，就可以不需要为这个函数分配栈空间。另外，对`process()`的调用是尾调用。

```
1   int length, width, radius;
2   enum figure {RECTANGLE, CIRCLE};
3   main( )
4   {  int area = 0, volume = 0, height;
5      enum figure kind = RECTANGLE;
6      for (height = 0; height < 10; height++)
7      {   if (kind == RECTANGLE)
8          {   area += length * width;
9              volume += length * width * height;
10         }
11         else if (kind == CIRCLE)
12         {  area += 3.14 * radius * radius;
13            volume += 3.14 * radius * radius * height;
14         }
15     }
16     process(area,volume);
17  }
```

图21-1　本章用作例子的一个C函数

第二个例子是图21-2中给出的Fortran 77程序。其中主程序简单地给数组`a()`的元素赋初值，然后便调用`s1()`。`s1()`中的第二个和第三个数组引用所引用的是同一个元素，因此其地址计算应当隶属于公共子表达式删除。`s1()`中的最内层循环应当减少到至多含8条指令，包括取两个值、将它们相加、存储其结果、更新两者的地址、终止测试条件以及分支（终止条件应当用基于其中一个数组元素的地址的线性函数测试来修改）。对于有存储器到存储器的指令、带地址更新的取/存指令、或比较然后分支指令的体系结构而言，这个指令序列应当更短。我们预

bbs.theithome.com

期其中有些编译器能够对这个循环的最内层循环进行循环展开和软流水。

```
1               integer a(500,500), k, l
2               do 20 k = 1,500
3                   do 20 l = 1,500
4                       a(k,l) = k + l
5         20    continue
6               call s1(a,500)
7               end
8
9               subroutine s1(a,n)
10              integer a(500,500),n
11              do 100 i = 1,n
12                 do 100 j = i+1,n
13                    do 100 k = 1,n
14                       l = a(k,i)
15                       m = a(k,j)
16                       a(k,j) = l + m
17        100   continue
18              end
```

图21-2 本章用作例子的一个Fortran 77程序

706 我们也期望这些编译器中至少有一些可以将过程s1()集成到主程序中。事实上，关于如何处理这个问题可有如下4种选择:

1. 分开编译主程序和s1()（即，不做过程集成）。
2. 集成s1()到主程序，同时也编译s1()的一个独立副本。
3. 集成s1()到主程序，同时将s1()编译为一个只做返回的虚过程。
4. 集成s1()到主程序，并完全删除s1()。

后面两种选择是合理的，因为这个程序的整个调用图是明确的——主程序调用s1()，并且两者都没有调用其他子程序，因此这个程序不可能存在另外的分开编译的模块对s1的使用。如我们将看到的，Sun和Intel编译器采用了第二种选择，但这些编译器都没有采用第三和第四种选择。如果这个过程没有被内联，就应使用过程间常数传播来确定n在s1()中的值是500。

21.1 Sun用于SPARC的编译器

21.1.1 SPARC体系结构

SPARC体系结构有两种主要的版本，Version 8和Version 9。相对而言，SPARC Version 8是一种最基本的32位RISC系统。处理器由一个整数部件、一个浮点部件和一个可选的（但从未实现）协处理器组成。整数部件包含32个通用寄存器，并执行取、存、算术、逻辑、移位、分支、调用和系统控制指令。它也计算整数和浮点取和存指令中的地址（寄存器＋寄存器，或寄存器＋位移）。整数寄存器由8个全局寄存器（其中r0总是提交一个0值并忽略对它的写）和若干个由24个寄存器组成的重叠窗口组成，每一个窗口由8个in寄存器、8个local寄存器和8个out寄存器组成。寄存器窗口的溢出和填充由操作系统通过响应自陷来处理。

整数部件实现了若干特殊指令，如用于支持动态类型语言的带标志的加、减指令；控制多处理机中存储器操作顺序的同步存指令；以及独立于调用和返回指令的用于切换寄存器窗口的保护和恢复指令。整数部件具有可由整数操作和浮点比较操作可选设置的条件码，分支指令用这些条件码来决定是否发生转移。

bbs.theithome.com

超星公司制作 请尊重作者版权

在体系结构上，SPARC的设计采用流水线结构，程序员能见到流水线的某些特征。分支和调用指令含有一个跟随在其后的延迟槽，这个延迟槽的执行在分支转移发生之前[⊖]，并且当从存储器取数指令的下一条指令使用的是该取数指令所取的值时，会导致一个互锁。

707

浮点部件有32个32位的浮点数据寄存器，并且执行ANSI/IEEE浮点标准，但该体系结构也允许某些由操作系统实现的次要特征。这些寄存器可以成对地容纳双精度值，或4个寄存器一组地容纳四精度值。整数寄存器和浮点寄存器集合之间不能进行数据移动。浮点部件执行取、存、算术指令、求平方根、整数与浮点之间的转换，以及比较指令。

典型的SPARC指令有三个操作数——两个源操作数和一个结果操作数。第一个源操作数和结果操作数几乎总是寄存器，第二个源操作数可以是寄存器，也可以是小常数。在汇编语言中，操作数的顺序是第一、第二是源操作数，第三是结果操作数。有关SPARC汇编语言的更多细节请参见附录A.1。

SPARC Version 9是一个与Version 8完全向上兼容的较新的体系结构。它将整数寄存器扩充到了64位，但同时提供了关于32位和64位操作结果的条件码。带特权的程序状态字中的一位指出虚地址转换应当是32位还是64位地址。整数指令也扩充包含了64位专用的指令（例如，除了Version 8的设置一对寄存器的64位取数指令之外，还有取数到一个64位寄存器的指令），根据64位条件码或根据寄存器中的值进行转移的新的分支指令，以及有条件的寄存器传送指令。这个体系结构的特权部分也有重大的重新设计，它可以允许多达4级自陷，并且使得寄存器窗口的溢出和填充更快。

SPARC的实现经历了先是用两片门阵列，并且整数和浮点部件在不同的芯片上（实际上是与Version 7稍有不同的体系结构），到高度集成的具有超标量执行功能的单芯片设计的变化。

21.1.2 Sun SPARC编译器

Sun为SPARC提供了C、C++、Fortran 77和Pascal等编译器。所支持的C语言是完整的ANSI C，C++ 遵循AT&T关于该语言的规范，Fortran 77支持DEC和Cray兼容的特征，Pascal遵循ANSI标准并具有与Apollo兼容的扩充。这些编译器源于Berkeley 4.2 BSD UNIX软件发布版本，并在1982年之后由Sun开发。原来的后端是用于Motorola 68010的，之后它被成功地移植到Motorola系列较后的体系结构，再又移植到Sun的SPARC。全局优化有关的工作开始于1984年，过程间优化和并行化开始于1989年。优化器的结构采用混合模式，它有两个优化器，一个用于代码生成之前，另一个用于代码生成之后。这4个编译器共享它们的后端部分，即全局优化器和包含后遍优化器在内的代码生成器，其结构如图21-3所示。

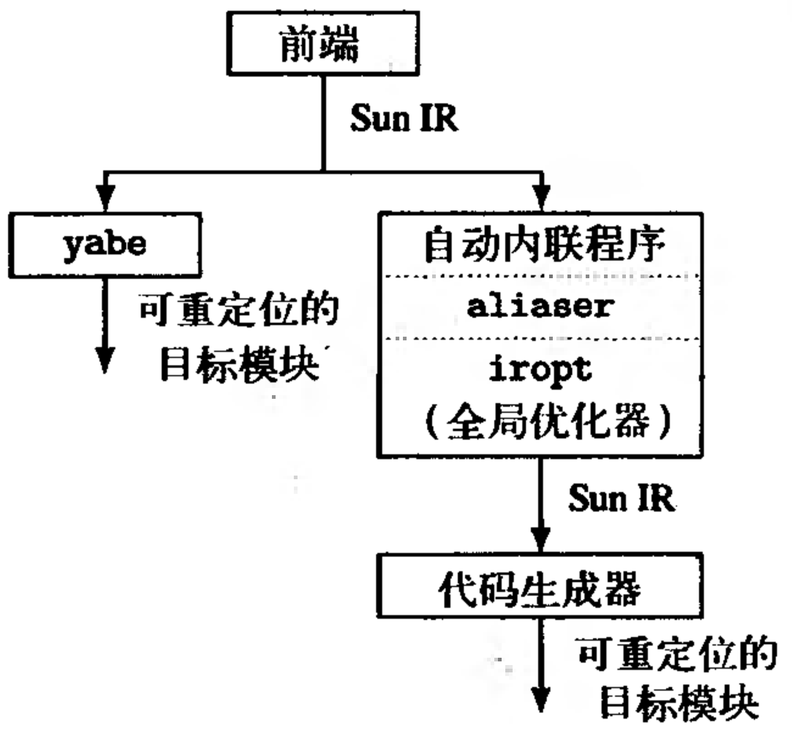


图21-3 Sun SPARC编译器的结构

708

前端的目标代码是称为Sun IR的中间语言，这种中间语言将程序表示成由三元式组成的链表和表示说明信息的若干张表，三元式表示可执行的操作。图21-4给出了由前端生成的Sun IR形式的例子；它与图21-2中第9~17行的代码相对应。其中的运算符大部分是不言自明的；CBRANCH运算符测试其后的第一个表达式，生成一个1或0作为结果，并根据结果分别选择第二个或第三个操作数

⊖ 分支指令中的作废位根据该分支指令是否导致分支转移来规定是否执行延迟槽中的指令。

bbs.theithome.com

作为转移目的地的标号。

如果没有打开优化选项，则使用yabe（“Yet Another Back End”的缩写）代码生成器来处理由前端产生的Sun IR代码。yabe的设计主要顾及代码生成的速度，它没有优化，能产生可重定位的目标模块。如果打开了优化选项，则将Sun IR传送给iropt全局优化器和代码生成器来进行处理。

```
ENTRY "s1_"  { IS_EXT_ENTRY,ENTRY_IS_GLOBAL  }
        GOTO LAB_32;
LAB_32: LTEMP.1 = ( .n { ACCESS V41});
        i = 1;
        CBRANCH (i <= LTEMP.1, 1:LAB_36, 0:LAB_35);
LAB_36: LTEMP.2 = ( .n { ACCESS V41});
        j = i + 1;
        CBRANCH (j <= LTEMP.2, 1:LAB_41, 0:LAB_40);
LAB_41: LTEMP.3 = ( .n { ACCESS V41});
        k = 1;
        CBRANCH (k <= LTEMP.3, 1:LAB_46, 0:LAB_45);
LAB_46: l = ( .a[k,i] { ACCESS V20});
        m = ( .a[k,j] { ACCESS V20});
        *(a[k,j]) = l *m { ACCESS V20, INT };
LAB_34: k = k + 1;
        CBRANCH (k > LTEMP.3, 1:LAB_45, 0:LAB_46);
LAB_45: j = j + 1;
        CBRANCH (j > LTEMP.2, 1:LAB_40, 0:LAB_41);
LAB_40: i = i + 1;
        CBRANCH (i > LTEMP.1, 1:LAB_35, 0:LAB_36);
LAB_35:
```

图21-4　图21-2中Fortran 77程序第9～17行对应的Sun IR代码

这些编译器支持如下4级优化（不优化选项除外）：

O1　这一级只包含代码生成器中的某些优化部分。

O2　这一级和其他较高的级别包含了全局优化器和代码生成器两者中的部分优化。在级别O2，含有全局变量和等价变量、有别名的局部变量、或易失变量的表达式都不是优化的候选；不进行自动内联、软流水、循环展开以及早期的指令调度。

O3　这一级对含有全局变量的表达式进行优化，但假定指针引起的潜在别名有最坏的情况，并且不进行早期的指令调度和自动内联。

O4　这一级积极地跟踪指针可能指向哪些对象，并只在必要时才作出最坏的假设；它依赖特定语言的前端识别潜在有别名的变量、指针变量和潜在别名的最坏情形集合；它也进行自动内联和早期指令调度。

如果选择了全局优化，优化器驱动读一个过程的Sun IR，标识基本块，并建立基本块的前驱和后继链表。如果选择了更高级别的优化（O4），自动内联程序则如15.2节描述的那样，用被调用过程的过程体副本替代对在同一编译单元中的某些子程序的调用。之后执行尾递归删除优化，并且为代码生成器的优化过程标志出其他的尾调用。由此产生的Sun IR送给aliaser处理，aliaser使用由特定语言前端提供的信息来确定在过程的某点上，哪些变量集合可能映射到相同的存储位置。如前面讨论的，在使得有别名的变量集合最小方面，aliaser的激进程度取决于所选择的优化级别。别名信息附加在每一个需要此信息的三元式中，由全局优化器使用。

bbs.theithome.com

超星公司制作请尊重作者版权

除了并行化程序做自己需要的结构分析外，控制流分析通过标识必经结点和回边来进行。所有数据流分析都采用迭代方式。

然后，全局优化器iropt处理每一个过程体，它首先计算附加的控制流信息；具体地，在此时标识循环体，包括显式的循环（例如Fortran 77的DO循环）和由if和goto构造的隐式循环。然后对过程体施加一系列的数据流分析和转换，如果需要的话，每一遍转换首先计算（或重新计算）数据流信息。这一系列转换的结果是该过程的Sun IR代码经改造后的版本。全局优化器依次进行下述转换：

709 ~ 710

1. 聚合量的标量替代和扩展Fortran复数算术运算为实数运算序列；
2. 基于依赖关系的分析和转换（仅O3和O4级），这些分析和转换将在后面描述；
3. 数组地址线性化；
4. 代数化简和地址表达式的重结合；
5. 循环不变代码外提；
6. 强度削弱和归纳变量删除；
7. 全局公共子表达式删除；
8. 全局复写和常数传播；
9. 死代码删除。

基于依赖关系的分析和转换遍是为支持并行化和数据高速缓存优化而设计的，它们可以在优化级别选择为O3或O4时进行。组成它的步骤（依次）如下：

1. 常数传播；
2. 死代码删除；
3. 结构控制流分析；
4. 循环发现（包括确定循环控制变量，循环下界、上界和增量）；
5. 隔离循环体内含调用和过早出口的循环；
6. 依赖关系分析，使用GCD和Banerjee-Wolfe测试，产生方向向量和循环携带的标量du链和ud链；
7. 循环分布；
8. 循环交换；
9. 循环合并；
10. 数组元素的标量替换；
11. 归约识别；
12. 数据高速缓存的循环铺砌；
13. 用于并行化代码生成的效益分析。

全局优化完成后，代码生成器首先将输入给它的Sun IR代码转换为所谓的asm+表示，这种表示由汇编语言指令加上表示控制流和数据依赖信息的结构组成。图21-5展示了asm+代码，它对应于图21-2中Fortran 77代码的第9 ~ 12行，由代码生成开始时的扩展遍产生。我们省略了除第一个BLOCK项之外其他所有BLOCK项之后的注释。这些指令除ENTRY外都是SPARC的原始指令。ENTRY是层次较高的运算符，它表示代码的入口点。注意，形如%r*nnn*的寄存器号表示的是符号寄存器。

711

bbs.theithome.com

```
BLOCK: label = s1_
   loop level = 0
   expected execution frequency = 1
   number of calls within = 0
   attributes:  cc_alu_possible
   predecessors
   successors
       ENTRY  ! 2 incoming registers
       or     %g0,%i0,%i0
       or     %g0,%i1,%i1
BLOCK: label = .L77000081
       or     %g0,%i1,%r118
       or     %g0,%i0,%r119
       add     %r119,-2004,%r130
       ba     .L77000088
       nop
BLOCK: label = .L77000076
       add     %r125,500,%r125
       add     %r124,1,%r124
       cmp     %r124,%r132
       bg     .L77000078
       nop
       ba     .L77000085
       nop
BLOCK: label = .L77000078
       add     %r131,500,%r131
       add     %r133,1,%r133
       cmp     %r133,%r134
       bg     .L77000080
       nop
       ba     .L77000087
       nop
```

图21-5 与图21-2中Fortran 77程序的第9～12行对应的asm+代码

在此之后，代码生成器按如下顺序执行一系列的处理：

1. 指令选择；
2. 其计算效果已明确的汇编语言模板内嵌；
3. 局部优化，包括死代码删除、伸直化、分支链接、将setis指令外提出循环，用无分支的机器方言替换分支代码序列，以及公共条件代码优化；

712

4. 宏扩展，第1遍（扩展二叉分支和其他几种结构）；
5. 活跃值的数据流分析（O2以上）；
6. 软流水和循环展开（O3以上）；
7. 早期指令调度（仅O4）；
8. 图着色寄存器分配（O2以上）；
9. 栈帧分配；
10. 宏扩展，第2遍（扩展存储到存储的传送、max、min、与常数值的比较、入口、出口等）；
11. 延迟槽填充；
12. 后期指令调度；
13. 其计算效果不明确的汇编语言模板内嵌（O2以上）；
14. 宏扩展，第3遍（以简化代码发射）；

bbs.theithome.com

15. 可重定位目标代码的发射。

对于O1优化，寄存器分配采用类似于16.2节描述的基于代价的局部方法。

Sun的编译系统提供了静态的（先于执行的）和动态的（运行时的）两种连接方式。选择静态或动态的，或者选择部分动态部分静态的，是由连接时的选项控制的。

图21-6所示SPARC汇编代码是图21-1中的C函数用O4优化编译后得到的结果列表。注意kind的常数值已传播到条件中，并且死代码已被删除（取存储在.L_const_seg_900000101的常数3.14和存储它至%fp-8的指令除外），循环不变量length*width已提出循环。这个循环已展开4次，乘以height的乘法已削弱为加法，局部变量都已分配到寄存器中，已执行了指令调度，并且也优化了对process()的尾调用。但另一方面，在第一个循环之前有些指令是不必要的，area的累加本应当转换为一个乘法。另外，有点奇怪的是循环展开标准导致展开了前8个迭代，但没有包括最后两个迭代。

```
                sethi   %hi(length),%o0
                sethi   %hi(width),%o1
                sethi   %hi(.L_const_seg_900000101),%o2
                ld      [%o0+%lo(length)],%o0
                ld      [%o1+%lo(width)],%o1
                or      %g0,0,%i1
                ldd     [%o2+%lo(.L_const_seg_900000101)],%f0
                smul    %o0,%o1,%o0
                std     %f0,[%fp-8]
                or      %g0,0,%l0
                or      %g0,0,%l1
                add     %i1,%l1,%i1
                add     %l0,1,%l0
                or      %g0,0,%i0
.L900000111:    add     %l1,%o0,%o1
                add     %o1,%o0,%o2
                add     %i1,%o1,%o1
                add     %o1,%o2,%o1
                add     %o2,%o0,%o2
                add     %i0,%o0,%o3
                add     %o2,%o0,%l1
                add     %o1,%o2,%o1
                add     %o3,%o0,%o3
                cmp     %l0,3
                add     %o1,%l1,%i1
                add     %o3,%o0,%o3
                add     %l0,4,%l0
                bl      .L900000111
                add     %o3,%o0,%i0
.L900000112:    add     %l1,%o0,%l1
                cmp     %l0,10
                bge     .L77000021
                add     %i0,%o0,%i0
.L77000015:     add     %i1,%l1,%i1
                add     %l0,1,%l0
                add     %l1,%o0,%l1
                cmp     %l0,10
                bl      .L77000015
                add     %i0,%o0,%i0
.L77000021:     call    process,2     ! (tail call)
                restore %g0,%g0,%g0
```

图21-6 图21-1中的程序由SPARC C编译器用O4优化产生的机器代码所对应的SPARC汇编代码

bbs.theithome.com

图21-7所示的SPARC汇编代码是图21-2中的主程序由Sun Fortran 77编译器用O4优化编译后得到的（我们已省略了赋初值的循环所产生的代码，并只留下了调用s1()所产生的代码，s1()已自动内联进来）。因为s1()已被内联，该编译器能够知道n的值是500，因此它用因子4展开最内层循环，并且没有产生卷起的副本。展开的循环从标号.L900000112到标号.L900000113，其中包括8个取、4个存、7个加、1个比较和1个分支。除了线性函数测试替换本来还可以删除一个加法外，对于这个展开因子它是最少的指令序列。局部变量都已分配到寄存器中，并且循环已经软流水（注意，取数指令在循环开始标号的前面）。循环中的临时变量的分配方式是使得调度自由度最大的方式。但是，编译器既为主程序生成代码，也为s1()生成了代码，而这是不必要的——因为很明显主程序只调用了s1()，并且s1没有调用其他函数。

713 ~ 714

```
MAIN_:          save    %sp,-120,%sp
                . . .
.L77000057:     sethi   %hi(GPB.MAIN.a),%o1
                add     %o1,%lo(GPB.MAIN.a),%o1
                or      %g0,-2004,%o2
                add     %o2,%o1,%g1
                or      %g0,1,%o5
                or      %g0,500,%o7
.L77000043:     add     %o5,1,%l1
                cmp     %l1,500
                bg      .L77000047
                sll     %l1,5,%o1
                sub     %o1,%l1,%o1
                sll     %o1,2,%o1
                add     %l1,%o1,%o1
                sll     %o1,2,%l0
.L77000044:     add     %o7,1,%o2
                add     %l0,1,%o1
                sll     %o2,2,%o2
                add     %g1,%o2,%l2
                sll     %o1,2,%o1
                or      %g0,1,%l3
                ld      [%l2],%o2
                add     %l2,4,%l2
                add     %g1,%o1,%o0
                add     %l3,1,%l3
.L900000112:    ld      [%o0],%o1
                cmp     %l3,493
                add     %o2,%o1,%o1
                st      %o1,[%o0]
                add     %o0,16,%o0
                ld      [%l2],%o2
                ld      [%o0-12],%o3
                add     %l2,16,%l2
                add     %o2,%o3,%o2
                st      %o2,[%o0-12]
                ld      [%l2-12],%o1
                add     %l3,4,%l3
                ld      [%o0-8],%o4
                add     %o1,%o4,%o1
                st      %o1,[%o0-8]
                ld      [%l2-8],%o2
```

图21-7 图21-2中的程序由SPARC Fortran 77编译器用O4优化产生的机器代码所对应的SPARC汇编代码

bbs.theithome.com

```
                ld      [%o0-4],%o3
                add     %o2,%o3,%o2
                st      %o2,[%o0-4]
                ble     .L900000112
                ld      [%12-4],%o2
.L900000113:    ld      [%o0],%o1
                cmp     %13,500
                add     %o2,%o1,%o1
                st      %o1,[%o0]
                bg      .L77000046
                add     %o0,4,%o0
.L77000056:     ld      [%12],%o1
                add     %13,1,%13
                ld      [%o0],%o2
                cmp     %13,500
                add     %o1,%o2,%o1
                st      %o1,[%o0]
                add     %12,4,%12
                ble     .L77000056
                add     %o0,4,%o0
.L77000046:     add     %11,1,%11
                cmp     %11,500
                ble     .L77000044
                add     %10,500,%10
.L77000047:     add     %o5,1,%o5
                cmp     %o5,500
                ble     .L77000043
                add     %o7,500,%o7
.L77000049:     ret
                restore %g0,%g0,%g0
```

图21-7 （续）

21.2 IBM POWER和PowerPC体系结构的XL编译器

21.2.1 POWER和PowerPC体系结构

POWER体系结构是一种增强的32位RISC机器，它由分支、定点、浮点和存储控制处理器组成。系统中除了寄存器是共享的，并且只有一个分支处理器之外，对于不同的实现，每一种类型的处理器可以有多个。

分支处理器包括条件寄存器、连接寄存器和计数寄存器，并执行条件和无条件分支、调用、系统调用，以及条件寄存器传送和逻辑运算。条件寄存器由8个4位的条件域组成，其中之一由有选择的定点指令来设置，另一个由浮点指令设置，其他的条件域可用于保存多个条件，并且都可以用来控制分支。连接寄存器主要用于存放调用地址，计数寄存器用于循环控制，它们两者都可以读写整数寄存器。

715 ~ 716

定点处理器包含32个32位的整数寄存器，其中gr0当在地址计算以及取地址指令中用作操作数时提交的值是0。定点部件有两种基本寻址方式，寄存器＋寄存器和寄存器＋位移，另外还具有用已计算的地址更新基寄存器的能力。它实现取和存（包括对具有反转字节的半字、对多个字以及对字符串的操作形式）、算术、逻辑、比较、移位、旋转和自陷指令，以及系统控制指令。

bbs.theithome.com

浮点处理器有32个64位的数据寄存器，并且只对双精度值实现了ANSI/IEEE浮点标准。它包括取和存指令、算术指令、整数与单精度值的转换指令、比较指令，以及某些特殊的操作，这些操作在乘法运算之后不立即进行截断便做加法或减法运算。

存储控制处理器提供段式主存以及与高速缓存和后备转换缓冲区的接口，并负责虚实地址转换。

PowerPC体系结构几乎是POWER向上兼容的扩充，它可以有32位和64位的实现。64位的实现总是同时允许64位和32位的操作模式，两种模式不同在于有效地址的计算和出现的指令序列。PowerPC处理器由分支、定点和浮点处理器组成，并且同POWER一样，系统可以有一个以上的定点和浮点处理器，但只有一个分支处理器。

分支处理器有一个32位的条件寄存器、一个64位的连接寄存器和一个64位的计数寄存器，它们的作用与POWER中的基本相同。

定点处理器有32个64位的整数寄存器，其中gr0的作用与POWER中的相同。PowerPC包含与POWER相同的寻址方式，除了使某些难于处理的情形（如使用基地址寄存器作为一条具有更新能力的取数指令的目标地址，或者取包含基寄存器作为目标地址的多个字）成为非法情形之外，它实现了与POWER定点部件相同种类的指令，但增加了存储用法控制指令。有少数指令被删除了，其原因或者是由于难于实现，例如差或零指令（有助于求最大和最小值）和循环左移屏蔽插入指令，或者是由于改变到64位操作而不再需要了。另外也增加了一些新指令，其中许多是用于处理高速缓存和后备转换缓冲区的。

PowerPC浮点处理器有32个64位的数据寄存器，并实现单精度和双精度值的ANSI/IEEE标准。除了处理32位形式的新指令外，指令集实际上与POWER相同。

典型的POWER和PowerPC指令有三个操作数——两个源操作数和一个结果。第一个源操作数和结果几乎总是寄存器，第二个源操作数可以是寄存器，也可以是小常数。在汇编语言中，操作数的顺序是结果操作数、第一个源操作数，然后是第二个源操作数。有关汇编语言的更多
717 细节请参见附录A.2。

21.2.2 XL编译器

用于IBM POWER和PowerPC体系结构的编译器是著名的XL系列，包括PL.8、C、Fortran 77、Pascal和C++ 等编译器，除了第一个之外，它们都可提供给用户。它们源于1983年开始为IBM的一种RISC体系结构提供编译器的项目，这种RISC体系结构是IBM 801和POWER之间的一个中间阶段，它从未作为产品发布。但是第一个基于XL技术的两个编译器事实上是用于PC RT的一个优化Fortran编译器和一个C编译器，其中Fortran编译器只发布给经过选择的少数几个用户，C编译器只用于IBM内部的开发。这两个编译器的后端是可改变的，因此能移植到IBM 370体系结构、前面提到的那个未发布的体系结构、PC RT、POWER，以及最近的PowerPC。

XL编译器的结构如图21-8所示，它遵循在低级代码上进行优化的模式。每一种编译器由称为转换器的前端、全局优化器、指令调度器、寄存器分配器、指令选择器，以及称为最后汇编的一遍组成；这个最后汇编产生可重定位映像和汇编语言列表。此外，有一个称为根服务的模块，它与所有遍打交道，并通过诸如区分有关如何生成列表和报错之类的信息而使编译器能与多个操作系统兼容。有一个反汇编器能够由可重定位模块产生汇编语言。这些编译器都是用PL.8编写的。

bbs.theithome.com

转换器通过调XIL库例程将源语言转换成称为XIL的中间形式。这些XIL生成例程不仅仅生成指令，例如，它们也生成常数替代原本应计算该常数的指令。转换器可以由一个转换源语言到不同中间语言的前端，后随一个从其他中间形式转换到XIL的转换器组成。System/370的C转换器是这样做的，370、POWER和PowerPC的C++ 也是这样做的，它们都首先转换源语言到370的中间语言。

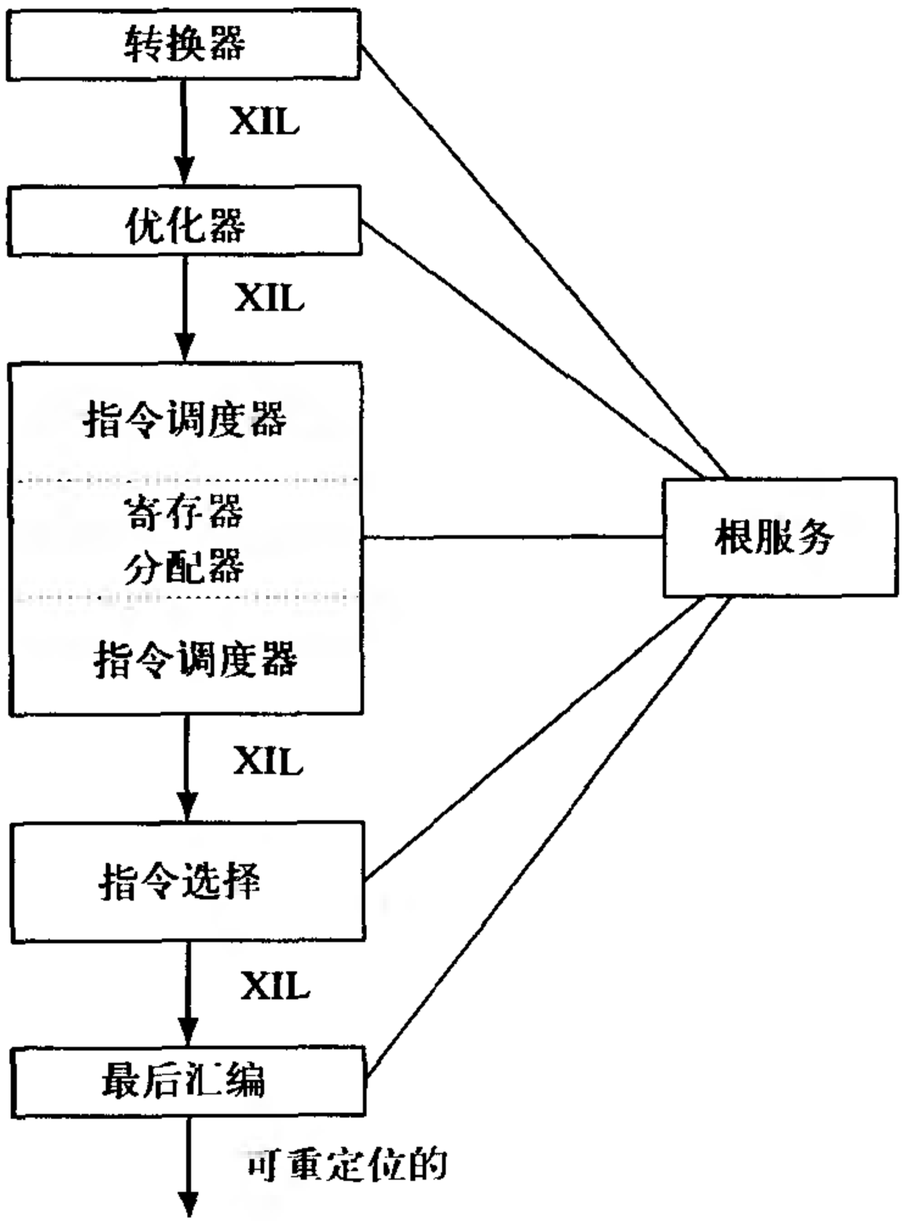


图21-8 IBM XL编译器结构

编译器的后端（除了源到XIL的转换器之外的所有遍）称为TOBEY，即“TOronto Back End with Yorktown”首字母的缩写词，它象征着后端继承的是801的PL.8编译器，并且是POWER系统开发的预演，尽管后来几乎每一个模块都已有重大改变或已被替换。

一个编译单位的XIL由一个*过程描述符表*（procedure descriptor table）和一个指向它的代码表示的指针组成。过程描述符表由每一个过程有关的信息（如栈帧的大小）和它所涉及的全局变量有关的信息组成。代码的表示是一个*过程表*（procedure list），它包含指向表示指令的XIL结构的指针；指令是非常低级的形式并且与源语言无关。每一条指令由*计算表*（computation table）中的一项来表示，计算表也简称CT，它是一个变长记录数组，这些记录是指令的中间代码的前序遍历表示。过程中相同的指令共享同一个CT项。每一个CT项由一个操作码后随可变个操作数组成，操作数可以是范围从0到$2^{16}-1$的整数、指向大整数和负整数的索引、浮点数、寄存器号、标号、符号表引用，等等。操作码可以是RISC风格的操作符，取或存，复合操作符（如MAX或MIN），管理性的操作符（例如过程头或基本块的开始），或者控制流操作符，包括无条件的、有条件的以及多路形式的（后者由一个选择符和一张标号表组成）。变量和中间结果用符号寄存器表示，它们中的每一个构成了*符号寄存器表*（symbolic register table）中的一项；每一个符号寄存器表项指向定义它的CT项。图21-9展示了过程表、CT和符号寄存器表之间的关系。

bbs.theithome.com

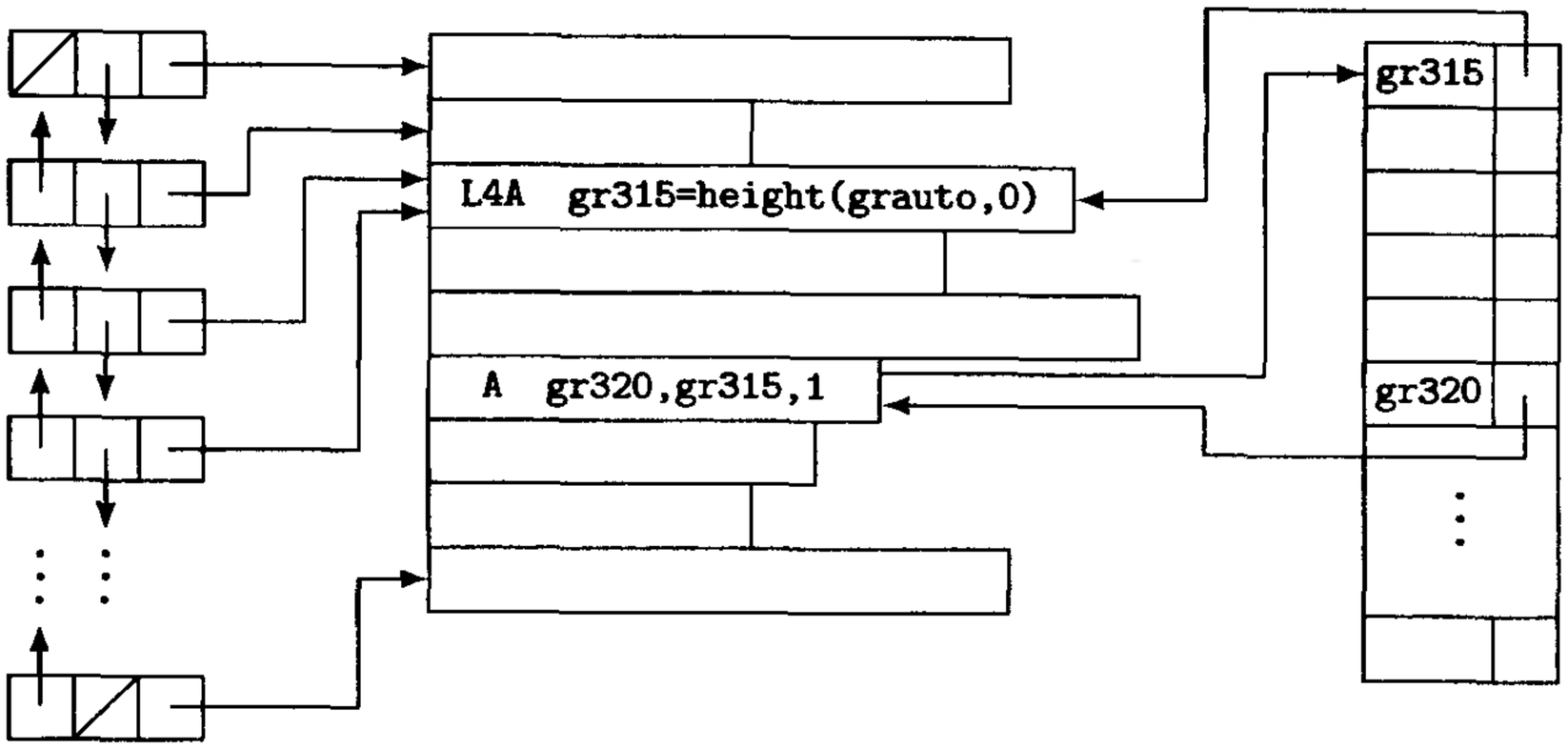


图21-9 过程的XIL结构

图21-10给出了图21-1中第一部分代码由C转换器产生的XIL代码的外部表示的一个例子。注意，这个外部表示有许多隐含的和未表示出来的代码结构。其中操作码的含义如下:

1. PROC——过程入口点
2. ST4A——存字指令
3. L4A——取字指令
4. C4——比较字指令
5. BF——根据条件为假分支指令
6. M——乘运算指令
7. A——加运算指令
8. B——无条件分支指令

718 ~ 719

形式为 *.variable*的操作数表示*variable*的地址，gr*n*表示符号寄存器*n*，cr*n*表示符号条件寄存器*n*。

```
        PROC
        ST4A    area(grauto,0)=0
        ST4A    volume(grauto,0)=0
        ST4A    kind(grauto,0)=0
        ST4A    height(grauto,0)=0
        L4A     gr315=height(grauto,0)
        C4      cr316=gr315,10
        BF      CL.2,cr316,0x1/lt
CL.1:   L4A     gr317=kind(grauto,0)
        C4      cr318=gr317,0
        BF      CL.3,cr318,0x4/eq
        L4A     gr319=area(grauto,0)
        L4A     gr314=.length(gr2,0)
        L4A     gr320=length(gr314,0)
        L4A     gr313=.width(gr2,0)
        L4A     gr321=width(gr313,0)
        M       gr322=gr320,gr321
        A       gr323=gr319,gr322
        ST4A    area(grauto,0)=gr323
```

图21-10 图21-1中C程序第1~10行对应的XIL代码

bbs.theithome.com

```
L4A     gr324=volume(grauto,0)
L4A     gr320=length(gr314,0)
L4A     gr321=width(gr313,0)
M       gr322=gr320,gr321
L4A     gr315=height(grauto,0)
M       gr325=gr315,gr322
A       gr326=gr324,gr325
ST4A    volume(grauto,0)=gr326
B       CL.4
```

图21-10 （续）

XIL的取和存操作有详细表述的地址。例如，一条一般的取字指令可视为具有下面的形式:

L4A *reg* = *name* (*rbase*, *rdisp*, *rindex*, …)

为了从这种形式生成370的代码，*rbase*、*rdisp*、*rindex*三部分都将被用到，只要$0<rdisp<2^{12}$；如果这个不等式不满足，则首先生成把*rdisp* + *rindex*或*rbase* + *rdisp*放至寄存器的附加指令。为了生成POWER的代码，既可以在其他部分为0时使用*rdisp* + *rindex*或*rbase* + *rdisp*（如果$0<rdisp<2^{16}$），也可以首先生成合并这些部分的附加指令。优化器负责保证指令选择器能够生成所选择目标机体系结构的合法指令。

除了MAX和MIN外，XIL包含有任意个数操作数的字节连接运算符、乘法运算符、除法运算符等。优化器将MAX和MIN运算符转换成另一种形式，针对POWER的指令选择器能够从这种形式生成对应的由两条指令组成的无分支序列。乘法运算符生成由移位、加和减指令组成的指令序列，或者生成一条硬件乘指令，但无论使用那一种，总是选择对于给定的操作数和目标机较高效的一种。除以1、3、5、7、9、25、125的除法以及这些整数与2的幂的乘积生成一个长乘法和一个双字移位；其他的除法生成一个除法运算指令。

编译器的后端使用了另外一种中间语言，叫做YIL，它用于存储有关的优化，并可以在将来用于并行化。一个过程的YIL由TOBEY根据它的XIL生成，它除了包含XIL中的结构外，还包含关于循环结构的表示、赋值语句、下标运算，以及语句一级的条件控制流。它也表示SSA形式的代码。当然，其目的是为了产生适合依赖关系分析和循环转换的代码。在这种分析和转换执行之后，YIL将被重新转换成XIL。

别名信息由特定语言的转换器提供给优化器，优化器通过调用前端的例程而得到这些信息。除了每一种语言定义提供的别名规则之外，并没有做进一步的分析。

控制流分析是简单的。它标识过程内基本块的边界，建立流图，构造流图的深度为主搜索树，并将它划分为区间。基本块结构记录在用基本块编号作为索引的一张表中，对于每一个基本块，它包含指向该基本块在过程表中对应的第一项和最后一项的两个指针，以及这个基本块的前驱和后继基本块的编号表。

720
~
721

数据流分析通过区间分析来进行，对非可归约区间则采用迭代方法，数据流信息记录在位向量中。对于某些优化，如重结合和强度削弱，这些位向量被转换成du和ud链。在这些遍的处理过程中，活跃寄存器和du与ud链随需要而更新。

优化器进行如下一系列的转换（按执行它们的顺序）:

1. 根据标号的多少，将多路分支转换成由比较与条件分支构成的序列，或者转换成通过跳转表的转移;

2. 映射已分配局部栈的变量到寄存器＋位移地址;

3. 如果要求进行内联并且几种启发式标准也允许的话，内联来自当前编译模块中的例程;

bbs.theithome.com

4. 非常激进的值编号（比任何已发表的版本都更先进）；

5. 全局公共子表达式删除；

6. 循环不变代码外提；

7. 存数指令下移；

8. 死代码删除；

9. 重结合（参见12.3.1节）、强度削弱，以及POWER和PowerPC的带更新形式的取和存指令的生成；

10. 全局常数传播；

11. 死代码删除；

12. 某些体系结构特殊的优化，它们的非正式名称为“wand waving”，例如将MAX和MIN转换成无分支的序列；

13. 宏操作扩展，即，将所有操作码和地址表达式的层次降低到目标机所支持的形式，将调用转变成访问参数和执行调用的指令序列，转变大常数为生成其值的指令序列，等等；

14. 值编号；

15. 全局公共子表达式删除；

16. 死代码删除；

17. 已死归纳变量删除，包括浮点变量。

浮点除法被转换成三条指令的序列，它包含有一个乘法和一个加法。如果在Fortran 77编译过程中打开了边界检查，则紧接着重结合和强度削弱遍之后，进行自陷移动分析和代码移动。

TOBEY包含两个寄存器分配器，一个是“虽快却粗糙”的局部分配器，它在不需要优化时使用；一个是图着色全局寄存器分配器，该分配器基于Chaitin的方法，但其溢出代码类似于Briggs的方法（参见16.3节）。在寄存器分配之前有一遍对过程入口和出口进行细化，并执行尾调用化简和叶例程优化。图着色寄存器分配也进行“清扫”（scavenging），即值编号的一种版本，它将溢出临时变量的取和存指令移出循环。该分配器对每一个过程尝试16.3.12节讨论的所有三种选择溢出代码的启发式方法，并从中选用最好的一个来选择溢出代码。

好几篇论文中都有其指令调度器的介绍（参见21.7节）。除了基本块和分支调度外，指令调度器也进行全局调度。全局调度在无环路的流图上工作，并使用程序依赖图（参见9.5节）来描述对调度的约束。在优化处理过程中，指令调度在寄存器分配之前运行，并且当寄存器分配生成了溢出代码时，在其后再运行一次。

最后的汇编遍对XIL进行两遍处理，一遍做若干种窥孔优化，例如，合并由与对应算术运算相比较而设置的条件代码，并删除这些比较（有关例子参见图21-11）；另一遍输出可重定位的映像和列表。最后汇编调用特定语言转换器中的例程来获得要包含在可重定位模块中的调试信息。

```
s      rx=ry,2
. . .
ci     cc0=ry,2
bt     label
```

a)

```
s.     rx=ry,2
. . .
bt     label
```

b)

图21-11 POWER中将条件代码设置和算术运算合并的例子。a) 中的ci已在b) 中通过设置它的记录位而合并到s中

图21-12给出的是图21-1的程序的POWER汇编代码，由XL反汇编器根据XL C编译器在选择O3优化级别下生成的该程序的目标代码生成。kind的常数值已传播到条件中，并且死代码已被删除；循环不变量length*width已从循环中移出；循环已用因子2展开；局部变量都已分配到寄存器中；并且已执行了指令调度。另一方面，对process()的尾调用没有被优化，area的累加也没有转变成单一乘法。

bbs.theithome.com

```
.main:  mfspr   r0,LR
        l       r5,T.22.width(RTOC)
        stu     SP,-64(SP)
        l       r4,T.26.length(RTOC)
        st      r0,72(SP)
        l       r0,0(r4)
        l       r5,0(r5)
        cal     r4,5(r0)
        muls    r0,r0,r5
        cal     r3,0(r0)
        mtspr   CTR,r4
        cal     r4,0(r3)
        cal     r5,0(r3)
__L34:  ai      r6,r5,1
        a       r3,r3,r0
        muls    r5,r5,r0
        a       r4,r4,r5
        muls    r5,r6,r0
        a       r4,r4,r5
        a       r3,r3,r0
        ai      r5,r6,1
        bc      BO_dCTR_NZERO,CR0_LT,__L34
        bl      .processPR
        cror    CR3_SO,CR3_SO,CR3_SO
        l       r12,72(SP)
        ai      SP,SP,64
        mtspr   LR,r12
        bcr     BO_ALWAYS,CR0_LT
```

图21-12 图21-1的程序的POWER汇编代码，由XL反汇编器根据XL C编译器在选择O3优化级别下生成的该程序的目标代码生成

图21-13给出了图21-2的子程序s1()的POWER汇编代码，由XL反汇编器根据XL Fortran编译器在选择O3优化级别下生成的该子程序的目标代码生成。子程序s1()没有内联。内层循环已用因子2展开（从标号__Lfc到下一条bc指令）。这个展开的循环含有4条取（其中两个带更新）指令、两条带更新的存指令、两条加指令以及一条分支指令。对于POWER体系结构，这是最少的情形。局部变量已分配到寄存器并且已执行了指令调度。

722
~
723

```
.s1:    l       r10,0(r4)
        st      r31,-4(SP)
        cal     r8,0(r10)
        cmpi    1,r8,0
        bc      BO_IF_NOT,CR1_FEX,__L1a4
        ai      r6,r10,-1
        cal     r12,0(r6)
        cmpi    0,r12,0
        ai      r7,r3,1996
        ai      r9,r3,-4
        cal     r11,2(r0)
        bc      BO_IF_NOT,CR0_GT,__L154
        rlinm   r3,r10,31,1,31
        rlinm.  r4,r10,0,31,31
        cmpi    1,r3,0
```

