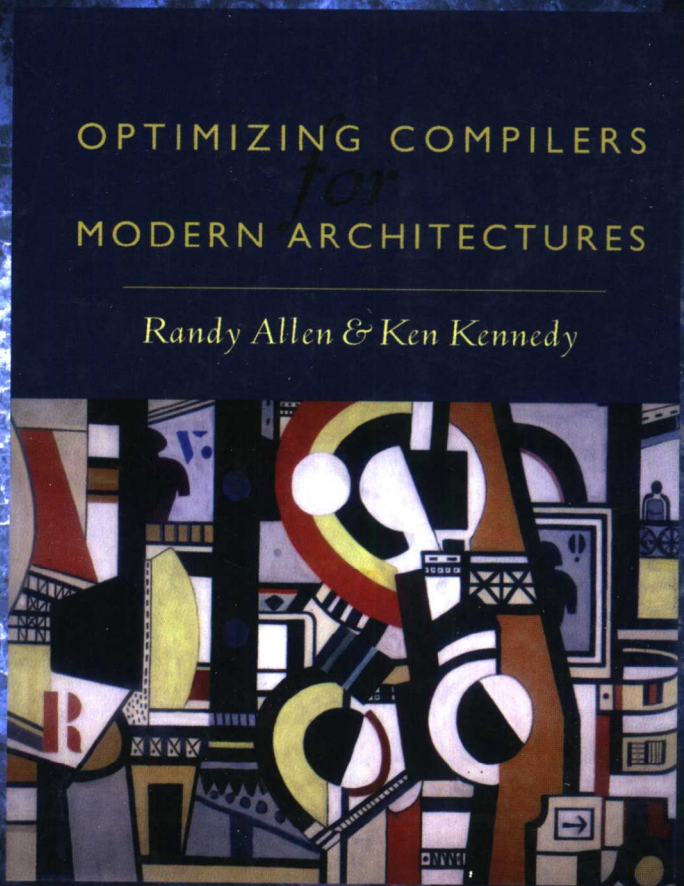
计 算 机 科 学 丛 书

现代体系结构的优化编译器

(美) Randy Allen Ken Kennedy 著
张兆庆 乔如良 冯晓兵
吴承勇 连瑞琦 刘 旸 译



Optimizing Compilers for Modern Architectures

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

设计具有高性能微处理器的现代计算机体系结构，能够极大地提高计算机在性能方面的潜在优势。然而其高度的复杂性使得产生有效代码和实现其全部优势变得愈加困难。这本出自两位学术权威的具有里程碑意义的教科书，重点阐述了编译器对于解决这个至关重要问题所起到的关键作用。

数据依赖是在高性能微处理器和并行体系结构上优化程序的基本编译器分析工具。它能使所编写的编译器自动地将简单的串行程序转换成具有现代体系结构特征的程序。数据依赖支持许多变换策略，也应用于一些重要的优化问题，本书对此做了全面介绍，并对基于数据依赖的编译器优化的重要性和广泛应用性进行了论证，给出了理解和实现它们所需要的基础，同时还为手工转换程序提供了详细说明。

书中介绍的方法是基于过去二十多年的研究成果，取材于在美国Rice大学的研究原型和几个有关的商业系统中实现的策略。致力于现代计算机体系结构设计和优化编译器的研究人员、业界专家和研究生都可以从本书中获益。

本书特点：

- 提供一种简单实用的算法和方法的指南，在高性能微处理器和并行系统中是最有效的
- 用处理过的例子示范每个变换
- 用实例分析编译器如何实现每一章中描述的理论和实践
- 介绍存储层次结构问题的最完善的处理方法
- 全书用依赖图来阐明排序关系
- 涉及各种语言，包括Fortran 77、C、硬件定义语言、Fortran 90和High Performance Fortran

封面设计 蓝子宇

ISBN 7-111-14122-9

9 787111 141228

HZ BOOKS

华章图书

网上购书：www.china-pub.com

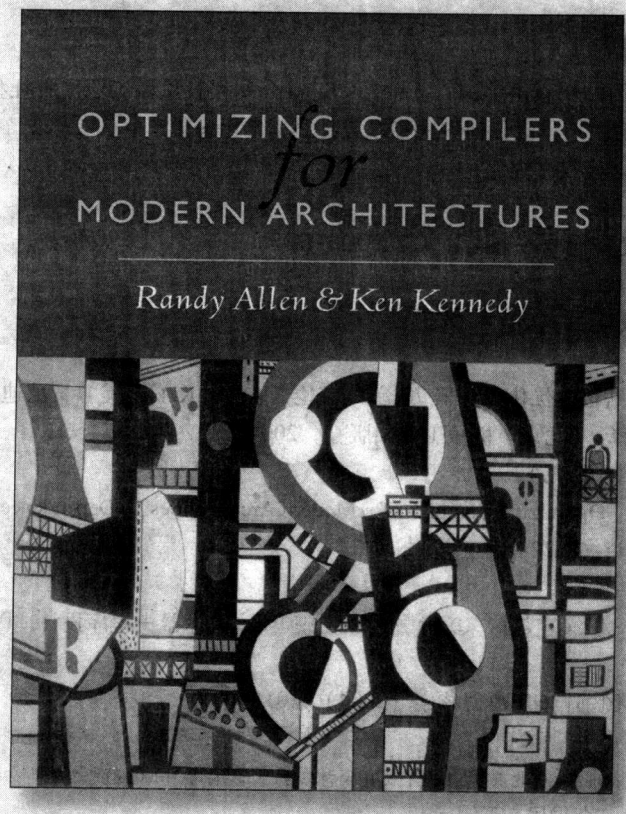
北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com

ISBN 7-111-14122-9/TP · 3499
定价：69.00 元

计 算 机 科 学 丛 书

现代体系结构的优化编译器

(美) Randy Allen Ken Kennedy 著
张兆庆 乔如良 冯晓兵
吴承勇 连瑞琦 刘 旸 译



Optimizing Compilers for Modern Architectures

A Dependence-Based Approach

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

本书介绍对现代体系结构的编译器进行优化的方法，理论基础是基于循环依赖的。分析基于依赖的变换的正确性论述和依赖测试的详细过程。剖析怎样扩展依赖去处理循环嵌套中的控制流以及跨越整个程序的过程。本书还讨论怎样能用依赖来回答现代计算机系统编译中的众多重要问题，包括支持不同类型体系结构（例如，向量、多处理器、超标量）的并行化，存储层次结构的编译器管理，带指令级并行性的机器的指令调度。最后，介绍一些不大为人熟知的应用，如硬件设计、数组语言实现以及消息传递系统的编译。

Randy Allen & Ken Kennedy: Optimizing Compilers for Modern Architectures, A Dependence-Based Apprach (ISBN 1-55860-286-0).

本书版权登记号：图字：01-2001-5271

图书在版编目（CIP）数据

现代体系结构的优化编译器 /（美）艾伦（Allen, R.），（美）肯尼迪（Kennedy, K.）著；张兆庆等译. －北京：机械工业出版社，2004.7

（计算机科学丛书）

书名原文：Optimizing Compilers for Modern Architectures, A Dependence-Based Approach

ISBN 7-111-14122-9

Ⅰ.现… Ⅱ.① 艾… ② 肯… ③ 张… Ⅲ.编译器－程序设计 Ⅳ.TP314

中国版本图书馆CIP数据核字（2004）第018060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：李伯民 王镇元

北京忠信诚胶印厂印刷 · 新华书店北京发行所发行

2004年7月第1版第1次印刷

787mm × 1092mm 1/16 · 37.5印张

印数：0 001 - 4 000册

定价：69.00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

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，还揭示了学术的源变，既遵循学术规范，又具有学者个性，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，又是挑战；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、翻译国外优秀教材上。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

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30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

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		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

序

Randy Allen和Ken Kennedy编写的“现代体系结构的优化编译器”一书是并行优化编译技术领域中一部权威性的著作。有幸由乔如良教授和张兆庆教授将全书译成中文，无疑对国内这一领域的教学、科研和工程的发展有着深远意义。

编译优化理论和技术是决定现代高性能计算机发展的十分关键的技术领域之一。优化编译技术的研究发展是和先进计算机结构——从微处理器到超级计算机——紧密相关联的。纵观优化编译技术最为超前的美国，各大学和科研机构中对这一领域的研究多是以电脑主流制造业的密切支持与合作研究为背景的。先进计算机体系结构和编译优化技术真可说是休戚与共、缺一不可。

本书的两位作者是长期从事优化编译技术研究的知名专家。特别是K. Kennedy教授更是学术界公认的泰斗。肯尼迪教授领导主持的一系列优化编译课题，既具有杰出的技术水平，又有着重大的学术影响。早在20世纪70年代末期，肯尼迪的研究工作对自动向量化编译技术的奠基和发展起过重大作用。80年代，肯尼迪和他在RICE大学的研究室对自动并行编译技术的贡献更为卓著。例如，他们在过程间优化的研究成果首创被商用编译器成功采用的先例。

基于他在学术上的成就和对计算机界的卓越贡献，肯尼迪教授先后被选为美国国家工程学院（National Academy of Engineering）院士，美国高科技协会（American Association for Advanced of Science）院士，IEEE院士和美国计算机学会（ACM）院士。肯尼迪教授在其三十多年的学术历程中，亲自指导了三十多名博士生。其中许多都是优化编译技术领域的精英，大多在知名大学任教或成为工业界的骨干。这本巨著中的许多内容正是作者和他的学生们多年经验的结晶。1997年，肯尼迪被美国总统任命为HPCC委员会两主席之一。

张、乔二教授是国内优化编译方面的资深专家和学术带头人。他们主持领导的中国科学院计算所先进编译技术试验室曾承担多项这一领域中重大长期研究项目，成果硕硕。近年来承担的Motorola和Intel等大公司委托的优化编译技术上的工程项目在国际上也颇具影响。由张、乔二教授和冯晓兵、吴承勇、连瑞琦、刘旸主持本书的翻译本出版是中国计算机学术界的一件大喜事，值得我们庆贺。

我有幸与Kennedy教授及张、乔、冯、吴等译者相识多年，这次能为这本书的中译本做序是我的荣幸。希望我的寥寥数言能有助于本书在国内计算机界广为推广。

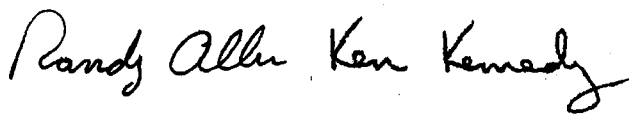
美国特拉华大学（University of Delaware）电子与计算机工程系终身教授
特拉华生物研究所生物信息中心主任
SUX excellence中心主任
CAPSL实验室主任和中国科学院计算技术研究所客座教授及联合实验室主任

Note To Chinese Readers

We are delighted that our book "Optimizing Compilers for Modern Architectures" will now be available to Chinese readers in their native language. This book was intended to serve several purposes. First we wanted it to be a good text for advanced students of compiler construction for high performance computers. Second, we wanted to produce a useful reference for professional compiler writers. Third, we wanted the material to be of value to scientists and engineers who develop application for modern high performance computers by providing an understanding of how to transform programs to achieve the very best performance on these machines.

We would like to thank the researchers at Institute of Computing Technology, Chinese Academy of Sciences who have done such a remarkable job of translation. Not only have they made this edition possible but, with their help, we have been able to correct many of the errors in the original English language edition. We sincerely hope that you, our Chinese readers, find the result of this effort both useful and interesting.

Randy Allen and Ken Kennedy



致中国读者

对于我们的著作《Optimizing Compilers for Modern Architectures》中文版的即将面世，我们感到由衷的高兴。本书可以满足不同读者的需要。首先，我们希望它能成为面向大学高年级学生的关于高性能计算机编译器构造的优秀的教科书。其次，我们希望它能作为专业编译器开发人员的有用的参考书。最后，对于那些为现代高性能计算机开发应用软件的科学家和工程师，我们希望本书的内容有助于他们了解应该如何变换程序从而能够在现代计算机上获得最佳的性能。

我们要感谢中国科学院计算技术研究所的同行们，他们不仅出色地完成了本书中文版的翻译工作，并且帮助我们更正了英文版中的许多错误。在此真诚地希望中国读者发现这是一本非常有价值的书。

Randy Allen和Ken Kennedy

译 者 序

Intel公司David Kuck院士在评论Ken Kennedy和Randy Allen的新书“Optimizing Compilers for Modern Architectures: A Dependence-Based Approach”时，这样写道：“编译程序是计算机科学与技术的皇后。编译器一直是应用与系统之间的桥梁。现在它们不仅决定应在新的硬件中实现哪些体系结构特色，而且说明对软件开发者而言，哪些新的语言特色将是有效的。”

Kuck院士和Kennedy教授都是并行优化编译理论与实践方面的最著名学者。Kuck将编译程序称为“计算机科学与技术的皇后”是十分恰当的。

自1958年ALGOL58问世后，20世纪60年代在程序设计语言理论、数据结构、算法设计与分析、可计算性理论等方面的研究，取得了令人惊叹的硕果，编译理论、技术和实验起了很大的推动作用。至今编译领域中仍有很多具有挑战性的问题，吸引广大计算科学与技术研究人员投身于此项事业。

现代计算机体系结构大都具有各种粒度的并行机制和分层存储结构，访存仍是当今各类计算机体系结构的瓶颈。优化编译器的主要任务可以说是：扬计算机结构之长—挖掘程序中的并行性，避其之短—挖掘程序的局部性（重用性），实施有效的寄存器分配和Cache管理。这正是Kennedy和Allen在书中讲述的主要内容。第5章（提高细粒度并行性），第6章（开发粗粒度并行性），第8章（改进寄存器的使用），第9章（管理Cache）和第10章（调度）是优化的主要手段。第2、3、4章是全书的理论基础和基本的程序变换方法。为使程序分析和优化更精确和有效，第7章和第11章是必须的，但它们增加了编译器的实现难度。最后三章可以看成是编译技术的特定应用。

Kuck院士在他的评论中接着写道：“作为该领域的创新者和研发者，作者依据丰富的经验写出了此书。该书对Cache管理、向量化、并行化等优化，做了非常好的综合。论题涉及到现代体系结构，这些题材确实可以应用于从台式计算机到世界上最快的超级计算机上。书中的例子是从Fortran中抽取出来的，但其理论可以应用到许多程序设计语言上。我认为该书将起到极好的教科书作用，并为软件开发者广泛地利用。”

正如Kuck院士所说，书中用大量的Fortran例子说明基本概念、理论、方法和优化的效果，使全书通俗易懂。每章的后面有一节实例研究，介绍作者在编译器的实践中的真实材料，以及有关策略有效性的实验论据。最后还有若干练习题。这些材料和习题无论是对学生，还是对学术界、工业界的软件开发者都大有益处。这里软件开发者不仅指编译器的设计与实现者，还包括应用软件开发者。他们可以从中学到如何写一个优化程序的许多概念、方法和规范，特别是应使用程序设计语言中那些有效的语言成分，摒弃那些使程序可读性、可维护性差和难以优化的语言成分。

我们希望此书的翻译出版，能对我国高校的软件教学与科研有所帮助。这是我们翻译此书的初衷。

本书的翻译组织工作由张兆庆研究员全面负责。作者介绍、前言、前四章和附录由张兆庆和乔如良研究员译校，第5、6章由刘旸博士翻译，第7、11、12章由连瑞琦副研究员翻译，

第8、10章由吴承勇副研究员翻译，第9、13、14章由冯晓兵副研究员翻译。译者自知水平有限，译文中难免有不妥之处，敬请读者批评指正。

感谢机械工业出版社组织翻译这部巨著，感谢李伯民老师仔细阅读译稿，提出许多宝贵意见，并对译稿做了大量编辑加工。

译 者

2003年11月25日

前　言

现代体系结构的编译器设计

影响人们对世界理解的主要方面之一是描述它的方法，人们往往借助于语言中的概念表述来传达他们的理解。在使用计算机语言时这更为明显。计算机科学家使用的语言影响着问题描述到最终实现的算法方法。LISP程序员解决问题的方法与IBM 370汇编程序员的方法就很不相同。

可见语言在应用设计中具有强烈影响，而高级语言具有内在的高效优点，语言研究的主要目标是开发人类更容易理解的计算机语言。尽管语言研究获得的成果使之更容易使用，但是在产品应用的开发中未被广泛采用。主要原因很简单：有效性——从这些语言写的代码产生的可执行程序对于产品使用太慢。而且，如果一个问题描述超出了计算机本机指令，将这些描述映射到指令集则变得更加困难。

如果我们曾经有过这样的实际语言，先进编译器技术将成为克服它们固有性能缺陷的基础工具。换句话说，编译器优化是使用高级语言的基本保证技术。数据依赖是基本的编译器分析工具，用来优化现代机器体系结构上的程序，因为它能使编译器对不同的代码段是否访问相同的或不同的存储单元进行推断。这样可以用它来确定一个给定的程序是否可以并行化，或者可以更加频繁地重用在寄存器和高速缓存中的数据。至今初级编译器课程中很少讲授依赖分析。这样，大多数大学生计算机科学主修课从不讲述。

本书的一个主要目标是集中在一册书里对数据依赖作广泛的介绍，它可以同样由从业人员和学生们用于学习关于数据依赖及其对重要优化问题的应用，如并行化和编译器的存储分层结构管理。虽然在20世纪60年代编译器早期研究人员就提出了数据依赖，大多数有说服力的依赖应用是在使用循环和数组的情况下。这些结构本质上是重复计算，而重复计算提供了最适宜的优化场所。扩充依赖到支持数组和循环是本书的关注点。

这本著作介绍的理论基础是基于循环依赖的。它包括基于依赖的变换的正确性论述和依赖测试的详细过程。说明怎样扩展依赖去处理循环嵌套中的控制流以及跨越整个程序的过程。本书也讨论怎样能用依赖来回答现代计算机系统编译中的众多重要问题，包括支持不同类型体系结构（例如，向量、多处理器、超标量）的并行化，存储层次结构的编译器管理，带指令级并行性的机器的指令调度。最后，介绍一些不大为人们熟知的应用，包括硬件设计、数组语言的实现以及消息传递系统的编译。

除了基本理论，本书还应用依赖处理真实编译器优化中的实际问题。通过介绍一些有效的实现算法，以及我们从结合研究和商用实现的经验中获得的认识，达到上述目的。只要有可能我们就用例子来传授这些认识，经常用图形化的依赖关系描述来传授对讨论中的问题的直观理解。本书介绍的大多数算法是基于我们工作中已经构造的实现。设计的这些算法已达到很高的精度，不需要再花运行时间就可以将它们放在编译器中实际使用。最后，每章包含一些材料，按照我们自己的经验，讨论我们介绍的和已经付诸实现的东西之间的关系，包括优点和缺点。

由于本书将理论与实践融合一体，既适合作为学校的教科书，又可以作为产业界从业人员的参考书。我们仅假定读者了解大学的初级编译器课程，以及具有Hennessy和Patterson的《Computer Organization and Design: The Hardware/Software Interface》水平的机器设计原理。过去几年，我们在Rice大学开设了一门一学期的研究生课程，内容涵盖本书的所有材料。然而我们相信这些材料也适宜包含在一门大学高年级课程中。本书是按指导风格写成的，使读者能获得足够的见解，容易去理解介绍的概念。另外，书中包含为了帮助学生更清楚地思考那些基本问题的讨论和设计的习题。

为了符合商用编译器作者的需要，书中包含对依赖测试和变换算法的详细介绍。这样做的主要理由是可以让实际编译器开发者避免过去20多年我们自己在实现中遇到过的陷阱。例如，第3章中的依赖测试的细节如果从首要的原理推导，那么往往很难精确地得到正确结果。因此在可能时，我们尽力介绍完整的算法。使用书中包含的材料，我们已帮助一些公司在商业产品中实现依赖测试、向量化和并行化。我们确信这些策略在实践中是有效的。

本书是在Rice大学一个为期20年的研究项目的产物，该项目为向量和并行计算机系统研发基础的编译技术。在1979年开始这个研究项目时，我们相信源-源翻译器对向量机的支持才是必要的，而不是基于依赖的编译器。理由是基于实验：依赖实现需要的编译时间对实用来说太长了，并且它们的输出通常可以通过简单的检查加以改善。我们认为在产品编译器中太长的编译时间需求是不能容忍的，但对只做一次的源-源翻译器来说是很好的。此外，可以让有经验的程序员调整源-源翻译器的输出，使之达到更佳的性能。我们研究的最令人满意的结果之一，在于发现我们最初的设想何等错误——依赖不仅对产品编译器是有效的，而且比起我们最初的猜想，它是更加强有力的工具和可用的理论。任何现代优化编译器已将依赖作为基本构件，并且优化的结果不仅应用于向量机（它们做得极为出色），而且也应用于标量机，甚至应用到硬件设计中。

因为机器设计的不断进步，人们对几年前看起来似乎过时的论题（例如向量化），现在重新发现了它们的重要性。向量化方法对改善VLIW和超标量机的性能还是有用的，办法是在最内层循环提供额外的指令级并行性。然而，由于更加重要的原因这本书是适时的。因为处理器和存储器性能之间的距离增加了，在现代体系结构上必须实施的多数优化与存储访问有关。依赖提供一个关于存储访问的合理框架，它将继续增加有用性。

虽然我们打算覆盖许多研究人员的重要工作，但是我们自然将注意力集中在过去二十几年里作者和同事们于Rice大学做过的工作上。总之，我们的目标是将我们从该领域的经历中获得的综合知识付印。因此不应该将本书看成是文献的详尽的综述，而是开发基于依赖的编译器的实用指南。为表达清楚，有时不得不牺牲一些技术细节。我们的总目标是给读者充分的直观知识，使其在编译技术这个引人入胜的领域中能有效地工作。

内容概述

本书由14章组成，它们的内容在下面各段中概述。材料的核心包含在前9章中；余下各章集中于基本材料的扩充和在不同的问题领域的应用。

第1章，*高性能体系结构对编译器的挑战*，是对现代计算机体系结构所做的综述和分类，并讨论对每一类体系结构提出的主要编译挑战。重要的主题包括流水线并行性，向量指令，异步处理器并行性，超标量指令和VLIW指令，以及存储层次结构管理。引入依赖概念作为保证并行性安全使用的机制。

第2章，*依赖：理论与实践*，讨论依赖的基本概念，以及它的若干性质，包括方向向量和距离向量。主要的目标是证明一些基本定理，它们确立维持依赖的变换的正确性，特别是在循环嵌套中。这一章以一个样例向量化算法结束，算法说明依赖的有用性，并用作后面讨论向量化和并行化的模型。

第3章，*依赖测试*，提供对引用偶之间测试依赖的系统介绍。包含对下标分类、单下标测试和耦合下标组中依赖测试的讨论。这部分材料强调在可能的地方彻底地消除一个依赖，并得到尽可能接近的依赖性质，如方向向量。

第4章，*初等变换*，覆盖精确依赖分析的基本变换，包括循环正规化，标量数据流分析，表达式传播和替换，归纳变量替换，以及标量重命名。

第5章，*提高细粒度并行性*，集中于需要支持向量指令和VLIW或超标量处理器的内循环并行性。这一章探讨对第2章中介绍的分层并行代码生成过程的变形和扩展。特别将重点放在循环交换、标量扩展、结点分裂，数组重命名和向量化过程中的循环倾斜。

第6章，*开发粗粒度并行性*，探讨对称多处理器的代码生成策略，在一致共享存储系统中利用异步并行性。这种机器的主要问题是寻找充足大并行性粒度，以补偿较高的任务启动和同步的代价。这个代价是必须交换一个并行循环到最外层合法位置，而不是向量化需要的最内层位置。这一章探讨循环交换的使用、数组私有化、循环对齐和代码复制、循环倾斜和各种索引集分裂策略，以达到在原始程序中寻找可用并行性的目的。

第7章，*处理控制流*，探讨在程序的分析和变换中由分支引入的复杂性。它深入地讨论两个重要的策略。第一个策略是if*转换*，这是一种通过将控制流分支下的所有语句转换成条件赋值语句来消除控制流的机制。为支持向量化引入的if转换，是重写条件语句为向量语句所需要的。第二个策略是使用*控制依赖*边，它们是从条件语句到执行时依赖于它们的那些语句的边。这一章证明如果数据依赖和控制依赖都得到遵从，那么程序的含义便得到维持。最后，该章说明控制依赖如何适应标准的变换算法，包括并行代码生成算法。

第8章，*改进寄存器的使用*，说明依赖（正确地讲扩展到包含输入依赖）是怎样的一种有效方法，用来探测处理器的寄存器中数据重用的可能性。这一章介绍*标量替换*，本质上它是将下标变量引用赋给一个寄存器。另外还论述改善各层重用的变换，包括展开和压紧、循环交换、循环对齐和循环合并。

第9章，*管理高速缓存*，越过简单的数据重用，探讨改善带有高速缓存存储层次结构机器性能的变换。虽然大多数改善寄存器重用的变换也能增强高速缓存的重用，但高速缓存存储器更加复杂，因为它们比典型的寄存器集合要大许多，并且是在若干个字的块中处理数据。这些不同为一些变换提供了机会，如分段循环嵌套使它在适合装在高速缓存的数据集上进行迭代，以及在内循环中引起跨距为1的访问的循环交换。这一章还探讨*软件预取*——使用显式指令，将数据块在需要用它们之前从存储器移到高速缓存中。

第10章，*调度*，论述几种基于依赖的方法，通过编译时指令布置来改善超标量机器和VLIW机器的性能。主题包括表调度、踪迹调度、软件流水线和向量操作链接。

第11章，*过程间分析和优化*，讨论扩展编译器分析（包括依赖分析）的策略到整个程序。这一章介绍分析几个重要的过程间问题的算法，包括副作用、别名和常数传播。然后说明如何将副作用分析扩展到数组子段上。其他主题包括过程间优化（如内联替换和过程克隆），以及过程间分析和优化的管理。

第12章，C语言和硬件设计中的依赖，应用本书中介绍的分析策略到两个新的问题领域。第一个领域是应用依赖测试和程序变换优化C程序。第二个领域是将依赖应用到硬件设计上，包括加速设计模拟的方法和从高层抽象规格说明综合低层硬件的方法。

第13章，编译数组赋值，其中的材料涵盖的方法为Fortran 90中数组赋值语句在标量机和具有定长向量寄存器机器上的实现。主要主题是将数组赋值语句转换成不需要无限制的临时存储的标量循环。

最后一章，编译高性能Fortran，应用依赖分析从高性能Fortran生成有效的消息传递代码，高性能Fortran是Fortran 90的变体，它包含说明如何在一可伸缩并行计算机的存储器中布局数组数据的设施。主题包括根据拥有者计算规则的计算划分和通信生成及优化。

这十四章包含的材料，可以用几种不同的方法组织成多个课程。作为一门高年级的大学生课程，第2章至第7章提供并行化中有关问题的合理的完整的论述。作为集中于单处理器优化的课程，可以跳过第5章中的某些材料（5.1节、5.2节和5.7～5.9节除外，这几节涵盖重要的变换），第6章（6.2节和6.3节除外），以利于包含第8章和第9章。另一方面，熟悉并行化的读者可能想了解其他的应用，包括存储层次结构管理，这方面的内容集中在第8章到第10章，以及第12章到第14章。如前所说，我们已成功地在Rice大学开设的一个学期的研究生课程中讲授本书中的所有材料。

我们力求使本书成为自含的，只假设读者熟悉编译器和机器设计的基础。这种期望必须包含第4章中关于数据流分析（4.4节）的某些材料，它们可能会包含在一门基础编译器课程中。我们感觉到在书中包含这些材料是重要的，所以你能在依赖和变换理论的上下文中看到它们。我们希望在使本书具有可读性和包含针对编译课程的高年级大学生和实际编译器编写者的主题上已获得成功。

每章的结构和特色

我们尽力使每章结构一致。每章包含一个引言，在开始论述详细的技术之前，提供主题的概述。引言通常用例子说明一章将要回答的问题。

技术材料按指导的风格介绍，包括用依赖图说明许多例子。使用变换“前和后”的代码序列描述各种变换。我们介绍程序变换的完整算法，使实践者根据这些材料开发一个编译器的实现变得简单明了。多数情况下，这些算法是在我们自己的研究和商用系统中已实现的直接变体。为有效地处理大程序，我们力求在这些实现中保证算法有可接受的渐进的复杂度。算法是用直观的类Algol表示法表示的，这种表示法对规格说明非常有用。然而，我们增加了某些特殊的构造，用于在元素集合上的迭代（例如，for each循环），这有助于我们回避说明集合实现的细节。

每章的末尾是一个小结，回顾这一章涵盖的要点。随后是实例研究，介绍我们在真实实现中的材料，以及有关策略有效性的实验论据。这里，你将可以找到有关产品编译器需要的设计折衷处理方案的材料。最后一节是历史评述与参考文献，回顾相关的文献，讨论一章中介绍的概念的由来和发展，具体参考文献的引文可以在本书的末尾找到。

在每一章的结尾是少量习题，这是针对书中内容为我们的研究生课程设计的。这些习题的范围从对材料的简单练习到实现作业和研究问题。

辅助的在线材料

与本书关联的网页（www.mkp.com/ocma/）包含勘误表。

致谢.

本书是在Rice大学20年研究的产物，而这项研究又依赖在其他学术机构多年做的相关研究工作。本书本身的发展过程已有10年并使用了6种不同的字处理系统。如此大量的工作必然要依靠两位作者以外的更多人的工作，我们要感谢许多人，他们以自己的工作和思想对本书做出了贡献。

列在这张名单前面的是Rice大学过去的博士生们，他们的研究确定了本书的大部分内容。学生们的论文材料直接包含在各章中，他们是Vasanth Bala、David Callahan、Steve Carr、Keith Cooper、Ervan Darnell、Chen Ding、Gina Goff、Mary Hall、Paul Havlak、Kathryn McKinley、Allan Porterfield、Carl Rosene、Jerry Roth、Linda Torczon、Ajay Sethi和Chau-Wen Tseng。特别要感谢贡献习题的人们：Chen Ding、Kathryn McKinley和Jerry Roth。同样重要的是众多的研究生、大学生以及研究和技术人员，他们对PFC，ParaScope，D系统和Ardent Titan编译器的开发做出了贡献。这些人中，我们要特别感谢协助领导了这些开发项目的技术人员：Vikram Adve、David Callahan、Keith Cooper、Mary Hall、Seema Hiranandani、Steve Johnson、Kathryn McKinley、John Mellor-Crummey和Linda Torczon。

另外，在开发这个材料的过程中与杰出的同事们密切地交流使我们获益匪浅。在开发PFC的过程中，IBM的Randy Scarborough花了大量的时间与我们一起工作，提供给我们作为一位有阅历的业界编译器开发人员的很多见解和观点。Ben Wegbreit在Ardent Titan编译器开发过程中给我们带来许多令人兴奋的讨论，Steve Wallach在他的Convex向量化编译器开发过程中与我们交流，导致对我们的方法的很多改进，包括我们对C的关注。

其他应当特别提到的是那些对优化和依赖的理论与实践基础做出贡献的研究人员。这些研究人员（我们不受约束地汲取了他们的工作成果）包括Leslie Lamport、David Kuck、Utpal Banerjee、Michael Wolfe、Ron Cytron、John Cocke、Jack Schwartz、Frances Allen、Susan Graham、Monica Lam以及其他很多人。

没有IBM、国家科学基金会、国防部高级研究计划局以及能源部（通过Los Alamos计算机科学研究所）的经费支持，就不能完成本书的研究工作。我们很幸运有一支强大的程序管理员队伍，他们不仅帮助指导研究，还提供有力的精神支持。这些管理员包括Horace Flatt、Fred Ris、Bill Harris、Kamal Abdali、Rick Adrion、Gil Weigand、Bob Lucas、Frederica Darema和John Reynders。

本书写作的一个重要部分是用于研究生程度的课程。我们非常感谢那些学生和指导老师，他们忍受由于最初版本中错误带来的麻烦，并对我们造成的许多错误提出反馈意见。由于他们提出的改进意见，使本书成为非常优秀的著作。

本书的出版得到了Morgan Kaufmann的编辑们令人惊叹的帮助。Emilia Thiuri在本书准备出版的过程中提供一贯的和有力的支持。Edward Wade耐心和持久地关注每个细节，对本书从内容到外观的卓越性是极为重要的。稿件编辑Ken DellaPenta完成了校正书中错误的出色工作，保证书的表述的一致性。其余制作人员——排版员Nancy Logan、校对员Jennifer McClain和索引编写员Ty Koontz——对最后的质量做了巨大贡献。我们还要对Rebecca Evans & Associates

公司的内部设计和Ross Carron Design公司的封面设计表示感谢，对Dartmouth Publishing出版社所绘制的精美插图表示感谢。

复审人员Tarek Abdelrahman、Benjamin Goldberg、Kathryn McKinley、Steven Carr、Allan Porterfield、Andrew Chien、Monica Enand、Rohit Chandra和Bill Appelbe对这册书的结构和内容的改进提出了许多有益的建议。深深地感谢我们的编辑Denise Penrose，没有他的耐心而又坚定的指导和积极的支持态度，不可能完成此书。

最后也是最重要的，我们要感谢我们的家庭，感谢他们的耐心和鼓励，以及他们带给我们的安定生活。我们的妻子Vicky Allen和Carol Quillen对我们难以置信地支持，即使这意味着我们必须长时间把注意力放在别处。当我们处于重大压力下时，她们保持在我们的周围。Jennifer和Caitlin的降生几次延缓写作进程，但是他们使生活更加愉快和满足。

作者简介

Randy Allen以优异成绩获得Harvard大学化学专业学士学位，在Rice（赖斯）大学获得数学科学硕士和博士学位。成为Rice大学研究员之后，Allen博士参加了业界编译器构造的实践活动。他经历了在Ardent Computers、Sun Microsystems、Chronologic Simulation、Synopsys和CynApps等公司的研究、高级开发以及管理工作。他本人以及和他人合作在各种学术会议和杂志上发表了15篇关于计算机优化、编译器重构和硬件模拟等方面的论文。他出任Supercomputing和Conference on Programming Language and Design Implementation等会议的程序委员会成员。目前，他是几家公司（包括IBM、Intermetrics、Microtes Research和Mentor Graphics）的优化编译器顾问，是Catalytic Compilers公司总裁和CEO。

Ken Kennedy是Rice大学计算工程的Ann and John Doerr教授和高性能软件研究中心（HiPerSoft）主任。他是电气和电子工程师学会（IEEE）、计算机协会（ACM）以及美国科学促进会协会（AAAS）的会士，自1990年起，他是美国工程院院士。从1997年到1999年，任美国总统信息技术顾问委员会（PITAC）副主席。由于他在拟定关于建立信息技术研究基金的PITAC报告中的领导作用，获得计算研究协会杰出贡献奖（1999）和RCI Seymour Cray HPCC Industry Recognition奖（1999）。Kennedy教授发表了150多篇学术论文，并指导34篇有关高性能计算机系统程序设计支撑软件的博士论文。他对高性能计算软件方面的贡献得到了公认，1995年他获得W.Wallace McDowell奖，这是IEEE Computer Society的最高研究奖项。1999年他被提名为ACM SIGPLAN程序设计语言成就奖的第三位接受者。

目　录

第1章

高性能体系结构对编译器的挑战

1.1 概述和目标

过去的二十年一直在激励人们从事高性能计算。当今台式个人计算机展示的计算能力相当于20世纪70年代后期最强的超级计算机。其间，由于在存储层次结构中使用并行机制和革新技术，超级计算机在实际应用中已超出持续的万亿次浮点运算，现在正将它们的目标定在10到100万亿次浮点运算上。高端超级计算系统可看成是用于软件研究和开发的实验室，因为在这样的系统上开发的革新技术最终找到了用于台式计算机系统的方法。

计算机速度的惊人改进产生了两个有影响的结果。首先，已经看到根据基础技术构造的机器按Moore定律预言的速度飞速地推进。图1-1描绘从1950到2000年最快的超级计算机的峰值性能。很好地遵循Moore定律的回归拟合表明，超级计算机性能每十年增长两个数量级。可是技术自身已显不足。画出的四个区域表明，计算机体系结构的差异——从标量到超标量、向量和并行——是保持性能不脱离轨道所必需的。显然，对超级计算的性能来说某种形式的并行性是基本要素。

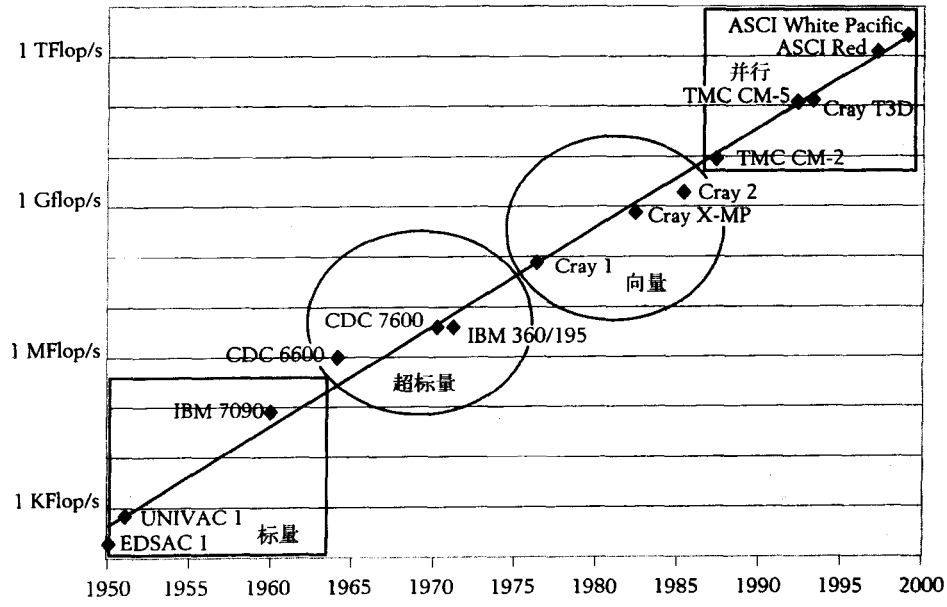


图1-1 过去50多年最快的超级计算机性能

但是，并行性不再是超级计算机独有的。即使现今的单处理器达到惊人的性能水准，它们仍然不能满足台式计算机的图像处理应用和多媒体的渴求。为满足这样的需要，提出了单处理器设计包含许多并行特性的要求，诸如多功能部件，多发射指令处理，以及附加的向量

部件。甚至这些策略也不足以满足计算服务器的需求，通常服务器使用适中数量（4到32个）处理器。显然，在市场上并行性已变成一个重大的因素。

高性能计算机不是惟一用并行性来区分的。由于快速增长的处理器速度和相对较慢的存储技术改进速度，今天多数处理器有二级或三级高速缓存存储器。设计这些高速缓存是用来隐藏处理器速度与存储访问时间失配所固有的长时间延迟。它们也改善有效的数据存储带宽，数据从高速缓存中得到重用。

然而计算能力方面的进展并非没有问题。当为维持Moore定律的高速度而使体系结构变得更复杂时，对编程来说就变得更困难了。多数高端应用程序开发者痛苦地意识到需要特殊的技巧，用来显式管理存储器层次结构和现今可扩展并行系统中的通信。为了尽力从各个处理器中汲取出更高的性能，程序员们已学会了如何手工变换他们的代码来改善多发射单处理器
1~2 上的指令调度。

我们的看法是，为高性能计算机所做的大多数手工变换事实上应当由编译器、库函数以及运行系统来实施。编译器必须起特殊的作用，因为它的传统责任就是将适合于人类应用程序开发人员使用的语言程序翻译成目标机器的本机语言。虽然正在说服人们相信编译器的责任在于结束时产生源程序的一个正确的机器语言翻译，但仅此而已是不够的。编译器必须产生一个适当有效的程序。如果做不到这一点，应用程序开发人员将抛弃使用这种语言的打算。

编译得到代码的效率的重要性不是新近的关注点。从Fortran第一个版本开始，它已包含在每一个成功的计算机语言的开发过程中。在John Backus的一段话中，反映了1978年对Fortran I所做出的努力[30]:

> 在Fortran开发的前几个月，我们的信念是，如果Fortran将任何合理的“科学计算”源程序翻译成目标程序，其速度仅为相应的手工代码的一半，那么我们的系统的可接受性会处于严重的危险境地。……直到现在我仍相信，把重点放在目标程序的效率上而不是语言的设计上，基本上是正确的。我相信如果不能产生有效的程序，那么一定会严重地推迟像Fortran这种语言的普及使用。
>
> 事实上，我相信今天的处境是类似的，但未被认识到：尽管所有争论是围绕大量语言细节产生的，当前传统语言仍然只是非常弱的程序设计工具，如果有人发现能力更强的语言并使语言以相当有效的方法运行，那么今天就会使用此语言。

四十年前在第一次开发出Fortran时说的这些话，今天仍是正确的。事实上，当机器变得更复杂时，编译器技术甚至变得更为重要。为计算机体系结构中每一项革新成功提供有效的语言实现，要视编译器技术能力而定；换句话说，将达到高性能的责任从硬件转向软件已成为现代机器体系结构的趋势。在此项任务中，编译器技术仅部分地获得成功。已为向量化，指令调度和多级存储层次结构的管理研究出了卓越的技术。另一方面，自动并行性仅对具有少量处理器的共享存储并行系统取得成功；对可扩展机器，编译器并行化仍是一个未解决的问题。

幸好今天针对高性能计算机体系结构可利用的多种多样组合，编译需要的基本分析结构已展示出有实用价值的共性。多数关键的编译任务能用变换加以处理，这些变换对原程序中
3 的语句实例重新排序。这些变换的安全性（即它们能否保持程序的含义）是由依赖概念决定的。已证明依赖是一个非常持久的和广泛可应用的概念。事实上，本书以《基于依赖的编译：理论和实践》作为书名可能是个好主意，因为依赖是整册书的统一主题。

本书的主要目标是介绍依赖及其支持的许多变换策略。这些编译器技术已在过去二十多年中开发出来，支持高端的机器复杂性。我们的目的是为先进编译技术的学生以及同样为实际的

编译器开发人员提供有用的资源。本章其余部分包含一个高性能计算机体系结构的简短介绍，以及处理它们所需的编译器策略。用一个扩展的例子说明这些策略的折衷，其中用相同的程序说明为不同机器组织所做的优化。最后，我们介绍依赖的概念，它将构成本书大部分的基础。

1.2 流水线

在计算机体系结构中，并行性的最早应用之一是使用流水线——将一个复杂的操作分成一系列独立的段，如果不同的段使用不同的资源，一旦它的前一个操作完成了第一段，立即启动一个操作，这样就能重复执行其他的操作。

1.2.1 流水线指令部件

考虑作为最早流水线之一的一个例子：指令执行流水线。IBM 7094是20世纪60年代中期的高端计算机，它的每条指令分两个阶段执行：取指阶段和执行阶段。因为可以安排这两个阶段从不同的数据体中取指令，因此下一条指令取指往往可以同当前这条指令的执行阶段重迭。根据70年代IBM 370系列机采用4级流水线的执行阶段，进一步提炼了指令执行流水线的概念。

过去的几年，指令流水线已变得更加复杂。图1-2给出DLX机器的指令流水线，Hennessy和Patterson以它为例说明精简指令集计算机（RISC）体系结构的原理[145]。DLX类似于许多现代的RISC机器，包括由SGI制造的机器中可以找到的MIPS体系结构。

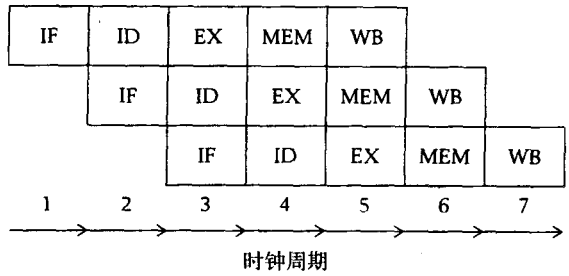


图1-2 DLX指令流水线[145]

在现代RISC机器中，典型的指令以三种形式之一出现：

- 寄存器到寄存器ALU操作：包括所有的算术操作，如整数和浮点数加、减等。
- 存储器操作：这些指令需要存储访问。从存储器取进寄存器和从寄存器存入存储器是典型的例子。
- 分支指令：依据一个条件的值这些指令改变下一条被执行指令在存储器中的位置。如果条件为真，程序计数器增加指令中的立即值；否则PC（程序计数器）继续指向存储器中的下一条指令。

设计的DLX流水线用来处理这三类指令。流水包括下面的这些级：

（1）取指令（IF）
（2）指令译码（ID）
（3）执行（EX）
（4）存储访问（MEM）
（5）回写（WB）

取指和译码是自明的。为执行的目的，将EX 和MEM级组织在一起。如果指令是一条寄存器到寄存器操作，它能在算术逻辑部件(ALU)中实施，则这一条指令在EX级完成。如果它是一条存储访问指令，则地址计算在EX级实现，而存储访问实际上是在MEM级发生。注意，如果需要的存储单元不在高速缓存中，出现不命中，那么这条指令将停顿（stall）直到将需要的块(block)装入高速缓存中。如果指令是分支指令，那么在执行阶段用一个指定寄存器与0比较，并在MEM级将PC置成正确的值。WB级用来将数据写回到寄存器中；在分支指令中不使用它。

如果每个流水线级使用不同的资源，则相继指令的不同级能够重迭。图1-2说明一种能在每个机器周期[⊖]上发射一条指令的调度。换句话说，在这种理想调度下，不计启动流水线需
4~5 要的时间，每条指令的平均周期数将是1。然而这种理想的调度不是总能实现的，因为可能受到很多状态（称为相关）的干扰。这些将在1.2.4节中讨论。

1.2.2 流水线执行部件

遗憾的是许多操作需要花费比这些短周期指令更长的时间。最显著的是浮点操作，它们可能要花4个或更多的周期才能完成。例如，考虑在一个典型的浮点加法器中的步骤。一旦两个运算对象已从存储器中取来，它们必须被规格化，使它们有相同的阶。下一步，必须将两个尾数加在一起。最后，在将相加的结果存入目的地之前，得到的相加结果可能不得不重新规格化。图1-3说明这一过程。

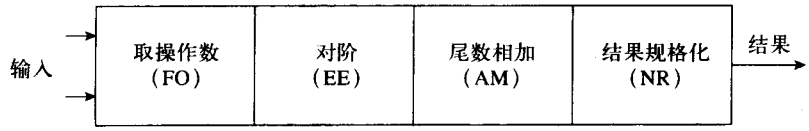


图1-3 典型的浮点加法器

由于加法部件中每一段独立于其他段，没有理由不让每一段同时对不同的操作数进行运算。因此，如果需要对几对数求和，通过加法部件各个阶段的重迭执行，能加快计算。如图1-4中的说明。

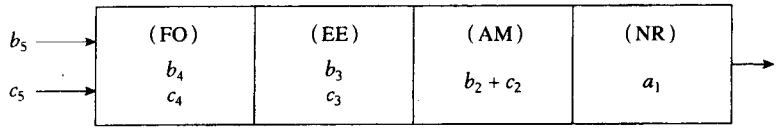


图1-4 计算$a_i = b_i + c_i$的流水线执行部件的瞬像

如果每一段在一对操作数上工作需要的时间是一个周期，那么在没有流水线的情况下实施n个加法操作需要的时间是$4n$个周期。如果各段重迭，使得计算是流水线的，那么n个加法需要$n+3$个周期。因为一旦流水线被充满，加法器每个周期产生一个结果。因此，对极大量的计算，用流水线执行一条加法指令的时间有效地从4个周期降至1个周期。

一个流水线功能部件是有效的，仅当流水线保持充满；即仅当在每个段时钟周期上的操作有可使用的运算对象。很遗憾，用户的计算很少满足此需求。其结果，有效地利用一个流
6 水线功能部件需要（通常指编译器）重排用户计算的顺序，使得递送必需要的运算对象足够地快，以保持流水线充满。

⊖ 以后文中出现的周期均指机器周期或时钟周期。——译者注

1.2.3 并行功能部件

如果复制了多个功能部件，那么如图1-5所描绘的每个部件就能对一条无关指令工作。当执行部件发射一条指令时，它将这条指令送到一个空闲的功能部件上，如果存在这样的部件的话。否则，它等待一个功能部件变为空闲的。如果有n个功能部件，每个以m个周期完成它的操作，则机器平均每个周期能发射n/m个操作。这类并行性（也称为*细粒度并行性*），乍一看可能比流水线显得更有吸引力，因为它允许操作的自由度更宽。然而，这里关联到代价问题。首先，一个流水线部件的代价比非流水线部件稍高一些[⊖]，而多功能部件的代价显然比单功能部件多若干倍。第二，功能部件比起流水线并行性需要更加复杂的执行逻辑。当然，可复制流水线的功能部件，因此，也可同时使用流水线和多部件并行性。

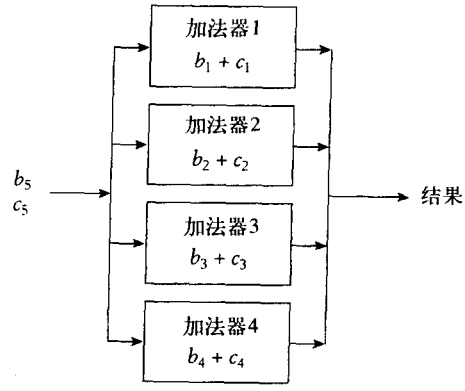


图1-5 多功能部件

7

1.2.4 标量流水线编译

在流水线的体系结构中，关键的性能屏障是存在*流水线停顿*（pipeline stall）。当一组新的输入不能注入流水线时，就发生停顿。这是由于一种称为*相关*（hazard）的状态造成的。Hennessy和Patterson[145]将相关分为三类：

（1）*结构相关*，它的发生是由于机器资源不支持有可能发生的指令覆盖组合。

（2）*数据相关*，它是发生在一条指令产生的结果是一条后续指令需要的时候。

（3）*控制相关*，它的发生是由于分支处理。

对于本章前面讨论的各种流水线，我们将用例子说明这三类相关。

在任何时候，一种特定体系结构的实现没有足够的资源支持某些种类的覆盖就会发生结构相关。例如，如果一台机器仅有一个到存储器的端口，它就不能覆盖带取数据的取指。这种限制基本上会使IBM 7094串行化，降低它的处理器性能。在一台仅有一个存储端口的DLX上，在每条load指令之后的第3条指令上会发生停顿，因为数据和指令的读取会发生冲突（见图1-6）。因此DLX的编译器策略不可能避免这类结构相关。

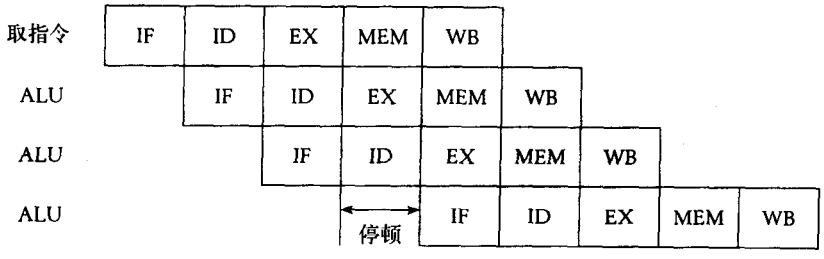


图1-6 具有一个存储器端口的DLX的结构相关

⊖ Seymour Cray在20世纪60年代设计Control Data 6600（大约1965年，它使用了多个部件）和Control Data 7600（大约1969年，它使用了流水线）之间发现了这一点。

数据相关会发生在像DLX这样的多级流水线上。在现代机器上，利用从ALU向ALU的下一流水线级传递结果从而避免大多数停顿。因此像

```
ADD     R1, R2, R3
SUB     R4, R1, R5
```

这样的指令序列能没有延迟地被执行，因为加法指令的结果是直接被送到减法指令的执行级，而不用等待回写到寄存器中。然而，这在

```
LW      R1, 0(R2)
ADD     R3, R1, R4
```

序列是不可能的，因为在存储周期结束前，取数指令的结果是不可能得到的。因此，我们看到如图1-7所描绘的一个周期的停顿。

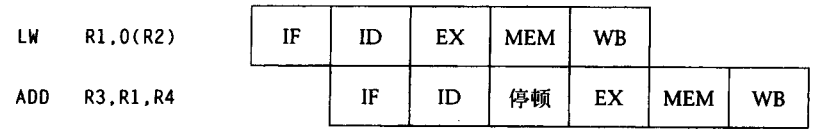


图1-7 由于存储时延导致DLX数据相关需要一个停顿

在上面load和add指令之间插入一条这样的指令，它不使用那个正在取数的寄存器，编译器调度就能消除此问题。

流水线功能部件向编译器调度提出一个类似的挑战，因为一条指令可能需要等待一个由前一条指令计算的输入，这就是执行部件流水线中的一个停顿（still）。在DLX上，多周期操作占有几个执行级，取决于执行操作需要的流水线级数。假设我们希望在一台机器上执行Fortran表达式

```
A + B + C
```

其中浮点加的流水线需要两级。如果按从左到右的顺序对表达式求值，那么第二条加法指令在执行前将不得不等待一个周期，如图1-8所示。

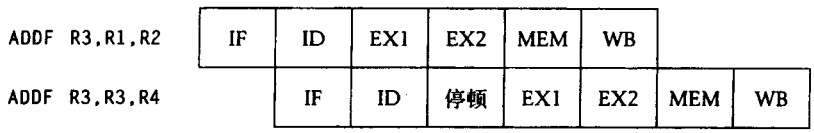


图1-8 由于指令时延导致DLX数据相关需要一个停顿

编译器调度再一次能帮助避免这些停顿。例如，假设求值的表达式是

8~9

```
A + B + C + D
```

在第一个加操作之后，每个加法将不得不等两个周期，等前一个加法做完，这需要两个停顿。另一方面，如果编译器能重组这些加法，如像

```
(A + B) + (C + D)
```

在第一个加法操作之后一个周期，能进行第二个加法操作一个周期。在第三个加法操作中将仍有一个停顿，但是停顿的总数已减少了一个。

控制相关是由分支指令引起的。在DLX上，一个控制相关能导致3个周期的停顿，如图1-9所示。假设不取分支，处理器开始取分支之后的指令。如果此假设证明是正确的，那么

流水线将进行而不用中断。但是，在分支指令的MEM阶段之前是否取分支是不知道的。如果取它，那么必须在新位置上重新开始处理这条指令，并且在分支之后和向存储器存入或回写到寄存器之前，必须将该条指令产生的中间结果抛弃掉。因此当取分支时，我们得到一个实在的3周期停顿。

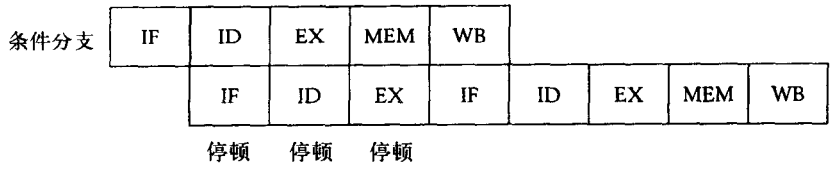


图1-9 使用自然实现取分支的DLX控制相关

因为3周期停顿是极为不利的后果，机器设计者花费很长时间去降低这种代价。一种方法是增加硬件使得能在指令译码（ID）级确定条件的输出和分支的目标。如果条件是简单的，比如测试等于0，只要有一个附加的ALU就能做到这两件事。因此，对这些简单的分支，执行分支的流水线停顿能减少至一个周期，如图1-10所示。注意，如果不取分支，那么就没有停顿。

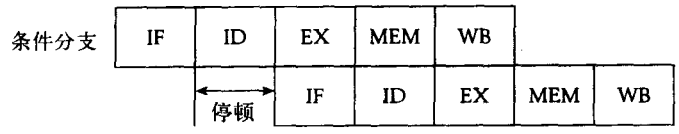


图1-10 通过及早识别分支目标减少流水线停顿

用这种方法实现的DLX，通过程序设计或编译器优化无法在被取的分支上避免一个周期的停顿。然而，某些RISC体系结构提供一条“ branch-and-execute”指令，在执行分支之前，总能执行指令序列中的下一条指令。这就允许编译器去重新安排指令，使得循环体中的最后一个操作（通常是一数据存储）在“此分支下”执行。 10

正如我们通过这些例子已看到的，克服由于相关引起的性能问题，编译器的主要策略是重新安排指令，使得停顿——特别是那些由于数据相关引起的停顿——决不发生。这种策略称为指令调度，将推迟到1.4.2节讨论，它涉及超标量和VLIW体系结构。

1.3 向量指令

在20世纪70年代用一个标准指令流使流水线保持充满的任务变得很繁重。高端超级计算机过去一直使用硬件策略在指令流中前视可以被启动的操作而不用等待另一个操作完成。这些策略使得读取指令和发射逻辑极端地复杂。

机器设计者努力去简化指令处理，在70年代中期转向一种替代的方法，在其中某些指令（称为向量指令）自身能充满浮点流水线。这一节讨论向量指令及使用中涉及的发射问题。

1.3.1 向量硬件概述

一条典型的向量指令启动存储器中或定长向量寄存器中两个向量按元素的操作，能用单个操作从存储器中取向量。1975年发布的Cray 1有7个向量寄存器，每个寄存器有64个元素。每个向量操作（包括取向量）在启动之后每个周期能产生一个结果，启动延迟的时间等于流水线的长度。

为了避免对连接的向量指令过多的启动延迟，许多处理器支持向量操作的链接。由于要

等待前一个向量操作计算的结果，所以一个向量操作的链接在第一个结果可以使用时立即着手启动。因此指令序列

```
VLOAD       VR1, M
VADD        VR3, VR2, VR1
```

在延迟时间等于向量取和向量加的启动延迟时间之和时，随后就会开始递交结果给VR3。如果不
11 用链接，在启动向量加之前需要等待，一直等到向量取完成了才能启动这条向量加指令。

向量指令极大地简化了发射足够的操作使流水线保持充满的过程。然而包括这些指令在内有若干缺点。首先，它们需要为向量寄存器大量地增加处理器状态，这就增加了处理器的耗费并使现场切换更复杂。第二，它们扩大了机器的指令总数，因此要做大量的、复杂的指令译码。最后，向量指令使存储层次结构设计复杂化，因为它们妨碍高速缓存的操作。为了避免过多的高速缓存回收问题，以及保持带宽高度，多数向量机在向量存取上绕过高速缓存。这又引入了向量操作之后维护存储器与高速缓存之间的一致性问题。为回避此问题，Cray Research公司完全不用高速缓存，代之使用一组显式管理的标量暂存寄存器。

1.3.2 向量流水线编译

虽然向量指令简化填满指令流水线的任务，但是给编译器和程序员带来了新问题。其中最重要的是要保证向量指令正确地实现用来编码的循环。下面的向量操作序列将取自A和B的两个有64个元素的向量[⊖]相加并将结果存入C中。

```
VLOAD       V1, A
VLOAD       V2, B
VADD        V3, V1, V2
VSTORE      C, V3
```

注意，在寄存器上发生load-store冲突时，多数向量操作会互锁，但是对存储器不会（不像单处理器，如DLX）。

为了说明向量机的主要编译器问题，我们介绍Fortran 90向量表示法。在Fortran 90中向量操作的语义反映多数向量指令集所具有的语义。确切地讲，它们为向量机提供一种较容易的表示编译器问题的方法。前面的向量加操作用Fortran 90将写成

```
C(1:64)= A(1:64) + B(1:64)
```

在Fortran 90标准语义下，数组赋值语句必须实现的行为犹如在存入任何结果元素前，先
12 从存储器中取出赋值语句右端的每个输入。换言之，凡提到右端的所有输入时，均指此语句执行之前的值。

虽然Fortran 90支持显式向量操作，但是由于历史原因，多数程序是用Fortran 90的方言Fortran 77编写的，因此用循环说明可以由向量硬件加速执行的操作。从这些程序中抽取向量并行操作是对编译器的挑战，这些程序将用类似下面形式的简单循环编写：

```
DO I = 1, 64
    C(I) = A(I)*B(I)
ENDDO
```

在编译器能调度此向量硬件的循环之前，它必须确定此循环是否在语义上等价于上述Fortran 90的数组赋值。如果等价，那么循环内的赋值语句能向量化，将循环归纳变量的引用

⊖ 贯穿全书，在没有说明特定的硬件向量长度时，假设长度为64，它是最常见的向量长度。

直接翻译成相应的三元组表示法。然而，问题并不那么简单，因为循环在开始执行下一次迭代之前，应执行本次迭代的所有的取数和存数，而数组语句是在任何存数之前执行所有的取数。在上面这种情况下，两种执行顺序产生相同的最后结果，所以含义精确相同——因此，此循环称为可向量化的。然而稍有差别的情况说明潜在的问题:

```
DO I = 1, 64
   A(I + 1) = A(I) + B(I)
ENDDO
```

这里，Fortran 77循环的每一次迭代使用前一次迭代产生的结果，不同于直译的Fortran 90数组语句:

```
A(2:65) = A(1:64) + B(1:64)
```

在此Fortran 90数组语句中，右端所有的输入引用旧值。因此第二个Fortran 77循环是不可向量化的。区分出这两种情况是向量化的基本问题，并推动数据依赖理论的发展。

倘若向量机的主要编译器问题是暴露向量操作的话，那么使用包含显式数组操作的语言（如Fortran 90）也许能解决向量化问题。不幸显式数组操作有一组类似的编译问题，正如我们将在第13章见到的（或者也是幸运，如果你使自己像编译器编写者一样生活！）。 13

1.4 超标量处理器和VLIW处理器

向量操作的主要缺点是复杂的指令集设计。除了完备的标量指令——在最现代的机器上有几百种——向量处理器必须支持一个相称的巨大的向量指令集，不仅包括执行计算的指令，还要包括创建向量操作的指令，以及在位向量掩码下操作的条件指令。

这种复杂性是能够避免的。如果我们可以在每个周期发射一条或多条流水线的指令，那么可能充满执行部件流水线，并以与向量处理器可比的速度产生结果。这就是在超标量指令字和超长指令字(VLIW)体系结构包含的基本思想。

1.4.1 多发射指令部件

在超标量和VLIW模式中，假设所有的输入已就绪，那么设计的处理器就要尽可能快地发射指令。典型的这类机器有能力在每个周期发射多条指令，直至由硬件确定的某个上限。

超标量机器通过硬件实现多发射，在操作的指令流中前视那些执行就绪的操作。因此，一个超标量处理器能连续发射指令，只要遇到的每条指令是“就绪”的。某些机器甚至有能力乱序发射指令。

另一方面，VLIW处理器每个周期用执行单条“宽指令”的办法发射多条指令。一条宽指令装有几条常规的指令，它们在同一时刻被发射出去。典型情况下，这些指令的每一条对应于不同的功能部件上的一个操作。因此，如果一台VLIW机器有两个流水线的浮点乘法部件，每个周期它能发射两个浮点乘法。在一个VLIW系统上，期待程序员或编译器去管理执行调度和正确地组装宽指令字——使得在它的所有输入就绪之前没有指令被发射出去。因此，在这类机器上，不需要特殊的前视硬件。

虽然超标量和VLIW系统结构能达到向量执行的速度，但它们有若干缺点。首先，它们需要更高的从存储器读取指令的带宽，必须要有足够大的指令高速缓存能容纳一个典型循环中的所有指令。另外，典型的数据读取如同简单处理器那样使用相同的存储层次结构，所有的运算对象要通过高速缓存传递，当高速缓存尺寸受到限制而运算对象仅使用一次时，这就引

起问题。这是多数向量机上设计的向量取数绕过标量高速缓存的理由。

通过高速缓存传递值引起的另一个问题是，跨距为1的数据访问对良好的性能变得更为要
14 紧。如果循环中访问方式不是连续的，对不是马上使用的运算对象来说，浪费了存储器和高速缓存之间多数可利用的带宽。倘若对这样的机器来说带宽限制是主要问题的话，那么这是一个严重的问题。

理论上讲，可以把超标量机器和VLIW机器想像成向量处理器，因为它们能有效地利用由向量化暴露的并行性。确实，许多现代的微处理器（如像在苹果公司Macintosh中使用的PowerPC G4），包含的“向量部件”实际上是VLIW协处理器。

1.4.2 多发射处理器的编译

为了充分达到它们的潜能，超标量机器和VLIW机器需要小心地规划操作，让机器资源极尽所能地得到使用。因为大多数应用程序开发者不是用机器语言写程序，实现规划的过程是编译器的任务。这涉及到两方面的挑战：

（1）编译器必须识别何时操作没有依赖关系。无依赖操作可以按相对于其他操作的任何顺序执行；有依赖的操作则不能。

（2）编译器必须调度计算中的指令，使其需要的周期数尽可能地少。

向量化能应付第一个挑战，因为向量化暴露许多能并行执行的操作。另一方面，第二个挑战需要一种编译器策略，称为指令调度，这是第10章的主题。按其最简单的方式来说，指令调度是在处理器资源和程序依赖的限制内尽可能早地执行指令。

一个经常被重复的神话是，现代超标量处理器不需要调度，因为它们使用积极的强有力的前视策略。确实，超标量体系结构比起VLIW处理器来说，调度的必要性较少。然而，所有的超标量系统前视指令流的窗口都受到尺寸的限制，并且当硬件并行性数量增加时，并行操作的搜索将需要拓宽范围，很可能超出指令流前视的限制。编译器通过重新安排指令流，对保证尽可能多的并行操作适应此前视窗口会有帮助。

在这一节，我们集中于VLIW处理器，因为它们必须显式地调度，从而使发射透明清晰。另外，一个好的VLIW处理器调度也为有相同资源的超标量处理器提供一种好的调度——列出从第一个周期到最后一个周期的每个周期中一个超标量程序应产生的指令，该程序至少与生
15 成调度好的VLIW程序一样工作。

为调度直线代码，编译器必须了解哪些指令依赖于另外一些指令，对于每个依赖关系，在第1条和第2条指令之间需要多长的延迟。为了说明这一点，考虑为一台每个周期能发射一条指令的机器调度下面的指令序列：

```
LF      R1, A
LF      R2, B
ADDF    R3, R1, R2
STF     X, R3
LF      R4, C
ADDF    R5, R3, R4
STF     Y, R5
```

如果从高速缓存取数有2周期延迟——即不能发射任何使用load结果的指令，一直要等到发射load指令两个周期之后——并且浮点加指令也有2周期延迟，通过将取C的load指令向前移动，将存X的store指令向后移动，则能改善上面序列的调度：

```
LF      R1, A
LF      R2, B
LF      R4, C
ADDF    R3, R1, R2
ADDF    R5, R3, R4
STF     X, R3
STF     Y, R5
```

第一个调度发射所有的指令需要用11个周期，因为在指令间必须插入4个1周期的延迟——每条加法指令和每条store指令前面各一个周期。另一方面，第二个调度发射所有的指令仅用8个周期，因为仅在两条加法指令之间需要延迟。

在每个周期能发射多于一条指令的机器上，不能改善上面的代码段，因为指令之间有过多依赖。然而，考虑下面的序列，它执行两个无依赖的加法操作:

```
LF      R1, A
LF      R2, B
ADDF    R3, R1, R2
STF     X, R3
LF      R4, C
LF      R5, D
ADDF    R6, R4, R5
STF     Y, R6
```

16

在一台每个周期能发射两条load指令和两条add指令的VLIW机器上，我们完全能用第一个表达式计算覆盖第二个表达式计算。另一方面，如果机器每个周期能发射两条load指令，但只能发射一条add指令，那么我们需要一个额外的周期，如下面的调度所示:

LF R1, A	LF R4, C
LF R2, B	LF R5, D
Delay	Delay
ADDF R3, R1, R2	Delay
STF X, R3	ADDF R6, R4, R5
empty	STF Y, R6

在循环中，调度变得更复杂，这里的目标是构造一个长度最短的等价循环，办法是用原循环的不同迭代覆盖计算。这类调度也称为“软流水线”，将在第10章讨论。

1.5 处理器并行性

虽然流水线是加速单处理器或功能部件执行的有效方法，但处理器并行性同时使用多处理器完成不同的任务，或在不同的数据集上完成相同的任务，从而减少应用程序的运行时间。

1.5.1 处理器并行性概述

通常有两种处理器并行性的使用形式，以它们的同步粒度来区分。

- *同步的处理器并行性*: 此策略复制全部处理器，使每个处理器在数据空间的不同部分上执行相同的程序。这类并行系统的例子有Thinking Machines CM-2，MasPar MP-2和AMT DAP，都是在20世纪80年代晚期或90年代早期引入的。同步系统的主要优点是同

步操作价廉，因为指令是按锁步方式执行，所以这些机器有利用更细的并行性粒度的能力。另一方面，对于有分支的代码，同步机器不是十分有效，因为它们必须执行序列中
17 分支的两边，不能在每一边使用不同的处理器。

- *异步的处理器并行性*：并行性的第二种形式复制全部处理器，但是让每个处理器以粗粒度显式同步执行不同的程序或相同程序的不同部分。图1-11显示有代表性的带共享全局存储器的异步并行机器结构。使用这种设计的多处理器称为*对称的多处理器*（SMP），从各种供货商那里可以得到这类产品，包括Compaq，Hewlett-Packard，IBM，Intel，Sequent，Silicon Graphics和Sun Microsystems。在这些机器上必须显式地说明任何需要的处理器之间的同步，因为在同步点之间各处理器的执行是独立的。异步并行性的问题是，启动并行任务的代价比较高——必须为每个处理器产生一个进程，并且在访问任何共享的数据之前必须使各处理器同步。由于这种高的开销，仅当有足够的工作补偿开销时，使用并行执行才是必要的。

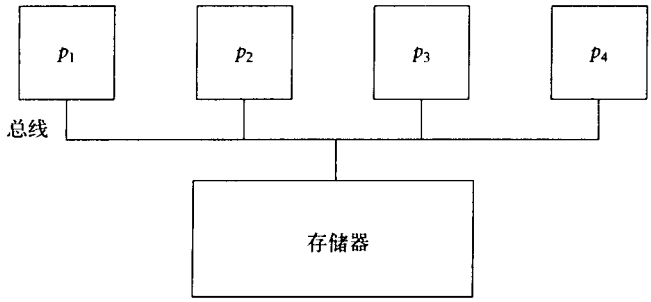


图1-11 异步共享存储多处理器

尽管并行性有优点，但它对硬件和软件设计者提出了许多问题。当一台机器包含多个处理器时，需要有某种机制在不同的处理器之间共享资源。这些处理器还必须有能力互相通信，以便传递数据和协调计算，这涉及到复杂的硬件机制。最后，并行机器的软件一般来说比标量机和向量机的软件复杂得多。为此，并行处理比向量处理花了更长的时间才获得广泛的接受。然而，在本书出版时，具有适当数量处理器的共享存储并行计算机已成为科学和工程计算工作站的标准机器。

虽然用各种形式的并行处理能达到令人印象深刻的速度，但是，快速硬件仅仅是快速程
18 序的一种必要条件。除非软件能有效地利用处理器内部提供的并行性，否则硬件简直变成浪费昂贵的资源。因此，增加应用程序的有效计算速率需要在语言级为挖掘并行性开发出若干机制。

1.5.2 异步并行性的编译

虽然乍一看异步并行性的编译应当粗略地显现出与向量化相同的挑战，但是，它有另外的一些复杂性。首先，在并行机上执行调度允许有更多的灵活性。为了支持对这一主题的讨论，我们为并行循环引进一种Fortran表示法。

PARALLEL DO语句类似于许多Fortran方言中使用的结构，保证在它的迭代之间没有调度限制。因此，按照任何调度可以并行地执行不同的迭代。换句话说，该语句是程序员给系统的*断言*，让系统无约束地并行执行这些迭代。Bernstein在1966年的一篇文章[40]中，确认两个迭

代I_1和I_2能安全地并行执行，如果

（1）迭代I_1不写入被迭代I_2读取的单元

（2）迭代I_2不写入被迭代I_1读取的单元

（3）迭代I_1不写入也要被I_2写入的单元

作为具体说明，考虑下面的例子，它违反Bernstein的条件:

```
PARALLEL DO I = 1, N
   A(I + 1) = A(I) + B(I)
ENDDO
```

这种简单情况类似于向量化中使用的例子。这里，在一次迭代中存入A(I+1)，修改下一次迭代中从A(I)读取的相同单元。因为没有说明调度，所以从这些迭代的第二个迭代中得到的结果会有差异，这取决于在迭代1中是否先发生存入，在先发生存入情况下，它将产生与串行循环相同的值，或者迭代2中的读取发生在迭代1的存入之前，在这种情况下，结果将是不同的。注意，一次执行与另一次执行的答案可能不同。

下面的循环表现出一种更微妙的情况:

```
PARALLEL DO I = 1, N
   A(I-1) = A(I) + B(I)
ENDDO
```

这里存入的一个单元是在前一次迭代中使用过的。在多数向量体系结构中，能安全地向量化这种循环，但是并行化编译器必须更加小心。例如，考察特殊的迭代I=2和I=3。按串行模拟，在迭代2中发生的取A(2)，总是先于在迭代3中对A(2)的存入。在并行版本中，取数可能在存入之前或之后来到，导致循环对A计算出不同的值。

最后，考察一个违背Bernstein第三个条件的例子:

```
PARALLEL DO I = 1, N
   S = A(I) + B(I)
ENDDO
```

19

按串行类比，循环执行后，S总是包含相同的值——迭代N中赋予的值。在并行版本中，任何迭代都可能是最后一次执行，赋给S的值是高度不确定的。

为了保证正确性，现代并行化编译器仅当它能验证所有Bernstein条件成立时，才将一个顺序循环转换成一个并行循环。

由异步并行机引入的第二个新问题是*并行粒度*。因为异步并行进程有很大的启动和同步开销，所以不应当无条件地启动一个并行循环，除非有足够的工作补偿增加的代价。与此对比，向量部件上的同步开销相当地小，以至允许单语句循环的有益向量化。因此，由于异步处理器的高代价，程序员应当力求将并行循环生成的同步频率最小化，这就相当于增加并行迭代的粒度。

对编译器来说，这意味着必须并行外循环而不是内循环（在向量化时选择的是内循环），这样才能使处理器必须同步的次数最小化。类似地，编译器必须能并行化带子程序和函数调用的循环，因为子程序调用是计算的好的来源，并且任何具有大量计算的循环几乎都包含若干调用。这些考虑使得并行化编译器的工作要比向量化编译器更困难，因为它们要负责分析更大的区域，包括跨越多个过程的区域。

异步并行机编译器面对的最后挑战是由访问大的全局存储器引发的。历史上某些最重要的并行系统（如Intel iPSC 860）没有全局共享存储器。替代的是每个处理器仅可以访问和它封装在一起的存储器。这样的机器有时称为*多计算机*，以便将它们与*多处理器*（典型地共享存储器）区分开。现今的SMP*机群*（例如IBM SP）在单结点上少量处理器之间共享存储器，但不能直接访问不同结点上的存储器。多计算机和SMP机群的并行化编译器必须决定诸如这
20 样的问题，哪些存储器保存哪些变量，以及何时要求通信原语将数据移到一个不拥有它的计算结点上。这些问题是很难回答的，第14章将讨论它们。

1.6 存储层次结构

存储层次结构是现代体系结构的一个复杂方面。当不断改善的处理器速度比存储器的速度快时，主存与处理器之间的距离变得更大了（以一个寄存器取数的周期数度量）。二十年前，从存储器取数很少有多于4个周期的；今天，超过50个周期是很普遍的。在并行机上，这种趋势特别明显，这里需要复杂的内部连接使每个处理器能访问所有的存储器。在并行处理器上，取数耗时可能高达几百个机器周期。结果，多数机器包括这样一些功能部件，能用它们来改善由于长的访存时间而引起的性能问题。遗憾的是，处理器速度与存储器速度的比率不可能很快得到改善，因为总的存储容量也在增长，技术转移的代价太昂贵了，必须认真地考虑。

1.6.1 存储系统概述

有两个通用的存储系统性能测度：

- *时延*是从存储器传送单个数据元素需要的处理器周期数。
- *带宽*是每个周期从存储器系统能传送到处理器的数据元素个数。

对分析性能来说，这两种测度都是重要的。时延确定处理器从主存取需要的值必须等待的时间。许多处理器停顿直到完成从主存取数；在这些处理器上，最小化对存储器的请求量是重要的。其他的处理器将对未完成的存储请求继续工作，但是当另一个操作需要结果时，它们必须停顿；在那些处理器上，重要的是力求在取数和使用它的结果之间调度足够的操作以保持处理器不停地工作。带宽决定了每个周期能支持多少个存储操作；带宽越高，一次能取到的存储值就越多。

有两种对待处理器时延的方法：避免和容许。*避免时延*涉及到这样的策略：减少计算中
21 遇到的有代表性的时延。存储层次结构是避免时延的最常用机制。如果多次被引用值存放在快速中间存储器（如像处理器的寄存器或高速缓存）中，那么第一次引用后其余引用的代价是很低的。避免时延技术也改善存储带宽的有效利用。

*延迟容忍*是指在取数据时做一些延迟容忍其他事情。使用显式预取或无阻塞取数是两种容许时延的方法。另一个有趣的延迟容忍机制称为*同步多线程*，是在Cray/Tera MTA上使用的。该机器提供快速的现场切换，使每个周期改变一新的执行流成为可能。如果有足够的流是现役的，并且控制以一种轮式风格连续地从一个流切换到另一个流，那么对每个流来说将出现很小的时延。

本书中，我们集中于用存储分层结构达到避免时延和采用高速缓存行预取的方法实现延迟容忍，因为它们在目前的实践中更常用。虽然存储层次结构的意图在于透明地以层次结构

最慢层的代价提供最快层的性能，但是达到这个目标的程度取决于程序如何有效地重用在高速缓存或寄存器中存储的值。为提供更多的重用机会，重构程序往往能导致处理器性能的显著改善。

1.6.2 存储层次结构的编译

虽然存储层次结构的意图在于克服相对较差的系统存储性能，但是它们并不总是很成功的。当它们失败时，其结果是非常差的性能，因为处理器总在等待数据。对具有高速缓存存储器的机器来说，在小的测试题目（通常是在购买前用来测试机器的）上达到高性能是很普遍的，但是当问题增长到一个现实的规模时，性能就很差。这通常意味着较小问题的整个数据集能适合装入高速缓存，但是大问题不能。其结果，对大问题来说，每次数据访问可能发生高速缓存不命中，但对较小的数组，每个数组元素仅不命中一次。

用一个简单的例子来说明此问题：

```
DO I = 1, N
   DO J = 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

假设在一台机器上执行此循环嵌套，该机器有一个字的高速缓存块，并且当需要空间时，高速缓存总是置换最近最少使用的（LRU）块。虽然此例代码有效地访问数组A的元素（每个元素仅导致一次不命中），但是当M足够大时，对B(J)的访问总是不命中高速缓存。按字计算当M的增长比高速缓存的容量大时，就会发生性能由好到坏的变化。因为B的一个元素在使用B的 22
其他M-1个元素之前，不能被重用，当M足够大时，B的元素有机会在I循环的下一次迭代中被重用前，LRU高速缓存已将B的每一个元素收回了。

缓解此问题的一个方法是对内循环分段，使其大小适合高速缓存，并将“按段步进”的循环交换到最外的位置：

```
DO JJ = 1, M, L
   DO I = 1, N
      DO J = JJ, JJ + L-1
         A(I) = A(I) + B(J)
      ENDDO
   ENDDO
ENDDO
```

这里L必须小于高速缓存中的字数。第二个例子B的每个元素仅有一次不命中高速缓存，因为B的一个元素留在高速缓存中直到它的所有使用结束。代价是对A的引用不命中的次数会增加。然而对A引用不命中的总数近似于 $(NM)/L$，相反，在原代码中引用B的不命中为 NM。

虽然程序员能对他们的源程序做如此的改变，但这样会导致需用手工的方法从特定机器代码重新产生每台新机器的代码。我们相信这类特定机器优化应当是优化编译器做的事情。后面几章将说明如何改造由向量化和并行化编译器发展起来的编译器技术，用于优化存储层次结构的使用。

1.7 实例研究：矩阵乘法

为说明机器体系结构如何影响为获得最高效率应对特殊计算编写程序的方法，现在我们

说明矩阵乘法代码的几种不同方案，矩阵乘法是处于许多重要科学应用中心位置的一项计算。计算两个矩阵A和B的乘积，用如下所示的典型Fortran循嵌环套:

```
DO I = 1, N
   DO J = 1, N
      C(J, I) = 0.0
      DO K = 1, N
         C(J, I) = C(J, I) + A(J, K)*B(K, I)
      ENDDO
   ENDDO
ENDDO
```

23

此代码段采用一个计算积矩阵C的简单策略：两个外循环选择积矩阵的一个特定元素做计算，内循环则计算该元素，取第一个矩阵的行和第二个矩阵的相应列作内积。在一台标量机上（没有对并行处理或向量操作的支持），此代码使硬件得到极好的使用。因为内循环将乘积累加到C(J,I)中。在循环过程中对C不需要访问存储器（或取或存）。只要中间结果放在一标量寄存器中，直到此循环完成，然后将结果存入存储器中。当呈现这样的代码段时，一个好的优化编译器生成的代码应当接近于标量机器可能有的最优性能，可是这要求识别数组量C(J,I)是内循环的不变量，并且能分配到一个寄存器。

在一台带有流水线浮点部件的标量机器上，相同的代码很可能不会有同样的进展。此代码在一台无流水线的标量机器上能有效地运行，因为最内层循环一次迭代的结果立即在下一次迭代中被使用，使重用寄存器中的结果成为可能。在一台流水线的机器上，每次迭代在可以开始做自身最后的加操作之前，必须等待前一次迭代中最后的加操作是有效的（见图1-12）。克服此问题的一个办法是在相同的时刻让一个外循环在四个不同的迭代上工作，如此用四个无关的计算填满四级流水线，如图1-13所示。下面给出的是完成此项工作的Fortran代码（这里假设N是4的倍数）:

```
DO I = 1, N
   DO J = 1, N, 4
      C(J, I) = 0.0
      C(J + 1, I) = 0.0
      C(J + 2, I) = 0.0
      C(J + 3, I) = 0.0
      DO K = 1, N
         C(J, I) = C(J, I) + A(J, K)*B(K, I)
         C(J + 1, I) = C(J + 1, I) + A(J + 1, K)*B(K, I)
         C(J + 2, I) = C(J + 2, I) + A(J + 2, K)*B(K, I)
         C(J + 3, I) = C(J + 3, I) + A(J + 3, K)*B(K, I)
      ENDDO
   ENDDO
ENDDO
```

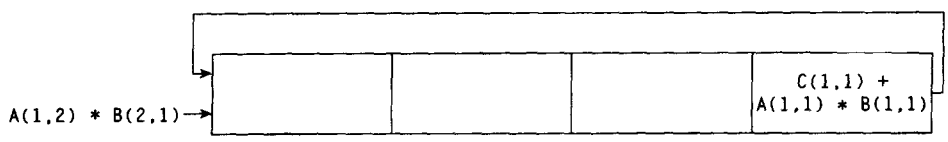


图1-12 矩阵乘法执行流水线互锁

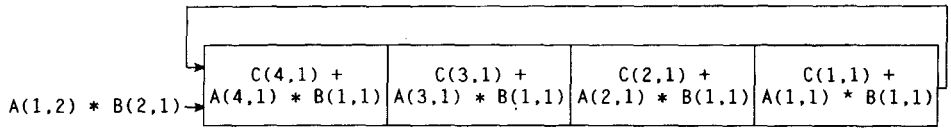


图1-13 通过外循环展开使流水线充满

在一向量机上不能向量化标量矩阵乘法的内循环，因为计算是递归的——即需要此标量代码的交替存取顺序，以保持代码的含义（因为精度的要求，假设加法的重新结合是禁止的）。在一台像Cray T90的向量机器（向量长度为64）上，我们需要将64个元素的向量操作移至内循环，使得在每次迭代中能重用T90的向量寄存器。因此对T90来说，最好的代码看上去像下面的形式: 24

```
DO I = 1, N
   DO J = 1, N, 64
      C(J:J + 63, I) = 0.0
      DO K = 1, N
         C(J:J + 63, I) =  C(J:J + 63, I) +  A(J:J + 63, K)*B(K, I)
      ENDDO
   ENDDO
ENDDO
```

矩阵乘法的这种形式将第一个代码的标量寄存器特征扩展为Cray的向量寄存器。现在k-循环执行积矩阵一个向量段需要的所有计算；其结果就像以标量形式累加单个元素一样，向量段可以被累加到一个向量寄存器中。

在一台带有四路同时发射、四个浮点乘法-加法器和四级流水线的VLIW机器上，一个好的代码形式可能是 25

```
DO I = 1, N, 4
   DO J = 1, N, 4
      C(J:J + 3, I) = 0.0
      C(J:J + 3, I + 1) = 0.0
      C(J:J + 3, I + 2) = 0.0
      C(J:J + 3, I + 3) = 0.0
      DO K = 1, N
         C(J:J + 3, I) = C(J:J + 3, I) + A(J:J + 3, K)*B(K, I)
         C(J:J + 3, I + 1) = C(J:J + 3, I + 1) + A(J:J + 3, K)*B(K, I + 1)
         C(J:J + 3, I + 2) = C(J:J + 3, I + 2) + A(J:J + 3, K)*B(K, I + 2)
         C(J:J + 3, I + 3) = C(J:J + 3, I + 3) + A(J:J + 3, K)*B(K, I + 3)
      ENDDO
   ENDDO
ENDDO
```

这里的意图是，可以在相同的周期中发射C(J:J+3,I)的四个乘法-加法，随后是索引为I+1的那些乘法-加法，等等。这种代码形式将使四个浮点部件保持忙碌工作。

必须考虑到对称式多处理器与向量机使用的不同。因为向量循环是由硬件“同时地”执行，它们必须是给定嵌套中的最内层循环。另一方面，并行循环是异步执行的，因此要求将它们移到最外层位置上，以保证有足够的计算去补偿启动和同步的开销。因此，在像Sun Starfire那样SMP机器上，矩阵乘法的最佳形式应是

```
PARALLEL  DO I = 1, N
   DO J = 1, N
      C(J, I) = 0.0
      DO K = 1, N
         C(J, I) = C(J, I) + A(J, K)*B(K, I)
      ENDDO
   ENDDO
ENDDO
```

在这种形式中，每个处理器能独立地计算积矩阵的一列，而无需与其他处理器同步，直至完成积的计算。这种形式要求所有处理器访问整个A矩阵和B矩阵的一列，这在SMP中真正是平常的事实，其中存储在所有处理器之间是共享的。

在一台没有流水线浮点部件的单标量处理器上，它的高速缓存足以装下多于$3L^2$个浮点数而不发生高速缓存冲突（假设高速缓存是全相联的），我们可以期望将高速缓存分块，用于一
26 次乘一个子矩阵。在下面的代码中，假设L能整除N:

```
DO II = 1, N, L
   DO JJ = 1, N, L
      DO i = II, II + L-1
         DO j = JJ, JJ + L-1
            C(j, i) = 0.0
         ENDDO
      ENDDO
      DO KK = 1, N, L
         DO i = II, II + L-1
            DO j = JJ, JJ + L-1
               DO k = KK, KK + L-1
                  C(j, i) =  C(j, i) + A(j, k)*B(k, i)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

这里的想法是，第一个循环嵌套初始化C的一个L × L块，第二个循环嵌套计算该块的值。通过对K层循环分块，我们能获得A的L × L块和B的L × L块的重用。换句话说，在KK-循环的每次迭代中，我们将A的一个L × L块和B的一个L × L块相乘，并将结果加到C的一个L × L块中。

从这里矩阵乘法的研究中，有两个重要的经验应该说是很明显的:

- 用源程序的并行性的显式表示对保证并行硬件的优化使用是不充分的。六种类型机器的每一种需要并行性的不同表示——在某些情况下，基本上不相同。此外，为得到特定机器的最好形式，需要该机器体系结构的详细知识。这个观察结果启示我们必须要为各种体系结构定制显式并行程序；否则当从一台机器向另一台机器移植时，它们会丢失效率。
- 尽管矩阵乘法的最好形式对每一类机器是不同的，但是，所有这些形式能从最初的非并行源程序用相对简单的程序变换导出。通过简单的交换循环的执行顺序就能获得大多数程序形式。

已知软件的生存期在增加，而硬件的生存期在减少，这些经验启示我们为特定机器体系

结构定制代码最好留给编译器去做。下一节介绍为能实现此目标而设计的编译器技术的基本要素。 27

1.8 先进编译技术

编译器的任务就是将人们容易理解的计算的高级表示转换成机器能执行的低级表示。人们的高级表示很少打算去适应机器体系结构的细节，所以生硬的翻译过程可能在生成的机器级程序中引进低效率。编译器优化阶段的目的就是要消除这些低效率，并将计算的表示转换成一个针对特定体系结构的调整好的表示。

为特定体系结构定制程序，是编译器应负的责任，其基本原理的自然含义是让编译器负责自动转换代码去利用并行性。在这种观点下，把源程序当成一份“规格说明”，它定义程序计算的结果。编译器用任何有意义的方法自由地变换程序，只要求变换后的程序计算出与原规格说明相同的结果。尽管这种观点看起来似乎很自然，但是，经常会受到怀疑，因为需要彻底而完善的变换去挖掘并行性。例如，矩阵乘法需要循环交换、循环分裂、循环分布、向量化和并行化，从而达到各种并行体系结构的优化结果。在传统的优化器中往往找不到这些变换。

本书介绍在编译器方法背后的理论和实践理念，这种编译方法是基于积极的变换串行规格说明的范例。我们主要集中精力于揭示Fortran 77程序中并行性的方法，因为Fortran是当今科学计算的通用算法语言。我们将不争论这种方法与从并行计算出发的方法相比的价值问题；这两种方法在它们的推崇者之间有很大的争议。然而，我们将说明从一串行语言出发有非常实际的优点，那就是它把大量现存的程序作为目标。另外，本书将演绎串行程序中揭示并行性使用的理论也同样能应用到优化显式并行程序中去。

1.8.1 依赖

当程序员用串行程序设计语言编写应用程序时，他希望程序计算的结果，首先是从第一个语句得到的，然后是从第二个语句得到的，等等，其中只有分支语句和循环语句这样的控制流结构属于例外情况。实质上，程序员已说明了他希望计算机执行操作的具体顺序。显然，直接从这样的规格说明中利用并行性是不可能的，因为并行化要改变操作的顺序。 28

先进编译器必然会遇到的基本挑战是，确定何时一个与程序员说明的顺序不同的执行顺序（包括并行执行或向量执行）总会计算出相同的结果。换言之，串行语言引入的约束对维持计算的含义并不是关键性的；变换这样的程序为并行形式的关键是，找到保证变换后的程序对每次输入将产生正确结果所必需的最少的约束。如果能精确地刻画这些约束，那么就能使编译器用不改变这些约束的任何方法重排程序的执行顺序。

在本书中，我们研究一组称为依赖的约束，它们足以保证程序变换不改变用计算结果表示的程序含义。这些约束不是精确的；有些情况下可以违背它们而不会改变程序的含义。然而，它们抓住保持命令式语言中正确性的一个重要策略：它们保持程序中对每一存储单元取数和存数的相对顺序。（它们不保持相同单元读数的相对顺序，不过这不影响程序的含义。）我们会看到能够扩展依赖概念来保持控制判定对其后操作的影响。

特别地，依赖是程序中语句上的关系。语句偶 $\langle S_1, S_2 \rangle$ 在此关系中，如果按任何有效的程序重排的顺序（倘若维持了访存的序），S_2必定在S_1之后执行。

为了说明依赖概念，考虑下面的简单代码段：

```
S1    PI = 3.14159
S2    R = 5
S3    AREA = PI*R**2
```

此代码段的结果定义为取执行顺序〈S_1,S_2,S_3〉时发生的那些结果。然而，没有理由要求代码段中S_2在S_1之后执行；事实上，执行顺序〈S_2,S_1,S_3〉产生与原序完全相同的结果（即变量AREA的值相同）。与之不同的是，S_3的执行时刻是要紧的；如果它在S_1或S_2之前执行，会计算出不正确的AREA值，因为尚未设置输入操作对象的值。根据依赖，语句偶〈S_1,S_3〉和〈S_2,S_3〉是在此代码段的依赖关系中，而〈S_1,S_2〉不在。

直线型代码中的依赖是一个容易理解的概念。不过，仅检测直线代码不能保证并行性的有效使用。为了达到高性能，我们必须将依赖概念扩展到程序最频繁执行的部分，使它有可
29 能处理循环和数组。下面的例子说明由这些扩展引入的复杂性:

```
      DO I = 1, N
S1      A(I) = B(I) + 1
S2      B(I + 1) = A(I)-5
      ENDDO
```

此循环呈现出依赖〈S_1,S_2〉，因为在每次迭代中计算的A值立即在S_2中使用，而依赖〈S_2,S_1〉是因为除了第一次使用B值之外，每次循环迭代使用前一次迭代中计算的B值。检测这些依赖是相当困难的，因为不同的循环迭代涉及到不同的数组元素。然而，这还仅仅是问题的一部分；依赖图中的环（指明此循环的周期性质）使利用并行性的调度算法复杂化。

循环和数组仅仅是编译器必须学会去处理的一些结构。IF语句也引入了问题。事实上，根据仅在运行时得到的有效值可能条件地执行这类语句，在变换过程中这是一个独特的复杂问题。关于这一点，一个程序的依赖可能有条件地遵从某些关键变量的值（仅在运行时可以使用的信息）。

本书前面过半部分专注于研究依赖性理论和在程序中精确地构造它们的方法。

1.8.2 变换

正如迄今所描述的，依赖用于挖掘程序中隐含的并行性仅仅是一种被动的导向。然而它远非如此。因为依赖说明可以如何重排程序的执行顺序，依赖还能构成强有力的变换系统的基础，增强程序中并行性的显现。例如，简单地交换两个数组

```
DO I = 1, N
   T = A(I)
   A(I) = B(I)
   B(I) = T
ENDDO
```

不能直接向量化，因为标量临时变量T构成向量计算的瓶颈：正如说明的那样，程序中每次仅能传输一个元素。如果将标量暂存扩展为一个向量暂存

```
DO I = 1, N
   T(I) = A(I)
   A(I) = B(I)
   B(I) = T(I)
30 ENDDO
```

那么循环就能直接向量化。通过检查程序的依赖性能确定这种变换的合法性。（这种特殊的变

换称为标量扩展，在第5章中讨论。）

第5、6章以及本书后半部分讨论依赖对程序变换支持的应用。为了用例子说明程序变换的概念，必须要有一种语言能显示这些例子。因为当今在并行和向量计算机上Fortran仍是最有份量的使用语言，最合理的语言选择是Fortran的扩展版本，它具有向量和并行操作。为此，我们将使用增加并行循环语句的Fortran 90。附录含有对Fortran 90特点的简单介绍。

1.9 小结

这一章介绍了编译高级语言的基本问题，即产生的代码在高性能计算机系统上达到可接受的性能。本章概括了重要的体系结构特色，包括流水线、向量并行性、VLIW和超标量处理、异步并行性以及存储层次结构。对每一特色，我们介绍了那些有助于改善性能的优化。其中的策略包括指令调度、自动向量化、自动并行化以及存储层次结构中改善重用的循环变换。

实施这些变换的编译系统的工作是：为维持计算的结果，确定总体排序中哪些是必须遵守的执行顺序。这样的执行序是在称为依赖的关系中捕捉到的——我们说语句S_2依赖于语句S_1，如果在串行程序的执行中S_2跟随在S_1之后，并且在任何变换后的程序中，S_2必须在S_1之后，这样可计算出相同的结果。按这个定义，编译器可以自由地重排语句实例的顺序，只要不改变涉及到依赖中任何两个语句的相对顺序。

1.10 实例研究

本书的每一章中包含一节“实例研究”，讨论作者在本章中描述的概念的实际实现方面的经验。这些经验多数集中于两个主要的系统：PFC和Ardent Titan编译器。

PFC系统是由两位作者在Rice大学开发的，项目来自IBM的一个重点研究合同。最初PFC专注于向量化，它作为IBM后来的一个向量化编译器的模型。PFC是在依赖和程序变换两个孪生概念上构建的，并作为一些研究的框架，这些研究提出的题目除了向量化以外，还包括并行化、存储层次结构管理以及过程间分析和优化。它是Rice大学一系列相关研究系统的起点，其中包括PTOOL（一个程序并行化工具，显示源代码中直接妨碍并行性的依赖（种类）），ParaScope（一个并行程序设计环境），以及D系统（高性能Fortran（HPF））的一个编译器和编程环境。 31

Allen离开Rice大学后，加入了Ardent计算机公司，领导他们的编译器开发工作。在那里，他构造了一个商用性的编译器，用于Titan计算机系列。Titan编译器使用了许多在PFC中用过的相同算法。可是，PFC主要是一个源码到源码的翻译器，而Titan编译器为一台真实计算机产生机器语言代码，该计算机具有许多独特的和挑战性的特色。另外，该编译器必须处理C和Fortran，它提出的一系列特殊挑战在第12章讨论。

这两个系统提供丰富的经验来源，我们希望举例阐明对基本原理的讨论，在每一章的主要部分中可以找到这些讨论。

1.11 历史评述与参考文献

研发作为积极程序变换基础的依赖性，是许多研究工作努力的结果。在Massachusetts Computer Associates所作的自动向量化和并行化，即后来称之为 Compass的传世之作，在历史上具有特别重要的意义。20世纪70年代早期，Compass为Illiac IV建立了一个变换工具，称为Parallelizer。描述Parallelizer的主要论文包括由Lamport撰写的理论和技术报告[195，196]，

以及由Loveman撰写的优雅文章，他推崇有关优化的一个原理性方法[204]。

在David Kuck指导下Illinois大学最先提出最有影响的自动程序变换理念。Illinois大学在Illiac系列计算机非常早的发展阶段就已经涉及到并行性；20世纪70年代中期，Parafrase项目一开始就把研究方向对准了自动程序变换。从项目的名称就能猜测到，Parafrase试图将串行程序重新表述为等价的并行程序和向量程序。此项目中产生了许多重要的文章，其中有关于依赖测试[32，190，283]、循环交换[280]、辅助变换[285，259]和低层并行性[219]的文章。Parafrase项目的自然产物是Kuck and Associates公司，这是在市场销售基于依赖的先进程序变
32 换系统的一家商业企业。

20世纪70年代后期，在Ken Kennedy指导下，Rice大学开始了并行Fortran转换器(PFC)项目[20]。最初该项目集中于扩展Parafrase，增加新的能力。然而，由于在内部数据表示上受到限制，因此搁下了Parafrase方案，并实现了一个完整的新系统。PFC的进步包括一个依赖的明晰分类，弄清楚了能应用依赖的许多领域，对许多变换有效的算法（特别是向量代码生成和循环交换），处理程序内条件语句的方法，以及增强并行性的新变换[16，21，23]。PFC项目本身作为IBM 3090向量功能[243]和Ardent Titan编译器[17]的向量化编译器的原型；构成PFC基础的理论也是Convex向量化编译器的基础。

习题

1.1 为表达式X*Y*2+V*W*3产生DLX代码。假设取数需要3个周期，立即取数一个周期，浮点加法一个周期，浮点乘法2个周期，为此代码找出最佳调度。假设寄存器数量是你所需要的，但是只有一个load-store部件和一个浮点算术部件。所有指令是流水线的，但机器每个周期至多能发射一条指令。你使用什么策略寻找最佳调度？

1.2 在下面的假设下重做习题1.1：每个时钟周期你能发射两条指令，有两个load-store部件和两个浮点算术部件。

1.3 根据本章的$N \times N$矩阵乘法例子，产生一个在有32个处理器的T90这样的并行-向量机上能顺利执行的版本。

1.4 产生一个$N \times N$矩阵乘法版本，它能在一台对称多处理器上顺利执行，系统中每个处理器有一个非流水线的执行部件和一个全相联的高速缓存，它的容量足以保存略多于$3L^2$个矩阵元素，这里L能整除N。

1.5 你可以利用任何共享存储高性能计算机系统，用Fortran或者C实现一个1000 × 1000矩阵乘法版本，要达到最高可能的性能。特别要注意存储层次结构，尝试找出正确的高速缓存
33 块尺寸。

第2章

依赖：理论与实践

2.1 引言

正如我们在第1章中已学到的，对于程序设计语言被接受，优化有重要的作用——如果优化器做不好此项工作，那么无人愿意使用这种语言。其结果导致优化技术变得十分复杂。然而优化技术的大多数研发工作集中在标量机上。在这类机器上，主要的优化是寄存器分配、指令调度和减少数组地址计算的开销。

在优化过程中并行性引进了更多的复杂性。对并行计算机来说，主要的优化变成了在串行代码中寻找并行性，以及将并行操作定制到目标机，其中包括它的存储层次结构。探寻有用的并行性的主要策略是寻求*数据分解*，对分解的数据，并行任务在数据数组的不同元素上执行类似的操作。在Fortran中，此策略自然地映射到并行DO循环的不同迭代上。对科学计算问题来说数据分解是有效的，因为当问题的规模增大时，分解是*可扩展的*，所以产生了有效的并行性。

假定将目标集中在Fortran的循环迭代上，一个先进的编译器必须能够确定每一对迭代是否满足Bernstein条件[40]（见1.5.2节）。检查循环迭代是相当复杂的；事实上在数据-并行循环内，多数代码会引用下标数组变量，这就增加了复杂性。因此，编译器必须要具有分析这种引用的能力，判定两个不同的迭代是否访问相同的存储单元。已知一个数组包含多个元素，何时两个迭代或语句引用相同数组的不同元素这件事，并非总是明显的。由于这种增加的复杂性，第1章介绍的依赖的简单定义纵然雅致而且易于理解，在出现循环和数组引用的场合是无效的。

34
~
35

本章的目的是详尽阐述依赖的定义和性质，作为一个受限的系统，它维持对循环嵌套有关的存储的数据访问顺序。这一章还将建立若干基本结果，它们构成后面几章的基础。一个主要目标将是确立依赖对自动并行化和向量化的适用范围。为说明依赖的能力，用一个向量化算法结束本章，该算法是几个商用编译器的核心。

2.2 依赖及其性质

1.8节勾画的基本方法，在本书中用来显露并行性。这种方法的基础是*依赖关系*，或*依赖图*，对一个给定的串行程序，依赖图包含语句到语句的执行顺序集，能用它指导选择和应用某些变换，这些变换保持程序的含义。图中每一对语句称为一个*依赖*。给出程序的一个正确的依赖图，任何基于序的优化，只要它不改变程序的依赖关系，将保证不改变程序的结果。

依赖代表对程序变换的两类不同约束。首先，设计某些约束保证数据按正确的顺序生产和消费。由这些约束产生的依赖称为*数据依赖*。回顾第1章的例子

```
S1      PI = 3.14
S2      R = 5.0
S3      AREA = PI*R**2
```

语句S_3不能移到S_1或S_2前，避免对变量PI和R产生可能不正确的值。为了阻止这么做，我们将构造从语句S_1和S_2到语句S_3的数据依赖。S_1和S_2之间不需要执行约束，因为执行顺序S_2，S_1，S_3将正确地产生PI的值，它与执行顺序S_1，S_2，S_3的值相同。

另一类引发依赖的约束是控制流。例如，考虑代码

```
S1      IF(T.NE.0.0) GOTO S3
S2         A = A/T
S3      CONTINUE
```

36 在正确变换后的程序中，S_2不能在S_1之前执行，因为S_2的执行是以S_1中的分支执行为条件。在S_1前执行S_2可能引发以0做除数的异常，这在原始版本中是不可能发生的。由控制流引发的依赖称为*控制依赖*。

当正确地并行化一个程序时，虽然必须考虑数据依赖和控制依赖，但是下面几章将专门集中于数据依赖，这种依赖对于理解和说明多数重要的原理更为简单。第7章将说明如何把这些原理用于控制依赖，或通过使用称之为“if转换”的技术将它们转换成数据依赖，或者将控制依赖纳入其扩展的算法之中。

我们回到数据依赖，如果保证以正确的顺序生产和消费数据，则我们必须要保证没有交换对相同单元的load和store操作；否则load操作可能得到错误的值。此外，我们还必须要保证按正确的顺序做两次store操作，使随后的store操作会得到正确的值。由这些概念地形式化引出下面的数据依赖定义。

定义2.1 从语句S_1到语句S_2存在*数据依赖*（语句S_2*依赖于*语句S_1），当且仅当（1）两个语句访问相同的存储单元，并且其中至少有一个语句存入此单元，（2）存在一条从S_1到S_2的可能的运行时执行路径。

下面几小节将讨论依赖的各种性质，据此可对它们进行分类。这些性质对理解本书后面介绍的算法是十分重要的。

2.2.1 存-取分类

按照存-取顺序的表示，在程序中可能有发生依赖的三种方式。

1. *真依赖*。第一个语句对一个单元存入数据而后由第二个语句读出：

```
S1   X  =  ...
S2   ...  =  X
```

此依赖保证第二个语句接收到由第一个语句计算的值。这种类型的依赖也称为*流依赖*，并用$S_1\delta S_2$表示（读作S_2依赖于S_1）。为用图形显示依赖，通常是画一条边表示从先执行的语句（源
37 点）流向稍后执行的语句（汇点）。

2. *反依赖*。第一个语句从一个单元读出数据，随后第二个语句对此单元存入数据：

```
S1   ...  =  X
S2   X  =  ...
```

这种依赖阻碍S_1和S_2的交换，交换可能导致S_1不正确地使用S_2计算的值。本质上，此种依赖的出现阻碍程序变换，因为它会引入在原始程序中不存在的一个新的真依赖。这样的反依赖用$S_1\delta^{-1}S_2$表示。在某些教科书中，反依赖也表示成$S_1\delta^{-}S_2$。

3. 输出依赖。两个语句对同一单元写入数据:

```
S1   X = ...
S2   X = ...
```

这种依赖阻碍交换，因为交换可能引起后面的语句读入错误的值。例如，在代码段

```
S1   X = 1
S2   ...
S3   X = 2
S4   W = X * Y
```

中，不允许将语句S_3移到S_1之前，以免在S_4中Y不正确地乘1而不是乘2。这类依赖称为输出依赖，并表示为$S_1\delta^o S_2$。

在硬件设计行文中，依赖通常称为相关或停顿，这是由于它们对流水线的影响。真依赖与RAW（写后读）相关相同；反依赖等价于WAR（读后写）相关；输出依赖是一种WAW（写后写）相关[145]。

2.2.2 循环内的依赖

将依赖概念扩展到循环，需要某种方法使循环迭代过程中执行的语句参数化。例如，在

```
      DO I = 1, N
S1      A(I + 1) = A(I) + B(I)
      ENDDO
```

这样简单的循环嵌套中，任何循环迭代的语句S_1依赖于自身前一次迭代的实例。在这种情况下，说“语句S_1依赖于自己”是真依赖，但是不精确。例如，对单索引做简单的改变能导致 38
此语句依赖于两次迭代前的实例:

```
      DO I = 1, N
S1      A(I + 2) = A(I) + B(I)
      ENDDO
```

因此，精确刻划循环中的依赖需要用循环迭代的某种表示使迭代中出现的语句参数化。为此，我们将构造一个整型向量，表示每个循环的迭代编号，语句嵌在这些循环中。对于简单循环

```
DO I = 1, N
   ...
ENDDO
```

迭代号正好等于循环索引值；第一次迭代的迭代号等于1，第二次迭代的迭代号等于2，如此等等。然而，在循环

```
DO I = L , U , S
   ...
ENDDO
```

中，当I等于L时，迭代号是1，当I等于I+S时，迭代号是2，如此等等[⊖]。

在某些情况下，使用正规化的迭代编号方案是更合适的，其中迭代从1开始以增量1步进到某个上界。下面的定义对这些概念予以形式化:

⊖ 某些教材中定义迭代号等于循环索引I。然而，当步长是负值时，会引起问题。在本书中我们将使用正规化的定义，除非另有注释。在正规化循环中，两者没有差别。

定义2.2 对任意一个循环，其中循环索引I以步长S从L步进到U，一个特定迭代的（正规化）迭代号i等于值$(I-L+S)/S$，其中I是该迭代中索引变量的值。

在一循环嵌套中，一个特定循环的嵌套层等于包围它的循环个数加1。循环从最外层到最
39 内层编号，开始层为1。这样得到的是多重循环中迭代号的自然定义。

定义2.3 给定n个循环的嵌套，最内层循环的一个特定迭代的迭代向量i是一个整型向量，它包含按嵌套层顺序的每层循环的迭代号。换句话说，迭代向量为

$$i = \{i_1, i_2, \ldots, i_n\} \tag{2-1}$$

其中i_k，$(1 \leqslant k \leqslant n)$表示在嵌套层$k$上的循环迭代号。

用特定执行向量参数化的一个语句，表示循环归纳变量具有此执行向量中的值时被执行语句的实例。例如，在

```
DO I = 1, 2
   DO J = 1, 2
      S
   ENDDO
ENDDO
```

中S[(2,1)]表示在I循环的第2次迭代和J循环的第1次迭代中发生的语句S的实例。对一个语句来说，所有可能的迭代向量集合是一个迭代空间。上例中S的迭代空间是{(1,1),(1,2),(2,1),(2,2)}。

对依赖来说，因为执行顺序的重要性，迭代向量需要排序，此序对应于它们的循环执行顺序。假设记法：i是一个向量，i_k是向量i的第k个元素，i[1:k]是一个长度为k的向量，它由i的最左k个元素组成，我们能对长度为n的迭代向量定义词典顺序如下：

定义2.4 迭代i居于迭代j之前，用i<j表示，当且仅当（1）i[1: $n-1$]<j[1: $n-1$]，或者（2）i[1: $n-1$] = j[1: $n-1$]且$i_n < j_n$。

换言之，一个迭代向量i居于另一个迭代向量j之前，当且仅当i描述的迭代中任何执行语句是在j描述的迭代中任何语句之前执行。很容易定义迭代向量的等同关系——迭代编号的对应元素相等。通过对词典顺序的自然扩展，还能在迭代向量上定义关系≤，>和≥。

40 现在我们定义公共循环嵌套中语句之间的依赖。

定理2.1 循环依赖 在公共嵌套循环中存在从语句S_1到语句S_2的依赖，当且仅当此嵌套循环存在两个迭代向量i和j，使得（1）i<j，或i = j并且在循环体中有一条从S_1到S_2的路径，（2）在迭代i中语句S_1访问存储单元M，而在迭代j中语句S_2访问存储单元M，（3）这两个访问之中至少有一个是写入。

该定理是直接从依赖的定义得到的必然结果。条件（2）保证有一条从依赖源点到汇点的路径。

2.2.3 依赖和变换

意欲使程序中的依赖成为一种工具，用来确定何时做某些程序变换是安全的。当我们说

一个变换是“安全”的，通常是指变换后的程序具有与原程序相同的“含义”。换言之，我们不仅关心提供给编译器的原始程序的正确性，而且关注变换后程序是否做与原始程序相同的事情。

但是这里引起这样的问题：必须保持什么样的程序行为？当然我们不需要去保持运行的时间。因为这些变换的目标正是改善性能。非形式地讲，保持可看见的程序效果（如输出的值和顺序）看来抓住了我们想要得到的东西的本质。

变换下的程序等价

传统上，依赖与命令式语言有关联，在这类语言中，每个语句从存储器中读出和存入数据。在命令式程序中，逻辑上程序的含义多半是借助于计算的状态来定义的。计算的状态是保存在存储单元中的所有值的集合；值（和单元）的每个不同集合构成各个不同的状态。显然，两个正好以相同的状态集合进行的计算，它们是等同的，但是如此强的等价定义对优化有过多的限制，因为它对重新安排程序步骤提供很少的灵活性。实际问题是：程序状态详细说明同时发生的程序的全部值。这意味着不允许程序交换对不同的存储单元更新顺序，即使这种变换对输出没有影响。

我们需要一个等价的定义，允许在变换方面有更多的包容性，同时保持程序员希望的正确性。例如，如果一个不同的算法准确地计算出可以替代的相同答案，那么这应该是可接受 41
的。因此，应当允许用任何稳定的排序算法替代冒泡排序法。

为了达到这种效果，我们应当集中在可见程序行为的一致性上。在计算的多数步骤中，内部状态是外部不可见的。恰好存在输出语句使得内部状态“有趣的”方面——程序员试图去计算的东西——成为可见的。因此，下面的说法更加有用：

定义2.5 两个计算是*等价的*，如果对相同的输入，它们按相同的顺序执行输出语句时，输出变量产生等同的值。

此定义允许用不同的指令序列（其中有些比其他的更有效）去计算相同的输出。此外，它还抓住这样的概念：优化必须维持的计算方面仅限于它的输出。如果对术语“输出”定义得足够清楚，那么此定义将充分达到本书的目的。

讨论引起这样的问题：计算的副作用是什么？副作用的一个例子是异常，特别是与一个错误关联的异常。这样的副作用，对优化编译器来说已经成为问题的传统根源，由此产生关于变换“安全性”的广泛文献[167]。很清楚，编译器变换决不应当引入在原程序中不会出现的错误。然而，在维护输出等价的同时，应当允许消除错误异常的变换吗？对于Fortran，常规的答案是“允许”，但是在像Java这样的语言中，它们具有较严格的语义和显式异常，这样回答可能是不可接受的。因为本书集中于Fortran或类似的语言，我们将允许消除异常的变换或调整那些异常发生的时间，只要对给定的输入变换不引入异常，而这种异常在原程序中对相同的输入不会发生。

基于依赖的变换的正确性

用现有的这些变换正确性概念，我们已能确定依赖在建立正确变换中所起的作用。本书中讨论的多数优化是下面定义的“重排序变换”：

定义2.6 *重排序变换*是任何这样的程序变换，它仅改变代码的执行序，不增加或取消任何语句的任何执行。 42

因为重排序变换不取消任何语句的执行，因此在重排序变换前任何两次执行引用一个公共的存储单元，变换后仍将引用相同的存储单元。因此，如果在变换前语句之间存在一个依赖，那么变换后仍然有一个依赖。不过要注意，变换可能*逆转*语句引用公共存储单元的顺序——从而逆转依赖（即变换前为$S_1\delta S_2$而变换后为$S_2\delta S_1$）。这将明显地导致运行时的不正确行为。

定义2.7 重排序变换*维持*依赖，如果它维持该依赖的源点和汇点的相对执行顺序。

现在我们已作好证明关于变换中依赖作用重要结果准备。

定理2.2 *依赖的基本定理* 维持程序中每一个依赖的任何一种重排序变换，将维持该程序的含义。

证明 我们从不包含条件语句的无循环程序开始。令$\{S_1, S_2, ..., S_n\}$是原程序的执行顺序，而$\{i_1, i_2, ..., i_n\}$是语句索引的一个置换，它表示在重排序后程序中语句的执行顺序；即$\{i_1, i_2, ..., i_n\}$是$\{1, 2, ..., n\}$的置换，它对应于语句的重排序。假设依赖得到维持，但含义变了。这意味着某个输出语句产生不同于原程序中相应的输出语句的结果。

为讨论的目的，我们将输出语句简单地看成是计算另一个结果，即正要产生的结果。用这种模型，在变换后的程序中语句序列至少包含一个语句，即输出语句，它产生一个“不正确的”结果，即不同于由原程序中相应语句产生的结果。令S_k是新序中产生不正确结果的第一个语句。因为此语句恰与原程序中的语句相同，在它的输入存储单元之一中必定会找出一个不正确的值。因为在S_k之前所有已执行过的语句产生的值与原程序中的值相同，这样就只有三种情形使S_k能看到在一特定单元M中一
43 个不正确的输入：

1. 在语句S_k读M中结果之前，原先由语句S_m将其结果存入M中，在S_k读之后现在S_m又存入M。这意味着在原程序中重排序未能维持真依赖$S_m\delta S_k$，违背了假设。

2. 语句S_m最初存入M是在语句S_k读M之后，现在S_m又在S_k读它之前写入M。这意味着在原程序中重排序未能维持反依赖（$S_k\delta^{-1}S_m$），违背了假设。

3. S_k读M之前，两个语句按原来的顺序写入M，现逆转它们的执行顺序，引起错误的值留在M中。这将意味着重排序未能维持输出依赖，违背了假设。

既然这里详述S_k可能得到错误值的各种情形，由于与假设矛盾结论得到证明。

循环。为了将此结果扩展到循环上，只要注意到列表中的语句执行可以看成是用迭代向量编上索引号的语句实例。在循环体中任何语句执行之前，提供计算得到的正确的循环迭代次数，然后应用相同的论证。这对保证维持语句实例的全集是必要的。实施这种限制的规范约定是从循环头到循环体中每个语句有一个控制依赖。

带条件的程序。因为在有条件的时候，我们还没有一个重排序变换的定义，目前我们将假设条件是单个宏语句（即if-then-else语句），重排序变换将它当成一个单位来对待。在第7章中将此结果扩展到更一般的变换上。用此单项限制定理得到证明。

定理2.2使我们得到下面的定义：

定义2.8 一个应用到程序上的变换，如果它维持程序中的所有依赖，就说它对程序是*有效的*。

从定理和它的证明应清楚看到：一组有效变换维持程序中对每个存储单元读和写的顺序——只有输入访问能被重排序。因此，有效变换维持的条件比定义2.1说明的等价条件更强，如下例所示：

44

```
L0   DO I = 1, N
L1      DO J = 1, 2
S0         A(I, J) = A(I, J) + B
        ENDDO
S1      T = A(I, 1)
S2      A(I, 1) = A(I, 2)
S3      A(I, 2) = T
     ENDDO
```

在此代码中，从S_0到S_1，S_2，S_3中的每一个有一依赖。因此，一个基于依赖的编译器应禁止交换循环L_1与语句块$\{S_1, S_2, S_3\}$，即使此交换在数组A中保留相同的值。为了看清这一点，注意A(I,1)和A(I,2)都接受等同的更改。所以更改发生在互换前或互换后都不要紧。

我们从定理2.2能立刻得到这样的结论：在一个无循环程序中，如果在两个语句之间不存在依赖，那么这两个语句能并行地执行。这是因为，没有依赖意味着两个语句的相对顺序对于用输出表示的程序含义来说是不重要的。不过此说法不是非常有用的，因为无循环的程序没有多少有趣的计算。为了将此概念扩展到循环上，需要引入一些概念帮助我们理解有关循环嵌套中的语句实例。

2.2.4 距离向量和方向向量

在包含涉及依赖语句的循环嵌套迭代空间中，用依赖的源点和汇点之间的距离来刻划依赖是方便的。我们利用*距离向量*和*方向向量*来表达这一点[278]。

定义2.9 假设从循环嵌套迭代i中语句S_1到迭代j中语句S_2有一依赖，则依赖距离向量d(i, j) 定义为长度为n的向量，使得$d(i, j)_k = j_k - i_k$。

在某些情况下，使用距离向量对工作是有益的，距离向量通过依赖跨越循环迭代的数目来表示。为此我们能将一个规范化的距离向量$d_N(i, j)$ 定义为d(i, j)/s，其中$s = \{s_1, s_2, \ldots s_n\}$，为循环步长向量。为本章讨论之用，我们将假设所有距离向量是规范化的。这隐含着：对一循环嵌套内依赖涉及到的语句实例，给定两个迭代号i和j，$i<j$当且仅当$d(i, j)>0$。

45

距离向量导致方向向量定义如下：

定义2.10 假设从n层循环嵌套的迭代i中语句S_1到迭代j中语句S_2有一依赖；那么依赖方向向量D(i, j) 定义为长度为n的向量，使得

$$D(i, j)_k = \begin{cases} \text{“}<\text{”}, & \text{如果} d(i, j)_k > 0 \\ \text{“}=\text{”}, & \text{如果} d(i, j)_k = 0 \\ \text{“}>\text{”}, & \text{如果} d(i, j)_k < 0 \end{cases}$$

记住方向向量中项的一种方便的方法是将“<”和“>”看成是箭头。利用这种方法，箭头总是指向依赖的源点和汇点的迭代向量偶中首次出现的循环迭代。

依赖的方向向量与在依赖的源点上的迭代向量到在依赖的汇点上的迭代向量有关。例如，

在循环

```
      DO I = 1, N
        DO J = 1, M
          DO K = 1, L
S1          A(I + 1, J, K-1) = A(I, J, K) + 10
          ENDDO
        ENDDO
      ENDDO
```

中，语句S_1有一个到它自身的真依赖，具有方向向量(<,=,>)，意味着在依赖的源点上最外层循环索引小于依赖汇点的索引，在源点和汇点上居中的循环索引相等，而在源点上最内层循环的索引大于汇点的索引。注意，如果一个方向向量的最左非“=”分量不是“<”，那么依赖不存在，因为那样意味着依赖的汇点出现在源点之前，而这是不可能的。

将距离向量、方向向量和差向量限制在公共循环中的理由是这些循环仅影响到语句的相对执行顺序。如果两个语句实例包含在n个公共循环中，对这些语句来说，两个迭代向量的前n个分量是相等的，于是它们的相对执行顺序由原文的位置确定，而不管向量余下的任何成分。

方向向量与变换

46 方向向量能用作理解循环重排序变换的基础，因为它们概括了依赖的源点和汇点上索引向量之间的关系。第5章将说明在受到变换影响的循环嵌套中如何判断依赖方向向量上作各种变换的效果。下面的定理解释方向向量如何能用来测试变换的合法性。

定理2.3 *方向向量变换* 令T是一个应用到循环嵌套上的变换，它不重排循环体中的语句。在应用变换后，如果循环嵌套中源点和汇点的依赖方向向量中没有一个最左非“=”分量是“>”，那么此变换是有效的。

证明 直接从定理2.2得到此定理，因为所有的依赖仍然存在，并且没有一个依赖被倒转。

定理2.3的主要效果是：如果能说明循环嵌套的变换如何影响到循环中依赖的方向向量，我们就能用此定理判断何时此变换是合法的。在第5章中将用它来确立循环交换和其他几个变换的正确性。

依赖个数

在处理依赖时经常发生的一个问题是：循环嵌套中给定的一对语句之间有多少依赖？从技术上讲，我们必须说从依赖原点的每个语句实例有一个到相同循环嵌套中另一个语句实例的依赖。因此，在下面的循环中，对每个迭代向量(i, j)，$1 \leqslant i \leqslant 9$和$1 \leqslant j \leqslant 10$，有一个从语句S到自身的依赖。

```
      DO J = 1, 10
        DO I = 1, 10
S         A(I + 1, J) = A(I, J) + X
        ENDDO
      ENDDO
```

这是因为语句S在索引值I = i和J = j时，在数组A中创建了一个值，当索引I具有值i + 1和J具有j时，在语句S中使用该值。在I循环的最后一次迭代中，语句实例上没有依赖发生，因为没有后继迭代消费此值。作为此分析的结果，我们看到在此循环中有90个不同的依赖。

实际上，没有一个编译器有能力为每一个嵌套循环中的每个语句保留这么多依赖。因此我们必须寻找能概括依赖的方法。例如，在外层循环的不同迭代中发生的依赖之间没有差别，只要它至少在一次迭代中发生过。如果将这些迭代的所有依赖组合在一起，我们能将依赖数目降至9。其次注意所有这些依赖有大量的对称性：每次迭代中生产的值在下一次迭代中消费，所以，所有的依赖是真依赖或流依赖，并且所有的依赖距离是1。这提示我们为每一种不同的依赖类型仅保留不同的距离向量，就能减少依赖数目。然而，即使这样，这种简化还会给我们留下过多的依赖，如下面的循环嵌套例示：

47

```
      DO J = 1, 10
        DO I = 1, 99
S1        A(I, J) = B(I, J) + X
S2        C(I, J) = A(100-I, J) + Y
        ENDDO
      ENDDO
```

当I=1时，此循环在语句S_1存入A(1,J)。随后在内循环I=99的最后一次迭代中使用同一单元。因此存在一个从S_1到S_2的依赖，距离为98。在I=2时，存入A(2,J)，在I=98时使用，产生一个距离96的依赖。继续用这种方法，当I=50时迭代中的距离降至0。

在下一次迭代中，I=51，且距离变为负2。然而，我们知道合法的依赖不能有一个负的距离，因为这将表明依赖的源点是在汇点之前执行。这意味着I大于50的迭代中依赖的源点和汇点被倒转了；即它们是从语句S_2到语句S_1的反依赖。因此，此循环中有反依赖，直至所有距离升至98（包含98）。从S_1到S_2的真依赖总共有50个不同的距离，而从S_2到S_1的反依赖有49个不同的距离。再一次表明，这是太多了。

为了进一步减少编译器必须记住的不同依赖的数目，本书将采用惯常做法：给定的一对引用之间的依赖的数目等于不同的方向向量的数目，它是对这些引用之间所有依赖类型的概括。这是充分的，因为仅用方向向量的知识能处理那些将被应用的多数变换。为了讨论需要距离的变换，我们仅保留从一个迭代到另一个迭代不改变距离情况下的距离。在上面的最后例子中，从S_1到S_2的真依赖仅有两个方向向量(=,<)和(=,=)。从S_2到S_1的反依赖有惟一的方向向量(=,<)。因此，根据我们的约定，在此循环嵌套中总共有三个不同的依赖。

48

2.2.5 循环携带依赖和循环无关依赖

迄今描述的数据依赖理论，对于语句S_2数据依赖于语句S_1强加了两个必须满足的要求：

1. 必须存在一条可能的执行路径，使得S_1和S_2两者引用相同的存储单元M。
2. S_1引用M的执行发生在S_2引用M的执行之前。

为使S_2依赖于S_1，S_1的某次执行必须引用一个存储单元（如果考虑的是真依赖，引用被认为是存数），而随后由S_2的一次执行引用（对真依赖是使用）。出现这种模式有两种可能的方式：

1. 在循环的一次迭代中，S_1能引用公共单元；在随后的迭代中S_2能引用同一单元。
2. 在相同的循环迭代中，S_1和S_2两者能引用公共单元，但是在循环迭代的执行过程中S_1居于S_2之前。

第一种情况是*循环携带依赖*的例子，因为此依赖仅当循环在迭代时存在。第二种情况是一个*循环无关依赖*的例子，依赖存在是由于循环内代码位置的关系。下面几小节详述这些依赖类型。

循环携带依赖

发生循环携带依赖是因为循环迭代的缘故。下面的Fortran程序段说明此概念:

```
    DO I = 1, N
S1      A(I + 1) = F(I)
S2      F(I + 1) = A(I)
    ENDDO
```

除第一次迭代外，在I循环的每一次迭代中，S_2使用在前一次迭代中由S_1计算的A的值；因此语句S_2有一个对S_1的真依赖。同样地，语句S_1使用在前一次迭代中语句S_2计算的F值（第一次除外），因此真依赖于语句S_2。这两个依赖都是由循环携带的。如果选取循环的任何特定迭
49 代去单独执行，没有依赖存在。

定义2.11 语句S_2对语句S_1有一个*循环携带依赖*，当且仅当S_1在迭代i中引用单元M，而S_2在迭代j中引用M，且d(i, j)>0（即D(i, j) 包含一个“<”作为它的最左非“=”分量）。

在d(i, j) 中非0分量的出现，保证相应的循环迭代至少有一次是在两个公共引用之间——因此命名为“循环携带依赖”。这将有助于根据循环体中涉及到的语句相对顺序分出携带依赖的类别。

定义2.12 从语句S_1到语句S_2的循环携带依赖被称为*后向的*，如果在循环体中S_2出现在S_1前，或者S_1和S_2是相同的语句。携带依赖被称为*前向的*，如果在循环体中S_2出现在S_1之后。

循环携带依赖的一个重要性质是依赖的层。

定义2.13 循环携带依赖的*层*是此依赖的D(i, j) 的最左非“=”的索引。

换言之，依赖的层是最外层循环索引的嵌套层，该索引在依赖源点和汇点之间变化，其中最外层循环是在嵌套层1。在前面的例子中，所有依赖的层是1，因为对每个依赖，D(i, j) 是(<)。在下面的循环中依赖层是3:

```
    DO I = 1, 10
      DO J = 1, 10
        DO K = 1, 10
S1        A(I, J, K + 1) = A(I, J, K)
        ENDDO
      ENDDO
    ENDDO
```

这是因为D(i, j) 是(=,=,<)。注意，在这种情况下，此依赖实际上是依赖集，每个依赖对应集合{1:10,1:10,1:9}中的一个迭代向量。以后，我们将在每一对不同下标的数组引用之间，联系一个独立的依赖。我们将把单个引用偶的所有依赖看成是一个依赖，但是此依赖可
50 以有许多方向向量。

由于许多理由，依赖层是有用的概念。理由之一是：层非常方便地概括了依赖。例如，上面的程序段含有900个依赖。因为每个迭代向量偶（i, j）引发的依赖具有d(i, j) = (0,0,1)，依赖层方便地用单一性质刻划所有的依赖。

我们也能用依赖层来帮助我们选择把哪些变换应用到程序中，以及阻止应用哪些变换。有时我们可以决定阻止一个特定类型的变换，因为这样做保证我们希望做的其他变换的有效性。

定义2.14 如果排除那些不能维持依赖的变换，则称依赖得到满足。

为了看到此定义如何有用，考察这样的情况：我们希望满足由一特定循环层携带的所有依赖。

定理2.4 任何重排序变换如果满足，(1) 维持k层循环的迭代顺序，(2) 在小于k的层上，不与k层循环内的任一位置交换循环，(3) 在大于k的层上，不与k层循环外的任何位置交换循环，那么这样的变换维持所有k层依赖。

证明 任何k层依赖的方向向量D(i, j) 的第k项必须是它的最左“<”。这意味着在位置1到$k-1$中，方向都是“=”。因此k层依赖源点和汇点是在循环1到$k-1$的相同迭代中。因此，这些循环的任何迭代重排序都不能改变此依赖的意思，因为根据假设在1到$k-1$层上将保持这些循环。另外，原在k层循环内的循环没有一个循环能变为依赖之一的携带者，因为那将需要将它交换到该循环的外边，按照假设这是被排除的。

因为保持在k层迭代的顺序，D(i, j) 第k个位置上的方向将保持“<”。因此，依赖必定得到保持。

定理2.4告诉我们：由于拒绝重排k层循环的迭代顺序，我们能满足任何k层的依赖。能用它来确立某些强有力的变换的有效性。例如，程序段

```
    DO I = 1, 10
S1    A(I + 1) = F(I)
S2    F(I + 1) = A(I)
    ENDDO
```

51

等价于

```
    DO I = 1, 10
S2    F(I + 1) = A(I)
S1    A(I + 1) = F(I)
    ENDDO
```

因为在1层所有的依赖是循环携带的，因此我们保持1层循环的迭代顺序。对循环携带的依赖来说，语句顺序是没有关系的。

定理2.4还确立：任何变换是有效的，如果它们是在最深的依赖层实施的。例如，在下面的代码段中，我们一旦决定按原顺序串行执行外层循环的话，那么我们能实施循环重排序和逆转内层两个循环。

```
    DO I = 1, 10
      DO J = 1, 10
        DO K = 1, 10
S         A(I + 1, J + 2, K + 3) = A(I, J, K) + B
        ENDDO
      ENDDO
    ENDDO
```

在此例中仅在1层上是携带的依赖。所以此代码等价于

```
      DO I = 1, 10
        DO K = 10, 1, -1
          DO J = 1, 10
S           A(I + 1, J + 2, K + 3) = A(I, J, K) + B
          ENDDO
        ENDDO
      ENDDO
```

这是通过交换J和K循环并逆转K循环获得的代码，因为我们维持1层I循环的迭代顺序。

已经了解依赖层的重要性，我们将使用特殊的记法（依赖符号的下标）来表示它。就是说，S_1和S_2之间的l层依赖将用$S_1\delta_l S_2$表示。

循环无关依赖

52 与循环携带依赖相反，循环无关依赖是由相对语句位置引起的。因此，循环无关依赖决定循环嵌套中代码执行的顺序，而循环携带依赖决定循环必须迭代的顺序。

定义2.15 语句S_2有一个对语句S_1的*循环无关依赖*，当且仅当存在两个迭代向量i和j，使得（1）S_1在迭代i中引用存储单元M，S_2在迭代j中引用M，且i = j；并且（2）迭代中有一条从S_1到S_2的控制流路径。

直观上看，定义2.15陈述这样的事实：在两个语句的公共循环的单次迭代中，如果两个语句引用相同存储单元，则存在一个循环无关依赖。一个非常明显的例子是

```
      DO I = 1, 10
S1      A(I) = ...
S2        ... = A(I)
      ENDDO
```

在I循环的每一迭代中，语句S_2使用的值，恰好是由语句S_1计算的，因此建立了循环无关依赖。

一个不太明显的例子是

```
      DO I = 1, 9
S1      A(I) = ...
S2        ... = A(10-I)
      ENDDO
```

在循环的第5次迭代中，语句S_1存入A(5)，而语句S_2从A(5)中取出。该依赖是循环无关的。此代码段中所有其他的依赖是由循环携带的。用下面的例子说明定义中出现分离的迭代向量i和j的理由：

```
      DO I = 1, 10
S1      A(I) = ...
      ENDDO
      DO I = 1, 10
S2      ... = A(20-I)
      ENDDO
```

语句S_2使用A(10)的值是由语句S_1在第一个循环的第10次迭代中计算的，建立一个循环无关依赖。公共循环对循环无关依赖不是必要的，因为它们是由语句位置引起的。

53 注意，如果我们维持循环无关依赖涉及的语句顺序，那么我们也就保证那些依赖得到满足。

定理2.5 如果从S_1到S_2存在循环无关依赖，任何不在迭代之间移动语句实例并且维持循环体中S_1和S_2相对顺序的重排序变换，将维持该依赖。

证明 根据定义，S_1和S_2在迭代向量i和j中引用单元M，使得$i = j$，且S_2在S_1之后。一个重排序变换将i映射到i'，j映射到j'，必须有$i' = j'$。因为没有一个语句能移出它的原循环迭代，并因为S_2在S_1之后，所以循环无关依赖的判别标准仍然符合。

为了看清为什么我们需要禁止语句迭代的移动，注意代码

```
    DO I = 1, N
S1      A(I) = B(I) + C
S2      D(I) = A(I) + E
    ENDDO
```

可以变换成为

```
    D(1) = A(1) + E
    DO I = 2, N
S1      A(I-1) = B(I-1) + C
S2      D(I) = A(I) + E
    ENDDO
    A(N) = B(N) + C
```

这仍然是一种重排序变换，因为语句S_1和S_2的所有实例被执行。另外，此变换维持这两个语句在循环体内的顺序。然而，由于将语句实例移动到循环外面，它把循环无关真依赖转换成后向携带的反依赖，使变换无效。

已知定理2.5中确立的循环无关依赖的性质，循环携带依赖中使用的层表示法的自然扩展是用无穷层来表示循环无关依赖（即S_2依赖于S_1的循环无关依赖，表示为$S_1\delta_\infty S_2$）。此层表示没有循环排序能维持依赖。注意，循环无关依赖的方向向量的全部项都是“=”。

定理2.4和2.5清楚地说明循环携带和循环无关依赖是如何互补的。只要肯定循环按原来顺序迭代，那么循环携带依赖就得到满足，而不管在一特定迭代中的语句顺序。只要维持语句 54
顺序，一个循环无关依赖就得到满足，而不管循环迭代顺序。

循环无关依赖和循环携带依赖划分所有可能的数据依赖。为了看清这一点只需注意依赖$S_1\delta S_2$的存在需要S_1在S_2之前执行。这只能在两个实例中发生：

1. 当依赖距离向量大于0时，或者
2. 当依赖距离向量等于0且在原文上S_1出现在S_2之前。

它们正好分别是循环携带依赖和循环无关依赖的准则。如果不是这两种情况，且S_1和S_2引用一公共存储元素，那么S_2是在S_1之前执行，而依赖实际上是$S_2\delta S_1$。

迭代重排序

我们用有关迭代重排序的合法性的一个结果结束这一节。

定理2.6 *迭代重排序* 一个变换重排k层循环迭代的顺序，此外不做任何其他的改变，如果该循环不携带依赖，那么它是有效的。

证明 假设k层循环不携带依赖，但是其迭代的某种顺序不能维持原程序中的依赖。因此在原程序中必存在一个被此变换倒转了的依赖。根据定理2.5，它不能是循环无关依赖。所以它必是一个携带的依赖。有两种情况：

1. 依赖是由讨论的循环之外的一个循环携带的。因此，它的层必须是$k-1$或更低的层。因为在k层重排迭代不影响在1到$k-1$层上的循环，根据定理2.4必定维持在这些层上任何携带的依赖。因此，此依赖不能由一个外循环携带。

2. 依赖是由讨论的循环之内的一个循环携带的。因为依赖的方向向量在第k个位置必须有“=”，重排循环不能改变此方向。所以方向向量在最左位置上必须仍有“<”，根据定理2.3此变换是有效的。

55 由于矛盾，定理得到证明。

2.3 简单的依赖测试

依赖测试方法将在第3章进行广泛的讨论。然而，为理解基于依赖的变换是如何工作的，有必要从直观上看依赖是怎样计算的。这一节我们对依赖的计算作一个直观的介绍。

我们用更具体的术语来说明循环依赖的一般条件，由此开始我们的介绍。

定理2.7 令α和β是下面循环嵌套的迭代空间内的迭代向量:

```
      DO i1 = L1, U1, S1
        DO i2 = L2, U2, S2
          ...
          DO in = Ln, Un, Sn
S1          A(f1(i1, ..., in), ..., fm(i1, ..., in)) = ...
S2          ... = A(g1(i1, ..., in), ..., gm(i1, ..., in))
          ENDDO
          ...
        ENDDO
      ENDDO
```

从S_1到S_2存在依赖，当且仅当存在α和β的值，使得（1）α按词典顺序小于或等于β，（2）下面的依赖方程组得到满足:

$$f_i(\alpha) = g_i(\beta) \text{ 对于所有的} i, 1 \leqslant i \leqslant m \tag{2-2}$$