图21-13 图21-2的子程序s1()的POWER汇编代码，由XL反汇编器根据XL Fortran编译器在选择O3优化级别下生成的该子程序的目标代码生成

bbs.theithome.com

```
          cal     r31,0(r7)
          mtspr   CTR,r3
__Lec:    ai      r12,r12,-1
          cal     r5,0(r31)
          cal     r4,0(r9)
__Lf8:    bc      BO_IF,CR1_VX,__L124
__Lfc:    lu      r3,4(r4)
          l       r0,4(r5)
          a       r3,r3,r0
          stu     r3,4(r5)
          lu      r3,4(r4)
          l       r0,4(r5)
          a       r3,r3,r0
          stu     r3,4(r5)
          bc      BO_dCTR_NZERO,CR0_LT,__Lfc
          bc      BO_IF,CR0_EQ,__L134
__L124:   lu      r3,4(r4)
          l       r4,4(r5)
          a       r3,r3,r4
          stu     r3,4(r5)
__L134:   cmpi    0,r12,0
          ai      r31,r31,2000
          bc      BO_IF_NOT,CR0_GT,__L154
          rlinm   r3,r10,31,1,31
          rlinm.  r4,r10,0,31,31
          cmpi    1,r3,0
          mtspr   CTR,r3
          b       __Lec
__L154:   ai.     r8,r8,-1
          ai      r9,r9,2000
          ai      r7,r7,2000
          ai      r11,r11,1
          ai      r6,r6,-1
          bc      BO_IF_NOT,CR0_GT,__L19c
          cal     r12,0(r6)
          cmpi    0,r12,0
          bc      BO_IF_NOT,CR0_GT,__L154
          rlinm   r3,r10,31,1,31
          rlinm.  r4,r10,0,31,31
          cal     r31,0(r7)
          cmpi    1,r3,0
          mtspr   CTR,r3
          ai      r12,r12,-1
          cal     r5,0(r31)
          cal     r4,0(r9)
          b       __Lf8
__L19c:   l       r31,-4(SP)
          bcr     BO_ALWAYS,CR0_LT
__L1a4:   bcr     BO_ALWAYS,CR0_LT
```

图21-13 （续）

21.3 DEC用于Alpha的编译器

21.3.1 Alpha体系结构

Alpha是一种全新的体系结构，它是由数字设备公司（DEC）作为VAX和基于MIPS的系统

bbs.theithome.com

的后一代产品而设计的，它是一种非常现代且新式的64位RISC设计。Alpha有32个64位的整数寄存器（其中r31当作为源操作数时交付值0，并且忽略写给它的结果）和32个64位的浮点寄存器（其中f31具有与r31类似的作用）。它只有一种寻址方式，即寄存器＋位移，并且没有条件码。

整数指令包括取和存，无条件分支，基于寄存器中的值与0相比较的有条件分支，分支到子程序和从子程序返回的跳转，以及在协例程之间进行分支的跳转，算术指令（包括其中一个操作数可以缩放4或8倍的加和减指令），在指定的寄存器中设置一位的有符号和无符号比较指令，逻辑、移位和有条件传送指令，以及有利于字符串处理的一组丰富的操纵字节的指令。

浮点设计实现了ANSI/IEEE标准，但对于一些边角情况需要来自操作系统的很多协助。它实现了VAX和ANSI/IEEE两种单精度和双精度格式，所提供的指令包括取和存、分支、条件传送、VAX和ANSI/IEEE两种算术运算，以及整数与VAX浮点格式的转换和整数与ANSI/IEEE浮点格式的转换。

系统控制指令提供有预取提示，实现了关于弱系统存储模式的存储器和自陷栅栏，并实现了所谓的有特权的体系结构库（Privileged Architecture Library）。其中最后一种是为了给那种可以从一种系统实现变化到另一种系统实现的操作系统提供方便而设计的。

典型的Alpha指令有三个操作数——两个源操作数和一个结果操作数。第一个源操作数和结果操作数几乎总是寄存器，第二个源操作数可以是寄存器或小常数。在汇编语言中，操作数的顺序是结果操作数、第一源操作数，然后是第二源操作数。关于其汇编语言的更多细节请参见附录A.3。

21.3.2 Alpha的GEM编译器

DEC关于Alpha编译器的开发是著名的GEM项目[⊖]。GEM不是首字母的缩写词，尽管它全是大写字母，这个名字是从为Prism开发编译器的项目沿袭下来的。Prism是DEC早先的一种RICS设计，但它从未成为一种产品。GEM项目开始于1985年，与其并行开发的还有几种内部的RISC。需要为好几种目标机提供编译器的需求导致GEM项目产生了易于重新移植到不同目标机的编译器，并且这些编译器已经若干次被移植到不同的目标机。1987年Prism被暂时地选择作为DEC进入RISC市场的第一个项目，那时已经有Prism上基于GEM的Fortran 77编译器。但是，DEC不久便将其DEC工作站系列改为使用MIPS平台，并且在1988年开始了将基于GEM的编译器移植到基于MIPS的DEC工作站上的工作，其中基于GEM的Fortran 77编译器于1991年首先被移植到基于MIPS的DEC工作站。

724 ~ 726

移植GEM Fortran 77编译器到Alpha的工作开始于1989年11月。到1990年夏季完成了BLISS编译器。

Alpha上可用的编译器包括支持VAX VMS和UNIX扩充的Fortran 77、Fortran 90、ANSI C（具有其他方言的选项）、C++、Ada、COBOL、Pascal、PL/I以及BLISS。这些编译器的结构如图21-14所示。一组称为GEM shell的实用程序提供一个共同的操作系统接口[⊖]，此接口与VMS、Ultrix、OSF/1以及Windows NT兼容；它包括正文输入/输出、诊断实用程序、语言敏感的编辑器、虚存管理、调试工具，它给所有的编译器和环境提供了一种共同的外貌和感觉。每一个编译器有它自己的前端，但其他所有部分是共享的。编译器的所有部分，除C前端是用C编写的外，都

⊖ GEM项目也产生了Digital的VAX和基于MIPS系统的编译器。

⊖ 实际上存在着GEM shell的若干种版本，对于若干组主机操作系统和体系结构中的每一对组合，均各有一个版本。

bbs.theithome.com

是用BLISS编写的。BLISS宏被用来生成中间语言的元组的运算符记号表和其他一些后端表格。

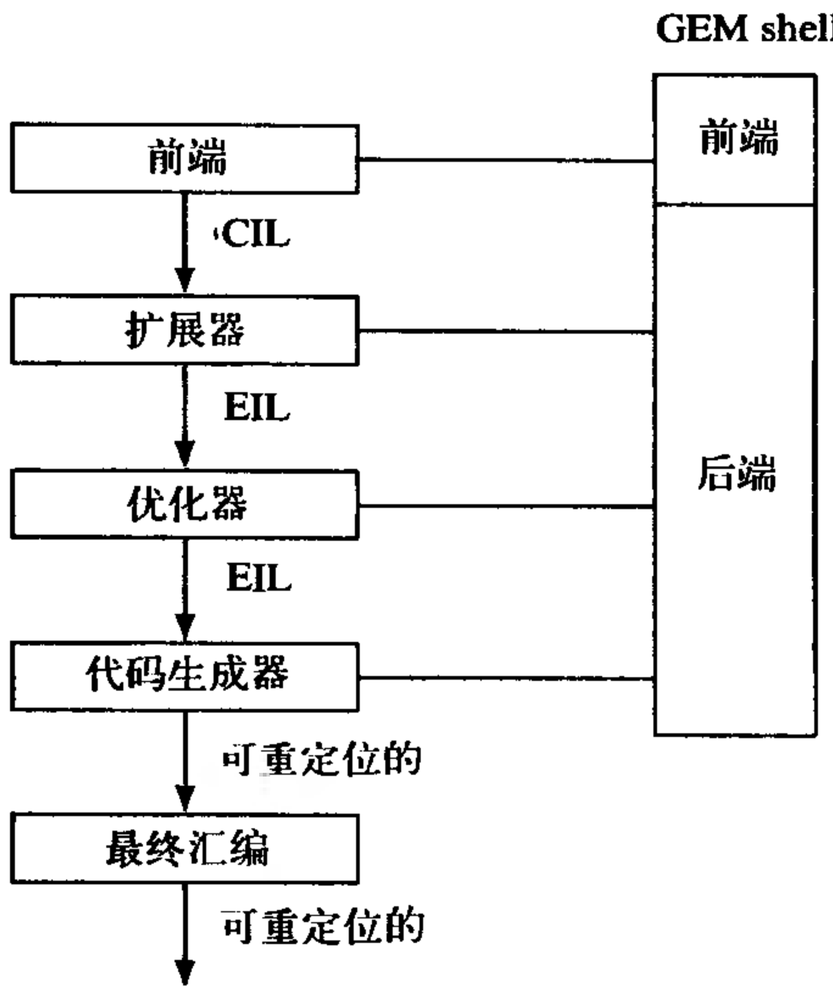


图21-14 DEC Alpha 编译器的结构

在每一种编译器中，前端每次处理一个文件，进行词法、语法和静态语义分析，并生成一种叫做CIL（Compact Intermediate Language）的中间形式。前端之后的所有遍都每次处理一个过程。

CIL和EIL（Expanded Intermediate Language）都用结点的双向链表来表示一个编译单位。在CIL中，这个表对应于一个编译单位，结点具有固定大小。每一个结点是一个记录，它的各个域表示该结点的种类和子类、标志、前向和后向链、属性，以及指向它的操作数结点的指针。有若干种运算符具有比较高级的表示，例如，取一个变量的值引用的是该变量的名字，下标表是用含有下标表达式和跨步的表来表示的。图21-15a展示了语句a=x[i]的CIL形式，其中省略了某些次要的子域。

```
$1: fetch(i)

$2: subscr($1,[4],[0]; posn=1)

$3: aref(x,$2)

$4: fetch($3)

$5: store(a,$4)
```

a)

```
$1: symref(; symbol=i)
$2: fetch($1)
$3: litref(; literal=[4])
$4: mul($1,$3)
$5: symref(; symbol=x)
$6: aplus($5,$4)
$7: fetch($6)
$8: symref(; symbol=a)
$9: store($8,$7)
```

b)

图21-15 a) C语句a=x[i]的CIL代码的例子，b) 将它扩展到EIL之后

该编译器使用混合优化模式，代码选择位于全局优化和寄存器分配以及指令调度之间。后端的第一遍，即扩展器，转换CIL为低级的EIL形式。在EIL中，结点表对应一个过程，结点的大小是可变的。结点是元组，它表示具有“适度强”机器类型的运算。每一个元组由运算符、操作数和结果的类型、它的各个操作数，一个属性表，以及链接其他元组的前向和后向链所组成。与CIL相比，取变量的值在EIL中被扩展成对一个符号的引用，以及通过该符号引用来取数的操作，下标表被表示成一系列计算该下标的线性形式的元组。从CIL到EIL的这种扩展的另一

bbs.theithome.com

超星公司制作 请尊重作者版权

个例子是过程调用：在CIL中，过程调用是用描述实参表以及如何传递每一个参数和如何接受结果的元组详细表示的；而在EIL中，参数表被扩展成一系列实际计算这些参数的元组。EIL也包含专门为优化器而设计的结点，如cseref结点，它表示使用的是公共子表达式。图21-15b展示的EIL代码是由扩展器从图21-15a所示的CIL代码产生的，其中同样省略了次要的子域。

该编译器有如下6级优化：

O0 这一级只进行窥孔优化，并给每一个局部变量分配一个不同的栈存储单元。

O1 这一级产生用于调试的代码；它进行局部公共子表达式删除和使得变量可共享栈单元和寄存器的生命期分析。

727 ~ 728

O2 这一级增加了不会显著增加代码体积的传统全局优化。

O3 这一级增加了循环展开和删除分支的代码复制（尾融合的逆转）。

O4 这一级增加了同一编译单位中的过程内联。

O5 这一级增加了依赖分析和软流水。

从O1级开始，控制器进行过程间分析，以确定在它能看到的范围内的调用图，并进行同一编译单位内的过程内联。然后对组成一个编译单位的这些过程进行优化，并且从调用树的叶结点至根每次一个过程地为它们生成代码。

优化器首先构造过程的流图和它的必经结点树，在此过程中删除空基本块、不可到达代码和不必要分支。基本块表按“循环序”排序，它类似于深度为主搜索树，但保持每一个循环体中的基本块是相邻的，以便改善代码的局部性。每一个循环之前插有一个空的前置块，循环之后相应也插有一个块。另外，while循环被转换成repeat循环，因为不变量外提遍的设计不将代码移出至while循环之外。数据流分析采用了Reif和Lewis（[ReiL77]和[ReiL86]）的符号计算方法，但没有构造他们的全局值图。别名分析有两遍，第一遍在数据流分析之前，它用符号访问和可能的别名信息标注数据访问结点。第二遍在数据流分析期间进行，它上下遍历必经结点树，计算表示赋值潜在副作用的位向量。

一系列的窥孔优化在优化的三个点上进行；这三个点是数据流分析之前、数据流分析之后以及优化结束时。这些窥孔优化包括代数化简，扩展乘以或除以常数的乘、除法，扩展位和字节域访问为取字、抽取以及相应位和字节域的扩展，以及其他一些优化。

所执行的全局优化包含下述一些内容：

1. 归纳变量强度削弱；
2. 线性函数测试替换；
3. 循环展开；
4. 全局公共子表达式删除，包括确定那些重新计算它比用公共子表达式更好的表达式；
5. 循环不变代码外提；
6. 全局复写和常数传播；
7. 无用存储操作删除；
8. 基地址捆绑，即确定那种减去一个小常数且此小常数足以放入存取指令位移域的地址表达式；
9. 软流水。

代码生成器的设计受到了Wulf等人[WulJ75]在产品质量编译器的编译器（PQCC，Production-Quality Compiler Compiler）中的工作和PDP-11 BLISS编译器的启发。它由如下6遍组成：

1. Context 1遍用代码模板匹配一个过程的EIL树，每个结点的每一个模板有一个代价。过

bbs.theithome.com

越是公司制作请尊重作者版权

程对应的EIL树的一个特定部分可以有好几个模板——实际上，每个结点大约有5到6个可应用的模板。然后Context 1选择一组对于整个树具有最小代价的模板。

2. 接下来，IL Scheduling遍对一个基本块所选择的模板进行充分的交替分析，以便产生高质量的寄存器分配。它也使用Sethi-Ullman数计算每一个表达式需要的最小寄存器个数。

3. 接下来，Context 2遍为需要临时变量的对象创建临时变量，并计算它们的生存期。

4. 然后，Register History遍追踪每一个已分配有存储单元的临时变量的重取（reload），标注那种有多次使用的临时变量，以便可以将它们分配到相同的寄存器。

5. TN Pack遍根据Context 1所选择的模板中包含的动作，对临时变量名进行装包（如16.2节介绍的）来进行寄存器分配。

6. 最后，Code遍根据所选择的模板和寄存器分配流出实际的目标代码。

代码生成器合并一个编译单位中所有过程的代码，产生一个可重定位目标模块，并确定何处可以使用（短位移）分支，以及何处需要跳转指令。

最后，编译器的最后遍进行窥孔优化，如机器方言、转移化简、交叉转移以及代码复制（即与交叉转移相反的做法）；并利用详细的机器模型和一种以确定直线代码关键路径为基础的算法进行指令调度。这个调度器也进行某种与安全前瞻执行[⊖]有关的跨基本块调度。

性能剖面分析结果的反馈信息可用来指导过程集成和对已生成的基本块排序，并裁剪频繁调用过程的调用约定。

图21-16给出了GEM Fortran编译器对图21-1中所示的函数用O5优化产生的汇编语言代码。注意，kind的常数值已传播到条件中，并且已经删除死代码；循环不变量length*width已从循环内移出；循环已用因子5展开；所有局部变量都已分配到寄存器中；并且已执行了指令调度，包括采用安全前瞻调度将取r27的指令从后继基本块中前移到标号L$3之后一条指令的位置。但另一方面，栈帧的分配是不必要的，area的计算本应当归约成单独一条乘法，对process()的尾调用也没有优化。还有，其结果被累加到一起以产生valume值的那些乘法本应当强度削弱为加法。

729 ~ 730

```
main:   ldah    gp,(r27)
        lda     gp,(gp)
        ldq     r1,8(gp)
        lda     sp,-16(sp)
        ldq     r0,16(gp)
        clr     r16
        stq     r26,(sp)
        clr     r17
        ldl     r1,(r1)
        ldl     r0,(r0)
        mull    r0,r1,r0
        clr     r1
L$3:    mull    r1,r0,r2
        ldq     r27,(gp)
        addl    r1,1,r3
        addl    r1,2,r4
        addl    r1,3,r5
```

图21-16 由GEM C编译器对图21-1中的函数用O5优化产生的Alpha汇编语言

⊖ 基本块中的一条指令的执行是前瞻的（speculative），如果该指令来自于其间间隔有一个条件的后继基本块中。如果执行该条件的另一个分支时，不需要对该前瞻指令的执行效果进行补偿，则这个前瞻执行是安全的。

bbs.theithome.com

超星公司制作 请尊重作者版权

```
        addl    r1,4,r6
        addl    r16,r0,r16
        addl    r16,r0,r16
        addl    r16,r0,r16
        addl    r1,5,r1
        addl    r16,0,r16
        addl    r16,r0,r16
        mull    r3,r0,r3
        addl    r17,r2,r2
        mull    r4,r0,r4
        addl    r2,r3,r2
        addq    r1,-10,r3
        mull    r5,r0,r5
        addl    r2,r4,r2
        mull    r6,r0,r6
        addl    r2,r5,r2
        addl    r2,r6,r17
        blt     r3,L$3
        jsr     r26,r27
        ldah    gp,(r26)
        ldq     r26,(sp)
        lda     gp,(gp)
        clr     r0
        lda     sp,16(sp)
        ret     r26
```

图21-16 （续）

图21-17给出了GEM Fortran编译器对图21-2中子程序s1()用O5优化产生的汇编语言。从s1_到L$21的代码是该子程序和循环的初始化代码，从L$13开始的代码是关于外层循环的循环控制代码。余下的是最内层循环的代码。最内层循环已经用因子4展开（代码开始于L$21）并且还产生了一个卷起的循环副本，它开始于L$24。在这个卷起的循环中有9条指令，因为没有实现线性函数测试替换。展开的循环有21条指令，其中只有一条不是必须的，这也是因为没有实现线性函数测试替换。局部变量都已分配到寄存器中，并且已执行了指令调度。但另一方面，没有实现将过程s1()集成到主程序中。这样做本来可以节省调用和返回开销，并且可以传播n的值（=500）到该子程序中，使得不需要最内层循环的卷起循环副本，因为500能够被4整除。

731
~
732

```
s1_:    ldl     r17,(r17)
        mov     1,r0
        mov     2,r1
        ble     r17,L$4
L$5:    mov     r1,r2
        cmple   r2,r17,r3
        beq     r3,L$7
        nop
L$8:    sll     r2,4,r3
        ble     r17,L$13
        sll     r2,11,r5
        s4subq  r3,r3,r3
        subq    r5,r3,r3
        sll     r0,4,r5
        sll     r0,11,r6
        s4subq  r5,r5,r5
```

图21-17 由GEM Fortran编译器对图21-2中子程序s1()用O5优化产生的Alpha汇编语言

bbs.theithome.com

```
        subq    r6,r5,r5
        subl    r17,3,r6
        addq    r16,r3,r3
        addq    r16,r5,r5
        cmplt   r17,r6,r7
        mov     1,r4
        lda     r3,-2000(r3)
        lda     r5,-2000(r5)
        bne     r7,L$24
        ble     r6,L$24
L$21:   ldl     r8,(r5)
        addl    r4,4,r4
        ldl     r19,(r3)
        addl    r8,r19,r8
        stl     r8,(r3)
        ldl     r19,4(r5)
        ldl     r8,4(r3)
        addl    r19,r8,r8
        stl     r8,4(r3)
        ldl     r19,8(r5)
        ldl     r8,8(r3)
        addl    r19,r8,r8
        stl     r8,8(r3)
        ldl     r19,12(r5)
        ldl     r8,12(r3)
        lda     r3,16(r3)
        lda     r5,16(r5)
        addl    r19,r8,r8
        cmple   r4,r6,r19
        stl     r8,-4(r3)
        bne     r19,L$21
        cmple   r4,r17,r8
        beq     r8,L$13
        nop
L$24:   ldl     r7,(r5)
        addl    r4,1,r4
        ldl     r19,(r3)
        lda     r3,4(r3)
        cmple   r4,r17,r8
        lda     r5,4(r5)
        addl    r7,r19,r7
        stl     r7,-4(r3)
        bne     r8,L$24
L$13:   addl    r2,1,r2
        cmple   r2,r17,r19
        bne     r19,L$8
L$7:    addl    r0,1,r0
        cmple   r0,r17,r7
        addl    r1,1,r1
        bne     r7,L$5
L$4:    ret     r26
```

图21-17 （续）

21.4 Intel 386体系结构上的Intel参考编译器

21.4.1 Intel 386体系结构

Intel 386体系结构系列包括Intel 386和其后续微处理器，如486、Pentium、Pentium Pro等，

版权公司制作请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

它们都以更快的执行方式实现了本质上相同的指令集[⊖]（参见 [Pent94]）。Intel的这个体系结构是彻底的CISC设计，但某些实现利用了RISC原理，如流水线和超标量，以达到显著改善速度以超过系列中以前成员的目的。它的体系结构在很大程度上受到了要求与早期的Intel处理器，如8086，向上兼容的限制，其中8086只包含了字节和半字数据，以及一个相当难用的分段寻址配置。我们只讨论几种兼容的特征。

Intel 386体系结构有8个32位的整数寄存器，名字为eax、ebx、ecx、edx、ebp、esp、esi和edi。每一个寄存器的低16位另有一个名字，这个名字是8086中的名字，并且只由指令集中的8086子集使用。每一个16位寄存器的名字是32位寄存器名去掉e字母。前4个16位寄存器的每一个又进一步分为两个字节寄存器，如ah和al，它们分别构成了ax的高部和低部。此外，有6个32位的段寄存器，用于计算取、存、分支和调用有关的地址。这些寄存器中有一些有专门的用途，例如ebp和esp，它们分别指向当前栈帧的基址和栈顶，而其他的专门用在某些类别的指令中，如ecx、esi和edi用在操作字符串的指令中。

存储器地址由一个段寄存器（多数情况下它是由指令类型选择的）、一个基寄存器、一个可选的带缩放的（1、2、4、8倍）索引寄存器，以及一个8位或32位的位移组成，其中每一个部分都是可选的（但至少必须有一个部分）。

条件控制是通过比较操作来实现的，这个操作设置一个标志寄存器中的某些位，并根据它们进行分支转移。

该体系结构包括的指令有数据传送（在寄存器和存储器之间、寄存器与寄存器之间），逻辑、移位和旋转操作，条件和无条件转移，调用和返回，循环控制，字符串管理，过程入口和出口，浮点，字节转换，字节交换和系统控制。

浮点程序设计模式包括8个80位的浮点寄存器。浮点格式有32位的单精度、64位的双精度和80位的扩展精度。所有数据一旦被取到寄存器便被转换成扩展精度，并且当存储时可以转换回原来的精度。除了浮点取和存、算术和比较运算，以及转换之外，有取7个特殊常数（例如π）中任意一个常数的指令，以及执行三角、指数、对数运算的指令。

Intel体系结构的多数指令有两个操作数，尽管也有相当的指令只有一个或没有操作数。在较典型的两操作数情况下，第一个操作数一般作为第一个源操作数，第二个操作数既是第二个源操作数，也是目的操作数，并且在汇编语言中也按此顺序书写。所允许的操作数类型随指令的不同而不同，但在多数情况下，一个操作数可以是存储器地址、寄存器或常数，而另一个可以是常数或寄存器。关于其汇编语言的进一步细节请参见附录A.4。

733 ~ 734

21.4.2 Intel编译器

Intel 为386体系结构系列提供了称之为参考编译器（reference compiler）的C、C++、Fortran 77和Fortran 90编译器。

这些编译器的结构采用混合方式的优化组织，其结构如图21-18所示。每一个编译器由一个特定语言的前端（来源于Multiflow和Edison Design Group）、过程间优化、存储优

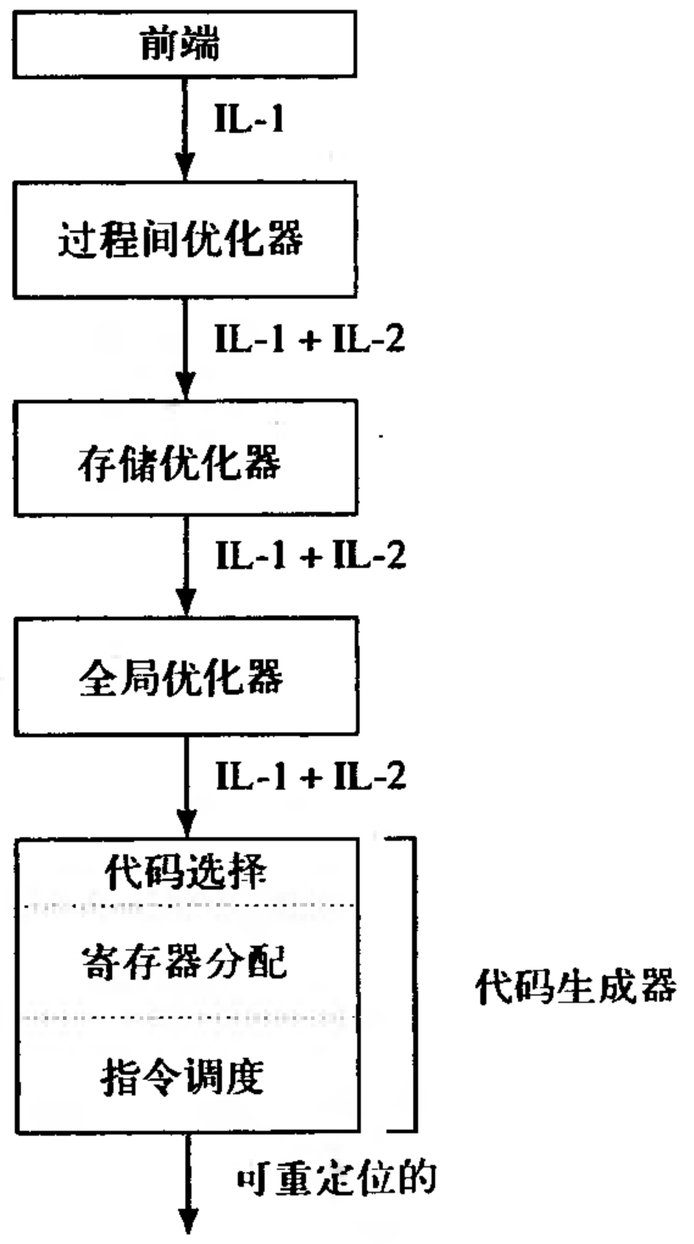


图21-18 Intel参考编译器结构

⊖ Pentium Pro实际上有一些新的指令，包括整数和浮点条件传送，以及一个设置整数条件码的浮点比较指令。

bbs.theithome.com

化、全局优化以及一个代码生成器组成，其中代码生成器由代码选择、寄存器分配和指令调度三遍组成。过程间优化和存储优化是1991年增加进来的，同时还有重新设计的一个代码生成器，称为Proton，其目的是为了增加Pentium处理器和它的下一代处理器的优化范围。从那时起，全局优化器就基于部分冗余代码删除广泛地进行了重写。

前端产生称为IL-1的中级中间代码，它包括某些值得注意的特征，如数组索引，它由基于指针的数组遍历重新生成，其形式与在C和C++中可以出现的形式相同。图21-19给出了图21-2中主程序对应的IL-1代码（由Fortran前端产生）作为例子。其中一些运算符的含义如下:

1. 指令1中的ENTRY表示过程的入口。

2. 指令3、9、25和29中的SSTORE表示存储一个整数到存储器。

3. 指令20中的VOGEN和指令21中的ASTORE分别表示生成一个数组下标和存储到索引所指的数组位置。

4. 指令5和11中的LOOP_BEGIN表示一个循环的开始。

5. 在第7行和13行的IF_REL.LE表示一个循环的关闭测试。

```
Entry bblocks: 0(MAIN)

BBLOCK 0:  (an entry bblock), preds: , succs: 1, stats:
    1 ENTRY.ARGS_REGS.ENT_GLOBAL
    3 SSTORE.SI32 5 1(SI32) __1.MAIN.k
BBLOCK 1:  preds: 0 6, succs: 4 2, stats:
    5 LOOP_BEGIN 5 500
    6 SLOAD.ND.NREG.SI32 5 t0 __1.MAIN.k
    7 IF_REL.LE.SI32 5 99% 0% t0 500(SI32)
BBLOCK 2:  preds: 1, succs: 3, stats:
    31 CALL.ARGS_REGS.CALLER_SAVES s1
BBLOCK 3:  preds: 2, succs:  , stats:
    33 RET.Sr
BBLOCK 4:  preds: 1, succs: 5, stats:
    9 SSTORE.SI32 6 1(SI32) __1.MAIN.l
BBLOCK 5:  preds: 4 7, succs: 7 6, stats:
    11 LOOP_BEGIN 6 500
    12 SLOAD.ND.NREG.SI32 6 t1 __1.MAIN.l
    13 IF_REL.LE.SI32 6 99% 0% t1 500(SI32)
BBLOCK 6:  preds: 5, succs: 1, stats:
    27 SLOAD.ND.NREG.SI32 5 t10 __1.MAIN.k
    28 GADD.SI32 5 t11 2 [t10,1(SI32)]
    29 SSTORE.SI32 5 t11 __1.MAIN.k
BBLOCK 7:  preds: 5, succs: 5, stats:
    15 SLOAD.ND.NREG.SI32 7 t2 __1.MAIN.k
    16 SLOAD.ND.NREG.SI32 7 t3 __1.MAIN.l
    17 GADD.SI32 7 t4 2 [t2,t3]
    18 SLOAD.ND.NREG.SI32 7 t5 __1.MAIN.k
    19 SLOAD.ND.NREG.SI32 7 t6 __1.MAIN.l
    20 VOGEN.2 7 t7 __1.MAIN.a 2
    21 ASTORE.2.SI32 7 t4 __1.MAIN.a t7 2
    23 SLOAD.ND.NREG.SI32 6 t8 __1.MAIN.l
    24 GADD.SI32 6 t9 2 [t8,1(SI32)]
    25 SSTORE.SI32 6 t9 __1.MAIN.l
```

图21-19 图21-2中主程序由Fortran前端产生的IL-1形式

结果操作数第一个出现在运算符之后，后面是源操作数。注意SI32限定符使得操作数是

bbs.theithome.com

超星公司制作 请尊重作者版权

32位整数，IF_REL运算符包含两个分支的预期执行频率。

只有一个可选择的全局优化级别（不包括不优化），过程间优化和存储优化使用另外独立的选项。

过程间优化器跨文件进行优化（通过保存每一个过程的IL-1形式），并可以由执行剖面分析的结果来驱动。它执行一系列的分析，这些分析收集与过程使用有关的信息，以及它们的某些特征信息，如它们的大小、常数参数、以及模块级静态变量的使用。过程间优化器然后执行一系列的优化，包括内联、过程克隆、参数替换和过程间常数传播（参见19.3节）。指导内联的是过程是否在循环内被调用、循环体的代码大小，以及是否出现有常数值参数，并且既考虑到有利于存储优化又考虑了有利于减少调用开销。过程克隆创建过程的各种副本，这些副本具有不同的常数参数，因此有可能使得循环和数组的边界是已知常数。参数替换追踪常数值实参数，并传播它们到使用它的哑参数。过程间优化器也能够判别出传递给特定过程的实参在寄存器中而不是在运行栈中（UNIX除外，它的应用程序二进制接口要求将它们传递在栈中）。

过程间优化器的输出是一种较低级的IL-1版本，叫做IL-2，它带有IL-1的程序结构信息，这种中间形式用于编译器余下的主要部分，直到用作代码生成器的输入。图21-2中主程序的最内层循环的IL-2形式（在优化之后）如图21-20所示。结果操作数在前，之后是运算符，其后是源操作数。注意，多数运算符的操作级别都降低了，下标已经扩展成地址计算（尤其注意第23行），并且已执行了循环倒置和代码外提。

735 ~ 737

```
BBLOCK 7:  preds: 7 4, succs: 7 9, stats:
    26 LOOP_BEGIN 6 500
    30 ADD.SI32 7 t6 t4 t5
    25 ASSIGN.N32 7 t14 500(SI32)
    29 IMUL.SI32 7 t7 t14 t5
    24 ADD.SI32 7 t8 t4 t7
    23 SST.SI32 7 t6 t8 (addr(__1.MAIN.a)(P32) - 2004(SI32))(P32) __1.MAIN.a
    21 ADD.SI32 6 t5 1(SI32) t5
    2 IF_REL.GE.SI32 6 99% 0% 500(SI32) t5
```

图21-20 提交给代码生成器的图21-2中主程序的内层循环的IL-2形式

过程内数据流分析作为存储优化或全局优化的第一步，这两种优化谁先进行取决于所选择的编译选项。它包括将循环转换成规范形式和表示它们的嵌套结构。

存储优化器关注于改善存储器和高速缓存的使用，执行的几乎全是循环转换。它首先执行基于SSA的稀有条件常数传播（参见12.6节），然后用Banerjee测试[Bane88]对已知边界的循环进行数据依赖关系分析。在依赖关系分析之前，先运行称为“循环净化”（loop cleanup）的一遍,尝试使得循环嵌套成为理想的嵌套，并使其边界和跨步成为已知常数。可施加的转换有循环交换、循环分布、循环分块（strip mining）、软预取、铺砌、以及可选循环（alternate loop）的创建。其中最后一种方法处理这种循环，如果它们的循环体满足某种依赖（或不依赖）关系，则可执行重要的优化，但是判别这种依赖关系的条件所依赖的信息只在运行时才能得知。瓦片循环的大小用Lam、Rothberg和Wolf [LamR91]开发的技术进行选择。存储优化器计算的迭代距离被传递给代码生成器控制循环展开和代码调度。注意，由过程间优化器执行的过程内联和过程克隆导致更多的循环可通过依赖关系测试算法，从而增加了存储优化的有效性。

全局优化器做一系列的数据流分析和关于边界已知循环的Banerjee数组依赖关系测试，后者识别访问存储器的顺序限制。数据流分析采用Lengauer和Tarjan的方法（参见7.3节）通过确定必经结点和回边来进行。别名分析假定所有指针引用会相互冲突，但Fortran的参数引用除外，

bbs.theithome.com

超星公司制作 请尊重作者版权

因为该语言标准规定如此。全局优化器执行的优化依次是:

1. 提升局部和文件级的静态变量作为寄存器分配的候选;
2. 常数传播;
3. 死代码删除;
4. 局部公共子表达式删除;
5. 复写传播;
6. 部分冗余删除;
7. 第二遍复写传播;
8. 第二遍死代码删除。

代码生成器叫做Proton，它使用自己的中间形式，称为Proton IL（PIL）。直接转换图21-20
738 的IL-2得到的PIL代码如图21-21所示。它的操作是三元式，其中，源操作数在运算符之后，结果用三元式编号来表示。例如，第3行加第2行和第1行的结果，第7行存储第3行的结果于指定的存储器位置，此位置以三元式6的结果为索引并具有一个由三元式4的结果给出的偏移量。imerge运算符是一个SSA形式的ϕ函数。

```
esp based stack
Stack frame size: 8

BLOCK=3 Phys_pred=2 Phys_succ=4 Loop=2
    CFlow_preds= B3 B2
    CFlow_succs= B3 B4
B3  opcode    op1     op2
1   imerge    B2.1    8
2   imerge    B1.3    B4.2
3   add       2       1
4   movi      $500
5   imul      4       1
6   add       2       5
7   st        3       ..1.MAIN.LOCALSTATIC.a-2004(6,4)
8   addi      1       $1
9   movi      $500
10  cjge      9       8       B3 p70% m0%
```

图21-21 图21-2中主程序的内层循环从图21-20的IL-2形式转换到PIL形式之后

Proton执行指令选择、寄存器分配、指令调度和一系列的低级优化，这4个任务合在一起成为一个重要的部分。首先做的是形成CISC形式的地址。这是一个不那么简单的工作，因为地址可以由基地址寄存器、变址寄存器、缩放因子以及位移组成，并且在该体系结构较为先进的实现中，地址计算可以与其他操作并行执行。优化是通过沿du链应用窥孔优化来进行的。归纳变量优化也在代码选择之前进行，以便形成与体系结构对应的代码。

指令选择相对较直接，但指令归并除外。指令归并是由少数可用的寄存器以及在Pentium及其后继处理器上并行运行指令的可能性来驱动的。例如，存储器与存储器加和取-加-存序列在Pentium上都需要3个时钟周期，但是后者在双流水的实现中更有可能与其他指令组成对。作为另一个例子，PIL将一字节符号扩展操作表示成一个取后随一个左移24位和一个右移24位，但是使用这个体系结构的带符号扩展的取指令在某些情况下需要的时钟周期更少。类似地，生成包括一个基址、一个变址和一个位移的有效地址，可以如前面指出的，用一条没有结果寄存
739 器的指令，但是也可以用将两个寄存器相加并将结果放至一个寄存器的指令——如果这个地址

bbs.theithome.com

使用多次，并且如果有一个寄存器可以用来存放它，这种方法可能就更划算。

寄存器分配采用基本块内的局部方法与Chaitin风格的跨基本块的图着色方法相结合的方式。代码生成根据给定程序的循环结构将寄存器划分为局部的和全局的。具有高执行频率（预期的或实测的）循环的程序分配给局部使用的寄存器较多，并且寄存器分配是从循环的最内层由里向外进行。

8个浮点寄存器组成的栈结构限制了寄存器分配的效率，因为为了使得它们的栈相匹配，需要进行路径合并。但是，在Pentium和Pentium Pro中，fxch（浮点寄存器内容交换）指令可以在V流水线以0结果延迟执行，因此至少在基本块内可以将浮点栈看成是一个寄存器集合。在486处理器上情况则不同，它的fxch需要4个时钟周期，将浮点栈作为等价的寄存器集合是十分不利的。

整数寄存器的全局分配似乎有很大的不确定性，因为只有8个32位的寄存器，且它们之中的若干个还有专门的用途。但是这些32位的寄存器中有4个其低部16位可作为一对字节寄存器，因此增加了可用寄存器个数；另外，只有栈指针寄存器是固定专用的，不能用于其他目的。此外，尽管8个寄存器肯定不如32个更有用，但全局寄存器分配研究已证明8个寄存器毕竟比1个或2个寄存器要好得多。

指令调度采用表调度方法并包括局部寄存器分配，其风格很像Goodman和Hsu [GooH88]开发的方法。它基本上每次只处理一个基本块，尽管在有限程度上也考虑了前驱和后继基本块。调度主要是表驱动的，因为其体系结构的不同实现在流水线组织和指令时间上有很大的不同。例如，386完全没有流水线，而Pentium有双流水线，Pentium Pro具有消除了相互间影响且具有乱序执行能力的多个流水线。

代码生成器中执行的低级优化包括如下：

1. 归纳变量优化，如前面提到的，优化寻址模式和寄存器的使用，强度削弱，线性函数测试替换，包括用移位和加法替代乘法。

2. 机器方言，沿du链寻找加1和减1指令并用自增和自减指令替代它们。

3. 将循环对齐在高速缓存的边界。

4. 转换对半字操作数运算的所谓的前缀指令到对应的字节或字形式，这种形式在Pentium和较新的实现上执行得更快。

5. 代码重选择，在寄存器压力希望这样做或迫使它必须这样做的地方，用基于存储器的操作数替换寄存器与寄存器操作的指令序列。

6. 沿窗口调度线进行软流水（参见17.4节）。

7. 重结合，将循环不变量运算集中到一起并从包含它们的循环中移出。

8. 循环展开。

9. 伸直化和基本块重排，以允许短分支指令。

代码生成器包括一个为UNIX系统产生位置无关代码的选项，并且公共子表达式删除同处理其他表达式一样，也作用于GOT引用（参见5.7节）。

图21-1中C程序例子的Pentium汇编代码如图21-22所示。注意，kind的常数值已传播到条件中，并且已删除了死代码，循环不变量length*width也已从循环内移出，乘以height的乘法已强度削弱为加法，局部变量已分配到寄存器中，指令调度也已执行。但另一方面，循环没有展开，没有优化对process()的尾调用，并且对area的累加本来应当转换成单一乘法指令。

bbs.theithome.com

```
main: .B1.1:
        pushl     %ebp
        movl      %esp,%ebp
        subl      $3,%esp
        andl      $-8,%esp
        addl      $4,%esp
        pushl     %edi
        pushl     %esi
        pushl     %ebx
        subl      $8,%esp
        movl      length,%esi
        xorl      %ebx,%ebx
        imull     width,%esi
        movl      %esi,%edi
        xorl      %ecx,%ecx
        movl      $-10,%edx
        xorl      %eax,%eax
.B1.2:  addl      %esi,%ecx
        addl      %eax,%ebx
        addl      %edi,%eax
        incl      %edx
        jne       .B1.2
.B1.3:  movl      %ecx,(%esp)
        movl      %ebx,4(%esp)
        call      process
.B1.4:  xorl      %eax,%eax
        popl      %edx
        popl      %ecx
        popl      %ebx
        popl      %esi
        popl      %edi
        movl      %ebp,%esp
        popl      %ebp
        ret
```

图21-22 由Intel参考编译器产生的图21-1中C程序的Pentium汇编代码

图21-23和图21-24所示的Pentium汇编代码是Fortran 77编译器在过程间优化、存储优化和全局优化都起作用的情况下，由图21-2中的例程序产生的，但我们省略了主程序中做初始化的那个循环，以及s1()中除最内层循环之外的其他所有部分。因为s1()已被内联，编译器可以利用n的值是500的信息，从而使得它可在循环控制中使用常数值。最内层循环（开始于.B2.9）没有被展开，但对它执行了线性函数测试替换。局部变量都已分配到寄存器中，除最内层循环之外，其他地方的代码完全是CISC风格的。

741～743

但是，该编译器为s1()和主程序都产生了代码，尽管这是不必要的——主程序显然只调了s1()，而s1()没有调其他过程。另外，对同样的循环嵌套而言，为s1()生成的代码和为主程序生成的代码之间存在一些有意思的差别。在主程序的代码中，循环没有被展开，并且是CISC风格；而在子程序的代码中，循环已用因子4展开（从标号.B1.7开始），并且完全是RISC风格的代码。注意到该循环除了展开的版本外还有一个卷起的版本(开始于标号.B1.18)，可以看出它没有做过程间常数传播。根据[Sava95]的解释，这种情形是因为做了过程内联而没有创建s1()的克隆。另外，该编译器根据在双流水线中形成成对指令的机会在CISC和RISC风格的代码生成之间进行选择；如果不能成对，则生成CISC风格的指令。展开子程序中的最内层循环创造了成对组成指令的机会，因此产生了RISC风格的代码。

bbs.theithome.com

```
s1: .B1.1:
          . . .
.B1.7:    movl      -2004(%ecx),%edx
          movl      -2004(%eax),%ebx
          addl      %ebx,%edx
          movl      %edx,-2004(%eax)
          movl      -2000(%ecx),%edx
          movl      -2000(%eax),%ebx
          addl      %ebx,%edx
          movl      %edx,-2000(%eax)
          movl      -1996(%ecx),%edx
          movl      -1996(%eax),%ebx
          addl      %ebx,%edx
          movl      %edx,-1996(%eax)
          movl      -1992(%ecx),%edx
          movl      -1992(%eax),%ebx
          addl      $16,%ecx
          addl      %ebx,%edx
          movl      %edx,-1992(%eax)
          movl      16(%esp),%edx
          addl      $16,%eax
          cmpl      %edx,%eax
          jle       .B1.7
.B1.17:   cmpl      %ebp,%eax
          jg        .B1.8
.B1.18:   movl      -2004(%ecx),%edx
          movl      -2004(%eax),%ebx
          addl      %ebx,%edx
          addl      $4,%ecx
          movl      %edx,-2004(%eax)
          addl      $4,%eax
          cmpl      %ebp,%eax
          jle       .B1.18
.B1.8:    movl      8(%esp),%ebx
          . . .
```

图21-23 由Intel参考编译器产生的图21-2中Fortran 77程序的子程序s1()的Pentium汇编代码

```
MAIN: .B2.1:
          pushl     %esi
          pushl     %ebp
          pushl     %ebx
          movl      $2000,%ebx
          movl      $1,%ebp
          . . .
.B2.5:    movl      $1,%esi
          movl      $500,%ebp
.B2.6:    leal      1(%esi),%eax
          cmpl      $500,%eax
          jg        .B2.12
.B2.7:    movl      %eax,%edx
          shll      $2,%edx
          subl      %eax,%edx
          leal      (%eax,%edx,8),%eax
          leal      (%eax,%eax,4),%ebx
          shll      $4,%ebx
```

图21-24 由Intel参考编译器产生的图21-2中Fortran 77程序的主程序的Pentium汇编代码

bbs.theithome.com

```
.B2.8:   movl    $-500,%eax
         movl    %ebp,%ecx
         shll    $2,%ecx
.B2.9:   movl    ..1.MAIN.LOCLSTATC.a.1.0(%ecx,%eax,4),%edx
         addl    %edx,..1.MAIN.LOCLSTATC.a.1.0(%ebx,%eax,4)
         incl    %eax
         jne     .B2.9
.B2.10:  addl    $2000,%ebx
         cmpl    $1000000,%ebx
         jle     .B2.8
.B2.12:  incl    %esi
         addl    $500,%ebp
         cmpl    $500,%esi
         jle     .B2.6
.B2.13:  popl    %ebx
         popl    %ebp
         popl    %esi
         ret
```

图21-24 （续）

21.5 小结

表21-1中总结了这4个编译器中每一个关于第一个例程序（C）的特征，表21-2中总结了它们关于第二个例程序（Fortran）的特征。

表21-1 4个编译器关于C例程序的比较

	Sun SPARC	IBM XL	DEC GEM	Intel 386系列
kind的常数传播	yes	yes	yes	yes
死代码删除	almost all	yes	yes	yes
循环不变代码外提	yes	yes	yes	yes
height的强度削弱	yes	yes	no	yes
area计算的归约	no	no	no	no
循环展开因子	4	2	5	none
卷起的循环	yes	yes	no	yes
寄存器分配	yes	yes	yes	yes
指令调度	yes	yes	yes	yes
已删除栈帧	yes	no	no	no
已优化尾调用	yes	no	no	no

表21-2 4个编译器关于Fortran例程序的比较

	Sun SPARC	IBM XL	DEC GEM	Intel 386系列
a(i)的公共子表达式地址	yes	yes	yes	yes
s1()过程集成	yes	no	no	yes
循环展开因子	4	2	4	none
卷起的循环	yes	yes	yes	yes
最内层循环的指令	21	9	21	4
线性函数测试替换	no	no	no	yes
软流水	yes	no	no	no
寄存器分配	yes	yes	yes	yes
指令调度	yes	yes	yes	yes
删除子程序s1()	no	no	no	no

bbs.theithome.com

超星公司制作 请尊重作者版权

21.6 编译器设计和实现未来的趋势

关于高级编译器设计和实现的未来，有几个显然的主要趋势:

1. SSA形式将被越来越多的优化采用，主要是因为它允许原来设计作用于基本块或扩展基本块的方法可以作用于整个过程，并且它一般能导致性能有显著的额外改善。

2. 部分冗余删除会使用得更频繁，部分因为它的现代版本非常高效，而且也比原来的形式效果好得多，另外也因为它的数据流分析为人们构思其他优化和执行其他优化提供了一个基础。

3. 诸如SSA形式和部分冗余删除的技术结合到一起将产生能改善其能力、效率和/或速度的优化版本。

4. 某些面向标量的优化，包括我们涉及到的大部分优化，将与并行化和向量化更为紧密地集成到一起产生并行编译系统。

744 ~ 745

5. 数据依赖测试、高速缓存优化和软流水在下一个十年都将会有显著的进展。

6. 在标量编译中最活跃的研究领域是并且仍将继续是优化。

所有这些趋势的例子都可以在关于程序设计语言实现年会所发表的论文中看到。

21.7 进一步阅读

正式介绍本章所讨论的处理器体系结构的技术资料如下:

体系结构	文献
SPARC Version 8	[SPAR92]
SPARC Version 9	[WeaG94]
POWER	[POWE90]
PowerPC	[Powe93]
Alpha	[Alph92]
Intel 386 family	[Pent94]

已出版的介绍这些编译器套件的有关文献是:

编译器	文献
Sun SPARC compilers	[Much88]
Digital's GEM compilers	[BliC92]
Intel 386 family reference compilers	[Inte93]

不幸的是，没有已出版的关于IBM XL编译器的更深入的介绍，尽管[BerC92]、[GolR90]、[OBrH90]和[Warr90]介绍了它的某些方面。IBM XL编译器和它的中间语言XIL和YIL是在[OBrO95]中讨论的。[BerG89]集中讨论了寄存器分配器，[Warr90]、[GolR90]和[BerC92]集中介绍了指令调度。

AT&T的C++ 规范是[EllS90]。IBM的PL.8语言的概要介绍是在[AusH82]中找到的。

IBM 801 RISC系统的有关介绍见[Radi82]和[Hopk87]。

Banerjee的关于已知边界循环的数组依赖关系测试，其介绍见[Bane88]。Banerjee-Wolfe测试是在[Wolf89b]中找到的。

Sun的编译和操作系统对动态连接支持的描述是在Gingell等人[GinL87]的论文中找到的。

GEM编译器中的数据流分析利用了Reif和Lewis在[ReiL77]和[ReiL86]中介绍的符号执行方法。GEM代码生成器是根据PDP-11 BLISS 编译器[WulJ75]中的思想设计的。

UNIX系统V版本4 关于Intel 386 系列的ABI增补是[UNIX93]。Intel编译器选择循环瓦片大小使用了由Lam、Rothberg和Wolf开发的技术（参见[LamR91]和[WolL91]）。Goodman和Hsu的将指令调度与局部寄存器分配结合的技术，其介绍见[GooH88]。

746

bbs.theithome.com

版权所有 请勿制作公司盗版

bbs.theithome.com

越是公句制作请尊重作者版权

附录A　本书使用的汇编语言指南

在这个附录中，我们对本书例子中用到的每一种体系结构的汇编语言给出一个简要的描述。这些描述并不是汇编语言的手册——它们提供的信息仅够阅读我们的例子。

A.1　Sun SPARC V8和V9汇编语言

在SPARC汇编语言中，一条指令由一个以冒号结束的可选标号域、一个操作码、以逗号分隔的操作数，以及一个以惊叹号开始的可选注释域组成。目标操作数是最后一个操作数。取和存指令中的地址书写为中括弧内的一个寄存器与另一个寄存器或一个位移之和。寄存器操作数可以是表A-1所示的形式。寄存器r0（等同于g0）是特殊的：当它作为操作数时产生值0，并忽略写给它的结果。操作符%hi()和%lo()分别抽取其操作数的高22位和低10位。

表A-1　SPARC寄存器操作数形式

名　字	含　义
%r*i*	整数寄存器*i*，0 ≤ *i* ≤ 31
%g*i*	全局整数寄存器*i*，0 ≤ *i* ≤ 7
%i*i*	In整数寄存器*i*，0 ≤ *i* ≤ 7
%l*i*	局部整数寄存器*i*，0 ≤ *i* ≤ 7
%o*i*	Out整数寄存器*i*，0 ≤ *i* ≤ 7
%f*i*	浮点寄存器*i*，0 ≤ *i* ≤ 31
%sp	栈指针（%o6）
%fp	帧指针（%i6）

例子中用到的操作码列出在表A-2中。其中有些操作码是机器指令集的扩充——例如，ld可以产生一条整数取指令或者一条浮点取指令，取决于其操作数的类型。完成符*a*（它的值可以是“,a”或省略）作废分支延迟槽。在操作数表的开头带有“i,”或“x,”的分支指令分别根据32位和64位条件码进行分支。

表A-2　正文中用到的SPARC操作码

名　字	操　作
add	加
ba,*a*	总是分支
bg,*a*	大于分支
bge,*a*	大于或等于分支
bl,*a*	小于分支
ble,*a*	小于或等于分支
bne,*a*	不等于分支
call	调用
cmp	比较
faddd	双精度浮点加
fadds	单精度浮点加

bbs.theithome.com

越是公司制作请尊重作者版权

（续）

名　字	操　作
fdivs	单精度浮点除
fdtoi	转换双精度到整数
fitod	转换整数到双精度
fmuld	双精度浮点乘
fsubs	单精度浮点减
iprefetch	指令预取（仅SPARC－V9）
ld	取字
ldd	取双字
ldf	取浮点字
ldh	取半字
mov	传送
move	相等条件传送
nop	空操作
or	或
restore	恢复寄存器窗口
ret	返回
save	保护寄存器窗口
sethi	设置高22位
sll	逻辑左移
smul	有符号乘
st	存字
std	存双字
stf	存浮点字
sub	减
subcc	减并设置条件码
umul	无符号乘
unimp	未实现的

SPARC的伪操作以点打头。例子中用到的伪操作给出在表A-3中。

尽管SPARC-V9用许多向上兼容的方式扩充了V8，在多数情况下这与我们的例子没有关系。

表A-3　SPARC的伪操作

名　字	含　义
.align	设置对齐（按字节）
.data	切换到数据段
.double	双字常数
.end	内嵌模板的结束
.global	全局符号
.seg	段切换
.template	内嵌模板开始
.text	切换到正文段
.word	字常数

A.2　IBM POWER和PowerPC汇编语言

在POWER和PowerPC汇编语言中，一条指令由一个以冒号结束的可选标号域、一个操作

bbs.theithome.com

版权所有 侵权必究

码、以逗号分隔的一组操作数，以及一个以美元符开始的可选注释域组成。目标操作数是第一个操作数。在取和存指令中的地址书写为一个位移后随一个括弧内的基寄存器，或者一个变址寄存器后随一个用逗号分隔的基寄存器。通用寄存器操作数是范围在0到31之间的一个数字，或者字母r后随这样一个数字；每一个寄存器的类型由操作码区别。寄存器r0是特殊的：在地址计算中，当它作为地址计算的操作数时产生值0。形如CR*n* 的操作数，其中$0 < n < 7$，表示一个条件寄存器；CR0可以由其操作码用一个点结束的任何整数指令设置。比较指令可以设置任何条件寄存器，分支指令可以对条件寄存器中的任意一个进行测试。

747 ~ 749

寄存器SP和RTOC分别是栈指针和指向全局对象表的指针。

例子中用到的操作码列出在表A-4中。PowerPC中已删除了POWER的零或差指令（doz）。

表A-4 正文中用到的POWER和PowerPC的操作码

POWER名字	PowerPC名字	操 作
a	addc	加
ai	addic	加直接数
b	b	无条件转移
bbt	bbt	条件寄存器位为真分支
bc	bc	条件分支
bcr	bcr	条件寄存器分支
bl	bl	分支并连接
cal	addi	计算地址低部
cmp	cmp	比较
cmpi	cmpi	比较直接数
cror	cror	条件寄存器或
doz	---	差或零
fa	fadd	浮点加
l	lwz	取
lbz	lbz	取字节和0
lhau	lhau	更新地址取半字并代数扩展
lu	lwzu	更新地址取
mfspr	mfspr	从特殊寄存器传送
mtspr	mtspr	传送至特殊寄存器
muls	mullw	短乘
rlinm	rlwinm	循环左移直接数并屏蔽
st	stw	存
stu	stwu	更新地址存

A.3 DEC Alpha汇编语言

在DEC Alpha汇编语言中，一条指令由一个以冒号结束的可选标号域、一个操作码、以逗号分隔的一组操作数，以及一个以分号开始的可选注释域组成。目标操作数是最后一个操作数。在取和存指令中的地址书写为一个位移后随一个括弧内的基寄存器。整数寄存器操作数可以是表A-5所示的形式。寄存器r31是特殊的：当它作为操作数时产生值0，并忽略写给它的结果。

750

例子中用到的操作码列出在表A-6中。其中有些操作码是机器指令集的扩充——例如，clr和mov两者都可以做逻辑或操作。

bbs.theithome.com

记住，DEC的“长字”和“4字”分别是我们的“字”和“双字”。

表A-5 Alpha整数寄存器名

名 字	含 义
r*i*	整数寄存器*i*，0 < *i* < 31
sp	栈指针（r30）
gp	全局指针（r29）

表A-6 正文中用到的Alpha操作码

名 字	操 作
addl	长字加
addq	4字加
beq	寄存器等于0分支
bis	逻辑或
ble	寄存器小于或等于0分支
blt	寄存器大于或等于0分支
bne	寄存器不等于0分支
clr	清寄存器
cmple	有符号的长字小于或等于比较
cmplt	有符号的长字小于比较
cvttq	转换IEEE浮点到整数
insbl	插入字节低部
jsr	转移到子程序
lda	取一个地址
ldah	取地址高部
ldl	取长字
ldq	取4字
ldq_u	取未对齐的4字
mov	寄存器传送
mskbl	屏蔽字节低部
mull	长字乘
nop	空操作
ret	从子程序返回
s4subq	用4放大4字然后减
sll	逻辑左移
stl	存长字
stq	存4字
stq_u	存未对齐的4字
subq	4字减

A.4 Intel 386体系结构汇编语言

在Intel 386体系结构系列的汇编语言中，一条指令由一个以冒号结束的可选标号域、一个操作码、以逗号分隔的一组操作数，以及一个以分号开始的可选注释域组成。指令可以有0~2个操作数，取决于操作码和用法（例如，返回指令ret可以有一个操作数，但它是可选的）。对于两操作数指令，第二个操作数通常既是第二操作数，也是目标操作数。

bbs.theithome.com

越是公司制作请尊重作者版权

存储器地址书写为一个位移后随一个括弧内的表；表由一个基寄存器后随一个变址寄存器以及一个缩放因子（它取决于变址寄存器）组成，其中每一项都是可选的，但如果没有出现变址寄存器，就不能有缩放因子。整数寄存器操作数是一个百分号后随寄存器名。常数以美元符号开头。

整数寄存器名给出在表A-7中。为了与较早的Intel处理器兼容，某些32位寄存器有16位的子寄存器，而且还有8位的子寄存器。虽然这些寄存器中有6个是通用的，但在某些情况下，其中有一些按指定的方式由特定的指令来使用，例如用于字符串操作的ecx、esi和edi。

表A-7 Intel 386体系结构整数寄存器名

32位的名字	16位的名字	8位的名字	用 法
eax	ax	al,ah	通用寄存器
ebx	bx	bl,bh	通用寄存器
ecx	cx	cl,ch	通用寄存器
edx	dx	dl,dh	通用寄存器
ebp	bp		基（或帧）指针
esi	si		通用寄存器
edi	di		通用寄存器
esp	sp		栈指针

751 ~ 752

浮点寄存器的长度是80位并形成了一个栈。它们的名字从%st(0)（或仅为%st）到%st(7)。典型的指令（如加）可有如下形式:

```
fadd    %st(2), %st
fadd    1.0
```

其中的第一条指令相加%st和%st(2)的内容，并将结果放置在%st中，第二条指令将1.0加到%st中。

例子中用到的操作码列出在表A-8中。所有整数指令中的后缀“l”指出它们操作的是长（32位）操作数。

表A-8 正文中用到的Intel 386体系结构操作码

名 字	操 作
addl	加
andl	逻辑与
call	调用
cmpl	比较
fadd	浮点加
imull	乘
incl	增加
jg	大于转移
jle	小于或等于转移
jne	不等于转移
leal	取有效地址
movl	传送
popl	退栈
pushl	压栈
ret	返回
shll	逻辑左移
subl	减
xorl	逻辑异或

bbs.theithome.com

超星公司制作 请尊重作者版权

A.5 Hewlett-Packard PA-RISC汇编语言

在PA-RISC汇编语言中，一条指令由一个可选标号域、一个操作码、以逗号分隔的一组操
753 作数，以及一个可选注释域组成。目标操作数是最后一个操作数。在取和存指令中的地址书写为一个位移或一个变址寄存器后随一个括在括弧内的基寄存器。寄存器操作数可以是表A-9所示的形式。操作符LR'和RR'分别抽取它们的操作数或常数的高16位和低16位。

表A-9 PA-RISC寄存器名

名 字	含 义
%r*i*	整数寄存器*i*，0 ≤*i*≤ 31
%fr*i*	浮点寄存器*i*，0 ≤*i*≤ 31
%fr*i*L	浮点寄存器*i*，0 ≤*i*≤ 31，左半部
%fr*i*R	浮点寄存器*i*，0 ≤*i*≤ 31，右半部

例子中用到的操作码列出在表A-10中。其中有些操作码是机器指令集的扩充——例如，COPY实际上是OR的一种子情形，COMB是COMBF和COMBT（分别是比较结果为假分支和比较结果为真分支）的简写。完成符*mod*指出修改取或存指令中的基寄存器；具体地，MA在形成有效地址之后通过给它加上一个位移来修改基寄存器。完成符*cond*指出一个算术（或其他）条件（例如，SDC指出“某个数字进位”），完成符*n*（其值可以是*N*）指出作废分支之后的指令。

表A-10 PA-RISC操作码

名 字	操 作
ADD	加
ADDB	加然后分支
ADDBT	加然后为真则分支
ADDI	加直接数
ADDIB,*cond*	加直接数然后条件分支
ADDIL	加直接数左部
B	分支
BL	分支与连接
BV	引导分支
COMB,*cond*,*n*	比较然后分支
COMBF,*cond*	比较为假则分支
COMCLR	比较并清除
COMIBF, *cond*,*n*	比较直接数然后为假则分支
COPY	复制
DCOR	十进制修正
FLDWS,*mod*	短位移取浮点字
FSTWS,*mod*	短位移存浮点字
LDHS,*mod*	短位移取半字
LDI	取直接数
LDO	取偏移
LDWM,*mod*	取字并更新地址
LDWS,*mod*	短位移取字
LDWX,*mod*	变址取字
MOVE	传送寄存器
NOP	空操作

bbs.theithome.com

超星公司制作 请尊重作者版权

（续）

名　字	操　作
OR	逻辑或
SH1ADD	移1位后相加
SH2ADD	移2位后相加
SH3ADD	移3位后相加
SHD	双字移位
STW	存字
STWM, *mod*	存字并更新地址
SUB	减
XMPYU	无符号定点乘

754 ~ 755

bbs.theithome.com

超星公司制作 请尊重作者版权

bbs.theithome.com

超星公司制作 请尊重作者版权

附录B　集合、序列、树、DAG和函数的表示

选择抽象数据结构的一种适当的具体表示，可以使得一个算法的运行时间从它需要的最大多项式或甚至指数时间降低到线性的或2次的时间。因此如果我们想要将本书或其他地方介绍的算法转变成有效运行的代码，就必须理解数据结构可能的表示、对它们的各种操作所需要的时间以及它们所占用的空间。

这个附录的目的不是提供数据结构的课程，而主要是为了使读者在实现如本书给出的有关算法时，应当考虑到和回忆起的一些有用的数据结构的具体表示。

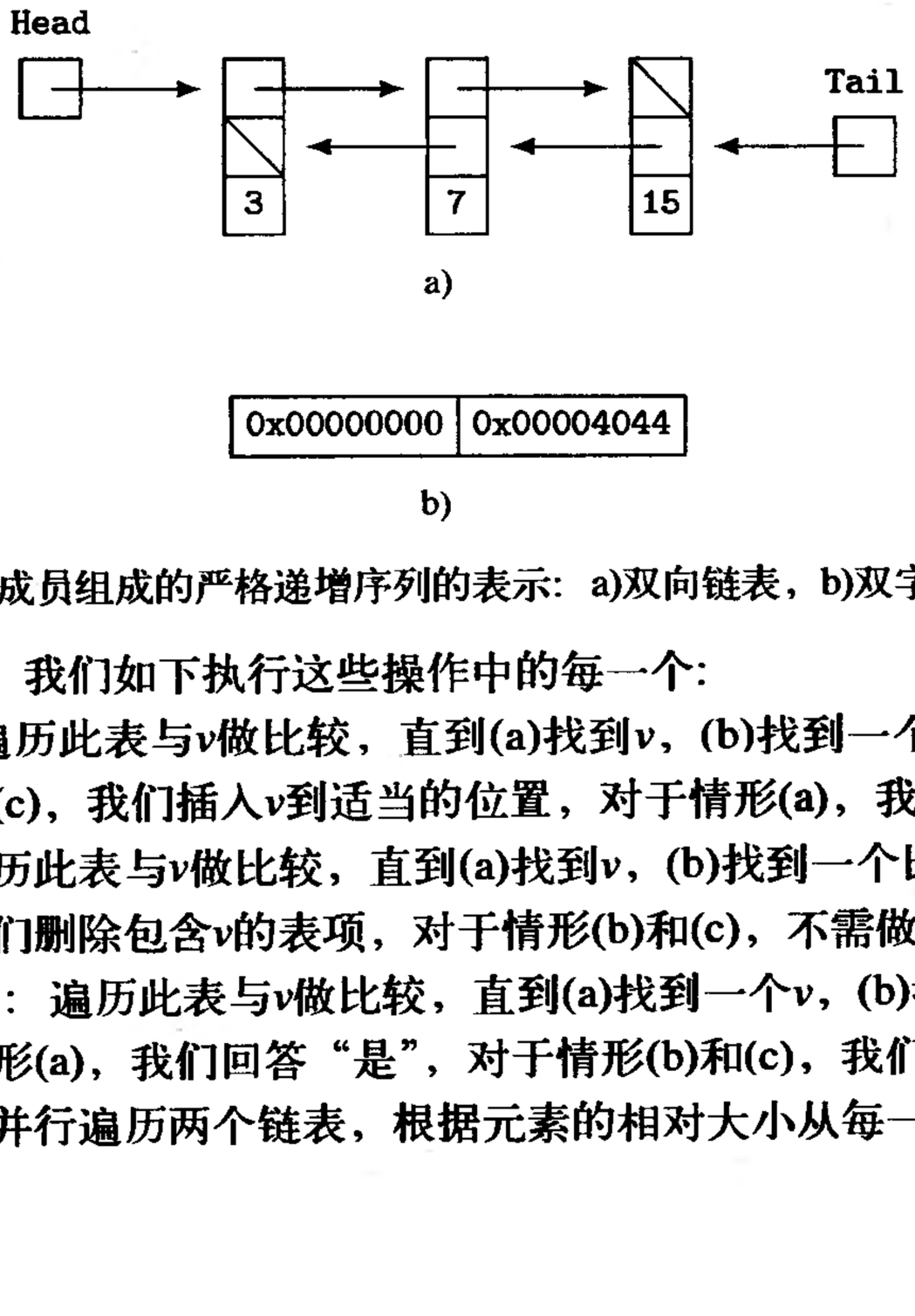
例如，假设我们需要表示一个范围在0至$u-1$之间，且至多有s个成员的严格递增的整数序列，则对于$u=64$，

1, 3, 5, 11, 23, 43, 53, 62

是这种序列，而

3, 1, 15

不是，因为在第二个序列中，第一个成员大于第二个。我们可以用一个双向链表来表示这种序列，其中的表项包含整数；也可以用一个双字的位向量来表示它，其中位置i是1当且仅当i属于该序列，如图B-1a所示。令s是该序列当前具有的全部元素的个数。假设我们需要对这个序列执行的操作是(1)增加一个值，(2)删除一个值，(3)测试一个值的成员关系，(4)合并两个序列和(5)确定序列的长度。



图B-1　由3, 7, 15三个成员组成的严格递增序列的表示：a)双向链表，b)双字位向量(位从右数起)

对于链表实现方式，我们如下执行这些操作中的每一个：

1. 增加一个值v：遍历此表与v做比较，直到(a)找到v，(b)找到一个比v大的值，或者(c)到达链尾。对于情形(b)和(c)，我们插入v到适当的位置，对于情形(a)，我们不需做什么。

2. 删除一个值：遍历此表与v做比较，直到(a)找到v，(b)找到一个比v大的值，或者(c)到达链尾。对于情形(a)，我们删除包含v的表项，对于情形(b)和(c)，不需做什么。

3. 测试v的成员关系：遍历此表与v做比较，直到(a)找到一个v，(b)找到一个比v大的值，或者(c)到达链尾。对于情形(a)，我们回答“是”，对于情形(b)和(c)，我们回答“不是”。

4. 合并两个序列：并行遍历两个链表，根据元素的相对大小从每一个链表中选择元素并合

bbs.theithome.com

并它们到一个表中，在处理中删除重复的元素。

5. 确定序列的长度：遍历此链表，对成员计数。

其中每一种操作需要的时间在最坏的情况下为$O(u)$，在典型的情形下为$O(s)$，并且该序列需要$3 \times s$个字的存储空间。

对于位向量的实现，我们首先构造一个由$u=64$个双字组成的数组mask[1..64]，它的每一个元素有一位被设置为1，即第i个双字中的第i位为1。称此序列的当前值为seq。

我们执行这些操作如下：

1. 增加一个值v：mask[v]与seq求逻辑或。

2. 删除一个值v：mask[v]取反码的结果与seq求逻辑与。

3. 测试成员关系：mask[v]与seq求逻辑与。如果结果非0，回答“是”，否则回答“不是”。

758

4. 合并两个序列：这两个序列求逻辑或。

5. 确定序列的长度：从$v=1$到u，mask[v]与seq求逻辑与，并且计数结果中为1的位的个数。

现在，除了(5)之外的每一种操作都具有常数执行时间，确定长度需要的时间为$O(u)$，并且每一个位向量占两字。

因此，已知要执行上面的一组操作，我们会愿意选择位向量表示，除非确定序列的长度是执行频率非常高的操作。另外注意到通过让其他操作分担确定序列长度的一部分代价，可以使它的速度得到显著的提高，即我们可以保持一个序列长度计数器，并且每当加入或删除一个元素时便对它进行修改。对于位向量表示，当我们合并两个序列时，仍然需要计数它的成员。

另一方面，假设序列允许的范围是$u=1\ 000\ 000$，但是它实际出现的序列决不会多于$s=50$个元素。现在，经过权衡后我们趋向于选择双向链表实现：每一种链表操作至多花费我们50个运算，并且链表的每一个元素至多占用150个字；尽管每一种向量操作所花的时间是$O(u)$，但每一个向量占了$\lceil 1\ 000\ 000/32 \rceil = 31\ 250$个字。

注意两种实现都可以采用动态分配和释放方法，以应付数据结构大小的变化，只要它们是通过指针来访问的。

B.1 集合的表示

同大多数数据结构的情形一样，我们如何选择集合的表示取决于元素的特征、构成集合全集的势、集合的典型规模和最大规模，以及要对它们执行的运算。

如果全集U的值域是u个整数，且对于某个u较可取的值是$0, \cdots, u-1$，则可以有几种比其他表示更简单的表示。如果不是这种形式，则常常有帮助的是通过散列将U的元素映射到值域$0, \cdots, u-1$。

是否需要散列函数是易于逆转的，取决于对集合要执行的运算。例如，如果我们想要合并两个集合，然后枚举这个并集中的成员，可逆的散列函数就是必须的。如果不需要我们枚举集合成员的运算，就不需要可逆性。

集合上的基本运算是并(“$\cup$”)、交(“$\cap$”)、差(“$-$”)、相等(“$=$”)、子集(“$\subset$”)和从属关系(“$\in$”)。某些情况下也可能需要集合的积(“$\times$”)和其他运算。

位向量是一种最容易执行这些基本运算的集合表示，尤其是当$U=\{0, \cdots, u-1\}$，且u相对较小时。大部分数据流分析都使用位向量表示，因为对应的运算较快而且其表示也较紧凑。对

759

于一个有u个元素的U，要求每一个集合占$\lceil u/32 \rceil$个字，其基本运算的执行如下：

bbs.theithome.com

集合操作	位向量操作
$c := a \cup b$	$bv(c) := bv(a)$ or $bv(b)$
$c := a \cap b$	$bv(c) := bv(a)$ and $bv(b)$
$c := a - b$	$bv(c) := bv(a)$ and $!bv(b)$
$t := a = b$	$t := (bv(a)$ xor $bv(b)) \neq \vec{0}$
$t := a \subset b$	$t := (!bv(a)$ and $bv(b)) \neq \vec{0}$
$t := a \in b$	$t :=$ (mask[a] and $bv(b)) \neq \vec{0}$

其中$bv(x)$是x的位向量表示；$\vec{0}$是全为0的位向量；or(或)、and(与)、xor(异或)和“！”(非)是按位逻辑运算；mask[]是一个位串，其中mask[i]的第i位是1，其余所有的位是0。因此，典型运算的执行时间都可以在$O(u)$内，与集合本身的元素个数无关。

链表是表示全集的较小子集合S(大小为s)的一种最容易的表示。一个典型的双向链表如图B-1a所示。S的这种表示的大小是$O(s)$，而不像位向量那样是$O(u)$，但执行基本运算比位向量要困难一些。例如，图B-2给出的计算A和B的并集的代码中，first()、next()和last()遍历链表，append()将它的第二个参数加到它的第一个参数的末尾，它们需要的时间是$O(ab)$，或$O(s^2)$，其中a和b分别是A和B的势。当然，在最坏的情况下这个时间是$O(u^2)$，但我们假定大多数情况下集合A和B的元素个数都大大少于u。为了执行其他运算还需要其他一些类似的例程，这些例程除了测试从属关系的时间是$O(s)$之外，它们都具有类似的时间上界。 760

```
procedure Set_Union(A,B) returns set of U
   A, B: set of U
begin
   S := A: set of U
   x, y: U
   for x := first(B) to last(B) do
      for y := first(A) to last(A) do
         if x = y then
            goto L1
         fi
         y := next(A)
         x := next(B)
      od
      S := append(S,x)
L1:   od
      return S
   end    || Set_Union
```

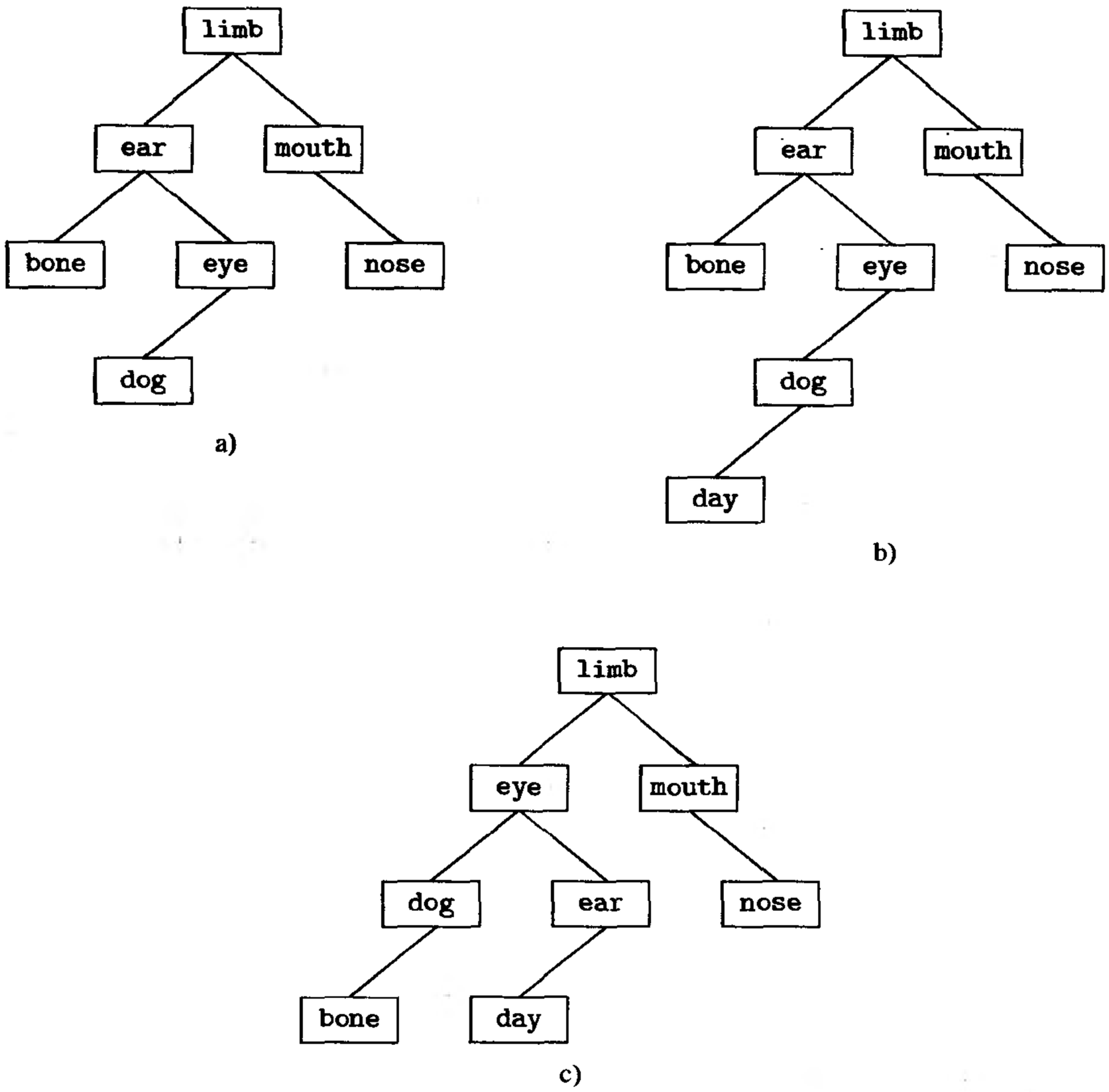
图B-2 计算用链表表示的两个集合的并集的ICAN代码

对于那种元素之间具有易于计算的全序（或需要保持有序）的集合而言，例如符号表中的标识符集合，平衡二叉树是它们的一种重要表示。在这种树中，每一个结点的值大于它的左子树中每一个结点的值，且小于它的右子树中每一个结点的值，并且，从根到一个叶子的最长路径与最短路径至多相差1。图B-3a给出了一个表示标识符集合{bone,dog,ear,eye,limb,mouth, nose}的平衡二叉树的例子。在图b)中，我们加入了一个值day到该集合，使得该树变得不平衡。图c) 展示了使它重新平衡后的结果。注意，由平衡树表示的集合S的从属关系的测试时间可在$O(\log s)$内。计算并、交、差、相等和子集需要的时间是$O(s \log s)$。

散列是另一种重要的集合表示。如通常阐述的，它需要计算一个函数hash()，此函数将集合$S \subseteq U$的元素映射到某个适当的范围0, …, $u-1$内。然后在数组Hash[0..$u-1$]的一个登 761

bbs.theithome.com

记项*hash*(*a*)中找出指向登记项链表（通常叫做“散列桶”(buckets)）的指针，链表中的每一个登记项对应于已散列到*hash*(*a*)的*S*的元素*a*。集合运算的效率在很大程度上取决于所选择的散列函数。假定被散列到每一个登记项的元素不会多于2*u*/*n*个元素，则当每一个散列链是有序的时，测试从属关系需要的时间为*O*(*u*/*n*)；并、交、相等和子集需要的时间为*O*(*u*)。



图B-3 a) 标识符的平衡二叉树；b) 加入day导致树变得不平衡；以及c) 使此树重新平衡后的结果

Briggs和Torczon [BriT93]开发了一种新的稀疏集合表示，这种表示的基本运算时间是常数。它用两个含*u*个元素的数组*s*[ ]和*d*[ ]以及一个标量*c*表示稀疏集合$S \subseteq U = \{1, \cdots, u\}$。*c*的值是*S*的势。数组*d*[ ]按任意顺序存放*S*的*c*个元素于位置1, … *c*，并设置*s*[ ]的元素使得

$1 \leq s[i] \leq c$且$d[s[i]] = i$，当且仅当$i \in S$

即，*s*[ ]的第*i*个元素给出*i*在数组*d*[ ]中的位置。*s*[ ]和*d*[ ]中其他项的值不重要。例如，图B-4说明了集合{2, 5, 7, 4}在*u*=8时是如何表示的。基本运算是增加和删除一个元素，测试从属关系，以及确定集合的大小。我们执行这些运算如下：

1. 增加一个元素*v*：检查是否有$1 \leq s[v] \leq c$且$d[s[v]] = v$。如果不是，设置*c*为*c*+1，*d*[*c*]为*v*，和*s*[*v*]为*c*。

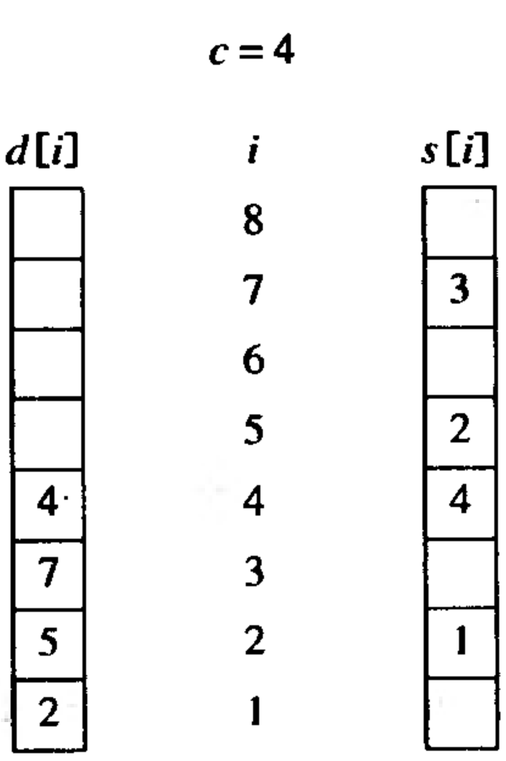
2. 删除一个元素*v*：检查是否有$1 \leq s[v] \leq c$且$d[s[v]] = v$。如果是，设置*d*[*s*[*v*]]为*d*[*c*]，*c*为*c* − 1，和*s*[*v*]:=0。

bbs.theithome.com

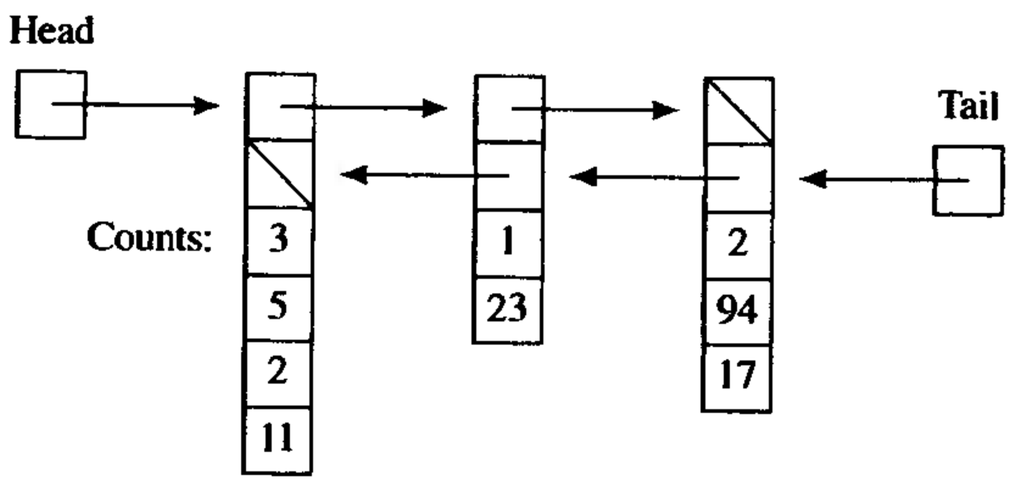
3. 测试v的从属关系：检查是否有$1 \leq s[v] \leq c$且$d[s[v]]=v$。如果是，回答“yes”，否则回答“no”。

4. 确定大小：返回c。

表示特定问题的稀疏集合有两种有用的混合表示方法。链接段表结合了数组的快速遍历与链表的动态分配。例如，图B-5展示了集合{5, 2, 11, 23, 94, 17}的链接段表表示。位向量链表与之类似，不同的只是表元素中存储的值是位向量。它们有助于这种位向量问题，在这种问题中，大部分的位在整个分析过程中都是常数。



图B-4 u=8时集合{2, 5, 7, 4}的稀疏表示



图B-5 用双向链接段表表示的集合{5, 2, 11, 23, 94, 17}

B.2 序列的表示

序列几乎同集合一样重要，并且事实上许多表示都是从集合延伸到序列的，不同的只是我们按预先定义的顺序保持序列的成员。

序列的一些重要运算是本附录开始时列出的那些运算，即增加一个值，删除一个值，测试从属关系，合并两个序列，以及确定序列的长度。序列的连接也是重要的。

链表和链接段表是序列的常用表示，它允许运算在时间$O(s)$内执行，其中s是序列的长度（注意，这也包括链接的时间，因为我们所要做的只是接合第一个序列的尾部与第二个序列的开始）。

另一方面，位向量对序列表示一般没有什么帮助，因为它们对元素强制有一种通常并不希望的顺序。

平衡二叉树是相当有用的，因为它们允许对其登记项强加任意选择的顺序（我们只需要使每一个元素在位置上按那个顺序毗连，并在重新平衡的树中使用它），并且也因为对它们的运算较快。

B.3 树和DAG表示

树在编译过程中有若干重要的用途，如语法分析、代码生成和表达式优化。在多数情况下，树是二叉树——尤其是在代码生成和优化中——因此，我们关注的是二叉树。

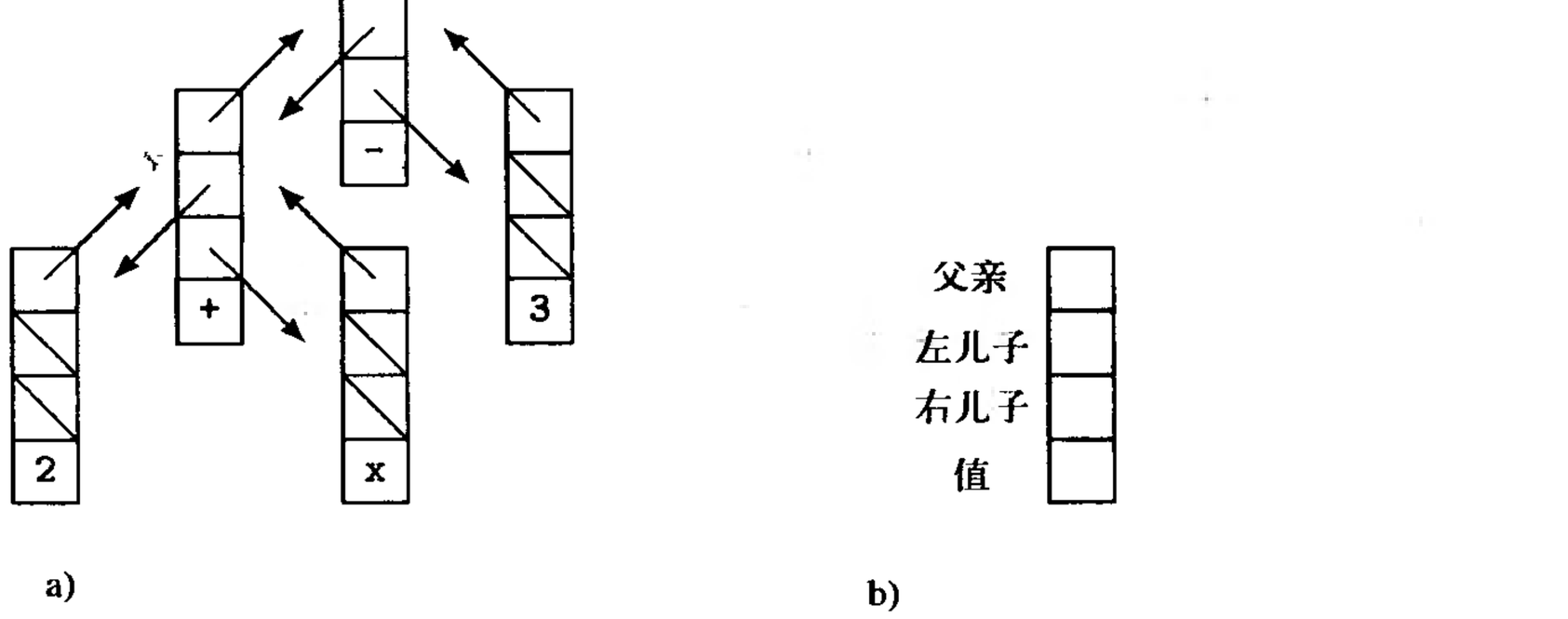
常用的二叉树表示有两种，链的和线性的。链形式中的结点一般至少含有4个域：父亲、左儿子、右儿子和值，如图B-6所示。每一个结点也可能还有用来表示其他值的另外的域。

762
~
763

树的线性表示依赖于这一事实：二叉树总是可以用它的根，其后跟随它的子树来表示，即，用前缀波兰表示来表示。在这种形式中，图B-6中链的表示变成了-+2x3。这种类型的表示在某些编译器中被用作为中间代码，并且也是Graham-Glanville风格的代码生成（参见6.2节）的基础。

bbs.theithome.com

越是公司制作 请尊重作者版权



a) b)

图B-6 a) 二叉树的链形式表示，即表示(2+x) − 3的表达式树；b) 它的各个域的含义

DAG有助于基本块的优化、代码生成以及代码调度（参见17.1.2），它几乎总是用链结构来表示的。

B.4 函数的表示

在编译器中，函数用于表示各种映射，例如映射基本块和流图的边到它们的执行频率，以及表示别名，等等。

最具效率的函数表示要求函数有一个由某种简单类型组成的定义域和一个简单类型的结果，并且从定义域的元素计算这个函数较容易，例如f(x, y, z) = x+2*y-z。当然，这种函数一般是用代码来表示的。

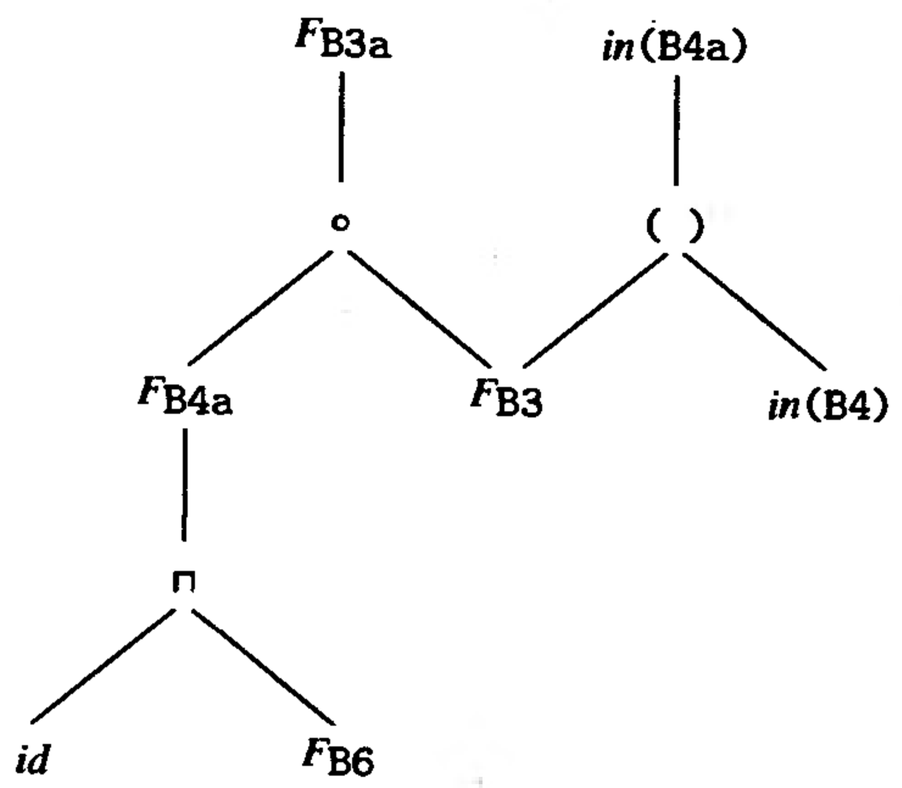
效率位于其次的函数表示需要用到一个数组。为了使得这种方法可用，定义域的组成必须是某个范围的整数、或者易于映射到某个整数范围，而且函数值必须是一致的、或可以通过存储在数组元素中的指针而得到。

散列方法为那种不易于映射到数组的函数提供了一种有吸引力的可选表示方法。在这种方法中，定义域元素首先被散列到一个确定的散列链表中，然后通过查找此表来寻找与参数匹配的存放对应函数值的登记项。对函数执行得最频繁的一种运算是，当给定一组参数时计算它的值，上面讨论的三种表示做到这一点都十分容易。

764

虽然不太频繁，但有时我们也可能会需要复合两个函数，或通过改变函数的定义域或一个值，或同时改变两者的方式来修改一个函数。

另一种函数表示是那种设计成通过程序来构造，然后重复使用这个程序来计算其值的表示，例如像在基于控制树的数据流分析中（参见8.6节）所表示的一样。在这种情形中，我们需要构造一个较后能够被解释执行的数据结构，如8.7.3节所讨论的图。图B-7是这类图的一个例子。其中，根是要计算的函数的名字，其下的每一个子图表示如何计算它（“∘”表示函数复合，“()”表示函数应用，“⊓”表示格的交运算，*id*表示所标识的这个函数）。这样的图不难构造，并且可以随计算数据流信息的需要来解释。



图B-7 结构化数据流方程式的DAG表示

bbs.theithome.com

越是公句制作请尊重作者版权

B.5 进一步阅读

有关如何保持二叉树的平衡，以及对编译器的构造而言有价值的其他数据结构的描述，可参见任何关于数据结构的优秀原文，例如Knuth的[Knut73]。

[BriT93]中描述了Briggs和Torczon关于稀疏集合的新表示。 765

bbs.theithome.com

版权所有 请勿制作非营利公司

bbs.theithome.com

超星公司制作 请尊重作者版权

附录C 软件资源

这个附录介绍可通过Internet或其他发布途径自由获取的、用于研究性编译器构造项目的软件。其中大部分软件包都带有限制其用于教育项目的许可协议。在本书出版商的网站上可以访问到本附录的所有部分，其中有所有互联网、WWW的链接，以及本附录中给出的ftp地址。本书的网址是http://books.elsevier. com/us//mk/us/subindex.asp? maintarget= companions/defaultindividual.asp&isbn=1558603204。