证明 这是定理2.1的直接应用。如果$\alpha < \beta$，则定理2.1中的条件（1）成立。否则，如果$\alpha = \beta$，则条件（1）成立，因为显然在循环嵌套中有一条从S_1到S_2的路径。因此定理2.7中条件（1）等价于定理2.1中的条件（1）。

方程（2-2）意味着在迭代α和β中数组A的索引值是相等的，所以这些方程等价于定理2.1的条件（2）。因为S_1是写入，定理2.1的条件（3）也成立，所以此定理必适用，希望的结果得到证明。

尽管定理2.7能很容易推广，我们仍然暂时假设潜在依赖的源点和汇点正好包含在相同的循环中。

56 为了研究依赖背后的直观感觉，我们介绍一种简单的Δ记法，它是3.4.1节中讨论的Delta测试的基础。在某些简单情况下,这种记法有助于直观上更容易形象化简单情形中的依赖。

回顾一下前一节，一个依赖的存在，只要潜在的依赖源点在迭代i中访问一个存储单元M，那么随后在迭代j的某些迭代中，依赖汇点将访问同一单元M。已知绝大多数下标是简单的，每个下标项具有单循环索引加或减一个常数，理解依赖的一种简易方法是将潜在的依赖源点和依赖汇点看成是相互之间的距离为Δ。即假设源点在迭代I_0中访问M，而汇点在随后的

$I_0+\Delta I$中访问M，这里ΔI是两次访问之间的迭代数。对这种多数的简单下标形式，利用此记法提供一种计算依赖的快捷、直观的方法。

作为一个示例，考察下面的循环：

```
      DO I = 1, N
S       A(I + 1) = A(I) + B
      ENDDO
```

为了测试真依赖，假设在迭代I_0中语句S左端访问M，而在随后的ΔI迭代中语句右端访问同一单元。那么$A(I_0+1)$和$A(I_0+\Delta I)$二者都必引用存储单元M。为此我们必须有

$$I_0 + 1 = I_0 + \Delta I$$

化简后，产生$\Delta I=1$。因为1大于0（表示它的确在随后访问），并且小于上界（所以它确实被执行），这里有一个真依赖，具有距离1和方向向量（<）。应用同一方法去测试一个反依赖产生$\Delta I=-1$，说明没有依赖。因为下标形式是简单的，对此循环的每次迭代，Δ都是相同的，因此依赖距离是一致的。

在具有多个下标的情况下，如代码段

```
DO I = 1, 100
   DO J = 1, 100
      DO K = 1, 100
         A(I + 1, J, K) = A(I, J, K + 1) + B
      ENDDO
   ENDDO
ENDDO
```

我们得到下面的索引表达式方程：

$$I_0 + 1 = I_0 + \Delta I;\ J_0 = J_0 + \Delta J;\ K_0 = K_0 + \Delta K + 1;$$ 57

这些方程的解是

$$\Delta I = 1;\ \Delta J = 0;\ \Delta K = -1$$

并且相应的方向向量是(<,=,>)。

对于稍微复杂的例子，考察下面的代码段：

```
DO I = 1, 100
   DO J = 1, 100
      A(I + 1, J) = A(I, 5) + B
   ENDDO
ENDDO
```

解由此循环导出的方程组，给出

$$\Delta I = 1,\ J_0 = 5$$

因为I是无约束的，第一个结果告诉我们I维中对I的所有值依赖距离是1。另一方面，第二个结果告诉我们在依赖源点上J_0是常数5，因为依赖距离是无约束的，我们必须设想任何距离是在-4和95（循环界允许的最小值和最大值）之间实现的。

此例中的一个次要结果是任何时刻循环索引不在依赖源点或汇点的任何下标中出现，它的距离是无约束的；就是说，它能取任何合法的距离。特别，对应此索引的方向是“*”，表示三种方向的联合。因此，在循环

```
DO I = 1, 100
   DO J = 1, 100
      A(I + 1) = A(I) + B(J)
   ENDDO
ENDDO
```

中，由I循环携带的依赖方向向量是（<，*）。

作为最后的注释，可能发生这样的情况：如果假设一个引用是源点而另一引用是汇点，则测试可能导致在最左方向向量位置上出现“>”。这并不意味着依赖是非法的，只不过表示在反方向上有一个相反类型的依赖。例如，如果前面的代码段以J循环

```
DO J = 1, 100
   DO I = 1, 100
      A(I + 1) = A(I) + B(J)
   ENDDO
ENDDO
```

58

作为最外层循环，测试将产生方向向量(*,<)或{(<,<),(=,<),(>,<)}。第一个方向对应于1层上的真依赖。第二个方向向量对应2层上的真依赖，第三个方向对应1层上具有方向向量(<,>)的反依赖。它的存在是因为当J=1和I=2时，语句读出A(2)，当J=2和I=1时，语句存入A(2)。

2.4 并行化和向量化

我们以将依赖应用于自动并行化和向量化的讨论来结束这一章。虽然向量化可以看成是并行化的特殊情况，我们将把对它的处理放在第二位，因为它涉及到另外一个循环分布变换。

2.4.1 并行化

并行化一个循环的标准方法是将每一个独立的迭代转换为一个并行线程，并且异步地运行这些线程。在某种意义上，这是一个重排序变换，其中迭代的原有顺序能转换成一个不确定的顺序。此外，因为循环体不是一个原子操作，两次不同迭代中的语句能同时运行。根据定理2.2，如果它不逆转任何依赖的含义，则并行化是有效的。能保证这一点的惟一方法是用简短的显式同步，禁止我们所希望的并行运行的迭代之间的任何依赖。

定理2.8 *循环并行化* 如果循环不携带依赖，则转换串行循环为并行循环是有效的。

证明 变换被称为有效的，是指变换后的程序对任何执行调度必须是有效的。从迭代重排序定理（定理2.6）我们知道，如果循环不携带依赖，那么将迭代安排成任何顺序是有效的。然而，在并行情况下，交叉并行迭代中的个别语句是可能的。为

59

证明此定理，我们必须证明不会引起依赖逆转。

假设在并行循环的某个调度中依赖被逆转。这意味着在该调度中依赖的汇点出现在依赖的源点之前。显然，在此并行循环内的一层上，依赖不能是循环无关的或循环携带的，因为在那种情况下，源点和汇点同在单一迭代中出现。因为迭代是用单线程执行的，其中所有语句实例的顺序得以保持。所以，依赖必定是由并行循环之外的一个循环携带的。但根据定理2.4，此依赖是由任何重排序变换维持的，它不影响携带者循环的顺序。因此定理得到证明。

2.4.2 向量化

回顾1.3.2节，向量化的任务是判断内层循环中的语句是否能直接用Fortran 90重写它们而被向量化。前一节的定理2.8告诉我们，不携带依赖的任何单语句循环能直接向量化，因为该循环能并行运行。因此循环

```
DO I = 1, N
   X(I) = X(I) + C
ENDDO
```

能安全地重写成

```
X(1:N) = X(1:N) + C
```

另一方面，在

```
DO I = 1, N
   X(I + 1) = X(I) + C
ENDDO
```

中，携带有一个依赖，变换成语句

```
X(2:N + 1) = X(1:N) + C
```

是不正确的。在每次迭代中，串行版本使用X的值是在前一次迭代中计算出来的，而Fortran 90语句使用的只是X的原有值。

已知循环并行化定理（定理2.8），一个自然要问的问题是：循环中携带依赖的任何语句能直接向量化吗？这个问题受到了下面例子的启发：

60

```
      DO I = 1, N
S1      A(I + 1) = B(I) + C
S2      D(I) = A(I) + E
      ENDDO
```

此循环携带依赖（$S_1\delta S_2$），因为在一次迭代中存入A中的值，而在下一次迭代中从A中取出。

然而，将此循环直接转换成Fortran 90具有与下面串行循环相同的含义。

```
S1   A(2:N + 1) = B(1:N) + C
S2   D(1:N) = A(1:N) + E
```

第二个语句的并行版本使用由前一循环定义的元素A(2:N)，正如在串行循环中所为。这种显而易见的矛盾能用这样的事实来解释：向量化过程加入一种额外的变换，称为*循环分布*。它的效果宛如首先将循环转换成两个不同的循环

```
      DO I = 1, N
S1      A(I + 1) = B(I) + C
      ENDDO
      DO I = 1, N
S2      D(I) = A(I) + E
      ENDDO
```

其中每个循环能直接向量化。在这种情况下，携带的依赖是前向的，但是，即使携带的依赖是后向的，向量化也能进行，如下例所示：

```
      DO I = 1, N
S2      D(I) = A(I) + E
S1      A(I + 1) = B(I) + C
      ENDDO
```

显然，如果交换循环体中的语句，此循环能向量化，因为它变成了等同于前面的情况。此交换是合法的，因为语句之间没有循环无关依赖。

另一方面，如果在语句之间有一个后向携带依赖和一个循环无关依赖，那么不能对它们向量化，因为上述交换是非法的:

```
      DO I = 1, N
S1      B(I) = A(I) + E
S2      A(I + 1) = B(I) + C
      ENDDO
```

61

这里不能用循环分布消除后向循环携带依赖，因为涉及到数组B的循环无关依赖，我们不能交换循环体中的语句。这在直观上与我们对向量化的理解是一致的。为了向量化一个语句，我们必须有可能去分布围绕它的循环。为此在进入分布之前，我们必须有可能计算分布循环的任何迭代的所有输入。依赖环使得这一点成为不可能的。

在下面的定理中，将这些观察得到的结果形式化，确立了循环中哪些语句可以向量化的一般条件。

定理2.9 *循环向量化* 至少包含在一个循环中的一个语句，如果不包含在任何依赖环中，则能直接用Fortran 90重写使之向量化。

证明 假设语句不包含在任何环中。则图2-1给出的算法将正确地向量化该语句。

回顾早先的讨论，Fortran 77 DO循环中的一个语句和与它类似的Fortran 90向量，其间的关键语义差别在于: 向量版本必须在存入任何输出之前，取出所有的输入，而DO循环可能相互混合取出和存入。因此，如果包含在一个循环中的语句在循环的开始处具有所有有效的输入，那么此语句能被正确地重写成Fortran 90的一个数组赋值。

图2-1中的算法将同一循环中的语句组成整体有序的语句组，组中的每个语句或是依赖图中环（也称依赖环）的一部分，或者不是环的一部分的单个语句。这种重排序等价于围绕依赖环分布循环和向量化不是任何环的一部分的语句。这种分布是正确的，因为没有从较晚的环到较早的环的后向边（拓扑排序保证了这一点），所以在每个分布的循环开始处，所有必需的输入将是可使用的。这种向量化是合法的，因为在一组开始执行前，该组外面的一个组需要的所有输入值是可使用的，给出的排序与依赖一致。结果是由此算法实施的向量化是正确的。

```
Procedure vectorize (L, D)
  // L是包含该语句的最大循环嵌套。
  // D是L中语句的依赖图。
  在依赖图D中求受囿于L的最大强连通区域的集合{S1, S2, ..., Sm}（使用Tarjan强连通区域算法[256]）;
  通过约简每个Si成为单结点和计算Dπ来构造Lπ，由D自然地从依赖图导出Lπ;
  令{π1, π2, ..., πm}是Lπ的m个结点，其编号顺序与Dπ一致（使用拓扑排序生成编号）;
  for i = 1 to m do begin
    if πi 是一个依赖环then
      生成围绕πi中语句的DO循环;
    else
      用Fortran 90 直接重写此单个语句，向量化包含它的相应的每个循环;
  end
end
```

图2-1 简单向量化

定理2.9确立可向量化的充分条件，但它比需要的条件强得多，如我们将在下一节见到的那样。

2.4.3 一个先进的向量化算法

图2-1中简单向量化算法的问题是失去许多向量化的机会。考察下面的简单例子：

```
DO I = 1, N
   DO J = 1, M
S     A(I + 1, J) = A(I, J) + B
   ENDDO
ENDDO
```

如果我们使用2.3节的直观方法构造依赖图，就会看到有一个从S到自身的依赖，具有距离向量(1,0)和方向向量(<,=)。因此，语句S包含在一个依赖环中，所以简单算法将不对它向量化。

另一方面，在此循环嵌套中自依赖是由1层上携带的。定理2.4告诉我们：由于保证1层循环的迭代顺序不变，所以能保证依赖得到维持；就是说，串行运行外层循环就足以保证依赖得到维持。这就提示我们：一旦我们保证依赖将得到满足，就能向量化内层循环，产生

62
↓
63

```
    DO I = 1, N
S      A(I + 1, 1:M) = A(I, 1:M) + B
    ENDDO
```

一般说来，定理2.4告诉我们：通过顺序地运行包含k层循环在内的所有外层循环就能让k层依赖得到满足。因此，即使一个语句是在一依赖环中，我们也许能通过串行运行某些循环对它向量化。

这些观察的结果提示一种解决多维向量化问题的递归方法：首先，尝试在最外层循环生成向量代码。如果依赖妨碍这么做，则串行运行外循环，从而满足由此循环携带的依赖，并尝试更深一层，忽略由外层携带的依赖。这个方法是在图2.2[168，21]介绍的*codegen*过程中详细说明。

```
Procedure codegen (R, k, D)
  // R是我们必须为之生成代码的区域。
  // k是合理的并行循环的最小嵌套层。
  // D是R中语句之间的依赖图。
  在依赖图D中求受困于R的最大强连通区域的集合{S1, S2, ..., Sm}(使用Tarjan算法);
  通过约简每个Si成为单结点和计算Dπ来构造Rπ，由D自然地从依赖图导出Rπ;
  令{π1, π2, ..., πm}是Rπ的m个结点，其编号顺序与Dπ一致（使用拓扑排序生成编号）;
  for i = 1 to m do begin
    if πi是一个依赖环then begin
      生成k层DO语句;
      令Di是D中所有依赖边组成的依赖图，它们是在k + 1层或更高层上，并且是
      属于πi内部的;
      codegen(πi, k + 1, Di);
      生成k层ENDDO语句;
    end
    else
      在ρ(πi) - k + 1维中为πi生成一个向量语句，其中ρ(πi) 是包含πi的循环个数
  end
end codegen
```

图2-2 多层向量代码生成算法

最初在整个程序的1层（最外层）上调用*codegen*。第一步是将程序划成π块（piblock），这里π块是一个强连通区域，如Tarjan算法[256]定义的那样。强连通区域的定义允许有环和无环两种π块；然而，任何无环块都是一些不依赖于自身的单个语句。接下来，根据依赖关系对强连通区域做拓扑排序[185]。最后按序检查每个区域。如果区域是无环的（因此必须由一个语句组成），则在余下的维中生成此语句的并行形式。如果区域是有环的，则为该区域生成1层DO循环，1层的依赖被消除，因此保证它们得到满足，然后在2层和更深的层上对带有依赖集合的区域递归地调用*codegen*。

为了说明*codegen*的能力，考虑它对下面程序段的应用。

```
      DO I = 1, 100
S1      X(I) = Y(I) + 10
        DO J = 1, 100
S2        B(J) = A(J, N)
          DO K = 1, 100
S3          A(J + 1, K) = B(J) + C(J, K)
          ENDDO
S4        Y(I + J) = A(J + 1, N)
        ENDDO
      ENDDO
```

图2-3给出此程序的依赖图。程序包含下列依赖：

- 从S_2到S_3有一个1层依赖和一个循环无关真依赖，这是由于在两个语句中使用B(J)导致的。为了看清这一点，我们看一下2.3节中简单的依赖测试过程。因为索引I不在任何一个下标中出现，对外层循环来说，距离向量是无约束的，并且方向是“*”。另一方面，在依赖源点和汇点上对J的引用导致方程$J_0 = J_0 + \Delta J$。这意味着$\Delta J = 0$，并且J循环的方向是“=”。因此与B(J)相关联的方向向量集是(*,=)或{(<,=),(=,=),(>,=)}。第一个向量告诉我们是1层真依赖，第二个告诉我们是循环无关真依赖。第三个方向对应于逆方向的反依赖，在下一小段讨论。
- 因为在I循环的下一次迭代中S_2存入B(J)之前，S_3使用B(J)，所以有一个由I循环携带的从S_3到S_2的反依赖。
- 由于在S_3中定义A(J + 1,K)和在语句S_4中使用A(J + 1,N)，有一个从S_3到S_4的循环无关真依赖。认定依赖存在是因为缺少关于N的值的其他信息，必须假定N能落在K的范围内（1到100之间）。注意，如果应用我们的依赖测试过程于这一对语句上，以S_3作为依赖源点，那么我们得到的方向集为(*,=)或{(<,=),(=,=),(>,=)}。这意味着，这两个引用之间还有一个由I循环携带的真依赖和一个逆方向的反依赖，它们分别对应第1和第3方向向量。在下一小段讨论反依赖。
- 有一个由I循环携带的从S_4到S_3的反依赖，这是因为在I循环的一次迭代中S_4使用A(J +

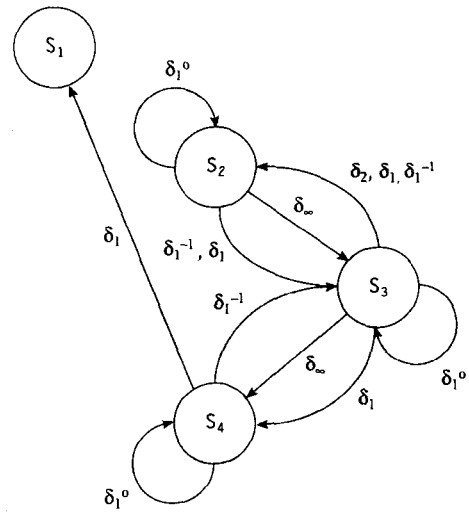


图2-3 例子的依赖图

1,N)；而在该循环的后继的一次迭代中（当J的值正好相同且K=N时），S_3存入同一单元。这对应于前一小节中的第3个方向。

- 有一个由I循环（1层）携带的从S_4到S_1的真依赖，这是因为在一次迭代中S_4能存入Y(I + 1)，而在I循环的下一次迭代中S_1从中读取。使用简单的依赖测试过程，以S_4为依赖源点，产生方程

$$I_0 + J = I_0 + \Delta I$$

它意味着ΔI=J。因为J总是大于0，我们有方向“<”，且依赖为真。

64 ~ 66

- 从S_3到S_2有J循环（2层）携带的真依赖，这是因为J循环的某一次迭代中S_3存入A(J + 1,K)，而在下一次迭代中通过对A(J,N)的访问，从相同的单元中读入。再一次，必须假定N是在1和100之间。此依赖的方向向量是(*,<)，它还引发一个真依赖和一个由I循环携带的逆向反依赖，这在图2-3中用一个额外的标号显示。
- 语句S_2，S_3和S_4都有到它们自身的输出依赖，这是因为它们左端具有的维数比包含它们的循环个数少一。因此，外循环必引起在不同的迭代中存入相同的数组元素。在简单的依赖测试过程中，我们会看到这一点，因为I循环的方向应是“*”。

在最外层调用*codegen*将产生两个π块：一个由S_2，S_3和S_4组成的有环π块和一个由S_1组成的无环π块。其结果将对S_1向量化，但必须放在多语句π块的代码之后，这是由于受到拓扑排序的限制。因此在这一层产生的代码是

```
DO I = 1, 100
   codegen({S2, S3, S4}, 2, D2})
ENDDO
X(1:100) = Y(1:100) + 10
```

为了有效地在2层上调用*codegen*，剥去1层上的所有依赖，留下图2-4中描绘的依赖图。

现在可为S_4生成向量代码，但S_2和S_3仍在一个依赖环中，至此生成的代码是

```
DO I = 1, 100
   DO J = 1, 100
      codegen({S2, S3}, 3, D3})
   ENDDO
   Y(I + 1:I + 100) = A(2:101, N)
ENDDO
X(1:100) = Y(1:100) + 10
```

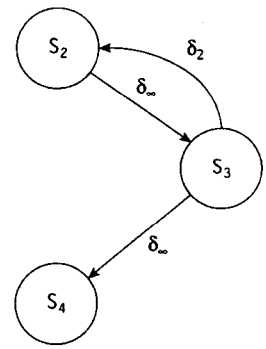


图2-4 在消除1层依赖后{S_2,S_3,S_4}的依赖图

最后调用*codegen*需要消除2层上的依赖，留下的图如图2-5。

留下的两个语句能按向量操作执行。S_2没有用于向量执行的维，所以产生一个简单的标量语句，最后的代码是

```
DO I = 1, 100
   DO J = 1, 100
      B(J) = A(J, N)
      A(J + 1, 1:100) = B(J) + C(J, 1:100)
   ENDDO
   Y(I + 1, I + 100) = A(2:101, N)
ENDDO
X(1:100) = Y(1:100) + 10
```

尽管*codegen*是一个简单而优雅的算法，但在程序中能产生显著的变化。在前面的例子中，

通过算法的直接应用暴露了所有可用的向量化性质，在语句排序方面，需要非常引人注目的改变。

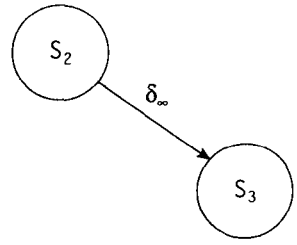


图2-5 在消除2层依赖之后{S_2,S_3}的依赖图

从本章的几个定理中很容易得出过程*codegen*的正确性。如果一个区域是有环的，那么定理2.4通过运行k循环（根据算法的性质，循环1到$k-1$）保证满足任何k层依赖。因此，在then子句中的语句仅忽略已由串行循环得到满足的依赖。通过串行执行DO循环，一旦到达已消除所有的环的某一层，定理2.9保证余下的循环可正确地并行执行。注意，必须到达这样的层（保证终止），因为循环无关依赖是固有无环的，只要串行执行所有的循环（所以生成的代码在0维是并行的）所有循环携带的依赖将最终得到满足。最后，根据定理2.5拓扑排序保证循环无关依赖会得到维持。依赖环外的循环携带依赖能由拓扑排序得到维持也是正确的，而不需要这些语句包含在公共循环中，虽然这一点没有显式地证明。

应当很清楚，当应用*codegen*到一个循环嵌套上，它的有效性受到依赖图的精确性限制。在第3章中，我们提供有关如何构造一个有效的和精确的依赖测试过程的细节。我们在第5章回到算法*codegen*上。

2.5 小结

依赖是编译器使用的主要的工具，用于分析和变换程序，使之在并行机和向量机上执行。如果变换维护了程序中每个依赖的源点和汇点的顺序，则重排程序中语句执行顺序的任何变换维持程序的正确性。可以用它作为有效的工具来确定何时并行化或向量化循环是安全的。

可以用几个不同的属性来刻划依赖。依赖的*类型*——真依赖、反依赖或输出依赖——告诉我们哪一个对应到读前写、写前读或写前写。循环中的依赖具有几个特殊的性质。依赖*方向向量*描述嵌套循环在依赖源点和汇点上循环索引值之间的关系（<,=或>）。*距离向量*给出循环嵌套中每个索引的依赖所跨越的迭代个数。依赖被称为是*循环无关的*，如果它的方向向量项都是“=”。否则它是*循环携带的*。循环携带依赖的*层*是循环的嵌套层，它对应于方向向量中“最左非‘=’”方向。

67～69

这些概念的有用性在一个简单而有效的向量化算法中说明，如图2-2所示。在这一章简要介绍了构造依赖图的方法，第3章将详细讨论这些方法。

2.6 实例研究

原始设计的*codegen*过程处于PFC向量化系统的核心位置。在PL/1中的实现使用一组PL/1预处理器宏指令实现语句集，使其实际的实现与图2-2中的样例代码非常类似。过去几年已将此过程一般化使其包含若干个变换，这些将在第5章介绍。所有添加功能始终保留基本结构。后来IBM VS Fortran Vectorizer[243]和Ardent Titan编译器[17]采纳了*codegen*结构。

PFC的原始实现使用了依赖的一种简化表示，它不包含全部方向向量，仅标识出程序中每个携带依赖的层号，并区分携带依赖与循环无关依赖。在3.8节中对这种表示有更详细的描述。

2.7 历史评述与参考文献

数据依赖及其对向量化和程序变换的应用，是Lamport[195，196]和Kuck，Muraoka和Chen[191，219]最早陈述的。虽然Kuck[189]是第一个对这些概念精确刻划和命名者，但在

Parallelizer系统背景中，Lamport发展了迭代空间、差向量的概念以及类似于方向向量的概念。他也提出类似于真依赖、反依赖和输出依赖的记法。语句依赖是用简单而有效的数据依赖测试判定的。Lamport提出一个向量化算法，它基于这样一个准则：从那些不能向量化的循环中精确地划分出能正确地向量化的循环。此准则称为“真不相容性”，它等价于在2.4.2中讨论的依赖图中环的记法。

在Parafrase系统中，Kuck和他的同事们[191，219]发展一种依赖的记法，比起在Parallelizer中使用的更清楚：数据依赖清晰地分类为真依赖、反依赖和输出依赖，并使用精细的测试去判断出现或不出现依赖。因为对依赖的清晰定义，Parafrase有一个检测向量语句的简单测试：任何语句只要它不依赖于自己，就能按向量执行。其结果，用一个依赖图的传递闭包来划分可向量化和不可向量化语句。Wolfe[283]提出了如本章介绍的方向向量方法，并将它们应用到包括循环交换和向量化在内的许多重排序变换中。

70

Kennedy第一个指出依赖层的重要性，并将Tarjan算法应用到这个问题上；本书中介绍的两个算法归功于他[168]。Allen对重排序变换精确地刻划了某些变换范围，在这个范围内可以应用依赖。基于依赖，许多研究人员已提出基于依赖的特殊变换，包括循环交换[280，19]、循环倾斜[195，281]、并行化[52，25]、结点分裂[190]和循环合并[1，269]。

循环携带依赖和循环无关依赖的特征首先是由Allen和Kennedy[16，21]指明的。基于依赖层的每个类的性质以及逐步求精的向量化算法也是由Allen和Kennedy提出的。

已有众多的作者讨论过依赖测试[21，32，283，285]。非形式化的测试策略来自Goff，Kennedy和Tseng[123]。第3章将更详细地讨论依赖测试。

习题

2.1 使用依赖的简单过程，为下面的嵌套循环构造所有的依赖，并对每一依赖提供（a）方向向量，（b）距离向量，（c）循环层，（d）类型。

```
DO K = 1, 100
   DO J = 1, 100
      DO I = 1, 100
         A(I + 1, J + 2, K + 1) = A(I, J, K + 1) + B
      ENDDO
   ENDDO
ENDDO
```

2.2 为下面的循环构造所有的方向向量，并指明与每个方向向量相联的依赖类型。

```
DO K = 1, 100
   DO J = 1, 100
      DO I = 1, 100
         A(I + 1, J, K) = A(I, J, 5) + B
      ENDDO
   ENDDO
ENDDO
```

2.3 习题2.2中的循环能并行化吗？如果能，给出一个并行版本。

2.4 对图2-3中所示的依赖，构造所有的方向向量。为对应固定距离的每个方向，明示其距离。

71

2.5 考虑下面的循环:

```
      DO K = 1, 100
        DO J = 1, 100
S1        B(1, J, K) = A(1, J-1, K)
          DO I = 1, 100
S2          A(I + 1, J, K) = B(I, 100-J, K) + C
          ENDDO
        ENDDO
      ENDDO
```

语句S_2依赖于语句S_1吗？语句S_1依赖于语句S_2吗？对于每个存在的依赖，给出依赖类型、方向向量以及涉及的数组变量。应用过程*codegen*于此嵌套循环上会产生什么？

2.6 循环反转是一种变换，它将一个给定循环的迭代顺序反转过来。换言之，循环反转变换将具有循环头

```
DO I = L, H
```

的循环变换成具有相同循环体但是循环头为

```
DO I = H, L, -1
```

的循环。陈述并证明循环反转有效性的充分条件。给出一个简单的循环例子，说明你的条件不是必要的（例子违反条件但仍能反转而不改变结果）。

2.7 在下面的嵌套中，对I循环的循环反转有效吗？为什么有效或没有效？

```
DO J = 1, N
  DO I = 1, M
    A(I + 1, J + 1) = A(I, J) + C
  ENDDO
ENDDO
```

2.8 一家著名的并行计算公司的代表喜欢说："如果你要判断一个给定的循环是否能并行化，只要反转它——如果你得到相同的答案，那么就能安全地将它转换成一个并行DO。"这种
72 说法正确吗？为什么正确或不正确？

第3章

依赖测试

3.1 引言

依赖测试是用来判断循环嵌套中在两个相同数组的下标引用之间是否存在依赖的方法。计算数组的数据依赖是复杂的，因为数组包含许多不同的单元。这一章提供各种高度精确依赖测试方法的详细描述。为了阐述的目的，忽略了控制流（循环自身除外）。

回顾第2章，在最一般的情况下，依赖测试的几乎全部原则能用在下面典型循环嵌套中判断从语句S_1到语句S_2是否存在依赖的问题来说明：

```
      DO i1 = L1, U1
        DO i2 = L2, U2
          ...
          DO in = Ln, Un
S1          A(f1(i1,...,in),...,fm(i1,...,in)) =...
S2          ... = A(g1(i1,...,in),...,gm(i1,...,in))
          ENDDO
          ...
        ENDDO
      ENDDO
```

令α和β是上面描述的循环嵌套迭代空间中的迭代向量。因此，α和β是长度为n的向量，它的第k项是一个整数值，该值在嵌套的第k个循环的下界和上界之间。回顾定义2.1，两个语句之间存在数据依赖必须满足两个要求：两个语句必须访问相同的存储单元，并且在相同的访问之间必须存在一条适宜的控制路径。按照这个例子的表示，该定义是说：从S_1到S_2存在依赖，当且仅当存在α和β的值使得按词典顺序α小于或等于β（控制流要求），满足下面的依赖方程组（公共访问要求）：

$$f_i(\alpha) = g_i(\beta), \text{对于所有} i, 1 \leqslant i \leqslant m \tag{3-1}$$

方程（3-1）是基于这样的简单观察：两个数组访问相同的存储单元，当且仅当每一个对应的下标项是等同的[⊖]。否则两个引用是无关的。

依赖测试有两个目标，第一个目标（并且是最想要的结果）是证明给定的一对相同数组变量的下标引用之间不存在依赖。达到这个目标使用的机制是证明在适当的α和β的区域中方程3-1无解。当不能达到此目标时，依赖测试企图用某种方法刻划可能的依赖，通常用距离向量和方向向量的最小的完全集来刻划。自始至终，测试必须是保守的；即不能明确地证明依赖不存在，就必须假设任何可能的依赖存在。

⊖ 如果程序使数组访问超出了说明的数组界，那么这种说法是不正确的。在Fortran中这样的访问是非法的；本书假定只有合法的数组引用。

本章余下的部分讨论自动解方程3-1（证明无解）的方法。

背景和术语

在研究判定方程3-1有解或无解的方法之前，我们需要一种适当的表示法。第2章已经介绍过距离向量、方向向量以及它们对数据依赖的应用。本章自始至终采用这种认知。依赖测试的目标就是构造距离向量和方向向量的完全集，用来表示相同数组变量的任意一对下标引用之间潜在的依赖关系。因为距离向量可以被当成精确的方向向量，故本章余下的部分将始终仅使用方向向量。

索引和下标

74 为了依赖测试的目的，将使用术语*索引*表示围绕两个引用的某循环的索引变量。另外，假设已检测出所有的*辅助归纳变量*，并用循环索引的线性函数替换（更多的细节见第4章）。

术语*下标*将用于指一对数组引用中下标的位置之一。因为依赖测试总是考虑一对数组引用，所以我们总是使用术语下标指一对下标位置。例如，在循环嵌套

```
      DO i
        DO j
          DO k
S1          A(i,j) = A(i,k) + C
          ENDDO
        ENDDO
      ENDDO
```

中对数组A的一对引用索引i出现在第一个下标中，索引j和k出现在第二个下标中。

为简单起见，假设所有循环的步长为1。非单位步长值在需要时立即可以用第4章讨论的方法正规化。

非线性

就其全部原则来说，显然依赖测试是一个不可判定问题。下标值可能是任意表达式，它们的值一直要等到运行时才能知道，使得编译时不可能确定依赖。虽然在实际程序中会出现某些不确定实例（需要保守假设，假定所有可能的依赖存在），但大多数下标是较简单的——通常为归纳变量的多项式——并因此是编译时分析的主体形式。即使是一般的多项式，用当前的数学理论去求方程3-1的解也是太复杂了。所以，我们将做进一步的简化：除非另有说明，将假设数组下标是循环索引变量的线性表达式。就是说，所有的下标表达式的形式为

$$a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + e \tag{3-2}$$

其中i_k是循环在嵌套层k上的索引，所有的a_k，$(1 \leqslant k \leqslant n)$是整常数，$e$是一个表达式，可能包含循环不变的符号表达式。任何不符合这个限制的下标将归类为*非线性下标*类，并将不对它们做测试。这个限制不是很严重的，因为实际上遇到的大多数下标是线性的。可是，由于各种各样的缘由也出现非线性下标。例如，如果在循环的一个下标中出现一个变量，它从
75 一输入介质中读值，那么该下标将是非线性的。另外，多数依赖测试程序将把包含其他下标数组引用的任何下标当成非线性的。因为这样的下标在“不规则的”或“自适应的”数值代码中是很常见的，处理成非线性会导致近似于依赖的过度保守。

保守测试

带有线性的限制，依赖测试等价于求线性丢番图方程组整数解的问题，这是一个NP完全问题。精确地解此问题是非常困难的；因此，多数依赖测试寻找有效的近似解。一般来说，最常见的近似是保守地做测试——保守测试尝试证明依赖方程（方程（3-1））不存在解。它不试图证明依赖实际上的存在。换言之，如果保守测试判断没有依赖存在，那么编译器就能信赖这一结论。然而，可能是这样一种情况：引用是无关的，但是保守测试没有能力去证明它。另一方面，精确测试是这样一种依赖测试：当且仅当依赖存在时检测依赖。注意，保守依赖测试的不精确将决不会导致编译器生成不正确的代码，仅仅是代码较少优化而已。

每当保守依赖测试程序遇到非线性下标时，它将假定下标表达式可等同于任意的距离和方向。就是说，非线性下标不能用来求精它所包含的任何索引上的依赖测试。因此，为了测试的目的，将把非线性下标作为实际上不存在的一个下标处理。其他的下标仍可用来求精可以在依赖中出现的方向集，或者甚至能证明无依赖。

复杂性

复杂性与出现在下标中的索引个数有关——在一个下标位置中出现的不同循环索引越多，依赖测试就变得越复杂。说一个下标是ZIV（零索引变量），如果无论在哪一个引用中，该下标位置不包含索引。说一个下标是SIV（单索引变量），如果在该位置上仅有一个索引。任何有多于一个索引的下标被说成是MIV（多索引变量）。例如，考虑下面的循环：

```
      DO i
        DO j
          DO k
S1          A(5,i + 1,j) = A(N,i,k) + C
          ENDDO
        ENDDO
      ENDDO
```

76

当对此样本段中对A的两个引用之间测试一个真依赖时，第一个下标是ZIV，第二个下标是SIV，第三个是MIV。再一次请记住约定，在此上下文中“下标”是指一对相应的下标。

可分性

为了依赖测试的目的，可分性描述一个给定的下标是否与其他下标相互影响。当测试多维数组时，说一个下标位置是可分的，如果它的索引不在其他下标中出现[16，51]。如果两个不同的下标包含相同的索引，它们是耦合的。例如，在循环

```
      DO i
        DO j
          DO k
S1          A(i,j,j) = A(i,j,k) + C
          ENDDO
        ENDDO
      ENDDO
```

中，第一个下标是可分的，但第二个和第三个是耦合的，因为它们两个都包含索引j。ZIV下标是可分的，因为它们不含循环索引。

术语“可分的”来源于线性代数和差分方程，将它应用到带有不同变量的方程组中，这

样的方程组能独立地求解。然后这些独立的解能合并在一起，形成一个精确的解集合。

这种独立的性质对可分的下标也成立。如果一个下标的位置是可分的，那么它所含的循环索引不在其他下标中出现。所以只要检查该下标位置，就能独立地测试对应这些索引的方向集合。得到的方向向量能进行以位置为基础的完全精确的合并。例如，在循环嵌套

```
      DO i
        DO j
          DO k
S1          A(i+1,j,k-1) = A(i,j,k) + C
          ENDDO
        ENDDO
      ENDDO
```

77 中，通过测试第一个下标确定方向向量中最左方向，通过测试第二个下标确定中间方向，通过测试第三个下标确定最右方向。得到的结果方向向量(<,=,>)是精确的。应用相同的方法到距离上，使我们计算得到精确的距离向量(1,0,-1)。

另一方面，当测试耦合组时，我们必须将组中的所有下标放在一起考虑，以便得一个精确的方向集合。注意，这里测试中不精确的意思是指依赖测试程序可能报告了实际上不可能存在的某些方向。

耦合下标组

正如我们在前一节指出的，一个不可分的下标必须包含一个索引的出现，该索引至少也在相同数组引用偶的另一个下标中出现。包含相同索引的任何两个下标被称为耦合的。

识别耦合是重要的，因为带有耦合下标的多维数组引用能导致依赖测试的不精确。为看出这一点，考虑下面的循环例子:

```
      DO I = 1,100
S1      A(I+1,I) = B(I) + C
S2      D(I) = A(I,I) * E
      ENDDO
```

如果分开检查数组A的下标引用，我们会发现不能消除语句S_2对语句S_1依赖的可能性。S_1中在存储时的第一个下标等于S_2中下一次迭代使用中的第一个下标。在存储时的第二个下标等于相同迭代中的第二个下标。因此，没有一个下标能独立地用来消除依赖。然而，如果将它们放在一起考虑，我们看到不可能有依赖存在——依赖不能同时既是循环携带的，又是循环无关的。

很容易看出，耦合是等价关系的一种形式。因此，我们可将一个（最小的）耦合组定义为耦合关系下的非平凡等价类——一个至少有两个下标的组使得（1）每个下标是耦合的，在该耦合组中至少还要有另外一个下标，（2）该组不能划分成更小的组，除非将两个耦合的下标放进不同的子组中。对非平凡等价类加上这种限制是必要的，因为耦合组的大小为1就是一个可分下标。

在例子

```
      DO i
        DO j
          DO k
S           A(i+1,j,k-1) = A(i,j+i,k) + C
```

```
        ENDDO
      ENDDO
    ENDDO
```

78

中，当测试语句S依赖于自身时，第一个和第二个下标形成一个耦合组，而第三个下标是可分的。

耦合的MIV下标形式 $\langle a_1 i + c_1, a_2 j + c_2 \rangle$ 称为受限的双索引变量（RDIV）下标。除了i和j是不同的索引外，这些下标类似于SIV下标。这种特殊情况的测试将在3.4.1节讨论。

注意，可以将下标耦合组想像成一个可分实体，因为它能精确地测试一个方向集合，这些方向对应于在组中出现的循环索引。然后这些方向能与其他测试产生的方向合并，合并方法与合并来自可分下标方向相同。

3.2 依赖测试概述

虽然我们没有明确地说过，将下标分成ZIV，SIV或MIV类的主要理由是能同等地测试这些较简单的下标形式（依赖的核心），比起测试更复杂的形式来更为简单和更加精确。例如，用简单检查就能确定ZIV偶A(5)和A(6)决不会相等（因此是无关的）。更复杂的MIV偶A(I+J)和A(I-J)需要更加复杂、更不精确的分析。有效依赖测试的关键是应用的测试方法不能比下标测试需要的方法更为复杂。

这一节详述迄今讨论过的每种下标类型的依赖测试的特性。讨论中隐含一个有力的算法，它能确定引起依赖的一对下标的位置，确定围绕这些引用的公共循环个数，根据类型划分引用的下标，并调用相应的测试来发现基于下标类型的方向向量。算法数学上的复杂部分涉及到一旦为测试识别了一对特定的引用后要做的工作，它是这一节余下部分的主题。算法的这一部分执行下面的步骤：

（1）将下标划分成可分的和最小耦合组。

（2）将每个下标分类为ZIV，SIV或MIV。

（3）对每个可分的下标，基于下标的复杂度应用适宜的单下标测试（ZIV，SIV，MIV），或者证明无依赖，或者对下标中出现的索引产生方向向量。如果证明无依赖，那么在其他位置上无需进一步测试。

79

（4）对每个耦合组，应用多下标测试对该组中出现的索引产生方向向量集合。

（5）如果任何测试得出无依赖的结论，由于无依赖存在而不需要进一步测试。

（6）其他情况下将前面步骤中计算的所有方向向量合并成两个引用的单一的方向向量集合。

将数组引用偶中所有的下标分类为可分的或某个最小耦合部分，算法得以利用可分性。一个耦合组是最小的，如果它不能被划分成具有不同索引集的两个非空子组。一旦实现了划分，每个可分的下标和每个耦合组有完全不相交的索引集。然后，分开测试每个划分，并且不失精度地合并得到距离向量或方向向量。

3.2.1 下标划分

图3-1介绍的算法详细说明划分下标的第1步。一开始*partition*将每一对下标放入它自己的划分中，虽然初看起来算法的时间复杂度是$O(m^2)$，但当数据结构允许用常数时间的联合操作做划分时，实际上算法复杂度是线性的。

```
procedure partition(S, P, n_p)
  // S是具有索引I_1, I_2, ..., I_n的n个循环中围着的单个引用偶的m个下标偶
  //  S_1, S_2, ..., S_m的集合,
  // P是输入变量,它包含由下标划分成可分的或最小耦合组构成的集合,
  // n_p是划分的数目
  n_p = m;
  for i = 1 to m do P_i ← {S_i};
  for i = 1 to n do begin
    k ← <none>
    for each余下的划分P_j do
      if存在s ε P_j使得s包I_i then
        if k = <none> then k ← j;
    else begin p_k ← p_k ∪ p_j; 丢弃p_j; n_p: = n_p - 1; end
  end
end partition
```

图3-1 下标划分算法

3.2.2 合并方向向量

在测试算法中描述的合并操作值得另作解释。因为每个可分的和耦合的下标组包含惟一的索引子集，可以将合并想像成笛卡尔积。在循环嵌套

```
      DO I
        DO J
S1        A(I+1,J) = A(I,J) + C
        ENDDO
      ENDDO
```

中，第一个位置产生I-循环的方向向量(<)，而第二个位置产生J-循环的方向向量(=)。得到的笛卡儿积是单个向量(<,=)。

一个更复杂的例子是

```
      DO I
        DO J
S1        A(I+1,5) = A(I,N) + C
        ENDDO
      ENDDO
```

第一个下标产生I循环的方向向量(<)。因为在J-循环中没有变化（J不出现在任何下标中，且N不直接随J改变），所以J-循环的每一次迭代中，对A的两个引用自始至终访问相同的存储单元。因此，如果它们访问一个公共存储元素，则对该元素的所有访问模式都会出现。换句话说，对J-循环必须假设为方向向量全集{(<),(=),(>)}。合并产生下面的方向向量集合：{(<,<),(<,=),(<,>)}。

对ZIV下标依赖测试结果要特殊对待。如果一个ZIV下标证明是无依赖的，则依赖测试算法立即停止。如果未证明无依赖，则ZIV不产生方向向量，故无需合并。

3.3 单下标依赖测试

一旦下标得到划分和分类，就是应用特定测试的时候，由此判断依赖是否存在，并刻画

80 ~ 81

它们的行为。这一节研究可用的最简单测试：那些能应用到单下标的测试。稍后几节将研究耦合组测试。

3.3.1 ZIV测试

因为ZIV不包含对任何循环归纳变量的引用，在任何循环内，它们不变化（假设所有的符号项是循环不变量）。因此，如果能证明两个表达式不相等，那么相应的数组引用是无依赖的。如果不能说明这些表达式是不同的，那么下标不分布到任何方向向量上（因为它不包含循环归纳变量），并可被忽略。ZIV测试能很容易扩展到符号表达式上，做法是构造表示两个下标表达式的差表达式。如果差表达式可以简化为一个非0常数，则这样的下标是无依赖的。

3.3.2 SIV测试

复杂度从ZIV下标上升一级是SIV下标。SIV下标实际上是最常出现的，很多作者（著名的如Banerjee，Cohagan和Wolfe[34，77，285]）发表了用于线性SIV下标的单索引精确测试。这些方法都是基于求两个变量的简单丢番图方程的全部解。这一节介绍的方法比那些精确测试要稍微简单一些。通过将SIV下标分离成强SIV和弱SIV两个类，这些方法得到简化。

强SIV下标

对索引i，说一个SIV下标是*强的*，如果它有$\langle ai+c_1, ai'+c_2\rangle$形式；就是说，如果它是线性的，同时索引$i$两次出现的系数是常数，并且相等。图3-2给出对应强SIV偶的方程的几何图形。因为循环系数对每个引用是相同的，强SIV偶映射到一对平行直线。标记A(m)的水平线指示两个下标访问相同元素的点。因为直线的平行性质，访问公共元素将被相同的循环迭代距离隔开。这个距离称为*依赖距离*，能用下面的公式计算：

$$d=i'-i=\frac{c_1-c_2}{a} \qquad (3\text{-}3)$$

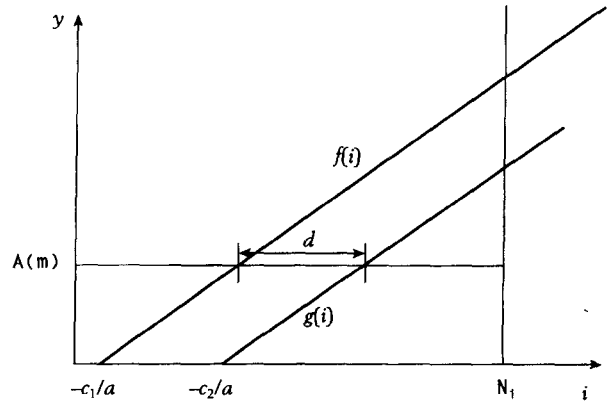


图3-2 强SIV测试的几何视图

两个引用之间存在依赖，仅当对公共元素的访问发生在由循环所设置的界内（给定SIV，两个引用必定只涉及到一个循环）。仅当d是一个整数（如果不是一个整数，两个下标对相同元素的引用不能出现在一次迭代中），并且

$$|d| \leqslant U-L \qquad (3\text{-}4)$$

时，才会发生这样的访问，这里U和L是循环上界和下界（否则，最多有一个引用能出现在循环的迭代空间内）。这些事实为我们提供一个非常简单的依赖测试：计算d；如果它是整数，且绝对值落在循环区域内，则存在依赖，它的方向能用d的符号确定[⊖]。否则无依赖存在。

强SIV测试的一个优点是很容易扩展它用来处理循环不变的符号表达式。首先扩展对符号

⊖ 这里一个值得注意的例外是距离为0，它仅能产生一个循环无关依赖。单个语句中的引用之间的0距离不表示一个依赖。

表示的依赖距离d求值。如果结果为常数，那么可以实施上述的测试。否则可计算循环界之差，并与d做符号上的比较。下面的循环是可能出现的代码例子:

```
      DO I = 1,N
S1      A(I+2*N) = A(I+N) + C
      ENDDO
```

82
~
83

强SIV测试能求出依赖距离d为2N-N，简化为N。与循环界做符号化比较，证明无依赖，因为N>N-1。

弱SIV下标

与强SIV下标不同，弱SIV下标的循环归纳变量上有不同的系数，并因此取形式$\langle a_1 i + c_1, a_2 i' + c_2 \rangle$。正如前面说明，用单索引精确测试可以求解弱SIV下标。然而，从几何角度看此问题也可能有帮助，其中依赖方程

$$a_1 i + c_1 = a_2 i' + c_2 \tag{3-5}$$

描述二维平面中的一条直线，以i和i'为坐标轴[51]。然后能将弱SIV测试公式化为在内循环上界和下界限定的空间中，判断从依赖方程推导出的直线是否相交于任何整数点，如图3-3所示。这里有两个特殊情况对我们思考有帮助。

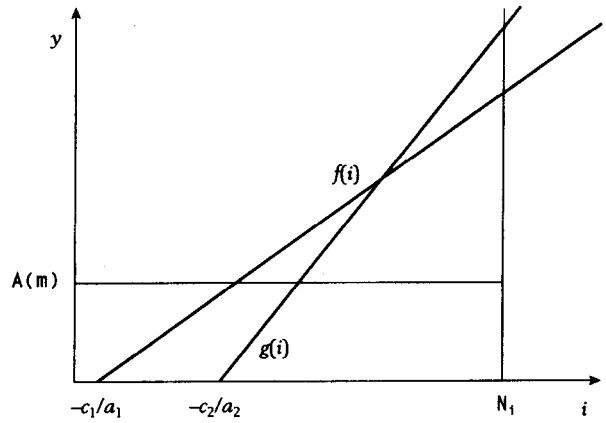


图3-3 弱SIV下标的几何视图

弱-0 SIV下标

如果两个系数之一是0（即$a_1=0$或$a_2=0$）。那么此下标是弱-0 SIV下标。如果a_2等于0，则依赖方程变成

$$i = \frac{c_2 - c_1}{a_1} \tag{3-6}$$

系数为0时的引用在循环过程中仅引用一个数组元素（图3-4中的水平直线）。假定其他的系数不是0，两条直线只在一点相交——方程（3-6）定义的那一点。因此，测试仅包括检查计算的值是一个整数，并且是在循环的界内。当a_1是0时，应用与其对称的测试。

弱-0 SIV测试求由一特殊迭代引起的依赖。在科学计算代码中，这个迭代通常是循环的第一次或是最后一次，对这样的依赖消去一个可能的方向向量。更加重要的是，由第一个或最后一个循环迭代引起的弱-0依赖可以用循环剥离消除。例如，考虑下面*tomcatv*程序中简化

的循环，此程序来源于SPEC基准测试程序包：

```
      DO I = 1 , N
S1      Y(I,N) = Y(1,N) + Y(N,N)
      ENDDO
```

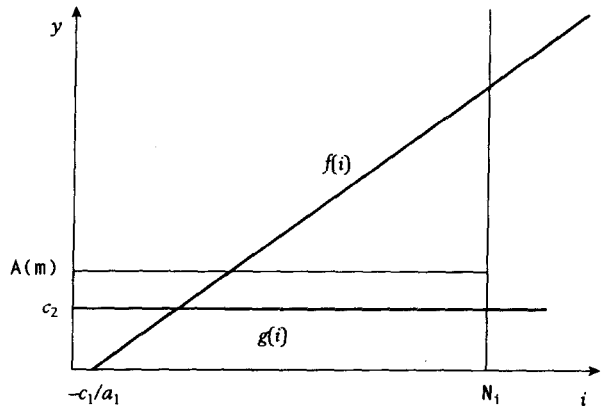


图3-4 弱-0 SIV下标的几何视图

弱-0 SIV测试能确定，Y(1,N)的使用引起一个从第一个迭代到所有其他迭代的循环携带真依赖。类似地，借助于符号分析，弱-0 SIV测试能发现Y(N,N)的使用引发从所有的迭代到最后一次迭代的循环携带反依赖。仅在引起依赖的情况下标记第一次和最后一次迭代，弱-0 SIV测试能通知用户或编译器剥离循环的第一次和最后一次迭代，得到下面的并行循环：

84 ≀ 85

```
      Y(1,N) = Y(1,N) + Y(N,N)
      DO I = 2 , N-1
S1      Y(I,N) = Y(1,N) + Y(N,N)
      ENDDO
      Y(N,N) = Y(1,N) + Y(N,N)
```

弱-交叉SIV下标

在$a_2 = -a_1$下的下标称为弱-交叉SIV。在这些情况中，化简分析是对称的，这是一个重要的性质。假设直线斜率的绝对值是相同的，那么它们总是以相同的速率从给定的一点“移动”，尽管一个是向上移动，而另一个向下移动。其结果，这两条直线对穿过它们的交点的垂直线是对称的（见图3-5）。

由于这种对称性，所有依赖的端点[⊖]是在一个交叉点（两条直线的交点）的相对两边。Cholesky分解中就出现弱-交叉SIV下标，这是一个常见的例子。为了确定交叉点的位置，将i'设置成i并做$a_2 = -a_1$代换，推导出

$$i = \frac{c_2 - c_1}{2a_1} \tag{3-7}$$

因为所有的依赖跨越交叉点，所以判断依赖是否存在成为一种简单的检查：查i的值是在循环界内，并且是一个整数或其非整数部分等于1/2。条件中出现第二部分是因为从交叉点对称地移动每条直线的结果；如果该点不在两个整数的当中，那么两条直线不能在x轴上的整数

⊖ 指源点和汇点。——译者注

点处与y轴上的整数点相交。用一个例子来说明这一点。假设N等于4，则我们有下面的i和$N-i+1$的值:

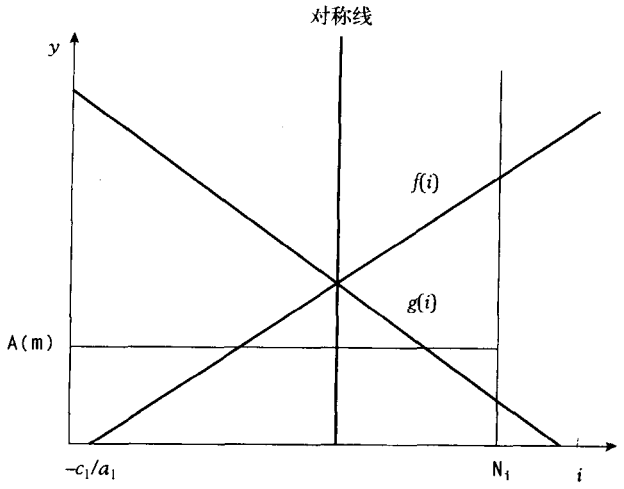


图3-5 弱-交叉SIV下标的几何视图

交叉点

i:	1	2	3	4
$N-i+1$:	4	3	2	1

由于存在依赖，交叉点必定恰好在2.5。

弱-交叉SIV依赖可以用循环分裂消除。为说明这一点，考虑下面取自Callahan-Dongarra-Levine向量测试包[57]的循环:

```
      DO I = 1, N
S1      A(I) = A(N-I+1) + C
      ENDDO
```

弱-交叉SIV测试确定在A的定义与使用之间存在依赖，并且它们都穿过迭代(N+1)/2，这里按Fortran语义需要对非整数值进行截断。在该迭代处将循环分裂产生两个并行循环:

```
      DO I = 1,(N+1)/2
         A(I) = A(N-I+1) + C
      ENDDO
      DO I = (N+1)/2 + 1 , N
         A(I) = A(N-I+1) + C
      ENDDO
```

弱SIV测试的两种形式对测试3.1.1节描述的耦合下标也是有用的。在3.3.2节末尾描述的精确SIV测试能用于任何其他的SIV下标的精确测试。

复杂的迭代空间

迄今描述的SIV测试已被应用到矩形迭代空间，其中所有的循环界不依赖于循环值本身。它们不能直接应用到三角形循环（其中循环界之一至少是一个其他循环索引的函数）。然而，可对它们加以扩展，以损失一定精度为代价使其可以应用到这样的循环上。

86
~
87

为了了解如何做到这一点，考虑循环中一种强SIV下标的特殊情况，其中的上界至多是一

个其他循环索引的函数。

```
      DO I = 1,N
        DO J = L0 + L1 * I, U0 + U1 * I
S1        A(J+D) = ...
S2             ... = A(J) + B
        ENDDO
      ENDDO
```

根据强SIV测试，我们可以看到：如果依赖距离d的绝对值比从循环下界到循环上界的距离小，则有一个语句S_2对语句S_1的依赖，它是由J循环携带的。换句话说，有一个依赖，如果

$$|d| \leqslant U_0 - L_0 + (U_1 - L_1)\mathrm{I}$$

因此，如果$|d|$小于不等式右端的最大值，我们能假设有一个依赖。如果$U_1 - L_1$是正的，当I取它的最大值时右端取最大值；否则当I取它的最小值时右端取最大值。

但是这并未告诉我们整个的情况，因为对每个I的值可能不存在依赖。特别，要想依赖存在，我们必须有

$$\mathrm{I} \geqslant \frac{|d| - (U_0 - L_0)}{U_1 - L_1} \tag{3-8}$$

对I的其他值，依赖不存在。除非能说明在I的迭代区域中，I的所有值违反不等式，不然我们必须假设在循环中有一个依赖，因为对I的某个值将有一个依赖。在这种意义下，依赖测试是精确的。

另一方面，由于对所有的值不发生依赖，我们能用表达式作为在5.7节中讨论的索引集分裂的机制。我们能附加上条件

$$\mathrm{I} < \frac{|d| - (U_0 - L_0)}{U_1 - L_1} \tag{3-9}$$

作为消除条件，它是一个谓词，确定一个条件或一些条件，在其下依赖不存在。能用消除条件对一个语句做部分向量化。例如，在循环

```
      DO I = 1,100
        DO J = 1,I
S1        A(J+20) = A(J) + B
        ENDDO
      ENDDO
```

88

中，使得下面的消除条件成立的每个I的值，语句S_1能向量化：

$$\mathrm{I} < \frac{|d| - (U_0 - L_0)}{U_1 - L_1} = \frac{20 - (-1)}{1} = 21$$

因此循环嵌套能再分成两个嵌套：

```
      DO I = 1,20
        DO J = 1,I
S1a       A(J+20) = A(J) + B
        ENDDO
      ENDDO
      DO I = 21,100
        DO J = 1,I
```

```
S1b        A(J+20) = A(J) + B
        ENDDO
     ENDDO
```

其中第一个嵌套的内循环能向量化。正如我们将在5.7.1节中看到的，也能用索引集分裂对第二个循环向量化。

对弱-0测试，类似的分析成立。

```
     DO I = 1,N
        DO J = L0+L1 * I , U0+U1 * I
S1         A(c) =
S2               = A(J) + B
         ENDDO
     ENDDO
```

这里仅当c在循环界内，即

$$L_0 + L_1 I \leqslant c \leqslant U_0 + U_1 I$$

有一个依赖。对所有的两个不等式成立的I值，依赖存在。然而，对迭代范围内I的任何值，如果不等式成立，我们必须报告这个依赖，即使它仅在由下面不等式给出的I值的非空循环内成立:

89

$$\frac{c - U_0}{U_1} \leqslant I \leqslant \frac{c - L_0}{L_1} \tag{3-10}$$

如果不等式的无论那一端分母中的项是0，那么我们假设在那一端是无界的。

技术上讲，如果在一循环中循环索引变量有一上界或下界，它是出现在相同引用偶另一个下标中的另一循环的索引，那么该下标的位置是耦合的。为了看清为什么，考虑例子:

```
     DO I = 1,100
        DO J = 1,I
S1         A(J+20, I) = A(J,19) + B
        ENDDO
     ENDDO
```

用三角形强SIV测试，第一个下标恰似我们较早的例子——不可能有依赖，除非I ≥ 21。应用弱-0测试于第二个下标，测试告诉我们仅当I=19时两个下标是相等的。因此，虽然我们假设分别测试两个下标时有依赖，但可能没有依赖。

具有2个SIV下标耦合的例子如下:

```
     DO I = 1,100
        DO J = I,I + 19
S1         A(I+20, J) = A(I,J) + B
        ENDDO
     ENDDO
```

独立的SIV下标测试指示有一个依赖，距离向量为（20，0）。然而，我们注意到在依赖源点和汇点上J-循环的迭代区域是不相交的: 当I=1时，语句S_1存入A(21,1:20)，而在汇点上，语句S_1从A(21,21:40)读入。因为在两个存储片中没有公共存储单元，故不可能有依赖。

事实上，这类情况很少发生，以致不值得担心。然而，我们将看到某些更强的耦合组测试，这些情况很容易碰到。

符号的SIV依赖测试

当所有涉及到的所有项是编译时常数时，迄今已介绍的多数依赖测试是精确的，但是对带有符号量的情况，它们不能很好地对付。有少数例外：例如，假设能用符号形成和化简差$(c_2 - c_1)$，然后就像一个常数那样使用，那么循环不变的符号加常数便得到了处理。符号量频繁地出现在下标中起因于程序设计的实践，诸如将多维数组传给一个过程，而该过程只希望接受一个一维数组。因此，符号测试是重要的。

90

这一节描述一种特殊的测试，它测试包含在两个不同循环的相同嵌套层上引用之间无依赖。例如，这种测试能应用到下面的一对循环上。

```
      DO I = L1, U1
S1      A(a1*I+c1) = ...
      ENDDO
      DO J = L2, U2
S2      ... = A(a2*J+c2)
      ENDDO
```

为简单起见，假设a_1大于或等于0。如果满足下面的依赖方程，则存在一个依赖：

$$a_1 i - a_2 j = c_2 - c_1 \tag{3-11}$$

对某个i和j的值，$L_1 \leqslant i \leqslant U_1$和$L_2 \leqslant j \leqslant U_2$。有两种情况要考虑。第一种情况，$a_1$和$a_2$可能有相同的符号。在这种情况下，假设$a_1 i - a_2 j$对$i = U_1$和$j = L_2$取最大值，对$i = L_1$和$j = U_2$取最小值（记住$a_1$和$a_2$是非负的）。因此，仅当

$$a_1 L_1 - a_2 U_2 \leqslant c_2 - c_1 \leqslant a_1 U_1 - a_2 L_2 \tag{3-12}$$

时存在依赖。如果违反任何一不等式，则依赖不存在。

在第二种情况，a_1和a_2有不同的符号。在这种情况下，假设$a_1 i - a_2 j$对$i = U_1$和$j = U_2$取最大值，所以仅当

$$a_1 L_1 - a_2 L_2 \leqslant c_2 - c_1 \leqslant a_1 U_1 - a_2 U_2 \tag{3-13}$$

时存在依赖。如果违反了任何一个不等式，则依赖不存在。

需要注意的是，这些不等式正好是Banerjee不等式的特殊情况，在3.3.3节中将介绍Banerjee不等式。然而所述的这种形式显然能对c_1，c_2，L_1，L_2，U_1和U_2的符号值公式化。另外，这种测试还能用来测试相同循环（即$L_1 = L_2$和$U_1 = U_2$）中的依赖。

作为符号测试的一个例子，考虑下面的循环：

```
      DO I = N+1,2*N
S1      A(I+N) = A(I) + B
      ENDDO
```

替换方程（3-13）中的循环界，我们得到下面的符号不等式：

$$N+1-2N \leqslant 0-N \leqslant 2N-(N+1)$$

91

化简成

$$1-N \leqslant -N \leqslant N-1$$

如果假设循环至少执行一次迭代（否则问题是不切实际的），$1-N$决不会小于$-N$，所以没有依赖。一个好的简化式将会捕捉到这种情况，以及更复杂的情况。

消除条件

用另一种方法看前面例子，循环中的依赖距离是符号量N。为了看清这一点，我们能构成由强SIV测试给出的依赖距离表达式。因为在循环中N的值不变，所以有一个符号距离是可能的。

现在考察与前面例子非常类似的循环:

```
      DO I = 1, L
S1      A(I+N) = A(I) + B
      ENDDO
```

因为变量N不在循环上界中出现，在编译时我们不能判断依赖距离实际上是否存在，所以我们必须假定它存在。然而，我们知道如果循环上界L不大于依赖距离的话，就不存在依赖。换句话说，如果L<=N，则没有从语句S_1到自身的依赖。谓词“L<=N”称为依赖消除条件。许多依赖测试程序，当面对像这样简单的情况时，将注释为依赖，这种依赖不能在编译时用消除条件证明其不存在，希望以后能用运行时测试或者从程序员的输入消除此依赖。

在上面的例子中，向量化程序能生成选择代码，它依赖于运行时消除条件的值:

```
      IF (L<=N)THEN
         A(N+1:N+L) = A(1:L) + B
      ELSE
         DO I = 1, L
S1          A(I+N) = A(I) + B
         ENDDO
      ENDIF
```

用循环分段把循环恰好分为N段能进一步改善此代码，如果N的值大到足以使向量化是值得做的。

92 另一个有趣的消除条件起因于弱-0测试，如下面例子所示:

```
      DO I = 1, N
S1      A(I) = A(L) + B
      ENDDO
```

假设不知道N和L的值之间的关系，弱-0测试不能证明语句S_1没有自依赖。然而，依赖测试程序能为这种依赖记录下消除条件

```
(L<1) .OR. (L>N)
```

能用它有条件地向量化循环。

```
      IF((L.LT.1).OR.(L.GT.N))THEN
         A(1:N) = A(L) + B
      ELSE
         DO I = 1, N
S1          A(I) = A(L) + B
         ENDDO
      ENDIF
```

如前所说，对串行循环分段，能进一步改善此代码。

ZIV测试也提供许多使用消除条件的机会。考虑下面的循环:

```
      DO I = 1, N
S1      A(L) = B(I) + C
```

```
S2      B(I+1) = A(L+K) + X(I)
        ENDDO
```

在不知道L和K在循环中的值的情况下，我们不能证明语句S_2依赖于语句S_1的假设不成立。然而很清楚，除非K为0，否则不可能有依赖。因此，谓词K.NE.0是依赖和依赖环的消除条件。

消除条件在某些程序库中特别有用。例如，在LAPACK线性代数库中，一个基本的线性代数子程序有一个如像下面的循环段：

```
        DO I = 1,N
S1      A(S*I) = A(S*I) + B(I)
        ENDDO
```

这里变量S保存循环的跨距(>0)，由调用程序将它传递给此子程序。在大多数程序中跨距为1。然而，如果在退化的情况下S=0，则循环变成了归约。在此例中，依赖测试程序必须假设从语句S_1到自身有三个依赖——一个真依赖、一个反依赖和一个输出依赖。然而，这些依 93
赖都有相同的消除条件S.NE.0，它能用来提供选择循环的版本，在运行时可选择一个向量版本和一个归约版本。

精确的SIV测试

在SIV下标位置上，下标具有形式

$$a_1 i + a_0 \text{和} b_1 i + b_0$$

通过构造线性丢番图方程

$$a_1 x - b_1 y = b_0 - a_0 \tag{3-14}$$

的全部解，能做精确的测试。

此方程组有解，当且仅当a_1和b_1的最大公约数整除$b_0 - a_0$。众所周知，计算最大公约数的Enclid算法,可以将它们扩展为计算n_a和n_b，使得

$$n_a a_1 + n_b b_1 = \gcd(a_1, b_1) \tag{3-15}$$

一旦可以得到这些值，下面的公式给出此丢番图方程的所有解：

$$\begin{aligned} x_k &= n_a\left(\frac{b_0 - a_0}{g}\right) + k\frac{b_1}{g} \\ y_k &= n_b\left(\frac{b_0 - a_0}{g}\right) + k\frac{a_1}{g} \end{aligned} \tag{3-16}$$

这里对每个k的整数值 (x_k, y_k) 是方程$a_1 x - b_1 y = b_0 - a_0$的一个解。另外，对于任何一个解$(x, y)$，存在一个$k$使得$x = x_k$和$y = y_k$。

单独一个解并不意味着有依赖。依赖存在，解必须出现在循环上下界指示的区域中。对一特殊方向向量和给定的循环界，确定是否存在一个意味着有依赖的解，需要一个搜索过程，这超出了本书的范围。

3.3.3 多归纳变量测试

当限制下标为循环归纳变量的线性函数时，ZIV下标和SIV下标是相对简单的从Z（自然数集合）到Z的线性映射。MIV下标要复杂得多，是从Z^m到Z的线性映射，其中m是出现在下标中的循环归纳变量的个数。为精确地确定依赖，这种增加的复杂性需要更高级的数学。因 94

此这一节回顾一下一般的依赖方程，更换适于MIV下标和相应数学使用的术语，作为MIV测试本身的前奏。

为了说明这些问题，让我们考虑循环的一般嵌套：

```
      DO i1 = L1,U1
        DO i2 = L2,U2
          ...
          DO in = Ln,Un
S1          A(f(i1,...,in)) = ...
S2          ... = A(g(i1,...,in))
          ENDDO
          ...
        ENDDO
      ENDDO
```

判断是否有一个带方向向量$\mathbf{D}=(D_1, D_2, ..., D_n)$的依赖，等价于判断方程组

$$f(x_1, x_2, ..., x_n)=g(y_1, y_2, ..., y_n) \tag{3-17}$$

在由

$$L_i \leqslant x_i, y_i \leqslant U_i, \forall i, 1 \leqslant i \leqslant n \tag{3-18}$$

定义的空间中是否存在一整数解，其中由于方向向量而有附加的限制

$$x_i D_i y_i \ \forall i, 1 \leqslant i \leqslant n \tag{3-19}$$

（回顾一下，方向向量的每一项是“<”，“=”或“>”之一。）令R表示由方程（3-18）和（3-19）定义的区域。如果

$$h(x_1, x_2, ..., x_n, y_1, y_2, ..., y_n)=f(x_1, x_2, ..., x_n)-g(y_1, y_2, ..., y_n)=0 \tag{3-20}$$

在区域R内某处有一整数解，则方程（3-17）有一个解。

因为需要的是整数解，此问题取决于丢番图方程理论。正如我们早先提到的，在一个受限的空间中精确地解丢番图方程是困难的。因此，寻求简化方法对编译器是有益的，这些方法不精确，但提供合理的精确性。一种这样的简化方法是去掉解是整数的限制，使解空间是连续的，而不是离散的。换言之，在区域R中寻找方程（3-20）的实数解（或更精确地说，无
95 实数解）是有益的。如果方程无实数解，那么它不能有整数解，并因此不可能有依赖存在。暂时假设函数f和g在区域R上是连续的。由初等分析直接得到下面的定理。

定理3.1 如果f和g是区域R上的连续函数，方程（3-20）存在实数解，当且仅当

$$\min_R h \leqslant 0 \leqslant \max_R h \tag{3-21}$$

证明 从中值定理立即可得结论[⊖]。

此定理是几个依赖测试的基础。

本章余下的部分假设f和g是两个仿射函数；即它们有形式

$$f(x_1, x_2, ..., x_n)=a_0+a_1x_1+...+a_nx_n \tag{3-22}$$

⊖ 严格地讲，区域R必须是封闭的，以便用最大值和最小值替换下确界和上确界。对于本书中的应用来说，所有区域将是封闭的。

$$g(y_1, y_2, ..., y_n) = b_0 + b_1y_1 + ... + b_ny_n \tag{3-23}$$

而求解的依赖问题是在区域R上求解线性丢番图方程

$$a_0 - b_0 + a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n = 0 \tag{3-24}$$

GCD测试

重新排列方程（3-24）的项产生方程

$$a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n = b_0 - a_0 \tag{3-25}$$

它是线性丢番图方程的标准形式。线性丢番图方程作为广泛研究的课题已有几百年。关于这些方程的一个基本定理是下面的定理:

定理3.2 GCD测试 方程（3-25）有一个解，当且仅当$\gcd(a_1, ..., a_n, b_1, ..., b_n)$整除$b_0 - a_0$。

因此，如果循环归纳变量的所有系数的gcd不能整除两个常数附加项之差，那么方程根本不可能有解——因此无依赖存在。另一方面，如果系数的gcd确实整除$b_0 - a_0$，则在某处有一 96
个解，虽然不需要它是在我们关心的区域R内。

当对特定的方向向量$\mathbf{D} = (D_1, ..., D_n)$做测试时，它的某些方向是“=”，这个条件可以更加严格。假设被测试的依赖的方向向量是$\mathbf{D}$，并且仅有一个“=”分量D_i，则任何可接受的解必须有$x_i = y_i$，使方程

$$a_1x_1 - b_1y_1 + ... + (a_i - b_i)\,x_i + ... + a_nx_n - b_ny_n = b_0 - a_0 \tag{3-26}$$

成立。显然，gcd应包含$(a_i - b_i)$，并且不含a_i和b_i，在这种情况下结果稍微精确一些。在一般情况下，应实施一个类似的替代条件，即正被测试的方向向量的任何位置上均是“=”。

很清楚，根据这种意见3.3.2节中描述的强SIV测试是定理3.2中GCD测试的一种特殊情况。

Banerjee不等式

虽然GCD测试在某些情况下极为有用，但是它不适合作为通用的依赖测试。理由是实际上遇到的大多数gcd是1，它整除任何整数。另外，每当依赖方程无论在何处而恰好不在区域R中有一个整数解时，GCD测试指出有依赖。从公用下标导出方程组通常在某处有整数解，即使解不在我们关心的区域中。考虑用迭代极限消除此问题的一种测试是Banerjee不等式。

在f和g是仿射函数的情况下，从定理3.1直接可得到Banerjee不等式:

$$h(x_1, x_2, ..., x_n, y_1, y_2, ..., y_n) = a_0 - b_0 + a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n \tag{3-27}$$

在介绍此测试之前，需要某些预备定义和表示法。

定义3.1 令$h_i^+ = \max_{R_i} h_i(x_i, y_i)$和$h_i^- = \min_{R_i} h_i(x_i, y_i)$，其中

$$h_i(x_i, y_i) = a_ix_i - b_iy_i \tag{3-28}$$

并且区域R_i由不等式

$$L_i \leqslant x_i, y_i \leqslant U_i \text{且} x_i D_i y_i \tag{3-29}$$

定义，其中D_i是第i个位置上的方向向量元素。

换句话说，h_i^+是一个区域上函数h_i的最大值，而h_i^-是最小值。为了使这些值是有用的，必须用某种方法计算它们。下面的定义有助于达到此目的。 97

定义3.2 令a是实数，a的正部表示为a^+，用表达式

$$a^+ = \text{if } a \geqslant 0 \text{ then } a \text{ else } 0 \tag{3-30}$$

给出。a的负部表示为a^-，用表达式

$$a^- = \text{if } a \geqslant 0 \text{ then } 0 \text{ else } -a \tag{3-31}$$

给出。

a^+和a^-都是非负的，并且下面的关系成立:

$$a = a^+ - a^- \tag{3-32}$$

根据这些定义，可证明下面的引理。

引理3.1 令t，s和z表示实数。如果$0 \leqslant z \leqslant s$，则

$$-t^- s \leqslant tz \leqslant t^+ s$$

另外，存在值z_1，z_2使得

$$tz_1 = -t^- s \text{和} tz_2 = t^+ s$$

证明 情况1: $t \geqslant 0$。在这种情况下$t^+ = t$和$t^- = 0$。所以不等式满足，因为$0 \leqslant tz \leqslant t^+ z \leqslant t^+ s$。由于$0 \leqslant z \leqslant s$，在这种情况$z_1 = 0$和$z_2 = s$。

情况2: $t < 0$。在这种情况下$t^+ = 0$且$t^- = -t$。因此，不等式变为$-t^- s \leqslant -t^- z = tz \leqslant 0$，

且我们有$z_1 = s$和$z_2 = 0$。

下一步将不等式推广到任意的界。

引理3.2 令t，l，u和z表示实数。如果$l \leqslant z \leqslant u$，则

$$-t^- u + t^+ l \leqslant tz \leqslant t^+ u - t^- l$$

并且存在值z_1，z_2，使得

$$tz_1 = -t^- u + t^+ l \text{和} tz_2 = t^+ u - t^- l$$

证明 假设$l \leqslant z \leqslant u$。重排各项，给出$0 \leqslant z - l \leqslant u - l$。引理3.1则给出

$$-t^-(u-l) \leqslant t(z-l) \leqslant t^+(u-l)$$

再次重排各项产生

98

$$-t^- u + (t + t^-) l \leqslant tz \leqslant t^+ u - (t^+ - t) l$$

重排方程（3-32）给出

$$t^+ = t + t^- \text{和} t^- = t^+ - t$$

不等式由此证明。z_1和z_2的存在直接从引理3.1推出。

这个不等式提供证明关于h的最小值和最大值的重要结果的基础。然而，在开始之前，我们需要介绍一些术语。

定义3.3 量$H_i^-(D)$和$H_i^+(\mathrm{D})$（其中D是一方向）定义如下:

$$H_i^-(<) = -(a_i^- + b_i)^+(U_i - 1) + [(a_i^- + b_i)^- + a_i^+]L_i - b_i$$
$$H_i^+(<) = (a_i^+ - b_i)^+(U_i - 1) - [(a_i^+ - b_i)^- + a_i^-]L_i - b_i$$
$$H_i^-(=) = -(a_i - b_i)^- U_i + (a_i - b_i)^+ L_i$$
$$H_i^+(=) = (a_i - b_i)^+ U_i - (a_i - b_i)^- L_i$$
$$H_i^-(>) = -(a_i - b_i^+)^-(U_i - 1) + [(a_i - b_i^+)^+ + b_i^-]L_i + a_i$$
$$H_i^+(>) = (a_i + b_i^-)^+(U_i - 1) - [(a_i + b_i^-)^- + b_i^+]L_i + a_i$$
$$H_i^-(*) = -a_i^- U_i^x + a_i^+ L_i^x - b_i^+ U_i^y + b_i^- L_i^y$$
$$H_i^+(*) = a_i^+ U_i^x - a_i^- L_i^x + b_i^- U_i^y - b_i^+ L_i^y$$

其中L_i^x，U_i^x，L_i^y和U_i^y在后两个方程中用于处理依赖的源点和汇点是在具有不同上界和下界的不同循环中的情况。

99

在方向“*”的定义中这样使用不同的上界和下界，许可处理像下面的情况：

```
    DO I = 1,100
       DO J = 1,50
S1        A(I,J) = B(I,J) + C
       ENDDO
       DO J = 51,100
S2        A(I,J) = A(I,J) + D
       ENDDO
    ENDDO
```

因为第一个和第二个循环的迭代区域不相交，所以S_1和S_2无依赖。允许上界和下界是不同的，这使得可能捕捉到用Banerjee不等式的这种情况。

引理3.3 令$h_i(x_i, y_i) = a_i x_i - b_i y_i$，并且$R_i$是由方程（3-29）——$L_i \leqslant x_i, y_i \leqslant U_i$和$x_i D_i y_i$——描述的区域。则$h_i$的最小值和最大值由

$$\min_R h_i = h_i^- = H_i^-(D_i) \tag{3-33}$$
$$\max_R h_i = h_i^+ = H_i^+(D_i) \tag{3-34}$$

给出，其中H的值由定义3.3给出。

证明 情况1：D= “=”。在这种情况下$L_i \leqslant x_i = y_i \leqslant U_i$。在方程中用$x_i$替换$y_i$得到：

$$L_i \leqslant x_i \leqslant U_i$$
$$h_i = a_i x_i - b_i y_i = (a_i - b_i)x_i$$

应用引理3.2产生想要的不等式。因为引理3.2保证在R内存在一些点，在这些点上获得左端和右端的值，结论得证。

情况2：D= “<”。在这种情况下$L_i \leqslant x_i < y_i \leqslant U_i$。为了使用引理3.2，不等式需要变换成

$$L_i \leqslant x_i \leqslant y_i - 1 \leqslant U_i - 1$$

注意，h_i（被最小化和最大化的量）由下式给出：

$$h_i = a_i x_i - b_i y_i = a_i x_i - b_i(y_i - 1) - b_i$$

引理3.2应用到$L_i \leqslant x_i \leqslant y_i - 1$上，能消去$x_i$：

$$-a_i^-(y_i-1)+a_i^+L_i-b_i(y_i-1)-b_i\leqslant h_i$$
$$\leqslant a_i^+(y_i-1)-a_i^-L_i-b_i(y_i-1)-b_i$$

接着应用引理3.2到$L_i\leqslant y_i-1\leqslant U_i-1$上，消去$y_i$:

$$-(a_i^-+b_i)^+(U_i-1)+(a_i^-+b_i)^-L_i+a_i^+L_i-b_i\leqslant h_i$$
100
$$\leqslant(a_i^+-b_i)^+(U_i-1)-(a_i^+-b_i)^-L_i-a_i^-L_i-b_i$$

它证实了这种情况的结论。引理3.1再一次保证在R内存在一些点，在这些点上出现最小值和最大值。

情况3: $D=$ “$>$”。在这种情况下$L_i\leqslant y_i<x_i\leqslant U_i$。首先将不等式转换成

$$L_i\leqslant y_i\leqslant x_i-1\leqslant U_i-1$$

重排h_i产生

$$h_i=a_ix_i-b_iy_i=a_i(x_i-1)-b_iy_i+a_i$$

应用引理3.2消去y_i:

$$a_i(x_i-1)-b_i^+(x_i-1)+b_i^-L_i+a_i\leqslant h_i$$
$$\leqslant a_i(x_i-1)+b_i^-(x_i-1)-b_i^+L_i+a_i$$

第二次引用引理3.2消去x_i:

$$-(a_i-b_i^+)^-(U_i-1)+(a_i-b_i^+)^+L_i+b_i^-L_i+a_i\leqslant h_i$$
$$\leqslant(a_i+b_i^-)^+(U_i-1)-(a_i+b_i^-)^-L_i-b_i^+L_i+a_i$$

这正是想要的不等式。

情况4: $D=$ “$*$”。在这种情况下$L_i^x\leqslant x_i\leqslant U_i^x$，$L_i^y\leqslant y_i\leqslant U_i^y$，$x_i$和$y_i$之间没有隐含的关系。对表达式$h_i=a_ix_i-b_iy_i$两次应用引理3.2产生不等式

$$H_i^-(*)\leqslant h_i\leqslant H_i^+(*)$$

由于引理3.2保证在R内存在某些点，在这些点上出现最小值和最大值，证实了这种情况的结果。

因为h中的每一项涉及到一个不同的归纳变量，每一项能独立地最大化和最小化。这就产生了下面的重要结果。

定理3.3 Banerjee不等式 $h=0$（方程（3-20））对方向向量$\mathbf{D}=(D_1, D_2, \ldots, D_n)$存在一个实数解，当且仅当满足下面两端的不等式:

$$\sum_{i=1}^{n}H_i^-(D_i)\leqslant b_0-a_0\leqslant\sum_{i=1}^{n}H_i^+(D_i) \tag{3-35}$$

证明 依赖方程对h提供了下面的公式:

$$h=a_0-b_0+\sum_{i=1}^{n}h_i=a_0-b_0+\sum_{i=1}^{n}(a_ix_i-b_iy_i)$$

于是能应用引理3.3使h最小化和最大化:

$$\min_R h=a_0-b_0+\sum_{i=1}^{n}H_i^-(D_i)$$

$$\max_R h = a_0 - b_0 + \sum_{i=1}^{n} H_i^+(D_i)$$

将这些公式代入定理3.1中方程（3-21）的不等式，并且从不等式的三部分减去$a_0 - b_0$就得到想要的结果。

101

处理Banerjee不等式中的符号

在前一节中我们介绍了一种方法，用来处理简单的SIV和ZIV测试中的符号量。这里我们将说明如何用Banerjee不等式测试将符号集成到MIV测试中。我们主要关心的将是存在循环的符号上界和下界。在此项任务中，将使用三条原则:

（1）除非显式说明步长为－1，我们可以假设下界不大于上界。如果不是这样的话，依赖测试是不切实际的。

（2）0乘一个未知值（在这种情况下，指一个循环上界或下界）的积总是0。

（3）如果一个循环上界或下界不受上述规则的限制，那么必须假设它取任意值。所以我们能对依赖测试做出可能最坏的假设。

在我们将使用的依赖测试策略中，第一条原则让我们消除任意小的循环上界值，或任意大的循环下界值。例如，如果我们有循环

```
DO I= 1,N
   DO J= 1,M
      DO K= 1,100
         A(I,K) = A(I+J,K) + B
      ENDDO
   ENDDO
ENDDO
```

我们知道N和M必须至少等于1。假设我们正在对方向向量(=,*,*)进行测试。注意

$$a_1 = 1;\quad a_2 = 0;\quad b_1 = 1;\quad b_2 = 1$$

在Banerjee不等式中对第一个下标位置（我们不需要k-循环，因为k不出现在该下标中）使用这些值给出

$$\begin{aligned}
&H_1^-(=) + H_2^-(*)\\
&= -(a_1 - b_1)^- U_1 + (a_1 - b_1)^+ L_1 - (a_2^- + b_2^+)U_2 + (a_2^+ + b_2^-)L_2\\
&= -(1-1)^- N + (1-1)^+ 1 - (0+1)M + (0+0)1 = -M\\
&\leqslant b_0 - a_0 = 1 - 1 = 0\\
&\leqslant H_1^+(=) + H_2^+(*)\\
&= (a_1 - b_1)^+ U_1 - (a_1 - b_1)^- L_1 + (a_2^+ + b_2^-)U_2 - (a_2^- + b_2^+)L_2\\
&\leqslant (1-1)^+ N - (1-1)^- 1 + (0+0)M - (0+1)1 = -1
\end{aligned}$$

102

右边的不等式不成立；因此不存在依赖。注意，此结果是由第二个原理得到的——两个包含符号上界的项系数为0。在左边，我们必须假设不等式成立，因为$-M$可以是一个任意的大负数，但是它不得大于－1。

梯形Banerjee不等式

至今介绍的Banerjee测试，假设循环嵌套中所有循环的上界和下界与其他循环归纳变量的值无关。然而，不是所有的循环会满足此要求，实际上经常遇到的是循环嵌套的内循环的迭

代范围依赖于外循环索引的值。下面是一个这样的例子:

```
DO I = 1,100
   DO J = 1,I-1
S1    A(J) = A(I+J-1) + C
   ENDDO
ENDDO
```

在这种情况下，内层循环不含携带依赖，因为对一给定的I值，左端的引用落在子数组A(1:I-1)中，而右端的引用落在子数组A(I:I*2-2)中。由于这两个子数组不相交，故对方向向量(=,*)不存在依赖。然而，如果在Banerjee不等式中替换系数的值

$$a_1=0;\quad a_2=1;\quad b_1=1;\quad b_2=1$$

我们得到

$$\begin{aligned}
&H_1^-(=)+H_2^-(*)\\
&=-(a_1-b_1)^-U_1+(a_1-b_1)^+L_1-(a_2^-+b_2^+)U_2+(a_2^++b_2^-)L_2\\
&=-(0-1)^-(100)+(0-1)^+1-(0+1)(I-1)+(1+0)1=-I-100\\
&\leqslant b_0-a_0=1-1=0\\
&\leqslant H_1^+(=)+H_2^+(*)\\
&=(a_1-b_1)^+U_1-(a_1-b_1)^-L_1+(a_2^++b_2^-)U_2-(a_2^-+b_2^+)L_2\\
&\leqslant(0-1)^+(100)-(0-1)^-1+(1+0)(I-1)-(0+1)1=I-3
\end{aligned}$$

103

由于I取大于3的值时，这些不等式确实能得到满足。因此必须假设有依赖。

问题是目前公式化的Banerjee不等式不能利用循环归纳变量在上界表达式中的值。为了缓解此不足之处，我们将推导出一个特殊的Banerjee不等式形式，我们称之为梯形Banerjee不等式，所以这样命名是因为它处理梯形循环。

让我们从假设上界和下界表达式能重写成循环归纳变量的仿射组合开始:

$$U_i=U_{i0}+\sum_{j=1}^{i-1}U_{ij}i_j\qquad L_i=L_{i0}+\sum_{j=1}^{i-1}L_{ij}i_j \tag{3-36}$$

想要的结果是一个修改的Banerjee不等式，它考虑了这些界表达式。这个结果不是简单明了的，因为对每个循环求最小值和最大值不再与其他循环无关。例如，在最内层循环上求最小值和最大值可能修改与外层循环关联的系数。

为了理解这一切是怎样发生的，考虑$H_i^-(<)$的求值。根据引理3.2最小化h_i产生

$$\begin{aligned}
H_i^-(<)=&-(a_i^-+b_i)^+\left(U_{i0}-1+\sum_{j=1}^{i-1}U_{ij}i_j\right)\\
&+[(a_i^-+b_i)^-+a_i^+]\left(L_{i0}+\sum_{j=1}^{i-1}L_{ij}i_j\right)-b_i
\end{aligned} \tag{3-37}$$

其中i_j可以是x_j或y_j，只要它在单重求和中始终是这一个或是那一个。遵照引理3.3中对情况2的证明，x_j的下界是在第一步中引入的，用来消除x_j，并产生

$$\begin{aligned}
&-a_i^-(y_i-1)+a_i^+L_i-b_i(y_i-1)-b_i\leqslant h_i\\
&\leqslant a_i^+(y_i-1)-a_i^-L_i-b_i(y_i-1)-b_i
\end{aligned}$$

因此，对x_j求和应当在上式左边对下界乘以a_i^+而在右边乘以a_i^-。当消除剩余的项时，对y_j

使用上界和下界是适当的。因此最后的表达式是 104

$$H_i^-(<) = -(a_i^- + b_i)^+\left(U_{i0} - 1 + \sum_{j=1}^{i-1} U_{ij}y_j\right) + (a_i^- + b_i)^-\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij}y_j\right) + a_i^+\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij}x_j\right) - b_i \tag{3-38}$$

这意味着系数x_j和y_j（$1 \leqslant j < i$）用在最小化h_i中必须用方程（3-38）中相应的项校准。对不等式的最大值方面的系数也必须做类似的校准。

图3-6（连同图3-7至3-10）包含对任何方向向量求梯形banerjee不等式的算法。注意求值不等式是从最内层循环开始并向外移动。在每一级更新不等式累加的左端和右端，然后在需要时校准对应于外层循环的系数。这些不等式左边的系数存放在临时数组A_l[]和B_l[]中，而A_r[]和B_r[]存放不等式右边的系数。

```
boolean function Banerjee(D)
  // A_l[i]是最小方面中使用的已校准的系数a_i
  // B_l[i]是最小方面中使用的已校准的系数b_i
  // A_r[i]是最大方面中使用的已校准的系数a_i
  // B_r[i]是最大方面中使用的已校准的系数b_i
  // V_min是累加的Banerjee左端
  // V_max是累加的Banerjee右端
  for i := 1 to n do begin
    A_l[i] := a_i; A_r[i] := a_i; B_l[i] := b_i; B_r[i] := b_i;
  end;
  V_min := 0; V_max := 0;
  for i := n to 1 do begin
    if D_i = "=" then
      ComputePositionEqual(V_min, V_max, A_l, B_l, A_r, B_r);
    else if D_i = "<" then
      ComputePositionLessThan(V_min, V_max, A_l, B_l, A_r, B_r);
    else if D_i = ">" then
      ComputePositionGreaterThan(V_min, V_max, A_l, B_l, A_r, B_r);
    else if D_i = "*" then
      ComputePositionAny(V_min, V_max, A_l, B_l, A_r, B_r);
  end;
  if V_min ≤ b_0 - a_0 and b_0 - a_0 ≤ V_max then return true else return false;
end Banerjee
```

图3-6 梯形Banerjee不等式求值

注意，对方向“*”的测试假设在依赖的源点和汇点上我们可以有不同的循环界。这是从要求在没有相交的迭代范围的分离循环中消除有端点的依赖推导出来的。（见前一小节对“Banerjee不等式”的讨论。）在这种情况下，我们假设每一个出现有不同的上界和下界表达式：

```
procedure ComputePositionEqual(V_min, V_max, A_l, B_l, A_r, B_r)
  // A_l[i]是最小方面中使用的已校准的系数a_i
  // B_l[i]是最小方面中使用的已校准的系数b_i
  // A_r[i]是最大方面中使用的已校准的系数a_i
  // B_r[i]是最大方面中使用的已校准的系数b_i
  // V_min是累加的Banerjee左端
  // V_max是累加的Banerjee右端
    V_min := V_min - (A_l[i] - B_l[i])^- U_i0 + (A_l[i] - B_l[i])^+ L_i0;
    V_max := V_max + (A_r[i] - B_r[i])^+ U_i0 - (A_r[i] - B_r[i])^- L_i0;
    nU := 0; nL := 0;
    // v(<) = -1, v(=) = 0和v(>) = 1
    for j := 1 to i-1 do begin
      nL := nL + L_ij v(D_j); nU := nU + U_ij v(D_j);
    end
    if nL < 0 then
      for j := 1 to i-1 do begin
        // 为更新B而反号
        B_l[j] := B_l[j] - (A_l[i] - B_l[i])^+ L_ij ;
        B_r[j] := B_r[j] + (A_r[i] - B_r[i])^- L_ij ;
      end
  else
    for j := 1 to i-1 do begin
      A_l[j] := A_l[j] + (A_l[i] - B_l[i])^+ L_ij ;
      A_r[j] := A_r[j] - (A_r[i] - B_r[i])^- L_ij ;
    end
  end
  if nU < 0 then begin
      for j := 1 to i-1 do begin
      A_l[j] := A_l[j] - (A_l[i] - B_l[i])^- U_ij ;
      A_r[j] := A_r[j] + (A_r[i] - B_r[i])^+ U_ij ;
    end
  else begin
    for j := 1 to i-1 do begin
      // 为更新B而反号
      B_l[j] := B_l[j] + (A_l[i] - B_l[i])^- U_ij ;
      B_r[j] := B_r[j] - (A_r[i] - B_r[i])^+ U_ij ;
    end
  end
end ComputePositionEqual
```

图3-7 方向为“=”的梯形Banerjee不等式

$$U_i^x = U_{i0}^x + \sum_{j=1}^{i-1} U_{ij}^x x_j \qquad L_i^x = L_{i0}^x + \sum_{j=1}^{i-1} L_{ij}^x x_j \tag{3-39}$$

$$U_i^y = U_{i0}^y + \sum_{j=1}^{i-1} U_{ij}^y y_j \qquad L_i^y = L_{i0}^y + \sum_{j=1}^{i-1} L_{ij}^y y_j \tag{3-40}$$

当然，如果两个出现都在相同的循环内的话，这些值将是相同的。

```
procedure ComputePositionLessThan(V_min, V_max, A_l, B_l, A_r, B_r)
  // A_l[i], B_l[i], A_r[i], B_r[i]是已校准的系数
  // V_min是累加的Banerjee左端
  // V_max是累加的Banerjee右端
  V_min := V_min − (A_l[i]⁻ + B_l[i])⁺(U_i0 − 1) + ((A_l[i]⁻ + B_l[i])⁻ + A_l[i]⁺)L_i0 − B_l[i]);
  V_max := V_max + (A_r[i]⁺ − B_r[i])⁺(U_i0 − 1) − ((A_r[i]⁺ − B_r[i])⁻ + A_r[i]⁻)L_i0 − B_r[i]);
  for j := 1 to i−1 do begin
    // 为更新B而反号
    A_l[j] := A_l[j] + A_l[i]⁺L_ij;
    B_l[j] := B_l[j] + (A_l[i]⁻ + B_l[i])⁺U_ij − (A_l[i]⁻ + B_l[i])⁻L_ij;
    A_r[j] := A_r[j] − A_r[i]⁻L_ij;
    B_r[j] := B_r[j] − (A_r[i]⁺ − B_r[i])⁺U_ij + (A_r[i]⁺ − B_r[i])⁻L_ij;
  end
end ComputePositionLessThan
```

图3-8 方向为“<”的梯形Banerjee不等式

```
procedure ComputePositionGreaterThan(V_min, V_max, A_l, B_l, A_r, B_r)
  // A_l[i], B_l[i], A_r[i], B_r[i]是已校准的系数
  // V_min是累加的Banerjee左端
  // V_max是累加的Banerjee右端
  V_min := V_min − (A_l[i] − B_l[i]⁺)⁻(U_i0 − 1) + ((A_l[i] − B_l[i]⁺)⁺ + B_l[i]⁻)L_i0 + A_l[i];
  V_max := V_max + (A_r[i] + B_r[i]⁻)⁺(U_i0 − 1) − ((A_r[i] + B_r[i]⁻)⁻ + B_r[i]⁺)L_i0 + A_l[i];
  for j := 1 to i−1 do begin
  // 为更新B而反号
    B_l[j] := B_l[j] − B_l[i]⁻L_ij;
    A_l[j] := A_l[j] − (A_l[i] − B_l[i]⁺)⁻U_ij + (A_l[i] − B_l[i]⁺)⁺L_ij;
    B_r[j] := B_r[j] + B_r[i]⁺L_ij;
    A_r[j] := A_r[j] + (A_r[i] + B_r[i]⁻)⁺U_ij − (A_r[i] + B_r[i]⁻)⁻L_ij;
  end
end ComputePositionGreaterThan
```

图3-9 方向为“>”的梯形Banerjee不等式

```
procedure ComputePositionAny(V_min, V_max, A_l, B_l, A_r, B_r)
  // A_l[i], B_l[i], A_r[i], B_r[i]是已校准的系数
  // V_min是累加的Banerjee左端
  // V_max是累加的Banerjee右端
  V_min := V_min − A_l[i]⁻U_i0^x + A_l[i]⁺L_i0^x − B_l[i]⁺U_i0^y + B_l[i]⁻L_i0^y;
  V_max := V_max + A_r[i]⁺U_i0^x − A_r[i]⁻L_i0^x + B_r[i]⁻U_i0^y − B_r[i]⁺L_i0^y;
  for j := 1 to i−1 do begin
  // 为更新B而反号
    A_l[j] := A_l[j] − A_l[i]⁻U_ij^x + A_l[i]⁺L_ij^x;
    B_l[j] := B_l[j] + B_l[i]⁺U_ij^y − B_l[i]⁻L_ij^y;
    A_r[j] := A_r[j] + A_r[i]⁺U_ij^x − A_r[i]⁻L_ij^x;
    B_r[j] := B_r[j] − B_r[i]⁻U_ij^y + B_r[i]⁺L_ij^y;
  end
end ComputePositionAny
```

图3-10 方向为“*”的梯形Banerjee不等式

正确性 为了说明函数*Banerjee*的正确性，我们必须先说明图3-6中由函数*Banerjee*计算的V_{min}和V_{max}值，使得

$$V_{min}+a_0-b_0 \leqslant \min_R h \text{和} \max_R h \leqslant V_{max}+a_0-b_0$$

换句话说，如果*Banerjee*返回*false*，则无依赖存在。对于$D_i=$“<”，$D_i=$“>”和$D_i=$“*”的情况，沿着方程（3-37）和（3-38）讨论中描述的直线对外层循环调整系数，直接从定理3.3的证明可得。事实上，根据引理3.3不等式对于这些方向都是精确的。

105 ~ 108 惟一难以处理的情况发生在$D_i=$“=”时。在这种情况下，下面的表达式对$H_i^-(=)$和$H_i^+(=)$是有效的:

$$\begin{aligned} H_i^-(=) &= -(a_i-b_i)^-\left(U_{i0}+\sum_{j=1}^{i-1}U_{ij}i_j\right)+(a_i-b_i)^+\left(L_{i0}+\sum_{j=1}^{i-1}L_{ij}i_j\right) \\ H_i^+(=) &= (a_i-b_i)^+\left(U_{i0}+\sum_{j=1}^{i-1}U_{ij}i_j\right)-(a_i-b_i)^-\left(L_{i0}+\sum_{j=1}^{i-1}L_{ij}i_j\right) \end{aligned} \tag{3-41}$$

问题是依赖的源点和汇点可以有不同的上界和下界，因为外循环索引在源点和汇点上的值很可能是不同的（除非所有外层循环方向都是“=”）。因为需要$x_i=y_i$，所以依赖不能存在，除非x_i大于两个下界和小于两个上界。遗憾的是，简单地将两个求和的最大值代回方程式不会对外层循环产生有意义的系数。因此，必须选择一个下界和一个上界，丢失一些精度。

选取保持尽可能高精度的上界和下界需要一种智能的试探方法。因为方向向量是已知的，所以能对

$$nL=\sum_{j=1}^{i-1}L_{ij}v(D_j) \quad \text{和} \quad nU=\sum_{j=1}^{i-1}U_{ij}v(D_j)$$

求值，其中

$$v(D_i)=\begin{cases} -1, & \text{如果}D_i=\text{“<”} \\ 0, & \text{如果}D_i=\text{“=”} \\ 1, & \text{如果}D_i=\text{“>”} \end{cases}$$

如果$nL<0$，则y_i的下界很可能大于x_i的下界，所以使用方程（3-41）中的下界。类似地，如果$nU<0$，则x_i的上界很可能比y_i的上界小，所以使用方程（3-41）中x_i的上界。在函数*Banerjee*的代码中，这种试探方法（它是精确的，如果只有一个非零L_{ij}的话）是多分支调整的根源。

回到例子上，主要的循环常数有下列值:

$$a_1=0;\ a_2=1;\ b_1=1;\ b_2=1$$
$$N_{10}=99;\ N_{20}=-1;\ N_{21}=1$$

109 梯形Banerjee不等式对这种情况不成立，我们把验证留给读者。

测试所有的方向向量

为使依赖测试有用，应当在给定的语句偶之间构造依赖方向向量的完全集。Burke和Cytron首创了一个实现这种构造的有效算法[49]。算法包含的基本思想是从测试方向向量最一般的集合开始，然后逐步细分测试，每当显示出不可能存在方向向量的完全集时，做修剪。图3-11例示这个思想（算法本身再现于图3-12中）。这里通过对（*，*，*）测试，此过程证实某些依赖存在，然后对第一个位置上的方向单独测试来细分测试。如果取得任何成功，则

对第二个位置上的不同方向做测试，进一步细分测试，如此等等。

图3-12中所示的测试是通过函数*try*实现的，用全是“*”的方向向量和一个空集*dvlist*（即*D*:=(*, *, ..., *); *dvlist*:=*try*(*D*, 1, Ø);）调用该函数。然后每当一次测试成功时该函数递归地调用自己。

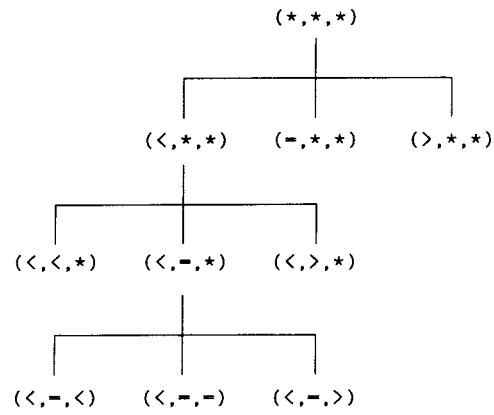


图3-11 对所有的方向向量作测试

虽然在最坏的情况下，此过程有与穷举测试相同的渐进复杂性，但在一般情况下，每当沿着一个特殊的分支测试表明没有依赖存在时，通过修剪树能有效地减少测试的开销。

3.4 耦合组中的测试

可分下标使用的测试也能用于耦合组的每个下标上：如果任何测试证明无依赖，则没有依赖存在。然而，我们已在3.1.1节中看到，在耦合组中逐个下标测试可能产生假的依赖。

一个比逐个下标测试稍强的方法是单独地测试每个下标，并对得到的方向向量集合求交集[283]。这对3.1.1节中的例子是有效的：

```
      DO I = 1,100
S1      A (I+1,I) = B(I) + C
S2      D(I) = A(I,I) * E
      ENDDO
```

这里测试第一个下标产生方向向量（<），而在第二个位置上测试产生方向向量（=）。当求这些方向的交时，我们发现不能存在依赖。

```
set function try(D, k, dvlist)
  if not Banerjee(D) then return dvlist;
  if k=n then return dvlist » {D};
  for d := "<", "=", ">" do begin
    D_i := d;
    dvlist := try(D, k+1, dvlist);
  end
end try
```

图3-12 MIV方向向量测试

这个方法允许简单而有效的测试，在某些情况下还提供一种保守的近似于耦合组中的方向集。就是说，它可能产生依赖不存在的方向向量。例如，考虑循环

```
      DO I = 1,100
S1      A(I+1,I+2) = A(I,I) + C
      ENDDO
```

用方向向量交集做逐个下标测试会产生单个方向向量（<，<）。然而，细心检查此语句发现这个方向向量是无效的，因为实际上没有依赖存在——这对下标不能同时发生相同的方向。

为了回答此问题，我们强调依赖测试是对相交的距离向量而不是方向向量。当应用SIV测试到上面的例子时，两个下标产生两个不同的距离向量：（1）和（2）。这两个距离向量的交显然是空的。这种策略极为有效，因为实际上很大比例的这种依赖测试是强SIV测试，它产生的是距离。

110 ~ 111

许多研究人员在多下标测试方面做了研究工作[202，266，286]。多数研究工作集中在一些快速方法上，用它们来判断联立线性丢番图方程是否能有一个解。3.4.2节中提供此项研究工作的一个概述。

3.4.1 Delta 测试

这一节我们讨论一个简单直观的策略，称为Delta测试，用来测试多下标中的依赖。设计

的Delta测试对普通的耦合下标是精确而又有效的。图3-13给出算法的概貌。

```
procedure Delta_test(subscripts, DVset, dV)
  // subscripts是一组全耦合的SIV和/或MIV下标
  // DVset是一个输出参数，包含所有的方向向量
  // dV是一个输出参数，包含已知的距离向量
  while在subscripts中存在未被测试的SIV下标do begin
  应用SIV测试所有未被测试的SIV下标，
  返回无依赖或推导新的约束向量C';
  C':=C∩C';
  if C'=Øthen 返回无依赖;
  else if C≠C' then begin
    C:=C';
    传播约束C到MIV下标，
      也许创建新的ZIV或SIV下标;
    应用ZIV测试于未被测试的ZIV下标上，
      返回无依赖或继续;
    end
  end
  while存在未被测试的RDIV下标do
    测试并传播RDIV约束;
  测试余下的MIV下标并将得到的方向向量与C求交集;
  从C构造DVset和dV;
end Delta_test
```

图3-13 Delta测试算法

Delta测试包含的主要思想是建立起关于距离向量交集的直觉。因为在实践中发现多数下标是SIV，并且在多数情况下SIV测试是简单而精确的，所以从中收集到的信息能用来化简相同耦合组中其他下标的测试。在Delta测试中，我们检查耦合组中的每个SIV下标，产生一些约束条件并能传播给相同耦合组中的其他下标。通常传播结果是一种简化，它产生一个精确的方向向量集合。因为在科学计算的Fortran代码中多数耦合下标是SIV，所以Delta测试是一个实用、快速和在多数情况下精确的多下标测试。

名称“Delta测试”是缘于用ΔI的非形式用法表示I-循环中源点索引和汇点索引之间的距离。因此我们假设在依赖源点上的索引是I的一个特定的值，而在汇点上的索引是此相同的值加上距离：I+ΔI。

为了看清楚Delta测试是如何工作的，考虑一个简单的例子：

```
      DO I = 1,100
        DO J = 1,100
S1        A(I+1,I+J) = A(I,I+J-1)+C
        ENDDO
      ENDDO
```

如果我们将SIV测试应用到第一个下标上，得到的距离为1，即ΔI=1。根据第二个下标我们有方程

$$I_0 + J_0 = I_0 + \Delta I + J_0 + \Delta J - 1$$

此方程通过用源点上索引的增加值替换汇点上的索引而导出。如果我们将ΔI=1代入此方程，得到

$$I_0 + J_0 = I_0 + 1 + J_0 + \Delta J - 1$$

现在我们消去常数值I_0和J_0，化简得到

$$\Delta J = 0$$

因此仅有的一个合法的距离向量是(0,1)，仅有的合法方向向量是(=,<)。

本质上我们在第二个下标的依赖测试中所做的是析取出I因子，转换到下面对S_1的测试：

```
    DO I = 1, 100
      DO J = 1, 100
S1      A(I+1,J) = A(I,J) + C
      ENDDO
    ENDDO
```

112
~
113

注意，我们已经用强SIV下标将一个MIV下标降至强SIV形式。

Delta测试可用于检测无依赖，若它的组成成分SIV测试检测出无依赖的话。否则它将所有的SIV下标转换成约束，并将它们传播到凡是可能的MIV下标中。重复此过程直至找不到新的约束。然后这些约束传播到耦合的RDIV下标中。测试余下的MIV下标并且将得到的结果与现存的约束求交集。下面几小节更详细地研究Delta测试。

约束

在本书的上下文中，*约束*是对索引的断言，它对存在的依赖必须成立。已知对依赖的联系，约束通常是从下标推导出来的。作为一个例子，依赖方程应用到下标 $\langle a_1 i + c_1, a_2 i' + c_2 \rangle$ 上，对索引i产生约束$a_1 i - a_2 i' = c_2 - c_1$。另一个简单约束的例子是依赖距离。

约束向量$\mathbf{C} = (\delta_1, \delta_2, \ldots, \delta_n)$是一个向量，它对于耦合下标组中$n$个索引的每一个索引有一个约束。在Delta测试中，约束向量用来存放从SIV测试中产生的约束。因为SIV下标的简单性质，这些约束能很容易转换成距离向量或方向向量。

一个*约束*δ可能有下列形式：

- *依赖直线*：表示依赖方程的一条直线 $\langle ax + by = c \rangle$。
- *依赖距离*：依赖距离的值 $\langle d \rangle$；它等价于依赖直线 $\langle x - y = -d \rangle$。
- *依赖点*：表示从迭代x到迭代y的依赖的一个点 $\langle x, y \rangle$。

从强SIV测试和弱SIV测试直接导出依赖距离和依赖直线。依赖点是从相交约束得到的结果，如下一小节的描述。

相交约束

因为从所有下标得出的依赖方程对任何存在的依赖必须同时有解，所以对每个下标相交约束能改善精度。如果交集是空集，则不可能有依赖。这似乎有些背离直观，因为术语“约束”通常隐含一种限制或抑制，因此对依赖缺少有关索引的约束似乎会隐含所有的依赖可能存在。记住这个重要的事实，约束实际上是逻辑表达式，当某个依赖存在时，它保证为真。如果没有这样的逻辑表达式，那么就不可能存在依赖。我们已经看到了应用到方向向量和耦合SIV下标上的约束相交。 114

最容易求交的是依赖距离，做法是简单地对距离做比较。如果距离不全相等，则它们不能同时成立，因而没有依赖存在。例如，正如我们早先在循环嵌套中

```
      DO I = 1,N
S1      A(I+1,I+2) = A(I,I) + C
      ENDDO
```

所见到的，强SIV测试应用到第一个下标上，呈现出依赖距离为1；应用到第二个下标上，呈现出依赖距离为2。通过对它们做比较，实现将两个约束求交。它们是不相等的，产生空集做为约束集合，证明无依赖。这反映了这样的事实：代入左边下标的任何值i'，不能比代入右边下标的任何i值同时小于1和2。

甚至SIV下标中的复杂约束也能精确地求交。因为从一个SIV下标导出的每个依赖方程可以看成是二维平面上的一条直线，从多SIV下标求约束交对用于计算平面上多条直线的相交点。如果在循环界内这些直线不交于一个公共点，或者此点的坐标没有整数值，则没有依赖存在。如果所有依赖方程相交于单个依赖点，则它的坐标实际上是引起依赖的仅有两个迭代。

```
      DO I = 1,N
S1      A(I,I) = A(1,I-1) + C
      ENDDO
```

在这个例子循环中，测试A的一对引用中的第一个和第二个下标，分别推导出依赖直线<I=1>和<I=I'-1>。这两条依赖直线相交于依赖点<1,2>，表示仅有的依赖是从第一次迭代到第二次迭代。因为能精确地实施计算平面上直线的相交，约束交集是精确的。

这里将不给出完全约束交集算法，因为推导是直截了当的。

约束传播

约束传播的目标是用从测试一个下标中获取有用的信息，改善相同耦合组中其他下标的
115 依赖测试。这里我们将讨论与约束关联的几个问题，以及它们在耦合测试中的使用。

SIV约束 Delta测试的一个主要贡献是它有能力将从SIV下标中推导出的约束传播到耦合下标中，通常不丢失精度。然后，得到的带约束的下标能受到更有效和更精确的测试。虽然这里我们将不介绍一个完全约束传播算法，但是很容易构造它。这样一个算法的目标是利用对每个索引的SIV约束，来消除那个索引在目标MIV下标中的距离。为使此算法更加具体，考察下面的例子：

```
      DO I
        DO J
S1        A(I+1,I+J) = A(I,I+J)
        ENDDO
      ENDDO
```

强SIV测试应用到数组A的第一个下标上，揭示索引I的依赖距离为<1>。将此约束传播到第二个下标中，消除I的两个出现，结果为带约束的SIV下标<J-1,J>。强SIV测试应用到此下标上给出J-循环的距离是-1。由于它完成了对所有下标的测试，也就完成了Delta测试。合并约束向量的元素给出一个依赖，具有距离向量(1,-1)。

在这个例子中约束传播是精确的，因为带约束下标中索引I的两个距离被消除了。经验研究[123]表明这是科学计算代码中频繁出现的情况。通常此算法只能消去索引的一个出现。当测试耦合组时，此结果改善了精度，但不是精确的。如果需要的话，可以获得额外的精度，做法是利用此约束去简化余下的索引的区域，如Fourier-Motzkin消去法[244]所做的那样。

多遍 迭代Delta测试算法，看MIV下标是否简化为SIV下标，因为这样的动作可产生新的约束。下面的循环嵌套为我们提供一个例子：

```
      DO I
        DO J
          DO K
S1            A(J-I, I+1, J+K) = A(J-I, I, J+K)
          ENDDO
        ENDDO
      ENDDO
```

在Delta测试的第一遍中，测试第二个下标，产生I-循环的依赖距离为<1>。然后将此约束传播到第一个下标中，得到下标<J+1, J>。

因为已经建立了一个新的SIV下标，算法重复执行。在第二遍，测试新下标，产生J-循环的距离为1。然后此约束传播到第三个下标中，导出下标<K-1,K>。新的SIV下标引起另一遍，它发现K-循环的距离为-1。因为所有的SIV下标已被测试，Delta测试在这个点停止，返回距离向量(1,1,-1)。

改善精度 Delta测试能改善在余下的受约束MIV下标上所做的其他依赖测试的精度。

```
      DO I = 1,100
        DO J = 1,100
S1        A(I-1, 2*I) = A(I, I+J+110)
        ENDDO
      ENDDO
```

Banerjee不等式应用到此例子循环的下标上，不能独自证明无依赖。通过首先将最左下标转换成一依赖距离约束<-1>，然后此约束传播到最右下标中产生受约束的MIV下标<2,J-I+110>，它能由Banerjee不等式成功地处理（即检测出无依赖），Delta测试对这一点有改善。

```
      DO I
        DO J
S1        A(I,2*J+I) = A(I,2*J-I+5)
        ENDDO
      ENDDO
```

类似地，在此例子循环中，GCD测试表明对两个下标有整数解。然而，将从第一个下标得到的I的距离约束<0>传播到第二个下标中，产生受约束的MIV下标<2*J,2*J-2*I+5>。现在GCD测试能检测无依赖，因为所有索引系数的GCD是2，它不能整除常数项5。

距离向量 Delta测试对产生耦合组中MIV下标的距离向量特别有用。下面的循环嵌套作为例证，显示数字代码中相当常见的一个模式，在变换后能改善并行性:

```
      DO I = 1,N
        DO J = I+1,I+N
S1        A(I,J-I) = A(I-1,J-I) + C
        ENDDO
      ENDDO
```

例如，在此例子中由于在第二个位置的MIV下标，多数依赖测试不能计算距离向量。然而，Delta测试允许精确分析这些下标。测试将从第一个下标得到的有关I的距离约束传播到第二个下标，推导出距离向量(1,1)。

受限的DIV约束 前一节说明如何传播SIV约束。MIV约束也可以传播，但是在一般情况，传播的代价昂贵。然而，在耦合的RDIV下标的特殊情况下（在3.1.1节介绍），我们介绍一个方法，用来处理一种重要的特殊情况。为简明起见，只考虑下面这种类型的数组引用:

```
      DO i
        DO j
S1        A(i1+c1, i2+c2) = A(i3+c3, i4+c4)
        ENDDO
      ENDDO
```

当i_1和i_2是索引i的两个实例而i_3和i_4是索引j的两个实例时，从第一个下标推导出i和j之间的一个约束。此约束可以传播到第二个下标中，使用的是前面讨论过的用于SIV下标的算法。另外要考虑的只是i和j的界可能有差别。

更一般的情况，i_1和i_4是索引i的实例，i_2和i_3是索引j的实例。在这种情况下得到下面的依赖方程组:

$$i+c_1=j'+c_3$$
$$j+c_2=i'+c_4$$

当检查依赖时，每个依赖方程可被单独测试，而不丢失精度。然而，当确定距离向量或方向向量哪一个是可能的时候，必须同时考虑两个方程。

将索引i在第二个引用中的实例看成是$i+\Delta_i$，也能传播这些约束，其中Δ_i是i两次出现之间的依赖距离。用相同的方式处理索引j，产生下面的依赖方程组:

$$i+c_1=j+\Delta_j+c_3$$
$$j+c_2=i+\Delta_i+c_4$$

118 将这两个方程加在一起，并稍微重排一下各个项，产生下面的方程:

$$\Delta_i+\Delta_j=c_1+c_2-c_3-c_4$$

当对一特定的距离向量或方向向量做测试时，则能检查此依赖方程。

为了更具体地说明，考虑下面的循环例子:

```
      DO I = 1,N
        DO J = 1,N
S1        A(I,J) = A(J,I) + C
        ENDDO
      ENDDO
```

传播RDIV约束得到依赖方程$\Delta_i+\Delta_j=0$。因此距离向量必须有$(d,-d)$形式，并且仅方向向量(<,>)和(=,=)是有效的。因此，所有的依赖是由外循环携带的；内循环可以并行执行。

精度和复杂性

Delta测试的精度取决于被测试的耦合下标的性质。在第一阶段中应用的SIV测试是精确的。约束求交算法也是精确的，因为在平面上任意条直线相交能精确计算。因此，Delta测试对任意数量的耦合SIV下标是精确的。

在约束传播阶段，总能精确地应用弱-0 SIV约束和依赖点，因为它们对下标中出现的索引赋值。(强SIV下标中的)依赖距离，当相应的索引系数相等时，也可以在不失精度的情况下传播到MIV下标中。所幸这种情况在科学计算代码中是频繁出现的。

当约束能精确地传播，并且通过消除共享的索引解耦所有的下标时，Delta测试防止多下标丢失精度。在结束时，如果Delta测试用ZIV和SIV测试做了所有下标的测试，那么答案是准确的。如果仅留下可分的MIV下标，则Delta测试受到应用于每个下标的单下标测试的精度限制。研究表明，对单个下标Banerjee-GCD测试通常是准确的[32，184，201]，所以Delta测试

对这些情况很可能也是准确的。

Delta测试的不精确有三个来源。第一，依赖线和依赖距离的约束传播可能是不精确的，如果不能完全消除来自目标下标中两个引用的索引的话。第二，诸如三角形循环这样的复杂迭代空间可以迫使Delta测试不使用下标之间的约束。最后，Delta测试不传播来自一般的MIV下标的约束。因此在Delta测试结束时可能留下耦合的MIV下标。在这些情况下，可以使用下几小节中讨论的，如λ测试或Power测试这些更一般但代价很大的多下标依赖测试。 119

因为对耦合组中每个下标最多做一次测试，所以Delta测试的复杂度是下标数目的线性关系。然而约束可能多次传播到下标中。

3.4.2 更强有力的多下标测试

许多最早的多下标测试利用Fourier-Motzkin消去法，一种基于逐对比较线性不等式的线性规划方法。Kuhn[192]和Triolet等[261]阐述凸形区域中的数组访问，可以用Fourier-Motzkin消去法求交得到这种凸形区域。也可以用这种区域对整个程序段汇总存储访问。这些技术很灵活但代价高。Triolet发现使用Fourier-Motzkin消去法做测试要22到28次，比传统的依赖测试时间长[260]。

Li等人提出的λ测试，是Banerjee不等式的多维形式，λ测试同时检查受约束的实数值解[202]。λ测试构造下标的线性组合，消除一个或多个索引实例，然后得到的结果用Banerjee不等式进行测试。同时产生的实数值解存在，当且仅当Banerjee不等式在生成的所有线性组合中求得解。

λ测试能测试方向向量和三角形循环。它的精度也可以应用GCD或单索引准确测试生成的伪下标得到增强。然而，没有明显的方法扩展λ测试，以证明整数解同时存在。如果无约束的整数解存在，并且索引变量的系数全是1，0或－1，那么λ测试对二维下标是准确的[201]。然而，即使带有这些限制，对三维或更多的耦合维，它仍是不准确的。

Delta测试可以看成是λ测试的受限制形式，它以通用性换取更高的效率和精度。

因为对依赖测试线性下标函数等价于在循环界限内求同时存在的整数解，因此一种途径是使用整数规划方法。然而，使用这些方法必须谨慎，因为它们很高的初始化代价和很长的运行时间——它们通常有指数级复杂度——使其在产品系统依赖测试中不适合需要，产品系统在一次编译过程中要应用依赖测试几千次。然而，如果使用时小心地筛选出一些特定的情况，那么它们比起较简单的方法，能有效地消除更多地依赖。这种策略早期的两个例子是Wallace的约束－矩阵测试（修正的整数规划单纯形算法，以及Banerjee的多维GCD测试），它应用修改的整数高斯消去法建立一个紧凑的系统，其中所有的整数点约定为原依赖系统的整数解[34]。 120

增加依赖测试能力的第三种策略是将整数规划与Fourier-Motzkin消去法组合在一起，判断依赖方程是否存在一个整数解。Wolfe和Tseng的Power测试利用Fourier-Motzkin消去法将循环界应用到从多维GCD测试得到的稠密系统上[286]。Power测试的代价很大，但是适合用于提供精确的依赖信息，诸如非紧嵌循环中的方向向量，带有复杂界的循环，以及不涉及方向向量的约束中的依赖信息。

Maydan，Hennessy和Lam[207]提出一种依赖测试，已在SUIF系统中实现，它基于整数规划方法使用一串特殊情况的精确测试。如果所有的特殊情况测试失败，则用Fourier-Motzkin消去法做补充测试。为了增加这种测试的有效性，它们用一张表维护前面计算得到的测试结果，使能快速识别出重复的测试。

Pugh的Omega测试[228]是基于Fourier-Motzkin消去法对测试整数规划的扩展。自身是想判断依赖方程是否存在一个解，而不是去求出所有这样的解。虽然此过程有可能是指数复杂度，但已显示出有较低的复杂度，在许多情况中为多项式复杂度，往往更廉价的方法是精确的。Omega测试还能用来将整数规划问题投影到变量的一个子集上，而不仅仅是判定它们，它使精确计算依赖方向和距离向量成为可能。随着时间的推移，Omega测试已发展成一种通用工具，称为Omega计算器，用来化简和验证Presburger公式。这个系统能用来解决各种程序分析问题，包括消除伪依赖[229]。

3.5 实验研究

为了帮助读者理解在依赖测试的真实实现中考虑的某些折衷，我们介绍由Goff，Kennedy和Tseng[123]指导研究的一种较重要的依赖测试的若干结果，在PFC中使用了此依赖分析系统[20]，这是Rice大学的程序分析系统。

测试时，PFC使用了本章中描述的依赖分析系统，包括下标划分和下面的依赖测试：ZIV（包含符号）测试，强SIV（符号）测试，对特殊情况的弱SIV（弱-0，弱交叉，精确）测试，MIV测试（GCD，具有单循环三角形性质的Banerjee不等式），以及Delta测试（带有一般的约束交集，但仅传播距离约束）。对PFC应用的每种依赖测试，做了很多次度量研究，当时处理四组Fortran程序：RiCEPS（Rice大学编译评估程序集），Perfect和SPEC基准测试程序集[92，254]，以及两个数学库EISPACK和LINPACK。这些测试程序集包含了28个完整的程序和两个
121 大的子程序库，有986个子程序和99 440行Fortran 77代码。表3-1总结了每种依赖测试相对于其他测试的效果，以每一种测试在总的应用程序数、成功次数和无依赖个数中所占的百分比表示。对此表而言，“成功”是能消去一个或多个方向的任何测试应用。表中提供的百分比是汇总所有程序的结果。

表3-1 依赖测试的频率和效果

	SIV							
百分比	ZIV	强	弱-0	弱-交叉	精确	MIV	Delta	使用符号测试
全部测试	44.7	33.9	6.7	0.7	0.1	5.1	8.4	—
成功的测试	30.9	51.8	7.6	0.6	0.2	2.6	6.0	28.5
证明为无依赖	85.4	4.8	1.5	0.1	0.0	2.7	5.3	9.9
每个应用的成功率	43.9	97.0	71.7	47.9	87.7	33.0	45.4	—
每个应用的无依赖率	43.9	3.2	5.2	3.9	0.0	12.4	14.3	—

表3-1还显示了每种测试的绝对效果；即每种测试证明了无依赖或者成功地消除了一个或多个方向向量的应用百分比。

在这项研究中，PFC应用依赖测试74 889次（占所有下标偶的88%）。如果下标偶是非线性的（6%），或者在相同多维数组中对其他的下标测试已证明为无依赖，则不对它们做测试。在测试的所有数组引用偶中，多数下标偶是ZIV（45%），或强SIV（34%）。少数测试过的下标是MIV（5.1%）。多数成功的测试是ZIV和强ZIV测试组合（83%）。ZIV测试几乎占据了证明为无依赖的所有引用偶（85%）。

Goff，Kennedy和Tseng还报告说程序中多数下标是可分的。耦合下标（占总体的20%）集中在少数几个程序中，特别是EISPACK库，它约占所有耦合下标的75%。在8 499个耦合组

中发现大多数是两个；也遇到某些耦合组是三个。Delta测试约束求交算法精确地测试了6 570耦合组（占所有这样的耦合组的78%）。在376个测试用例中应用了距离约束传播（占耦合组的4.4%），仅在28个测试用例中将MIV下标转换成SIV形式。因此使用Delta测试精确地测试了6 198个耦合下标（82%），仅使用依赖距离的约束交集和传播。

结果表明SIV和Delta测试精确地测试了大多数下标。MIV测试，如Banerjee-GCD测试，只在所有下标的一小部分（5%）中需要用它们，但是对某些程序来说它们是重要的。许多成功的测试要求PFC有能力处理符号加常数（28.5%）。

在相关的研究中，Li等人说明在诸如EISPACK这样的库中，对于耦合下标测试，多下标测试可检测到的无依赖比逐个下标测试多36%的测试用例[202]。Shen等人[247]做了数组下标 122
和常规依赖测试的综合实验研究。

从这些统计中，有几个结论是显然的。第一，在提到的框架中，ZIV、强SIV、弱-0 SIV和MIV测试，以及某种形式的耦合下标测试，对精确的依赖测试是基本的。弱-交叉和精确的SIV测试几乎不会用到，但是当调用它们时，显示出它们能捕捉到重要的特殊情况。然而，在依赖测试的实现中，它们应当有较低的优先级。因为在超过28%的测试中使用符号测试，未计入允许符号循环界（几乎每个循环）的测试，符号处理也是基本的。

最后一个观察到的结果是，调用MIV和复杂的耦合下标测试次数很少，所以使用某种更强有力的（更大代价）测试是合理的，如3.4.2节中讨论的多下标测试。

3.6 各种测试的集成

图3-14和3-15包含原先在3.2节中描述的依赖测试算法的具体形式。关于此算法有几个重点。首先，正在做依赖测试的每一对数组引用，调用测试程序一次。做出关于哪个引用是源点和哪个引用是汇点的假设。然而，依赖测试程序完全有能力报告以“>”作为领头方向的方向向量。在这种情况，调用过程将颠倒对应此方向向量的依赖含意，并反转依赖的方向。

```
boolean procedure test_dependence(R1, R2, L, n, DVset)
  // R1和R2是在n个循环集中包围着的源点和汇点数组引用。
  // L是一组循环描述符 (LD1, LD2, ..., LDn)，其中每个描述符LDi是一个
  // 四元组 (Ii, Li, Ui, Si)，分别表示循环索引、下界、上界和步长
  // DVset是一个输出变量，表示两个引用之间求得的依赖的方向向量集。
  令m是引用偶中下标位置的数目;
  分配S[1:m]，下标偶的数组;
  for j := 1 to m do S[j] := (R1[j], R2[j]);
  for j := 1 to m do
    linear[j] := analyze_subscript(S[j], nx[j], In[j][0: nx[j]],
      a[j][0: nx[j]], b[j] [0: nx[j]]);
  // In是一个输出数组，使得In[i]是S[j]中循环索引的集合。

  Partition(S, P, np);
  // P是一个输出变量，它包含一组划分P[1:p]，其中每个P[k]是该划分中下标的集合

  DVset := {(*,*,...,*)};

  // 首先测试所有可分下标
```

图3-14 完整的依赖测试算法

```
        for k := 1 to n_p do
          if ‖ P[k] ‖ = 1 then begin
            depExists = test_separable(P[k], DVset);
            if not depExists then return depExists;
          end
        // 现在从头到尾迭代划分，再次测试耦合组。
        for k := 1 to n_p do
          if ‖ P[k] ‖ > 1 then begin
            InP := ∅;
            for all j∈P[k] do InP := InP ∪ In[j];
            depExists = Delta_test(P[k], DV, dV);
            if not depExists then return depExists;
            else merge_vector_sets(InP, DVset, DV);
          end
        return true;
      end test_dependence
```

图 3-14（续）

```
boolean procedure test_separable(P, DVset)
  // P是下标划分，它必须至少包含一个下标
  // DVset是迄今已有的方向向量集
  令j是P中单个下标划分;
  if linear[j] then begin
    if nx[j] = 0 then depExists = ZIV_test(a[j][0], b[j][0]);
    else if nx[j] = 1 then
      depExists = SIV_test(In[j], a[j][0:1], b[j][0:1], DV, dV);
  else
      depExists = MIV_test(In[j], a[j][0:nx[j]], b[j][0:nx[j]], DV, dV);
  if not depExists then return depExists;
  else begin
    merge_vector_sets(In[j], DVset, DV);
    return true;
    end
  end
  else return true;
end test_separable
```

图3-15 可分的下标测试

为了理解这一点，考虑下面的循环:

```
    DO I = 1,N
S0      T = B(I,J)
        DO J = 2,N
S1          A(J-1) = T
S2          T = A(J) + B(I,J)
```

```
        ENDDO
S3      C(I) = C(I) + A(J)
    ENDDO
```

如果我们检查由于对数组A的引用而引起的语句S_2对语句S_1的依赖，那么我们可能将引用A(J-1)作为R_1传递，假设为源点引用，将引用A(J)作为R_2传递，作为汇点引用。当测试程序返回时，将产生方向向量集(*,>)。因此方向向量的完全集是(<,>),(=,>)和(>,>)。第一个方向对应由外循环携带的真依赖，而后两个依赖分别对应于内循环和外循环携带的反依赖。在得到此结果后，调用的程序会报告实际的依赖集是一个具有方向向量(<,>)的真依赖，一个具有方向向量(=,<)的反依赖，和一个具有方向向量(<,<)的反依赖。注意，在两个反依赖中所有的方向已被反转，当一个依赖的含意被颠倒时，这是正确的操作。

此种产生并报告所有方向向量（即使是不合法的那些方向向量）的策略，对每个不同的引用偶仅许可调用测试程序一次。因此，如果程序中按依赖排序的引用集是$\{R_1, R_2, \ldots, R_N\}$，那么在驱动程序中我们可能看到下面的循环：

```
for i := 1 to N do
  for j := i to N do begin
    depExists := test_dependence(Ri, Rj, L, n, DVset);
    if depExists then begin
    // 记录每个不同的DV作为图中的一个依赖
    // 对非法的DV，逆转依赖的意思
    ...
    end
  end
```

预分析 *test_dependence*先要做的事情之一是将引用偶重组到下标偶集合中，引用中的每个位置上是一个下标偶。一旦完成此事，为每个下标位置调用例程*analyze_subscript*。该例程分析下标偶并判断此下标是否能表示成包含下述引用偶的循环索引的线性组合：

123 ≀ 125

$$\langle a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + a_0, b_1 i_1 + b_2 i_2 + \ldots + b_n i_n + b_0 \rangle$$

如果报告此下标偶是线性的，那么对所有的i，a_i和b_i的每一个值必须是常数，或是一个符号表达式，它的值在此循环嵌套中不变。（某些依赖测试程序还要求只有a_0和b_0是符号。）如果这些下标不能变成正确的形式，*analyze_subscript*报告此下标是非线性的，并因此不能测试它。用返回值*false*报告此事。

过程*analyze_subscript*（在此我们将不详细介绍）需要分析下标表达式和分解因子。这需要做大量工作。除了返回布尔值外，过程还返回在下标中实际找到的循环索引个数（存储在*nx*[*j*]中），以及存储在*In*[*j*][0:*nx*[*j*]]中的数组，它包含找到的循环索引的层号。它还要返回在下标中找出的*ai*和*bi*的实际值（存放在数组*a*[*j*][0:*nx*[*j*]]和*b*[*j*][0:*nx*[*j*]]中）。因此，如果在下标位置*j*找出层1，3和4的索引，则将分别在*a*[*j*][1]，*a*[*j*][2]和*a*[*j*][3]中找到系数a_1，a_3和a_4。因此，这种表示是相当地紧凑的。

一旦分析了这些下标，我们就可以调用过程*patition*，在图3-1中以简要的形式描述过它。注意*patition*必须返回一个划分的数组，每个划分是引用偶中下标位置的不相交子集，这里下标位置是1和*m*之间的整数。

最后，正式开始测试。首先，访问所有的下标，并用ZIV和SIV测试对线性可分的下标做

测试，如图3-15中过程*test_separable*所示。如果这些测试没有一个产生无依赖，那么测试耦合组。除ZIV测试外，每个不能证明无依赖的测试产生一个可能的方向向量集合DV。然后必须由图3-16中给出的*merge_vector_sets*过程将此集合与依赖聚集的集合合并。

```
procedure merge_vector_sets(In, DVset, DV)
  // In是在方向向量中表示的索引表
  // DVset是当前的方向向量集，它将被扩充
  // DV是表In中索引的距离向量集
  // 此例程仅在表In中索引的所有当前方向向量的列为"*"时被调用
  // 因此我们能简单地复制所有当前的方向向量到DV的基数中，并为每个
  //   新向量填入方向
  nV := ‖DV‖; nI := ‖In‖; nDVset := ‖DVset‖
  newDVset := new DVarray[nDVset*nV];
  lastNewDV := 0
  for i := 1 to nV do begin
    for j := 1 to nDVset do begin
      thisDV := DVset[j]的拷贝;
      for k := 1 to nI do thisDV[In[k]] := DV[i][k];
      lastNewDV := lastNewDV + 1;
      newDVset[lastNewDV] := thisDV;
    end
  end
  free DVset;
  DVset := newDVset;
end merge_vector_sets
```

图3-16 方向向量和距离向量合并

方向向量合并 根据每一种测试应用于下标的单个划分上这样的事实，合并过程是比较容易的。回顾一下，每个划分包含惟一的循环索引集。即没有这样的划分，它包含的循环索引是在任何其他划分中找到的。所以，我们能保证当我们实施合并时，在正被测试的划分中找到的索引集的相应位置上，依赖的所有方向向量有"*"项。所以，我们基本上能做笛卡儿积，产生一个方向向量集，它的大小等于划分的向量数目乘以迄今已有的这样依赖向量的数目。

我们用一个例子说明合并过程，假设我们有下面的代码:

```
   DO I = 1,N
     DO J = 2,N
       DO K = 1,N
S        A(I,J+1,J+K) = A(I+1,J,J+K) + C
       ENDDO
     ENDDO
   ENDDO
```

对语句S中的两个引用进行测试，我们从系统设定的方向向量(*,*,*)开始。第一个下标是可分的，所以先对它应用SIV测试。返回方向">"和距离-1。合并过程简单地将此方向插入对应I（外循环索引）的位置上，产生(>,*,*)。接下去测试由下标2和3组成的耦合组。应

用SIV测试到第二个下标上，产生对应J-循环的方向“=”。这意味着J的值在源点和汇点上是相等的。当它被代入第三个位置时，J的值可以从方程中删去，我们同样得到K-循环的方向“=”。 126~127

因此由J-循环和K-循环组成的划分的方向向量是(=,=)。当它与当前的方向向量集(>,*,*)合并时，得到的结果是(>,=,=)。由此我们看到S的自依赖是一个反依赖，方向向量为(<,=,=)。注意，合并总是在当前方向向量集中具有“*”的位置上发生。

距离和消除条件 虽然我们已经讨论了方向向量的处理，但是还没有说明如何扩展这些算法来处理距离和其他诸如交叉阈值和消除条件这样的注释。在多数的依赖测试程序中，这些附加的量是对方向向量的注释。在前面的例子中，惟一的未知距离的方向是与距离-1关联的“>”。此信息可以作为对此方向的注释附加上，它含有一个布尔变量（指示有一个关联的距离），一个类型（指示距离的种类）和一个值。所以，实际上方向向量看上去有点像：

```
(>[fixed: -1],=,=)
```

可以对符号距离和消除条件做类似的处理。

注意，依赖的消除条件是所有具有这种条件的方向向量的消除条件的析取。当然，ZIV测试产生的消除条件不与任何特定的方向有关。可以将它作为一个整体附加到方向向量上。这种表示利用了这样的事实：距离和消除条件普遍比方向少，所以在每个循环索引位置上不必为这样的注释分配空间。

标量依赖 最后一个实用问题是与标量变量关联的依赖表示。为了理解此问题，考虑下面的例子：

```
      DO I = 1,N
        DO J = 1,N
          DO K = 1,N
S           T = T + A(I,J,K)
          ENDDO
        ENDDO
      ENDDO
```

因为标量变量T不是任何变量的索引，我们可以看出它有一个依赖，是由三个循环的每一个循环携带的。事实上，对T的两个引用引发出三类依赖：由于左边出现的T引起输出依赖，从右边的出现到左边出现的反依赖，以及从左边出现到右边出现的真依赖。方向向量集如下： 128

(1) 输出依赖：(<, *, <),(<, *, >),(=, <, <),(=, <, >)

(2) 反依赖：(<, *, <),(<, *, >),(=, <, <),(=, <, >)

(3) 真依赖：(<, *, <),(<, *, >),(=, <, <),(=, <, >)

换句话说，对每一种依赖有8个不同的方向向量。如果我们纪录下这些方向向量的每一种不同的依赖，正如我们因数组引用而做的依赖，那么空间需求将很快变得不可管理。事实上，这在PFC的一个早期版本中发生过，其中一个相当小的子程序，引起了最大达64 000个项，超出了依赖表的容量。