C.1 寻找Internet上的软件

这里列出的资源只是许多可能得到的、有助于编译器相关项目的软件包中的几个。通过某个搜索引擎，如Yahoo、Alta Vista、Lycos、Infoseek等，可以在WWW上找到其他的信息。有众多的参考书（如[Krol92]和[Keho93]）提供了关于访问和使用这种搜索引擎的信息。

C.2 机器模拟器

C.2.1 Spim

威斯康星大学的James Larus编写了一个MIPS R2000和R3000的模拟器，叫做SPIM，可获得它用于学习项目。[HenP94]的附录A中详细地介绍了这个模拟器。

SPIM能够读入并执行含汇编语言语句的文件和MIPS a.out文件。它是一个包括调试器和操作系统接口的自包含系统，并且至少可在DEC、Sun、IBM和HP等工作站上运行。 767

SPIM实现了几乎整个MIPS扩充汇编指令集（某些复杂的浮点比较指令和页表管理指令除外）。它包含完整的源代码和所有使用文档，有一个简单的终端风格的界面，也有一个基于X Windows的界面。

当本书即将印刷时，SPIM的一个DOS版本和一个新的Windows版本正在开发中。Windows版本将运行在Windows 3.1、Windows 95和Windows NT之上。可在出版商的网站http://www.mkp.com/cod2e.htm查看到关于这些新版本的信息。

为了取回一个包含SPIM系统和文档的压缩tar文件，可访问http://www. cs.wisc. edu/~larus/spim.html并遵循那里找到的使用说明。为了及时得到这个系统进一步的更新信息，可发送你的电子邮件到larus@cs.wisc.edu。

C.2.2 Spa

Spa是澳大利亚阿德雷德大学的Gordon Irlam编写的一组工具，其中包含了一个SPARC模拟器。要获得有关它的更多内容，可通过http://www.base.com访问文件gordoni/spa.html。

C.3 编译器

C.3.1 GNU

GNU编译器由自由软件基金会的自愿者们开发，并且可以自由发布和修改。如果满足某些

bbs.theithome.com

超星公司制作 请尊重作者版权

条件的话，它可以包含在商业产品中。有关信息可与基金会联系，地址为675 Massachusetts Avenue，Cambridge，MA 02139。

为了获得GNU编译器的源代码，建立一个匿名ftp连接到prep.ai.mit.edu、ftp.uu.net或者世界其他地方的一些机器，取回压缩tar文件pub/gnu/gcc-version.tar.gz，其中version是可用的最高版本号。同一目录中的文件GETTING.GNU.SOFTWARE对初学者会有所帮助。

C.3.2 LCC

lcc是由Christopher Fraser和David Hanson编写的一个易于移植到其他目标机的ANSI C编译器。它可通过匿名ftp.princeton.edu在目录pub/lcc中得到。建议你先从那个目录取得文件README开始。它的前端已被修改用于SUIF系统（参见下一小节）。lcc也可以从http://www.cs.princeton.edu/software/lcc获得。

768 这个编译器没有优化器，但是它的中间代码适合于本书讨论的大部分优化。

在[FraH91b]中有关于该编译器的简要介绍，在[FraH95]中有详细的介绍。如果你希望加入lcc邮件列表，可发送电子邮件信息subscribe lcc到majordomo@cs.princeton.edu。有关更多的信息，请访问http://www.cs.princeton.edu/software/lcc。

C.3.3 SUIF

SUIF是由Monica Lam和她在斯坦福大学的同事一起开发的一个实验型编译系统[⊖]，它由中间格式、组成编译器的核心部分，以及一个将C或Fortran 77转换为MIPS代码且具有并行化功能的编译器组成。这个发布版本还包含以下一些内容：

1. C和Fortran 77前端。
2. 数组数据依赖关系分析库。
3. 循环转换库（基于么模变换和循环铺砌）。
4. 矩阵和线性不等式运算库。
5. 并行代码生成器和运行库。
6. 标量优化器。
7. MIPS后端。
8. C后端（因此，本地C编译器可以作为非MIPS系统的后端）。
9. 线性不等式计算器（用于原型算法）。
10. 可用于编译课程的简化接口。

SUIF可从http://suif.stanford.edu获得。

SUIF的发布版本不包含担保，也没有关于支持的承诺，它可以自由地用于非商业目的，但禁止重新发布。

SUIF并不打算作为产品质量的编译器。它运行得既不快，也不能像gcc或你的机器上的编译器所能做到的那样高效地生成代码。但是，它能够编译主要的基准测试程序，并且能作为学生项目或编译研究的一个基础。

C.4 代码生成器的产生器：BURG和IBURG

BURG是一个基于BUGS技术（参见[Pele88]和[PelG88]）的代码生成器的产生器，是由

⊖ SUIF的名字代表Stanford University Intermediate Form。

bbs.theithome.com

Christopher Fraser、Robert Henry和Todd Proebsting编写的，可以通过匿名ftp从kaese.cs.wisc.edu获得。它作为一个压缩的shar档案文件提供在位于pub/burg.shar.Z的文件中。[FraH91a]提供了该系统的概述以及它的使用说明。

IBURG是另一个代码生成器的产生器，它也基于BUGS技术，但能进行编译时的动态程序设计，IBURG的编写者是Fraser、Hanson和Proebsting [FraH92]，通过匿名ftp可以从ftp.cs.princeton.edu获得它。这是一个压缩的tar文件pub/iburg.tar.Z或pub/iburg.tar.zip。[FraH92]给出了关于这个系统以及如何使用它的概述。

C.5 剖面分析工具

C.5.1 QPT

QPT是由威斯康星大学的Thomas Ball和Jamea Larus编写的一个准确和高效的程序剖面分析器和追踪系统。它重写一个程序的可执行文件（a.out），在其中插入代码记录每一个基本块或控制流边的执行频率或执行序列。另一个叫做QPT_STATS的程序则从这些信息中计算程序的各个过程的执行代价。与UNIX工具prof和gprof不同，QPT记录精确的执行频率而不是采样统计。当追踪一个程序时，QPT产生一个轨迹再生成程序，这个程序读入高度压缩的轨迹文件并生成完整的程序轨迹。要获得QPT和它的文档，可访问http://www.cs. wisc.edu/~larus/qpt.html，并遵循在那里找到的说明。

当用来进行剖面分析时，QPT使用两种模式。在慢模式中，它在程序的每一个基本块放置一个计数器。在快模式中，它在程序控制流图边集合的非频繁执行边的子集上放置计数器。这样放置可以将剖面分析的代价减少3到4倍，但稍微增加了产生剖面或轨迹信息所需要的时间。

QPT当前运行在SPARC系统并且被编写成可移植的——所有与机器相关的特征都集中在少数几个文件中。移植这个程序到一个新机器大约需要花两个人一个月的时间。QPT的发布包含了源代码和少量的文档。

QPT也是一个叫做WARTS（Wisconsin Architectural Research Tool Set）的大项目的一部分，可用http://www.cs.wisc.edu/~larus/warts.html访问这个项目。

C.5.2 SpixTools和Shade

SpixTools是由Sun Microsystems公司的Robert Cmelik编写的允许对SPARC应用程序进行指令级性能分析的一组程序。[Cmel93] 给出了它的一个指南性介绍，并提供了关于SpixTools的参考手册。

Spix创建用户程序的一个插有测量桩的副本。当这个带测量桩的副本运行时，它追踪每一个基本块的执行情况，并写出基本块的最终执行次数统计。有一些工具可以显示和总结这些统计。Spixstats可打印出列表，列出操作码的使用、分支行为、寄存器使用、函数使用等数据。Sdas可反汇编应用程序，并用注释标出反汇编代码指令的执行次数。Sprint能打印出应用程序的源代码，同时标注出语句或指令的计数。

使用SpixTool的应用程序必须静态连接，并且不能使用自我修改的代码。此外还有其他几个限制。

Shade是一个指令集模拟器和定制的轨迹生成器，它是由David Keppel和Robert Cmelik编写的。它在用户提供的轨迹分析器的控制下执行和追踪应用程序。为了减少通信开销，Shade和分析器在同一个地址空间运行。为了进一步改善性能，模拟和追踪应用程序的代码是动态生成的，并保存下来可以重用。当前的实现运行在SPARC系统，并且在不同程度上模拟SPARC

769
770
越是公句制作请尊重作者版权
bbs.theithome.com

Version 8和Version 9以及MIPS I体系结构。[CmeK93]介绍了Shade的功能、设计、实现和性能，并讨论了一般的指令集模拟。

Shade提供了细粒度的追踪控制，因此只有实际需要的数据才有数据搜集开销。Shade是可扩充的，这使得可使用能够考察任意状态的分析器，并搜集Shade本身不知道如何搜集的特殊信息。

为了获得SpixTools和Shade，可向Robert Cmelik申请许可表格，联系地址是Sun Microsystems，Inc.，2550 Garcia Avenue，Mountain View，CA94043-1100，电话415-336-1709，
771 电子邮件rfc@sun.com。

超星公司制作 请尊重作者版权

bbs.theithome.com

版权所有 请尊重作者版权

参考文献

[Ada83] *The Programming Language Ada Reference Manual, ANSI/MIL-STD-1815A-1983,* U.S. Dept. of Defense, Washington, DC, 1983.

[AdlG93] Adl-Tabatabai, Ali-Reza and Thomas Gross. Detection and Recovery of Endangered Variables Caused by Instruction Scheduling, in [PLDI93], pp. 13–25.

[AhaL93] Ahalt, Stanley C. and James F. Leathrum. Code-Generation Methodology Using Tree-Parsers and High-Level Intermediate Representations, *J. of Programming Languages,* Vol. 1, No. 2, 1993, pp. 103–126.

[AhoC75] Aho, Alfred V. and M.J. Corasick. Efficient String Matching: An Aid to Bibliographic Search, *CACM,* Vol. 18, No. 6, June 1975, pp. 333–340.

[AhoG89] Aho, Alfred V., Mahadevan Ganapathi, and Steven W.K. Tjiang. Code Generation Using Tree Pattern Matching and Dynamic Programming, *ACM TOPLAS,* Vol. 11, No. 4, Oct. 1989, pp. 491–516.

[AhoH74] Aho, Alfred V., John Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[AhoJ76] Aho, Alfred V. and Steven C. Johnson. Optimal Code Generation for Expression Trees, *JACM,* Vol. 23, No. 3, July 1976, pp. 488–501.

[AhoS86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1986.

[AigG84] Aigran, Philippe, Susan L. Graham, Robert R. Henry, M. Kirk McKusick, and Eduardo Pelegri-Llopart. Experience with a Graham-Glanville Style Code Generator, in [Comp84], pp. 13–24.

[Aike88] Aiken, Alexander. *Compaction-Based Parallelization,* Ph.D. thesis, Tech. Rept. TR-88-922, Dept. of Comp. Sci., Cornell Univ., Ithaca, NY, June 1988.

[AikN88a] Aiken, Alexander and Alexandru Nicolau. Optimal Loop Parallelization, in [PLDI88], pp. 308–317.

[AikN88b] Aiken, Alexander and Alexandru Nicolau. Perfect Pipelining: A New Loop Parallelization Technique, *Proc. of the 1988 Euro. Symp. on Programming,* Springer-Verlag, Berlin, 1988, pp. 308–317.

[AikN91] Aiken, Alexander and Alexandru Nicolau. A Realistic Resource-Constrained Software Pipelining Algorithm, in Nicolau, Alexandru, David Gelernter, Thomas Gross, and David Padua (eds.). *Advances in Languages and Compilers for Parallel Processing,* MIT Press, Cambridge, MA, 1991.

[AllC72a] Allen, Frances E. and John Cocke. A Catalogue of Optimizing Transformations, in Rustin, Randall (ed.). *Design and Optimization of Compilers,* Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 1–30.

[AllC72b] Allen, Frances E. and John Cocke. Graph Theoretic Constructs for Program Control Flow Analysis, Research Rept. RC3923, IBM Thomas J. Watson Research Ctr., Yorktown Heights, NY, 1972.

[AllC76] Allen, Frances E. and John Cocke. A Program Data Flow Analysis Procedure, *CACM,* Vol. 19, No. 3, Mar. 1976, pp. 137–147.

bbs.theithome.com

[AllC81] Allen, Frances E., John Cocke, and Ken Kennedy. Reduction of Operator Strength, in [MucJ81], pp. 79–101.

[AllC86] Allen, J. Randy, David Callahan, and Ken Kennedy. An Implementation of Interprocedural Data Flow Analysis in a Vectorizing Fortran Compiler, Tech. Rept. COMP TR-86-38, Dept. of Comp. Sci., Rice Univ., Houston, TX, May 1986.

[Alle69] Allen, Frances E. Program Optimization, in Halprin, Mark I. and Christopher J. Shaw (eds.). *Annual Review of Automatic Programming*, Vol. 5, Pergamon Press, Oxford, UK, 1969, pp. 239–307.

[Alph92] *Alpha Architecture Handbook*, Digital Equipment Corp., Maynard, MA, 1992.

[AlpW88] Alpern, Bowen, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs, in [POPL88], pp. 1–11.

[AlsL96] Alstrup, Stephen and Peter W. Lauridsen. A Simple Dynamic Algorithm for Maintaining a Dominator Tree, Tech. Rept. 96/3, Dept. of Comp. Sci., Univ. of Copenhagen, Copenhagen, 1996.

[AndS92] Andrews, Kristy and Duane Sand. Migrating a CISC Computer Family onto RISC via Object Code Translation, in [ASPL92], pp. 213–222.

[ANSI89] *American National Standard X3.159-1989, The C Programming Language*, ANSI, New York, NY, 1989.

[ASPL82] *Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, published as *SIGPLAN Notices*, Vol. 17, No. 4, Apr. 1982.

[ASPL87] *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, IEEE Comp. Soc. Order No. 805, Oct. 1987.

[ASPL89] *Proc. of the 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, published as *SIGPLAN Notices*, Vol. 24, special issue, May 1989.

[ASPL91] *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, published as *SIGPLAN Notices*, Vol. 26, No. 4, Apr. 1991.

[ASPL92] *Proc. of the Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, published as *SIGPLAN Notices*, Vol. 27, No. 9, Sept. 1992.

[AusH82] Auslander, Marc and Martin Hopkins. An Overview of the PL.8 Compiler, in [Comp82], pp. 22–31.

[BalL92] Ball, Thomas and James R. Larus. Optimally Profiling and Tracing Programs, in [POPL92], pp. 59–70.

[BalL93] Ball, Thomas and James R. Larus. Branch Prediction for Free, in [PLDI93], pp. 300–313.

[Bane76] Banerjee, Utpal. *Dependence Testing in Ordinary Programs*, M.S. thesis, Dept. of Comp. Sci., Univ. of Illinois, Urbana-Champaign, IL, Nov. 1976.

[Bane88] Banerjee, Utpal. *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, MA, 1988.

[Bane93] Banerjee, Utpal. *Loop Transformations for Restructuring Compilers*, Kluwer Academic Publishers, Boston, MA, 1993.

[Bane94] Banerjee, Utpal. *Loop Parallelization*, Kluwer Academic Publishers, Boston, MA, 1993.

[Bann79] Banning, John P. An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables, in [POPL79], pp. 29–41.

[Bart78] Barth, John M. A Practical Interprocedural Data Flow Analysis Algorithm, *CACM*, Vol. 21, No. 9, Nov. 1978, pp. 724–736.

bbs.theithome.com

[Bell90] Bell, Ron. IBM RISC System/6000 Performance Tuning for Numerically Intensive Fortran and C Programs, Document No. GG24-3611, IBM Intl. Tech. Support Ctr., Poughkeepsie, NY, Aug. 1990.

[BerC92] Bernstein, David, Doron Cohen, Yuval Lavon, and Vladimir Rainish. Performance Evaluation of Instruction Scheduling on the IBM RISC System/6000, *Proc. of MICRO-25,* Portland, OR, published as *SIG MICRO Newsletter,* Vol. 23, Nos. 1 and 2, Dec. 1992.

[BerG89] Bernstein, David, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers, in [PLDI89], pp. 258–263.

[BerG95] Bergin, Thomas J. and Richard G. Gibson (eds.). *The History of Programming Languages-II,* ACM Press, New York, NY, 1995.

[BerK87] Bergh, A., Keith Keilman, Daniel Magenheimer, and James A. Miller. HP 3000 Emulation on HP Precision Architecture Computers, *Hewlett-Packard J.,* Dec. 1987.

[BerR91] Bernstein, David and M. Rodeh. Global Instruction Scheduling for Superscalar Machines, in [PLDI91], pp. 241–255.

[Bird82] Bird, Peter. An Implementation of a Code Generator Specification Language for Table Driven Code Generators, in [Comp82], pp. 44–55.

[BliC92] Blickstein, David S., Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, and William B. Noyce. The GEM Optimizing Compiler System, *Digital Tech. J.,* Vol. 4, No. 4, special issue, 1992.

[BodC90] Bodin, François and François Charot. Loop Optimization for Horizontal Microcoded Machines, *Proc. of the 1990 Intl. Conf. on Supercomputing,* Amsterdam, June 1990, pp. 164–176.

[Brad91] David G. Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors,* Ph.D. thesis, Tech. Rept. UW-CSE-91-08-07, Dept. of Comp. Sci. and Engg., Univ. of Washington, Seattle, WA, June 1991.

[BraE91] Bradlee, David G., Susan J. Eggers, and Robert R. Henry. Integrated Register Allocation and Instruction Scheduling for RISCs, in [ASPL91], pp. 122–131.

[BraH91] Bradlee, David G., Robert R. Henry, and Susan J. Eggers. The Marion System for Retargetable Instruction Scheduling, in [PLDI91], pp. 229–240.

[BriC89] Briggs, Preston, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation, in [PLDI89], pp. 275–284.

[BriC94a] Briggs, Preston, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation, *ACM TOPLAS,* Vol. 16, No. 3, May 1994, pp. 428–455.

[BriC94b] Briggs, Preston and Keith D. Cooper. Effective Partial Redundancy Elimination, in [PLDI94], pp. 159–170.

[BriC94c] Briggs, Preston, Keith D. Cooper, and Taylor Simpson. Value Numbering, Tech. Rept. CRPC-TR94517-S, Ctr. for Research on Parallel Computation, Rice Univ., Houston, TX, Nov. 1994.

[Brig92] Briggs, Preston. Register Allocation via Graph Coloring, Tech. Rept. CRPC-TR92218, Ctr. for Research on Parallel Computation, Rice Univ., Houston, TX, Apr. 1992.

[BriT93] Briggs, Preston and Linda Torczon. An Efficient Representation for Sparse Sets, *ACM LOPLAS,* Vol. 2, Nos. 1–4, Mar.–Dec. 1993, pp. 59–69.

[CalC86] Callahan, David, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation, in [Comp86], pp. 152–161.

[CalC90] Callahan, David, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables, in [PLDI90], pp. 53–65.

bbs.theithome.com

[CalG95] Calder, Brad, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin G. Zorn. Corpus-Based Static Branch Prediction, in [PLDI95], pp. 79–92.

[CalK91] Callahan, David and Brian Koblenz. Register Allocation by Hierarchical Tiling, in [PLDI91], pp. 192–203.

[Call86] Callahan, David. Dependence Testing in PFC: Weak Separability, Dept. of Comp. Sci., Rice Univ., Houston, TX, Aug. 1986.

[Call88] Callahan, David. The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis, in [PLDI88], pp. 47–56.

[CamK93] Campbell, Philip L., Ksheerabdhi Krishna, and Robert A. Ballance. Refining and Defining the Program Dependence Web, Tech. Rept. CS93-6, Univ. of New Mexico, Albuquerque, NM, Mar. 1993.

[Catt79] Cattell, Roderic G.G. Code Generation and Machine Descriptions, Tech. Rept. CSL-79-8, Xerox Palo Alto Research Ctr., Palo Alto, CA, Oct. 1979.

[CF7790] *CF77 Compiling System, Volume 1: Fortran Reference Manual,* Publication SR-3071 4.0, Cray Research, Inc., Mendota Heights, MN, 1990.

[ChaA81] Chaitin, Gregory, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register Allocation Via Coloring, *Computer Languages,* Vol. 6, No. 1, 1981, pp. 47–57; also in [Stal90], pp. 88–97.

[Chai82] Chaitin, Gregory. Register Allocation and Spilling via Graph Coloring, in [Comp82], pp. 98–105.

[ChaW90] Chase, David R., Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures, in [PLDI90], pp. 296–310.

[CheM92] Chernoff, Anton and Maurice P. Marks. Personal communication, Digital Equipment Corp., Nashua, NH, Mar. 1992.

[Cher92] Chernoff, Anton. Personal communication, Digital Equipment Corp., Nashua, NH, May 1992.

[ChiD89] Chi, Chi-Hung and Hank Dietz. Unified Management of Registers and Cache Using Liveness and Cache Bypass, in [PLDI89], pp. 344–355.

[ChoH84] Chow, Frederick and John Hennessy. Register Allocation by Priority-Based Coloring, in [Comp84], pp. 222–232; also in [Stal90], pp. 98–108.

[ChoH90] Chow, Frederick and John Hennessy. The Priority-Based Coloring Approach to Register Allocation, *ACM TOPLAS,* Vol. 12, No. 4, pp. 501–536.

[Chow83] Chow, Frederick. A Portable Machine-Independent Global Optimizer—Design and Measurements, Tech. Rept. 83-254, Comp. Systems Lab., Stanford Univ., Stanford, CA, Dec. 1983.

[Chow86] Chow, Paul. MIPS-X Instruction Set and Programmer's Manual, Tech. Rept. No. CSL-86-289, Comp. Systems Lab., Stanford Univ., Stanford, CA, May 1986.

[Chow88] Chow, Frederick. Minimizing Register Usage Penalty at Procedure Calls, in [PLDI88], pp. 85–94.

[ChoW92] Chow, Frederick and Alexand Wu. Personal communication, MIPS Computer Systems, Inc., Mountain View, CA, May 1992.

[ChrH84] Christopher, Thomas W., Philip J. Hatcher, and Ronald C. Kukuk. Using Dynamic Profiling to Generate Optimized Code in a Graham-Glanville Style Code Generator, in [Comp84], pp. 25–36.

[CliR91] Clinger, William and Jonathan Rees (eds.). *Revised^4 Report on the Algorithmic Language Scheme,* Artificial Intelligence Lab., MIT, Cambridge, MA, and Comp. Sci. Dept., Indiana

bbs.theithome.com

Univ., Bloomington, IN, Nov. 1991.

[CloM87] Clocksin, W.F. and C.S. Mellish. *Programming in Prolog*, third edition, Springer-Verlag, Berlin, 1987.

[CmeK91] Cmelik, Robert F., Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An Analysis of SPARC and MIPS Instruction Set Utilization on the SPEC Benchmarks, in [ASPL91], pp. 290–302.

[CmeK93] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling, Tech. Rept. SMLI 93-12, Sun Microsystems Labs, Mountain View, CA, and Tech. Rept. UWCSE 93-06-06, Dept. of Comp. Sci. and Engg., Univ. of Washington, Seattle, WA, 1993.

[Cmel93] Cmelik, Robert F. SpixTools User's Manual, Tech. Rept. SMLI TR-93-6, Sun Microsystems Labs, Mountain View, CA, Feb. 1993.

[CocS69] Cocke, John and Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*, Courant Inst. of Math. Sci., New York Univ., New York, NY, 1969.

[ColN87] Colwell, Robert P., Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler, in [ASPL87], pp. 180–192.

[Comp79] *Proc. of the SIGPLAN '79 Symp. on Compiler Constr.*, Denver, CO, published as *SIGPLAN Notices*, Vol. 14, No. 8, Aug. 1979.

[Comp82] *Proc. of the SIGPLAN '82 Symp. on Compiler Constr.*, Boston, MA, published as *SIGPLAN Notices*, Vol. 17, No. 6, June 1982.

[Comp84] *Proc. of the SIGPLAN '84 Symp. on Compiler Constr.*, Montreal, Quebec, published as *SIGPLAN Notices*, Vol. 19, No. 6, June 1984.

[Comp86] *Proc. of the SIGPLAN '86 Symp. on Compiler Constr.*, Palo Alto, CA, published as *SIGPLAN Notices*, Vol. 21, No. 7, July 1986.

[CooH92] Cooper, Keith D., Mary W. Hall, and Linda Torczon. Unexpected Effects of Inline Substitution: A Case Study, *ACM LOPLAS*, Vol. 1, No. 1, pp. 22–32.

[CooH93] Cooper, Keith D., Mary W. Hall, Robert T. Hood, Ken Kennedy, K.S. McKinley, J.M. Mellor-Crummey, Linda Torczon, and S.K. Warren. The ParaScope Parallel Programming Environment, *Proc. of the IEEE*, Vol. 81, No. 2, 1993, pp. 244–263.

[CooK84] Cooper, Keith D. and Ken Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary, in [Comp84], pp. 247–258.

[CooK86] Cooper, Keith D., Ken Kennedy, and Linda Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation, in [Comp86], pp. 58–67.

[CooK88a] Cooper, Keith D. and Ken Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary—A Correction, *SIGPLAN Notices*, Vol. 23, No. 4, Apr. 1988, pp. 35–42.

[CooK88b] Cooper, Keith D. and Ken Kennedy. Interprocedural Side-Effect Analysis in Linear Time, in [PLDI88], pp. 57–66.

[CooK89] Cooper, Keith D. and Ken Kennedy. Fast Interprocedural Alias Analysis, in [POPL89], pp. 49–59.

[CooS95a] Cooper, Keith D., Taylor Simpson, and Christopher Vick. Operator Strength Reduction, Tech. Rept. CRPC-TR95635-S, Ctr. for Research on Parallel Computation, Rice Univ., Houston, TX, Oct. 1995.

[CooS95b] Cooper, Keith D., and Taylor Simpson. SCC-Based Value Numbering, Tech. Rept. CRPC-TR95636-S, Ctr. for Research on Parallel Computation, Rice Univ., Houston, TX, Oct. 1995.

[CooS95c] Cooper, Keith D., and Taylor Simpson. Value-Driven Code Motion, Tech. Rept. CRPC-TR95637-S, Ctr. for Research on Parallel Computation, Rice Univ., Houston, TX, Oct.

超星公司制作 请尊重作者版权

bbs.theithome.com

1995.

[CouH86] Coutant, Deborah, Carol Hammond, and John Kelly. Compilers for the New Generation of Hewlett-Packard Computers, *Proc. of COMPCON S'86,* 1986, pp. 48–61; also in [Stal90], pp. 132–145.

[Cout86] Coutant, Deborah. Retargetable High-Level Alias Analysis, in [POPL86], pp. 110–118.

[CytF89] Cytron, Ronald, Jean Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form, in [POPL89], pp. 23–25.

[CytF91] Cytron, Ronald, Jean Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Program Dependence Graph, *ACM TOPLAS,* Vol. 13, No. 4, Oct. 1991, pp. 451–490.

[DamG94] Damron, Peter C., Vinod Grover, and Shahrokh Mortazavi. Personal communication, Sun Microsystems, Inc., Mountain View, CA, May 1994.

[DanE73] Dantzig, George and B.C. Eaves. Fourier-Motzkin Elimination and Its Dual, *J. of Combinatorial Theory* A, Vol. 14, 1973, pp. 288–297.

[DeuS84] Deutsch, L. Peter and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System, in [POPL84], pp. 297–302.

[Deut94] Deutsch, Alain. Interprocedural May-Alias Analysis for Pointers: Beyond *k*-Limiting, in [PLDI94], pp. 230–241.

[Dham88] Dhamdhere, Dhananjay M. A Fast Algorithm for Code Movement Optimization, *SIGPLAN Notices,* Vol. 23, No. 10, Oct. 1988, pp. 172–180.

[Dham91] Dhamdhere, Dhananjay M. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise, *ACM TOPLAS,* Vol. 13, No. 2, Apr. 1991, pp. 291–294.

[DhaR92] Dhamdhere, Dhananjay M., Barry K. Rosen, and F. Kenneth Zadeck. How to Analyze Large Programs Efficiently and Informatively, in [PLDI92], pp. 212–223.

[DonB79] Dongarra, Jack, James Bunch, Cleve Moler, and G. Stewart. *LINPACK Users Guide,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979.

[EbcG94] Ebcioğlu, Kemal, Randy Groves, Ki-Chang Kim, Gabriel Silberman, and Isaac Ziv. VLIW Compilation Techniques in a Superscalar Environment, in [PLDI94], pp. 36–46.

[EisW92] Eisenbeis, Christine and D. Windheiser. A New Class of Algorithms for Software Pipelining with Resource Constraints, Tech. Rept., INRIA, Le Chesnay, France, 1992.

[Elli85] Ellis, John R. *Bulldog: A Compiler for VLIW Architectures,* Ph.D. dissertation, Dept. of Comp. Sci., Yale Univ., New Haven, CT, 1985.

[EllS90] Ellis, Margaret A. and Bjarne Stroustrup. *The Annotated* C++ *Reference Manual,* Addison-Wesley, Reading, MA, 1990.

[Enge75] Engelfriet, Joost. Tree Automata and Tree Grammars, DAIMI Rept. FN-10, Dept. of Comp. Sci., Univ. of Aarhus, Aarhus, Denmark, Apr. 1975.

[FarK75] Farrow, Rodney, Ken Kennedy, and Linda Zucconi. Graph Grammars and Global Program Flow Analysis, *Proc. of the 17th IEEE Symp. on Foundations of Comp. Sci.,* Houston, TX, Nov. 1975.

[Farn88] Farnum, Charles. Compiler Support for Floating-Point Computation, *Software—Practice and Experience,* Vol. 18, No. 7, July 1988, pp. 701–709.

[Feau91] Feautrier, P. Data Flow Analysis of Array and Scalar References, *Intl. J. of Parallel Programming,* Vol. 20, No. 1, Jan. 1991.

[FerO87] Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization, *ACM TOPLAS,* Vol. 9, No. 3, July 1987, pp. 319–349.

bbs.theithome.com

[Fish81] Fisher, Joseph A. Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. on Comps.*, Vol. C-30, No. 7, July 1981, pp. 478–490.

[FisL91] Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler With* C, Benjamin-Cummings, Redwood City, CA, 1991.

[Fort78] *Programming Language FORTRAN*, ANSI X3.9-1978 and ISO 1539-1980(E), ANSI, New York, NY, 1978.

[Fort92] *Programming Language Fortran 90*, ANSI X3.198-1992 and ISO/IEC 1539-1991(E), ANSI, New York, NY, 1992.

[FraH91a] Fraser, Christopher W., Robert R. Henry, and Todd A. Proebsting. BURG—Fast Optimal Instruction Selection and Tree Parsing, *SIGPLAN Notices*, Vol. 27, No. 4, Apr. 1991, pp. 68–76.

[FraH91b] Fraser, Christopher W., David A. Hanson. A Retargetable Compiler for ANSI C, *SIGPLAN Notices*, Vol. 26, No. 10, Oct. 1991, pp. 29–43.

[FraH92] Fraser, Christopher W., David R. Hanson, and Todd A. Proebsting. Engineering a Simple Code-Generator Generator, *ACM LOPLAS*, Vol. 1, No. 3, Sept. 1992, pp. 213–226.

[FraH95] Fraser, Christopher W. and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*, Benjamin-Cummings, Redwood City, CA, 1995.

[Frei74] Freiburghouse, Richard A. Register Allocation via Usage Counts, *CACM*, Vol. 17, No. 11, Nov. 1974, pp. 638–642.

[GanF82] Ganapathi, Mahadevan and Charles N. Fischer. Description-Driven Code Generation Using Attribute Grammars, in [POPL82], pp. 107–119.

[GanF84] Ganapathi, Mahadevan and Charles N. Fischer. Attributed Linear Intermediate Representations for Code Generation, *Software—Practice and Experience*, Vol. 14, No. 4, Apr. 1984, pp. 347–364.

[GanF85] Ganapathi, Mahadevan and Charles N. Fischer. Affix Grammar Driven Code Generation, *ACM TOPLAS*, Vol. 7, No. 4, Oct. 1985, pp. 560–599.

[GanK89] Ganapathi, Mahadevan and Ken Kennedy. Interprocedural Analysis and Optimization, Tech. Rept. RICE COMP TR89-96, Dept. of Comp. Sci., Rice Univ., Houston, TX, July 1989.

[GarJ79] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, CA, 1979.

[GhoM86] Ghodssi, Vida, Steven S. Muchnick, and Alexand Wu. A Global Optimizer for Sun Fortran, C, and Pascal, *Proc. of the Summer 1986 USENIX Conf.*, Portland, OR, June 1986, pp. 318–334.

[GibM86] Gibbons, Phillip A. and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Processor, in [Comp86], pp. 11–16.

[GilG83] Gill, John, Thomas Gross, John Hennessy, Norman P. Jouppi, Steven Przybylski, and Christopher Rowen. Summary of MIPS Instructions, Tech. Note 83-237, Comp. Systems Lab., Stanford Univ., Stanford, CA, Nov. 1983.

[GinL87] Gingell, Robert A., Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared Libraries in SunOS, *Proc. of the 1987 Summer USENIX Conf.*, Phoenix, AZ, June 1987, pp. 131–146.

[GlaG78] Glanville, R. Steven and Susan L. Graham. A New Method for Compiler Code Generation, in [POPL78], pp. 231–240.

[GofK91] Goff, Gina, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing, in [PLDI91], pp. 15–29.

[Gold72] Goldstine, Hermann H. *The Computer from Pascal to Von Neumann*, Princeton Univ. Press, Princeton, NJ, 1972.

bbs.theithome.com

[Gold84] Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, MA, 1984.

[Gold91] Goldberg, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys,* Vol. 23, No. 1, Mar. 1991, pp. 5–48.

[GolR90] Golumbic, M.C. and Victor Rainish. Instruction Scheduling Beyond Basic Blocks, in [IBMJ90], pp. 93–97.

[GooH88] Goodman, J.R. and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. of the Intl. Conf. on Supercomputing,* St. Malo, France, July 1988, pp. 442–452.

[GosJ96] Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification,* Addison-Wesley, Reading, MA, 1996.

[GraH82] Graham, Susan L., Robert R. Henry, and Robert A. Schulman. An Experiment in Table Driven Code Generation, in [Comp82], pp. 32–43.

[GraJ79] Graham, Susan L., William N. Joy, and Olivier Roubine. Hashed Symbol Tables for Languages with Explicit Scope Control, in [Comp79], pp. 50–57.

[GriP68] Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language,* Prentice-Hall, Englewood Cliffs, NJ, 1968.

[GroF92] Grove, Richard B. and R. Neil Faiman, Jr. Personal communication, Digital Equipment Corp., Littleton, MA, Sept. 1992.

[GroT93] Grove, Dan and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementation, in [PLDI93], pp. 90–99.

[Grov94] Grove, Richard B. Personal communication, Digital Equipment Corp., Littleton, MA, May 1994.

[GupS89] Gupta, Rajiv, Mary Lou Soffa, and Tim Steele. Register Allocation via Clique Separators, in [PLDI89], pp. 264–274.

[Gupt90] Gupta, Rajiv. *Compiler Optimization of Data Storage,* Ph.D. dissertation, California Inst. of Technology, Pasadena, CA, July 1990.

[Gupt93] Gupta, Rajiv. Optimizing Array Bounds Checks Using Flow Analysis, *ACM LOPLAS,* Vol. 2, Nos. 1–4, Mar.–Dec. 1993, pp. 135–150.

[HalB90] Hall, Mark and John Barry (eds.). *The SunTechnology Papers,* Springer-Verlag, New York, NY, 1990.

[Hall91] Hall, Mary Wolcott. *Managing Interprocedural Optimization,* Ph.D. thesis, Dept. of Comp. Sci., Rice Univ., Houston, TX, Apr. 1991.

[Hay92] Hay, William. Personal communication, IBM Canada Lab., Toronto, Canada, May 1992.

[Hay94] Hay, William. Personal communication, IBM Canada Lab., Toronto, Canada, June 1994.

[Hech77] Hecht, Matthew S. *Flow Analysis of Computer Programs,* Elsevier North-Holland, New York, NY, 1977.

[HecU75] Hecht, Matthew S. and Jeffrey D. Ullman. A Simple Algorithm for Global Data Flow Problems, *SIAM J. of Computing,* Vol. 4, No. 4, Dec. 1975, pp. 519–532.

[HenD89a] Henry, Robert R. and Peter C. Damron. Performance of Table-Driven Code Generators Using Tree-Pattern Matching, Tech. Rept. 89-02-02, Comp. Sci. Dept., Univ. of Washington, Seattle, WA, Feb. 1989.

[HenD89b] Henry, Robert R. and Peter C. Damron. Algorithms for Table-Driven Code Generators Using Tree-Pattern Matching, Tech. Rept. 89-02-03, Comp. Sci. Dept., Univ. of Washington, Seattle,

bbs.theithome.com

WA, Feb. 1989.

[Hend90] Hendren, Laurie J. Parallelizing Programs with Recursive Data Structures, *IEEE Trans. on Parallel and Distributed Systems,* Vol. 1, No. 1, Jan. 1990, pp. 35–47.

[HenG83] Hennessy, John and Thomas Gross. Postpass Code Optimization of Pipeline Constraints, *ACM TOPLAS,* Vol. 5, No. 3, July 1983, pp. 422–448.

[HenP94] Hennessy, John L. and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface,* Morgan Kaufmann, San Francisco, CA, 1994.

[Henr84] Henry, Robert R. Graham-Glanville Code Generators, Rept. UCB/CSD 84/184, Comp. Sci. Division, Dept. of Elec. Engg. and Comp. Sci., Univ. of California, Berkeley, CA, May 1984.

[HewP91] *PA-RISC Procedure Calling Conventions Reference Manual,* HP Part No. 09740-90015, Hewlett-Packard, Palo Alto, CA, Jan. 1991.

[HimC87] Himelstein, Mark I., Fred C. Chow, and Kevin Enderby. Cross-Module Optimization: Its Implementation and Benefits, *Proc. of the Summer 1987 USENIX Conf.,* Phoenix, AZ, June 1987, pp. 347–356.

[Hime91] Himelstein, Mark I. Compiler Tail Ends, tutorial notes, *ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation,* Toronto, Ontario, June 1991.

[HofO82] Hoffman, C.W. and Michael J. O'Donnell. Pattern Matching in Trees, *JACM,* Vol. 29, No. 1, Jan. 1982, pp. 68–95.

[Hopk87] Hopkins, Martin. A Perspective on the 801/Reduced Instruction Set Computer, *IBM Systems J.,* Vol. 26, No. 1, 1987, pp. 107–121; also in [Stal90], pp. 176–190.

[HumH94] Hummel, Joseph, Laurie J. Hendren, and Alexandru Nicolau. A General Data Dependence Test for Dynamic, Pointer-Based Data Structures, in [PLDI94], pp. 218–229.

[IBMJ90] *IBM J. of Research and Development,* special issue on the IBM RISC System/6000 Processor, Vol. 34, No. 1, Jan. 1990.

[IEEE83] *IEEE Standard Pascal Computer Programming Language,* Inst. of Elec. and Electronic Engrs., New York, NY, 1983.

[IEEE85] *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std 754-1985, Inst. of Elec. and Electronic Engrs., New York, NY, 1985.

[Inge61] Ingerman, Peter Z. Thunks, *CACM,* Vol. 4, No. 1, Jan. 1961, pp. 55–58.

[Inte93] *Intel Reference Compiler for the Intel 386, Intel 486, and Pentium Microprocessor Family: An Overview,* Intel Corporation, Santa Clara, CA, 1993.

[Jain91] Jain, Sunil. Circular Scheduling: A New Technique to Perform Software Pipelining, in [PLDI91], pp. 219–228.

[JaiT88] Jain, Sunil and Carol Thompson. An Efficient Approach to Data Flow Analysis in a Multiple Pass Global Optimizer, in [PLDI88], pp. 154–163.

[JohM86] Johnson, Mark S. and Terrence C. Miller. Effectiveness of a Machine-Level, Global Optimizer, in [Comp86], pp. 99–108.

[John78] Johnson, Steven C. A Portable Compiler: Theory and Practice, in [POPL78], pp. 97–104.

[JohP93] Johnson, Richard and Keshav Pingali. Dependence-Based Program Analysis, in [PLDI93], pp. 78–89.

[JonM76] Jones, Neil D. and Steven S. Muchnick. Binding Time Optimization in Programming Languages: Some Thoughts Toward the Design of an Ideal Language, in [POPL76], pp. 77–94.

[JonM78] Jones, Neil D. and Steven S. Muchnick. TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages, *Lecture Notes in Comp. Sci.,* Vol. 66, Springer-Verlag, Berlin, 1978.

bbs.theithome.com

[JonM81a] Jones, Neil D. and Steven S. Muchnick. Flow Analysis and Optimization of LISP-Like Structures, in [MucJ81], pp. 102–131.

[JonM81b] Jones, Neil D. and Steven S. Muchnick. Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language Due to Dijkstra, in [MucJ81], pp. 380–393.

[KamU75] Kam, J.B. and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks, Tech. Rept. No. 169, Dept. of Elec. Engg., Princeton Univ., Princeton, NJ, 1975.

[KanH92] Kane, Gerry and Joe Heinreich. *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Karr84] Karr, Michael. Code Generation by Coagulation, in [Comp84], pp. 1–12.

[Keho93] Kehoe, Brendan P. *Zen and the Art of the Internet,* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Keil94] Keilman, Keith. Personal communication, Hewlett-Packard, Cupertino, CA, June 1994.

[Kenn71] Kennedy, Ken. *Global Flow Analysis and Register Allocation for Simple Code Structures,* Ph.D. thesis, Courant Inst., New York Univ., New York, NY, Oct. 1971.

[Kenn75] Kennedy, Ken. Node Listing Applied to Data Flow Analysis, in [POPL75], pp. 10–21.

[Kenn81] Kennedy, Ken. A Survey of Data Flow Analysis Techniques, in [MucJ81], pp. 5–54.

[Kenn86] Kennedy, Ken. PTOOL, Tech. Rept., Dept. of Math. Sci., Rice Univ., Houston, TX, 1986.

[KerE93] Kerns, Daniel R. and Susan J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency Is Uncertain, in [PLDI93], pp. 278–289.

[KheD99] Khedker, Uday P. and Dananjay M. Dhamdhere. Bidirectional Data Flow Analysis: Myths and Reality, *ACM SIGPLAN Notices,* Vol. 34, No. 6, June 1999, pp. 47–57. (Added in 2000, 4th printing; supports corrections made on pages 229, 264, and 407 in that printing.)

[Kild73] Kildall, Gary A. A Unified Approach to Global Program Optimization, in [POPL73], pp. 194–206.

[KnoR92] Knoop, Jens, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion, in [PLDI92], pp. 224–234.

[KnoR93] Knoop, Jens, Oliver Rüthing, and Bernhard Steffen. Lazy Strength Reduction, *J. of Programming Languages,* Vol. 1, No. 1, 1993, pp. 71–91.

[KnoR94] Knoop, Jens, Oliver Rüthing, and Bernhard Steffen. Partial Dead Code Elimination, in [PLDI94], pp. 147–158.

[KnoZ92] Knobe, Kathleen and F. Kenneth Zadeck. Register Allocation Using Control Trees, Tech. Rept. No. CS-92-13, Dept. of Comp. Sci., Brown Univ., Providence, RI, Mar. 1992.

[Knut62] Knuth, Donald E. A History of Writing Compilers, *Computers and Automation,* Dec. 1962, pp. 8–10, 12, 14, 16, 18.

[Knut71] Knuth, Donald E. An Empirical Study of Fortran Programs, *Software—Practice and Experience,* Vol. 1, No. 2, 1971, pp. 105–134.

[Knut73] Knuth, Donald E. *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.

[KolW95] Kolte, Pryadarshan and Michael Wolfe. Elimination of Redundant Array Subscript Checks, in [PLDI95], pp. 270–278.

[Kou77] Kou, Lawrence T. On Live-Dead Analysis for Global Data Flow Problems, *JACM,* Vol. 24, No. 3, July 1977, pp. 473–483.

[Kris90] Krishnamurthy, Sanjay. *Static Scheduling of Multi-Cycle Operations for a Pipelined RISC Processor,* M.S. paper, Dept. of Comp. Sci., Clemson Univ., Clemson, SC, May 1990.

[Krol92] Krol, Ed. *The Whole Internet User's Guide and Catalog,* O'Reilly & Associates, Sebastopol,

bbs.theithome.com

CA, 1992.

[Kuck74] Kuck, David J. Measurements of Parallelism in Ordinary Fortran Programs, *Computer,* Vol. 7, No. 1, Jan. 1974, pp. 37–46.

[Lam88] Lam, Monica S. Software Pipelining: An Efficient Scheduling Technique for VLIW Machines, in [PLDI88], pp. 318–328.

[Lam90] Lam, Monica S. Instruction Scheduling for Superscalar Architectures, in Joseph F. Traub (ed.). *Annual Review of Comp. Sci.,* Vol. 4, Annual Reviews, Inc., Palo Alto, CA, 1990, pp. 173–201.

[LamR91] Lam, Monica, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimization of Blocked Algorithms, in [ASPL91], pp. 63–74.

[LanJ82] Landwehr, Rudolf, Hans-Stephan Jansohn, and Gerhard Goos. Experience with an Automatic Code Generator Generator, in [Comp82], pp. 56–66.

[LarH86] Larus, James and Paul Hilfinger. Register Allocation in the SPUR Lisp Compiler, in [Comp86], pp. 255–263.

[Laru89] Larus, James R. Restructuring Symbolic Programs for Correct Execution on Multiprocessors, Tech. Rept. UCB/CSD/89/502, Comp. Sci. Division, Univ. of California, Berkeley, CA, May 1989.

[Laru90] Larus, James R. Abstract Execution: A Technique for Efficiently Tracing Programs, *Software—Practice and Experience,* Vol. 20, No. 12, Dec. 1990, pp. 1241–1258.

[LawL87] Lawler, Eugene, Jan Karel Lenstra, Charles Martel, and Barbara Simons. Pipeline Scheduling: A Survey, Comp. Sci. Research Rept. RJ 5738, IBM Research Division, San Jose, CA, July 1987.

[Lee89] Lee, Peter H. *Realistic Compiler Generation,* MIT Press, Cambridge, MA, 1989.

[Lee91] Lee, Peter H. (ed.). *Topics in Advanced Language Implementation,* MIT Press, Cambridge, MA, 1991.

[LenT79] Lengauer, Thomas and Robert E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph, *ACM TOPLAS,* Vol. 1, No. 1, July 1979, pp. 121–141.

[LevC80] Leverett, Bruce W., Roderic G.G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz, and William A Wulf. An Overview of the Production-Quality Compiler-Compiler Project, *Computer,* Vol. 13, No. 8, Aug. 1980, pp. 38–49.

[LoEg95] Lo, Jack L. and Susan Eggers. Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism, in [PLDI95], pp. 151–162.

[LowM69] Lowry, E. and C.W. Medlock. Object Code Optimization, *CACM,* Vol. 12, No. 1, 1969, pp. 13–22.

[MahC92] Mahlke, Scott A., Pohua P. Chang, William Y. Chen, John C. Gyllenhaal, Wen-mei W. Hwu, and Tokuzo Kiyohara. Compiler Code Transformations for Superscalar-Based High-Performance Systems, *Proc. of Supercomputing '92,* Minneapolis, MN, Nov. 1992, pp. 808–817.

[MahR94] Mahadevan, Uma and Sridhar Ramakrishnan. Instruction Scheduling over Regions: A Framework for Scheduling Across Basic Blocks, *Proc. of Compiler Constr. '94,* Edinburgh, *Lecture Notes in Comp. Sci.,* Vol. 786, Springer-Verlag, Berlin, 1994.

[MauF81] Mauney, John and Charles N. Fischer. ECP—An Error Correcting Parser Generator: User Guide, Tech. Rept. 450, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Oct. 1981.

[MayH91] Maydan, Dror E., John L. Hennessy, and Monica S. Lam. An Efficient Method for Exact Dependence Analysis, in [PLDI91], pp. 1–14.

[McFa89] McFarling, Scott. Program Optimization for Instruction Caches, in [ASPL89], pp. 183–191.

bbs.theithome.com

[McFa91a] McFarling, Scott. Procedure Merging with Instruction Caches, in [PLDI91], pp. 71–79.

[McFa91b] McFarling, Scott. *Program Analysis and Optimization for Machines with Instruction Cache,* Ph.D. dissertation, Tech. Rept. CSL-TR-91-493, Comp. Systems Lab., Stanford Univ., Stanford, CA, Sept. 1991.

[Mill92] Miller, James A. Personal communication, Hewlett-Packard, Cupertino, CA, Apr. 1992.

[MilT90] Milner, Robin, M. Tofte, and R. Harper. *The Definition of Standard ML,* MIT Press, Cambridge, MA, 1990.

[MitM79] Mitchell, James G., William Maybury, and Richard Sweet. Mesa Language Manual, Version 5.0, Tech. Rept. CSL-79-3, Xerox Palo Alto Research Ctr., Palo Alto, CA, Apr. 1979.

[MorR79] Morel, Etienne and Claude Renvoise. Global Optimization by Suppression of Partial Redundancies, *CACM,* Vol. 22, No. 2, Feb. 1979, pp. 96–103.

[MorR81] Morel, Etienne and Claude Renvoise. Interprocedural Elimination of Partial Redundancies, in [MucJ81], pp. 160–188.

[Morr91] Morris, W.G. CCG: A Prototype Coagulating Code Generator, in [PLDI91], pp. 45–58.

[MowL92] Mowry, Todd C., Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching, in [ASPL92], pp. 62–73.

[Mowr94] Mowry, Todd C. *Tolerating Latency Through Software-Controlled Prefetching,* Ph.D. dissertation, Dept. of Elec. Engg., Stanford Univ., Stanford, CA, Mar. 1994.

[Much88] Muchnick, Steven S. Optimizing Compilers for SPARC, *SunTechnology,* Vol. 1, No. 3, summer 1988, pp. 64–77; also appeared in [HalB90], pp. 41–68, and in [Stal90], pp. 160–173.

[Much91] Muchnick, Steven S. Optimization in the SPARCompilers, *Proc. of the Sun Users' Group Conf.,* Atlanta, GA, June 1991, pp. 81–99; also appeared in *Proc. of Sun USER '91,* Birmingham, England, Sept. 1991, pp. 117–136, and in *README,* Sun Users' Group, 1991, pp. 1–13 and 20–23.

[MucJ81] Muchnick, Steven S. and Neil D. Jones (eds.). *Program Flow Analysis: Theory and Applications,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Myer81] Myers, E.A. Precise Interprocedural Data Flow Analysis Framework, in [POPL81], pp. 219–230.

[Nick90] Nickerson, Brian. Graph Coloring Register Allocation for Processors with Multi-Register Operands, in [PLDI90], pp. 40–52.

[Nico86] Nicolau, Alexandru. A Fine-Grain Parallelizing Compiler, Tech. Rept. TR-86-792, Dept. of Comp. Sci., Cornell Univ., Ithaca, NY, Dec. 1986.

[OBrH90] O'Brien, Kevin, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, and Matthew Zaleski. Advanced Compiler Technology for the RISC System/6000 Architecture, in Misra, Mamata (ed.). *IBM RISC System/6000 Technology,* Publication SA23-2619, IBM Corp., Austin, TX, 1990, pp. 154–161.

[OBrO95] O'Brien, Kevin, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ron Unrau. XIL and YIL: The Intermediate Languages of TOBEY, *Proc. of the ACM SIGPLAN Workshop on Intermediate Representations,* San Francisco, CA, Jan. 1995, published as Microsoft Tech. Rept. MSR-TR-95-01, Microsoft Corp., Redmond, WA, Jan. 1995, and as *SIGPLAN Notices,* Vol. 30, No. 3, Mar. 1995, pp. 71–82.

[PaiS77] Paige, Bob and Jack T. Schwartz. Expression Continuity and the Formal Differentiation of Algorithms, in [POPL77], pp. 58–71.

[Patt95] Patterson, Jason R.C. Accurate Static Branch Prediction by Value Range Propagation, in [PLDI95], pp. 67–78.

bbs.theithome.com

[Pele88] Pelegri-Llopart, Eduardo. *Rewrite Systems, Pattern Matching, and Code Generation*, Ph.D. dissertation, Rept. No. UCB/CSD 88/423, Comp. Sci. Division, Univ. of California, Berkeley, CA, June 1988.

[PelG88] Pelegri-Llopart, Eduardo and Susan L. Graham. Code Generation for Expression Trees: An Application of BURS Theory, in [POPL88], pp. 294–308.

[Pent94] *Pentium Family User's Manual, Volume 3: Architecture and Programming Manual*, Intel Corp., Mount Prospect, IL, 1994.

[PetH90] Pettis, Karl and Robert C. Hansen. Profile Guided Code Positioning, in [PLDI90], pp. 16–27.

[Pint93] Pinter, Shlomit S. Register Allocation with Instruction Scheduling: A New Approach, in [PLDI93], pp. 248–257.

[PLDI88] *Proc. of the SIGPLAN '88 Symp. on Programming Language Design and Implementation*, Atlanta, GA, published as *SIGPLAN Notices*, Vol. 23, No. 7, June 1988.

[PLDI89] *Proc. of the SIGPLAN '89 Symp. on Programming Language Design and Implementation*, Portland, OR, published as *SIGPLAN Notices*, Vol. 24, No. 7, July 1989.

[PLDI90] *Proc. of the SIGPLAN '90 Symp. on Programming Language Design and Implementation*, White Plains, NY, published as *SIGPLAN Notices*, Vol. 25, No. 6, June 1990.

[PLDI91] *Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation*, Toronto, Ontario, published as *SIGPLAN Notices*, Vol. 26, No. 6, June 1991.

[PLDI92] *Proc. of the SIGPLAN '92 Symp. on Programming Language Design and Implementation*, San Francisco, CA, published as *SIGPLAN Notices*, Vol. 27, No. 7, July 1992.

[PLDI93] *Proc. of the SIGPLAN '93 Symp. on Programming Language Design and Implementation*, Albuquerque, NM, published as *SIGPLAN Notices*, Vol. 28, No. 7, July 1993.

[PLDI94] *Proc. of the SIGPLAN '94 Conf. on Programming Language Design and Implementation*, Orlando, FL, published as *SIGPLAN Notices*, Vol. 29, No. 6, June 1994.

[PLDI95] *Proc. of the SIGPLAN '95 Conf. on Programming Language Design and Implementation*, La Jolla, CA, published as *SIGPLAN Notices*, Vol. 30, No. 6, June 1995.

[PLDI96] *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, Philadelphia, PA, published as *SIGPLAN Notices*, Vol. 31, No. 5, May 1996.

[POPL73] *Conf. Record of the ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Boston, MA, Oct. 1973.

[POPL75] *Conf. Record of the 2nd ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Palo Alto, CA, Jan. 1975.

[POPL76] *Conf. Record of the 3rd ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Atlanta, GA, Jan. 1976.

[POPL77] *Conf. Record of the 4th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Los Angeles, CA, Jan. 1977.

[POPL78] *Conf. Record of the 5th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Tucson, AZ, Jan. 1978.

[POPL79] *Conf. Record of the 6th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, San Antonio, TX, Jan. 1979.

[POPL80] *Conf. Record of the 7th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Las Vegas, NV, Jan. 1980.

[POPL81] *Conf. Record of the 8th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Williamsburg, VA, Jan. 1981.

[POPL82] *Conf. Record of the 9th ACM SIGACT/SIGPLAN Symp. on Principles of Programming*

bbs.theithome.com

Languages, Albuquerque, NM, Jan. 1982.

[POPL84] *Conf. Record of the 11th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Salt Lake City, UT, Jan. 1984.

[POPL86] *Conf. Record of the 13th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1986.

[POPL88] *Conf. Record of the 15th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, San Diego, CA, Jan. 1988.

[POPL89] *Conf. Record of the 16th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Austin, TX, Jan. 1989.

[POPL90] *Conf. Record of the 17th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, San Francisco, CA, Jan. 1990.

[POPL91] *Conf. Record of the 18th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Orlando, FL, Jan. 1991.

[POPL92] *Conf. Record of the 19th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Albuquerque, NM, Jan. 1992.

[POPL94] *Conf. Record of the 21st ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, Portland, OR, Jan. 1994.

[POWE90] *POWER Processor Architecture, Version 1.52*, IBM Corp., Austin, TX, Feb. 1990.

[Powe93] *PowerPC Architecture*, first edition, IBM Corp., Austin, TX, May 1993.

[ProF94] Proebsting, Todd A. and Christopher W. Fraser. Detecting Pipeline Structural Hazards Quickly, in [POPL94], pp. 280–286.

[PugW92] Pugh, William and David Wonnacott. Eliminating False Data Dependences Using the Omega Test, in [PLDI92], pp. 140–151.

[Radi82] Radin, George. The 801 Minicomputer, in [ASPL82], pp. 39–47.

[RaoS95] Rao, Suresh, William A. Savage, and Kevin J. Smith. Personal communication, Intel Corp., Santa Clara, CA, Mar. 1995.

[RauG81] Rau, B.R. and C.D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing, *Proc. of the 14th Annual Microprogramming Workshop*, Chatham, MA, published as *SIGMicro Newsletter*, Vol. 12, No. 4, Dec. 1981, pp. 183–198.

[ReiL77] Reif, John R. and Harry R. Lewis. Symbolic Evaluation and the Global Value Graph, in [POPL77], pp. 104–118.

[ReiL86] Reif, John R. and Harry R. Lewis. Efficient Symbolic Analysis of Programs, *J. of Comp. and System Sci.*, Vol. 32, No. 3, June 1986, pp. 280–313.

[Reyn68] Reynolds, John C. Automatic Computation of Data Set Definitions, *Proc. of the IFIP Congress 1968*, Aug. 1968, pp. B69–B73.

[RicG89a] Richardson, Stephen E. and Mahadevan Ganapathi. Interprocedural Optimization: Experimental Results, *Software—Practice and Experience*, Vol. 19, No. 2, Feb. 1989, pp. 149–169.

[RicG89b] Richardson, Stephen E. and Mahadevan Ganapathi. Interprocedural Analysis vs. Procedure Integration, *Information Processing Letters*, Vol. 32, Aug. 1989, pp. 137–142.

[Rich91] Richardson, Stephen C. *Evaluating Interprocedural Code Optimization Techniques*, Ph.D. dissertation, Tech. Rept. CSL-TR-91-460, Comp. Sci. Lab., Stanford Univ., Stanford, CA, Feb. 1991.

[RogL92] Rogers, Anne and Kai Li. Software Support for Speculative Loads, in [ASPL92], pp. 38–50.

bbs.theithome.com

[Rose77] Rosen, Barry K. High-Level Data Flow Analysis, *CACM*, Vol. 20, No. 10, Oct. 1977, pp. 712–724.

[Rose79] Rosen, Barry K. Data Flow Analysis for Procedural Languages, *JACM*, Vol. 26, No. 2, Apr. 1977, pp. 322–344.

[Rose81] Rosen, Barry K. Degrees of Availability as an Introduction to the General Theory of Data Flow Analysis, in [MucJ81], pp. 55–76.

[RutG96] Ruttenberg, John, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler, in [PLDI96], pp. 1–11.

[Ryma82] Rymarczyk, J.W. Coding Guidelines for Pipelined Processors, in [ASPL82], pp. 12–19.

[SanO90] Santhanam, Vatsa and Daryl Odnert. Register Allocation Across Procedure and Module Boundaries, in [PLDI90], pp. 28–39.

[Sava95] Savage, William A. Personal communication, Intel Corp., Santa Clara, CA, Sept. 1995.

[Schl91] Schlansker, Michael. Compilation for VLIW and Superscalar Processors, tutorial notes, Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, Apr. 1991.

[SchS79] Schwartz, Jacob T. and Micha Sharir. A Design for Optimizations of the Bitvectoring Class, Comp. Sci. Rept. No. 17, Courant Inst. of Math. Sci., New York Univ., New York, NY, 1979.

[Schw73] Schwartz, Jacob T. *On Programming: An Interim Rept. on the SETL Project*, Installment II, Comp. Sci. Dept., Courant Inst. of Math. Sci., New York Univ., New York, NY, 1973.

[SetU70] Sethi, Ravi and Jeffrey D. Ullman. The Generation of Optimal Code for Arithmetic Expressions, *JACM*, Vol. 17, No. 4, Oct. 1970, pp. 715–728.

[Shar80] Sharir, Micha. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers, *Computer Languages*, Vol. 5, Nos. 3/4, 1980, pp. 141–153.

[ShaS69] Shapiro, R.M. and H. Saint. The Representation of Algorithms, Tech. Rept. RADC-TR-69-313, Volume II, Rome Air Development Center, Griffiss Air Force Base, NY, Sept. 1969; also published as Tech. Rept. CA-7002-1432, Massachusetts Computer Associates, Wakefield, MA, Feb. 1970.

[ShiP89] Shieh, J.J. and C.A. Papachristou. On Reordering Instruction Streams for Pipelined Processors, *Proc. of the 22nd Annual Intl. Symp. on Microarchitecture*, Dublin, Ireland, Aug. 1989, pp. 199–206.

[Simp96] Simpson, Loren Taylor. *Value-Driven Redundancy Elimination*, Ph.D. thesis, Dept. of Comp. Sci., Rice Univ., Houston, TX, Apr. 1996.

[SitC92] Sites, Richard L., Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary Translation, *Digital Tech. J.*, Vol. 4, No. 4, special issue, 1992; also appeared in slightly different form in *CACM*, Vol. 36, No. 2, Feb. 1993, pp. 69–81.

[SmoK91] Smotherman, Mark, Sanjay Krishnamurthy, P.S. Aravind, and David Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling, *Proc. of the 24th Annual Intl. Symp. on Microarchitecture*, Albuquerque, NM, Nov. 1991, pp. 93–102.

[SPAR92] *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[SPEC89] *SPEC newsletter*, Vol. 1, No. 1, Systems Performance Evaluation Cooperative, Fremont, CA, 1989.

[SriW93] Srivastava, Amitabh and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time, *J. of Programming Languages*, Vol. 1, No. 1, 1993, pp. 1–18.

[SriW94] Srivastava, Amitabh and David W. Wall. Link-Time Optimization of Address Calculation on a 64-Bit Architecture, in [PLDI94], pp. 49–60.

bbs.theithome.com

[Stal90] Stallings, William. *Reduced Instruction Set Computers,* second edition, IEEE Comp. Society Press, Los Alamitos, CA, 1990.

[Stee84] Steele, Guy L., Jr. *COMMON LISP: The Language,* Digital Press, Burlington, MA, 1984.

[SteH89] Steenkiste, Peter and John Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP, *ACM TOPLAS,* Vol. 11, No. 1, Jan. 1989, pp. 1–32.

[Stro88] Stroustrup, Bjarne. Type-Safe Linkage for C++, *Computing Systems,* Vol. 6, No. 4, 1988, pp. 371–404.

[Tarj72] Tarjan, Robert Endre. Depth First Search and Linear Graph Algorithms, *SIAM J. of Computing,* Vol. 1, No. 2, 1972, pp. 146–160.

[Tarj74] Tarjan, Robert Endre. Testing Flow Graph Reducibility, *J. of Comp. and System Sci.,* Vol. 9, No. 4, Dec. 1974, pp. 355–365.

[Tarj81] Tarjan, Robert Endre. Fast Algorithms for Solving Path Problems, *JACM,* Vol. 28, No. 3, July 1981, pp. 591–642.

[Tene74a] Tenenbaum, Aaron. Type Determination for Very High Level Languages, Rept. NSO-3, Comp. Sci. Dept., New York Univ., New York, NY, Oct. 1974.

[Tene74b] Tenenbaum, Aaron. Compile Time Type Determination in SETL, *Proc. of the ACM Annual Conf.,* San Diego, CA, Nov. 1974, pp. 95–100.

[Thom92] Thompson, Carol. Personal communication, Hewlett-Packard, Cupertino, CA, May 1992.

[Tiem89] Tiemann, Michael D. The GNU Instruction Scheduler, CS 343 Report, Dept. of Comp. Sci., Stanford Univ., Stanford, CA, June 1989; also appeared in an updated form as Cygnus Tech. Rept. CTR-0, Cygnus Support, Mountain View, CA, 1989.

[Tjia93] Tjiang, Steven W.K. *Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers,* Ph.D. dissertation, Dept. of Comp. Sci., Stanford Univ., Stanford, CA, July 1993.

[TjiH92] Tjiang, Steven W.K. and John L. Hennessy. Sharlit—A Tool for Building Optimizers, in [PLDI92], pp. 82–93.

[TjiW91] Tjiang, Steven W.K., Michael E. Wolf, Monica S. Lam, K.L. Pieper, and John L. Hennessy. Integrating Scalar Optimization and Parallelization, *Proc. of 4th Workshop on Languages and Compilers for Parallel Computing,* Santa Clara, CA, Aug. 1991, pp. 137–151.

[Towl76] Towle, Robert A. *Control and Data Dependence for Program Transformations,* Ph.D. thesis, Dept. of Comp. Sci., Univ. of Illinois, Champaign-Urbana, IL, Mar. 1976.

[Ullm73] Ullman, Jeffrey D. Fast Algorithms for the Elimination of Common Subexpressions, *Acta Informatica,* Vol. 2, Fasc. 3, July 1973, pp. 191–213.

[Unga87] Ungar, David M. *The Design and Evaluation of a High Performance Smalltalk System,* MIT Press, Cambridge, MA, 1987.

[UngS91] Ungar, David M. and Randall B. Smith. SELF: The Power of Simplicity, *Lisp and Symbolic Computation,* Vol. 4, No. 3, June 1991, pp. 187–205.

[Unic90] Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding, Version 1.0,* Addison-Wesley, Reading, MA, 1990.

[UNIX90a] UNIX Software Operation. *System V Application Binary Interface,* UNIX Press/Prentice-Hall, Englewood Cliffs, NJ, 1990.

[UNIX90b] UNIX Software Operation. *System V Application Binary Interface: Motorola 88000 Processor Supplement,* UNIX Press/Prentice-Hall, Englewood Cliffs, NJ, 1990.

[UNIX90c] UNIX Software Operation. *System V Application Binary Interface: SPARC Processor Supplement,* UNIX Press/Prentice-Hall, Englewood Cliffs, NJ, 1990.

bbs.theithome.com

[UNIX91] UNIX System Labs. *System V Application Binary Interface: Intel i860 Processor Supplement,* UNIX Press/Prentice-Hall, Englewood Cliffs, NJ, 1991.

[UNIX93] UNIX System Labs. *System V Application Binary Interface: Intel 386 Architecture Processor Supplement,* UNIX Press/Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Wall86] Wall, David W. Global Register Allocation at Link Time, in [Comp86], pp. 264–275.

[Wall88] Wallace, David R. Dependence Among Multi-Dimensional Array References, *Proc. of the 1988 ACM Intl. Conf. on Supercomputing,* St. Malo, France, July 1988, pp. 418–428.

[Wall91] Wall, David W. Predicting Program Behavior from Real or Estimated Profiles, in [PLDI91], pp. 59–70.

[Wall92] Wallace, David R. Cross-Block Scheduling Using the Extended Dependence Graph, *Proc. of the 1992 Intl. Conf. on Supercomputing,* Washington, DC, July 1992, pp. 72–81.

[WanE93] Wang, Jian and Christine Eisenbeis. Decomposed Software Pipelining, Tech. Rept. No. 1838, INRIA, Le Chesnay, France, Jan. 1993.

[Warr90] Warren, Henry S. Instruction Scheduling for the IBM RISC System/6000, in [IBMJ90], pp. 85–92.

[WeaG94] Weaver, David L. and Tom Germond. *The SPARC Architecture Manual, Version 9,* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[WegZ91] Wegman, Mark N. and F. Kenneth Zadeck. Constant Propagation with Conditional Branches, *ACM TOPLAS,* Vol. 13, No. 2, Apr. 1991, pp. 181–210.

[Weic84] Weicker, Reinhold P. Dhrystone: A Synthetic Systems Programming Benchmark, *CACM,* Vol. 27, No. 10, Oct. 1984, pp. 1013–1030.

[WeiC94] Weise, Daniel, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation Without Taxation, in [POPL94], pp. 287–296.

[Weih80] Weihl, William E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables, in [POPL80], pp. 83–94.

[Wexe81] Wexelblat, Richard L. *History of Programming Languages,* Academic Press, New York, NY, 1981.

[WhiS90] Whitfield, Debbie and Mary Lou Soffa. An Approach to Ordering Optimizing Transformations, *Proc. of the Second ACM Symp. on Principles and Practice of Parallel Programming,* Seattle, WA, published as *SIGPLAN Notices,* Vol. 25, No. 3, Mar. 1990, pp. 137–146.

[Wism94] Wismüller, Roland. Debugging Globally Optimized Programs Using Data Flow Analysis, in [PLDI94], pp. 278–289.

[Wolf89a] Wolfe, Michael R. More Iteration Space Tiling, Tech. Rept. No. CS/E 89-003, Dept. of Comp. Sci. and Engg., Oregon Graduate Inst., Beaverton, OR, May 1989.

[Wolf89b] Wolfe, Michael R. *Optimizing Supercompilers for Supercomputers,* MIT Press, Cambridge, MA, 1989.

[Wolf90] Wolfe,. Michael R. Scalar vs. Parallel Optimizations, Tech. Rept. No. CS/E 90-010, Dept. of Comp. Sci. and Engg., Oregon Graduate Inst., Beaverton, OR, July 1990.

[Wolf92] Wolf, Michael E. *Improving Locality and Parallelism in Nested Loops,* Ph.D. dissertation, Tech. Rept. CSL-TR-92-538, Comp. Systems Lab., Stanford Univ., Stanford, CA, Aug. 1992.

[Wolf96] Wolfe, Michael R. *High-Performance Compilers for Parallel Computing,* Addison-Wesley, Redwood City, CA, 1996.

[WolH90] Wolfe, Michael R. and Robert Halstead (eds.). *Proc. of a Workshop on Parallelism in the Presence of Pointers and Dynamically Allocated Objects,* Tech. Note SRC-TN-90-292, Inst. for Defense Analyses Supercomputing Research Ctr., Bowie, MD, Mar. 1990.

bbs.theithome.com

[WolL91] Wolf, Michael E. and Monica S. Lam. A Data Locality Optimizing Algorithm, in [PLDI91], pp. 30–44.

[WolT90] Wolfe, Michael R. and Chau-Wen Tseng. The Power Test for Data Dependence, Tech. Rept. No. CS/E 90-015, Dept. of Comp. Sci. and Engg., Oregon Graduate Inst., Beaverton, OR, Aug. 1990.

[WulJ75] Wulf, William A., Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler,* American Elsevier, New York, NY, 1975.

[WulR71] Wulf, William A., David B. Russell, and A. Nico Habermann. BLISS: A Language for Systems Programming, *CACM*, Vol. 14, No. 12, Dec. 1971, pp. 780–790.

[Zade84] Zadeck, F. Kenneth. Incremental Data Flow Analysis in a Structured Program Editor, in [Comp84], pp. 132–143.

[ZimC91] Zima, Hans with Barbara Chapman. *Supercompilers for Parallel and Vector Machines*, ACM Press/Addison-Wesley, Reading, MA, 1991.

'Begin at the beginning, the King said, gravely,
'and go until you come to the end; then stop.'
—Carrol, Lewis, *Alice in Wonderland*, Chap. 11.

bbs.theithome.com

版权公司制作 请勿盗作者版权

索　引

注：索引中的页码为英文原书页码，即本书页边所标页码。

符　号

Ox, begins hexadecimal number, in HIR, MIR, and LIR (Ox, 十六进制数前缀，HIR、MIR和LIR中), 74, 76

+ (addition operator), in ICAN (+ (加法运算符)，ICAN中)，27, 29

+ (addition operator), in HIR, MIR, and LIR (+ (加法运算符)，HIR、MIR和LIR中)， 74

| (alternation operator), in XBNF notation (| (选择运算符)，XBNF表示中)，19, 20

δ^a, antidependence operator (δ^a，反依赖运算符)，268

± (arbitrary distance), in direction vectors (± (任意距离)，方向向量中)，278

* (arbitrary distance), in direction vectors (* (任意距离)，方向向量中)，278

?, arbitrary relational operator (?，任意关系运算符)，449, 456

[...] (array constant delimiters), in ICAN ([...] (数组常数分界符)，ICAN中)，24, 27, 30

← (assignment operator), in HIR, MIR, and LIR (← (赋值运算符)，HIR、MIR和LIR中)，73, 74, 79, 80

:= (assignment operator), in ICAN (:= (赋值运算符)，ICAN中)，24, 36, 37

B (back-edge label)，in depth-first presentation of flowgraph (B (后向边标号)，流图的深度为主表示中)，178

B (binding graph)，in interprocedural analysis (B (结合图)，过程间分析中)，623-627, 643-646

%r (carriage return character), in ICAN (%r (回车字符)，ICAN中)，29

'...' (character delimiters), in ICAN ('...' (字符分界符)，ICAN中)，28,29

"..." (character string delimiters), in ICAN ("..." (字符串分界符)，ICAN中)，32

|| (comment delimiter), in HIR，MIR, LIR (|| (注释分界符)，HIR、MIR、LIR中)，76

|| (comment delimiter),in ICAN (|| (注释分界符)，ICAN中)，21

← (...) (conditional assignment operator), in HIR, MIR, and LIR (←(...) (条件赋值运算符)，HIR、MIR和LIR中)，74, 75

δ^c control-dependence operator (δ^c，控制依赖关系运算符)，267-268

C (cross-edge label)，in depth-first presentation of flowgraph (C (横向边标号)，流图的深度为主表示中)，178

D, ends double-precision floating-point number in HIR, MIR, and LIR (D (结束双精度浮点数)，HIR、MIR和LIR中)， 74

⇒(defined to be operator), in machine grammars (⇒ (定义运算符)，机器描述语法中)，140

→ (defined to be operator), in XBNF notation (→ (定义运算符)，XBNF表示中)，19

<...>, dependence vector delimiters (<...>，依赖向量的分界符)，277, 278

$\overset{*}{\Rightarrow}$(derives operator), in machine grammars ($\overset{*}{\Rightarrow}$ (导出运算符)，机器描述语法中)，147

<...>, direction vector delimiters(<...>,方向向量分界符)，277-278

<...>, distance vector delimiters(<...>,距离向量分界符)，278

/ (division operator), in HIR, MIR, and LIR (/ (除法运算符)，HIR、MIR和LIR中)，74

/ (division operator), in ICAN (/ (除法运算符)，ICAN中)，27, 29

∤, does not divide operator (∤ ，不能整除运算符)，281

...→..., edge in directed graph (...→...，有向图中的边)，175

[] (empty sequence value), in ICAN ([] (空序列值)，ICAN中)，24, 27, 32

∅ (empty set value), in ICAN (∅ (空集合值)，ICAN中)， 24, 27, 31

ϵ (empty string), in grammars (ϵ (空串)，语法中)，19, 140

{...} (enumerated type delimiters), in ICAN ({...} (枚举

bbs.theithome.com

超星公司制作 请尊重作者版权

类型分界符)，ICAN中)，23, 26, 29

= (equal to operator), in HIR, MIR, and LIR (=（相等运算符)，HIR、MIR和LIR中)，74

= (equal to operator), in ICAN (=（相等运算符)，ICAN中)，27，28，29，30

∃ (existential quantifier), in ICAN（∃（存在量词)，ICAN中)，27, 28

E (exponent delimiter), in HIR, MIR, and LIR (E（指数分隔符），HIR、MIR和LIR中)，74

E (exponent delimiter), in ICAN (E（指数分隔符），ICAN中)，28, 29

↑ (exponentiation operator), in ICAN（↑（幂运算符)，ICAN中)，27, 29

. (field indirection operator), in HIR, MIR, and LIR (.（域间接运算符)，HIR、MIR和LIR中)，74

. (field selection operator),in HIR, MIR, and LIR (.（域选择运算符)，HIR、MIR和LIR中)，74,79

. (field selection operator),in ICAN (.（域选择运算符)，ICAN中)，27, 34

δ^f, flow-dependence operator (δ^f，流依赖运算符)，268

F (forward-edge label), in depth-first presentation of flowgraph (F（前向边标号)，流图的深度为主表示中)，178

∘, function composition (∘，函数复合)，235

→ (function type constructor),in ICAN（→（函数类型构造符)，ICAN中)，23, 26, 28, 34

> (greater than operator), in HIR, MIR, and LIR (>（大于运算符)，HIR、MIR和LIR中)，74, 75

> (greater than operator), in ICAN (>（大于运算符)，ICAN中)，27, 29

>= (greater than or equal to operator), in HIR, MIR,and LIR (>=（大于等于运算符)，HIR、MIR和LIR中)，74, 75

≥ (greater than or equal to operator), in ICAN（≥（大于等于运算符)，ICAN中)，27,29

{...} (grouping operator), in XBNF notation ({...}（成组运算符)，XBNF表示中)，19

∞ (infinite value), in ICAN（∞（无穷值)，ICAN中)，24, 28, 29

δ^i, input-dependence operator (δ^i，输入依赖运算符)，268

★ (Kleene closure operator, unary postfix), applied to functions (★（克林闭包运算符，一元后缀)，作用于函数)，235

: (label separator), in HIR，MIR，and LIR (:（标号运算符)，HIR、MIR和LIR中)，74, 75, 79, 80

: (label separator), in ICAN (:（标号运算符)，ICAN中)，37

⊥, lattice bottom value（⊥，格的底值)，223

⊥(lattice bottom value), in ICAN（⊥（格的底值)，ICAN中)，225, 364

⊐, lattice greater than operator（⊐，格的大于运算符)，225

⊒, lattice greater than or equal to operator（⊒，格的大于等于运算符)，225

⊔, lattice join operator（⊔，格的并运算符)，223

⊏, lattice less than operator（⊏，格的小于运算符)，225

⊏ (lattice less than operator), in ICAN（⊏，(格的小于运算符)，ICAN中)，639

⊑, lattice less than or equal to operator（⊑，格的小于等于运算符)，225

⊓, lattice meet operator（⊓，格的交运算符)，223

⊓ (lattice meet operator), in ICAN（⊓（格的交运算符)，ICAN中)，232, 367, 639

⊤, lattice top value（⊤，格的顶值)，223

⊤ (lattice top value), in ICAN（⊤（格的顶值)，ICAN中)，225, 232, 364

< (less than operator), in HIR, MIR, and LIR (<（小于运算符)，HIR、MIR和LIR中)，74

< (less than operator), in ICAN (<（小于运算符)，ICAN中)，27, 29

<= (less than or equal to operator), in HIR, MIR, and LIR (<=（小于等于运算符)，HIR、MIR和LIR中)，74

≤ (less than or equal to operator), in ICAN（≤（小于等于运算符)、ICAN中)，27, 29

≺ lexicographically less than operator（≺，词典序小于运算符)，275

⪯, lexicographically less than or equal to operator（⪯，词典序小于等于运算符)，276

↑ (load operator, unary), in machine grammars（↑（取数运算符，一元操作)，机器语法中)，139

& (logical and operator), in ICAN (&（逻辑与运算符)，ICAN中)，27, 28

! (logical not operator, unary), in HIR, MIR, and LIR (!（逻辑非运算符)，HIR、MIR和LIR中)，74, 75

! (logical not operator, unary), in ICAN (!（逻辑非运算符)，ICAN中)，27, 28

∨ (logical or operator), in ICAN（∨（逻辑或运算符)，ICAN中)，27, 28

∈ (member of set operator), in ICAN（∈（集合成员运算符)，ICAN中)，27, 28, 31, 34

% (modulo operator), in ICAN (%（模运算符)，ICAN中)，27, 29

* (multiplication operator), in HIR, MIR, and LIR (*

bbs.theithome.com

超星公司制作请尊重作者版权

(乘法运算符)，HIR、MIR和LIR中)，74, 75
* (multiplication operator), in ICAN (* (乘法运算符)，ICAN中)，27, 29
- (negation operator, unary), in HIR, MIR, and LIR(- (取负运算符，一元操作)，HIR、MIR和LIR中)，74, 75
- (negation operator, unary), in ICAN (- (取负运算符，一元操作)，ICAN中)，27, 29
> (negative distance), in direction vectors (> (负距离)，方向向量中)，278
- (negative distance), in direction vectors (- (负距离)，方向向量中)，278
+ (non-empty closure operator, unary postfix), applied to functions (+ (非空闭包运算符，一元后缀)，作用于函数)，235
!= (not equal to operator), in HIR, MIR, and LIR (!= (不相等运算符)，HIR、MIR和LIR中)，74, 75
≠ (not equal to operator), in ICAN (≠ (不相等运算符)，ICAN中)，27, 28, 29, 30
∉ (not member of set operator), in ICAN (∉ (不属于集合成员运算符)，ICAN中)，27, 31
+ (one or more copies operator, unary postfix), in XBNF notation (+ (一次以上复制运算符，一元后缀)，XBNF表示中)，19, 20
[...] (optional operator), in XBNF notation ([...] (可选运算符)，XBNF表示中)，19, 20
δ^o, output-dependence operator (δ^o，输出依赖运算符)，268
∏ (pair binding graph), in interprocedural alias analysis (∏ (成对结合图)，过程间别名分析中)，643,649-653
%fr*i*, PA-RISC floating-point register *i* (%fr*i*，PA-RISC浮点寄存器*i*)，754
%fr*i* L, PA-RISC floating-point register *i*, left part (%fr*i* L，PA-RISC浮点寄存器*i*的左部)，754
%fr*i*R, PA-RISC floating-point register *i*, right part (%fr*i*R，PA-RISC浮点寄存器*i*的右部)，754
%r*i*, PA- RISC integer register *i* (%r*i*，PA-RISC整数寄存器*i*)，754
%% (percent character), in ICAN (%% (百分号字符)，ICAN中)，29
Φ function, static single assignment form (Φ函数，静态单一赋值形式)，252, 253
* (pointer indirection operator, unary), in HIR, MIR, and LIR (* (指针间接运算符，一元)，HIR、MIR和LIR中)，74, 75, 79
. (position indicator), in LR(1) items (. (位置指示符)，LR(1)项目中)，144
< (positive distance), in direction vectors (< (正距离)，方向向量中)，278
+ (positive distance), in direction vectors (+ (正距离)，方向向量中)，278
◁, precedes-in-execution-order operator (◁，执行顺序上的先于运算符)，267, 275
%" (quotation mark character), in ICAN (%" (引号字符)，ICAN中)，29
<...> (record constant delimiters), in ICAN (<...> (记录常数分界符)，ICAN中)，27, 33
{...} (record type constructor), in ICAN ({...} (记录类型构造符)，ICAN中)，23, 26, 33
◆ (select from set operator, unary}, in ICAN (◆ (从集合中选择运算符，一元)，ICAN中)，24, 27, 31
⋈ (separated list operator), in XBNF notation (⋈ (分开的表运算符)，XBNF表示中)，19, 20
⊕ (sequence concatenation operator), in ICAN (⊕ (序列连接运算符)，ICAN中)，24, 27, 32
[...] (sequence constant delimiters), in ICAN ([...] (序列常数分界符)，ICAN中)，24, 27, 32
⊖ (sequence member deletion operator), in ICAN (⊖ (序列成员删除运算符)，ICAN中)，24, 27, 32
↓ (sequence member selection operator), in ICAN (↓ (序列成员选择运算符)，ICAN中)，27, 32
{...} (set constant delimiters), in ICAN ({...} (集合常数分界符)，ICAN中)，27, 31
∩ (set intersection operator}, in ICAN (∩ (集合交运算符)，ICAN中)，27, 31
∪ (set union operator), in ICAN (∪ (集合并运算符)，ICAN中)，27, 31
%' (single quotation mark character), in ICAN (%' (单引号字符)，ICAN中)，29
|...| (size operator), in ICAN (|...| (大小运算符)，ICAN中)，24, 27, 36
%f*i*, SPARC floating-point register *i* (%f*i*，SPARC浮点寄存器*i*)，748
%g*i*, SPARC integer general register *i* (%g*i*，SPARC整数通用寄存器*i*)，748
%i*i*, SPARC integer in register *i* (%i*i*，SPARC整数输入寄存器*i*)，748
%l*i*, SPARC integer local register *i* (%l*i*，SPARC整数局部寄存器*i*)，748
%o*i*, SPARC integer out register *i* (%o*i*，SPARC整数输出寄存器*i*)，748
%r*i*, SPARC integer register *i* (%r*i*, SPARC整数寄存器*i*)，748
←sp (speculative load operator), extension to LIR (←sp (前瞻取运算符)，扩充到LIR的)，547, 548
; (statement separator), in ICAN (; (语句分隔符)，ICAN中)，21,37
←(store operator), in machine grammars (← (存储运算

bbs.theithome.com

符），机器语法中），139
..(subscript range operator), in ICAN（..（下表范围运算符），ICAN中），26, 30
－(subtraction operator), in HIR, MIR, and LIR（－（减法运算符），HIR、MIR和LIR中），74, 75
－(subtraction operator), in ICAN（－（减法运算符），ICAN中），27, 29
<...> (tuple constant delimiters), in ICAN（<...>（元组常数分界符），ICAN中），27
@ (tuple element selection operator), in ICAN（@（元组元素选择操作符），ICAN中），24, 27, 33
× (tuple type constructor), in ICAN（×（元组类型构造符），ICAN中），23, 26, 28, 33
= (type definition operator), in ICAN（=（类型定义操作符），ICAN中），25, 26, 35
∪(union type constructor), in ICAN（∪（联合类型构造符），ICAN中），23, 26, 28, 34
∀(universal quantifier), in ICAN（∀（全体量词），ICAN中），27, 28
δ^{wl} (write-live dependence),（δ^{wl}（写－活跃依赖））571
= (zero distance), in direction vectors（=（零距离），方向向量中），278
* (zero or more copies operator, unary postfix), in XBNF notation（*（零或多次重复运算符，一元后缀），XBNF表示中），19, 20

A

ABI. *See* Application Binary Interface (ABI)standards（ABI，参见应用程序二进制接口（ABI）标准）
abstract flowgraphs（抽象流图），198
abstract nodes（抽象结点），198
abstract syntax tree（抽象语法树），70-71
accumulator-variable expansion（累加器变量扩张），562-564
Action/Next tables, Graham-Glanville code generator（Action/Next表，Graham-Glanville代码生成器），149-150, 158
acyclic control structures（无环控制结构）
  backward structural data-flow analysis（向后结构数据流分析），245,246
  forward structural data-flow analysis（向前结构数据流分析），239-240
  structural control-flow analysis（结构控制流分析），202-204, 207-208
acyclic test（无环测试），284
Ada
  call by value-result in（传值得结果），117
  enumerated value representation（枚举值表示），107
  scoping issues（作用域问题），52
  unnecessary bounds-checking elimination for（不必要边界检查的消除），454-457
addition operator (+)（加法运算符 (+)）
  in HIR, MIR, and LIR（HIR、MIR和LIR中），74
  in ICAN（ICAN中），27, 29
addressing expressions, algebraic simplification and reassociation（地址表达式、代数化简和重结合），334-341
addressing method, symbol-table variables（寻址方法，符号表变量），54-55
adjacency-list representation for interference graphs（冲突图的邻接表表示），487, 496-497, 498,499, 530
  examples（例），498, 514, 516, 517, 520
adjacency-matrix representation for interference graphs（冲突图的邻接矩阵表示），487, 495-496, 497, 530
  examples（例），496, 513-516, 519
affix-grammar-based code-generator generators（基于词缀文法的代码生成器的生成器），159-160
aggregation of global references（全局引用的聚合），663
  importance of（重要性），664
  order of optimizations（优化的顺序），665-666
Aho, Alfred V., 160, 165, 351
algebraic simplifications and reassociation（代数化简和重结合），333-343
  of addressing expressions（地址表达式的），334-341
algebraic simplifications（代数化简）
  defined（定义），333
  for binary operators（用于二元运算符），333
  for bit-field types（用于位域），333
  for Booleans（用于布尔量），333
  canonicalization（规范化），335-336
  of floating-point expressions（浮点表达式的），342-343
  order of optimizations（优化顺序），333, 372
  overview（概述），371
  reassociation, defined（重结合，定义），333
  for relational operators（用于关系运算符），333-334
  strength reductions（强度削弱），334
  tree transformations（树转换），337-341
  for unary operators（用于一元运算符），333
  using commutativity and associativity（使用交换率和结合率），334
ALGOL 60
  call by name in（传名字），118
  call by value in（传值），117
  labels as arguments（标号作为实参），118-119
ALGOL 68

bbs.theithome.com

超星公司制作 请尊重作者版权

call by reference in（传地址），118
call by value in（传值），117
algorithms（算法）. *See also specific types of analyses and optimizations*（同时参见分析和优化的各种特定类型）
binding graph construction（结合图构造），625, 643, 645
call graph nesting levels（调用图的嵌套层），629-630
call-graph construction（调用图构造），609-611
call-graph construction with procedure-valued variables（带过程值变量的调用图构造），612-618
code hoisting（代码提升），418-419
combining alias information with alias-free version（别名信息与免除别名的版本合并），651, 654
computing aliases for nonlocal variables（计算非局部变量的别名），643-646, 647
computing formal-parameter alias information（计算形式参数的别名信息），650-651, 653
constant folding（常数折叠），330
dead-code elimination（死代码删除），592-593, 596
dependence DAG construction（依赖DAG构造），273-274
depth-first search for dominator computation（必经结点计算的深度为主搜索），187
dominator computation using Dom_Comp（使用Dom_Comp的必经结点计算），182
dominator computation using Domin_Fast（使用Domin_Fast的必经结点计算），186-187
dynamic programming algorithms（动态规划算法），163-165
global common-subexpression elimination（全局公共子表达式删除），388-390, 391-394
GMOD( ), 630-631
Graham-Glanville code-generation algorithm（Graham-Glanville代码生成算法），142-143
greedy or maximal munch（贪婪的或贪吃的），141
IMOD$^+$ (), 628
instruction scheduling（指令调度），540-541
interprocedural constant propagation（过程间常数传播），637-639
intraprocedural code positioning（过程内的代码放置），678-681
inverting nonlocal alias function（颠倒非局部别名函数），646-647, 648
label evaluation for dominator computation（用于必经结点计算中的标号计算），188
linking for dominator computation（用于必经结点计算中的链接），188
local common-subexpression elimination（局部公共子表达式删除），380-381
local copy propagation（局部复写传播），356-358
loop-invariant code motion（循环不变代码外提），397-400, 403-404
marking parameter pairs in pair binding graph（标志成对结合图中的参数偶对），650, 652-653
MOST pipelining algorithm（MOST流水算法），567
natural loop（正常循环），192
pair binding graph construction（成对结合图构造），649-650
partitioning for global value numbering（全局值编号的划分），351-355
path-compression for dominator computation（必经结点计算的路径压缩），187
percolation scheduling（渗透调度），572
procedure sorting（过程排序），673-676
RBMOD() on binding graph（结合图的*RBMOD*()），626
sparse conditional constant propagation（稀有条件常数传播），364-366
strength reduction（强度削弱），436-438
strongly connected components（强连通分量），195
structural control-flow analysis（结构控制流分析），205-206
unreachable-code elimination（不可到达代码删除），580-581
unroll-and-compact software pipelining（展开－压实软件流水），557-558
window scheduling（窗口调度），552-553
worklist for iterative data-flow analysis（迭代数据流分析的工作表），232-233
alias analysis（别名分析），293-317. *See also* interprocedural alias analysis（同时参见过程间别名分析）
alias gatherer（别名搜集器），203-207, 315
alias propagator（别名传播器），307-314, 315
aliases in C（C中的别名），300-301,305-306
aliases in Fortran 77（Fortran 77的别名），298-299
aliases in Fortran 90（Fortran 90的别名），301-302, 305
aliases in Pascal（Pascal的别名），299-300, 304, 305
Alpha compilers（Alpha 编译器），729
cases（情形），295-296
described（描述），293
flow-insensitive may information（流不敏感可能信息），295
flow-insensitive must information（流不敏感一定信息），295
flow-sensitive may information（流敏感可能信息），

bbs.theithome.com

295-296
flow-sensitive must information（流敏感一定信息），296
flow-sensitive vs. flow-insensitive information（流敏感与流不敏感信息），294-295, 315
granularity of information（信息粒度），297, 315
importance of（重要性），293-294, 314-315
interprocedural（过程间的），641-656
may vs. must information（可能与一定信息），294, 315
sources of aliases（别名的原因），296-297
alias gatherer（别名搜集），203-207
aliases in C（C中的别名），305-307
described（描述），315
examples of aliasing concepts（别名概念的例子），303
nontermination（不终止），304
program points（程序点），303
sources of aliases（别名的原因），296-297
alias propagator（别名传播器），307-314
for C code（C代码情形），309-314
described（描述），296, 315
examples of aliasing concepts（别名概念的例子），303
Allen, Francis E., 449, 459
alloca(), pointers and（alloca()，指针），113
Alpern, Bowen, 348, 355
Alpha architecture（Alpha体系结构），726
Alpha compilers（Alpha 编译器），726-733
alias analysis（别名分析），729
assembly code examples（汇编代码的例子），730-733
assembly language（汇编语言），750-752
CIL code（CIL代码），727-728
code generator（代码生成器），730
data prefetching（数据预取），698
data-flow analysis（数据流分析），729
development（开发），726-727
EIL code（EIL代码），727, 728, 730
global optimizations（全局优化），729
inlining（内联），729
instruction prefetching（指令预取），672
languages supported（支持的语言），727
loop inversion（循环倒置），729
mixed model of optimization in（优化的混合方式），9
optimization levels（优化级别），728-729
order of optimizations（优化的顺序），705
peephole optimizations（窥孔优化），729, 730
register allocation（寄存器分配），483,485
alternation operator (|), in XBNF notation（选择运算符(|)，XBNF表示中），19, 20
ancestor（祖先），185
anticipatable registers, in shrink wrapping（可预见的寄存器，收缩包装中），473
anticipatable values（可预见的值），410-411
antidependence（反依赖）
definition（定义），268
operator (δ^a)（运算符δ^a），268
Application Binary Interface (ABI) standards（应用二进制接口（ABI）标准），105, 134
arbitrary distance, in direction vectors（任意距离，方向向量中）
±, 278
*, 278
arbitrary relational operator (?)（任意关系运算符(?)），449, 456
architectures（体系结构）
Alpha, 726
branch architectures（分支体系结构），533
Intel 386 family（Intel 386系列），734-735
loop unrolling and（循环展开），560-561, 562
pipeline architectures（流水线体系结构），531, 532-533
POWER and PowerPC（POWER和PowerPC），716-717
SPARC, 707-708
arguments（实参）. *See also* parameter-passing (run-time); parameters（同时参见参数传递（运行时）；哑参）
defined（定义），117
described（描述），111
array constant delimiters ([...]), in ICAN（数组常数分界符([...])，ICAN中），24, 27, 30
array...of, (array type constructor), in ICAN（array...of（数组类型构造符），ICAN中），23
array types (ICAN)（数组类型，ICAN的）
binary operators（二元运算符），30
constants（常数），30, 33
overview（概述），30
arrays（数组）
column-major order（列为主顺序），107
in copy propagation（复写传播），356
data-flow analysis（数据流分析），258-259
function representation（函数表示），764
HIR representation（HIR表示），68-69
last-write trees（最近写树），259
LIR representation（LIR表示），68-69