为了避免发生此类问题，用一种概要形式表示标量依赖更好一些，这种形式指明携带依赖的循环集合，以及是否有可能是循环无关的依赖。对于前面的例子，我们应当纪录下三个循环可能携带依赖，但是不可能是循环无关的依赖。这能表示成循环层的区域加上一个布尔变量。

3.7 小结

依赖测试是这样一个过程：判断在给定的一组循环中，对相同变量的两个引用是否可能访问相同的存储单元。如果一个依赖是可能的，那么测试过程必须标识方向向量集，在某些情况下，还要标识出距离，它们描述给定的引用偶之间所有可能的依赖。

在多数情况下，依赖测试相当于确定丢番图方程组在循环嵌套的迭代界限内是否有解，下标的引用出现在此循环嵌套中。在多数情况下，由于仅仅集中于下标能描述成循环归纳变量的仿射组合上，使此问题得到了简化。即使这样，由于在线性表达式中存在符号系数，问题仍然是复杂的。

在这一章我们介绍了一个依赖测试过程，它使用基于情况的分析，保证对大多数经常出现的情况能得到有效而精确的处理。此过程由下列步骤组成：

（1）划分下标（这里下标指一对引用中匹配的下标位置偶）成可分的和最小耦合组。如果在一下标中出现的归纳变量，没有一个在其他的下标中出现，则称此下标是可分的。一个
129 耦合组是最小的，如果它不能划分成两个具有不同的索引集合的非空子集合。一旦完成了划分，那么每个可分的下标和每个耦合组有完全不相交的索引集。然后，能分离开测试每个划分，并得到合并的距离向量或方向向量而不丢失精度。

（2）将每个下标位置分类成ZIV（含0个归纳变量），SIV（含一个归纳变量）或MIV（含多个归纳变量）。

（3）对每个可分的下标，根据下标的复杂性应用适宜的单下标测试（ZIV，SIV，MIV）。这将产生无依赖或在该下标中出现的索引的方向向量。如果证明为无依赖，则对其他位置不需要做进一步测试。

（4）对每个耦合下标，应用诸如Delta测试这样的多下标测试，产生此组中出现的索引的方向向量集。

（5）如果任何测试的结果是无依赖，则没有依赖存在，无需做进一步的测试。否则合并在前几步中计算的所有方向向量，使成为两个引用的一个方向向量集。

这一章对ZIV和SIV测试做了详细的描述，并详细地讨论了GCD测试和Banerjee不等式——一个快速而又精确的MIV偶测试。为使测试真正有效，任何测试过程必须处理符号系数和梯形循环。还介绍了测试的扩展，以便处理这些情况。

3.8 实例研究

PFC和Ardent Titan编译器都是按照这一章描述的依赖测试方法实现的。原始的PFC向量化程序仅对含有依赖偶的一个循环携带的下标做测试。例如，如果一依赖偶包含在三个循环中，它会对下面的方向向量做测试：

（1）（<，*，*）——由最外层循环携带

（2）（=，<，*）——由次最外层循环携带

（3）（=，=，<）——由最内层循环携带

对每个携带者循环，还构造阈值（距离）信息，并测试交换最内层与次最内层循环是否
130 合法。在三个循环的嵌套中，这就组成了对方向向量（=，<，>）的测试。所有这些测试的导出使用三角形Banerjee不等式和GCD测试。

稍后增强的PFC，包含ZIV，全部SIV和MIV测试，以及如本章中所描述的受限制的Delta

测试版本。三角形Banerjee不等式限制在任何循环上界中只包含最多一个外层循环索引[170]。

这个PFC的第二版本计算完整的方向向量，只要可能，还计算距离向量。如果在表达式中符号量始终是循环不变量，则将符号表达式记为一个距离。在符号依赖测试中可以使用这些距离。

PFC还记录了消除条件，并利用它们插入运行时的依赖测试。另外，在一个称为PTOOL[18]的交互程序设计工具中，用这些条件来消除依赖，PTOOL使用了PFC作为依赖测试的引擎。ParaScope Editor[31，79，132]在合并它自己的测试系统之前，也使用了PFC做测试依赖。

PFC的一个重要特色是支持过程间分析，使之有可能用第11章中描述的方法做过程间的依赖测试。

3.9 历史评述与参考文献

最早的依赖测试工作集中在从强SIV下标导出距离向量[191，195，219]。Cohagan[77]描述了一种测试方法，可从符号上分析一般的SIV下标。Banerjee和Wolfe[33，283]研究了目前的单索引精确测试形式。Callahan描述用在PFC中的可分性方法[51]。

对于MIV下标，GCD测试可用来检查无约束的整数解[32，168]。Banerjee不等式提供有用的通用单下标测试，求受约束的实数解[33]。另外，还对它进行修改用来提供许多不同类型的依赖信息[21，34，49，168，170，283]。研究表明在许多常见的情况下，Banerjee不等式是精确的[32，184，201]，虽然结果还没有被扩展到方向向量或复杂的迭代空间。

由Kong等人研究的I-测试集成GCD和Banerjee测试，通常能证明整数解[188]。Gross和Steenkiste提出一个有效的区间分析方法，用来计算数组的依赖[126]。不过他们的方法不处理耦合下标，并由于没有计算距离向量和方向向量，不适合于多数循环变换。

Lichnewsky和Thomasset描述在VATIL向量化程序中使用的符号依赖测试[203]。Haghighat和Polychronopoulos提出的流分析框架有助于符号测试[128]。

执行条件也可以用来精炼依赖测试。Wolfe的All-Equals测试在循环内的控制流上检查无效的循环无关依赖[283]。Lu和Chen的子域测试结合来自循环体内条件的有关索引的信息[205]。Klappholz和Kong已对Banerjee不等式做了扩展，使之做相同的事情[183]。 131

测试多维数组的早期方法强调同时性，包括对每一维求交方向向量和线性化[49，121]；在许多情况中已证明它们是不精确的。真正的多下标测试提供的精确度，是以同时考虑所有下标的性能损失为代价。这些测试已在3.4.2节中讨论。本章描述的Delta测试是由Goff，Kennedy和Tseng建立的[123]。

习题

3.1 在下面的每个例子中，假设你正在测试语句S到自身的依赖。哪些下标位置是可分的？哪些是耦合的？用本章描述的依赖测试过程哪一种依赖测试将会应用到每个位置上？

a.
```
      DO K = 1, 100
        DO J = 1, 100
          DO I = 1, 100
S           A(I+1,J+1,K+1) = A(I,J,1) + C
          ENDDO
        ENDDO
      ENDDO
```

```
b.      DO K = 1, 100
           DO J = 1, 100
              DO I = 1, 100
S                A(I+1,J+K+1,K+1) = A(I,J,K) + C
              ENDDO
           ENDDO
        ENDDO

c.      DO K = 1, 100
           DO J = 1, 100
              DO I = 1, 100
S                A(I+1,J+K+1,I) = A(I,J,2) + C
              ENDDO
           ENDDO
132     ENDDO
```

3.2 在下面的例子中，为循环中所有可能的依赖计算完整的方向向量集。具体涉及每种情况的依赖类型。描述对此例用本章中的依赖测试程序将使用的测试。

```
     DO K = 1, 100
        DO J = 1, 100
           DO I = 1, 100
S1            A(I+1,J+4,K+1) = B(I,J,K) + C
S2            B(I+J,5,K+1) = A(2,K,K) + D
           ENDDO
        ENDDO
     ENDDO
```

3.3 对下面的例子构造有效的消除条件。

```
a.      DO I = 1, 100
S          A(I+IX) = A(I) + C
        ENDDO

b.      DO K = 1, 100
           DO J = 1, 100
              DO I = 1, 100
S                A(I+1,J+1,K+1) = A(I,JX,K) + C
              ENDDO
           ENDDO
        ENDDO

c.      DO K = 1, 100
           DO J = 1, 100
              DO I = 1, 100
S               A(I+1,J+K+JX) = A(I,J) + C
              ENDDO
           ENDDO
        ENDDO
```

3.4 3.3.2节考虑梯形迭代空间对强SIV依赖测试的影响。说明将3.3.3节的梯形Banerjee不等式用于同样的例子上你会获得什么结果。结果相同呢还是不同？为什么？

3.5 3.3.2节也考虑迭代的梯形区域中的弱-0 SIV测试。其结果与你应用梯形Banerjee不等
133 式得到的结果相同吗？

3.6 用引理3.2证明，在区域$L_i^x \leqslant x_i \leqslant U_i^x$，$L_i^y \leqslant y_i \leqslant U_i^y$中有

$$-a_i^- U_i^x + a_i^+ L_i^x - b_i^+ U_i^y + b_i^- L_i^y \leqslant a_i x_i - b_i y_i$$
$$\leqslant a_i^+ U_i^x - a_i^- L_i^x + b_i^- U_i^y - b_i^+ L_i^y$$

（这是引理3.3证明中的情况4。）

134

第4章

初等变换

4.1 引言

第3章介绍的多数依赖测试要求下标表达式是循环归纳变量的线性函数或仿射函数，下标表达式的系数是已知常数，并且最多只有一个符号加常数。如果想要为这些测试构造出一个精确的依赖图，那么程序中的多数下标必须取这种形式。

不过，编写程序通常并未考虑依赖测试。程序员写代码倾向于利用他们偏爱的Fortran语言或它的编译器的不同版本。另外，许多特有的实践已经被用来攻击编译器优化策略中的弱点，结果是得到的代码往往可击败最好的依赖分析器。下面是一个有代表性的例子：

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      KI = KI + INC
      U(KI) = U(KI) + W(J)
   ENDDO
   S(I) = U(KI)
ENDDO
```


此循环嵌套中仅有两个下标——W(J)和S(I)——是循环归纳变量的仿射函数。特别地，表达式U(KI)（在每一种依赖测试中都会涉及到这种表达式）不能用所写的这种形式测试，因为KI在循环内变化。如果要想将第3章中的那些测试成功地应用到此例上，必须变换此代码。

为了解决这类问题，在依赖测试之前可以先做若干变换，使测试的目标更精确。这些变换使更多的下标能接受第3章中描述的精确测试。

此过程中，一个重要的变换是*归纳变量替换*。在上例中，变量INC在内循环是不变的。所以在J-循环中对KI的赋值是在每次循环迭代中将它的值增加一个常量。此增量使KI成为一个*辅助归纳变量*；本质上它是另一个循环变量，它跟踪正规的循环索引(J)，但是具有不同的增量或起点。归纳变量替换将每个辅助归纳变量的引用变换成一个循环索引的直接函数。对内循环应用这样的变换产生

```
      INC = 2
      KI = 0
      DO I = 1, 100
         DO J = 1, 100
            ! 已被删去: KI = KI + INC
            U(KI+J*INC) = U(KI+J*INC) + W(J)
         ENDDO
S1       KI = KI + 100*INC
         S(I) = U(KI)
      ENDDO
```

在循环内，KI的使用已变成循环归纳变量的一个函数；删去了增量（变为一个CONTINUE语句）；并且插入了S_1，用来在循环外对KI置正确的值。注意，在循环中仍有一个对KI的引用，但是现在它含有该变量的循环不变的初始值。

由于在外循环插入一个KI的增量语句，此变换已使KI变成一个与该循环有关的辅助归纳变量。需要第二次应用归纳变量替换，完全消除作为辅助归纳变量的KI:

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      U(KI+(I-1)*100*INC+J*INC) = &
136       U(KI+(I-1)*100*INC+J*INC) + W(J)
   ENDDO
   ! 已被删去: KI = KI + 100*INC
   S(I) = U(KI+I*(100*INC))
ENDDO
KI = KI+100*100*INC
```

现在所有下标都是循环归纳变量的仿射函数，虽然系数都是符号的。

化简此程序的下一个步骤是识别在循环中使用的INC和RI的值实际上恰好是常数值。循环外的值的常数传播将消去这些符号量，化简后得到

```
INC = 2
! 已被删去: KI = 0
DO I = 1, 100
   DO J = 1, 100
      U(I*200+J*2-200) = U(I*200+J*2-200) + W(J)
   ENDDO
   S(I) = U(I*200)
ENDDO
KI = 20,000
```

虽然现在此例对依赖测试来说是一种可处理的形式，但是还有一些无用的冗余——某些常数值赋值可能绝不会用到。特别地，对KI和INC的赋值仅当在程序中稍后使用它们时才是需要的。不让这些赋值浪费代码空间和执行时间，我们可以通过死代码消除发现并删去它们。假设在此循环之后不再使用KI和INC，将产生

```
DO I = 1, 100
   DO J = 1, 100
      U(I*200+J*2-200) = U(I*200+J*2-200) + W(J)
   ENDDO
   S(I) = U(I*200)
ENDDO
```

事实上，许多程序含有类似于上例的代码。因此变换代码的能力对依赖测试的成功是重要的。这一章介绍对程序实施初等变换的方法，变换使程序适合依赖测试的需要。因为在有
137 控制流时，这些变换变得更困难，我们将延缓对控制的讨论到第7章。

4.2 信息需求

第3章讨论依赖测试方法时，假设了循环的一些性质。例如，如前所述的多数依赖测试假

设循环的步长为1。为了实现这些测试，必须收集有关循环符合这些需求的信息，并做一些尚未做的事情。另外，实现前一节描述的变换需要有关程序中数据结构和使用的知识。这些必需的信息包含下面所列的各项：

（1）循环跨距：虽然依赖测试能重新计算非单位跨距，但是，如果跨距是1的话，那么依赖测试最容易实现。

（2）循环不变量：如果编译器想要寻找机会做辅助归纳变量替换，则编译器必须要有能力识别循环不变的变量和表达式。

（3）常数值赋值：对常数传播来说，识别常数值赋值是一个重要的开端。

（4）变量的使用：传播一个常数值赋值需要知道哪些语句使用由赋值定义的变量。类似地，死代码识别涉及到识别那些输出绝不会被使用的语句。

收集这些信息的过程涉及众所周知的标量优化技术。特别地，最后这三项通常归标量数据流分析计算。本章余下的部分对程序变换和数据流分析策略需要计算的必要信息提供一个简介。

4.3 循环正规化

为使依赖测试过程尽可能简单，许多先进的编译器对所有的循环正规化，使它们从下界为1运行到某个上界，跨距为1——基本上使所有的循环“数点”执行迭代，用新的归纳变量的仿射函数代替对原循环归纳变量的引用。这种变换称为循环正规化。

作为原始定义，对此变换特别地使用了“循环正规化”这个术语。然而，依赖测试需要很多有关循环的信息——例如，哪些循环围绕哪些语句，哪些归纳变量控制哪些循环，以及什么样的循环界控制每个循环。因为必须在真正的测试之前的某个时刻收集这些信息，并因循环正 138
规化无论如何必须要检查所有的循环，所以许多编译器将实现循环正规化归为一般的正规化和信息收集遍。结果，该术语往往不仅用于特定的变换，还用于收集有关循环信息的总的遍中。对变换或总的遍我们都将互换地使用“循环正规化”；上下文应当能澄清它所指的意思。

图4-1介绍一个循环正规化的简单算法，它能应用到任何带有整型控制参数的Fortran 90 DO循环上。它用1替换被正规化的原循环索引变量，并用等价表达式代入新的索引变量中替换对原始索引的引用。注意，为简化起见，我们将小写变量i看成是编译器引入的变量，它不同于大写的I，即使多数Fortran 90的实现将它们看成是相同的。

```
procedure normalizeLoop(L0)
  // L0是被正规化的循环
  令i是惟一的由编译器生成的循环索引变量;
S1: 用校准的循环头
      DO i = 1, (U - L + S)/ S
  替换L0的循环头
      DO I = L, U, S;
S2: 用i * S - S + L替换循环中对I的每一个引用;
S3: 在循环结束后立即插入一个最后完成的赋值语句
      I = i * S - S + L;
end normalizeLoop
```

图4-1 循环正规化算法

正确性 为了说明*normalizeLoop*具有要求的效果，我们只需说明在步骤S_1中计算的循环界，在步骤S_2中计算的替换的索引值，以及在步骤S_3中计算的最后定值是正确的。即替换后的循环具有相同的迭代数目，并且替换的表达式与原循环归纳变量是等价的。

在Fortran 90循环中，恰好在循环外对循环索引置循环下界并且做测试以保证它小于上界。在后继的迭代中当前的索引值加上增量，如果结果小于或等于上界，则执行循环体。因此对新的归纳变量i满足

139

$$L+(i-1)*S \leqslant U$$

的每一个值，执行循环体。换言之，对i满足

$$i*S \leqslant U-L+S$$

的每一个值，执行循环体。

因此在循环迭代区域内i的最大值必须是小于或等于

$$(U-L+S)/S$$

的最大整数，这里将除解释成产生一个实数。然而，在Fortran中整数除产生的最大整数小于等于真正的商。所以在步骤1中计算的循环计数是正确的。

接下来我们必须说明原循环归纳变量I替换后的值有如I在每次迭代中相同的值。显然在第一次迭代中（当i=1时）值是正确的，因为该值是L。如果在一次迭代中值是正确的，那么在后继迭代中的值必定是正确的，因为在后继迭代中替换的值大于正确迭代中替换的值，大出的值为循环增量S。因此步骤2中的替换是正确的。

在Fortran循环出口处，循环归纳变量取值为该变量在最后一次循环迭代的值加步长S的值。根据上述推理，它必定是

$$i*S-S+L$$

其中，i是生成的归纳变量的出口值。倘若所有下标值与正规化循环中的值相同，那么正规化总是安全的，因为它不改变循环中任何语句的顺序。因此所有依赖关系得到保持。而且，原归纳变量的值在循环之后被正确地重构。所以变换维持原程序的含意。

循环正规化提供若干优点。除了简化依赖测试外，它创建等价于该循环迭代计数的循环索引。这使得像归纳变量替换这样的变换更容易实施。

然而，循环正规化也有某些重大的缺点。最突出的缺点是有歪曲依赖性质的可能性。下面的循环嵌套说明这种缺点：

```
      DO I = 1, M
         DO J = I, N
S1          A(J,I) = A(J,I-1) + 5
         ENDDO
140   ENDDO
```

S_1到自身的真依赖有方向向量(<,=)。对内循环正规化产生下面的代码：

```
      DO I = 1, M
        DO J = 1, N-I+1
S1         A(J+I-1,I) = A(J+I-1,I-1) + 5
         ENDDO
      ENDDO
```

因为这是一个安全的重排序变换，所以依赖必定仍然存在。然而，方向向量已从(<,=)变成(<,>)。此变换不是完全无害的。例如，考虑交换I-循环和J-循环（5.2节对循环交换有充分的讨论）。循环交换对方向向量的影响是互换对应循环交换后的项。其结果是，将循环交换应用到原方向向量(<,=)上，产生一个新的方向向量(=,<)，它维持依赖，并且显然是安全的。应用循环交换到方向向量(<,>)上，产生一个新的方向向量(>,<)，它逆转依赖，显然是不安全的。因此，正规化已使交换无效，虽然我们将会看到能通过其他的变换克服这个问题。

当原循环中步长是符号时，正规化也会带来问题。在这种情况下，正规化为原归纳变量引入的表达式中产生一个符号系数，使得在任何出现这些表达式的下标上测试依赖变得很困难。在这样的情况下，实际上应用一个简单的正规化版本更好些，它取步长正好是1。这会丢去精度，但使测试下标成为可能。事实上，符号下标在运行时的值往往等于1，如在使用LINPACK库的应用程序中那样，其中用户选择的跨距的大多数几乎是不必要的。

尽管存在这些不足，正规化是一个有用的变换，并且已应用到几乎所有的向量化编译器中。虽然不是必需的，但是为简单起见本书中的例子假设已实行了正规化。然而要注意，第3章中依赖测试的所有工作都是对任意下界的。

4.4 数据流分析

数据流分析的目的在于理解程序中数据元素是如何建立和使用的，为的是支持维持程序含意的优化变换。关于标量数据流分析有大量的文献（见Kennedy[169]，Muchnick[218]），我们不打算在此充分地表述。代之，我们对构造辅助数据结构（定义-使用链和静态单赋值形式）的方法做一简单的介绍，它们对本章以及本书后面讨论的依赖分析和变换的支持都是有用的。 141

4.4.1 定义-使用链

所有描述的初等变换，处处需要能够容易地得到从一个变量的定义到达可以消费该定义值的所有单元。定义-使用链是一个精确定义的数据结构，用它容易实现这样的操作：

定义4.1 定义-使用图是一个图，它包含这样的边：程序中从每个定义点到运行时该变量的每一个可能的使用有一条边。

我们使用术语“定义-使用图”代替更传统的“定义-使用链”是因为“图”更正确地刻画它所包含的信息的性质。定义-使用图本质上是程序中真依赖的标量版本。

在代码的单个直线块[⊖]中构造定义-使用边是非常简单的。你在基本块中按顺序经过每个语句，注视由每个语句定义的变量（也称为它的定义），以及被每个语句使用的变量（称为它的使用）。对每个使用加一条边到定义-使用图中，此边从基本块中变量最后暴露的定义指向该使用——换言之，从那个定义到达这个使用。每当遇到一个变量的新定义，新定义消除已有的定义，使得后面的使用只连接到新定义而不是老的定义上。当到达块的结尾处时，完成局部图。

除局部图外，基本块计算还产生若干有用的集合，它们刻画基本块的行为。这些集合如下：

⊖ 这种块也称为基本块。一个基本块是语句的最大组，执行组中一个语句，当且仅当每个语句被执行。换句话说，除了恰好在它的开始处或它的结尾处，没有控制流进入或退出基本块，更多有关的知识见Kennedy[169]。

uses(*b*)：在基本块*b*中使用的所有变量集合，在此基本块内这些变量没有预先定义。换句话说，在此块内它们是不“满足的”使用，所以对从前面的基本块到达该基本块的任何定义
142 来说它们是暴露的。

defsout(*b*)：基本块*b*内所有定义的集合，这些定义在块内未被消除。换句话说，它们是能到达*b*之外的其他基本块的所有的定义。

killed(*b*)：被基本块*b*内其他定义消除的定义变量的所有定义的集合。从其他基本块来的任何定义，如果它们是在*killed*(*b*) 中，将禁止它们试图“通达”*b*。

这些集合为构造整个程序（而不是个别基本块）的定义-使用图提供基本工具。给定这些局部集合，全局边计算需要一个遗漏的集合*reaches*(*b*)——从所有基本块（包括*b*）有可能到达*b*的所有的定义集合。对于任何设定的块*b*，可以这样获得所有的全局定义-使用边：对于*reaches*(*b*) 中每个元素达到*uses*(*b*) 的所有相应的元素，就加上一条边。其结果是，如果我们能找到一种对所有的块计算*reaches*的方法，那么就能建立定义-使用图。

为了理解如何全局地计算*reaches*，首先看一下非常受限制的图上的问题是有好处的：一个基本块*b*和它的某些前驱，他们之中的每块能通过控制流的某种形式到达*b*。在这个简单的图中，沿着任何一个前驱*p*，*reaches*(*b*) 是到达*p*的所有定义的集合（*reaches*(*p*)），并且在*p*内不是被消除的（非*killed*(*p*)），加上那些*p*中到达*p*的出口的定义（*defsout*(*p*)）。更形式化地表示，*reaches*(*b*) 能用下面的方程定义：

$$reaches(b) = \bigcup_{p \in P(b)} (defsout(p) \cup (reaches(p) \cap \neg killed(p))) \tag{4-1}$$

在上面描述的受限制图上求解此方程显然是简单的，但是在一般的控制流图上求解就没有那么简单了。复杂性是由于对每个块*b*计算*reaches*可能立即改变所有其他的*reaches*（包括*b*自身），因为*reaches*(*b*) 是其他*reaches*方程的输入。获得正确解需要同时解所有的单个方程。

所幸的是，此问题的基本数学方法保证在程序中每个结点上以迭代方式应用方程（4-1），最终将会以一个稳定的解终止，此解是同时求所有方程获得的一个精确解。对它的证明超出本书的范围[⊖]。然而图4-2介绍的结果产生概念上解数据流方程组的简单迭代方法。虽然此方法的实现是最简明的，但是它的渐近最坏情况界限是最复杂的。算法以自然的初始近似开始求解——作为方程（4-1）输入的所有*reaches*集皆为null。然后此方法在所有顶点上反复地迭代，
143 直到它到达一个不动点（每遍通过各基本块不再产生改变的一个点）。得到的解为全局解。

在许多情况下，如果在控制流图中选择正确顺序访问顶点，则收敛会得到加速。通常最常用的顺序是深度优先顺序——假定从程序入口结点开始，这种序以深度优先搜索访问各个结点。

过程*iterate*不仅是（甚至在最坏的情况是最快的）构造全局*reaches*集的方法，而且由此能得到定义-使用链。一般来说它需$O((N+E)/N)$ 个集合操作，其中N是图G中的顶点数，E是边数[169，164]。然而，它是最简单的实现方法，并且实际上收敛得非常快[⊜]。对解数据流问题其他方法有兴趣的读者，可以参阅Kennedy的综述[169]或Muchnick的教科书[218]。

一旦构造了定义-使用图，归纳变量替换、常数传播和死代码消除等问题就能着手求解。

⊖ 更多的细节见Kennedy[169]。

⊜ 准确地说，算法实际需要$O((N+E)D(G))$，其中$D(G)$ 是一个图的“循环连通性”，它与程序中循环嵌套深度有关。对多数程序来说，$D(G)$ 比N小得多。

在一个编译器中，通常按此顺序解决这些问题。然而依据理解的难度，归纳变量替换最难，而死代码消除最简单。余下的几节按相反的顺序讨论这些问题。

```
procedure iterate(G)
  // G=(N, E) 是输入控制流图，其中
  // N是基本块集合
  // E是控制边集合
  // defsout(b) 是在b中到达b的出口的定义集
  // killed(b) 是由于一个中间赋值而不能到达b的末尾的定义集
  // reaches(b) 是到达块b的定义集

  for each b ε G do reaches(b)=Ø

  // 对不动点迭代
  changed := true;
  while changed do begin
    changed := false;
    for each b ε N do begin
      newreaches := reaches(b);
      for each p ∈ predecessors(b) do
        newreaches := newreaches ∪ (defsout(p) ∪ (reaches(p) ∩ ¬killed(p)));
      if newreaches ≠ reaches(b) then begin
        reaches(b) := newreaches;
        changed := true;
      end
    end
  end
end iterate
```

图4-2 到达定义的迭代方法

4.4.2 死代码消除

死代码是这样的代码，它的结果绝不会在任何有用的语句中使用。先以一种近似的说法，可将“有用的语句”简单地说成是输出语句，因为这种语句仅是执行从外部直接看到结果的操作语句。当然，任何这样一种语句也是有用的，它们计算为输出语句使用的值如同计算为有用语句使用的值的语句一样。这种粗糙的定义构成死代码消除算法的基础；寻找所有的有用语句并删除所有的其他语句。图4-3中介绍的算法以全部绝对有用语句（即输出语句，控制流语句和输入语句[⊖]）的*worklist*集合开始工作。然后，算法反复加入语句，这对计算*worklist*当前成员是必需的，直至不再需要加入语句。到那时，*worklist*包含程序中所有有用语句，并可以将所有其他的语句删除。

此算法的能力经常在用户漫不经心编写的基准测试程序上显示出来。在一基准测试程序中，程序员往往关心计算需要的时间，而不是计算的结果。过去有些基准测试程序的设计者记住了打印执行时间，但忘记了打印任何结果。当编译器实施死代码消除处理时，整个计算变成了死代码，得到极快的执行时间——生成的代码通常将读时钟一次，再读一次，取其差

⊖ 如果任何输入语句未被执行的话，即使它的结果没有用处，那么其他的输入语句很容易接受到错误的值。

并将它打印出来。

```
procedure eliminateDeadCode(P)
  // P是在其中做死代码消除的过程
  // 假设P中的所有语句存在定义-使用链
  令worklist := {绝对有用的语句};
  while worklist ≠ Ø do begin
    x := worklist的任意一个元素;
    标记x为有用;
    worklist := worklist - {x};
    for all (y, x) ∈ defuse do
      if y未被标记为有用的 then worklist := worklist ∪ {y};
  end
    消除未被标记为有用的每个语句;
end eliminateDeadCode
```

图4-3 死代码消除

应当指出，像图4-3中那样的死代码消除算法，对处理确定控制流的表达式往往会遇到麻烦。即使这些表达式不产生在其他语句中使用的值，但它们控制着某些语句的执行，依据确定控制流的测试结果能绕过这些语句的执行。简单的死代码消除系统将每个条件语句标记为绝对地有用，然后如果它们控制的所有语句被删除的话，则删除条件语句。一个更加复杂的方法是扩展定义-使用链，使之带有第7章中描述的控制依赖边。如果这样做，图4-3中的算法将产生精确的结果。下一节介绍一种更加复杂的基于定义-使用的算法：常数传播。

4.4.3 常数传播

常数传播试图用常数值替换所有那些能证明在运行时具有常数值的变量。分析此问题的一种方法是利用图4-4中描绘的常数值格架。此格架表示收集到的有关变量的信息。在顶层（“未知”）表示有关一个特定变量没有信息可以利用。中间层表示变量有一个已知的常数值——常数传播希望的状况。注意，这一层是无限宽的，因为能出现任何整常数（假定我们正在传播整数；我们同样能在这层放置浮点数）。底层表示已知一个变量取多个值，或者在编译时不知道它的常数值。虽然此格架有无限的宽度，但它有有限的深度，因为最长的向下链长度为2。

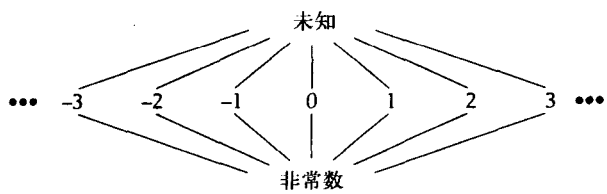


图4-4 常数传播格架

常数传播算法的基本想法是从格架顶层元素接近的每个变量开始。每当发现一个语句有常数输出时，则它的输出变量值被降低为常数值。然后用定义-使用边安放所有使用该输出值的指令。在每一使用处，用格架中老值和新值匹配来调整输入变量的近似值。图4-5给出了完

整的算法。

```
procedure propagateConst(P)
  // P是在其中做常数传播的过程
  // valout(w, s) 是输入w到s的最近似值
  // valin(v, s) 是从s输出到v的最佳值
  // μ(s)（输入到s）是它的输入格值上语句s的符号解释的结果。此输出是此语句的输
  // 出的格值。
  for all程序中的语句do begin
    for each s的输出v do valout(v, s) := unknown;
    for each s的输入w do
      if w是一个变量then valin(w, s) := unknown;
      else valin(w, s) := w的常数值;
  end;
  worklist := {所有常数形式的语句，例如，x=5};
  while worklist ≠ Ø do begin
    从worklist中选取并删除任意语句x;
    令v指称x的输出变量;
    // 语句x的符号化解释
    newval := μ(x) (valin(v, x), 对x的所有输入v);
    if newval ≠ valout(v, x) then begin
      valout(v, x) := newval;
      for all (x, y) ∈ defuse do begin
        oldval := valin(v, y);
        valin(v, y) := oldval ∧ valout(v, x);
        if valin(v, y) ≠ oldval then worklist := worklist ∪ {x};
      end
    end
  end
end propagateConst
```

图4-5 常数传播算法

直观上，常数传播以所有这样的赋值语句集合开始：它们对一个变量赋予一个常数值。从此集合中选取一个元素；用定义-使用边求出该定义能到达的所有输入。对每一个这样的输入，向后追踪定义-使用边，求出能到达一特定输入的所有定义。如果所有定义有相同的常数值，那么用该值替换此输入；否则不知其为常数。如果输入被替换，那么可能建立一个新的常数赋值；如果做了以上这一步，则将此赋值语句加到*worklist*集合中。

常数传播需要的时间是$O(N+E)$，其中N是程序中语句的数目，E是定义-使用图中边的数目。为了看清这一点，注意一个语句最多只有两次被放入*worklist*中，仅当它的输出值在格中被降低时才将它加到*worklist*中。因为最长的向下链不多于两条边，输出值能最多下降两次。因此主循环体执行$O(N)$次。最内层forall循环在定义-使用边上迭代，这些边从单个语句发射。累计起来，循环体对每条边最多执行两次，因为从*worklist*中能取到边的源点不超过两次。因此最内层forall循环体需要的时间为$O(E)$，而整个过程为$O(N+E)$。

4.4.4 静态单赋值形式

在图4-5中有一个与常数传播算法有关的问题，是在有控制流的时候定义-使用边的数量能生长得非常大。图4-6介绍一个说明此问题的例子。语句S_1，S_2和S_3都定义了变量X。这些定义经由语句S_4都达到语句S_5，S_6和S_7中的使用。因为每个定义能到达每个使用，所以定义-使用边的数量与语句数量的平方成正比。在这种特殊情况下，有9条边：(S_1,S_5)，(S_1,S_6)，(S_1,S_7)，(S_2,S_5)，(S_2,S_6)，(S_2,S_7)，(S_3,S_5)，(S_3,S_6)和(S_3,S_7)。因为常数传播需时$O(N+E)$，且E与N^2成正比，所以对整个算法来说，需要的时间是语句数量的二次方。

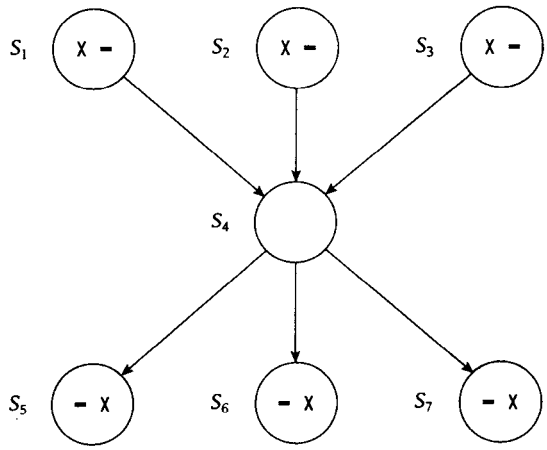


图4-6 定义-使用的例子

减少操作数量的一种方法是在语句S_4的结点中放一个特殊的伪操作：

X=X

因为此定义消除语句S_1，S_2和S_3中建立的X的值，故在修改后的程序中定义-使用边的总数是6条：(S_1,S_4)，(S_2,S_4)，(S_3,S_4)，(S_4,S_5)，(S_4,S_6)和(S_4,S_7)。

这个思想能与一种方法结合，为每个标量变量域提供惟一的名字，产生*静态单赋值形式*(通常缩写成SSA)，这是定义-使用图的一种变化，具有下列性质：

(1) 每个赋值语句创建一个不同的变量名。

(2) 在控制流汇合点插入一特殊操作将相同变量的不同实现合并在一起。

图4-7给出的是图4-6中例子的静态单赋值图。此图类似于原控制流图，这不是巧合，因为合并的结点插在控制流路径的合并处。

静态单赋值形式表示的定义-使用图对分析来说有许多优点，其中最重要的一点是改善像常数传播这样的算法的性能，并且减少图的大小。

构造SSA通常分为两个主要阶段进行：

(1) 需要标识放合并函数（称为φ函数）的地方，

(2) 变量重命名，为每个定义点创建惟一的名字。

在我们能标识插入φ函数的点之前，需要考虑一下我们的目标。为了保持图很小，我们应当仅在以下一些点上引入φ函数：它们对维持所要求的图的性质是必要的，最重要的是仅有一个定义到达每个变量使用。因此，我们将在这种块的开始处为一给定的变量x插入一个φ函数：该块

144
~
149

有多个前驱，如果通到那些前驱之一的每条路径包含一个定义，其中到达其他前驱之一的某条路径被绕过。在该块开始处需要x的ϕ函数，为了保证不会有多于一个x的定义到达后继的使用。

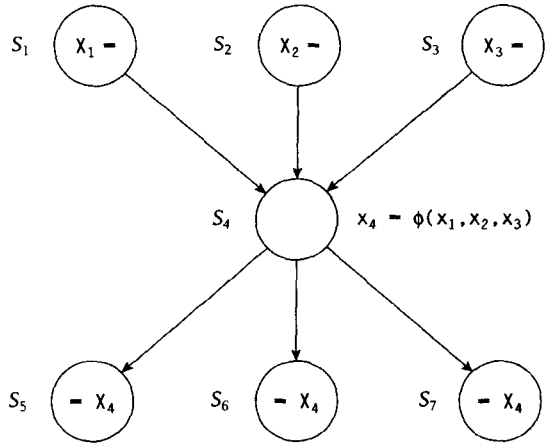


图4-7 图4-6的静态单赋值形式

控制边界 为了标识要求的插入点，我们将引入控制结点（dominator）和控制边界（dominance frontier）的概念。

定义4.2 在具有单出口结点的有向图G中，结点X在G中支配（或控制）结点Y，如果从G的入口结点到Y的任何路径必经过X。结点X严格地控制Y，如果X控制Y且$X \neq Y$。

在有向图中计算控制结点的问题已为众多研究者探讨过[198，138]。计算控制结点的最简单方法是将它作为一个数据流问题。令$dominators(b)$是顶点的集合，这些顶点控制顶点b。根据常规，我们将总假定$b \in dominators(b)$。下面的数据流方程组对计算控制结点是充分的：

$$dominators(x) = \{x\} \cup \bigcap_{y \in preds(x)} dominators(y) \tag{4-2}$$

可以用图4-2介绍的迭代方法的变种求解此方程，其中所有的$dominators$集合初始化为泛集，使其到达一个最大的不动点（见图4-8）。 150

此算法继承迭代方法的渐近运行时间——$O((N+E)N)$个集合操作。然而，对于控制流图是可规约的特殊情况（即它不包含多入口循环），如大多数良结构图那样，假设以深度优先顺序处理图的顶点，很容易看出算法一遍就收敛了。这是因为在可规约图中，一条回边的源点的控制结点集是汇点的控制结点集的超集。这样，回边不再进一步减少它们的汇点的控制结点集。因此对于可规约图，此算法仅需$O(N+E)$个集合操作。

定义4.3 在图G中指定一个顶点x，它的直接控制结点是顶点$y \in dominators(x) - \{x\}$，使得如果$z \in dominators(x) - \{x\}$，则$z \in dominators(y)$。

换句话说，顶点x的直接控制结点必须严格地控制x，并且受x的每一个其他控制结点的严格控制。

一旦得到控制结点集，就容易构造直接控制结点树。注意，由于x的直接控制结点y必须受到x的每个控制结点的控制（x自身除外），那么x的直接控制结点必须是$dominators(x) - \{x\}$

的成员，它具有最大的控制结点集。基于此观察的结果，我们可以构造分两遍的过程，用来建立直接控制树，每遍费时不超过$O(N^2)$。

```
procedure iterateDom(G)
  // G=(N, E) 是输入控制流图，其中
  // N是基本块集合
  // E是控制流边的集合
  // dominators(b) 是块b的控制结点集
  for each b ε N do
  if predecessors(b)=ø then dominators(b)={b}
    else dominators(b)=N;
  changed := true;
  while changed do begin
    changed := false;
    for each b ε N do begin
      newDoms := dominators(b);
    for each p ε predecessors(b) do
      newDoms := newDoms ∩ dominators(p);
    newDoms := newDoms ∪ {b};
    if newDoms != dominators(b) then begin
      dominators(b) := newDoms;
      changed := true;
    end
  end
 end
end iterateDom
```

图4-8 迭代控制结点的构造

（1）对于每个顶点x，加注它控制结点集中的顶点数目。

（2）对于每个顶点x，设置$idom(x)$为$dominators(x)-\{x\}$中具有的最大控制结点集的顶点。

Lengauer和Tarjan研究出一个算法，用来构造直接控制结点关系，最坏情况下需时$O(E\alpha(N, E))$，其中N和E分别是控制流图中的结点数和边数，α是一个增长非常慢的函数，它与Ackerman函数的逆函数有关[198]。由于函数α增长很慢，所以算法实际上与输入图大小成线性关系。Harel改善了此算法，使它在最坏的情况下是线性的[198]。

定义4.4 对给定的块x，控制边界$DF(x)$是这样的块y的集合，y的某个前驱结点在控制流图中受x的控制，但y自身不受x严格控制。

图4-9给出一个计算控制流图中每个块的控制边界的算法。

我们必须说明算法*ConstructDF*正确地构造控制边界。就是说，在算法执行后$y \in DF(x)$，当且仅当y是在x的控制边界中。

为了看清这一点，假设y是在x的控制边界中。如果它是x的一个后继结点，但不受x的控制，则在步骤L_3中将它加入$DF(x)$中。假设它不是x的后继结点，则在控制结点树中存在某个控制结点序列$\{x_1, x_2, \ldots, x_n\}$，使得

$$x \text{ idom } x_1 \text{ idom } x_2 \ldots \text{idom } x_n$$

这里y是x_n的后继结点，但x不是y的严格控制结点。注意没有顶点x_i能控制y，因为在控制结点树中x_i的所有前驱结点（包括x）必须控制y，这与假设矛盾。由于顶点是按逆控制结点顺序处理的，将首先处理x_n，并由L_3将y加入到$DF(x_n)$中。随后当处理控制链上的其他元素时，由L_4将y加到它们的控制边界的每一个当中去。因此最终将y加到$DF(x)$中。

```
procedure ConstructDF(G, DF)
  // G是输入控制流图
  // DF(x) 是x的控制边界中集本块集
  // idom(y) 是控制流图中基本块y的直接控制结点。

L1: 对控制流图G求直接控制结点关系idom; （对一个具有单入口的控制流图，此关系形成一棵树，以入口
    结点作为根。）
    令l是控制结点树的逆拓扑列表，使得如果x控制y，则在l中x跟随y之后;

L2: while l ≠ Ø do begin
        令x是l的第一个元素;
        从l中删除x;

L3: for all x的控制流后继结点y do
        if idom(y) ≠ x then DF(x) := DF(x) ∪ {y};

L4: for all z使得idom(z) = x do
        for all y ∈ DF(z) do
          if idom(y) ≠ x then DF(x) := DF(x) ∪ {y};
    end
end ConstructDF
```

图4-9 控制边界构造算法

另一方面，如果此算法将一顶点加入$DF(x)$中，则该顶点必须是在x的控制边界中。显然，在L_3中加语句就是这种情况，因为此循环简单地实现控制边界定义。假设步骤L_4将某个顶点y不正确地加到了$DF(x)$中。（我们可以假定y是第一个被不正确地加入的顶点。）因此，存在一个z使得x idom z，并且y在$DF(z)$中。因为z是在x之前处理的，因此，根据假设y必须已被正确地加到它的控制边界中。但是惟一不能正确地将y加入到$DF(x)$的情形是当x idom y时。然而，如果这样的话，它必须是直接控制结点，因为如果x和y之中的任何结点控制y，那么y可能不在z的控制边界中。但这是不可能的，因为如果x是y的直接控制结点，则绝不会在L_4中被加到$DF(x)$中。

不计算控制结点的构造，算法*ConstructDF*在最坏的情况下需要时间为$O(\max(N+E, |DF|)$。为了看清这一点，注意拓扑排序需时$O(N+E)$，而对控制流图中每个结点在标号L_2处执行循环头一次，或者说需要$O(N)$次。对每个结点的每个控制流后继结点，在L_3进入此循环一次，总共需$O(E)$次，因为循环体能在常数时间内实现，此循环总共需要的时间为$O(E)$。

在标号L_4的循环更加复杂，对控制结点树中的每条边执行此循环一次，但是由于每个结点最多有一个直接控制结点，因此执行循环头只需$O(N)$次。对于给定的结点x的控制边界中的每个元素，内循环最多执行一次，所以此循环嵌套需时$O(|DF|)$。

151 ~ 153

注意，*ConstructDF*所花的时间与它的最大输入和输出量成比例。因为一个算法必须至少要和二者之中任何一个所需要的时间一样，所以这个算法是优化的。

确定插入位置 一旦我们有了有效的控制边界，我们就能用图4-10中的过程确定ϕ函数的

所有位置。此算法基于这样的简单观察：如果块x包含变量y的一个定义，那么对y的φ函数必须插在$DF(x)$中每块的头上，因为有一条可选的路径达到这些块中的每一块而不经过x，因此包含y的一个不同定义。一旦在块z中插入了y的φ函数，那么根据传递性，我们还需在$DF(z)$的每个元素中插入一个y的φ函数。

```
procedure LocatePhi(G, DF, PutPhiHere)
  // G是输入控制流图
  // DF是控制边界图，其中
  // DF(x) 是基本块x的控制边界中块的集合，即DF图中x的后继结点，
  // Def(x)是在基本块x中定义的变量集。
  // PutPhiHere(x)是变量集，有关的φ-函数必须插在块x的开始处。

  求控制边界图DF中的最大强连通区域的集合{S1, S2, ..., Sm}（使用Tarjan深度优先搜索算法）；

  通过约简每个Si成为单结点，DF中的一条边 (x, y) 变为DFπ中的一条边 (Sx, Sy)，其中Sx是包含x的区域，
  Sy是包含y的区域，这样从DF构造出DFπ；（注意：删除从一个区域到自身的那些边。）

  令 (π1, π2, ..., πm) 是DFπ的m个结点，其编号顺序与Dπ一致（使用拓扑排序得到的序）；

  for i:=1 to m do begin
    Def(πi)置为πi中对所有x，Def(x) 的并集;
    PutPhiHere(πi) 置为空;
  end

  for i:=1 to m do
    for each DFπ(πi) 中的πj do
      PutPhiHere(πj) := PutPhiHere(πj) ∪ Def(πi);
  for i:=1 to m do begin
    if πi是一强连通区域或自循环then
      PutPhiHere(πi) := PutPhiHere(πi) ∪ Def(πi);
    for each x ∈ πi, πi外有一个前趋结点do
      PutPhiHere(x) := PutPhiHere(πi);
  end
end LocatePhi
```

图4-10 确定φ函数的位置

另一个观察到的结果使算法更加有效。在DF图中可能有一个环。由两个顶点构成的循环是最简单的例子，循环外面的单个顶点直接转移到循环中的每个顶点。于是每个顶点是在另外一个的控制边界中。然而，处理这样的循环是简单的，因为如果在此循环的任何结点中必须放置一个φ函数的话，那么也必须在循环的每个入口结点中放置一个，这是由于上面所述的传递性需要的。因此，为了传播的目的，我们能将DF图中的强连通区域当成一些单个的顶点处理。

从这些观察结果应清楚地看到，算法为φ函数计算了正确的位置，并且需要的时间是DF图中结点数的线性关系。应当看到DF图比控制流图大得多。事实上，在最坏的情况下能与控制流图大小的平方成比例，虽然这种情况是少见的[97]。目前已证明确定φ函数位置需要的时间与控制流图的大小成比例[253]。

应当指出，不在目标变量废弃的交汇结点的开始处放置任何这样的函数，能有效地减少φ函数的数量。图4-10中算法的简单改进，与通过迭代分析构造的每个块上的活变量集相结合，

将足以实现此目标。

在SSA形式的构造中留下的是重命名变量和建立SSA边。这能用一种简单的方法实现：将惟一的索引名赋给在每个定义点产生的值，包括φ函数。然后应用4.4.1节中描述的*reaches*传播算法能确定哪个值到达每个使用，因为一旦插入了φ函数，则仅有一个定义能到达任何使用。

虽然SSA形式不直接修改早先描述的死代码消除和常数传播算法的结构，但是对下一节中要处理的表达式前向替换和归纳变量替换算法将从它的特殊性质获得重大好处。

4.5 归纳变量暴露

死代码消除和常数传播两种变换依靠的都是定义-使用图中相当简单的模式。归纳变量替换是更加复杂的变换，并需要相当复杂的模式的识别。因此，归纳变量替换通常需要一个复杂的分析程序框架。鉴于存在这样的架构并已知以下的事实：归纳变量替换的原始需要起因于将普通程序设计行为变成一种更符合依赖测试需要的形式的必要性，所以归纳变量替换阶段通常不只是复制辅助归纳变量。一个普遍的扩展是将区域不变的表达式前向代入下标中。

154 ~ 155

在这一节我们将介绍前向替换和循环归纳变量替换的算法。然后我们将这些材料捆绑在一起，讨论如何使它们一起工作。

4.5.1 前向表达式替换

下面是一个有用的前向替换的例子：

```
      DO I = 1, 100
         K = I+2
S1       A(K)=A(K) + 5
      ENDDO
```

这里，程序员在他的代码中作了某些编译器的工作，实施公共子表达式消除，将两个下标计算压缩成单个变量。可惜依赖测试程序计算来自对数组A的引用的依赖将出现问题，这是因为K的使用，它不是一个归纳变量而又在循环中A的下标里变化。前向替换K产生

```
      DO I = 1, 100
S1       A(I+2) = A(I+2) + 5
      ENDDO
```

得到一种依赖测试容易处理的形式。另外，如果对一向量机编译这个例子，那么生成的代码将是非常有效的。假定确定第一种形式是可向量化的（的确可以），将通过扩展K为一临时向量，并用此临时向量作为A的分散-聚集索引，它会被向量化。第二种形式产生一种更简单的向量操作。

实施归纳变量替换和前向表达式替换不仅需要定义-使用边，还需要某种控制流分析。一个定义归纳变量的语句，在循环的每个迭代中必定被执行。类似地，在一个定义被向前替换之前，必须保证在它替换到语句中之前总是在一循环迭代中执行该定义。这些性质很容易用来确定我们是否有效地使用了SSA图，以及我们是否有一个简单的过程来确定哪些语句是在给定的循环里面和哪些不在里面。

156

为了确定一特定的语句是否在一给定的循环中，我们需要对每个语句维护一个独立的数据结构，用来确定语句嵌在其中的一组循环。如果给定的循环L是在包含S的循环嵌套中，则语句S是在循环L中。特别地，如果循环层是k，我们只需测试包含S的k层循环是否等于L。如

果我们维护一个包括每个语句的循环标识符栈，那么很容易做到这一点。

使用这种机制，我们能开发一个简单的前向替换的过程，处理只涉及循环不变量和循环归纳变量的任何表达式。SSA图使它很容易判断一个给定的表达式是否符合这种需求。给定一个语句S做为前向替换的候选，我们简单地检查到S的每条SSA边。如果此条边来自循环中的一个语句，那么该语句必须是循环归纳变量在循环体开始处的ϕ结点。否则此量包含一个表达式而不是在循环内变化的循环归纳变量，并且我们必须排除前向替换。

实际上，上面的陈述并不完全正确。在实际使用之前可能有一个循环不变的赋值语句，如下所示:

```
    DO I = 1, N
    ...
S1  IC = IB !IB是循环不变的
    ...
S2  IX = IC + 5
    ...
    ENDDO
```

表达式“IC+5”能被前向替换，因为赋给IC的是一个循环不变的值IB。然而，我们建议的过程通过依序处理循环中的语句将避免所述的这个问题。因此为前向替换考虑S_2右端之前，对IC的赋值将被前向替换到语句S_2中。

回到前向替换算法，下面的程序说明在一个DO循环被扩展成更简单的代码之后，标准的循环归纳变量是如何增加的:

```
    DO I = 1,N
    ...
    ENDDO
```

变成

```
    I = 1
L   IF(I>N) GO TO E
      ...
    I = I+1
    GO TO L
157 E
```

其中L和E是惟一的数字标号。归纳变量的更新发生在循环结尾处，是在所有使用它的语句之后。所以，标准循环归纳变量的每次使用必须有一条来自ϕ结点的边，ϕ结点正好在循环入口之后合并归纳变量的值。

一旦确定语句S能被前向替换，那么过程能被执行: 检查从目标语句到另一个在循环中不是ϕ结点的语句的每条边。对于每条这样的边，在SSA边的汇点上的语句中我们用S的右端替换S左端的每一次出现。图4-11介绍遵循此方法的前向表达式替换算法。

正如我们早先指出的，此算法必须依序从头到尾应用到循环中的各个语句上。按此做法，前向替换链能减少循环中使用的变量个数，它以循环中附加的表达式复杂性为代价。这种有序的应用是图4-16（4.5.3节）中介绍的变换驱动程序的一部分。

*ForwardSub*还需要SSA图的一个值得注意的性质。算法中第一个循环追踪从外面进入一个语句的所有的边。图中的第二个循环追踪从一个语句出来进到其他语句的所有的边。为使

此算法工作，实现SSA图所选择的任何数据结构必须保持这种双向性。

```
boolean procedure ForwardSub(S, L)
  // S是前向替换的候选语句
  // L是正在其中实施替换的循环
  // 假设有关语句嵌套的SSA图和信息
  // 如果因为S有循环可变的输入，未尝试替换，则返回true，指出应当试IV替换
  // ForwardSub实现L中每个语句的替换，它的右端变量只含L的归纳变量或L的不变量。

  if S应用一个φ-函数or S有副作用then return false;

  // 确定循环不变量
  for each进入S的SSA边e do begin
    S_s = source_stmt(e);
    if S_s是在循环L中and
      S_s定义了一个变量而不是L的归纳变量
    then return true;// 不能是循环不变量，但可试调用IV-sub
  end

  // 用该语句替换循环中所有的限定的使用。
  all_uses_gone = true;
  all_loop_uses_gone = true;
  for each从S发射出的SSA边e do begin
    S_t = target_stmt(S);
    if S_t是在循环L中
      if operation(S_t) ≠ φ then begin
        用rhs(S) 置换target(e);
        // 更新SSA边
        for each进入rhs(S) 的SSA边ie do
          加上从source_stmt(ie) 到S_t的新边;
        删去边e;
      end
      else all_loop_uses_gone = false;
    else all_uses_gone = false;
  end
  if all_uses_gone and all_loop_uses_gone then delete(S);
  else if all_loop_uses_gone then S移出循环外;
  return false;
end ForwardSub
```

图4-11 前向表达式替换算法

使用返回值确定是否要对语句尝试做归纳变量替换。此返回值将在4.5.3节中介绍的驱动例行程序中使用。因为对S的某个输入是在循环中定义的，故仅当未尝试前向替换时，返回值才是true（真）。注意，如果语句S已被删除或移去，那么例行程序绝不会返回true。

4.5.2 归纳变量替换

讨论前向替换后，现在是集中注意力于归纳变量替换的时候。第一个任务是识别辅助归纳变量。

定义4.5 在一个以

```
DO I = LB,UB,S
```

为循环头的DO-循环中，一个辅助归纳变量是它在循环中使用的每个位置L处能正确地表示成

$$cexpr * \mathrm{I} + iexpr_L$$

158 ~ 159 的任何变量，其中$cexpr$和$iexpr_L$是表达式，它们在循环中不变化，虽然在循环中不同的位置上可能需要替换$iexpr_L$的不同的值。

用这种最简单的形式，一个辅助归纳变量将用像

$$\mathrm{K} = \mathrm{K} \pm cexpr \tag{4-3}$$

这样的语句来定义，再次强调，其中$cexpr$是循环不变式。

归纳变量识别具有很大的普遍性。某些程序定义了一组归纳变量，如下例所示:

```
DO I = 1, N
   J = K + 1
   ...
   K = J + 2
ENDDO
```

这里J和K是循环的辅助归纳变量。

我们可以用4.5.1节的过程判断一个语句是否在一给定的循环L中，以帮助我们判断一给定的语句S是否定义一个归纳变量。语句S可能为L定义一个辅助归纳变量，如果S是包含在一个SSA边的简单环中，此环只涉及到S和循环中的另一个语句（一个ϕ结点）。要求ϕ结点是在循环的入口处，用来合并循环外面初始化的循环归纳变量的值和循环内增加的值。如果此环包含循环内的其他ϕ结点，那么它不是一个归纳变量，因为它在每次迭代中不被更新。

作为例子，考察下面的循环:

```
DO I = 1, N
   A(I) = B(K) + 1
   K=K+4
   ...
   D(K) = D(K) + A(I)
ENDDO
```

它有图4-12中定义的SSA图，为简便起见图中省略了索引变量名字。此图表明SSA图仅有一部分涉及循环中变量K的定义和使用。带有单ϕ结点的环是一个辅助归纳变量的指示信号。

如果它满足环的条件，保证归纳变量是在每次循环迭代中被定义，那么余下的惟一要求是定义的表达式有正确的形式。为了本书的目的，我们将介绍一个简单的识别算法，它寻找
160 的仅是方程（4-3）中表现的那种归纳变量的形式。图4-13给出该识别算法。

过程*isIV*开始时保证候选语句S在循环的每次迭代中被执行（如果不被执行，它应当包含在一个环中，此环在循环的结束处也包含一个ϕ结点），并且定义的变量在每次循环迭代中被更新（这是自边的需要）。其次，它保证将可能的归纳变量加到右端，并且在每次迭代中没有一个乘系数和对候选归纳变量加上或减去一个常量（$cexpr$）。如果循环不变部分是被减项，那么它被求反，所以在替换阶段可以假定加法是控制操作。

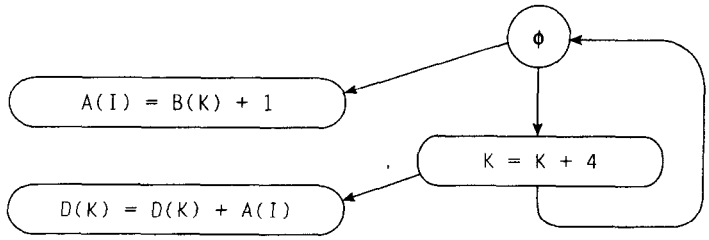


图4-12 归纳变量识别的示例SSA图

```
boolean procedure isIV(S, iV, cexpr, cexpr_edges, iV)
  // S是候选语句，它可能定义了一个辅助归纳变量（IV）
  // 如果S定义了一个归纳变量isIV返回true
  // iV是归纳变量
  // cexpr是一个表达式，它被加到S的归纳变量中
  // cexpr_edges是SSA边集，这些边是从循环外到cexpr的
  // isIV检查赋值语句S，以确定它是否定义该循环中包含它的归纳变量。
  // 此过程仅识别归纳变量，修改I=I±cexpr的形式，这里cexpr在循环中是常数。

  is_iv=false;
  If S是循环中SSA图的环路部分，它只涉及到自身和循环头中φ结点
  then begin
    令iV是S左端中的变量;
    令cexpr是，iV已被消除的rhs(S);
    if iV是加到rhs(S) and
      cexpr或者加入rhs(S)中，或者从rhs(S)中减去

    then begin
      loop_invariant := true;
      cexpr_edges := Ø;
      or each边e到达cexpr do
        if source(e) 是在当前循环内
          then loop_invariant := false;
          else cexpr_edges := cexpr_edges ∪ {e};
      if cexpr从rhs(S) 中被替换then cexpr := - cexpr
      if loop_invariant then is_iv := true ;
    end
  end
  return is_iv
end isIV
```

图4-13 归纳变量识别

161

图4-14和图4-15介绍的算法，在归纳变量一旦被识别时，实现对它的使用的置换。

替换算法对正在被置换的辅助归纳变量使用两个不同的表达式。对于在循环中出现在S前面的语句，使用的乘数比当前的迭代数小1。S之后的语句乘数是当前的迭代数（I是循环索引；L是它的下界；U是它的上界；S是跨距）。

```
procedure IVSub(S, L)
  // S是候选语句，它可能定义一个辅助IV
  // L是被实施替换的循环
  // IVSub检查赋值语句，确定它是不是一个归纳变量，如果是，则替换它。

  If not isIV(S, iV, cexpr cexpr_edges) then return;
  // iV是定义的辅助归纳变量
  // cexpr是加入iV中的常数表达式
  // cexpr_edges是边的集合，它们定义在cexpr中使用的变量

  令包含S的最内层循环的头是
        DO I = L, U, S

  令S_h指称iV的循环头中的φ-结点;

  // SSA保证只有一条从循环外进入S_h的边
  令S_0指称从循环外到S_h的单条边的源点;

  for each从S_h到相同循环中一个结点的边e do begin
    // 在循环体中，target(e) 在s之前到来
    用“target_expr(e) + ((I - L) / S) * cexpr”替换target_expr(e)
    update_SSA_edges(e, cexpr_edges, S_0);
  end

  for each从S到相同循环中一个结点的边e do begin
    //在循环体中，target(e)在S之后到来
    用target_expr(e) + ((I - L + S) / S) * cexpr替换target_expr(e);
    update_SSA_edges(e, cexpr_edges, S_0);
  end

  if存在从S到循环外顶点的边then begin
    将S移出循环外，
      将cexpr改为 ((U - L + S) / S) * cexpr
    加一条从S_0到S的边；删去从S_h到S的边;
  end
  else删去S和从S_h到S的边;
  删去S_h和从S_0到S_h的边;
  return;
end IVSub
```

图4-14 归纳变量替换算法

```
procedure update_SSA_edges(e, cexpr_edges, S_0)
  // e是边，沿着它已实施iV替换
  // cexpr_edges是边的集合，它们是从循环外进入更新的表达式的边
  // S_0是循环外iV的惟一定义点。

  for each edge ie ∈ cexpr_edges do
    加上一条从source(ie) 到target(e) 的新边;
  加上一条从S_0到target (e) 的边;
  删除边e;
end update_SSA_edges
```

图4-15 在归纳变量替换后更新SSA图

4.5.3 驱动替换过程

在研究应用于前向替换和归纳变量替换的各个算法后，现在来研究将各个变换捆绑在一起的驱动算法。有两个关键性考虑。第一，由于在一内循环实施归纳变量替换，可能在外循环引入一个新的归纳变量（例如见4.1节的那些例子），归纳变量替换应当由内到外实施。第二，前向替换循环不变的表达式可能建立某些新的归纳变量，表明应当先实行循环不变的表达式替换。这两点意见浓缩在图4-16表示的算法*IVDrive*中。

```
procedure IVDrive(L)
  // L是正被处理的循环，假设SSA图有效
  // IVDrive实施循环L上的前向替换和归纳变量替换，这里需要递归地调用它自己。
  for each按序对L中的语句S do begin
    case(kind(S))
      assignment:
        FS_not_done := ForwardSub(S, L);
        if FS_not_done then IVSub(S, L);
      DO-loop:
        IVDrive(S);
      default:
    end case
  end
end IVDrive
```

图4-16 归纳变量替换驱动程序

对于归纳变量替换，最后有两个有价值的评注。第一个是关于循环正规化（4.3节）和归纳变量替换之间的交互问题。在一个非正规化循环上实施归纳变量替换时，会得到极为低效的代码。

例如，考察下面非常一般的例子：

```
DO I = L,U,S
   K = K + N
   ... = A(K)
ENDDO
```

应用图4-14中描述的归纳变量替换产生

```
DO I = L,U,S
   ... = A(K+(I-L+S)/S*N)
ENDDO
K = K + (U-L+S)/S*N
```

在许多机器上不能执行硬件整数除法和乘法；即使执行，它们也是使用非常低效的指令。由于在循环中引入整数除法和乘法，归纳变量替换产生了许多低效率的代码。通常应用众所周知的强度削减优化试图减少多数常用的整数乘法，但是出现在循环内的那种形式不能用这种变换来消除。另外，新下标的非线性性质意味着对它做依赖测试将会失败。总之，将归纳变量替换应用到此循环上是无济于事的。

然而，如果在归纳变量替换前此循环被正规化，

162 ~ 164

```
I = 1
DO i = 1,(U-L+S)/S, 1
   K = K + N
   ...= A(K)
   I = I + 1
ENDDO
```

则从归纳变量替换得到的代码是更合乎人意的:

```
I = 1
DO i = 1,(U-L+S)/S,1
   ...= A(K+i*N)
ENDDO
K = K + (U-L+S)/S*N
I = I + (U-L+S)/S
```

此代码在循环内只有一个整数乘法，强度削减容易将它消除。另外，此下标能容易地用第3章描述的依赖测试处理。总之，应在归纳变量替换前应做循环正规化，使稍后的变换更有效。

第二个评注涉及这里介绍的归纳变量替换算法的简明性。在图4-14中介绍的算法是一个简单的版本，设计用来演示有关的一般原理。通常在循环中会有更复杂的辅助归纳变量，如下面的例子所示:

```
DO I = 1, N, 2
   K = K + 1
   A(K) = A(K) + 1
   K = K + 1
   A(K) = A(K) + 1
ENDDO
```

像这样的情况，归纳变量识别仍是十分简单的——含有定义归纳变量的语句将在循环内形成一个SSA图环，如图4-17所示。此环只有一个ϕ结点，这个结点强制放在循环的开始。当然应用正规的约束——仅允许循环不变量的加和减。再一次要注意的是，为了简明起见，索引变量名（通常是在SSA图中）在图4-17中已经被省略了。

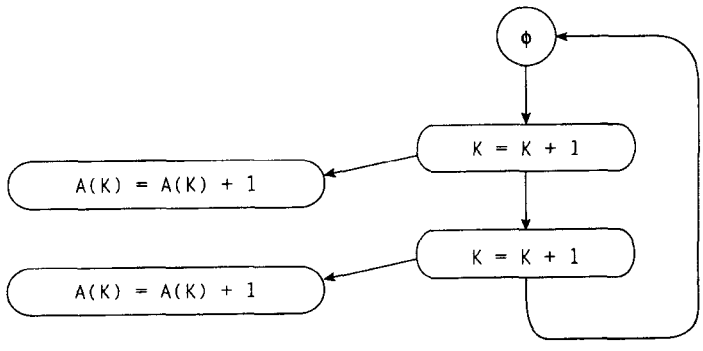


图4-17 复杂归纳变量替换的SSA图

一旦识别出这样的变量，归纳变量替换很容易适应处理更复杂的情况。首先将定义归纳变量的语句从ϕ结点开始编号为{S_0, S_1, ..., S_n}。然后，对每个S_i赋给一个不同的替换公式，这

取决于特定语句S_1在刚执行后归纳变量的聚集值。实现的细节是直截了当的。

一个替代策略是增强前向替换以识别区域不变量。当做到这一点时，对K的第一个赋值可以前向传播，产生

```
DO I = 1, N, 2
   A(K+1) = A(K+1) + 1
   K = K + 1 + 1
   A(K) = A(K) + 1
ENDDO
```

至此普通的归纳变量替换将完成对K的消除。扩展前向替换以识别这种情况是直截了当的[20]。

4.6 小结

我们已介绍了几个变换，让更多的下标成为标准形式，来支持依赖测试。

- *循环正规化*是一种变换，使循环从标准下界到上界执行，执行的步长为1。在许多编译器中用它来简化依赖测试，虽然它有若干不足之处。 165 ~ 166
- *常数传播*在编译时用已知的常数置换未知值的变量。置换是由程序中一个数据流图表示上的算法执行的。
- *归纳变量替换*消除辅助归纳变量，用标准的循环归纳变量的线性函数置换它们。归纳变量替换算法的一种简单变化，可用来实现循环嵌套中的表达式折叠。

这些变换由许多数据流分析策略支持，包括数据流算法的迭代解和定义-使用链或静态单赋值（SSA）形式的构造。

4.7 实例研究

原PFC实现实施了所有循环嵌套的正规化。其后，它应用了归纳变量识别、前向替换和归纳变量替换，都是遵循这一章介绍的算法。由于实现的时间先于SSA的开发，所有算法采用了定义-使用链。这对某些过程增加一点额外的复杂性，但是很有效。

PFC的归纳变量替换阶段是极为系统化的，并且能处理同一循环内的相同归纳变量的多次更新。它从循环最内层到最外层进行，处理初始化，以及将内层循环生成的最终语句处理成外层循环进一步替换的候选。结果表明在PFC系统中归纳变量替换阶段对向量化的成功总的来说是很关键的。

PFC在每一个主要变换阶段之后，也执行系统化的死代码删除。这使代码更清晰和更易于理解。因为死代码删除程序不包含控制依赖边，它标记的所有控制结构绝对有用。因此它必须包括一后置遍，用来删去带有空子句的循环和if语句。

Ardent Titan 编译器对Fortran使用一组类似的算法，但是附带有处理C优化的任务。它将附加的重任放在归纳变量替换中。因为C的前端对带有++操作符的表达式不做精细的副作用分析，故为生成的中间代码提供了许多归纳变量替换的机会。例如，源代码如

```
While(n){
   *a++ = *b++;
   n--;
}
```

167

C前端会将它翻译成

```
while(n) {
   temp_1 = a;
```

```
    a = temp_1+4;
    temp_2 = b;
    b = temp_2 + 4
    *temp_1 = *temp_2;
    temp_3 = n;
    n = temp_3-1;
}
```

一旦将所有无用的东西清除掉，此循环很容易向量化（毕竟它仅是一份向量拷贝）。在此发生之前，必须将关键赋值语句向量化成

```
*(a+4*i)=*(b+4*i);
```

这样的形式，其中i是生成的循环归纳变量。问题是产生这种形式只能用对temp_1和temp_2的赋值前向替换星号赋值。然而，在对a和b的更新被前向移动之前，不能做这种替换。因此，简单的技术不能处理这种循环。

虽然处理这一问题的理论已产生非常通用的技术（使用部分冗余消除的各种策略，例如[215，75]），因为实用性的理由，特别是空间方面的理由，Titan编译器设计者们不选用它们。编译器另行使用一种简单的启发式解法：无论何时一个语句被替换拒绝，只是因为其后的语句重新定义了该语句使用的一个变量，那么后面的语句被标记为“封锁”前面的语句。当一个封锁语句被前向替换时，则要重新检查它所封锁的所有语句的替换。因此，在当它保证产生某个替换时才需要回溯。另外，替换此语句需要的多数分析不需要被重复。因此，极少调用回溯，但是当调用回溯时，它是极为有效的。

对C中使用的变换（特别是向量化）的广泛讨论，一直推迟到第12章。

4.8 历史评述与参考文献

从Parafrase系统[190]开始，许多向量化编译器实施循环正规化。数据流分析有大量文献
168 [15，125，164，169，182，218，264]。这里描述的定义-使用链构造，以及常数传播和死代码消除算法，在概念上是由John Cocke建立的。这里所用的系统表示归功于Kennedy[169]，由Wegman和Zadeck做了修改[272]。

控制结点构造的数据流方法是传统的方法（例如，见Muchnick[218]）。然而，从控制结点集构造直接控制结点的算法是新的，并且逐渐胜过Muchnick介绍的方法。然而，Lengauer-Tarjan[198]和Harel[138]的直接控制结点关系的直接构造方法比这里描述的方法逐渐地更加有效。

SSA形式的构造归功于Cytron，Ferrante，Rosen，Wegman和Zadeck[97]，尽管基于DF图计算φ函数插入位置的明确表述是他们的方法的一个变种。线性时间的SSA构造算法已由Sreedhar和Gao[253]建立。

在每一个向量化编译器中，已有各种不同形式的归纳变量替换。如Allen，Kennedy和Callahan[16，21，20]描述的，这里介绍的归纳变量替换算法已在PFC系统中改写成SSA形式。Wolfe[284，285]介绍了一种更加复杂的方法。

习题

4.1 基于第3章的测试策略，一个非正规化循环的哪一个性质（非单位开始索引或非单位跨距）会引起最大困难。为什么？

4.2 手工正规化下面的循环:

```
DO I = 1000, 1, -2
   A(I) = A(I) + B
ENDDO
```

4.3 对下面的循环手工做正规化、归纳变量替换和常数折叠:

```
IS = 5
DO I = 1, 100
   IS = IS + 10
   DO J = 2, 200, 3
      A(IS) = B(I) + C(J)
      IS = IS + 1
   ENDDO
ENDDO
```

169

4.4 为活跃分析问题建立类似于方程（4-1）的数据流方程。目标是计算变量的*live*(*b*) 集，表示在程序中每个基本块*b*的入口处这些变量是“活跃的”。在程序中某一点上一个变量是活跃的，如果有一条控制流路径从该点到此变量的使用，并且在它的使用之前，该路径上不包含此变量的定义。提示：考虑两种情况——变量活跃是由于同一块中有它的使用，以及变量活跃是由于在某个后面的块中有它的使用。

4.5 考虑循环嵌套，这里L1，L2，H1和H2都是未知的:

```
      DO J = L1, H1, 7
         DO I = L2, H2, 10
S           A(I+J+3) = A(I+J) + C
         ENDDO
      ENDDO
```

除开从语句S到自身关联的方向向量(=,=)，消除方向向量的前景如何？为什么？提示：如果你正规化和使用第3章的MIV测试会发生什么？

170

第5章

提高细粒度并行性

5.1 引言

本章和下一章介绍依赖在串行代码的自动并行化中的应用。按照历史发展过程，我们从非最小粒度的并行性开始。这种类型的并行性在向量机和指令级并行性机器（如VLIW处理器和超标量处理器）中是很有用的。因为大多数理论都是首先从向量机发展而来的，所以将着重于向量化的讨论。粒度敏感的并行性处理将推迟到第6章讨论。

在第2章，我们提出了图2-2所示的并行化算法*codegen*，用于自动寻找Fortran程序中的细粒度并行性。这个算法基本上仅使用循环分布和语句重排变换寻找所有可能的并行性。在本章中，我们在这个基本的代码生成算法的基础上进行扩展，方法是通过使用其他一些变换，例如循环交换，将细粒度并行性增加到一定的量，从而使得自动向量化可行而且有效。

我们用矩阵乘法的一个非常典型的串行编码为例来说明两个这样的高级变换的能力：

```
DO J = 1, M
   DO I = 1, N
      T = 0.0
      DO K = 1, L
         T = T + A(I,K) * B(K,J)
      ENDDO
      C(I,J) = T
   ENDDO
ENDDO
```


尽管矩阵乘法可以很容易地手工并行化，但是*codegen*应用于这个循环嵌套却找不到任何向量操作。问题在于代码中使用了标量临时变量T，它引入由嵌套中的每个循环携带的若干依赖。大部分这样的依赖可以通过*标量扩展*消除，在其中把标量T替换为临时数组T$：

```
DO J = 1, M
   DO I = 1, N
      T$(I) = 0.0
      DO K = 1, L
         T$(I) = T$(I) + A(I,K) * B(K,J)
      ENDDO
      C(I,J) = T$(I)
   ENDDO
ENDDO
```

现在I-循环可以完全围绕它所包含的语句分布：

```
DO J = 1, M
   DO I = 1, N
      T$(I) = 0.0
   ENDDO
```

```
      DO I = 1, N
         DO K = 1, L
            T$(I) = T$(I) + A(I,K) * B(K,J)
         ENDDO
      ENDDO
      DO I = 1, N
         C(I,J) = T$(I)
      ENDDO
   ENDDO
```

最后，I-循环和K-循环被交换，把向量并行性移到最内层的位置。通过循环交换，所有的最内层I-循环都可以被向量化，得到

```
   DO I = 1, L
      T$(1:N) = 0.0
      DO K = 1, N
         T$(1:N) = T$(1:N) + A(1:N,K) * B(K,J)
      ENDDO
      C(1:N,J) = T$(1:N)
172   ENDDO
```

标量扩展和循环交换这两种变换创建了原来并不存在的并行性。经验说明，如果没有这样的变换，自动并行化很难真正有效。

5.2 循环交换

循环交换——交换紧嵌循环中两个循环的嵌套顺序——是提高程序性能最有效的变换之一。下面的例子显示它在向量化中的重要性:

```
      DO I = 1, N
         DO J = 1, M
S           A(I,J+1) = A(I,J) + B
         ENDDO
      ENDDO
```

这个例子中的最内层循环携带一个从语句S到其自身的真依赖。因此，图2-2中给出的向量代码生成算法*codegen*无法作任何向量化。然而，如果交换这两个循环，

```
      DO J = 1, M
         DO I = 1, N
S           A(I,J+1) = A(I,J) + B
         ENDDO
      ENDDO
```

依赖关系就变为由外层循环携带，而内层循环不携带依赖。对此嵌套应用*codegen*算法，内层循环将被向量化并得到如下结果:

```
      DO J = 1, M
S        A(1:N,J+1) = A(1:N,J) + B
      ENDDO
```

在本例中，循环交换通过把可向量化的循环移动到最内层而提高向量化效果。对于在第6章讨论的粗粒度并行化而言，这个过程被颠倒过来，变为把内层并行循环移动到最外层以增加并行粒度和减少同步开销。

为了看出循环交换实际上是一种重排序变换，可以把循环看作它所包含的语句的一系列参数化实例。循环交换改变这些参数化实例的执行顺序，但是并没有产生任何新的实例。在 173
以下代码里，令S(I,J)表示参数为I和J的语句S的实例。换句话说，S(I,J)是语句S在迭代向量为(I,J)的迭代中执行时的实例。

```
DO J = 1, M
   DO I = 1, N
      S
   ENDDO
ENDDO
```

使用这种新记法，我们可以看到在这段代码中，S(1,2)在S(2,1)之后执行，但是在循环交换后，它变成在S(2,1)之前执行。因此循环交换实际上是一个重排序变换。因为是重排序变换，故其合法性可以通过数据依赖关系判定：任何重排序某个依赖端点的循环交换是不合法的。

5.2.1 循环交换的安全性

并非所有循环交换都是合法的，下面的例子说明这一点：

```
DO J = 1, M
   DO I = 1, N
      A(I,J+1) = A(I+1,J) + B
   ENDDO
ENDDO
```

按原来的执行顺序，A(2,2)在I=2,J=1时被赋值；它在后面J-循环的一次迭代中，当I=1且J=2时被使用。如果作了循环交换，A(2,2)先在外层循环的第一个迭代（迭代向量(1,2)）被使用，而后在第二次迭代（迭代向量(2,1)）中被赋值。因此，这个例子中的循环交换破坏依赖关系，导致A(2,2)使用获得错误的值。

图5-1图示说明此类对依赖关系的破坏。该图显示与语句S(2,1)有关的各种不同的依赖关系。箭头从语句S出发指向在它后边执行的语句。灰色箭头表示前边描述过的被循环交换颠倒

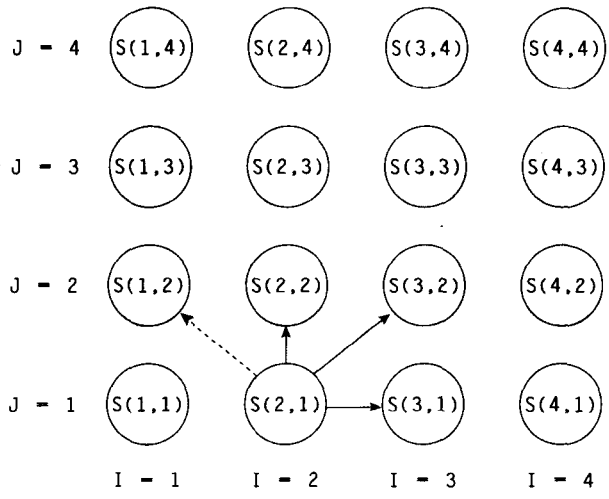


图5-1 循环交换的合法性

方向的依赖；如果J-循环被交换为内层循环，这条依赖边尾部的语句将在它的头语句之前执行。黑色箭头表示不管循环如何交换都保持不变的依赖。

下面给出与循环交换的安全性相关的两类依赖的定义：

定义5.1 对于给定的一对循环来说，一个依赖是*阻止交换的*，如果交换这两个循环会改变该依赖的端点的顺序。

定义5.2 一个依赖是*交换敏感的*，如果在循环交换以后该依赖仍然由同一循环携带。即是，一个交换敏感的依赖随着原来携带它的循环移动到新的层次。

在图5-1中，灰色的依赖是阻止交换的；水平和垂直的箭头表示交换敏感的依赖。不管哪个循环迭代得“更快”，水平方向的依赖将由I-循环携带，而垂直方向的依赖将由J-循环携带。剩余的箭头（从S(2,1)指向S(3,2)的）表示总是由这对循环中的外层循环携带的依赖。这样的依赖通常称为*交换不敏感的*。

定理5.1 设D(i, j) 是一个有n层循环的紧嵌嵌套中某依赖关系的方向向量。那么在对嵌套中的循环作置换后，该依赖的方向向量可以通过对D(i, j) 的元素作同样的置换确定。

证明 由定义2.9和2.10，可知方向向量中各元素的值是由依赖的源点和汇点的迭代向量i和j中各元素的相对值按下式决定的：

$$D(i,j)=\begin{cases}\text{“}<\text{”}, & \text{如果}i_k<j_k\\ \text{“}=\text{”}, & \text{如果}i_k=j_k\\ \text{“}>\text{”}, & \text{如果}i_k>j_k\end{cases}$$

因为循环置换只是置换迭代向量的各元素，所以方向向量必须以完全相同的方式置换。

根据定理5.1的结论可见，方向向量是判断循环交换结果的一个重要工具。从这个结论以及循环交换反转阻止交换依赖的方向这一事实容易看出，阻止交换依赖的依赖方向向量是(<,>)。循环交换后，这样一个依赖的方向向量变成(>,<)，显然，这将导致以不同的顺序访问数据。为了将这些观察扩展到包含多个依赖的循环嵌套的循环交换合法性测试，我们还需要一个定义。

定义5.3 一个循环嵌套的*方向矩阵*是这样一个矩阵：矩阵的行表示包含在嵌套中的语句间的某个依赖的方向向量，并且每个这样的方向向量都被表示为矩阵中的一行。

注意这个定义允许用一行表示几个相同的方向向量。我们用下面的循环嵌套来说明这个定义：

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J+1,K) = A(I,J,K) + A(I,J+1,K+1)
    ENDDO
```

```
   ENDDO
ENDDO
```

174 ~ 176

其方向矩阵为

$$\begin{bmatrix} < & < & = \\ < & = & > \end{bmatrix}$$

矩阵的第一行表示从A(I+1,J+1,K)到A(I,J,K)的真依赖，第二行表示到A(I,J+1,K+1)的真依赖。

不难将定理5.1扩展到方向矩阵。因为循环置换在某个特定的依赖上的效果能反映在置换其方向向量的分量上，故同一置换在所有依赖上的效果即可反映在置换依赖矩阵的各列上。例如，若上例中最外层循环被移到最内层位置，而另外两个循环相应地向外移动，就得到如下的方向矩阵:

$$\begin{bmatrix} < & = & < \\ = & > & < \end{bmatrix}$$

矩阵的第二行现在以一个“>”作为最左的非“=”符号。这样的依赖是非法的，反映上面所描述的变换是非法的。

定理5.2 对紧嵌循环的置换是合法的，当且仅当其方向矩阵在对各列作相同的置换后不含以“>”方向作为最左非“=”方向的行。

定理5.2可以直接由定理5.1和定理2.3得出。它提供一种测试循环交换安全性的简单有效的方法——构造循环嵌套中所有依赖的方向向量，组成一个方向矩阵，并对该矩阵作相应的置换，看是否产生非法的方向矩阵。

5.2.2 循环交换的有利性

图2-2给出的向量化算法*codegen*向量化所有嵌套在携带最内层依赖环的循环以内的循环。循环交换必须增加在这个依赖环以内的循环个数，才能使该算法发现更多的向量化。换句话说，循环交换必须改变它所作用的循环嵌套的依赖模式才能有效。

定理5.2告诉我们，一个特定循环交换对循环嵌套内依赖关系的影响可以通过改变方向矩阵而看到；换句话说，所得到的依赖模式可以不需要修改程序本身就能确定。因此，为了确定循环交换的有利性，我们可以分析可能有利的循环置换并置换方向矩阵以检查它们是否合法。 177

目标机器的体系结构通常是确定最有利的循环交换模式的主要因素。因此，正确的循环嵌套不能以机器无关的方式确定。为了说明最佳循环交换模式如何随目标体系结构而变化，考虑下面的例子:

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
S        A(I+1,J+1,K) = A(I,J,K) + B
      ENDDO
   ENDDO
ENDDO
```

S由一个方向向量为(<,<,=)得到真自反依赖。如果用图2-2中的*codegen*来处理这个嵌套，

内层的两个循环将被向量化（依赖由外层循环携带）。结果产生Fortran 90代码

```
      DO I = 1, N
S       A(I+1,2:M+1,1:L) = A(I,1:M,1:L) + B
      ENDDO
```

这样的代码对于有大量同步的功能部件的SIMD机器（例如Connection Machine）来说可能是有效的，但是对于向量寄存器机器来说就不是最有效的。这是因为向量寄存器数量通常比单个循环的迭代次数少，因此向量化多个循环产生很少的附加益处。而另一方面，选择适当的内存访问模式对于这些机器来说却有很大的影响。

在大多数向量机上，最好向量化那些内存访问跨距为1的循环，因为存储器技术在传送连续的数据块时效率最高。由于Fortran是按列存储数组的（最左边一维的内存访问是连续的），这就是说最好向量化第一维——当前这个例子里的I-循环。如果I-循环被移动到最内层而另外两个循环的相对顺序保持不变，得到的方向向量就是(<,=,<)。这样I-循环不携带任何依赖，*codegen*对其向量化产生

```
      DO J = 1, M
        DO K = 1, L
S         A(2:N+1,J+1,K) = A(1:N,J,K) + B
        ENDDO
178   ENDDO
```

如果目标机是一个有向量执行单元的MIMD并行机，那么这个循环顺序虽然很好，但并非最佳选择。I-循环仍然是最佳的向量化循环（选择），但是K-循环作为剩下的惟一可被并行执行的循环，却不在最优的位置上。如果它被移动到最外层，生成方向向量(=,<,<)，它将可并行执行并将同步代价减少到原来的1/M。结果代码为

```
PARALLEL DO K = 1, L
  DO J = 1, M
    A(2:N+1,J+1,K) = A(1:N,J,K) + B
  ENDDO
END PARALLEL DO
```

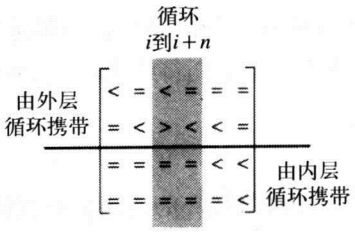
这些例子显示方向矩阵可以用于决定对某种特定目标机而言最好的循环置换。然而，穷举所有的变换不是最有效的办法；最好是先预测对给定目标机的最佳（循环）顺序，然后用方向矩阵来确认该顺序的安全性。这样一种“预测的方法”被证明不仅对于循环交换有利，而且对其他变换也是有好处的。下面一节将讲述如何把循环交换合并到*codegen*算法中并针对向量机作性能微调。

5.2.3 循环交换和向量化

对于向量化，从一个简单的观察可以得出一个有效和实用的循环交换策略。这个观察是这样的：一个不携带依赖的循环不会携带任何会阻止它和嵌套在其中的其他循环进行循环交换的依赖。并且，这样一个循环不可能携带任何交换敏感的依赖。没有阻止交换的依赖意味着将该循环移动到更深的嵌套层次总是合法的。在原来的层次上无依赖保证该循环在新的、更深的嵌套层次上也不会携带任何依赖。因此，向内移动的过程可以持续进行直到该循环被移到最内层，并且在那里该循环也不会携带任何依赖。因为一个无依赖的循环也是一个无依赖环的循环，而且向量化的目标是得到无依赖环的内层循环，这个交换方法对于向量化算法来说是理想的。以下的定理更形式化地陈述这些观察。

定理5.3 在一个紧嵌循环中，如果从第i层到第$i+n$层的循环不携带依赖（也就是说，所有的依赖关系由层号小于i或者大于$i+n$的循环携带），则将这些循环移动到第$i+n+1$层循环之内总是合法的。并且，这些循环在它们的新位置上也不携带任何依赖。

证明 为了证明定理的正确性我们考虑两种情况。首先，如果依赖由外层循环携带，如下面方向矩阵中的头两行所示，那么不管对内层的列做何种置换，相应的变换总是合法的，因为最外层的“<”将保证依赖的源语句在汇语句之前执行。



另一方面，如果依赖由内层循环携带，如图中最后两行所示，那么对应于i到$i+n$层的列只能包含为“=”方向的项，因为最外层的非“=”的项位于大于$i+n$的层。同样地，这些循环可以被移到内层，而依赖的携带循环保持不变。

定理5.3为把一组循环移动到最内层提供理论基础。这种循环交换基本上是把一组循环“移动”到另一个循环内，因此称为*循环移动*。关于怎样把这个结果用于执行循环交换，下面常见的矩阵乘法编码给出一个极好的例子:

```
      DO I = 1, N
        DO J = 1, N
          DO K = 1, N
S           A(I,J) = A(I,J) + B(I,K) * C(K,J)
          ENDDO
        ENDDO
      ENDDO
```

语句S有到其自身的真依赖、反依赖和输出依赖、它们都由K-循环携带且其方向向量均为(=,=,<)。如果将*codegen*直接应用于以上循环嵌套，将不会产生任何向量化语句，因为最内层循环有依赖环。在从最外层循环开始的每一级，递归的代码生成器会发现当前的层携带一个依赖环，这样就只能生成一个串行的DO循环，尽管外层的两个循环没有携带任何依赖。

179
~
180

然而，基于本节开始时的观察，很容易发现因为最外层的依赖由第3层携带，而循环1和2不携带依赖，可以被移动到循环3以内并且仍然不携带依赖。这样得到循环嵌套

```
      DO K = 1, N
        DO I = 1, N
          DO J = 1, N
S           A(I,J) = A(I,J) + B(I,K) * C(K,J)
          ENDDO
        ENDDO
      ENDDO
```

具有方向向量(<,=,=)。当*codegen*用于这个循环嵌套时，它会发现内层的两个循环既没有依赖也没有依赖环，因此可以在两维上被向量化:

```
DO K = 1, N
   FORALL (J=1,N)
      A(1:N,J) = A(1:N,J) + B(1:N,K) * C(K,J)
   END FORALL
ENDDO
```

为清楚起见这里使用FORALL循环。虽然二维向量语句可以写成Fortran 90语句而不使用FORALL结构，但是得到的代码却很难理解。

这个观察可以通过一个简单的方法合并到图2-2的算法*codegen*中——把携带依赖的最外层遗留循环移动到k层，然后为其生成串行执行的循环，而不是为k层的循环生成一个串行DO循环。根据定理5.3这总是合法的。在下一节我们将给出*codegen*的一个修改版本，在其中把这一修改合并到一个更通用的框架里。

一个代码生成框架

图5-2给出一个包含循环移动和打破依赖环的通用并行代码生成算法。在本章其余大部分地方，它将被用作代码生成的框架。

```
procedure codegen(R, k, D)
  // R是代码必须生成的区域
  // k是可能的并行循环最小嵌套层次
  // D是R中语句的依赖图
  在R的依赖图D中找到最大强连通分量集合[S1, S2, ..., Sm]（使用Tarjan算法）
  通过把每个Si约简为一个节点，从R构造出Rπ，相应地从D构造Dπ
  使[π1, π2, ..., πm]为Rπ中对应于Dπ的m个顺序排列的节点（按照拓扑排序进行排列）
  for i = 1 to m do
    if πi是带环的then
      if k是πi中最深的循环
      then try_recurrence_breaking(πi, D, k);
      else begin
           select_loop_and_interchange(πi, D, k);
           生成一个k层的DO语句;
           令Di为包含D中所有在k+1层或大于k+1层并且在πi内部的边的依赖图;
           codegen(πi, k+1, Di);
           生成k层的ENDDO语句;
      end
    else
      为πi在ρ(πi) - k+1维上生成一个向量语句，这里ρ(πi) 是πi所包含的循环个数;
end codegen
```

图5-2 带循环选择和打破依赖环的代码生成框架

这个版本与原来相比有两处改变。首先，如果一个区域是有环的，就用一个测试检查这是不是嵌套的最内层循环。如果是最内层循环，那么就试图打破依赖环。否则，就尝试由例程*Select_loop_and_interchange*选择一个循环进行串行化。回忆在原来的算法中，总是选择一个有环的嵌套的最外层循环。然而，在当前版本中，我们可以使用图5-3中的过程实现循环移

动的启发式方法。这个过程对于本节前边给出的例子，能够得到我们想要的结果。

通用的循环选择和交换

实际上，图5-3给出的循环移动算法是十分有效的。它不仅很容易实现，通常只需要很少的代码，并且也能采用最常见的下标编码形式得到最好的代码。不过，这个算法并不是完美的。比如在下面的例子中，它有可能错过向量化的机会：

181 ~ 182

```
procedure select_loop_and_interchange(π_i, D, k)
  if π_i所携带的最外层依赖在层p > k上，then
    把在k, k+1, ..., p-1层上的循环都移动到p-层循环内;
    使得它在k-层循环上;
  return;
end select_loop_and_interchange
```

图5-3 实现简单循环移动交换的循环选择

```
   DO I = 1, N
      DO J = 1, M
S        A(I+1,J+1) = A(I,J) + A(I+1,J)
      ENDDO
   ENDDO
```

在这个例子中，S有两个对自身的真依赖，方向矩阵为

$$\begin{bmatrix} < & < \\ = & < \end{bmatrix}$$

因为两个循环都携带依赖，所以循环移动算法将无法发现任何向量循环。然而，对方向矩阵的简单检查显示，交换这两个循环可以使得交换后的内层循环不携带依赖，从而允许它被向量化。得到向量代码如下：

```
DO J = 1, M
   A(2:N+1,J+1) = A(1:N,J) + A(2:N+1,J)
ENDDO
```

并且，这个向量循环有连续的内存访问，其好处前面已讨论过。鉴于在循环交换且向量化的版本和无循环交换也无向量化的版本之间执行时间的差别，对*codegen*进行修改以处理这样的情况是非常有益的。

像前边提到的，一个显然但不一定高效的处理任意循环交换的方法，是检查方向矩阵的所有合法的置换以得到最优形式，然后生成对应于置换的循环顺序。这个方法会在后面所有变换的完整的上下文都有时再重新考虑。对于向量化而言，所希望的结果是得到一个算法，比循环移动获得更多向量化，但是比检查所有置换的工作量小。 183

做这件事情的一个一般的方案是用通用的*循环选择*替换图5-3中的*循环移动*代码：选择一个在$p \geqslant k$层上的循环，它可以被安全地移动到层次k，并将$k, k+1, \ldots, p-1$层上的循环移动到它的内部。为了使这个策略有效，需要有选择在p层的循环的理由。特别是，它至少要携带一个依赖。

除了循环选择以外，也尝试了很多不同的启发式方法。在本节中，我们将讨论一种略好于循环移动的借助于检查方向矩阵的启发式方法。其基本策略如下：

（1）如果k层循环不携带依赖，那么令p层为携带依赖的最外层循环。（这种情况下与循环移动的启发式方法效果相同。）

（2）如果k层循环携带依赖，那么令p是可以被向外移动到k层的最外层循环，并且它携带一个依赖d，其方向向量在除了第p个分量以外的其他位置上都是“=”。如果没有这样的循环存在，就令$p=k$。

这个启发式方法的优越性在于它可以串行化任何必须被串行执行的循环，因为它携带一个串行执行其他循环所无法满足的依赖。这样它就可以正确地处理前边简单的循环移动失败的例子。图5-4给出实现这个更复杂的启发式方法的代码。

```
procedure select_loop_and_interchange(R, k)
  // k是在区域R中的当前嵌套层次
  // 当仅考虑在k层和更深层次的循环时，R是强连通的
  设N是最深层次的循环嵌套
  设p是携带依赖的最外层次
  if p = k then begin
    not_found = true;
    while (not_found and p < N) do
      if p-层循环可以被安全地向外移动到k层and
        存在一个循环携带的依赖d，它的方向向量在除了p以外的位置上都是“=”
      then not_found := false;
      else p := p + 1;
    end
    if p > N then p = k;
  end
  if p > k then把在k, k+1, ..., p-1层的循环移动到p-层循环以内;
end select_loop_and_interchange
```

图5-4 循环选择启发式方法

在充分探讨循环交换在向量化中的应用后，我们现在转向*codegen*中的依赖环打破。

5.3 标量扩展

如5.1节的矩阵乘法原来的编码显示，Fortran 77程序在涉及数组和向量的计算时经常使用标量临时变量。一个典型的例子是下面代码段，其中交换两个向量的元素：

```
      DO I = 1, N
S1      T = A(I)
S2      A(I) = B(I)
S3      B(I) = T
      ENDDO
```

图5-5中的这个循环的依赖图显示，这个循环在此形式下是不可向量化的。

本质上，向量交换必须读一个数组的元素然后再重写相同的元素。因此，由数组A和B导致的循环无关反依赖（$S_1\ \delta_\infty^{-1}\ S_2$和$S_2\ \delta_\infty^{-1}\ S_3$）无法消除。其他的依赖都是由标量T引起的。由于T被用于保存原存储单元将被重写的值，其导致的循环无关真依赖很难被消除或者改变。然而，其他的标量依赖都是由于同样的存储单元在不同的循环迭代中被用来临时存储空间而引起的。

特别是，循环携带的真依赖$S_1\delta_1S_3$是因为2.2节给出的依赖定义——S_1写入由S_2在下一次迭代中读出的存储单元——的蕴涵性导致的。如果每个迭代使用一个不同的单元作为临时变量，那么这些依赖就不复存在，从而得到图5-6所示的可以向量化的依赖图。

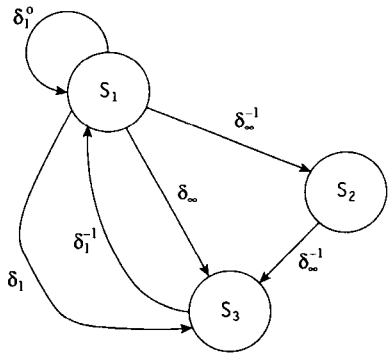


图5-5 向量交换的依赖图

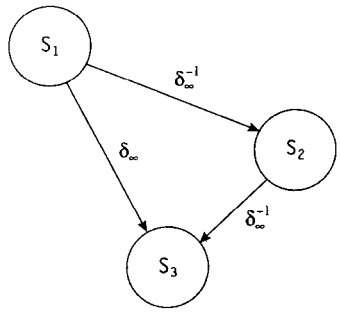


图5-6 向量交换修改后的依赖图

为了变换这个程序使其对应于这个循环嵌套，对标量T的引用必须用对编译器生成的临时数组T$的引用替换，这个数组对每个迭代有一个不同的存储单元。这个叫做*标量扩展*的变换产生如下的代码:

```
      DO I = 1, N
S1      T$(I) = A(I)
S2      A(I) = B(I)
S3      B(I) = T$(I)
      ENDDO
      T = T$(N)
```

在循环之后对T的赋值捕获在循环中计算出的T$的最后的值。将过程*codegen*应用于该循环得到

```
S1  T$(1:N) = A(1:N)
S2  A(1:N) = B(1:N)
S3  B(1:N) = T$(1:N)
    T = T$(N)
```

184
~
186

在6.2.1节，我们将讨论一种略有不同的方法——*私有化*，它通过声明存储单元T对于并行循环的各迭代是私有的来得到相同的结果。但不管是标量扩展还是标量私有化都不是没有代价的，因为这需要额外的存储和更复杂的寻址方式。因此，除非可以增加向量化或者并行性，否则不应该使用这两个变换。

为了说明标量扩展（像向量化）并不总是有利的，考虑下面的代码段，这是在有限差分计算中常会遇到的典型的循环:

```
DO I = 1, N
   T = T + A(I) + A(I+1)
   A(I) = T
ENDDO
```

由于在第一次赋值给T之前有一个对T的使用，标量扩展算法必须小心地插入初始化代码（和前边例子中的结束代码对称）。正确地进行扩展后的代码如下:

```
      T$(0) = T
```

```
      DO I = 1, N
S1        T$(I) = T$(I-1) + A(I) + A(I+1)
S2        A(I) = T$(I)
      ENDDO
      T = T$(N)
```

图5-7给出这个代码段的依赖图。我们看到，即使进行标量扩展，也不可能进行任何向量化。

从这两个例子读者容易看出标量扩展总是安全的。我们将要描述的扩展的实际过程简单且有些乏味，可以用于任何循环嵌套。可是，如我们所看到的，标量扩展并不总是有利的，而主要的挑战在于确定什么时候值得付出扩展所付出的代价。

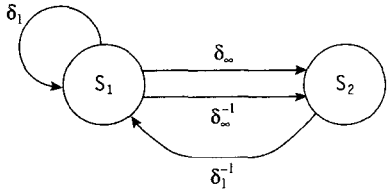


图5-7 扩展无利的依赖图

一个显然的办法是扩展所有的标量，生成最优的向量化，然后收缩所有不必要扩展的标量。虽然这个方法是可行的，但是更好的做法是确定什么时候变换肯定是有利的，由此避免不必要的扩展、测试和收缩标量的工作。和循环交换的情形一样，在实际变换程序之前，我们将尝试预测标量扩展在依赖图上的效果。这样，在进行代价高昂的修改循环的过程之前即可计算出获益，可以省掉不必要的变换的恢复工作。

一个确定标量扩展在依赖上的效果的方法可由这一节开始的观察直接得到。某些依赖由对内存单元的重用引起，另外一些则由对值的重用引起。由对值的重用引起的依赖必须被保持，因为任何删除这样一个依赖的变换都会导致计算结果发生变化。另一方面，由对内存单元的重用引起的依赖可以通过扩展相应的标量删除。如果一些边可以预先确定通过标量扩展是可删除的，那么就可以通过如下方式进行标量扩展：删除可删除的依赖边，计算新的依赖图中的强连通区域，只要新的依赖图中有可以向量化的语句才作标量扩展。

确定可删除的依赖边的一个关键概念由下面的定义引入。

定义5.4 某个标量*S*的定值*X*对循环*L*是一个*覆盖定值*，如果*S*在循环*L*开始处的某个定值不能到达出现在*X*之后的任何使用。如果在循环中有多个覆盖定值，那么“覆盖定值”一词指的是最早的那个定值。

在下面的循环中，S_1是T的覆盖定值：

```
      DO I = 1, 100
S1        T = X(I)
S2        Y(I) = T
      ENDDO
```

任何置于循环开始处T的定值将立即被S_1注销。因此不可能到达在S_2的引用。

类似地，S_1在下面的例子中也是一个覆盖定值：

```
      DO I = 1, 100
         IF (A(I) .GT. 0) THEN
S1          T = X(I)
S2          Y(I) = T
         ENDIF
      ENDDO
```

187
~
188

即使对T的赋值可能不是在循环的每次迭代都执行，但是它在每次引用T的迭代都会执行。因此，任何在循环开始的定值都不能到达S_1之后的引用。

覆盖定值并不一定总是存在:

```
      DO I = 1, 100
         IF (A(I) .GT. 0) THEN
S1         T = X(I)
         ENDIF
S2       Y(I) = T
      ENDDO
```

由于S_2是无条件执行而S_1是条件执行的，一个在循环开始处的定值总是有机会到达S_2。因此，T没有覆盖定值。用4.4.4节讨论过的静态单赋值形式来表述即为，如果从T的第一个赋值出发的边稍后到达循环中的一个φ函数，该函数将T的该值与其在另外一条通过循环的控制流路径中所得的值合并，那么变量T就没有覆盖定值。

为标量扩展的目的，考虑覆盖定值的一种更一般的解释是有用的，即将其扩展为在循环的不同路径上一组定值。如果在一个给定的循环中下面的两个条件之一成立，我们就说在该循环中存在一个T的覆盖定值的集合C:（1）在循环的开始不存在合并循环外和循环内T的不同版本定值的φ函数；（2）循环内的φ函数不存在到任何φ函数（包括它本身）的SSA边。下面是一个这样的循环的例子:

```
      DO I = 1, 100
         IF (A(I) .GT. 0) THEN
S1         T = X(I)
         ELSE
S2          T = -X(I)
         ENDIF
S3       Y(I) = T
      ENDDO
```

这里S_1和S_2形成一个覆盖定值的集合，因为在循环的开始处不需要φ函数，虽然在语句S_3之前需要一个φ函数。

按照这个扩展的定义，任何包含一个T的定值的单个循环都可以按下面的SSA图上的过程，通过在T未被覆盖定值的路径插入哑赋值而转换成有一个覆盖定值集合的循环，该过程同时计算出该覆盖定值的集合C。

（1）令S_0是T在循环开始处的φ函数，如果没有这样一个函数则为空。令C为空并初始化一个空栈。 189

（2）令S_1是T在循环中的第一个定值。把S_1加入到C中。

（3）如果S_1的SSA后继是一个不等于S_0的φ函数S_2，那么把S_2压入栈中并作标记。

（4）当栈非空时:

a）从栈中弹出φ函数S;

b）把所有不是φ函数的SSA前驱加入到C中;

c）如果存在一个从S_0到S的SSA边，则沿该边插入赋值T=T作为最后一条语句，并把它加入到C中;

d）对于每个是S的SSA前驱且未被标记的φ函数S_3（除S_0以外），标记S_3并且把它压入到栈中;

e）对于每个可从S经一条SSA边到达且在控制流图中不是由S控制的未标记的φ函数S_4，标记S_4并把它压入到栈中。

这个过程简单地持续寻找从第一个赋值到一个φ函数的并行路径，直到找不到为止。然而，在步骤（4）e）中，它避免把被当前φ函数控制的φ函数压入栈中，因为不存在到这样一个节点的未经考虑的路径。

当这一过程被应用到前面没有覆盖定值的循环时，将产生如下代码段：

```
      DO I = 1, 100
         IF (A(I) .GT. 0) THEN
S1         T = X(I)
         ELSE
S2         T = T
         ENDIF
S3       Y(I) = T
      ENDDO
```

覆盖定值是很重要的，因为它们决定标量被扩展的方式并因此而决定哪些边是可删除的。下面是一个在正规化的循环中对T作标量扩展的过程，假定覆盖定值被确定。

（1）创建一个适当长度的数组T$。

（2）对覆盖定值集合C里的每个S，把语句左部的T替换为T$(I)。

（3）对T的其他定值以及循环体中可以经SSA边到达而不越过S_0（在循环开始处的φ函数）的T的每个使用，用T$(I)替换T。

190 （4）对于在覆盖定值前的每个使用（SSA图中S_0的直接后继），用T$(I-1)替换T。

（5）如果S_0不为空，那么在循环前插入T$(0)=T。

（6）如果存在一条循环内的定值到循环外的使用的SSA边，则在循环之后插入T=T$(U)，这里U是循环的上界。

本章前面的例子说明了有覆盖定值的表达式将如何被扩展。下面是前面循环的一个扩展，前面的循环原来是没有覆盖定值的：

```
      DO I =1, 100
         IF (A(I) .GT. 0) THEN
S1         T$(I) = X(I)
         ELSE
            T$(I) = T$(I-1)
         ENDIF
S2       Y(I) = T$(I)
      ENDDO
```

有了覆盖定值的定义，我们现在可以确定所有可删除的依赖了。

定理5.4 如果一个标量T在覆盖定值的任一成员之前的所有使用都被扩展为T$(I-1)，而所有其他引用和定值都被扩展为T$(I)，那么用标量扩展后将被删除的边是（1）后向携带的反依赖，（2）所有循环携带的输出依赖，（3）在覆盖定值之前的循环无关反依赖，（4）冗余的前向携带真依赖。

为证明这一结论，下面逐个考查每一类依赖：

（1）*后向携带的反依赖*：覆盖定值集合的每个成员在循环体内都是向上暴露的；所有携

带的反依赖必须以这些覆盖定值之一或者某个后面的定值作为目标。类似地，由于反依赖是反向的，其源也在覆盖定值之后出现。这样两端点的引用都被扩展成T$(I)，但是由于每个迭代使用一个不同的存储单元，两端点引用的是数组内不同的单元，从而消除依赖。

(2) 后向携带的输出依赖：这种情况只需对前边的论证做一点小的修改。

(3) 前向携带的输出依赖：由于所有定值都出现在一个或多个覆盖定值之后，所有定值都被扩展为T$(I)，从而消除跨越循环迭代的重用。注意这种情况包括覆盖定值中的自输出依赖。 191

(4) 到覆盖定值的循环无关反依赖：在覆盖定值之前的引用将被扩展为T$(I-1)；覆盖定值将被扩展为T$(I)。由于引用不再出现在相同的迭代中，所以循环无关反依赖被消除。

(5) 从覆盖定值到覆盖定值后的使用的循环携带真依赖：覆盖定值注销任何迭代间的值的传递。

为了更具体地说明这些原则，再次考虑图5-6给出的向量交换的依赖图。在图5-8中，依赖图中可删除的边以Ds标明。边D_1是后向携带的反依赖，根据第1种情况是可删除的。边D_2是前向携带的输出依赖，根据第3种情况是可删除的。边D_3是一个前向携带的真依赖，根据第5种情况可以删除。

第4条原则从前边的例子来看不是很直观，但是一个新的例子可以清楚地说明这一点：

```
      DO I = 1,100
S1      A(I) = T
S2      T =  B(I) + C(I)
      ENDDO
```

图5-9是这段代码的依赖图，上面标出了可以被删除的边。边D_1是一个前向循环无关反依赖，根据情况4，它可被消除。标量扩展后的代码是

```
      T$(0) = T
      DO I = 1, 100
S1      A(I) = T$(I-1)
S2      T$(I) = B(I) + C(I)
      ENDDO
       T = T$(100)
```

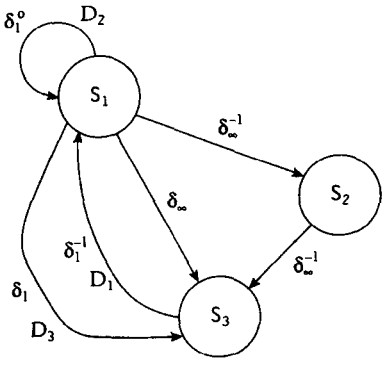


图5-8 向量交换中的可删除边

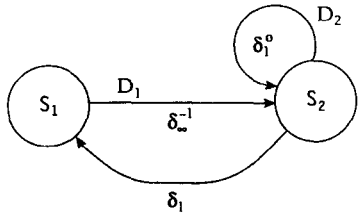


图5-9 循环无关的可删除边

这里反依赖消失了，因为存储操作总是在使用的前一个迭代。通过语句重排序，这个例子实际上可以被向量化为（正像变换后的图所示）

```
      T$(0) = T
S2    T$(1:100) = B(1:100) + C(1:100)
```

```
S1  A(1:100) = T$(0:99)
    T = T$(100)
```

值得注意的是定理5.4仅适用于标量在单个循环内被扩展的情形。它可以扩展为允许多个循环内的扩展，但是这一过程非常复杂，且带来的好处也远小于最初在单个循环内的扩展。

使用依赖图上预测标量扩展结果的能力，很容易把*codegen*修改为扩展标量。图5-10包含*try_recurrence_breaking*过程的一个版本，它由图5-2给出的代码生成框架调用。值得注意的是在“扩展可删除依赖边指示的标量”这一步，那些必须被扩展的标量只是那些由跨入或跨出一个向量π块的可删除依赖边指出的标量。

```
procedure try_recurrence_breaking(π_i, D, k)
  if k是π_i中最深层次的循环then begin
    清除π_i中可删除的边;
    找到依赖图D中仅限于π_i范围内的最大强连通区域集合[SC_1, SC_2, ..., SC_n]
    if在SC_i中有向量语句then begin
      扩展可删除边指示的标量
      codegen（π_i, K, D中仅限于π_i的依赖图）
    end
  end
end try_recurrence_breaking
```

图5-10 在最内层嵌套通过标量扩展打破依赖环

标量扩展的一个明显缺点是增加程序的内存需求。如果不小心处理，那么这一障碍可能会消除向量化和并行化所带来的好处。幸运的是，有一些技术可以缓和标量扩展所带来的存储问题。其中之一是只在单个循环中扩展标量，以最小的代价得到最大的利益。在多个循环内扩展会急剧增加内存需求，但是性能方面增加的好处却很小。第二种技术是在标量扩展前先进行循环分段，并仅扩展段循环。

例如，对程序段

```
DO I = 1, N
      T =  A(I) + A(I+1)
      A(I) = T + B(I)
ENDDO
```

针对一个寄存器长度为64的向量寄存器机器（假定N是64的偶数倍）分段，生成如下代码段

```
DO I = 1, N, 64
   DO j = 0, 63
      T =  A(I+j) + A(I+j)
      A(I+j) = T + B(I+j)
   ENDDO
ENDDO
```

在段循环中扩展更容易处理，此时得到

```
DO I = 1, N, 64
   DO j = 0, 63
      T$(j) =  A(I+j) + A(I+j)
```

```
      A(I+j) = T$(j) + B(I+j)
    ENDDO
ENDDO
```

192 ~ 194

这时临时存储的大小较小且在编译时就可以知道。并且，因为向量寄存器长度为64，可以把T$分配到向量寄存器中，而不需要额外的内存。本质上，原本要分配到标量寄存器的标量临时存储被扩展为向量临时存储，并被分配到向量寄存器。

第三种减少对存储需求的技术是前向替换。回到前边的例子，另一个消除任何额外存储（以及对标量扩展的需求）的方法是把T前向替换到它的惟一的引用，产生

```
DO I = 1, N
   A(I) = A(I) + A(I+1) + B(I)
ENDDO
```

在这个例子里，前向替换很明显是一个较好的方法，因为临时变量仅有一个引用。然而，一般说来，必须作一个折中：当临时存储被使用较多时，必须比较为存放扩展的标量而导致的存储的代价和重新计算替换的表达式所需的额外计算的代价。这种替换很容易并入归纳变量替换（见4.5节）。

5.4 标量和数组重命名

标量扩展有效地消除一些由对内存单元的重用而导致的依赖，即使是以多用内存为代价。实际上，对内存单元的重用尽管减小了程序的大小，但是也导致很多可以通过使用额外内存消除的“人为”依赖。以下代码段是另一种可被消除的标量依赖的例子。

```
     DO I = 1,100
S1     T = A(I) + B(I)
S2     C(I) = T + T
S3     T = D(I) -B(I)
S4     A(I+1) = T * T
     ENDDO
```

如果依赖图构造得不够精细，那么就可能出现依赖$S_1\delta S_4$。而且，即使是一个较精细的构造也会加入一个标量扩展无法消除的输出依赖$S_1\delta^0 S_3$，虽然根据依赖的定义，第一个依赖是存在的，但是这个依赖并不涉及值的传递，因为S_3阻止S_1的输出值到达S_4。第二个依赖的存在只是因为内存单元的重用。第二个依赖表示有两个不同的变量恰好存放在同一个内存单元中。如果它们被放到不同的内存单元，如像

195

```
     DO I = 1, 100
S1     T1 = A(I) + B(I)
S2     C(I) = T1 + T1
S3     T2 = D(I) - B(I)
S4     A(I+1) = T2 * T2
     ENDDO
     T = T2
```

则那个“人为”依赖就消失了，使循环可以完全向量化：

```
S3  T2$(1:100) = D(1:100) - B (1:100)
S4  A(2:101) = T2$(1:100) * T2$(1:100)
S2  T1$(1:100) = A(1:100) + B(1:100)
S1  C(1:100) = T1$(1:100) +T1$(1:100)
    T = T2$(100)
```

为了看出是标量重命名而非标量扩展，使得可以进行向量化，考虑下面标量扩展而没有重命名的代码片段：

```
      DO I = 1, 100
S1      T$(I) = A(I) + B(I)
S2      C(I) = T$(I) + T$(I)
S3      T$(I) = D(I) - B(I)
S4      A(I+1) = T$(I) * T$(I)
      ENDDO
```

如图5-11所示，这个例子的依赖图仍然是有环的。

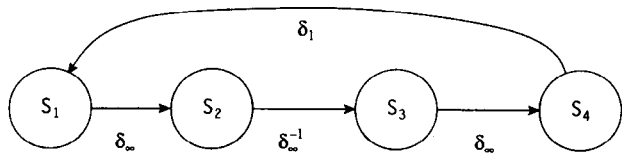


图5-11 不能仅由标量扩展打破的依赖环

定值-使用图可作为标量重命名的基础。图5-12描述一个把标量划分为不同等价类的算法，每个等价类可以被重命名为一个不同的值。简单地说，该算法反复地选取一个定值，加入该定值到达的所有使用，然后加入所有到达这些使用的定值，如此直到达到不动点。最坏情况
196 是所有定值和使用都在同一划分中，意味着无法进行任何重命名。

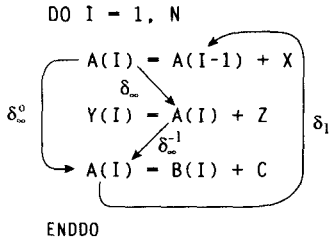
```
procedure rename(S, D)
  // S是被考虑重命名的标量
  // D是循环的定值使用图
  // rename把S的所有定值和使用划分为等价类，其中每个可以占用不同的内存单元
  partitions = Ø;
  defs_to_examine = Ø;
  令unmarked_defs = S中的所有定值;
  while (unmarked_defs != Ø) do begin
    从unmarked_defs中选出并删除一个元素s;
    把s加入到defs_to_examine;
    创建一个新的划分p;
    while (defs_to_examine != Ø) do begin
      从defs_to_examine中选出并删除一个元素s;
      把s加入到p中;
      for each s可到达的使用u do begin
        把u加入到p中;
        for each可到达u的def d ∈ unmarked_defs do begin
          从unmarked_defs中删除d;
          把d加入到defs_to_examine中;
        end
      end
    同样命名p中的所有元素;
    end
  end
end rename
```

图5-12 标量重命名

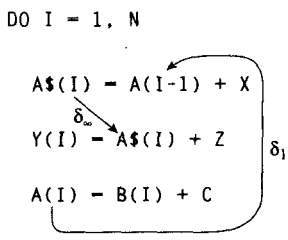
这个算法假定如果同一变量同时出现在赋值表达式的左边和右边，左边的定值不同于右边的使用。注意这个算法有一个略微不同的SSA形式的版本（见4.4.4节）。这种情况下，φ节点是同一变量的定值和使用。所有其他指令对相同变量的定值和引用都被作为不同的定值和使用。

在建立一个安全地重命名标量的算法以后，剩下的问题是什么时候标量重命名是有利的。一般说来，在类似于图5-11所示的情况下，标量重命名将打破依赖环，其中，依赖环形成的关键元素是一个循环无关的输出依赖或反依赖。不过，标量重命名基本上没有什么不好的副作用，它所导致的内存的增加在即使最坏情况下也是很少的，同时这种情况几乎不存在，因为几乎每个命名的标量都能分配到一个寄存器。而且，大多数优化编译器都是在寄存器分配中确定活跃区间时作标量重命名的。 197

标量重命名几乎没有代价，而类似的*数组重命名*却不是。正像标量内存单元有时为不同的值所重用一样，数组的存储单元有时也被重用，导致不必要的反依赖和输出依赖。考虑如下例子：



这个循环含有一个依赖环，因此不能用至目前为止讲过的算法将其向量化。这个依赖环和标量重命名例子中的一个依赖环类似：A(I)在循环中被用来传递两个不同的值。最后一条语句中的定值和第一条语句中的引用涉及一个值，而第一条语句中的定值和第二条语句中的引用涉及另外一个值。如果对第二组值用A$(I) 替换A(I)，计算结果相同，而代码变为：



正像在标量重命名的例子中那样，关键的反依赖和输出依赖被消除了。这打破依赖环，使得该循环能够被向量化：

```
A(1:N) = B(1:N) + C
A$(1:N) = A(0:N-1) + X
Y(1:N) = A$(1:N) + Z
```

198

很明显，数组重命名的安全性和有利性都比标量重命名的情形要复杂。标量重命名可以在有控制流图的情况下相对容易地实现，因为标量的“注销”可以容易地确定；而确定数组的注销就复杂多了。而且，数组重命名需要和数组大小成比例的额外内存空间（虽然我们将

简略地讨论如何避免这一点）——这一代价可能严重影响程序的性能。因此，数组重命名需要更加谨慎地实现。

对于向量化而言，数组重命名的好处是正确地消除重命名的引用和数组之间的循环无关反依赖和循环无关输出依赖。当这些依赖中的某个是依赖环存在的关键时，数组重命名可以打破那个环，从而可能产生向量语句。

理想情况下，通过检查依赖图和确定打破依赖环且可以由数组重命名消除的“关键边”的最小集合，数组重命名的有利性可以预先确定。可惜这个问题在这种形式下是NP完全的，因此没有精确有效的解决方法。然而，像标量扩展一样，如果可以找出通过数组重命名能消除的依赖边，就可以通过检查依赖图来预测数组重命名的效果，而不需要实际执行开销很大的程序变换。

和标量重命名一样，当同一数组单元在循环内重新定值时，使用数组重命名来增加并行性是很有效的。但数组重命名更为复杂，因为即使在没有控制流的情况下，确定两个数组引用是否访问同一数组元素也是很困难的，在有控制流变化时就更为困难。

幸好，如果没有控制流，可以利用依赖图将(可能的)数组引用划分成可重命名的区域。图5-13给出一个将依赖图划分为可重命名区域的算法，算法以下述简化假设为前提：

（1）循环体中没有控制流。

（2）数组A的每个使用或引用以A(I+c)的形式出现，c是一个常数。

实际的名字空间构造可以用6.2.5节将要介绍的一个合并算法来处理。

```
procedure array_partition(l, D, A)
  // l是当前考虑的循环
  // D是依赖图
  // array_partition代表引用数组A相同值的引用集合，
  // 使得所有输出依赖和反依赖边为可删除

  令{d1, d2, ..., dn}为数组A的定值集合;
  令人工定值d0表示在循环外的所有定值;
  把依赖图看作单个引用的集合;
  nameSpace(d0) := {所有依赖边的源点在循环中而汇点不在循环中的引用}
  for each定值di do begin
    nameSpace(di) := {di};
    for each从循环中di出发的真依赖δ do begin
      令u为δ的汇点引用
      if依赖图中不存在从di到u且走过输出依赖或者反依赖（循环无关或者由循环l携带的）的路径
      then把u加入到nameSpace(di);
    end
  end
end array_partition
```

图5-13 简单循环中数组重命名的划分算法

为了证明该算法是有效的，我们必须证明每个形如$u=A(i+c_u)$的引用被指派到恰好一个名字空间。特别，它被指派到与循环的迭代空间中离u“最近”的定值相关联的名字空间。设$d_x=A(I+c_x)$是使距离c_x-c_u最小的那个定值。如果这个距离是0，那么d_x在循环体中必须出现在u之前。

如果存在多个有相同最小距离的定值，选取循环体中较迟出现的定值（或者循环体中最靠近u的定值，如果距离为0的话）。我们可以断言不会有通过其他定值d_y的从d_x到u的依赖路径，因为那样要么从d_y到u的依赖有更小的距离，要么在循环中d_y在d_x之后，而这两种情况都违反假设。这样，循环中的每一个使用都在最近之前的定值名字空间中，或者是在缺省的$nameSpace(d_0)$。

一旦有了名字空间，代码生成就是很简单直接的了。每个名字空间都被给予一个不同的名字，其中一个名字空间仍然使用原数组名。不过，还有一个问题：我们怎样把最终的值存回到原数组中呢？这里的诀窍在于如何使得在开始或者最后的拷贝操作最少。我们使用的策略是选取将在最后作最多赋值操作的定值的名字空间，让它保持原数组名。这可能会导致在开始时生成一定数量的拷贝操作。我们也将对缺省名字空间使用原数组名，虽然这将在代码中留下一些反依赖。下面是该过程的概要：

（1）令$\{c_1, c_2, \ldots, c_m\}$是在循环内的定值中找到的增量常数，按照升序排列，集合中没有重复元素，并令$\{e_1, e_2, \ldots, e_n\}$是在循环内的引用中找到的增量常数，也按照升序排列，且没有重复。令e_k是某引用中的第一个常数，$e_k > c_m$或者$e_k = c_m$且存在一个引用A(I+e_k)位于任一形如A(I+c_m)的定值之前。然后令$nameSpace(d_0)$由{A(I+e_k),A(I+e_{k+1}),...,A(I+e_n)}组成。 199 ~ 200

（2）令d_1是循环体中常数为c_1的最后一个依赖。把原数组名和$nameSpace(d_1)$相关联；因为没有其他的定值会覆盖d_1写入的值，这保证将最大数量的值写到正确的数组中去。至于与循环内定值相关联的其他名字空间，只需任意选择名字即可。

（3）对$nameSpace(d_0)$也使用原数组名，因为在这个名字空间中的引用不可能是循环携带的真依赖环的一个部分。不过，可能需要作节点分裂以消除一些反依赖（见下节）。

（4）为原数组中没有指派到原数组名的每一个值插入收尾代码。如果我们采用这样的约定，d_i是与对A(I+c_i)的最后一个赋值相关联的定值，而A$i是和$nameSpace(d_i)$相关联的名字，那么我们需要在循环的最后插入如下所示的一系列从$i=2$到m的循环：

```
DO j = c(i-1) + 1, ci
      A(U+j) = A$i(U+j)
ENDDO
```

其中U是原来循环的上界。

（5）最后，我们需要为后续迭代中最后的定值$\{d_i\}$的名字空间中的任一使用插入初始化代码。令A(I+g_i)是A(I+c_i)名字空间中具有最小增量系数的使用，其中$g_i < c_i$。那么我们需要插入循环

```
DO j = gi + 1, ci
      A$i(j) = A(j)
ENDDO
```

我们用一个例子来对此加以说明：

```
   DO I = 1, 100
S1    A(I+2) = A(I+1) + B1
S2    A(I+1) = A(I+3) + B2
S3    A(I-1) = A(I) +B3
   ENDDO
```

这里语句S_1中定值A(I+2)的名字空间包含A(I+1)在相同语句中的使用；语句S_2中定值A(I+1)的名字空间包含A(I)在S_3中的使用；而语句S_3中定值A(I-1)的名字空间只包含它自己。 201
对循环输入的缺省名字空间仅包含在语句S_2中A(I+3)的使用。如果按照前面的算法并对S_3中的定值和S_2中的使用采用原数组名，我们将得到包括初始化的变换后的代码，在把单迭代循

环缩为单个语句以后，代码如下:

```
        A$1(1) = A(1)
        A$2(2) = A(2)
        DO I = 1, 100
S1        A$2(I+2) = A$2(I+1) + B1
S2        A$1(I+1) = A(I+3) + B2
S3        A(I-1) = A$1(I) + B3
        ENDDO
        DO j = 0, 1
          A(100+j) = A$1(100+j)
        ENDDO
        A(102) = A$2(102)
```

5.5 节点分裂

在某些特殊情况下，当应用上一节的重命名算法以后，我们仍然不能消除某个关键的反依赖。这是因为，为了避免复制，我们对同一语句的同一名字有两个不同的名字空间。考虑下面的例子:

```
DO I = 1, N
δ1  A(I) = X(I+1) + X(I)
      δ∞^-1
    X(I+1) = B(I) + 10
ENDDO
```

如果上述重命名算法应用于这个循环，在第一条语句中两个X的不同引用会在两个不同的名字空间中，因为第一个引用的是X在循环入口处的值（例如X(I+1)），而第二个（例如X(I)）是在循环内计算的值，只除了在第一个迭代。而且我们的命名算法试图对这两个名字空间都使用原数组名。这样，就使反依赖和依赖环保留下来。

当一个依赖环包含这样一个关键的反依赖时（即如果这个反依赖不存在，就不会有依赖环），它有可能可以通过熟知的节点分裂技术消除。节点分裂对反依赖的源节点生成一个拷贝；如果没有依赖以这个节点为汇点，那么这个依赖环将被打破。节点分裂生成了一个X数组的拷贝以提供对旧值的访问，从而允许语句被重排序。（在需要时，这个优化可以在重命名操作中自动地实现，只需给缺省的$nameSpace(d_0)$一个生成的名字并在循环外初始化即可。）

202

```
DO I = 1, N
      X$(I) = X(I+1)
δ1  A(I) = X$(I) + X(I)   δ∞^-1
    X(I+1) = B(I) + 10
ENDDO
```

当将*codegen*应用于变换过的代码时，至此就可以线性化这些依赖（因为环已经被打破），生成如下的向量代码:

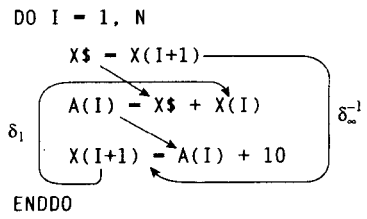
```
X$(1:N) = X(2:N+1)
X(2:N+1) = B(1:N) + 10
A(1:N) = X$(1:N) + X(1:N)
```

一旦确定需要消除的反依赖，实现节点分裂变换就很简单了。反依赖的惟一需要是源点引用只能使用数组的“旧”值。具有常数阈值的反依赖总能满足这个要求。图5-14给出假定在反依赖被删除时分裂一个节点的算法。这个算法只是以一个标量（假定后面将调用标量扩展）替换引用，并更新依赖图以反映程序的变化。

```
procedure node_split(D)
  // node_split对一个已知的，循环无关的反依赖，
  //  分裂这个依赖相应的节点，并且相应地调整依赖图
  设T$是一个还未在程序中使用的新标量;
  创建一个新的赋值x:  T$ = source(D)并且把它插入到source(D) 之前;
  用T$替换source(D);
  加入一个从x到source(D) 的循环无关真依赖;
  把source(D)改为x;
end node_split
```

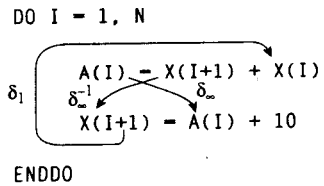
图5-14 节点分裂算法

容易看出，*node_split*是通过把源点移到一个不可能包含在任何依赖环中的语句来消除反依赖的。尽管节点分裂可以通过某种数组重命名来实现，但它常可以应用于比重命名的适用范围更广的循环。例如，考虑原来例子的一个变形:

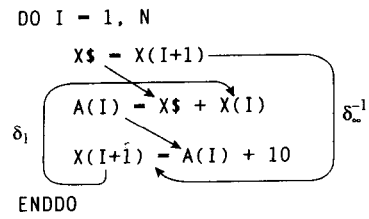


因为对X(2) 的引用，5.4节的数组重命名算法在这里不能使用，但是节点分裂依然可以进行。

虽然容易应用节点分裂，但是要确定它在某种特定情况下是否有利却不是很容易。下面的例子说明节点分裂并不总是能够打破依赖环:



应用图5-14的变换得到如下代码:



标量扩展可以使得生成的对X$的赋值能够被向量化，但其余的语句仍限制在一个依赖环内。
203 这里的问题是反依赖对于依赖环来说并不是关键的。为了使节点分裂生成能够有效的被向量
~ 化的代码，被分裂的依赖对于依赖环必须是“关键”的；也就是说，消除该依赖必然会打破
204 依赖环。遗憾的是，在一个给定的环中确定这样一个最小关键依赖集的问题是一个NP完全问题。因此，没有求最优解的有效方法。换句话说，对于产品编译器的开发人员来说，作一个完美的节点分裂可能不太现实。

所幸有了图5-14的方法，我们并不一定需要那么完美的节点分裂，因为变换的主要开销发生在临时标量被扩展时。如果仅当有把握产生有利的向量化时才作标量扩展，且有利性算法知道人为加入的赋值不会产生有利的向量化，那么只要可能就可以作节点分裂而不必担心有降低性能的危险。

由这些观察可以得出一个简单的节点分裂策略：选择依赖环中的一个反依赖，删除它，看结果是否无环。如果是，那么就应用节点分裂来消除那些反依赖。

我们把证明节点分裂的正确性作为一个习题留给读者，即证明当节点分裂应用于依赖环上的关键反依赖时，依赖环会被打破。然而，需要注意的是，即使依赖环被打破了，仍有可能不能向量化，因为原来的依赖环可能只是被分裂成了一组较小的依赖环。

5.6 归约识别

很多不能直接向量化的操作在程序中经常出现，以至于可以考虑在向量体系结构中使用一些特殊的硬件来处理它们。例如，把一个向量或数组中的元素累加起来是一个极为常见的操作，但它不能直接向量化。通过合并数组各元素而得到单个元素的过程叫做*归约*——因为这个操作把向量归约为一个元素。累加一个向量叫做sum*归约*。求一个向量的最大/最小元素是一个max/min*归约*。计算一个向量中的真元素的个数叫做count*归约*。

因为归约在很多重要的程序中出现，所以很多机器都有特殊的硬件或软件过程来计算它们。例如，考虑下面的sum归约：

```
S = 0.0
DO I= 1, N
   S = S + A(I)
205 ENDDO
```

如果假设浮点加是交换的和结合的（这个假设几乎不会是真正正确的，但是通常情况很接近，因此不必让程序员过于为此费心），那么归约可以被重写为一些并行的部分和，并在最后求它们的和。在一个有四级加法流水线的目标机上，一种自然的变换是把归约分解为四个部分求和归约：

```
S = 0.0
DO k = I, 4
   SUM(k) = 0.0
   DO I = k, N, 4
      SUM(k) = SUM(k) + A(I)
   ENDDO
   S = S + SUM(k)
ENDDO
```

将k-循环分布可以自然地把归约分解为三部分：初始化，计算，结束处理。

```
S = 0.0
DO k = 1, 4
   SUM(k) = 0.0
ENDDO
DO k = 1, 4
   DO I = k, N, 4
      SUM(k) = SUM(k) + A(I)
   ENDDO
ENDDO
DO k = 1, 4
   S = S + SUM(k)
ENDDO
```

因为目标机有四级流水线，我们希望对计算嵌套内的循环进行交换，得到

```
DO I = 1, N, 4
   DO k = I, min(I+3,N)
      SUM(k-I+1) = SUM(k-I+1) + A(I)
   ENDDO
ENDDO
```

内层循环不携带依赖，因此可以被向量化，得到

```
DO I = 1, N, 4
   SUM(1:4) = SUM(1:4) + A(I:I+3)
ENDDO
```

206

如果一个向量计算被抽象地视为同时发生的（实际上并非如此），则此形式同时将A的四个元素加到四个不同的部分和上。这些部分和再由外层循环累加起来（通常存入一组叫做*累加器*的特殊寄存器中）。图5-15图示显示在四级流水情况下的这个过程。

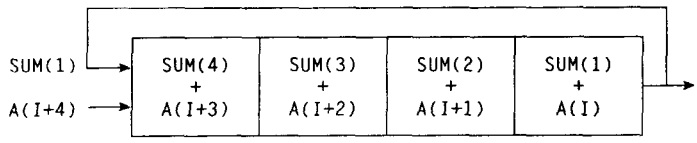


图5-15 sum归约的流水线

尽管向量的长度较短，这一形式仍然较类似的标量计算提高了速度。然而，需要注意的是，如果SUM(1)的结果可以在得到后就立即反馈给流水线（如图5-15左端所示），这个计算基本可以按向量速度执行——当结果从流水线一端出来，就立即被反馈到开头。一旦求得在SUM(1:4)中的四个部分和，总和可以通过三个浮点加法计算出来，其中两个可以重叠。类似的技术可以被用于计算乘积、min/max以及其他的归约。

Fortran 90的内部函数SUM被用来提供一种尽可能快的可交换和可结合的sum归约。这样，如果编译器能识别出sum归约循环并用适当的内部函数调用

```
S = SUM(A(1:N))
```

替换它，那么在任何有sum归约硬件的机器上都将得到高效的代码。Fortran 90内部函数PRODUCT，MINVAL以及MAXVAL分别在乘法、最小值和最大值的计算中扮演着相同的角色。

因此，对于先进编译器的编写者来说，问题是识别归约并在可能的情况下用等价但更快速的内部过程替换归约。归约有三个基本的性质:

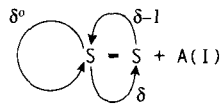
（1）它们把某个向量或者数组维的元素归约为一个元素。

（2）只有归约的最终结果会在后面用到；任何对中间结果的使用都避免归约。

207 （3）在中间累加过程中没有变化；也就是说，归约只对向量进行操作。

幸运的是，这些性质很容易用依赖图来确定。考虑前面提到的sum归约循环的依赖图：

```
DO I = 1, N
```




```
ENDDO
```

这个到自身的真依赖、输出依赖和反依赖的模式对于一个归约的存在是必要的。真依赖反映在每个迭代中语句加到部分累加结果的事实；输出依赖反映只有最后结果被使用的事实；而反依赖反映部分累加和会被重写的事实。这些依赖对于满足归约的第一个性质都是必要的。

第二个和第三个性质由不出现其他真依赖反映出来的。为举例说明不满足这些要求的一段代码，考虑下面的例子：

```
      DO I = 1, N
S1      S = S + A(I)
S2      T(I) = S
      ENDDO
```

归约的中间值被S_2使用——如果归约作为单个的操作执行就无法得到这些值。

第三个性质是与中间值的使用对称的，但是同传入归约的值有关。这个条件在没有真依赖传入归约时得到满足。

验证第二个和第三个性质使用的最简单的方法是在循环被分布时识别归约。如果两个条件都满足，那么在循环分布后，归约将变为单个语句的依赖环。如果其中任一条件不满足，那么归约因其累加到单个元素的性质，将会把其他语句也拉到依赖环里来。

由于归约很少可以充分利用向量机硬件，在识别归约时必须小心，确保归约的识别不会掩盖另外一种更高效的循环形式。考虑下面的例子：

```
DO I = 1, N
   DO J = 1, M
      S(I) = S(I) + A(I,J)
208   ENDDO
ENDDO
```

内层循环是一个sum归约且代码可以被替换为

```
DO I = 1, N
   S(I) = S(I) + SUM( A(I,1:M))
ENDDO
```

然而，外层的循环对S的所有元素进行归约。因此，它可以通过循环交换直接向量化，得到

```
DO J = 1, M
   S(1:N) = S(1:N) + A(1:N,J)
ENDDO
```

这个形式在大多数向量机上更加有效，表现为有适当的标量化和内存优化得以执行（如第8和第13章的描述）。因此，不能过早地在向量代码生成过程中通过插入等价的归约对代码

任意进行变换。相反，归约的插入应当在其他选择都考虑过以后再进行。

5.7 索引集分裂

有很多这样的情况，循环中的依赖环无法用到目前为止讨论过的方法打破，但是依赖的模式只在循环的部分迭代中保持。在这种情况下，*索引集分裂变换*，一种将循环划分为不同迭代范围的变换，可以被用来达到部分的并行化。在这一节我们将讨论一些这样的变换。

5.7.1 阈值分析

回忆一下，依赖的阈值是它最左边的非零距离的值。换句话说，阈值就是携带循环在访问依赖的源点和访问依赖的汇点之间的迭代的次数。因为必须有这两个访问依赖才能存在，一种生成向量化代码的方法是把循环分割成比依赖阈值还小的块。例如，下面的循环不携带依赖，因为它的上界恰好等于依赖的阈值: 209

```
DO I = 1, 20
   A(I+20) = A(I) + B
ENDDO
```

因此，它可以被直接重写成一个向量语句:

```
A(21:40) = A(1:20) + B
```

然而，如果迭代的数量增加了，依赖成为既成事实，而循环不能被向量化，如下面的版本所示:

```
DO I = 1, 100
   A(I+20) = A(I) + B
ENDDO
```

这个问题很容易通过循环分段将循环划分为不大于20的块而得到解决:

```
DO I = 1, 100, 20
   DO j = I, I+19
      A(j+20) = A(j) + B
   ENDDO
ENDDO
```

因为阈值的作用，内层循环是无依赖的；而外层的循环携带所有的依赖。因此，内层循环可以被向量化:

```
DO I = 1, 100, 20
   A(I+21:I+40) = A(I:I+19) + B
ENDDO
```

遗憾的是，这个变换在实践中不是很有用的，因为实际程序中的依赖的阈值通常都比较小——特别是，最常见的阈值是一个迭代。不过，这一分析对于某些特殊类型的程序是很有用的——例如，模拟动态存储分配的程序（它通过给一个主数组的不同段赋予不同的抽象数组）以及将多维数组作为一维向量接收的子程序。

虽然常数阈值是最常见的，但是*跨越阈值*在实际程序中也经常出现，因此值得对它们进行分析。在3.3.2节中简略介绍过的跨越阈值，出现在依赖的距离会变化，而所有依赖都跨越一个特定迭代的依赖中。下面的循环是一个例子:

```
DO I = 1, 100
```

```
S1      A(101-I) = A (I)  + B
210     ENDDO
```

S_1到自身的依赖的距离从I=1时的99变化到I=50时的1。在那之后引用反过来了，真依赖变成了反依赖。所有的依赖，不管是真依赖或者反依赖，其源点都在迭代50之前，而其汇点都在迭代50之后。因此，如果循环按长度50作循环分段，则所有依赖都将留在外层循环中，从而使得内层循环没有依赖环:

```
DO I = 1, 100, 50
   DO j = I, I+49
      A(101-j) = A(j) + B
   ENDDO
ENDDO
```

将*codegen*用于变换过的循环，得到下面的向量代码:

```
DO I = 1, 100, 50
   A(101-I:51-I) = A(I:I+49) + B
ENDDO
```

跨越阈值可以由依赖分析器以一种自然的方式计算出来，如3.3.2节所示。

鉴于这些简单的情况的处理实现起来很容易，且检测这些情况所需的编译时间也很少，因此尽管应用这一技术而获利的情况很少，在产品编译器中加入一些基于阈值的索引分裂还是可行的。特别是，后面几章将介绍阈值分析在向量化之外的一些实际应用。

5.7.2 循环剥离

一种更常出现的情况是循环有一个以单个迭代为源点的携带依赖:

```
DO I = 1, N
   A(I) = A(I) + A(1)
ENDDO
```

每个迭代中的计算都使用第一个迭代计算出来的A(1)的值。这个携带依赖可以通过剥离循环
211 的第一个迭代放到循环前缀块中的方法将其转化为从循环外来的循环无关依赖:

```
A(1) = A(1) + A(1)
DO I = 2, N
   A(I) = A(I) + A(1)
ENDDO
```

这样得到的循环不携带任何依赖，可以直接被向量化:

```
A(1) = A(1) + A(1)
A(2:N) = A(2:N) + A(1)
```

循环剥离可能涉及除第一个和最后一个迭代之外的其他迭代；在这种情况下，循环必须沿导致依赖的迭代进行分裂。例如，假设在下面的例子中，N恰能被2整除:

```
DO I = 1, N
   A(I) = A(N/2) + B(I)
ENDDO
```

我们希望通过循环分裂将循环携带依赖转化为两个循环间的循环无关依赖。这样得到以下代码段:

```
M = N/2
DO I = 1, M
   A(I) = A(N/2) + B(I)
ENDDO
DO I = M + 1, N
   A(I) = A(N/2) + B(I)
ENDDO
```

显然这是另一种形式的跨越阈值，并且循环的大小可以在依赖测试时很容易地确定。在此情况下，确定这种变换的测试是在3.3.2节中介绍过的*弱-0测试*。

5.7.3 基于区域的分裂

一个循环剥离的变体可以用下面的循环嵌套说明，这里，我们再次假设N恰能被2整除：

```
      DO I = 1, N
         DO J = 1, N/2
S1          B(J,I) = A(J,I) + C
         ENDDO
         DO J = 1, N                                    212
S2          A(J,I+1) = B(J,I) + D
         ENDDO
      ENDDO
```

在这个例子中有两个值得注意的依赖：从S_1到S_2（由B引起）的循环无关依赖和从S_2到S_1（由A引起）的由I-循环携带的依赖。因此，J-循环可以被向量化，但是外层循环被限制在一个依赖环内。因为S_1只定值B的一部分（B(1:N/2)），故一个消除循环无关依赖的很自然的方法就是把第二个循环划分为两个循环：一个使用S_1的结果而另一个不使用S_1的结果。下面的代码说明这样一个划分：

```
      DO I = 1, N
         DO J = 1, N/2
S1          B(J,I) = A(J,I) + C
         ENDDO
         DO J = 1, N/2
S2          A(J,I+1) = B(J,I) + D
         ENDDO
         DO J = N/2+1, N
S3          A(J,I+1) = B(J,I) + D
         ENDDO
      ENDDO
```

S_3现在与其他两条语句无关，因此*codegen*将分布I-循环并依照拓扑序把S_3移动到开始处：

```
      DO I = 1, N
         DO J = N/2+1, N
S3          A(J,I+1) = B(J,I) + D
         ENDDO
      ENDDO
      DO I = 1, N
         DO J = 1, N/2
S1          B(J,I) = A(J,I) + C
         ENDDO
```

```
        DO J = 1, N/2
S2         A(J,I+1) = B(J,I) + D
        ENDDO
     ENDDO
```

213 这样，S_3可在两维上被向量化，而其余的部分将只在J-维上被向量化：

```
     M = N/2
S3   A(M+1:N, 2:N+1) = B(M+1:N, 1:N) + D
     DO I = 1, N
S1   B(1:M,I) = A(1:M,I) + C
S2   A(1:M,I+1) = B(1:M,I) + D
     ENDDO
```

这种变换需要有对流经依赖边的数组区域的复杂知识和分析。例如，这个例子是基于我们知道仅有B的一部分在I-循环中被定义和使用这一事实。和阈值一样，如果对所有的循环都应用这一分析可能太复杂且代价太高。然而，在有过程调用的情形，这样的分析可能是值得的。第11章将介绍为确定过程间的影响而开发的这种分析。

5.8 运行时符号解析

在很多代码中的符号变量出现在下标中，这使得依赖测试变得复杂了。在那种情况下，依赖分析器必须作保守的假设。例如，在循环

```
DO I = 1, N
   A(I+L) = A(I) + B(I)
ENDDO
```

中，未知变量L阻止向量化。如果L大于0，那么该语句包含一个到自身的真依赖，而显然不能被向量化。如果L小于或者等于0，那么语句或者不包含依赖，或者包含一个到自身的反依赖，并不妨碍向量化。因为L很有可能在运行时定值，无法在编译时确定，因此在重构阶段通常只能假设真依赖和反依赖都存在。一种缓和这种保守性的方法是给依赖边加注*消除条件*，它们是一些逻辑表达式，其值为真时依赖即可被消除（见3.3.2节）。例如，在这个例子中，真依赖的消除条件是（L.LE.0），而反依赖的消除条件是（L.GT.0）。

当循环中有消除条件时，循环通常可以被有条件地向量化，即将其置于一个IF语句中以保证不存在的依赖足以打破依赖环。在前边的例子中，消除真依赖即足以打破依赖环，因此
214 循环可以如下向量化：

```
IF ( L .LE. 0) THEN
   A(L:N+L) = A(1:N) + B(1:N)
ELSE
   DO I =1, N
      A(I+L) = A(I) + B(I)
   ENDDO
ENDIF
```

消除条件的一个很常见的应用是在可变跨距的计算中。这些跨距在非常通用的支持对任意数组操作的库程序包中特别流行，如像LINPACK[106]。例如，下面的代码类似DAXPY（一种常见的线性代数操作，是LINPACK的基础）：

```
DO I = 1, N
   A(I*IS-IS+1) = A(I*IS-IS+1) + B(I)
ENDDO
```

如果IS恰好是零（实际上IS几乎从不等于零），这个循环就是一个归约，有一个由循环携带的真依赖、输出依赖和反依赖。如果IS是任意非零值，那就没有依赖，而这个循环可以被完全向量化。这种类型的结构很容易被依赖分析器检测到，从而生成如下检查消除条件时的代码:

```
M = N*IS - IS + 1
IF (IS .NE. 0) THEN
   A(1:M:IS) = A(1:M:IS) + B(1:N)
ELSE
   A(1) = A(1) + SUM(B(1:N))
ENDIF
```

在这个例子中，使用消除条件使得无论其值为真或假都能生成向量代码，因为在依赖存在时，循环就变成了一个求和归约。

一般说来，一个循环可以包含任意数量的消除条件。但正如确定通过数组重命名可以打破依赖环的最小依赖的集合是一个NP完全问题一样，确定打破一个依赖环所需的最少的消除条件也是一个NP完全问题。因此，在产品编译器中试图在很一般的情况下处理消除条件可能是不现实的。然而，本节给出的简单例子是从一些重要的数值计算程序包中摘出来的，说明至少应有一种可以处理这些情况的方法。这样我们建议使用一种类似于结点分裂中使用过的分析方法，来确定何时可以通过消除条件而有条件地消除一个关键依赖。 215

5.9 循环倾斜

到目前为止所给出的所有程序变换都是在探讨如何发现并行的循环迭代。尽管大多数程序有很规则的并行性，可以表示为并行循环，但一些重要的例子不是这样。*循环倾斜*是一种改变迭代空间形式的变换，可以把存在的并行性用传统的并行循环的形式表示出来。作为例子，考虑下面的循环:

```
      DO I = 1, N
         DO J = 1, N
                      (=,<)
S:          A(I,J) = A(I-1,J) + A(I,J-1)
                     (<,=)
         ENDDO
      ENDDO
```

从依赖关系便很容易看出，没有循环能够并行执行；I-循环和J-循环都携带依赖。然而，这段代码确实包含并行性，以图形的方式最容易看出。图5-16显示这个例子散布在迭代空间中的依赖。显然，没有循环可以并行执行，因为依赖边跨越两个坐标轴。任何沿某个坐标轴向下移动一行并行性的尝试都会失败，因为依赖关系使得任何和坐标轴平行的行只能串行执行。然而，在从左上角到右下角的对角线行上却存在着并行性。例如，一旦迭代S(1,1)执行完，迭代S(1,2)和S(2,1)就可以并行执行；类似地，在迭代S(1,2)和S(2,1)执行完以后，S(1,3),S(2,2)和S(3,1)可以并行地执行。换句话说，包含并行性的对角线可以从迭代空间中提取出来，从左下角开始进行到右上角。问题是，这一并行性并不能直接应用到任一循环。

一种旋转有并行性的对角线使其通过一个循环的方法是重映射迭代空间，同线性代数中找特征值非常相似。在这个例子中，一个可行的映射是创建一个如下的新索引变量j:

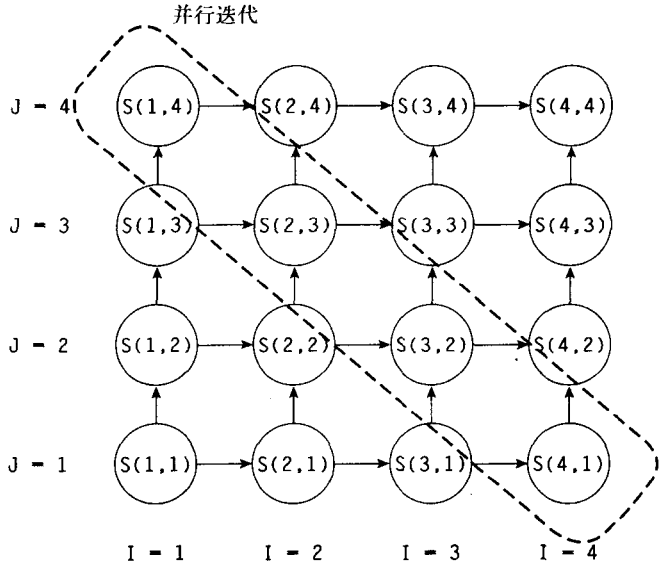


图5-16 循环倾斜前的依赖模式

$$j = J + I$$

使用以j替换J给出的逆映射

$$J = j - I$$

替换原来例子中的J，得到如下的代码:

```
DO I = 1, N
   DO j = I + 1, I + N
                    (=,<)
S:       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
                 (<,<)
   ENDDO
ENDDO
```

注意在迭代空间的变化会影响I-循环携带的依赖——在例子中位于图下方的依赖——的方向向量。为了使得左边的下标位置相等，在存储中使用的I的值必须比右边使用的值小1。当这个约束被传播到右边的下标位置时，它迫使j在引用中的值比在存储中的值大1，使得两个循环的方向都变为“<”。用Δ记号表示即为:

216
~
217

$$\Delta I = 1 \text{和} j - I = j + \Delta j - I - \Delta I$$

$$\text{因此，} \Delta j = \Delta I = 1$$

映射到迭代空间的新形状上的新的迭代模式如图5-17所示，图中的结点仍然以原来循环中的索引I和J标记。

因为两个循环都仍携带依赖，所以到目前为止仍不清楚循环倾斜除了使单纯的读者感到迷惑之外还做了些什么。然而，请注意现在外层循环携带的依赖是交换敏感的（图中对角线方向的依赖），从而可以这样生成一个并行循环: 交换两个循环，把所有的依赖移到新的外层循环中，留下没有依赖的内层循环。经过循环交换后例子变成

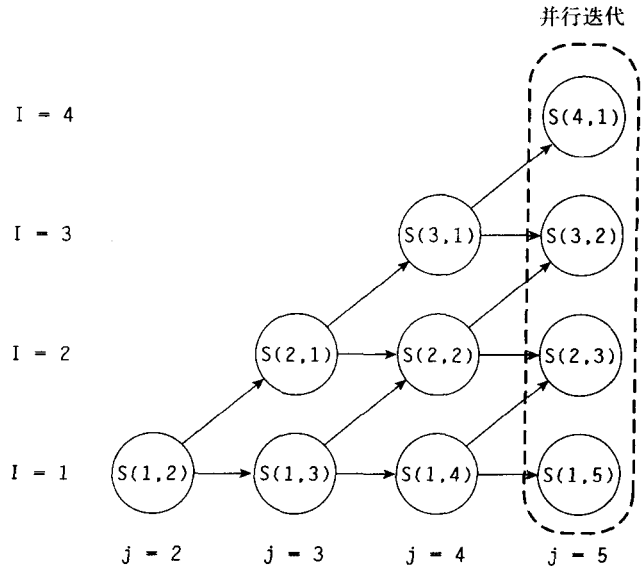


图5-17 循环倾斜后的依赖模式

```
DO j = 2, N+N
   DO I = max(1,j-N), min(N,j-1)
      A(I, j-I) = A(I-1,j-I) + A(I,j-I-1)
   ENDDO
ENDDO
```

例子中循环交换后的循环界是很典型的由循环倾斜和循环交换产生的循环界；倾斜的循环产生梯形的迭代空间，这种空间在循环交换后生成复杂的循环界。对这一形式的循环应用*codegen*，得到如下的向量代码: 218

```
DO j = 2, N+N
   FORALL (I=max(1,j-N),min(N,j-1))
      A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
   END FORALL
ENDDO
```

这里FORALL语句是需要的，因为向量语句无法直接用三元组记号表示。

虽然这个例子中通过循环倾斜得到的向量化可能会提高性能，但这种形式的向量代码存在某些缺点。主要的缺点是变化的向量长度——向量长度从1开始变到N，然后又回到1，平均的长度为N/2。如果向量的开始时间很大而N很小，那么这种向量形式很容易运行得比标量程序还慢。另一个缺点是向量界在外层循环的每次迭代都必须重新计算。

为了在更一般的形式下应用循环倾斜，假定我们有一个循环嵌套，其I索引的外层循环和J索引的内层循环都已正规化。我们希望倾斜内层循环使内层循环位置携带依赖的距离比原来循环大k。为此，我们引入一个新的内层循环索引:

$$j = J + k*I \tag{5-1}$$

并且对循环做变换以使用这个值。这意味着循环界必须被改变并且原循环中J的实例要被替换为表达式J=k*I的实例。这个变换的安全性可由如下事实得出: 我们并没有改变任何计算顺序——我们只是重新标注迭代空间。把证明这个变换可以在方向向量上得到所需要的效果留作习题。

相对于循环I以因子k倾斜循环J的效果是改变循环J的依赖距离。特别是，如果一个依赖相应于循环I的距离为ΔI而相应于循环J的距离为ΔJ，且这两个距离在整个循环上都是常数，那么在倾斜后内层循环的依赖距离就是

$$\Delta j = \Delta J + k*\Delta I \tag{5-2}$$

为看出这一点，令$f_1(I,J)$是在依赖的源点的（多维）下标表达式而$f_2(I,J)$是在汇点的下标表达式。我们知道

$$f_1(I,J)=f_2(I+\Delta I, J+\Delta J)$$

在等式的两边用j-k*I替换J，得到

219

$$f_1(I,j-k*I)=f_2(I+\Delta I, j-k*I-k*\Delta I+\Delta j)$$

但是因为该距离对I和J的各个值是一致的，我们即知f_2的第二个参数和f_1的第二个参数之差必然是ΔJ。这样就有

$$\Delta J = \Delta j - k*\Delta I$$

由此式立即可得式（5-2）。

式（5-2）告诉我们循环倾斜可以用于从两个都携带依赖的循环中产生一个无依赖的循环，只要所有的依赖距离是一致的。为此，使用本节前面描述的方法——相对于外层循环倾斜内层循环，使得外层循环携带的依赖在内层循环位置上的距离至少是1。如果外层循环携带的某些依赖在相应于内层循环的位置上以负距离开始，则可能需要以一个较大的常数倾斜。一旦内层循环的所有依赖距离都是正的，即可将其交换到外层循环，而它将会携带所有的依赖，使得新的内层循环可以被并行化。用一个最后的例子将说明其复杂性：

```
DO I = 1, N
   DO J = 1, N
      A(I,J) = A(I-1,J) + A(I,J-1) + A(I-1,J+1)
   ENDDO
ENDDO
```

这个例子与本节开始时的例子的不同之处在于它右边增加第三个引用，它导致一个方向向量为(<,>)和距离向量为(1,-1)的依赖。为了使循环倾斜后的外层循环和内层循环可以交换，我们必须以因子2倾斜以使这第三个依赖的距离在对应于J的位置上是正数。

```
DO I = 1, N
   DO j = 2*I+1,2*I+N
      A(I,j-2*I) = A(I-1,j-2*I) + A(I,j-2*I-1)$ + A(I-1,j-2*I+1)
   ENDDO
ENDDO
```

如我们已经看到的那样，循环倾斜不是没有代价的。由此变换得到的循环是高度梯形化的。这样一个不规则的循环在向量机上导致的代价最高；它们在异步多处理器上引起的问题则较小。然而，即使在那些体系结构上，应用循环倾斜也必须小心以避免过大的负载不平衡。

220

5.10 各种变换的集成

本章介绍了若干用于发现或提高并行性的变换：循环交换，标量扩展，标量重命名，数组重命名，结点分裂，归约识别，索引集分裂，符号解析，以及循环倾斜。有这样为数众多的变换的好处是为发掘并行性提供多种选择。不利的一面是，在这么多的选择中找出正确的

变换是一项十分复杂的工作，如果要将变换选择的过程自动化就更困难了。在选择变换时，至少有两点是必须要考虑的：确保所选择的变换确实使得程序相对于其原来的形式有所改进，以及确保选择的变换不会和其他对程序有更大好处的变换有冲突或者相互干涉。

关于第一个问题，本章中给出了很多例子。几乎对每种变换，我们都是从两个方面来处理的：确定何时变换是安全的以及确定何时变换是有利的。安全性问题总是可以明确地回答。但另一个方面，回答有利与否的问题却要困难一些，因为确定最有利的变换常常需要解一个NP完全问题。

不过，对于向量机，总是可以找到一个关于有利性的明确的测试：即选择导致最多向量化的变换。这可以通过执行变换来确定——最好只是在依赖图上进行，但如果必要也可以在程序上进行——并且把依赖图重划分为强连通区域。抽象地说，较多的向量化总是较好的。然而实际上，真实机器的体系结构总有一些问题领域，很难确定变换（一些例子见5.11节）。

第二个问题——避免变换间的相互干扰——则更加复杂。5.6节有一个这种干扰的例子：过早插入的一个求和归约阻碍了循环交换。此外有大量与循环倾斜有关的干扰的例子：5.9节关于循环交换的复杂性就是一个非常好的例子。

我们用下面的简单求和归约作为最后的例子，说明为什么过于简单的程序变换方法是不可行的：

```
DO I = 1, N
   S = S + A(I)
ENDDO
```

在这个形式下，这个循环很容易归约为一个求和归约调用，这在任何体系结构上都应该是最优的。然而，一个过于简单的编译器可能会在进行归约识别之前尝试标量扩展（尽管5.3节将说明这一扩展是没有意义的），产生如下的代码：

221

```
S$(0) = S
DO I = 1, N
   S$(I) = S$(I-1) + A(I)
ENDDO
S = S$(N)
```

这个循环仍然是一个归约，但是，要识别出它是一个求和归约就困难多了。在这种形式下，它更有可能被当作一个简单的线性递归并被一个求解例程替换。虽然这种结果比标量代码好些，但是不如直接用一个求和归约替换高效。

在开发一个算法将所有的变换集成起来时，有两点必须要考虑：首先，任何生成高效代码的方法必须以全局的观点看待被变换的代码，而不仅仅从局部去看。也就是说，仅就一个循环来考虑某个变换是否最有效是很困难的(如果不是不可能的话)。为确定这一点，至少需要考虑整个循环嵌套。下面的例子是关于所需的范围的类型：

```
DO I = 1, M
   DO J = 1, N
      A(I,J) = A(I-1,J-1) + B(INDEX(I),J)
   ENDDO
ENDDO
```

循环交换可以用于向量化其中一个循环，但不能两个都向量化——其中一个必须以标量形式执行。当只考虑其中一个循环时，变换算法很容易确定这个循环能否被向量化。然而，很难

确定向量化一个循环将妨碍另一个循环的向量化。在这个例子中，虽然I-循环的跨距是1，但向量化J-循环更为有利，因为向量化I-循环所需的B上的收集操作在大多数机器上都是低效的。

这一原则可以很好地扩展到向量机以外的体系结构上。从全局的观点来决定最优代码的生成是相当困难的；但若纯粹从局部的观点看则完全不可能。

其次，对于一个给定的程序，决定哪种变换最好需要目标机体系结构的知识。在一个高度并行的机器上，例如 Connection Machine，发掘所有可能的并行性是最基本的，必须着眼于对程序进行变换以建立尽可能多的并行循环。在较少并行性的体系结构上，例如单处理器的超标量体系结构，选择一个合适的并行循环远比生成大量并行循环重要。因此，以同一种策略将程序变换集成起来是不可能同时满足两个极端（或其间的大多数）的体系结构的需求的。

为了说明全局变换中所涉及的一些原则，本节考察一个为向量寄存器体系结构而调整过
222 的算法。为了说明方便起见，这些体系结构适合于处理中等程度的并行性，其好处较易确定且易于以Fortran 90的向量操作在源代码级表示出来。我们假定主要的目标是找出一个好的向量循环；向量化其他循环的好处太小，不值得花那个代价。这一假设对于目前大部分商用的向量寄存器体系结构来说都是正确的。

给定目标体系结构，从标量源程序生成向量代码的过程可以自然地分为三个阶段：

（1）检测：找出每条语句都可以向量方式执行的所有循环。

（2）选择：从每条语句都可以用向量方式执行的所有可能候选循环中选出最佳的。

（3）变换：对选定的循环执行向量化所需的变换。

这一策略很适合本章所介绍的变换。检测阶段涉及修改依赖图以反映所有可能的变换，而不必在程序上执行它们。因为这里描述的变换只会提高并行性(没有变换会阻碍并行性)，以这种方式修改依赖图不会造成不良影响。选择阶段则是与机器高度相关的，并且涉及到详细的下标和循环分析。最终的变换阶段使用原来的依赖图来驱动所需的对程序的修改以得到最佳的向量循环。

图5-18的过程*mark_loop*给出对检测阶段的更为详细的描述。这个算法从依赖图中删除可以被标量扩展、数组重命名、节点分裂和符号解析消除的所有依赖。循环交换的处理方法是搜索不携带依赖的循环。索引集分裂和循环倾斜作为最后的手段——在没有其他的向量化机会时才使用它们。归约在标记过程中识别，因为它们在循环分布后最容易发现。

```
procedure mark_loop(S, D)
  // mark_loop对一个语句集合S和相应的依赖图D（其中标出了所有的可删除边）在
  //  所有可能向量语句上标记所有循环
  for each可以通过数组重命名，标量重命名，节点分裂或者符号处理可删除的边e do begin
    把e加入deletable_edges;
    从D中删除e;
  end

  mark_gen(S, 1, D);

  for each没有向量循环标记的S中的语句x do
    尝试索引集分裂和循环倾斜，标记找到的向量循环;
  for each deletable_edges中的边e do
    把e重新放回到D中;
end mark_loop
```

图5-18 向量循环检测

在依赖图上作了这些改动之后，我们将调用*codegen*的一个变体，称为*mark_gen*（见图5-19）。然而，*mark_gen*只是简单地为检测到的向量循环和所有相关的语句加上标记，并不生成向量代码或执行循环交换。

```
procedure mark_gen(S, k, D)
  在依赖图D中仅限于区域R的范围内找出最大强连通区域集合{S1, S2, ..., Sm}（使用Tarjan算法）;
  for i = 1 to m do begin
    if Si是带环的then
      if最外层的携带依赖在层p > k then begin
        对所有在Si中的语句标记所有在层k, k+1, ..., p-1的循环为向量;
        mark_gen(Si, p, Di)
      end
      else if Si是一个归约then begin
        为Si标记循环k为向量;
        标记Si中的语句为层k的归约;
      end
      else begin
        设Di是依赖图D中πi范围内所有在k层或大于k层上的依赖边组成的依赖图
        mark_gen(Si, k+1, Di);
      end
    else
      为在k和更深层嵌套的循环Si中的语句标记为向量
  end
end mark_gen
```

图5-19 标记向量循环和语句的*codegen*的变形

这一过程使用简单的循环移动作为其交换策略。不过，容易对其加以修改，采用像图5-4介绍的那样更为复杂的循环选择启发式方法。

全局代码生成算法的第二个阶段，从一条语句所有可能的向量循环中选出最佳的向量循环。这个阶段按照一些机器相关的标准，例如5.11节讨论的标准，对每条语句作局部的检查。然而，一些非局部的分析对于保证在应用变换方面的一致性是必要的。例如，在标量扩展中，单个标量必须为每个重命名的划分跨越相同循环进行扩展。尽管把循环

```
DO I = 1, M
   DO J = 1, N
      T = A(I,J) + B(I,J)
      C(J,I) = T + B(J,I)
   ENDDO
ENDDO
```

扩展为下面的形式可能有吸引力(它通过建立跨距为1的访问对单独的语句产生最大化结果):

```
DO I =1, M
   DO J = 1, N
      T$(I) = A(I,J) + B(I,J)
      C(J,I) = T$(J) + B(J,I)
   ENDDO
ENDDO
```

但结果显然是不正确的。如果只是单独考察这些语句而没有全局的观点，很容易出现这种类型的代码结果。

一旦对每条语句找到了一个最佳向量循环的一致集合，剩下的任务就是在程序中执行变换——换句话说，使程序对应于依赖图。依赖图提供一种简单的机制，可有效地发现一种简单的路径以识别用来向量化所选择的循环的正确变换。原来的依赖图被恢复(即那些可以删除的边被恢复)，然后*codegen*被再次调用。一旦*codegen*找到一个没有向量化的“最佳向量”循环，它就知道必须调用一个变换。由于所需的变换已知，那些确定潜在的程序变换的边很容易在这个过程中找到。因此，*codegen*可以有效地寻找能够产生所需结果的正确的变换集合。图5-20和图5-21详细描述实现这一过程的*codegen*版本。它一开始在最外层循环被调用。如果它返回时没有生成向量代码，但最佳向量循环已被标出，那么就必须进行循环倾斜或者索引集分裂。

223
~
225

```
procedure transform_code(R, k, D)
  找到依赖图D中R范围内的最大强连通区域集合{S1, S2, ..., Sm}（使用Tarjan算法）
  通过把Si归约为单个的节点，从R构造Rπ，然后自然地根据D从Rπ推导出来的依赖图，计算Dπ
  设{π1, π2, ..., πm}是Rπ的m个节点，它们按照某种顺序排列（使用拓扑排序以计数）
  for i = 1 to m do
    if k是πi中某些语句最好的向量循环then
      if πi是带环的then begin
        select_and_apply_transformation(πi, k, D);
        // 重试在新依赖图上的向量化
        transform_code(πi, k, D);
      end
      else
        为循环k中的πi生成一个向量语句
    else begin
      生成一个k层DO语句
      设Di为依赖图D中仅限于πi范围内在层k+1以及大于k+1上所有依赖边组成的依赖图
      transform_code(πi, k+1, Di);
      生成层k的ENDDO语句;
    end
end transform_code
```

图5-20 程序变换的驱动程序

```
procedure select_and_apply_transformation(πi, k, D)
  if循环k不携带πi内的依赖then
    移动循环k到最内层的位置;
  else if πi在k层是归约then
    用归约替换并且把和归约相连的边消除掉
  else // 变换并调整依赖
    if数组重命名是可能的then
      应用数组重命名并调整依赖关系，在需要的地方使用结点分裂;
    else如果结点分裂是可能的then
      应用结点分裂并调整依赖关系;
    else if标量扩展是可能的then
      应用标量扩展并且调整依赖关系;
    else
      应用倾斜或索引集分裂并调整依赖关系;
end select_and_apply_transformation
```

图5-21 变换的选择

5.11 实际机器的复杂性

向量代码生成的选择阶段决定最佳向量循环，这个阶段看起来可能是最简单的，实际上，它是较为困难的过程之一，因为它必须考虑目标机器的体系结构。当为细粒度并行的机器生成代码时，重要的是保证在最内层循环选择的操作和可用的功能部件匹配。当为粗粒度多处理器生成代码时，保证并行分解能提供足够的计算以克服同步的开销且保持负载平衡则是一个困难的问题。

由于有这些问题，你可能认为向量机是最简单的目标机，而且前一节的简短讨论已覆盖了这个题目。但实际上正相反；确定最佳向量循环（甚至只是保证在向量化以后执行得更快的循环）是一个非常困难的问题。本节提出试图为某个特定的机器确定最佳向量循环时必须考虑的一些问题。当面对真实而非理想的机器时，这些问题都是很典型的。

内存跨距访问 体系结构设计中很难达到的一个平衡是CPU性能和内存性能之间的平衡。这一平衡的问题在向量机中尤为棘手，因为高度流水线的CPU在完成向量操作的过程（可能较长）中的每个时钟周期内都会需要操作数。速度快到能够满足这样的CPU的内存部件远比CPU昂贵，很难让人认为那是值得的，因为向量机器要执行的远不仅是向量访存操作。

为了避免导致过高的代价，系统设计者通常使用比CPU慢一些的存储部件，但是它们被设计为能够在大多数操作上提供与CPU相同的速度。两个标准的体系结构特性是多体存储器和预取。多体存储器将所有的内存划分为一些体（通常是2的较小的幂——8或16是常见的）。单个体内的访问需要的时间多于一个周期，但是对多个体的访问可以重叠，在启动延迟之后即可每个周期提供一个操作数(和流水线功能部件的工作方式基本相同)。当然，只有在连续的访问是到不同的体时，才能维持这样的访问速率；重复访问相同的内存体会导致类似于流水线的互锁问题，且同样会导致内存访问速率的下降。由于连续的内存地址通常是以巡回的方式分配给不同的体（即地址x分配到体0，地址$x+1$分配到体1，依此类推），跨距为1的访问在这样一个系统上肯定有较好的效果，因为连续的内存访问肯定是到不同的体上的。另一方面，跨距等于体的大小的向量访问的性能将会很差，因为重复访问同一个内存体。在一个以这种方式使用内存体的系统上，在向量化操作时避免体冲突是非常关键的。注意跨距为1的访问并不总是能避免体冲突；如果内存是沿字边界来划分体的，则对半字元素的向量访问在跨距为2时性能会更好。

226 ~ 227

采用硬件预取时，对某个特定字的访问会使得后续的一些字也被访问。当向量跨距为1时，这一技术自动地把后边的一些操作数从内存取出，使得处理单元可以全速执行。在跨距不等于1时，预取就不是那么有效了，因为仅有部分(或没有)预取的操作数被用到。当选择最优的向量循环时，从预取的角度，肯定倾向于较小的向量跨距。

分散-收集 一个和跨距密切相关的内存因素是分散-收集操作。收集出现在程序使用一个索引向量将稀疏的操作数收集到一起的时候，例如在

```
DO I = 1, N
   A(I) = B(INDEX(I))
ENDDO
```

中。类似地，一个分散操作把压缩的操作数散布到未压缩的内存单元中去:

```
DO I = 1, N
A(INDEX(I)) = B(I)
ENDDO
```

因为收集和分散都涉及到变化的、未知的跨距，它们总是比直接的内存访问效率低一些，由于同样的原因不等于1的跨距也会引起问题。分散-收集在科学计算代码中是很常见的，因此向量化这样的结构是很重要的；然而，也应认识到，即使是向量化了的分散和收集循环，通常也要比直接内存访问的执行效率低很多。

循环长度 所有向量部件在一开始注满流水线时都会导致一些开销，相应地需要有一定数量的操作以使分段执行的加速比能超过启动开销。向量化的循环越长(当只有一个循环被向量
228 化时)，向量部件就越能够有效地分摊启动开销。当循环长度在编译时间全部已知时（这种情况很少出现），编译器就可以估计每个循环的向量化效率。当循环界是符号表达式时，在循环长度与步长和其他参数之间作出折中是非常困难的。如果程序员不提供其他的输入，编译器通常必须假定所有具有未知界的循环对于有效的向量执行来说足够长。这一假设会在很多领域导致非常令人不快的低效率。例如，图形程序通常有长度为4的循环；物理学和化学的代码通常的循环长度为3（x，y，z方向各为1）。当这样的程序被简单地向量化以后，生成的变换过的代码通常运行比原先的程序慢很多。

操作数重用 优化内存访问的最优方法是最小化内存访问的数量。向量化以使操作数可在寄存器中被重用是实现这一目标的一个显然的方法。本章给出了一些基于这种考虑的例子；在第8章和第13章会有更多例子。

不存在的向量操作 不是所有的算术操作都可以有效地分段，也不是所有的体系结构都会为所有的指令提供向量版本的支持。很难用流水线的算术操作的一个常见的例子是浮点除法。浮点除法是一条复杂的指令，需要基本指令序列的很多个迭代才能完成。因为这个迭代的过程，除法在向量化时很少加速，因此很多体系结构都不浪费指令位去支持向量化的除法。因而向量化一个执行向量除法的循环就成为编译器必须仔细考虑的问题。在下面例子

```
DO I = 1, M
   DO J = 1, N
      A(I,J) = B(J) / C(I)
   ENDDO
ENDDO
```

中，在大多数机器上不考虑步长和内存的因素，J-循环是首选的向量循环。当J-循环被向量化以后，除法可以通过计算标量倒数并将除法变为乘法而得以高效执行（假定程序员愿意接受精确度稍差的结果）：

```
DO I = 1, M
   T = 1.0 / C(I)
   DO J = 1, N, 32
      A(I,J:J+31) = B(J:J+31) * T
   ENDDO
ENDDO
```

229 如果I-循环被向量化，绝大多数机器都不会有任何加速。

条件执行 向量部件持续重复相同的操作，因而在处理一系列规则的操作数时性能最优。引入条件语句（因此一些操作被跳过）破坏这一规则性，将大大降低向量化效率。大多数向量部件能够通过一组控制单个操作结果是否覆盖结果寄存器的*掩码寄存器*来执行条件操作。然而，掩码寄存器通常不允许条件置位（就是说，掩码寄存器不能在掩码寄存器的控制下置位），因此只能支持一层IF嵌套。即使只是一层的条件执行也要比纯粹的向量执行效率低，因

此应当尽量避免。这样，在例子

```
DO I = 1, M
   DO J = 1, N
      IF (A(J) .GT. 0 ) THEN
         B(J,I) = B(J,I) + 1.0
      ENDIF
   ENDDO
ENDDO
```

中，最好是向量化I-循环，因为它可以被变换成

```
DO J = 1, N
   IF (A(J) .GT. 0 ) THEN
      DO I = 1, M
         B(J,I) = B(J,I) + 1.0
      ENDDO
   ENDIF
ENDDO
```

从而消除了向量流水中的条件执行。

像这些例子所指出的那样，即使是选择最优的向量循环这样似乎很简单的任务其实也并不容易。当这个任务在有粗粒度并行性(以多个CPU的形式)或不规则的细粒度并行性的情况下被进一步复杂化时，编译器的任务就实在是非常困难了。

5.12 小结

本章所发展的理论可支持很多基于依赖的用来提高程序中并行性的变换。这些面向循环的变换，或者侧重于重新安排程序以求得到更好的并行性，或者着重在打破依赖环来创造并行性: 230

- *循环交换*，用于调节循环的嵌套顺序从而将并行循环放在最优位置。它不仅对于向量化而且对于本书中介绍的大多数方法都是很关键的。因为串行执行外层循环会使得内层循环有更多向量化的可能性，一个好的循环交换策略的要点在于选择适当的外层循环串行执行。
- *标量扩展*，当标量瓶颈限制执行顺序时，它可通过把标量扩展为向量而消除依赖，使并行成为可能。代价是额外的内存开销。
- *数组重命名*、*标量重命名*以及*节点分裂*，也通过使用额外的内存来删除依赖。
- 归约在很多代码中是很重要的一部分，使得*归约识别*成为任何先进编译器的一个完整的组成部分。
- *索引集分裂*、*符号解析*以及*循环倾斜*，是应用于特殊情况的另外一些暴露并行性的技术。

这些变换必须在考虑到可能应用于同一个循环嵌套的其他操作的情况下小心地使用，以确保得到最大限度的并行性。本章介绍一个向量机上这些变换的驱动过程。这一过程包括三个阶段: *确定*可以被向量化的循环，*选择*应被向量化的最优循环，*变换*代码来向量化选择的循环。作为结束，本章最后讨论在真实的向量机上为每条语句选择最优向量循环时需要考虑的一些问题。

5.13 实例研究

因为向量化是PFC系统和Ardent Titan编译器的一个主要侧重点，本节将详细讨论它们的

能力。此外，关于PFC和一些商用编译器在处理一组用于测试向量化覆盖宽度的循环时的表现，本节给出一些研究结果。

5.13.1 PFC

由作者在Rice大学开发的PFC系统，使用图5-2和5-3的循环移动策略来向量化代码。如果*codegen*在到达最内层循环时还未找到任何向量化的机会，它将尝试对最内层的两个循环作通常的循环交换。这一方法可以通过在依赖边上设置一个属性——即是否阻止和下一个内层循
231 环的交换——来实现，而不必通过计算和检查整个方向矩阵。这一属性很容易在依赖测试时通过在携带循环和位于其内层的循环上寻找显式的方向偶“<，>”进行计算。

为了暴露更多的向量化机会，PFC在标量扩展、标量重命名和节点分裂作了相当完整的工作。它使用了较为复杂的索引集分裂，包括阈值分析、交叉阈值以及循环剥离。它还包括一种简单形式的运行时解析，但没有尝试作循环倾斜和数组重命名。

鉴于PFC生成Fortran 90而不是任何目标机的代码，因此它没有包含与机器相关的考虑。变换的选择使用了特殊的方法。确切地说，若*codegen*在到达最内层循环时仍未产生任何向量化，且对内层的循环进行交换也于事无补，那么就从标量扩展开始，按照特定的顺序尝试各种变换，以消除依赖边。

5.13.2 Ardent Titan编译器

在Ardent Titan上的向量寄存器文件的灵活性允许使用范围很广的一组向量化策略。在一个极端，寄存器可以被当作4个长度为2K的向量寄存器。在这一模式下，Titan本质上可以是一个向量存储机，而正确的策略将是向量化尽可能多的循环以建立可能达到最长的向量操作。

另外一个选择更加合理，并且允许向量寄存器可能的重用——这是4个向量寄存器所不允许的。Titan Ardent编译器把向量寄存器文件大致划分为64个长度为32的向量寄存器。[⊖]这个长度足以补偿向量启动代价，但允许更大量的寄存器重用，且给操作系统留下一些空间，使得上下文切换更为有效。

小的向量启动代价和短的向量长度使得Titan可以对其向量化策略作一个重要的简化。Ardent编译器侧重于向量化一个循环而不是向量化尽可能多的循环。这里有几个理由支持这一决策：

（1）由于向量寄存器的长度被设为32，有很大的可能性只需向量化一个循环就能提供足以充满向量部件的操作。

（2）由于向量部件是异步的，因此如果采用正确的编译器调度，那么调度大量小的向量操作可以和发出一个长的向量操作一样快。因此，即使一个向量化的循环碰巧没有充满向量
232 寄存器，也不会有太大的性能损失。

（3）仅向量化一个循环可以简化很多表示的问题。当仅有一个循环被向量化时，所得到的线性化向量操作总是可以简单地用一个三元组表示。并非所有的多重向量化的循环都能如此，并且，选定一个循环意味着优化器可以避免为了确定一个语句是否可以在多个循环内被正确地向量化而作的很多代价高昂的分析。

（4）把向量化限制在一个循环内可以大大简化编译时许多关于最优循环顺序的查找问题。很多涉及向量循环顺序的问题有相对于循环个数的指数级时间复杂度。尽管循环嵌套通常不

⊖ 由于标量寄存器得到特别处置，这样说不完全真实。

是很深，但因为指数级的解决方案，所以即使是一个小的循环嵌套也会导致潜在的编译时间问题。这些问题当指数限制为1时就不存在了。

使用这样的策略，Ardent优化器的基本的出发点就变成选择一个最优的循环进行向量化，而不是向量化尽可能多的循环。这种转变导致几乎所有的代码生成算法的改变。确切地说，它产生了在5.10节描述的代码生成策略。

当时的一个顾虑是把向量化限制在一个循环可能会对以后把编译器移植到其他向量机时产生负面影响，但是实际情况证明不是这样。Ardent优化器后来被移植到一个基本上是向量存储体系结构的机器上，并且生成的代码仍然非常好。

最后一个向量化策略方面的关键决策涉及到向量化的总体方法。本书给出的算法从一个纯粹语义的观点来考虑向量化，确定何时一个向量操作和相应的标量操作在语义上是等价的。遗憾的是，这对于绝大多数实际的机器来说是不够的，因为机器必须能够执行这个向量操作才行。例如Titan I，它没有向量除法指令，因此虽然从理论的观点来看如果编译器能够向量化一个除法是很好的，但是这并不能导致任何的程序执行加速。[⊖]

面对固定的硬件这一困难的现实，Titan向量化编译器必须或者（1）避免向量化那些机器不支持的操作，或者（2）把这样的向量化操作转换成一系列可以为机器所接受的操作。这里采用的方法是第二种：从纯粹语义的观点来向量化循环，然后进行一遍机器相关的“反向量化”，把机器中没有的向量操作转换为有效的指令。这样做的理由是编译器最终总会被移植到其他目标体系结构上，因此先作一遍向量化（按机器的约束参数化，例如按分段大小和可用的并行性）然后再作一遍反向量化，这是一种比较容易进行移植的方法。

事实证明，这也是所有方法中最有效的一种，特别是在最终的代码方面。例如，即使Titan没有向量除法，但在某些情况下，它仍可以向量化一个向量除以一个标量的除法，其作 233
法是计算标量的倒数，然后做向量和这个结果的乘法（假设$x/a = x*1/a$）。[⊖]这种变换在预向量化阶段是很难识别的，因为在知道哪个循环被向量化之前，你无法知道除数是否是一个标量。这在预向量化完成后的阶段是很容易实现的。

5.13.3 向量化的性能

为了说明在PFC中使用和本章中描述过的向量化技术的性能，我们给出了不同编译器在Callahan-Dongarra-Levine基准测试程序包上的有效性研究结果。该测试程序包包含100个循环，用于考察向量化编译器的不同方面的能力。测试程序包（可以在www.mkp.com/ocma上找到源代码）分为四类：

（1）*依赖*，用于考察依赖测试的精确度。

（2）*向量化*，测试各种向量化变换（例如循环分布、循环交换、标量扩展以及交叉阈值等）是否存在。

（3）*idioms*，测试各个编译器在识别常见的归约、封装、查找和一些特殊依赖环的能力。

（4）*完整性*，测试处理复杂的语言结构，例如控制流、等价和内部函数等的能力。

PFC和一些商用编译器的实际性能如表5-1所示。PFC的实验是由PFC项目组的Ervan Darnel完成的，而所有其他数据均来自Callahan，Dongarra和Levine的论文[57]。在这些商用编

⊖ 当CPU试图执行向量除法时，可执行的向量操作立即转储核心，在这个意义下程序的执行得到加速。

⊖ 数值方面的纯粹主义者很容易发现，如果不对标量加一些限制，结果并非完全相同。

译器中，Ardent Titan编译器，Convex C Series编译器和IBM 3090 VF 编译器[243]都是直接基于PFC中的方法和算法的。每组都有三列：标“V”的列表示完全向量化的循环的个数，标“P”的列表示部分向量化的循环的个数。100和这两列的和之差就是没有被向量化的循环个数。

表5-1 测试性能（Callahan-Dongarra-Levine测试）

	总 计		依 赖		向 量 化		Idioms		完 全	
Vectorizing compiler	*V*	*P*	*V*	*P*	*V*	*P*	*V*	*P*	*V*	*P*
PFC	70	6	17	0	25	4	5	0	23	2
Alliant FX/8, Fortran V4.0	68	5	19	0	20	5	10	0	19	0
Amdahl VP-E, Fortran 77	62	11	16	1	21	8	11	1	14	1
Ardent Titan-1	62	6	18	0	19	5	9	0	16	1
CDC Cyber 205, VAST-2	62	5	16	0	20	5	7	0	19	0
CDC Cyber 990E/995E	25	11	8	0	6	8	3	1	8	2
Convex C Series, FC 5.0	69	5	17	0	25	4	11	0	16	1
Cray series, CF77 V3.0	69	3	20	0	18	3	9	0	22	0
CRAX X-MP, CFT V1. 15	50	1	16	0	12	1	10	0	12	0
Cray Series, CFT77 V3.0	50	1	17	0	8	1	7	0	18	0
CRAY-2, C FT2 V3.1 a	27	1	5	0	3	1	8	0	11	0
ETA-10, FTN 77 V1.0	62	7	18	0	18	7	7	0	19	0
Gould NP1, GCF 2.0	60	7	14	0	19	7	8	0	19	0
Hitachi S-810/820	67	4	14	0	24	4	14	0	15	0
IBM 3090/VF, VS Fortran	52	4	12	0	19	3	5	1	16	0
Intel iPSC/2-VX, VAST-2	56	8	15	0	17	8	6	0	18	0
NEC SX/2, F77/SX	66	5	17	0	21	5	12	0	16	0
SCS-40, CFT × 1 3g	24	1	7	0	6	1	5	0	6	0
Stellar GS 1000, F77	48	11	14	0	20	9	4	1	10	1
Unisys ISP, UFTN 4.1.2	67	13	21	3	19	8	10	2	17	0

如表5-1所示，PFC在向量化和完整性这两个组都做得很好，这是因为它所采用的程序变换的系统方法（例如数组重命名和交叉阈值）、对条件语句的处理（见第7章）以及过程间分析（见第11章）。在依赖分析方面，PFC做得一般，但是实验时，它尚未使用第3章所描述的那一套完整的测试方法。比较起来，PFC在idioms方面做得比较差。这是因为PFC项目中的一个设计决策，决定不在那些处理特殊情况的模式匹配上投入过多资源。

直接基于PFC的编译器——来自Ardent，Convex和IBM——也做得比较好，虽然IBM向量化编译器在依赖测试方面有所欠缺，因为它是在低层中间表示上而不是在源程序级作依赖测试。Ardent编译器做得相当好，尽管只用了一年半的时间进行开发。

总而言之，这些结果证明本书描述的变换和测试广泛涉及向量化编译器所面临的各种挑战。然而，Callahan-Dongarra-Levine测试还显示另外一点。那些做的最好的编译器都是在细节上一丝不苟的。idioms是说明这一点的一个例子，但是我们给出另外一个例子——测试用例171（它是一个PFC不能向量化的例子，但是绝大部分编译器都能正确地处理）：

234
|
235

```
subroutine s171 (a,b,n)
  integer n
  real a(*),b(*)
  do 1030 i = 1, n
```

```
      a(i*n) = a(i*n) + b(i)
1030 continue
```

PFC在这个测试失败是因为它在处理下标中的符号系数方面作得很差。因为n可能等于0，所以它没有向量化这个循环。但它忽视了n同时还是循环上界这一事实，这样若n等于0，循环即退化了。这类例子说明，即使有了正确的框架，要取得成功还依赖于对实际中出现的特殊情况的系统处理。本章的一个目标就是提供一个大大简化这一处理的框架。

5.14 历史评述与参考文献

在基于依赖的程序变换方面最早的文献包括Lamport[195，196]和Kuck[190]的论文。Lamport开发了一种用于向量化的循环交换，以及用于并行化的波前方法，循环倾斜的一种早期形式。Lamport也讨论标量扩展的重要性，虽然他没有给出实现方法。Wolfe[278，280，283]进一步地研究了循环交换，本章讨论的扩展是由Allen和Kennedy[16，19，20]提出的。用方向矩阵驱动交换是在本书中首先提出的，但是它基本上是 Wolfe提出的方向向量[280]以及Lamport所使用的并被Wolf和Lam[277]发展了的距离矩阵方法的一种变形。

Wolfe的硕士论文[278]讨论一种标量扩展的机制。他的方法机械地扩展所有的标量，不管是否有利，而依靠后面一遍收缩那些没有好处的扩展。Pieper[224]的博士论文考察多种不同的标量扩展方法，并比较它们在实际程序中的效果。

用于消除依赖的各种重命名技术在很多地方都作了一般的讨论[20]，例如索引分裂。特别地，Padua的论文给出一个标量重命名的精确算法[221]。数组重命名和结点分裂是从Allen和Kennedy的一篇文章[21]中来的，不过在本书对它进行了修改。目前这种形式的循环倾斜是由Wolfe[28]作为Lamport的波前变换的一种实用的形式而引入的。对程序变换的有用的评述包括Padua和Wolfe[222]、Kuck等[190]、Wolfe的博士论文[283]以及Wolfe后来写的教科书等。

这里给出的向量变换的全局选择策略是Allen在Ardent Titan编译器中工作的一部分。 236

习题

5.1 把循环移动代码生成的启发式方法应用于下例:

```
DO I = 1, N
   DO J = 1, N
      DO K= 1, N
         A(I,J+1) = A(I,J) + B(K,J)
      ENDDO
   ENDDO
ENDDO
```

5.2 修改5.4节的数组重命名算法用于处理循环体内有控制流的代码。提示：采用和标量扩展类似的方法保证在每个控制流分支上都有一个定值。

5.3 证明等式（5.1）中的循环倾斜替换具有给外层循环携带的依赖的内层循环距离增加k的作用。它对内层循环携带的依赖具有什么作用?

5.4 证明存在关键反依赖时，图5-14的节点分裂算法可以在应用标量扩展以后打破依赖环。

5.5 把5.10节的代码生成算法应用于下边循环嵌套，会有哪些变换被用到?

```
DO I = 1, N
   DO J = 1, N
      DO K= 1, N
```

```
            T = A(K,J) + A(K+1,J)
            A(K+1,J) = T + B(K)
            A(K+1,J) = A(K+1,J) + C(K)
         ENDDO
      ENDDO
   ENDDO
```

5.6 下边的循环应该生成怎样的向量代码?

```
DO I = 1, 100
   A(I) = B(K) + C(I)
   B(I+1) = A(I) + D(I)
ENDDO
```

237

第6章

开发粗粒度并行性

6.1 引言

第5章讨论了用于提高内层循环并行性的程序变换。那些变换提高了向量和超标量体系结构的性能，其中并行性是细粒度的，且主要存在于最内层循环中。本章我们转向为用共享全局存储的异步多处理器发掘并行性的问题。这类处理器以对称多处理器（SMP）——如Sun，Silicon Graphics及Compaq等公司的工作站——以及分布式共享存储（DSM）系统（典型系统有SGI Origin 2000和HP/Convex SPP-2000）为代表。

多处理器级的粗粒度并行性要求不同的侧重点：在这种体系结构上，利用并行性的方法是在每个处理器上创建一个线程，并行执行一段时间，其间偶有同步，并在最后通过一个栅障实现同步。在这些系统上获得高性能的关键在于找到并封装有足够粒度的并行性，以弥补并行初始化和同步所带来的开销。这样侧重点就在于找到循环体内有大量计算的并行循环。这通常意味着外层循环的并行化而不是内层循环的并行化，并且常常意味着含有子例程调用的循环的并行化，这是在第11章要讨论的题目。

在为粗粒度体系结构生成并行代码时，许多复杂微妙的折中需要仔细考虑。其中最困难的问题之一是在保持所有处理器上的负载平衡的同时最小化通信和同步开销。在一个极端，当整个程序运行在一个处理器上时，我们得到绝对的最小开销；因为没有并行性，也就没有处理器间的通信和同步开销。自然，负载平衡和相应的并行加速比也很差。在另一个极端，当程序被分解成最小可能的并行单元，且所有并行单元均匀地分布在空闲处理器上时，可以得到最好的负载平衡。因为并行程序单元很小时，不会出现一个处理器长时间执行某个很大的程序单元而其他处理器无事可做的情形。然而，同步和通信开销在此情况下达到最大化，而这一开销几乎总是超过完美的负载平衡所带来的好处。在这两个极端之间有一个最有效的并行分解——并行化编译器面临着找到这个最佳点的挑战。

第1章介绍了PARALLEL DO语句（见1.5.1节），它表示其迭代按任何顺序都能正确运行的循环。文献中也把这种类型的循环称为DOALL循环。粗粒度并行循环的另一种形式是DOACROSS循环，它利用迭代间的同步以流水方式执行并行循环迭代（基本上是把多处理器作为高层向量处理器使用）。本章侧重于PARALLEL DO循环的生成，因为任何可以以DOACROSS循环的形式有效利用并行性的循环，都可以被分成一系列PARALLEL DO循环，虽然这可能失去流水线并行性。6.6.2节更详细地讨论流水并行性的引入。

本章的其余部分描述可用于程序建立新并行机会的变换，从而将这个简单的代码生成算法改进成为一个复杂的，积极的代码生成策略。这其中的很多变换曾经在讨论向量化时讲述过；本章给出在粗粒度并行性下必须作的一些（通常是较小的）修改。

6.2 单循环的处理方法

在试图改进从单个循环中发掘出来的并行性时，可以尝试两个通用的策略。如果循环是

一个串行循环，（也就是说，它携带依赖），找到一些方法将其并行化是改进并行性的明显方法。任何消除循环携带依赖的变换（例如循环分布）都可以用于这个目的。如果一个循环是并行的，增大暴露出的并行性粒度通常是有效的。这一节讨论为达到这些目的不同变换。

6.2.1 私有化

第5.3节介绍了标量扩展和标量私有化变换。标量扩展在向量化中是有效的，因为它消除
240 很多和原来的标量相关联的依赖——既有循环携带也有循环无关的依赖。因为删除循环携带依赖能把串行循环转变成为并行循环，标量扩展和标量私有化都是增加粗粒度并行性的重要变换。

标量扩展的机制和确定它的安全性的条件在第5.3节已经讨论过了。不管是应用于向量化或者是并行化，这些都是相同的。在这一节我们将集中于私有化，确定一个循环内赋值的变量仅仅在其被赋值的迭代内引用。这样的变量可以跨越各个迭代复制，消除可能表面上阻碍并行化的循环携带依赖。

作为一个例子，考虑5.3节中的一个简单代码段，在其中交换两个数组的值：

```
      DO I = 1,N
S1      T = A(I)
S2      A(I) = B(I)
S3      B(I) = T
      ENDDO
```

这个循环的依赖模式在图6-1中给出。因为循环携带的反依赖和输出依赖，这个循环在此形式下不能被并行化。

幸运的是，所有的循环携带依赖都是由于标量变量T的赋值和引用导致的，像我们在5.3节指出的那样，如果每个迭代有它自己的变量T的拷贝，所有这些循环携带依赖都可以被消除掉，像在以下变换和并行化后的代码中那样：

```
      PARALLEL DO I = 1,N
         PRIVATE t
S1       t = A(I)
S2       A(I) = B(I)
S3       B(I) = t
      END PARALLEL DO
```

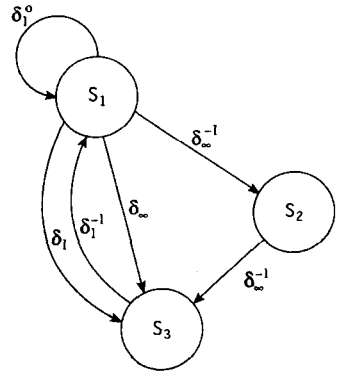


图6-1 数组交换的依赖图

在上边的例子中，所进行的变换使得每个迭代有一个私有的变量T的拷贝是正确的，因为循环体内所有T的引用都是引用一个相同迭代内的赋值。换句话说，循环体内没有向上暴露的引用。这个条件通过下边的定义来形式化。

定义6.1 一个在循环内定义的标量x对这个循环来说是*可私有化的*，当且仅当从循环体的开始到循环体内x的任一引用的每条路径在到达该引用前必须经过x的一个定值点。

能私有性可以通过4.4节描述的标准数据流分析来确定。首先我们求解循环体上的一组数据流方程，来确定在每一基本块x的开始处是向上暴露的变量集合$up(x)$：

$$up(x) = use(x) \cup (\neg def(x) \cap \bigcup_{y \in succs(x)} up(y)) \quad (6\text{-}1)$$

其中$use(x)$是在块x中有向上暴露引用的变量集合。$def(x)$是在块x中有定义的变量集合。如果b_0是循环体的入口基本块，那么在循环内定义的一个变量是可私有化的，当且仅当它不在集合$up(b_0)$里。在循环体B中所有可私有化变量的集合由下式给出:

$$private(B) = \neg up(b_0) \cap (\bigcup_{y \in B} def(y)) \quad (6\text{-}2)$$

确定某个特定的标量是否可私有化的另一方法是通过4.4.4节的SSA图。

定理6.1 一个在循环中定义的变量x可以被私有化，当且仅当该变量的SSA图在循环入口处没有ϕ结点。

这一命题是对的，因为标准SSA构造会对没有在程序中显式赋值的变量插入哑初始化。因此，我们可以假定每个标量都是在循环体外赋值的。如果一个给定的变量x有一个向上暴露的引用，那么循环体外的定义可以到达这个引用。另外，任何循环内的定义也可以通过一条跨越迭代的路径到达该引用。这样由两个不同的定义所产生的值必然会在该循环的SSA构造过程中通过插入ϕ结点解决。显然，如果循环体中没有向上暴露的引用，就不需要ϕ结点。因此在循环入口处有ϕ结点，当且仅当循环体中有向上暴露的引用。

241 ~ 242

如果某个特定循环携带的所有依赖均涉及到可私有化的变量，那么通过私有化所有这些变量，该循环可以被并行化。

比起标量扩展，明显地我们倾向于用私有化的方式消除在粗粒度循环内的循环携带依赖，但标量扩展也可以在一些不能做私有化的情况下有所帮助。下面就是一个这样的例子:

```
DO I = 1, N
   T = A(I) + B(I)
   A(I-1) = T
ENDDO
```

因为存在由T和A导致的循环携带的依赖，(对T的)私有化不能产生任何并行性。然而，标量扩展T允许循环被分布，产生了两个并行循环，其间有一个栅障:

```
PARALLEL DO I = 1, N
   T$(I) = A(I) + B(I)
ENDDO
PARALLEL DO I = 1, N
   A(I-1) = T$(I)
ENDDO
```

标量扩展在这种情况下的应用可能需要大量存储空间(特别是如果扩展应用于多个循环时)，故其使用须视情况而定。

好的私有化对于并行化大多数应用来说是很关键的。很多应用如果没有对数组和标量的私有化就不能被并行化。考虑一维数组的私有化问题。因为我们不为下标变量构造SSA图，定理6.1给出的私有化条件的简单测试不能用于数组。遗憾的是，循环携带依赖的存在不足以说明一个数组是不能被私有化的，如下例所示:

```
      DO I = 1, 100
S0      T(1) = X
L1      DO J = 2, N
```

```
S1          T(J) = T(J-1) + B(I,J)
S2          A(I,J) = T(J)
         ENDDO
243   ENDDO
```

虽然这可能不很明显，但实际上I-循环的循环体内对于数组T的每个元素的引用之前都有对该元素的赋值。然而，一个标准的依赖分析器将会构造一个从语句S_1到它自身的I-循环携带的依赖，因为T(J-1)和T(J)在I-循环的不同迭代中引用了相同的数组元素。这样，为了私有化数组T，我们必须确定数组T的任一元素都没有向上暴露的引用。

如果我们是处理单个的循环，那么可以使用一种类似标量数据流分析的方法。假设我们把数组T的下标变量的不同实例作为单个的标量引用看待。这样，T(1)就会被看作和T(J)以及T(J-1)不同的变量。如果我们来解决循环体的向上暴露引用问题，结果将会是那些我们必须假定在循环体内定义它之前被引用的变量。如果该方法应用于上例的内层循环

```
L1   DO J = 2, N
S1      T(J) = T(J-1) + B(I,J)
S2      A(I,J) = T(J)
     ENDDO
```

它确定引用T(J-1)是向上暴露的。注意B(I,J)在循环内也是向上暴露的，但是我们现在将注意力集中在T上，因为它是惟一的既在循环内引用又在其中定义的变量。

为了把这个分析扩展到循环的嵌套中，我们需要一些方法来确定那些在内层循环中向上暴露的引用集合。为了计算这个集合，我们对那些由循环J迭代导致的向上暴露的变量，构造一个表示。在迭代J向上暴露的变量集合就是循环体内从迭代J开始向上暴露的变量集合，减去在某些前面的迭代中定义的变量的集合。

对于前边的内层循环，我们已经看到T(J-1)在迭代J上是向上暴露的。显然，在某些前边迭代中定义的变量集合由所有前边迭代的定义集合的并集给出。在这个例子中很清楚它是T(2:J)。J的所有变量的这些差集的并给出T中向上暴露的元素的完全集合。

$$up(L_1) = \bigcup_{J=2}^{N}\{T(J-1)\} - T(2:J)$$

容易看出，除了J等于2时，表达式T(J-1)-T(2:J)都为空。这样内层循环向上暴露单元的完全集合是{T(1)}。

一旦子循环被处理了，确定哪个变量可私有化的算法也使用单循环的处理方法，内层循
244 环被作为一条语句处理，该语句先引用其向上暴露的变量表中的所有变量，然后定义那些必须在循环的某些迭代中定义的变量。在我们的例子中，L_1定义T(2:N)。

这个分析用于本例的最终结果是T(1)恰好在L_1之前的语句中定义，因此在外层循环中没有向上暴露的元素。这样整个数组在I-循环中可以被私有化，从而外层循环可以被并行化:

```
      PARALLEL DO I = 1, 100
         PRIVATE t
S0         t(1) = X
L1         DO J = 2, N
S1            t(J) = t(J-1) + B(I,J)
S2            A(I,J) = t(J)
         ENDDO
      ENDDO
```

注意这个代码假设在循环后没有数组T的引用（这里t和T被认为是不同的）。如果存在这样的引用，私有化变换需要插入一个条件赋值语句，在循环的出口处把t的值拷贝到T中。在这个例子中，会得到如下代码:

```
      PARALLEL DO I = 1, 100
         PRIVATE t(N)
S0       t(1) = X
L1       DO J = 2, N
S1          t(J) = t(J-1) + B(I,J)
S2          A(I,J) = t(J)
         ENDDO
            IF (I==100) T(1:N) = t(1:N)
      ENDDO
```

6.2.2 循环分布

细粒度并行化算法*codegen*（图2-2）隐式依赖于循环分布把循环携带依赖转变成为循环无关依赖。例如，在

```
      DO I = 1,100
         DO J = 1, 100
S1          A(I,J) = B(I,J) + C(I,J)
S2          D(I,J) = A(I,J-1)*2.0
         ENDDO
      ENDDO
```

245

中，当J-循环跨越从语句S_1到语句S_2的循环携带依赖分布时，循环嵌套变成

```
      DO I = 1,100
         DO J = 1, 100
S1          A(I,J) = B(I,J) + C(I,J)
         ENDDO
         DO J = 1, 100
S2          D(I,J) = A(I,J-1)*2.0
         ENDDO
      ENDDO
```

在原始代码中，由数组A导致的从S_1到S_2的依赖跨越J-循环的不同迭代，因此是被循环携带的依赖。通过循环分布，这个依赖不再被任何循环携带。

循环分布可用于把一个串行循环转变为多个并行循环。然而，因为它把循环携带的依赖转变成为分布的循环间的循环无关依赖，这个变换在依赖的两端点之间（第一个并行循环的尾端）隐式地插入一个同步栅障。这减小了并行性粒度且增加额外的通信和同步开销。因此，在采用循环分布之前，应先尝试其他能够消除依赖但是不会减小并行性粒度的方法。在下边几小节我们将讨论一些这样的变换。

6.2.3 对齐

循环分布通过先执行所有依赖中源点的语句然后执行汇点的语句来变换循环携带依赖。在分布之前，一组值在循环的一个迭代中计算而在后边的迭代中被引用。分布以后，这些值在一个循环中计算而在不同的循环中引用。另一个能够达到相同目的的方法是重新对齐循环使得值的计算和引用在同一迭代中。下边从6.2.2节分布例子改写的例子，说明这个思想:

```
DO I = 2, N
   A(I) = B(I) + C(I)
   D(I) = A(I-1)*2.0
ENDDO
```

这个循环不能并行执行，因为对迭代I计算的A的值在迭代I+1被引用。这两条语句可以通过加上一个额外的迭代并调整其中一条语句中数组引用的索引来对齐，使得值的计算和引用
246 在同一迭代中进行，从而得到

```
DO i =1, N+1
   IF (i .GT. 1) A(i) = B(i) + C(i)
   IF (i .LE. N) D(i+1) = A(i)*2.0
ENDDO
```

这样得到的循环不携带任何依赖，所以它能够并行执行。这个变换称为循环对齐。循环对齐的思想如图6-2所示。

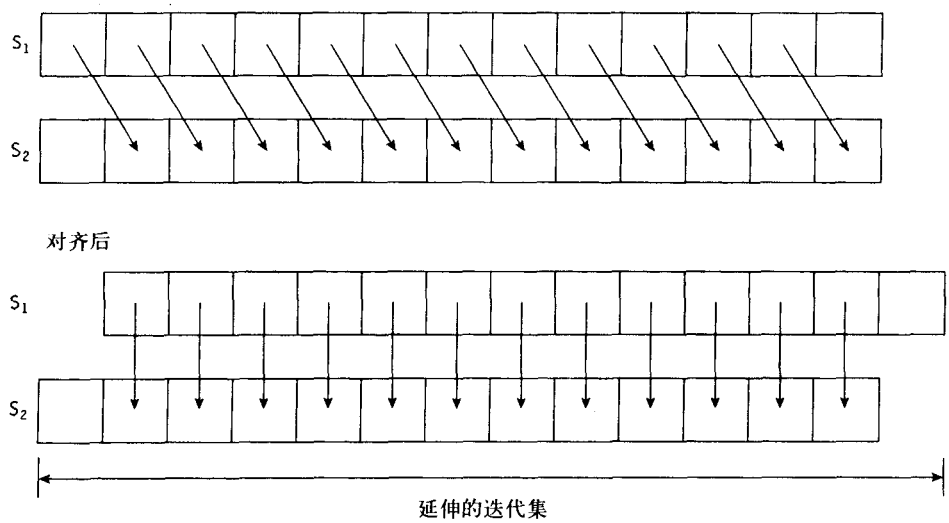


图6-2 循环对齐的图解

更一般的情况，循环对齐这样实现：通过增加循环迭代次数以及在与那些迭代略有不同的子集上执行各个语句，使循环携带依赖变成循环无关的依赖。循环对齐确实会导致一些的开销——一个额外的循环迭代和额外的条件判断。这些开销可以通过执行第一条语句的最后一个迭代和第二条语句的第一个迭代来减少，也就是

```
DO i = 2, N
    j = i - 1; IF (i .EQ. 2) j = N
    A(j) = B(j) + C(j)
    D(i) = A(i-1)*2.0
247 ENDDO
```

对于除了第一个迭代以外的所有其他迭代，j比i小1，因此赋值是对A(i-1)的。在第一个迭代，j=N，因此对A的最后一个元素的赋值可以正确执行。结果是，循环迭代的总数被恢复到它的最初计数，但是仍有对j条件赋值的开销。

另一个办法是，可以通过剥离每条语句的第一次和最后一次执行来消除条件语句，得到

```
D(2) = A(1)*2.0
DO I = 2, N-1
   A(I) = B(I) + C(I)
   D(I+1) = A(I)*2.0
ENDDO
A(N) = B(N) + C(N)
```

这个形式允许有效的并行化，代价是增加两条语句的开销，一条在循环前，一条在循环后，它们不能并行执行。

是否可能通过对齐来消除一个循环内的所有携带依赖？显然，如果携带的依赖在一个依赖环中，回答是否定的，如下边例子所示:

```
DO I = 1, N
   A(I) = B(I) + C
   B(I+1) = A(I) + D
ENDDO
```

在这个例子中，对B的引用产生一个循环携带依赖。在这种情况下为了成功应用循环对齐，我们需要交换循环体内两条语句的顺序。然而，由A导致的循环无关依赖阻碍对齐前的语句交换，所以我们希望能够在一步内完成循环对齐和语句交换来消除循环携带依赖:

```
DO I = 1, N +1
   IF (I .NE. 1) B(I) = A(I-1) + D
   IF (I .NE. N+1) A(I) = B(I) + C
ENDDO
```

虽然B现在被对齐，对于A的引用却没有对齐，产生一个新的循环携带依赖。考察这个例子，有理由相信循环对齐不能消除一个环内的所有携带依赖。为了更形式化地证明这一点，让我们假设有一个依赖环，并且希望变换程序把所有携带依赖转变成循环无关依赖。为了保证变换的正确性，必须在变换后的代码中保持原有的每个依赖。这意味着依赖环中每条语句 248
必然有一个从它到环内其他语句的依赖，并且，根据我们的假设，那个依赖必然是循环无关的。这对于重排序后代码中的最终语句显然是不成立的，因为任何从它开始的依赖必然是后向的，因此不能是循环无关的。

直观上，依赖环显然会阻碍对齐。每个依赖环有一个固定的总阈值，也就是说，环内的每条语句实例都和它自身在循环内的若干迭代前的实例相关连。减少循环体内前向依赖的阈值必然需要增加反向携带依赖的阈值。这样循环内所有依赖的阈值不能同时减少到0。

但是不包含在依赖环内的循环携带依赖会怎样呢？它们是否总能够通过对齐转换而不引入新的携带依赖？回答仍然是否定的，因为存在着对齐冲突的可能性——两个或者更多的依赖不能被同时对齐。考虑下边的例子:

```
DO I = 1, N
   A(I+1) = B(I) + C
   X(I) = A(I+1) + A(I)
ENDDO
```

这个循环包含两个和数组A关连的依赖、一个循环无关依赖和一个循环携带依赖。如果用对齐来消除循环携带依赖，将得到下边的代码:

```
DO I = 0, N
   IF (I .NE. 0) A(I+1) = B(I) + C
```

```
   IF (I .NE. N) X(I+1) = A(I+2) + A(I+1)
ENDDO
```

原来的循环携带依赖已经被消除，但是消除它的过程把最初的循环无关依赖转变成了循环携带依赖。循环仍然不能正确地并行执行。幸运的是，另一种变换——*代码复制*，在很多情况下能够消除对齐冲突。

6.2.4 代码复制

由于从相同源点出发进入到相同汇点的两个或者多个依赖（或者依赖链）有着不同阈值，导致对齐冲突。若源点或汇点被调整到对齐某一个依赖，则其他的依赖将以该依赖的阈值偏
249 移，因此不能对齐。

如果导致对齐冲突的依赖可以被改变为具有不同的源点或者不同的汇点，那么这些依赖可以被分别对齐而不引发冲突。一个分裂依赖源点的方法是复制源点进行的计算。例如，在前一节的对齐冲突例子中，第二条语句引用的A(I)的值在第一个迭代之后的每个迭代中都来自第一条语句（因此是循环携带依赖）。因为第一条语句的输入操作数在循环内部不改变，因此对需要的迭代重新计算A(I)的值是可能的。下面修改后的对齐冲突例子通过复制消除循环携带依赖。

```
DO I = 1, N
   A(I+1) = B(I) + C
   !复制的语句
   IF (I .EQ. 1) THEN
      t = A(I)
   ELSE
      t = B(I-1) + C
   END IF
   X(I) = A(I+1) + t
ENDDO
```

第一个迭代必须被分离出来，因为A(1)的值是循环入口以前的旧值而不是来自循环内的其他语句。假设对变量t作私有化，则这个循环没有携带依赖，可以并行执行。由于这里变换的本质是加入一些冗余的重复计算，变换称为*代码复制*。

代码复制、循环对齐和语句重排序的组合是否足以消除无依赖环的循环内的所有循环携带依赖？下边的定理表明这个问题的回答是肯定的。

定理6.2 对齐、复制和语句重排序足以消除单个循环内的所有携带依赖，前提是循环不包含依赖环，且每个依赖的距离是独立于循环索引的一个常数。

我们通过构造一个产生想要的结果的算法来证明这个定理。根据一个循环没有任何依赖环这一事实，它的依赖图可以表示为一个有向无环对齐图$G=(V, E)$。其中顶点（V）表示语
250 句；边（E）表示依赖，并在边上标明依赖距离。

在一个对齐图中，任何语句在任一时刻的对齐用一个*偏移量*保存，它表示语句的当前对齐和原来在循环中的位置之间的迭代次数。下边的例子可以解释这个概念：

```
      DO I = 1, N
S1       A(I+2) = B(I) + C
S2       X(I+1) = A(I) + D
```

```
S3      Y(I) = A(I+1) + X(I)
    ENDDO
```

这个例子最初的对齐图如图6-3所示。依赖距离在d字段表示；偏移量（在变换前它们的值都是0）保存在o字段。在证明对齐的作用以前，给出下边的定义是有用的:

定义6.2 如果$o(v)$表示顶点v的偏移量，$d(e)$表示依赖边e的距离，一个对齐图$G=(V, E)$被称为*无携带的*，如果对于每个边$e=(v_1, v_2)$有

$$o(v_1)+d(e)=o(v_2)$$

图6-3中的对齐图不是无携带的。因为总的目标是产生一个并行循环，对齐和复制算法的目标是通过偏移量调整和顶点复制构造一个无携带的对齐图。给定这样一个图，为调整过的循环生成代码是很简单的。

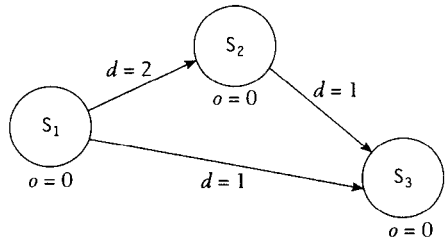


图6-3 对齐图

图6-4表示的对齐和复制算法的基本方法如下:

（1）创建一个工作表，表中最初包含任意未访问过的顶点。

（2）当工作表非空时，重复以下三个步骤:

a）从工作表中选取并删除一个结点。

b）把它和对齐图中与之相邻的结点对齐，当需要不同的对齐时复制代码。

c）把这些结点加入工作表。

（3）如果没有其他未访问过的顶点，回到步骤（1）。

```
procedure Align(V, E, d, o)
  // V和E分别是顶点和边的集合
  // d是依赖距离表
  // o是偏移量表，一个输出变量
  // W是算法的任务表
  while V≠Ø do begin
    从V中选出并且删除任意一个元素v0
    W:={v0}; o{v0}:=0;
    While W≠Ø do begin
    for each和v相关连的边e do
      if e=(w, v) then
        if w∈V then
        begin
          W:=W∪{w}; V:=V-{w};
          o(w):=o(v)-d(e);
        end
        else if o(w)≠o(v)-d(e) then begin // 冲突
          创建一个新的顶点w';
          把边e=(w, v) 用e'=(w', v) 替换;
```

图6-4 对齐和复制算法

```
                for each到w的边do
                    复制这条边，把w用w'替换;
                    W:=W∪{w'}; o(w'):=o(v)-d(e);
                end
            else //e=(v, w)
                if w∈V then
                begin
                    W:=W∪{w}; V:=V-{w};
                    o(w):=o(v)-d(e);
                end
                else if o(w)≠o(v)+d(e) then begin // 冲突
                    // 一直复制上流的顶点
                    创建一个复制的顶点v';
                    把边e=(v,w) 用e'=(v', w) 替换
                    for each到v的边do
                        复制这条边，把v用v'替换;
                    W:=W∪{v'}; o(v'):=o(w)-d(e);
                end
            end
        end
    end Align
```

图 6-4（续）

如果我们把该算法应用到图6-3的对齐图中，选择S_3作为根，可以得到图6-5中修改过的图。

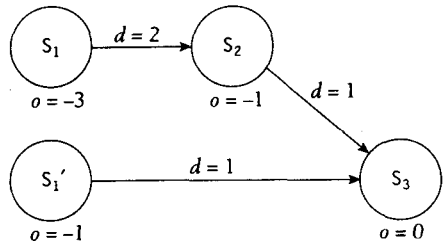


图6-5 修改过的对齐图

接下来描述如何从对齐图生成代码。注意偏移量表示在对齐空间中每个语句应该从哪里开始。下面是一个生成图6-3的循环的简单对齐:

```
        DO I = 1, N+3
S1      IF (I .GE. 4) A(I-1) = B(I-3) + C
S1'     IF (I .GE. 2 .AND. I .LE. N+1) THEN
          t = B(I-1) + C
        ELSE
          t = A(I+1)
        ENDIF
S2      IF (I .GE. 2 .AND. I .LE. N+1) X(I) = A(I-1) + D
S3      IF (I .LE. N) Y(I) = t + X(I)
        ENDDO
```

251 ~ 253

和前面的例子一样，我们可以把这段代码重新压缩如下:

```
        DO I = 1, N
          i1 = I - 3; IF (i1 .LE. 0) i1 = i1 + N
S1        A(i1+2) = B(i1) + C
          i2 = I -1; IF (i2 .LE. 0) i2 = i2 + N
S1'       IF (i2 .GE. 2) THEN
            t = A(i2) + D
          ELSE
```

```
          t = X(I)
        ENDIF
S2      X(i2+1) = A(i2) + D
S3      Y(I) = t + X(I)
    ENDDO
```

图6-6给出生成对齐代码的未压缩版本的算法。可以通过使用条件赋值计算循环索引而将此算法改写为产生压缩版本，我们把这个修改留作习题。

```
procedure GenAlign(V, E, o)
  // V和E分别是顶点和边的集合
  // o是偏移量表，一个输出变量

  h_i: = V中任意顶点的最大偏移量;
  l_o: = V中任意顶点的最小偏移量;
  设Ivar为初始循环增量
  设Lvar为初始循环下界
  设Uvar为初始循环上界

  生成循环语句 "DO Ivar = Lvar - h_i, Uvar + l_o";

  把V中的顶点按照拓扑排序，首先取出具有最小偏移量的顶点;

  for each v ∈ V按照排序后的顺序do begin
    if o(v) = l_o then那么在和v相连的语句前加上前缀
      "IF (Ivar. GE. Lvar - o(v))" ;
    else if o(v) = h_i then在和v相连的语句前加上前缀
      "IF (Ivar. LE.Uvar - o(v))" ;
    else在和v相连的语句前加上前缀
      "IF (Ivar. GE. Lvar - o(v).AND.Ivar.LE.Uvar - o(v))";
    if v是一个复制的节点then begin
      替换IF后边的语句S通过
        THEN
          t_v = RHS(S) 使用Ivar + o(v)替换Ivar
        ELSE
          t_v = LHS(S) 使用Ivar + o(v)替换Ivar
        ENDIF
        这里t_v是惟一的一个标量变量
      把从v发出的依赖边的汇点引用替换为t_v;
    end
  end

  生成循环结束语句 "ENDDO"
end GenAlign
```

图6-6 对齐和复制代码的生成

6.2.5 循环合并

至此，我们探讨了进行循环变换增加并行性而不使用循环分布的方法。如在6.2.2节讨论的那样，我们不愿意做分布因为那样会导致较小的并行粒度和额外的同步开销。但是，如果循环中一些语句可以并行执行而另一些不能并行执行呢？我们需要寻找重新构造现有循环的

方法，把可能并行执行的部分与那些必须串行执行的代码分开，而循环分布是这样做的明显途径。有效的循环分布需要回答下边两个问题:

（1）我们如何才能有效地找出循环中可以并行执行的部分，并把它们与串行代码分开?

（2）我们如何才能避免过多地损失并行性粒度?

当考虑细粒度并行性时，我们可以尽可能地使用循环分布，试图把每条语句放到由它自己构成的一个循环中去。经过循环分布，某些循环可以并行执行，而另外一些仍然不能，因为它们是依赖环中的一部分。当我们为异步并行性生成代码时，这个方法存在问题，因为生成的循环粒度太细了。另一方面，它将并行部分清晰地与串行执行部分分开。恢复粒度的方法是重新组合，或者说“合并”那些并行循环成为较大的并行循环。这种思想可以用下边的例子来说明:

```
      DO I = 1, N
S1       A(I) = B(I) + 1
S2       C(I) = A(I) + C(I-1)
S3       D(I) = A(I) + X
      ENDDO
```

254
~
255

因为从S_2到它自身有一个携带依赖，对齐、复制或倾斜不能够并行化该循环。循环分布把循环转化为三个分开的循环:

```
L1    DO I = 1, N
         A(I) = B(I) + 1
      ENDDO
L2    DO I = 1, N
         C(I) = A(I) + C(I-1)
      ENDDO
L3    DO I =1, N
         D(I) = A(I) + X
      ENDDO
```

容易看出循环L_1和L_3不携带依赖，因此可以并行化。然而，因为每个循环是一个单语句的循环，结果得到的并行性不大可能是有利的，除非循环上界足够大，值得通过循环分段把一些迭代聚集在一起。

为了增加并行粒度，我们试图把不同的并行区域合并到一个较大的循环。为了显示如何做到这一点，我们把完全分布得到的循环结果表示为一个有向图，图中的顶点表示循环，而边表示不同循环中语句间的依赖。图6-7说明前边例子的这种抽象表示。在这个图中，双环用于表示从循环分布得到的并行子循环，单环表示串行子循环。

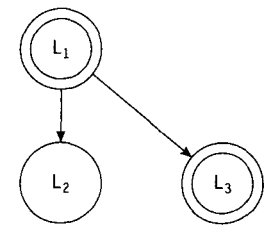


图6-7 循环间依赖图

从图6-7明显看出，循环L_1和L_3可以被合并以增大并行粒度，因为它们原来是在相同循环中，并且从L_2到L_3没有阻止合并的依赖。下边的并行代码由合并L_1和L_3得到:

```
L1    PARALLEL DO I = 1, N
         A(I) = B(I) + 1
L3       D(I) = A(I) + X
      ENDDO
L2    DO I = 1, N
         C(I) = A(I) + C(I-1)
      ENDDO
```

这个变换称为循环合并。像通常重排序变换一样，有些情况下循环合并是无效的。特别是，就像存在阻止交换的依赖一样，也会有阻止合并的依赖——指那些当两个循环合并时被反转的依赖。这样的依赖不可能是由循环分布产生的子循环造成的，因为合并这些子循环显然是有效的（它们本来是一个循环，因此把它们重新合并为一个循环不可能是非法的）。然而，这样的依赖可能存在于原来不同的循环间，如下边例子所示:

```
      DO I = 1, N
S1      A(I) = B(I) + C
      ENDDO
      DO I = 1, N
S2      D(I) = A(I+1) + E
      ENDDO
```

从S_1到S_2的依赖由于A是循环无关的——S_2中引用的全部A值是由S_1生成的，惟一的例外是最后一个迭代引用的值。如果这两个循环合并成

```
      DO I = 1, N
S1      A(I) = B(I) + C
S2      D(I) = A(I+1) + E
      ENDDO
```

循环无关依赖就变成了一个后向的循环携带反依赖。由于这个明显的事实，在合并后的循环中，S_2引用的A的值没有一个是由S_1生成的。

像6.2.2节讨论的那样，循环分布把一些循环携带依赖转变成循环无关依赖。因为由循环分布造成的循环可以被正确地合并看来是很合理的，因此在循环分布过程中从循环携带依赖转变成的循环无关依赖不会阻碍循环合并——通过循环合并，它们将被正确地重新转换为循环携带依赖。这些依赖和前边例子中的携带依赖的区别，在于它们是前向的携带依赖。如果例子中的两个循环如下: 256 ~ 257

```
      DO I = 1, N
S1      A(I) = B(I) + C
      ENDDO
      DO I = 1, N
S2      D(I) = A(I-1) + E
      ENDDO
```

它们可以被正确地合并并生成

```
      DO I = 1, N
S1      A(I) = B(I) + C
S2      D(I) = A(I-1) + E
      ENDDO
```

因为在两种情况下S_1都生成S_2中引用的所有A的值，除了第一个迭代。

前边循环合并例子中的问题，在于合并这两个循环把一个前向的真依赖转变为后向的循环携带的反依赖。换句话说，依赖的源点和汇点被颠倒了。这明显违反由基本依赖定理（定理2.2）给出的排序限制。这样，任何在循环合并以后变为循环携带依赖且反转其两端点的前向循环无关依赖，被称为是阻止合并的。

定义6.3 在两个不同循环中语句间的循环无关依赖（即从S_1到S_2）是阻止合并的，如果合并这两个循环导致依赖由合并后的循环反向携带（从S_2到S_1）。

显然，阻止合并和阻止并行化的依赖代表着安全性和有利性的考虑，是面向并行性通用代码生成算法中必须考虑的因素。另一个安全性方面的考虑是排序：合并循环而不违反任何依赖隐含的排序限制必须是可能的。下边对原来例子的修改说明这个条件：

```
      DO I = 1, N
S1      A(I) = B(I) + 1
S2      C(I) = A(I) + C(I-1)
S3      D(I) = A(I) + C(I)
      ENDDO
```

尽可能地分布该循环再次产生三个子循环。然而，循环间的依赖图已经改变，如图6-8
258 所示。

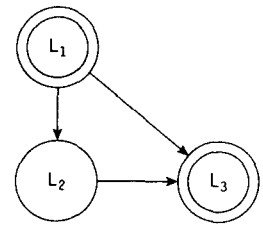


图6-8 由排序限制阻止的合并

合并循环L_1和L_2必然导致或者L_3在L_2前执行（如果合并后的循环首先执行），或者L_1在L_2后执行（如果合并后的循环第二个执行）。任一顺序都违反一个依赖。在循环分布和循环合并中，都必须遵守依赖所要求的拓扑排序。

总结起来，循环合并有两个安全限制：

（1）阻止合并的依赖限制：两个循环不能被合法地合并，如果它们之间存在一个阻止合并的依赖。

（2）排序限制：两个循环不能被合法地合并，如果它们之间存在一条循环无关依赖路径，这条路径包含一个未与它们合并的循环或语句。

虽然不作显式的证明，但这两个限制足以保证循环合并的安全性（不同于那些次要而明显的条件，如循环有相同的界，等等）。

尽管阻止合并和排序限制是决定循环合并安全性的主要约束条件，但它们不足以保证有利性。对于任何机器体系结构都有效的一个保证有利性的最小要求，是合并不能破坏并行性——至少在循环合并以后有和原来一样多的并行语句，否则合并就是没有好处的。这样的要求看上去可能对于合并是很容易满足的，但是实际情况并不是那样。

作为循环合并减少并行性的例子，考虑下边的一对循环：

```
L1    DO I = 1, N
        A(I) = B(I) + 1
      ENDDO
L2    DO I = 1, N
        C(I) = A(I) + C(I-1)
259   ENDDO
```

L_1不包含携带依赖，可以并行执行；L_2有一个依赖环，必须串行执行。这两个循环可以合法地被合并，但是因为L_2必须串行执行，合并循环强制L_1也串行执行，消除了并行的机会。任何时候只要一个串行循环被合并到一个并行循环，就会损失并行性，这提示我们以下的循环合并限制：

（1）分离限制：一个串行循环不应和一个并行循环合并，因为结果必然是串行执行的，减少并行性的总量。

然而，仅分离限制不足以保证在并行性情况下循环合并的有利性。可能合并两个并行循环得到一个串行循环，如下面例子所示：

```
      DO I =1, N
```

```
S1      A(I+1) = B(I) + C
        ENDDO
        DO I = 1, N
S2      D(I) = A(I) + E
        ENDDO
```

任一循环可以单独并行运行，但是如果它们被合并到一起，合并后的循环

```
        DO I =1, N
S1      A(I+1) = B(I) + C
S2      D(I) = A(I) + E
        ENDDO
```

携带一个依赖，阻止并行性。虽然这些依赖可以用对齐和复制来处理（因为不可能存在一个依赖环，否则循环合并就是无效的），为简单起见，我们将假定这样情况下的合并应当被禁止。这样我们引入一个新的定义来涵盖这一情形：

定义6.4 两个不同循环内的语句间的一条边被称为*阻止并行性的*，如果合并这两个循环以后，该依赖被合并后的循环携带。

这个定义是第二个有利性限制的基础：

（2）*阻止并行性的依赖限制*：两个并行循环不应被合并，如果它们之间存在一个阻止并行的依赖。 260

在花费大量时间讨论循环合并在什么情况下不能用来发掘并行性后，现在可以提出一个合并过程说明如何利用合并。

带类型的合并

我们将从一个严格分开并行和串行代码的代码生成模型开始；换句话说，它试图通过循环合并构造尽可能大的并行循环，但是不判定什么时候串行循环可以和其他串行循环或并行循环并行执行。在这个模型中，每个并行循环将会被一个栅障操作中止，所有必须在该并行循环之后或下一并行循环之前执行的串行代码将直接位于该栅障之后。在这个模型中，代码生成的中心任务可以概述如下：

给定一个图，其中边表示依赖而顶点表示循环，使用正确且有利的循环合并生成一个等价的程序，使得其中的并行循环个数最少。

我们可以把这个问题概括为*带类型的合并问题*：

定义6.5 一个*带类型的合并问题*是一个五元组$P=(G,T,m,B,t_0)$，其中

（1）$G=(V,E)$是一个图，

（2）T是一个类型的集合，

（3）$m:V\to T$是一个映射，对任意$v\in V$，$m(v)$是v的类型，

（4）$B\subseteq E$是一个坏边的集合，

（5）$t_0\in T$是对象类型。

带类型的合并问题P的解是一个图$G'=(V',E')$，其中V'是由V按以下限制合并判别为t_0类型的顶点导出：

（1）*坏边限制*：由坏边连接的两个顶点不能被合并；

（2）*排序限制*：由一条包含类型$t\neq t_0$的顶点的路径相连的两个顶点不能被合并。

带类型合并问题的*最优解*有最小数目的边。

容易看到，带类型的合并是并行合并问题的一个模型，顶点对应于循环，边对应于依赖，
261 类型对应于并行和串行，坏边对应于阻止合并或者阻止并行边，以及一个特殊的类型并行。问题的最优解拥有最小数量的并行循环，因此可能有最粗的粒度。

在图6-9到图6-12中，我们给出由Kennedy和McKinley[177]提出的算法*TypedFusion*，该算法能够在一个包含坏边和多种类型结点的图中产生给定类型t_0的结点的最优贪婪合并。这是快速算法，在最坏情况下需要$O(N+E)$步，其中N是顶点的数目，E是图G中边的数目。因为任何算法至少遍历图一遍，这明显是最小可能性。

```
procedure TypedFusion(G, T, type, B, t0)
  // G=(V, E) 是最初的带类型图
  // T是类型的集合
  // type(n) 是返回节点类型的函数
  // B是坏边的集合
  // t0是我们要为之找出最小合并的特定类型
  // 初始化
  lastnum:=0; lastfused:=0; count[*]:=0; fused:=0; node[*]:=0;
  for each边e=(m, n)∈E do count[n]:=count[n]+1;
  for each节点n∈V do begin
    maxBadPrev[n]:=0; num[n]:=0; next[n]:=0;
    if count[n]=0 then W:=W∪[n];
  end
  // 对类型t0的工作集合，访问节点和合并节点做迭代
  while W≠Ø do begin
    设n为W中的任意节点; W:=W-{n}; t=type(n);
L1: if t=t0 then begin // 一个该类型的节点
      // 计算需要合并的节点
S1:   if maxBadPrev[n]=0 then p:=fused;
      else p:=next[maxBadPrev[n]];
      if p≠0 then begin // 和在p的节点合并
        x:=node[p]; num[n]:=num[x];
        update_successors(n, t); // 在合并前访问后继
        合并x和n并把结果称为n;
          使得原来从n出发的边现在都从x出发
      end
      else begin // 把第一个节点放到一个新的组里
        creat_new_fused_node(n);
        update_successors(n, t);
      end
    end
    else begin // t≠t0
      create_new_node(n);
      update_succcessors(n, t);
    end
  end
end TypedFusion
```

262 图6-9 带类型的合并算法

```
procedure create_new_node(n)
    lastnum := lastnum + 1;
    num[n] := lastnum;
    node[num[n]] := n;
end create_new_node
```

图6-10 在归约图中创建一个新的结点

```
procedure create_new_fused_node(n)
    create_new_node(n);
    // 加节点n到fused末尾
    if lastfused = 0 then begin
        fused := lastnum;
        lastfused = fused;
    end
    else begin
        next[lastfused] := lastnum;
        lastfused = lastnum;
    end
end create_new_fused_node
```

图6-11 创建和初始化一个新的合并组

```
procedure update_successors(n, t)
l2: for each node m such that (n, m) ∈ E do begin
    count[m] := count[m] - 1;
    if count[m] = 0 then W := W ∪ {m};
    if t ≠ t0 then
        maxBadPrev[m] := MAX(maxBadPrev[m], maxBadPrev[n]);
    else // t = t0
        if type(m) ≠ t0 or (n, m) ∈ B then // bad edge
            maxBadPrev[m] := MAX(maxBadPrev[m], num[n]);
        else // 等类型，不阻止合并
            maxBadPrev[m] := MAX(maxBadPrev[m], maxBadPrev[n]);
    end
end update_successors
```

图6-12 访问后继、加入工作表以及更新*MaxBadPrev* 263

该算法的基本思想是，当每个结点第一次被访问时，算法在常数时间内确定它能被合并到相同类型的一个确定的结点中。如果没有这样的结点存在，对被访问结点分配一个分离的结点号。本质上，从图6-9到6-12的算法为单个选定的类型执行贪婪合并。

定义6.6 我们定义类型t的一个*坏路径*为一条从类型为t的结点开始的路径，它或者包含一条两个类型为t的结点间的坏边，或者包含一个类型不为t的结点。

算法把一条坏路径看作一个坏边。只有相同类型的结点才考虑合并。这个计算中使用的主要数据结构是*maxBadPrev*[*n*]，对图中的每个顶点*n*都作计算。对于原来图中给定的一个顶点*n*，*maxBadPrev*[*n*]是合并后图中的一个顶点号，该顶点是与*n*类型相同但是因为在该顶点（或者它表示的集合中的某些顶点）和*n*之间存在一条坏边而不能与*n*合并的顶点中编号最高的那个顶点。显然，*n*不能和*maxBadPrev*[*n*]合并。我们将阐明它也不能和任何结点号比 *maxBadPrev*[*n*]低的结点合并。因此它能合并的第一个结点是在简化图的编号中位于*maxBadPrev*[*n*]之后的第一个和*n*类型相同的结点。由于总是将*n*与这样的结点合并，*TypedFusion*实现一种贪婪的策略。

除了*maxBadPrev*[*n*]，该算法还使用下面的中间量：

- *num*[*n*]是*n*与t_0合并的结点集合中第一次访问的结点的号。换句话说，就是在输出图中包含*n*的合并后结点的编号。
- *node*[*i*]是一个把编号映射到结点的数组，例如*node*[*i*]是输出图中第*i*个集合的代表结点。注意*node*[*num*[*x*]]=*x*。
- *lastnum*是最近分配的结点号。
- *fused*是图中类型为t_0的第一个结点号。
- *lastfused*是类型为t_0的最近访问过的结点号。
- *next*[*i*]是合并表中下一个结点的编号，其中*i*是合并表中的一个结点号。
- *W*是将被访问的结点工作集。

264

该算法有点类似于拓扑排序。它从初始化所有这些数据结构，把没有前驱的结点放到工作表*W*中开始。然后在循环L_1中，重复地从工作表删除一个结点。如果该结点不是需要的类型t_0，它就为输出图建造一个新结点。另一方面，如果它是类型t_0的，算法按前边描述的那样找到可与它合并的最早结点，或者，如果没有那样的结点存在，它建造一个新的合并图并且把那个组加到合并的结点列表的最后。

随着每个结点被处理，所有的依赖后继被访问（在过程*update_successors*中），这样为这些后继更新*maxBadPrev*值。这个过程在检测到一个后继的所有前驱都被处理过以后，就把它加入到工作表*W*中。

*TypedFusion*达到一个最优的时间复杂度$O(E+V)$，上界为访问选定类型的每个新结点时，选出正确的可以在常数时间内合并的结点的时间。这出现在语句S_1中。如果我们能证明这样选择了要合并的正确结点，则利用与拓扑排序的复杂度分析类似的分析即可得到所需的时间上界。

正确性 为了证明正确性，我们必须证明问题定义中的限制是得到保证的。首先，算法明显只合并类型为特殊类型t_0的结点。为了证明算法遵守坏边限制，我们必须证明算法从未合并被坏边连接的两个结点。为了证明该算法满足排序限制，我们必须证明它从未合并由一条经过不同类型结点的路径连接的两个结点。这些都可由选择过程的正确性得到证明——如果*maxBadPrev*[*n*]计算正确，我们将不会与一个存在一条坏路径（包含一个不同类型的结点或者一条坏边）的结点合并。对图6-12中过程*update_successor*的仔细检查显示，在访问顶点*n*的每个后继*m*时，*maxBadPrev*[*m*]取其现有值和*maxBadPrev*[*n*]的最大值，或者如果$type(n)=t_0$而(n,m)是一条坏边或$type(m)\neq t_0$，则取合并的结点*n*的编号。这显然产生正确的值。

为了证明算法得到贪婪解，我们必须证明它把类型为t_0的顶点*n*和它最早能够合并的结点作了合并。这是因为从*maxBadPrev*[*n*]之前的任何类型为t_0的合并结点必然有一条坏路径到顶点*n*。显然，存在一条坏路径从*maxBadPrev*[*n*]到*n*。设*x*是某个*maxBadPrev*[*n*]之前的类型为t_0

的合并结点。那么必然有一条从x到$maxBadPrev[n]$的坏路径，否则那两个顶点就可以被合并。因为n和合并结点表中$maxBadPrev[n]$后边的第一个结点合并，那么该顶点必然是它能够合并的第一个结点。

最优性 为了证明该算法所得的解有最少的合并顶点，我们必须证明该贪婪解具有这个性质。虽然我们不去形式化地证明这个结果，但是将给出一个非形式化的论证。假设存在一个满足问题定义但是有着比*TypedFusion*计算出的贪婪集合C_G更少的合并组的集合C。那么在C中必然有一个第一组，它不同于C_G中相应编号的组。该组必然是贪婪组的一个子集，因为我们 265
把不能和先前的组合并的所有顶点都放到了贪婪组中。这样我们可以把贪婪组中的所有顶点移动到集合C的相应组中，这样使得C中某些更迟的组成为不同于C_G中的第一个组。如果持续这个过程，我们只会**缩小**C使得C的大小和C_G相同，因为我们是把C中较后组中的元素移动到较前的组里去。这样C就不能比C_G小。

复杂性 像我们前边说过的那样，这个算法的时间复杂性是$O(E+V)$的，这里E是原来图中边的数目而V是顶点的数目。这可以通过一个类似于对拓扑排序的分析得知。每个顶点最多可以从工作表W中取出一次，因此循环L_1里的例程和调用的其他例程中的常数时间工作需要$O(N)$时间。惟一需要非常数时间的地方在*update_successor*，其中每个给定结点的后继被访问。因为在整个计算中每条边最多被通过一次，这个过程执行任务的总量以调用数$O(N)$加图中边的数量$O(E)$为界。这样渐近的时间复杂度上界是$O(N+E)$。

为了了解这个划分算法是如何工作的，考虑图6-13中的例子。当算法应用于用双环标出的*并行*类型结点时，它会产生如图6-14所示的中间注释。每个并行结点用形式如

$$(maxBadPrev, p) \rightarrow num$$

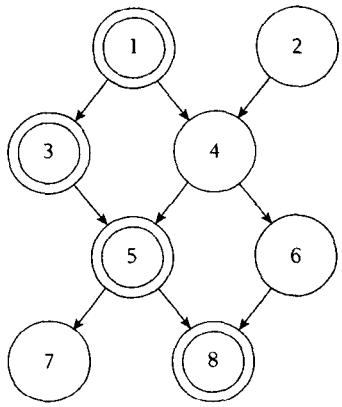


图6-13 一个子循环图的例子

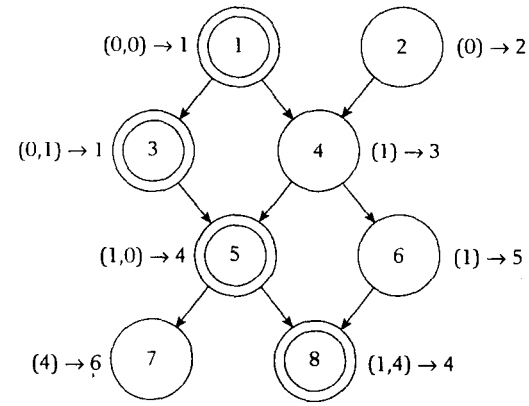


图6-14 对t_0=*parallel*（*并行*）的例子加上标注后的图

的一个元组标记，其中p是在算法中计算的而num是赋给结点的最后编号。原图中的顶点用带环的结点表示。*串行*类型的结点由一个单环标记，由算法对t_0:

$$(maxBadPrev) \rightarrow num$$

以外类型的结点计算。

合并以后的结果如图6-15所示，旁边是新的结点编号。

注意我们现在可以令t_0=*sequential*（*串行*）而再次使用算法*TypedFusion*来合并所有的串

行循环。我们不是一定需要合并它们，但是这对于了解需要多少个串行“段”是很有用的。最终的结果如图6-16所示。

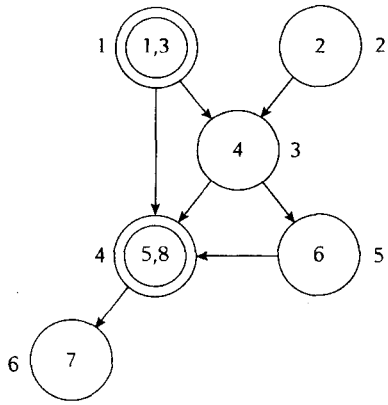


图6-15 合并并行循环后的例子

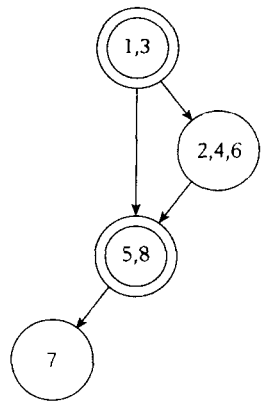


图6-16 合并顺序循环后的例子

需要的调度是:

(1) 包含原循环1和3的并行循环

(2) 包含原循环2，4和6的串行循环

(3) 包含原循环5和8的并行循环

(4) 包含原循环7的串行循环

无序和有序的带类型合并

一旦我们发展了类型合并的概念，就可以想像很多它的应用。本章后边将会给出一个重要的例子。现在我们考虑处理不相容循环头的问题。合并两个有着不相容循环头的循环可能不是很方便并且也不一定有什么好处，因为它们有着不同的迭代范围。如果我们以每个循环头作为一种不同的类型，则可以通过每次对一种类型应用*TypedFusion*算法来限制只对有相容循环头的循环作合并。

266 ~ 268

很自然会问，这是否总能产生总数最少的结点？为了理解这个问题，考虑图6-17的简单依赖图，它显示一个类型冲突的情况。虽然我们可以分开合并任一类型，但是我们不能两种类型都合并，因为一种类型的合并会产生对另一种类型的坏路径。这样我们必须从两种类型中选择一种进行合并。

图6-18的例子显示顺序如何能使合并的质量产生差异。如果我们在类型0或类型2之前合

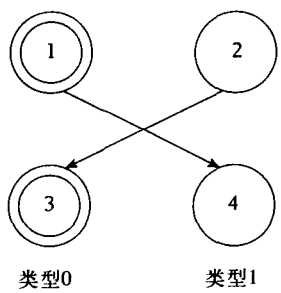


图6-17 一个类型冲突

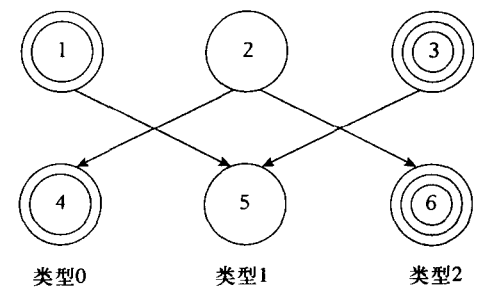


图6-18 顺序敏感的合并问题

并类型1，就会对类型0和类型2都产生坏路径，从而阻止其他的类型合并。然而，如果我们选择首先合并类型0，将仅对类型1造成坏路径，但是仍然允许类型2的两个结点的合并。第一种方法总共产生5个结点，而第二种方法只产生4个结点，明显结果较好。

是否有什么方法可以确定按照怎样的顺序合并类型最好？答案是肯定的，但是代价很高。Kennedy和McKinley已经证明这个算法是对类型数目的NP难题[176，177]。幸运的是，类型冲突是相当少见的，所以一个好的选择顺序的启发式方法可以是有效的。

对于有序的带类型合并问题，情况要简单得多。在这种情况下，类型可以根据某个标准排定优先级，而不管会对低优先级的类型造成多少坏路径，使具有更高优先级类型的结点数量达到最少总是更加重要。这种情况下，将一个调用*TypedFusion*的简单算法按照优先级降序 269
应用于每种类型将产生“最优”结果——即对于每种类型而言，相当于只要可能就在较高优先级给出类型合并。

一个在实践中经常出现的重要的有序的带类型合并问题有表示三种类型结构的结点：

（1）一个并行循环

（2）一个串行循环

（3）不在任何循环内的单个语句

很明显，合并并行循环总是最好的。因为合并两条语句没有任何好处，这种合并只是简单地把它们放到一起，而合并两个串行循环可以减少开销，所以总是优先进行循环的合并。这样对这三种类型就有一个清楚的优先顺序。

实际上，类型的全序是很难得到的。找到一个类型的偏序要典型得多。换句话说，在实践中出现的合并问题，通常仅有少量优先规则倾向于某种类型。我们处理这些问题时，可以通过使用拓扑排序来为图建立一个全优先序，然后按照降序来调用*TypedFusion*。

混杂合并

如果你不仅希望并行化循环，还要并行执行两个串行循环，或者并行执行一个串行循环和一个并行循环，那么就提出了另外一个有些不同的问题。某种程度上说，这里的问题就是找出所有可以和另外一个循环并行执行的循环，对它们或者进行合并，或者由于它们之间没有依赖关系。这个模型综合数据和任务并行性的特点。

算法的主要思想是在每个阶段识别出一个*混杂集*，它是一个可以并行执行的循环集合。这个集合可能包含若干并行循环，它们可以被合并成一个并行循环，以及若干可以和并行循环同时执行的串行循环。

定义6.7 *一个并行混杂集*是一个并行和串行循环的集合，它满足两个条件：（1）混杂集中任意两个并行循环之间没有阻止合并或者阻止并行性的边。（2）混杂集中任意并行循环或串行循环之间没有执行限制（即依赖边）。

换句话说，混杂集中仅有的循环间依赖边是两个串行循环间的边，或并行循环间既不是 270
阻止合并也不是阻止并行性的边。如果混杂集中的一个串行循环依赖于另一个，只要连接它们的边不是阻止合并的，那么它就可以和它的前驱合并。

总的来说，在一个混杂集中合并串行循环或许并不合乎需要，除非循环间有依赖。如果两个串行循环间没有执行限制条件，它们就可以（做为一个整体）相互并行执行。合并循环强制一对循环同时在同一处理器上执行，结果损失并行性。如果两个串行循环间存在依赖边，

那么合并这两个循环可能会因为内存层次结构（见第8章）或者简化的执行控制而节省一些执行时间。然而，节省的时间可能相对较小。

这个问题可以通过两次应用*TypedFusion*来解决。首先，所有结点被视为一个类型，而后边的边被定义为坏边：

（1）阻止合并的边

（2）阻止并行的边

（3）连接一个串行循环和一个并行循环的边

这一遍将生成最小数量的混杂集，从而产生最少的栅障。第二遍应用*TypedFusion*可以用于合并每个混杂集内的所有并行循环，虽然这不是必要的，因为根据定义一个混杂集内的所有并行循环都可以被合并。

虽然这个方法最小化了检查点栅障的数量，但是这个系统中对串行循环的处理可能仍是不令人满意的。当把串行循环分裂为两个混杂集和正确的大小划分迭代而取得更好的负载平衡时，可能有一个很大的串行循环将支配这个算法生成的混杂集。

6.3 紧嵌循环套

到目前为止，这一章给出了一些在单个循环中发掘并行性的技术以及一个通过改变循环顺序提高并行性的技术（循环交换）。本节将开始组合这些技术，着眼于开发一个在任意循环嵌套中发掘并行循环的通用算法。为达到此目标的第一步是处理紧嵌循环技术的开发。

6.3.1 为并行化的循环交换

循环交换是一种很有价值的变换，因为它可以完全改变大量语句的执行顺序。如1.5.2节
271 所指出的，并行化给循环交换（以及通用的变换）提出了一些不同的问题。特别是，平衡并行性和通信以及同步付出的代价要求并行区域有足够大的粒度以克服开始和同步并行计算的开销。换句话说，一个成功的并行分解必须使得并行粒度足够细以提供合理的负载平衡，同时尽量减少通信和同步的代价以避免影响结果并行性。

因为循环通常执行足够多时间来提供合理的负载平衡，把这个原则应用于循环嵌套通常意味着并行化外层循环。对于循环交换来说，这个原则意味着依赖无关的循环应当被移到尽可能的最外层位置，只要那样不会导致它们携带依赖。这和向量化正相反，那里目标是把依赖无关的循环移动到最内层位置。下面的例子说明这个区别：

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

外层（I）循环携带一个依赖环，而内层的（J）循环没有任何依赖。对于向量化，这个循环顺序是合理的，因为它允许内层循环被向量化。然而，对于粗粒度并行性来说，这个顺序是有问题的，因为只有内层循环能够被并行化，它可能在运行时要求N个栅障同步——串行执行的外层循环中每个迭代一个。

另一方面，如果并行循环被交换到最外层的位置（那里它仍然没有依赖），

```
PARALLEL DO J = 1, M
   DO I = 1, N
```

```
      A(I+1,J) = A(I,J) + B(I,J)
   ENDDO
END PARALLEL DO
```

那么在循环嵌套执行期间只需要一个同步。

当然，把一个循环移动到最外层并使它保持无依赖并不总是可能的。例如，如果这个例子稍微改动一下，

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J+1) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

272

(使得依赖方向向量成为(<,<)而不是(<,=)) 循环交换就不再有效了。在这个改动过的例子中，依赖不是交换敏感的，外层循环在交换后仍然携带依赖。没有其他变换的帮助，内层循环并行性以及结果所需要的同步操作次数，是我们能够得到的最好的结果。

```
DO I = 1, N
   PARALLEL DO J = 1, M
      A(I+1,J+1) = A(I,J) + B(I,J)
   END PARALLEL DO
ENDDO
```

我们在第5章看到循环倾斜和循环交换结合可以在这个例子上产生内层循环并行性，但是内层循环并行性对粗粒度并行是不能令人满意的，除非我们在内层循环中有足够多的循环迭代能使得每个处理器运行很多迭代的一个大块。

这些例子中自然引出的问题是，什么时候一个嵌套内的循环可以被移动到最外层位置且保证是并行的？下边的定理回答了这个问题。

定理6.3 在一个紧嵌循环套中，一个特定的循环可以在最外层被并行化，当且仅当该嵌套的方向矩阵的这一列仅包含“=”项。

证明 *充分性*：显然，其方向矩阵列中仅包含“=”项的循环可以在最外层被并行化；它能被移动到最外层位置是因为在它内层的循环将保持同样的相对顺序且方向矩阵中的所有向量仍然保持它们的最外层非“=”位置全为“<”项。因为只有“=”项，故此循环无论在哪一层都不携带依赖，包括最外层。这样按定理2.8它可以被并行化。

必要性：现在我们必须证明仅有这些是可以在最外层被并行化的循环。考虑任何其他情况。如果这一列包含一个“>”，那么该循环不能被移动到最外层位置，因为在这行的依赖将被反转（见定理5.2）。如果列中包含一个“<”，它不能在最外层被并行化，因为在那层携带一个依赖。由于没有其他可能的项，只能是这样的情况，当在该列中仅包含“=”项时，循环能够被移动到最外层位置并被并行化。

由此定理可以得出一个基于方向矩阵的紧嵌循环套的通用代码生成策略：

273

(1) 当方向矩阵包含只有“=”项的列时，任选一个有这样列的循环（最外层这样的循环是一个合乎逻辑的选择，但是任一个都可以），将其移动到最外层的位置并作并行化，然后从方向矩阵中删除它的列。继续该步骤直到所有这样的列被删除为止。

（2）一旦所有仅含“=”项的列被消除，选出它的方向向量中包含“<”项最多的循环，把它移动到剩下的最外层位置上，为它生成一个串行循环。删除这一列，同时删除表示这个循环携带的依赖的行（即那些在新的最外层位置包含“<”项的）。删除这些行可能产生新的全为非“=”的列，于是重复从步骤（1）开始的算法。

后面几小节将推广这个算法。然而，这个简单版本说明我们将给出的所有算法的操作模式：重复地选择一个循环（不管是并行的还是串行的）到最外层的位置。如果一个串行循环被选出，它应当是在剩余的循环中能发掘出最多并行性的循环。

下面三个循环组成的嵌套说明这个方法：

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
         A(I+1,J,K) = A(I,J,K) + X1
         B(I,J,K+1) = B(I,J,K) + X2
         C(I+1,J+1,K+1) = C(I,J,K) + X3
       ENDDO
   ENDDO
ENDDO
```

这个嵌套的方向矩阵是

$$\begin{bmatrix} < & = & = \\ = & = & < \\ < & < & < \end{bmatrix}$$

由于没有哪一列的所有项都是“=”，没有任何循环可以在最外层被并行化。在最外层位置，I循环和K循环都将携带两个依赖，因此可以选择任一循环在最外层串行执行。选择I循环剩下
274 一个方向矩阵[= <]，这将使J循环并行化而迫使K循环串行执行。

```
DO I = 1, N
   PARALLEL DO J = 1, M
      DO K = 1, L
         A(I+1,J,K) = A(I,J,K) + X1
         B(I,J,K+1) = B(I,J,K) + X2
         C(I+1,J+1,K+1) = C(I,J,K) + X3
      ENDDO
   END PARALLEL DO
ENDDO
```

这个简单的算法发掘相当数量的并行性，但很显然其他的变换（如循环分布）能够增加总的并行性。例如，如果该循环嵌套被分布成三个分离的循环，每个循环将拥有两维并行性。这样分布的有利性取决于目标机器的最小有效粒度。

6.3.2 循环选择

给定一个紧嵌循环套，并行性生成的主要挑战在于生成有足够粒度的最大并行性。应对这一挑战的关键在于选择适当的循环并行执行。6.3.1节通过展示如何利用循环交换提高并行性来对付这个挑战作了介绍。这一节将扩展那里的方法，并说明循环选择问题是非常困难的。

回忆6.3.1节提出的非形式化的并行代码生成策略。

（1）当有可以并行执行的循环时，把它们移动到最外层位置，将其并行化。

（2）选择一个串行循环，串行执行它，然后寻找可能暴露出来的新的并行性。

图6-19给出这个方法的一个更为形式化的说明。

```
procedure parallelizeNest(N, success)
  // N是需要并行化的循环嵌套
  // success如果至少有一个循环被并行化了则返回真
  // M是该循环嵌套的方向矩阵
  // L是N中需要处理的循环（紧嵌）的工作表

  为嵌套计算方向矩阵M
  设L是嵌套N中循环的列表;

  success:=false;
  while L≠Ø do begin
    while在M中存在一列，该列的所有方向都是“=”do begin
      success:=true;
      l:=对应于所有非“=”列中的最外层循环
      把l从L中删除;
      在最外层位置为l生成一个并行循环;
      从M中删除l对应的那一列;
    end;

    if L≠Ø并且¬success then begin
      //（启发式地）选择一个循环进行串行化
      // 它必须是可以移动到最外层的
      // 它可以导致并行性的发现
S:    select_loop_and_interchange(L);
      设l是L中的最外层循环; 把l从L中删除;
      为l生成一个串行循环;
      从M中删除l所对应的一列;
      从M中删除所有循环l携带的依赖所对应的行;
    end
  end
  if ¬success then恢复最初的循环; // 在Parallelize中再试
end parallelizeNest
```

图6-19 并行化循环嵌套的算法

对该算法而言，成功的关键步骤是步骤*S*：选择串行执行的循环的启发式方法。为了说明这一选择的重要性，考虑例子

```
DO I = 2, N+1
   DO J = 2, M+1
      DO K = 1, L
         A(I,J,K+1) = A(I,J-1,K) + A(I-1,J,K+2)
      ENDDO
   ENDDO
ENDDO
```

它的方向矩阵如下：

$$\begin{bmatrix} = & < & < \\ < & = & > \end{bmatrix}$$

275 ~ 276

没有仅包含“=”项的列，因此没有循环可以立即被并行执行。另外，内层循环有一个“>”方向，使得它不能被移动到最外层。因此，并行化这个嵌套的第一步就必须使用选择的启发式方法。在这个例子中，选择是显然的——最外层循环必须在某处被串行执行，因为它必须覆盖内层循环的“>”项。一旦这样做了，剩余的循环没有能够立即被并行化的，这就需再次使用启发式方法。串行执行剩下的循环中的任何一个都将允许另一个并行执行，因此最终将得到一个并行循环。

```
DO I = 2, N+1
   DO J = 2, M+1
      PARALLEL DO K = 1, L
         A(I,J,K+1) = A(I,J-1,K) + A(I-1,J,K+2)
      ENDDO
   ENDDO
ENDDO
```

是否可能找到一个选择的启发式方法来得到最优代码？寻求这样一个启发式规则似乎是不可能的，因为不难证明选择适当的循环是一个NP完全问题（见习题6.2）。目前，采取选择有最多“<”方向的循环这一简单方法，因为理论上这样一个循环能从方向矩阵中消除最多的行。但是把这个策略应用到下面的方向矩阵就不对了：

$$\begin{bmatrix} < & < & = & = \\ < & = & < & = \\ < & = & = & < \\ = & < & = & = \\ = & = & < & = \\ = & = & = & < \end{bmatrix}$$

尽管最外层循环携带最多的依赖，但是串行化执行它没有任何好处，因为三个内层循环也需要被串行执行。然而，如果三个内层循环被串行执行，最外层循环可以被移动到最内层的位置并被并行化。

一个避免这个例子所指出的问题的方法是在发掘并行性之前，优先选择必须串行执行的循环。这样，如果有一个循环能够被合法地移动到最外层位置，且存在一个依赖，循环对于它只有“<”方向，那么就串行化该循环。如果有多个这样的循环，它们全部需要在此过程中某处被串行化。

为了证明循环选择是一个NP完全问题，可将循环视为位向量，对于最外层位置将由循环携带的依赖，在向量的相应位置用“1”表示。前面的方向矩阵用下述矩阵说明：

277

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

然后循环选择问题就等同于在循环中找到一个最小基，以“逻辑或”作为合并操作。而这个问题就和最小集合覆盖问题一样，是一个NP完全问题。这是循环选择最好采用启发式方法的原因。

我们用一个简单的例子来说明启发式循环选择的一些原则，作为本节的结束。考虑下面类似模板计算的代码:

```
DO I = 2, N
   DO J = 2, M
      DO K = 2, L
         A(I,J,K) = A(I,J-1,K) + A(I-1,J,K-1) + A(I,J+1,K+1) + A(I-1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

这段代码有四个依赖，右边的每个引用有一个。注意第三个引用引起一个反依赖。方向矩阵如下，右边引用的依赖向量按照从上到下的顺序排列:

$$\begin{bmatrix} = & < & = \\ < & = & < \\ = & < & < \\ < & = & > \end{bmatrix}$$

这个例子中有意思的一点是最内层的循环不能被移动到最外层的位置，因为它有一个“>”项。因此，这一项必须被一个在外层串行执行的循环的某些项覆盖。这使得外层循环必须串行执行。另外，次外层循环也必须被串行化，因为方向矩阵的第一行表示通过串行化其他任何循环都不能满足的一个依赖。任何启发式方法都应该发现这些事实。这种情况下，先前描述的启发式方法将首先选择J-循环串行化，因为第一个方向向量在该循环中仅有“=”。然后它将选择I-循环来覆盖内层循环。结果是 278

```
DO J = 2, M
   DO I = 2, N
      PARALLEL DO K = 2, L
         A(I,J,K) = A(I,J-1,K) + A(I-1,J,K-1) + A(I,J+1,K+1) + A(I-1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

很明显这个级别的复杂性很难由一个程序员来处理。

6.3.3 循环反转

考虑在6.3.2节开始的例子的一个变形

```
DO I = 2, N + 1
   DO J = 2, M + 1
      DO K = 1, L
         A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

这段代码的方向向量在最内层循环的所有方向都是“>”:

$$\begin{bmatrix} = & < & > \\ < & = & > \end{bmatrix}$$

这为提高并行性提供了一个机会。虽然我们不能立即把内层循环移到最外层循环的位置，但我们可以反转内层循环的迭代方向（即从L执行到1，步长为－1），它反转该循环每个依赖的方向。一旦这个变换（称为循环反转）执行了，我们就可以把循环移动到最外层的位置，得到下边的方向矩阵:

$$\begin{bmatrix} < & = & < \\ < & < & = \end{bmatrix}$$

279

因为现在所有依赖都由外层循环携带，串行执行它将使得两个内层循环可以并行执行。生成的代码是

```
DO K = L, 1, -1
   PARALLEL DO I = 2, N + 1
      PARALLEL DO J = 2, M+1
         A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
      END PARALLEL DO
   END PARALLEL DO
ENDDO
```

这个例子显示循环反转可以增加循环选择启发式方法的选择范围——它不再强制覆盖每个“>”方向，如果它能通过反转该循环来改变这些方向。

6.3.4 为并行化的循环倾斜

在第5章，循环倾斜被用来将“=”方向转变为“<”方向以产生内层循环并行性。这个变换对生成异步并行性也可以同样有帮助。考虑下面的例子:

```
DO I = 2, N + 1
   DO J = 2, M + 1
      DO K = 1, L
         A(I,J,K) = A(I,J-1,K) + A(I-1,J,K)
         B(I,J,K+1) = B(I,J,K) + A(I,J,K)
      ENDDO
   ENDDO
ENDDO
```

这个例子的方向矩阵是

$$\begin{bmatrix} = & < & = \\ < & = & = \\ = & = & < \\ = & = & = \end{bmatrix}$$

把这些依赖更精确地表达为距离矩阵，得到

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

280

由于每个循环携带一个依赖，没有立即生成并行性的明显方法。5.9节描述的循环倾斜可以被用于在一个内层循环上把“=”方向转变为“<”。当内层循环被移动到最外层并被串行化时，它可能使得很多其他循环被并行化。

例子中的最内层循环可以通过使用替换k=K+I+J相对于两个外层循环被倾斜，生成如下代码：

```
DO I = 2, N + 1
   DO J = 2, M + 1
      DO K = I + J + 1, I + J + L
         A(I,J,K-I-J) = A(I,J-1,K-I-J) + A(I-1,J,K-I-J)
         B(I,J,K-I-J+1) = B(I,J,K-I-J) + A(I,J,K-I-J)
      ENDDO
   ENDDO
ENDDO
```

这段代码变换后的方向矩阵如下

$$\begin{bmatrix} = & < & < \\ < & = & < \\ = & = & < \\ = & = & = \end{bmatrix}$$

注意现最内层循环在每个对应于携带依赖的位置都是“<”，因此如果它被移动到最外层位置并被串行化，将使得内层循环没有携带依赖。对这段代码应用这个变换得到一个循环嵌套，其内层的两个循环都可以被并行化：

```
DO k = 5, N + M + 1
   DO I = MAX(2,k-M-L-1), MIN(N+1,k-L-2)
      DO J = MAX(2,k-I-L), MIN(M + 1, k-I-1)
         A(I,J,K-I-J) = A(I,J-1,K-I-J) + A(I-1,J,K-I-J)
         B(I,J,K-I-J+1) = B(I,J,K-I-J) + A(I,J,K-I-J)
      ENDDO
   ENDDO
ENDDO
```

像我们将会看到的那样，循环倾斜有两个对于并行化很关键的性质。首先，它可以用于把倾斜后的循环变换成可以交换到最外层位置的循环而不改变程序含义。第二，它可以用于 281
以这样一种方法变换倾斜后的循环，在向外交换后，它将携带先前未倾斜时该循环携带的所有依赖。

这两个性质都是因为相同的原因所致。我们从前面的讨论了解到一个循环可以被移动到最外层的位置，如果在方向矩阵中它的列的所有项是“=”或者“<”。如果一个循环在某个依赖上有方向“>”，那么这个依赖必须被一个外层循环携带。因此，如果相对于这个外层循环用一个足够大的常数k来倾斜目标循环，我们就能够把方向转变为“=”或者“<”，这就证明了第一个属性。

第二个属性用相同的论证可以得到。对于外层循环携带的每一个依赖，在携带循环的列中总有一个正的距离。相对于携带循环，以一个足够大的因子倾斜该循环可以保证被倾斜循环在距离矩阵中的列只有大于零的项。因此，这个循环被移动到最外层的位置时将携带所有的依赖。

这些观察导致我们得出一个选择循环的启发式方法，用于图6-19具体实现循环倾斜的并行化算法中。这个启发式方法中使用的策略是尝试最多串行化一个外层循环而在下一层循环中找到并行性。而且，如果可能的话，试图在最外层的位置并行化循环。步骤是首先尝试找到一个循环，它可以覆盖当前最外层的循环且可以被移动到最外层的位置。如果能够找到这样一个循环，即可将其串行化使得在下一步对当前的外层循环并行化成为可能。

如果第一次尝试失败，那么我们将尝试循环倾斜。如果我们能够相对于外层循环倾斜某个循环并把它交换到最外层位置，我们将这样做。如果不能，我们试图找到可以移到两个最外层位置的一对循环，且内层循环可以相对于外层循环被倾斜。最后，如果这样做也失败了，我们选择串行化这样一个循环，它能够被移动到最外层并能够覆盖最多的其他循环。图6-20中给出这个选择的启发式方法。

```
procedure select_loop_and_interchange(L)
  // L是输入的循环嵌套
  // 返回的最外层循环需要被串行化执行
  设{l1, l2, ..., lk}是L中剩余循环的一个集合;

  if l2, ..., lk（称为lp）中的任何一个可以被移动到最外层的位置并且在那个位置lp包含l1
  then把lp交换到最外层位置;
  else if l1包含l2, ...lk中的任意一个then把l1留在最外层位置上;

  else begin
    设li和ls是最外层的一对循环
    (1) li可以被合法地移动到最外层的位置
    (2) ls可以被合法地移动到次最外层的位置
    (3) li和ls在距离矩阵中都是常量距离;
  if这样一对循环存在then begin // 尝试倾斜
    把li交换到最外层位置;
    把ls相对于li按照如下方法倾斜
      (a) ls可以被交换到li的外层
      (b) ls在距离矩阵中每个入口相应于li在最外层携带的循环至少为1;
    把ls交换到最外层位置;
  end
  else begin // 为串行化找到一个covering的循环
      // 选择一个有机会发掘并行性的循环
      l:=最外层的循环并且具有以下属性:
        (a) 可以被移动到最外层位置
        (b) 必须被串行化（如果存在一个这样的循环）且
        (c) 在它那一列中有最多的“<”方向;
      交换l到最外层的位置;
  end
end select_loop_and_interchange
```

图6-20 用循环倾斜的选择启发式算法

注意*select_loop_and_interchange*不试图相对于一个以上的循环进行倾斜，但是能够很容易把它扩展成那样。下边的代码能够说明如何把该算法用于循环倾斜:

```
DO K = 1, L
   DO J = 1, M
      DO I = 1, N
         A(I+1,J+1,K+1) = A(I,J,K+1) + A(I,J+1,K) + A(I+1,J+1,K)
      ENDDO
   ENDDO
ENDDO
```

该循环的方向矩阵是

$$\begin{bmatrix} = & < & < \\ < & = & < \\ < & = & = \end{bmatrix}$$

显然，所有距离都是常数（1或者0）。最外层的循环不覆盖任何其他循环，因此我们选择相对于外层循环倾斜中间循环，使用替换j=J+K。循环中所有对于J的引用被替换为j-K。 282 ~ 283

通过这些替换，变换后的循环嵌套变成

```
DO K = 1, L
   DO j = K+1, K+M
      DO I = 1, N
         A(I+1,j-K+1,K+1) = A(I,j-K,K+1) + A(I,j-K+1,K) + A(I+1,j-K+1,K)
      ENDDO
   ENDDO
ENDDO
```

这个循环被修改过的方向矩阵是

$$\begin{bmatrix} = & < & < \\ < & < & < \\ < & < & = \end{bmatrix}$$

当它被交换到最外层位置，两个内层循环都可以被并行化，虽然内层循环仍然可以串行执行，以便优化内存层次结构性能（像我们在下一小节将看到的那样）：

```
DO j = 2, L+M
   PARALLEL DO K = MAX(1,j-M), MIN(L,j-1)
      DO I = 1, N
         A(I+1,j-K+1,K+1) = A(I,j-K,K+1) + A(I,j-K+1,K) + A(I+1,j-K+1,K)
      ENDDO
   ENDDO
ENDDO
```

图6-20中的循环选择启发式方法在次最外层循环寻找并行性，但所得到的并行性不一定是负载平衡的。在我们的例子中，K-循环执行的迭代次数是从1到L之间的可变数目。然而，在异步并行性的情况下，不像向量并行那样，这并不是一个很大的问题，因为大多数并行循环都是自调度的，可以处理可变数量的工作。在10.5节我们将再回到这个主题。

6.3.5 幺模变换

循环交换、循环倾斜以及循环反转都是称为幺模变换的更为一般的一类变换的例子。“幺模”一词是从线性代数中借用的，它描述一个置换它的域而不改变域大小的映射（或矩阵）。 284

定义6.8 一个矩阵T表示的变换是*幺模变换*，如果

（1）T是方阵，

（2）T中所有元素都是整数，

（3）T的行列式的绝对值为1。

如果矩阵T_1和T_2都是幺模矩阵，那么它们的乘积$T_1 \cdot T_2$也是幺模矩阵（两个方阵的乘积是方阵；它的元素都是整数，而整数的乘积以及和也都是整数；两个矩阵的乘积的行列式是两个矩阵的行列式的乘积）。换句话说，任何幺模变换的组合仍是幺模变换。

幺模变换理论已被用于支持非常有效的目标制导的并行化策略（见Banerjee[36]以及Wolf和Lam[277]）。很多这样一些策略是使用方向矩阵方法的更为形式化的版本，因此我们在这里不进一步讨论它们。

6.3.6 基于有利性的并行化方法

由于有利的并行化有最小粒度的要求，也因为在并行代码生成中有多种选择，实践中必须采用某种方法来估计不同代码排列的代价。我们在Rice大学做的一个研究使用一种非常有效的策略来找出Fortran程序中可以并行化的循环。这个策略在一些现有商用编译器无法并行化的循环上取得成功。可惜一旦这些循环被并行化，运行速度反而比先前慢很多，因为缺乏足够的并行性粒度。

因此，需要某种形式的性能评估，使得并行代码生成有效。我们将考虑使用一个静态性能评估函数，该函数可以被用于任何代码段评估代码段的运行时间。这样一个函数不必是很精确的，但是它应能够从两种代码排列中选择出较好的一种。在开发静态评估函数时，关键的考虑是代码段中内存引用的代价以及并行循环的粒度是否足以使并行化是有利的。

在代码生成时使用代价函数来评估循环嵌套的一种方法是考虑所有可能的排列和并行化，然后选出最好的一种。但这个方法是不实际的，其原因有二。第一，所有选择的个数与嵌套
285 内循环的个数成指数关系，且代价评估自身的代价通常也很大，使得这一策略慢得无法忍受。第二，循环嵌套中很多循环的上界在编译时是未知的。这样，精确地评估运行时间是不可能的。可以生成一些不同的循环排列，并在运行时选择一个最好的，用这样的方法可以克服这种策略的缺点，但是排列的数目可能会导致代码大小的爆炸。

因为这些原因，基于代价函数本身的性质，通常只考虑可能代码排列的一个子集。作为例子，我们给出一个代码生成的启发式方法，它在选择一个紧嵌循环套的正确排列时，主要考虑内存引用的代价。这一方法是基于Kennedy和McKinley的工作[175]。

我们从开发循环嵌套中最内层循环的代价度量C_L开始。对于最内层循环，这个代价基本上就是这个循环产生的高速缓存不命中次数的上界。我们把这个度量叫做*循环代价*。为了估价一个给定循环的循环代价，我们执行下边的三步：

（1）把循环体内的所有引用划分为*引用组*。两个引用之间如果存在一个循环无关的依赖或者常数距离的携带依赖，则它们属于同一个组。直觉上，同一个引用组中，第一个引用发生的高速缓存不命中应是该组经历的惟一的高速缓存不命中，因为第二个引用会在高速缓存内找到所需的行。

（2）对于每个引用组，确定对相同引用的访问是否是（a）循环不变量，（b）单位跨距，（c）非单位跨距。对于情况（a）令该组的引用代价为1，因为对于该组，在整个循环中只会

遇到一次高速缓存不命中。对于情况（b)令该组的引用代价是迭代次数除以高速缓存行的大小（以所引用类型的数据项为单位），因为存在高速缓存行内的重用。对于情况（c），假定在高速缓存行内没有重用，令该组的代价等于循环内的迭代次数。

（3）对于最内层循环，令循环的代价等于各组的引用代价的和乘以循环总的执行次数。这基本上等于各外层循环的循环界的乘积。

作为例子，考虑矩阵乘法的最内层循环；

```
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
```

286

当K-循环是最内层循环时C的引用代价是1，因为它是循环不变量。A的引用代价是N，因为它不是单位跨距的，B的引用代价是N/L，其中由于单位跨距，L是高速缓存行大小。因此K-循环总的循环代价是$N^3(1+1/L)+N^2$。当J-循环是最内层循环时，它的代价是$2N^3+N^2$，因为C和B的引用是非单位跨距，并且A的引用是循环不变量。当I-循环是最内层的时候，我们有两个单位跨距引用组和一个循环不变量，代价为$2N^3/L+N^2$。这样，I-循环有最低的循环代价，因此从高速缓存重用的观点看最适合作为最内层循环。当然，这个符号值的比较假定上界很大。(否则，迭代数目就会很小而这样循环顺序就不重要了。）表6-1汇总循环代价。

表6-1 循环代价汇总

内部循环索引	成 本
I	$2N^3/L+N^2$
J	$2N^3+N^2$
K	$N^3(1+1/L)+N^2$

为了找到一个最好的全序，我们简单地按循环代价增加的顺序从最内层到最外层排列循环，如果高速缓存足够大，则理论上内层循环重用也可以是外层循环重用。这意味着为了得到最好的高速缓存性能，应当选择下边的循环顺序:

```
DO J = 1, N
   DO K = 1, N
      DO I = 1, N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
      ENDDO
   ENDDO
ENDDO
```

即使外层循环高速缓存重用无法得到，它仍可能通过循环分段来获得。

一旦通过循环代价的启发式方法得到想要的循环顺序，下一步则是重新排列循环，使之尽可能接近想要的顺序，尽管有时不可能得到恰好是我们想要的循环顺序，因为某些排列可能是非法的。图6-21给出一个重排循环的算法。