版权所有 翻印必究
bbs.theithome.com

MIR representation（MIR表示），68-69
row-major order（行为主顺序），107
run-time representation（运行时表示），107-108
scalar replacement of array elements（数组元素的标量替换），682-687
separable references（可分的引用），282
sparse set representation（稀疏集合表示），762-763
storage binding（存储绑定），58
unnecessary bounds-checking elimination（消除不必要的边界检查），454
weakly separable references（弱可分的引用），282, 283
ASCII, DEC VAX support（ASCII, DEC VAX支持），106
assembly code examples（汇编代码的例子）
Alpha compiler（Alpha编译器），730-733
Pentiums（奔腾），741-744
POWER compiler（POWER编译器），723-725
SPARC compilers（SPARC编译器），713-716
assembly language（汇编语言），747-755
Alpha, 750-752
Intel 386 family（Intel 386系列），752-753
PA-RISC, 753-755
POWER and PowerPC（POWER和PowerPC），749-750
relocatable binary form vs.（可重定位二进制形式），138
SPARC, 747-749
assignment operator（赋值运算符）
←, in HIR, MIR, and LIR（←，HIR、MIR和LIR中的），73, 74, 79, 80
:=, in ICAN（:=，ICAN中的），24, 36, 37
assignment statements (ICAN), overview(赋值语句(ICAN)，概述), 36-38
assignments（赋值）
HIR, 79
LIR, 80
MIR, 75
associativity, algebraic simplifications using（结合率，代数化简中的使用），334
attribute-grammar-based code-generator generators（基于属性文法的代码生成器的生成器），159-160
attributes of symbols in symbol tables（符号表中符号的属性），45-47
list of（表），46
Auslander, Marc, 804
automatic generation of code generators（代码生成器的自动生成），137-167. *See also* Graham-Glanville code-generator generator（同时参见Graham-Glanville代码生成器的生成器）
Graham-Glanville code-generator generator（Graham-Glanville代码生成器的生成器），139-158
issues in generating code（生成代码中的问题），137-138
overview（概述），6, 137-139
semantics-directed parsing（语义制导的分析），159-160
software resources（软件资源），769-770
syntax-directed technique（语法制导的技术），139-158
tree pattern matching and dynamic programming（树模式匹配和动态规划），160-165
automatic inlining（自动内联）. *See* procedure integration（参见过程集成）
automatic storage class（自动存储类），44
automating data-flow analyzer（数据流分析器自动化）
construction（构造），259-261
automating instruction-scheduler（指令调度器自动化）
generation（生成），543
available expressions（可用表达式）
analysis（分析），229
global common-subexpression elimination（全局公共子表达式删除），385-387, 390
local common-subexpression elimination（局部公共子表达式删除），379
available registers, in shrink wrapping（可用寄存器，收缩包装中），473

B

Babbage, Charles, 459
back edges（回边）
in depth-first presentation（深度为主表示中），178
label (B), in depth-first presentation of flowgraph（标号（B），流图的深度为主表示中），178
in natural loops（正常循环中），191
Backus-Naur Form, Extended (XBNF)（巴科斯－诺尔范式（XBNF)），19-20
backward data-flow analysis（向后数据流分析）
iterative（迭代的），229, 235
live variables analysis（活跃变量分析），445-447
structural data-flow analysis（结构数据流分析），244-247
balanced binary trees（平衡二叉树），48, 761,763
balanced scheduling（平衡式调度），545
Ball, Thomas, 598
Banerjee, Utpal, 289, 738

bbs.theithome.com

basic blocks（基本块）. *See also* basic-block scheduling; cross-block scheduling（同时参见基本块调度；跨基本块调度）
in copy propagation（复写传播中的），356
correspondence between bit-vector positions, definitions, and basic blocks（位向量的位置、定值和基本块之间的对应），219, 220
data-flow analysis and（数据流分析），218
dependence DAGs（依赖DAG），269-274
extended（扩展的），175-177, 361,565
identifying（标识），173-174
intraprocedural code positioning（过程内代码安置），677-681
local common-subexpression elimination（局部公共子表达式删除），382-384
local copy propagation for extended basic blocks（扩展基本块的局部复写传播），361
placement in instruction cache（放置在指令高速缓存中），676-677
predecessor（前驱），175,484
reverse extended（逆扩展的），175
successor（后继），175,484
value numbering applied to（用于值编号），344-348
basic induction variables（基本归纳变量），426, 427
basic-block scheduling（基本块调度）. *See also* instruction scheduling（同时参见指令调度）
balanced scheduling（平衡式调度），545
combining with branch scheduling（与分支调度结合），573
described（描述），531
filling stall cycles（填充停顿时钟周期），535
list scheduling（列表调度），535, 537-543
loop unrolling and（循环展开），532
order of optimizations（优化的顺序），574-575
performance and（性能），573
register allocation and（寄存器分配），545
register renaming and（寄存器重命名），532
trace scheduling（踪迹调度），569-570
variable expansion and（变量扩张），532
basis, of a class of induction variables（基，归纳变量类的），428
Bell, Ron, 670, 671
Bernstein, David, 521,548
bidirectional data-flow analysis（双向数据流分析），229
bin-packing, register allocation and（装包，寄存器分配），484-485, 730
binary operators（二元运算符）
algebraic simplifications（代数化简），333
MIR, 75
binary operators (ICAN)（二元运算符（ICAN））
array types（数组类型），30
boolean types（布尔类型），28
character types（字符类型），29
enumerated types（枚举类型），30
integer types（整数类型），29
real types（实数类型），29
record types（记录类型），34
sequence types（序列类型），32
set types（集合类型），31
tuple types（元组类型），33
union types（联合类型），34
binding graph（结合图）
flow-insensitive interprocedural alias analysis（流不敏感过程间别名分析），643, 645
flow-insensitive side-effect analysis（流不敏感副作用分析），623-627
pair binding graph（成对结合图），649-650
bit-field types, algebraic simplifications（位域类型，代数化简），333
bit vectors（位向量）
correspondence between bit-vector positions, definitions, and basic blocks（位向量的位置、定值和基本块之间的对应），219, 220
defined（定义），218-219
lattice of（格），224, 226
linked-list representation vs.（链表表示），757-759
sequence representation（序列表示），763
set operations（集合运算），759-760
set representation（集合表示），759-760
BLISS compilers（BLISS编译器）
for Alpha（Alpha的），727
reducible flowgraphs and（可归约流图），196
register allocation（寄存器分配），483,484, 528
block and procedure placement in instruction caches（指令高速缓存中基本块和过程的放置），676-677
block schema（块的图解），203
blocks, basic（块，基本的）. *See* basic blocks（同时参见基本块）
boolean ICAN type（ICAN布尔类型）
binary operators（二元运算符），28
Boolean-valued expressions（布尔值表达式），28
overview（概述），28
symbol-table entries（符号表项），46
Booleans（布尔量）
algebraic simplifications（代数化简），333
run-time support（运行时的支持），107

版权所有 侵权必究

bbs.theithome.com

超星公司制作 请尊重作者版权

boosting（上推），548
bottom of lattices（格的底），223
bottom-up rewriting systems(BURS)（自下而上重写系统（BURS）），165
bounds checking（边界检查），454. *See also* unnecessary bounds-checking elimination（同时参见消除不必要的边界检查）
braces（花括号）. *See* curly braces（参见花括号）
brackets（方括号）. *See* square brackets（参见方括号）
Bradlee, David G., 526
branch node（分支结点），175
branch optimizations（分支优化），589-590
  cases（情形），590
  described（描述），580
  detecting branches to branch instructions（检测分支到分支的指令），589
  order of optimizations（优化的顺序），589, 604
branch prediction（分支预测），597-599
  described（描述），580, 597
  dynamic methods（动态方法），597-598
  heuristics（启发式），598-599
  importance of（重要性），579, 603
  instruction prefetching and（指令预取），673
  loop and nonloop branches（循环和非循环分支），598-599
  order of optimizations（优化的顺序），604
  perfect static predictor（完美静态预测器），598
  static methods（静态方法），598
branch scheduling（分支调度），533-535
  branch architectures（分支体系结构），533
  combining with basic-block scheduling（与基本块调度结合），573
  described（描述），533
  filling delay slots（填充延迟槽），534-535
  filling stall cycles（填充停顿时钟周期），535
  flowchart（流程图），536
  nullification（作废），534-535
  order of optimizations（优化的顺序），574-575
  performance and（性能），573
breadth-first order（宽度为主顺序），181
breadth-first search（宽度为主搜索），181
Briggs, Preston, 355, 414, 485, 495, 523,525, 762
BURG, 769
BURS, 165
byte, usage in this book（字节，本书中的用法），16-17

C

C
  alias propagator（别名传播器），309-314
  aliases in（别名），300-301,305-307, 641
  basic-block boundaries（基本块边界），174
  call by reference in（传地址），118
  call by value in（传值），117
  local variable declarations（局部变量声明），58
  pointer aliasing（指针别名），294
  pointers（指针），258
  tail-recursion elimination in（尾递归删除），461,462
  well-behaved loops（规则循环），425
C++
  call by reference in（传地址），118
  call by value in（传值），117
  name mangling by `cfront` preprocessor（`cfront`预处理器的名字重塑），10
  scoping issues（作用域问题），52
caches（高速缓存）. *See also* data-cache optimization; instruction-cache optimization（同时参见数据高速缓存优化；指令高速缓存优化）
  combined or unified cache（合并的或统一的高速缓存），670
  data-cache impact（数据高速缓存的影响），670-671
  data-cache optimization（数据高速缓存优化），687-700
  effectiveness（效果），670
  instruction-cache impact（指令高速缓存的影响），672
  instruction-cache optimization（指令高速缓存优化），672-682
  stride values（跨步值），670
  translation-lookaside buffer (TLB)（后备转换缓冲区(TLB)），670
`call`（调用）
  LIR, 80
  MIR, 76
call-by-name parameter passing（传名字参数传递），118
call-by-reference parameter passing（传地址参数传递），117-118
call-by-result parameter passing（传结果参数传递），117
call by value（传值）
  interprocedural alias analysis and（过程间别名分析），654-656
  interprocedural optimization（过程间优化），656, 658
  parameter passing（参数传递），117
call-by-value-result parameter passing（传值得结果参数传递），117
call graphs（调用图）. *See also* interprocedural control-flow analysis（同时参见过程间控制流分析）
  interprocedural control-flow analysis（过程间控制流分析），609-618

bbs.theithome.com

interprocedural data-flow analysis（过程间数据流分析），628-631
procedure sorting（过程排序），673-674
Callahan, David, 525, 530, 634, 637, 694
callee-saved registers（被调用者保护的寄存器），120
caller-saved registers（调用者保护的寄存器），120
canonical form for loops（循环的规范形式），689
canonical loop test（规范循环的测试），275
canonicalization（规范化），335-336
capitalization, XBNF notation（大写，XBNF表示），19
Carr, Steve, 694
carriage return character (%r), in ICAN（回车字符(%r)，ICAN中），29
case statements (ICAN)（case语句（ICAN））
form（形式），39
internal delimiters（内部分界符），25
case studies（实例分析），705-745
Alpha compiler（Alpha编译器），726-733
example programs for（例程序），705-707
IBM XL compilers（IBM XL编译器），716-725
Intel reference compilers for 386 family（Intel 386系列的参考编译器），734-744
SPARC compilers（SPARC编译器），707-716
case/switch schema（case/switch图解），203
CDGs. *See* control-dependence graphs (CDGs, 参见控制依赖图(CDGs))
chain loops, eliminating in Graham-Glanville code generator（链循环，Graham-Glanville 代码生成器中的删除），152-154
Chaitin, Gregory, 485,494, 498
Chandra, Ashok, 804
`character`, ICAN type（字符，ICAN类型）
binary operators（二元运算符），29
delimiters (' ... ')（分界符（' ... ')），28, 29
escape sequences（转义序列），29
overview（概述），29
symbol-table entries（符号表项），46
character string delimiters ("..."), in ICAN（字符串分界符("...")，ICAN中），32
character strings, run-time representation（字符串，运行时表示），108-109
characters, run-time representation（字符，运行时表示），106, 108-109
Chow, Frederick, 473,524, 662
CIL code（CIL代码），727-728
circular scheduling（环调度），565
CISCS
aggregation of global references（全局引用的聚合），663
character representation supported（所支持的字符表示），106
instruction decomposing（指令分解），602
pipelining（流水），531
register assignment（寄存器指派），482
register usage（寄存器用法），110
class of induction variables（归纳变量的类），428
clique（集团），525
clique separators（分离子团），525-526
cloning, procedure（克隆，过程），607-608, 657
closed scopes（闭作用域），52-54
Cmelik, Robert, 770, 771
coalescing registers（合并寄存器）. *See* register coalescing（参见寄存器合并）
Cocke, John, 449, 485, 804
code generation（代码生成）. *See also* automatic generation of code generators（同时参见代码生成器的自动生成）
Alpha compiler（Alpha编译器），730
assembly language vs. relocatable binary form（汇编语言与可重定位二进制形式），138
described（描述），3
Graham-Glanville code-generator generator（Graham-Glanville代码生成器的产生器），140-144
issues（问题），137-138
code hoisting（代码提升），417-420
control-flow analysis and（控制流分析），175
example（例），418,420
implementation（实现），418-419
order of optimizations（优化的顺序），421
overview（概述），377, 417, 420
very busy expressions（非常忙表达式），417
code positioning in cache（高速缓存中代码定位）. *See* instruction-cache optimization（同时参见指令高速缓存优化）
code scheduling（代码调度）. *See* instruction scheduling（参见指令调度）
code sharing（代码共享）. *See* shared objects（参见共享对象）
code-generator generators（代码生成器的产生器）. *See* automatic generation of code generators（参见代码生成器的自动生成）
column-major order for arrays（数组的列为主顺序），107
combined or unified cache（合并的或统一的高速缓存），670
comment delimiter (||)（注释分界符（||））

bbs.theithome.com

in HIR, MIR, LIR（HIR、MIR和LIR中），76
in ICAN（ICAN中），21
comments（注释）
ICAN, 21
MIR, 76
COMMON statement (Fortran 77), aliases and（COMMON语句（Fortran 77），别名），298-299
common storage class (Fortran)（公用存储类（Fortran），44
common subexpressions（公共子表达式）
defined（定义），378
if simplifications（if化简），586
common-subexpression elimination（公共子表达式删除），378-396
combining with constant propagation（与常数传播结合），394
combining with copy propagation（与复写传播结合），394-395
forward substitution（向前替代），395
global（全局的），378, 385-395
local（局部的），378, 379-385, 395
order of optimizations（优化的顺序），421
overview（概述），377, 378-379, 420
reassociation with（重结合），385, 415-416
register use issues（寄存器使用问题），396
repeating（重复），394-395
value numbering vs.（值编号），343
commutativity, algebraic simplifications using（交换率，代数化简中的使用），334
compensation code, in trace scheduling（补偿代码，踪迹调度中），569
compilation process（编译过程）
high-level intermediate languages in（高级中间语言），69-70
phases（阶段），2-3
compilation, separate（编译，分开的）. *See* separate compilation（参见分开编译）
compile-time interprocedural register allocation（编译时过程间寄存器分配），662
compiler structure（编译结构），1-3
optimizing compilers（优化编译器），7-11
placement of optimizations（优化的安排），11-14
compiler suites（编译套件），9
compiler-specific types (ICAN), overview（编译专用的类型（ICAN），概述），24, 35
compilers, defined（编译器，定义），1-2
composition operation (o), in lattices（复合运算符(o)，格中），235
compound statements, in ICAN（复合语句，ICAN中），25
computation nodes（计算结点），571
concatenation operation, XBNF notation（连接运算，XBNF表示），19-20
condition codes, dependency computation and（条件代码，依赖关系计算），269
conditional assignment operator(←(...)), in HIR, MIR, and LIR（条件赋值运算符(←(...))，HIR、MIR和LIR中），74, 75
conditional moves（条件传送）
described（描述），580, 591
order of optimizations（优化的顺序），604
overview（概述），591-592
congruence of variables（变量的重合），348-351
co-*NP*-complete problems（co-*NP*完全问题），634, 637
conservative optimizations（保守的优化），319-320
constant folding（常数折叠），329-331
algorithm（算法），330
for floating-point values（用于浮点值），330-331
increasing effectiveness（增加效率），331
order of optimizations（优化的顺序），331, 372
overview（概述），329-330, 371
constant propagation（常数传播）. *See also* sparse conditional constant propagation（同时参见稀有条件常数传播）
combining with common-subexpression elimination（与公共子表达式删除结合），394
induction-variable optimizations and（归纳变量优化），426
interprocedural（过程间的），637-641,656, 658
overview（概述），362-363
constant-expression evaluation（常数表达式计算）. *See* constant folding（参见常数折叠）
constant-propagation analysis（常数传播分析），230, 261-262
interprocedural（过程间的），631-647, 656, 665-666
constants（常数）
global constant-propagation analysis（全局常数传播分析），230
HIR, MIR, and LIR integer constants（HIR、MIR和LIR整常数），76
for jump and return-jump functions（关于转移和返回转移函数），640
interprocedural constant-propagation analysis（过程间常数传播分析），631-641,656, 665-666
constants（常数）(ICAN)
array types（数组类型），30, 33

bbs.theithome.com

function types（函数类型），34
record types（记录类型），33
sequence types（序列类型），32, 33
set types（集合类型），31
syntax of generic simple types（普通简单类型的语法），28
tuple types（元组类型），33
constrained Diophantine equations（约束的丢番图方程），280
constraint matrix test（约束矩阵测试），284
constructed types（构造的类型）(ICAN), 28
array types（数组类型），30, 33
function types（函数类型），34
record types（记录类型），33
sequence types（序列类型），32, 33
set types（集合类型），31
tuple types（元组类型），33
control dependence（控制依赖）
defined（定义），267-268
dependence graphs and（依赖图），268-269
operator (δ^c)（运算符(δ^c)），267-268
control trees（控制树），198, 199
control-tree based data-flow analysis（基于控制树的数据流分析），235-236
control-dependence graphs(CDGs)（控制依赖图），284-286
control-dependent nodes of PDGs（PDG的控制结点），284, 286
control-flow analysis（控制流分析），169-216. *See also* flowgraphs and interprocedural control-flow analysis（同时参见流图和过程间控制流分析）
Alpha compilers（Alpha 编译器），726
basic-block identification（基本块标识），173-174
branch node（分支结点），175
breadth-first search（宽度为主搜索），181
depth-first search（深度为主搜索），178-179, 180
dominance tree（必经结点树），171-172
dominator-based approaches（基于必经结点的方法），173
dominators（必经结点），171-172, 181-191
flowgraph construction（流图构造），174-177
Intel compilers（Intel编译器），738
interprocedural（过程间的），609-618
interval analysis and control trees（区间分析和控制树），197-202
interval-based approaches（基于区间的方法），173
join node（汇合结点），175
natural loops（正常循环），191-193
postorder traversal（后序遍历），180, 181
POWER and PowerPC compilers（POWER和PowerPC编译器），721
predecessors of a node（结点的前驱），175
preorder traversal（前序遍历），179-181
reducibility（归约性），196-197
simple example（简单的例子），169-172
SPARC compilers（SPARC编译器），710
strongly connected components（强连通分量），193-196
structural analysis（结构分析），202-214
successors of a node（结点的后继），175
usefulness of（有效性），169-170
control-flow and low-level optimizations（控制流和低级优化），579-605
branch optimizations（分支优化），580, 589-590
branch prediction（分支预测），580, 597-599, 603
conditional moves（条件传送），580, 591-592
dead-code elimination（死代码删除），580, 592-597, 602-603
described（描述），322, 579
if simplifications（if化简），580, 585-586
loop inversion（循环倒置），580, 587-588
loop simplifications（循环化简），580, 586-587
machine idioms and instruction combining（机器方言和指令归并），580, 599-602
postpass or peephole optimizations（后遍或窥孔优化），579, 603-604
straightening（伸直化），579, 583-585
tail merging or cross jumping（尾融合或交叉转移），580, 590-591
unreachable-code elimination（不可到达代码删除），579, 580-582
unswitching（无开关化），580, 588-589
control-flow path, defined（控制流路径，定义），218
control-tree-based data-flow analysis（基于控制树的数据流分析），236-251
Cooper, Keith D., 355, 414, 443, 469, 620, 637, 642, 664
copy propagation（复写传播），356-362
combining with common-subexpression elimination（与公共子表达式删除结合），394-395
described（描述），356
global（全局的），358-362
local（局部的），356-358,361
order of optimizations（优化的顺序），372
overview（概述），356, 371
register coalescing（寄存器合并），487, 497-501
copy-propagation analysis（复写传播分析），230
Coutant, Deborah, 309
Cray Fortran extensions（Cray Fortran扩充），299, 305,

bbs.theithome.com

708
critical edges（关键边），407, 408,474
cross-block scheduling（跨基本块调度），546-547
described（描述），531
order of optimizations（优化的顺序），574-575
performance and（性能），573
cross edge, in depth-first presentation of flowgraph（横向边，流图的深度为主表示中）
defined（定义），178
label (C)（标号（C）），178
cross jumping（交叉转移）. *See* tail merging（参见尾融合）
cyclic control structures（有环控制结构）
backward structural data-flow analysis（向后结构数据流分析），245-246
forward structural data-flow analysis（向前结构数据流分析），240
structural control-flow analysis（结构控制流分析），202-204, 207-208

D

D-cache optimization（D-cache优化）. *See* data-cache optimization（参见数据高速缓存优化）
DAGs. *See* Directed Acyclic Graphs (DAGs)（参见无环有向图）
data dependence（数据依赖），268-269
data prefetching（数据预取），688,698-700
order of optimizations（优化的顺序），702-703
data structure representation（数据结构的表示），757-765
functions（函数），764-765
linked-list vs. bit-vector（链表与位向量），757-759
sequences（序列），763
sets（集合），759-763
trees and DAGs（树和DAG），763-764
data structures in structural control-flow analysis（结构控制流分析中的数据结构），205
data types（数据类型）. *See also* type definitions(ICAN)（同时参见ICAN中的类型定义）
arrays (ICAN)（数组（ICAN）），30
compiler-specific types (ICAN)（编译专用的类型（ICAN）），24, 35
constructed types (ICAN)（构造的类型（ICAN）），28
enumerated types (ICAN)（枚举类型（ICAN）），29-30
floating-point numbers（浮点数），81
functions (ICAN)（函数（ICAN）），34-35
generic simple types (ICAN)（普通简单类型（ICAN）），23, 28-29
integers（整数），81
nil value for（关于nil值），24, 35
records (ICAN)（记录类型（ICAN）），33-34
registers（寄存器），82
run-time representation（运行时的表示），106-109
sequences (ICAN)（序列（ICAN）），32-33
sets (ICAN)（集合（ICAN）），31-32
size operator (||)（大小运算符(||)），36
for symbol-table attributes（用于符号表的属性），46
syntax of ICAN definitions（ICAN定义的语法），25-26
tuples (ICAN)（元组（ICAN）），33
unions (ICAN)（联合（ICAN）），34
variables（变量），81
data-cache optimization（数据高速缓存优化），687-700. *See also* memory-hierarchy optimizations（同时参见存储层次优化）
data prefetching（数据预取），688,698-700
data-cache impact（数据高速缓存的影响），670-671
interprocedural arrangement of data（过程间的数据安排），689
locality and tiling（局部性与铺砌），695-698
loop transformations（循环转换），689-695
matrix multiplication and（矩阵乘），671
optimizer structure and（优化器的结构），10-11
order of optimizations（优化的顺序），701,702
overview（概述），687-689, 701
scalar vs. memory-oriented optimizations（标量与面向存储器的优化），700
data-flow analysis（数据流分析），217-266. *See also* interprocedural data-flow analysis（同时参见过程间数据流分析）
Alpha compiler（Alpha编译器），729
approaches to solving problems（解问题的方法），230-231
arrays, structures and pointers（数组、结构和指针），258-259
automating data-flow analyzer construction（数据流分析器构造自动化），259-261
available expressions analysis（可用表达式分析），229
backward（向后），229, 244-247
bidirectional（双向），229
conservatism in（保守性），217
constant-propagation analysis（常数传播分析），230, 261-262
control-flow path（控制流路径），218

bbs.theithome.com

超星公司制作 请尊重作者版权

control-tree based（基于控制树的）, 235-236
copy-propagation analysis（复写传播分析）, 230
du-chains（du链）, 251
factorial computation example（阶乘计算的例子）, 261-262
fixed points（不动点）., 226-227
flow functions（流函数）, 226-227
for global copy propagation（关于全局复写传播）, 358-360
forward（向前）, 229, 236-244
in induction-variable optimizations（归纳变量优化中）, 428
independent attributes（属性无关）, 229
Intel compilers（Intel编译器）, 738
interprocedural（过程间的）, 619-637
interprocedural data-flow information and（过程间数据流信息）, 658
interval analysis（区间分析）, 249-250
iterative analysis（迭代分析）, 231-235
lattices（格）, 223-226
lattices of flow functions（流函数的格）, 235-236
live variables analysis（活跃变量分析）, 229
overview（概述）, 217-218
partial-redundancy analysis（部分冗余分析）, 230
POWER and PowerPC compilers（POWER和PowerPC编译器）, 722
program verification, connection to（程序证明，关联）, 261-262
purpose（目的）, 217
reaching definitions analysis（到达定值分析）, 218-223,229
slotwise analysis（槽式分析）, 250-251
SPARC compilers（SPARC编译器）, 710
SSA form（SSA形式）, 252-258
structural analysis（结构分析）, 236-249
taxonomy of problems and solution methods（问题和求解方法的分类）, 228-231
for type determination and dependence analysis（用于类型判定和依赖分析）, 262-263
ud-chains（ud链）, 251
upwards exposed uses analysis（向上暴露使用分析）, 229-230
webs（网）, 251-252
dead instructions（死去的指令）, 592
dead variables（死去的变量）, 445, 592-593
dead-code elimination（死代码删除）, 592-597
algorithm（算法）, 592-593, 596
described（描述）, 580, 592
determining dead code（确定死代码）, 592-593
ICAN code（ICAN代码）, 593-594
importance of（重要性）, 602-603
order of optimizations（优化的顺序）, 327, 603-604
repetition of（重复）, 13-14, 579, 603
unreachable-code elimination vs.（不可到达代码删除）, 580
debugging, intermediate-code ontput（调试，中间代码输出）, 69
DEC GEM compilers（DEC GEM编译器）. *See* Alpha compilers（参见Alpha 编译器）
DEC VAX, ASCII support（DEC VAX, ASCII支持）, 106
declarations (ICAN)（声明（ICAN））. *See also* type definitions(ICAN)（同时参见类型定义(ICAN)）
procedure declarations（过程声明）, 23, 24, 27
syntax（语法）, 26-27
variable declarations（变量声明）, 23, 26-27
dedicated registers（专用的寄存器）, 120
defined to be operator（定义运算符）
⇒, in machine grammars（⇒，机器文法中）, 140
→, in XBNF notation（→，XBNF表示中）, 19
degree, in register allocation by graph coloring（度，图着色寄存器分配中）, 494
degree < *R* rule（度< *R*规则）, 488, 503-506
delay slots, filling（延迟槽，填充）, 532, 534-535
delayed expressions in partial-redundancy analysis（部分冗余分析中的延迟的表达式）, 411-412
delete-node transformation（删除结点转换）, 571-572
delimiters（分界符）
between statements (ICAN)（语句之间的（ICAN））, 21
of compound statements (ICAN)（复合语句的(ICAN)）, 25
Delta test（Delta测试）, 284
dependence, defined（依赖关系，定义）, 267
dependence analysis（依赖关系分析）, 267-291
Alpha compilers（Alpha 编译器）, 729
basic-block dependence DAGs（基本块依赖DAG）, 269-274
data-flow analysis for（数据流分析的）, 262-263
dependence relations（依赖关系）, 267-269
dependence testing（依赖关系测试）, 279-284
dependences between dynamically allocated objects（动态分配对象之间的依赖关系）, 286-288
dependences in loops（循环中的依赖关系）, 274-279
in high-level intermediate languages（高级中间语言

bbs.theithome.com

中），71
Intel compilers（Intel编译器），738
POWER and PowerPC compilers（POWER和PowerPC编译器），721
program-dependence graphs (PDGs)（程序依赖图(PDG)），284-286
SPARC compilers（SPARC编译器），711
dependence DAGs（依赖DAG），269-274
algorithm（算法），273-274
constructing（构造），271-274
edges in（边），269
for LIR code（LIR代码的），270-271
list scheduling（列表调度），535,537-543
nodes in（结点），269
resource vectors（资源向量），271
structural hazards（结构停顿），271
for window scheduling（用于窗口调度），551-552, 554, 555
dependence graphs（依赖图）
described（描述），268-269
program-dependence graphs (PDGs)（程序依赖图(PDG)），284-286
dependence relations（依赖关系），267-269
antidependence (δ^a)（反依赖(δ^a)），268
control dependence (δ^c)（控制依赖(δ^c)），267-268
data dependence（数据依赖），268
flow or true dependence (δ^f)（流或真依赖(δ^f)），268
input dependence (δ^i)（输入依赖（δ^i)），268
loop-carried（循环携带的），278
loop-independent（循环无关的），278
notation（表示法），267-268,276
output dependence (δ^o)（输出依赖(δ^o)），268
write-live dependence (δ^{wl})（写-活跃依赖(δ^{wl})），571
dependence testing（依赖测试），279-284
constrained Diophantine equations（约束的丢番图方程），280
GCD (greatest common divisor)（GCD（最大公因子)）281-283
integer programming（整数规划），280
other tests（其他测试），283-284
separable array references（可分的数组引用），282
weakly separable array references（弱可分的数组引用），282
dependence vector（依赖向量）
defined（定义），278
delimiters (<...>)（分界符<...>），277, 278
distance vector, set of（距离向量，集合），278
dependent induction variables（依赖归纳变量），426-427, 429
depth-first number（深度为主编号），178
depth-first presentation（深度为主表示），178
depth-first search（深度为主搜索），178-179, 180
computing dominators（计算必经结点），187
interprocedural data-flow analysis（过程间数据流分析），632
depth-first spanning tree（深度为主生成树），178
computing（计算），180
in structural control-flow analysis（在结构控制流分析中），204
tree edges（树边），178
derives operator ($\stackrel{*}{\Rightarrow}$), in machine grammars（推导运算符，机器文法中），147
Deutsch, Alain, 133,287, 290
Dhamdhere, Dhananjay M., 250, 407
Directed Acyclic Graphs (DAGs)（有向无环图(DAG)），100-101
basic-block dependence DAGs（基本块依赖DAG），269-274
representation（表示），764, 765
direction vector（方向向量）
defined（定义），277-278
delimiters (<...>)（分界符(<...>)），277-278
distance vector（距离向量）
defined（定义），277
delimiters (<...>)（分界符(<...>)），278
lexicographically positive（词典序为正的），690-691
distance vector set（距离向量集合），278
distributive lattices（分配格），224, 236
division operator (/)（除法运算符(/)）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74
in ICAN（ICAN中），27, 29
does not divide operator ($\nmid$)（不能整除运算符），281
dominance frontiers（必经边界），253-256
computing for flowgraphs（流图的计算），254-256
iterated（迭代的），254-256
dominance tree（必经结点树）
global value numbering for（全局值编号），355
identifying loops（标识循环），171-172
dominators（必经结点），181-191
algorithms（算法），182, 186-187
computing using Dom_Comp()（使用Dom_Comp()计算），182-184
computing using Domin_Fast（使用Domin_Fast计算），185-191
disadvantages of（缺点），173
immediate dominators（直接必经结点），183-184,

版权所有 翻印必究
bbs.theithome.com

189-191
label evaluation computation（标号评估计算），188
linking computation（链接计算），188
overview（概述），181-182
path-compression computation（路径压缩计算），187
postdominators（后必经结点），182
semidominators（半必经结点），185
simple example（简单的例子），171-172
strict dominators（必经结点），182
double-precision floating-point number, ends with D, in HIR, MIR, and LIR（双精度浮点数，以D结束，HIR、MIR和LIR中），74
double-precision matrix multiplication（双精度矩阵乘），671
du-chains（du链），251,486. *See also* webs（同时参见网）
determining dead code（确定死代码），593
in SSA form（SSA形式中），252
dynamic branch-prediction methods（动态分支预测方法），597-598
dynamic extent（动态作用域），45
dynamic languages, run-time support（动态语言，运行时支持），131-133
dynamic link（动态链）. *See also* shared objects（同时参见共享对象）
described（描述），110-111, 114
global offset table (GOT)（全局偏移表(GOT)），129, 130
procedure linkage table (PLT)（过程连接表(PLT)），130, 131
semantics of（语义），127-128
transferring control between shared objects（在共享对象之间转换控制），129-130
transferring control within shared objects（在共享对象内转换控制），128-129
dynamic linker, run-time（动态连接，运行时），127-128
dynamic programming（动态规划），160-165
algorithms（算法），163-165
described（描述），160
optimality principle（最优性原理），160
pattern-matching process（模式匹配过程），160-164
tree-matching automaton（树匹配自动机），163, 164
tree-rewriting rules（树重写规则），160-162
uniform-register-machine assumption（统一的寄存器机器的假设），163-165
dynamic scoping（动态作用域），45
dynamically allocated objects, dependences between（动态分配的对象，之间的依赖），286-288
dynamically allocated storage, alias gatherer and（动态分配的存储单元，和别名搜集），302

E

earliestness of expressions in partial-redundancy analysis（部分冗余删除中的最早性），411
early optimizations（前期优化），329-375
algebraic simplifications and reassociation（代数化简和重结合），333-343
constant folding（常数折叠），329-331
constant-expression evaluation（常数表达式计算），329-331
copy propagation（复写传播），356-362
described（描述），321
scalar replacement of aggregates（聚合量标量替代），331-333
sparse conditional constant propagation（稀有条件常数传播），362-371
value numbering（值编号），343-355
EBCDIC, IBM System/370 support（EBCDIC, IBM System/370支持），106
Ebcioğlu Kemal, 548
edge in directed graph (...→...)（有向图中的边(...→...)），175
edge splitting, of critical edges（边分割，关键边的），407, 474, 509
effective height, of a lattice（有效高度，格的），226
efficiency（效率）. *See also* performance（同时参见性能）
one-pass compilers and（一遍编译器），3, 6
run-time support and（运行时支持），4-6
Eggers, Susan J., 526, 545
EIL code（EIL代码），727, 728
elimination methods（消去法）
control-tree based data-flow analysis（基于控制树的数据流分析），235-236
interval control-flow analysis（区间控制流分析），173, 197-202
empty sequences, in ICAN（空序列，ICAN中）
defined（定义），24
value ([])（值([])），24, 27, 32
empty sets, in ICAN（空集合，ICAN中）
defined（定义），24
value (ø)（值（ø)），24, 27, 31
empty string (∈), in grammars（空字符串(∈)，文法中），19, 140
enum, enumerated type constructor, in ICAN（enum，枚举类型构造符，ICAN中），23

bbs.theithome.com

enumerated type delimiters ({...}), in ICAN（枚举类型分界符({...})，ICAN中），23, 26, 29
enumerated types, run-time representation（枚举类型，运行时的表示），107
enumerated types (ICAN)（枚举类型（ICAN））
  binary operators（二元运算符），30
  overview（概述），29-30
  symbol-table entries（符号表项），46
epilogue of procedures（过程的出口处理），120
  shrink wrapping and（收缩包装），472, 473
equal to operator (=)（相等运算符(=)）
  in HIR, MIR, and LIR（HIR、MIR和LIR中），74
  in ICAN（ICAN中），27, 28, 29, 30
EQUIVALENCE statement (Fortran 77), aliases and（EQUIVALENCE语句(Fortran 77)，别名），298-299
escape sequences for character values（字符值的转义序列），29
essential value or instruction（必要的值或指令），592
examples in this book, target machines（本书中的例子，目标机器），16
exception handling, in interprocedural side-effect analysis（异常处理，过程间副作用分析中），637
existential quantifier (∃), in ICAN（存在量词(∃)，ICAN中），27, 28
exit blocks of a loop（循环的出口基本块），403
exponent delimiter (E)（指数分界符(E)）
  in HIR, MIR, and LIR（HIR、MIR和LIR中），74
  in ICAN（ICAN中），28, 29
exponentiation operator (↑), in ICAN（幂运算符，ICAN中），27, 29
expressions（表达式）
  algebraic simplification and reassociation of addressing expressions（地址表达式的代数化简和重结合），334-341
  algebraic simplification of floating-point expressions（浮点表达式的代数化简），342-343
  available expressions analysis（可用表达式分析），229
  common-subexpression elimination（公共子表达式删除），378-396
  delayed（延迟的），411-412
  earliestness（最早性），411
  globally anticipatable values（全局可预见值），410-411
  HIR, 79
  isolated（孤立的），413
  latestness（最迟性），412-4 13
  LIR, 80
  locally anticipatable values（局部可预见值），410
  MIR, 84-85
  very busy expressions（非常忙表达式），417
expressions (ICAN)（表达式（ICAN））
  Boolean-valued（具有布尔值的），28
  nil value in（nil值），35
  overview（概述），23, 28
  syntax（语法），27
Extended Backus-Naur Form (XBNF)（扩展的巴科斯－诺尔范式(XBNF)），19-20
  syntax of HIR instructions（HIR指令的语法），78-79
  syntax of LIR instructions（LIR指令的语法），79-81
  syntax of MIR instructions（MIR指令的语法），73-75
  syntax of MIR programs and program units（MIR程序和程序单位的语法），74
extended basic blocks（扩展基本块），175-177
  local copy propagation for（局部复写传播），361
  register renaming on（寄存器重命名），565
extended formal parameters（扩展的形式参数），643
extended GCD test（扩展的GCD测试），283
extent of variables（变量的作用域），44
  dynamic extent（动态作用域），45
extern storage class（外部存储类），44

F

factorial computation, data-flow analysis example（阶乘计算，数据流分析的例子），261-262
Farnum, Charles, 342
field indirection operator (*.), in HIR, MIR, and LIR（域间接运算符(*.)，HIR、MIR和LIR中），74
field selection operator (.)（域选择运算符(.)）
  in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 79
  in ICAN（ICAN中），27, 34
file storage class（文件存储类），44
finite differences, method of（有限差分，方法），435
Fischer, Charles N., 159, 160
fixed points（不动点），226-227
  maximum (MFP)（最大不动点），227
floating-point expressions, algebraic simplification（浮点表达式，代数化简），342-343
floating-point registers, ICAN representation（浮点寄存器，ICAN表示），82
floating-point values（浮点值）
  constant folding for（常数折叠相关的），330-331
  data-cache optimization and（数据高速缓存优化），688-689
  in ICAN（ICAN中），81

bbs.theithome.com

in HIR, LIR, and MIR（HIR、LIR和MIR中）, 76
parameter-passing in registers（用寄存器传递参数）, 121
run-time representation（运行时的表示）, 106-107
flow dependence（流依赖）
defined（定义）, 268
in dependence graphs（依赖图）, 268
operator (δ^f)（运算符(δ^f)）, 268
flow functions（流函数）
alias propagator（别名传播器）, 308
backward structural data-flow analysis（向后结构数据流分析）, 244
fixed point（不动点）, 226
forward structural data-flow analysis（向前结构数据流分析）, 237-239
lattices of monotone（单调格）, 235-236
overview（概述）, 226-227
flow sensitivity（流敏感性）
alias information（别名信息）, 294-296, 315
flow-insensitive interprocedural alias analysis（流不敏感过程间别名分析）, 642-654
flow-insensitive interprocedural side-effect analysis（流不敏感过程间副作用分析）, 619-633
flow-sensitive interprocedural alias analysis（流敏感过程间别名分析）, 642
flow-sensitive side-effect analysis（流敏感副作用分析）, 634-636
global optimization and（全局优化）, 323
interprocedural optimization and（过程间优化）, 608-609, 664
optimizations（优化）, 323
flow-insensitive interprocedural alias analysis（流不敏感过程间别名分析）, 642-654
binding graph construction（结合图构造）, 643,645
combining alias information with alias-free version（别名信息与免除别名的版本结合）, 651, 654
computing formal-parameter aliases（计算形式参数的别名）, 650-651,653
example program（例程序）, 643, 644
extended formal parameter set（扩展的形式参数集合）, 643
inverting nonlocal alias function（颠倒非局部别名函数）, 646-647, 648
marking algorithm（标志算法）, 650, 652-653
nonlocal aliases（非局部别名）, 643-644, 646-648
pair binding graph construction（成对结合图构造）, 649-650
steps in computing（计算的步骤）, 643
flow-insensitive side-effect analysis（流不敏感副作用分析）, 619-633
binding graph construction（结合图构造）, 623-627
call graph construction（调用图构造）, 628-631
depth-first search（深度为主搜索）, 632
strongly connected components（强连通分量）, 631-632
flow insensitivity（流不敏感性）. *See* flow sensitivity（参见流敏感性）
flow-sensitive interprocedural alias analysis（流敏感过程间别名分析）, 642
flow-sensitive side-effect analysis（流敏感副作用分析）, 634-636
flowgraphs（流图）. *See also* control-flow analysis（也参见控制流分析）
abstract（抽象）, 198
constant propagation and（常数传播）, 363
construction（构造）, 174-177
dominance frontier computation（必经边界计算）, 254, 255
dominator tree for（关于必经结点树的）, 257
extended basic blocks（扩展基本块）, 175-177
intraprocedural code positioning（过程内代码安置）, 681
iterated dominance frontier computation（迭代的必经边界计算）, 254, 256
predecessor basic blocks（前驱基本块）, 175
reducibility（可归约性）, 196-197
for register allocation by graph coloring（关于图着色寄存器分配的）, 512, 513,518, 521, 522
reverse extended basic blocks（逆转的扩展基本块）, 175
successor basic blocks（后继基本块）, 175
translating to SSA form（转换到SSA形式）, 256-257, 349-350
fonts, XBNF notation（字体，XBNF表示）, 19
for statements（for语句）
HIR, 78-79
ICAN, 39
iterators in ICAN（迭代符，ICAN中）, 39-40
Fortran
aliased variables and（有别名的变量）, 641
aliases in Fortran 77（Fortran 77中的别名）, 298-299
aliases in Fortran 90（Fortran 90中的别名）, 301-302, 305
arrays, column-major order（数组，列为主顺序）, 107
call by reference in（传地址）, 118
call by value-result in（传值得结果）, 117

bbs.theithome.com

common storage class（公用存储类），44
Cray extensions for Fortran 77（Fortran 77的Cray扩充），299, 305, 708
enumerated value representation（枚举值的表示），107
Fortran H compiler（Fortran H编译器），484, 528
labels as arguments（标号作为实参），118-119
lexical and syntactic analysis in（词法和语法分析），3
preprocessor for（关于预处理），10
reducibility in Fortran 77（Fortran 77中的可归约性），196-197
forward data-flow analysis（向前数据流分析）
defined（定义），218
iterative（迭代的），231-234
iterative forward bit-vector（迭代的向前位向量），218-223
structural（结构的），236-244
forward edges, in depth-first presentation of flowgraph（前向边，流图的深度为主表示中）
defined（定义），178
label (F)（标号（F）），178
forward substitution（向前替代），396
Fourier-Motzkin test（Fourier-Motzkin测试），284
frame pointer（帧指针）(fp), 112-114
alloca（ ）and, 113
described（描述），110, 112
stack pointer vs.（栈指针），112-113
tail-call elimination and（尾调用删除），462
frames（帧）. *See* stack frames（见栈帧）
Fraser, Christopher, 273,768, 769
Free Software Foundation（自由软件基金会），768
Freiburghouse, Richard A., 483, 528
function composition (∘)（函数复合（∘）），235
function types, in ICAN（函数类型，ICAN中）
constants（常数），34
constructor (→)（构造符(→)），23, 26, 28, 34
overview（概述），34-35
functions, representation（函数，表示），764-765
fundamental induction variables（基本归纳变量），426, 427
future trends in compiler design（编译器设计中的未来趋势），744

G

Ganapathi, Mahadevan, 159, 160, 608,664
Gao, G. R., 817
GCD (greatest common divisor) dependence test（GCD（最大公因子）依赖测试），281-283
GEM compilers（GEM编译器）. *See* Alpha compilers（参见Alpha 编译器）
generating code（生成代码）. *See* automatic generation of code generators; code generation（参见代码生成器的自动生成；代码生成）
generation scavenging（阶段清除），133
generic simple constants (ICAN), syntax（普通简单常数（ICAN），语法），28
generic simple types (ICAN)（普通简单类型（ICAN））
list of（表），23, 28
overview（概述），28-29
Geschke, Charles M., 820
global common-subexpression elimination（全局公共子表达式删除），385-395
algorithms（算法），388-390, 391-394
available expressions（可用表达式），385-387, 390
combining with local form（与局部形式结合），394
control-flow analysis and（控制流分析），175
dealing with individual expression occurrences（处理单独的表达式的出现），390-394
local vs.（局部的），378
repeating（重复），394-395
using *AEin*( ) data-flow function（使用*AEin*（ ）数据流函数），388-390
global copy propagation（全局复写传播），358-362
data-flow analysis for（数据流分析），358-360
data-flow equations（数据流方程），359-360
local copy propagation and（局部复写传播），361
performing（实现），360-361
some copy assignments not detected（某些复写赋值没有被删除），362
global data structures in structural control-flow analysis（结构控制流分析中的全局数据结构），205
global offset table (GOT)（全局偏移表（GOT）），129, 130
global offset table pointer (gp)（全局偏移表指针（gp））
described（描述），111
used（使用），129, 663
global storage classes（全局存储类），44
global symbol-table structure（全局符号表结构），49-54
closed scopes（闭作用域），52-54
stack-plus-hashing model（栈加散列模式），52-54
symbol-table stacks（符号表栈），49, 51
tree of local symbol tables（局部符号表树），49, 50
global value numbering（全局值编号），348-355
congruence of variables（变量的叠合），348-351
extending to dominator trees（扩充到必经结点树），355

bbs.theithome.com

generalizations to approach（方法的推广），355
local value numbering and（局部值编号），344
partitioning algorithm（划分算法），351-355
translating flowgraphs into SSA form（转换流图到SSA形式），349-350
value graphs（值图），349-351
globally anticipatable values（全局可预见值），410-411
GNU compilers（GNU编译器），768
Goldberg, David, 331
Goldin, Dina Q., 803
Goldstine, Herman H., 459
Golumbic, Martin C., 548, 803
Goodman, James R., 740
GOT. *See* global offset table (GOT)（参见全局偏移表）
goto statements（goto语句）
ICAN form（ICAN形式），38
labels as arguments（标号作为实参），118
MIR, 75-76
Graham, Susan L., vii-ix, 52, 139-158
Graham-Glanville code-generator（Graham-Glanville代码生成器）
generator（生成器），139-158
Action/Next table issues（Action/Next表的问题），158
algorithm（算法），142-143
chain loop elimination（链循环删除），152-154
code generator（代码生成器），140-144
code-generation algorithm（代码生成算法），142-143
code-generator generator（代码生成器的生成器），144-151
components（部分），139
example for LIR instructions（LIR指令的例子），139-140, 141
machine-description rules（机器描述规则），140-142
overview（概述），139-140
Polish-prefix input to code generator（代码生成器的前缀波兰输入），139-158,764
syntactic block elimination（句法阻滞删除），154-158
granularity of aliasing information（别名信息的粒度），297, 315
graph coloring（图着色）. *See* register allocation by graph coloring（参见图着色寄存器分配）
greater than operator (>)（大于运算符（>））
in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
in ICAN（ICAN中），27, 29
greater than or equal to operator（大于或等于运算符）
>=, in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
> in ICAN（ICAN中），27, 29
greatest common divisor (GCD) dependence test（最大公因子（GCD）依赖关系测试）），281-283
greedy algorithms（贪婪算法），141,544
greedy schedule, unconstrained（贪婪调度，无约束的），555-556
Gross, Thomas, 542, 543
group I optimizations（第一组优化），323-324
group II optimizations（第二组优化），324
group III optimizations（第三组优化），324-325
group IV optimizations（第四组优化），325
grouping operator ({...}), in XBNF notation（成组运算符({...})，XBNF表示中），19
Groves, Randy, 807
Gupta, Anoop, 699
Gupta, Rajiv, 525, 689

H

Hall, Mary W., 469,664, 806
Hanson, David, 768, 769
hashing（散列）
function representation（函数表示），764, 765
local common-subexpression elimination（局部公共子表达式删除），384-385
local symbol-table management（局部符号表管理），48-49
set representation（集合表示），761-762
stack-plus-hashing model（栈加散列模式），52-54
height of a lattice（格的高度），226
Hendren, Laurie J., 262, 287, 290
Hennessy, John, 259, 260, 524, 542, 543, 662
Henry, Robert, 158,526, 769
heuristics for register spilling（寄存器溢出启发式）. *See* register allocation by graph coloring（参见图着色寄存器分配）
Hewlett-Packard PA-RISC compiler（Hewlett-Packard PA-RISC编译器）. *See* PA-RISC compiler（参见PA-RISC编译器），
hexadecimal notation（十六进制表示），17
hexadecimal number, begins with 0x, in HIR, MIR, and LIR（十六进制数，以0x打头，HIR、MIR和LIR中），74, 76
hierarchical reduction（层次归约），568-569
order of optimizations（优化的顺序），574-575
high-level data-flow analysis（高级数据流分析），202
high-level intermediate languages（高级中间语言），69-71. *See also* HIR
abstract syntax tree（同时参见HIR抽象语法树），70-71

bbs.theithome.com

in compilation process（编译过程中），69-70
dependence analysis in（依赖分析），71
order of optimizations（优化的顺序），325-326
in preprocessors（预处理器中），69-70
High-level Intermediate Representation（高级中间表示）. *See* HIR（参见HIR）
Hilfinger, Paul, 525
HIR, 78-79. *See also* intermediate code（同时参见中间代码）
canonical form for loops（循环的规范形式），274
definition of（定义），78-79
described（描述），4
differences from MIR（与MIR的不同），78-79
ICAN representation（ICAN表示），81-82, 85-86
instructions（指令），78-79
loop code（循环代码），5
operators in IROper（IROper中的操作），82
Hobbs, Steven O., 820
Hood, Robert T., 637
Hopcroft, John, 351
Hopkins, Martin, 494
HP PA-RISC compiler（HP PA-RISC编译器）. *See* PA-RISC compiler（参见PA-RISC编译器）
Hummel, Joseph, 287

I

I-cache optimization（I-cache优化）. *See* instruction-cache optimization（参见指令高速缓存优化）
IBM 801 RISC system（IBM 801 RISC系统），485, 718
IBM PL/1 compiler（IBM PL/1编译器），528
IBM System/370 compiler（IBM System/370编译器），106, 484, 528, 531,575
IBM XL compilers（IBM XL编译器）. *See* POWER and PowerPC compilers
IBURG, 769-770
ICAN. *See* Informal Compiler Algorithm Notation (ICAN)（参见非形式编译算法表示法（ICAN））
ICP lattices（ICP格）. *See* integer constant-propagation (ICP)lattices（参见整数传播（ICP）格）
identifiers, MIR（标识符，MIR），76
identity function, (*id*())（恒等函数，(*id*())），233
if simplifications（if化简），585-586
common subexpressions（公共子表达式），586
constant-valued conditions（值为常数的条件），585-586
described（描述），580, 585
for empty arms（空的分支），585
if statements（if语句）
code representations for structural data-flow analysis（结构数据流分析的代码表示），247-248
flow functions for structural data-flow analysis（结构数据流分析的流函数），237-238, 244-245
HIR, 78
ICAN, 38-39
MIR, 75-76
IL-1 code (Intel compilers)（IL-1代码，Intel编译器），736, 737
IL-2 code (Intel compilers)（IL-2代码，Intel编译器），736-738
immediate dominators（直接必经结点），183-184, 189-191
implicit resources, dependency computation and（隐含资源，依赖关系计算），269
improper intervals or regions（非正常区间或区域），196, 203,240-241
in-line expansion（内嵌扩展），470-472
advantages（好处），470-471
mechanisms required（需要的机制），471-472
order of optimizations（优化的顺序），477-478
summary（概要），476
independent-attributive data-flow analysis（属性无关数据流分析），229
index vectors（索引向量），lexicographic ordering of（词典序），275
induction variables（归纳变量）
basic or fundamental（基本的或基础的），426, 427
basis（基本的），428
class of（类），428
dependent（依赖），426-427, 429
identifying（标识），426-435
induction-variable expansion（归纳变量扩张），562-563
overview（概述），425-426
removal of（删除），447-453
replacing (linear-function test replacement)（替换，线性函数测试替换），447, 448-453
strength reduction（强度削弱），435-443, 444
induction-variable optimizations（归纳变量优化），425-453
addressing modes and（寻址方式），426
constant propagation and（常数传播），426
data-flow analysis in（数据流分析），428
identifying induction variables（识别归纳变量），426-435
linear-function test replacement（线性函数测试替换），447, 448-453

bbs.theithome.com

live variables analysis（活跃变量分析），443-446
order of optimizations（优化的顺序），458
removal of induction variables（基本归纳删除），447-453
strength reduction（强度削弱），435-443, 444
summary（概要），458
infinite loop（无限循环）. *See* nonterminating computation（参见不终止的计算）
infinite value (∞), in ICAN（无穷值，ICAN中），24, 28, 29
Informal Compiler Algorithm Notation (ICAN)（非形式编译算法表示(ICAN)），19-42
comments in（注释），21
constants of constructed types（构造类型的常数），24
constructors（构造符），23
data types（数据类型），27-36
expressions（表达式），23, 27, 28
Extended Backus-Naur Form (XBNF)（扩展的巴科斯－诺尔范式），19-20
generic simple constants（一般简单常数），28
generic simple types（一般简单类型），23, 28-29
HIR representation in（HIR表示），81-82, 85-86
intermediate-code operators（中间代码运算符），82
keywords, reserved in（关键字，保留的），40
lexical conventions（词法约定），21-22
LIR representation in（LIR表示），81-82, 86-92
MIR representation in（MIR表示），81-85
naming of data structures and routines（数据结构和例程的命名），92-95
overview（概述），23-25
procedure declarations（过程说明），23, 24, 26-27
program syntax（语法），25
statements（语句），36-40
syntax conventions（语法约定），21
type definitions（类型定义），23, 24, 25-26, 27-36
variable declarations（变量说明），23, 26-27
inlining（内联）. *See* in-line expansion（参见内嵌扩展）
inlining（内联），automatic. *See* procedure integration（参见过程集成）
input dependence（输入依赖）
defined（定义），268
operator (δ)（运算符（δ)），268
inside-out order（由内向外顺序），609
instruction buffer（指令缓冲区），672
instruction-cache optimization（指令高速缓存优化），672-682
combining intra- and interprocedural methods（结合过程内和过程间的方法），682
instruction prefetching（指令预取），672-673
instruction-cache impact（指令高速缓存的影响），672
intraprocedural code positioning（过程内代码安置），677-681
order of optimizations（优化的顺序），702-703
overview（概述），701
procedure and block placement（过程和基本块放置），676-677
procedure sorting（过程排序），673-676
procedure splitting（过程分割），681-682
instruction combining（指令归并）. *See* machine idioms and instruction combining（参见机器方言和指令归并）
instruction decomposing（指令分解），602
instruction prefetching（指令预取）
branch prediction and（分支预测），673
hardware prefetching（硬件预取），672
order of optimizations（优化的顺序），702-703
software prefetching（软件预取），672-673
instruction scheduling（指令调度），531-577
algorithm（算法），540-541
automating instruction- scheduler generation（指令调度生成器自动化），543
balanced scheduling（平衡式调度），545
basic-block schednling（基本块调度），531
branch scheduling（分支调度），533-535, 536
combining with register allocation（与寄存器分配结合），526
cross-block scheduling（跨基本块调度），531, 546-547
DEC GEM compilers（DEC GEM编译器），729, 730
described（描述），322
Intel 386 family compilers（Intel 386系列编译器），739, 740
list scheduling（表调度），535, 537-543
loop unrolling（循环展开），532
order of optimizations（优化的顺序），532, 574-575
overview（概述），531-532
percolation scheduling（渗透调度），532, 571-573
POWER and PowerPC compilers（POWER和PowerPC编译器），723
register allocation and（寄存器分配），545
register renaming（寄存器重命名），532
repetition of（重复），14
software pipelining（软流水），531,532, 548-569
SPARC compilers（SPARC编译器），713
speculative loading and boosting（前瞻取和上推），547-548

bbs.theithome.com

speculative scheduling（前瞻调度），548
for superscalar implementations（实现），543-545
trace scheduling（踪迹调度），532, 569-570
variable expansion（变量扩张），532
instruction-scheduler generation, antomating（指令调度生成器，自动化），543
integer constant-propagation (ICP) lattices（整常数传播（ICP）格），224-225,228
integer ICAN type（整型ICAN类型）
binary operators（二元运算符），29
overview（概述），29
integer programming（整数规划），280
integer registers（整数寄存器）
ICAN representation（ICAN表示），82
parameter-passing in（参数传递），121-123
integers（整数）
ICAN representation（ICAN表示），81
run-time representation（运行时的表示），106
Intel reference compilers for 386 family（Intel 用于386系列的参考编译器），735-744
assembly language（汇编语言），752-753
code generator (Proton)（代码生成(Proton)），735, 738-739
code reselection（代码重新选择），740
data-dependence testing（数据依赖测试），738
data-flow analysis（数据流分析），738
fxch instruction（fxch指令），740
global optimizer（全局优化），738
IL-1 code（IL-1代码），736, 737
IL-2 code（IL-2代码），736-738
instruction combining（指令归并），739-740
instruction scheduling（指令调度），740
instruction selection（指令选择），739-740
Intel 386 family architecture（Intel 386系列体系结构），734-735
interprocedural optimizer（过程间优化），736-737
languages supported（所支持的语言），735
low-level optimizations（低级优化），740-741
memory optimizer（存储优化），738
Pentium assembly code examples（Pentium汇编代码的例子），741-744
position-independent code（位置无关代码），741
Proton intermediate language (PIL)（Proton中间语言(PIL)），738-739
register allocation（寄存器分配），740
sparse conditional constant propagation（稀有条件常数传播），738
structure（结构），735-736
interference graphs（冲突图），494-497
adjacency-list representation（邻接表表示），487, 496-497, 498,499
adjacency-matrix representation（邻接矩阵表示），487, 495-496, 497
degree < *R* rule（度<*R*规则），503-506
described（描述），481,485
examples（例），486, 503,513-516, 519-522
overview（概述），494-495
pruning（修剪），488,503-506
reducing register pressure（减少寄存器压力），506
register pressure（寄存器压力），506
spilling symbolic registers（溢出符号寄存器），488, 506-511
interferences, defined（冲突，定义），481
interlocks（互锁），532-533
intermediate code（中间代码）. *See also* HIR; LIR; MIR; SSA form（参见HIR; LIR; MIR; SSA形式）
algorithms operating on（其上运算的算法），4
code-generation approach and（代码生成方法），138
debugging ontput（调试输出），69
Directed Acyclic Graphs (DAG)（有效无环图），100-101
external representation issues（外部表示的问题），69
high-level intermediate languages（高级中间语言），69-71
HIR, 78-79, 85-86
ICAN naming of data structures and rontines（数据结构和例程的ICAN命名），92-95
ICAN representation of MIR, HIR, and LIR（MIR、HIR和LIR 的ICAN表示），81-92
LIR, 79-81, 86-92
low-level intermediate languages（低级中间语言），71-72
medium-level intermediate languages（中级中间语言），71
MIR, 73-78, 82-85
multi-level intermediate languages（多级中间语言），72-73
new intermediate code design vs. existing one（新的中间代码涉及与现存的中间代码设计），67-69
optimization and（优化），67-68
Polish-prefix notation（前缀波兰表示），101
program-dependence graphs (PDGs)（程序依赖图），284-286
quadruples（四元式），96
translating between forms（各种形式之间的转换），

bbs.theithome.com

68-69
trees（树），97-100
triples（三元式），96-97
variety in design（设计中的变化），4
Internet software resources（互联网软件资源），767
interprocedural alias analysis（过程间别名分析），641-656
example（例），642
flow-insensitive（流不敏感），642-654
flow-sensitive（流敏感），642
for languages with call by value and pointers（传值和传指针语言），654-656
overview（概述），641-642
interprocedural analysis and optimization（过程间分析和优化），607-668
aggregation of global references（全局引用的聚合），663
alias analysis（别名分析），641-656
constant propagation（常数传播），637-641
control-flow analysis（控制流分析），609-618
data-flow analysis（数据流分析），619-637
flow insensitivity and（流不敏感性），608-609
flow sensitivity and（流敏感性），608-609
in-line expansion（内嵌扩展），470-472
may and must information and（可能和一定信息），608-609
optimizations（优化），656-659
order of optimizations（优化的顺序），665-666
overview（概述），607-609
parameter-passing and（参数传递），608
performance and（性能），608
procedure cloning（过程克隆），607-608, 657
procedure integration（过程集成），465-470
"programming in the large" issues（大规模程序设计中的问题），663-664
register allocation（寄存器分配），659-662
separate compilation and（分开编译），608
tail-call elimination（尾调用删除），461-465
interprocedural constant propagation（过程间常数传播），637-641, 665-666
algorithm（算法），637-639
choices for jump and return-jump functions（转移和返回转移函数的选择），640-641
importance of（重要性），664
jump functions（转移函数），637
order of optimizations（优化的顺序），665-666
procedure cloning and（过程克隆），641
return-jump functions（返回转移函数），637
site-independent form（位置无关形式），637
site-specific form（位置特殊形式），637
support of a function（函数的支持），637
interprocedural constants, for jump and return-jump functions（过程间参数，关于转移和返回转移函数），640
interprocedural control-flow analysis（控制流分析），609-618
ICAN code for call graph construction（调用图构造），609-611
inside-out order（由内向外顺序），609
invocation order（调用顺序），609
outside-in order（由外向内顺序），609
with procedure-valued variables（有过程值变量的），612-618
reverse invocation order（逆调用顺序），609
separate compilation and（分开编译），611
types of actions for procedure-valued variables（过程值变量的动作类型），615-616
interprocedural data-cache usage（过程间数据高速缓存使用），689
interprocedural data-flow analysis（数据流分析），619-637
binding graph construction（结合图构造），623-627
call graph construction（调用图构造），628-631
depth-first search（深度为主搜索），632
flow-insensitive side-effect analysis（流不敏感副作用分析），619-633
flow-sensitive side-effect analysis（流敏感副作用分析），634-636
other issues in computing side effects（计算副作用中的其他问题），637
overview（概述），619
program summary graph（程序概要图），634-636
program supergraph（程序超图），634
strongly connected components（强连通分量），631-632
interprocedural register allocation（过程间寄存器分配），659-662
annotations（注释），660-661
compile-time（编译时），662
importance of（重要性），664
issues requiring special handling（需要特殊处理的问题），661-662
link-time（连接时），659-662
order of optimizations（优化的顺序），665-666
interval control-flow analysis（区间控制流分析），197-202. *See also* structural control-flow analysis（同

bbs.theithome.com

超星公司制作 请尊重作者版权

时参见结构控制流分析）
advantages（好处），173
basic steps（基本步骤），200-202
control trees（控制树），198, 199
minimal vs. maximal intervals（极小与极大区间），199,200
overview（概述），173
structural analysis compared to（结构分析比较），202
interval data-flow analysis（区间数据流分析），249-250
in POWER and PowerPC compilers（POWER和PowerPC编译器），722
reaching definitions example（到达-定值的例子），249-250
structural analysis compared to（结构分析比较），249
intraprocedural code positioning（过程内代码安置），677-681
algorithm（算法），678-681
order of optimizations（优化的顺序），678
overview（概述），677-678
procedure splitting and（过程分割），681
intraprocedural optimizations（过程内优化），607
invocation order（调用顺序），609
Irlam, Gordon, 768
irreducibility（不可归约性）
approaches to（方法），197
prevalence of（流行），196-197
See also improper intervals（同时参见非正常区间）
isolated expressions（孤立的表达式），413
iterated dominance frontiers（迭代的必经边界计算），254-256
iteration space（迭代空间），275
iteration-space traversal（迭代空间遍历），275,276, 277
iterative, defined（迭代，定义），218
iterative data-flow analysis（迭代的数据流分析），231-235
for backward problems（向后问题），235
iterative forward bit-vector problem（迭代的向前问题），218-223
worklist algorithm（工作表算法），232-233
iterative forward bit-vector problem（迭代的向前位向量问题），218-223
iterators, `for` statements (ICAN)（迭代符，`for`语句(ICAN)），39-40

J

Java, run-time support（Java，运行时支持），131-133
Johnson, Steven C., 165
Johnsson, Richard K., 820
join node（汇合结点），175
join operation, for lattices（汇合操作，关于格的），223, 225
Jones, Neil D., 225,262, 287, 290, 306
Joy, William N., 52
jump functions（转移函数），637-641

K

Kam, J. B., 227
Kennedy, Ken, 449, 620, 637, 642, 664, 694
Keppel, David, 771
Kerns, Daniel R., 545
keywords (ICAN)（关键字（ICAN））
list of（表），40
reserved words（保留字），25, 40
Kildall, Gary A., 227, 259
killing definitions（杀死定值），220
Kim, Ki-Chang, 807
Kleene closure, unary postfix operator (*), applied to functions, defined（克林闭包，一元前缀运算符(*)作用于函数，定义），235,248
Knobe, Kathleen, 525, 530
Knoop, Jens, 407, 443,597
Koblenz, Brian, 525, 530
Krawczyk, Hugo, 803
Kuck, David J., 637

L

labels, passing as arguments（标号，作为实参传递），118-119
label separator (:)（标号分割符）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75, 79, 80
in ICAN（ICAN中），37
label variables, in interprocedural side-effect analysis（标号变量，过程间副作用分析），637
Lam, Monica, 289, 568, 690, 698, 699, 738,769
Larus, James, 525,598, 767
last-write trees（最近写树），259
latestness of expressions, in partial-redundancy analysis（表达式的最迟性，在部分冗余删除中），412-413
lattices（格）
of bit vectors (BV^n)（位向量(BV^n)），224, 226
bottom value (⊥)（底值），223
bottom value (⊥), in ICAN（底值(⊥)，ICAN中），225, 364

bbs.theithome.com

distributive（分配的），224
greater than operator (⊐)（大于运算符(⊐)），225
greater than or equal to operator (⊒)（大于或等于运算符(⊒)），225
height（高度），226
integer constant-propagation (ICP) lattice（整常数传播(ICP)格），224-225, 228
join operator (⊔)（并运算符），223
less than operator (⊏)（小于运算符），225
less than operator (⊏), in ICAN（小于运算符，ICAN中），639
less than or equal to operator (⊑)（小于或等于运算符(⊑)），225
meet operator (⊓)（交运算符），223,225
meet operator (⊓), in ICAN（交运算符，ICAN中），232, 367, 639
monotone（单调的），226
overview（概述），223-226
partial order on（偏序），225
product of lattices（格的乘积），224
product operator (×)（乘积运算符），224
strongly distributive（强可分配的），236
top value (⊤)（顶值），223
top value (⊤), in ICAN（顶值，ICAN中），225,232, 364
lattices of monotone flow functions（单调流函数的格），235-236
composition operator (∘)（复合运算符），225
effective height（有效高度），226
fixed points（不动点），226-227
identity function (*id*)（恒等函数），235
Kleene closure operator, unary postfix (*)（克林闭包运算符，一元后缀（*)），235,248
nonempty closure operator, unary postfix (+)（非空闭包运算符，一元后缀（+)），235
lazy code motion（懒惰代码移动），407
lcc, 768
leaf routine, defined（叶例程，定义），472
leaf-routine optimization（叶例程优化），472-473. *See also* shrink wrapping（同时参见收缩包装）
applicability（实用性），472-473
order of optimizations（优化的顺序），477-478
overview（概述），477
Lee, Peter H., 131,132
Lengauer, Thomas, 185, 738
less than operator (<)（小于运算符(<)）
in LIR, MIR, and HIR（LIR、MIR和HIR中），74
in ICAN（ICAN中），27, 29
less than or equal to operator (<=)（小于或等于运算符(<=)）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74
in ICAN（ICAN中），27, 29
Lewis, Harry R., 348,729
lexical analysis（词法分析）
combining with syntactic analysis（与语法分析结合），3
described（描述），2
for Fortran（关于Fortran），3
lexicographically less than operator (≺)（词典序小于运算符(≺)），275
lexicographically less than or equal to operator (⪯)（词典序小于等于运算符(⪯)），276
lexicographically positive distance vector（词典序为正的距离向量），690-691
Li, Kai, 548
libraries（库）
interprocedural issues（过程间问题），664
shared（共享的），127-131
Lichtenstein, W., 817
lifetimes of variables（变量的生命期），44
linear-function test replacement（线性函数测试替换），447, 448-453
described（描述），447
ICAN code（ICAN代码），450-453
procedure（过程），448-450
link-time interprocedural register allocation（连接时过程间寄存器分配），659-662
linked-list representation bit-vector representation vs.（位向量表示的链表表示），757-759
of sequences（序列的），763
of sets（集合的），760
Linpack
procedure integration and（过程集成），465-466
propagating information（传播信息），657
LIR, 79-81. *See also* Graham-Glanville code-generator generator; intermediate code（参见Graham-Glanville代码生成器的产生器；中间代码）
array representation（数组表示），68-69
assignments（赋值），80
definition of（定义），79-81
dependence DAGs（依赖DAG），270
described（描述），4
differences from MIR（与MIR的不同），79-81, 492-494
Graham-Glanville code-generator generator（Graham-Glanville代码生成器的产生器），139, 140

bbs.theithome.com

ICAN representation（ICAN表示），81-82, 86-92
instructions as ICAN tuples（指令作为ICAN元组），91-92
loop code（循环代码），5
memory addresses（存储器地址），91
operators in IROper（IROper中的运算符），82
tail-call optimization（尾调用优化），463
LISP
compile-time interprocedural register allocation（编译时过程间寄存器分配），662
dynamic scoping（动态作用域），45
register allocator（寄存器分配器），525
run-time support（运行时支持），131-133
list scheduling（表调度），535, 537-543
algorithm（算法），540-541
approach to（方法），540
basic-block boundaries（基本块边界），537
delay function computation（延迟函数的计算），538, 539
dependence DAG（依赖DAG），537-543
heuristics（启发式），542
ICAN code（ICAN代码），538-541
NP-hard problem（NP困难的问题），537
performance and（性能），437
literal constants, as jump and return-jump functions（文字常数，作为转移和返回转移函数），640
live range, in register allocation by priority-based graph coloring（活跃范围，基于优先级的图着色寄存器分配中），524-526
live variables（活跃变量）
defined（定义），443,445
in interference graphs（冲突图中），494
live range（活跃范围），524
in register renaming（寄存器重命名中），564
live variables analysis（活跃变量分析），229, 443-446
backward data-flow analysis（向后数据流分析），445-447
live vs. dead variables（活跃变量与死去的变量），443,445
Lo, Jack L., 545
load operator, unary (↑), in machine grammars（取数运算符(↑)，机器语法中），139
loads, generating in symbol tables（取数指令，在符号表中生成），59-63
local common-subexpression elimination（局部公共子表达式删除），379-385
algorithm（算法），380-381
available expressions（可用表达式），379
binary case（两种情形），379-382
combining with global（与全局结合），394
example（例），382-384
fast implementation（快速实现），384-385
global vs.（全局的），378
overview（概述），379
repeating（重复），394-395
local copy propagation（局部复写传播），356-358
algorithm（算法），356-358
generalizing for extended basic blocks（推广到扩展基本块），361
global copy propagation and（全局复写传播），361
local methods of register allocation（寄存器分配的局部方法），483-485
local stack frames（局部栈帧）. *See* stack frames（参见栈帧）
local symbol-table management（局部符号表管理），47-49
balanced binary tree（平衡二叉树），48
hashing（散列），48-49
local transparency（局部透明性），408-410
locally anticipatable values（局部可预见值），410
logical and operator (&), in ICAN（逻辑与运算符(&)，ICAN中），27, 28
logical not operator, nnary (!)（逻辑非运算符，一元(!)）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
in ICAN（ICAN中），27, 28
logical or operator (∨), in ICAN（逻辑或运算符(∨)，ICAN中），27, 28
loop dependence tests（循环依赖测试），279-284
loop distribution（循环分布），694
loop fusion（循环合并），684, 693-694
loop header（循环首结点），191
loop interchange（循环交换），684, 691
loop invariants, removal of induction variables and（循环不变量，归纳变量删除），449
loop inversion（循环倒置），587-588
in DEC GEM compilers for Alpha（Alpha DEC GEM编译器中），729
described（描述），580, 587
determining that a loop is entered（判定进入一个循环），587-588
nested loops（嵌套循环），587-588
loop optimizations（循环优化），425-460
induction-variable optimizations（归纳变量优化），425-453
order of optimizations（优化的顺序），458-459
overview（概述），322, 425,457-459

bbs.theithome.com

transformations for data-cache optimization（数据高速缓存有关的转换），689-695
unnecessary bounds-checking elimination（不必要的边界检查删除），454-457
loop peeling（循环剥离），374, 684
loop permutation（循环置换），691
loop reversal（循环逆转），692
loop simplifications（循环化简），580, 586-587
loop skewing（循环倾斜），692-693
loop tiling（循环铺砌），694-695
data-cache organization and（数据高速缓存组织），697
locality and（局部性），695-698
loops not fully tilable in（循环不完全可铺砌），696, 697
tile loop（铺砌循环），694
tile size and（瓦片大小），697-698
loop transformations for data-cache optimization（数据高速缓存有关的循环转换），689-695
lexicographically positive distance vector（词典序为正的距离向量），690-691
locality and loop tiling（局部性与循环铺砌），695-698
loop distribution（循环分布），694
loop fusion（循环合并），693-694
loop interchange（循环交换），691
loop permutation（循环置换），691
loop reversal（循环逆转），692
loop skewing（循环倾斜），692-693
loop tiling（循环铺砌），694-698
overview（概述），689-691
types（类型），690
unimodular transformations（么模转换），690-693
loop unrolling（循环展开），559-562
architectural characteristics and（体系结构特征），560-561,562
combining with window scheduling（与窗口调度结合），553, 555
described（描述），532, 573
ICAN code（ICAN代码），561
instruction-level parallelism increase（指令级并行性增加），562
loop characteristics and（循环特征），561-562
order of optimizations（优化的顺序），573, 574-575
overview（概述），559
register renaming and（寄存器重命名），561
rolled loop（卷起的循环），559
trace scheduling and（踪迹调度），569
unrolling factor（展开因子），559
loop-carried dependence（循环携带的依赖），278
loop-independent dependence（循环无关的依赖），278
loop-invariant code motion（循环不变代码外提），397-407
algorithm（算法），397-400, 403-404
examples（例），400-403,404-406
identifying loop-invariant computations（标识循环不变计算），397-398
order of optimizations（优化的顺序），421
overview（概述），377, 397, 420
preventing errors in（预防错误），401-403
reassociation with（重结合），415-416
reductions（归约），406-407
loops（循环）.*See also* dominators; `for` statements; `if` statements; `while` statements（也参见必经结点; `for`语句; `if`语句; `while`语句）
canonical form（规范形式），689
canonical loop test（规范循环测试），275
control-flow analysis（控制流分析），191-196
dependences in（依赖关系中），274-279
dominance tree for（必经结点树），171-172
doubly nested（双层嵌套的），275-276
flow functions for structural analysis（结构分析的流函数），237-239
headers（首结点），191
iteration space（迭代空间），275
iteration-space traversal（迭代空间遍历），275, 276, 277
kinds of optimizations（优化的种类），6-7
natural loops（正常循环），191-193
nesting depth and register allocation（嵌套深度和寄存器分配），483,484
strongly connected components（强连通分量），193-196
well-behaved, in C（规则的，C中），425
low-level intermediate languages（低级中间语言），71-72. *See also* LIR (也见LIR)
order of optimizations（优化的顺序），326-327, 328
Low-level Intermediate Representation（低级中间表示）. *See* LIR（参见LIR）
low-level model of optimization（优化的低级模式），7-9
low-level optimizations（低级优化）. *See* control-flow and low-level optimizations（参见控制流低级优化）

M

machine idioms, defined（机器方言，定义），599
machine idioms and instruction combining（机器方言和

bbs.theithome.com

版权所有 翻印必究

指令归并），599-602
described（描述），580
examples（例），599-602
instruction combining（指令归并），599
Intel 386 family compilers（Intel 386系列编译器），739-740
machine idioms（机器方言），599
order of optimizations（优化的顺序），604
machine-level DAGs（机器级DAG），542-543
machine simulators（机器模拟器），767-768
Mansour, Yishay, 803
Markstein, Peter, 494
matrix multiplication（矩阵乘），671
maximal clique separators（最大分离子团），525-526
maximal intervals（极大区间），198-199
maximal munch algorithms（最大贪婪算法），141
maximal strongly connected components（强连通分量），194
maximum fixed point (MFP)（最大不动点(MFP)），227
may information alias analysis（可能信息别名分析），294, 295-296, 315
global optimization and（全局优化），323
interprocedural optimization and（过程间优化），608,664
optimization（优化），323
McFarling, Scott, 682
McKinley, K. S., 806
medium-level intermediate languages（中级中间语言），71, 72. *See also* MIR（也参见MIR）
DEC CIL, 727-728
Intel IL-1, 736-737
order of optimizations（优化的顺序），325-327
register allocation（寄存器分配），482-483
Sun IR, 709-711
Medium-level Intermediate Representation（中级中间表示）. *See* MIR（参见MIR）
meet operation（交运算）
for lattices（关于格的），223,225
meet-over-all paths (MOP) solntion（满足全路径(MOP)的解），227
meet-over-all paths (MOP) solution（满足全路径(MOP)的解），227
Mellor-Crummey, J. M., 806
member of set operator (∈), in ICAN（集合成员运算符(∈)，ICAN中），27, 28, 31, 34
memory addresses, LIR（存储器地址），91
memory hierarchy optimizations（存储层次优化），669-704
combined or unified cache（合并的或统一的高速缓存），670
data-cache impact（数据高速缓存的影响），670-671
data-cache optimization（数据高速缓存优化），687-700
effectiveness of caches（数据高速缓存的有效性），670
instruction-cache impact（指令高速缓存的影响），672
instruction-cache optimization（指令高速缓存优化），672-682
order of optimizations（优化的顺序），701-703
scalar replacement of array elements（数组元素的标量替换），682-687
scalar vs. memory-oriented optimizations（标量与面向存储器的优化），700
translation-lookaside buffer (TLB)（后备转换缓冲区），670
memory speed, disparity with processor speeds（存储器速度，与处理器速度的不一致），669
Mesa, alias determination（Mesa，别名判定），642-643
MFP. *See* maximum fixed point (MFP)（参见最大不动点(MFP)）
minimal intervals（最小区间），199, 200
MIPS compilers（MIPS编译器）
branch architecture and（分支体系结构），533
MIPS machine simulator（MIPS机器模拟器），767-768
mixed model of optimization in（优化的混合方式），9
MOST pipelining algorithm（MOST流水算法），567
procedure integration and（过程集成），469-470
type checking（类型检查），132
MIR, 73-78. *See also* intermediate code（也参见中间代码）
alternative PA-RISC code sequences（可选的PA-RISC代码序列），72
array representation（数组表示），68-69
assignments（赋值），75
changes to define HIR（定义为HIR的改变），78-79
changes to define LIR（定义为LIR的改变），79-81, 492-494
comments（注释），76
definition of（定义），78-79
described（描述），4
examples（例），76-78
expressions（表达式），84-85
forward substitution in（向前替代），396
`goto` instruction（`goto`指令），75-76
ICAN representation of（ICAN表示），81-85
identifiers（标识符），76
instructions（指令），75-76

bbs.theithome.com

instructions as ICAN tuples（作为ICAN元组的指令），82-84
integer constants（整常数），76
loop code（循环代码），5
operators（运算符），75
operators in IROper（IROper中的运算符），82
receives（接收），75
XBNF syntax of instructions（指令的XBNF语法），73-75
XBNF syntax of programs and program units（程序和程序单位的XBNF语法），74
mixed model of optimization（优化的混合方式），7-10
ML, run-time support（运行时支持），131-133
Modula-2
enumerated value representation（枚举值的表示），107
reducible flowgraphs and（可归约流图），196
scoping issues（作用域问题），52
module storage class（模块存储类），44
modulo operator (%), in ICAN（求模运算符(%), ICAN中），27, 29
monotone flow functions（单调流函数），226
MOP solution（MOP解）. *See* meet-over-all paths (MOP) solution（参见满足全路径(MOP)的解）
Morel, Etienne, 407, 422
MOST pipelining algorithm（MOST流水算法），567
move conditional（条件传送），571-572
move operation（传送操作），571-572
Mowry, Todd C., 699
Muchnick, Steven S., 225, 262, 264, 265, 287, 290, 306, 316, 373,542, 575, 576
multi-level intermediate languages（多级中间语言），72-73
multiplication operator (*)（乘法运算符(*)）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
in ICAN（ICAN中），27, 29
must information（一定信息）
alias analysis（别名分析），294, 295-296, 315
global optimization and（全局优化），323
interprocedural optimization and（过程间优化），608, 664
optimization（优化），323
Myers, E. A., 634

N

Nahshon, Itai, 803
name mangling（名字重塑），10
name scoping（名字作用域），663
names, in register allocation by graph coloring（名字，图着色寄存器分配中）. *See* webs（参见网）
naming conventions, ICAN data structures and rontines（命名约定，ICAN数据结构和例程），92-95
NaN (not a number), in ANSI/IEEE floating point（NaN（非数字），ANSI/IEEE浮点），342
natural loops（自然循环），191-193
algorithm（算法），192
preheaders（前置块），193
natural loop schema（自然循环图解），203
negation operator, unary (-)（取负运算符，一元(-)）
in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
in ICAN（ICAN中），27, 29
negative distance, in direction vectors（负距离，方向向量中）
>,278
-, 278
nested regions（嵌套的区域），198
Nickerson, Brian, 523
Nicolau, Alexandru, 287, 571,573
nil value (ICAN), overview（nil值(ICAN)，概述），24, 35
node splitting（结点分割），197
non-empty closure operator, nnary postfix (+), applied to functions（非空闭包运算符，一元后缀(+)，作用于函数），235
nonterminating computation（不终止的计算）. *See* infinite loop（参见无限循环）
nop, in delay slots（nop，在延迟槽中），535
normalized form for loops（循环的正规化形式），689
notation（表示法）. *See also* Informal Compiler Algorithm Notation (ICAN)（也参见非形式编译算法表示）
for numbers in this book（关于本书中的数字），17
not equal to operator（不相等运算符）
!=, in HIR, MIR, and LIR（!=，HIR、MIR和LIR中），74, 75
≠, in ICAN（≠，ICAN中），27, 28, 29, 30
not member of set operator (∉), in ICAN（不属于集合成员运算符，ICAN中），27, 31
NP-hard and NP-complete problems（NP困难的和NP完全的问题），537, 544, 573, 602, 637, 689
nullification（作废）. *See* branch scheduling（参见分支调度）
number notations（数字表示法），17

O

Odnert, Daryl, 662
"off-by-one" errors（差1错误），454

版权所有 侵权必究

bbs.theithome.com

超星公司制作 请尊重作者版权

Omega test（Omega测试），284
one or more copies operator, unary postfix (+), in XBNF notation（一次以上复制运算符，一元后缀(+)，XBNF表示中），19,20
one-pass compilers（一遍编译器）
  described（描述），3
  efficiency of（效率），3, 6
opcodes（伪代码）
  Alpha, 751-752
  Intel 386 family architecture（Intel 386系列体系结构），753
  PA-RISC, 754-755
  POWER and PowerPC（POWER和PowerPC），750
  SPARC, 748-749
operating-system interface（操作系统接口），3
operators（运算符）
  assignment statements (ICAN)（赋值语句(ICAN)），36, 38
  ICAN, 27-36
  HIR, 82
  LIR, 82
  MIR, 75, 82
  sets（集合），759
  size operator (ICAN)（大小运算符(ICAN)），36
  XBNF notation（XBNF表示），19-20
optimality principle（最优原理），160
optimization（优化）. *See also specific types of optimization*（也参见各种特殊类型的优化）
  criteria for（标准），320
  group I（第一组），323-324
  group II（第二组），324
  group III（第三组），324-325
  group IV（第四组），325
  importance of（重要性），6-7, 320-321, 323-325
  intermediate code and（中间代码），67-68
  as misnomer（用词不当），319
  order of optimizations（优化的顺序），325-328
  performance and（性能），319, 320, 321
  safe or conservative（安全的或保守的），319-320
optimizing compilers（优化编译器）
  aggressive, placement of optimizations in（激进的，优化安排中），11-14
  low-level model（低级模式），7-9
  mixed model（混合模式），7-10
  placement of optimizations（优化安排），11-14
  structure of（结构），7-11
optional operator ([...]), in XBNF notation（可选运算符([...])，XBNF表示中），19, 20
order of optimizations（优化的顺序）
  control-flow and low-level optimizations（控制流与低级优化），603-604
  early optimizations（前期优化），372-373
  instruction scheduling（指令调度），532, 574-575
  interprocedural optimizations（过程间优化），665-666
  loop optimizations（循环优化），458-459
  memory hierarchy optimizations（存储层次优化），701-703
  overview（概述），11-14, 325-328
  percolation scheduling（渗透调度），532, 574
  postpass or peephole optimizations（后遍或窥孔优化），603
  procedure optimizations（过程优化），477-478
  redundancy elimination（冗余删除），327, 421-422
  register allocation（寄存器分配），481, 482-483,527-528
  trace scheduling（踪迹调度），532, 574
output dependence（输出依赖）
  defined（定义），268
output-dependence operator(δ^o)（输出依赖运算符(δ^o)），268
outside-in order（由外向内顺序），609

P

PA-RISC
  branch architecture（分支体系结构），533
  floating-point register *i* (%fr*i*)（浮点寄存器*i* (%fr*i*)），754
  floating-point register *i*, left part (%fr*i*L)（浮点寄存器*i*，左部(%fr*i*L)），754
  floating-point register *i*, right part (%fr*i*R)（浮点寄存器*i*，右部(%fr*i*R)），754
  integer register *i* (%r*i*)（整数寄存器*i* (%r*i*)），754
PA-RISC compilers（PA-RISC编译器）
  alternative sequences for MIR（可选的MIR序列），72
  assembly language（汇编语言），753-755
  branch architecture and（分支体系结构），533
  low-level model of optimization in（优化的低级模式），9
  machine idioms and instruction combining（机器方言和指令归并），599, 600-602
  SLLIC in（SLLIC中），68
  type checking（类型检查），132
  UCODE in（UCODE中），68
packed records（压缩的记录），108
pair binding graph（成对结合图），649-650
  marking algorithm（标记算法），650, 652-653
ParaFrase programming environment（ParaFrase程序设计

bbs.theithome.com

环境），637
parallel computation graphs (PSGs)（并行计算图），571-573
parameter-passing（参数传递）
in C, aliases and（C中，别名），301
in Fortran 77, aliases and（Fortran 77中，别名），298-299
in-line expansion and（内嵌扩展），471-472
interprocedural optimization and（过程间优化），608
jump and return-jump functions（转移和返回转移函数），641
in Pascal, aliases and（Pascal中，别名），299-300
procedure integration and（过程集成），468-469
parameter-passing (run-time)（参数传递（运行时）），116-119
call by name（传名字），118
call by reference（传地址），117-118
call by result（传结果），117
call by value（传值），117
call by value-result（传值得结果），117
in flat register file（平面寄存器文件），121-123
invoking procedures（调用过程），119-126
labels（标号），118-119
mechanisms（机制），116-117
procedure-valued variables（过程值变量），126
with register windows（用寄存器窗口），123-126
on run-time stack（在运行时栈上），123
parameters（哑参）. *See also* arguments; parameter-passing (run-time)（同时参见实参；参数传递（运行时））
defined（定义），117
parsing（分析）. *See* syntactic analysis（参见语法分析）
partial redundancy（部分冗余），defined（定义），407
partial-redundancy analysis（部分冗余分析），230
partial-redundancy elimination（部分冗余删除），407-415
advantages of modern formulation（现代公式的优点），378,421
critical edges（关键边），407, 408
delayed expressions（延迟的表达式），411-412
earliestness（最早性），411
future trends（未来的趋势），744
globally anticipatable valnes（全局可预见值），410-411
implementing（实现），414
isolated expressions（孤立的表达式），413
latestness（最迟性），412-413
lazy code motion（懒惰代码移动），407
local transparency（局部透明性），408-410
locally anticipatable values（局部可预见值），410
order of optimizations（优化的顺序），421
overview（概述），377, 407, 420
reassociation with（重结合），415-416
value numbering vs.（值编号），343
partitioning algorithm for global value numbering（全局值编号的划分算法），351-355
Pascal
alias determination（别名判别），642-643
aliases in（别名），299-300, 304, 305
basic-block boundaries（基本块边界），174
character string representation（字符串表示），108-109
enumerated value representation（枚举值的表示），107
loop unrolling（循环展开），559
pointers（指针），258
scoping and symbol-table structure（作用域和符号表结构），49, 50
unnecessary bounds-checking elimination for（不必要的边界检查），454-457
pass-through parameter, for jump and return-jump functions（经传递的参数，关于转移和返回转移函数），641
path expressions（路径表达式），260
path strings（路径字符串），163
PCGs. *See* parallel computation graphs (PCGs)（参见并行计算图(PCG)）
PDGs. *See* program-dependence graphs (PDGs)（见程序依赖图(PDG)）
PDP-11 BLISS compiler（PDP-11 BLISS编译器）. *See* BLISS compiler（参见BLISS编译器）
peephole optimizations（窥孔优化）
Alpha compilers（Alpha 编译器），729, 730
branch prediction（分支预测），580, 597-599
described（描述），579, 597, 603-604
Intel compilers（Intel编译器），739
machine idioms and instruction combining（机器方言和指令归并），580, 599-602
order of optimizations（优化的顺序），603
POWER and PowerPC compilers（POWER和PowerPC编译器），723
SPARC compilers（SPARC编译器），713
Pelegri-Llopart, Eduardo, 165
Pentium. *See* Intel 386 family architecture（见Intel 386系列体系结构）
PentiumPro. *See* Intel 386 family architecture（见Intel 386系列体系结构）
branch architecture（分支体系结构），533

bbs.theithome.com

percent character (%%), in ICAN（百分比符号(%%), ICAN中），29
percolation scheduling（渗透调度），571-573
algorithm（算法），572
computation nodes（计算结点），571
described（描述），532
meta-transformations（元转换），573
order of optimizations（优化的顺序），532, 574
transformations（转换），571-573
write-live dependence and（写－活跃依赖），571
perfect static predictor（完美静态预测器），598
performance（性能）. *See also* efficiency（也参见效率）
algebraic simplifications and（代数化简），334-335
instruction scheduling and（指令调度），573
interprocedural optimization and（过程间优化），608
list scheduling and（表调度），437
as optimization criterion（作为优化标准），320
optimizations and（优化），319, 320, 321
procedure integration and（过程集成），468, 469-470
processor speeds vs. memory speeds（处理器速度与存储器速度），669
register allocation by graph coloring and（图着色寄存器分配），481
trace scheduling and（踪迹调度），570
period, of a recurrence（周期，重现），683
ϕ-function, static single assignment form（ϕ函数，静态单赋值形式），252, 253
Pinter, Ron Y., 803
Pinter, Shlomit S., 526
pipeline architectures（流水体系结构），531, 532-533
interlocks（互锁），532-533
pipelining（流水）. *See* software pipelining（参见软流水）
PL.8 compiler（PL.8编译器），485, 718
PL/I, character string representation（PL/I，字符串表示），108-109
placement of optimizations（优化的安排）. *See* order of optimizations（见优化的顺序）
PLT. *See* procedure linkage table (PLT)（参见过程连接表(PLT)）
pointer indirection operator, unary (*), in HIR, MIR, and LIR（指针间接运算符，一元(*)，HIR、MIR和LIR中），74, 75, 79
pointers（指针）
in C, aliases and（C中，别名），294, 300-301
data-flow analysis（数据流分析），258
described（描述），110-111
in Fortran 77 Cray extensions, aliases and（Fortran 77 Cray扩充中，别名），299
in Fortran 90, aliases and（Fortran 90中，别名），301
interprocedural alias analysis and（过程间别名分析），654-656
in interprocedural side-effect analysis（过程间副作用分析中），637
in Pascal, aliases and（Pascal中，别名），299, 300
to procedure descriptors（指向过程描述字），126
run-time representation（运行时的表示），108
Polish-prefix notation（前缀波兰表示）
in Graham-Glanville code-generator generator（Graham-Glanville代码生成器的产生器中），142
overview（概述），101
semantics-directed parsing（语义制导的分析），159-160
polymorphic languages, run-time support（多态语言，运行时支持），131-133
polynomial functions, for jump and return-jump functions（多项式函数，转移和返回转移函数），641
portability, low-level vs. mixed optimization（可移植性，低级与混合的优化），8-9
postdominance（后必经关系）
defined（定义），182
tree（树），284-285
position-independent code（位置无关代码），128, 713, 741
position indicator (.), in LR(1) items（位置指示符(.)，LR(1) 项目中），144
positive distance, in direction vectors（正距离，方向向量中）
<,278
+,278
postdominators（后必经结点），182
postorder traversal（后序遍历），180, 181
postpass optimizations（后遍优化）. *See* peephole optimizations（参见窥孔优化）
POWER and PowerPC compilers（POWER和PowerPC编译器），716-725
alias information（别名信息），721
assembly code examples（汇编代码的例子），723-725
assembly language（汇编语言），749-750
branch architecture and（分支体系结构），533
computation table (CT)（计算表(CT)），719
control-flow analysis（控制流分析），721
data-flow analysis（数据流分析），722
data prefetching（数据预取），698
development（开发），718
instruction scheduler（指令调度器），723
languages supported（支持的语言），718

bbs.theithome.com

low-level model of optimization in（优化的低级模式），9
machine idioms and instruction combining（机器方言和指令归并），599, 601-602
optimizer transformations（优化器转换），722
order of optimizations（优化的顺序），705
peephole optimizations（窥孔优化），723
POWER architecture（POWER体系结构），716-717
PowerPC architecture（PowerPC体系结构），717
procedure descriptor table（过程描述符表），719
procedure list（过程列表），719
register allocators（寄存器分配器），523,723
register coalescing（寄存器合并），723
scavenging, in register allocation（清扫，寄存器分配中），723
structure（结构），718-719
symbolic register table（符号寄存器表），719
TOBEY common back end（TOBEY公共后端），719-723
"wand waving"，722
XIL code（XIL代码），719-721
YIL code（YIL代码），721
power test（Power测试），284
precedes-in-execution-order operator（◁）（执行顺序上先于运算符(◁)），267, 275
predecessor basic blocks（前驱基本块），175
register allocation and（寄存器分配），484
prefetching（预取）
data prefetching（数据预取），688,698-700
instruction prefetching（指令预取），672-673
prefix unary operators (ICAN)（前缀一元运算符(ICAN)）. *See* unary operators (ICAN)（参见一元运算符（ICAN））
preheaders of natural loops（正常循环的前置块），193
preorder traversal（前序遍历），179-181
structural control-flow analysis（结构控制流分析），207
preprocessors（预处理）
described（描述），10
high-level intermediate languages in（高级中间语言中），69-70
preserving definitions（保持定值），220
priority-based graph coloring（基于优先级的图着色），524-525. *See also* register allocation by graph coloring（同时参见图着色寄存器分配）
procedure and block placement in instruction cache（指令高速缓存中过程和基本块的放置），676-677
procedure-call side effects（过程调用副作用）. *See* side effects of procedure calls
procedure-call statements (ICAN), form（过程调用语句(ICAN)，形式），38
procedure cloning（过程克隆），607-608,657
order of optimizations（优化的顺序），665
procedure declarations (ICAN)（过程说明（ICAN））
overview（概述），23, 24
syntax（语法），26-27
procedure descriptors（过程描述字），126
procedure integration（过程集成），465-470
breadth of（宽度），466-468
effects of（效果），469-470
implementation（实现），468-469
order of optimizations（优化的顺序），477
performance and（性能），468
summary（概要），476
procedure linkage table (PLT)（过程连接表(PLT)），130, 131
procedure optimizations（过程优化），461-479
described（描述），322, 461
in-line expansion（内嵌扩展），470-472
leaf-routine optimization（叶例程优化），472-473
order of optimizations（优化的顺序），477-478
overview（概述），476-477
procedure integration (automatic inlining)（过程集成(自动内联)），465-470
shrink wrapping（收缩包装），472, 473-476
tail-call and tail-recursion elimination（尾调用和尾递归删除），461-465
procedure parameters, in interprocedural side-effect analysis（过程参数，过程副作用分析中），637
procedure sorting（过程排序），673-676
algorithm（算法），673-676
overview（概述），673
procedure splitting and（过程分割），681
procedure specialization（过程说明）. *See* procedure cloning（参见过程克隆）
procedure splitting（过程分割），681-682
procedure-valued variables（过程值变量），126
interprocedural control-flow analysis（过程间控制流分析），612-618
procedures（过程），119-126. *See also* parameter-passing (run-time)（也参见参数传递（运行时））
call（调用），119
epilogue（出口处理），120
extended formal parameters（扩展的形式参数），643
Fortran 90, aliases and（Fortran 90，别名），301-302
Pascal, aliases and（Pascal，别名），300
phases in run-time execution（运行时执行的阶段），119-120