容易证明，如果存在一个合法的排列使得我们想要的最内层循环正好在最内层位置，这个算法就可以找出这样一个排列，使得我们想要的循环恰在最内层[176]。这个基于内存的循

环交换算法是一个将有利性信息加入代码生成策略中的启发式方法，且不会导致指数级执行
287 时间。实验证明它在实践中是很有效的[176]。

```
procedure select_permutation(O, P)
  // Input: 一个希望得到的循环排列顺序O={i1, i2, ..., in}和
  // 循环的初始方向矩阵D
  // 输出: 一个靠近的排列P
  P:=Ø;
  for i=1 to n do begin
    在O中选择最左边的循环l使得{P1, P2, ..., Pi-1, l}没有非法的方向矩阵前缀;
    从O中删除l;
    把l附加在P的后边，这样它就变成Pi;
  end
end select_permutation
```

图6-21 选择一个接近的排列

一旦我们有了按照最优内存顺序排列的循环，并行代码生成系统可以指望将内层循环标记为串行执行的，如果它能够在高速缓存方面得到好的性能的话——也就是说，如果它被并行化，可能会失去跨距为1的访问。这个标记将会阻止循环被并行化以及从最内层位置移出。如果这是仅有的并行循环，一个折中是对其作循环分段，并将按分段迭代的循环交换到外层并行执行。当然，只有当循环有足够的迭代，可以同时利用跨距为1的访问并提供足够的并行性以得到显著的加速时，这样做才是可行的。

将此策略加入代码生成算法*ParallelizeNest*相当容易，因此我们不在这里给出修改过的算法。

6.4 非紧嵌循环套

当循环在最外层是非紧嵌的，且最外层的循环不能被直接并行化的时候，最大限度的循环分布可以是一个有效的变换，因为它产生一个循环集合，每一个都可能是紧嵌的。如果有非紧嵌的循环产生，是因为有涉及某个语句和某个内层循环的紧依赖环，在那种情况下串行化外层循环并继续处理可能较好。

在这一节，我们考虑一个代码生成策略，在其中首先尝试最大循环分布，然后用多层循
288 环合并增加粒度。

6.4.1 多层循环合并

当处理循环嵌套的时候，合并问题变得更为困难，因为每个循环嵌套可能有不同的最优排列。如果我们假设仅合并在循环分布前原先合并的循环，那么不同的循环排列可能干扰循环合并。例如，考虑下面的循环嵌套:

```
DO I = 1, N
   DO J = 1, M
      A(I,J+1) = A(I,J) + C
      B(I+1,J) = B(I,J) + D
   ENDDO
ENDDO
```

在循环分布以后，很明显每个语句的外层循环恰好不同:

```
PARALLEL DO I = 1, N
   DO J = 1, M
      A(I,J+1) = A(I,J) + C
   ENDDO
END PARALLEL DO
PARALLEL DO J = 1, M
   DO I = 1, N
      B(I+1,J) = B(I,J) + D
   ENDDO
END PARALLEL DO
```

虽然每个嵌套有外层循环的并行性，这两个嵌套更难合并，因为外层循环有不同的循环头和不同的循环迭代界。另外，由于循环索引变量的数组下标位置不同，出于内存层次结构的考虑，合并这两个循环可能是不明智的。

已经证明包含循环交换的循环合并在循环个数上是NP完全问题。下面的循环说明这个问题如此困难的原因之一:

```
DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J) + X
      B(I+1,J) = A(I,J) + B(I,J)
      C(I,J+1) = A(I,J) + C(I,J)
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
   ENDDO
ENDDO
```

这个嵌套被分布于所有四个语句，产生四个循环: 289

```
DO I = 1, N ! 可以并行执行
   DO J = 1, M ! 可以并行执行
      A(I,J) = A(I,J) + X
   ENDDO
ENDDO
DO I = 1, N ! 串行执行
   DO J = 1, M ! 可以并行执行
      B(I+1,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
DO I = 1, N ! 可以并行执行
   DO J = 1, M ! 串行执行
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
ENDDO
DO I = 1, N ! 串行执行
   DO J = 1, M ! 可以并行执行
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
  ENDDO
ENDDO
```

这个循环的问题可以用图6-22中的图表明。在这个图中，每个子循环用一个结点表示，结点标有在子循环内赋值的变量。在每个结点内列出可以并行的循环索引。

现在的问题是确定哪个循环可以被合并到A循环中去。A循环产生既在B循环又在C循环内引用的值。因此我们希望与两个循环都合并。然而，我们不能合并两个循环而又不放弃并行性。如果我们合并A循环和B循环，则需要使J循环在结果嵌套中并行。但是那样以后，与C循环的合并将强制结果的I层和J层成为串行的。因此我们必须做出选择。

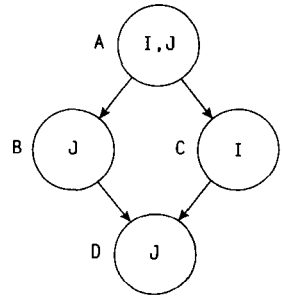


图6-22 两层合并的例子

问题在于仅仅检查后继不能确定最优的选择。虽然它们在这一点看起来一样好，但实际上与C循环合并肯定更有利。为了理解为什么会这样，我们可以假定与B循环合并。这样，由于我们前边描述过的理由，得到的结果循环就不能和C循环合并。并且，它也不能和D循环合并，因为如果两个循环在依赖图上有依赖关系，并且依赖图中存在一条路径经过一个不能被合并到同一组的循环，那么这两个循环就不能被合并。在这种情况下，穿过C循环的路径迫使我们只能留下循环D不合并。最终结果有三个并行循环和两个栅障:

```
PARALLEL DO J = 1, M
   DO I = 1, N
      A(I,J) = A(I,J) + X
      B(I+1,J) = A(I,J) + B(I,J)
   ENDDO
END PARALLEL DO
PARALLEL DO I = 1, N
   DO J = 1, M
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
END PARALLEL DO
PARALLEL DO J = 1, M
   DO I = 1, N
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
   ENDDO
END PARALLEL DO
```

另一方面,如果我们合并A循环和C循环，那么我们就可以合并B循环和D循环，那样就只会留下一个栅障和两个循环嵌套:

```
PARALLEL DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J) + X
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
END PARALLEL DO
PARALLEL DO J = 1, M
   DO I = 1, N
      B(I+1,J) = A(I,J) + B(I,J)
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
  ENDDO
END PARALLEL DO
```

290 ~ 291

就数据重用和处理器个数来说，这个版本明显要好一些。问题是在作关于如何做循环合并的决定时需要前瞻。有时甚至可能需要任意长度的前瞻。

因为这些问题，大多数并行代码生成系统使用某种启发式方法来决定如何合并多层的循环。一个有趣的启发式方法是与那些不能与其后继合并的循环合并。这个恰好能解决例子中问题的启发式方法的背后理由是，如果一个后继不能和它自己的后继合并，那么你就需要至少一个栅障。因此为什么要立即把另一个栅障放到那个后继的前边而选择两个呢？这个启发式方法被用于PFC的并行代码生成系统中。

6.4.2 一个并行代码生成算法

在本节我们提出一个代码生成算法，用于一般的情形——有层次的循环嵌套。这里的主要问题是确定什么时候使用循环分布和合并，以及在各个嵌套并行化以后决定哪些循环需要合并。

按照常规我们使用自顶向下的方法，首先尝试并行化最外层的循环嵌套。如果这样能够成功地找到足够的并行性，那就不需要再作什么了。否则就对外层循环作分布，尝试在内层循环中找到更多的并行性。结果过程——*Parallelize*——如图6-23所示。为了处理紧嵌嵌套，使用图6-19给出的例程*ParallelizeNest*，以及图6-20的循环选择启发式方法。我们假定*ParallelizeNest*已如6.3.6节讨论过的那样被修改为用于对循环做重排序以提高内存层次结构的性能，并当循环需要在单个处理器上运行时将内层循环标记为串行的，以便有效地使用高速缓存。

```
procedure Parallelize(l, D_l)
  // l是需要并行化的循环嵌套
  // D_l是限于l中语句的依赖图
  Parallelize(l, success);
  if ¬success then begin
    if l可以被分布then begin
      把l分布到循环嵌套l_1, l_2, ..., l_n中;
      for i:=1 to n do begin
        设D_i是l_i中语句间依赖关系的集合
        Parallelize(l_i, D_i);
      end
      Merge({l_1, l_2, ..., l_n});
      end
    else begin
      // 或者l携带一个依赖环,或者它包括一个因为内存性能而必须在单个处理器上运行的语句
      // 注意ParallelizeNest为嵌套中的每个语句生成了一个串行循环。然而，它没有试图
      //    立即把这个循环分布到外层循环中去;
      for each嵌套在l中的外层循环l_o do begin
        设D_o是l_o中语句间的依赖关系集
          去掉l携带的依赖;
        Parallelize(l_o, D_o);
      end
      设S是l中剩下的外层循环和语句循环的集合;
      if ‖S‖ >1 then Merge(S);
    end
  end
end Parallelize
```

图6-23 一个通用代码生成算法

*Parallelize*首先调用*ParallelizeNest*，尝试6.3节讨论的紧嵌嵌套策略。如果这一步失败了，就尝试把循环分布为一系列子嵌套，递归地并行化每个子嵌套，并尽可能地合并得到的结果嵌套。如果循环不能被分布，就串行化最外层循环，并在其循环体内的顶层循环递归调用自己。当这个过程结束，尝试合并尽可能多的结果嵌套。

这个算法的关键部分是图6-24的例程*Merge*，它把并行化的循环尽可能地重新合并起来。*Merge*使用一种类似图6-9给出的用于串行-并行合并的算法。其基本思想是把不同的循环按照类型分类，每种类型表示原来嵌套中的最外层循环，且在分布后的循环中也能放在最外层，并且不会改变关键的性能因素，例如并行性、粒度或最内层位置循环的内存性能。若两个循环在正则形式下一个的最外层循环是并行的而另一个的是串行的，则此二循环的类型不同。作为例子，考虑下面的循环嵌套:

292
~
293

```
procedure Merge(S)
  // S是需要合并的循环嵌套集合
  // 注意: 单个语句可以被视为一种特别类型的循环

  设{t1, t2, ..., tm}是循环类型的集合;
  按照顺序对S中每种类型的最外层循环进行合并
    使用图6-9给出的TypedFusion;
  for each合并的组G do begin
    设{l1, l2, ..., lk}为合并后的组G内的循环嵌套;
    Merge({l1, l2, ..., lk});
  end
end Merge
```

图6-24 一组循环嵌套的循环合并

```
DO J = 1, M
   DO I = 1, N
      A(I+1,J+1) = A(I+1,J) + C
      X(I,J) = A(I,J) + C
   ENDDO
ENDDO
```

这个嵌套中的哪一个循环都不能并行化，因此对外层循环作分布:

```
DO J = 1, M
   DO I = 1, N
      A(I+1,J+1) = A(I+1,J) + C
   ENDDO
ENDDO
DO J = 1, M
   DO I = 1, N
      X(I,J) = A(I,J) + C
   ENDDO
ENDDO
```

这些循环中的每一个都可以并行化，第二个是在两维上并行化，尽管代码生成器可能串行化第二个嵌套的内层循环以保证它具有可接受的内存性能:

```
PARALLEL DO I = 1, N
   DO J = 1, M
      A(I+1,J+1) = A(I,J+1) + C
   ENDDO
ENDDO
PARALLEL DO J = 1, M
   DO I = 1, N ! 为了内存层次结构的性能保留串行
      X(I,J) = A(I,J) + C
   ENDDO
ENDDO
```

294

第一个循环嵌套的类型是[I-循环，并行]。第二个循环嵌套是[J-循环，并行]。因此这两个循环有不同的类型，从而不能在最外层被合并。

一个稍微复杂一些的嵌套如下所示，类似于6.4.1节的例子:

```
DO J = 1, M
   DO I = 1, N
      A(I,J) = A(I,J) + X
      B(I+1,J) = A(I,J) + B(I,J)
      C(I,J+1) = A(I,J) + C(I,J)
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
   ENDDO
ENDDO
```

在循环分布和并行化以后，它变成

```
L1   PARALLEL DO J = 1, M
        DO I = 1, N ! 为了内存层次结构的性能而串行化
           A(I,J) = A(I,J) + X
     END PARALLEL DO
        ENDDO
L2   PARALLEL DO J = 1, M
        DO I = 1, N
           B(I+1,J) = A(I,J) + B(I,J)
        ENDDO
     END PARALLEL DO
L3   PARALLEL DO I = 1, N
        DO J = 1, M
           C(I,J+1) = A(I,J) + C(I,J)
        ENDDO
     END PARALLEL DO
L4   PARALLEL DO J = 1, M
        DO I = 1, N
           D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
        ENDDO
     ENDDO
```

第一、第二和第四个循环被划分为同一个类型。合并图如图6-25所示。

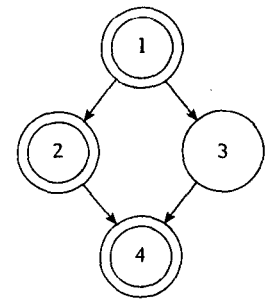


图6-25 并行和串行子图

一个贪婪合并算法将会合并循环L_1和L_2，而单独留下L_4。此外，对*Merge*的递归调用也将合并内层的串行循环。结果如下:

```
L1   PARALLEL DO J = 1, M
```

```
        DO I = 1, N
           A(I,J) = A(I,J) + X
           B(I+1,J) = A(I,J) + B(I,J)
        ENDDO
     END PARALLEL DO
L3   PARALLEL DO I = 1, N
        DO J = 1, M
           C(I,J+1) = A(I,J) + C(I,J)
        ENDDO
     END PARALLEL DO
L4   PARALLEL DO J = 1, M
        DO I = 1, N
           D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
        ENDDO
     ENDDO
```

像我们在6.4.1节指出的那样，对于这个例子，有可能得到更少的并行循环。不过我们打算接受这个结果，因为它有更好的内存性能，而且最小化并行循环的数量问题是一个NP完全问题。

295 ~ 296

关于*Parallelize*有最后两点需要说明。首先，把它应用于给定程序中一组循环嵌套的最外层经常是有用的。为此需要一个驱动程序，它可以包括在*Parallelize*的顶层或者单独存在。驱动程序如图6-26所示。它将用于6.5节扩充的例子。

```
procedure DriveParallelize(L, D)
  // L是在最外层的循环嵌套（以及单个语句）的列表
  // D是L中的语句集的依赖图
  for each循环嵌套l ∈ L do begin
    设Dl是l中语句间的依赖关系集合;
    Parallelize(l, Dl);
  end
  if ‖ L ‖ >1 then merge(L);
end DriveParallelize
```

图6-26 循环和语句集并行化过程的驱动程序

其次，*Parallelize*如同写出的那样，在尝试任何分布之前，总是更倾向于并行化整个循环，即使它不是最外层循环。这可能不会对任何机器都是最佳的策略。同时尝试两种策略，并根据某种标准评价这两种方法所得结果，然后选择最好的，这样可能得到一个更好的结果。我们把修改*Parallelize*以完成以上任务留作一个习题（习题6.4）。

6.5 一个扩充的例子

我们将用对一段特殊示例代码的讨论来结束对循环嵌套的代码生成的处理。我们选择的代码是Erlebacher的一个子例程，它是由Thomas M. Edison编写的一个解NASA微分方程的程序。原始代码见图6-27。注意循环已经被最大限度地分布。

如果我们对各个嵌套应用图6-23的通用多层代码生成算法，那么在每个嵌套中J-循环都会被并行化，而I-循环则仍是串行的，以保证在单个处理器上的顺序访问。因为得到的最外层循环是相容的——它们都是并行的，并且从J=1执行到JMAXD——这些最外层的循环都可以在*TypedFusion*的第一次应用中被合并而生成图6-28所示的代码。

```
L1    DO J = 1, JMAXD
         DO I = 1, IMAXD
            F(I,J,1) = F(I,J,1) * B(1)
         ENDDO
      ENDDO
L2    DO K = 2, N - 1
         DO J = 1, JMAXD
            DO I = 1, IMAXD
               F(I,J,K) = (F(I,J,K)-A(K)*F(I,J,K-1)) * B(K)
            ENDDO
         ENDDO
      ENDDO
L3    DO J = 1, JMAXD
         DO I = 1, IMAXD
            TOT(I,J) = 0.0
         ENDDO
      ENDDO
L4    DO J = 1, JMAXD
         DO I = 1, IMAXD
            TOT(I,J) = TOT(I,J) + D(1) * F(I,J,1)
         ENDDO
      ENDDO
L5    DO K = 2, N - 1
         DO J = 1, JMAXD
            DO I = 1, IMAXD
               TOT(I,J) = TOT(I,J) + D(K) * F(I,J,K)
            ENDDO
         ENDDO
      ENDDO
```

图6-27 Erlebacher的子例程*tridvpk*

```
L1    PARALLEL DO J = 1 , JMAXD
         DO I = 1, IMAXD
            F(I,J,1) = F(I,J,1) * B(1)
         ENDDO
L2       DO K = 2, N - 1
            DO I = 1, IMAXD
              F(I,J,K) = (F(I,J,K)-A(K)*F(I,J,K-1)) * B(K)
            ENDDO
         ENDDO
L3       DO I = 1, IMAXD
           TOT(I,J) = 0.0
         ENDDO
L4       DO I = 1, IMAXD
           TOT(I,J) = TOT(I,J) + D(1) * F(I,J,1)
         ENDDO
L5       DO K = 2, N - 1
           DO I = 1, IMAXD
             TOT(I,J) = TOT(I,J) + D(K) * F(I,J,K)
           ENDDO
         ENDDO
      ENDDO
```

图6-28 单个嵌套并行化以后Erlebacher中的子例程*tridvpk*

*TypedFusion*接着用于下一层循环，它们都是串行的。这产生图6-29的合并图。这里循环L_1，L_3和L_4（在I上的外层循环）被赋予一种类型，而L_2和L_5（在J上的外层循环）被赋予另一种类型。这个第二遍应用*TypedFusion*生成图6-30所示的最终的程序，在最外层有一个并行循环而所有内层循环以跨距1访问关键数组。

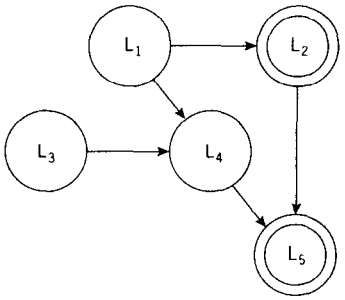


图6-29 Erlebacher的合并图

6.6 并行性的封装

通常并行代码的性能不仅依赖于能够找到多少并行性，也依赖于它们如何封装。我们在6.3.6节已经看到在并行性和内存层次结构性能之间的一个折中。在这一节我们将考虑并行性和同步粒度之间的折中。像我们在引言中所说的那样，在不同机器之间，每个同步操作的代价可以是不同的。如果没有并行线程的启动和同步的代价，那么按最小粒度封装并行性就总是最好的。然后零-代价调度程序总可以保持每个处理器的忙状态，直到所有任务完成。最大的负载不平衡等于并行处理器上执行的任务的最大粒度，而开销的时间则是零。

297 ~ 299

```
PARALLEL DO J = 1, JMAXD
   DO I = 1, IMAXD
      F(I,J,1) = F(I,J,1) * B(1)
      TOT(I,J) = 0.0
      TOT(I,J) = TOT(I,J) + D(1) * F(I,J,1)
   ENDDO
   DO K = 2, N - 1
      DO I = 1, IMAXD
         F(I,J,K) = (F(I,J,K)-A(K)*F(I,J,K-1)) * B(K)
         TOT(I,J) = TOT(I,J) + D(K) * F(I,J,K)
      ENDDO
   ENDDO
ENDDO
```

图6-30 Erlebacher中子例程*tridvpk*的最终代码

因为在真实系统上，线程启动和同步的代价都是大于零的，选择一个合适的折中就变得更加复杂。较大粒度的工作单位意味着在较小并行性和较差负载平衡的代价下可以较少执行调度和同步操作。这样的折中需要编译器针对不同的目标平台做出不同的选择。

在这一节我们将讨论两个用于处理并行性和同步之间折中的工具。循环分段是一种通过减少可用并行进程总数来直接增加粒度的策略。另一方面，在没有其他方法可用时，流水线给出一种通过显式同步来得到并行性的方法。像我们将会看到的那样，流水线还涉及到另一个在粒度和并行性之间的折中。

6.6.1 循环分段

很多不含依赖的循环，其迭代可以正确地并行执行，但在调度算法比较简单的情况下可能无法有效地并行执行。常见的情况是单个循环迭代中的工作量可能不足以使其成为调度的单位。在这种情况下，把多个迭代组成集合，每个集合作为一个调度单元，这样可以更有效

地利用并行性。与向量化类似的变换是循环分段——把可用的并行性转换成一种更适合硬件的形式。下面的简单循环将说明这种变换对并行性的优点。 300

```
DO I = 1, N
   A(I) = A(I) + B(I)
ENDDO
```

如果恰有P个处理器可用于执行循环，那么最佳的负载平衡和最小的同步可以由如下程序得到:

```
k = CEIL(N/P)
PARALLEL DO I = 1, N, k
   DO j = I, MIN(I+k-1,N)
      A(j) = A(j) + B(j)
   ENDDO
END PARALLEL DO
```

在像这样的循环中，每个迭代很明显需要相同的计算量，一旦知道循环迭代的个数和可用的处理器个数，找到适当的平衡点是很容易的。由于那些值经常到运行时才能知道，这种循环分段经常由特别的硬件完成（像Convex C2和C3中的情况）。

如果迭代的执行时间是不同的，如像在

```
DO I = 1, N
   DO J = 2, I
      A(I,J) = A(I,J-1) + B(I)
   ENDDO
ENDDO
```

这样的程序中，给出一个有效的平衡点更加困难。通常宁愿选择一个较小的块大小，而不是通过循环分段把并行循环（在这个例子中是外层）平均分到处理器。这样，当负担较重的处理器还在执行时，执行任务较少的处理器（也就会较快结束）可以承担一些过剩的工作，从而提供某种负载平衡。较小的块大小在处理器开始交错工作时也更有利，因为这允许首先开始的处理器多承担一些负载。已经提出若干种不同的并行进程的动态调度方案。这些技术将在本章后面讨论。

6.6.2 流水线并行性

另一种形式的粗粒度并行循环是DOACROSS，它用迭代间的同步实现并行循环迭代的流水执行（基本上用多个处理器作为高层向量处理器）。例如，循环 301

```
      DOACROSS I = 2, N
S1       A(I) = B(I) + C(I)
         POST(EV(I))
         IF (I .GT. 2) WAIT(EV(I-1))
S2       C(I) = A(I-1) + A(I)
      ENDDO
```

可以完全像用一个PARALLEL DO循环那样并行地执行S_1的所有实例，但是S_2的各个迭代必须在所需的结果可用时按流水方式执行。在这个例子中，POST(EV(I))发布事件EV(I)已经发生的信号而WAIT(EV(I))阻塞直至事件被发送。通常，所有事件都初始化为未发送状态。

一个更精巧的例子是流水线版的有限差分松弛法的波前并行化。串行的循环嵌套如下所示:

```
DO I = 2, N-1
   DO J =2, N-1
      A(I,J) = 0.25 * (A(I-1,J) + A(I,J-1) + A(I+1,J) + A(I,J+1))
   ENDDO
ENDDO
```

我们已经知道如何用循环倾斜来并行化这个循环。循环倾斜也可以看做是一种实现流水线的方法。现在我们将说明如何通过DOACROSS循环来实现它。在这个实现中，我们将并行执行外层循环而串行执行内层循环。这里重要的一点是保证我们在计算出A(I,J)之后才开始A(I+1,J)的计算。像上面一样，我们将使用一个二维事件数组来保证计算正确地同步。

```
DOACROSS I = 2, N-1
   POST(EV(1))
   DO J =2, N-1
      WAIT(EV(J-1))
      A(I, J) = 0.25 * (A(I-1, J) + A(I, J-1) + A(I+1, J) + A(I, J+1))
      POST(EV(J))
   ENDDO
ENDDO
```

这个并行循环的执行过程如图6-31所示。注意在这个图中的箭头表示在一个公共事件上的post-wait同步。

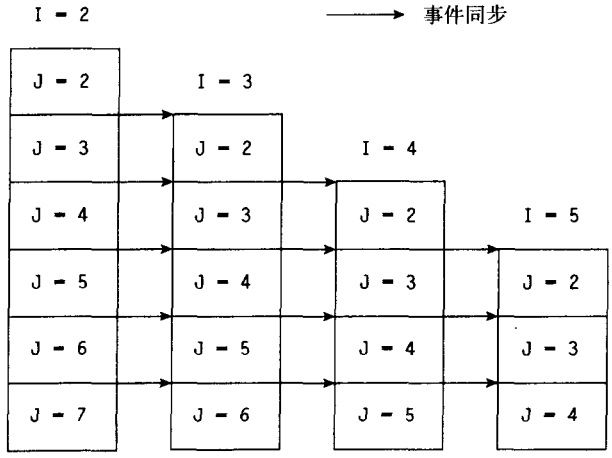


图6-31 流水线并行化

如果同步的开销较大，我们可能希望把迭代分组以减少同步的频率。例如，如果我们希望同步次数比通常少，可以如下把迭代分为两个一组：

```
DOACROSS I = 2, N-1
   POST(EV(1))
   K = 0
   DO J = 2, N-1, 2
      K = K+1
      WAIT(EV(K))
      DO m = J, MAX(J+1, N-1)
         A(I, m) = 0.25 * (A(I-1, m) + A(I, m-1) + A(I+1, m) + A(I, m+1))
      ENDDO
```

```
      POST(EV(K+1))
    ENDDO
  ENDDO
```

注意，我们为降低同步的频率付出了减少并行性的代价。从图6-32可以看到这一差别。在这个例子中，虽然I-循环的最后一个迭代开始得很迟，但如果同步代价很高的话，减少同步的执行频率仍然可以弥补开始时间延迟的代价。 302 ~ 303

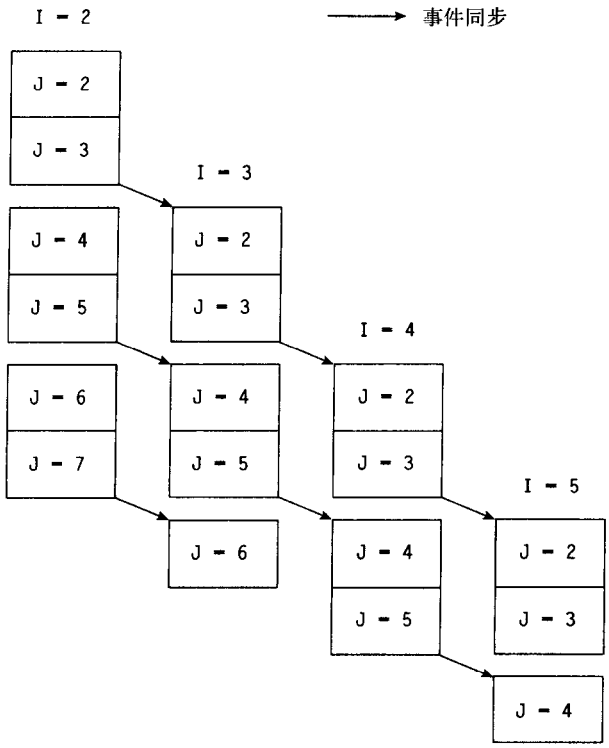


图6-32 粗粒度流水线并行化

由于DOACROSS的效果是与机器密切相关的(它的执行效率很大程度上依赖于处理器同步)，它应当只在流水线并行性是提高性能的惟一方法时才使用。

6.6.3 调度并行任务

在多个处理器间成功地调度并行循环迭代是很困难的，并且这曾经是并行程序设计得到广泛接受的一个主要障碍。在一方面，大多数必需的原语是很简单的且在很多操作系统上已经实现。例如在Unix机器上，所有需要的只是一个轻量级线程（通常通过一个系统调用fork的变形得到)，共享地址、数据空间和堆栈，但是有它自己的“线程局部的”存储区（通常在页面调度机制中使用写拷贝语义创建)。轻量级线程必须能够创建自身的栈以支持子例程调用;这通常在线程局部存储以外实现。 304

在这样的环境下，一个相当标准的调度并行循环迭代的技术是把循环变量保存在一个同步的共享单元。当一个处理器空闲时，循环迭代就被分配到这个处理器上，和顾客在一个“取号”的面包店里得到服务的情形很相似（因此也称作“面包店计数器”调度)。在这样一

个方案下，并行程序在开始时执行一个系统调用获取可用的处理器。当它得到处理器以后，程序将除主处理器（启动处理器）之外的其他处理器都设为低优先级，等待任务。主处理器然后继续串行执行程序。当遇到一个并行循环时，主处理器初始化循环变量并为所有正在等待的处理器设置一个跳转向量，将这些处理器释放出来让它们从相应的并行循环开始执行（每个线程都拥有相同的代码空间的拷贝）。每个处理器通过同步到计数器取一个循环迭代来执行。这个过程轮流进行直到所有任务完成为止，这时从处理器重新进入空闲循环而主处理器将继续在串行区域中执行。大体上，面包店计数器方法在减少同步开销和平衡处理器间的任务方面达到了较好的平衡。然而，这种方法在某些情况下也有可能效率很低——例如，如果一个被分配到某个迭代的处理器被操作系统中断了很长一段时间去执行其他任务。在这种情况下，该迭代就会阻塞后边的计算。

面包店计数器算法说明成功的并行性调度中所涉及的一些固有的矛盾——适当地解决负载平衡和同步的问题。为了达到最佳的负载平衡，在一个并行过程内的任务应当被划分为可能的最小单位（最小的粒度）。那样，在任何时间都有最大数量的单位用于分配。这就减少了一个处理器忙于完成一个任务而其他处理器空闲地等待任务的可能性。

然而，缩小并行性的粒度的代价是同步的开销。把一个单位的任务分配到一个处理器上涉及到一些同步——在面包店计数器方法中，同步是对共享循环变量加锁。当粒度变得更细时，就有更多单位要分配；每当一个单位被分配，就多一些同步开销。显然，将一个并行过程作为单个线程（串行执行）来执行不会引入任何开销；而把每条指令作为一个独立线程并行执行则显然会引入最大的开销。严格地最小化开销等于支持串行执行，当然这会导致最差的负载平衡。换句话说，最低的开销是在负载平衡最差点达到的，而最佳的负载平衡是在最
305 高的开销点达到的。在这中间的某些点，可以有效地使用处理器而不引入过多的开销；找到那些点是有效的并行调度的关键。

为了具体地说明这一平衡多么重要，考虑下面的并行三角矩阵操作：

```
DO PARALLEL I = 1, 100
   DO J = 2, I
      A(J,I) = A(J-1,I) * 2.0
   ENDDO
ENDDO
```

第一个并行循环迭代不执行乘法，第二个迭代执行1次乘法，第三个迭代执行2次乘法，依此类推，直到最后一个迭代执行N次乘法。如果要在一个有4个处理器的机器上调度它，一个达到最小同步开销的方法是把并行循环分成4块执行，每块执行$N/4$个迭代。当然，问题是，第一个处理器将执行大约$N/8$次乘法，第二个执行$3/8N$次乘法，第三个执行$5/8N$次乘法，而最后一个处理器大约要执行$7/8N$次乘法。换句话说，前3个处理器将很快会无事可作，而最后一个处理器将忙于执行大部分的任务。把循环分成最细的粒度——每个处理器每次执行一个循环迭代——提供最佳的负载平衡，但是也导致最大的同步开销。

平衡这相互矛盾的两个因素是得到有用并行性的关键。作为一般的原则，任何能够达到这一平衡的有效的系统都需要用到动态处理器调度（和静态调度相对），因为并行性的量和处理器的可用性只有在运行时才能知道。面包店计数器算法是动态调度的一个例子；它也是自调度的一个例子，即每个处理器自行决定下一个执行的任务是什么（相对于有一个全局控制单元控制执行的情形）。动态自调度算法对于达到负载平衡且避免过大的开销是必要的。下一

节我们讨论制导的自调度——一种为此目标而提出的动态自调度方法。

6.6.4 制导的自调度

如前所指出的那样，有效地调度循环级并行性的关键在于平衡负载使得每个处理器都保持忙碌，同时又不引入过多的开销而破坏并行性带来的好处。在并行计算中，有很多种类的开销，包括循环变量的访问，对全局变量的同步访问，以及辅助处理器导致的额外的高速缓存不命中。这些开销很容易超过并行执行所带来的好处。例如，简单地假定每个处理器的开 306
销是一个常数因子σ_0（对于p个处理器会导致$p\sigma_0$的总开销），那么如果

$$\sigma_0 \geqslant (NB)/p \tag{6-3}$$

其中N是循环迭代的个数而B是执行一个循环迭代所需要的时间，则并行执行将总是比串行执行慢。换句话说，如果用p个处理器执行，每个处理器的开销等于总的处理器执行时间的$1/p$，则执行时间将大于串行执行的时间，意味着并行执行不能得到任何好处。要达到这样量级的开销根本不难，特别是当处理器的个数p增加的时候。

制导的自调度（GSS）采用某种静态调度来指导动态自调度。基本上，GSS试图做两件事情：（1）最小化同步开销，（2）保持所有处理器在任何时间都是忙碌的。GSS通过按迭代组调度到处理器来减少同步开销，而不是像在面包店计数器算法中那样按单个迭代。然而，当把一组迭代打包发送给某个处理器执行时，它试图保证留有足够的任务以保持所有其他处理器忙碌，直到被调度的处理器完成它的任务。结果是这样一种动态平衡：在并行区域处理的较早阶段，大块的任务被分配到处理器；而当接近并行区域的末尾，就分配较小的块。希望是较早分配的块没有大到使得负载不平衡超过后面迭代的所能补偿的范围。

更形式化地说，GSS首先采用循环变换，如循环交换和循环合并等，来得到尽可能大的并行循环。然后它把循环的迭代以面包店计数器的方式分配到处理器，在时间t时分配的迭代个数为

$$x = \left\lceil \frac{N_t}{p} \right\rceil \tag{6-4}$$

其中N_{t+1}被设为$N_t - x$。也就是说，每个处理器在任何给定时间取得第$1/p$个剩余迭代，得到一个大小合理的块来工作，同时留下足够多的迭代以确保当它工作时其他处理器保持忙碌。使用GSS在一个有4个处理器的机器上调度一个有20个迭代的循环，得到如表6-2的分配结果。

表6-2 GSS下的迭代分配

步骤（t）	处 理 器	剩 余 数	已 分 配 数	已完成的迭代
1	P1	20	5	1-5
2	P2	15	4	6-9
3	P3	11	3	10-12
4	P4	8	2	13, 14
5	P4	6	2	15, 16
6	P3	4	1	17
7	P2	3	1	18
8	P4	2	1	19
9	P1	1	1	20

这里处理器的分配和所需要的时间假定所有循环迭代需要相同的时间来完成，且没有外部中断发生。根据这些假定，P1执行6个迭代，P2执行5个迭代，P3执行4个迭代，而P4执行5个迭代——不是一个很完美的平衡，但是由于实际上会有一些因为同步开销而导致的出入，所以这是一个好的平衡。总的同步执行次数为9次，而面包店计数器算法需要20次。

图6-33给出一个GSS的形式化描述。该算法执行一些静态的初步变换（循环合并，循环交换，循环分布），使得并行循环尽可能大，然后设置运行时分配所需的静态信息。显然，GSS没有（也不能）为每种可能的并行循环的变体找到最优调度——例如，在表6-2描述的例子中，如果前5个循环迭代都需要执行很长时间而剩余的迭代的执行时间都很短，那么显然把它们单个地分配出去会更好。然而，如果给什么是“最优”一个合理的限定，那么可以证明在任何初始处理器配置下，GSS可以得到最优调度并且需要最少数量的同步点。

```
// 给出任意一个至少包含一个DOALL的循环嵌套L
// 为p处理器调度

尽可能地分布L中的循环。
交换DOALL到尽可能的最外层位置。
尽可能地塌缩所有的内层循环到外层。

调度每个并行循环如下:
N_t := N; // 并行循环迭代总数
While N_t > 0 do begin
  If存在空闲处理器then begin
    调度x := (N_t + p - 1)/p个迭代;
      N_t := N_t - x;
  end
end
```

图6-33 制导自调度

307～308

这里，上取整操作符被改写为 $(N_t+p-1)/p$，这是等价的。注意最后的4次分配都是单个的循环迭代。这并非这个特定例子中的一种巧合；GSS在其最简单形式下（也被称为GSS（1））保证最后$p-1$个分配都是单个循环迭代。这些分配可通过对GSS算法稍加修改而消除。特别是，如果在每次同步中发送出去的迭代数是 (N_t+2p-1)，那么例子中的迭代流就变为（6, 5, 4, 3, 2）。这个变形就是GSS（2），一般地说，GSS（k）保证所发送出去的迭代块包含不少于k个迭代。基本的GSS算法可以通过调节发出迭代数公式中p的系数而改变为GSS（k）。当然，需要为终止条件作一些小的修改，因为只有GSS（1）是保证最后正好剩下0个迭代，其他的则可能超过。

制导的自调度通过使用编译时信息来指导运行时的调度，在静态和动态调度之间达到出色的平衡。特别是，编译器可以针对不同循环根据迭代的数量和每个迭代的预期执行时间来调节k的值——迭代数对处理器数的比值越高，k的值就应该越大。由于没有一个通用的算法可以完美地调度任意的并行循环，GSS确实以少得多的同步开销提供面包店计数器算法所能达到的平衡。

6.7 小结

为生成在对称多处理器上执行的程序，所面临的主要挑战是处理并行性和并行粒度之间

的折中。如果没有足够的并行性，处理器就不能被有效地利用。另一方面，如果计算的粒度太细，并行执行的启动和同步代价将会超过性能上的收益。因此，挑战在于发现可能的最粗粒度的并行性。

我们提出在三种情况下解决这个问题的方法：

- 在单个循环中，包括私有化、对齐和复制在内的许多变换可被用于消除携带依赖，从而避免循环分布，那样会减小并行粒度。如果需要作循环分布，可以在其后尽量作循环合并以使粒度减小的影响降至最小。
- 在紧嵌循环嵌套中，循环交换、循环反转和循环倾斜可被用于发现并行性并将其移动到尽可能外层的位置。在很多情况下如何组织循环以求最优性能不仅是一个最大化并行性的问题，还要考虑体系结构因素，例如内存层次结构的性能。
- 在一般的循环嵌套中，即不一定是紧嵌循环嵌套，可以先作多层循环分布，然后对结果循环嵌套进行并行化并尽量作循环合并，这样可以发掘可用的并行性并高效地加以封装。 309

在一个真正有效的编译器中，所有这些策略都是需要的，并且它们必须与其他提高性能的策略合作，特别是内存层次结构管理。

6.8 实例研究

Rice大学在PFC和ParaScope以及在Ardent Titan编译器上的研究都把提高粗粒度并行性作为一个主要目标。这一节将对在这两个项目中所使用的方法作一个回顾。

6.8.1 PFC和ParaScope

PFC系统按本章描述的方法作并行化。对于嵌套的循环，它将尝试把并行循环移到尽可能外层的位置上。内层循环或者非紧嵌循环套将会被分布、并行化，然后像在6.2.5节描述的那样再重新合并到一起。但这里没有使用带类型的合并。

因为PFC最初是作为一个向量化编译器构造的，它没有执行循环嵌套外的依赖分析。这是并行化的问题，对于并行化而言，考虑在一个子例程内跨越循环嵌套的依赖是很重要的。多年来，这个问题的改善是通过把子例程看成有一个循环包围在例程体外，目的是作依赖分析，但是不对这个循环携带的依赖进行测试。

Kathryn McKinley为她的博士论文使用ParaScope系统进行了实验，ParaScope系统是PFC的后继，使用了PFC的依赖信息以确定变换的安全性和有利性[209]。变换的驱动程序相当接近于图6-23的算法并且是完全自动的，合法性和安全性测试也是这样。然而，变换本身是使用一个交互式的变换系统——ParaScope编辑器，按照驱动程序生成的指令手工进行的。这样得到的结果就和完全自动系统所能达到的效果非常相似。在McKinley的测试中，自动系统和有经验的并行编程用户的手写程序在一个有19个处理器的Sequent Symmetry SMP上进行了加速比的比较，这些用户都不是PFC或者ParaScope开发组的成员。结果如图6-34所示。

注意在实验时还没有LINPACKD基准测试程序的手工并行化版本。基准测试程序“Control”的加速是在有8个处理器的配置下得到的。更多实验的细节见McKinley的博士论文[209]。 310

图6-34显示，在中等大小的对称多处理器上，不需要对算法做太多改动，本节所描述的自动并行化策略与一般用户的手写代码是不相上下的。惟一较大的失败是出现在“Multi”基准测试程序，其中用户使用了一个临界区来同步对共享变量的更新。自动系统未能发现这可以得到相同的结果，因为那实际上也是对内存访问的重排序。

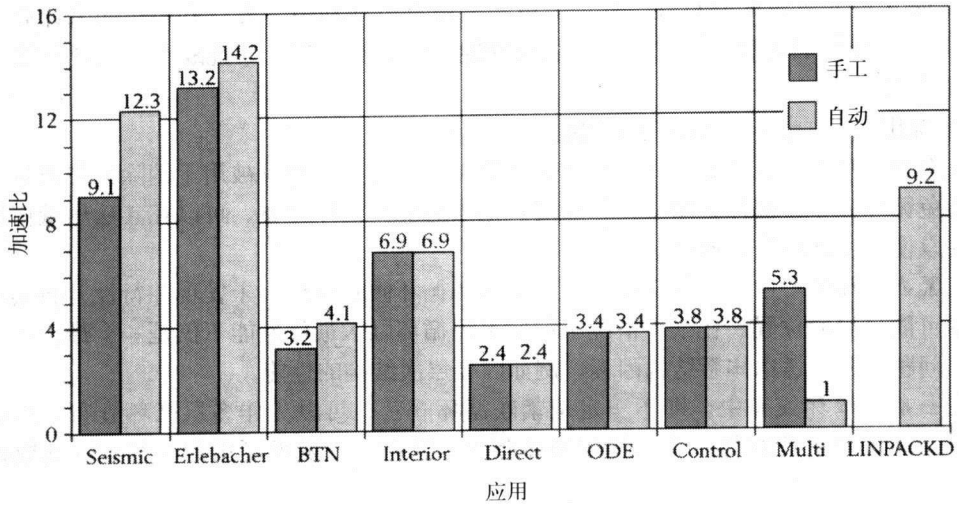


图6-34 在Sequent Symmetry上使用PFC/ParaScope得到的并行化加速比

6.8.2 Ardent Titan编译器

Ardent Titan有多个向量处理器。为简单起见，它试图向量化和并行化各一个循环。一旦这样决定，循环交换就变得比较容易了。在理想情况下（也就是，在一个能产生一个向量循环和一个并行循环的循环嵌套中），在Titan上进行循环交换的目标如下：

（1）把向量循环移动到最内层位置。根据向量化的定义，这个变换一定是合法的。并且，因为这个循环将在并行代码生成阶段被“三角化”，它将在*codegen*的过程中消失。

311

（2）把并行循环移动到可能的最外层位置。

（3）把对重用来说最佳的循环正好移动到“tripled”循环的外面——生成代码中的真正最内层位置。因为向量寄存器是向量部件上仅有的一种“高速缓存”，因此目标是尽可能地重用向量寄存器。这是Titan最大的成功之处，而剩下的把循环放在正确位置等其他的次要工作，对于讨论编译时间的代价来说就不能被认为那么重要了。

这一策略简化了依赖测试和代码生成。这个受限制形式的循环交换不需要完整的依赖矩阵。为确定一个给定的外层循环是否可以被向量化，Ardent编译器仅使用简单最内层化测试——一个不携带依赖的循环总是可以被放在最内层位置并且向量化。把并行循环移动到最外层位置是通过反复交换并行循环和它直接外层的循环而实现的这是合法的——只是简单地并入到代码生成的过程。因此，移动到最外层的测试变成了一个向内交换的测试。如果它确实携带了一个依赖，那么一个重要的标准是当它向内交换时仍然携带这个依赖（因为否则向外移动就会破坏并行性）。这样，移动到最外层的重要标准是如果外层循环包含一个依赖，这个依赖会随着交换移动到内层。这和上边第三个条件的要求其实完全相同——得到一个将重用置于最内层位置的循环。因此，循环交换所要求的惟一的特殊测试是在向量化意义下的“禁止交换”条件之一，虽然那些正是Titan编译器会交换的依赖。

很多在并行化中使用的循环变换是用户也可以作的。例如，循环分布就很容易在源代码级进行，而那些最老练的用户可以判断他们什么时候可以安全地分布循环。循环合并是用户自然会使用的一种变换——当他们写一个循环时，他们把尽可能多的代码放到循环里。基于这个观点，Ardent团队决定在Ardent编译器中不实现循环合并。这个决定的原因有如下几点：

（1）依赖图的大小：为了实现任意循环合并，依赖图的计算必须跨越不重叠的循环嵌套和其间的语句。这很快就会变得很大。

（2）依赖测试的简单性和速度：很多关于循环交换和标量扩展的决策已经简化了编译器需要的依赖的属性。去掉计算合并限制条件的要求又多消除了一个需要作决策之处。

（3）可用性：如前所述，我们认为循环合并用于一般的程序可能只能得到很少的好处，因为绝大多数程序员自然倾向于合并[⊖]。在某种我们确实希望合并不相交循环（展开和压紧）的情况下，安全性可以在需要时动态地计算。 312

重要的是注意这个限制条件不是要避免所有的循环合并，而是想简化本来不相交的循环的合并。Ardent优化器会合并从同一个循环分布而来的循环，像在*codegen*中那样，但是这种类型的合并不需要任何安全性计算。这些循环原来就是同一个,这一事实保证它们可以被安全地合并。同样需要注意Ardent编译器不支持Fortran 90，它需要通过合并从数组语句得到性能。

Ardent Titan编译器的总的并行代码生成策略可以总结为以下两个原则:

（1）最多向量化一个循环并且并行化一个循环。

（2）不进行任何辅助变换（例如标量扩展、if转换或循环交换），除非这些变换确实可以提高向量化、并行化或者内存重用。

第一个原则意味着代码生成算法应集中于找到最优的向量循环和并行循环，而不是所有向量循环和并行循环。第二个原则意味着代码生成应按需求驱动辅助变换，只在需要时才实现变换。尽管这两个原则初看起来较难于加入到通用的*codegen*算法中去，但它们实际上只需要很小的改动就可以融和到一般的方案中。Ardent优化器最后采取的总策略是:

（1）所有依赖边都带有一个叫做“可删除性”的属性，扩展了标量扩展中引入的概念。这个属性为真时表示这条边可以通过某种变换删除。

（2）表示控制流的控制边被显式地加入到依赖图中，返回边也加进去，强制所有在一个控制区域内的语句在整个*codegen*过程中都在一起。分布产生的循环在给定层次标记上“可并行化”和“可向量化”等不同属性。所用的Tajan算法在求“parallelizable”时，忽略了所携带的控制边。

（3）第一遍的代码生成用于找出所有可能的向量和并行循环。这一信息如下收集：首先从图中暂时删除所有可删除的边，调用*codegen*算法，不在任一步生成并行或向量代码，而是仅记录在该层的循环能否以向量或并行方式执行。这一遍基本上完成所有可能的依赖环消除变换的工作，但是仅在图中实现它们，而不是在源代码中实现。 313

（4）在确定了所有可能的并行循环和向量循环以后，代码生成器对这些循环评分以确定最佳的向量循环和并行循环。考虑的因素包括跨距为1的访问、分散-收集的不足、循环长度以及全局一致性。

（5）代码生成器最终调用一遍*codegen*过程，主要针对最优的向量循环和并行循环。当这一遍*codegen*遇到一个还没有向量化或者并行化的最优向量循环或并行循环时，就知道需要作某些打破依赖环的变换，并作出相应的选择。类似地，在这个方案中合并类似区域也是简单的。

⊖ 当然，这是因为没有考虑到机器生成代码的可能性，这违反了所有正常的需求，对于任何编译器编写人员来说都是一件糟糕的事情。但是，总是有些程序员会违反合理的编程规范，这也是事实。

图6-35是Ardent优化器使用的代码生成策略高层算法概要。

```
procedure generate(T)
  // T是需要并行化的中间表示
  为T计算依赖图D，使得所有可以通过某种方式的变换消除的边e的属性deletable(e)为真
  通过删除所有deletable(e)为真的边e，从D计算D_d
  mark(D_d, S); // 把所有可以向量化或者并行化的循环和语句存储到S中
  score(D_d, S); // 将每个语句存入最优的向量和并行循环
  codegen(D, S, T); // 实际生成代码
end generate
```

图6-35 Ardent Titan代码生成概览

作为一个说明该策略有效性的简单例子，考虑矩阵乘法。在线性代数教科书中的标准代码把求和归约循环放在最内层的位置，如下所示：

```
DO I = 1, 512
   DO J =1, 512
      C(I,J) = 0.0
      DO K = 1, N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
      ENDDO
   ENDDO
ENDDO
```

314

其他方面都一样，Ardent编译器会选择I-循环，因为它保证向量语句的跨距为1，这样在Titan上可能有最好的存/取性能。因为Titan的向量寄存器的长度为32，它把循环分段长度设为32并把它移动到最内层的位置。另一完全并行的循环是J-循环，它可以被移动到最外层位置。结果代码与Titan体系结构匹配得极好：

```
PARALLEL DO J = 1, 512
DO I =1, 512, 32
  C(I:I+31,J) = 0.0
  DO K = 1, N
    C(I:I+31,J) = C(I:I+31,J) + A(I:I+31,K)*B(K,J)
  ENDDO
ENDDO
ENDDO
```

6.9 历史评述与参考文献

标量私有化是Cytron和Ferrante [95]以及Allen等[14]在PTRAN自动并行化项目中提出的。数组私有化的必要性是由Eigenmann等[108]在对Perfect基准测试程序的研究中确认的，而Li[199]描述了进行优化的方法。对齐和分布最早是由Allen，Callahan和Kennedy[25]提出。

循环交换方法已被广泛地使用。Wolfe为此引入了方向向量[280]。Allen和Kennedy对这个问题进行了进一步的研究[19]。基于方向矩阵的形式化表述是新的，但是可以被看作么模变换的一个变形，如Wolf和Lam [277]所作的讨论。循环倾斜是Lamport[195]提出的波前方法的一个变形。这个变换本身是由Wolfe提出的[281]。

数组操作和流操作优化的循环合并有很长的历史[124，118，163]。Allen，Callahan和

Kennedy [25]最早讨论了将循环分布和合并结合起来的方法。带类型的合并算法和它在单层多循环并行化的应用由Kennedy和McKinley [176]提出。Kennedy和McKinley [176]也描述了6.3.6节所提到的在内存性能和并行性之间找到平衡点的策略。

Cytron在Cedar项目中[93，94]提出了DOACROSS的概念和流水线并行性的处理方法。为多处理器调度设计的面包店计数器算法是一个在操作系统文献中经常引用的古老的策略。制导 315
的自调度则是Polychronopoulos和Kuck [226，225]提出的。

在依赖和并行化的一般性处理方面的一些出色的工作可见Allen等[14]、Amarasinghe [28]等、Kuck等[190]以及Wolfe等[283,285]。

习题

6.1 为下面的循环嵌套构造方向矩阵，并说明如何并行化这些循环以达到最大的粒度。描述并行化这些循环嵌套需要的变换。

a.
```
DO J = 1, M
   DO I = 1, N
      A(I+1,J+1) = A(I,J+1) + C
      B(I+1,J+1) = B(I,J+1) + A(I+1,J) + D
   ENDDO
ENDDO
```

b.
```
DO K = 1, L
   DO J = 1, M
      DO I = 1, N
         A(I+1,J+1,K+1) = A(I,J+1,K+1) + A(I+1,J-1,K) + A(I+1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

c.
```
DO K = 1, L
   DO J = 1, M
      DO I = 1, N
         A(I+1,J+1,K+1) = A(I,J+1,K+1) + A(I+1,5,K)
      ENDDO
   ENDDO
ENDDO
```

6.2 证明下列问题是NP完全问题：给定一个有N个循环和M（$M>N$）个依赖的循环嵌套的方向矩阵，找出能够覆盖所有剩余循环的最小个数的循环。

6.3 说明如何利用图6-20的选择启发式方法把循环反转并入到图6-19给出的例程*ParallelizeNest*中去。 316

6.4 开发例程*Parallelize*的一种版本；同时尝试全循环并行化（没有结束变换）和外层循环分布（如果全循环并行化不能生成一个并行的外层循环），然后根据某种评价标准从中选择最佳的一个。评价标准不需要写出。

6.5 适应性调度对计算量进行划分以得到负载平衡。一个对立的方向是划分数据。例如，一个并行的Web服务器接收对大量数据文件的请求。你能否扩展适应性调度的思想，把它用于在并行服务器之间划分数据？ 317

第7章

处理控制流

7.1 引言

在本书前面几章，我们主要关注的是针对除循环外，不含其他控制流的程序变换。换句话说，我们忽略了由条件分支引发的问题。这些问题非常难处理，因为控制流引入对程序变换的一类新的限制，这类问题不是数据依赖所能处理的。

我们以下面的循环为例，说明由控制流引入的执行限制：

```
    DO 100 I = 1, N
S1      IF (A(I-1) .GT. 0.0) GOTO 100
S2          A(I) = A(I) + B(I) * C
100 CONTINUE
```

如果我们只考虑数据依赖，那么可以看到，由于对A的引用，S_1对S_2有一个循环携带依赖，除此之外没有其他的数据依赖。因此，单纯的数据依赖分析将显示这两个语句可以被向量化。但是，由于我们不知道如何向量化一个带有GOTO的IF语句，所以我们只能仍然以循环的方式来实现这个语句，结果是

```
S2  A(1:N) = A(1:N) + B(1:N) * C
    DO 100 I = 1, N
S1      IF (A(I -1) .GT. 0.0) GOTO 100
100 CONTINUE
```


这里，由于原代码中从S_2到S_1的依赖，带IF语句的循环只能在向量语句之后出现。

容易看出这段代码是不正确的。在原来的代码中，语句S_2只有在A(I-1)比0大的循环迭代中才执行；在转换后的版本中，这个语句将被无条件地执行。很明显，我们忽略了阻止我们交换S_1和S_2顺序的某种限制。如何才能避免这样错误的变换呢？根据定理2.8，只要语句不是依赖图中存在的依赖环的一部分，这些语句就可以被向量化。所以，前面的错误变换一定是由于我们忽略了一个从S_1到S_2的依赖。这个依赖是一个*控制依赖*，是由于S_2的执行与否依赖于S_1的输出而导致的。

在本章中，我们将逐步介绍控制依赖的概念，并且将前面几章中的结论扩展到包含条件控制的程序。本章还将介绍计算和处理控制依赖的方法。

处理控制依赖主要有两种基本的策略：其一是通过把控制依赖转换成数据依赖来消除控制依赖。这是一种在自动向量化的过程中经常应用的策略，被称为*if转换*，我们将在下一节详细介绍。

第二种策略是将控制依赖作为数据依赖的扩展来处理，并且在依赖图中加入控制依赖边。由于这种策略可以产生较为简单的代码，所以它适合运用在自动并行化中，我们将在7.3节中介绍这种策略。尽管这种策略企图避免做系统的if转换，但是，为了产生正确的代码，我们仍然可以看到有些情况下需要类似于if转换的变换。

7.2 if转换

如果将引言中描述的问题改写为如下不带GOTO的形式，这个问题就比较容易解决了。

```
DO I = 1, N
   IF ( A( I - 1).LE.0.0 ) A( I ) = A( I ) + B( I ) * C
ENDDO
```

在这种形式中，条件控制可以被看成是这个语句的另一个输入，这样依赖环就很明显了。

下面我们看一个稍微复杂一些的例子:

```
    DO 100 I = 1, N
S1      IF (A(I-1).GT.0.0) GOTO 100
S2          A(I) = A(I) + B(I) * C
S3          B(I) = B(I) + A(I)
320 100 CONTINUE
```

在这样的形式下，又一次很难看出我们是否可以对S_2和S_3进行向量化。

但是，上面的例子如果转化成条件赋值语句，

```
    DO 100 I = 1, N
S2      IF (A(I-1).LE.0.0) A(I) = A(I) + B(I) * C
S3      IF (A(I-1).LE.0.0) B(I) = B(I) + A(I)
100 CONTINUE
```

数据依赖分析就可以告诉我们第二个语句可以被向量化。用Fortran 90中的WHERE语句将上面的例子以向量化的形式重写，得到

```
    DO 100 I = 1, N
S2      IF (A(I-1).LE.0.0) A(I) = A(I) + B(I) * C
100 CONTINUE
S3      WHERE (A(0:N-1).LE.0.0) B(1:N) = B(1:N) + A(1:N)
```

对于这样一种简单清楚的从分支到条件执行的变换，一个很自然的疑问是：是否这种变换可以被普遍地运用于所有形式的语句和程序结构。换句话说，是否通过将语句转化成有条件控制的形式（其中控制语句执行的条件可以看作是语句的输入），就能将控制依赖转化成数据依赖？从本章将会看到，答案是肯定的。if转换可以被广泛地运用在向量化编译器中。if转换可以把所有的控制依赖转换成数据依赖从而把所有依赖统一成为一种形式，理论上它是一种处理控制依赖的完美的方法。

7.2.1 定义

7.1节用一个简单的例子介绍了if转换的概念。简单地讲，if转换是一个删除程序中所有分支的过程。当然，要保证程序的正确执行，分支不能被简单地删除，而是需要用其他成分代替它。if转换引入一个控制执行的概念，就是说每个语句都隐式地包含一个逻辑表达式来控制它的执行。只有当它的控制表达式的值为真时，语句才会被执行。

为了说明控制执行这个概念，我们再来考虑本章的第一个例子中循环内的部分:

```
S1  IF (A(I-1).GT.0.0) GOTO 100
S2      A(I) = A(I) +B(I) *C
321 100 CONTINUE
```

把其中的分支去掉，用控制执行的形式来改写这段代码，结果是

```
S1   M = A( I-1) .GT.0.0
S2   IF (.NOT.M) A(I) = A(I) + B(I) *C
100 CONTINUE
```

这里第二条语句上的控制条件（.NOT.M）是非常自然地使用Fortran中的IF语句来表示的。如果没有控制条件显式地出现，那么我们假设控制条件为真，语句总是执行。

if转换的目标是删除程序中的所有分支，代之以等价的控制语句的集合。有了Fortran 90中的WHERE语句，if转换在向量化中就有明显的作用，因为在数据依赖允许的条件下，控制执行可以很自然地翻译成WHERE语句。

尽管在上面的例子中，if转换是一个很简单的过程，但是某些情况下它是很复杂的。为了把这个过程讲述清楚，这里有必要先将分支归为不同的类型。

7.2.2 分支的分类

为了便于分析，分支可以分为三种类型：

（1）前向分支：把控制转向同一循环嵌套层中本分支语句之后的位置。

（2）后向分支：把控制转向同一循环嵌套层中按词法序在本分支语句之前的位置。

（3）出口分支：结束一个或多个循环，将控制转向循环嵌套之外的位置。

本章中第一个例子表示一个前向分支。后向分支和前向分支类似，但后向分支会引入一个隐式的循环，这一点与前向分支不同。后向分支经常在Fortran 66和Fortran 77中被用于编写while循环，如

```
10 I = NEXT (I)
   A( I) = A(I) + B(I)
   IF ( I.LT.1000) GOTO 10
```

322

出口分支的确定性质是跳出循环，这同时也是决定如何删除出口分支的性质。跳出循环的方向（向前跳转还是向后跳转）是不重要的。出口分支经常被用于类似于如下的“搜索循环”中：

```
      DO I = 1, N
         IF (ABS(A(I) - B(I)).LE.DEL) GOTO  200
      ENDDO
      ...
200 CONTINUE
```

在这个“搜索循环”中，在数组A和数组B中的一对元素在数值上足够接近时，就发生跳出。

这个分类方法中没有包含跳转到循环内的分支。这种现象在Fortran中是非法的，但在C和其他语言中可能出现。本书不处理这类分支（完整的处理参见Allen等的论文[23]）。

if转换实际上是两种不同变换的组合：

（1）分支重定位将分支移出循环，直到分支语句和它的目标位置处在同样的DO嵌套循环中——这个过程是把出口分支转换成前向分支或后向分支。

（2）分支删除通过计算分支控制范围内的操作语句的控制表达式，并依据这些控制表达式的条件执行，删除前向分支。

下面几小节将详细介绍这两种技术。

7.2.3 前向分支

if转换中最简单的变换是分支删除，该变换通过插入相应的控制表达式删除循环内的前向

分支。分支删除的基本思想是遍历程序，维护一个布尔表达式，表示只有这个逻辑条件必须为真时，当前语句方能执行。遇到一个新的分支时，把这个分支的控制表达式结合到当前的逻辑控制条件当中。当遇到一个分支的目标，则把这个分支的控制表达式从当前的逻辑控制条件中分离。

前面的例子说明了非常简单的分支删除，只涉及到一个GOTO。一般来说，分支删除可以非常复杂，如下例所示：

```
      DO I = 1, N
C1      IF (A(I).GT.10) GOTO 60
20        A(I) = A(I) +10
C2        IF (B(I).GT.10) GOTO 80
40          B(I) = B(I) +10
60    A(I) = B(I) +A(I)
80    B(I) = A(I) -5
323   ENDDO
```

为了在这个循环中执行分支删除，确定对每条赋值语句有效的控制表达式是必需的。很明显，当且仅当C_1中的条件为假时，语句20才会执行。同样，当且仅当C_1和C_2的条件都为假时，语句40才会执行。语句60和80比较复杂。语句60可以在C_1和C_2都为假时直接达到，也可以在C_1为真时由语句C_1跳转到。语句80可以在执行完语句60后达到（这种情况下，它和语句60由同样的条件控制），也可以直接从C_2跳转到（C_1为假而C_2为真）。正确的分支删除应产生下面的代码：

```
      DO I = 1, N
        m1 = A(I) .GT.10
20      IF (.NOT.m1) A(I) = A(I) +10
        IF(.NOT.m1) m2 = B(I) .GT.10
40      IF(.NOT.m1 .AND. .NOT.m2) B(I) = B(I) + 10
60      IF(.NOT.m1 .AND. .NOT.m2 .OR.m1) A(I) = B(I) + A(I)
80      IF(.NOT.m1 .AND. .NOT.m2 .OR.m1 .OR. .NOT.m1.AND.m2)
          B(I) = A(I) -5
      ENDDO
```

表面上看，循环结束的控制条件显得复杂。但是，当这个条件从符号上简化后，控制流变得比较清晰和简单了。语句80事实上被无条件执行，语句60当第一个分支跳转或第二个分支不跳转时执行。简化后的循环变为下面的形式：

```
      DO I = 1, N
        m1 = A(I).GT.10
20      IF (.NOT.m1)  A(I) = A(I) +10
        IF(.NOT.m1)   m2 = B(I).GT.10
40      IF(.NOT.m1.AND..NOT.m2)  B(I) = B(I) + 10
60      IF(m1.OR..NOT.m2) A(I) = B(I) + A(I)
80      B(I) = A(I) -5
      ENDDO
```

在计算数据依赖和进行标量扩展（见5.3节）后，这种形式的循环向量化就很简单了，其结果可以用Fortran 90中的WHERE语句表示如下：

```
      m1(1:N) = A(1:N).GT.10
20      WHERE (.NOT.m1(1:N)) A(1:N) = A(1:N) +10
```

```
      WHERE(.NOT.m1(1:N)) m2(1:N) = B(1:N).GT.10
40    WHERE(.NOT.m1(1:N).AND..NOT.m2(1:N)) B(1:N) = B(1:N) + 10
60    WHERE(m1(1:N).OR..NOT.m2(1:N)) A(1:N) = B(1:N) + A(1:N)
80    B(1:N) = A(1:N) -5
```

324

这个例子中有两点需要说明。第一，仔细观察转换后的循环可以发现变量m2未被初始化值可能在语句40和60中引用。事实上，这一点是真的；但是，这些未被初始化的值不会导致任何不利的结果。在任一个特定的迭代中，m2在m1为真时不会被赋值。语句40只有当m1为假而m2为真时才会被执行。由于m1为假会迫使m2被正确地赋值，所以，语句40不可能出现由于m2未被初始化而导致的错误执行。同样，语句60只有在m1为真或m2为假时执行。如果m1为真，m2不被赋值，但是这时不管m2的值如何，这个语句都将被执行。如果m1为假，从而由m2单独控制这个语句的执行时，m2被正确地初始化。

第二点涉及到布尔控制表达式的复杂度。很明显，一些形式的布尔简化对于最后得到理想的代码是十分重要的。有关简化的问题将在7.2.7节讨论。

图7-1给出前向分支删除的一个更为形式化的说明。其中假定IF-THEN-ELSE的删除是很容易的，不予讨论，而且假设只有有标号的语句才能由分支语句达到。在算法记号中，控制条件是Fortran 90中用的字符串。这是为了让读者记住，通过生成化简后的文本代码，条件将被用于在编译时修改程序。

```
procedure remove_branches(B)
// 输入: 一个循环B，循环体内有一系列的语句
// 输出: 一个没有条件转移的循环

current := ".TRUE.";
for each B中带标号语句S do cond(S) := ".FALSE.";

for each B中带标号语句S按顺序do begin
  if S是带标号的
  then current := Simplify (current || ".OR." || cond(S));
  if S 是条件转移: IF(C) GO TO L
  then begin
    设S_L是带标号L的语句;
    设m是编译器产生的新的逻辑变量，初始化为true;
    转换为 "IF(current) m:= C";
    cond(S_L) := cond(S_L) || ".OR." || current || ".AND.m");
    current := current || ".AND..NOT.m" ;
  end
  else if S的形式是: IF(C)S, 其中S不是转移
    then转换为 "IF(C.AND.current) S";
  else
    转换为 "IF(current) S" ;
end remove_branches
```

图7-1 前向分支删除

正确性 我们现在来讨论分支删除的正确性。回忆在2.2.3节中，我们定义了转换后的程序等价于原程序的条件是：（1）在输出点，两个程序的输出变量的值相同；（2）输出语句以

相同的顺序执行；（3）转换后的程序不能引入在原程序中不出现的错误异常。在此定义下，我们必须证实以下三个事实来证明转换的正确性：

（1）新程序的语句实例的控制表达式为真，当且仅当相应的语句在原程序中会被执行，除非新语句的引入是为了计算控制条件的值，这些语句必须在原程序中计算条件表达式的点被执行。

（2）新程序中控制条件为真的语句执行顺序应同原程序中那些语句执行的顺序一样。

（3）计算有副作用的表达式的次数在新旧程序中应相同。

上述的第一个条件可以通过语句顺序方面的归纳来证实。我们的目标是证实每条语句上的控制条件确实是使原程序中语句执行的条件集合。在处理第一句时，空条件集合显然是正确的。假设前面的所有语句的控制条件都正确的。我们必须证明当前语句的控制条件是正确的。当前语句的控制条件是由与它关联的两部分析取构成的，一部分是通过前面的分支跳转到达这个语句的条件$cond(S_L)$，另一部分是由它前一条语句传递的控制条件。这说明控制可以通过分支到达一个语句，也可以通过这条语句的前一条语句直接到达这个语句。

由分支到达语句的条件一定是正确的，因为每个这样的条件是由那个分支的当前条件（按照归纳这个条件是正确的）和这样一个变量的合取得到的，该变量用于保存控制这个分支的条件表达式的值。所以，所有这些条件的析取$cond(S_L)$为真，当且仅当某个控制条件为真的分支语句跳转到S_L。

这样，我们就只需证明直通下来的这部分条件是正确的。按照归纳在前一语句当前条件是正确的。因此，当那个语句不是一个条件分支语句时，当前语句的条件和前面语句的条件是相同的，也必然是正确的。对于条件分支语句，直通下来的当前条件是如下两个条件的合取：当前条件和保存控制分支的条件的变量的“非”。这样，由上一语句传到当前语句的当前条件是上一条语句的当前条件和一个变量的“非”的合取，该变量保存控制分支的表达式的值。因此，当前语句的控制条件为真，当且仅当前一条语句被执行并且分支跳转不被执行。这显然是正确的。

325
~
326

以上第二个条件显然是为真，因为我们没有重排语句的顺序。第三个条件可以由第一个条件和以下这个观察结果证实：控制每个条件分支的条件只在分支将被定位的点计算一次。所有其他语句都准确地和原程序中的相应语句在相同迭代中的相同位置被执行。

请注意，无条件的前向分支语句也可以用上面的算法处理，只需把它们看作是条件为永真的条件分支。

7.2.4 出口分支

出口分支的消除比前向分支更加复杂。原因是前向分支只影响分支之后的语句的控制条件，而出口分支则对它之前和它之后的语句都有影响。考查下面的简单程序段：

```
    DO I = 1, N
      S1
      IF (p(I)) GOTO 100
      S2
    ENDDO
100 S3
```

如果p(I)为真的第一个循环迭代为I_0，那么整个循环将在这一点停止执行。当这个循环转化为控制执行的形式后，这个DO循环将执行包括I_0和其后的所有迭代，因为它不能通过分支来早结束这个循环。在目前的形式下，S_2将在迭代I_0到N不被执行，S_1在迭代i+1到N不被执

行。出口分支影响整个循环中所有语句的事实使它们比简单的前向分支更加难被消除。

由于出口分支最复杂的特性是它们将会使循环结束，消除这个特性就是将它们转化成控制执行的形式的关键问题。如果分支能被重定位在它能结束的所有循环之外，此时得到的分支就是一个简单的前向分支。为了说明如何实现这个重定位，看如下的例子: 327

```
      DO J = 1, M
         DO I = 1, N
            A(I,J) = B(I, J) + X
S           IF (L(I,J)) GOTO  200
            C(I,J) = A(I,J) +Y
         ENDDO
         D(J) = A(N,J)
200      F(J) = C(10, J)
      ENDDO
```

分支语句S跳出一个循环。如果，可以把S从一个出口分支转化成一个前向分支，那么，一定会有一个变换产生和如下代码类似的结果:

```
      DO J = 1, M
         DO I = 1, N
            IF (C1)  A(I,J) = B(I, J) + X
Sa          Code to set C1 and C2
            IF (C2)  C(I,J) = A(I,J) +Y
         ENDDO
Sb       IF (.NOT.C1.OR..NOT.C2) GOTO 200
         D(J) = A(N,J)
200      F(J) = C(10, J)
      ENDDO
```

在这种形式下，原来的出口分支已经被移出循环外，变成了语句S_b。这是一个简单的前向分支，可以用图7-1中的算法来删除。于是，这个问题就转化为寻找合适的条件去控制内层循环中的语句。

简单地说，内层循环中的语句只有在过去的迭代中出口分支没有跳出的情况下才会执行。换句话说，一旦一个迭代执行，其中控制出口分支的条件为真，后面的迭代就不会再执行了。这样，我们需要迭代地计算一个变量，即一旦它在任何一个出口分支条件成立的迭代中被置为假，则它以后就一直为假。这样的变量是循环中所有其他语句的正确的控制条件。计算这样一个变量并不难，它是所有前面迭代中出口分支的控制条件取非的“合取”:

$$lm = \bigcap_{k=1}^{I} \neg L(k, J) \tag{7-1}$$

在前面的例子中应用这个变换，得到下面的代码: 328

```
DO J = 1, M
   lm = .TRUE.
   DO I = 1, N
      IF (lm) A(I,J) = B(I,J) +X
      IF (lm) m1=.NOT. L(I,J)
      lm = lm .AND. m1
      IF (lm) C(I,J) = A(I, J) + Y
   ENDDO
```

```
        m2 = lm
        IF (m2) D(J) = A(N,J)
200     F(J) = C(10, J)
      ENDDO
```

注意变量m1的引入是为了计算条件L(I,K)，这个条件的计算可能有副作用，所以这里保证了只有当原程序执行这个计算时，变换后的程序中才会执行。向前替换m2并把lm标量扩展成为二维数组，结果是

```
      DO J = 1, M
        lm(0,J) = .TRUE.
        DO I = 1, N
          IF (lm(I-1,J)) A(I,J) = B(I,J) +X
          IF (lm(I-1,J)) m1=.NOT. L(I,J)
          lm(I,J) = lm(I-1,J) .AND. m1
          IF (lm(I,J)) C(I,J) = A(I, J) + Y
        ENDDO
        IF (lm(N,J)) D(J) = A(N,J)
200     F(J) = C(10, J)
      ENDDO
```

当对这段代码应用*codegen*之后，就得到下面向量化的循环:

```
      DO J = 1, M
        lm(0,J) = .TRUE.
        DO I = 1, N
          IF (lm(I-1,J)) m1=.NOT. L(I,J)
          lm(I,J) = lm(I-1,J) .AND. m1
        ENDDO
      ENDDO
      WHERE (lm(0:N-1,1:M)) A(1:N,1:M) = B(1:N,1:M) +X
      WHERE (lm(0:N-1,1:M)) C(1:N,1:M) = A(1:N, 1:M) + Y
      WHERE (lm(N,1:M)) D(1:M) = A(N,1:M)
329   200 F(1:M) = C(10, 1:M)
```

这样尽可能地实施向量化，产生4个向量化循环，看上去可能是非常有效的。但是，这不是最有效的方案。更好的方法是只扩展内层循环中的标量lm，而不是在两层循环中都扩展它:

```
      DO J = 1, M
        lm(0) = .TRUE.
        DO I = 1, N
          IF (lm(I-1)) A(I,J) = B(I,J) +X
          IF (lm(I-1)) m1=.NOT. L(I,J)
          lm(I) = lm(I-1) .AND. m1
          IF (lm(I)) C(I,J) = A(I,J) + Y
        ENDDO
        IF (lm(N)) D(J) = A(N,J)
200     F(J) = C(10, J)
      ENDDO
```

向量化后变为

```
      DO J = 1, M
        lm(0) = .TRUE.
```

```
      DO I = 1, N
         IF (lm(I-1)) m1=.NOT. L(I,J)
         lm(I) = lm(I-1) .AND. m1
      ENDDO
      WHERE (lm(0:N-1)) A(1:N,J) = B(1:N,J) +X
      WHERE (lm(1:N)) C(1:N,J) = A(1:N,J) + Y
      IF (lm(N)) D(J) = A(N,J)
200   F(J) = C(10, J)
   ENDDO
```

聪明的归约识别算法可以识别出保留的I-循环实际上是个“FirstTrue”归约，所谓“FirstTrue”归约是可以找到一个向量中第一个为真的分量的索引的函数。在一个有适当硬件支持的机器上（这个硬件非常简单，容易被加入），相应的代码为:

```
      DO J = 1, M
         n = FirstTrueX(L(1:N,J)) -1
         A(1:n+1,J) = B (1:n+1,J) +X
         C(1:n, J) = A(1:n,J) + Y
         IF (n. GT.N) D(J) = A(N,J)
200   F(J) = C(10,J)
   ENDDO
```

330

其中内置向量函数FirstTrueX返回逻辑参数数组中的第一个为 · TRUE · 值的索引，当其中没有为真的值时，返回参数数组的长度加1。在大多数向量机上，这样的代码比二维向量化代码更为有效。原因是，二维形式的代码工作在变长的向量上，也就是说WHERE语句在整个I-循环上操作，但事实上这个向量可以在循环上被安排得很密集（即无条件地执行），直到出口分支被执行时完全跳出循环，从而完全跳过后面的部分。

和前向分支删除一样，分支重定位的基本过程是对于要进行出口分支消除的循环，精确地计算控制循环内每条语句执行的布尔控制表达式。这个算法在图7-2中给出。循环中出口分支条件x的计算仍然是不带控制表达式的，这样是为了使它被扩展成一个向量或一个数组时，仍然能像上面的例子一样得到正确的结果。这样不会导致含义上的差异，因为语句本身被设计成无副作用的。

```
procedure relocate_branches(l)

    // l是循环中的DO语句
    // lg是用来控制循环中每条语句的循环控制表达式
S1: for each循环l中的DO语句d do
      relocate_branches(d);
    设lg=null;

S2: for each出口分支 "IF(p) GO TO S" do begin
      建立一个新循环出口标记x;
      在l之前插入赋值 "x=.TRUE." ;
      if lg=null then lg:= "x" ; else lg:=lg|| ".AND.x";
      在循环后插入分支 "IF (.NOT.x) GO TO S" ;
      产生一个新的布尔变量m并用一对语句 "m=.NOT.p" 后跟 "x=x.AND.m" 代替出口分支
```

图7-2 分支重定位算法

```
        end
        for each循环l中非-DO语句s do
          if guard(s) = null begin
            if s中赋值的变量不是lg中的标记
            then guard(s) := lg;
          end
          else guard(s) := guard(s).AND.lg;
    end relocate_branches
```

331

图 7-2（续）

过程*relocate_branches*消除它处理的循环和包含在这个循环内的任何循环中的出口分支。算法中在S_1处的递归调用是删除内部循环中的出口分支的关键，这个过程把出口分支从包含的循环中删除。这个递归调用必须在循环体的语句S_2开始之前执行。原因是一个跳出多个循环的内层循环分支也可能跳出循环*l*，因此，必须在以S_2开始的循环体中予以删除。

正确性 图7-2中的*relocate_branches*算法必须在以下情形才是有用的：（1）对一个循环应用这个算法不会改变程序的含义，就是说这个算法不会重排序、添加或删除语句实例，（2）这个算法消除它处理的循环中的所有出口分支。

首先，我们证明算法*relocate_branches*删除所有的出口分支，这是比较简单的情形。按定义，语句S_2处的循环可以找到输入循环中的所有出口分支，并用赋值语句代替这些分支。算法在所检查的循环外构造新的分支，所以它们不可能是那个循环的出口分支。通过简单的归纳可得，对*relocate_branches*的递归调用删除内层循环中的所有出口分支。结果，通过对*relocate_branches*的调用，被检查的循环内部的所有出口分支都被删除了。

证明程序的含义不变比较复杂一些。由于没有在循环中添加或删除语句，证明算法的正确性需要证明循环内的语句以原顺序执行，直到出口分支的条件为真时，且此后没有再执行下去。在任何出口分支条件变为真的迭代中，与此分支相关的控制变量x都会在原出口分支所在的精确位置被置为假。由于循环控制表达式是循环中各个出口分支的出口控制变量的合取。所以，只要任何一个出口控制变量为假，循环控制变量都会立即变为假，并且将一直保持为假。又由于循环控制表达式被加入到循环体中每个语句的控制表达式中，所以，每个语句的控制表达式也会同时变为假，故循环中没有语句会被执行。由于循环中语句的执行顺序没有变，并且计算出口分支条件的点也与原程序中计算这些条件的点相同，所以，变换后的程序与原程序执行同样的语句且保持语句的执行顺序。

留下的问题是循环结尾处的条件分支是否正确？请注意，只有一个出口分支控制变量会被置成假，因为一旦一个分支控制变量被置成假，不可能有其他控制变量被置为假，由于此时的循环控制表达式已经是假了。所以，紧接在循环后的条件分支中只有一个会被执行。如果这些条件都为假，将开始执行原程序中循环后的第一条语句。这一点完备了正确性证明的

332

论证。

这个正确性的论证不能用于控制所跳出的循环的归纳变量的值；如果从出口分支跳出，循环归纳变量在经过算法*relocate_branches*之后将有一个不同的值。为了能正确处理这种情况（由于出口分支经常用于结束搜索循环，此时循环变量保存了搜索结果，所以正确处理这种情况是很必要的），可以用编译器产生的变量i来代替循环控制表达式中的循环归纳变量I。这样，原来用来控制循环控制条件的语句

```
IF (lg) x =.NOT.p
```

可以替换为条件语句块

```
IF (lg) THEN
   x =.NOT.p
   I = i
ENDIF
```

所以循环归纳变量最后的值被保存下来了。同时，这要求在所有被移动的分支由*relocate_branches*插入紧接循环的位置之后，也保存这个值。

7.2.5 后向分支

到现在为止，已经介绍了两类前向分支的处理方法（包括跳出循环和不跳出循环）。剩下的一种可能性（跳入循环）在Fortran 的标准中是禁止的。因此，可以认为7.2.4节完整地介绍了所有前向分支的处理方法，接下来需要介绍剩下的一种分支——后向分支的处理方法。但是，后向分支会使前向分支的处理变得复杂，因为它们会造成前向分支中没有介绍的那种可能性——跳入循环的分支。总体上讲，后向分支的复杂至少是由于以下两个原因:

（1）它们造成了隐式的循环。后向控制流是循环的基本要素，循环不能用一个简单的控制条件（像前向分支那样）来模拟。消除后向分支需要其他的能代表后向控制流的机制。

（2）由于它们造成前向分支可能跳入的循环，使前向分支的消除变得复杂。

下面的例子显示这种复杂性: 333

```
      IF (P) GOTO 200
      ...
100   S1
      ...
200   S2
      ...
      IF (Q) GOTO 100
```

应用图7-1所示的前向if转换算法后，产生

```
      m1 = .NOT. P
      ...
100   IF (m1) S1
      ...
200   S2
      ...
      IF (Q) GOTO 100
```

如果在跳转到200的前向分支后面接一个跳转到100的后向分支，则控制变量m1的值将为.FALSE.，因此在再次到达200之前的语句不会被执行。但事实上，这些语句在原程序中是应当被执行的。

由于这些复杂性，后向分支不能被隔离出来简单地处理。相反，它们应当和那些涉及到的前向分支合并起来同时处理。最简单的处理方案是将后向分支隔离出来，让这些后向分支控制之下的语句（称为*隐式迭代区域*）保持不变。这种方案引入了一个较为严重的限制，使隐式迭带区域中的前向分支不能被删除。所以，这种方案过于简单，对于面向Fortran 77的产品编译器来说，是不实用的。

Allen等人在1983年发表的一篇论文[23]示范了如何通过if转换的一种变形，使其能消除Fortran程序中所有后向分支。虽然这个过程的细节超出了本书范围，我们在这里也介绍这个过程的基本思想。

前一个例子中的关键问题在于语句S_1的控制条件。为了保证转换的正确，这个控制条件必须反映以下两个事实:

(1) 在这段代码第一次执行时，S_1只有当P为假时才执行;

(2) 如果后向分支被执行，则S_1总是被执行。

因为代码的第一遍执行是挑选出来的，所以，很明显需要一组条件来控制一个隐式迭代区域的第一遍执行，并且需要另一组条件来控制其后的各遍执行。解决这个问题的一个办法是引入一个后向分支控制条件bb，当与它有关的后项分支被执行时，它的值为真。我们用前面的
334 例子来说明这个方法:

```
      m = p
      ...
      bb = .FALSE.
100   IF (.NOT.m.OR(m.AND.bb)) S1
      ...
200   S2
      ...
      IF (Q) THEN
         bb = .TRUE.
         GOTO 100
      ENDIF
```

只有当后向分支被执行时，后向分支变量bb才变为真。S_1在以下情况下总是被执行:

(1) 前向分支未被执行，此时m为.FALSE.

(2) 后向分支被执行。

如果前向分支未被执行，无论bb的值如何，第一个条件满足。如果前向分支被执行，则bb和m都为.TRUE.，使控制条件为真。

总体上讲，有两条路径可以从程序的起点到达一个后向分支的目标语句:

(1) 从前面的语句直接执行下来，在这种情况下，它的执行的条件包含在前驱的当前条件中。

(2) 分支跳转绕过该语句，随后通过后向分支又跳回该语句，这种情况下，正确的控制条件是绕过它的分支条件和返回它分支条件的逻辑与。

这样，如果在目标语句y之前的当前条件用cc表示，前向分支条件如果用m表示，后向分析条件用bb表示，则y处的控制条件应为

```
cc.OR.(m1.AND.bb)
```

这个条件与例子中给出的条件是一致的。

Allen等人的论文[23]接下来讨论如何处理跳入后向分支引起的隐式迭代区域的前向分支。并且说明如何将隐式迭代区域转换成Fortran 90的WHILE循环。但是，在现代程序开发实践中，是不鼓励使用GOTO语句的，尤其是因为会造成隐式迭代区域的原因。而且，Fortran 90语言包括WHILE循环，这样就消除对多种隐式迭代结构的需求。由于这些原因，本书下文中假设: 对
335 于编译器而言，程序中控制流结构仅包括显式循环、前向无条件跳转或前向条件分支。这个假设明显地简化了编译器的任务。

7.2.6 完全前向分支消除

现在，我们已经准备好介绍一种完整的前向分支消除算法。这个过程（图7-3给出详细算法）按照输入语句列表中语句的出现顺序处理语句，同时维护一个用于控制当前被处理的语句的当前条件。

```
procedure remove_branches(R, current)
    // R是即将考虑的一组语句
    // current是语句列表开始处的当前条件
    for each R中可能成为分支目标的语句S do
      cond(S) := ".FALSE.";
S1: for each R中循环l do relocate_branches(l);
    for each R中语句S do begin
      if S是分支的目标then
        current := Simplify (current || ".OR." || cond (S));
      case statement_type(S) in
If:     begin // IF (P) GOTO L——前向分支
          令m是编译器生成的新的逻辑变量，初始化为true;
          插入赋值语句 "IF (current) m := P" ;
          令SL表示标号L处的语句;
          cond(SL) := cond(SL) || ".OR." ||current|| ".AND.m" ;
          current := current || ".AND..NOT.m" ;
          删除语句S;
        end
Goto:   begin // GOTO L
          SL表示标号L处的语句;
          cond(SL) := cond(SL)|| ".OR." ||current;
          current := ".FALSE." ;
          删除语句S;
        end
Loop:   begin // DO B ENDDO——一个循环
          remove_branches(B, current);
          // 由于事先执行了relocate_branches，所以循环之后的当前条件和循环之前一样
        end
Other:  begin // 除continue之外的其他语句
          if S的形式是 "IF (P) S1" then
            用 "IF (current.AND.P) S1" 代替这条语句;
          else用 "IF (current) S" 代替这条语句;
        end
      end case
    end
end remove_branches
```

图7-3 完全前向分支删除

处理每条语句时，根据语句的类型，算法将执行如下步骤：

（1）如果当前语句是一个分支语句的目标语句，当前条件必须以简化的形式计算。为了做到这一点，必须把与跳到目标的分支相关的条件和从语句的词典序前驱传递来的当前条件

合并。这个条件的简化算法将在下一小节介绍。然后，这个语句的处理转到下面的第二步。

（2）如果当前语句是除DO，ENDDO或CONTINUE（这几种语句在if转换保留不被控制）外的任何一种，当前条件（由步骤1计算所得，或在当前语句不是分支的目标语句时由词典序前面语句继承下来）应和当前语句的控制条件结合起来成为当前语句的控制条件。如果当前语句没有被控制，则当前条件成为它的控制条件。

（3）如果当前语句是一条DO语句，首先对循环调用*relocate_branches*以消除它可能包含的任何出口分支。这可能会在这些语句应当被当前条件控制的循环之前产生一些语句。此外，也可能会产生一些新的语句插在修改后的循环的语句表中。这个分支删除的过程应该对在分支重定位之后出现的循环体递归地执行。

（4）如果当前语句是一个条件分支语句，算法把当前条件复制两份。编译器产生的与新条件相关的变量连接到一个拷贝的开始，其结果添加到与分支目标语句相关的表中。这个变量的“非”连接到当前条件的另一个拷贝的开始，成为程序下一语句的当前条件。

（5）如果当前语句是一个无条件跳转语句，当前条件添加到跳转目标语句的条件表中。下一语句的当前条件被设置为空（假）。如果下一语句不是分支的目标语句，则它将不会被执行到。

336
~
337

（6）对下一条语句继续执行步骤1。

这个过程的正确性证明，可由图7-1中前向分支删除算法的正确性证明和图7-2中分支重定位算法的正确性证明加上对递归调用的简单归纳得到。算法剩余的部分是简化的过程，这部分在下一小节讨论。

7.2.7 化简

从上面介绍的几个例子中，我们可以看出语句的控制条件的化简是if转换的一个重要方面。仅仅简单地运用条件转换算法的结果是引入复杂的、臃肿的控制条件，而且这些控制条件在运行时将会重复地被计算，所以，在编译时尽可能地简化它们是非常重要的。但可惜的是不难证明，布尔化简是一个NP完全问题。给定一个能将表达式化为最简形式的布尔化简过程，一个一般的布尔表达式将是不可满足的，当且仅当这个表达式化简为“假”。结果，可满足性可归约为化简，并且我们知道可满足性问题是NP完全问题。也容易证明if转换可以产生任意的表达式，所以，没有简便的方法可以简化转换算法的输出结果。

由于布尔化简程序在每个可能的分支的目标语句被处理时都被调用，所以，效率是一个重要的因素。虽然通常的程序中很少有复杂的控制流，但即使是一个很小的程序（尤其是带有Fortran中的赋值GOTO语句时），也可能会有足够深的层次造成指数级的化简时间复杂度。注意，某特定语句的控制表达式的复杂度是与能影响这个语句执行的分支的数目成比例的。

被最广泛运用的化简程序是Quine和McCluskey[230，208]提出的，它包括三个操作步骤：

（1）将原来的逻辑表达式简化为一些小项的“析取”，其中每个小项都是一个包括原表达式中每个逻辑变量的“合取”，逻辑变量在表达式中是其自身或是取“非”的形式。

（2）计算一组*基本蕴含项*，基本蕴含项表示原表达式中小项的“析取”，但是不被其他蕴含项包含。计算蕴含项的策略是合并这样的小项对：小项在每个因式中相同，除非一个变量在一个小项中以取“非”的形式出现，而在另一个小项中以不取“非”的形式出现。

（3）选择基本蕴含项的一个最小合法子集。一个子集“合法”是指覆盖原表达式。也就是说，基本蕴含项合法选择的“析取”为真，当且仅当原布尔表达式为真。

338

为了说明这个过程，我们介绍一个例子，在本节我们将一直使用这个例子：

```
    DO I = 1, N
1      IF (P) GO TO 10
2         A(I) = B(I)
3         IF (Q) GO TO 20
10           C(I) = 0
20     A(I) = A(I) +C(I)
    ENDDO
```

在不用化简的if转换之后，变为

```
    DO I = 1, N
1      m1 = P
2         IF (.NOT.m1)  A(I) = B(I)
3         IF (.NOT.m1)  m2 = Q
10           IF ((.NOT.m2.AND..NOT.m1) .OR.m1) C(I) = 0
20     IF (((.NOT.m2.AND..NOT.m1).OR.m1).OR.m2) A(I) = A(I) + C(I)
    ENDDO
```

首先，我们先对语句20应用Quine-McCluskey过程。第一步将它的控制条件简化为一组小项：

$$(\neg m2 \wedge \neg m1) \vee (m2 \wedge m1) \vee (\neg m2 \wedge m1) \vee (m2 \wedge \neg m1)$$

为了找到基本蕴含项，我们继续简化各项。第二和第三小项 $(m2 \wedge m1) \vee (\neg m2 \wedge m1)$ 可以简化为$m1$，而第一个和最后一个小项 $(\neg m2 \wedge \neg m1) \vee (m2 \wedge \neg m1)$ 可以简化为$\neg m1$。然后，这两个表达式可以简化为一个基本蕴含项.TRUE.，这代表整个表达式。所以，语句20的控制条件为空。运用类似的过程可以将语句10的控制条件简化为一对基本蕴含项：

$$(\neg m2 \wedge \neg m1) \vee (m2 \wedge m1) \vee (\neg m2 \wedge m1) = \neg m2 \vee m1$$

遗憾的是，Quine-McCluskey算法相对于原表达式的大小来说是指数级的，这一点使这个算法不能实际运用在一般的if转换中。但是，我们可以通过重定义符合if转换最低要求的化简问题来消除算法中的指数特性。

如果牺牲掉一些表示上的简化，而把注意力集中在化简的真正原因上，即决定什么时候可以把一个编译器生成的分支变量从当前的控制条件中剔除，就可以得到一个更快的算法。我们仍然用刚才的例子，来说明其中的区别。用Quine-McCluskey算法化简语句10和20的控制表达式，我们得到 339

```
    DO I = 1, N
1      m1 = P
2         IF (.NOT.m1)  A(I) = B(I)
3         IF (.NOT.m1)  m2 =Q
10           IF (.NOT.m2 .OR.m1) C(I) = 0
20     A(I) = A(I) + C(I)
    ENDDO
```

这里，应当清楚语句20的控制条件的化简比语句10的控制条件的化简重要得多，因为化简使我们从语句20的控制条件中消除变量，而对于语句10来说，仅仅是对于同样的变量得到一个较为简单的表达式。如果我们不对语句10化简，我们将得到一个略微复杂的形式：

```
      DO I = 1, N
1        m1 = P
2           IF (.NOT.m1)  A(I) = B(I)
3           IF (.NOT.m1)  m2 =Q
10             IF ((.NOT.m2.AND..NOT.m1).OR.m1) C(I) = 0
20       A(I) = A(I) + C(I)
      ENDDO
```

这个例子的重要之处在于：虽然变量m1的分支目标在语句10达到，但直到达到m2的分支目标后才会把m1从语句的控制条件中化简掉。概括地说，我们只能以与算法引入逻辑变量的顺序的相反顺序把这些逻辑分支变量从表达式中化简掉。也就是说，最新引入的逻辑变量被最先化简掉。所以，m1直到处理到语句20（m2的分支目标）时才能被删除掉。那时，两个变量都会通过化简被删除掉，结果语句是不被控制的。

以上的讨论给出一个使化简程序变得简单的好方法：只关心化简中最近被引入的变量。就是说，化简程序应集中消除最后引入的变量，而不是试图化简给定语句的当前条件中所有的变量。采用这个化简的形式的结果将使上一个例子被化简成最后一个版本；例子中，语句10的控制条件变得更为复杂，这是简化整个化简程序付出的代价。主要的区别在于：简化了的算法只在有可能从控制条件中消除变量从而消除被控制语句的一个输入时，才进行化简。这一点正是支持if转换的化简的关键任务。

有了以上的观察，现在可以给出一个快速化简算法。这个算法将把条件都维护成项的
340 “析取”，每个项代表因子的“合取”，其中一个因子可以表示变量或变量的“非”。在一个项中，因子将以被引入次序的相反顺序出现，因此，最近被引入的因子将在项中第一个出现。最近被引入的因子称为*关键因子*，它是在考虑删除其他任何变量之前，必须从项中化简掉的变量。

图7-4具体地给出这个化简算法，这个算法是假设它将被用在图7-3给出的*remove_branches*过程中。记住过程*remove_branches*是以语句在程序中的出现顺序来处理语句的，并且维护一个*当前条件*，这个当前条件是用于控制当前语句的条件。

这个化简程序只在遇到一个可能是分支目标的语句时被调用。它必须将跳转到这个目标的分支语句的条件集合和从语句词典序的前驱传递的当前条件合并。这些条件被组合成一组项，每一个项包含一个关键因子。如果存在两个项，两个项的关键因子包含同样的变量，那么它们一定符合以下情况：除了一个变量在一个项中以不取“非”的形式出现，而在另一个项中以取“非”的形式出现而外，这两项完全相同。（为了看出这点，回忆两项代表两条不同的路径，路径上最后一个分支是关键因子变量代表的分支。由于这两条路径是不同的，所以这两条路径一定由那个分支转向不同的方向。）如果找到了这样一对项，它们将被一个单独的项所代替，这个用来替换它们的项包含原来两项中剥离关键因子后的部分。换句话说，关键因子已被消除。在新项中的第一个因子成为新的关键因子。这一步将被重复直到没有两项的关键因子包含相同变量为止。这时，保留下来的这组项的“析取”就是新的当前条件。

为了更具体地说明这个算法，让我们再一次考虑前面的例子。在语句1被处理时，当前条件是由空条件表示的.TRUE.。在语句1处理完之后，语句10带有其目标表的条件“m1”，而当前条件是“.NOT.m1”。在语句3处，条件

```
m2.AND..NOT.m1
```

被加在语句20的条件表中，并且

.NOT.m2.AND..NOT.m1

成为新的当前条件。结果，当处理语句10时（调用算法中的语句4），要考虑的两个条件分别是当前条件和目标条件，也就是

".NOT.m2.AND..NOT.m1" and "m1"

341

以这两个条件的"析取"作为新的当前条件。由于其中的关键变量不同，所以不能进一步化简。最后，处理到语句20时，用以下三个项:

"m2.AND..NOT.m1," ".NOT.m2.AND..NOT.m1," and "m1"

前两项的关键因子中有相同的项，合并后的结果为

".NOT.m1" and "m1"

这两项的关键因子中也有相同的变量。结果可化简为.TRUE.，这正是我们想要的结果。注意，这里我们对语句10的条件没有化简。

图7-4给出在前向if转换过程中使用的简化后的化简算法。

```
procedure Simplify(cond)
  // cond是各个项的"或"，每项都有一个关键变量
  // seen是一组曾经被看作关键变量的变量
  // sofar是已经加入输出条件列表的一组项
  // worklist是在当前输出条件中正在处理的一组项

  seen := Ø; sofar := Ø;
  worklist := cond中项的集合;
  while worklist ≠ Ø do begin
    令t是worklist中的任一元素;
    worklist := worklist - {t};
    k是t的关键因子中的变量
    while k ∈ seen do begin
      从sofar中取k的对应项q;
      sofar := sofar - {q};
      t := 从q中剥去关键因子后剩余的项;
      k := t起始处关键因子中的变量;
    end
    sofar := sofar ∪ {t};
    seen := seen ∪ {k};
  end
  return sofar中项的"或";
end Simplify
```

图7-4 化简

342

在每个分支目标处，这个算法的工作量和当前条件中项的数目成比例。为了看清这点，注意最外层循环对当前条件中的每一项执行一次。内层循环的每一步将会消除当前条件的一项，所以，整个算法内层循环的迭代总数是受原当前条件中项的数目限制的。

请注意，当前条件的项的数目大致和程序中通过当前点的控制流路径（在控制流图上被这个点切断的边）的数目成比例。如果我们只对前向分支做if转换，这个数目就以程序中语句数目为界。这样，对于前向分支来说，整个程序的化简的总体代价不会比$O(N^2)$更差，这里N是程序中语句的数目。

7.2.8 迭代依赖

一旦一个程序中运用了条件转换，由操作语句、分支语句和占位符语句造成的所有依赖都会表现为数据依赖。但是这些语句不是造成控制依赖的全部。迭代语句也可以造成控制依赖，如下面的例子所示：

```
20 DO I = 1, 100
40    L = 2 * I
60    DO J = 1, L
80       A(I, J) = 0
      ENDDO
   ENDDO
```

如果我们假设迭代语句不会携带任何控制依赖，这个例子将会被不正确地向量化为：

```
20 DO I = 1, 100
40    L = 2 * I
   ENDDO
80 A(1:100, 1:L) = 0
```

原来例子将结果数组中的一个三角形区域中的元素变为0。向量化后的例子是将一个矩形区域内的元素变为0。向量化后的结果不能复制由外层循环造成的可变的边界。因为我们忽视了依赖图中不出现的一些依赖，所以这个边界的变化也被忽视了。要找到这些依赖就必须注意到这样一个概念：DO语句控制着一些特定语句的执行次数。为了模型化这个概念，我们引入下
343 述概念：循环中的每条语句有一个隐式的迭代范围输入，在例子中我们把它表示在代码中语句的最后，并用括号将它括起来。这样，

```
A(I, J) = 0  (irange2)
```

表示这个语句被迭代范围irange2控制着，irange2是由编译器产生的用来保存第二层循环的迭代范围的标量。

一个给定循环的迭代范围将在原程序中计算迭代范围处被赋值。在Fortran中，循环边界在恰好进入循环之前计算，所以，对迭代范围的赋值将被放在DO语句所控制的循环之外。对于像WHILE循环这样的条件循环来说，控制条件应在通过循环的每次迭代中都被计算；因此，WHILE循环的赋值应放在循环内并和循环内的语句在同一层次上。

运用这种策略，开始这个讨论时所用的例子将被转换成

```
20  irange1 = (1, 100)
    DO I = irange1
40     L = 2 * I (irange1)
60     irange2 = (1,L) (irange1)
       DO J = irange2
80        A(I, J) = 0 (irange2)
       ENDDO
    ENDDO
```

在决定依赖之前，编译器应当运用类似于第4章中介绍的方法前向替换任何常数和循环无关的变量。这样，以上程序将被简化为

```
20  DO I = 1, 100
40     L = 2* I(1,100)
60     DO J = 1, L (1,100)
```

```
80        A(I,J) = 0 (1,L) (1,100)
       ENDDO
    ENDDO
```

为了依赖测试的目的，常数输入将被忽视，这样，图7-5给出上例中的依赖模式。注意其中由第一层循环携带的反依赖的存在是由于循环语句和赋值语句80都隐式地使用内层循环的标量上界。

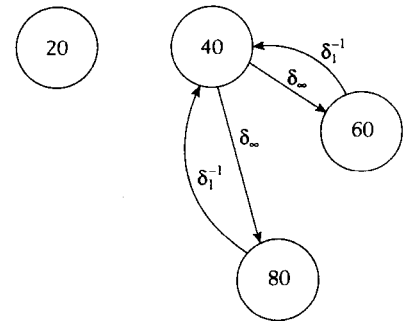


图7-5 迭代依赖例子

标准的向量化过程作用于这个依赖图，将生成代码

```
20  DO I = 1, 100
40     L = 2 * I
80     A(I, 1:L) = 0
    ENDDO
```

其中内层循环由于是空的而被消除。

为了得到最佳的性能，范围的计算应当移到尽可能多的循环之外，并尽可能地作前向替换。我们用下面的例子来说明这个问题:

```
10  READ 5, L, M
    DO I = 1, L
20     N = 2*I
       DO J = 1, M
          DO K = 1, N
30           A(I, J, K) = 0.0
          ENDDO
       ENDDO
    ENDDO
```

转换之后，代码变为

```
10  READ 5, L, M
    irange1 = (1, L)
    DO I = irange1
20     N = 2 * I  (irange1)
       irange2 = (1,M) (irange1)
       DO J = irange2
          irange3 = (1, N) (irange2)
          DO K = irange3
30           A(I, J, K) = 0.0 (irange3)
          ENDDO
       ENDDO
    ENDDO
```

用4.5.1节中给出的方法进行前向替换后，我们得到如下代码:

```
10  READ 5, L, M
    DO I = 1, L
20     N = 2 * I  (1,L)
       DO J = (1,M),(1,L)
          DO K = (1,N) (1,M)(1,L)
30           A(I, J,K) = 0.0 (1,N) (1,M) (1,L)
          ENDDO
```

344
~
345

```
      ENDDO
   ENDDO
```

本例的依赖图在图7-6中给出。

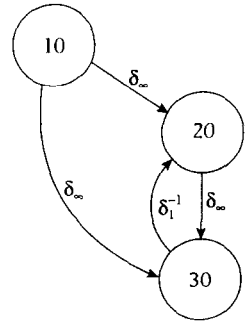


图7-6 迭代依赖例子

当对这个例子应用向量化算法后，将产生

```
READ 5, L, M
DO I = 1, L
   N = 2 * I
   A(I, 1:M, 1:N) = 0.0
ENDDO
```

WHILE循环中的迭代依赖可进行类似处理，但循环条件要在循环
346 中计算。例如:

```
DO I = 1, L
   M = I * 2
   J = 0
   DO WHILE (J.LT.M)
      J = J + 1
      A(I, J) = 0
   ENDDO
ENDDO
```

这种情况的变换将产生

```
irange1 = (1, L)
DO I =irange1
   M = I * 2 (irange1)
   J = 0 (irange1)
   irange2 = (J.LT.M) (irange1)
   DO WHILE (irange2)
      irange2 = (J.LT.M) (irange2)
      J = J + 1 (irange2)
      A(I,J) = 0 (irange2)
   ENDDO
ENDDO
```

由于条件irange2在内层循环每次执行时都要重新计算，所以，如果不做进一步的修改，这个例子就不可能进行向量化。为了将内层WHILE循环转化为迭代循环，我们引入编译器产生的归纳变量jtemp:

```
irange1 =L
DO I =irange1
   M = I * 2 (irange1)
   J = 0 (irange1)
   irange2 = ( 0.LT.M) (irange1)
   DO jtemp = 1, nolimit WHILE (irange2)
      irange2 = ( J.LT.M) (irange2)
      J = J + 1 (irange2)
      A(I, J) = 0 (irange2)
   ENDDO
ENDDO
```

347 接下来，实施归纳变量替换和前向表达式合并将得到如下代码:

```
DO I =1, L
   M = I * 2 (1,L)
   J = 0 (1,L)
   irange2 = ( 0.LT.M) (I,L)
   DO jtemp = 1, nolimit WHILE (irange2)
      irange2 = ((jtemp-1).LT.M) (irange2)
      A(I, jtemp-1) = 0 (irange2)
   ENDDO
ENDDO
```

现在控制内层循环语句的irange2只是简单地测试了(jtemp-1.LT.M)，它可以被并入循环控制条件，产生一个完整的迭代循环:

```
DO I =1, L
   M = I * 2 (1,L)
   J = 0 (1,L)
   irange2 = (1,M+1) (1,L)
      A(I, jtemp-1) = 0 (irange2)
   ENDDO
ENDDO
```

这个循环版本可以在第二维被向量化，产生

```
DO I =1, L
   M = I * 2 (1,L)
   A(I, 0:M) = 0
ENDDO
```

这个例子除了说明如何处理WHILE循环中的迭代依赖外，还显示利用第4章介绍的变换的变形形式将WHILE循环转换为迭代循环的一般过程。

7.2.9 if重构

虽然if转换是非常有用的变换，但是在不能进行向量化的情况下，它对性能有负面的影响。比如考察下面的例子:

```
      DO 100 I = 1, N
         IF (A(I).GT.0) GOTO 100
         B(I) = A(I) *2.0
         A(I+1) = B(I) +1
100 CONTINUE
```

348

if转换之后，循环变为

```
      DO 100 I = 1, N
         m1 = (A(I).GT.0)
         IF (.NOT.m1) B(I) = A(I) * 2.0
         IF (.NOT.m1) A(I+1) = B(I) +1
100 CONTINUE
```

由于一个依赖环的存在，这个循环不能被正确地向量化。在不进行变换的例子中，只计算一次条件。根据这次计算的结果，或者是一个分支跳转被执行，或者是两个赋值语句被无条件地执行。在进行变换的例子中，还是要计算一个条件，此外，在每个赋值语句前都要计

算条件。在分支跳转不被执行的迭代中，变换后的例子导致每个赋值语句前计算条件的额外开销。在分支跳转被执行的循环迭代中，变换后的代码执行一系列的分支，而原代码只执行一次。这就是说，对于普通的机器（“普通的机器”是指没有条件执行但有条件跳转的机器），变换后的例子将产生与下面循环类似的代码：

```
      DO 100 I = 1, N
         m1 = (A(I) .GT.0)
         IF (m1) GOTO  10
            B(I) = A(I) * 2.0
10       IF (m1)  GOTO 20
            A(I+1) = B(I) +1
20       CONTINUE
100   CONTINUE
```

由于控制执行引起的额外开销，if转换将使这个例子的性能变差。为了避免if转换使这些没有向量化的循环性能变差，我们可以采用称为*条件重构*的逆转换过程。简单地说，if重构就是将控制执行转换为一系列条件分支语句。

if重构的目标使将各部分被控制执行的代码用一个最小的分支集合来代替，这些分支可以显式地控制那些代码的执行。如果一个未被向量化的循环输入*codegen*的语句顺序和输出的语句顺序相同，那么重构一组效果不比原来差的分支语句是相当容易的。但是，*codegen*经常会改变语句的顺序。而且，程序员在编码时也不总是使用最少数量的分支。这些都说明我们需
349 要一个更加智能的方法来决定由等价条件控制的代码的各个部分。前向分支指令的处理很简单，只需要对依赖图进行一次简单的拓扑序遍历，尽可能长时间地维护控制条件。出口分支具有将语句紧密地锁定在一个依赖环中的属性，所以，它们的相关区域容易确定。但是，许多可以用向量硬件处理的常见的依赖环也写成出口分支的形式（比如，在一个条件向量中搜索第一个和最后一个为真的元素）。因此，重构过程的一个重要部分就是识别这些依赖环并把它们转换成可以有效利用向量部件的代码。这种依赖环的识别在7.2.4节中讨论过了。

虽然if重构可以删除大部分由if转换引入的低效的部分，if转换、布尔化简和if重构的过程在没有向量化情况下仍然为编译器引入了大量无用工作。而且，对于诸如标量扩展这样的变换来说，一些形式的控制依赖还是需要的。这些缺点使另一种替代if转换的方法具有吸引力。这种替代方法在下一节介绍。

7.3 控制依赖

虽然if转换巧妙地解决了存在条件分支时的向量化问题，但它也有一些不希望出现的副作用，最成问题的是它不必要地使不能向量化的代码变得复杂了。更为理想的解决方法是先分析代码确定进行向量化还是并行化，然后只在需要产生并行代码时转化IF语句。但不幸的是，if转换过程中无法做到这一点，因为在这个变换之后，我们才能确信数据依赖是否刻画了为了保证正确性而需要的所有限制。

在本节中，我们找到一种不同的方案，这种方案用另一种依赖来表示控制流引入的限制，这种依赖称为*控制依赖*。直观地说，如果依赖边的源点处的语句是分支语句而且它能决定依赖边汇点处的语句是否执行，那么控制依赖就存在于这两个语句之间。在本节剩余的部分，我们将形式化这个直观的定义并且说明这种依赖如何运用在向量化和并行化的过程中。

我们从控制依赖的具体定义开始：

定义7.1 如果（1）存在一条从语句x到语句y的非空路径，使得在这条路径上的每一语句$z(z \neq x)$被语句y后控制，（2）x不被y后控制，就说语句y控制依赖于另一语句x。 350

这个定义的直观含义是x必须是一个条件分支，这样分支转向一个方向时y必然被执行到，如果分支转向另外的方向，控制流就不会执行到y。

因为基本块是具备以下性质的语句的最大的组：执行组中所有语句的充分必要条件是组中的任一条语句被执行，所以，为了方便起见，我们将控制依赖作为基本块的性质来讨论。在基本块中控制流是没有变化的，所以一个基本块中的每条语句有相同的控制依赖。下面来看图7-7中的例子。

在这个图中，每个结点代表一个基本块，每条边代表一个可能的控制转移。每个结点处标出这个基本块控制依赖的基本块的集合。值得注意的是块4和块5都不依赖于块1，因为控制流由块1处转向它的某个后继并不能决定块4和块5是否执行。而且，块6也不依赖于块3，因为只要执行到块3就会执行到块6。

控制依赖图（就是代表基本块之间控制依赖的图）可能比它相应的控制流图大，如图7-8所示。在这个例子中，图左边的每个结点都控制依赖于惟一的一个其他结点。但是，图右边的每一个结点都控制依赖于结点1和每一个能到达它的偶数结点。这样，包含$2n+2$个结点的控制依赖图中的控制依赖个数为左端的n加上

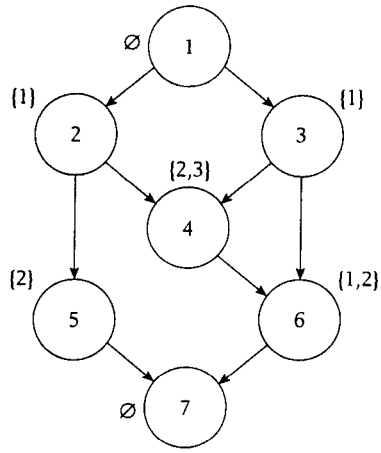


图7-7 控制依赖例子

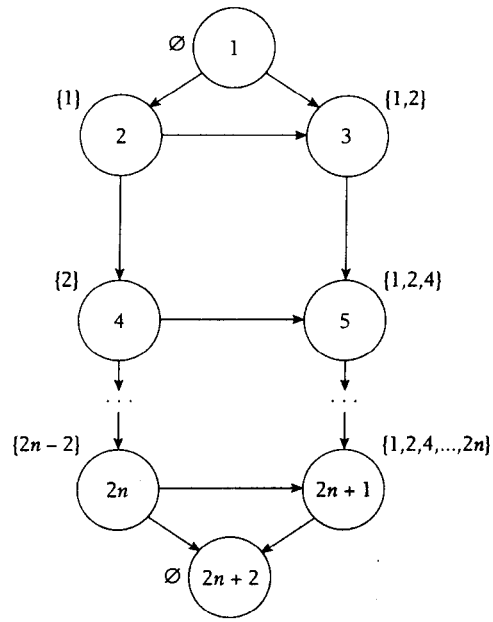


图7-8 控制依赖图的平方级增长

$$\sum_{i=1}^{n}(i+1)=\frac{n(n+1)}{2}+n$$

总数为

$$\frac{n(n+1)}{2}+2n=\frac{n(n+5)}{2}$$

或者说有$O(n^2)$个控制依赖。

7.3.1 构造控制依赖

351
~
352

在向量化过程中运用控制依赖图之前，我们必须先构造一个依赖图。图7-9所示的Cytron等人[97]给出的算法*ConstructCD*是已知的由控制流图构造控制依赖图通用的最快算法。

```
procedure ConstructCD(G, CD)
    // G是输入控制流图
    // CD(x)是x所控制依赖的块的集合
    // ipostdom(x) 是块x在控制流图G中的直接后控制结点
L1: 为控制流图G找出直接后控制结点关系ipostdom；（对于单出口结点的控制流图来说，直接后控制结点
    关系构成一棵树，出口节点是这棵树的根结点。）
    令l是该后控制结点树的拓扑排序列表，其中如果x后控制y，则在l中x在y之后；
L2: while l ≠ Ø do begin
      x是l中的第一个元素;
    从l中删除x;
L3: for all x的控制流前驱y do
      if ipostdom(y) ≠ x then CD(x) = CD(x) ∪ {y};
L4: for all z，其中ipostdom(z) = x, do
      for all y ∈ CD(z) do
        if ipostdom(y) ≠ x then CD(x) = CD(x) ∪ {y};
    end
end ConstructCD
```

图7-9 控制依赖的构造算法

算法*ConstructCD*依赖于后控制结点树的构造。

定义7.2 在只有一个出口结点的有向图G中，如果任何从结点y到图G的出口结点的路径都经过结点x，则结点x*后控制*结点y。

图7-10包含图7-7所示控制流图的后控制结点树。注意，每个结点都被出口结点7后控制。

计算有向图的后控制结点的问题曾经被很多人研究过。Lengauer-Tarjan算法[198]是广泛应用的算法，它在最坏的情况下的时间复杂度为$O(E\alpha(N, E))$，其中N和E分别表示控制流图中结点的数量和边的数量，α表示和Ackerman函数的反函数相关的一个增长非常慢的函数。事实上，由于α增长非常慢，所以在实际情况下，算法相对于输入的图的大小是线性的。最近，Harel发表的算法达到线性时间界限[138]。我们推荐把图4-8中给出的迭代算法进行修改后用于计算后控制结点。Cooper，Harvey和Kennedy已经证明如果巧妙地实现集合数据结构，这个算法对于实际程序中的控制流图是十分高效的[84]。

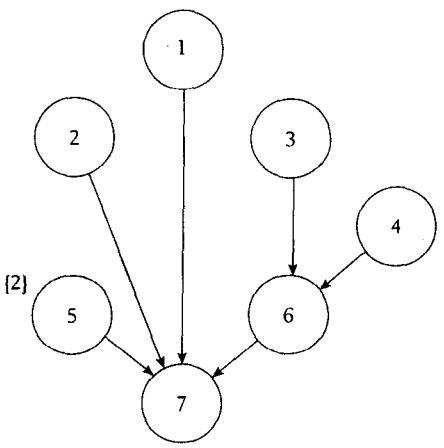


图7-10 图7-7的后控制结点树

正确性 要证明算法*ConstructCD*能正确构造控

制依赖关系，我们必须证明：在算法执行之后，$y \in CD(x)$ 当且仅当x控制依赖于y。

注意算法处理结点的先后顺序，是同后控制结点关系一致的。这就保证了如果x后控制z，则z将在处理x之前被处理。这样，我们就可以通过归纳证明，假设当x被处理时，这个性质对于所有的x后控制的结点成立。这个证明的剩余部分留给读者，如习题7.4。

复杂度 如果不考虑后控制结点的构造，算法*ConstructCD*在最坏的情况下需要$O(\max(N+E, |CD|))$ 的时间。为了证明这一点，注意到按照拓扑排序遍历需要$O(N+E)$ 的时间，而在标号L_2处的循环头对控制流图中每个结点执行一次，或者说这部分需要$O(N)$ 的时间。标号L_3处的循环对每个结点的每个控制流前驱执行一次，共需要$O(E)$ 次，由于这个循环体可在常数时间内完成，这个循环需要的总的时间是$O(E)$。

在标号L_4处的循环更复杂一些，它对后控制结点树上的每条边执行一次，但是由于每个结点只有一个后控制结点，这个循环头只执行$O(N)$次。内层循环最多对控制依赖图上的每条边执行一次，所以循环嵌套需要$O(|CD|)$ 次。总的时间界限可立即推出。

353 ~ 354

由于这个时间界限是算法输入和输出大小的最大值，所以没有算法可以比它性能更好。

7.3.2 循环中的控制依赖

虽然标准算法可以通过以下方法处理循环，先把循环转换成一个控制流图，然后运用算法*ConstructCD*，但是将循环作为特例来处理会更好一些。比如，有时我们需要一个循环控制结点，因为我们需要把一些循环迭代的信息附在上面。这个结点还可以在各种变换中代表这个循环，如可以通过克隆这个循环结点来实现循环分布。

在这里的讲述中，我们将使用循环控制结点来代表循环。我们可以把这个结点看成是在计算循环控制表达式，进而决定是否执行下一次迭代。问题立即出现：程序中哪些语句是控制依赖于循环控制结点的？回忆一下，如果从语句S_1出来的一个分支会导致S_2的执行而另外的分支不会导致S_2的执行，则存在一个从S_1到S_2的控制依赖。很明显，循环控制结点会导致循环体一个迭代的执行，所以循环中不控制依赖于循环中其他语句的所有语句都是控制依赖于循环控制结点的候选者。一般来说，从循环控制结点到循环中当循环体执行时就会被执行的每个结点都有一条边。

下面的例子可以说明这一点：

```
10  DO I = 1, 100
20     A(I) = A(I) + B(I)
30     IF (A(I).GT.0) GOTO 50
40        A(I) = -A(I)
50     B(I) = A(I)+C(I)
    ENDDO
```

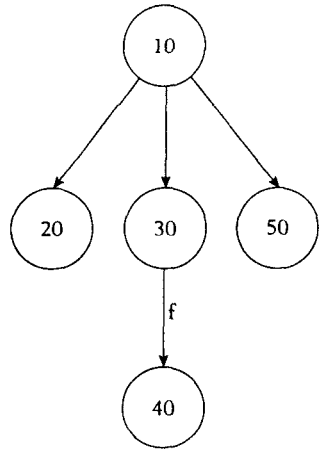


图7-11 控制依赖例子

这个循环的控制依赖图在图7-11中给出。语句30到语句40之间的控制依赖上的标号f是用来说明这个语句是在if条件为假时执行。我们注意到与while语句类似，循环控制依赖被标注为真，这表示相关的while 条件为真。因此，当while条件为假，语句就不被执行。

循环控制语句的特殊之处是它建立了一组可以被执行的语句实例。对于一个迭代范围固定为1到100和步长为1的DO循环，循环控制变量可以被认为产生循环体中每条语句的100个实例。

这些语句实例的正确执行将在下一小节中讨论。

7.3.3 控制依赖的一个执行模型

在下面几段中，我们将证明涉及到控制依赖的变换是正确的。但是，为了阐明正确性问题，我们必须首先为标有控制依赖和数据依赖的程序建立一个执行模型，并且必须证明这个模型是正确的。我们已经知道数据依赖表示语句必须按照一个特定顺序执行。控制依赖实质上也表明同样的事情；但是控制依赖更复杂一些，因为它同时会告诉我们一个语句是否会执行。

回想我们在第2章建立的一个模型，这个模型允许我们在不颠倒依赖的源点和汇点的情况下重排语句或语句实例的顺序。当时没有控制流，只有语句和循环。因此我们可以把程序的执行看作是一组语句实例$S(i)$，每一条语句实例都带有包含这条语句的循环嵌套的迭代向量i作为索引。在这个模型中，当$S(i)$所依赖的每个语句实例都被执行后，$S(i)$就可以执行。换句话说，如果$S(i)$依赖于$S_0(j)$，其中$j \leqslant i$，则$S_0(j)$必须在$S(i)$之前执行。我们把这个模型看作是这样一个执行模型：一个语句可以在它依赖的所有语句都执行过之后的任何时间执行。

355
~
356

控制依赖引入另外一种可能性：S_2依赖于S_1，但是S_1从不会被执行。在上面的执行模型中，S_2也不会被执行到。下面的伪码可以说明这种情况：

```
      DO I = 1, N
S0      IF (P) GOTO 100
            S1
100     S2
      ENDDO
```

在分支跳转被执行的迭代中，$S_2(I)$不会等待$S_1(I)$执行之后才执行，因为$S_1(I)$不会被执行。虽然，这个依赖在其他迭代中是真实存在的，但从$S_1(I)$到$S_2(I)$的路径在那些迭代中不被执行，所以不存在依赖。

为阐明这个问题，我们可以把每个语句实例看作带有一个执行变量$S(i).doit$。不控制依赖于其他语句实例的语句实例有它们自己的*doit*标记，初始化为真。所有其他语句实例的*doit*标记初始化为假。于是，如果语句实例$S(i)$的*doit*标记被设置为真并且它依赖的每个语句实例的*doit*标记为假或已经被执行过，我们就可以执行这个语句实例。

在这个模型中，一个语句的*doit*标记如何才能设置为真呢？规则很简单：对于所有控制依赖于条件并且它们的执行由条件决定的语句，它们的*doit*标记被设置为真。

这一点促使我们在控制执行的控制依赖上标记上条件的真值。所有那些控制依赖于一个条件并且当条件为真时才会执行的语句，在它们的控制依赖边上标记为真，而条件为假时执行的语句，在它们的依赖边上标记为假。这样前面例子中的循环体的依赖图如图7-12所示。

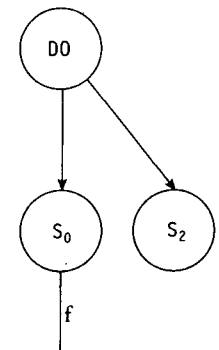


图7-12 控制依赖实例

我们观察到，如果一个给定语句S的*doit*标记被设置为真，那么存在一个控制语句序列$S_0, S_1, \ldots, S_m=S$，其中，S_0不控制依赖于其他语句，因而在原程序中无条件执行，接下来，对于每个k, $0 \leqslant k < m$, S_k处决定促使S_{k+1}的执行。这样，控制依赖序列定义惟一一条在原程序中必须采取的执行路径。

在这个执行模型中有待说明的是如何定义循环控制结点在控制依

赖图的执行中的行为。我们先从一个简单的循环结点开始，这个循环结点不包括在任何由它开始的循环携带的控制或数据依赖环中。在这种情况下，当循环结点的*doit*标记被设置为真时，循环的迭代范围可以完全计算出来。然后，循环头的执行将产生一组语句实例，每一个循环迭代的每一个语句对应一个语句实例。然后，处理过程将会把每个依赖于循环头语句的语句实例的*doit*标记设置为真。在这一步，所有其他语句的*doit*变量被设置为假。*doit*标记为真的语句实例应当像原来一样执行。

在有依赖环存在的情况下，迭代范围的计算依赖于循环中计算出的量，DO循环结点的执行更复杂一些。如果迭代的范围不为空，那么它将产生一个其本身的新的语句实例来调整剩余的迭代。如果反向依赖于DO结点的依赖是数据依赖，则给这个DO结点一个为真的*doit*标记，如果反向依赖于它的依赖是控制依赖，则给它一个为假的*doit*标记。于是，循环中的每个语句都产生语句实例，并且所有控制依赖于循环头的语句的*doit*标记都设置为真，其他语句的标记都设置为假。这样，执行就能正常进行。

这个模型的正确性在下面给出。

定理7.1 按照本节给出的执行模型执行的依赖图在含义上等价于创建出这个依赖图的原程序。

证明 首先，这个执行模型要求任何语句实例不能在它所依赖的语句之前执行。这样，我们仅需要考虑控制依赖引入的额外的复杂性。这个证明的本质是要证明在此图执行模型下执行的依赖图精确地执行了与原程序相同的语句集合。为了证明这一点，我们必须证明当且仅当一条语句在原程序中执行，它在图中的*doit*标记才会被设置为真。

357 ~ 358

假设情况相反，一些语句实例$S_x(i_0)$得到错误的*doit*标记。那么存在某个语句实例序列$S_0, S_1, \ldots, S_m$，其中S_0不控制依赖于任何其他语句（因而必定会执行），并且，$S_m = S_x(i_0)$。而且，不失普遍性，假设S_m是序列中第一个得到错误*doit*标记的语句。进一步，不失普遍性，假设这是一条到达错误标记的最短序列。因此，这个序列就代表一系列控制判定，它表示原程序中一条到达$S_m = S_x(i_0)$的执行路径。由于这条路径上除最后一个在S_{m-1}的判定外，其他所有的判定都是相同的，因此我们必须考虑在S_{m-1}处作的判定。由于所有数据依赖都是满足的，所以，在这条路径上，S_{m-1}所有直接或间接依赖的语句都已经在S_{m-1}这个判定之前执行过了，它们的*doit*标记一定是被正确设置的，因为如果它们的标记不正确的话，将会产生一个长度为$m-1$的判定序列到达另一个*doit*标记错误的语句，这与我们前面假设长度为m的路径是到达错误标记语句的最短路径矛盾。

同理，不可能存在原程序中没有执行而在图中执行并影响S_{m-1}处的控制判定的语句，因为这样的语句的标记必然是错误的，且到达它的判定序列也一定比m短，又和假设矛盾。所以，在语句S_{m-1}处的判定必然和原程序中的一致。

这个证明可以不加修改地用在含有DO结点的图中，因为上面证明中的语句也可以是语句实例。

这个定理给出一个工具，它能建立保持数据依赖和控制依赖正确的变换。此外，它也可以作为依赖图的生成代码过程的基础。

7.3.4 控制依赖在并行化中的应用

在具备建立控制依赖图的能力后，下一个问题是如何将它运用到并行代码生成中去。更明确地说，有以下两个问题需要解决:

(1) 修改代码生成中采用的变换（即循环分布、循环合并和循环交换），用于处理控制依赖。

(2) 把抽象的语句、数据依赖和控制依赖转换到可执行的代码中去（换句话说，就是根据控制依赖图重新构造控制流）。

359 下面几段讨论这两个问题。

控制依赖和变换

在检测控制依赖时，最好是先只考虑循环无关控制依赖，也就是说，只考虑那些循环内由于前向分支造成的依赖。后向分支造成的隐式循环，应当单独被处理；出口分支造成一些复杂的循环携带依赖，这一点在7.2.4节中讨论过。为简单起见，本节只讨论由前向分支造成的循环无关控制依赖和由循环造成的控制依赖。

我们在第2章中讲到，大多数循环变换不受循环无关依赖的影响，无论是数据依赖还是控制依赖。例如，循环反转不会影响循环无关依赖。同样地，循环倾斜、循环分段、索引集分裂和循环交换也不会影响循环无关依赖。两种会影响循环无关依赖的循环变换是循环合并和在有控制依赖时的循环分布。当可能把两个循环间的循环无关依赖转换为一个循环内的循环携带依赖时，循环合并是不允许的。但是，如果出口分支被排除，就不可能有这种控制依赖(这种分支必须跳出一个循环进入另一个循环)。这样，就只剩下循环分布能够非法地影响控制依赖。如果一个循环无关控制依赖的源点和汇点分别分布在两个单独的循环里，循环无关依赖将会非法地成为循环携带的依赖。例如，在代码

```
      DO I = 1, N
S1      IF (A(I).LT.B(I)) GOTO  20
S2         B(I) = B(I) + C(I)
20      CONTINUE
      ENDDO
```

中，惟一的依赖是从S_1到S_2的控制依赖。将这两条语句分布到两个单独的循环中显然是错误的

```
      DO I = 1, N
S1      IF (A(I).LT.B(I)) GOTO 20
      ENDDO
      DO I = 1, N
S2         B(I) = B(I) + C(I)
      ENDDO
20      CONTINUE
```

因为由于S_1在某个迭代中执行分支跳转而简单地跳过S_2的所有迭代显然是不正确的。这种情形
360 类似于跨越两个没有进行标量的循环分裂一个标量依赖。

到目前为止，所有的代码生成算法的中心要素就是把一个单个循环分解为尽可能小粒度的单元，所以，要想成功地把控制依赖纳入这个框架，正确处理循环分布是非常关键的。

循环分布可以被看作是将循环控制语句克隆，并把原来依赖于它的语句分成两组，每一个新循环里放一组。由于任何与依赖环有关的语句都不能被分开放到这两个组中，所以我们有明确的条件来说明循环分布什么时候是合法的。其中难于处理的部分是：当一个控制依赖

跨越分布后的两个循环时该怎么办。当然，也可以克隆条件语句，但这样做是有风险的，因为这样可能把一个有副作用的表达式计算两次。这里我们真正需要的是某种机制，我们用这种机制可以以某种可以在不同的数组中使用的表示方式来捕获被控制语句的*doit*标记。换句话说，由于可能存在一个跨越循环分布后的两个小循环的循环无关控制依赖，这两个循环一个可以被向量化，而另一个不能，所以我们需要的基本变换是把一个条件计算的结果保存到需要它的时候，这和标量与标量扩展的关系一样。

这就给出一种类似于if转换的变换，它用一个逻辑数组保存条件表达式计算的结果，这些结果以后可以被查询到。比如，假设你想要分布前面例子中的循环（事实上，你更可能不想分布它），得到的正确的结果是

```
      DO I = 1, N
S1      e(I) = A(I).LT.B(I)
      ENDDO
      DO I = 1, N
S2      IF (e(I).EQ..FALSE.) B(I) = B(I) + C(I)
      ENDDO
```

这里进行的基本变换和if转换是类似的，但是一个重要的区别是怎样运用变换。在循环分布过程中，if转换是无条件地把所有的控制依赖统一转换成数据依赖。而这里的转换是在循环已经被充分分布并重新合并后，弥补那些被变换过程打破的控制依赖。

图7-13是一个更为复杂的例子，我们用它来说明存在控制依赖时的循环分布的问题。

```
      DO I = 1, N
1       IF (A(I) .NE. 0) THEN
2         IF (B (I) / A(I) .GT. 1) GOTO 4
        ENDIF
3       A(I) = B(I)
        GOTO 8
4       IF (A(I) .GT. T) THEN
5         T = (B(I) - A(I)) + T
        ELSE
6         T = (T + B(I)) - A(I)
7         B(I) = A(I)
        ENDIF
8       C(I) = B(I) + C(I)
      ENDDO
```

图7-13 一个存在控制依赖的循环分布的例子

这个例子的控制流图在图7-14中给出，相应的控制依赖图在图7-15中给出。当加入数据依赖时（在图7-16中数据依赖用虚线箭头表示），可以很明显地把它再划分为n块。

在图7-16中，虚线框出的部分表示先进行充分的循环分布再尽可能把类似区域合并后的循环结构。循环1是一个并行循环，循环2是一个串行循环，循环3也是一个并行循环。为了保证变换的正确性，跨越这些划分边界的控制流边必须转换成能永久的形式。得到这种永久性是通过构造一些称为*执行变量*的数组，用这些数组来保存在跨越循环的边的源点处进行的分支判定。上面的例子需要两个执行变量：用来保存在语句2处的分支结果的E2(I)和用来保存

在语句4处的分支结果的E4(I)。为了理解执行变量可能具有的值，让我们来考虑在这两个语句处可能保存的结果。在语句2处，有三种可能保存的结果:

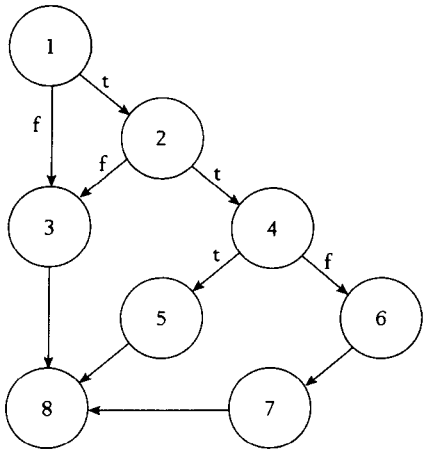


图7-14 图7-13的控制流图

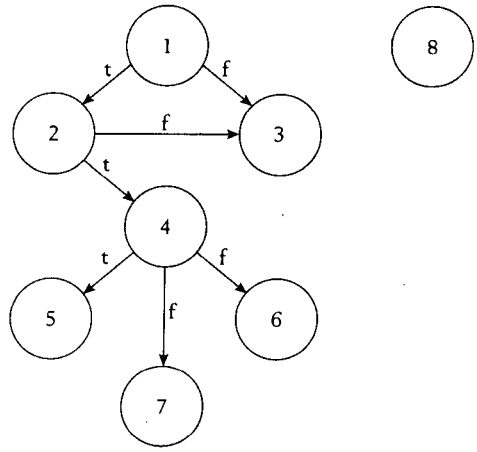


图7-15 图7-13的控制依赖图

(1) 语句2执行，且执行到语句4的真分支，

(2) 语句2执行，且执行到语句3的假分支，

(3) 语句2不执行，因为在语句1处执行了假分支。

这三种情况可能保存在任何跨循环控制依赖的源点的相应语句处。这同我们的执行模型中的要求对应，如果从语句S_0到语句S有一个控制依赖且S_0的*doit*标记已经设置，那么以上三种情况可以用以下的形式来保存: *doit*标记在给定的语句S处被设置，且条件表达式的值被标记在跳转到S的分支上（即它强制S的执行)。

因此，看来自然可以模拟这些情况，用执行变量元素的三个值*真*、*假*和*未定义*（有时用“⊤”表示）与上述三种情况对应。当确定需要某些执行变量时，这些执行变量首先被初始化为⊤。然后，程序变换将在代码的适当位置设置这些执行变量。图7-17中的算法实现这样的转换。但是请注意，算法并没有产生并行代码，而只是选择性地把控制依赖转化成数据依赖，并在此过程中改变某些语句。

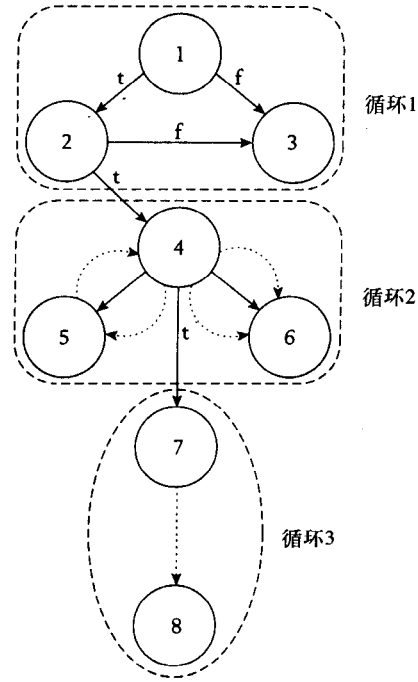


图7-16 图7-13的数据依赖和控制依赖图

正确性 为了证明图7-17中算法*DistributeCDG*是正确的，我们必须证明它产生的依赖图和原来的依赖图含义是一样的。这一点可以从定理7.1给出的执行模型的正确性证明和以下观察中直接得出: 循环分布之后的新图中某语句的*doit*标记会被设置，当且仅当它的*doit*标记在原程序中会被设置。我们观察到，证明这一点惟一的问题在于当一个控制依赖会跨越分割的边界时出现的问题。在那种情况下，循环中每一个语句实例有一个*doit*标记。由于两个循环中的语句实例相同，我们必须证明执行变量数组保存由跨循环控制依赖

边所控制的语句索引的正确的*doit*标记。

```
procedure DistributeCDG(G, G_cd, P)
    // G是输入的控制流图
    // G_cd是输入的控制依赖图
    // P={P_1, P_2, ..., P_k}是表示循环在分布之后的一组分割
    //输出: 用执行变量修改后的G_cd

L_1: for each分割P_i do begin
    for each n∈P_i, 使得存在一条边(n, o)l∈G_cd, 其中l是控制依赖边的真/假标号, 使得o∉P_i
    do begin
      在P_i的开始处插入"EV (I)=T";
      test是n的分支条件;
      if存在边(n, m)_l∈G_cd, 其中m∈P_i then
        用"EV_n (I)=test""IF (EV_n (I). EQ.. TRUE.)"代替n;
        // 现在, 考虑条件依赖于后面这个测试
      else
        用"EV_n (I)=test"代替n;
      for each包含n的一个后继的P_k, P_k≠P_i do begin
        构造一个新语句N:
          "IF (EV_n (I). EQ.TRUE.)";
        将N加入P_k;
        为EV_n插入一个数据依赖;
        for each (n, q)_l, 使得q∈P_k, do begin
          // 更新控制依赖
          从G_cd中删除(n, q)_l;
          将(N, q)_true加入G_cd;
        end
      end
    end
  end
  复制一份原始的DO结点, 把所有的依赖也一同复制, 并且从这个新的DO结点到P_i中每条语句插入
  控制依赖边;
end DistributeCDG
```

图7-17 执行变量和控制的生成

这样，我们的目标是证明在修改后的程序中，*S*的每个对应实例的*doit*标记会被设置，当且仅当*S*的*doit*标记在原来未被分布的循环中被设置。我们首先注意到，这两个循环控制语句有相同的控制前驱集合，所以，如果原来的循环会被执行到，则这两个循环也会被执行到。而且，这两个循环产生的两组语句实例是相同的。

361
~
365

充分性：假设在迭代*i*中，控制前驱S_c在原程序中执行。那么它的*doit*标记会被设置为真，同时，所有控制后继（由标号控制）的*doit*会被设置为和控制分支的条件相同的值。在转换后的图中，这些标号保存在执行变量EV(*i*) 中，并传递给另一个条件，这个条件是在给被控制的语句的*doit*标记设置值之前测试这些真值是否正确。这样，语句*S*的*doit*标记将在转换后的程序中被正确地设置。

必要性：如果控制前驱S_c在原程序迭代*i*中不执行，执行变量数组元素EV(*i*) 将会被置为

“⊤”，并且在转换后程序的第二个循环中对这个变量的比较测试将会产生“假”的结果，这就保证这个测试条件所控制的语句不会被执行。另一方面，如果在计算条件时产生错误的真值，那么在另一个循环中对该真值的测试将会失败，相应语句的*doit*标记不会被设置。证明完毕。

将算法*DistributeCDG*用在图7-13的例子中并且配以适当的代码生成，得到的结果在图7-18中给出。“适当的代码生成”将是下面一段的主题。