版权公司制作请勿盗作者版权

bbs.theithome.com

prologue（入口处理），119-120
static link（静态链），114
value graphs（值图），349-351
product of lattices（格的乘积），224
production, XBNF notation（产生式，XBNF表示），19-20
Production-Quality Compiler Compiler (PQCC)（产品质量编译器的编译器（PQCC）），730
Proebsting, Todd, 273,769
profiling tools（剖面分析工具），770-771
program management, interprocedural issues（程序管理，过程间的问题），663-664
program points alias-propagator flow functions（程序点别名传播流函数），308
defined（定义），295, 303
functions mapping（函数映射），304
program summary graph（程序概要图），634-636
program supergraph（程序超图），634
program syntax (ICAN)（程序语法（ICAN）），25
program verification, data-flow analysis for（程序证明，数据流分析用途），261-262
program-dependence graphs (PDGs)（程序依赖图（PDG）），284-286
constructing CDGs for（构造CDG），284-286
control-dependent nodes（控制依赖结点），284, 286
in instruction scheduling（指令调度中），723
region nodes（区域结点），284, 285
"programming in the large" issues（大规模程序设计中的问题），663-664
Prolog, run-time support（Prolog，运行时支持），131-133
prologue of procedures（过程的入口处理），119-120
shrink wrapping and（收缩包装），472, 473
proper ancestor（固有祖先），185
proper interval（正常区间），204
pruned SSA form（修剪过的SSA形式），258
pruning interference graphs（修剪冲突图），488, 503-506
pseudo-operations, SPARC（伪操作，SPARC），749
PSPACE-hard（PSPACE困难的），612

Q

QPT, 770
QPT_STATS, 770
quadruples（四元式）
infix form（中间形式），96
translation from trees（从树转换），99-100
translation to trees（转换到树），98-99
translation to triples（转换到三元式），97
trees vs.（树），98
triples vs.（三元式），96
quotation mark character (%"), in ICAN（双引号字符(%")，ICAN中），29

R

R-coloring（*R*着色）. *See also* register allocation by graph coloring（参见图着色寄存器分配）
defined（定义），485
degree < *R* rule（度<*R*规则），503-506
Rainish, Victor, 548
range checking（范围检查），454. *See also* unnecessary bounds-checking elimination（同时参见不必要的边界检查删除）
reaching-definitions analysis（到达－定值分析），218-223
basic blocks and（基本块），218
correspondence between bit-vector positions, definitions, and basic blocks（位向量的位置、定值和基本块之间的对应），219, 220
described（描述），229
forward structural data-flow analysis（向前结构数据流分析），242-244
interval data-flow analysis（区间数据流分析），249-250
iterative forward bit-vector problem（向前迭代位向量问题），218-223
killing definitions（杀死定值），220
preserving definitions（保持定值），220
undecidability in（不可判定的），219
reading flow among chapters in this book（阅读本书各章节的流程），14-16
`real` ICAN type（ICAN实类型）
binary operators（二元运算符），29
overview（概述），29
reassociation（重结合） *See also* algebraic simplifications and reassociation（也参见代数化简和重结合）
with common-subexpression elimination（公共子表达式删除），385,415-416
with loop-invariant code motion（循环不变代码外提），415-416
with partial-redundancy elimination（部分冗余删除），415-416
redundancy elimination and（冗余删除），415-416
receives, MIR instruction type（接受，MIR指令类型），75
recompilation, interprocedural issues（重新编译，过程间问题），664

bbs.theithome.com

record, (record type constructor), in ICAN（记录，（记录类型构造符），ICAN中），23
records（记录）
  data-flow analysis（数据流分析），258, 259
  packed and unpacked（压缩的和非压缩的），108
  run-time representation（运行时的表示），108
records, in ICAN（记录，ICAN中）
  binary operators, in ICAN（二元运算符，ICAN中），34
  constant delimiters (<...>)（常数分界符(<...>)），27, 33
  constants（常数），33
  overview（概述），33-34
  type constructor (record {...})(类型构造符(record {...})),23, 26, 33
recursion（递归）
  in C, aliases and（C中，别名），301
  in Fortran 90, aliases and（Fortran 90中，别名），301
  inlining and（内联），468
  in interprocedural side-effect analysis（过程间副作用分析中），637
  in Pascal, aliases and（Pascal中，别名），300
  tail-recursion elimination（尾递归删除），320-321
reducibility of flowgraphs（流图的归约性），196-197
reductions, loop-invariant code motion（归约，循环不变代码外提），406-407
redundancy elimination（冗余删除），377-423
  code hoisting（代码提升），417-420
  common-subexpression elimination（公共子表达式删除），378-396
  described（描述），322
  loop-invariant code motion（循环不变代码外提），397-407
  order of optimizations（优化的顺序），327, 421-422
  overview（概述），377-378
  partial-redundancy analysis（部分冗余分析），230
  partial-redundancy elimination（部分冗余删除），407-415
  reassociation（重结合），415-416
region（区域），175
region nodes of PDGs（PDG的区域结点），284, 285
register allocation（寄存器分配），481-530, 659-662. *See also* interprocedural register allocation; register allocation by graph coloring（也参见过程间寄存器分配，图着色寄存器分配）
  candidates for allocation（分配的候选者），483-484
  combining with scheduling（与调度结合），526
  DEC compilers（DEC编译器），730
  described（描述），322, 482
  by graph coloring（采用图着色），485-524
  importance of（重要性），481
  instruction scheduling and（指令调度），545
  Intel 386 family compilers（Intel 386系列编译器），740
  interprocedural（过程间的），659-662
  local methods（局部方法），483-485
  order of optimizations（优化的顺序），481, 482-483,527-528
  other approaches（其他方法），525-526
  overview（概述），481-483
  POWER and PowerPC compilers（POWER和PowerPC编译器），723
  priority-based graph coloring（基于优先级的图着色），524-525
  scalar replacement of array elements（数组元素的标量替换），682-687
  SPARC compilers（SPARC编译器），713
  SPUR LISP compiler（SPUR LISP编译器），525
register allocation by graph coloring（图着色寄存器分配），485-524
  adjacency lists（邻接表），487, 496-497, 498,499, 530
  adjacency matrixes（邻接矩阵），487, 495-496, 497, 530
  basic steps（基本步骤），485
  candidates for allocation（分配的候选者），489-494
  degree（度），494
  degree < *R* rule（度<*R*规则），503-506
  edge splitting（边分割），509
  examples（例），485-486, 510-521
  heuristics used in register spilling（寄存器溢出所使用的启发式），501,522-523
  interference graph（冲突图），485, 486, 487, 488,494-497, 503-506
  interferences（冲突），481
  machine independence（机器依赖），524
  NP-completeness（NP完全的），482
  order of optimizations（优化的顺序），527-528
  other issues（其他问题），521-524
  overview（概述），481-482, 485-486
  performance and（性能），481
  priority-based graph coloring（基于优先级的图着色），524-525
  register assignment（寄存器指派），488, 506, 508
  register coalescing（寄存器合并），487, 497-500
  register pairs（寄存器偶对），523,529
  register pressure（寄存器压力），407, 506
  rematerialization（重回寄存器），488, 501, 509, 523-524, 529, 530

bbs.theithome.com

spill costs computation（溢出代价计算），487-488,501-503
spilling symbolic registers（溢出符号寄存器），488, 506-511
top-level structure（高层结构），486-489
two-address instructions（两地址指令），499, 529
webs（网），486, 489-494
register allocation by priority-based graph coloring（基于优先级的图着色），524-525
register assignment overview（寄存器指派概述），482
register allocation by graph coloring（图着色寄存器分配），488,506, 508
register coalescing（寄存器合并），487, 497-500
functions performed by（所执行的函数），499
ICAN code（ICAN代码），498-500
overview（概述），487, 497-498
POWER and PowerPC compilers（POWER和PowerPC编译器），723
register pairs, in register allocation by graph coloring（寄存器偶对，图着色寄存器分配中），499, 523, 529
register pipelining（寄存器流水）. *See* scalar replacement of array elements（参见数组元素的标量替换）
register pressure（寄存器压力），407, 506
register renaming（寄存器重命名），564-567
in branch scheduling（分支调度），534
described（描述），532, 573
determining register sets（确定寄存器集合），564-565
in extended basic blocks（扩展基本块），565
ICAN code（ICAN代码），565, 566
loop unrolling and（循环展开），561
order of optimizations（优化的顺序），574-575
register windows, parameter-passing with（寄存器窗口，参数传递），123-126
registers（寄存器）
Alpha integer register names（Alpha寄存器名），751
anticipatable（可预测的），473
common-subexpression elimination and（公共子表达式删除），396
contention for（内容），110-111
DEC Alpha register names（DEC Alpha寄存器名），751
dividing into classes（划分为几类），120
flat register file, parameter-passing in（平面寄存器文件，参数传递），121-123
IBM POWER and PowerPC register names（IBM POWER和PowerPC寄存器名），749-750
ICAN representation（ICAN表示），82
importance of（重要性），109-110
Intel 386 family register names（Intel 386系列寄存器名），752
issues and objectives（问题和目标），110
managing for procedures（关于过程的管理），120-123
PA-RISC register names（PA-RISC寄存器名），754
partial-redundancy elimination and（部分冗余删除），407
register windows, parameter-passing with（参数传递），123-126
run-time usage（运行时的用法），109-111, 121-126
shrink wrapping and（收缩包装），473-474
SPARC register names（SPARC寄存器名），748
Reif, John R., 348, 729
related topics not covered in this book（本书没有包含的相关主题），16
relational operators, algebraic simplifications（关系运算符，代数化简），333-334
rematerialization, in register allocation by graph coloring（重回寄存器，图着色寄存器分配），488,501,509, 523-524,529, 530
removal of induction variables（归纳变量删除），447-453
ICAN code（ICAN代码），450-453
loop invariants and（循环不变量），449
overview（概述），447-448
procedure（过程），448-450
situations where advisable（可取的情形），447
Renvoise, Claude, 407, 422
repeat statements（repeat语句）
ICAN, 40
loop inversion（循环倒置），587-588
reserved words (ICAN)（保留字（ICAN)），25, 40
resource vectors（资源向量），271
results（结果）. *See* return values（参见返回值）
`return`, MIR, 76
`return-`, 655
return statements (ICAN), form（返回语句（ICAN)，形式），38
return values（返回值）
call by result parameter-passing (run-time)（传结果参数传递（运行时)），117
described（描述），111
return-jump functions（返回转移函数），637-641
reverse extended basic blocks（反扩展基本块）
definition（定义），175
in instruction scheduling（指令调度中），598
Richardson, Stephen E., 608, 664
RISCS

bbs.theithome.com

aggregation of global references（全局引用的聚合），663
branch architectures（分支体系结构），533
constant propagation and（常数传播），362-363
data prefetching（数据预取），698
instruction prefetching（指令预取），672
machine idioms and instruction combining（机器方言和指令归并），599-602
pipelining（流水），531
register allocation（寄存器分配），483
register assignment（寄存器指派），482
register usage（寄存器用法），110
Rodeh, M., 548
Rogers, Anne, 548
rolled loop（卷起的循环），559
Rosen, Barry K., 202, 250
Rosenberg, Scott, 768
Rothberg, Edward E., 738
Roubine, Olivier, 52
routines, ICAN naming（例程，ICAN命名），92-95
row-major order for arrays（数组的行为主顺序），107
run-time stack（运行时栈）
overview（概述），114-116
parameters passed on（其上的传递参数），123
run-time support（运行时支持），105-136
Application Binary Interface (ABI) standards（应用二进制接口(ABI)标准），105, 134
importance of（重要性），4-6
local stack frames（局部栈帧），111-114
parameter-passing disciplines（参数传递规则），116-119
procedure prologues, epilogues, calls, and returns（过程的入口处理、出口处理、调用和返回处理），119-126
register usage（寄存器用法），109-111
run-time stack（运行时栈），114-116
shared objects（共享对象），127-131
symbolic and polymorphic language support（符号和多态语言支持），131-133
Rüthing, Oliver, 407, 443, 597
Ruttenberg, John, 567
Rymarczyk, J. W., 531,575

S

safe optimizations（安全优化），319-320
safe positions, in instruction scheduling（安全位置，指令调度中），543
Santhanam, Vatsa, 662
saxpy(), Linpack procedure（saxpy()，Linpack过程），465-466, 657
scalar optimizations future trends（标量优化未来的趋势），744
memory-oriented optimizations vs.（面向存储器的优化），700
scalar replacement of aggregates（聚合量标量替代），331-333
main procedure（主过程），331, 332, 333
order of optimizations（优化的顺序），372
overview（概述），371
simple example（简单的例子），331-332
scalar replacement of array elements（数组元素的标量替换），682-687
examples（例），682-683
loop fusion（循环合并），684
loop interchange（循环交换），684
for loop nests with no conditionals（用于不含条件的循环嵌套），683-684
loops with ifs（含if的循环），684, 685
for nested loops（用于嵌套循环），684, 686-687
order of optimizations（优化的顺序），702-703
overview（概述），701
SCCs. *See* strongly connected components（强连通分量）
scheduling（调度）. *See* instruction scheduling（见指令调度）
Scheme, run-time support（Scheme，运行时支持），131-133
Schiffman, Alan M., 133
scope（作用域）
closed scopes（闭作用域），52-54
defined（定义），43
dynamic scoping（动态作用域），45
global symbol-table management and（全局符号表管理），49-54
name scoping（名字作用域），663
types of storage classes（存储类的类型），44
visibility of variables（变量的可见性），43-44
scratch registers（草稿寄存器），120
search-variable expansion（搜索变量扩张），562-563
select from set operator, unary (◆), in ICAN（从集合中选择运算符，一元(◆)，ICAN中），24, 27, 31
SELF, run-time support（SELF，运行时支持），131-133
semantic checking, described（语义检查，描述），2
semantics-directed parsing（语义制导的分析），159-160
semidominators（半必经结点），185
separable array references（可分的数组引用），282
separate compilation（分开编译）
interprocedural analysis and optimization and（过程间

bbs.theithome.com

分析和优化），608
interprocedural control-flow analysis and（控制流分析），611
separated list operator（⋈），in XBNF notation（分隔表运算符，XBNF表示中），19, 20
separators, in ICAN（分隔符，ICAN中），21
sequence instruction, MIR（sequence指令，MIR），76
sequence of, (sequence type constructor), in ICAN（sequence of，（序列类型构造符），ICAN中），23
sequence of optimizations（优化序列）. *See* order of optimizations（参见优化的顺序）
sequence types, in ICAN（序列类型，ICAN中）
binary operators overview（二元运算符，概述），32
concatenation operator (⊕)（连接运算符(⊕)），24, 27, 32
constants（常数），32, 33
constant delimiters ([...])（常数分界符([...])），24. 27, 32
member deletion operator (⊖)（成员删除运算符(⊖)），24, 27, 32
member selection operator (↓)（成员选择运算符(↓)），27, 32
overview（概述），32-33
sequences, representation（序列，表示），763
set of (set type constructor), in ICAN（set of（集合类型构造符），ICAN中），23
sets（集合）
operators（运算符），759
representation（表示），759-763
run-time representation（运行时的表示），109
set types, in ICAN（集合类型，ICAN中）
binary operators（二元运算符），31
constant delimiters ({...})（常数分界符({...})），27, 31
constants（常数），31
intersection operator (∩)（交运算符(∩)），27, 31
overview（概述），31-32
selection (◆), unary operator（选择(◆)，一元运算符），31-32
union operator (∪)（并运算符），27, 31
sgefa(), Linpack procedure（sgefa()，Linpack过程），465-466, 467, 657
Shade, 771
shared compiler components（共享编译器成分），9-10
shared libraries（共享库）. *See* shared objects（参见共享对象）
shared objects（共享对象），127-131
accessing external variables（访问外部变量），128, 129
advantages（优点），127
global offset table (GOT)（全局偏移表(GOT)），129, 130
performance impact（性能影响），128
position-independent code（位置无关代码），128
procedure linkage table (PLT)（过程连接表(PLT)），130, 131
semantics of linking（连接的语义），127-128
table of contents example（内容例子的表），128
transferring control between（在其间转换控制），129-130
transferring control within（在其内转换控制），128-129
Sharir, Micha, 205, 210
Sharlit, 259-261
shrink wrapping（收缩包装），473-476. *See also* leaf-routine optimization example（参见叶例程优化例子），474-476
implementation（实现），473-474
order of optimizations（优化的顺序），477-478
overview（概述），472, 473,477
side effects of procedure calls（过程调用副作用）
flow-insensitive side-effect analysis（流不敏感副作用分析），619-633
flow-sensitive side effect analysis（流敏感副作用分析），634-636
other issues in computing（计算中的其他问题），637
Silberman, Gabriel, 807
simple loop residue test（简单循环余数测试），284
Simpson, Taylor, 355, 414, 443
single quotation mark character (%'), in ICAN（单引号字符(%')，ICAN中），29
single-valued（单值的），34
site-independent interprocedural constant propagation（位置无关的过程间常数传播），637
site-specific interprocedural constant propagation（位置特殊的过程间常数传播），637
SIV tests（SIV测试），283
size operator (|...|), in ICAN（大小运算符，ICAN中），24, 27, 36
SLLIC, integer multiply and divide operands in（SLLIC，整数乘和除操作数），73
slotwise data-flow analysis（位置式数据流分析），250-251
Smalltalk, run-time support（Smalltalk，运行时支持），131-133
Smotherman, Mark, 542
SNOBOL, run-time support（SNOBOL，运行时支持），

bbs.theithome.com

131-133
Soffa, Mary Lou, 525, 530, 700
software pipelining（软流水）, 548-569
circular scheduling（环调度）, 565
described（描述）, 532
disambiguating memory references（消除存储引用的二义性）, 551
hierarchical reduction（层次归约）, 568-569
loop unrolling（循环展开）, 532, 559-562
for loops containing conditionals（含条件的for循环）, 568-569
MOST pipelining algorithm（MOST流水算法）, 567
order of optimizations（优化的顺序）, 574-575
other approaches（其他方法）, 565-568
overview（概述）, 548-551,573
performance and（性能）, 573
pipeline architectures（流水线体系结构）, 531, 532-533
register pipelining（寄存器流水）(scalar replacement of array elements)（数组元素的标量替换）, 682-687
register renaming（寄存器重命名）, 532, 564-567
testing loop iterations（测试循环迭代）, 551
unroll-and-compact（展开－压实）, 555-558
variable expansion（变量扩张）, 532, 562-564
window scheduling（窗口调度）, 551-555
software resources（软件资源）, 767-771
code-generator generators（代码生成器的产生器）, 769-770
compilers（编译器）, 768-769
machine simulators（机器模拟器）, 767-768
profiling tools（剖面分析工具）, 770-771
the Internet（互联网）, 767
sorting, local variables in stack frame（栈帧中的局部变量）, 56-58
sources of aliases（别名的原因）, 296-297
Spa, 768
space, as optimization criterion（空间，作为优化标准）, 320
SPARC
architecture（体系结构）, 591,707-708
floating-point register *i* (%f *i*)（浮点寄存器*i* (%f *i*)）, 748
integer general register *i* (%g*i*)（整数通用寄存器*i* (%g*i*)）, 748
integer in register *i* (%i*i*)（整数输入寄存器*i* (%i*i*)）, 748
integer local register *i* (%l*i*)（整数局部寄存器*i* (%l*i*)）, 748
integer out register *i* (%o*i*)（整数输出寄存器*i* (%o*i*)）, 748
integer register *i* (%r*i*)（整数寄存器*i* (%r*i*)）, 748
machine simulator（机器模拟器）, 768
SPARC compilers（SPARC编译器）, 707-716
asm+ code（ams+ 代码）, 711-712
assembly code examples（汇编代码的例子）, 713-716
assembly language（汇编语言）, 747-749
branch architecture and（体系结构）, 533
code generator phases（代码生成遍）, 712-713
conditional moves（条件传送）, 591
data prefetching（数据预取）, 698
dependence analysis（依赖分析）, 711
development（开发）, 707
global optimizer（全局优化）, 710-711
instruction prefetching（指令预取）, 672
languages supported（支持的语言）, 708
levels of optimization supported（所支持的优化级别）, 710
mixed model of optimization in（优化的混合方式）, 9
optimizer transformations（优化转换）, 711
position-independent code（位置无关代码）, 713
procedure linkage table (PLT)（过程连接表(PLT)）, 131
register windows for successive procedures（连续过程的寄存器窗口）, 125
stack frame layout with register windows（带寄存器窗口的栈帧安排）, 126
Sun IR, 71, 72, 709-710
type checking（类型检查）, 132
yabe back end（yabe后端）, 709-710
sparse conditional constant propagation（稀有条件常数传播）, 362-371. *See also* constant propagation（也参见常数传播）
algorithm（算法）, 364-366
code for（代码）, 364-367
constant propagation（常数传播）, 362-363
examples（例）, 366-371
order of optimizations（优化的顺序）, 372
overview（概述）, 371-372
repetition of（重复）, 13
symbolic program execution（符号程序执行）, 363-364
transforming the flowgraph into SSA form（转换流图到SSA形式）, 363
value numbering vs.（值编号）, 343
sparse set representation（稀疏集合表示）, 762-763
speculative load, extension to LIR（前瞻取指令，扩充到LIR）

bbs.theithome.com

defined（定义），547
operator (← sp)（运算符(← sp)），547, 548
SPEC92 benchmarks（SPEC92基准测试程序），567
speculative loading（前瞻取），547-548
speculative scheduling（前瞻调度），548
performance and（性能），573
safe and unsafe（安全的和非安全的），548,730
unspeculation（反前瞻调度），548
speed（速度）. *See* performance（参见性能）
spill costs computation（溢出代价计算），487-488, 501-503
heuristics（启发式）. *See* register allocation by graph coloring（参见图着色寄存器分配）
spilling symbolic registers（溢出符号寄存器），488, 506-511
SPIM, 767-768
SpixTools, 770-771
splitting critical edges（分割关键边）. *See* edge splitting（参见边分割）
SPUR LISP compiler（SPUR LISP编译器），register allocator（寄存器分配器），525
Srivastava, Amitabh, 658, 664
SSA form（SSA形式），252-258. *See also* intermediate code（也参见中间代码）
described（描述），4
dominance frontiers（必经边界），253-254
du-chain in（du链），252
future trends（未来的趋势），744
loop code（循环代码），4, 5
in partial-redundancy elimination（部分冗余删除），415
in POWER and PowerPC compilers（POWER和PowerPC编译器中），721
pruned（修剪），258
standard translation example（标准转换的例子），253
strength reduction on（强度削弱），443
translating into（转换到），252-258, 349-350
in value numbering（值编号），349-355
stack frames（栈帧）. *See also* parameter-passing (run-time)（同时参见参数传递（运行时））
dynamic link（动态链），114
local symbol tables（局部符号表），49, 51, 52-54, 56-58
local variables in（局部变量），56-58
overview（概述），111-112
run-time support（运行时支持），111-114
stack-plus-hashing model（栈加散列模式），52-54
static link（静态链），114-116
structures for procedure calling（过程调用的结构），122, 124, 126
tail-call elimination and（尾调用删除），462-463
stack of local symbol tables（栈的局部符号表），49, 51
local variables in（局部变量），56-58
stack-plus-hashing model（栈加散列模式），52-54
stack pointer (sp)（栈指针(sp)），112-114
alloca( ) and, 113
described（描述），110, 112
frame pointer vs.（帧指针），112-113
tail-call elimination and（尾调用删除），462
stack, run-time（栈，运行时）. *See* run-time stack（参见运行栈）
stack storage class（栈存储类），44
stall cycles, filling（停顿时钟周期，填充），534-535
standards（标准）
ANSI/IEEE-754 1985, 106, 331
Application Binary Interface(ABI)（应用二进制接口(ABI)），105, 134
constant folding and（常数折叠），331
Unicode（统一码），106
statement separator (;), in ICAN（语句分隔符(;)，ICAN中），21,37
statements (ICAN)（语句（ICAN)），36-40
assignment statements（赋值语句），36-38
case statements（case语句），25, 39
described（描述），24-25
for statements（for语句），39-40
goto statements（goto语句），38
if statements（if语句），38-39
keywords in（关键字），25, 40
overview（概述），36
procedure call statements（过程调用语句），38
repeat statements（repeat语句），40
return statements（返回语句），38
syntax（语法），37
while statements（while语句），39
static branch-prediction methods（静态分支预测方法），598
static link（静态链），114-116
described（描述），111, 114
in procedure descriptors（过程描述字），126
procedure sorting（过程排序），672-673
setting（设置），114-115
up-level addressing（上层寻址），115-116
static single-assignment form（静态单赋值形式）. *See* SSA form（参见SSA形式）
static storage classes（静态存储类），44
static variables (C)（C的静态变量），44
procedure integration and（过程集成），469

bbs.theithome.com

static-semantic validity, described（静态语义的合法性，描述），2
Steele, Guy L., 525, 530
Steenkiste, Peter, 662
Steffen, Bernhard, 407, 443,597
storage binding（存储绑定），54-58
  addressing method（寻址方法），54-55
  described（描述），54
  large local data structures（大的局部数据结构），58
  symbolic registers（符号寄存器），55-56
  variable categories（变量类别），54
storage classes of symbol tables（符号表的存储类），43-45
  automatic or class（自动的或栈的），44
  file or module（文件或模块），44
  global（全局的），44-45
  lifetime of variables（变量的生命期），44
  scope（作用域），43
  static（静态的），44
  visibility of variables（变量的可见性），43-44
store operator (←), in machine grammars（存储运算符(←)，机器语法中），139
stores, generating in symbol tables（存数指令，在符号表中生成），59-63
Stoutchinin, A., 817
straightening（伸直化），583-585
  described（描述），579, 583
  ICAN code（ICAN代码），584-585
  overview（概述），583-584
strength reduction（强度削弱），435-443
  algorithm（算法），436-438
  MIR example（MIR例子），437-443,444
  overview（概述），435-436
  on SSA form（在SSA形式上），443
strict dominance（严格必经结点），182
stride（步长），670
strongly connected components（强连通分量），193-196
  algorithm（算法），194-195
  interprocedural data-flow analysis（过程间数据流分析），631-632
  maximal（最大的），194
  natural loops（自然循环），191-193
strongly distributive（强可分配的），236
structural control-flow analysis（结构控制流分析），202-214
  acyclic and cyclic control structures（无环和有环控制结构），202-204, 207-208
  advantages（优点），173
  algorithm（算法），205-206
  data structures（数据结构），205, 208-209, 211-212
  example（例），210-214
  global data structures（全局数据结构），205
  interval analysis compared to（与区间分析的比较），202
  preorder traversal（前序遍历），207
  region-reduction routine（区域归约例程），207, 209-210
  smallest improper region determination（最小非正常区域的确定），210, 211
structural data-flow analysis（数据流分析），236-249
  backward problems（向后问题），244-247
  described（描述），236
  flow functions for `if-then`（用于`if-then`的流函数），237-238
  flow functions for `if-then-else`（用于`if-then-else`的流函数），238
  flow functions for `while`（用于`while`的流函数），238-239
  forward problems（向前问题），236-244
  interval analysis compared to（与区间分析的比较），249
  reaching-definitions analysis（到达-定值分析），242-244
  representing equations（表示方程），247-249
structural hazards（结构停顿），271
structure of compilers（编译器的结构），1-3
  optimizing compilers（优化编译器），7-11
  placement of optimizations（优化的安排），11-14
subscript range operator (..), in ICAN（下标范围运算符(..)，ICAN中），26, 30
subsumption（归类）. *See* register coalescing（参见寄存器合并）
subtraction operator (−)（减法运算符(−)）
  in HIR, MIR, and LIR（HIR、MIR和LIR中），74, 75
  in ICAN（ICAN中），27, 29
successor basic blocks（后继基本块），175
  register allocation and（寄存器分配），484
SUIF, 769
Sun compilers（Sun编译器）. *See* SPARC compilers（见SPARC编译器）
Sun IR, 71, 72, 709-710
supergraph（超图），634
superscalar implementations（超标量实现）
  percolation scheduling for（渗透调度），574
  scheduling for（调度），543-545
  software pipelining and（软流水），548

bbs.theithome.com

trace scheduling for（踪迹调度），574
superscript operators, in XBNF notation（上标运算符，XBNF表示），19-20
support of a function（函数的支持），637
symbol names, procedure integration and（符号名，过程集成），469
symbol tables（符号表），43-65
compiler structure diagrams and（编译器结构图示），3
generating loads and stores（生成取和存指令），59-63
global symbol-table structure（全局符号表结构），49-54
local symbol-table management（局部符号表结构），47-49
run-time and（运行时），114
storage binding（存储绑定），54-58
storage classes（存储类），43-45
symbol attributes（符号属性），45-47
symbol-table entries（符号表项），45-47
variety in design（设计中的变化），4
symbol-table entries（符号表项），45-47
typical fields（典型字段），46
symbolic execution（符号执行）
for jump and return-jump functions（用于转移和返回转移函数），641
for sparse conditional constant propagation（用于稀有条件常数传播），372
symbolic languages, run-time support（符号语言，运行时支持），131-133
symbolic registers（符号寄存器）. *See also* webs（也参见网）
ICAN representation（ICAN表示），82
spilling（溢出），488,506-511
storage binding（存储绑定），55-56
syntactic analysis（语法分析）. *See also* parsing（也参见分析）
combining with lexical analysis（与词法分析结合），3
described（描述），2
in Fortran（Fortran中），3
syntactic blocks, eliminating in Graham-Glanville code-generator generator（句法阻滞，在Graham-Glanville代码生成器的产生器中的删除），154-158
syntax（语法）
abstract syntax tree（抽象语法树），70-71
XBNF syntax of changes to MIR to make HIR（为变化到HIR而对MIR的XBNF语法的改变），79
XBNF syntax of changes to MIR to make LIR（为变化到LIR而对MIR的XBNF语法的改变），80
XBNF syntax of MIR instructions（MIR指令的XBNF语法），74
XBNF syntax of MIR programs and program units（MIR程序和程序单位的XBNF语法），74
syntax (ICAN)（语法(ICAN)）
conventions（约定），21
declarations（说明），26-27
expressions（表示），27
generic simple constants（一般简单常数），28
statements（语句），37
type definitions（类型定义），25-26
whole programs（完整的程序），25
syntax-directed code generators（语法制导的代码生成器）. *See* Graham-Glanville code-generator generator（参见Graham-Glanville代码生成器的产生器）

T

tail call, defined（尾调用，定义），461
tail merging（尾融合）
described（描述），580
order of optimizations（优化的顺序），604
overview（概述），590-591
tail-call optimization（尾调用优化），461-465
addressing modes and（寻址方式），465
effect of（效果），461-463
identifying tail calls（识别尾调用），463
implementation（实现），463-465
order of optimizations（优化的顺序），477
overview（概述），476
stack frames and（栈帧），462-463
tail-recursion elimination（尾递归删除），461-463
effect of（效果），461
implementation（实现），463
importance of（重要性），320-321
order of optimizations（优化的顺序），477
overview（概述），476
tail-recursive call, defined（尾递归调用，定义），461
target-machine code, examples in this book（目标机代码，本书的例子），16
Tarjan, Robert E., 185, 631,738
temporaries, described（临时变量，描述），111
Tenenbaum, Aaron, 287
terminals, in XBNF notation（终结符，XBNF表示中），19
thunk（形实转换程序），118
tiling（铺砌）. *See* loop tiling（参见循环铺砌）
Tjiang, Steven S. K., 160, 259, 260, 265
TLB. *See* translation-lookaside buffer (TLB)（参见后备

bbs.theithome.com

版权所有 翻印必究

转换缓冲区(TLB))
TOBEY, back end of POWER and PowerPC compilers (TOBEY，POWER和PowerPC编译器后端)，719-723
tokens, in ICAN（单词，ICAN中），22
top, of lattices（顶，格的），223. *See also* lattices（也参见格）
Torczon, Linda, 469, 495, 637, 664, 762
Towle, Robert A., 289
trace, defined（踪迹，定义），569
trace scheduling（踪迹调度），569-570
  compensation code（补偿代码），569
  described（描述），532, 569
  example（例），569-570
  order of optimizations（优化的顺序），532, 574
  performance and（性能），570
translation-lookaside buffer (TLB)（后备转换缓冲区(TLB)）
  described（描述），670
  misses and stride values（缺失与步长值），670
tree edges（树边），178
tree-pattern-matching code generation（树模式匹配代码生成）. *See* dynamic programming（参见动态规划）
tree transformations, for simplification of addressing expressions（树转换，用于地址表达式简化），337-341
trees（树）
  balanced binary trees（平衡二叉树），48, 761, 763
  intermediate code form（中间代码形式），97-100
  Polish prefix form（前缀波兰形式），101
  quadruples vs.（四元式），98
  representation（表示），763-764
  translation from quadruples（从四元式转换），98-99
  translation to quadruples（转换到四元式），99-100
trends in compiler design（编译器设计的趋势），744
triples（三元式）
  intermediate code form（中间代码形式），96-97
  quadruples vs.（四元式），96
  translation to quadruples（转换到四元式），97
true dependence（真依赖）. *See* flow dependence（参见流依赖）
tuple types, in ICAN（元组类型，ICAN中）
  binary operators（二元运算符），33
  constant delimiters(<...>)（常数分界符(<...>)），27
  constants（常数），33
  element selection operator (@)（元素选择运算符(@)），24, 27, 33
  type constructor (×)（类型构造符），23, 26, 28, 33
tuples, in ICAN（元组，ICAN中）
  HIR instructions, representation as（HIR指令，表示为），87
  LIR instructions, representation as（LIR指令，表示为），91-92
  MIR instructions, representation as（MIR指令，表示为），82-84
twig, 160-165
two-address instructions in register allocation by graph coloring（图着色寄存器分配中的两地址指令），499, 529
type definitions（类型定义）(ICAN)
  arrays（数组），30
  compiler-specific types（编译专用的类型），24, 35
  constructed types（构造的类型），28
  described（描述），23, 28
  enumerated types（枚举类型），29-30
  floating-point numbers（浮点数），81
  functions（函数），34-35
  generic simple types（一般简单类型），23, 28-29
  integers（整数），81
  nil value（nil值），24, 35
  operator (=)（运算符(=)），25, 26, 35
  records（记录），33-34
  sequences（序列），32-33
  sets（集合），31-32
  size operator（集合运算符），36
  syntax（语法），25-26
  tuples（元组），33
  unions（联合），34
type determination, data-flow analysis for（类型判定，用于数据流分析），262-263

U

UCODE, design issues using（UCODE，设计问题中用到的），67-68
ud-chain（ud链），251
  determining dead code（确定死代码），593
Ullman, Jeffrey D., 227, 351
unary operators（一元运算符）
  algebraic simplifications（代数化简），333
  MIR, 75
unary operators (ICAN)（一元运算符(ICAN)）
  negation（取负），28
  set types（集合类型），24, 31
unconstrained greedy schedule（无约束的贪婪调度），555-556
Unicode, run-time support（统一码，运行时支持），106

bbs.theithome.com

unification（统一）. *See* code hoisting（见代码提升）
unified or combined cache（一致的或组合的高速缓存）, 670
uniform register machine（一致的寄存器机器）, 163
unify transformation（统一转换）, 571-572
unimodular loop transformations（幺模循环转换）, 690-693
  loop interchange（循环交换）, 691
  loop permutation（循环置换）, 691
  loop reversal（循环逆转）, 692
  loop skewing（循环倾斜）, 692-693
  overview（概述）, 690-691
unimodular matrixes（幺模矩阵）, 690-693
unions, in interprocedural side-effect analysis（联合，过程间副作用分析中）, 637
union types (C), aliases and（联合类型（C），别名）, 300
union types, in ICAN（联合类型，ICAN中）
  binary operators（二元运算符）, 34
  overview（概述）, 34
  type constructor (∪)（类型构造符）, 23, 26, 28, 34
universal quantifier (∀), in ICAN（全体量词(∀)，ICAN中）, 27, 28
unnecessary bounds-checking elimination（不必要边界检查删除）, 454-457
  implementation（实现）, 455-457
  importance of（重要性）, 454-455
  languages requiring（语言要求）, 454
  order of optimizations（优化的顺序）, 458
  overview（概述）, 458-459
unpacked records（非压缩的记录）, 108
unreachable code, defined（不可到达代码，定义）, 580
unreachable-code elimination（不可到达代码删除）, 580-582
  algorithm（算法）, 580-581
  dead-code elimination vs.（死代码删除）, 580
  described（描述）, 579, 580
  ICAN code（ICAN代码）, 581
  MIR procedure example（MIR过程的例子）, 581-582
  repetitions of（重复）, 579
unroll-and-compact software pipelining（展开－压实软流水）, 555-558
  algorithm to find repeating pattern（找出重复模式的算法）, 557-558
  overview（概述）, 555-557
  unconstrained greedy schedule（无约束的贪婪调度）, 555-556
unrolling. *See* loop unrolling（循环展开）
unrolling factor（展开因子）, 559
unspeculation（反前瞻调度）. *See* speculative scheduling（见前瞻调度）
unswitching（无开关化）, 580, 588-589
up-level addressing（上层寻址）, 115-116
upwards-exposed uses（向上暴露使用）
  analysis（分析）, 229-230
  in instruction scheduling（指令调度）, 542
user environment, described（描述）, 3

V

value graphs of procedures（过程的值图）, 349-351
value numbering（值编号）, 343-355
  applied to basic blocks（作用于基本块）, 344-348
  applied to extended basic blocks（作用于扩展基本块）, 344
  described（描述）, 343
  global（全局的）, 348-355
  optimizations with similar effects（具有类似效果的优化）, 343
  order of optimizations（优化的顺序）, 372
  overview（概述）, 371,372
  very aggression version in IBM POWER and PowerPC compilers（POWER和PowerPC编译器中非常激进的版本）, 722
variable declarations (ICAN)（变量说明(ICAN)）
  described（描述）, 81
  overview（概述）, 23
  syntax（语法）, 26-27
variable expansion（变量扩张）, 562-564
  accumulator variables（累加器变量）, 562-564
  algorithms（算法）, 563
  described（描述）, 532, 573
  induction variables（归纳变量）, 562-563
  order of optimizations（优化的顺序）, 574-575
  search variables（搜索变量）, 562-563
variables（变量）
  addressing method in symbol tables（符号表中的寻址方法）, 54-55
  congruence of（叠合）, 348-351
  dead（死去的）, 445,592-592
  induction variables（归纳变量）, 425-426
  induction-variable optimizations（归纳变量优化）, 425-453
  lifetimes of variables（变量的生命期）, 44
  live and dead（活跃的与死去的）, 443,445
  live variables analysis（活跃变量分析）, 229, 443-446
  local variables in stack frame（栈帧内的局部变量）, 56-58
  procedure-valued, interprocedural control-flow analysis

bbs.theithome.com

with（过程值的，过程间数据流分析），612-618
procedure-valued variables（过程值变量），126
register use by（寄存器使用），111
in shared objects（共享对象），128
storage-binding categories（存储绑定分类），54
up-level addressing（上层寻址），115-116
visibility of symbol-table variables（符号表变量的可见性），43-44
VAX compiler, ASCII support（VAX编译器，ASCII支持），106
very busy expressions（非常忙表达式），417
Vick, Christopher, 443
VLIW (very long instruction word systems)（VLIW（超长指令字系统）），548,562, 566, 570, 574, 575, 576
visibility, of variables（可见性，变量的），43-44
volatile storage class（易变存储类），45

W

Wall, David W., 658,659-662, 664
"wand waving", in POWER and PowerPC compilers（"wand waving"，POWER和PowerPC编译器中），722
Warren, S. K., 806
WARTS, 770
weakly separable array references（弱可分的数组引用），282, 283
webs（网）. *See also* symbolic registers（也参见符号寄存器）
data-flow analysis（数据流分析），251-252
defined（定义），486
example（例），489
interferences among（之间的冲突），490
priority-based graph coloring（基于优先级的图着色），524-525
register allocation by graph coloring（图着色寄存器分配），486, 489-494
spill costs computation（溢出代价计算），487-488,501-503
Wegman, Mark N., 348, 355,363
Weihl, William E., 612, 637
Weicker, Reinhold P., 121
Weinstock, Charles B., 820
well-behaved loops in C（C的规则循环），425
well-structured flowgraph（结构良好的流图），196
while statements（while语句）
flow functions for structural analysis（用于结构分析的流函数），238-239
ICAN form（ICAN形式），39
loop inversion（循环倒置），587-588
whitespace, in ICAN（空白符，ICAN中），21
Whitfield, Debbie, 700
window scheduling（窗口调度），551-555
algorithm（算法）. 552-553
combining with loop unrolling（与循环展开结合），553,555
overview（概述）. 551-552
Wolf, Michael E., 289, 690, 698, 738
Wolfe, Michael R., 694, 700
word, usage in this book（字，本书中的用法），16-17
worklist algorithm for iterative data-flow analysis（迭代数据流分析的工作表算法），232-233
write-live dependence（写活跃依赖），571
Wulf, William, 730

X

XBNF. *See* Extended Backus-Naur Form (XBNF)（扩展的巴科斯-诺尔范式）
XIL code（XIL代码），719-721
XL compilers（XIL编译器）. *See* POWER and PowerPC compilers（参见POWER和 PowerPC编译器）

Y

YIL code（YIL代码），721

Z

Zadeck, F Kenneth, 250, 348, 355, 363,525, 530
zero distance (=), in direction vectors（零距离(=)，方向向量中），278
zero or more copies operator, unary postfix (*), in XBNF notation（零次或多次复制运算符(*)，XBNF表示中），19,20
Ziv, Isaac, 807

bbs.theithome.com