```
        PARALLEL DO I = 1, N
          E2(I) = ⊤;
1       IF (A(I) .NE. 0) THEN
2          E2(I) = (B(I) / A(I).GT. 1)
        ENDIF
3       IF (E2(I) .NE. .TRUE.) A(I) = B(I)
        ENDDO
        DO I = 1, N
           E4(I) = ⊤;
           IF (E2(I) .EQ. .TRUE.) THEN
              E4(I) = (A(I) .GT. T)
4             IF (E4(I) .EQ. .TRUE.) THEN
5                T = (B(I) - A(I)) + T
              ELSE
6                T = (T + B(I)) - A(I)
              ENDIF
           ENDIF
        ENDDO
        PARALLEL DO I = 1, N
           IF (E4(I) .EQ. .FALSE.) THEN
7             B(I) = A(I)
           ENDIF
8       C(I) = B(I) + C(I)
        ENDDO
```

366

图7-18 图7-13在循环分布后的代码例子

生成代码

虽然控制依赖图可以表示任何的控制流，但是真实的机器只能执行非常有限的一组控制流操作。当执行了循环分布和其他变换后，带有控制依赖的依赖图就不能很明显地再被映射成一种可执行的形式。为了说明问题可能的困难程度，我们假设目标机可以执行Fortran中的控制流操作，从而目标语言是Fortran。请看例子

```
DO I= 1, N
S1        IF (p1)  GOTO 3
          S2
          GOTO 4
3         IF (p3) GOTO 5
4         S4
5         S5
   ENDDO
```

图7-19是为这个例子构造的控制依赖图，它同时也显示在循环分布和合并之后想要得到的分割。

按照图7-19中给出的分割进行循环分布所产生的控制依赖图在图7-20中给出。结点1a和1b是循环分布算法在第二个分割中产生的两个新结点，它们都数据依赖于结点1。

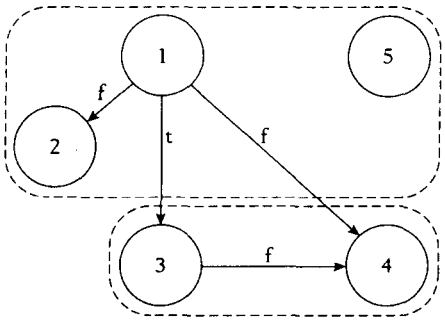


图7-19 示例代码段的控制依赖图

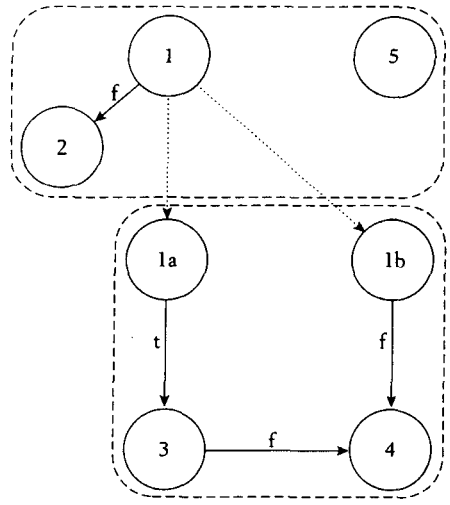


图7-20 示例代码段在循环分布之后的控制依赖图

当两个分割的代码产生之后，第一个分割的代码成为

```
DO I = 1, N
   E1(I) = p1
   IF(E1(I).EQ.FALSE) THEN
      S2
   ENDIF
   S5
ENDDO
```

第二个分割的代码不是如此简单。语句4控制依赖于两个不同的条件结点，所以它不能不加任何变换地被放到一个结构化的条件语句中去。解决这个问题的一个简单的方法是使用条件变换的一种变形。第二个分割的图中惟一的非条件语句是语句4。如果我们用if转换时的方法计算出它所依赖的条件集合，我们就能得到如下代码:

```
DO I = 1, N
   IF ((E1(I).EQ..TRUE.).AND..NOT.p3).OR.
       (E1(I).EQ..FALSE.)) THEN
       S4
   ENDIF
ENDDO
```

367
~
368

但是，情况不总是这么简单。下面考虑同一个原始循环的简单变形:

```
    DO I = 1, N
S1      IF (p1)  GOTO 3
        S2
        GOTO 5
3       IF (p3)  THEN
           S4
           GOTO 6
```

```
        ENDIF
5       S5
6       S6
    ENDDO
```

上面循环在循环分布之后的控制依赖图在图7-21中给出，其中循环分布是按照图中的虚线进行的。进一步假设从语句S_4到S_5存在一个数据依赖。这个例子的代码生成比较困难，因为我们只能为语句S_4产生一个结构化的IF语句而不能为语句S_5也生成一个。为了解决这个问题，我们使用条件转换的变形为每一个有多控制依赖前驱的语句生成一个IF语句。这种方法得到的代码是：

```
DO I = 1, N
   P1a3 = (E1(I).EQ..TRUE.).AND..NOT.p3
   IF (P1a3)   S4
   IF ( P1a3.OR.(E1(I).EQ..FALSE.))  S5
ENDDO
```

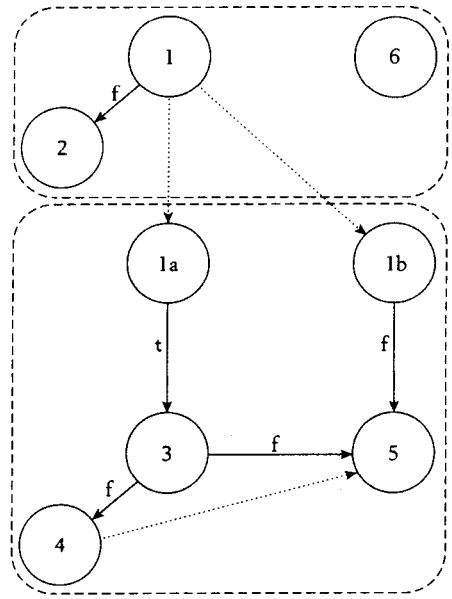


图7-21 示例代码段的控制依赖图

很容易看到，为每个结点最多只有一个控制依赖前驱的图产生代码是比较容易的——只要在控制条件的真分支或假分支内生成每条语句。利用这个观察，带控制依赖的无循环依赖图的一般代码生成算法分两个阶段操作。第一个阶段把控制依赖图变换成一个含有一组控制依赖树的*规范形式*，这个形式具有以下性质：

（1）每条语句最多控制依赖于一条其他的语句（即每条语句至多是一棵依赖树的成员）。

（2）依赖树可以按这样的顺序排序，使得所有树之间的数据依赖都从该序列中前面的树中流出，并流入到该顺序中后面的树中去。

在第二个阶段，我们将运用简单的递归过程为这组规范形式的依赖树产生代码。

我们现在开始讨论图7-22中给出的依赖图。在这个图中，实线边代表控制依赖，虚线边代表数据依赖。我们假设所有发出控制依赖的结点都是if语句，它计算某个谓词，而其他结点都是简单语句。每个结点上的标号都是语句的编号。

图7-22中有几件事是明显的。第一，语句3和7有多个控制依赖前驱。这些结点必须首先分离出去，依次形成它们的树。想要的结果在图7-23中给出，图中产生了初始化语句1a，5a，6a和8a，它们分别用来保存在语句1，5，6，8中计算的谓词的结果。使用这个策略是为了避免重复谓词计算。每个初始化赋值都在等式左侧有一个编译器生成的逻辑变量。

新结点“1|5&6”和“5&6|8”是用来控制结点3和7的组合结点，结点3和7先前有多个控制依赖前驱。注意，没有语句在结点6的控制之下，所以它可以被删除。另一方面，结点5不能被删除，因为它现在包含结点6中谓词的初始化语句。

369
ι
370

下一步是看这些依赖树是否能被线性排序。由于从语句2到语句3和从语句3到语句4的数据依赖，这个例子中的依赖树不能被线性排序。这需要进一步分裂这些树，为语句2和语句4分别构造依赖树。这样做的结果显示在图7-24中，这个结果就是想要的规范形式。现在可以按照与数据依赖相同的顺序1a，1，5a，5，1|5&6，1b，8a，5&6|8，8为依赖树产生代码。

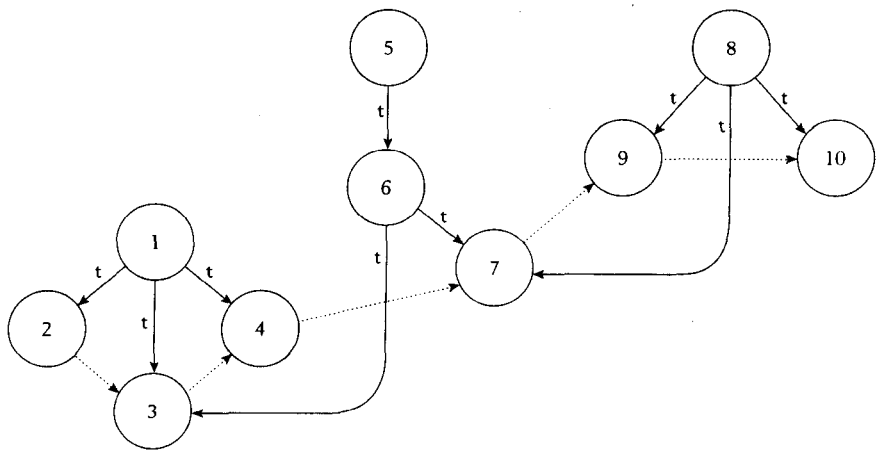


图7-22 一个控制依赖图的例子

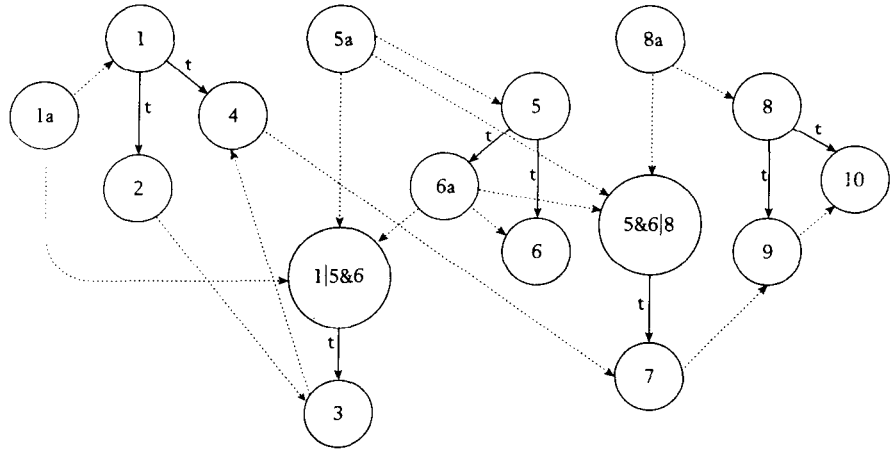


图7-23 分裂结点后的控制依赖图示例

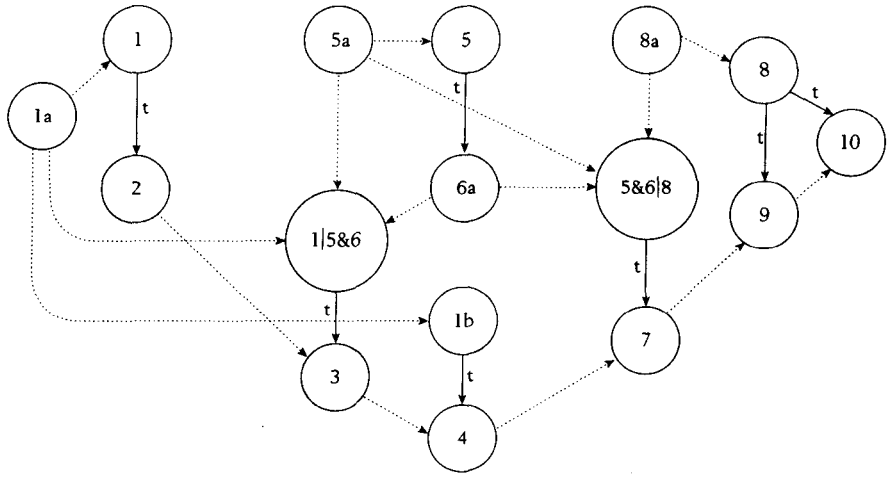


图7-24 数据依赖分裂后的控制依赖图示例

剩余的一个问题是如何做第二步的分裂，即把控制语句的控制依赖后继组织成条件相同的语句组。如果在一些语句之间没有这样的依赖路径：路径经过的语句不全是带有相同标号的相同条件结点的子结点，那么这些语句可以放在同一组里。从这个描述中，容易看出这个分组问题是一个带类型的合并问题，可以用6.2.5节中带类型的合并问题算法来解决这个问题，只要把每条语句用二元组（p, l）来分类，其中p是语句惟一的控制依赖前驱，l是从p到该语句的控制依赖边上的标号。

371
~
372

控制依赖分裂的代码在图7-25中给出。这一遍完成之后，我们就有了这样一个依赖图，图中包含一组排好序的依赖树，树中每一个结点都最多只有一个控制依赖前驱。

```
procedure CDGsplit(G_cd, P)
  // G_cd是输入的控制依赖图
  // P是循环体中的语句集合

    for each P具有多个CDG前趋中的结点v do begin
      令{c_1, c_2, ..., c_m}是v的CDG前趋的集合;
      为其中每个没有初始化的条件c_i构造一个初始化结点e_i，并且插入依赖图中的原始谓词层;
      如果不存在w结点，建立一个用条件c_1|c_2|...|c_m标记的新的w结点，否则令w为构造的结点，作为最远
      公共祖先的子结点;
      从每个e_i到w插入数据依赖边;
      将v作为w的控制依赖后继;
    end
    删除所有没有后继的条件结点（迭代地删除所有后继结点都被删除的结点）;
    for each作为图中其他w结点控制依赖后继的结点v do
      把v的类型设为(c_w, l_wv) 其中，c_w是与w相关的条件，l_wv是w和v之间的边的标号;
    for each不同的类型t do begin
      对图进行类型t的带类型合并;
      if类型t有多个簇then
        为第一个之后的每个簇构造一棵新树，在需要时生成新的条件计算结点，以及相应的数据依赖
        （在最深的可能层构造新树）;
  end
end CDGsplit
```

图7-25 控制依赖分裂算法

我们现在可以使用简单的递归过程生成代码。这些过程在图7-26和图7-27中给出。这组过程按照和数据依赖一致的顺序为组成控制依赖图的每个子树生成代码。图7-27中的例程是为数据依赖于带有相同标号的一个给定结点的一组语句生成代码。由于在规范形式中每条语句只有一个控制依赖前驱，这个过程中忽略集合的控制依赖前驱并且只为集合中每一条语句递归调用*gencode*。

373

关于这个代码生成算法，我们仍需说明最后两点：首先，相对于原依赖图的大小来说，这个代码生成过程的运行时间大致是线性的，这里的依赖图包括数据依赖和控制依赖。算法中惟一不是严格线性的部分是重复地应用带类型的合并，这部分所用的时间和分裂后的图的大小（分裂后的图的大小可以仅仅比原来的大一个线性的系数）与不同类型个数的乘积成比例。通过只与多个语句对应的类型应用合并的方式，可以减少类型的数目。

```
procedure CDGgenCode(G_cd, P)
  // G_cd是规范形式的输入控制依赖图
  // P是循环体中的语句集合

  令S是P中没有控制依赖前趋或数据依赖前趋的语句的集合;
  while S≠Ø do begin
    令n是S中的任一元素;
    从S中删除n;
    gencode(n);
    将没有控制依赖前趋且数据依赖前趋已经由gencode处理的结点加入S;
  end
end CDGgenCode

procedure gencode(n)
  if n是IF(p)的形式then begin
    令T是语句m的集合，(n, m)_true ∈ G_cd;
    令F是语句m的集合，(n, m)_false ∈ G_cd;
    if T≠Ø then begin
      生成“IF (p) then”;
      genset(T, n);
      if F≠Ø then begin
        生成“ELSE”;
        genset(F, n);
      end
      generate “ENDIF”;
    end
    else if F≠Ø then begin
      生成“IF (.NOT. p) THEN”;
      genset(F, n);
      生成“ENDIF”;
    end
  end
  else // n不是一个条件
    生成n;
end gencode
```

图7-26 从控制依赖图生成代码

```
procedure genset(S, n) // 结构化代码的版本

  // 依次为集合S中的每条语句生成代码，其中控制依赖前趋n被忽略
  while S≠Ø do begin
    令m是S中这样一个任意元素，它的所有数据依赖前趋的代码已经生成;
    从S中删除m;
    gencode(m);
  end
end genset
```

图7-27 为结构化代码的一组结点生成代码的过程

第二点观察是，算法试图执行有限制的if转换来生成代码。在实现这个意图时，我们必须很小心，如果结果令人满意，就不要强制执行if转换。这说明在分裂算法中，为什么我们必须在控制依赖树可能的最深层次构造重复结点。当我们生成一个新的条件来控制一个先前有多个控制依赖前驱的结点时，我们希望把它构造成那些控制依赖前驱的公共最深祖先的子结点。当执行带类型合并时，我们应当生成条件的多个拷贝，作为原结点和从原结点到重复结点的路径上的任意结点的最近公共祖先的子结点。最近公共祖先可以通过直接修改带类型合并算法计算得到。

我们举一个例子来说明代码生成的过程，以下是这个过程为图7-24中的控制流图生成的代码的概况：

```
p1 = pred1
IF (pred1) S2
p5 = pred5
IF (p5) p6 = pred6
IF (p1.OR.(p5.AND.p6)) S3
IF (p1) S4
p8 = pred8
IF ((p5.AND.p6).OR.p8) S7
IF (p8) THEN
   S9
   S10
ENDIF
```

374
~
375

代码中用斜体字表示的*pred*变量代表谓词表达式。每个表达式只被计算一次。注意，为了简短起见，在只有一条语句被一个条件控制的地方使用了简单IF语句。

7.4 小结

本章介绍了处理循环中控制流的两种不同的方法：

（1）if转换通过把程序的语句转换成带控制条件语句的形式来删除所有分支语句，其中控制条件反映语句被执行时实际的条件集合。如果控制条件被看作是输入，这个方法起到把控制依赖转换成数据依赖的作用，使得可以直接把前几章中的代码生成过程用在带有分支的代码当中。这种转换在向量化编译器中是非常有效的。

（2）控制依赖是由控制流引入的一种特殊依赖。如果语句S_0是一个条件分支，并且S_0处的分支方向决定语句S的执行与否，则语句S控制依赖于语句S_0。控制依赖可以像数据依赖一样用在分析算法中。但是，它使得代码生成过程变复杂了。

为有条件分支的程序生成代码是本章讨论的主题。

7.5 实例研究

PFC和Ardent Titan编译器的最初版本都使用了系统的if转换来处理循环中的控制依赖。PFC系统实现了本章中讨论的完整的if转换的策略，包括从隐式循环到while循环的转换。但是它并没有取得完全的成功。PFC可以充分地向量化只有前向控制流且控制流只限制在循环范围内的循环。在其他情况下，这种方案没有很大作用。

随后用于并行化的两个PFC版本引入了前面介绍的if重构和执行变量的方案，尽管其中循环分布没有完全按照这里介绍的方案实现。

if转换可能消耗大量编译时间。首先，是转换本身的代价，在7.2.7节中介绍的快速化简算法就是一次失败经验的结果，算法中实现的Quine-McCluskey过程是指数级的。即使是PFC中

实现了的if转换，速度也是很慢的。此外，条件操作一旦被向量化，就需要选择执行形式（在 376
有掩码的硬件上执行或将操作压缩在一个密集的向量上），条件向量操作很容易比等价的标量操作运行得还慢，这取决于实际执行了多少个元素。而且，由于隐式地建立了向量掩码，受控制的变量和数组的存和取还会引起内存储层次结构和存储方面的问题。

带掩码的向量操作不总是向量机上简单的或快速的方法：掩码硬件的确使得掩码位的设置和检测都比较容易，但是一般都不支持直接操作在掩码上的存、取操作或逻辑操作。例如，控制Titan向量部件的掩码是一组特殊的位，任何指令都不能直接访问这些位。相应的标志向量比较和逻辑操作的结果可以设置这些掩码位；以后的标志向量操作可以使用它们。为了把一个向量掩码保存到内存中，需要首先做一个带条件的向量移动，把一个全“1”向量移到一个全零的寄存器中（这样那些掩码位为真的对应位的1就移动到寄存器），然后再把这个寄存器的值存到内存中去。同样，从内存中恢复一个向量掩码需要先把它从内存中取出，再和一个全“1”的向量进行比较。一旦设置掩码的向量操作执行完，这个掩码就不能直接被其他逻辑操作重新设置；代替的方法是必须把它取到一个向量寄存器，然后在其上进行操作。结果是，控制变量的“与”或“或”作为向量操作实现的代价是很高的。

基于这些考虑，加上Titan具有多个处理器用于有效地加速条件代码，Ardent编译器所执行的if转换是十分有限的。基本上，只有结构非常好的IF-THEN-ELSE语句或构成结构化的IF-THEN-ELSE的分支模式才被转换成控制条件的形式。这些语句的正确嵌套也被转换了，但这在Titan中证明是无用的，因为即使被很好地向量化了，一旦控制条件表示多于一个的IF条件，结果执行起来就比标量执行还慢。由于只转换了结构化的条件结构，化简过程就不需要了，而且if重构也就不重要了。所有其他的分支结构（包括C语言中允许的跳入循环的跳转）将由一种控制依赖的变形来处理，这将在第12章介绍。

7.6 历史评述与参考文献

第一个在向量机上对控制流变化的处理由Towle提出，他的处理方法通过维护一个位向量的守恒集合来模拟简单的控制流。本章介绍的if转换和布尔化简方案是Allen等[23]提出的；布尔化简中的关键结果是由Warren和Kennedy在PFC系统中建立的。Besaw[41]在Univac向量化编译器中实现了用类似布尔化简器的if转换。条件转换的有限形式是很多向量化编译器中条件 377
处理的基础[270，196]。

控制依赖在编译器方面的文献中有很长的历史，但是它在程序变换领域内的初步运用开始于Ferrante，Ottenstein和Warren的论文[112]。其中构造算法是Cytron等[97]提出的。

在存在控制流的情况下分布循环的算法由Kennedy和McKinley[174，209]提出，他们的算法是对Callahan与Kalem[58]和Dietz[102]的早期工作的改进。Towle[259]和Baxter与Bauer[39]提出了在向量化中由控制依赖的一种形式建立条件数组。本章介绍的代码生成算法基于Kennedy和McKinley[174，209]的算法，但是算法能生成结构化的代码，并克服了他们的算法在某些情况下可能产生错误代码的缺点。很多研究者触及到了“重构”非结构化代码的问题，包括Ferrante，Mace和Simons[110，111]，他们讨论存在控制依赖情况下的循环合并、死代码删除和分支删除。

习题

7.1 为图7-28中的控制流图构造控制依赖图。

7.2 在控制依赖存在的情况下，循环分布变换使用执行变量，执行变量可以取真、假和未定义三个值之一。为什么这里需要三个值而不仅仅是真、假两个值？给出一个为了保证正确性而需要未定义的例子。

7.3 在图7-29给出的控制依赖图中，虚线区域表示循环分布之后的理想区域。给出把语句分布在两个循环中之后的控制依赖图，并给出通过7.3.4节中“生成代码”部分描述的过程将生成的结果循环的代码。

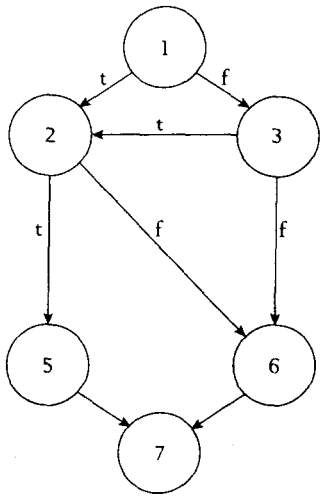


图7-28 习题7.1的控制依赖图

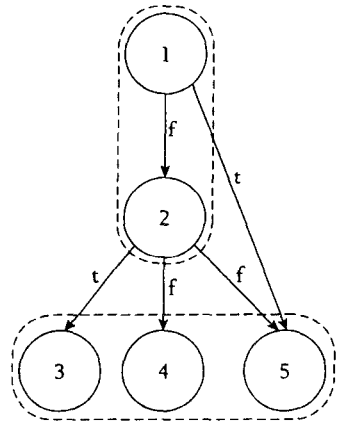


图7-29 习题7.3的控制依赖图

7.4 证明图7-9中的*ConstructCD*算法能够正确地构造控制依赖关系。换句话说，在算法执行之后，$y \in CD(x)$ 当且仅当x控制依赖于y。提示：证明应对算法中处理结点的顺序采用归纳法，这样可以保证在结点x在被处理之前，它的所有后控制结点都被处理过了。

378
~
379

第8章

改进寄存器的使用

8.1 引言

现代计算机系统的性能越来越取决于其存储器层次结构的性能。在芯片内操作的速度得到极大提高的同时，内存的性能却基本上保持不变。结果，以处理器周期计算的访存时延持续增加。要使机器的性能跟上处理器的性能，就必须缩短这些延迟。

在本章中，我们将讨论通过编译变换提高存储层次结构性能的若干方法中的第一种。当前使用的每一种计算机系统都有某种处理器寄存器集。在RISC结构上，寄存器特别重要，在这种结构中，除了取数和存数，所有的操作都要求操作数来自处理器中的寄存器，并且把所有结果写入寄存器。

大多数现代的处理器都有某种高速缓存存储器；但有的没有。没有高速缓存的机器包括向量处理器和Tera MTA系列机器（其中每一个线程都有自己的寄存器集）。我们将在第9章讨论高速缓存的编译器管理问题。

8.2 标量寄存器分配

在大多数编译器课程上，学生被告知标量寄存器分配的问题基本上已用寄存器着色技术解决了。这一技术由Chaitin和他在IBM研究院的同事提出[69，68]，后来又由Stanford大学的Chow和Hennessy[76]以及Rice大学的Briggs等[45]加以了改进。 381

这些技术试图为单个“活跃区间”（给定的变量在其中活跃的程序区域）中每个标量变量的所有引用分配一个寄存器。为此，编译器通常执行下面的步骤：

（1）确定变量的活跃区间并给每一活跃区间一个惟一的名字。

（2）建立干涉图表示哪些活跃区间不能分配同一寄存器。

（3）使用一种快速的启发式着色算法尝试为得到的干涉图着色，所用的颜色数代表可用的寄存器的个数。

（4）如果着色失败，编译器会选择至少一个活跃区间，取消分配给它的寄存器，然后重复步骤3，继续尝试着色。

就作者所知道的RISC处理器而言，其C和Fortran的编译器中均采用了这种方法。实践证明，这种方法是十分有效的。对于大多数小的例行程序，使用这种方法可以得到完美的分配，每一个非数组变量在其各个活跃区间均分配到一个寄存器。这样，人们通常认为单处理器中寄存器使用的效率已无太多改进的余地。

然而，这些常规的技术在处理浮点寄存器方面却行不通，这类寄存器通常用来暂时保存数组变量的单个元素。你可以把浮点寄存器看作一个很小的信息窗口，其大小恰等于数组中的一个字。为了得到最高的性能，我们必须在这个窗口移到另一个元素之前，尽可能多地使用其中的数据，因为每一次重用窗口中的元素就会减少一条取数指令的执行，而取数指令通常需要几十甚至上百个周期。

为了搞清楚这些技术的重要性，我们举一个简单的例子，这个例子可以看作矩阵乘法计算的一个抽象：

```
DO I=1, N
   DO J=1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

在这个例子未被广泛理解之前，几乎所有的产品编译器都不能识别出A(I)在内层循环中可以被保留在一个寄存器中，尽管很容易看出A(I)的地址在那个循环中没有改变。不能识别的原因在于单处理器的编译器只能很好地处理标量变量的寄存器分配——数组访问会被作为表达式在循环的每一次迭代中求值。尽管编译器可以使用强度消减来识别出地址没有改变，但是
382 只有极少的编译器会扩展强度消减以消除取数指令。

现在考虑把前面的例子稍微改变一下：

```
DO I=1, N
   T = A(I)
   DO J=1, M
      T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

这个版本在执行内层循环的操作时，用一个标量变量暂时存放A(I)的值。所有使用图着色寄存器分配方法的编译器都能正确地为这个标量变量分配一个寄存器。因此，第二个循环会比第一个循环快得多，尽管它们看上去只有表面上的不同。

这一观察导致一种叫做*标量替换*的优化方法，这种方法通过将数组引用转换为标量引用来提高编译器基于着色的寄存器分配器的效能。虽然标量替换是作为一种编译优化方法提出的，但实际上可被理解为是一种源-源变换。

在下面的几小节中，我们将介绍多种提高单处理器存储层次结构性能的源-源变换方法，以及实现它们的算法。

8.2.1 面向寄存器重用的数据依赖

到目前为止，本书重点讨论了作为重排序变换的安全约束的数据依赖。在这样的背景中，依赖是一种必须满足的操作约束，因为两个不同的语句实例可能访问同一个内存单元。就这一点来说，程序的依赖越少越好，因为越少的约束意味着编译器有更大的空间来对语句重排序。对于寄存器（和数据）的重用，我们将以另外的方式来利用依赖。一个准确的数据依赖代表两个访问同一内存单元的不同的语句实例。因此依赖就可以被看成指示那些经常被重用的存储单元，应保存在存储层次结构中最快部分。就这一点来说，一个程序中的依赖越多就越好，因为越多的依赖意味着越多的重用。依赖的两种不同的用法使其适用于各式各样的程序的优化。

为了弄明白在寄存器分配中是怎样使用数据依赖的，我们考虑各种不同的数据依赖：

- *真依赖*或*流依赖*，表明值在源点被计算且在汇点被使用。如果这个值能一直保持在寄存
383 器中，直至到达汇点，那么就不需要从内存中读取。同样地，如果包含被引用单元的块一直留在高速缓存中，直至到达汇点，那么就可以避免高速缓存不命中。

- 反依赖，其中变量的引用是在对其赋值之前，因此无法用反依赖来改进寄存器的分配。但是，如果在源点处被装入的块一直保留在高速缓存中直至汇点的话，则可以避免高速缓存不命中。
- 输出依赖，在高速缓存管理中也可能是有用的，但它对改进寄存器使用有特殊的用途。考虑下面的例子:

```
S1   A(I) =
     ...
S2   =A(I)
     ...
S3   A(I) =
```

其中在S_1和S_3之间有一个输出依赖，在S_1和S_2之间有一个真依赖。这个真依赖允许我们消除语句S_2中的一个取数，但输出依赖告诉我们A(I)在S_2中使用以后其值不再使用，不需要保存。

- 为了达到存储层次结构的目的，我们要使用第四种依赖，称为输入依赖，在这种依赖中，源点和汇点使用同一个单元。

```
S1   = A(I)
     ...
S2   = A(I)
```

从S_1到S_2的输入依赖不会导致执行约束，却会提供机会消除第二个引用中的取数。可以简单地修改标准的依赖测试过程来计算输入依赖。

回忆一下，我们提到过内存访问的类型是依赖的一个特性，这导致上面列出的重用的原理。然而，依赖还有一个特性：循环携带和循环无关。这个特性也导致在下一小节中讨论的重用的原理。

8.2.2 循环携带和循环无关的重用

非常明显，循环无关依赖可以作为对值的重用的指南。例如循环无关的真依赖

```
S1   A(I) =
S2       = A(I)
```

384

可以重新写为

```
S1  t =
S2    = t
```

只要确切地知道依赖，且两条语句之间没有其他存数语句。对第二种形式采用通常的标量寄存器分配就可以了。同样，循环无关输出依赖可以用来消除不必要的存数，循环无关输入依赖可以用来回避不必要的取数。

初看起来，循环携带依赖无法提供这样一种明显的重用机制。然而仔细研究会发现：阈值较小的后向循环携带依赖可以有与循环无关依赖相同的作用。例如

```
S2       = A(I)
S1   A(I+1) =
```

可以重新写为

```
S2    =t
S1  t=
```

同样，一个好的标量寄存器分配器应该能正确处理这段代码。在这个简单的检查中，循环携带依赖看来只能处理出现在源点（包括固有的源语句）之前的汇点，否则在下一次迭代中计算的新值将覆盖所需要的来自前次迭代的值。然而，除此之外它可以被扩展，如后面几小节中的说明。类似的变换对输入和输出依赖也同样有效。

如果循环携带依赖对寄存器重用有用，就必须使用一些约束。回忆一下，每一个循环携带依赖都有一个*阈值*，这个阈值正好是携带此依赖的循环的依赖距离。如果这个阈值在整个循环中始终是常数（即它在每一次迭代中都是相同的），就说它是*一致的*。具有一致阈值的循环携带依赖称为*一致依赖*。要对存储管理有用，一个携带的依赖就必须是一致依赖。实际上，带有小的编译时间常数的阈值的循环携带依赖是消除内存引用的最佳候选，因为他们需要用于保存依赖源点和汇
385 点之间的值的寄存器最少。注意，循环无关依赖若要对内存管理有用，也必须满足一致性要求。

8.2.3 寄存器分配的例子

尽管前一节所讲的原理十分简单，但它们在实际中却非常有用。下面的求和归约循环显示这些原理有什么样的用处:

```
DO I=1, N
   DO J=1, M
      A(I)= A(I)+B(J)
   ENDDO
ENDDO
```

这里，我们发现从循环中的语句到其自身有一个真依赖和一个输出依赖，它们都来自对A(I)的引用，并且它们两个都由J-循环携带。另外，从该语句到其自身有一个反依赖，也是由J-循环携带。最后，由于对B(J)的引用引发I-循环携带一个输入依赖。图8-1显示了这些依赖。

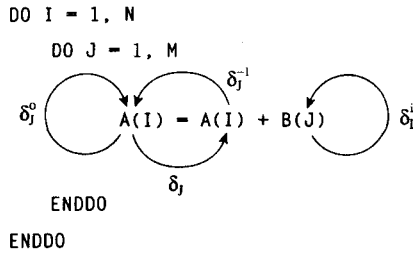


图8-1 带依赖的归约例子

在这个例子中，由J-循环携带的真依赖指出，如果A(I)能够一直保存在一个寄存器直到下一次循环迭代，就可以省去一个取数操作，而输出依赖和反依赖则显示在每一次迭代中最初的取数和存数操作完全可以移出循环。

变换之后的循环在每一次迭代中将只需要为B(J)作一次取数:

```
DO I=1, N
   T = A(I)
   DO J=1, M
      T = T + B(J)
   ENDDO
   A(I) = T
386 ENDDO
```

如前面所讲的，我们使用下标数组变量对标量的赋值来表示取数，因为大多数标量寄存器分配过程会将所有的标量变量放到寄存器中，至少在单一子程序里是这样的。

8.3 标量替换

在前面讨论过的简单原理的基础上，这一节讨论实施标量替换的算法。这个过程的第一

步是要删除依赖图中的那些无关依赖，以便（剩下的）每一个依赖都能够用来精确度量内存重用的收益。后面几小节我们将把这些原理加以扩展，用来处理更一般的依赖。

8.3.1 依赖图剪枝

在本节中，我们将介绍一种依赖图剪枝的算法，使每一条依赖边代表一种消除取数或存数操作的可能。用下面的代码来说明这个问题:

```
      DO I = 1, N
S1      A(I+1) = A(I-1) + B(I-1)
S2      A(I) = A(I)+B(I) + B(I+1)
      ENDDO
```

在这个例子中的A和B的依赖模式是有趣的。对语句S_1中A(I+1)的赋值能够到达语句S_2中的A(I)的使用，但不能到达语句S_1中A(I-1)的使用，因为它会被语句S_2中的赋值覆盖。S_2中对B(I+1)的引用所使用的单元在后面一次迭代的S_2中重用，在第三次迭代的S_1中再次重用。

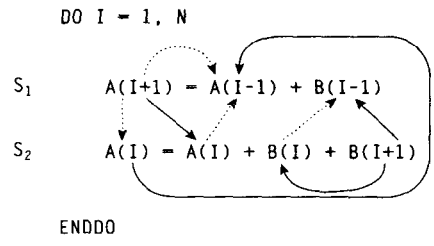


图8-2 剪枝的作用

图8-2是剪枝前后的依赖图。在剪枝后留下的边用实线表示。虚线表示用常规测试方法产生的边，但在剪枝后不再存在。

重要的是要注意到，在被剪枝的图中每一个引用最多只有一个前驱，且每一条边代表省去一次潜在的内存访问。在一组中所有边的源点称为那个组的母点。在这个例子中的三个母点分别是两个赋值语句的左部和语句S_2中B(I+1)的引用。

如果我们把所有与母点有关的引用都赋给一个惟一的临时标量，则在标量替换后将得到下面的代码:

```
      t0A = A(0); t1A0 = A(1); tB1 = B(0); tB2 = B(1)
      DO I = 1, N
S1      t1A1 = t0A + tB1
        tB3 = B(I+1)
S2      t0A = t1A0 + tB3 + tB2
        A(I) = t0A;
        t1A0 = t1A1; tB1 = tB2; tB2 = tB3
      ENDDO
      A(N + 1) = t1A1
```

这段代码在主循环中只有一次取数和存数。原来的代码中却有5次使用引用，因为剪枝后的依赖图中的三条边，除其中之一外的所有的引用都被消除。另外，由于从A(I+1)到A(I)的输出依赖使得A(I+1)的存数只在最后一次迭代之后才是必要的，所以消除了一次存数。

为了对图8-2描绘的图作剪枝，我们必须消除两种边:

（1）不代表潜在的重用的流依赖边和输入依赖边，因为源点所产生的值被插入的对同一单元的赋值所覆盖。在语句S_2中的从A(I+1)到A(1-1)的流依赖是这种需要删除的依赖的一个例子。

（2）由于另外的引用（通常是对两端点都有依赖的母点）而成为多余的输入依赖边。语句S_2中的B(I)到S_1中的B(I-1)的输入依赖是第二种需要删除依赖的例子。

387
~
388

注意输出依赖和反依赖不能直接导致寄存器的重用，所以总是被删除。

这启示我们提出一个剪枝依赖边的三步骤算法。

*第一步：消除被覆盖的依赖。*这一步删除所有这样的依赖，在其两端点间有一个对该依赖涉及的单元的存数操作。如果被删除的依赖是流依赖，可以如下确定一个覆盖存储的操作：从依赖的源点到赋值有一个输出依赖，而从赋值到依赖的汇点存在一个流依赖。

```
S1    A(I+1) =
      ...
S2    A(I) =
      ...
S3    ... = A(I)
```

这里，从S_1到S_3的流依赖将被删除。

如果将要删除的依赖是一个被循环中的存数操作所覆盖的输入依赖，则从源点到该操作将有一个反依赖，而从该操作到依赖的汇点有一个流依赖。

```
S1    ...= A(I+1)
      ...
S2    A(I) =
      ...
S3    ... = A(I-1)
```

这里，从S_1到S_3的输入依赖将被删除。

*第二步：识别母点。*在这一步中，所有的母点将被识别出来。在剪枝后的依赖图中，母点是带有至少一个从它到循环中另一语句的流依赖的赋值引用，或者是这样一个使用引用，至少有一个从它发源的输入依赖而且没有输入依赖或流依赖到达它。

*第三步：查找名字划分和消除输入依赖。*从每一个母点开始，对于从该母点沿流依赖或者输入依赖可以到达母点的每一个引用，我们将它们标记为那个变量的名字划分的一部分。名字划分是一引用集合，这些引用可以用对单个标量变量的引用来替换。如果依赖源点不是母点本身，则可以消除同一名字划分中的两个元素之间的任何输入依赖。

这种分析会给我们带来某种意想不到的结果：如果我们把每一次引用当作一个顶点，把每一条边看作连接两个引用，那么整个查找名字划分的过程可以被看作带类型的合并问题(6.2.5节)。在这一形式化表述中，我们用引用中的数组的名字作为类型，而把输出依赖和反依赖定义为坏边。当这种带类型的合并算法执行时，每一个合并的结点代表一个不同的名字划分，被添加到合并结点的初始结点就是母点。

在这个框架中，另外还有两个复杂的问题必须处理。首先，可能有某些引用是在循环的依赖环中。注意，这里的引用和语句之间的区别是很重要的。下面是一个例子：

```
DO I = 1, N
   A(J) = B(I) + C(I, J)
   C(I, J) = A(J) + D(I)
ENDDO
```

在这个例子中，带类型的合并算法不会访问A(J)，因为它位于图8-3中所示的一个依赖环中。

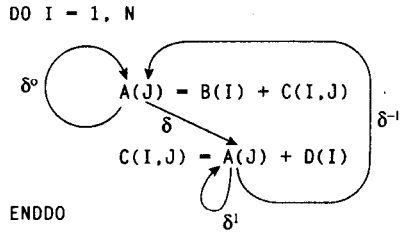


图8-3 依赖环中的引用

为了解决这个问题，我们注意到任何这样的环中

的引用都可能被赋给循环中的一个标量。这里不要求复制标量:

```
DO I = 1, N
   tA = B(I) + C(I, J)
   C(I, J) = tA + D(I)
ENDDO
A(J) = tA
```

由于对自身的赋值导致的输出依赖，最后的赋值可以被移出循环。(注意这段代码假定$N \geqslant 1$。)如果A(J)有一个向上暴露的使用，则需要在循环前插入一个对tA的赋值。

第二个复杂的问题是可能出现不一致的依赖。对于一个代表重用的依赖来说，它必定是一致的，因为在循环的迭代中它有一个常数的阈值。下面是一个例子，说明不一致的依赖:

389 ~ 390

```
      DO I = 1, N
S        A(2*I) = A(I) + B(I)
      ENDDO
```

尽管由于对A的引用而造成从S到它本身有一个流依赖，这个阈值随每次迭代改变，所以这个依赖是不一致的，故不适于重用。在上面带类型合并的框架结构中，不一致的依赖可以被标记为坏边。这会使得对相同数组的引用间的任一不一致依赖强制插入一个取数和一个存数操作。同样，对于依赖环中的引用，环应该用一个坏边打破，以强制插入取数和存数。以下面的循环为例:

```
      DO I = 1, N
S1       A(I) = A(I-1) + B(I)
S2       A(J) = A(J) + A(I)
      ENDDO
```

这里，在S_1中对A(I)的引用可以用一个标量替换，但是对A(I-1)的引用必须进行取数，因为我们必须假定对A(J)的存数会改变A(I)的值。这就导致下面的代码:

```
      DO I = 1, N
S1       tAI = A(I-1) + B(I)
         A(I) = tAI
S2       A(J) = A(J) + tAI
      ENDDO
```

这段代码可以通过使用索引集分裂的方法作出重大改进:

```
      tAI = A(0); tAJ = A(J)
      JU = MAX(J-1, 0)
      DO I = 1, JU
S1       tAI = tAI + B(I); A(I) = tAI
S2       tAJ = tAJ + tAI
      ENDDO
      IF (J.GT.0.AND.J.LE.N) THEN
         tAI = tAI + B(I); A(I) = tAI
         tAJ = tAI! can be forward-substituted
         tAJ = tAJ + tAI; tAI = tAJ
      ENDIF
      DO I = JU + 2, N
         tAI = tAI + B(I); A(I) = tAI
         tAJ = tAJ + tAI
```

```
    ENDDO
391 A(J) = tAJ
```

第二段代码在每一次迭代中要比原来少两个取数和一个存数。

名字划分是标量替换的基本的输入。不过，在我们介绍其在8.3.5节中的算法之前，先介绍一种实际的改进，克服现有标量寄存器分配方法的一些局限性。

8.3.2 简单替换

我们现在转向讨论标量替换的一般过程。我们从一个剪枝后的依赖图开始，这种图在前一节中已经介绍过。这个图中，每一个真依赖或者输入依赖都表示恰好消除一次取数或一次存数操作的机会。如果依赖是循环无关的，那么用对生成的标量的引用来替换第二个引用通常是没有问题的。例如，在下面的代码中，变换是简单的:

```
DO I = 1, N
   A(I) = B(I) + C
   X(I) = A(I)*Q
ENDDO
```

在剪枝后的依赖图中，从循环体的第一个语句到第二个语句有一个循环无关的依赖。在这种情况下，标量替换要求将第一个语句的计算结果保存在一个标量t[⊖]中，插入一个标量到A(I)的复制语句，以及在第二个语句中把对A(I)的引用替换为对t的引用，结果如下:

```
DO I = 1, N
   t = B(I) + C
   A(I) = t
   X(I) = t*Q
ENDDO
```

如果在此循环中t被分配到一个寄存器，如我们期望在大多数编译器中看到的那样，那么每一次迭代将比原来少一次取数。

8.3.3 处理循环携带依赖

这个一般的过程适用于任何依赖或者跨距小于一个循环迭代的一组依赖——在这组依赖
392 中，第一个和最后一个语句或者在同一个迭代中，或者如果在相邻的两个迭代中，则最后一个语句按词法顺序不会出现在这组的第一个语句之后。下面是一个例子:

```
DO I = 1, N
   A(I) = B(I-1)
   B(I) = A(I) + C(I)
ENDDO
```

通过引入两个标量tA和tB，并在循环之前插入tB的初始化，我们得到下面的结果:

```
tB = B(0)
DO I = 1, N
   tA = tB
   A(I) = tA
   tB = tA + C(I)
   B(I) = tB
ENDDO
```

⊖ 讨论中我们将总是用小写字母的标量变量（如像t）表示编译器引进的标量变量用于对寄存器赋值。

注意一个好的标量寄存器分配器会将标量tA和tB合并到同一个寄存器。然而，我们让标量编译器来作决定，因为这个问题可能与操作的时序相关，而这是机器相关的，这里不作讨论。

8.3.4 跨越多个迭代的依赖

如果生成的依赖跨越一个或多个完整的循环迭代，标量替换将变得更加复杂：

```
DO I = 1, N
   A(I) = B(I-1) + B(I+1)
ENDDO
```

在这个例子中，从B(I+1)到B(1-1)有一个输入依赖，因为一次迭代中对B(I+1)的引用和后两次迭代中对B(I-1)的引用都涉及相同的内存单元。这样，依赖的距离就是两个迭代。

跨越多个迭代的依赖的问题在于消除取数操作需要不止一个临时变量。这是因为在任何特定的迭代中计算出来的值需要被保存在寄存器中，直到下一个迭代中第一次引用此相关值。当前的例子跨越了两个迭代，我们将使用如下定义的三个不同的临时标量： 393

```
t1 = B(I-1)
t2 = B(I)
t3 = B(I+1)
```

这些关系可以认为是“循环不变”的，即在循环的每一次迭代中都成立。现在我们可以考虑应生成什么样的代码来消除依赖汇点中的取数操作：

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
   t3 = B(I+1)
   A(I) = t1+t3
   t1 = t2
   t2 = t3
ENDDO
```

8.3.5 删除标量拷贝

当这段代码实现减少取数的目标时，它却要付出代价，那就是引入了两条会被编译器转换成寄存器-寄存器拷贝的标量拷贝指令。尽管这样的拷贝代价很小，但仍需要指令发射槽，因而会造成不必要的性能下降。因为这些拷贝操作实际上是在置换机器寄存器中的值，因此我们可以通过展开置换的环的长度来消除对拷贝的需要，这样将产生下面的代码：

```
t1 = B(0)
t2 = B(1)
mN3 = MOD(N, 3)
DO I = 1, mN3
   t3 = B(I+1)
   A(I) = t1+t3
   t1 = t2
   t2 = t3
ENDDO
DO I = mN3+1, N, 3
   t3 = B(I+1)
   A(I) = t1+t3
   t1 = B(I+2)
```

```
      A(I+1) = t2+t1
      t2 = B(I+3)
      A(I+2) = t3+t2
394   ENDDO
```

第一个循环称为前置循环，它保证当进入主循环时，剩下的迭代次数是3的倍数，这样就不需要在主循环中对循环结束的条件进行特殊的测试。确切地说，前置循环是与前面讨论的简单解决方法一样。然而，现在在前置循环最多进行两次迭代后就会进入主循环，且主循环不需要寄存器-寄存器的拷贝，以及较少的循环结束条件测试。要证明最后的循环可以计算出所需要的答案是很容易的。

8.3.6 缓解寄存器压力

理想情况下，标量替换过程应该系统地应用于循环中的所有名字划分。但实际上，这会产生许多标量，竞争有限的浮点寄存器。这种过载是大多数标量寄存器分配器无法处理的。因此，对标量替换系统来说，限制产生的标量的个数少于可用的标量寄存器个数通常更好。

为此，系统对每一个名字划分R附加两个参数：

（1）名字划分的值$v(R)$：等于用对寄存器常驻标量的引用替换R中的每一个引用所减少的内存取数或存数操作的个数。

（2）名字划分的代价$c(R)$：是在消除R中的引用时需要用来存放所有临时标量值的寄存器的个数。

假定可用的寄存器的个数是n，我们需要的结果是引用集的一个子集$\{R_1, R_2, \cdots, R_m\}$，满足

$$\sum_{i=1}^{m} c(R_i) \leqslant n$$

且使得可被消除的内存访问的总数

$$\sum_{i=1}^{m} v(R_i)$$

取最大值。容易看出这是经典的装箱问题通称背包问题的一个实例。在最一般的形式下，装箱问题是*NP*完全的。但是，0-1背包问题可以利用图8-4所示的动态规划算法在多项式时间内解决。

```
procedure Pack(v, c, M, n, L, m)

  // v [1:M]是M个名字划分的值集。
  // c [1:M]是M个名字划分的代价集。
  // n是可供数组量使用的寄存器数目。

  // L [1:m]列出最佳装箱中名字划分的下标。

  // BP [0:n, 0:M]是一个概率矩阵，其中
  // BP [i, j]是对一个大小为i的箱仅使用引用集1到j的最佳可能装箱的值。

  for j := 0 to M do begin BP [0, j] := 0;  last[0, j] := 0 end
  for i := 1 to n do begin BP [i, 0] := 0;  last[i, 0] := 0 end

  for j := 1 to M do begin
    for i := 1 to n do begin
      BP [i, j] := BP [i, j - 1];  last[i, j] := last[i, j - 1];
```

图8-4 寄存器压力缓解

```
            if i - c[j] > 0 then
              if BP [i, j - 1] < BP[i - c[j], j] + v [j] then begin
                BP [i, j] : = BP[i - c[j], j] + v [j];
                last [i, j] : = j;
              end
          end
        end
        // 现在打开包含的索引的列表并存入l中
        l : = last [n, M]; m : = 0; isize : = M;
        while l ≠ 0 do begin
          m : = m + 1; L[m] : = l;
          sizeleft : = sizeleft - c[l];
          l : = last[sizeleft, l - 1];
        end
      end Pack
```

图 8-4（续）

众所周知，这个过程需要$O(nM)$个步骤。然而，如果这太耗时而无法在编译器中实现的话，还有一个好的启发式算法。如果对引用集按照$v(R)/c(R)$的比值结果从大到小进行排序，我们可以从表的起始选择元素直到寄存器用光为止。在实践中，这种启发式算法非常有效。

8.3.7 标量替换算法

我们现在准备介绍一种简单的针对不包含条件控制流的循环的标量替换算法（图8-5～图8-9）。这种算法可以粗略地划分为下面五个步骤：

395～396

```
procedure ScalarReplace(L, G, n)
  // L是我们要为之生成代码的循环嵌套。
  // G是L中语句间的增广的依赖图。
  // n是可供数组量使用的寄存器数目。
  对依赖图作带类型的合并，同时将输出依赖、反依赖和不一致依赖标记为坏边，产生一组名字划分P;
  使用装箱算法选择P的一组子集{p1, p2, ..., pm}，在使用不超过n个寄存器的前提下，最大化减少访存次数;
  for i : = 1 to m do begin
    if pi是一个无环划分then
        ScalarReplacePartition(pi, ki);
    else begin
        ScalarReplaceCyclicPartition(pi) ;
        ki : = 1;
    end
  for each循环中不一致依赖 δ do
        InsertMemoryRefs(δ);
  end
  // ki是标量替换引入的临时变量的数目
  令K是{k1, k2, ..., km}的最小公倍数;
  // 将循环展开为K个循环体以消除标量拷贝
  UnrollLoop(K);
end ScalarReplace
```

图8-5 标量替换

```
procedure ScalarReplacePartition(g, k)
  // g是名字划分中的依赖的集合。
  令k表示集合g所跨越的叠代的总数;
  引入惟一的临时量 {t1, t2, ..., tk};
  令I表示将要替换的循环的索引变量;
  令R(I + l) 表示在索引I的位置上下标I + l的组中的引用;
  令j等于g中下标引用中加到I上的最大的值;
    // 这意味着j - k + 1是最小的加数
  for each g中的每一个下标引用R(I + l) do begin
    用t(j-k+1)替换该下标引用;
    if该引用位于赋值语句左部then
      if没有到循环中另一个赋值的引用的输出依赖then
        在含有该引用的语句之后插入一条赋值语句 "R(I + l) = t(j-k+1)" ;
      else begin // 存在输出依赖
        令R(I + q) 为输出依赖的汇点引用;
        if q < l then
          for i: = q + 1 to l do
            在循环后插入赋值语句 "R(N + i) = t(j-k+i)" ; // N = 循环上界
      end
  end
  令l是组内某个下标引用中I的最大的加数，该引用有一个来自循环的流依赖;
  for i: = 1 to j - k + l do
    在循环开始前插入 "t(i) = R(i)" ;
  for i: = 1 to k - 1 do
  在循环的结束前插入 "t(i) = t(i+1);
end ScalarReplacePartition
```

图8-6 无环依赖集的标量替换

```
procedure ScalarReplaceCyclicPartition(g)
  // g是名字划分中的依赖的集合。
  if涉及g的引用不是循环不变的then
    不作任何替换return;
  令R是名字划分的循环不变的引用;
  令t是一个惟一的临时变量;
  for each g中的下标引用R do用t替换R;
  if至少有一个引用是位于赋值语句的左部then
    在该循环后插入 "R = t" ;
  if循环中有R的向上暴露的引用then
    在该循环前插入 "t = R" ;
end ScalarReplaceCyclicPartition
```

图8-7 有环依赖集的标量替换

```
procedure InsertMemoryRefs(δ)
  // δ是不一致依赖
  if δ连接一个循环改变的名字划分和一个循环不变的名字划分then
  begin // 使用索引集分裂解决此问题
    将循环划分为三部分:
      a) 到包含δ中涉及的循环不变引用的叠代为止(不包含该叠代)的所有叠代
      b) 包含该引用的叠代
      c) 位于包含该引用的叠代之后的叠代;
    对所得到的三个循环作标量替换;
  end
  else if该依赖是一个流依赖then begin
    插入一个对将源点临时变量存入相应的引用的存数操作;
    令R是循环中源点之后的汇点的名字划分中最早的引用;
    如果在该引用前没有对R的临时变量的取数操作,则插入一个这样的操作;
  end
  else if该依赖是一个输入依赖then begin
    令R2是当前源点引用之后的第一个对汇点的引用;
    令R1是R2之前的最后一个对源点的引用;
    如果不冗余的话,在R1后插入一个从R1的临时变量到R1的存数操作;
    如果不冗余的话,在R2后插入一个从R2到R2的临时变量的取数操作;
  end
end InsertMemoryRefs
```

图8-8 为不一致依赖插入存储引用

```
procedure UnrollLoop(K)
  // K是展开因子
  将循环分裂为两个循环,一个是前置循环
      DO I = 1, MOD(N, K)
    以及一个主循环
      DO I = MOD(N, K) + 1, N;

  消除主循环中所有的寄存器到寄存器的拷贝;
  展开主循环,使其包含K个原循环体的拷贝,并使其步长为K:
      DO I = MOD(N, K) + 1, N, K;
  // 如在所有的展开索引中一样,I在循环体的第p个拷贝中被替换为I + p

  在循环体的第q个拷贝(第一个拷贝是零号拷贝)中,将对惟一生成的常数ti替换
  为对tMOD(i+q-1,k)+1的引用,其中k是ti所在的临时组中最大的索引;

end UnrollLoop
```

图8-9 展开消除拷贝

(1) 首先用带类型的合并找出如8.3.1节中讲的一组名字划分。我们利用一个临时变量替换一组中的所有引用,或者复制一组临时变量来覆盖多次迭代。

(2) 缓解寄存器压力,选择能够放入可用寄存器的名字划分并最大限度地减少内存引用的个数。可以使用装箱算法或者在8.3.6中介绍的相关启发式算法。

（3）用对标量的引用替换每一个被选择的名字划分：

a）如果它是无环划分，就用一组惟一的临时变量来替换，该集合所跨越的每次迭代都用其中一个引用替换掉，如8.3.4节所示。在可能的情况下，利用输出依赖将存数操作移出循环，利用输入依赖把取数移出循环。

b）如果它是环状划分，就用一个临时变量替换所有的引用。

c）对不一致的依赖来说，可以用索引集分裂来确保正确的值被传递，或者在循环中插入取数和存数。

（4）最后，作循环展开以消除8.3.4节中所示的寄存器-寄存器拷贝操作。展开次数等于由选定的组中的各集合跨越迭代的个数的最小公倍数。

8.3.8 实验数据

为了说明标量替换的效果，这里我们报告它在众多流行的基准测试程序的核心和程序上的应用结果。这里给出的结果来自Carr的学位论文[64，67]。在实验中，他运行了程序的两个版本——原始的版本和经过标量替换而得到的新版本。这两种版本都在IBM RS/6000 540型机器上编译和运行。用原先的版本的运行时间除以使用标量替换后的新版本的运行时间的比值代表加速比。

图8-10显示在著名的Livermore循环中得到的加速比。图中仅给出有加速比的循环；其他所有的循环都没有变化。在一个有非常好的寄存分配器的编译器上，我们可以看到改进的范围从一般的1.03到令人吃惊的2.67。注意，除非目标机的编译器过于简单，否则标量替换绝不会导致性能损失。

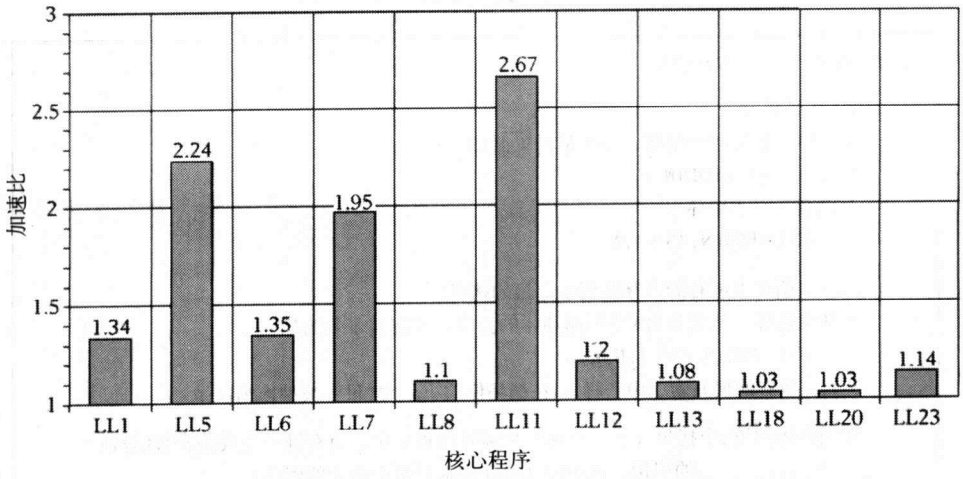


图8-10 对Livermore循环的标量替换

Carr也在许多著名的核心程序上测试了标量替换，比如LAPACK[29]中实现的LU分解，若干个NAS核心程序，以及他在Rice找到的一些核心程序[表8-1]。图8-11中显示的结果包括有选主元和不用选主元的LU分解的核心，并涵盖了点算法版本和分块的版本。这些核心程序都包含要用依赖分析来检测的不变数组引用，因此在只能处理标量的编译器里不可能得到这样的加速比。在Seval和Sor上得到两个异常的性能，是由于标量替换成功地优化了一个计算密集的循环，而几乎所有的运行时间都集中于此。

表8-1 测试核心的存储变换

程 序 包	核 心	描 述
Lin Alg	MM	矩阵乘法
	LU	LU分解
	LUP	带选主元的LU分解
	BLU	块LU分解
	BLUP	带选主元的块LU分解
NAS	Vpenta	五对角矩阵求逆
	Emit	涡流生成
	Gmtry	解涡流生成的高斯消去法
Geophysics	Fold	卷积
	Afold	卷积
Local	Seval	B-样条求值
	Sor	连续超松弛

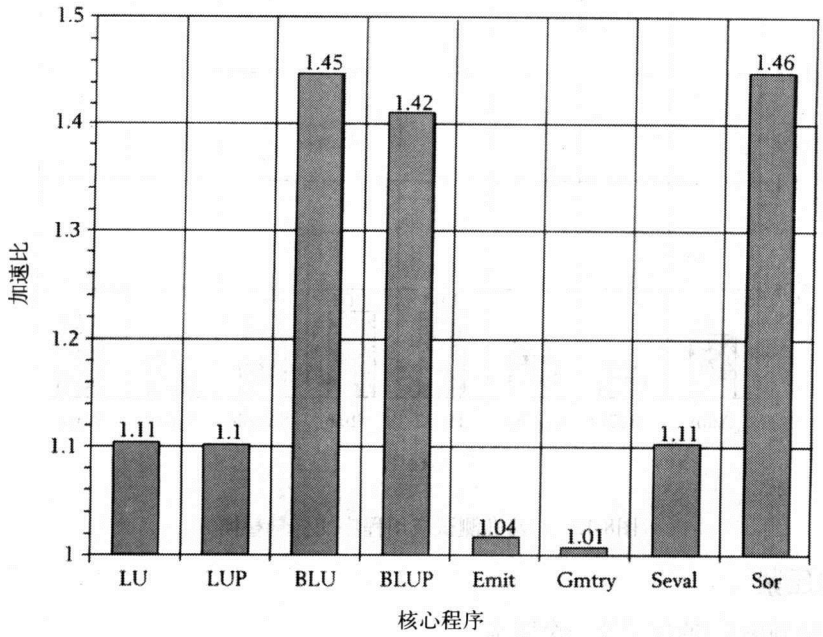


图8-11 对线性代数核心程序的标量替换

Carr也在许多基准测试应用程序[表8-2]上测试了标量替换。这些程序选自著名的程序包SPEC，Perfect，RiCEPS（Rice大学收集的一组编译器基准测试程序）以及另外三个来自Rice但不属于RiCEPS的应用程序。

表8-2 测试应用程序的存储变换

程 序 包	核 心	描 述
SPEC	Matrix300	矩阵乘法
	Tomcatv	网格生成
Perfect	Adm	伪光谱空气污染
	Arc2D	2维流体流量求解器

（续）

程 序 包	核 心	描 述
RiCEPS	Flo52	超音速无粘性流
	Onedim	时间无关的薛定鄂方程
	Shal	天气预报
	Simple	2维流体力学
	Sphot	粒子迁移
	Wave	电磁粒子模拟
Local	CoOpt	石油勘探

在图8-12中给出这些应用在标量替换之后的性能。尽管其中的一些循环得到了显著的加速比，但标量替换对这些应用的总的改进不大（没有得到改进的程序未列出）。

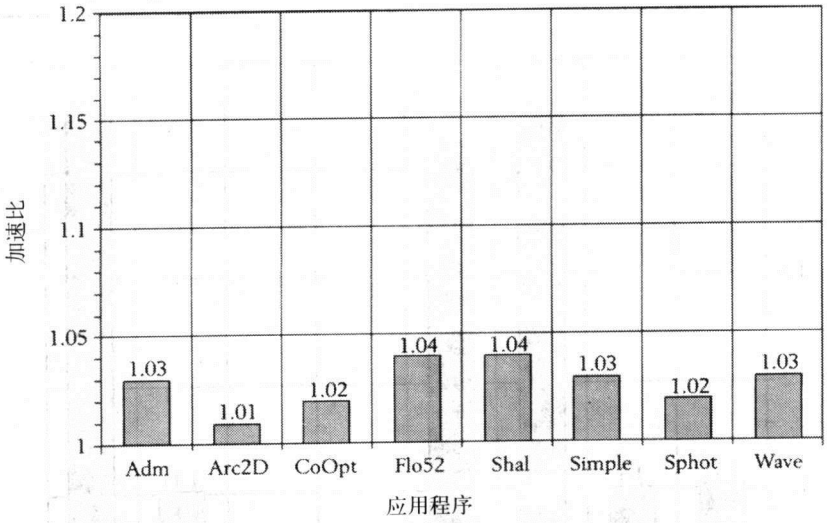


图8-12 对基准测试应用程序的标量替换

8.4 展开和压紧

我们现在回到图8-1的例子的一个版本：

```
DO I = 1, N*2
   DO J = 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

397 ~ 403

基于内层循环所携带的依赖，标量替换可以最大限度地减少对A的引用，且效果非常好。然而，外层循环也携带了一个对B的输入依赖，因此从那个依赖也可以得到一些重用的机会。

如现在构造的循环，我们不大可能得到重用，因为在J-循环的各次迭代中，B(J)的一个特定的值必须保留在一个寄存器里，直到I-循环的下一次迭代。既然J-循环的迭代次数N未知，我们必须假定任何一个给定的机器都没有足够的寄存器来得到对B更多的重用。然而，如果作一个简单的称为*展开和压紧*的变换，我们就能让各对迭代更紧密：

```
DO I  = 1, N*2, 2
   DO J = 1, M
      A(I) = A(I) + B(J)
      A(I+1) = A(I+1) + B(J)
   ENDDO
ENDDO
```

这个变换的本质是把外层循环展开为多个迭代，然后合并内层循环的副本。通过这个变转换，我们已将对B(J)的两次引使用放在一起，因此循环的这一新版本对每两次引用只对B(J)作一次取数操作。如果用两个标量来存放A(I)和A(I+1)的值，用一个标量来存放B(J)的值，那么在整个内层循环中我们将需要3个寄存器:

```
DO I = 1, N*2, 2
   s0 = A(I)
   s1 = A(I+1)
   DO J = 1, M
      t = B(J)
      s0 = s0+t
      s1 = s1+t
   ENDDO
   A(I) = s0
   A(I+1) = s1
ENDDO
```

内层循环现在对每两个浮点加法只需要一次取数。虽然内层循环总共仍然需要M次取数，I循环的执行次数减少为原来的一半。因此以取数操作的个数计算的总的计算代价也只是原来的一半。展开因素多于2可以节省更多的代价。

展开和压紧还可用于改进流水线功能部件的效能。考虑下面的循环: 404

```
DO J = 1, M*2
   DO I = 1, N
      A(I, J) = A(I+1, J) + A(I-1, J)
   ENDDO
ENDDO
```

这里，内层I循环的一次迭代计算出的值作为下一次迭代中计算的输入。这样尽管提供了好的重用，但导致执行流水线的问题。如果功能部件流水线包含两级，下一次迭代就需要从当前迭代开始起等待两个周期才能开始。因此执行这个循环嵌套的总的时间最少也要2*M*N个周期。

另一方面，如果对这个相同的例子使用展开和压紧，我们就得到

```
    DO J = 1, M*2, 2
       DO I = 1, N
S1        A(I, J) = A(I+1, J) + A(I-1, J)
S2        A(I, J+1) = A(I+1, J+1) + A(I-1, J+1)
       ENDDO
    ENDDO
```

这时，我们有两个共享同一个功能部件的独立的依赖环。因此，该功能部件能在这两个依赖环上交替进行工作，在执行语句S_1之后的周期，启动语句S_2的加法运算。这样，这两个依赖环只需要花费原来计算其中之一所需的时间就可以全部完成。既然外层循环只有一半的迭代，

因此执行循环嵌套就只需要一半的时间。在标量替换后，这段代码就变为如下的形式:

```
      DO J = 1, M*2, 2
         s0 = A(0, J)
         s1 = A(0, J+1)
         DO I = 1, N
S1          s1 = A(I+1, J) + s1
S2          s0 = A(I+1, J+1) + s0
            A(I, J) = s0
            A(I+1, J) = s1
         ENDDO
      ENDDO
```

405 对每一对浮点操作来说，内层循环要执行两次取数和两次存数。

8.4.1 展开和压紧的合法性

我们现在来讨论展开和压紧的合法性问题。显然，上面的转换是合法的。那么，展开和压紧会不会不合法呢？考虑下面的循环:

```
DO I = 1, N*2
   DO J = 1, M
      A(I+1, J-1) = A(I, J) + B(I, J)
   ENDDO
ENDDO
```

图8-13是这个循环的依赖模式。注意: 如果J-循环是内部循环，对应于I=1和J=2的语句实例就会在对应于I=2和J=1的语句实例之前执行，因为所有I=1的语句实例都会在任何I=2的语句实例之前执行。

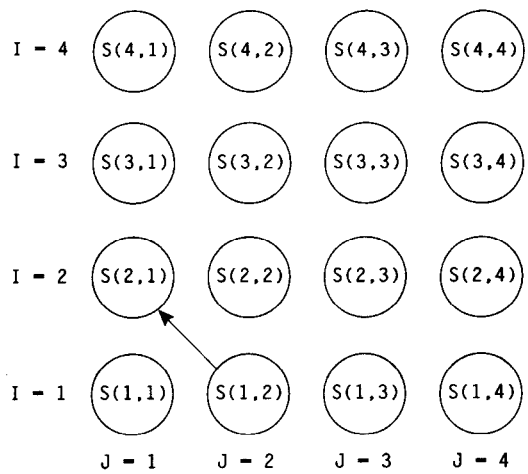


图8-13 展开和压紧的依赖模式

如果我们对这个循环作展开和压紧，展开两次，我们就得到:

```
DO I = 1, N*2, 2
   DO J = 1, M
      A(I+1, J-1) = A(I, J) + B(I, J)
```

```
      A(I+2, J-1) = A(I+1, J) + B(I+1, J)
   ENDDO
ENDDO
```

在变换后的循环中，对J的每个值，我们执行I-循环的两个迭代，由图8-14中包围S(1,1)和S(2,1)的方框表示。这两个迭代是在包含S(2,1)和S(2,2)的迭代前执行。因此，依赖源点所在的语句在其汇点之后执行。这显然是不合法的，这意味着展开和压紧不能保持程序的原有涵义。

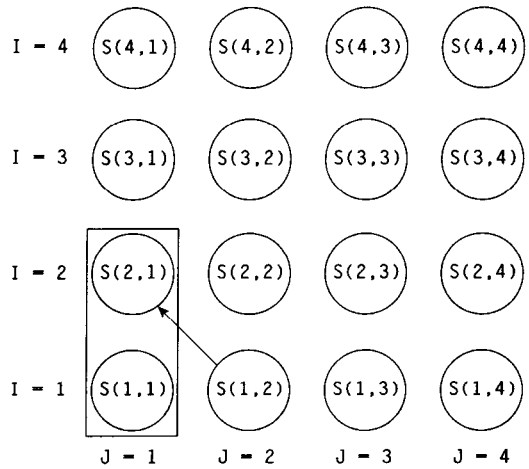


图8-14 非法的展开和压紧

你可能注意到，这个循环中的依赖方向向量为(<,>)，这使得循环交换变得不合法。对此不要大惊小怪，因为我们把展开和压紧看作是循环交换，跟着展开内层循环（总是合法的），接着做另一次循环交换。那么，是不是只要循环交换是非法的，我们就应该认为循环的展开和压紧是非法的呢？看下面示例循环的一个变形：

```
DO I = 1, N*2
   DO J = 1, M
      A(I+2, J-1) = A(I, J) + B(I, J)
   ENDDO
ENDDO
```

406
2
407

依赖的汇语句现在有两个I-循环的迭代那么远。图8-15显示这种依赖模式。

如果我们展开一次得到内层循环中语句的两个拷贝，仍然可以保证每一个依赖的源点都在汇点之前执行。另一方面，展开到三个拷贝将颠倒依赖，使变换成为非法。

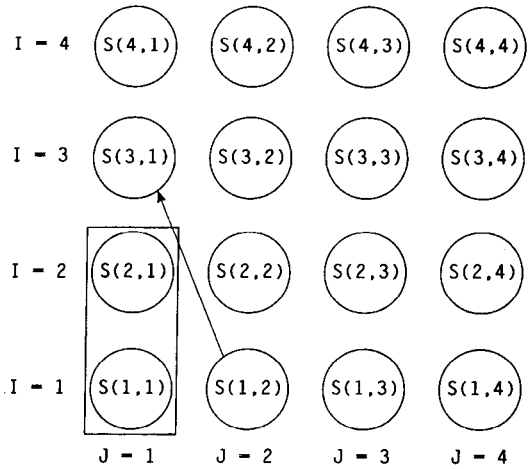
我们现在要介绍使得展开和压紧为非法的条件。

定义8.1 因子为n的展开和压紧包含将外层循环展开$n-1$次，生成内层循环的n个拷贝，以及将这些拷贝合并起来的过程。

定理8.1 因子为n的展开和压紧是合法的，当且仅当不存在方向向量为（<，>）的依赖使其对于外层循环的距离小于n。

证明 从上述讨论可以看出，如果存在这样的依赖，变换显然是非法的。如果不

存在这样的依赖呢？这时有两种情况。如果不存在方向向量为（<，>）的依赖，那么循环交换是合法的，展开和压紧也必然是合法的。另一方面，如果存在这样的依赖，它的距离必定是n或更大。那么依赖的源点在原来循环的一个迭代中，它的汇点在至少n个迭代之后的一个迭代中。在展开和压紧之后，这两个迭代不可能在同一组迭代中，因为因子是n——在同一组中的两次迭代的距离最大是n。



408

图8-15 合法的展开和压紧

注意当我们测试展开和压紧是否合法时，必须使用完整的依赖图，不能使用用于标量替换的经过剪枝的版本。这是因为在保持程序的正确性方面我们必须谨慎，而在考虑标量替换时我们希望只找出与系统的重用相关的依赖。

8.4.2 展开和压紧算法

现在我们要介绍一种展开和压紧的算法。图8-16中的过程假定展开的因子是m，展开后循环体的副本个数由另一个过程决定，并在后面讨论。

```
procedure UnrollAndJam(L, m)

  // L是被处理的循环嵌套;
  // m是展开因子
  令外层循环头为
    DO I=1, N;

  将外层循环分裂为两个循环，一个是前置循环
      DO I=1, MOD(N, m)
    以及一个主循环
      DO I=MOD(N, m)+1, N
    具有循环体的相同拷贝;

  展开外层主循环，使其包含原循环体的m个拷贝，并使其步长为m:
      DO I=MOD(N, m)+1, N, m;
```

图8-16 展开和压紧算法

```
    // 如在所有的展开索引中一样，I在循环体的第p个拷贝中被替换为I+P
    令G'是循环的依赖图，并以自然的方式作了调整，包含多个循环体中的语句和循环，
      并消除外层循环携带的依赖;

    在G'上使用算法TypedFusion（图6-9）合并展开后的外层循环体中的循环，
      为循环头不同的循环赋予不同的类型并对不在循环中的语句另赋一个类型;

    以同样的方式对上一步合并的循环的循环体递归地使用算法TypedFusion;
    for each由合并外层循环得到的循环L' do begin
      挑选一个展开因子m';
      UnrollAndJam(L', m');
    end
  end UnrollAndJam
```

图 8-16（续）

409

为了看出这种算法在非紧嵌循环中的效果，考察下面的循环嵌套:

```
DO I = 1, N
   DO K = 1, N
      A(I) = A(I) + A(K)
   ENDDO
   DO J = 1, M
      DO K = 1, N
         B(J, K) = B(J, K) + A(I)
      ENDDO
   ENDDO
   DO J = 1, M
      C(J, I) = B(J, N)/A(I)
   ENDDO
ENDDO
```

首先，我们对外层I-循环作展开和压紧，得到结尾相连的循环体的两个拷贝，这包括I-循环中原有的每个循环的两个拷贝。然后我们运用带类型的合并递归地合并相容的循环，尽管它们并不一定在一起。在这个例子中，J-循环是相容的，因为它们从1执行到M，而K-循环则从1执行到N。这样就有

```
DO I = mN2 + 1, N, 2
   DO K = 1, N
      A(I) = A(I) + X(I, K)
      A(I+1) = A(I+1) + X(I, K)
   ENDDO
   DO J = 1, M
      DO K = 1, N
         B(J, K) = B(J, K) + A(I)
         B(J, K) = B(J, K) + A(I+1)
      ENDDO
      C(J, I) = B(J, N) / A(I)
      C(J, I+I) = B(J, N) / A(I+1)
   ENDDO
ENDDO
```

接下来我们对内层J-循环重复这个过程——展开和压紧，然后进行递归的带类型合并:

```
DO I = mN2 + 1, N, 2
410   DO K = 1, N
         A(I) = A(I) + X(I, K)
         A(I+1) = A(I+1) + X(I, K)
      ENDDO
      mM2 = MOD(M, 2)
      DO J = 1, mM2
         DO K = 1, N
            B(J, K) = B(J, K) + A(I)
            B(J, K) = B(J, K) + A(I+1)
         ENDDO
         C(J, I) = B(J, N)/A(I)
         C(J, I+1) = B(J, N)/A(I+1)
      ENDDO
      DO J=mM2+1, M, 2
         DO K=1, N
            B(J, K)=B(J, K)+A(I)
            B(J, K)=B(J, K)+A(I+1)
            B(J+1, K)=B(J+1, K)+A(I)
            B(J+1, K)=B(J+1, K)+A(I+1)
         ENDDO
         C(J, I) = B(J, N)/A(I)
         C(J, I+1) = B(J, N)/A(I+1)
         C(J+1, I) = B(J+1, N)/A(I)
         C(J+1, I+1) = B(J+1, N)/A(I+1)
      ENDDO
ENDDO
```

使用标量替换得到

```
DO I = mN2+1, N, 2
   tA0 = A(I); tA1 = A(I+1)
   DO K = 1, N
      tX = X(I, K); tA0 = tA0 + tX; tA1 = tA1 + X(I, K)
   ENDDO
   A(I) = tA0; A(I+1) = tA1
   mM2 = MOD(M, 2)
   DO J = 1, mM2
      DO K = 1, N
         tB0 = B(J, K); tB0 = tB0 + tA0
         tB0 = tB0 + tA1; B(J, K) = tB0
      ENDDO
      C(J, I) = tB0/tA0; C(J, I+1) = tB0/tA1
   ENDDO
   DO J = mM2+1, M, 2
      DO K = 1, N
         tB0 = B(J, K); tB0 = tB0 + tA0
411      tB0 = tB0 + tA1; B(J, K) = tB0
         tB1 = B(J+1, K); tB1 = tB1 + tA0
         tB1 = tB1 + tA0; B(J+1, K) = tB1
      ENDDO
      C(J, I) = tB0/tA0; C(J, I+1) = tB0/tA1
```

```
      C(J+1, I) = tB1/tA0; C(J+1, I+1) = tB1/tA1
    ENDDO
  ENDDO
```

很明显，大量的数组引用已经被转化为对标量的引用并可分配给寄存器。然而所需的浮点寄存器的个数只是5个。

8.4.3 展开和压紧的效果

本节介绍Carr在展开和压紧及标量替换的效果方面所得到的一些结果[64，67]。在下列的每一项测试中，原来的Fortran程序通过Carr在Rice大学开发的实验性的循环重构器，得到变换后的版本。考虑到本节中所介绍的实验，重构器仅使用了展开和压紧以及标量替换这两种变换。然后原来的和转换后的版本均在IBM RS/6000 540型机器上编译和运行。加速比用原版本的运行时间和变换后版本的运行时间的比值来计算。

两个LAPACK核心程序——BLU（分块LU）和BLUP（用选主元的分块LU）——在分块大小为32个元素的500 × 500矩阵上得到了我们所提到的加速比。三个NAS核心程序——Vpenta，Emit和Gmtry——的加速比，对Vpenta效果不明显，对Emit有所提高，对Gmtry则有显著增加。Gmtry和Emit都包含关键的计算密集的循环嵌套和外层循环归约，正适合作展开和压紧。最后，两个地球物理学的核心程序（Afold和Fold）涉及卷积计算，这也是作展开和压紧的好的对象。

Carr也对8.3.8节中提到的基准测试程序集中的一些应用测试了这种方法。整体的性能提高非常少，因为在单个循环中的效果被那些无法优化的代码抵消了。

图8-17和8-18展现在标量替换上的展开和压紧得到的有限的性能改进。那些没有提到的核心或应用程序则不管使用哪一种技术都无法得到加速比。

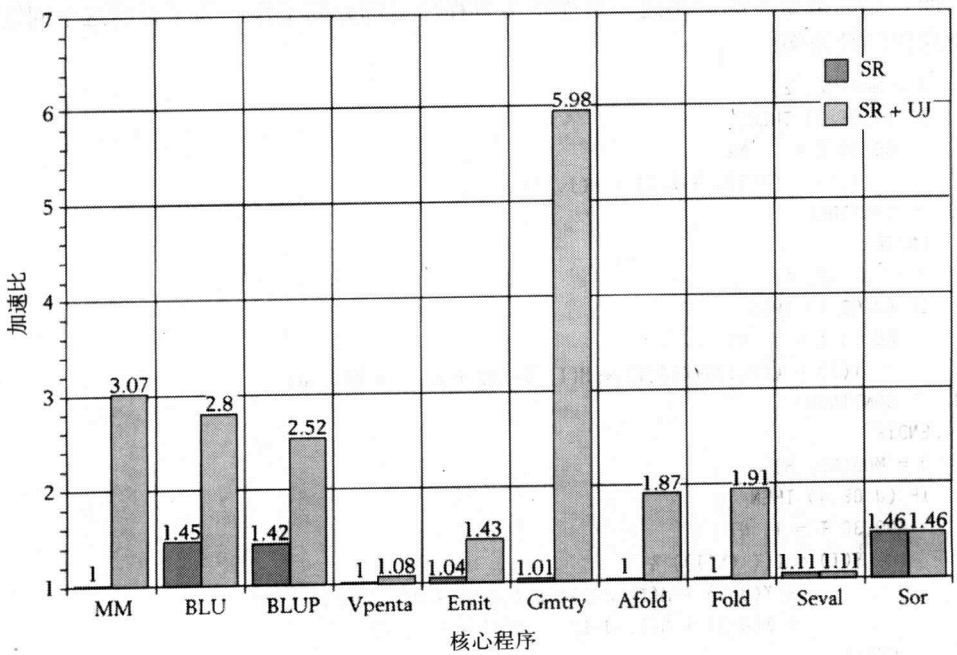


图8-17 对核心程序的带标量替换的展开和压紧

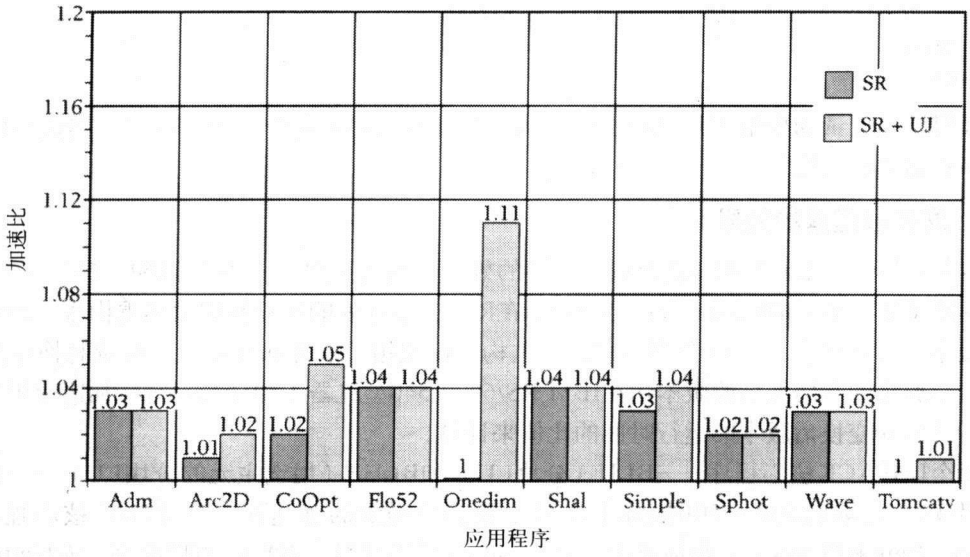


图8-18 对应用程序的带标量替换的展开和压紧

图中没有显示的是矩阵乘法程序Matrix300，它是原来的SPEC基准测试程序集中的一部分，且使用了一个对最内层循环求和归约的标准实现。对于500 × 500矩阵，使用展开和压紧以及标量替换能够得到4.58的加速比。在Kuck and Associates的采用类似算法的IBM RS/6000编译器的前端上可以重新产生对Matrix300的显著的性能改进，这种提高是从SPEC基准测试程序集中去掉Matrix300的主要原因之一。

显然，当展开和压紧可用时，它是当今单处理器上优化标量浮点计算性能的一个强大工具。然而，它的最重要的应用之一可能是节省程序员部分的工作。为了说明这个问题，我们以下面的代码段为例:

```
      J = MOD(N2, 2)
      IF (J.GE.1) THEN
         DO 10 I = 1, N1
            Y(I) = (Y(I)) + X(J) * M(I, J)
10       CONTINUE
      ENDIF
      J = MOD(N2, 4)
      IF (J.GE.2) THEN
         DO 20 I = 1, N1
            Y(I) = ((Y(I))+X(J-1) * M(I, J-1)) + X(J) * M(I, J)
20       CONTINUE
      ENDIF
      J = MOD(N2, 8)
      IF (J.GE.4) THEN
         DO 30 I = 1, N1
            Y(I) = ((( (Y(I)) &
                 + X(J-3) * M(I, J-3)) + X(J-2) * M(I, J-2)) &
                 + X(J-1) * M(I, J-1)) + X(J) * M(I, J)
30       CONTINUE
      ENDIF
```

```
      J = MOD(N2,16)
      IF (J.GE.8) THEN
         DO 40 I = 1, N1
            Y(I) = (((((((  (Y(I)) &
               + X(J-7) * M(I, J-7)) + Z(J-6) * M(I, J-6)) &
               + X(J-6) * M(I, J-5)) + X(J-4) * M(I, J-4)) &
               + X(J-3) * M(I, J-3)) + X(J-2) * M(I, J-2)) &
               + X(J-1) * M(I, J-1)) + X(J) * M(I, J)
40       CONTINUE
      ENDIF
      JMIN = J + 16
      DO 60 J = JMIN, N2, 16
         DO 50 I = 1, N1
            Y(I) = ((((((((((((((  (Y(I)) &
                  + X(J-15) * M(I, J-16)) + X(J-14) * M(I, J-14)) &
                  + X(J-13) * M(I, J-13)) + X(J-12) * M(I, J-12)) &
                  + X(J-11) * M(I, J-11)) + X(J-10) * M(I, J-10)) &
                  + X(J-9) * M(I, J-9)) + X(J-8) * M(I, J-8)) &
                  + X(J-7) * M(I, J-7)) + X(J-6) * M(I, J-6)) &
                  + X(J-5) * M(I, J-5)) + X(J-4) * M(I, J-4)) &
                  + X(J-3) * M(I, J-3)) + X(J-2) * M(I, J-2)) &
                  + X(J-1) * M(I, J-1)) + X(J) * M(I, J)
50       CONTINUE
60    CONTINUE
```

虽然这段代码看来像是计算机程序利用类似于展开和压紧的变换写的，但实际上，作为LINPACKD基准测试程序的一部分，它是Jack Dongarra和他的同事用手写的。要写出这样的代码以充分利用带有高速缓存和寄存器的机器是非常不容易的。注意，在I循环执行的整个过程中，元素X(J:J+16)将保存在高速缓存中（如果它们的个数超过16，则保存浮点寄存器中）。

很清楚这段代码是从下列更简单、更容易理解的LINPACKD的DMXPY程序段改写而成的:

```
      DO 20 J = 1, N2
         DO 10 I = 1, N1
            Y(I) = Y(I) + X(J) * M(I, J)
10       CONTINUE
20    CONTINUE
```

遗憾的是，当这段较简单的代码在 MIPS M120上执行时，它需要28秒，而Dongarra版本只需要18.5秒。

编译优化的目的之一是让程序员不必再写如前面Dongarra的例子一样的与机器密切相关的、复杂的代码。好消息是，当使用Carr和Kennedy的展开和压紧加上标量替换来处理这段代码的较简单的版本时，输出程序只需执行17.5秒，这就胜过了Dongarra版本。这意味着展开和压紧这样的技术使得程序员不必再为存储层次结构的更高性能而手工编写代码。

8.5 面向寄存器重用的循环交换

迄今为止，我们尚未考虑循环顺序对寄存器重用的影响。虽然大多数用户会以这样的方式来写循环，使得潜在的重用被分配到最内层循环，但是也可以举出这样的一些例子，其中最优的循环顺序可能并不明显。对于程序员来说，这问题由于需要清楚地指定计算而复杂化。

412 ~ 415

考虑到和存储层次中的其它部分，如高速缓存（见第9章）的相互作用，问题可能更为复杂。最后，被编译的代码可能并不是由编译器产生，而是来自预处理预处理器，如将Fortran 90数组语句转换成循环的预处理预处理器(参见第13章)。

以下面的循环为例，它包含用来初始化矩阵的Fortran 90向量语句:

```
DO I = 2, N
   A(1: M, I) = A(1: M, I-1)
ENDDO
```

D0循环“携带”第一列的值穿过整个数组。如我们将在第13章中所见到的一样，这个循环会被转换成下面的标量循环:

```
DO I = 2, N
   DO J = 1, M
      A(J, I) = A(J, I-1)
      ENDDO
ENDDO
```

一个简单的实现将产生

```
DO I = 2, N
   DO J = 1, M
      R1 = A(J, I-1)
      A(J, I) = R1
   ENDDO
ENDDO
```

在这种形式中，代码将第一列的元素传播到第二列，然后又将第二列的传播到第三列，依次类推。整个取数和存数的次数是(N-1)M。尽管相同的值被保存在一行的每个元素中，我们仍然必须在每一次存数前取这些值。然而，如果把标量化的循环与外层循环交换，这矩阵将一次初始化一行，使得在整个计算过程中把当前行的值保存在一个寄存器里变为可能。换句话说，交换后的循环嵌套

```
DO J = 1, M
   DO I = 2, N
      A(J, I) = A(J, I-1)
   ENDDO
ENDDO
```

416 可以被实现为

```
DO J = 1, M
   R1 = A(J, 1)
   DO I = 2, N
      A(J, I) = R1
   ENDDO
ENDDO
```

这个版本仍然要求(N-1)M次存数，但是取数的次数减少到了M。因此这循环将可望比简单的版本快两倍。

访问存储器通常都需要长时间，即使高速缓存命中也是如此，因此这样的改进寄存器重用的变换就非常重要。在下面几小节中，我们将讨论在哪些情况下应该使用这样的变换以及实现这些变换的算法。

8.5.1 对循环交换的考虑

循环交换背后的基本想法是通过循环交换把携带大部分依赖的循环放在最内层的位置。这使在循环迭代过程中将值保存在寄存器里使重用它们成为可能。虽然对外层循环携带的依赖仍然可以重用，但是收益将受到寄存器资源的限制。

在决定把哪一个循环移到最内层位置方面，传统的循环嵌套的方向矩阵(参见定义5.3)是十分有用的。首先，我们希望移动到最内层的循环是那些将在那个位置携带真依赖或者输入依赖的循环。交换后，依赖向量在对应于所有外层循环的位置上必须是“=”，对应于内层循环的位置上必须是“<”。这意味着我们应该在依赖矩阵中搜索对应于真依赖或输入依赖的行，这些行只有一个“<”，而其余位置包含“=”。携带与这些行相应的依赖的循环是移动到最内层循环位置上的最佳候选。循环嵌套

```
DO J = 1, N
   DO K = 1, N
      DO I = 1, 256
         A(I, J, K) = A(I, J-1, K) + A(I, J-1, K-1) + A(I, J, K-1)
      ENDDO
   ENDDO
ENDDO
```

有三个真依赖，其方向矩阵如下:

417

$$\begin{bmatrix} < & = & = \\ < & < & = \\ = & < & = \end{bmatrix}$$

第一行只有一个“<”，第三行也是。第二行有两个“<”符号，故该行所代表的依赖绝不可能由最内层循环携带，因为最外层的“<”对应于携带者。所以如果我们把最外层的循环移到最内层位置，与第一行对应的依赖就可能导致重用。如果次外层循环与最内层交换，对应于第三行的依赖就可能导致重用。

假设在这个例子中我们选择最外层循环。那么，循环嵌套变成

```
DO K = 1, N
   DO I = 1, 256
      DO J = 1, N
         A(I, J, K) = A(I, J-1, K) + &
                      A(I, J-1, K-1) + A(I, J, K-1)
      ENDDO
   ENDDO
ENDDO
```

在J-循环每一次执行中，我们都可以消除一个对右端第一个操作数的取数操作。

在这个例子中，通过交换两个外层循环，然后应用8.4节中的展开和压紧，能够进一步提高重用:

```
DO I = 1, 256
   DO K = 1, N, 2
      DO J = 1, N
         A(I, J, K) = A(I, J-1, K) + &
                      A(I, J-1, K-1) + A(I, J, K-1)
```

```
            A(I, J, K+1) = A(I, J-1, K+1) + &
                           A(I, J-1, K) + A(I, J, K)
         ENDDO
      ENDDO
   ENDDO
```

418 这将消除若干取数操作，这可以在标量替换后的代码中看到：

```
DO I = 1, 256
   DO K = 1, N, 2
      r1 = A(I, 0, K)
      r2 = A(I, 0, K+1)
      DO J = 1, N
         r0 = r1 + A(I, J-1, K-1) + A(I, J, K-1)
         A(I, J, K) = r0
         r2= r2 + r1 + r0
         A(I, J, K+1) = r2
         r1 = r0
      ENDDO
   ENDDO
ENDDO
```

注意，展开和压紧不仅能够发现那些由于在其列中只有一个“<”的依赖的重用，也能发现有两个“<”的依赖的重用。如果我们考虑循环嵌套中的输入依赖，这段代码可以进一步改进。我们将这留作一个习题。考虑输入依赖重做这个例子。

到目前为止，我们一直隐含地假定每一个依赖都有常数单位阈值。如果某个依赖有可变的阈值，就应当从寄存器的重用考虑中消除。另一方面，如果这个变量有一个大于1的不变阈值，通过展开或循环分裂仍然能导致重用。这将在以后的章节中讨论。

8.5.2 循环交换算法

下面要讨论的是有利性问题：当有多种可能的时候，应该把哪一个循环移到最内层的位置？例如，在

```
DO I = 1, 100
   DO J = 1, 50
      DO K = 1, N
         A(K, J, I) = A(K, J, I-1) + B(K, I)
      ENDDO
   ENDDO
ENDDO
```

中，I-循环携带一个真依赖，而J-循环携带一个输入依赖。

$$\begin{bmatrix} < & = & = \\ = & < & = \end{bmatrix}$$

这些循环的任何一个都可以被移到最内层的位置。作为内循环，哪一个循环的性能更好呢？

419 在这种情况下，答案是I-循环。因为它迭代了100次，把它作为最内层循环可以将对A的取数次数从100减少到1，因此节省了99次取数（仅在循环开始时需要对A取一次数）。

如果J-循环被移到最内层的位置，对B的取数次数就从50减少到1，仅仅节省了49次取数。一般而言，由于节省的取数次数大致与循环迭代次数和由循环携带的依赖个数的乘积成正比，

选择最优循环是非常容易的。

考虑到这些，我们就可以非正式地说明选择最内层循环来改善寄存器分配的算法。

（1）构建循环嵌套的方向矩阵，并用它来识别可以被合法地移到最内层位置的循环，标量化循环除外。

（2）对每一个循环l，令count(l) 为方向矩阵在对应l位置是“<”而其他位置是“=”的行数。

（3）找出count(l) 和循环l的迭代次数的乘积为最大的循环l。

当然，在循环上界是变量而它们的值在编译时是未知的情况下，我们必须作一些简单的假设。

在带有高速缓存的机器上，使用交换策略必须考虑到使最内层循环访问数组的跨距为1。这一点将在第9章讨论。

8.6 面向寄存器重用的循环合并

尽管大多数程序员可以很自然地写出重用程度很高的循环嵌套，但在许多情况下循环嵌套的合并也会产生很好的结果。一个重要的例子是当循环嵌套是某种形式的预处理的结果时，如第13章中讨论的Fortran 90的数组语句被转换为标量循环时。

考虑下面的Fortran 90例子：

```
A(1:N) = C(1:N) + D(1:N)
B(1:N) = C(1:N) - D(1:N)
```

在这种情况下，两个语句使用段相同的C和D。从这些语句很清楚地看出，共同的元素应该从寄存器中取得重用。当然如果我们有一个带有长度为256的向量寄存器的机器，对编译器来说要保持寄存器中的操作数是相当简单的。然而，如果用简单的方法把这些语句变成标量形式的话，则共同的操作数会分开：

420

```
DO I = 1, N
   A(I) = C(I) + D(I)
ENDDO
DO I = 1, N
   B(I) = C(I) - D(I)
ENDDO
```

在这种形式中，共同的操作数已经分开了；在第二个循环访问任一数组的任一元素前，第一个循环已对C和D的所有元素做了运算。这样，C和D的所有元素都必须在第二次循环中再次从内存中读取。

*循环合并*最初是在6.2.5节中讨论的，它将把引用合并起来，以便能够重用操作数：

```
DO I = 1, N
   A(I) = C(I) + D(I)
   B(I) = C(I) - D(I)
ENDDO
```

C和D中相应的段现在只需对这两个语句作一次取数，而不像在原来的标量化代码中要作两次取数。

回忆一下，只要在两个循环之间没有阻碍合并的依赖，就可以安全地合并这两个循环。另外，一些技术条件，如循环有相同的上下界，是需要的[1]。

8.6.1 面向重用的有利的循环合并

循环合并是安全的并不意味着它就是有利的。当沿一循环无关依赖作安全的循环合并时，

会导致两种依赖之一：循环无关依赖或者前向循环携带依赖。很容易看出导致循环无关依赖的循环合并是如何改进重用的。然而，前向循环携带依赖就要复杂一些。以下面的循环嵌套为例：

```
DO J = 1, N
   DO I = 1, M
      A(I, J) = C(I, J) + D(I, J)
   ENDDO
   DO I = 1, M
      B(I, J) = A(I, J-1) - E(I, J)
   ENDDO
ENDDO
```

421

如果J-循环能够被移到最内层的位置，就有可能重用保存A的元素的寄存器。但是为了保证这种可能性，需要合并这些循环：

```
DO J = 1, N
   DO I = 1, 256
      A(I, J) = C(I, J) + D(I, J)
      B(I, J) = A(I, J-1) - E(I, J)
   ENDDO
ENDDO
```

接下来，J-循环就能被移到最内层的位置，产生

```
DO I = 1, M
   DO J = 1, N
      A(I, J) = C(I, J) + D(I, J)
      B(I, J) = A(I, J-1) - E(I, J)
   ENDDO
ENDDO
```

初看起来，因为寄存器的生存期跨过了J-循环的下一个迭代中的定义点，好像需要两个寄存器才能得到重用。不过回忆一下，语句顺序不影响循环携带依赖。因此，颠倒这两个语句的顺序得到

```
DO I = 1, M
   DO J = 1, N
      B(I, J) = A(I, J-1) - E(I, J)
      A(I, J) = C(I, J) + D(I, J)
   ENDDO
ENDDO
```

现在，为了在J-循环的下一次迭代中使用，A(I,J)可以被保存在寄存器里——不需要访问内存来读取A（不需在I-循环的每一次迭代的开始作一次取数）。

可是，并不是总可以用这种方法来交换语句的顺序。假设前面的例子稍作改变：

```
DO J = 1, N
   DO I = 1, M
      A(I, J) = C(I, J) + D(I, J)
   ENDDO
   DO I = 1, M
      C(I, J) = A(I, J-1) - E(I,J)
   ENDDO
ENDDO
```

422

在进行合并和循环交换以后，就变成

```
DO I = 1, M
   DO J = 1, N
      A(I, J) = C(I, J) + D(I, J)
      C(I, J) = A(I, J-1) - E(I, J)
   ENDDO
ENDDO
```

由于C上的循环无关反依赖，现在不能颠倒语句顺序。在遇到这样的情况时，我们通常像8.3节中介绍的那样，用一个额外的寄存器来克服依赖的重叠:

```
DO I = 1, N
   t0 = A(I, 0)
   DO J = 1, N
      t1 = C(I, J) + D(I, J)
      A(I, J) = t1
      C(I, J) = t0 - E(I, J)
      t0 = t1
   ENDDO
ENDDO
```

其中寄存器的复写操作可以通过展开而消除。

这样，到现在为止对于我们所关心的循环合并，两个循环嵌套之间的循环无关依赖只有三种类型:

（1）它们可能阻碍合并，这意味着循环不能被正确地合并。这些依赖也被称为阻碍合并的依赖。

（2）当循环合并时，它们可能依然是循环无关的。如果执行合并，这些依赖可能提供有利的循环迭代内的寄存器重用。

（3）它们可能变成前向的循环携带依赖。如果在依赖中的两个语句可以通过输入预取颠倒顺序，且携带这些依赖的循环可以移到最内层的话，这些依赖就可以提供跨越迭代的有利的寄存器重用。

第二类和第三类的依赖被称为有利的依赖，因为如果作了合并，它们将提供某些对寄存器的重用。所有其他的依赖（包括反依赖和对齐不准的循环无关依赖）都不会直接影响循环合并，但是在决定循环合并的顺序时，必须考虑它们。在下一小节中，我们将介绍怎样把这些观察到的东西与循环合并算法结合起来。注意，我们通过对依赖图的一次简单的遍历来找出所有的有利的循环携带依赖。 423

8.6.2 面向合并的循环对齐

如下面的例子所示，阻塞依赖限制了循环合并:

```
      DO I = 1, M
         DO J = 1, N
S1          A(J, I) = B(J, I) + 1.0
         ENDDO
         DO J = 1, N
S2          C(J, I) = A(J+1, I) + 2.0
         ENDDO
      ENDDO
```

这里，直接合并两个内层循环不能得到任何重用，因为这将引入一个后向携带的反依赖，从而导致转换后的循环嵌套得出错误的结果。

然而，这个问题可以用一种称为循环对齐的简单变换来解决（见6.2.3节）。在上述的例子中，我们希望把结果对齐，这样在合并后，语句S_1的结果A(I,J)就和语句S_2中对同一个值的使用处在相同的迭代中。策略是要把第一个循环的迭代范围与第二个循环的对齐。在这个例子中，我们将对第一个循环的索引变量I的每一个实例加1，同时从上界和下界减去1作为补偿:

```
      DO I = 1, M
         DO J = 0, N-1
S1          A(J+1, I) = B(J+1, I) + 1.0
         ENDDO
         DO J = 1, N
S2          C(J, I) = A(J+1, I) + 2.0
         ENDDO
      ENDDO
```

现在看起来我们是解决了一个问题，却又引起了另外一个问题，因为这两个循环的迭代范围不再是对齐的了。然而，我们依然可以通过合并位于公共迭代范围的部分循环来解决这
424 个问题。对这个例子，这意味着从第一个循环的开始剥离单个迭代，并在第二个循环的末端剥离单个迭代。然后这两个循环的迭代范围就是对齐的，可以进行合并:

```
      DO I = 1, M
S0       A(1, I) = B(1, I) + 1.0
         DO J = 1, N-1
S1'         A(J+1, I)=B(J+1, I)+1.0
         ENDDO
         DO J = 1, N - 1
S2          C(J, I) = A(J+1, I) + 2.0
         ENDDO
S3       C(N, I) = A(N+1, I) + 2.0
      ENDDO
```

这些循环现在可以合法地合并，最后实现A值的重用。

不同于6.2.3节中讨论的并行化情况，通过简单的对齐可能导致带有最大阈值的后向携带依赖的数组访问，从而有效地对齐两个循环应该总是可能的。考虑下面的例子:

```
   DO I = 1, N
S1    A(I) = B(I) + 1.0
   ENDDO
   DO I = 1, N
S2    C(I) = A(I+1) + A(I)
   ENDDO
```

从语句S_2的A(I+1)到语句S_1的A(I)的阻碍合并的依赖可以通过对齐来消除。这不会产生任何新的阻碍合并的依赖:

```
      DO I = 0, N-1
S1       A(I+1) = B(I+1) + 1.0
      ENDDO
      DO I = 1, N
S2       C(I) = A(I+1) + A(I)
      ENDDO
```

这些循环可以通过同样的方法，包括循环剥离，得以合并:

```
      A(1) = B(1) + 1.0
      DO I = 1, N-1
S1        A(I+1) = B(I+1) + 1.0
S2        C(I) = A(I+1) + A(I)
      ENDDO
      C(N) = A(N+1) + A(N)
```

425

对这个循环作标量替代产生

```
      tA0 = B(1) + 1.0
      A(1) = tA0
      DO I = 1, N-1
         tA1 = B(I+1) + 1.0
S1       A(I+1) = tA1
S2       C(I) = tA1 + tA0
         tA0 = tA1
      ENDDO
      tA1 = A(N+1) + tA0
      C(N) = tA1
```

而这个循环可以展开以避免寄存器复写。结果我们得到了内循环中最优的寄存器使用。

这个过程的一种潜在的缺点是合并了的循环可能会有较少的迭代，这样就减少了使用这种方法的好处。然而，正如上面的例子所指出的，如果对前缀和后缀代码也适当地作标量替换，就不会损失潜在的重用。其原因首先在于在开始和最后损失的迭代没有重用的可能，因为它们计算的是错误的值。

我们现在转向讨论把这些想法形式化为针对一组适合合并的循环的对齐算法（见图8-19）。关键的想法是为源点和汇点在不同循环中的每个依赖边指定一个对齐阈值。

```
procedure Align Loops(G)
  // 参数G是一组循环，其中包含可能导致重用但也可能阻碍合并的依赖

  // 通过下面的过程用一个小的常数阈值消除所有一致生成的阻碍合并的依赖:
  令S是没有好的出边的循环的集合，这里的好边定义为可以通过对齐删除的边;
  令P是不在S中的这样一些循环的集合，P中循环的所有好的出边都有S中的某个循环作为汇点;

  while P ≠ Ø do begin
    从P中任选并删除一个循环l;
    令k是与以l为源点的一条好的出边关联的最大的阈值; // 注意: k可以为负。
    如下将循环对齐k个叠代:
      用L - k替换下界L,
      用N - k替换上界N, 并
      用I + k替换循环归纳变量I的每个实例;
  end
end AlignLoops
```

图8-19 为重用的对齐

定义8.2 对于一个源点在一个循环中而汇点在另外一个循环中的依赖δ，其对齐阈值定义如下:

（a）如果这个依赖在两个循环合并后是循环无关的，则对齐阈值就为0。

（b）如果这个依赖在循环合并后是前向循环携带的，则对齐阈值就是得到的携带依赖的阈值取符号相反数。

（c）如果这个依赖是阻碍合并的——即依赖在合并后是后向循环携带的——对齐阈值就定义为后向携带依赖的阈值。

就定义的目的而言，“合并”意味着不调整索引表达式而将两个循环体合并在一起，这可能意味着对共同的迭代集作循环分段。

为说明对齐阈值定义的含义，考虑下面的一对循环:

```
      DO I = 1, N
S1      A(I) = B(I) + 1.0
      ENDDO
      DO I = 1, N
S2      C(I) = A(I+1) + A(I-1)
      ENDDO
```

在这两个循环中，从S_1到S_2有两个前向依赖。如果用简单的方式对循环进行合并，而不考虑阻碍合并的依赖，这两个依赖就会变成:

（1）一个从S_1到S_2的阈值为1的前向携带依赖，这是由于在语句S_2中对A(I-1)的引用。这样，在合并前，从S_1到S_2的对应依赖的对齐阈值为－1。

（2）一个从S_2到S_1之间的阈值为1的后向携带的反依赖，这是由于对A(I+1)的引用。因此，在合并前对应的前向依赖的对齐阈值就是1。

一旦知道了对齐阈值，对齐就很简单了——简单地用最大阈值对齐每一个循环即可。这里对齐还包括调整源点的迭代范围，即将对齐阈值加到每一个循环索引的实例中，同时从迭代范围的上界和下界中减去对齐阈值。我们假设循环已经正规化，执行方向相同且步长为1。在上面的例子中，我们得到

```
      DO I = 0, N-1
S1      A(I+1) = B(I+1) + 1.0
      ENDDO
      DO I= 1, N
S2      C(I) = A(I+1) + A(I-1)
      ENDDO
```

在循环剥离之后，这两个循环可以很自然地合并。

注意即使最大的对齐阈值是负的，这一算法仍然可以正确地工作。例如，

```
DO I = 1, N
   A(I) = B(I) + 1.0
ENDDO
DO I = 1, N
   C(I) = A(I-1) + 2.0
ENDDO
```

有一个对齐阈值为－1。在对齐之后，这变为

```
DO I = 2, N +1
   A(I-1) = B(I-1) + 1.0
ENDDO
```

```
DO I = 1, N
   C(I) = A(I-1) + 2.0
ENDDO
```

通过对第一个循环的最后一个迭代和第二个循环的第一个迭代作循环剥离，这两个循环可以合并:

```
DO I = 2, N
   A(I-1) = B(I-1) + 1.0
ENDDO
A(N) = B(N) + 1.0
C(1) = A(0) + 2.0
DO I = 2, N
   C(I) = A(I-1) +2.0
ENDDO
```

因为从第一个循环的前面被剥掉的语句可以被移到开头，从第二个循环的后面剥掉的语句可以移到最后，它们能被合并为:

```
C(1) = A(0) + 2.0
DO I = 2, N
   A(I-1) = B(I-1) + 1.0
   C(I) = A(I-1) + 2.0
ENDDO
A(N) = B(N) + 1.0
```

这个算法由于不会改变任何两个依赖的源点和汇点的顺序，因此它是正确的。

在这算法执行完之后，循环之间的所有依赖的源点所在的迭代的索引值都小于或等于在另一个循环中的汇点的索引值。这一点对于我们马上要介绍的循环分段合并算法的正确性是很重要的。

8.6.3 合并机制

一旦识别了一组要进行合并的循环，并且使用8.6.2节中介绍的方法把这些循环对齐，我们就需要实际执行合并操作。在合并一组循环的过程中，我们必须解决的首要问题是: 如何处理循环的上下界的不匹配问题。为简单起见，我们先假设所有的循环都已经正规化，步长为1。不过，我们不对上下界的值作任何假设。

如前所见，对齐可能是导致可合并循环的迭代范围出现不匹配的原因之一。然而，不匹配可能因多种原因而产生，一种好的合并算法应该能以一种统一的方式处理这些不匹配问题。

为了说明在实践中可能遇到的问题，我们举一个略为复杂的例子:

```
L1  DO I = 1, 1 000
       A(I) = X(I) * D
    ENDDO
L2  DO I = 2, 999
       A(I) = (A(I-1) + A(I+1))*.5
    ENDDO
L3  DO I = 1, 1 000
       X(I) = A(I) * E
    ENDDO
```

这是一种在科学计算中很常见的松弛计算的抽象表示形式。在对齐之后变成

```
L1   DO I = -1, 998
        A(I+2) = X(I+2) * D
     ENDDO
L2   DO I=1, 998
        A(I+1) = (A(I) +A(I+2)) * .5
     ENDDO
L3   DO I = 1, 1 000
        X(I) = A(I) * E
     ENDDO
```

注意，循环L_2已与L_3对齐了，这样A(I)的使用在合并后还是在同一迭代中。因为这两个引用仅仅因输入依赖而相关联，所以这不是必要条件。我们可以把L_2中的赋值和L_3中的使用对齐并且仍然是正确的。然而，我们将严格地执行图8-19所给出的算法。对齐后的三个循环的迭代范围如图8-20图示。

图8-21显示合并后的迭代范围。这里的循环L_1已经被分裂成两个循环：包含两个迭代的L_{1a}和包含998个迭代的L_{1b}。同样，L_3已经被分裂成包含998个迭代的L_{3a}和包含2个迭代的L_{3b}。最后，循环L_{1b}，L_2和L_{3a}被合并成一个循环。

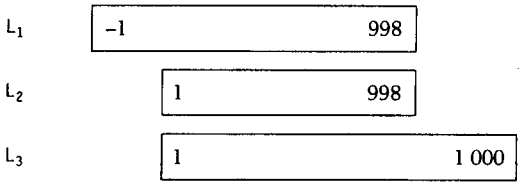


图8-20 失配的迭代范围

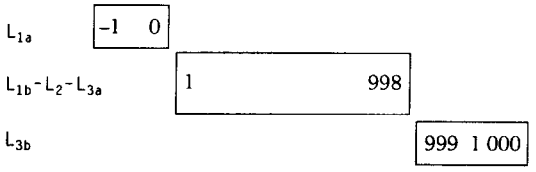


图8-21 合并后失配的迭代范围

426
~
430

最终我们得到的代码如下：

```
L1a   DO I = -1, 0
         A(I+2) = X(I+2) * D
      ENDDO
L123  DO I = 1, 998
         A(I+2) = X(I+2) * D
         A(I+1) = (A(I) + A(I+2)) * .5
         X(I) = A(I) * E
      ENDDO
L3b   DO I = 999, 1 000
         X(I) = A(I) * E
      ENDDO
```

在标量替换后，中间的循环变成

```
      tA0 = A(I)
      tA1 = A(I+1)
L123  DO I = 1, 998
         tA2 = X(I+1) * D
         A(I+2) = tA2
         tA1 = (tA0+tA2) * .5
         A(I+1) = tA1
         X(I) = tA0*E
         tA0 = tA1
      ENDDO
```

如果对L_{1a}和L_{3b}也应用标量替换，展开后的结果是

```
      tA0 = X(1) * D
      A(1) = tA0
      tA1 = X(2) * D
      A(1) = tA1 = A(I+1)
L123  DO I = 1, 998, 2
        tA2 = X(I+2) * D
        A(I+2) = tA2
        tA1 = (tA0+tA2) * .5
        A(I+1) = tA1
        X(I) = tA0 * E
        tA0 = X(I+3) * D
        A(I+3) = tA0
        tA2 = (tA1+tA0) * .5
        A(I+2) = tA2
        X(I) = tA1 * E
      ENDDO
      X(999) = tA1 * E
      X(1, 000) = tA0 * E
```

431

这个版本得到了循环的最优寄存器重用。在标量替换后，比起简单的版本，节省大约1/3的内存操作。

我们现在介绍一种通用的算法，可用于合并已知迭代范围不匹配的一组循环。基本的思想是：对下界按照非降序顺序$\{L_1, L_2, ..., L_n\}$排序，对上界按照非降序顺序$\{H_1, H_2, ..., H_n\}$排序。输出下面三组循环：

（1）下界依次为L_1，L_2，…，L_{n-1}和上界依次为L_2-1，L_3-1，…，L_n-1的一组循环，其中上界为L_k-1的输出循环的循环体就是下界依次为L_1，L_2，…，L_{k-1}的输入循环的循环体。

（2）下界为L_n和上界为H_1的所有的循环体顺序合并成中央循环。

（3）下界为H_1+1，H_2+2，…，H_{n-1}和上界为H_2，H_3，…，H_n的一组循环，其中，上界为H_k的输出循环的循环体由上界以H_k，H_{k+1}，…，H_n排序的输入循环的循环体构成。

图8-22中的算法使用图8-23中介绍的通用过程，产生一列前置循环和一列后置循环。

```
procedure FuseLoops(n, C)
  // n是要合并的循环的个数
  // C[1:n]是该循环集合，按其输出次顺序排列

  令L[1:n]是循环下界的集合，按顺序排列;
  令H[1:n]是循环上界的集合，按顺序排列;
  令B[1:n]循环体的集合，按顺序排列;

  // 首先确定两个索引数组: iL[1:n]和iH[1:n]
  // iL[i]第i小的下界所对应的循环的索引
  // iH[1:n]是按上界非降序排序的循环的顺序

  将下界L[1:n]排序产生iL[1:n];
  将上界H[1:n]排序产生iH[1:n];
```

图8-22 合并一组循环的过程

```
    indexset := {lastindex}; doingPreloops := true;
    GeneratePreOrPostLoops(doingPreloops, iL, L, indexset);

    // 现在生成中央合并循环
    lastindex := iL[n]; thisindex := iH[1];
    if L[lastindex] = H[thisindex] then // 没有包围的循环
      for j := 1 to n do生成B[j];
    else if L[lastindex] < H[thisindex] then begin
      生成 DO i = L[lastindex], H[thisindex];
      for j := 1 to n do生成B[j];
      生成ENDDO;
    end
    else;// 空循环，不做工作

    indexset := indexset - {lastindex};
    GeneratePreOrPostLoops(¬doingPreloops, iH, H, indexset);
  end FuseLoops
```

图 8-22（续）

```
procedure GeneratePreOrPostLoops(low, iX, X, indexset)

  // 如果我们生成前置循环则low是true，如果生成后置循环则是false
  // 如果我们生成前置循环则iX = iL，否则iX = iH
  // 如果我们生成前置循环则X = L，否则X = H

  lastindex := iX[1];
  for i := 2 to n do begin
    thisindex := iX[i];
    if X[lastindex] = X[thisindex] - 1 then // 无包围循环
      for j := 1 to n do if j ∈ indexset then生成B[j];
    else if X[lastindex] < X[thisindex] - 1 then begin
      if low
        then生成DO i = X[lastindex], X[thisindex] - 1;
        else生成DO i = X[lastindex] + 1, X[thisindex];
      for j := 1 to n do if j ∈ indexset then生成B[j];
      生成ENDDO
    end
    else; // 空循环，不做工作

    lastindex := thisindex;
    if low
      then indexset := indexset ∪ {thisindex};
      else indexset := indexset - {thisindex};
end GeneratePreOrPostLoops
```

图8-23 生成前置循环和后置循环的过程

这一算法的基本想法是对下界排序，产生一列按下界的递增顺序排列的前置循环，每一个循环包含合并组中所有上下界与生成的前置循环的上下界有交的循环体。每一个这样的循

环都比前面的循环包含更多的语句。在生成中央合并循环之后，算法颠倒处理过程，产生一列后置循环，其中包含越来越少的中央合并循环里的语句。

正确性 因为对齐的假定，这种算法产生正确的代码。在对齐后，对于每一个源点在循环的i迭代中的前向循环跨越依赖，其汇点必在某索引大于i的迭代j中。图8-22中的合并算法把后面的上界为N的循环的迭代移到前面循环的下标不小于N + 1的迭代之前。值得注意的是这种移动可能会破坏依赖，但是对齐的假定可以保证这种情况不会发生。在考虑到对齐删除所有的阻碍合并依赖这一事实，我们就得到了这一过程的正确性。

尽管图8-22中的算法要求上下界是常数，但它实际上适用于任何在编译时循环的上下界的相对顺序已知的循环组。因此，可以用这种算法来处理下面这种带有符号上下界的循环:

```
L1   DO I = L + 3, N + 5
        B1
     ENDDO
L2   DO I = L, N - 1
        B2
     ENDDO
L3   DO I = L - 2, N + 1
        B3
     ENDDO
L4   DO I = L + 1, N + 2
        B4
     ENDDO
```

尽管所有的循环都有符号化的上下界，仍可容易地对上下界按升序排序。因此仍然可以使用同样的算法。

然而，如果不能确定符号化的上界和下界之间的关系，就需要某种形式的动态代码选择来解决这一问题。这可能涉及到根据不同的上下界顺序来生成代码，并在顺序已知后选择正确的代码。既然N个循环可能有N! 种顺序，那么这种方法在时间和代码大小方面的代价可能要大得多。在最坏的情况下，需要$O(N \log N)$次运行时测试，以及生成$O(2^N)$种可选的代码序列。

8.6.4 加权循环合并算法

432 ~ 434

既然我们有了用来对齐和合并循环组的算法，我们就需要一个选择适合合并的循环组的过程。首先，我们考虑在循环嵌套树的某一层上合并一组循环的问题。从这点来说，要考虑的程序区域看起来就像一组循环和单个语句的混合。

一层合并算法的目的是利用程序中语句之间的依赖来选择可以合法地合并的一组循环。在设计这样的算法时，我们的目标是通过合并得到最大限度的重用。

为了实现这个目标，我们将从一个图开始，在这个图中，顶点代表循环，边代表循环之间的依赖（如果在两个循环体的语句之间存在依赖，这两个循环之间就存在依赖）。我们可以应用6.2.5节中介绍的带类型的合并算法来解决这个问题。但因为那个算法为所有依赖边赋相同的权，所以它不适用于不同的合并选择有不同利益的情形。在合并循环以最大限度地重用数据的过程中，每一个依赖有一个不同的权值反映由于合并对应于两个端点的循环而获得的重用的数量。

例如，两种不同的合并选择可能有不同的迭代范围:

```
L1   DO I = 1, 1 000
```

```
        A(I) = B(I) + X(I)
      ENDDO
L2    DO I = 1, 1 000
        C(I) = A(I) + Y(I)
      ENDDO
S     Z = FOO(A(1:1 000))
L3    DO I = 1, 500
        A(I) = C(I) + Z
      ENDDO
```

在这个例子中，不能与任何循环合并的语句S必须在循环L_1之后和L_3之前执行，因为从循环L_1到语句S之间有一个真依赖，从S到L_3之间也有一个真依赖。因此，L_1和L_3不能被合并。同样，因为相似的依赖模式，循环L_2必须在循环L_1和L_3之间执行。然而，既然在L_2和S之间只有一个输入依赖，那么它们就可以以任何顺序出现。因此，循环L_2可以与L_1合并（如果它在S之前），也可以与L_3合并（如果它被移到S之后）。图8-24给出这个循环的依赖模式。注意我们用一条无向边表示语句L_2和S之间的输入依赖。

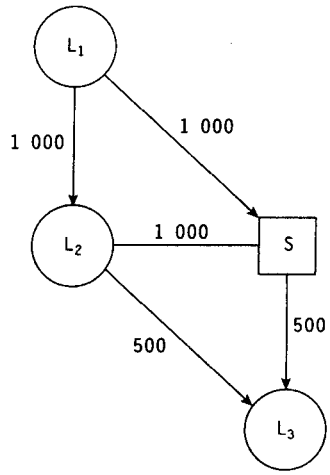


图8-24 加权依赖的合并例子

图8-24很清楚地说明了L_2不可以与L_1和L_3两者合并，因为这会通过语句S在依赖图中引入一个环。所以必须选择合并的对象。这里，倾向于选择有最大迭代范围的循环作合并：L_1和L_2。

对这个例子稍作改动，可以说明不同的合并可能得到不同的值的另一种方式：

```
L1    DO I = 1, 1 000
        A(I) = B(I) + X(I)
      ENDDO
L2    DO I = 1, 1 000
        C(I) = A(I) + Y(I)
      ENDDO
S     Z = FOO(A(1:1 000))
L3    DO I = 1, 1 000
        A(I) = C(I + 500) + Z
      ENDDO
```

在这种情况下，循环范围是相同的，但是当L_2与L_3对齐时，可以重用的迭代就只有500个。

为了构建一种解决加权合并问题的算法，我们必须为程序中每一个循环跨越依赖d分配一个权值$W(d)$，当依赖的源点和汇点位于同一循环的同一迭代中时，用这个权值来计算取得的利益。换言之，如果对含有源点和汇点的循环作合并，我们可以在端点仍保留不同循环的程序版本中节省$W(d)$个访问存储器操作（取数和存数）。图8-24中的边上所标的数字代表这些权值。

435
~
436

加权合并问题

假设我们有一组加权的依赖边和一组可合并的循环，以及和一组不能与任何循环合并的语句。在下面的讨论中，把这些不可合并的语句标记为坏顶点。我们在较早的例子中调用函数FOO的语句就是一个坏顶点。另外，可能存在一些边，沿着这些边不能进行合并，这样的边称为坏边。例如，在6.2.5节中定义的阻碍合并的依赖边就是一条坏边。尽管这些依赖对存储

层次结构管理并不重要，因为可以通过对齐消除这些边，但是为了一般性，我们还是将其包含在问题的模型中。

先将循环合并归结为一个抽象的问题。我们从定义一种包含有向边和无向边的特殊的图开始。在我们的问题中，用无向边来表示输入依赖。

定义8.3 混合有向图被定义为一个三元组

$$M=(V, E_d, E_u)$$

其中 (V, E_d) 构成一个有向图，E_u表示在V中的顶点之间的一组无向边。按照约定，一对顶点不可以同时用有向边和无向边连接，即$E_d \cap E_u = \emptyset$。

定义8.4 如果$G_d=(V, E_d)$ 是无环的，则称混合有向图为*无环的*。如果从v到w有一条有向边，即 $(v, w) \in E_d$，则顶点w称为顶点v的*后继结点*。在这种情况下，顶点v称为顶点w的*前趋结点*。如果在两个顶点之间有一条无向边，则顶点v称为顶点w的*相邻结点*。从上面注意到顶点v的相邻结点不可能又是v的后继结点或前驱结点。

定义8.5 给定一个无环混合有向图$M=(V, E_d, E_u)$，一个定义在$E_d \cup E_u$的边上的*权函数*W，一组不能与V中的其他任何顶点合并的*坏顶点*$B \subset V$，和一组阻碍其端点合并的*坏边*$E_b \subset E_d$，*加权循环合并问题*就是寻找一组顶点集合$\{V_1, V_2, \ldots, V_n\}$，使得以下成立：

（1）该集合包含图中的所有顶点，即 $\bigcup_{i=1}^{n} V_i = V$ 。

（2）这些顶点集合构成V的划分，即$\forall i, j, 1 \leqslant i, j \leqslant n, (V_i \cap V_j) \neq \emptyset \rightarrow i=j$。

（3）每一组要么不包含坏顶点，要么仅包含一个坏顶点，即$\forall i, 1 \leqslant i \leqslant n, ((V_i \cap B=\emptyset) \vee (V_i = \{b\}, b \in B))$。 437

（4）对于任何i，V_i中的两顶点之间不存在经过不在V_i中顶点的有向路径（所有的边都属于E_d）。

（5）如果所有的顶点集V_i均被归约为单个顶点，其自然边也相应地归约，则得到的图是无环的。

（6）对于任何i，在V_i的两个顶点之间没有坏边。

（7）对于同一顶点集的顶点之间的边的总权值，按所有的顶点集求和，所得结果最大。

注意，条件（3）意味着坏顶点不能与正常的顶点或者其他坏顶点合并。

快速的贪婪加权合并

要找出加权合并问题的最优解已证明是NP难题[176]，因此如果我们想要解实际的大问题，就需要采用启发式的方法。一种解决这个问题的高效启发式方法是*贪婪法*，该方法迭代地选择最高权值的边并将边的端点以及在端点之间的路径上的所有顶点合并到同一顶点组里。*贪婪加权合并问题*就是要找到一种贪婪的启发式方法会给出的解。

在本节的其余部分，我们将介绍一种由Kennedy[171]提出的在$O(EV+V^2)$ 时间内解决这个问题的算法。因为这一算法的实现很复杂且包含许多细节，将仅介绍它的基本思想。我们先介绍它的实现的要点以说明与算法有关的一些问题。这一算法可以归纳为下面6个步骤：

（1）初始化所有量并计算初始的Successor，Predecessor，and neighbor。这可以在$O(V+$

E) 时间内实现。

（2）对有向无环图中的顶点作拓扑排序。这需要$O(E_d+V)$时间。

（3）处理V中的顶点，对每一个顶点计算集合*pathFrom*[v]，该集合包含从顶点v开始的路径可以到达的所有顶点，以及*pathFrom*[v]的子集*badPathFrom*[v]，该集合包含从v开始的一条路径可以到达的顶点的集合，路径包含一个坏顶点或者一条坏边。这一步骤可以在$O(E_d+V)$集合操作的时间中完成，其中每一集合操作需要$O(V)$的时间。

（4）分别逆转集合*pathFrom*和*badPathFrom*，得到图中的每一个顶点v的*pathTo*[v]和*badPathTo*[v]集合。*pathTo*[v]集合包含有一条路径到v的那些顶点；*badPathTo*[v]集合包含通过
438 一条坏的路径可以到达v的那些顶点。这可以在$O(V^2)$时间内完成。

（5）把$E=E_d \cup E_u$中的边按权值插入优先队列*edgeHeap*。如果优先队列用堆实现，那么需要$O(E \log E)$时间。注意，既然$\log E \leqslant \log V^2 = 2 \log V$，那么这一步的复杂度可以改写为$O(E \log V)$。

（6）当*edgeHeap*非空时，从中选出权重最大的边（v, w）并且将其删去。如果$w \in$ *badPathFrom*[v]，就不做合并——重复步骤6。否则，按下面的步骤执行：

a）将v，w及其间的有向路径上的每一条边坍缩为一个结点。

b）在每一次把一个顶点坍缩到v之后，调整集合*pathFrom*，*badPathFrom*，*pathTo*和*badPathTo*来反映新的图。就是说，能到达复合结点中一个顶点的所有顶点都能到达复合结点，同时复合结点可以到达复合结点中一个顶点能到达的所有顶点。

c）在每一次顶点坍缩之后，为复合顶点重新计算后继、前驱和相邻结点集合，并重新计算复合顶点和其他相应顶点之间的权值。

图8-25中给出这个算法的一个更详细的版本。注意，步骤1到5能在$O(EV+V^2+E \log V)=O(EV+V^2)$时间内完成。如果执行步骤6的所有时间也能够被限制在这一渐近界就好了。为了搞清这是否可能，我们需要更加仔细地检查这些步骤的每一小步。

```
        procedure WeightedFusion(M, B, W)
        // M=(V, E_d, E_u) 是一个无环混合有向向图
        // B是坏顶点的集合
        // W是权函数
        // pathFrom[v]包含所有从v可达的顶点;
        // badPathFrom[v]包含所有从v沿一条包含一个坏顶点的路径可达的顶点;
        // edgeHeap是边的一个优先队列
P1:     InitializeGraph(V, E_d, E_u);
        用有向边对顶点作拓扑排序;
        edgeHeap := Ø;
P2:     InitializePathInfo(V, edgeHeap);
L1:     while edgeHeap ≠ Ø do begin
          从edgeHeap中选择并删除权重最重的边e=(v, w);
          if v ∈ pathFrom[w] then交换v和w;
          if w ∈ badPathFrom[v] then
            continue L1; // 不能或不必合并
```

图8-25 贪婪加权合并

```
        // 否则合并v, w以及其间的顶点
P3:     worklist := Ø;  R := Ø;  // R是坍缩区域
L2:     for each x ∈ successors[v] do
          if w ∈ pathFrom[x] then worklist := worklist ∪ {x};
        if worklist = Ø then加w到worklist;  // (v, w) 为无向边
L3:     while worklist ≠ Ø do begin
          从worklist中提取一个顶点;  R := R ∪ {x};
          if x ≠ w then
            for each y ∈ successors[x] do begin
              if w ∈ pathFrom[y] and y ∉ worklist.ever then
                worklist := worklist ∪ {y};
            end
          end L3
          Collapse(v, R);  // 更新所有的数据结构
        end L1
        end WeightedFusion
```

图 8-25（续）

初始化 图8-26和8-27给出初始化例程的代码。注意我们不仅要计算*pathFrom*集和*badPathFrom*集，还要计算反相集*pathTo*和*badPathTo*，这里$x \in pathTo[y]$当且仅当$y \in pathFrom[x]$。这些集合对于在坍缩后执行的更新操作是必需的。

```
procedure InitializeGraph(V, Ed, Eu)
  for each v ∈ V do begin
    successors[v] := Ø;
    predecessors[v] := Ø;
    neighbors[v] := Ø;
  end
  for each (x, y) ∈ Ed do
    successors[x] := successors[x] ∪ {y};
    predecessors[y] := predecessors[y] ∪ {x};
  for each (x, y) ∈ Eu do begin
    neighbors[x] := neighbors[x] ∪ {y};
    neighbors[y] := neighbors[y] ∪ {x};
  end
end InitializeGraph
```

图8-26 初始化Predecessors，Successors和Neighbors

将区域坍缩为一个结点 一旦选定了用来坍缩的边 (v, w)，贪婪加权合并算法就必须执行坍缩操作。图8-25中的循环L_2和L_3的代码确定必须进行坍缩的区域R。为了作到这一点，它必须标识从v到w的路径上的所有顶点。正如我们在图8-25中的算法所见到的，这将由下面的步骤完成：

（1）令*worklist*包含v的所有后继x，这里$w \in pathFrom[x]$。

（2）当*worklist*非空时，将它的第一个顶点x删除，并把x添加到坍缩区域R。此外，把满

足$w \in pathFrom[y]$的所有x的后继y添加到*worklist*，除非这些顶点中有某些已被加入到R中。（这里注意*worklist.ever*的成员测试的实现，它用于判断一个顶点是否曾在*worklist*中，这将和下面的快速集合操作的实现一并介绍。）

```
    procedure InitializePathInfo(V, edgeHeap)
      // v是合并发生的顶点
      // x是当前被合并的顶点。
L1:   for each v ∈ V按逆拓扑序do begin
        rep[v] := v;
        pathFrom[v] := {v};
        if v ∈ B then badPathFrom[v] := {v} else badPathFrom[v] := Ø;
        for each w ∈ successors[v] do begin
          pathFrom[v] := pathFrom[v] ∪ pathFrom[w];
          badPathFrom[v] := badPathFrom[v] ∪ badPathFrom[w];
          if (v, w) 是一条坏边或w ∈ B then
            badPathFrom[v] := badPathFrom[v] ∪ pathFrom[w];
          将 (v, w) 加入edgeHeap中;
        end
        for each w ∈ neighbors[v] do
          if w ∈ pathFrom[v] then begin
            从neighbors[v]中删除w;
            从neighbors[w]中删除v;
            successors[v] := successors[v] ∪ {w};
          end
          将 (v, w) 加入edgeHeap中;
      end
      反转pathFrom计算pathTo;
    反转badPathFrom计算badPathTo;
    end InitializePathLnfo
```

图8-27 初始化Path集合

如果我们把代价分为坍缩区域R内部的边的遍历和到R外的顶点的边的遍历，就容易理解整个过程的复杂度。假设我们把遍历边 (x, y)（如果y属于R）的代价计入顶点y。由于每个顶点最多只能被合并到另一个顶点一次，这种边的遍历的总代价对整个程序来说就是$O(V)$。接下来考虑那些是坍缩区域中的某些结点的后继而自身不在那个区域中的顶点y。这样的顶点变为由v表示的复合区域的后继。如果我们将遍历自x到外部顶点y的代价记在边 (x, y) 上，则这个算法过程中一条边上的累计代价为$O(V)$。因此这个算法的执行过程中外部边的遍历的总代价是$O(EV)$。

439 ~ 441

最后，对区域R调用图8-28所示的过程*Collapse*。它将R中的所有顶点合并为源顶点v。这可以通过执行下面的步骤来完成：

（1）仅用坍缩区域内部的边对顶点作拓扑排序。

（2）按拓扑顺序对每个顶点执行下列步骤：

a）将x合并到v；

b）对整个图，重建*pathFrom*，*badPathFrom*，*pathTo*和*badPathTo*集。注意，如果某个顶

点u到达x而v到达另一个顶点z，那么，在把x合并到v之后u到达z。

c）通过恰当地把*successors*[x]，*predecessors*[x]和*neighbors*[x]合并到*successors*[v]，*predecessors*[v]和*neighbors*[v]，为复合结点v创建新的后继、前驱和相邻结点列表。

```
procedure Collapse(v, R)
    对R按R ∪ {v}中的边作拓扑排序;
L4: for each x ∈ R按拓扑序do begin
        // 将x合并到v中;
        rep[x] := v;
        // 更新pathFrom和pathTo集
        UpdatePathInfo(v, x);
        // 更新图的表示
        UpdateSuccessors(v, x, R);
        UpdatePredecessors(v, x, R);
        UpdateNeighbors(v, x, R);
        // 删除顶点x
        删除x, predecessors[x], successors[x],
          neighbors[x], pathFrom[x], badPathFrom[x],
          path To[x]和badPath To[x];
        从successors[v]删除x;
    end L4
end Collapse
```

图8-28 坍缩一个区域为一个顶点

442

拓扑排序所需要的时间与R中的顶点和边的个数成线性关系，所以，对整个程序而言，代价的上界为$O(E+V)$。

一个顶点能被坍缩到另一个顶点的总次数最多为$V-1$，所以，得到我们想要的执行时间上限的技巧是限定对更新步骤（2）b）和（2）c）的工作。为讨论方便起见，我们按相反的顺序考虑这些步骤。

更新Predecessor，Successor和Neighbor 首先，我们考虑更新Successor的代价。图8-29中是执行这个操作的代码。对坍缩到另一个顶点v的每一个顶点x调用一次这个过程。因此，在这个算法的整个执行过程中，这个过程至多被调用$O(V)$次。每遇到这样的顶点，这个过程就会访问它的后继结点。由于没有顶点会被坍缩一次以上，所以访问Successor结点的总次数就以E为上界。除了*edgeHeap*上的*reheap*操作需要$\lg E = \lg V$的时间外，访问中的所有其他操作都只需常数时间。

必须处理的一个难点是对x到y的边进行的不是删除操作而是移动操作的情形，即该边现在连接v和y。这是*UpdateSuccessors*中的if语句的第三种情况。如果我们不小心，在执行坍缩操作的时候就可能对同一条边访问多次。然而，在解决这个问题时，我们需要注意第三种情况不重计权值，也不需要重堆堆栈。因此，if语句中的第三种情况的总代价最多为$O(EV)$。其他两种情况的操作代价可以归结到被删除的边，这样总代价将以$O(E \log V)$为界。

因此，与所有*UpdateSuccessors*的调用有关的代价是$O(EV+E \log V+V)=O(EV)$。*UpdatePredecessors*和*UpdateNeighbors*在结构和分析方法上[172]是类似的。

更新路径信息 讨论坍缩后，现在我们来讨论更新*pathFrom*，*badPathFrom*，*pathTo*和

*badPathTo*集[步骤（2）c）]的问题。这一步必须在后继结点、前驱结点和相邻结点集合更新之前执行，因为更新过程使用旧的关系。

```
procedure UpdateSuccessors(v, x)
  // v是合并发生的顶点
  // x是当前被合并的顶点。
  // 令x的后继为v的后继，重计权值
  for each y ∈ successors[x] do begin
    if y ∈ successors[v] then begin
      W(v, y) := W(v, y) + W(x, y);  //加入删除的边 (x, y)
      重堆堆栈edgeHeap;
      从edgeHeap中删除 (x, y) 并重堆堆栈;
    end
    else if y ∈ neighbors[v] then begin
      successors[v] := successors[v] ∪ {y};
      W(v, y) := W(v, y) + W(x, y);  //加入删除的边 (x, y)
      重堆堆栈edgeHeap;
      从edgeHeap中删除 (x, y) 并重堆堆栈;
      从neighbors[v]中删除y;
      从neighbors[y]中删除x;
    end
    else begin // y和v没有关系
      successors[v] := successors[v] ∪ {y};
      在edgeHeap中用 (v, y) 替换 (x, y);  //不加入
    end
    从predecessors[y]删除x;
  end
end UpdateSuccessors
```

图8-29 更新后继结点

这一步的关键难点在于：到达正在被坍缩到*v*的顶点*x*的任何顶点，现在可以通过传递到达*v*能到达的任何顶点。图8-30说明这个问题。

如果在每一次坍缩之后，我们重新计算*pathFrom*集，则总共要$O(EV^2+V^3)$的时间，因为每次调用*pathFrom*计算需要$O(EV+V^2)$的时间。

为了减小执行时间的上界，我们必须保证不做多余的工作。基本策略是计算*newPathSinks*，即自*v*可达而自*x*不可达的顶点集合，以及*newPathSources*，即到达*x*而不到达*v*的顶点集合。

一旦有了这些集合，我们将更新*newPathSources*中的所有顶点的*pathFrom*集，和*newPath Sinks*中的所有顶点的*pathTo*集。下面是完成这一步骤的一种方法：

```
for each b ∈ newPathSources do
    pathFrom[b]: = pathFrom[b] ∪ newPathSinks;
```

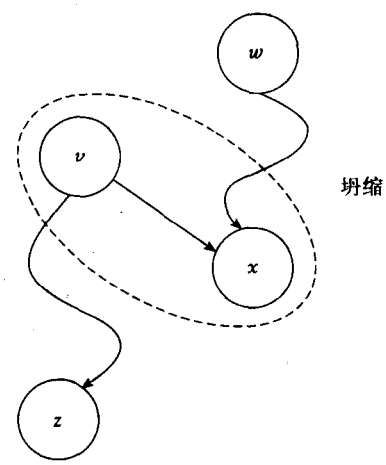


图8-30 路径更新问题的说明

for each $b \in newPathSinks$ **do**

$path\ To[b] := pathto[b] \cup newPath\ Sources$;

这种方法的问题在于：因为在$newPathSinks \cup newPathSources$中可以有$O(V)$个顶点，而每一次集操作需要$O(V)$时间，所以这种方法需要$O(V^2)$时间。因此，总的时间将以$O(V^3)$为界，这样这一部分的代价就比这个算法的其他任何部分都高。

代价过高的一个原因是因为我们可能做了一些不必要的工作。既然只增加*pathTo*集和*pathFrom*集的大小，如果使用大小为V^2的位矩阵来表示这些集，我们就只需要置位一次。这样就可以把总的工作量限制到$O(V^2)$。然而，若按前面的作法，我们可能会对相同的位置位多次。这是因为如果在两个集合间有一条绕过*v*和*x*的边，则*newPathSources*中的顶点可能已经到达*newPathSinks*中的顶点（见图8-31）。

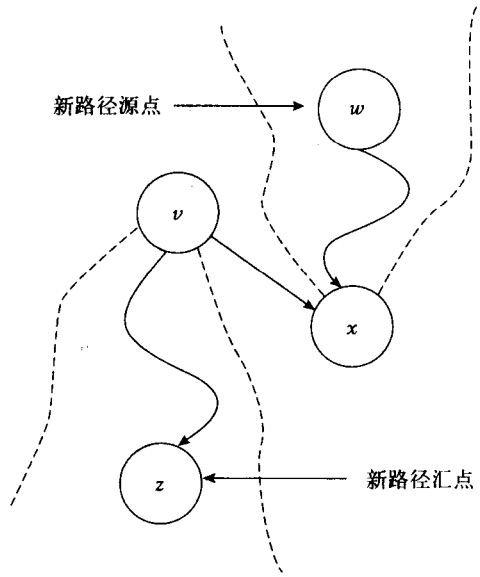


图8-31 解决路径更新问题

为避免这些冗余的工作，我们将通过*newPathSources*中的顶点后退，小心不要置位*pathTo*位超过一次。同样地，我们将向下移过*newPathSinks*以置位*pathFrom*位至多一次。关键的过程是图8-32中所示的*UpdateSlice*。

注意，设计*UpdateSlice*为在不同的调用中分别更新*pathFrom*和*badPathFrom*信息，方法是将不同的参数传递到*pathFrom*。此外，通过改变*cesors*和*pcesors*的定义（逆转图和大多数集合的角色），可调用*UpdateSlice*计算*pathTo*和*badPathTo*。这样，在每一次调用中，视遍历方向的不同，参数*pathFrom*可以用来代表*pathFrom*（或*badPathFrom*）集合，或者*pathTo*（或*badPathTo*）集合。可以证明需要最多调用*UpdateSlice* 8次来更新所有的路径信息。对沿向边的坍缩，4次调用就已足够。

既然对不同方向的调用结果是对称的，我们可以来分析一下更新*newPathSources*中的所有*pathFrom*集的一次调用的代价。对于由*newPathSources*表示的片中的每个顶点*w*，我们将计算*newPathFrom*[*w*]集合，它表示在坍缩之后从*w*到达的新顶点的集合。当处理每一个顶点时，我们将访问它的所有前驱。在每一个前驱*y*上，我们将检查*newPathFrom*[*w*]中的每一个顶点*z*，看它是否在*pathFrom*[*y*]中。如果*z*不在*pathFrom*[*y*]，我们就把它添加到*pathFrom*[*y*]和*newPathFrom*[*y*]。图8-32给出这一算法。

正确性 过程*UpdateSlice*的正确性可以通过对从*worklist*中移出元素的顺序作归纳来证明。因为*worklist*的第一个元素是*x*，我们可以确信它的*newPathFrom*集合得到正确的计算，因为按照我们的计算方法，它就是初始的*newPathSinks*集合。现在假定：对在顶点*w*前从*worklist*移出的所有顶点，*pathFrom*[*b*]和*newPathFrom*[*b*]是正确计算的。另外，我们假设当顶点*w*从*worklist*移出时，它有一个*pathFrom*集合是不正确的。这只能是因为*pathFrom*[*w*]有某个应该置位而没有置位的位。这样的位必与*newPathSinks*中的某顶点相对应。但是这意味着*w*的后继的*pathFrom*集合或*newPathFrom*集合中，这些位均未置位。但是既然从*w*到由该位表示的顶点

间必有一条边，那么某个需在w之前从*worklist*中移出的前驱也必有一个对应于该顶点集合的错误地置位的位。这就与假设矛盾。

```
procedure UpdateSlice(x, pathFrom, newPathSources, newPathSinks, cesors, pcesors)
  // x是要坍缩的顶点
  // pathFrom是要更新的集合
  // newPathSources是可以到达x的顶点的集合，但不包含那些被坍缩的顶点
  // newPathSinks是从那些被坍缩的顶点可达但从x不可达的顶点集
  // cesors是后继结点集合（如果从x后向遍历）
  // pcesors是前驱结点集合（如果从x后向遍历）
  // 在newPathSources中从x回溯更新pathFrom集，在newPathSinks中加入顶点
S0: newPathFrom[x] := newPathSinks;
L1: for each b ∈ newPathSources - {x} do newPathFrom[b] := Ø;
    worklist := {x};
L2: while worklist ≠ Ø do begin
      从worklist前端选一个元素w并将其从中删去;
L3:   for each y ∈ pcesors[w]且满足y ∈ newPathSources do begin
        if y ∉ worklist.ever then将y加入worklist;
L4:     for each z ∈ newPathFrom[w] do
S1:       if z ∉ pathFrom[y] then begin
            pathFrom[y] := pathFrom[y] ∪ {z};
            newPathFrom[y] := newPathFrom[y] ∪ {z};
          end
      end L3
    end L2
  end UpdateSlice
```

图8-32 对到达x或从x到达的部分顶点更新*pathFrom*集

复杂性 为了说明这种算法的时间复杂度在预期的上界之内，我们必须考虑实现的几个不同的方面。首先，在下一小节题为“快速集合实现”中，我们将描述如何表示整数集使得成员关系检查、插入和初始化都只需常数时间（初始化是最困难部分）。如果使用这种表示方法，我们就能在常数时间内把所有*newPathFrom*集和*worklist*初始化为空集。由于进入过程最多$O(V)$次，而且一片中最多有$O(V)$个顶点，循环L_1的代价以$O(V^2)$为界。同理，由于*newPathSources*中的所有顶点被加入*worklist*至多一次，循环L_2的循环体最多执行$O(V^2)$次。循环L_3访问一个顶点的所有前驱，因此它的循环体最多执行$O(EV)$次。

可惜在L_3的循环体中所做的工作包括循环L_4。我们必须采用某种方法来求出循环L_4的循环体所做工作的界，它包含一个if语句S_1。只有坍缩已使y到达z，且首次处理这一情形，才执行if语句的真分支。我们将此分支的常数时间记入被置为真的*pathFrom*矩阵的项中。在整个算法中，这个if语句的真分支至多执行$O(V^2)$次。

这样就剩下假分支。虽然这个假分支是空的，但为假分支测试本身表示需要常数时间。那么用产生的假条件执行多少次测试呢？因为$z \in newPathFrom[w]$，意味着*newPathFrom*集合中y的直接后继w已将其*pathFrom*集合中对应于z的位置为真。这样，我们将把此测试代价记入y和w之间的边上。由于任一给定的*pathFrom*位在这个算法中最多只能被置位一次，每一边最

443 ~ 447

多只能被计代价V次，对于每一个从那条边的汇点能到达的顶点计一次。这样，整个算法用于假分支条件测试的总代价就是$O(EV)$。

这些因素证明*pathFrom*和*badPathFrom*在整个算法中能够在$O(EV+V^2)$时间内更新。同样的分析可以得出对*pathTo*和*badPathTo*的更新也有相同的时间上界。

快速集合实现 实现这一算法的一个简单的方法是对所有集合，使用一个列表和一个相联位向量。遗憾的是，将位向量初始化为空集需要$O(V)$时间。这在过程*UpdateSlice*中是不可接受的，因为这将使得循环L_1的总代价为$O(V^3)$，而该循环的执行时间在这一算法中占主要部分。这样，我们需要找到一个允许常数时间初始化的表示，同时保持常数时间的成员关系检查和插入。此外，我们希望能使对整个集合进行迭代的时间与集合的大小成比例。

这些目标可以通过使用一种众所周知但很少讨论的策略来达到，这是一种避免由索引数组表示的集合的初始化代价的方法。这一算法的基本思想（作为练习包含在Aho，Hopcroft和Ullman[10]所著的书中)是用长度为V的两个整数数组作为集合的表示。数组Q按元素$Q[next]$到元素$Q[last]$的顺序包含队列中所有的顶点。数组In将用于元素测试——$In[v]$将包含顶点v在Q中的位置。

在队列的末尾加入一个新的顶点y需要执行下面的步骤:

$$last := last+1;\quad Q[last] := y;\ In[y] := last;$$

注意，这使得元素$In[y]$指向Q中的一个位置，其范围在[0: *last*]中而且包含y。这样，对顶点z是否是Q的一个成员的测试就是

$$next \leqslant In[z] \leqslant last \textbf{ and } Q[In[z]] = z;$$

图8-25中所需的z是否曾经是Q的成员的测试就是

$$0 \leqslant In[z] \leqslant last \textbf{ and } Q[In[z]] = z;$$

图8-33用图形描述队列表示。注意顶点4不可能被误认为是Q的一个成员，因为没有正确范围内的Q的元素指回4。

使用这种表示，我们可以在常数时间内执行所有的队列操作，包括初始化操作。这一数据结构将在这种算法中被用于许多集合的表示。

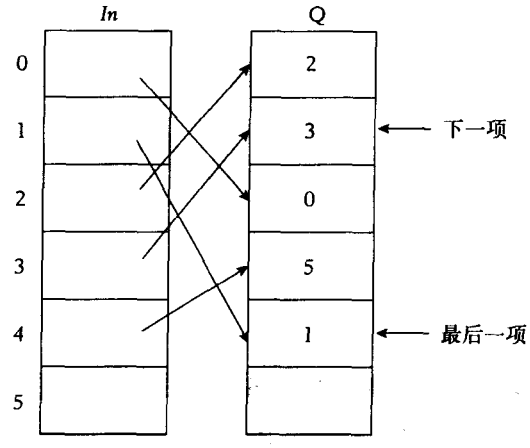


图8-33 快速队列表示

最后的观察 总计起来，这个算法的总代价包括：前5步为$O(EV+V^2)$、第（6）a）步为$O(EV)$，第（6）b）步为$O(EV+V^2)$。因此，整个算法需要$O(EV+V^2)$时间。

任何合并算法的一个关键点就是要重新计算坍缩的顶点和某个前驱或后继之间的权值。在前面介绍的算法中，通过将与原来的两个顶点相连的边上的代价加起来，计算出合并后的边的代价。在很多情况下，这并不是最好的方法。例如，在面向重用的合并问题中，我们可能在三个不同的循环中使用同样的数组变量。假设每个循环执行100次迭代，则三个循环的图看起来就像图8-34。

在这个例子中，把顶点x和y合并起来，并用加法重新计算权值将得出x和z之间的合计权值为200。实际上，在合并后总的额外的重用是100。在这种情况下，使用最大值作为重新计算权值的操作更为恰当。

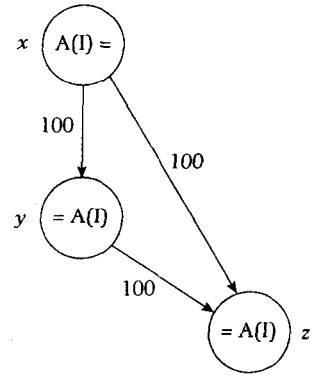


图8-34 权值计算例子

这种算法的一个优点是边的重新计算权值至多执行$O(E)$次。这意味着我们可以使用一种比较昂贵的重新加权算法而不会对执行时间的上界有太大影响。例如，假设在程序中有A个不同的数组，如果我们对每个顶点用一个长度为A的数组来表示循环中每个数组的使用次数，那么我们可以如下进行重新计算权值：首先取正在坍缩的两个顶点的数组的使用次数的最大值，然后对复合顶点的各个后继的对应数组之间的最小权值求和。这种以数组为单位的计算可以在$O(A)$时间内完成。因此，整个合并算法需要$O((E+V)(V+A))$时间。如果$A=O(V)$，那么这种算法就与使用加法进行重新计算权值的合并算法具有相同的渐近时间复杂度上界。

虽然这种算法在实际中应该产生好的效果，但是图8-35显示它并不总是产生最优结果。在这个例子中，最高的权值边就是带有权值11的(a, f)。如果我们坍缩这条边，最后的合并集将是$\{a, b, c, d, f\}$，而所有边的总权值将是16。然而，通过把c和d与b合并，我们已排除了c和d与e合并，因为从b到e的路径上有坏顶点。如果我们把c和d与e和f合并的话，节省的总的权值将是22，这是更好的结果。

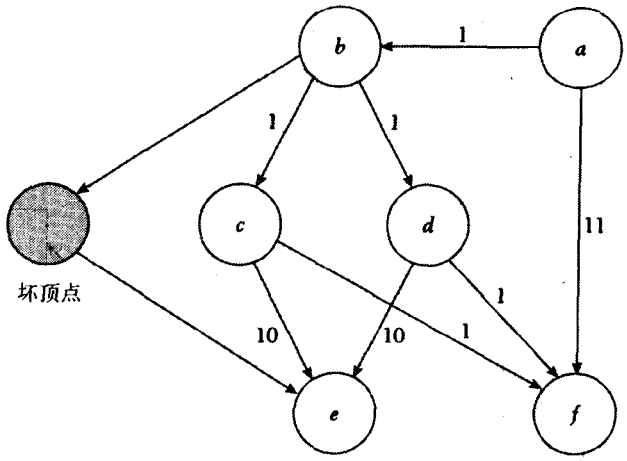


图8-35 贪婪加权合并的非优化性

8.6.5 面向寄存器重用的多层循环合并

下面我们将讨论多层循环嵌套问题。对于多层合并的基本策略是先在最外层进行合并，然后递归地合并所形成的循环的体。

这种特别简单的方法有一个非常重要的复杂情况。当在一外层循环上进行合并而作循环对齐时，我们必须仅对齐包含外层循环索引的索引，而假设内层循环索引将另行对齐。

为了说明这一原则，我们看下面的例子，这是用来表示二维松弛计算的：

```
DO J = 1, 1 000
   DO I = 1, 1 000
      A(I, J) = B
   ENDDO
ENDDO
DO J = 2, 999
   DO I = 2, 999
      A(I, J) = A(I+1, J) + A(I - 1, J) + A(I, J+1) + A(I, J - 1)
   ENDDO
ENDDO
```

外层循环的对齐应该仅仅考虑出现J的索引。这样，对齐将产生

```
DO J = 0, 999
   DO I = 1, 1 000
      A(I, J + 1) = B
   ENDDO
ENDDO
DO J = 2, 999
   DO I = 2, 999
      A(I, J) = A(I+1, J) + A(I-1, J) + A(I, J + 1) + A(I, J - 1)
   ENDDO
ENDDO
```

448 ~ 451

合并后变成

```
DO J = 0, 1
   DO I = 1, 1 000
      A(I, J + 1) = B
   ENDDO
ENDDO
DO J = 2, 999
   DO I = 1, 1000
      A(I , J + 1) = B
   ENDDO
   DO I = 2, 999
      A(I, J) = A(I + 1, J) + A(I - 1, J) + A(I, J + 1) + A(I, J - 1)
   ENDDO
ENDDO
```

现在，当对第二个循环的循环体作对齐时，我们需要关心的惟一引用就是A(I,J+1)，所以不需要作任何对齐。再一次合并产生

```
DO J = 0, 1
   DO I = 1, 1 000
      A(I, J + 1) = B
   ENDDO
ENDDO
DO J = 2, 999
   A(0, J + 1) = B
   DO I = 2, 999
      A(I, J + 1) = B
      A(I, J) = A(I + 1, J) + A(I - 1, J) + A(I, J + 1) + A(I, J - 1)
   ENDDO
```

```
   A(1 000, J + 1) = B
ENDDO
```

标量替换后，内循环将变成

```
DO J = 2, 999
   tA1 = tB ! moved into a register earlier
   A(0, J + 1) = tA1
   tA0 = A(1, J)
   DO I = 2, 999
      tA1 = tB
      A(I, J + 1) = tA1
      tA0 = A(I + 1, J) + tA0 + tA1 + A(I, J - 1)
      A(I, J) = tA0
   ENDDO
   A(1 000, J + 1) = B
ENDDO
```

452

这段代码在每一次迭代中节省两次引用（总计6次引用），所以这段代码将比简单生成的代码快1.5倍。这种改进一半应该归功于两层循环合并。使用展开和压紧可以取得更进一步的改进。

图8-36给出整个合并算法。

```
procedure CompleteFusion(R)
  // 参数R是一个包含一些循环和一些不在任何循环中的赋值语句的代码区域
  if R中没有循环then return;
  使用WeightedFusion算法选择一组循环作合并;
  for each合并集合中的集合S do begin
    沿外层循环的索引变量对齐S中的循环;
    使用FuseLoops产生一组合并的循环;
  end
  for each合并后R中的循环l do CompleteFusion(body(l));
end CompleteFusion
```

图8-36 多层循环合并

8.7 改进寄存器使用的变换综合

现在来讨论怎样把改进寄存器使用的不同变换综合起来，我们用矩阵乘法的完整处理过程来说明一些基本原理。

8.7.1 决定变换的顺序

在转向讨论寄存器改进策略的扩充例子之前，我们应先谈论一下本章中讨论过的变换的顺序问题。排序的目标之一是尽可能少地改变原来的代码。就是说，除非它能改进代码的性能，否则我们就不应作任何改变。假定有两个对性能的改进大致相同的选择，我们应该选择
453 对代码的改变最少的那个。尽可能地使代码保持原状有助于程序员理解这些改变。

按照前面的讨论，我们列出推荐的寄存器分配的变换顺序：

（1）循环交换：应该先做循环交换是因为它将重用带入最内层循环，这应是优先的。循环合并可能与此过程冲突。

（2）循环对齐与合并：它可以得到跨越循环的额外重用，特别是当所编译的代码是由像Fortran 90(见13.2节)前端这样的预处理器生成的时候。

（3）展开和压紧：当在循环交换之后存在由非最内层循环携带的依赖时，它可以实现外层循环的重用。

（4）标量替换：通过用标量变量来替换可以重用的数组引用，产生为标准的基于着色的寄存分配器。

这4个步骤的应用在下一节将用例子来说明。

8.7.2 例子：矩阵乘法

我们以典型的矩阵乘法为例来说明寄存分配变换的应用。我们先从适合于向量机算法版本开始。在科学计算程序中，将代码重排以便更好地向量化是非常常见的。当这些代码被移植到标量单处理器机器上时，它们很少会被重写以利用这些机器的特点。

```
      DO I = 1, N
        DO J = 1, N
S0        C(J, I) = 0.0
        ENDDO
        DO K = 1, N
          DO J = 1, N
S1          C(J, I) = C(J, I) + A(J, K) * B(K, I)
          ENDDO
        ENDDO
      ENDDO
```

通过将沿连续的列元素迭代的索引为J的循环置于最内层，这段代码可以在两个循环嵌套中进行长向量操作。

图8-37显示涉及循环中两个语句的依赖。从图中我们可以看出，K-循环携带与语句S_1有关的大部分依赖。 454

很清楚，我们应该把K-循环交换到围着S_1语句的最内层位置，产生

```
DO I = 1, N
  DO J = 1, N
    C(J, I) = 0.0
  ENDDO
  DO J = 1, N
    DO K = 1, N
      C(J, I) = C(J, I) + A(J, K) * B(K, I)
    ENDDO
  ENDDO
ENDDO
```

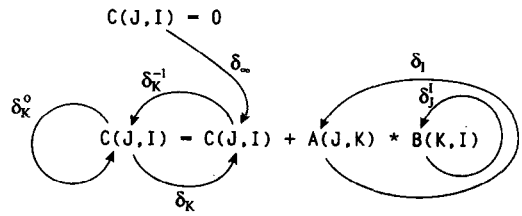


图8-37 矩阵乘法涉及的依赖

现在，我们对I-循环的循环体作循环合并，产生

```
DO I =1, N
  DO J = 1, N
    C(J, I) = 0.0
    DO K = 1, N
      C(J, I) = C(J, I) + A(J, K) * B(K, I)
```

```
      ENDDO
   ENDDO
ENDDO
```

接下来，我们对两个外层循环作因子为2的展开和压紧，得到如下代码:

```
DO I = 1, N, 2
   DO J = 1, N, 2
      C(J, I) = 0.0
      C(J + 1, I) = 0.0
      C(J, I + 1) = 0.0
      C(J + 1, I + 1) = 0.0
      DO K = 1, N
         C(J, I) = C(J, I) + A(J, K) * B(K, I)
         C(J + 1, I) = C(J + 1, I) + A(J + 1, K) * B(K, I)
         C(J, I + 1) = C(J, I + 1) + A(J, K) * B(K, I + 1)
         C(J + 1, I + 1) = C(J + 1, I + 1) + A(J + 1, K) * B(K, I + 1)
      ENDDO
   ENDDO
ENDDO
```

经过标量替换这段代码变为

```
DO I = 1, N, 2
   DO J = 1, N, 2
      s0 = 0.0;  s1 =0.0;  s2 = 0.0;  s3 = 0.0
      DO K = 1, N
         r1 = A(J, K);       r2 = B(K, I)
         r3 = A(J + 1, K);  r4 = B(K, I + 1)
         s0 = s0 + r1 * r2; s1 = s1 + r3 * r2
         s2 = s2 + r1 * r4; s3 = s3 + r3 * r4
      ENDDO
      C(J, I) = s0;       C(J + 1, I ) = s1
      C(J, I + 1) = s2;   C(J + 1, I + 1) = s3
   ENDDO
ENDDO
```

如果每一个标量变量都分配到一个寄存器的话，就需要8个寄存器，但是内层循环中的8个浮点操作的计算需要4次取数。此外，有足够的操作填满多达4个阶段的加法流水线。一般来说，如果展开因子是m，这个策略能够取得在内层循环中执行m^2个浮点操作和$2m$次取数的速度。

8.8 复杂的循环嵌套

实际上，真实程序中的循环通常比迄今为止我们作为例子使用的循环要复杂得多。例如，要在密集的线性代数计算上做出好的工作，需要有关变换系统来处理包含if语句的循环、三角形循环和梯形循环以及LU分解中的那些特殊的循环嵌套。本节将介绍处理这些循环嵌套的一些方法。

8.8.1 包含if语句的循环

遗憾的是，在8.3节中介绍的标量替换策略不能自然地扩展到有条件的代码。看下面的例子:

```
   DO I = 1, N
5     IF (M(I) .LT. 0) A(I) = B(I) + C
```

```
10      D(I) = A(I) + E
        ENDDO
```

由于对A(I)的赋值和使用产生的从语句5到语句10的真依赖不会显露出对A(I)的赋值是有条件的。只使用依赖信息，一个简单的标量替换方法会产生下面的代码:

```
        DO I = 1, N
5       IF (M(I) .LT. 0) THEN
           a0 = B(I) + C
           A(I) = a0
        ENDIF
10      D(I) = a0 + E
        ENDDO
```

这段代码是错误的，因为一个假的谓词导致a0没有定义，从而在语句10中产生不正确的a0值。

解决这个问题的方法非常简单：我们通过在if语句的假分支处插入一个对a0的从A(I)的取数，就可以保证a0具有正确的值:

```
        DO I = 1,N
5       IF (M(I) .LT. 0) THEN
           a0 = B(I) + C
           A(I) = a0
        ELSE
           a0 = A(I)
        ENDIF
10      D(I) = a0 + E
        ENDDO
```

用这种方法插入指令的危险在于可能增加沿某些路径执行这个程序时的时间。不过，在这个例子中，这并不成为一个问题——如果执行真分支，对A(I)的取数就可以避免，而如果执行假分支，就会插入一个取数，而另一个取数会被删除。在任何一种情况下，运行时间都不会增加，除非插入一个额外的分支语句。 457

我们算法的目标是要应用这些变换后不会造成运行时间的增加。一个与此目标相同的优化策略是*部分冗余消除*，该方法试图消除在给定的执行路径上的两个相同计算中的后一个。一个计算称为*冗余的*，如果在每一条到达它的路径上，都有一个相同的计算先于它出现。在这种情况下，我们就说它在每一条路径是都是*提前的*。如果这个计算只在某些到达它的执行路径上是提前的，我们就说它是*部分冗余的*。为了消除部分冗余计算e，编译器必须在每一条不提前的路径上复制e。正如文献[215]中的介绍，部分冗余消除的一个基本特性是，它保证不增加任何路径上的计算次数。

为使部分冗余消除适应标量替换，我们真正感兴趣的是对在每一条到达使用的路径上尚未初始化的变量插入初始化操作。为此我们注意到，在程序的某一给定点p处，如果一个变量是*活跃的*，它就有可能是未初始化的——存在一条不含赋值的到达使用的路径——且存在一条从程序的入口块开始的不包含初始化操作的路径。我们的目标是在每一条不包含初始化操作的路径上插入一个初始化表达式。进一步，我们将在不在未被初始化的任何路径上的最后基本块的末尾插入这个初始化操作。

为简单起见，我们假设程序中的每一条if语句都有else分支（可能为空）。这种人为的条件可以通过在为循环建立控制流图时插入空结点而加入。下面介绍几个变量:

- $live_{out}(b)$ 是在基本块b的出口处活跃的临时变量的集合。$live_{in}(b)$ 是在b的入口处活跃的变量集合。
- $alive_{out}(b)$ 是绝对活跃的变量集合，即在从b的出口到图的出口结点的任何路径上都是活跃的。$alive_{in}(b)$ 包含在b的入口处是绝对活跃的变量集合。
- $init_{in}(b)$ 是在b的入口处已被初始化的变量集合。$init_{out}(b)$ 是在b的出口处已被初始化的变量集合。
- $prnit_{in}(b)$ 是在b的入口处已部分被初始化的变量，即在到达b的入口的某些路径上被初始化的变量集合。$pinit_{out}(b)$ 是在到达b的出口已部分初始化的变量集合。

我们希望在这样的基本块中插入初始化操作，在其中变量不是部分初始化的，但是在它的后继中是部分初始化的。更进一步，该变量在这个基本块的出口处应该是绝对活跃的。这些条件保证我们绝不会在任何路径上将变为冗余的一点插入一个取数操作。

458 下面的等式定义变量的*alive*、*init*和*pinit*:

$$alive_{out}(b) = \bigcap_{c \in succ(b)} alive_{in}(c)$$
$$alive_{in}(b) = (alive_{out}(b) - killed(b)) \cup used(b) \tag{8-1}$$

$$init_{in}(b) = \bigcap_{a \in pred(b)} init_{out}(a)$$
$$init_{out}(b) = init_{in}(b) \cup assigned(b) \tag{8-2}$$

$$pinit_{in}(b) = \bigcup_{a \in pred(b)} pinit_{out}(a)$$
$$pinit_{out}(b) = pinit_{in}(c) \cup assigned(b) \tag{8-3}$$

现在我们可以给出插入的条件。$insert_{out}(b)$ 集合是必须在b的末端插入初始化操作的临时变量集合。$insert_{in}(b)$ 集合是必须在b的起始处插入初始化操作的变量的集合。首先，如果一个变量在一个基本块中使用但没有在到达该基本块的任何路径上初始化，我们就必须在该基本块的起始处插入这个变量:

$$insert_{in}(b) = used(b) - pinit_{in}(b) \tag{8-4}$$

如果一个变量在到达一个基本块的任何路径上未被初始化，在这个基本块的出口处是绝对活跃的，且在这个基本块的某个后继处是部分可用的，我们就在这个基本块的尾部插入对这个变量的初始化:

$$insert_{out}(b) = alive_{out}(b) \cap \neg pinit(b) \cap (\bigcup_{c \in succ(b)} pinit(b)) \tag{8-5}$$

在每个插入点，我们插入与临时变量相对应的下标引用到临时变量的赋值操作。在最初的标量替换中，这个引用是可以省去的。

8.8.2 梯形循环

许多在实际程序中出现的循环并不是规则的，对矩形结构的循环，本章所介绍的变换是很有效的。特别是，许多循环的上下界会随着外层循环的索引在同一循环嵌套中变化。在本小节中我们将介绍如何改造那些改进寄存器使用的变换来处理这些梯形循环。

三角形展开与压紧

如果恰当地划分这些循环，我们可以使用前面几节中介绍的技术将展开和压紧以及循环

分块用于梯形循环。以下面的循环为例: 459

```
DO I = 2, 99
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
   ENDDO
ENDDO
```

我们想要使用展开和压紧来变换这个循环以便不仅能重用A(I,I)的当前值，还能重用A(J,J)的值。问题是在梯形循环中不能直接用展开和压紧技术。

如果对外层循环展开两次，就得到

```
DO I = 2, 99, 2
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
   ENDDO
   DO J = 1, I
      A(I + 1, J) = A(I +1, I + 1) + A(J, J)
   ENDDO
ENDDO
```

现在注意，如果对第二个内层循环嵌套的最后一个迭代作循环剥离，我们就可以把头两个循环压紧在一起。这将产生

```
DO I = 2, 99, 2
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
      A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
   ENDDO
   A(I + 1, I) = A(I + 1, I + 1) + A(I, I)
ENDDO
```

如果现在执行标量替换，我们得到

```
tI = A(2, 2)
DO I = 2, 99, 2
   tI1 = A(I +1, I + 1)
   DO J = 1, I - 1
      tJ = A(J, J)
      A(I, J) = tI + tJ
      A(I + 1, J) = tI1 + tJ
   ENDDO
   A(I + 1, I) = tI + tI1;  tI = tI1
ENDDO
```

这段代码在内层循环的每次迭代中需要作一次取数和两次存数——这是对简单版本的每次迭代的两次取数的改进，假设即使是简单的寄存器分配器也能检测到内层循环中A(J,J)的重用。图8-38说明这种策略。这里我们看到三角形循环被分割为每块有两个J-循环迭代的矩形块和少于2次迭代（即1次迭

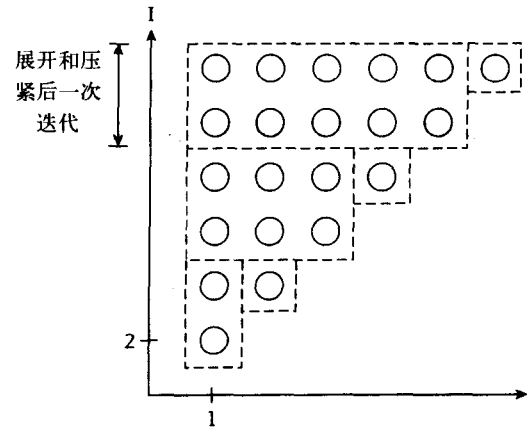


图8-38 三角形展开和压紧

代）的三角形循环。

这种方法可以推广为展开多于2次，但是代码会更复杂。看看外层循环展开3次的结果就清楚了。这里，我们已调节循环上下界使之可以被3整除，在实际中会有一个前置循环来处理这一点。

```
DO I = 2, 100
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
   ENDDO
ENDDO
```

如果将外层循环展开三次，就得到

```
DO I = 2, 100, 3
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
   ENDDO
   DO J = 1, I
     A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
   ENDDO
   DO J = 1, I + 1
      A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
   ENDDO
ENDDO
```

460 ~ 461

再次合并，我们得到

```
DO I = 2, 100, 3
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
      A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
      A(I + 2, J) = A(I + 2, I + 2) + A(J, J)
   ENDDO
   DO J = I, I
      A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
   ENDDO
   DO J = I, I + 1
      A(I + 2, J) = A(I + 2, I + 2) + A(J, J)
   ENDDO
ENDDO
```

展开最后两个内层循环，我们得到

```
DO I = 2, 100, 3
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
      A(I + 1, J) = A(I + 1, I + 1) + A(J, J)
      A(I + 2, J) = A(I +2, I + 2) + A(J, J)
   ENDDO
   A(I + 1, I) = A(I + 1, I + 1) + A(I, I)
   A(I + 2, I) = A(I + 2, I + 2) + A(I, I)
   A(I + 2, I + 1) = A(I + 2, I + 2) + A(I + 1, I + 1)
ENDDO
```

标量替换后的代码是

```
tI = A(2, 2);  tI1 = A(3, 3)
DO I = 2, 100, 3
   tI2 = A(I + 2, I + 2)
   DO J = 1, I - 1
      tJ = A(J, J);            A(I, J) = tI + tJ
      A(I + 1, J) = tI1 + tJ; A(I + 2, J) = tI2 + tJ
   ENDDO
   A(I + 1, I) = tI1 + tI;
   A(I + 2, I) = tI2 + tI
   A(I + 2, I + 1) = tI2 + tI1
   tI = tI1;   tI1 = tI2                                    462
ENDDO
```

这段代码在内层循环的每次迭代减少了三个取数。

梯形展开和压紧

为了理解怎样把前一节中的方法扩展到梯形循环，我们看一个非常重要的例子——简单的卷积。这个循环出现在许多地震分析应用中，特别是在石油工业中。下面的版本是从地球物理的地震分析的真实代码中抽取出来的，并且占据了那个应用代码的大部分运行时间:

```
DO I = 0, N3
   DO J = I, MIN(N1, I + N2)
      F3 (I) = F3 (I) + F1(J) * W(I - J)
   ENDDO
   F3 (I) = F3 (I) * DT
ENDDO
```

显然，变量F3(I)在整个内层循环的迭代中将被保存在某个寄存器。很容易看出，通过展开和压紧可以得到的变量F3(I)的重用。然而，展开和压紧还可以提供对变量W(I-J)的重用。

在对这个循环嵌套运用展开和压紧之前，我们把它分为两部分——I + N2 < N3的部分和余下的一部分——使得每一个循环嵌套中的内层循环有一个一致的上界:

```
DO I = 0, N3 - N2
   DO J = I, I + N2
      F3 (I) = F3 (I) + F1(J) * W(I - J)
   ENDDO
   F3 (I) = F3 (I) * DT
ENDDO
DO I = N3 - N2 + 1, N3
   DO J = I, N1
      F3 (I) = F3 (I) + F1(J) * W(I - J)
   ENDDO
   F3 (I) = F3 (I) * DT
ENDDO
```

第二个循环嵌套不再是梯形的，因为上界就是N3。因此，我们可以把注意力放在第一个循环嵌套。对外层循环展开一次，得到 463

```
DO I = 0, N3 - N2, 2
  DO J = I, I + N2
     F3 (I) = F3 (I) + F1(J) * W(I - J)
```

```
   ENDDO
   F3 (I) = F3 (I) * DT
   DO J =  I+1, I + N2 + 1
      F3 (I + 1) = F3 (I + 1) + F1(J) * W(I - J + 1)
   ENDDO
   F3 (I + 1) = F3 (I + 1) * DT
ENDDO
```

合并内层循环的公共迭代产生

```
DO I = 0, N3 - N2, 2
   F3 (I) = F3 (I) + F1(I) * W(0)
   DO J = I + 1, I + N2
      F3 (I) = F3 (I) + F1(J) * W(I - J)
      F3 (I +1) = F3 (I + 1) + F1(J) * W(I - J + 1)
   ENDDO
   F3 (I + 1) = F3 (I + 1) + F1(I + N2 + 1) * W( - N2 - 1)
   F3 (I) = F3 (I) * DT
   F3 (I +1) = F3 (I + 1) * DT
ENDDO
```

对这个循环作标量替换，产生

```
DO I = 0, N3 - N2, 2
   f3I = F3 (I);  f3I1 = F3 (I + 1)
   f1I = F1(I);   wIJ0 = W(0)
   f3I = f3I + f1I * wIJ0
   DO J = I + 1, I + N2
      f1J = F1(J);  wIJ1 = W(I - J)
      f3I = f3I + f1J * wIJ1
      f3I1 = f3I1 + f1J * wIJ0
      wIJ0 = wIJ1
   ENDDO
   f1J = F1(I + N2 + 1);  wIJ1 = W( - N2 - 1)
   f3I1 = f3I1 + f1J * wIJ1
   F3 (I) = f3I * DT;  F3 (I + 1) = f3I1 * DT
ENDDO
```

注意，因为在下标表达式中减去了循环索引J，所以对W(I-J)的重用发生在下一次迭代中的第二个语句实例上。

464

在Carr和Kennedy在MIPS M120上所作的实验中，改进的代码在只有100个元素的数组上取得了表8-3所示的加速比。

表8-3 卷积时间

原始代码	15.59秒
变换后代码	7.02秒
加速比	2.22

8.9 小结

我们介绍了许多改进现代单处理器CPU寄存器中数组值的重用的基于依赖的技术。这些方法对浮点寄存器尤其有用，因为它们经常被用来保存数组数据结构的单个元素。因此，传统的基于图着色的寄存器分配策略对这些寄存器是低效的。方法包括：

- 标量替换，将下标变量的重用暴露给有好的寄存器分配器的标量编译器。
- 展开和压紧，变换程序以开发外层循环中的寄存器重用。
- 循环交换，把携带重用最多的循环移到最内层的位置。

• 对齐的循环合并，把一个程序不同循环中的引用合并到一个循环中。

这些技术已经被证明适用于包含控制流和有梯形迭代范围的循环。本章的重要贡献是介绍了一种使用贪婪启发式方法来解决加权合并问题的快速算法，它是由于考虑循环合并的有利性而自然产生的。

8.10 实例研究

由Carr，Callahan和Kennedy[55，64，65，67]提出的策略的最早实现是在PFC系统中，然后被移植到ParaScope。在8.3.8节和8.4.3节介绍的所有有效性研究都是在这一框架结构上完成 465
的。实现方面的主要考虑是加入对输入依赖的测试。尽管测试本身可以通过简单地修改依赖分析加以实现，但输入依赖的总数可能大得足以给编译器造成麻烦。特别是标量的依赖可能造成依赖图大小的巨大膨胀，因为可以在循环嵌套的每一层上携带它们。在PFC系统中，我们使用标量依赖的概要表示来解决这个问题，即用一条边概括各层的依赖。可以相当简单地扩展这一策略来概括所有的方向向量。有了这些改动，尽管依赖图仍然很大，但是其大小已是可以处理的。

标量替换和相关的变换在最初的Ardent Titan中特别重要，这是因为浮点操作是在向量部件上执行，所有取数和存数都绕过了高速缓存。换句话说，浮点值不会在高速缓存中，所以有效地利用寄存器就非常关键。除了不构造输入依赖边，编译器像本章所介绍的那样作标量替换，因此它失去了很多改进的机会。然而，Titan编译器的标量替换阶段，展开与压紧和基于输出依赖的存数消除一起，产生了非常好的效果。10.5节中的表10-1介绍对Livermore循环应用依赖分析的研究结果。在6个标量替换作为重要优化的核心程序中，由依赖和标量替换所得的改进从7.4%直到200%。这些结果以及那些改进显著的核心与图8-10中报告的Livermore循环上的PFC结果吻合得很好。

最初的PFC实现和Ardent Titan编译器都没有作对齐的循环合并来改进寄存器分配。尽管对齐和合并已经作为改进高速缓存利用的策略而得到了研究[103，105]，但我们没有看到仅依靠合并后的更好的标量替换而导致单独改进的研究。

8.11 历史评述与参考文献

Allen和Kennedy在论文“向量寄存器分配”[22]以及Allen的论文[16]中首先提出了用依赖在向量机上优化寄存器使用。John Cocke提出在RS/6000（最早的由于高速缓存不命中而产生很长时延的机器之一）上使用展开和压紧技术。这一想法在Callahan，Cocke和Kennedy的论文[55]中首次发表。使用标量替换来取得单处理器上的寄存器重用的思想是由Carr和Kennedy提出的[67]。Carr在与Callahan和Kennedy合作的一系列的论文[65，66]中报告了标量替换以及展开与压紧的实现及其效果。依赖图剪枝的算法是由Brandes[43]，Rosene[237]和Carr[64]提出的。处理复杂循环嵌套的算法也是由Carr和Kennedy[66]提出的。 466

循环对齐最初是由Allen，Callahan和Kennedy作为一种在并行化中消除携带依赖的方法提出的[25]。在合并后运用对齐来改进寄存器重用的方法是新颖的，尽管它作为一种改进高速缓存重用的机制已被研究过[103，105]。

本章介绍的贪婪加权合并算法是由Kennedy[171，172]提出的。Kennedy和McKinley[176]给出了加权合并是NP完全问题的最早的证明，并提出了一种基于重复应用Goldberg和Tarjan的网络最大流算法的启发式策略[176]。该算法需时$O(kEV \log(V^2/E))$，其中$k < V$是所需应用的最

大流算法的次数。该算法所得的解比由贪婪加权合并算法求得的解的好还是坏并不清楚。Gao，Olsen，Sarkar和Thekkath[118]给出了边的惟一权值是1或0的0-1合并问题的解决方法。这种用来支持数组紧缩[242]的局部性优化方法先用了一个$O(EV)$的前置遍来减小图的大小，然后在简化图$G_R=(E_R, V_R)$上连续应用最大流算法。整个算法需要$O(EV+V_R^2E_R \log V_R)$时间。因此，Gao算法产生一个解决加权合并问题的一个子问题的启发式方法，其时间复杂度比贪婪加权合并略差。然而，因为这些算法所得到的解可能由于使用了不同的启发式规则而不同，因此很难比较它们之间孰优孰劣。Meggido和Sarkar对一个循环合并的最优算法做了实验，实验显示对小的例子其执行时间是合理的[211]。贪婪加权合并算法的各种替代算法都采用加法作为重新加权的操作，这使得它们在需要更复杂的重新加权策略的存储层次上的效果不够理想。

习题

8.1 用标量替换手工变换下面的程序。假设数组A只用在这个循环中。在标量替换后，这个程序还需要数组A吗？你能否形式化从一个程序中去掉一个数组的条件？

```
DO I=1, N
   A(I)=B(I)+3.0
   SUM=SUM+A(I)
ENDDO
```

467

8.2 手工变换下面的循环嵌套以求得到高度的寄存器重用。你使用了哪些变换？在变换前后浮点操作与取数的比率是多少？你假定有多少个寄存器？

```
DO I=1, N
   DO J=1, N
      A (I+1, J+1) =A (I, J+1) +A (I+1, J) +B (J)
   ENDDO
ENDDO
```

8.3 针对不同的N的值，包括非常大的（比高速缓存大的）值，在你最喜欢的机器上（使用Fortran编译器）运行习题8.2中的循环嵌套。报告并解释最后的结果。

8.4 访问同一数组A(1:N)的两个循环总可以使用循环对齐来合并吗？如果可以，予以证明；如果不可以，给出反例。

8.5 考虑一个简单的贪婪加权合并算法。该算法迭代地选择权值最大的边，然后沿从汇点到源点的所有路径前溯以决定合并是否合法。这个算法的渐进复杂度是什么？（或者说，为什么它如8.6.4节所给出的快速贪婪加权合并算法一样好？）

8.6 重作8.5.1节的例子，通过考虑输入依赖而改进性能。提示：这样可为每个迭代节省一个取数操作。

468

第9章

管理高速缓存

9.1 引言

虽然寄存器的重用和高速缓存的重用具有很多共性特点，但二者间也有显著的差别。首先，二者的基本储存单位不同。一个寄存器精确地包含一个字，高速缓存则被分成块（或行），每个块通常包含多个字。所以，寄存器的重用是由于连续访问同一个数据项而引起的。这种形式的重用通常称为时间重用，它在高速缓存的管理中同样是有用的。在另一方面，当一个高速缓存块被调入后，对同一块中的不同数据项的访问不会导致不命中。这种形式的重用——由于一些数据项的存储是邻近的，使得它们出现在同一个高速缓存块中——称为空间重用。

寄存器重用和高速缓存重用之间的第二类不同是由于高速缓存实现的方式导致的。例如，对于大多数高速缓存，向不在高速缓存中的块进行的写操作将导致一次不命中，使得在完成写操作前要先将涉及到的块读入高速缓存中。对寄存器的写入则无需这种读操作。这种性质的一个直接推论是反依赖在提高高速缓存的重用中起着重要的作用——如果引用可以被调整得与后续的写操作很接近，则写操作导致的高速缓存不命中是可以避免的。

最后，高速缓存的操作本质上讲是同步方式的；也就是说，在等待解决一次不命中时，处理器处于停顿的状态。当连续发生多次不命中时，这将严重地降低性能。这类性能问题迫使机器的设计者采取措施，使得多个高速缓存操作可以同时进行。一种常用的实现方式是增加一个预取操作，它可以作为机器的一条指令被使用。预取指令不会导致处理器的阻塞，除非在装入完成之前，对被装入的高速缓存块出现了一次引用。预取操作只是提供改善性能的机会；能否真正利用这些机会，取决于编译器能否适时地产生预取指令。这是9.5节要讨论的主题。

高速缓存的重用类型有两种，时间重用和空间重用，它们使得提高性能有两种常用的策略。在循环中的迭代访问连续的存储空间时——更确切地说，是当一个循环中每个迭代访问的存储位置与前面一个迭代访问的存储位置邻近时，可以获得最高的空间重用。所以，对于拥有长的高速缓存块的机器而言，获得好的高速缓存性能的关键是选择恰当的循环作为循环嵌套的最内层循环。例如，对于Fortran循环嵌套

```
DO I = 1, N
   DO J = 1, M
      A (I, J) = A (I, J) + B (I, J)
   ENDDO
ENDDO
```

最内层循环按行进行迭代。Fortran数组是以列优先的次序存储的，因此，对于引用A(I,J)，对于固定的J值，I-循环进行迭代时，对存储的访问将是相邻的。所以，对于拥有容纳多个字的高速缓存块的机器而言，对这两个循环进行交换将提高其性能：

```
DO J = 1, M
   DO I = 1, N
      A (I, J) = A (I, J) + B (I, J)
```

```
   ENDDO
ENDDO
```

对内层循环的正确选择不总是这么明了的。9.2节将讨论增强空间重用的循环交换策略。

在时间重用的情形面临的问题略有不同。假设目前我们在讨论的机器具有一个数据项占据一个单独的高速缓存行的特点。在这样一个机器上，时间重用是唯一的一种重用。现在再考虑8.1节的例子:

```
DO I = 1, N
   DO J = 1, M
      A (I) = A (I) + B (J)
   ENDDO
ENDDO
```

如果我们关注于高速缓存的行为，我们可以看到对于I的每个不同的值，引用A(I)只是在第一次访问时导致一次高速缓存不命中。在另一方面，由于几乎所有的高速缓存都采用“最
470 近最少使用”的替换策略，所以当M足够大时，对B(J)的每次访问都会导致一次高速缓存不命中。这意味着如果M大于整个高速缓存的字的数目，则B(1)很可能由于J-循环的后续迭代中的访问而被从高速缓存中替换出去。在这种情况下，这个循环嵌套导致的不命中次数是N * M。

对内层循环进行分段，使得一个分段的长度S与高速缓存相匹配，然后将在这些分段上进行迭代的循环移到最外层的位置，这样可以提高性能。结果如下:

```
DO J = 1, M, S
   DO I = 1, N
      DO jj = J, MIN(M, J + S - 1)
         A (I) = A (I) + B (jj)
      ENDDO
   ENDDO
ENDDO
```

如果S足够小，内层循环除了在对B(jj)进行第一次访问时不命中之外，不会再出现其他不命中。然而，我们在此引入了新的关于A(I)的不命中——对于J-循环的每一次迭代，A的每个元素将导致一次不命中，不命中的总次数是N * M/S。对于S的任何合理的值，这个不命中次数要远小于原有循环的不命中次数。

这种变换称为*循环分段和交换*，而通用的方法称为高速缓存分块。9.3节将更详细地讨论分块的策略。

9.2 适合于空间局部性的循环交换

考虑一个至少有两层循环的完全嵌套的循环嵌套。一个核心问题是，嵌套中的哪个循环应该在最内层？这个问题之所以重要是因为最内层循环决定了数组的哪一维被顺序地访问。当我们在8.5节中讨论适合于增强寄存器重用的循环交换时，我们只是试图使得内层循环拥有最多的依赖。然而，对于高速缓存而言，空间重用使得问题变得复杂了——通常，当以跨距1对连续的存储单元访问时，得到最好的效果。

既然Fortran的数组是以列优先的次序存储的，只有当循环是在列上迭代时才能获得跨距为1的访问。将这个循环移到最内层的位置可以提高空间局部性，使得每个高速缓存行的不命中率降为1次（假设下标系数是1）。作为一个例子，考虑下面的循环:

```
DO I = 1, N
   DO J = 1, M
      A (I, J) = B (I, J) + C
   ENDDO
ENDDO
```

471

按目前的形式，由于最内层循环是在数组不连续的维上进行依次访问，这个循环嵌套在每次访问数组A和每次访问数组B时都将导致一次不命中，总的不命中次数是2MN。如果交换循环，

```
DO J = 1, M
   DO I = 1, N
      A (I, J) = B (I, J) + C
   ENDDO
ENDDO
```

不命中次数将按一个因子b减少，b是一个高速缓存行容纳的字的个数，总共有2MN/b次不命中。

不幸的是，实际的情形通常不是这样显而易见，如下面代码所示:

```
DO I = 1, N
   DO J = 1, M
      D (I) = D (I) + B (I, J)
   ENDDO
ENDDO
```

在这个循环嵌套中，相对于外层循环的每次迭代，数组D的访问至多只有一次不命中。事实上，由于D(I)在内层循环的每个迭代都被访问，它及其包含它的整个高速缓存行将保留在高速缓存中，直至外层循环的下一次迭代开始。所以，实际的不命中次数是N/b，此处的b是一个高速缓存行容纳的字的个数。在另一方面，这个循环嵌套由于访问数组B导致的不命中次数是NM次。

循环交换有可能减少上述的不命中次数。循环交换可以将关于B的不命中次数减少至NM/b，但这将失去数组D中的原自然局部性，从而使得每访问（数组D）b次后引发一次不命中。对于修改后的循环，产生的不命中的总次数是2NM/b。如果我们将这两个结果进行比较，可以看到交换后的循环更好的条件是，如果

$$N/b + NM - 2NM/b > 0$$

上式可以化简为

$$M(b-2)+1>0$$

所以，当高速缓存行的大小至少是两个字大小时，循环交换才是有利的。

对循环的所有可能的置换进行详细的分析是代价巨大的，但是为了确定循环的次序，从而将时间重用性最大化，通常一个简单的启发式方法就可以获得非常好的结果。在下面的方法中，将循环嵌套中的每个循环看作是循环嵌套的最内层循环，对其导致的存储代价进行评估。在进行这个评估时，不必考虑循环是否可以移到最内层的位置，所以后续的循环重排阶段试图使得循环对齐的次序接近于这个启发方法建议的次序。 472

这种方法的基本思想是：假设被评估的循环位于最内层，为循环嵌套中的每个引用标注一个代价值。给定一个循环嵌套$\{L_1, L_2, \ldots, L_n\}$，我们将分三步为循环嵌套中的每个循环$L_i$定义一个*最内层存储访问代价函数*$C_M(L_i)$。首先，要找出循环嵌套中的引用并根据高速缓存不命中的情况为其指定一个存储代价，此处的高速缓存不命中是指当每个引用仅位于给定的循环时导致的不命中:

（1）一个引用如果与循环的索引变量无关，代价赋值为1。

（2）如果一个引用中出现的给定循环的索引变量是对非连续维进行步进，则给定代价值为N，此处的N是循环的迭代次数。

（3）如果一个引用中出现的索引变量是对连续维进行小跨距的步进，跨距为s，则代价由

$$\frac{N}{\left(\frac{b}{s}\right)}=\frac{Ns}{b}$$

确定，其中N是循环次数，b是高速缓存行的大小。

一旦这些代价值被确立，如下所述，代价被乘以与其他的每个循环相关的因子：

（1）对于每个引用，如果不随给定循环的循环索引变化，则代价值不变（用1相乘）。

（2）如果引用随给定循环的循环索引变化，则代价值与循环的迭代次数相乘。

当每个引用的代价值被确定后，对于给定循环，只需将每个引用的各自代价值累加起来，就可以得到其最内层存储访问代价。

为了使上述的方法有效，我们需要小心地定义引用的含义，以避免将对同一个高速缓存行的访问重复计算。所以，如果下面两个条件的任何一个被满足，则我们认为对于给定内层循环L，两个引用r_1和r_2是同一个引用组的一部分：

（1）这两个引用之间存在一个循环无关依赖，或者存在一个依赖，但这个依赖由循环阈值很小的循环L所携带，此处，“很小”的定义可能是变化的，但循环阈值为2总是可接受的。对于由于时间重用而命中同一个高速缓存行的引用，这个条件可以保证此类引用不会被重复计算。

（2）两个引用是对同一个数组的引用，且其形式只是在连续维上有很小的常数的不同，
473 此处的“很小”是指小于一个高速缓存行的大小。这个条件可以避免对由于空间重用而对同一个高速缓存行进行的访问重复计算。通常，对上述条件要推广：使得当两个引用在连续维上的距离大于一个高速缓存行时，要形成一个新的高速缓存组。所以如果高速缓存行包含4个元素，对于在同一个循环体中的引用A(I)，A(I+2)和A(I+4)，必须被看成是两个高速缓存组。

在做了上述修改后，代价函数保持不变，除非代价的计算是针对每个引用组的一个单独的引用并累加以确定给定循环的最内层存储访问代价。

一旦循环嵌套中的每个循环的最内层存储访问代价被计算出来，理想的循环次序可以如下得到：将最内层存储访问代价最小的循环放在最内层的位置，按照循环的最内层存储访问代价的递增顺序由最内层到最外层依次排列循环。

作为上述的启发方法应用的一个例子，考虑同我们的例子循环

```
DO I = 1, N
   DO J = 1, M
      D(I) = D(I) + B(I, J)
   ENDDO
ENDDO
```

相关的代价，其中有两个引用组，一个是对D(I)的，一个是对B(I,J)的。我们分别考虑每个循环的代价：

（1）当J-循环是最内层循环时，我们得到D(I)的代价是1，而B(I,J)的代价是M。这两个代价值都要与N相乘，在相加后得到总的代价是N+MN。

（2）如果I-循环是最内层循环，对于这两个引用，我们得到同样的代价N/b。对于外层循环，这两个代价值都要与M相乘，从而得到总的代价是2MN/b。

如果高速缓存行的长度超过两个字的长度，应该选择第二种排序，即将I-循环作为最内层循环，这是我们深入分析的结果。请注意在第一种情况中，启发式方法对不命中的次数重复计算了，因为没有注意到如果该循环是对连续维进行迭代，最内层循环的一个不变的引用在下一个最内层循环中会得到空间重用性。

当得到了一个理想的次序$O=\{\sigma(1), \sigma(2), ..., \sigma(n)\}$，其中$L_{\sigma(i)}$表示按照我们的启发式方法出现在第$i$个最外层位置的循环后，我们需要对循环重新排序，以便与理想的次序相一致。为了达到上述目的，我们使用一个观察结果，如果一个循环在最内层的位置可以得到比其他循环更多的重用，则它在外层的位置也会得到更多的重用。所以我们希望这个循环的放置在与 474
上述启发式方法给出的理想的位置最接近的位置。

我们将依据在图9-1中给出的过程完成这个处理，即反复进行移位操作，将P中的当前位置移到理想的次序中剩余的最外层的合法位置。这种方法得到的循环次序显然是合法的，因为在这个算法的整个过程中维持合法循环顺序的性质。

```
procedure PermuteLoops(N, O, n, P)
  // N={L1, L2, ..., Ln}是要进行置换操作的循环嵌套
  // O={σ(1), σ(2), ..., σ(n)}是理想次序的循环索引变量
  // n是循环嵌套中循环的个数
  // P={P1, P2, ..., Pn}是置换后最终的循环嵌套

  j:=1; // P中当前要填充的位置的索引
  P:=N; // 以原始的置换为起点
  while O≠Ø do begin
    令k是O中最左端的满足下述条件的元素: Lk可以被移到P中的位置j,
      且不会在这个循环嵌套的方向矩阵中引入任何非法的方向向量;
    将k从O中删除;
    将Lk移到Pj;
    j:=j+1;
  end
end PermuteLoops
```

图9-1 循环置换的算法

这个过程具有如下特点：如果N中的循环存在一个合法的置换序列，其中$L_{\sigma(n)}$是最内层的位置，则*PermuteLoops*将生成一个$L_{\sigma(n)}$位于最内层的置换序列。为看出这一点，请注意如果原来的循环是合法的，这个算法的每一步产生成一个合法的排序，因为只有当满足下述条件时循环才会被移位：在方向矩阵中，如果表示一个在先前的移动中被移到的外层位置的循环的行向量中，所有没有被其他方向覆盖的非“=”方向对应的列中，当前循环对应的位置是“<”。假设*PermuteLoops*不选择$L_{\sigma(n)}$作为最内层循环，而是将其向外移位到P_m的位置，其中$m<n$。那么移动过程必然经过每个在理想的次序中位于$L_{\sigma(n)}$外层的循环，只有当这些循环中的每个移位到P中的m位置是非法时，这种情况才会发生。或者说，在$L_{\sigma(n)}$外的每个循环，在$L_{\sigma(n)}$有“<”的某个位置它们有一个“>”方向并且这些循环都不会选择到包含m的外层位置。但是这不可能为真，因为那样就不会存在$L_{\sigma(n)}$在最内层的合法排列了——它将必须一直在那些循环外边。

从上述的论述中，我们知道由*PermuteLoops*放置在最内层的位置上的循环是携带有最多重用（以其代价来度量）的循环，并且是可以合法放置在那里的。这意味着如果代价模型是
475 准确的，则这种方法选择了最优的合法内层循环。

为了举例说明这个方法，我们回过头来，再一次看看许多教科书中出现的矩阵乘法的例子:

```
DO I = 1, N
   DO J = 1, N
      C(I, J) = 0
      DO K = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(I, J)
      ENDDO
   ENDDO
ENDDO
```

这段代码可以分布成两个循环嵌套

```
DO I = 1, N
   DO J = 1, N
      C(I, J) = 0
   ENDDO
ENDDO
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
```

在初始化的循环嵌套中，目前的结构导致的代价是N^2，而将I循环作为最内层循环时导致的代价是N^2/b。所以初始化的循环嵌套应该置换为

```
DO I = 1, N
   DO I = 1, N
      C(I, J) = 0
   ENDDO
ENDDO
```

在进行计算的循环体中，有三个引用组，一个用于C，一个用于A，另一个用于B。表9-1给出将这三个循环分别放在最内层位置时的代价。

表9-1 矩阵相乘的存储访问分析

循环	C(I,J)	A(I,K)	B(K,J)	总计
I	N^3/b	N^3/b	N^2	$2N^3/b+N^2$
J	N^3	N^2	N^3	$2N^3+N^2$
K	N^2	N^3	N^3/b	$N^3(1+1/b)+N^2$

上述的分析指导我们选择I-循环作为最内层循环，K-循环作为次最内层循环，而J-循环作为最外层循环:

```
DO J = 1, N
   DO K = 1, N
```

```
      DO I = 1, N
        C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
    ENDDO
ENDDO
```

在三种不同的机器上进行试验——一台Sun Sparc 2，一台Intel i860和一台IBM RS/6000，N=512时，置换后的循环比原始的循环快两倍[210]。在RS/6000的性能提高接近10倍，这个系统的访存延时与未来的机器特点类似。

9.3 分块

一旦循环交换选择了最好的循环次序，我们就可以对某些循环尝试用分块（也称为*tiling*）提高性能。考虑上一小节中经过循环交换后的例子:

```
DO J = 1, M
  DO I = 1, N
    D(I) = D(I) + B(I, J)
  ENDDO
ENDDO
```

如前面指出的，这个循环将导致2NM/b次不命中。通过循环分段和交换（在9.1节中介绍）可以进一步提高这个循环的性能，如下所示:

```
DO I = 1, N, S
  DO J = 1, M
    DO ii = I, MIN(I + S - 1, N)
      D(ii) = D(ii) + B(ii, J)
    ENDDO
  ENDDO
ENDDO
```

476 ~ 477

假设B中的每一列都从一个新的高速缓存行开始。那么，如果选择的S是b的倍数，对B(ii,J)的访问将得到所有的空间重用，不命中次数是NM/b次。假设对D的存储也是从新的高速缓存行开始的。那么如果S选择的足够小，使得在J-循环的不同迭代间，包含有D(ii)的高速缓存行保留在高速缓存之内，对D的访问可以既得到空间局部性，又得到时间局部性，产生总的不命中次数是N/b次。在这些前提下，总的失效次数是

$$\left(1+\frac{1}{M}\right)\frac{NM}{b}$$

当M很大时，此式将非常接近NM/b。

现在考虑如果我们对外层循环进行分段，并且将对分段得到的段进行迭代的循环交换到内层。这相当于对外层循环进行分段和交换:

```
DO J = 1, M, T
  DO I = 1, N
    DO jj = J, MIN(J + T - 1, M)
      D(I) = D(I) + B(I, jj)
    ENDDO
  ENDDO
ENDDO
```

再一次假设B的每一列都从一个新的高速缓存行开始。如果选择的T足够小，使得在I-循环的不同迭代间，包含有B(I,jj)的高速缓存行保留在高速缓存中，则对b的访问可以从空间局部性中得到好处，此时总的不命中次数仍然是NM/b次。然而，在D上的不命中次数是不一样的，因为此时在执行内层的两个循环期间，每个高速缓存行将有一次不命中发生，或者说共有N/b次不命中。由于这些循环共被执行M/T次，所以对于D的总的不命中次数是NM/(bT)，比未分块的版本优点是因子T的存在。所以分块版本的总的不命中次数是

$$\left(1+\frac{1}{T}\right)\frac{NM}{b}$$

既然我们希望M比T大很多，在相同的行对齐的前提下，这个公式得到的代价大于前面的内层循环分段的代价。

9.3.1 非对齐的数据

如果B的列不是从一个新的高速缓存行开始，上述两种情况的分析将发生变化。假设数组的整体内存分配从高速缓存行开始，且多维数组是连续存放的，则每个新的列可能从高速缓
478 存行的任何位置开始。在原始的分块

```
DO I = 1, N, S
   DO J = 1, M
      DO ii = I, MIN(I + S - 1, N)
         D(ii) = D(ii) + B(ii, J)
      ENDDO
   ENDDO
ENDDO
```

中，由于B的行可能从一个高速缓存行的中间开始，行的结束也早于其使用的最后一个高速缓存行的结尾，内层循环在B的连续片段上的访问可能对每一次迭代有一次额外的不命中。所以，访问B导致的额外不命中是其每一列对于外层的I-循环的每次迭代至多一次，或者说总次数是NM/S。所以总的不命中次数不会超过

$$(1+1/M+b/S)\,NM/b$$

在第二种分块

```
DO J = 1, M, T
   DO I = 1, N
      DO jj = J, MIN(J + T - 1, M)
         D(I) = D(I) + B(I, jj)
      ENDDO
   ENDDO
ENDDO
```

中，舍弃对列对齐的假设调整导致I-循环的每次迭代对B的访问至多增加一次额外的不命中，总共有M/T次新增的不命中。所以这个版本的程序的总共不命中次数是

$$\left(1+\frac{1}{T}+\frac{b}{TN}\right)\frac{NM}{b}$$

由于T被选择为b的倍数，且M比T大得多，括号的中第三项远小于第二项，所以其结果非常接近于

$$\left(1+\frac{1}{\mathrm{T}}\right)\frac{\mathrm{NM}}{b}$$

因此，如果第一种情形中的块大小S与第二种情形中的块大小T相近，第一种情形不再显现（相对于第二种情形）更好的性能。然而，对代码的分析显示：在第一种方法中，高速缓存必须存放D的S/b个不同的块，而在第二种方法中，同样大小的高速缓存必须能够存放B的T个不同的块。这意味着S比T大b倍，所以第一种方案的代价改进后为 479

$$\left(1+\frac{1}{\mathrm{M}}+\frac{1}{\mathrm{T}}\right)\frac{\mathrm{NM}}{b}$$

如果N和M的大小相当，则这种方案比第二种方案的性能要略差一些。换句话说，数据对齐的考虑使得我们转为优先选择第二种方案。

这些考虑说明当对循环同时进行分块时，选择最优的循环次序的复杂性——对于循环交换方案最好的次序未必是分块方法希望的最好的循环次序。从这个例子得到的特殊经验是如果多维数组的列不与高速缓存的边界对齐，可能付出的代价是分块后固定的最内层访问，而不是对邻接块的访问，因为这种方法将额外的不命中移到了内层循环之外。

在后面几小节中，我们将对嵌套循环分块的几种策略进行探讨。但是，我们将首先回答两个问题：

（1）何时分块是合法的？

（2）何时分块是合适的？

9.3.2 分块的合法性

分块的基本算法称为*循环分段和交换*。基本上，该算法有如下步骤：将一个给定循环分段，得到两个循环，一个循环在分段内进行连续的迭代，而一个外层循环进行逐段迭代，然后将对段进行迭代的循环交换到其他的包围着两个循环的循环之外。基本算法在图9-3中给出。

显然，循环分段这一步总是合法的，因为它没有对执行次序进行任何的改变。但是，对逐段循环实施交换不是必然合法的。考虑在5.2.1小节中给出的循环交换的合法性。基本上，如果对于每一个被循环$L_o...L_{k+1}$中任何循环携带的依赖的方向向量中第k个分量是一个“=”或“<”，使得循环移位后的方向矩阵依然合法，则循环交换是合法的。

事实上，合法性测试过于保守，因为当交换非法的时候，循环分段和交换仍然可以在特定的环境中实施，如图9-3的例子所示。如果分段的大小小于或等于可能阻碍进行循环交换的依赖距离的阈值，此时虽然单纯的循环交换是不合法的，但是循环分段和交换是合法的。在图9-3中，有一个被外层的J-循环携带的阻碍循环交换的依赖，其依赖距离为3，如果分段的大小是3或更小，则这个依赖不会阻碍循环分段和交换的实施。然而，基于依赖距离往往比有用的分段大小小得多的认识，使用保守的测试方法是一种简单而有效的方法。 480

9.3.3 分块的有利性

通常，如果在一个非最内层的循环的不同迭代间有重用，分块是有利的。如果将在9.2节中介绍的重排序方法从次最内层循环开始逐步向外实施，可望找到重用。

在两种情况下，可以得到外层循环的重用：

（1）循环携带一个有小的阈值的任何类型的依赖，包括输入依赖，循环携带依赖，或者

（2）循环的索引变量（拥有小的跨距）出现在一个多维数组的连续维中，而不出现在其他的维中。

```
procedure StripMineAndInterchange(L, m, k, o, S)
  // L={L1, L2, ..., Lm}是要变换的循环嵌套
  // Lk是要分段的循环
  // Lo是外层循环，在循环交换后，这个循环刚好在被分段的循环之内
  // S是一个记录分段大小的变量；它的值必须是正值
  令Lk的循环头是
    DO I=L, N, D;
  将这个循环分为两个循环，一个是对分段进行迭代的循环:
      DO I=L, N, S * D
    和一个在分段内进行迭代的循环:
        DO i=I, MIN (I+S * D-D, N), D
    围绕循环体;
  将对分段进行迭代的循环交换到刚好在Lo外的位置;
end StripMineAndInterchange
```

图9-2 循环分段和交换的算法

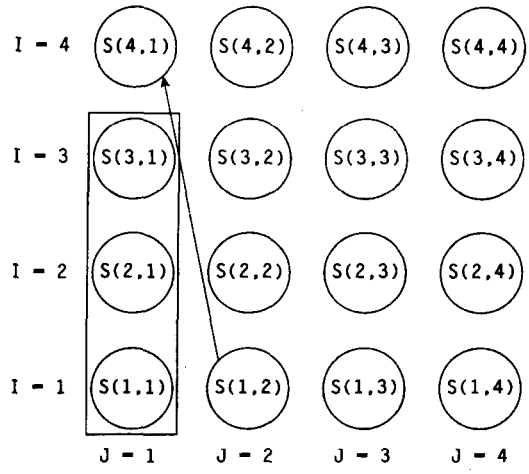


图9-3 循环分段和交换的合法性

在第一种情形中，对于每个引用，如果它是至少一个循环携带依赖的目标，则其不命中次数的减少与循环的分段个数N成比例。在第二种情形中，如果引用不是第一种情形涵盖的任何依赖的目标，不命中次数的减少与高速缓存行的大小b成比例。

由于有分段循环引入的不命中，循环分段和交换也是伴随有代价的。对于每个引用组，存在有两种可能的代价:

（1）对于每个引用，如果它是一个被内层循环携带的依赖的目标，可能会导致M/S次不命中，其中M是内层循环的迭代次数，而S是分段的大小。（我们假设S是b的一个整倍数。）

（2）对于每个引用，如果它不是循环携带依赖的目标，但是内层循环的索引变量出现在其连续维（的下标表达式）中，由于分段的界限与高速缓存行不对齐的可能性，所以可能引入M/S次不命中。（如果分段的界限与高速缓存行是对齐的，不会引入额外的不命中。）

所以，引入的总的不命中次数大约是R_0M/S，其中R_0是内层循环中无依赖引用的个数。总

的节省的不命中次数是

$$R_1N + R_2N\left(1+\frac{1}{b}\right) = (R_1 + R_2)N - \frac{R_2N}{b}$$

其中R_1是被外层循环携带的依赖的目标引用的总数，而R_2是访问具有存储邻接性质且不是任何循环携带依赖的目标的引用的总数。如果N和M的大小是可比的，则当循环分段和交换无法进行有效性测试时，在外层循环导致的减少的引用总数比其在最内层循环减少的携带引用次数要小，减小的倍数与分段的大小可比。所以，很多研究者对于任何可以实施循环分段的循环进行分段，以提高重用性[276]。

481
~
482

9.3.4 一个简单的分块算法

上一小节的结果使我们有了一个简单的分块算法，如图9-4所示。本质上，这个算法只是不断进行循环分段和交换，直至再也没有任何一个原始循环具有重用。

```
procedure BlockLoops(L, m)
  // L={L1, L2, ..., Lm}是将对其实施变换的循环嵌套，它已利用9.2节中的算法排列成最好的存储顺序;
  for i:=m+1 to 1 by -1 do begin
    if在循环Li中有重用then begin
      令o≥i-1是Li-1可以移动到其外层的最外层循环的索引变量;
      if o≥i then begin
        令Si-1是一个新的变量;
        StripMineAndInterchange(L, m, i-1, o, Si-1);
      end
    end
  end
end BlockLoops
```

图9-4 简单的分块

让我们看一看对于实施循环交换后的矩阵乘法，上述算法是如何实施的:

```
DO J = 1, N
   DO K = 1, N
      DO I = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

在选择最内层循环后，由于对C(I,J)的重用和对B(K,J)邻接引用，所以次最内层循环(K-循环)显然也带有重用。进而，我们对I-循环进行分段，并将段循环一直移动到最外层:

```
DO I = 1, N, S
   DO J = 1, N
      DO K = 1, N
         DO ii = I, MIN(I + S - 1, N)
            C(ii, J) = C(ii, J) + A(ii, K) * B(K, J)
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

由于A(ii,K)的自输入依赖，J-循环也具有少量的重用。这种重用可以通过对K-循环实施循环分段和交换来发掘:

```
DO K = 1, N, T
   DO I = 1, N, S
      DO J = 1, N
         DO kk = K, MIN(K + T - 1, N)
            DO ii = I, MIN(I + S - 1, N)
               C(ii, J) = C(ii, J) + A(ii, kk) * B(kk, J)
            ENDDO
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

这个循环的高速缓存不命中的总次数可以用9.2节中给出的各种代价公式来估计。在内层循环中，我们知道对于数组C和A各自共有S/b次不命中，而对于数组B则有1次不命中。假设S足够小，kk-循环使C数组的不命中次数乘以1，A数组的不命中次数乘以T，而B数组的不命中次数乘以T/b。J-循环使数组C和B的不命中次数乘以N，而数组A的不命中次数乘以1。外面两层循环使不命中次数乘以N^2/ST。这三个引用组的总的不命中次数如表9-2的总结。

表9-2 分块矩阵乘法存储分析

循环	C(I,J)	A(I,K)	B(K,J)	总次数
ii	S/b	S/b	1	$2S/b+1$
kk	S/b	ST/b	T/b	$S/b+ST/b+T/b$
J	NS/b	ST/b	NT/b	$NS/b+ST/b+NT/b$
I	N^2/b	NT/b	$N^2T/(Sb)$	$N^2/b+NT/b+N^2T/(Sb)$
K	$N^3/(Tb)$	N^2/b	$N^3/(Sb)$	$N^3/(Tb)+N^3/(Sb)+N^2/b$

如果S=T，本方法相对于最好的循环交换的版本（其导致的不命中次数大约是$2N^3/b+N^2$）有一个明显的改善，不命中次数降低的因子是S。而且，这接近于能保存一个$S \times S$块的高速缓存最可能的复杂性。

483 ~ 484

9.3.5 带倾斜的分块

有些循环由于不能进行交换而不能分块。下面是一个简单的例子:

```
DO I = 1, N
   DO J = 1, M
      A(J + 1) = (A(J) + A(J + 1)) / 2
   ENDDO
ENDDO
```

这个循环的依赖方向矩阵是

$$\begin{bmatrix} = & = \\ = & < \\ < & = \\ < & < \\ < & > \end{bmatrix}$$

这个矩阵产生图9-5中显示的依赖模式。

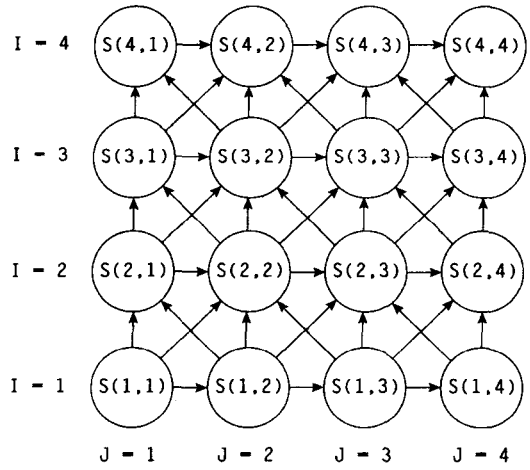


图9-5 例子中的依赖模式

循环置换过程对这些循环无法产生作用，由于循环不能进行交换，所以这些循环也不能进行分块。如果我们希望增加这个循环嵌套中的重用的数量，内层循环可以对外层循环倾斜， 485

```
DO I = 1, N
   DO j = I, M + I - 1
      A(j - I + 2) = (A(j - I + 1) + A(j - I + 2)) / 2
   ENDDO
ENDDO
```

这使得循环可以进行交换。因此，循环分段和交换也就可以实施了。下面是进行了循环分段步骤后得到的循环:

```
DO I = 1, N
   DO j = I, M + I - 1, S
      DO jj = j, MIN(j + S - 1, M + I - 1)
         A(jj - I + 2) = (A(jj - I + 1) + A(jj - I + 2)) / 2
      ENDDO
   ENDDO
ENDDO
```

将对分段循环向外交换，产生

```
DO j = 1, M + N - 1, S
   DO I = MAX(1, j - M + 1), MIN(j, N)
      DO jj = j, MIN(j + S - 1, M + I - 1)
         A(jj - I + 2) = (A(jj - I + 1) + A(jj - I + 2)) / 2
      ENDDO
   ENDDO
ENDDO
```

这种形式的循环可以利用外层循环的重用。如果原始的循环嵌套产生了MN/b次不命中，改进后的版本产生的不命中次数接近于

$$(S/b+1)\,(M+N)/S=(M+N)\,(1/b+1/S)$$

这个结果达到一个数量级的改善。分块的效果可以通过图9-6显示的修改后的依赖图看到。

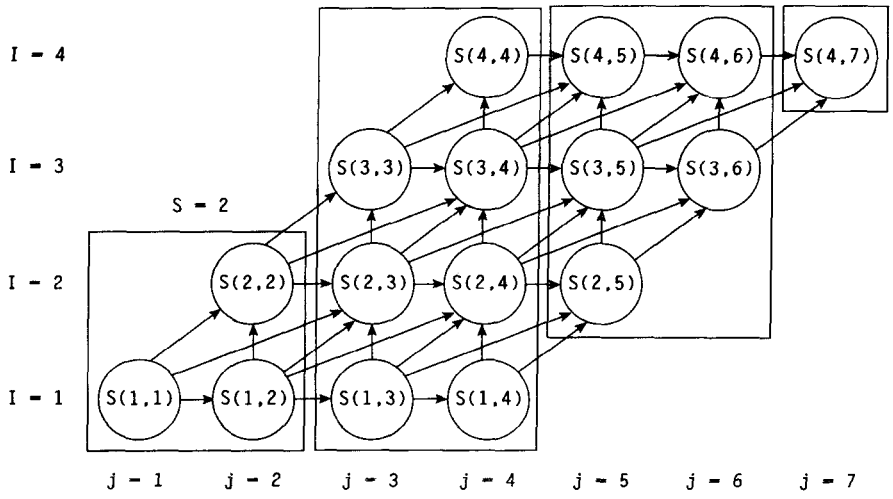


图9-6 倾斜和分块后的依赖模式

在实际使用中，虽然倾斜变换的应用可以获得很好的效果，但是为了分块的目的而应用倾斜变换的机会是很少见的[275]。我们建议在不可能用分块方法获得其他维的重用的情况下使用倾斜变换。这是图9-7的算法使用的方法。

```
procedure BlockLoopsWithSkewing(L, m)
  // L={L1, L2, ..., Lm}是将对其实施变换的循环嵌套，它已利用9.2节中的算法排列成最好的存储顺序;
  for i:=m+1 to 1 by -1 do begin
    if在循环Li中有重用then begin
      令o>i-1是Li-1可以移动到最外层循环的循环索引变量;
      if o>i then begin
        令Si-1是一个新的变量;
        StripMineAndInterchange(L, m, i-1, o, Si-1);
      else if i=m+1 then begin
        对于Li倾斜Li-1;
        令Si-1是一个新的变量;
        StripMineAndInterchange(L, m, i-1, o, Si-1);
      end
    end
  end
end BlockLoopsWithSkewing
```

图9-7 带倾斜的分块

9.3.6 循环合并和对齐

486
~
487

在8.6节中详细介绍过的循环合并，在与面向高速缓存的分块技术协同使用时，对于提高存储的层次结构管理性能也是非常有用的。在许多例子中，如果两个循环中任何一个都可以利用分块来提高层次存储结构的性能，这两个循环可以先合并，然后再分块，在某些情形中，

这种方法可以使性能倍增。

如8.6.2小节中的说明，利用对齐可以增加循环合并的机会。我们通过一个简单的例子解释这一点，这个例子首先对一个两维数组初始化，后面接着做内部松弛。

```
      DO I = 1, N
         DO J = 1, N
S1          A(J, I) = AINIT(J, I)
         ENDDO
      ENDDO
      DO I = 2, N - 1
         DO J = 2, N - 1
S2          A(J, I) = (A(J + 1, I + 1) + A(J - 1, I - 1)) * 0.5
         ENDDO
      ENDDO
```

依照8.6.2小节中的策略，我们能够将这两个循环对齐，使得语句S_2中的第二个引用与语句S_1中的赋值对齐:

```
      DO I =0, N - 1
         DO J = 0, N - 1
S1          A(J + 1, I + 1) = AINIT(J + 1, I + 1)
         ENDDO
      ENDDO
      DO I = 2, N - 1
         DO J = 2, N - 1
S2          A(J, I) = (A(J + 1, I + 1) + A(J - 1, I - 1)) * 0.5
         ENDDO
      ENDDO
```

现在，我们可以合法地将这两个循环嵌套合并，得到

```
      DO I = 0, 1
         DO J = 0, N - 1
            A(J + 1, I + 1) = AINIT(J + 1, I + 1)
         ENDDO
      ENDDO
      DO I = 2, N - 1
         A(1, I + 1) = AINIT(1, I + 1)
         A(2, I + 1) = AINIT(2, I + 1)
         DO J = 2, N - 1
S1          A(J + 1, I + 1) = AINIT(J + 1, I + 1)
S2          A(J, I) = (A(J + 1, I + 1) + A(J - 1, I - 1)) * 0.5
         ENDDO
      ENDDO
```

488

当内层循环分块后，A的每个高速缓存行都只导致一次不命中，而原来的循环中每次迭代发生两次不命中。

9.3.7 结合其他变换的分块

分块可以和本书中讨论的其他变换技术结合，从而产生非常有效的优化。此处我们将讨论其中的三种。

分块与寄存器使用的增强

此处描述的分块变换可以相当容易地与第8章介绍的标量替换与展开和压紧技术相结合。由于寄存器的管理只是利用了时间局部性，相对于也包含非时间的空间局部性的高速缓存管理而言，寄存器的管理是很特殊的。此外，高速缓存管理比寄存器管理要考虑更多的依赖；例如，（在大多数高速缓存的设计中）一个反依赖会引起高速缓存的重用，但不会引起寄存器的重用。所以，寄存器管理可以对完成了高速缓存管理而留下的循环实施。不过，由于高速缓存管理试图在时间和空间局部性两方面都进行优化，对寄存器管理有意义的循环应该是在高速缓存管理的变换完成后靠近最内层循环位置的循环。

存储层次结构的多个级别

除寄存器之外，在现代计算系统中通常可以发现多级的高速缓存。此外，不同的存储有着差异非常大的访问时间。在下一小节将讨论到的并行计算机系统是一种典型的“非一致”存储访问时间的结构。在一个典型的并行计算机系统中，对异地存储的访问时间可以比对本地存储的访问时间长数倍。例如，在一个有64个结点、128个处理器的SGI Origin 2000上，对异地存储的访问平均是对本地存储的访问时间的三倍[251]。在这种情况下，针对多于一个级别的高速缓存分块技术可能有作用。然而，除非在最快速的高速缓存级别不比下一级别的高速缓存快得多，或者最快的级别容量过小以至于不能起到作用，分块的实质是首先以最快速的高速缓存为目标进行变换，然后再处理其他的高速缓存层次。在这种情形中，可能会对由循环分段和交换变换得到的对分段的循环再进行分段变换，以得到对大容量的二级高速缓存的进一步的重用。另一种方案是，如果第一级高速缓存容量太小，以至于在每一维中都无法充分得到重用，则更大容量的高速缓存可以用来获得某种程度上的未发掘的重用性。

将二级的存储看成是内存层次结构的一部分也是可能的。已有的经验显示，利用类似于
489 优化高速缓存中描述的变换技术，一个典型的核外计算能够提高性能200倍或更多[173]。

分块与并行化

当高速缓存管理和并行化相结合时，会引起两个关键问题。

（1）如果被并行的维是连续访存的维，则增加并行性可能会妨碍存储层次结构的性能。

（2）如果每个处理器使用的数据不能很好地和高速缓存行的边界对齐，即使处理器实际上并没有访问同一个数据，但两个或多个处理器可能争用包含这些处理器需要使用的数据的同一个高速缓存行。这种现象称为*假共享*，它是共享存储的并行机器的一个重大问题。

既然最经常提到的未能充分挖掘并行计算机的性能的原因是低下的单计算结点的性能，因此在生成面向并行计算机的代码时，把增强数据局部性作为非常优先的工作是合乎情理的。这意味着，如果程序中有多个维可以并行，则在并行化时应避免访问跨距为1的那一维。

如果程序语言的语义允许，假共享也可以通过数据的非连续分配而减少。例如在多维数组中，如果列被不同的处理器访问，则有助于保证每一列从一个新的高速缓存行开始。这种处理违反了Fortran语言中的顺序和存储结合的限制，但是普遍认为，这些语言特性正在被语言所抛弃。

如果必须选择跨距为1的维进行并行化，则计算应该根据高速缓存行的边界来进行划分，以避免假共享的发生。HPF形式的CYCLIC(k) 分配应该只有在满足下述条件时才被使用：k声明的字数是高速缓存行大小的整数倍。

在未来，并行化工作可能会在更深程度上处理存储层次结构，这会使得本章中讨论的各

种变换技术更加重要。

9.3.8 有效性

通常，在可能的情况下利用分块技术提高高速缓存的性能可以取得显著的效果。Porterfield指出单单一个循环分段和交换变换，就可以把矩阵乘法中的高速缓存不命中次数从932 576次减少到40 000次，几乎将程序中的不命中完全消除[227]。Wolf对于同样的程序报告了类似的性能改善。

在实际程序中，问题在于变换的适用性。Porterfield和Wolf都指出对于分块变换技术的一些简单的障碍因素常常可以使他们的系统失效，导致许多可以被分块的程序根本得不到改善。 490
好消息是对于那些这种变换可以起作用的应用程序，它确实可以得到非常好的效果。例如，对于8.3.8小节中讨论的NASA核心程序Gmtry，Wolf的分块算法可以将性能提高3倍，而对于Vpenta程序，该算法得到了50%的性能提高。

Porterfield也指出即便不实施分块，结合对齐技术的循环合并也是非常有力的。对于一个小波分析应用的程序WANAL1，将循环交换、对齐和合并的综合使用，可以消除近一半的高速缓存不命中。最近，Ding报告在有带宽限制的机器上，通过使用循环合并，性能获得急剧的提高[103]。

这些成功的例子表明循环分块和合并技术是提高高速缓存性能的可行之路。但是，编译器必须有一个完善的程序变换库，以有效地使用这些变换。

9.4 复杂循环嵌套中的高速缓存管理

到目前为止，我们关注的是对矩形循环嵌套的分块。不幸的是，现实中处理的循环嵌套不是都以这种良好的规则形式出现的。在本节中，我们将集中讨论梯形循环嵌套和通常在线性代数中用到的特殊循环嵌套带来的问题。

9.4.1 三角形的高速缓存分块

可以用前面几节中介绍的通用过程对三角形循环进行循环分段，以提高高速缓存的性能。例如，如果我们希望对8.2.2节中例子的外层循环以因子K进行分段，此处的K整除外层循环的迭代次数（这种情况在前置循环的使用中也会出现）。

```
DO I = 2, N
   DO J = 1, I - 1
      A(I, J) = A(I, I) + A(J, J)
   ENDDO
ENDDO
```

在循环分段后，我们得到

```
DO I = 2, N, K
   DO ii = I, I + K - 1
      DO J = 1, ii - 1
         A(ii, J) = A(ii, ii) + A(J, J)
      ENDDO
   ENDDO
ENDDO
```

491

如果我们现在将中间的循环交换到最内层的位置（利用三角形循环交换方法），可以得到

```
DO I = 2, N, K
   DO J = I, I + K - 1
      DO ii = MAX(J + 1, I), I + K - 1
         A(ii, J) = A(ii, ii) + A(J, J)
      ENDDO
   ENDDO
ENDDO
```

假设精心选择K，这种形式应该有好的高速缓存性能。

9.4.2 特殊用途的变换

在线性代数中遇到的很多类型的程序是相当特殊的。然而，通过研究这些程序可以学到很多。我们从对LU分解的分析开始，LU分解是解线性方程组最常用的算法。

大多数版本的LU分解使用的是某种形式的选主元法以保证稳定性。然而，选主元法带来了特殊的问题，我们在后面会讨论。目前，我们首先以一个不使用选主元法的算法开始我们的讨论。

在不使用选主元法的LU分解中，中心循环嵌套如下：

```
      DO K = 1, N - 1
       ! 主元总是A(K, K)
        DO I = K + 1, N
S1         A(I, K) = A(I, K) / A(K, K)
        ENDDO
        DO J = K + 1, N
           DO I = K + 1, N
S2            A(I, J) = A(I, J) - A(I, K) * A(K, J)
           ENDDO
        ENDDO
      ENDDO
```

这段代码的问题是假设N足够大，这个循环嵌套的第二部分不能得到在赋值号右端的A(I,K)和A(K,J)的重用。为了得到重用，你应该对外层循环K进行分段，并将段内的循环向内交换到第二个循环嵌套中J-循环和I-循环的内层。实施循环分段后，我们得到

```
      DO K = 1, N - 1, S
492      DO kk = K, K + S - 1
      !     主元总是A(kk, kk)
           DO I = kk + 1, N
S1            A(I, kk) = A(I, kk) / A(kk, kk)
           ENDDO
           DO J = kk + 1, N
              DO I = kk + 1, N
S2               A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
              ENDDO
           ENDDO
        ENDDO
      ENDDO
```

为了将kk循环向内交换，我们必须首先将它分布在两个循环嵌套上。但是，由于存在包含语句S_1和S_2的依赖环，这是不能允许的。由于A(I,kk)的两次出现，存在一个循环无关的依赖看起来是显而易见的。不明显的是，由于J>kk，语句S_2中将A(I,J)存储到的位置在kk循环

后面的迭代中要执行读入。

这表明设想的分块似乎是不可能的。但是，如果我们注意到K循环的每次迭代中，语句S_2生成

```
A(K+1:N, K+1:N)
```

而语句S_1使用

```
A(K+1:N, K:K+S-1)
```

这意味着，在使用S_2的输出的迭代范围外，不存在依赖环。所以我们将围绕S_2的内层循环嵌套通过索引集分裂分为两个循环，这组循环嵌套中的第二个不产生在S_1中用到的值：

```
      DO K = 1, N - 1, S
         DO kk = K, K + S - 1
      !     主元总是A(kk, kk)
            DO I = kk + 1, N
S1             A(I, kk) = A(I, kk) / A(kk, kk)
            ENDDO
            DO J = kk + 1, K + S - 1
               DO I = kk + 1, N
S2                A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
               ENDDO
            ENDDO
            DO J = K + S, N
               DO I = kk + 1, N
S3                A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
```

493

现在kk循环可以被分布，产生一个循环嵌套包含S_1和S_2，另一个包含S_3：

```
      DO K = 1, N - 1, S
         DO kk = K, K + S - 1
      !     主元总是A(kk, kk)
            DO I = kk + 1, N
S1             A(I, kk) = A(I, kk) / A(kk, kk)
            ENDDO
            DO J = kk + 1, K + S - 1
               DO I = kk + 1, N
S2                A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
               ENDDO
            ENDDO
         ENDDO
         DO kk = K, K + S - 1
            DO J = K + S, N
               DO I = kk + 1, N
S3                A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
```

最后，在最终的循环嵌套中，kk循环可以被移动到I-循环和J-循环的内部：

```
      DO J = K + S, N
         DO I = K + 1, N
            DO kk = K, MIN(I - 1, K + S - 1)
S3             A(I, J) = A(I, J) - A(I, kk) * A(kk, J)
            ENDDO
         ENDDO
      ENDDO
```

内层循环极大地提高了高速缓存的重用性。

根据Carr和Kennedy在MIPS M120上进行的实验，对于含有100个元素的数组，改进后的
494 代码达得到的计时在表9-3中给出。

表9-3 LU分解的计时

原始代码	8.36 sec
变换后的代码	6.40 sec
Sorensen手写的代码	6.69 sec
循环展开和压紧后的代码	3.55 sec
加速比	2.35

我们注意到，与Dan Sorensen开发的作为LAPACK的实现工作的一部分的手工版本相比，自动的方法得到一个不同并稍好一些的版本。另外，对变换的结果使用三角形式的展开和压紧（这种方法显然也可以用于Sorensen版本）得到的总加速比是2.35。

9.5 软件预取

即便程序重构能取得应有的效能，仍然不能消除某些类型的高速缓存不命中：

（1）对首次使用数据不命中

（2）对某种在编译时无法判定的被重用的数据不命中

下面的循环给出上述每种不命中的例子：

```
DO I = 1, N
   A(I) = B(LOC(I))
ENDDO
```

当对A(I)的写操作映射到一个新的高速缓存行时，由于是在这个循环嵌套中第一次使用A(I)，所以会导致不命中。如果LOC(I)中的值有重复，则B(LOC(I))可能具有大量的时间重用性。但是，由于LOC(I)的值在编译时是不知道的，因此在编译时通过重构循环嵌套来发掘这种重用性是困难的。

为了改善这些问题，在20世纪80年代后期和90年代前期的机器设计者在现代处理器中引入了预取指令。一条预取指令的典型实现类似于读入指令，但是不会将一个值放入寄存器。实际上，预取是把包含指令指定的存储器中目标地址的高速缓存行预先读入到高速缓存中。第二点不同是预取操作通常不会导致处理器的停顿——如果目标数据不是在高速缓存中，将
495 包含这些数据的高速缓存块读入到高速缓存中的操作是和其他操作并行执行的[⊖]。因此预取指令提供一种将内存访问和其他的处理器指令重叠执行的机制。

为了使用处理器可用的预取指令，程序员或编译器必须生成预取指令——完成这个任务的处理过程称为软件预取（software prefetching）。如果编译器能够为循环中使用的每个高速缓存行在程序中足够提前的位置插入一条预取指令，所有不命中的延迟都可以被消除（假设高速缓存容量足够大）。由于仅靠程序重构技术对这些循环已经没有作用了，此时预取的价值是

⊖ 理想情况下，预取不会由于越界访问而导致异常的发生。

显然的。

虽然预取有明显的优点，但它也有一些显著的缺点：

（1）预取增加了必须执行的指令条数，每次预取都需要为预取本身增加一条指令，并可能为计算预取的地址而增加数条指令。

（2）预取可能使有用的高速缓存行被过早地替换掉。

（3）预取读入的高速缓存行可能在使用前被替换掉，或读入一些根本不会使用的高速缓存行，不必要地增加存储访问流量。

为了将这三方面缺点的影响降至最低限度，我们必须小心地设计预取的算法，使得（a）预取的次数与实际的需要相近，（b）数据不会过早地被预取，（c）预取很少被不需要预取的引用所调用。

在后面几小节中，我们给出了一个基于Callahan，Kennedy和Porterfield[61]以及Mowry[216]的工作的预取算法。然后，根据这些研究组得到的实验结果，我们将讨论软件预取的有效性。

9.5.1 一个软件预取算法

通常，预取算法将试图对循环中的引用插入预取指令。由于同一个源程序引用点导致了对不同内存位置的访问的特点，对下标变量的引用是预取的重要目标。但是，由于高速缓存块的长度通常超过一个字的长度，预取算法必须小心地只对一个新的高速缓存行的第一次访问生成预取指令，以免产生大量的无用的预取操作。因此，有效的预取算法的关键步骤是

（1）精确确定需要预取的引用，从而将不必要的预取减至最少；

（2）插入预取指令的位置足够提前，使得数据到达既不太迟，也不太早。 496

其中的第一步称为*预取分析*，而第二步称为*预取插入*。在本节的后面将详细讨论这两个阶段。

分析阶段必须精确决定哪一个引用需要预取，并对程序进行变换，使得需要预取的部分与不需要预取的部分区分开来。例如，考虑下面的循环嵌套：

```
DO I = 1, N
   DO J = 2, M
      A(J, I) = A(J, I) + B(J) * A(J - 1, I)
   ENDDO
ENDDO
```

由于对A(J,I)的引用在内存中是顺序访问的，每一个高速缓存行都会产生一次不命中。对B(J)的引用，每一个高速缓存行也会有一次不命中，这个行数要乘上J-循环执行的次数。

这意味着，即便在I-循环的每次迭代中B(J)的值被重用，但是如果循环上界M足够大，上述重用是不能实现的。通过对J-循环实施循环分段和交换的变换，我们可以得到这种重用性，但是段内循环的大小必须仔细选择，从而不会发生9.3节中描述的越出高速缓存。让我们假设实施了这个变换，并得到了下面的代码：

```
DO J = 2, M, S
   JU = MIN(J + S - 1, M)
   DO I = 1, N
      DO jj = J, JU
         A(jj, I) = A(jj, I) + B(jj) * A(jj - 1, I)
      ENDDO
```

```
    ENDDO
  ENDDO
```

一旦完成了这个变换，除了I循环的每一次执行的第一个迭代而外，我们已经消除了访问B(jj)的延迟。

但是，由于A(jj,I)和A(jj-1,I)这个循环中仍然有一些的不命中，这是可以通过预取改善的。分析的目的是确定循环嵌套中的哪些迭代需要对A的一次预取，而哪些迭代需要对B进行预取。下一节将讨论如何对预取进行调度。

如同我们前面指出的，在程序中有两类常见的局部性。当存在对同样的内存位置的重用时称为时间局部性，当两个引用访问相同的高速缓存行时称为空间局部性。在上面的例子中，由于A(jj-1,I)和A(jj,I)在循环的相邻迭代中访问的是相同的位置，所以这两个引用显现时间局部性。但是，在jj-循环的相邻迭代中对A(jj,I)的引用存在空间局部性。如果高速缓存
497 行的长度是L个字，则对A(jj,I)的引用每L次迭代会产生一次不命中。

如果我们采用8.3节中的为了提高寄存器分配的性能的对时间局部性的分析类似的分析方法，我们会发现对A(jj,I)的引用（无论是读或是写）和对A(jj-1,I)的引用构成了一个单独的名字划分，因此形成了一个时间重用组。进一步，A(jj,I)被看成是这个组的生成者。因此，目标是决定在哪些迭代预取A(jj,I)。请注意，对A(jj-1,I)的引用只是在jj循环的第一次迭代需要一次预取。

目前，假设对于I的每个值，A(1,I)位于一个高速缓存行的边界，并且S是高速缓存行长度L的倍数，我们可以看到在jj-循环中满足MOD(jj-J,L)=L-1的迭代或J+L-1，J+2 * L-1，J+3 * L-1等的迭代，需要对A(jj,I)进行预取。

对于B(jj)的引用情况如何呢？如果我们假设B(1)位于一个高速缓存的边界，则在I循环的第一次迭代中，我们在下列迭代中需要对B的预取：在jj-循环的第一次迭代之前，或每一个满足MOD(jj-J,L)=L-1的迭代或J+L-1，J+2*L-1，J+3 * L-1等的迭代。但是，由于选择的S使得所有B(J:JU)的值装入高速缓存，我们可以在jj-循环之前预取它们的全体。这个方式的实现称为预取向量化。

预取需求概括如下：

（1）当I=1且jj=J+k * L-1，$0 \leqslant k \leqslant$(JU-J)/L时，预取B(jj)。换句话说，在I-循环的第一次迭代预取B(J-1:JU:L)。

（2）对于所有的I和jj=J，预取A(jj-1,I)。

（3）对于所有的I和所有的jj=J+k * L-1，$1 \leqslant k \leqslant$(JU-J)/L，预取A(jj,I)。

按照这种方式决定需要预取操作的迭代是预取分析的目标。

一旦决定需要预取的迭代，我们可以划分循环的迭代空间，以确保只有在引用需要的时候才预取。所有对B的预取应该发生在I-循环的第一次迭代，可以将这个迭代从循环中剥离，并在剥离的迭代中插入预取。但是，在这个例子中，只需简单地将对B的预取外提到I-循环之外就可以达到同样的效果。

为了分离对A的预取，我们需要对jj-循环进行分段，分段的长度为L，但是第一个分段的长度为L-1。现在，假设对于I的每个值，A(1,I)位于一个高速缓存行的边界，我们可以看到下面循环的配置可以得到希望的结果：

```
DO J = 2, M, S
   JU = MIN(J + S - 1, M)
```

```
    ! 此处是对B的预取
    DO I = 1, N
       ! 此处是对A(J, I)的预取
       ! 对齐预取的前置循环
       DO jk = J, MIN(J + L - 2, JU)
          A(jk, I) = A(jk, I) + B(jk) * A(jk - 1, I)
       ENDDO
       DO jj = J, JU, L
          jju = MIN(jj + L - 1, JU)
          ! 此处是对A(jj, I)的预取
          DO jk = jj, jju
             A(jk, I) = A(jk, I) + B(jk) * A(jk -1, I)
          ENDDO
       ENDDO
    ENDDO
ENDDO
```

498

假设每个预取被刚好放置在与之相关的引用的前面，预取放置的结果如下：

```
DO J = 2, M, S
   JU = MIN(J + S - 1, M)
   DO jj = J - 1, JU, L
      prefetch(B(jj))
   ENDDO
   DO I = 1, N
      Prefetch(A(J, I))
      ! 对齐预取的前置循环
      DO jk = J, MIN(J + L - 2, JU)
         A(jk, I) = A(jk, I) + B(jk) * A(jk - 1, I)
      ENDDO
      DO jj = J, JU, L
         jju = MIN(jj + L - 1, JU)
         prefetch(A(jj, I))
         DO jk = jj, jju
            A(jk, I) = A(jk, I) + B(jk) * A(jk -1, I)
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

到现在为止，我们已经确定了预取数据被实际访问前可以发出预取指令的最后位置。但是，一个好的预取指令放置算法将把预取指令移动到程序中足够早的点，以确保在使用数据前预取已经完成了。在许多情况下，此种算法需要将预取指令跨越几个迭代进行移动。大多数预取系统将这个任务留给指令调度器（见第10章）实现，所以在此我们对这个问题不作进一步讨论。

499

预取分析

我们的预取算法的分析阶段包括确定哪些迭代将遭遇预取不命中。为了做到这一点，我们将使用在8.3节中介绍的依赖分析策略。请回忆用于确定名字划分和生成器的图分析过程。与那种情况不同的是，在处理高速缓存时写操作不会注销一个名字划分，因为在大多数机器上，高速缓存处理写的方式与处理读的方式非常相似——如果要写入的高速缓存块不在高速

缓存中，产生一个高速缓存不命中。因此，除非有对应于不相容依赖的边，否则图中不会有坏边。

在另一方面，为了用这个方法进行处理，我们必须保证不太可能与重用相关联的边从图中删除，所以假设边的目标点对应于一次不命中，除非由于时间局部性而省去这次不命中。

我们假设在为了提高局部性而对循环嵌套实施分段和交换后，开始预取分析阶段。如同9.2节中描述的那样，这个处理过程必然要将具有最好空间局部性的循环移动到可能的最内层的位置。然后，算法将从最内层到最外层循环进行遍历，以决定由于不可能重用而必须把哪些依赖标记为“无效的”。任何时候，只要在源点和汇点间被代码访问的数据量超过了假定的高速缓存大小，重用就是不可能的（在这个算法中，通常假定高速缓存比其实际的大小要小得多，以补偿映射的影响）。

为了完成这个分析，Porterfield[227]估计了每次迭代使用到的数据量，并且在其后决定溢出迭代，溢出迭代是指可以将其数据同时放置在高速缓存中的那些迭代个数多1的迭代。任何一个依赖，如果其阈值等于或大于溢出迭代的值，则其对于重用的目的是无效的。

一旦找出并删除了所有无效的边，我们确定所有不命中可能发生的点，这些点也是需要进行预取的点。为了实现这个目标，我们必须考虑两种情况：

（1）如果名字划分组的生成者不包含在一个依赖环中，每个迭代都预期有一次不命中，除非在后续迭代中对生成者的访问显现出时间局部性（换句话说，如果对生成引用的访问是内存中连续的单元）。这种情况下，在开始一个新高速缓存行的每次迭代中预期有一次失效。

（2）如果名字划分组的生成者包含在一个由某个循环携带的依赖环中，则仅在携带依赖的循环的开始几次迭代中预期有一次不命中，迭代的次数与携带依赖的距离相关。在这种情
500 况下，可以在携带依赖的循环前插入一个对引用的预取。

一个简单的例子可以阐明这两种情况：

```
DO J = 1, M
   DO I = 1, 32
      A(I + 1, J) = A(I, J) + C(J)
   ENDDO
ENDDO
```

对A(I,J)的引用是一组引用的生成者，其中也包含A(I+1,J)。既然这些引用不是由任何循环携带的依赖环的一部分，它们属于第一种情况，因而对命中一个新高速缓存行的内层循环的每一次迭代会产生一次不命中。在另一方面，对C(J)的引用包含在一个由内层循环携带的输入依赖中，所以它的预取可以放置在内层循环的入口之前。

注意，在第二种情况有一个特别的考虑。如果依赖边由外层循环携带，且这个名字划分的生成者引用由内层循环的索引变量确定，则这个生成者给出的整个引用向量可以被预取。如果在高速缓存中没有足够的空间，在对高速缓存大小进行分析时这个携带依赖边将被标记为“无效的”。作为一个例子，考虑下面的循环嵌套：

```
DO J = 1, M
   DO I = 1, 32
      A(I, J) = A(I, J) + B(I) * C(I, J)
   ENDDO
ENDDO
```

这个循环嵌套有一个包含B(I)的由外层循环携带的输入依赖。如果高速缓存大小分析确

定B(1:32)能够占据整个高速缓存，则包含B(I)的依赖边将被标记为“无效的”，且对B(1:32)的预取放置在J-循环之外。这与9.5.1节的介绍中描述的预取向量化对应。

注意，至少在指令调度之前，本例中对A(I,J)和C(I,J)的预取应放在I-循环之内。

无环名字划分的预取插入

我们开始讨论对于不构成强连通区域的名字划分的预取插入位置。考虑这样一种情况：在循环中有一个单独的名字划分，其中只有一个单独的生成者需要循环中的预取。这有两种情况：

（1）如果在循环中，对生成者的引用在高速缓存中不是顺序重复的（即循环中的引用没有空间重用），则只需在每次引用生成者前插入一条预取指令。 501

（2）如果循环中对生成者的引用具有空间局部性，则要决定在访问生成者引起一次不命中的初始迭代后失效的第一次迭代的索引i_0和高速缓存的两次不命中之间的迭代间距l。请注意，如果数组访问的跨距大于1，则间距l可能比高速缓存行的长度小。也请注意，由于我们已经以初始索引不能是第一次迭代这样一种方式定义初始的索引，因此$i_0 \leqslant l+1$。

a）将这个循环划分为两部分：一个初始子循环，执行从1到i_0-1的迭代，其余部分执行从i_0到结束的迭代。

b）对第二个循环进行分段，使得子循环的长度为l。在每个子循环的前面插入一条对生成者的预取指令。

c）为了避免在初始子循环中的不命中，在初始循环前插入所有需要的预取指令。请注意，如果在名字划分中有一个循环携带的数据依赖，且该依赖的汇点的引用是在比生成者引用更早的高速缓存行中，对含有生成者引用的高速缓存行的预取在第一次迭代中是不够的。

d）通过循环展开消除任何非常短小的循环。此处的“非常短小”可能依赖于特定机器的参数，但是显然，只有一次迭代的循环是非常短小的。

在两种情况下，预取指令的最后位置将由指令调度器决定。

作为这个过程的一个例子，考虑下面的循环：

```
DO I = 1, M
   A(I, J) = A(I, J) + A(I - 1, J)
ENDDO
```

包含对A(I,J)和A(I-1,J)引用的名字划分把对A(I,J)的写作为它的生成者。如果我们假设A(0,J)从一个新的高速缓存行开始，且高速缓存行有4个字的长度，则$i_0=4$且$l=4$。

因此，初始的子循环，或称为前置循环，由3次迭代组成，而其余的迭代可以分段，段长是4。

```
DO I = 1, 3
   A(I, J) = A(I, J) + A(I - 1, J)
ENDDO
DO I = 4, M, 4
   IU = MIN(M, I + 3)
   DO ii = I, IU
      A(ii, J) = A(ii, J) + A(ii - 1, J)
   ENDDO
ENDDO
```

显然，在内层循环的每次迭代前需要对A(I,J)的一次预取，但在前置循环的前面需要预取

502 哪些数据呢？A(0,J)是必须被预取的，而且这个预取操作将把在同一个高速缓存行的A(1,J)带入高速缓存中。使用这个预取指令后，代码变成

```
prefetch(A(0, J))
DO I = 1, 3
   A(I, J) = A(I, J) + A(I - 1, J)
ENDDO
DO I = 4, M, 4
   IU = MIN(M, I + 3)
   prefetch(A(I, J))
   DO ii = I, IU
      A(ii, J) = A(ii, J) + A(ii - 1, J)
   ENDDO
ENDDO
```

请注意，在第一个循环前必须被预取的高速缓存行的集合就是包含前置循环第一次迭代中的所有引用的高速缓存行的集合。通常这可以通过检查来确定。

如何将这个过程进行扩展，使其可以处理拥有不同生成者的多个名字划分呢？扩展是直接的：简单地确定每个名字划分的初始不命中索引，用最小的一个作为实际的i_0，且在分段循环展开版本的循环体中直接插入对于其他的不命中进行预取的指令。

为了举例说明这种扩展，我们给出另一个例子，假设A(0,J)和B(0,J)开始一个高速缓存行：

```
DO I = 1, M
   A(I, J) = A(I - 1, J) + B(I + 2, J)
ENDDO
```

当I=2时，对B(I+2,J)的引用将有一次初始的不命中，而对A(I,J)的初始不命中索引是4。采用小的索引2，在循环展开和插入预取后，我们得到下面的循环：

```
prefetch(A(0, J))
prefetch(B(0, J))
A(1, J) = A(0, J) + B(3, J)
DO I = 2, M, 4
   prefetch(B(I + 2, J))
   A(I, J) = A(I - 1, J) + B(I + 2, J)
   A(I + 1, J) = A(I, J) + B(I + 3, J)
   A(I + 2, J) = A(I + 1, J) + B(I + 4, J)
   prefetch(A(I + 3, J))
   A(I + 3, J) = A(I + 2, J) + B(I + 5, J)
ENDDO
```

503

在MOD(M-1,4)=0的前提下，这段代码是正确的。如果这个条件不满足，则对I的循环将有一个最多有3次迭代的后置循环，后置循环的生成是简单的。

有环名字划分的预取插入

对于有环的名字划分，预取指令正好插入在携带依赖环的循环前面。在最内层循环的情形下，这一点是相当简单的，因为生成者的引用（这种情形下，是循环携带依赖的目标）只是在预取指令中重复。

但是，在携带依赖的循环是一个外层循环的情形，预取是可以被向量化的。下面的过程实现这个功能：

（1）首先是将预取的循环嵌套放置在携带一个有环名字划分的后向依赖的循环之外。预取循环应该包括含有生成者引用的和使用相同迭代范围的循环的循环头副本。

（2）对循环嵌套进行重新排列，使得在高速缓存行上进行连续迭代的循环是最内层循环。

（3）将最内层循环分成两个循环——一个前置循环和一个主循环。前置循环到达最内层循环的第一次迭代，包含开始一个新高速缓存行的一个生成者引用（前置循环可以为空）；主循环从含有新高速缓存引用的迭代开始。用对第一个生成者引用的一个预取替代前置循环。令主循环的跨距为新的高速缓存引用的间距。

为了举例说明上述过程，我们给出下面的例子循环:

```
DO J = 1, M
   DO I = 2, 33
      A(I, J) = A(I, J) * B(I)
   ENDDO
ENDDO
```

对A(I,J)的预取是很简单的，可以只在例子的循环体中给出。由于包含B(I)的输入依赖被J-循环携带，对B的预取将放在这个循环之外。假设B(1)和A(1,J)与高速缓存行的边界对齐，且高速缓存行的长度是4，第一个预取的将是B(2)，其后的预取是B(5)，B(9)，等等。

```
prefetch(B(2))
DO I = 5, 33, 4
   prefetch(B(I))
ENDDO
DO J = 1, M
   prefetch(A(2, J))
   DO I = 2, 4
      A(I, J) = A(I, J) * B(I)
   ENDDO
   DO I = 5, 33, 4
      prefetch(A(I, J))
      A(I, J) = A(I, J) * B(I)
      A(I + 1, J) = A(I + 1, J) * B(I + 1)
      A(I + 2, J) = A(I + 2, J) * B(I + 2)
      A(I + 3, J) = A(I + 3, J) * B(I + 3)
   ENDDO
   prefetch(A(33, J))
   A(33, J) = A(33, J) * B(33)
ENDDO
```

504

请注意，在外层循环前安排对B的预取，是为了对于在计算循环的循环体中未出现的引用不发射预取指令，而在循环体中所有引用的高速缓存行都被预取。这项处理可以通过在算法中引用循环分裂策略实现。

不规则访问的预取

不规则访问带来了特殊的问题，因为不规则访问通常是由一个数组访问构成的，在数组中的一个下标位置出现下标变量:

```
A(IX(I), J)
```

在这些情形中，我们假设对于外面的数组不存在空间局部性，所以我们预取每一个引用。但

是，对于索引数组，可能存在空间局部性，所以它被作为一个通常的预取来处理。可以使用标准的预取算法。

考虑下面的例子:

```
DO J = 1, M
   DO I = 2, 33
      A(I, J) = A(I, J) * B(IX(I), J)
   ENDDO
ENDDO
```

其中对A的预取与前面的例子类似，但是在每一次迭代中都要预取B。对IX的预取通常会出现
505 在紧邻B的预取的前面，本例中的这个预取可以移到J-循环的外面，得到下面的代码:

```
prefetch(IX(2))
DO I = 5, 33, 4
   prefetch(IX(I))
ENDDO
DO J = 1, M
   prefetch(A(2, J))
   DO I = 2, 4
      prefetch(B(IX(I), J))
      A(I, J) = A(I, J) * B(IX(I), J)
   ENDDO
   DO I = 5, 32, 4
      prefetch(A(I, J))
      prefetch(B(IX(I), J))
      A(I, J) = A(I, J) * B(IX(I), J)
      prefetch(B(IX(I + 1), J))
      A(I + 1, J) = A(I + 1, J) * B(IX(I + 1), J)
      prefetch(B(IX(I + 2), J))
      A(I + 2, J) = A(I + 2, J) * B(IX(I + 2), J)
      prefetch(B(IX(I + 3), J))
      A(I + 3, J) = A(I + 3, J) * B(IX(I + 3), J)
   ENDDO
   prefetch(A(33, J))
   prefetch(B(IX(33), J))
   A(33, J) = A(33, J) * B(IX(33), J)
ENDDO
```

对间接引用预取的一个问题是，如果索引数组在循环中被修改，则预取的地址在预取执行时可能是无效的，这可能导致对一个非法的地址进行预取[216]。由于在大多数机器上，预取不会导致存储异常，对非法地址的预取不会产生不安全的代码。然而，如果我们试图通过两级间接地址预取，我们可能是从一个非法的地址进行读操作。为了避免这种问题，我们限于只对一层间接地址进行预取。

9.5.2 软件预取的有效性

针对一个与上述描述的方法接近的选择预取的方法，Mowry通过多种核心程序检验了它的有效性，结果显示与不进行预取的同样程序相比较，该方法得到从1.05到2.08之间的实际加速比[216]。这些结果重现在图9-8中。这些结果是在一个100Mhz的，拥有8K字节容量的一级高速缓存和256K字节容量的二级高速缓存的MIPS R4000上模拟得到。一级高速缓存不命

中对访问二级高速缓存命中的损失是12个周期，所有访问不命中对存储器访问总的损失是75个周期。 506

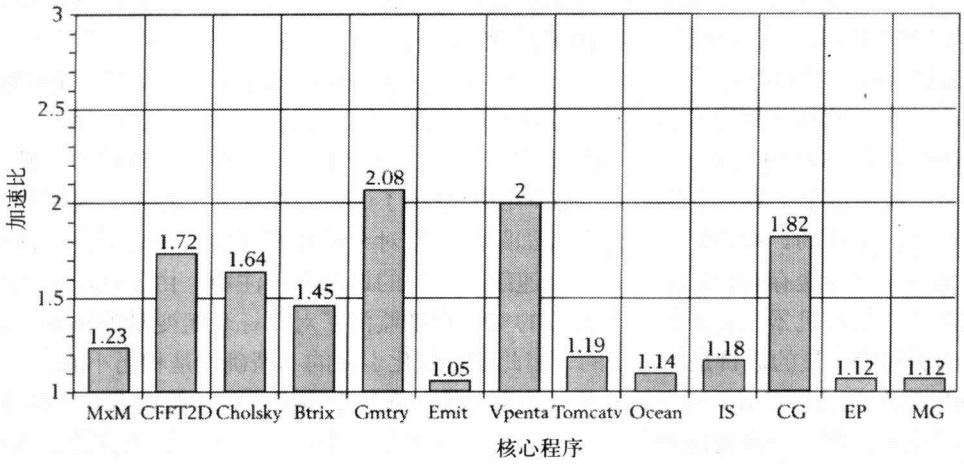


图9-8 软件预取的加速比

Mowry的加速比包括执行预取操作需要的额外指令的代价。在某些情形下，需要这些额外的指令重新计算预取的地址，在某些例子中，这些指令执行的开销可以高达15%。但是，由于减少内存访问导致的处理器停顿的收获对于抵消这些开销是绰绰有余的。

这些结果显示：对于具有现代存储器体系结构的机器，预取是一种重要的优化措施。

9.6 小结

在讨论高速缓存存储时，两种不同的重用是重要的。当两个不同的内存操作访问同一个数据元素时出现时间重用。在第8章中研究过这类重用。当两个不同的访问对应的内存单元接近到足够一起占据同一高速缓存块时，出现的是空间重用，这种重用是高速缓存所特有的。在本章中，为了增加时间和空间两方面的重用性，我们考虑了两种策略：

- 循环交换，用于提高内层循环中对长的高速缓存行的使用效率，既可以获得内层循环的小跨距访问（空间重用性），又可以减少对相同内存单元访问的距离（时间重用性）。 507
- 高速缓存分块，采用与第8章介绍的循环展开和压紧相类似的方式变换程序，发掘外层循环中的重用性。

对于包含有控制流和有梯形迭代范围的循环，这些技术已经显示出其可用性。这些技术的使用效果可以通过使用前面研究的变换技术得到增强，如考虑对齐的循环合并。

另外，本章介绍了软件预取，这是一种变换技术，通过预先读入由特定预取指令指明的高速缓存块来隐藏对主存访问的延迟的。

9.7 实例研究

为了支持其学位论文的研究，Porterfield对Rice大学的PFC系统作了扩充，使其可以实现本章中描述的循环分段和交换。另外，他实现了一个循环合并的基本形式。与本章描述的对齐方法不同，Porterfield的实现方法是当循环对齐使得直接的循环合并不再可能时，寻找应用另外一种称为循环剥离和压紧变换的机会。循环剥离和压紧变换类似于如下形式的伴有对齐的循环合并：为了使循环对齐可以实施，在实施循环合并之前将一个循环的几次迭代剥离。但是，

Porterfield并没有用一个通用的循环合并算法来驱动他的搜索过程，所以他只能发现很少的机会应用他的方法。不过，循环剥离和压紧对于提高某些重要程序性能是关键性的；在9.3.8小节中指出，对WANL1应用这种变换可以使其性能提高四分之一。对于一个主要的流体动力学基准测试程序SIMPLE，可应用循环剥离和压紧使循环中的不命中次数消除一半；但是，这些循环只代表计算中的读操作的一小部分，所以命中率整体的提高只是1%——从77%提高到78%。Porterfield在其研究中也描述了循环倾斜与循环分块的结合，但是没有发现其应用程序。

Porterfield在HPF中集成了一个高速缓存模型，由此产生一个事件驱动的模拟工具，称为PFC-Sim。他用这个工具对变换前后的代码性能进行了评估。在这个工具中，模拟器是和程序的执行并行工作的，减少对用于保存详细跟踪信息对大量数据存储空间的需要。这要求对每一个访问了高速缓存存储的语句用对高速缓存模型的调用进行注释。由于这项工作是在一个计算能力有限的机器上完成的，大部分的实验只是监测了对下标数组变量的访问。由于任何给定的例程中标量的数目是非常少的，因此这种简化引入的不精确性是相当小的。

Porterfield也将PFC用于研究软件预取的有效性。在这个研究中，他假设有一个高速缓存，
508 每一个浮点量占据一个高速缓存行。这个假设允许采用一个比本章中介绍的算法更简单的算法，即试图用长的高速缓存行减少不必要的预取次数。不过，这个预取算法基本上与Mowry，Lam，和Gupta使用的方法[217，216]相同，显现了预取的预期结果，特别是对在非规则计算中发现的间接引用形式。Porterfield的研究工作是已知最早的对软件预取的研究。

PFC的后继者ParaScope系统包括了为提高重用性而对高速缓存分块和实施循环交换的工具。这些工具被Kennedy与McKinley[175]及McKinley，Carr和Tseng[210]用来试验为了提高高速缓存的利用而将循环交换和分块结合使用的有效性。本章中介绍的策略就是从这项工作中得到的。

在最近，Chen Ding扩展了D系统（ParaScope的一个新的版本），实现激进的多层循环合并，从而减少应用程序的存储带宽的消耗[103]。这个系统将循环对齐纳入了一个全局算法，这个算法类似于第8章中介绍过并在本章中使用的算法。

Ardent Titan有一个高速缓存，但是没有将其使用于浮点量，因为所有的浮点操作都是由向量部件完成的，向量部件的读/写操作是绕过高速缓存的。由于这个原因，Titan编译器实现了第8章介绍的提高寄存器重用性的操作，而不包括高速缓存管理的策略。

9.8 历史评述与参考文献

针对高速缓存的分块技术实际已知有多年了。与Kuck一起工作的Abu-Sufah给出了第一个公开发表的基于依赖的程序变换处理方法[1]。在Abu-Sufah工作基础上，一些研究人员给出了许多由编译器实现高速缓存管理的论文，其中包括Gannon，Jalby和Gallivan[117]及Callahan，Kennedy和Porterfield[61，227]。Wolf和Lam的方法具有可以处理多维分块的特点，并显示出分块后，性能随着问题规模的扩大而提高[276]。McKinley，Carr和Tseng[210]将高速缓存分块和循环交换集成在一起[210]。

由于高速缓存分块在现代机器上是非常适合的，所以值得为了得到好的分块而付出更大的努力。Wolfe[282]及Kodukula，Ahmed和Pingali[186]都给出了更强有力的和复杂的变换技术的实例。

Ding和Kennedy最近的工作[103，105]显示出多层的循环合并和对齐对于提高高速缓存重用性并因此而减少存储带宽的需求是有效的。基于这样一种观察，通过增加重用性和提高存

储器有效带宽，其效益超出引入少量额外条件的代价，在实现中将不同形状和嵌套层次的循环进行合并。 509

多年来，软件预取作为一种提高高速缓存性能的方法，成为许多探索的主题。这个术语是由Callaham，Kennedy和Porterfield[61，227] 在他们最初的论文中提出的，论文中也给出了其实现的编译算法和显示其预期效果的模拟结果。本章中介绍的选择性预取技术是由Mowry，Lam和Gupta提出的，是Mowry学位论文的一部分[217，216]。

习题

9.1 图9-1中的循环重新排序算法对循环进行置换，以便减少对内存访问的跨距。它对非完全嵌套循环有作用吗？

9.2 针对下面的代码，确定循环的存储次序：

```
DO I = 1, N
   DO J = 2, M
      A(I, J) = A(I, J - 1) + B(I, J) + C(J)
   ENDDO
ENDDO
```

9.3 当模拟一个改变中的系统时，许多程序使用一个下面例子中的时间步循环：

```
DO TIME = 1, N
   DO I = 1, N
      POSITION(I) = F(POSITION(I))
   ENDDO
ENDDO
```

你可以使用本章中讨论的任何技术提高这个循环的高速缓存性能吗？

9.4 存储访问慢速不仅是由于高的延迟，也是由于低的带宽。在考虑带宽限制的前提下，预取的作用如何？至少给出两个说明为什么软件预取可能导致不必要的存储传输的原因。 510

第10章

调　度

10.1 引言

本书用大部分篇幅讨论如何以依赖关系的理论为基本工具，来发掘程序中的并行性。在本章中，我们研究如何利用相同的理论将程序中的并行性映射到计算机系统中有限的并行资源上。这个问题通常被称为调度，因为使用的主要的策略是通过牺牲一些执行时间而使给定的程序适应可利用的资源。调度问题的本质是最小化必须牺牲的时间总量。我们将会看到，对于在高性能计算机上编译后的程序的性能，好的调度可以产生巨大的效果。

我们将研究调度问题的两个不同的变形：指令调度和向量部件调度。

（1）指令调度是指定指令执行顺序的过程——在所有的体系结构上这都是一种重要的优化。在单处理器的层次上，利用体系结构提供的资源指令调度需要小心地平衡各种不同的指令在资源方面的需求。主要的目标是最小化一个指令序列中关键路径的长度。

（2）向量部件调度的目标是最有效地使用向量部件的指令和能力。这需要模式识别和同步的最小化。

在第 6.6 节中我们已讨论过多处理器调度，即在异步并行的协处理器间分配工作，使得各处理器负载平衡，并且最小化程序运行的时间。

虽然本章涉及这两类调度问题，但大量的论述集中在简单指令调度上。随着晶体管变得越来越小，芯片上的功能变得越来越强，目前大多数的设计者都致力于利用细粒度并行性功能。这一变化的后果是，指令调度从可有可无变为一个由编译器和硬件组成的系统的成功的关键。

10.2 指令调度

所有的现代机器体系结构都有在一个周期中发出多条指令的能力，因此理论上有获得高度的细粒度并行性的能力。有效地利用这种能力要求指令按照这样一种顺序排列，使得处理器能够找到并且发出那些能够被并行执行的指令。对于达到这一目标而言，主要有两方面的障碍：（1）指令间的依赖迫使指令顺序执行；（2）资源的限制迫使需要相同资源的指令串行化。单处理器指令调度的目标是以这样一种顺序生成指令，把有依赖的指令放置足够远从而使依赖不会导致延迟，同时使尽可能多的功能部件在每一周期处于忙状态。

大多数支持细粒度并行性的处理器属于两种类型之一：超标量（superscalar）和超长指令字（VLIW）（见第1.4节）。最常见的超标量处理器有由硬件控制和调度的多个功能部件，这意味着指令译码部件同时读入一组指令（通常为一个高速缓存的字长且与高速缓存字边界对齐）并确定它们之间的相关（或依赖）关系。如果有空闲的资源，并且多条指令可以被安全地并行调度，则该部件会将这些指令调度到多个功能部件上执行。超标量处理器通常通过停顿执行或寄存器重命名来避免执行一条操作数尚未就绪的指令，以此方式提供出现相关时的保护。很容易看出超标量执行为什么如此流行——因为指令部件可以接受已有的二进制码，一个超标量机器无需重新编译程序就能提供某种加速比。

VLIW（Very Long Instruction Word）结构较不常见，但是最近再度流行起来。超长指令字结构与超标量结构相反，获得高性能的主要的工作不在硬件设计方面，而在编译器这一边。
512 VLIW结构使用长指令字，其中包含控制每个可用的功能部件的指令域。一条这样的指令可以使得所有的功能部件都执行。对于VLIW机器，编译器必须显式地指出并行性,并且必须注意指令间的相关，因为VLIW结构通常不提供防止使用尚不可用的结果的保护。超长指令字机器的一个主要的缺点是缺乏二进制兼容性；由于其指令长度与功能部件的个数成比例，较新的有不同数目功能部件的机器如果不通过某种形式的翻译，将不能执行已有的二进制代码。结果是用户和软件厂商必须为每一代不同的VLIW机器重新编译他们的源代码。

对于超标量和超长指令字结构，有效开发指令级并行性的关键是重排指令序列，以使在每一周期利用尽可能多的功能部件。编译器为达到这一目标所采取的标准做法是首先发出一条顺序的已知是正确的指令流，然后重排这一指令流以便更有效地利用硬件的并行性。当然，这一重排的过程不能破坏原始指令流中存在的依赖关系。然而，指令调度中的主要问题之一是产生串行指令序列时必须考虑可用的资源，而这可能产生一些人为的在高层表示计算中并不存在的依赖。

寄存器是最常见的例子。举例来说，对于代码片断

```
a = b + c + d + e;
```

可能的指令流如下（假设所有在寄存器中的变量用相同的名字指称）：

```
add   a,b,c
add   a,a,d
add   a,a,e
```

这一指令流无法通过重排序来利用多个加法器。因为每条指令均依赖前一条指令的结果。然而，这些依赖关系纯属人为的，其所以发生这种情况只是因为所有的中间结果均累加到同一个寄存器中。如果使用不同的寄存器分配，例如

```
add   r1,b,c
add   r2,d,e
add   a,r1,r2
```

则前两个加法就可以并行完成。设计的原始指令流最小化所需的寄存器的个数；但这样作同
513 时也最小化可得的并行性。在这一层次上的反依赖都是由对资源的重用导致的。

前面的例子说明指令调度中最基本的冲突之一：产生原始的指令流。如果产生原始的指令流时已考虑到机器中可用的资源（特别是寄存器），就有可能引入由资源重用导致的人为依赖。另一方面，如果原始的指令流不考虑可用的资源，而只是用符号表示它们，就可能没有足够的资源正确运行调度后的指令流。举例来说，上面的第二个版本无法在一个只有5个寄存器的结构上执行，因为并行执行前两个加法就需要6个寄存器（假设我们不能破坏任何输入的内容）。这是在并行性和存储空间之间折衷的一个例子。

这一节介绍每个周期能发射几条指令的单处理器机器上指令调度的一些方法，而不论其是超标量或VLIW处理器。中心是重排目标程序的指令以最大限度地利用机器体系结构所提供的并行性（指令处理和功能执行中的并行性），而不危及程序的含义。假设部分资源（特别是寄存器）已被分配。其他的资源（如功能部件等）尚未被分配。

为了使调度处理更具一般性，我们将定义一个抽象机器模型，作为本章探讨的各种指令调度策略的目标机。这个机器有任意数目的功能部件，以及在一个周期中发出一条或多条各种类型指令的能力。这一机器模型应该相当接近未来新的计算机中实际的机器设计。

10.2.1 机器模型

基本的假设是一个机器包含若干不同类型的指令发射部件。每一个发射部件对应于一种机器资源，如整数或浮点算术运算部件。每个发射部件在每一周期可以发出一个操作到它所控制的那个功能部件（假设是流水线的）。每个发射部件将有一个关联的类型，指明它所控制的资源种类，还有一个延迟，是指从操作发出到操作的结果可用之间需要的周期数。

发射部件用两个整数来指定：部件类型和该类型中特定部件的序号。记号

$$\mathbf{I}_j^k$$

表示类型k的第j个部件。类型k的发射部件数目是m_k。这样机器中的发射部件的总数是

$$M=\sum_{i=1}^{l} m_i \tag{10-1}$$

514

其中l是机器中可以使用的发射部件类型的数目。由于每个发射部件每一周期能发出一个操作，机器的峰值发射率是每个周期M个操作。

为说明方便，假定我们的模型是VLIW体系结构，一条宽指令由M个子指令组成（每个子指令对应于一个发射部件）。编译器的工作为每一周期选择数量不超过M的一组操作构成一条宽指令，使得类型为k的操作个数$\leqslant m_k$。如果类型k的指令个数严格地小于m_k，那些未使用的指令发射槽填入空操作。通过列出能够按线性方式调度的指令表可以生成等价的超标量版本代码。

在这个模型之下，指令调度问题将被表示为一个图，其中顶点表示一条给定类型的指令，而边表示两条指令间的依赖关系，依赖边对应的延迟等于边的源顶点对应这类操作的延迟。

10.2.2 直线型代码的图调度

最简单的调度问题是一个基本块或直线型代码的调度。如可预见到的那样，基本的需求是一个依赖图，其上附有调度所需的额外信息。这个图（称为调度图）包含四个成分：

$$G=(N, E, type, delay) \tag{10-2}$$

其中，N是代码中指令的集合，对任一$n\in N$，$type(n)$给出它的类型，$delay(n)$给出它的延迟。若两条指令之间因共享一个寄存器而后一条指令必须等前一条指令执行完毕才能执行，则两条指令之间有一条边——真依赖（一次写后读或RAW通常称为这种语境中的相关）、反依赖（读后写）以及输出依赖（写后写）的概念在此都是很重要的。

一个正确的调度是图的顶点集到表示周期数的非负整数集的一个映射S，满足

（1）对所有的$n\in N$，$S(n)\geqslant 0$，

（2）如果$(n_1, n_2)\in E$，$S(n_1)+delay(n_1)\leqslant S(n_2)$，

（3）对任意类型t，最多只有类型t的m_t个顶点被映射到给定的整数。

直观上，条件（1）保证所有的指令均会在某一点被执行；条件（2）保证没有依赖被破坏；条件（3）保证在任一周期所使用的资源均未超过机器所提供的资源。 515

调度S的长度，记为$L(S)$，定义为

$$L(S)=\max_{n\in N}(S(n)+delay(n)) \tag{10-3}$$

直线型代码调度的目标是找到最短的正确的调度，其中一个直线型代码的调度S称为是最优的，如果对于同一个图的任何其他正确的调度S_1有

$$L(S)\leqslant L(S_1)$$

很明显，最短的调度也是执行时间最少的调度。

10.2.3 表调度

调度一个直线图的最简单的方法是使用拓扑排序的一个变形——构造并且维护一个在图中没有前驱的指令列表。任何在这个表中的指令均可被调度而不会破坏任何依赖，而且调度一个指令可能会导致把新的指令（被调度的指令的后继）加入到表中。图10-1给出这个算法，很自然地称它为表调度。

```
procedure list_schedule(G, instr)
  // G=(N, E, type, delay) 是调度图
  // instr[c]是输出的表，它将周期映射到指令的集合
  // W[MaxC]是一个工作表的数组，用来保存就绪的指令
  // MaxC比任何指令的最大延迟大1
  for each n∈N do begin count[n] := 0; earliest[n] := 0; end
  for each (n1, n2)∈E do begin
    count[n2] := count[n2] + 1;
    successors[n1] := successors[n1] ∪ {n2};
  end
  for i := 0 to MaxC - 1 do W[i] := Ø;
  Wcount := 0;
  for each n∈N do
    if count[n] = 0 then
      begin W[0] := W[0] ∪ {n}; Wcount := Wcount + 1; end
  c := 0; instr[c] := Ø; // c是周期号
  cW := 0; // cW是周期c的工作表个数
  while Wcount > 0 do begin
    while W[cW] = Ø do begin
      c := c + 1; instr[c] := Ø; cW := mod(cW + 1, MaxC);
    end
    nextc := mod(c + 1, MaxC);
    while W[cW] ≠ Ø do begin
      从W[cW]中选择并删除任意一条指令x;
      if在周期c有可用的类型为type(x)的发射部件then begin
        instr[c] := instr[c] ∪ {x}; // 调度该指令
        Wcount := Wcount - 1;
        for each y∈successors[x] do begin
          count[y] := count[y] - 1;
          earliest[y] := max (earliest[y], c + delay(x));
          if count[y] = 0 then begin
            loc := mod (earliest[y], MaxC);
            W[loc] := W[loc] ∪ [y]; Wcount := Wcount + 1;
          end
        end
      else W[nextc] := W[nextc] ∪ {x};
    end
  end
end list_schedule
```

图10-1 简单的表调度算法

基本的思想是在一条指令依赖的所有操作都执行完时立即调度该指令。*count*数组用来决定何时最后一个前驱已被调度，而*earliest*数组用来决定任一给定的指令最早可能的调度时间。当一条指令的最后一个前驱已被调度时，则它的*earliest*值迟到足以保证对该指令的所有的输入都是可用的。然后用这个值决定该指令存放在哪个工作表中。需要有足够的工作表以确保指令不会调度到它所依赖的指令之前。为保证这一点，工作表的数目只需等于最大的延迟加上1。这样，如果一条指令可被调度的最早周期是c，我们就把它存放在工作表$W[\mathrm{mod}(c, MaxC)]$中。

图10-1中的算法是一相当标准的基于拓扑排序的算法。这个算法典型的特征之一是随机地从工作表选择。如果一个周期的工作表中所有的指令都能在该周期中被调度，那么随机选择不会成为问题。但若没有足够的资源将所有的指令在同一周期调度，情况就有所不同了。在这种情况下，所有未调度的指令被放在下一个周期的工作表中，如果一条这样延迟的指令位于关键路径上，则调度的长度就会增加。下面用一个简单例子来说明这一点：

```
mult  c, a, b
mult  f, d, e
add   f, f, g
add   f, f, h
add   f, f, c
```

如果这个指令序列被调度到只有一个乘法部件的机器上执行，上面的调度算法必须从两个乘法操作中选择一个调度到第一个周期执行。这里，第二个是正确的选择；选择第一个将会因延迟位于关键路径上的加法的求值操作而增加调度的长度。可以通过修改算法，加入一遍对图的预先遍历，确定每条指令后面所需保留的最小周期数（利用项目调度中的一个标准算法）。图10-2给出这样一个图的反向遍历算法。

516 ~ 517

```
procedure find_remaining(G, remaining)

  // G=(N, E, type, delay) 是调度图
  // remaining[x]是输出变量，含有关键路径上从x开始到最后一个直接或间接地依赖于x的指令间的周期数

  for each n∈N do begin count[n] := 0; remaining[n] := delay(n); end
  for each (n1, n2)∈E do begin
    count[n1] := count[n1] + 1;
    predecessors[n1] := predecessors[n1] ∪ {n2};
  end
  W := Ø;
  for each n∈N do if count[n] = 0 then W := W ∪ {n};
  while W ≠ Ø do begin
    从W中任选并删除一条指令x;
    r := remaining[x];
    for each y∈predecessors[x] do begin
      count[y] := count[y] - 1;
      remaining[y] := max(remaining[y], r + delay(y));
      if count[y] = 0 then W := W ∪ {y};
    end
  end
end find_remaining
```

图10-2　确定关键指令

可以在表调度中加入对关键路径信息，这只需选择周期数保留值最高的一个，而不是随机地选择一条指令。这种选择的方式需要将工作表组织成一个优先队列，这会导致算法的时间复杂度中插入一个对数因子。这个方法在文献中被称为*最高层优先*（highest levels first，HLF）的启发式方法。

10.2.4 踪迹调度

表调度方法在基本块内可以得到很好的调度，但是在基本块的边界效果就不够好了。因为表调度方法不越过基本块的边界查找，因此它必须在基本块的尾部插入足够多的指令，以
518 保证所有操作的结果在调度下一个基本块之前是可用的。由于典型的程序中基本块的个数通常比较多，这些收尾的指令可能导致很大的开销。

消除这种开销的方法之一是产生一个非常长的基本块，叫做*踪迹*（trace），并对其应用表调度。简单地说，踪迹是由一些基本块组成的经过程序全部或部分的一条路径，*踪迹调度*假定控制沿这条路径中的基本块，一次调度整个踪迹中的指令。当然，在控制流能能够进入或退出这条路径的点上，必须插入一些修正代码。在理想情况下，被选中调度的第一条踪迹代表程序最常执行的路径，所以产生的调度对大多数情况来说是最优的。因为踪迹调度使用基本块调度技术，因此它不能调度有环的图。或者必须将循环展开，或者把循环体作为一个基本块调度，而在边界处插入修正代码。

对于一个给定的有向无环调度图，踪迹调度的第一步是把图划分为一组踪迹。第一条被选中的踪迹希望是最可能通过此图的路径。这一路径被作为一个大的基本块调度。接下来，选择经过那些剩余的基本块的最可能的路径作为下一条踪迹，并加以调度。如果在第二条踪迹中的某一点有分支进入第一条踪迹，则需在其间插入修正代码以满足第一条踪迹中对资源的假设。同样地，若在第一条踪迹中的某处有分支进入第二条踪迹，亦需在其间插入修正代码以满足第二条踪迹中对资源的假设。这个过程将重复直到程序中所有的基本块均被调度。

简而言之，踪迹调度可以被描述为以下三个步骤的重复应用：

（1）*选择一条穿过程序的踪迹*。踪迹的选择并不像初看起来那么容易，即使有描述信息也是如此。

（2）*调度*选择的踪迹。

（3）*插入修正代码*。因为修正代码是调度踪迹之外的新的代码，它建立必须反馈给踪迹调度的新的基本块。

第三步是踪迹调度最有趣的方面。为说明代码修正所需的类型的，考虑图10-3中简单例子。

图中，踪迹由图左边的基本块组成。如果这条踪迹被调度成代码序列

```
j = j + 1
if  e1
i = i + 2
```

那么对i的赋值被移到分支语句的下面，这将会使得对k的赋值得到错误的结果。这个问题可以通过沿控制图中的分裂边复制对i的赋值而解决（图10-4）。

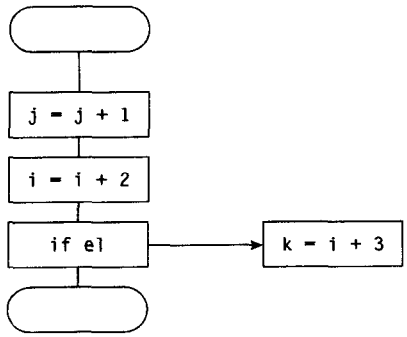


图10-3 需要修改代码的踪迹

新插入的指令可以独自作为一条踪迹被调度，也可以和对k赋值的基本块所在的踪迹一起

调度。一般来说，将一些操作移动到一个踪迹的分支点之下或一个踪迹的连接点之上，均需插入如图中所示的类似的修正代码。而在无法通过依赖分析确定对其他指令的影响的情况下，踪迹调度不会移动操作向上越过（控制流的）分支点或向下越过（控制流的）汇合点。

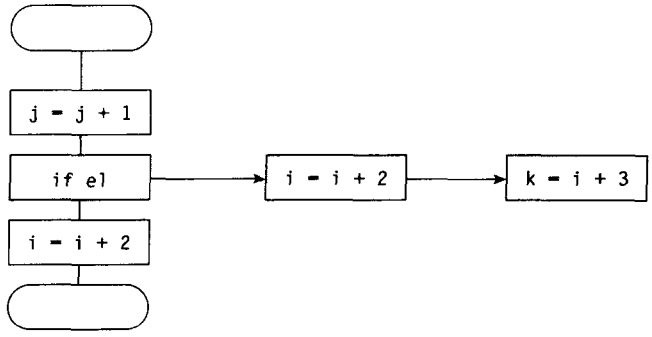


图10-4 调度后带有修改代码的踪迹

鉴于调度一条踪迹可能导致新踪迹的产生，一个很自然的问题是：踪迹调度是否收敛？也就是说，在踪迹调度的过程中，有没有可能出现这种情况：新插入的踪迹和正在调度的踪迹一样多或更多？幸运的是，对第二个问题的答案是不会；踪迹调度一定会收敛。但是，在最坏的情况下，虽然收敛，但会导致产生大量的修正代码。从Ellis的学位论文[109]摘引的图10-5的例子说明这种最坏的情况。

踪迹调度可以像移动其他操作那样移动分支操作越过多个基本块，因此，从这个例子可能得到这样的调度，其中调度的分支操作的顺序被颠倒过来了：

C_n
A_n
C_{n-1}
A_{n-1}
...
C_1
A_1

这样的调度将需要如图10-6所示的修正代码。对每一条离开踪迹的边，需要复制除控制该边的基本块之外的所有其他基本块。

这里，总的操作数目增加到$O(nn!)$，大约是$O(ne^n)$。尽管这样极端的情况在实际的踪迹调度的实现中尚未出现过，但要构造出类似上述的测试用例来也是容易的。假定踪迹调度利用循环展开来处理循环，一个仅包含一个条件判别的简单的内层循环展开足够多的次数后就可能产生上述情形。

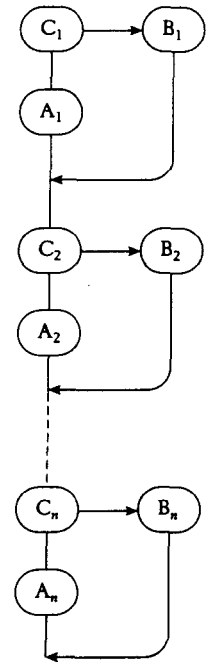


图10-5 说明最坏情况下代码膨胀的例子

程序设计语言中的循环结构所传递的关键信息之一是指出循环体会被执行许多次。依赖分析和向量化都利用这一迭代性质，试图分析并调度循环体的不同迭代。踪迹调度的缺点之一是不能以这种方式处理循环，而是必须要将它转换为直线型代码。后面几段将介绍一些利

用循环的迭代特性的调度方法，但在这之前，我们先介绍与直线型代码调度有关的一些关键问题是值得的。

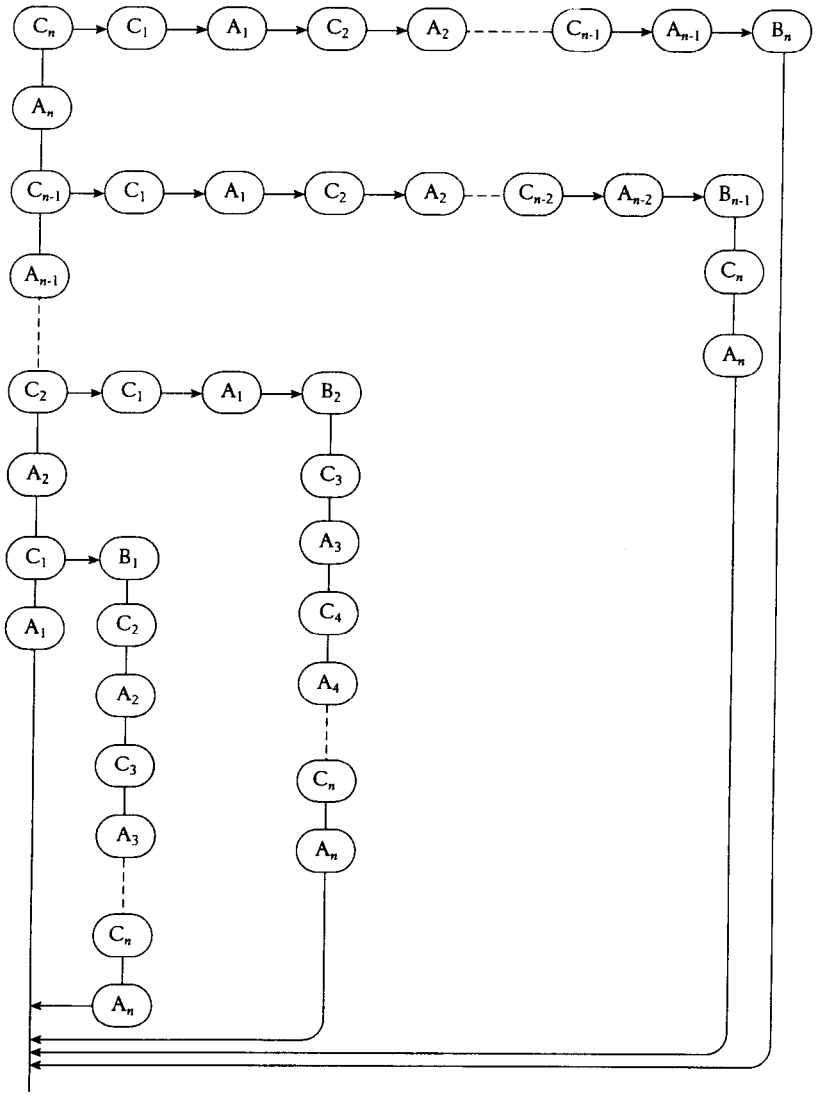


图10-6 踪迹调度的代码膨胀

直线型代码调度中的一些问题

在到目前为止的讨论中，指令调度看起来比项目管理——建一个PERT图再对其拓扑排序复杂不了多少。但实际上，指令调度所需的大部分复杂分析都作为细节隐藏在基本的算法背后。这一段较深入地说明一些细节。

到目前为止，在讨论指令调度时我们忽略了一个关键的机器资源——寄存器。介绍的每一种算法都假定在指令调度之前寄存器已经被分配了。尽管这样做会由于限制了一组资源而大大简化了指令调度，但同时也限制了所得到的并行度。一旦一个变量被限定到一个特定的寄存器，就很容易因另一个与之无关的变量被分配到同一个寄存器而人为地引发依赖。这些依

赖进而会阻碍两条指令的并发执行，尽管在原来的程序中并没有要求两者串行执行。寄存器分配和指令调度的相对顺序目前仍是许多计算机科学家争论的问题。

寄存器分配的第二个影响是掩盖有效调度所需的真正的依赖分析。容易这样假设，将变量归结为寄存器会使得构造依赖图更容易，因为寄存器是明确定义的标量。若仅对寄存器而言，这是对的；但这里忽略了内存的取数和存数操作。内存取数操作若未命中高速缓存会非常慢，而大多数计算必须等待所需值的取数操作完成。因此，能否将取数操作调度得尽可能早对于一个有效的指令排序是至关重要的。在这种情况下，依赖分析是必需的，因为取数操作不能移到对同一内存单元的存数操作之前。没有相当复杂的分析，在存储访问之间进行移动几乎是不可能的，而结果就是产生很差的调度。 519 ~ 523

10.2.5 循环内的调度

前面提到，直线型代码调度和踪迹调度的主要缺点是它们对循环的处理不够好。它们通常采用的作法是展开循环，使得循环体尽可能地大，然后再尽可能高效地调度循环体。这种方法忽略跨越循环迭代间的并行性。这一节介绍另一种方法，它试图最大限度地发掘迭代间的并行性，而不是只考虑直线型代码的调度。由于循环体现的计算将被重复执行，这种方法更着重在程序执行的热点区域上。

核心调度

调度循环的一种直接的调度策略是将循环分为三个部分：

1. 核心，包含当循环的执行到达稳态后每一轮迭代必需执行的代码
2. 前缀，包含循环的执行到达稳态前需要执行的代码
3. 后缀，包含核心执行完以后，为结束循环还需执行的代码

这里，调度的目标是最小化核心代码的执行时间，因为它代表循环重复执行的部分。而前缀和后缀部分只是为核心代码的执行作准备和收尾工作。正因为如此，所以有如下的定义。

定义10.1 核心调度问题是对一给定的循环寻找其最小长度的核心代码。

那么，最小化核心代码的长度能保证得到循环的最优调度吗？对于迭代次数很大的循环而言，显然是这样的。而对于迭代次数比较小的循环（迭代次数为零是最坏的情况，因为任何前缀都降低速度），就不一定了。不过，由于迭代次数比较少的循环在总的执行时间中所占的比例也比较小，如何调度它并不重要。因此，在本章其余部分，我们总假设最优的核心调度就代表最优的循环调度。 524

定义10.2 核心调度问题是一个图

$$G=(N, E, delay, type, cross)$$

其中，对E的每条边定义的$cross\ (n_1, n_2)$给出n_1和n_2之间依赖关系跨越的迭代次数。

再次说明，如果两个结点之间存在依赖，则两个结点之间存在一条边，并且$cross$仅代表依赖的阈值（距离）。循环无关依赖的阈值为0。

核心调度的目标是指令经过循环迭代时时间上的移动，而不是在一个迭代内空间上的移动。也就是说，那些很早就需要用到其结果的关键指令被移到前面的循环迭代中执行，这样，在当前迭代中需要时，它们的结果已经是可用的。同样，位于关键路径尾部的指令被移到后

面的迭代中去，以缩短当前迭代的完成时间。换句话说，循环的一个迭代的执行体以流水方式跨越多个迭代，以便在一次迭代中最大限度地利用可用资源。这一过程通常称为软流水方法，可以更形式化地定义如下：

定义10.3 核心调度问题的一个解是一对表的二元组 (S, I)，其中调度S将每一条指令n映射到核心内的一个周期，而迭代I将每一条指令映射到从零开始的一个迭代偏移量，使得对E中的每一条边 (n_1, n_2)，满足

$$S[n_1] + delay(n_1) \leqslant S[n_2] + (I[n_2] - I[n_1] + cross(n_1, n_2))L_k(S) \tag{10-4}$$

其中，$L_k(S)$ 是S的核心长度。

$$L_k(S) = \max_{n \in N}(S[n]) \tag{10-5}$$

用一个简单的例子可以使这个概念变得更清楚。假设我们调度下面的循环体：

```
      lw     r1, 0
      lw     r2, 400
      lf     fr1, c
10    lf     fr2, a(r1)
11    addf   fr2, fr2, fr1
12    sf     r2, b(r1)
13    addi   r1, r1, 8
14    comp   r1, r2
525   15    ble    10
```

同样，假定机器有三个功能部件：一个存取部件，一个整数运算部件，以及一个浮点运算部件。所有取数操作有2个周期的延迟；存数操作有1个周期的延迟；浮点加法有3个周期的延迟。整数部件处理所有分支操作，对blt操作，跳转到目标需要一个周期的延迟。所有其他的整数指令均有1个周期的延迟。对于上面循环中的指令，一个合法的需要3个周期的核心调度如下：

```
S[10] = 0; I[10] = 0;
S[11] = 2; I[11] = 0;
S[12] = 2; I[12] = 1;
S[13] = 0; I[13] = 0;
S[14] = 1; I[14] = 0;
S[15] = 2; I[15] = 0;
```

这个调度，如图10-7所示，指出在一个迭代的第一个周期，执行浮点取数和增量操作；比较操作在第二个周期执行；而分支、浮点加法和浮点存数操作则在第三个周期执行。

部件	取-存	整数	浮点
周期1	lf a(r1)	addi r1,r1,8	
周期2		comp r1,r2	
周期3	sf fr2,b-16(r1)	ble	fadd fr2

图10-7 核心调度的例子

需要注意的是，如果保持原程序中指令的形式，这个调度就是非法的；为得到正确的结果，必须对指令的形式作一定的修改。举例来说，在r1两次增量及下个迭代的浮点取数操作

改写结果之后，浮点存数操作被提前一个迭代执行。这意味着该存数操作需要对r1作调整，并需要一个额外的浮点寄存器来存放浮点加的结果，直到存数操作可以执行为止。假定fr3是可用的，我们得到如下正确的核心：

```
k1  lf fr2, a(r1);      addi r1, r1, 8
k2                      comp r1, r2
k3  sf fr3, b-16(r1);   ble  k1         addf fr3, fr2, fr1
```

当然，这段代码在第一次迭代中是不正确的，因为它假定fr3和r1已被之前的迭代定值。这意味着在进入核心之前必须执行该循环一次（除掉依赖于已完成的前一次迭代的那些代码），以完成必要的初始化。 526

```
    lw r1, 0
    lw r2, 400
    lf  fr1, c
p1  lf fr2, a(r1);   addi  r1, r1, 8
p2                   comp r1, r2
p3                   beq  e1;          addf fr3, fr2, fr1
```

并且，最后一次迭代的结果不被写回，这需要另外加入少量收尾指令：

```
e1  nop
e2  nop
e3  sf  fr3, b - 8(r1)
```

整个循环的调度在下面给出。核心的长是 3 个周期，假定整数部件在每个周期处于忙状态，因此不可能有更短的调度。

```
    lw r1, 0
    lw r2, 400
    lf fr1, c
p1  lf fr2, a(r1);      addi r1, r1, 8
p2                      comp r1, r2
p3                      beq  e1;         addf fr3, fr2, fr1
k1  lf fr2, a(r1);      addi r1, r1, 8
k2                      comp r1, r2
k3  sf fr3, b-16(r1);   ble  k1;         addf fr3, fr2, fr1
e1  nop
e2  nop
e3  sf fr3, b-8(r1);                     addf fr3, fr2, fr1
```

指令p1-p3形成前缀，k1-k3形成核心代码，而e1-e3形成后缀。

定义10.4 令N为循环上界。那么调度长度$L(S)$由下式给出：

$$L(S) = NL_k(S) + \max_{n \in N}(S[n] + delay(n) + (I[n]-1)L_k(S)) \qquad (10\text{-}6)$$

这个定义使我们注意到，对于非常大的N，最小化核心的长度也就是最小化调度的长度。因为N在编译时间通常是未知的，而迭代次数较小的循环无论怎样调度，对总的运行时间的影响也较小，看来假定N很大是合理的，并由此尝试构造可能最短的核心。 527

核心调度算法

存在最优的核心调度算法吗？为回答这个问题，必需为做好调度确定合理的下界。实际

上，根据不同的分析，我们可以得到两种不同的下界。

资源使用约束 假设在循环内没有依赖环。令#t表示每一次迭代中必须要发射到类型为t的功能部件上执行的指令的条数。那么

$$L_k(S) \geqslant \max_t \left\lceil \frac{\#t}{m_t} \right\rceil \tag{10-7}$$

换句话说，如果每个周期只能发出m_t个类型为t的运算，则发出n条指令需要至少$\lceil n/m_t \rceil$个周期。这样，如果我们对所有的操作类型取这些下界的最大值，核心不可能比此最大值短。

我们可以证明，对于一个没有依赖环和控制流变化的循环，存在调度S使得

$$L_k(S) = \max_t \left\lceil \frac{\#t}{m_t} \right\rceil \tag{10-8}$$

换句话说，在没有依赖环的情况下，找到一个最优长度的核心调度总是可能的。为了得到这个结果，我们构造一个如图10-8所示的算法，它对给定的长度L确定核心的调度。注意这个算法要求L满足式（10-7）中的不等式。

```
procedure loop_schedule(G, L, S, I)
  // G是循环体的调度图
  // S和I定义最终的调度中的核心
  // L是被调度的循环中指令的条数，该值至少为等式（10-8）给出的最小调度长度;
  对G作拓扑排序;
  for each G中的指令x，按拓扑顺序do begin
    earlyS := 0;  earlyI := 0;
    for each G中x的前驱y do
      thisS := S[y] + delay(y);  thisI := I[y];
      if thisS ≥ L then begin
        thisS := mod(thisS, L);  thisI := thisI + ⌊thisS/L⌋;
      end
      if thisI > earlyI或thisS > earlyS then begin
        earlyI := thisI;  earlyS := thisS;
      end
    end
    从周期earlyS开始，找到第一个周期c0
      其中，x所需的资源是可用的，
      如果需要，环绕到核心的开始;
    S[x] := c0;
    if c0 < earlyS then I[x] := earlyI + 1;  else I[x] := earlyI;
  end
end loop_schedule
```

图10-8 在没有依赖环的循环中寻找最小长度核心

这个算法实质上是一个表调度算法，其中加了记分板以记录跨越循环迭代时的资源使用情况。当调度器为每一条指令分配发射槽时，它同时在记分板上标记处于忙状态的资源。当它尝试在等式（10-8）指明的长度的核心结束之后的某一时刻发射一条指令n时，调度器要作

修改。它会绕回到核心的起始，并将增加$I[n]$的值，因此将指令n移到后面一个迭代。因为调度长度是由限制最紧的资源决定的，因此所有其他类型的指令都将轻而易举地归并到这个调度中。对于限制最紧的资源，把这样的指令移到后面的迭代中（由于没有依赖环这是可能做到的），这样我们总可以在核心中找到空槽，将指令放进去，因为正好有我们需要的那么多空槽。这对寄存器可能会有影响，但假设有足够多的寄存器，所以保证核心适合于给定长度。

为说明这种方法，考虑在一台有3个整数部件和两个访存部件的机器上调度下列并无多大意义的指令序列：

```
10  lw    a, x (i)
11  addi  a, a, 1
12  addi  a, a, 1
13  addi  a, a, 1
14  sw    a, x (i)
```

定理指出存在一个长度为1的核心调度：三条整数指令分配给三个整数部件。一个普通的表调度算法不可能找出这样一个调度，因为这里一个代码序列中每一条指令都依赖于前一条，故需要5个周期才能完成。 528 ~ 529

调度第一条指令是简单的，导致使用下列资源：

存储器1	整数1	整数2	整数3	存储器2
10: $S=0; I=0$				

第二条指令的普通调度将它压到该调度结束之后，所以它回绕到下一次迭代：

存储器1	整数1	整数2	整数3	存储器2
10: $S=0; I=0$	11: $S=0; I=1$			

对余下的每条指令类似地重复此过程，导致最后一个资源的使用

存储器1	整数1	整数2	整数3	存储器2
10: $S=0; I=0$	11: $S=0; I=1$	12: $S=0; I=2$	13: $S=0; I=3$	14: $S=0; I=4$

这个调度的长度是1；有意思的是，在每个周期，每个部件都在循环的一个不同的迭代上执行。这一事实意味着这一序列不能按原来书写的形式执行，因为现在每个部件都需要一个不同的寄存器用于它的迭代。最终的代码必须转换成某种与下面代码类似的形式：

```
10  lw    a, x (i)
11  addi  b, a, 1
12  addi  c, b, 1
13  addi  d, c, 1
14  sw    d, x (i)
```

这个例子说明结果的主要问题：未回答是否有足够的硬件寄存器。这个问题将在后面“寄存器资源”一段中进一步讨论。

在等式（10-8）中考虑依赖环的限制的好处在于能保证只要有资源可用，指令就能被调度。依赖环会导致指令被推向多个迭代，从而增大最小调度的大小。正如下面的限制所说明的。

有环数据依赖限制 给定一个依赖环$(n_1, n_2, \ldots, n_k)$，则 530

$$L_k(S) \geqslant \frac{\sum_{i=1}^{k} delay(n_i)}{\sum_{i=1}^{k} cross(n_i, n_{i+1})} \tag{10-9}$$

因此有

$$L_k(S) \geqslant \max_{c}\left(\frac{\sum_{i=1}^{k} delay(n_i)}{\sum_{i=1}^{k} cross(n_i, n_{i+1})}\right) \tag{10-10}$$

其中c遍取循环中的所有依赖环。

式（10-9）中的分母表示依赖环跨越的循环迭代次数的计数，分子表示执行一次依赖环上的所有指令所需的机器周期数。这样式（10-9）的右端代表每一次迭代中计算依赖环的所需的周期数。显然，一个正确的核心每一次迭代所需的周期数不可能更少，否则不可能计算依赖环上的正确值。

定义10.5 式（10-9）右端的值称为依赖环的斜率。

由式（10-7）和式（10-10）给出的两个约束条件，构成带有依赖环的循环的核心调度算法的基础，如图10-9所示。此算法先利用式（10-7）和式（10-10）计算核心调度的最小值，然后利用图10-8中的算法*loop_schedule*，试图找出该长度的一个调度。因为只要L满足式（10-7）中的不等式，此算法总能在长度为L的核心中找到调度，所以我们必须要问，究竟什么原因会导致失败？在这种情况下，如果任何依赖环的长度增加，就会导致核心完成计算的迭代次数的增加，我们就得不到满意的调度。这可以通过检查是否依赖环的某一条指令能延迟一个迭代，或者通过检查一个调度的延迟是否会导致依赖环中的最后一条指令产生的输出结果来不及传给依赖环的下一个迭代的第一条指令。

```
procedure kernel_schedule(G, S, I)
  // G是循环的调度图
  // sched是输出调度
  使用all-pairs shortest-path算法找出调度图G中斜率最大的环;
  将所有这样的环标为关键环;
  将G中所有在关键环上的指令标为关键指令;
  按式（10-10）给出的关键依赖环的最大斜率和式（10-7）给出的硬件约束计算循环的下界LB
  N:=原循环体中指令的条数;
  令G0是G通过消除循环体内依赖环上到最前面指令的边而得到的所有破坏的图;
  failed:=true;
  for L:=LB to N while failed do begin
    // 尝试按长度L调度循环
    loop_schedule(G0, L, S, I);
    // 测试调度是否成功
    allOK:=true;
```

图10-9 核心调度的算法

```
        for each依赖环C while allOK do begin
          for each C上的指令v while allOK do begin
            if I[v] > 0 then allOK := false;
            else if v是环C中的最后一条指令,
              v0是环中的第一条指令，且
              mod(S[v] + delay(v), L) > S[v0]
              then allOK := false;
          end
        end
        if allOK then failed := false;
      end
    end kernel_schedule
```

图 10-9（续）

如果算法未能找到一个合适的调度，它会增加正在搜索的最小长度的调度的值，并且再尝试一次，直到它最后成功。需要注意的是，循环总能被调度到一个具有循环体长度的核心中，所以最后总会成功的。但即使调度成功了，它所需要的寄存器的个数可能超过机器中的寄存器的数目。

图10-9中的核心调度算法的效果可以通过对图10-8中的*loop_schedule*稍作修改而得到提高，类似于前面讨论过的对表调度算法的改进一样。当选择下一条指令进行调度时，如果不止一条指令就绪，该算法应首先选择一条在关键依赖环上的指令，其次选择一条在任何依赖环上的指令，最后才调度不在任何依赖环上的指令。这一修改是简单的。

531
~
532

这一节中的算法计算循环的核心调度，但仍需要产生前缀和后缀代码。下一节描述这方面的细节。

前缀和后缀的生成

在前面的简单的核心调度的例子中，生成前缀和后缀的关键信息是达到计算的稳定状态所需的迭代次数。同样的信息在这些例子中也是需要的，并称之为调度的迭代的范围（range）：

$$range(s) = \max_{n \in N}(I[n]) + 1 \tag{10-11}$$

按照这一定义，调度的范围是对应于原循环发射的一个迭代中的所有指令所需的迭代次数。

范围是使循环达到稳定状态（注满流水线）所需执行的迭代次数，以及稳定状态后执行完循环（排空流水线）所需的迭代次数。要得到一个完整的核心（带有稳定状态中的每一条指令的拷贝）需要来自$range(S)$个不同迭代的指令，包括目前刚刚开始执行的指令。因此如果$r = range(S)$，执行的第一个完整迭代是第r个，则前缀的长度是

$$(r-1)L_k(S)$$

有了这一信息，可以如下生成前缀：首先产生核心的$r-1$个拷贝，然后对所有满足$I[n]=i<r-1$的指令n，将其在迭代1到迭代i中的拷贝用no-op替换。不插入no-op，前缀就不能被有效地重调度（保证是最优的），但是插入了no-op以后，就可能显著地压缩前缀。在那些情况下，用表调度算法重新调度前缀也是有用的，可以减小的前缀的长度。

我们可以用类似的方法得到后缀长度的界。一旦核心的最后一次完整迭代完成（这是原来代码的最后迭代中的指令开始发射的迭代），它就可以在$r-1$次迭代中完成。然而，这还不

够。不同于前缀的情况，后缀在完成最后的指令时需要一些额外的时间，以保证所有带有循环外的代码的相关是适合的。例如，如果一个调度的最后一条指令是一个store，并且store需要五个周期来完成，后缀代码必须包含5个额外的no-op以保证store得以完成，后面的load操作不会读取无效值。后缀代码所需的额外的时间是

533

$$\Delta(S) = (\max_{n \in N}(((I[n]-1)L_k(S) + S[n] + delay(n)) - rL_k(S))^+ \quad (10\text{-}12)$$

这里的上标“+”在第3章中定义，表示取表达式的正部，如果表达式的值大于零，它就等于表达式的值，否则就等于0。

因此后缀的长度有下面的上界：

$$(r-1)L_k(S) + \Delta(S)$$

跟前缀的情况相同，后缀的长度可以通过表调度减少。

这样，为了生成后缀代码，只要产生核心的$r-1$个拷贝，然后对所有满足$I[n]=i<r-1$的指令n，将其在迭代$i+1$到迭代$r-1$中的拷贝用no-op替换。

寄存器资源

迄今为止，尽管我们在讨论软流水时已提到了由于流水的影响导致寄存器需求增大的问题，但并未论及如何解决这个问题。自然可以说推迟到现在才讨论这个问题，是因为我们可以提出一种通用的软流水算法来漂亮地解决这个问题。不过实际上没有这样的漂亮的解决方案。当流水线调度所需要的寄存器的个数超过机器中寄存器的数目时，就必然导致寄存器溢出或者代码重新调度。在解决这个问题的过程中提出了许多启发式技术，但是完全令人满意的解决方案还没有找到。假定寄存器溢出很少发生（这似乎是一个合理的假定），当一个最优的调度没有足够的寄存器可用时，插入内存溢出指令并重新调度似乎是合理的。

然而，缺乏寄存器并不是软流水面临的唯一问题。即使有足够的寄存器，在使用现有的寄存器时也会发生寄存器名字冲突。这种冲突是由于试图使用同一寄存器存放不同的原来迭代中活跃区域重叠的两个不同的值所造成的。然而正如我们在第8章中所见的，这类名字冲突问题通常能通过循环展开解决。

控制流

迄今为止，我们只讨论了内部没有控制流的循环的软流水。这样的循环的执行过程是规则的、可预见的，我们能够比较有把握地通过调度得到改进的结果。控制流在两个方面使软流水复杂化：

（1）因为存在不同调度长度的控制流路径，就不可能再假定每一次迭代所需要的周期数
534 都相同。

（2）在分支条件控制下的条件执行指令必须在控制流汇合在一起之前发出。而来自不同迭代的指令不可以随意地混合在一起。如果不能在控制流汇合前发出一条有控制条件的指令，则必须实现保存控制判定的机制，而这样做既不容易也不高效。

因为带有控制流的循环难以预测，移动指令的空间也较小，结果是在软流水中处理控制流变得很困难。一种选择是在循环体中使用路径调度；这样做基本上是把赌注押在最有可能的控制流路径上。

第二种方法常用在对预测执行提供硬件支持的机器上。如果使用7.2节中讲到的if转换，我们可以通过把分支之下的指令转换为预测的形式，从而消除所有的前向分支指令。为了得

到最好的效果，7.2节中的算法需要修改，使得计算条件的值的指令数最少。但是，这样的修改是容易的。

对有预测硬件的机器，另一种策略如7.3节所述，先建立控制依赖，然后利用控制依赖进行调度，最后利用图7-25和7-26中的代码生成算法来生成最终的代码，对每一条指令的预测，是通过对控制它的最直接条件求值产生的布尔变量。目前，至少有一个英特尔IA-64体系结构的代码生成技术的研究组正在使用这种方法[107]。

最后一种在软流水中加入控制流的方法由Lam[194]提出，该方法首先使用非流水方法调度控制流区域，然后在流水处理时将这些区域当作黑盒子对待。换句话说，控制流区域被当作一条宏指令，执行这条指令需要在若干周期中占用所有的资源。在控制流区域外部的指令可以使用软件流水线调度到这个区域之前或之后执行。这种策略在图10-10中说明。

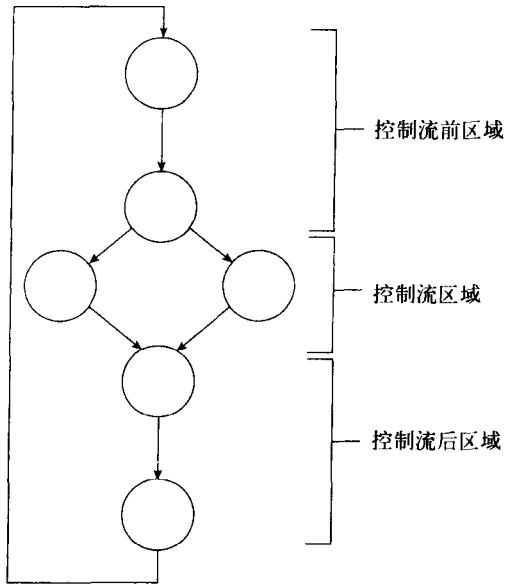


图10-10 有控制流的软流水

控制流区域中的每条路径被独立地调度，然后整个控制流区域被归约为一条超级指令，占用周期数等于最长路径长度的全部资源。这个步骤完成后，控制流前区域、超级指令和控制流后区域可使用标准软流水方法进行调度。

对这种策略的一种改进是允许这种超级指令在输出不同结果时有不同的延迟。也就是说，这种超级指令仍然被当作一条单一输出的指令发射，但其输出在不同的时间产生，并且在该超级指令执行完之前就释放一些资源。这样将使得控制流后区域中的部分指令能够被调度到控制流区域的所有路径上，从而缩短核心长度。

可以用类似的策略来缩短控制流前区域。假设在控制流前区域的指令i_1和控制流区域的指令i_2之间有延迟为d的依赖，我们不需要让i_1在超级指令前d个周期发射。如果i_2已被调度到超级指令的第k个周期执行，那么i_1可以在超级指令前$d-k+1$个周期发射。如果$d-k+1$是负数，当空资源在两条路径上都可用时，则i_1就可以与超级指令重叠。

10.3 向量部件调度

一般说来，向量指令的调度比指令部件的调度简单得多。首先，一条向量指令，按其定义涉及到许多标量指令的执行，所有这些指令都是以最高效的流水方式执行的。因而仅发射一条向量指令就已经表示显著的加速和有效的重排。其次，向量指令通常会执行很长时间以致于不可能让其他所有的执行部件都保持繁忙。向量指令像一把伞，它的代价隐藏其他所有指令的代价。

535
~
536

然而，对于大多数的向量处理器而言，指令调度仍有用武之地。向量处理器通常支持硬件中的操作链接；对这样的处理器花费少量时间对指令重新排序能显著缩短程序的执行时间。同样地，向量处理器经常被用作内存映射的协处理器。这样的协处理器通常能通过仔细调度同步指令而提高速度。本节将简要地讨论这些调度机会的几个重要方面。

10.3.1 链接

鉴于一个典型的向量操作所需的执行时间较长，加之向量部件通常拥有多个不同的执行部件这一事实，多向量部件的链接就成了常见的优化方法。链接是一种硬件机制，它可以用来识别一个向量操作的输出在什么时候可以作为另一个不同的向量操作的输入使用。在这种情况下，链接硬件会将第一个操作的输出结果直接传递到第二个操作的执行部件，从而实现两条指令的重叠操作。举一个简单的例子:

```
vload   t1, a
vload   t2, b
vadd    t3, t1, t2
vstore  t3, c
```

这段指令从内存中取两个向量，把它们相加，并把结果存回内存。假定有两个从存储器读出的流水线a和b（正如在load指令中指出的），且两个操作是独立的。如果简单地假设每一条向量指令要花费64个周期，且这两条load指令同时开始进行操作，那么，这段代码的非链接方式执行就需要192个周期：其中64个用来装载两个向量寄存器，64个用来做加法运算，余下的64个用来保存结果。而链接方式的执行只需要花费66个周期，速度几乎是非链接方式的3倍。在链接方式下，第一个周期开始向量加法运算（假定在第一个周期时，向量的第一个元素已经取到），它的第2个周期开始保存向量（假定加法运算在第一个周期已经完成），load，add和store都以流水方式重叠执行。

大多数支持链接操作的向量部件都有完整的记分板（记录向量寄存器和操作部件情况），识别链接机会的任务基本上就由硬件来完成（就像它识别超标量一样）。然而，编译器必须提供某种支持。在最坏的情况下，编译器必须在指令操作码中设置特殊控制位，通知硬件对两
537 个操作进行链接。然而，通常情况下，编译器只需要把两个操作尽可能地移到一起以便记分板识别链接机会。

如果只是要求将指令移近，那么产生链接机会只需对指令调度作一个很小的修改，即在指令调度中提升检索和识别满足链接限制的模式的能力。既然这里涉及的是向量而非标量，那么相关冒险检测显然需要全面的依赖分析。

初看起来，对链接的最优使用可能是容易的。在大多数实际情况中，确实最优的链接可以从简单的指令调度中得出。但也并不总是这样，下面的例子可以说明这一点:

```
i1   vload  a, x (i)
i2   vload  b, y (i)
i3   vadd   t1, a, b
i4   vload  c, z (i)
i5   vmul   t2, c, t1
i6   vmul   t3, a, b
i7   vadd   t4, c, t3
```

假定这段代码是在有两条取数流水线、一条加法流水线和一条乘法流水线的向量部件上执行，第一个加法运算将和load进行链接操作，第一个乘法运算将和第三个load进行链接操作，最后一个加法运算将和最后一个乘法运算进行链接操作。整个计算需要三个完整的向量操作。这是相当好的，但是重新排列代码有可能会做得更好:

```
vload  a, x (i)
vload  b, y (i)
```

```
vadd   t1, a, b
vmul   t3, a, b
vload  c, z (i)
vmul   t2, c, t1
vadd   t4, c, t3
```

在这个版本中，完整的向量操作的个数已经减少到了两个。技巧是把所有依靠相同操作结果作为输入的操作集中到一起。

链接问题可以通过使用8.6.4节中介绍的加权合并算法的变形得以解决。Ding和Kennedy [104]介绍了这个算法的一个考虑资源约束的版本。这一变形使我们可以在合并过程的每一步判定合并是否会产生一个需要过多资源的组。在合并的情况下，我们总是想把尽可能多的能够链接的向量指令集中到一起，只要我们有足够的资源以链接方式同时执行这些操作。例如，如果一台机器有两条取数流水线、一条存数流水线、两个加法部件和一个乘法部件，我们就 538
不会试图链接比两个取数、两个加法、一个乘法和一个存数更多的东西。

8.6.3节的合并算法和考虑资源限制的合并算法所共有的一个有用的特性是，在合并之后，重新动态地计算权值。这意味着如果选择一个加法运算和一个乘法运算进行链接操作的话，则对这两个指令进行合并后，另一个同时输入到这个加法和乘法运算的取数操作将被赋予一个更高的权值。

利用这种策略，这个算法的步骤如下:

(1) 为将要进行链接操作的直线代码构造依赖图。

(2) 用代表边的向量的长度作为边的权值。如果不能确定向量长度，就使用整个向量寄存器的长度。

(3) 用限制加权合并算法来确定最大的合并组。在选择下一条合并边的每一步，如果有几条边由于最高权值而绑在一起，就选择最近加入到合并组的一条边，其中偏向于选择源点和汇点在原来的顺序中最早的那条边。

如果我们对本节开始的那个例子使用此算法的话，就从建立如图10-11所示的依赖图开始，图中所有的边的权值相等。然后开始合并。

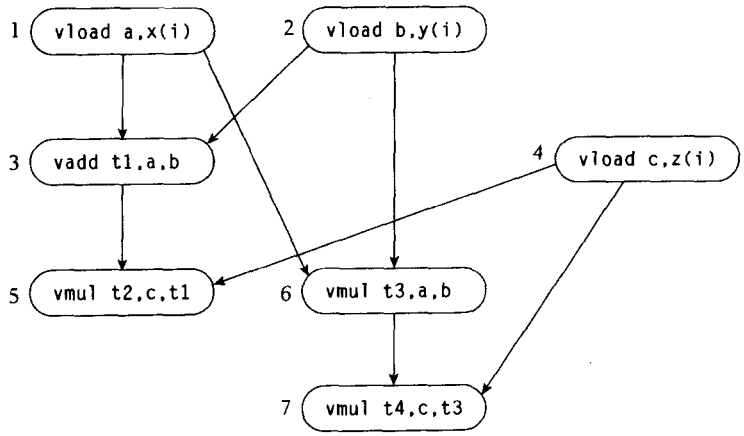


图10-11 链接例子的依赖图

这种算法会先合并指令1和指令3。然后再将指令2和前两条指令合并起来得到图10-12所 539

示的图。

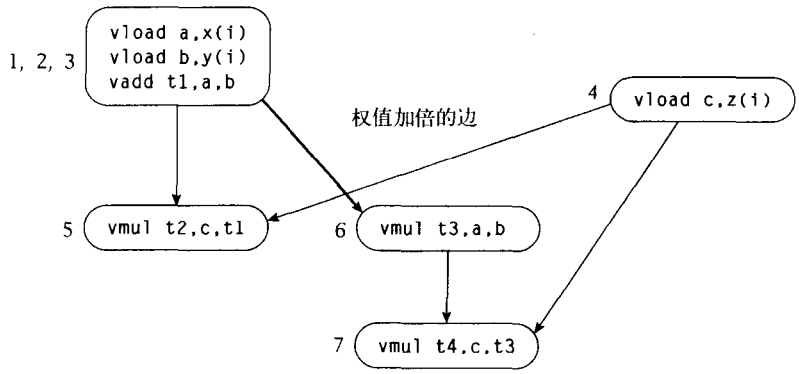


图10-12 部分链接后的依赖图

注意由于指令6有两个输入来自包含指令1、2、3的合并组，故该组与指令6之间边的权值在合并后重新计算，值增加一倍。这样就会跳过指令5，选择将下一条指令与第一组结合起来。从而用完了该组所有的资源。另外三条指令将被合并到第二组中，产生所需的结果。

10.3.2 协处理器

在许多特殊的应用中，通常“附加”一个协处理器来加快应用代码特殊部分的执行速度。例如，在大部分个人电脑中，图形协处理器分担主处理器的绘制和其他的基本图形计算，加快系统的运行速度。在浮点和整数处理器被集成到一个芯片上之前，通常会将向量和浮点协处理器连接到标准机器上。目前，现场可编程门阵列（FPGA）技术使动态构建专门针对某些计算的核心部分的协处理器成为可能。

在一定程度上，调度协处理器类似于调度VLIW或者超标量体系结构。协处理器只是另一个功能部件。比起那些细粒度机器上的典型的加法器或者乘法器部件，它具有更广泛的能力。一般来说，协处理器的功能是明确定义的，所以就很容易判断哪些代码应该转移到协处理器执行、哪些应该留在主处理器执行；拿图形协处理器为例，它一般是用来做图形操作的。然
540 而，有一点细微的差别使得协处理器的调度变得复杂。协处理器一般都是连接到系统总线上，主处理器通常通过“内存映射（memory-mapped）”的接口调用它们。也就是说，主处理器通过把一些特殊值存储到某些特殊的内存位置，来通知协处理器所需完成的工作；协处理器则通过系统总线读取这些值，以决定它需要完成的工作。协处理器只能在系统总线上访问主存储器并获取值，但是它能从主处理器得到的唯一的内部信息也就是主处理器发送给它的。

因为高速缓存被连接到中央处理器，协处理器没有办法看到它们。同样地，高速缓存通常也不知道协处理器的存在。结果，当协处理器存储一个计算的值到内存时，更新后的值可能不会立即反映到中央处理器的高速缓存上。同样地，中央处理器保存的值也可能不会立即出现在内存中，从而导致协处理器读取一个无效的值。解决这个问题的一种方法是提供一套特殊的内存同步指令，它可以保证内存与高速缓存一致。这样的指令暂停中央处理器，直到它发出的所有的写操作都已写入内存。另一种方法则将高速缓存置为无效，这样来自内存的新值就能被读取。不易发现的是，考虑到协处理器可能会在主处理器发出相关写操作之前发出读操作，但写操作却首先完成，这种情况还需要一条指令来强制暂停直到读操作在所有的

处理器上都完成为止。所有这些指令都是代价很高的，因此将它们的使用减至最少对于执行速度是至关重要的。

可以利用数据依赖来确定什么地方需要等待指令。当数据依赖的源点在一个处理器上被执行而汇点在另一个处理器上被执行时，两者之间就会需要某种形式的内存同步。如果是真依赖，主处理器必须保证在汇点处理器读内存之前，源点处理器上的所有存储操作已经完成。如果依赖是反依赖，主处理器必须保证汇点处理器在写操作发射之前，源点处理器上的所有读操作已经完成。如果是输出依赖，中央处理器必须保证第二个保存操作在第一个之后完成。由于这些等待指令是昂贵的，编译器必须最大限度地减少应用在指令操作中的等待指令的数量。操作的位置也是重要的，因为许多体系结构保证存数操作在给定的若干时钟周期内一定能写入内存。这样，如果源点和汇点能够用足够的其他指令分开，内存访问的正确性就可以无须借助等待指令而得以保证。

指令的位置之所以重要的第二个原因是：它能减少必要的等待指令的数量。考虑图10-13中的例子。第一个依赖能被在区域1或区域2中的等待指令覆盖，第二个依赖能被在区域2或区域3中等待指令覆盖。简单的作法会导致在区域1和区域3中都加入等待指令；更巧妙的作法是只在区域2中插入等待指令（恰好在A(I)的load之前）。

```
store A(I)
... region 1
store B(I)
... region 2
load coprocessor A(I)
... region 3
load coprocessor B(I)
```

图10-13 最小化生成的等待

在一个基本块之内，把等待指令减少到最低限度是简单的，而且可以在一遍中完成。从块的起点开始，依赖边的源点标注成被检查的指令。当遇到一条未覆盖的边的汇点时，就插入一条等待指令，而且因为该等待指令会保护所有的迄今为止其源点已遇到过的边，这些边将被标记为被覆盖的。因为所有边的源点和汇点都会被遇到，从而导致所有的边被覆盖，故这种算法可以保护基本块内的所有的边安全。

要弄明白这种算法怎样产生最少的等待指令并不困难。概要地讲它的证明方法就是，我们可以试图移去一条等待指令，并重新安排其他等待指令来覆盖已暴露的边。已安排好的等待指令的任何移动必然会导致另一条边变为不被覆盖。虽然这一算法产生的等待指令的个数最少，但是放置等待指令的方法不是唯一的，通常会有许多种方法。例如，一种对称的解决方案是从块的末端开始，向上遍历，当碰到边的暴露的源点时就插入等待指令。向下遍历的优点是使得有问题的访存操作和它的等待指令之间的时间间隔最大，从而减少在那些对存数操作有时间限制的体系结构中需要用到等待指令的可能性。

尽管在没有控制流的情况下我们可以用一种相当简单的算法产生最少数目的等待指令，而在有控制流的情况下使等待的数目达到最小是NP完全的。因此编译器必须使用启发式的算法产生好的解决方案。

在并行处理器中，也有处理器之间在访存上的同步问题，也同使用协处理器一样——对称的、共享内存的多处理器可被视为等价的协处理器的集合。然而在这样的系统中编译器的设计者通常不关心如何对访存操作进行显式调度。由于硬件设计者认识到这样的系统应当支持多处理器（他们并不见得总是了解协处理器），通常会在系统总线中提供支持，以避免许多类似这样的问题发生。此外，在进入和退出并行区域的时候通常会有栅障（barrier），这非常清楚地提示我们可以在这里插入同步操作。不过，已经提出了一些有关在并行循环中把所需的同步操作降低至最小限度的策略[62，214]。

541
~
542

10.4 小结

在本章中我们讨论了与调度有关的两个问题，这两个问题可以借助依赖分析的方法加以解决。

（1）单处理器上的*指令调度*涉及最大限度地减少在VLIW处理器或超标量处理器上执行一系列特定的操作所需的周期数。典型的作法是建立指令间的依赖图，并将调度算法用到该图上。在这一章介绍了三种不同的调度方法。*表调度*试图为的一块直线代码找到最佳的调度。*踪迹调度*扩展表调度，使之能够处理带有控制流的代码，该方法对称为踪迹的单一控制流路径进行调度，每次调度一条，并插入代码以解决接口问题。*核心调度*或*软流水*通过减少执行循环体的计算核心的周期数而把循环体的执行时间降低到最小限度，该核心在循环达到稳态后，执行循环体中的操作。

（2）*向量指令调度*试图把由于向量启动时间引起的延迟的影响降低到最小限度，方法是尽可能地重叠向量指令的执行。这一章介绍一种*向量链接*的算法，它使用加权合并算法的一个变形获得最大的重叠。

所有这些问题，即使在非常简单的假设条件下，也是NP完全的，因此本章所讨论的算法都是启发式的。

10.5 实例研究

由于PFC主要是一种源-源变换系统，不作任何指令调度。另一方面，Ardent Titan编译器则为RISC处理器和相连的向量部件生成代码。

对最初的Titan体系结构来说，最大的挑战之一就在于对向量和标量的浮点操作。因为在Titan 2000中的浮点部件是内存映像的异步协处理器，它也面对着10.3.2节中列出的那些挑战。编译器为了在向量部件和标量部件之间为向量和标量浮点指令同步内存访问，负责插入等待指令。编译器还必须正确同步同一浮点部件上取数和存数操作的内存访问。插入等待指令的算法基本上就是这章中所讲的那些算法。

最初决定采用在向量部件中执行标量浮点操作的方法并不容易，受到多方面因素的支配，
543 其中大部分涉及到硬件设计。在作出设计决定后，浮点部件和标量部件之间的同步就简化为只需考虑向量指令的情形，因为（假定）作为浮点数访问的数据不会作为整数访问。涉及到在循环外的标量浮点取数和存数操作的内存同步被认为极少发生，所以不会成为问题。

在协处理器上执行所有的浮点操作这一决定在多方面取得了成功。在循环内部，编译器清楚地了解控制流和数据流，很明显这一方法取得了成功。表10-1显示Titan编译器在Livermore循环上取得的结果，这些混有标量和向量计算的循环代表Livermore国家实验室中典型的计算的负载。在倒数第2列中是这些向量核心；在Titan编译器上，向量化获得了显著的加速比。

然而，表10-1也试图显示即使是标量核心也能在Titan系统上得到了显著的加速比。Titan编译器有一个选项支持在标量优化中调用依赖分析，但不作向量代码生成。标有“O1 Megaflops”和“O1 with Dependence”的列显示对每一个核心的标量优化（有依赖分析及没有依赖分析）所得到的性能。当依赖分析关闭时，主要的标量优化就是强度消减和协处理器指令调度，如10.3.2节中的介绍过。当没有依赖信息时，则只对标量的取数和存数把等待指令降到最低限度的优化。当有依赖信息时，这种优化方法就可扩展到数组变量。此外，编译器可以使用8.3节中介绍的标量替换技术把大部分内存访问提升为寄存器的读写，大量减少内存

访问需要的等待指令数。结果，所有的整数和浮点数部件都能全速异步运行，在无法向量化的核心中产生接近向量化的速度。

表10-1 Titan编译器利用依赖提高标量性能

核 心	*O1*有依赖	*O1*无依赖 (megaflops)	改进 百分比	可向量 化吗?	关键 优化
11	0.4514	1.3558	200.1	No	SR, IS
12	0.4397	1.2560	185.6	Yes	IS
5	0.7301	1.9633	168.9	No	SR, IS
10	0.5136	1.3211	157.2	Yes	IS
21	0.6787	1.6991	150.3	Yes	STR, IS
6	0.6894	1.7203	149.5	Yes	SR, IS
1	1.2589	2.1979	74.5	Yes	IS
13	0.2954	0.4237	43.4	Yes	STR, IS
23	1.8059	2.4598	36.2	No	SR, IS
2	1.1035	1.4349	30.0	Yes	IS
9	1.5206	1.7720	16.5	Yes	IS
20	1.6663	1.8809	12.9	No	SR, IS
7	2.3657	2.6227	10.9	Yes	IS
19	1.6792	1.9030	7.4	No	SR
14	0.4565	0.4775	4.6	Yes	IS
15	0.7601	0.7908	4.0	Yes	LI
18	1.8216	1.8262	0.3	Yes	
3	1.9713	1.9684	−0.1	Yes	
24	0.3825	0.3828	−0.1	Yes	
17	0.9663	0.9644	−0.1	No	
22	0.4924	0.4914	−0.2	Yes	
4	1.8936	1.8873	−0.3	Yes	
16	0.8522	0.8479	−0.5	No	
8	2.0733	1.9089	−7.9	Yes	
均值	1.1195	1.4773	32.0		
几何平均	0.9346	1.2959	38.7		
调和平均	0.7724	1.0786	39.6		
中值	0.9093	1.7097	88.0		

对因使用依赖信息而取得显著改进的核心而言，最后一列表示对这种改进起主要作用的优化方法，这里的“SR”代表标量替换，“IS”代表指令调度，“STR”代表强度消减，“LI”代表循环不变量识别——一种可以消除循环中在每次迭代中执行相同计算并把结果存储到相同内存地址的语句（使用这种优化方法的概率相当高）。

尽管有这些结果，同步问题仍然是Titan系统中的一个大问题。认为数据只会以整数和浮点数的形式被访问的想法是错误的——特别是数学库中，浮点数一般都被分解为指数和尾数。同时，在循环外的标量浮点数读写之间的同步化对于许多代码也是一个大问题。如果没有循环的依赖分析，代码生成器基本上只能假定所有浮点数访问都是有依赖的，因此而导致插入的等待指令会极大地降低程序性能。另一个在一开始并没有引起足够重视的问题是对遗漏必要的等待指令的程序进行调试所需要的时间和人力。这样的程序的行为很大程度上依赖于取

数操作和内存的情况，其运行十之八九都是正确的。要正确的插入等待指令需要开发大量工具以支持调试，找出在已编译好的程序中遗漏的等待指令。

由于上面提到的这些问题以及硬件设计的改变，第二代Titan系统结构——Titan 3000在整数部件中进行标量浮点化，因而只需要用等待指令来对向量指令作同步。结果得到了一个效率极高的编译器和硬件组成的系统，正如表10-1中的Livermore循环的测试结果所表明的。既然只有向量循环需要等待指令，其中编译器完全了解循环中控制流和数据流，所以这些循环核心的测试结果都不错。

544
~
545

Titan编译器为调度指令实现了另外两个有趣的优化。一个是Titan内部的内存体结构特别需要的。Titan对浮点运算有强的支持，其存－取体系结构按64位分体，适合执行64位操作。由于整数只有32位长，在Titan系统中进行连续的向量整数存取是不好的做法，因为在进行一对整数的第二个数的存取同时，内存体却依然还在忙于对第一对整数的存取。结果，硬件就会停止，直到内存体完成操作，这样就造成连续向量整数的存取速度比等价的双精度存取慢3-5倍。编译器解决这个问题的方法是把有连续整数操作的向量循环分为包含跨距为2的操作的两个循环，这种分法是安全的。根据循环中的依赖，容易确定这种分法的安全性。当这种分法可行时，可以得到向量整数代码中的3倍的加速比。

第二种优化有着更加广泛的可应用性，它涉及到归约操作。对Titan和其他大多数的支持向量归约操作的系统结构来说，归约的结果被存入一组累加器。在归约开始之前，需要对累加器赋恰当的初值，在归约结束的时候，还需要把归约结果移出累加器。

Titan编译器的经验证实依赖分析的价值，依赖作为总结执行约束的工具，以及作为最常在程序中被引用的内存位置的指示器。这种分析的价值早在向量机和并行机时代就被认识到了。另外，Titan系统结构的这些概念对较传统的机器（具有浮点协处理器、非同步内存访问和较深的存储层次结构）也是有价值的。鉴于具有类似特征的新型微处理器设计的流行——如Sony Playstation 2和PowerPC G4（这里仅举两个），很显然，依赖在可预见的未来仍将是一种有用的工具。

10.6 历史评述与参考文献

表调度很久以前就已出现在计算机科学文献中。它在微码压缩、指令调度和任务调度中被研究过。这方面的工作在Fisher的博士论文[113]中有很好的总结，这篇论文也提出10.2.3节中介绍的最高级优先的启发式方法HLF（highest levels first）。Adam，Chandy和Dickson[2]也讨论过HLF。Touzeau[258]，Gibbons和Muchnick[120]，Warren[268]讨论了利用表调度的基本块调度程度。Garey和Johnson[119]在其书中证明表调度是NP完全问题。Muchnick[218]在其书中提出他对调度策略的非常卓越的看法。

546

Fisher[113，114]提出了踪迹调度。基于Ellis在Yale大学Fisher的指导下所完成的博士论文[109]，该方法在Mutiflow编译器第一次成功地得到商业化的实现。

核心调度，也被称为*软流水*[72]或者*有环调度*，最初是由Rau和他的同事基于Davidson和其他人[98，99，100]在硬件流水线设计方面所作的一些早期研究工作而提出的[232，233]。最初对这个问题的形式化包含了最小迭代间隔的概念，这一概念建立在由资源约束和数据依赖环导致的下界的基础上。这些思想其后在Hsu[153，154]、Lam[193，194]以及Su和Wang[255]中得到提炼。Hsu和Lam独立地证明了对有资源约束和任意依赖环的循环的调度问题是NP完全问题[153，193]。

Rau和Fisher[231]对指令级并行和软件调度方法作了非常好的总结。

习题

10.1 构造一个例子简单的表调度例子，采用支持预先计算关键路径的最高级优先（highest levels first）的启发式方法的改进，产生次优的结果。

10.2 指令调度在代码生成后的汇编代码级进行。为什么在源代码级进行调度而后生成代码不是一种好的想法？

10.3 对数组访问的依赖分析需要知道数组引用的形式和与它们相关的循环嵌套。没有这些源级的信息，你怎样决定内存存取间的数据依赖关系？不精确的依赖分析会带来什么样的问题？

10.4 核心调度需要找出一个循环中的所有依赖环的最大斜率。图10-8中的算法用一种求最短路径的算法。它是怎样求最大斜率的？你能想出另外一种求最大斜率的方法吗？你能用一种有效的方法求出有最小斜率的依赖环吗？这些方法的复杂度如何？

10.5 对带有多级高速缓存的现代处理器，内存引用的时延与数据是否在高速缓存上和在哪一级高速缓存上有关。如果在调度中不考虑不同的内存时延，那么会出现什么问题？编译器如何确定内存引用的可能的时延？

10.6 当内存操作的时延很小时，指令调度的效果会很好。如果时延较大，例如从内存取数超过200个机器周期，那么它的效果依然很好吗？你有什么补救措施？ 547

第11章

过程间分析和优化

11.1 引言

由于机器和语言变得越来越复杂，编译技术必须也变得越来越复杂。随着向量机和并行机的出现，只分析单个过程不能再适应产生高质量并行代码的要求。本章我们介绍借助于过程间分析和优化系统能够解决的一些问题。我们也要介绍解决这些问题的方法，并总结过去15年来过程间编译方面的研究成果。

我们首先从术语“过程间分析”和“过程间优化”的定义开始。*过程间分析*是指在整个程序范围内而不是只在单个过程内收集信息。此处的信息和单个过程的数据流分析收集到的信息类似。过程间分析问题的例子有：找出由于过程调用的副作用而被修改的变量和确定一对变量在给定过程入口处是否可能互为别名。

*过程间优化*是涉及一个程序中多个过程的程序变换。最熟悉的过程间优化的例子是*内联*(inling)，通过内联过程的调用点用过程体替代。尽管过程间优化一般会修改多个子程序，但把任何基于过程间信息（由过程间分析阶段收集）的优化看作是过程内优化也是合理的。但是，本章中我们仍然坚持使用其较窄的范畴。

11.2 过程间分析

程序设计中使用过程的价值在于过程可以对程序员隐藏一些不必要的细节。这样做的一个必然结果就是它也对编译器隐藏了一些细节。但是，既然这些细节可能提供一些优化的机会，这些细节就可能不是无关紧要的了。过程间分析的目标是充分地重新找出有用的细节来支持更进一步的优化策略。

11.2.1 过程间问题

为了说明过程间分析方法的必要性，我们将通过一系列例子来介绍几个重要的问题。

修改和引用的副作用

我们从一个简单的向量化问题开始。假设有如下的代码段：

```
      COMMON X, Y
      ...
      DO 100 I = 1, N
S0      CALL S
50      X(I) = X(I) + Y(I)
100 CONTINUE
```

如果没有特定的过程间优化，我们不可能向量化其中的过程调用，但是，我们是否可以向量化赋值语句50呢？由于X和Y都在COMMON块里，所以我们必须考虑在S_0处的过程调用语句对这些变量的副作用。如果将包含语句S_0和50的循环进行循环分布是合法的，则语句是可以被向量化的。进而，如果不存在涉及两个语句的依赖环，这就是可能的。如果这个过程调用同时

满足以下两个条件，我们可以确信不存在这样的依赖环：

（1）过程调用既没有修改X也没有使用X。

（2）过程调用没有修改Y。

第一个条件保证不可能有涉及到过程调用和语句50中X的依赖环。第二个条件排除涉及到Y的依赖环。

550 为了讲明这个问题，我们介绍过程间修改（MOD）和引用（REF）的问题。

定义11.1 在给定的过程调用点S，*修改副作用集*MOD(S)是指由于S处的过程调用的副作用而可能被修改的所有变量的集合。*引用副作用集*REF(S)是指可能由于S处的过程调用的副作用而可能被引用的所有变量的集合。

有了这个定义，我们能够形式化地重新描述以上赋值语句可以向量化的条件，即

$X \notin REF(S_0)$且$X \notin MOD(S_0)$且$Y \notin MOD(S_0)$

别名分析

假设我们有以下子程序：

```
    SUBROUTINE S(A, X, N)
      REAL A(*), X, Y
      COMMON Y
      DO 100 I = 1, N
S0      X = X + Y*A(I)
100   CONTINUE
    END
```

为任何机器编译这个循环时，如果在整个循环的过程中把X和Y保存在寄存器里，并在循环结束之前不把X存回内存，会有好的效果。这看起来很简单，但是如果子程序S的以“CALL S(A,Y,N)”的形式被调用，情况又将如何呢？这种情况下，Y是X的*别名*，所以如果不在每次迭代中存储X，我们就可能忽略需要在循环内更新变量Y[⊖]。

为了避免类似这样的问题，编译器必须确定在子程序入口处两个变量是否互为别名。

定义11.2 对于过程P和传递给P的一个形式参数X，*别名集合*ALIAS(S, X)是指在S的入口点和X指向同一内存位置的变量集合。

551 在上面的例子中，如果Y ∉ ALIAS(S,X)，则X和Y可以被保存在寄存器里而不用存入内存。

调用图的构造

程序的调用图是模拟程序中过程之间调用关系的一种图。

定义11.3 程序的*调用图*是指图$G=(N, E)$，其中N中的结点代表程序中的过程，E中的边代表可能的调用。

程序中每一个调用点通常需要用图中一条单独的边来表示，这种情况下，调用图实际上

⊖ 有见识的读者会说，Fortran标准指明这种用法是非法的，即如果两个变量在循环的入口处互为别名，那么子程序对其中的任何一个写入数值是不符合规范的。虽然这种规定可以容易地解决上述的特殊问题，但这只出现在Fortran中，因为C语言中没有这种限制。

是一个多重图。我们将在本章的剩余部分也遵守这个约定。

调用图的准确性直接影响到所产生的数据流信息的精确性。但是构造一个精确的调用图本身就是一个过程间分析问题。对于每次调用都必须是带名字的恒定过程的语言，构造调用图十分容易——你只需检查每个过程p的过程体，为其中每个过程调用点s加入一条从p到在s处被调用过程q的边(p, q)。

但对于一种允许过程变量的语言，构造精确调用图就困难得多了。即使在Fortran中，程序中不允许有可赋值的过程变量，由于过程参数的存在仍然会导致问题——形式参数可以在调用点处和过程名绑定。考虑如下例子:

```
      SUBROUTINE S(X, P)
S0    CALL P(X)
      RETURN
      END
```

这里的问题是哪个过程或哪些过程可能在语句S_0处被调用？换句话说，在这个调用点P具有什么值？我们可以通过检查调用子程序S的所有调用点处的实际参数来试图解答这个问题，但是任何的实际参数本身可能又是一个过程参数。这样，我们就必须在构造好调用图之前在调用图上传播过程常数。

定义11.4 对于给定的过程p和p中的调用点s，调用集CALL(s) 是指可能在s处被调用的所有过程的集合。

虽然调用集是作为调用点的一种属性提出，但是，CALL(s) 不是一个副作用问题。它实际是在探求在包含s的过程入口处，哪些过程将被传给形式参数。由于这个问题依赖于包含s的过程被调用处的上下文，所以与其他副作用问题相比，该问题更类似于别名分析。 552

活跃分析和使用分析

一个已经在文献中得到广泛研究的重要的数据流问题是活跃变量的分析。如果存在一条从s点到x的一个使用点的控制流路径，且这条路径上在x的这个使用之前没有对x的定义，就说变量x在程序中给定点s是活跃的。

活跃分析的一个重要应用是判断在一个并行循环执行的末尾处，是否需要把一个循环内的私有变量赋给一个全局变量。考虑如下代码段:

```
DO I = 1, N
   T = X(I) * C
   A(I) = T + B(I)
   C(I) = T + D(I)
ENDDO
```

把T作为一个循环内的局部变量，这个循环可以被并行化。但是，如果T在后面程序中的使用是在它被重定义之前，那么这个并行化程序必须把局部变量T的最后的值赋给一个T对应的全局变量。换句话说，代码必须被转换成:

```
PARALLEL DO I = 1, N
   PRIVATE t
   t = X(I) * C
   A(I) = t + B(I)
   C(I) = t + D(I)
```

```
      IF (I .EQ. N) T = t
   ENDDO
```

这里我们有一个印刷上的约定，所有小写字母表示的变量都是由编译器引入的变量。

如果，我们能断定变量T在原始循环的出口处不活跃，那么这段代码可以被化简。在这种情况下，循环结尾处的条件语句可以被删除，于是代码变为

```
PARALLEL DO I = 1, N
   PRIVATE t
   t = X(I) * C
   A(I) = t + B(I)
   C(I) = t + D(I)
553 ENDDO
```

虽然，活跃分析本身可以看作是过程间问题，但是用另一个过程间分析问题来处理它也是方便地。使用分析问题是判断在一些通过某过程的某个特定调用点s的路径上，一个变量是否存在一个向上暴露使用。向上暴露使用是指从过程调用发生到这个使用点的某条路径上，在这个使用之前不存在对变量的定义。

定义11.5 对调用过程p的给定调用点s，使用副作用集USE(s) 是指在p中有向上暴露使用的变量集合。

有了这个定义，我们可以对一个变量何时活跃的问题给出一个更加形式化的说明。如果$x \in$USE(s)，或者存在一条通过s处过程调用的路径并在此路径上没有给x赋新的值，且x在s的控制流后继处活跃，则变量x在调用点s处是活跃的。

注销分析

REF，MOD和USE这类问题是问：在通过被调用子程序的某条路径上可能会发生什么。另外，提问在通过一个过程的每条路径上必定发生什么的问题也是十分有用的。下面的例子可以说明这点：

```
L    DO I = 1, N
S0      CALL INIT(T, I)
        T = T + B(I)
        A(I) = A(I) + T
     ENDDO
```

这里有两个问题可能会阻碍这个循环被正确地并行化。首先，我们不知道子程序INIT将会做什么，所以必须假设它对变量赋值的方式上将会造成依赖环。对程序的一个全局变量先使用后赋值就是造成依赖环的一种方式。例如，如果INIT被定义为如下代码，就不能被并行化了：

```
SUBROUTINE INIT(T, I)
   REAL T
   INTEGER I
   COMMON X(100)
   T = X(I)
   X(I+1) = T + X(1)
554 END
```

这里，从调用的程序中的循环的角度看，INIT造成一个含有COMMON数组X的依赖环。

但是，即使我们能证明这个循环不会修改任何全局变量或者甚至是任何静态局部变量（即“SAVE”变量），过程调用仍然会给我们带来更为微妙的问题。如果这个循环被并行化，就必定可能识别出变量T相对于每次迭代都是私有变量。这种情况是可能的，例如，如果子程序INIT只是为了初始化T，就像在下面的代码中那样：

```
SUBROUTINE INIT(T, I)
   REAL T
   INTEGER I
   COMMON X(100)
   T = X(I)
END
```

这里的T被循环的每次迭代使用之前都被初始化。这样，它就不会和循环内任何循环携带的依赖有关，于是它可以被私有化。

但是，我们如何才能发现这个事实呢？当然，MOD可以告诉我们子程序内没有改变过任何全局变量，因此可以假设类似的分析可以排除INIT中任何静态局部变量在修改之前被使用的可能性。所以，决定这个循环是否可以被并行化的关键在于证明，变量T在通过INIT的*每条路径*上被使用之前先被赋值。

判断一个变量是否在通过一个过程调用的每条路径上都被赋值的问题称为KILL（注销）问题，因为一个变量的赋值将“注销”变量先前的值。

定义11.6 对于给定的过程调用点s，*注销副作用集*KILL(p) 是指通过在s处调用的过程p和在p之内调用的过程的每条路径上都被赋值的变量的集合。

假设在集合MOD(p) 内没有全局变量，且p没有在赋值之前使用任何静态局部变量，那么，如果变量T在通过S_0处调用的过程的每条路径上都被注销，且不存在任何通往过程内部的路径，在此路径上T在注销之前被使用，循环L就可以被并行化。这一点可以形式化地表示为

$$T \in (KILL(S_0) \cap \neg USE(S_0))$$

假设调用点s是在一个单独的基本块内，在包含s的过程中，我们可以定义LIVE(s) 如下：

$$LIVE(s) = USE(s) \cup (\neg KILL(s) \cap \bigcup_{b \in succ(s)} LIVE(b)) \qquad (11\text{-}1)$$

555

如果我们通过额外分析知道过程出口的活跃变量集合，以上的公式将活跃变量的计算扩展到过程间的情况。

常数传播

常数传播是单过程数据流分析中最重要的问题之一，同时也是一个重要的过程间问题。考虑下面的从LINPACK代码中抽出来的例子：

```
      SUBROUTINE S(A, B, N, IS, I1)
         REAL  A(*), B(*)
L        DO I = 0, N-1
S0          A(IS*I+I1) = A(IS*I+I1)  + B(I+1)
         ENDDO
      END
```

如果我们想向量化这个子程序中的循环L，这里存在一个由于变量IS可能为0值所产生的问题。如果IS的值为0，就存在一个输出依赖。这样，语句S_0实际上是一个归约操作，不能用

常规的技术去向量化它。虽然，可以在运行时测试这种情形的出现，但如果我们可以在包含子程序S的程序中每个S的调用处判断出IS的值总是为1（最常见的情况）[⊖]，那么可以完全避免处理这种情形。

定义11.7 给定一个程序和程序中的过程p，过程间常数集合CONST(p) 包含每次调用p时值为已知常数的变量。对于一个变量$x \in$ CONST(p)，$valin(x, p)$ 是一个返回x在p入口处的值的函数。

虽然单过程的常数传播问题是比较难处理的，但是，已经证明单过程的近似解决方案是比较有效的[237]。同样，过程间常数传播的近似解决方案也能帮助判定很多对优化和并行化有用的事实[127，136]。

11.2.2 过程间问题分类

这里我们将探讨过程间数据流问题的各种分类。这些分类是十分有用的，因为同一类的
556 问题可以用同样的算法过程来解决。

可能问题和必定问题

对于询问某些事件是“可能”发生还是“必定”发生的问题，我们已经看到了问题之间的差异。MOD，REF和USE是可能问题，因为它们分别计算出的是可能被修改的、可能被引用的和可能在定义前被使用的变量的集合。相反，KILL是必定问题，因为它计算出的是必定被注销的变量的集合。

虽然，这些差异在文献中已经得到了广泛的讨论，但并不深入，因为每个可能问题的逆问题就是一个必定问题，反之亦然。例如，¬MOD(s) 是所有不在MOD(s) 中的变量的集合。所以，¬MOD问题是寻找那些必定不会在一个给定的调用点被作为调用的副作用所修改的变量。这样，¬MOD是一个必定问题。同样，¬REF也是一个必定问题。相反，¬KILL问题是为程序中每个调用点s寻找那些在被调用的过程执行中可能不被修改的变量集合，所以，必定问题KILL的逆问题是一个可能问题。由于任何集合问题的解可以从全集中相减（通常是位向量的补）转换为逆问题的解，这样做所需的时间与解集合数目呈线性关系，所以，必定问题和它对应的可能问题不存在复杂度的差别。

流敏感问题和流不敏感问题

Banning[37]给出了关于流敏感问题和流不敏感问题的一种看起来相关的概念。从直觉上讲，流敏感问题需要跟踪被调用子程序体中的不同控制流路径来求解。另一方面，求解流不敏感问题则可以只检查被调用子程序体，而不必考虑控制流。这样，MOD和REF问题就是流不敏感问题，这是因为假设子程序中的所有代码都是可达的，对子程序p的过程体中任何一个变量x的修改都意味着，对于任何调用p的调用点s来说，$x \in$MOD(s)。但是，KILL是一个流敏感问题，这是因为对于一个给定变量x，它的解需要检查通过过程的每条路径以确认在这个路径上包含一个对x的定义。

Banning[37]给出流敏感问题和流不敏感问题更加形式化的定义，这个定义基于将调用图的子区域中的解组合到更大区域的解中。考虑图7-11中描述的两个调用图区域。

⊖ 除了这种情况外，如果我们可以证明每次进入循环时IS≠0，那么也可以向量化。类似这样的谓词分析也可以用常数传播的变形形式来处理。

假设我们已知区域R_1和R_2的子区域A和B的各自MOD集合。那么，我们可以把这两个集合以如下的方式组合得到整个区域的解:

$$MOD(R_1) = MOD(A) \cup MOD(B)$$
$$MOD(R_2) = MOD(A) \cup MOD(B)$$

557

这样组合的原因是：对于上面任何一种子图分解方式，在任一子区域中修改某一变量，那么在整个区域内这个变量就有可能被修改。对于REF问题，也有类似的一对等式:

$$REF(R_1) = REF(A) \cup REF(B)$$
$$REF(R_2) = REF(A) \cup REF(B)$$

现在针对KILL解决同样的问题。在区域R_1的情形中，一个变量只要在第一个子区域被注销或在第二个子区域被注销，那么这个变量在整个区域被注销:

$$KILL(R_1) = KILL(A) \cup KILL(B)$$

对于区域R_2的情形，一个变量只有在两个子区域中都被注销，它才能在$KILL(R_2)$中:

$$KILL(R_2) = KILL(A) \cap KILL(B)$$

USE的等式很容易得到:

$$USE(R_1) = USE(A) \cup (\neg KILL(A) \cap USE(B))$$
$$USE(R_2) = USE(A) \cup USE(B)$$

其中第二个方程式同MOD和REF的对应等式一样。

察看这些等式，我们注意到：在MOD和REF的等式中只使用集合“并”作为连接符，而KILL和USE的等式更加复杂一些，用到其他连接符且通常用到其他局部集合。这就使我们有了以下的定义:

定义11.8 过程间数据流问题是*流不敏感*的，当且仅当，对问题的无参形式而言，在以串行方式和选择方式组合的区域（如图11-1的R_1和R_2）中，解的值都由子区域的解的并集得到。

优化方面的文献有时会涉及对流敏感问题的*流不敏感分析*。我们对这个术语的解释是用流不敏感问题的解逼近流敏感问题的解。举一个例子，假设你想要用一个和多个流不敏感问题的解逼近USE问题的解。请注意，当知道一个变量不在一个过程p的$USE(p)$（过程p被正在被优化的过程调用）中时，大多数优化才能进行。所以，我们希望近似结果APUSE是保守的，即包含USE集合中所有的元素:

$$USE(p) \subseteq APUSE(p) \text{ 或 } \neg APUSE(p) \subseteq \neg USE(p)$$

在这种情况下我们永远不基于不真的命题。一个可能的近似由

$$APUSE(p) = REF(p)$$

给出，这个集合很明显是$USE(p)$的一个超集，可以通过解一个流不敏感问题得到。我们将会看到，这样做可能是有用的，因为流不敏感问题比流敏感问题容易求解。但是，我们并不推

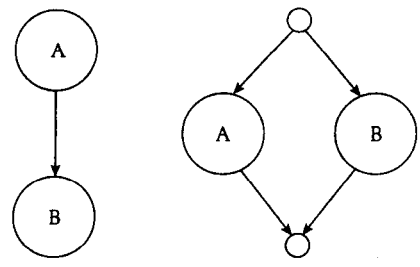


图11-1 调用图子区域

荐使用像这样的近似，因为使用流敏感分析能够在合理时间内得到更加精确的近似。

副作用问题和传播问题

过程间问题的最后一种分类是按照数据流的方向分类。一些过程间问题的是问：作为一个过程调用的副作用，什么事情可能发生或必定发生？我们称为副作用问题的这类问题包括MOD，REF，KILL和USE问题。这类问题类似于单过程分析中的后向数据流问题。

第二类问题是问：什么条件在当前过程入口处可能成立或必定成立（这些条件多半是我们在优化中所关心的）。我们称这些问题为传播问题。这类问题包括ALIAS，CALLS和CONST问题。在副作用问题中用到的把问题分为流敏感和流不敏感的方法不适用于传播问题，因为这类问题需要后向查看调用链。但是，通常认为Fortran中的别名问题是流不敏感问题，因为只有引用形式参数才会引入别名。这样，通过一个子程序到一个调用点的流信息是不重要的，因为别名只能在过程调用处引入。

558 ~ 559

另一方面，Fortran中的常数传播问题是流敏感的，因为对于传入某个被调用过程的一个常数，相应变量必须在通过调用链上每个过程的每条路径上都得到同样的常数值。基于同样的理由，有指针赋值的语言（如C语言）中的别名分析也是流敏感问题。

表11-1总结两维分类问题。将问题分类为流敏感和流不敏感是很重要的，因为Banning给出了一个算法，证明了流不敏感问题可以在调用图的大小的多项式时间内求解[37]，而Mayers证明了流敏感问题在有别名和任意嵌套的情况下是很难解决的[220]。

表11-1 过程间问题分类

问题类型	传播	副作用
流敏感	ALIAS, CALLS	MOD, REF
流不敏感	CONST	KILL, USE

11.2.3 流不敏感副作用分析

我们现在开始讨论流不敏感副作用分析问题的解。在本节中，我们用修改副作用分析（MOD）问题作为例子。引用副作用分析（REF）问题可以用完全相同的方法求解。

一些假设

首先，我们建立一些代表Fortran特点的假设，在某些情形中，也代表了C中的特点。首先，我们假设没有过程嵌套。就是说，变量被分为局部变量集合和全局变量集合。局部变量只在本过程中可见，而全局变量对每个过程都可见。虽然这看起来可能比较严格，但是这里介绍的方法很容易扩展到通常的嵌套情况下。

已经隐含的第二个假设是所有的参数都以引用方式传递并且没有指针变量。这个限制是为了简化各个算法必须处理的别名模式。在这个限制下，别名只会在过程调用点引入。虽然，Fortran 90和C中都有指针变量，但这个假设对Fortran 77是成立的，而Fortran 77是大多数自动并行化系统的输入语言。

虽然Fortran 77不支持递归调用，但这里所讲的算法在有递归过程调用的情况下仍是正确的。但是，我们做了一个关于过程的参数表大小的假设，即一个过程形式参数的最大数目不随程序的大小而增长。换句话说，程序员一般不通过增加过程接口的复杂性来处理大程序增加的复杂度。这样，我们将假设存在一个常数μ，任何过程p的形式参数数目都小于等于μ。

560

MOD问题陈述

这个问题的目标是给程序中每个过程调用点s计算MOD(s) 集合，这个集合包括可能作为s处的过程调用的副作用而被修改的每个变量。首先要注意的是，我们可以通过忽略其中潜在的别名来简化问题。特别，我们将计算*直接修改副作用集*DMOD(s)，这个集合包括所有在s处可见的作为调用的副作用而被直接修改的变量。DMOD(s) 可能比MOD(s) 小，因为它没有考虑过一个被副作用所修改的变量x在调用点处有几个可能的别名的情况。如果解是精确的，所有这些别名都应当在最后的MOD(s) 中。但是一旦DMOD(s) 被计算出之后，就可以借助于11.2.1节中介绍的别名集对其进行更新，从而得到MOD(s)。

前面提到，对于给定过程p，ALIAS(p, x) 包含所有在p的入口处可以作为x别名的变量。有了这些集合，我们可以按照如下公式把DMOD(s)更新为MOD(s):

$$\mathrm{MOD}(s) = \mathrm{DMOD}(s) \cup \bigcup_{x \in \mathrm{DMOD}(s)} \mathrm{ALIAS}(p, x) \tag{11-2}$$

其中p是包含调用点s的过程。ALIAS(p, x) 的构造将在11.2.4中讨论。

我们将为每个调用点计算DMOD(s)，首先为程序中每个过程p计算称为*综合修改副作用*的集合GMOD(p)。GMOD(p) 包含作为p被调用的结果而直接或间接被修改的全局变量和形式参数的集合。一旦有了这些集合后，我们可以通过以下公式计算DMOD(s):

$$\mathrm{DMOD}(s) = \{\{v \mid s \text{ 调用了 } p,\ v \xrightarrow{s} w \text{ 且 } w \in \mathrm{GMOD}(p)\} \tag{11-3}$$

其中$v \xrightarrow{s} w$ 表示在调用点s处实在参数v被绑定到被调用过程的形式参数w或v和w是对同一全局变量的引用。(换句话说，在这个公式里全局变量可以看作是被调用过程的一类参数。)

下面通过一个例子来说明DMOD和GMOD的定义。如果，我们有一个调用点

```
S0   CALL   P(A, B, C)
```

561

其中子程序P被定义为

```
SUBROUTINE P(X, Y, Z)
   INTEGER  X, Y, Z
   X = X * Z
   Y = Y * Z
END
```

那么，GMOD(P)={X,Y}和DMOD(S_0)={A,B}。

等式 (11-3) 把DMOD问题化简为对程序中每个过程p计算可能作为调用p的副作用而被修改的变量的集合GMOD(p)。通常来讲，GMOD(p) 包含两类变量:

(1) 在p的过程体中被显式修改的变量;

(2) 在p的过程体作为某个调用过程的副作用而被修改的变量。

如果*直接修改副作用集*IMOD(p) 代表p中显式修改的变量集合，那么以下公式成立:

$$\mathrm{GMOD}(p) = \mathrm{IMOD}(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w \text{ 且 } w \in \mathrm{GMOD}(q)\} \tag{11-4}$$

注意并集计算包括p中所有的调用点s。

问题分解

等式 (11-4) 中的数据流方程组可以用数据流分析的迭代法求解。但是，得到这个解需要比较长的时间，因为这个方程组不满足迭代法快速收敛的条件[164]。它甚至不能表示成可以使

用快速消去法的形式[125]。要快速得到解，就必须限制分析过程中遍历程序流图中循环的次数。对于流不敏感的过程间分析，问题图就是调用图，因而收敛问题就局限在递归调用的区域内。

进一步察看带有递归的问题发现这和引用形式参数有关。考虑下面的例子：

```
SUBROUTINE P (F0, F1, F2, ..., Fn)
   INTEGER X, F0, F1, F2, ..., Fn
   ...
S0 F0 = some expression
   ...
S1 CALL P(F1, F2, ..., Fn, X)
   ...
END
```

从这个例子中可以看出，为什么在一般情况下分析过程必须沿递归环迭代不确定的次数。要回答的问题是：子程序P有多少个参数将由于调用P的副作用而被修改？很明显，F_0会在语句S_0处被修改，但是我们必须检查在S_1处的递归调用，发现F_1被传给F_0，所以它也会被修改。沿递归循环再迭代一次发现F_2也可以被修改。迭代过程一直会持续到我们发现最后一个参数F_n也会被修改。如果n为不确定大小，那么递归环上的迭代也会进行不确定的次数。

这些观察清楚说明为什么我们对所有子过程要假设一个参数数目的上界——它使我们建立一个迭代次数的常数上界，这是迭代过程收敛所必需的。但是，我们可以通过对问题的进一步分解而得到更好的时间上界——即分别处理引用参数的副作用和全局变量的副作用。为做到这一点，先要引入直接修改副作用集IMOD(p)的一种扩展形式，称为IMOD$^+$(p)，这个集合除了包括所有IMOD(p)中的变量外，还包括所有由于从p中调用过程的形式参数引用的副作用而被修改的变量。换句话说，当以下条件之一成立时，变量x在IMOD$^+$(p)中：

(1) $x \in \text{IMOD}(p)$

(2) $x \xrightarrow{s} z$ 且$z \in \text{GMOD}(q)$，其中$s=(p,q)$且z是q的形式参数

如果我们能为程序中每个过程计算IMOD$^+$(p)，那么我们就可以用下面简单的方程组求解GMOD(p)：

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup \bigcup_{s=(p,q)} \text{GMOD}(q) \cap \neg \text{LOCAL} \tag{11-5}$$

其中LOCAL表示程序中所有的局部变量，所以它的补集是所有全局变量的集合。由于所有关于引用形式参数的副作用都反映在IMOD$^+$(p)，我们只需要确认其后继的GMOD集的并反映了所有关于全局变量的副作用。如果一个全局变量由于调用p而被修改，它必定属于以下三中情况之一：(1) 在子程序体中被直接修改，这种情况下，这个变量属于IMOD(p) $\subseteq$ IMOD$^+$(p)；(2) 作为实际参数传给了另一个子过程，而在另一个子过程中被修改，这种情况下，根据定义这个变量属于IMOD$^+$(p)；(3) 被过程p直接或间接调用的子程序作为全局变量修改，这种情况下，
563 这个变量属于p的某个后继q的GMOD(q)集合。由此证明公式(11-5)。

现在，我们已经把问题分解成了两部分：

(1) 为程序中每个过程p计算IMOD$^+$(p)。

(2) 按照等式(11-5)传播全局变量的修改。

我们将在以下两个小节中介绍这两部分的计算。

计算IMOD$^+$

下面我们建立一个特殊的数据结构，称为绑定图：

（1）为程序中每个过程f的每个形式参数构造一个结点。

（2）如果过程p的形式参数f_1在调用点$s=(p, q)$处被绑定到过程q的形式参数f_2上，我们构造一条从f_1到f_2的有向边。

我们可能问的一个很直接的问题是：绑定图会有多大？如果N是调用图中的结点的数目，E是边的数目，那么绑定图的结点的数目不会大于μN，其中μ是程序中任意过程的参数数目的上界。因为整个程序的绑定图中不可能有多于形式参数数目的结点，而形式参数数目是以μN为上界的，所以以上命题是正确的。同样，由于在一个独立的实参的位置至多只能出现一个形式参数的引用，所以由调用图中的每条边构造出的（绑定图）边不可能多于μ条。这样，绑定图中的边的总数就不会大于μE，绑定图也不会大于调用图大小的某个常数倍数，即它的大小是$O(N+E)$。

令RMOD(p) 表示在过程p中通过直接修改或由于被赋给被p调用的过程q的引用形式参数而被修改的p的形式参数集合。显然，下面的等式成立：

$$\mathrm{IMOD}^{+}(p)=\mathrm{IMOD}(p)\cup\bigcup_{s=(p,\,q)}\{z\mid z\xrightarrow{s}w\ \text{且}\ w\in\mathrm{RMOD}(q)\}\tag{11-6}$$

这个等式与等式（11-4）类似。这样，对程序中每个过程我们只需要构造RMOD(p)，然后应用等式（11-6）构造IMOD^{+}(p)。

我们利用一个在绑定图上的简单标记算法来构造RMOD(p)，这个算法中，绑定图中每个结点都附加用一个位实现的逻辑标记。最初，所有这些逻辑标记都为假，然后对于任何过程p，对于在IMOD(p) 中的p的任何形式参数的标记被置为真。为真的位会在图中传播，规则是形式参数f_1被绑定到标记为真的形式参数f_2上，f_1的标记也会被置为真。当不再有传播发生时，RMOD(p) 就是p中所有标记为真的形式参数的集合。算法在图11-2中给出。 564

```
procedure computeRmod(P, N_B, E_B, IMOD, proc, RMOD)
    // P包含程序中的所有过程
    // N_B包含绑定图中所有的形式参数
    // E_B包含绑定图中所有的边
    // mark[f]是从形式参数到它们的标记值的映射
    // proc[f]是从参数到它所属的过程的映射
    // IMOD[p]是从一个过程到它的直达修改副作用集的映射
    // RMOD[p]包含所有的输出集合
    // worklist是形式参数的一个工作集
L1: for each f∈N_B do mark[f] := false;
    worklist := Ø;
L2: for each f∈N_B，使得f∈IMOD[proc[f]] do begin
S1:     mark[f] := true;
        worklist := worklist ∪ {f};
    end
L3: while worklist≠Ø do begin
        f := worklist中的任一元素;
        worklist := worklist - {f};
L4:     for each满足 (v,f)∈E_B的v do begin
S2:         if mark[v] = false then begin
S3:             mark[v] := true;
```

图11-2 构造RMOD集合的算法

```
                worklist := worklist ∪ {v};
            end
        end
    end
L5: for each p ∈ P do RMOD[p] := Ø;
L6: for each f ∈ NB do
        if mark[f] then RMOD[proc[f]] := RMOD[proc[f]] ∪ {f};
end computeRmod
```

图 11-2（续）

正确性：为了证明图11-2中的算法*computeRmod*能正确计算RMOD集，我们必须证明在这个算法的出口处，$f \in \text{RMOD}[proc[f]]$当且仅当$f$可能被过程$p = proc[f]$的调用副作用所修改。

充分性：假设f可能被修改，则必然是以下两种情况之一：（1）f在某个以它为参数的过程中被修改，在这种情况下，它将在语句S_1处被标记为真。（2）绑定图上有一条从参数f到达形式参数f_0的路径，且f_0在$\text{IMOD}[proc[f_0]]$中。这样，f_0就被标记为真并且加到循环L_1的*worklist*中。

565 令$f = f_n, f_{n-1}, \ldots, f_1, f_0$表示绑定图上从$f$到$f_0$的路径上的参数的序列。假设$f = f_n$永远不会被标记为真。那么必定存在一个最小的$k$，使得$f_k$肯定不会被本算法标记为真而$f_{k-1}$被标记为真。但是，由于当$f_{k-1}$被标记为真时它被加入到*worklist*，在它最终被从*worklist*中删除时，每条进入f_{k-1}的边都会被检查，这时f_k必定被标记为真，与前面假设矛盾。所以，每个可能被作为调用$proc[f]$的副作用修改的参数都被放入$\text{RMOD}[proc[f]]$。

必要性：假设f是一个标记被置为真的参数，却没有被修改。在算法中，只把那些在过程体中被修改的形式参数或在绑定图中存在到达在其过程中修改的另一参数的一条路径的形式参数，其标记被置为真。由于只有当相应的过程调用可能发生时，绑定图中才会有一条边，所以必定是原参数f可能被修改了。

复杂度 算法*computeRmod*在最差的情况下将执行$O(N+E)$步，其中N和E分别代表调用图中的结点和边的数目。这是因为它的执行时间和绑定图的大小成比例。令N_B和E_B分别表示绑定图中的结点和边的数目。从前面的讨论中，我们知道$N_B \leqslant \mu N$且$E_B \leqslant \mu E$。

剩下的问题是要证明算法运行的时间和绑定图的大小成比例。这样我们就可以证明以上的结果。很明显，循环L_1需要N_B步。假设IMOD被实现为这样一种形式，使我们可以在常数时间（即通过一个位向量）检测某个元素是否为它的成员，那么循环L_2也需要N_B步。循环L_5所需的时间和程序中的过程数与初始化RMOD集的时间的乘积成比例。如果这些集合都实现为长度为μ的位向量，则这个循环需要的时间为$O(N_B)$。假设向RMOD中添加元素需要常数时间，那么循环L_6需要$O(N_B)$的时间，因为它将用一个位向量实现。

所以，问题的关键是循环L_3和L_4所需要的运行时间。如果我们假设其中的*worklist*是以链表的形式实现的，这样从前端取任意一个元素所需要的时间是常数，那么循环体的执行次数不会多于N_B次，因为每个结点至多只会被放入*worklist*中一次。如果我们假设像在拓扑排序算法中那样，边是以前趋表的形式排列的，从而特定结点的所有前趋可以在和前趋数目成比例的时间内找到。这样，循环体对程序中每个结点的每个前趋执行一次，总共E_B次。

这样我们已经证明了在最坏的情况下，算法需要$O(N_B + E_B) = O(N+E)$步。

为了把RMOD扩展为IMOD^+，我们需要访问程序中每个调用点和调用点处的每个参数，来确定这个参数是否被绑定到被调用过程p的$\text{RMOD}(p)$集中的一个变量上。如果RMOD集是用位向量的形式表示的，那么这个确定的过程需要常数时间，所以转换到IMOD^+的过程也需要$O(N+E)$

的时间。这里假设IMOD⁺可以在O($N+E$) 的时间内被初始化。如果IMOD⁺(p) 是用一个位向量表示的，那么这个位向量的长度必然和程序中全局变量的个数V加上p的形式参数个数的和成比例。由于此种形式参数的个数小于μ，因此为每个过程初始化位向量的时间是O($V+\mu$)=O(V)。由于这些工作对每个过程只做一次，所以初始化时间是O(NV)。这样，如果包含初始化的时间，IMOD⁺的计算需要O($NV+E$) 的时间。 566

计算GMOD

一旦为程序中的每个过程p构造了IMOD⁺(p) 集，我们必须按照等式（11-5）用它来计算GMOD(p)，这里我们重复写出这个公式:

$$\mathrm{GMOD}(p)=\mathrm{IMOD}^{+}(p)\cup\bigcup_{s=(p,q)}\mathrm{GMOD}(q)\cap\neg\mathrm{LOCAL}$$

这个等式的含义是：如果一个变量x在IMOD⁺(p) 中，或者x是全局变量且调用图中存在一条从p到另一过程q的非空路径（其中$x\in$ IMOD⁺(q)），那么这个变量在GMOD(p) 集中。换句话说，我们把这个问题归结为调用图中可达性问题的一个变形。众所周知，使用深度优先搜索算法（这个算法是基于Tarjan的找有向图中的强连通分量算法的）可以在程序大小的线性时间内求解可达性问题。图11-3中的算法*findGmod*就是这样算法的一个实现。

```
procedure findGmod(N, E, n, IMOD+, LOCAL)
  integer dfn[n], lowlink[n], nexdfn, p, q, d,
    IMOD+[n], GMOD[n], LOCAL;
  integer stack Stack;
  procedure search(p);
    dfn[p] := nexdfn; nexdfn := nexdfn + 1;
    GMOD[p] := IMOD+[p]; lowlink[p] := dfn[p];
    将p压入Stack;
    for each与p邻接的q do begin
      if dfn[q] = 0 then begin // 树的边
        search(q);
        lowlink[p] := min(lowlink[p], lowlink[q]);
      end
      if dfn[q] < dfn[p] and q ∈ Stack then
        lowlink[p] := min(dfn[q], lowlink[q]);
      else // 应用等式
        GMOD[p] := GMOD[p] ∪ (GMOD[q] ∩ ¬LOCAL);
    end
    // 检查是否为强连通分量的根
    if lowlink[p] = dfn[p] then begin
      // 对强连通分量的每个成员调整GMOD集合
      repeat
        从Stack弹出u;
        LOCAL[u] := LOCAL[u] ∪ (LOCAL[p] ∩ ¬LOCAL);
      until u = p
    end
  end search;
  // 子程序体
  nexdfn := 1; dfn[*] := 0; Stack = Ø;
  search(1); // 约定root = 1
end findGmod
```

图11-3 全局修改副作用传播的算法

复杂度 由于这个算法是深度优先遍历算法的一个直接改写形式，所以它需要运行$O(N+E)$步，其中每一步都涉及到一个长度为V的位向量操作，V表示程序中变量的数目。这样，算法在最坏的情况下需要$O((N+E)V)$个基本步骤。

正确性 我们现在来证明图11-3中的算法*findGmod*能正确地计算程序中每个过程p的GMOD(p)集。这个算法是Tarjan的寻找强连通分量算法的一个直接改写形式。但是，当算法沿调用顺序的逆序回退时，它不是在收集强连通区域的集合，而是在更新每个结点处GMOD(p)的计算。当到达一个强连通分量的头结点处，算法会更新这个分量中每个u的GMOD(u)集，把头结点的GMOD集中的非局部部分加入其中。这样，一个强连通分量中每个过程u的GMOD(u)的全局部分是相同的，它们本该如此。这样，由于遍历顺序本身的特点，可以保证不在循环中的结点的GMOD集是正确的，而由于算法会对环中所有过程的GMOD集进行更新，所以循环中的节点的GMOD集也是正确的。

把本节的结果和前一节的结果综合起来可以证明整个计算可以在$O((N+E)V)$步骤内完成。由于DMOD可以在$O(NV+E)$的时间内由GMOD计算得到，所以DMOD的完整的计算需要$O((N+E)V)$步。这也是最好的可能时间上界，由于我们必须在调用图的每个结点处至少计算一次等式(11-5)，这就需要$O((N+E)V)$的时间。

11.2.4 流不敏感别名分析

前面讲述的是不涉及到别名的副作用分析，下面我们转而讨论别名分析问题以及如何将它集成到常规的副作用问题算法中。

将DMOD更新为MOD

567
~
568

计算了直接修改副作用集后，剩余的问题是如何将别名因素考虑在内。我们用一个具有以下三个过程例子来说明这个问题:

```
      SUBROUTINE P
         INTEGER A
S0       CALL S(A,A)
      END
      SUBROUTINE S(X,Y)
         INTEGER X, Y
S1       CALL Q(X)
      END
      SUBROUTINE Q(Z)
         INTEGER Z
         Z = 0
      END
```

我们感兴趣的是计算集合MOD(S_1)。至今的分析结果告诉我们GMOD(Q)={Z}，这个结果被向后转移到调用点S_1后得到DMOD(S_1)={X}。但是为了真正确定哪些变量将在S_1处被修改，我们还必须知道由于S_0处调用S时把同一个变量传给两个参数，所以在S_1处X能够成为Y的别名。通过这些别名信息我们得到MOD(S_1)={X,Y}。

回忆一下等式（11-2）中的公式，我们重复写在下面:

$$\mathrm{MOD}(s) = \mathrm{DMOD}(s) \bigcup_{x \in \mathrm{DMOD}(s)} \mathrm{ALIAS}(p, x)$$

图11-4给出实现这个转换的一个很自然的算法。我们分析一下这个循环嵌套的复杂度。循环L_1

执行O(E) 时间，其中E表示调用图中的边的数目。语句S_1处的赋值所用的时间和位向量的长度成比例，或者说需要O(V) 步，其中V是程序中变量的数目。由于这个语句执行O(E) 时间，耗费在这个语句上的执行时间是O(EV)。在L_1的每次迭代中，L_2会对DMOD(s) 中的每个变量执行一次。由于这在根本上处理所有变量O(V)，所以这个循环总计会进入O(EV) 次。最后，L_3被进入O(EV) 次，且逐次对ALIAS(p, x) 中的每个变量执行一次迭代。如果我们假设ALIAS(p, x) 可能包含程序中的每个变量，那么这个循环体将被执行O(EV^2) 次。最后，如果我们采用位向量形式的实现，语句S_2需要常数时间。这样这个更新过程的总的运行时间为O(EV^2)。

```
procedure findMod(N, E, DMOD, ALIAS, MOD)
L1: for each调用点s do begin
S1:     MOD[s] := DMOD[s];
L2:     for each x ∈ DMOD[s] do
L3:         for each v ∈ ALIAS[p, x] do
S2:             MOD[s] := MOD[s] ∪ {v};
    end
end findMod
```

图11-4 由DMOD的简单别名更新产生MOD

如果我们不改善别名分析的运行时间，那么它将占用整个副作用分析的绝大部分时间，这样我们就不能在重要的别名模式的程序中使用这个算法。但是，如果给可能发生的别名模式仔细分类，我们可以大大改善这个算法。首先，我们注意到，在一个只包含全局变量和局部变量的两级命名层次结构中，两个全局变量之间绝不会互相成为别名。它们只可能是引用形式参数的别名。这样，一个全局变量x的ALIAS(p, x) 只可能包含过程p的引用的形式参数。这意味着ALIAS(p, x) 中的元素不可能多于μ项，其中μ是程序中任何过程的形式参数的最大数目。

另一方面，对于过程p一个给定的形式参数f，ALIAS(p, f) 可能包含任何全局变量或任何其他的形式参数，所以它的大小可能为O(V)。

从这些观察讯息中，我们知道我们应当把从DMOD到MOD的更新分为两类：一类是针对形式参数的，另一类是针对全局变量的。当考虑一个给定变量$x \in$ DMOD(s) 的别名时：

（1）如果x是一个全局变量，我们将把一个很小的集合（$\leqslant \mu$个元素）加入到MOD(s) 中，但是我们可能需要加入O(V) 次；

（2）如果x是包含s的过程p的一个形式参数，我们需要把可能多达O(V) 个的元素加入MOD(s) 中，但我们只需要加入很少的次数（$\leqslant \mu$次）。

在这两种情况下，工作量都是O(V)而不是O(V^2)。完整的算法在图11-5中给出。

通过前面的讨论应当清楚，图11-5中的合并策略对于每个调用点的全部运行时间是O(V)，总的运行时间是O(EV)。这就意味着如果我们能在O($(N+E)V$) 时间内计算得到ALIAS集，那么整个MOD集计算的时间上界就是O($(N+E)V$)。

计算别名

我们下面将集中讨论为每个过程p计算ALIAS(p, x) 集的问题，其中变量x是全局变量或某个过程的参数。为了尽快计算出结果，我们再一次利用全局变量只可能是形式参数的别名这个观察结果。首先，我们为程序中每个形式参数计算一个中间量$A(f)$，它定义全局变量的一个集合，这些全局变量通过绑定图上如下一系列参数绑定可能成为形式参数f的别名：

569 ≀ 570

```
procedure updateMODwithAlias
  for each程序中的调用点s do
    t := Ø; // t是一个长度为μ的临时位向量
    for each全局变量x ∈ DMOD(s) do      // O(V) 次迭代
        t := t ∪ {x};                   // 常数时间
        t := t ∪ ALIAS[p, x];           // O(μ) 时间
    end
    MOD[s] := MOD[s] ∪ t;               // O(V) 时间
    for each形式参数f ∈ DMOD(s) do begin // O(μ) 次迭代
      MOD(s) := MOD(s) ∪ {f};           // 常数时间
      MOD(s) := MOD(s) ∪ ALIAS(p, f);   // O(V) 时间
    end
  end
end updateMODwithAlias
```

图11-5 副作用和别名信息的快速合并

$$g \to f_0 \to f_1 \to f_2 \to \ldots \to f_{n-1} \to f_n = f$$

为了计算$A(f)$，我们将在绑定图的变形形式上进行前向传播，其中绑定图的变形形式是指将绑定图中的环归约成一个结点。注意，$A(f)$ 是ALIAS(p, f) 的近似，包含其中的所有全局变量。为程序中每个形式参数计算$A(f)$ 的算法在图11-6中给出。在这个算法中，集合$A(f)$表示为一个长度为O(V) 的位向量。

```
procedure computeA(N, E, N_B, E_B, A)
  for each形式参数f ∈ N_B do A[f] := Ø;
  for each调用点s ∈ N do begin
    for each在s处映射到形式参数f的全局量g do
      A[f] := A[f] ∪ {g};
  end
  将绑定图中每个环用一个单独的结点替换，将图归约为一个有向无环图的形式;
  for each归约图中依拓扑顺序出现的f do
    A[f] := A[f] ∪ ⋃_{(f0, f) ∈ E_B} A[f0];
  for each原来绑定图中的环do begin
    令C是归约的绑定图中代表这个环的结点;
    for each f ∈ C do A[f] := A[C];
  end
end computeA
```

图11-6 计算形式参数的近似别名集的算法

容易看出，最坏的情况下算法的前向传播阶段需要O(NV) 步，更新环中这些集合也需要O(NV) 步。

一旦我们得到了程序中每个形式参数的$A(f)$ 集，就能通过一个简单的反演变换得到每个全局变量的别名集合，算法在图11-7中给出。这个计算可以分成两个部分：初始化别名集合和计算这个变换。前面提到，我们可以把给定过程中的一个全局变量的别名集合表示为长度为μ的位向量。所以，图11-7中步骤S_0的初始化过程不超过O$(NV\mu)$=O(NV) 步。然而，因为一次更新可在常数时间内完成，所以语句S_1处的更新动作的总的开销也是O(NV)。

```
procedure computeAlias
  for each过程p do
    for each全局变量g do
S0:     ALIAS[p, g] := Ø;
  for each程序中的形式参数f do begin
    令p是声明形式参数f的过程;
    for each g ∈ A[f] do
S1:     ALIAS[p, g] := ALIAS[p, g] ∪ {f};
  end
end computeAlias
```

图11-7 变形为计算每个全局变量g的ALIAS(g)

剩余的工作就是计算形式参数的别名。注意，$A(f)$ 是ALIAS(p, f) 的近似，它包含所有为f别名的全局变量。要把$A(f)$ 扩展到ALIAS(p, f)，我们只需要把可能为f别名的形式参数加入ALIAS(p, f)。在我们考虑的简单的两层结构语言中，只有同一过程的其他形式参数可能为f的别名。这样，在任何给定的过程中，可能的形式参数对不会多于$\mu(\mu+1)/2$。

为了计算可能为其他形式参数别名的形式参数，图11-8中的算法*computePairs*跟踪给定过程p中可能互为形式参数别名的形式参数对的集合FPAIRS(p)。我们首先把所有FPAIRS集合初始化为空，然后检查每个调用点，察看是否有通过把同一变量传递给两个不同的参数而引入的别名。一旦发现这样的情况，我们就把别名的形式参数对加入工作表。接下来，我们在工作表上迭代，以查找可能的别名传播，这些传播是由于两个可能互为别名的参数都被传递到同一个过程中而引起的。如果被调用过程中的相应的参数对不在FPAIRS集中，那么我们将把这个参数对加入工作表。这个过程将持续到工作表为空。

```
procedure computePairs
    W := Ø;
L1: for each程序中的过程p do FPAIRS[p] := Ø;
L2: for each别名的引入点（例如，"CALL P(X, X)"）do
      将因此得到的形式参数对加入到FPAIRS[p]中和工作表W中
L3: while W ≠ Ø do begin
      从W中删除 <f1, f2>;
L4:   for each过程p中同时传递f1和f2的调用点s do begin;
        令q是s处调用的过程;
        if f1和f2在s处被传递给f3和f4 and
          <f3, f4> ∉ FPAIRS[q] then begin
          FPAIRS[q] := FPAIRS[q] ∪ {<f3, f4>};
          W := W ∪ {<f3, f4>};
        end
      end
    end
end computePairs
```

图11-8 计算可能互为别名的形式参数对的算法

让我们分析一下算法*computePairs*的运行时间。注意在任何子程序中参数对的数目不会超

过$\mu(\mu+1)/2$。由于对每个过程的初始化都只需要常数时间，所以循环L_1的初始化过程需要$O(N)$步。循环L_2需要察看每个调用点，所以需要的时间为$O(E)$，但是如果我们假设W和FPAIRS都是用某种链表结构实现的，那么所有的集合操作需要的是常数时间。对于每个被放到工作表的参数对，循环L_3都会被执行一次。由于每个调用点最多只有$\mu(\mu+1)/2$个参数对，所有L_3的总的迭代次数小于或等于$\mu(\mu+1)N/2$次，故进入L_3的次数为$O(N)$。同样，程序中的每个调用点被检查的次数不会超过$\mu(\mu+1)/2$次，所以进入循环L_4 $O(E)$次。由于循环L_4的循环体的执行需要常数时间，所以整个过程需要的时间为$O(N+E)$。

这样，我们证明了这个别名分析算法可以在$O((N+E)V)$的时间内完成，这就意味着整个MOD求解可以在这个时间内完成。

11.2.5 常数传播

571 ~ 573

常数传播是一个可以提高很多类优化性能的重要数据流分析问题[127][136]。可惜即使在单独的一个过程中，此问题也难于解决，因为得到这个问题的精确解被证明是不可能的[165]。即使通常的单过程近似问题在过程间的范畴中也是流敏感的，因而也是很难解决的[220]。

造成这些困难的一个重要原因是：当常数传播到一个程序区域内时，这个程序区域中的相关表达式可能会被计算，从而在出口处得到新的常数。下面例子可以说明过程间的情况：

```
SUBROUTINE PHASE(N)
   INTEGER N, A, B
   CALL INIT(A, B, N)
   CALL PROCESS (A,B)
END
SUBROUTINE INIT(A, B, N)
   INTEGER A, B, N
   A = N+1
   B = (N*A)/2
END
```

子程序INIT的目的是初始化变量A和B。所以，如果在过程PHASE的入口处N是常数10，则在INIT的出口处A是常数11，B是常数55。因此，这些常数值在PROCESS的入口处也是有效的。本节将会介绍一种确定这些事实的方法。

单过程中的常数传播算法已在4.4.3节中介绍。这个算法在图4-5中给出，它在定义-使用图或静态单赋值（SSA）式[96]上运用一个迭代过程。因为这个迭代的过程是基于图4-4中所示的常数传播格，所以它保证是收敛的。而且，这个算法相对于问题所在的图的大小来说是线性的，因为一条指令最多使它的输出值在格中的位置下降两次，所以，它也最多被放入工作表两次。

我们的策略是开发一个和迭代单过程常数传播过程类似的过程间方案。事实上，我们希望仍然能使用相同的算法，但是算法所基于的图不同于前面提到的定义-使用图，我们称这个过程间方案中用到的图为*过程间值传播图*。在这种图中，结点代表“跳转函数”，它根据进入一个过程时的已知值计算离开给定过程时的值。跳转函数类似于过程内和过程间区间分析中的“传递函数”[131]。

令s是过程p内的一个过程调用点，x是s处所调用的过程q的形式参数。x在s处的跳转函数标记为J_s^x，它根据包含s的过程p的输入值来决定x的值。J_s^x的*支持集*是指在计算J_s^x时实际使用的输入的集合。跳转函数可以通过检查程序中单独过程的预备阶段计算。

574

现在，我们回到过程间值传播图的构造上来。下面是构造过程间值传播图的步骤：

（1）为每个前向跳转函数J_s^t构造一个结点。

（2）如果$x \in \text{support}(J_t^s)$，其中$t$是在$s$处被调用的过程内的一个调用点，那么构造一条从$J_s^x$到$J_t^s$的边。

迭代常数传播算法可以应用到由以上步骤得到的图上。

我们现在给出这个过程的一个简单例子。考虑图11-9所示的程序。我们从这个例子容易导出跳转函数

$$J_\alpha^X = \{1\};\ J_\alpha^Y = \{2\}$$
$$J_\beta^U = \{X + Y\};\ J_\beta^V = \{X - Y\}$$

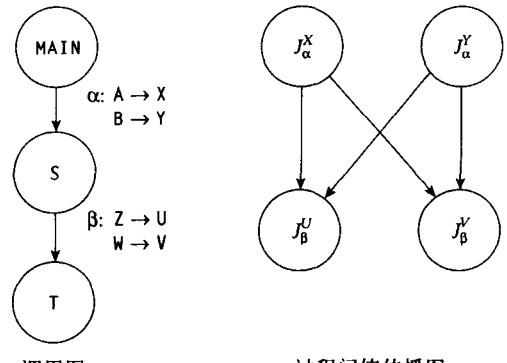
相应的调用图和得到的过程间值传播图在图11-10中给出。

```
      PROGRAM MAIN
         INTEGER A,B
         A = 1
         B = 2
α        CALL S(A,B)
      END
      SUBROUTINE S(X,Y)
         INTEGER X,Y,Z,W
         Z = X + Y
         W = X - Y
β     CALL T(Z,W)
      END
      SUBROUTINE T(U,V)
         PRINT U, V
      END
```

图11-9 过程间常数传播的例子

容易看出把常数传播算法应用到图11-10中的过程间值传播图会很快收敛到如下常数赋值：

X＝1; Y＝2; U＝3; V＝－1;



调用图　　过程间值传播图

图11-10 过程间值传播图的例子

为了估计这个算法的总开销，回忆一下，如果跳转函数的计算可以在常数时间内完成，那么本算法所基于的迭代常数传播算法的开销是和图的结点和边的数目成比例的。但是，我们不可能期望所有的跳转函数都能在常数时间内计算得到，因为，正如我们将看到的，构造跳转函数的不同策略会得到不同执行开销的函数。

因此，我们必须讨论跳转函数被计算的次数。一个跳转函数J有$support(J)$个输入，每个输入最多在格中被降低两次。这样一个跳转函数J被计算的次数不会超过$O(|support(J)|)$次。令$cost(J)$表示执行跳转函数J的开销，那

么每个跳转函数的执行的总开销为O(|*support*(*J*)|.*cost*(*J*))。所以，执行过程间常数传播算法的总开销为

$$O\left(\sum_{s}\sum_{x}\left|support(J_s^x)\cdot \cos t(J_s^x)\right|\right) \tag{11-7}$$

其中s的范围是程序中所有的调用点，x的范围是对应子程序的输入参数。

构造跳转函数

跳转函数会因近似估计的精确度和开销的不同而有很大不同。在复杂的情况下，一个跳转函数可能会对它所处理的过程应用完全的符号翻译。但是，在简单的情况下，它也可以只计算通过子程序的每条路径上且不包含在任何循环中的赋值。这种情况下，只在子过程的部分路径上或在循环内被赋值的变量将得到常数格中的⊥（底）值。

575 ~ 576

一个主要的问题是如何为一个过程体有函数调用的过程构造跳转函数。考虑图11-11给出的例子。为了在调用点γ为T构造跳转函数，我们必须知道在调用点β处调用的子程序INIT的行为。要解决这个问题，我们定义“返回跳转函数”，它汇总来自一个子程序在用一组特定的输入调用时的常数传播。

```
      PROGRAM MAIN
        INTEGER A
α       CALL PROCESS(15, A)
        PRINT A
      END
      SUBROUTINE PROCESS(N, B)
         INTEGER N, B, I
β        CALL INIT(I, N)
γ        CALL SOLVE(B, I)
         RETURN
      END
      SUBROUTINE INIT(X, Y)
         INTEGER X, Y
         X = 2 * Y
         RETURN
      END
      SUBROUTINE SOLVE(C, T)
         INTEGER C, T
         C = T * 10
         RETURN
      END
```

图11-11 一个复杂的过程间常数合并的例子

如果x是过程p的一个输出，返回跳转函数R_p^x决定x在对p的一次调用后对应于p的输入参数的值的返回值。R_p^x的支持集同前向跳转函数的相同。对于图11-11中给出的子程序INIT简单情况，我们有返回跳转函数

$$R_{\text{INIT}}^{\text{X}} = \{2 * \text{Y}\}$$

而对于过程SOLVE的情况，我们有

$$R^{C}_{SOLVE} = \{T * 10\}$$

我们可以把返回跳转函数用于常数传播的常规跳转函数中。例如，调用点γ处的跳转函数如下：

$$J^{T}_{\gamma} = \{\text{if } I \in MOD(\beta) \text{ then } R^{X}_{INIT}(N) \text{ else } undefined - const\}$$

其中，*undefined-cost*用于表示对那些未初始化的变量给出的一个特定值。为了完成这个例子的分析，我们给出剩余的跳转函数：

$$J^{N}_{\alpha} = \{15\} \text{and } J^{\gamma}_{\beta} = \{N\}$$

还有一个重要的跳转函数没有被定义，即子程序PROCESS的返回跳转函数，它对于决定在主程序中的输出语句处的变量A能否被一个常数所替换是十分重要的。这个返回跳转函数必须调用INIT和SOLVE的返回跳转函数，但是，如果我们利用前向跳转函数来决定SOLVE的输入形式参数的值，那么就会自动调用INIT的返回跳转函数。这个返回跳转函数可表示为：

$$R^{B}_{PROCESS} = \begin{array}{l}\{\text{if } B \in MOD(\gamma) \\ \text{then } R^{C}_{SOLVE} J^{T}_{\gamma}((N)) \\ \text{else } B\}\end{array}$$

利用这些跳转函数，我们可以看到变量A在过程PROCESS出口处的值由

$$R^{B}_{PROCESS} = (2 * (15)) * 10 = 300$$

Grove和Torczon[127]对于科学计算Fortran程序的研究说明MOD应当在计算跳转函数之前计算，并且也应当在执行用于初始化过程间常数传播的一个全局常数传播之前计算。因为在任何情况下，当一个变量不在给定调用点的MOD集中的时候，返回跳转函数是简单的恒等式。在这种情况下，返回跳转函数就不需要了，这就简化了作用域包含给定调用点的同一变量的其他跳转函数的构造。这是返回跳转函数最简单的有用类型。同一项研究还证实：

（1）简单跳转函数几乎能得到科学计算Fortran程序中的所有常数；

（2）返回跳转函数仅在很少的情况用于获得新常数，但是使用这些函数的代价是比较高的。

11.2.6 注销分析

我们可以改变用于解决MOD问题和常数传播问题的一些思想，以此来解决注销问题。回忆一下，KILL(p) 是所有必定在过程p的所有路径上都被修改的变量的集合。为了在算法中方便地用公式表示，我们将计算NKILL(p)=¬KILL(p)，这是没有作为调用过程p的副作用被注销的所有变量的集合。用一个与此类似的方法可以计算程序中每个过程p的USE(p)。

577
↓
578

让我们首先考虑如何解决单个过程的NKILL问题。对于每个扩展的基本块b和它的每个后继c，假设我们用集合THRU(b, c) 表示通过由b到c的某条路径没有被注销的所有变量。那么，以下的数据流方程组可以用来解决求NKILL(b) 的问题：

$$NKILL(b) = \bigcup_{c \in succ(b)} THRU(b, c) \cap NKILL(c) \tag{11-8}$$

如果e是过程的出口结点，那么

$$NKILL(e) = \Omega \tag{11-9}$$

其中Ω表示所有变量的集合。这个方程组可以用数据流分析的简单迭代方法求解。

用类似的过程可以对程序中每个过程计算NKILL(p)。首先，我们需要一个公式来计算单个过程的NKILL(p)。这个公式比计算单个基本块的公式复杂得多，因为它依赖于特定过程的控制流。为了进行这样的计算，我们首先需要构造过程的*归约控制流图*G_{THRU}。G_{THRU}中，每个结点是一个调用点、过程的入口结点或出口结点。G_{THRU}中的每条边 (x, y) 上标注有变量的THRU(x, y)集合，表示在从x到y的不包含调用点的某条路径上没有被注销的所有变量。用图11-12中给出的算法可以构造G_{THRU}。

```
procedure ComputeReducedCFG(G)
  从控制流图G中删除所有的回边;
  令b0表示过程的入口结点;
  标记b0已被处理;
  worklist := Ø;
  for each s ∈ successors(b0) do worklist := worklist ∪ {(b0, s)};
  while worklist ≠ Ø do begin
    从worklist任取一个元素 (b, s)，使得s的所有前趋结点
      已经被处理或者合并到b中;
    if s是一个调用点then begin
      for each t ∈ successors(s) do
        worklist := worklist ∪ {(s, t)};
      标记s已被处理;
    end
    else if s是一个出口结点then不做任何处理
    else begin // s是一个常规结点
      将s合并到b中;
      for each t ∈ successors(s) do
        if THRU[b, t]未被定义then
          THRU[b, t] := THRU[b, s] ∩ THRU[s, t];
        else
          THRU[b, t] := THRU[b, t] ∪ (THRU[b, s] ∩ THRU[s, t]);
    end
  end
end ComputeReducedCFG
```

图11-12 构造归约控制流图的算法

很容易看出，图11-12中的算法采用了一个拓扑排序算法的变形，算法可以在控制流图大小的线性时间内实现。

一旦有了归约控制流图，图11-13中给出的算法就可以用于计算一个过程p的NKILL(p)，其中在过程p内可以被调用的所有过程的NKILL值是给定的。

假设程序中所有的变量都是全局变量，那么我们利用一个简单的迭代数据流分析算法（采用图11-13中的过程）就可以计算NKILL。虽然，这个算法在最坏的情况下需要$O(N^2V)$的时间，但是，如果调用图是可归约的，算法仅仅需要$O((N+E)d)$个位向量操作步[143]，其中d是在调用图中非环路径中回边数的最大值。这种情况下需要的总时间为$O((N+E)dV)$，迭代方法实际上是非常快的。

到现在为止，我们所描述的算法仅仅能在程序中不引用形式参数的情况下快速计算

NKILL(*p*)。如果存在引用形式参数，这个算法也适合用来在已知正确的形式参数-实参映射关系的情况下产生正确的答案。但是，此算法可能会需要很长时间才能达到收敛，原因是可能会存在“移位寄存器效应”，在这种情况下，某些参数会被传递给一个递归调用环中的其他参数。我们在11.2.3节中处理MOD问题时观察到同样的问题。

```
procedure ComputeNKILL(p)
  for each G_THRU(p) 中逆拓扑排序的b do begin
    if b是一个出口结点then NKILL[b] := Ω;
    else if b是一个调用点then begin
      NKILL[b] := Ø;
      for each G_THRU(p)中b的后继结点s do
        NKILL[b] := NKILL[b] ∪ (NKILL[s] ∩ THRU[b, s]);
      NKILL[b] := NKILL[b] ∩ NKILL[q],
        其中q是在b处调用的过程;
    end
    else begin // b是G_THRU(p) 的入口结点
      NKILL[b] := Ø;
      for each G_THRU(p) 中b的后继结点s do
        NKILL[b] := NKILL[b] ∪ (NKILL[s] ∩ THRU[b, s]);
      NKILL[p] := NKILL[b];
    end
  end
end ComputeNKILL
```

图11-13 计算NKILL(*p*)

幸运的是，我们可以用同样的通用技术来改进这个问题的解决方法——使用形式参数绑定图。我们将构造和标记绑定图，算法在图11-14中给出。

```
procedure BindingComputeNKILL(P)
  最初，令绑定图包含程序P中的每个形式参数的一个结点;
  worklist := Ø;
  for each程序中的过程p do begin
    令NKILL0[p]是应用图11-13中的算法的结果，
      其中对p的每个后继q有NKILL[q] = Ω(q)，Ω(q) 表示q的形式参数集合;
    NKILL[p] := NKILL0(p);
    for each p的形式参数f do begin
      if f ∈ NKILL0[p] then
        killed(f) := false;
        for each f在p内的任意一个调用点被传递到的形式参数g do
          在绑定图中添加一条边 (f, g);
      end
      else begin
        killed(f) := true; worklist := worklist ∪ {f};
      end
```

图11-14 构造和标记NKILL的绑定图

```
    end
  end
  while worklist ≠ Ø do begin
    任取一个元素f ∈ worklist;
    worklist := worklist − {f};
    for each在绑定图中存在一条边 (g, f) 的g and killed(g) = false do begin
      令q是用g作为形式参数的过程;
      NKILL[q] := 应用图11-13中的算法的结果，其中q的后继结点采用当前的NKILL集合;
      if g ∉ NKILL[q] then begin
        killed(g) := true; worklist := worklist ∪ {g};
      end
    end
  end
end BindingComputeNKILL
```

图 11-14（续）

只有当我们发现某过程的一个参数的状态可能被改变为“被注销”，而改变的原因是在某个调用点传递的一个参数可能已经改变了状态，此时我们会更新这个过程的NKILL(p) 集合。除此之外，这个算法仅仅是迭代算法的一个简单变形。由于每个参数只能被加入到*worklist*之中一次并且算法是访问从*worklist*中取出的一个参数的每个前趋，所以更新NKILL的总次数就被限制在$O(E_B+N_B)=O(E+N)$ 次之内，其中E_B和N_B分别代表绑定图中边和结点的数目，而E和N分别代表调用图中边和结点的数目。如果我们仔细地选择从*worklist*中取出元素的顺序，那么更新的次数还可以进一步减少。

我们注意到这个过程计算出的注销集合是不精确的，因为这些集合的计算没有考虑别名关系。假设我们知道每次访问过程p时，变量$x_1, x_2, \ldots, x_n$引用的是同样的位置，那么如果一个变量的一个别名在s处调用的过程或任何在这个过程内调用的过程中的每条路径上都被注销，则这个变量属于KILL(s)。换句话说，我们可以通过考虑别名关系而增大KILL集合的大小。但是，这样做的代价很高，因为这需要计算在过程的某次调用中，引用同一位置的所有变量的元组的集合，这需要程序中用做参数的变量数目的指数级的时间。

11.2.7 符号化分析

除了到现在为止讨论的一些基本分析问题之外，成功的程序并行化还需要解决一些更复杂的过程间问题。其中两个尤为重要，它们是符号化分析和数组区域分析[142，136]。

虽然常数传播提供关于过程入口处变量的值信息，但大多数情况下，变量不是常数，因而无法证明它们是常数。然而，要改善全局程序分析，我们也许不必证明一个变量是常数。也许确定变量值的范围或证明它的值可以用程序中同一点的其他变量的值的函数表示就足够了。这样的符号关系可以被用于证明一些事实，例如，没有依赖关系等。考虑如下简单的代码：

579 ~ 582

```
SUBROUTINE S(A, N, M)
  REAL A(N+M)
  INTEGER N, M
  DO I = 1, N
    A(I+M) = A(I) + B
  ENDDO
END
```

如果我们能证实在S的入口处N=M，那么就能知道这个子程序的循环中没有携带循环依赖。

过程间符号化分析的目的是为了证明在某子过程的入口处或某调用处的返回点一些关于变量值可能满足的事实。经常进行的符号化分析有三类：

（1）符号化表达式分析：这种分析是试图借助于程序中同一点其他变量的值来确定做为一个变量值的符号化表达式。

（2）谓词分析：这种分析是试图建立一对变量在程序给定点可能具有的值的关系。

（3）范围分析：这种分析是试图建立一个变量在程序给定点的值的范围，这个范围由已知的常数上下界组成，有时也包括常数跨距。

通常情况下，上述某种分析的结果可以被另一种分析的结果所代替。例如，在前面例子中，符号化表达式分析和谓词分析都会发现M=N这个事实。两者之间的区别在于表达式分析所产生的值可能会涉及到很多其他的值，而谓词分析通常仅涉及一系列的变量对。另一方面，产生符号化表达式的分析要比简单的谓词分析复杂得多。

范围分析可以有效地用在程序分析中以排除某种可能性。例如，考虑下面的子程序：

```
SUBROUTINE S(A, N, K)
   REAL  A(0:N)
   INTEGER  N, K
   DO I = 2, N
      A(I) = A(I) + A(K)
   ENDDO
END
```

如果我们能证明在这个子过程的入口处K∈[0:1]成立，那么就能证明这个循环不携带循环依赖。

在一单个过程中，进行符号化表达式分析通常都采用某种形式的值计数[27，236]，它对每个表达式给定惟一的数值，这样，在程序的给定点，数值相同的表达式在运行时会具有相同的值。由于在整个程序范围内的值计数很可能是十分复杂的，所以典型的做法是只在过程范围内进行值计数。在过程间分析中，一个受限制的关系集合跨越过程边界进行传播[14]，例如用谓词表示的关系中只会涉及到两个变量。 583

容易看出，范围分析和符号化表达式分析可以用第11.2.5节中的常数传播算法的变形来解决。为了使用这个算法，我们需要首先定义以下三件事情：

（1）在程序中引入新符号信息的过程

（2）跳转函数：由包含某个调用点的过程的入口处信息产生该调用点处的信息

（3）返回跳转函数：由过程入口处成立的关系确定该过程输出结果的关系

让我们考虑如何综合利用这三个函数来计算范围信息。范围信息一般是在程序的控制流点引入的。例如，在一个以：

```
DO I = 1, N
```

为循环头的循环中，假设I ∈ [1,N]是安全的。同样，条件语句可以引入部分范围信息，而这些范围信息可以组合构成完整的范围信息。

在所有的符号分析方法中，跳转函数计算格中的值，这种网格比常数传播中的格复杂得多。例如，在范围分析的情况下，我们可能用到这样的格：格中的合并操作在由两个连接的控制流分支的一对范围中选取较大的上界和较小的下界。图11-15中描述的就是这样一个格中的一部分。如果我们可以限制一个上界在变成∞之前可以增加次数，并同样地限制一个下界可

以减小的次数，那么这个格点就可以用在所有需要用有限递减链格的地方，比如用在迭代算法中。

符号化谓词分析比范围分析更复杂，因为在谓词分析中需要察看一对变量之间的关系。例如，知道以下信息是非常有用的：两个变量x和y之间是否存在相等的关系或者从其中一个变量到另一个变量的偏移量是一个常数。这样的关系可能被表示为

$$x - y = c$$

其中c是一个编译时的常数。关于变量之间的更加通用的线性关系可以被刻划为

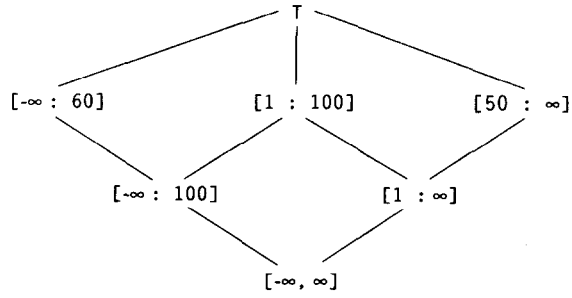


图11-15 用于范围分析的简单值的网格

$$c_1x + c_2y = c_0$$

其中c_0，c_1和c_2都是常数。注意，这个关系是传递性的：如果y和z之间具有关系

$$d_1y + d_2z = d_0$$

那么我们可以找到常数e_0，e_1和e_2，使得

$$e_1x + e_2z = e_0$$

特别地，

$$c_1d_1x - c_2d_2z = c_0d_1 - c_2d_0$$

这样，在程序的任何一点，我们可以找到相互之间具有线性关系的变量的集合。符号化谓词分析的目标就是在程序中传播这些集合。利用过程间常数传播的一种变形可以做到这一点。

11.2.8 数组区域分析

584 ~ 585

到现在为止，所讨论到的分析对于我们在自动程序并行化过程中必须解决的最重要的问题之一没有多大帮助，这个问题就是如何分析包含过程调用的循环中的依赖。考虑如下代码：

```
      SUBROUTINE S
        DIMENSION A(100,100)
        ...
        DO I = 1, N
S1        CALL SOURCE(A,I)     ! 对A赋值
S2        CALL SINK(A,I)       ! 使用A
        ENDDO
        RETURN
      END
```

如果希望将这个子程序中的循环并行化，我们必须判断这个循环中是否携带循环依赖。前面几节所描述的若干种过程间信息用处很小，只能告诉我们数组A被SOURCE修改并由SINK使用。如果没有更有用的信息，我们必须假设在SOURCE中存在对数组某个元素的赋值，这个赋值在循环的某次后续迭代中被SINK使用。换句话说，我们必须假设这个循环携带循环依赖，且不能被并行化。

如果我们能证明在两个例程中对A的访问只局限在第I列，情形就不一样了，这隐含把I作

为参数传递给两个例程。此时，我们就知道这个循环的不同迭代处理的是数组的不同部分，没有循环携带依赖存在。我们下面将提炼出一种过程间分析方法可以证明类似这样的条件，也就是说我们的分析应当能识别整个数组中的子数组。

假设我们能计算在循环的第I次迭代中SOURCE可能修改的数组内的元素集合$M_A(I)$，以及第I个迭代中SINK可能使用到的数组元素集合$U_A(I)$。用处理数组的MOD和USE版本可以分别计算这两个集合。那么，循环中存在真依赖的充分必要条件是存在I_1和I_2（$1 \leqslant I_1 \leqslant I_2 \leqslant N$）使得

$$M_A(I_1) \cap U_A(I_2) \neq \emptyset$$

为了在这些子数组上进行分析，我们需要一种表示子数组的方法。这种表示法也应当很容易表示“交”和“并”的操作。

很直接的想法是扩展标准数据流算法，原算法中使用了位向量，其中的位向量每一位只能表示两种状态（如可能被修改或必定不修改），扩展的方法是把位向量扩展为更通用的格元素的向量。如果能找到一个可以精确表示子数组的格，那么，我们就可以在过程间数据流分析过程中使用这个格来确定子数组的副作用。 586

格表示应具备如下几个重要的特性:

（1）表示应当尽可能地精确。

（2）条控制流路径汇合时所调用的合并操作必须高效。

（3）依赖测试也应当高效，因为依赖测试通常涉及到两个区间表示的交操作。

（4）应当能在分析框架内处理递归，这意味着格应当具有有限递减链的特性，换句话说，格中每个递减链在经过有限步骤后必须达到最小值。

（5）这个网格应当能处理调用点出现的参数传递。

让我们考虑在图11-16中给出的一个可能用于表示子数组的格，这个格是满足若干以上要求的。格中的元素为简单规则区域，因为它们只能表示非常有限数目的规则的子数组，即点、行、列和整个矩阵。注意，因为我们可能在下标中使用任意的变量或常数，所以这个格也可能扩展成无限大。但是它具有有限递减链的特性，因为格中没有元素可能被下降三次以上。

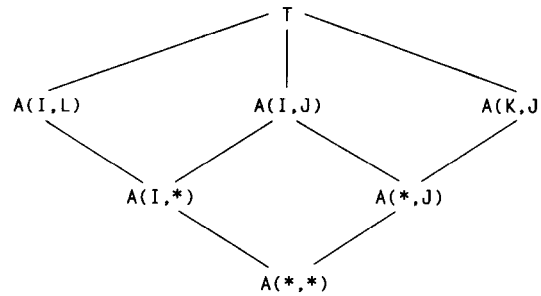


图11-16 简单规则区域格

在评价这个格表示时，我们发现其中的合并操作（表示类似于MOD计算中集合的“并”操作）具备如下特点:

（1）格的深度为$k+1$，其中k是所表示的数组的下标位置的个数。

（2）合并操作的代价是$O(k)$，因为对于两个引用的每个下标位置必须进行检查和比较，以决定什么必须合并。 587

（3）对于依赖测试来说很关键的“交”操作，在这里是有限形式的合一操作，它也可以在下标数目的线性时间内完成。

为了证实最后一个断言，注意在格中每个下标位置上，要么是一个符号表达式，要么是一个常数或表示整个行或列的符号“*”。这样，如果两个下标都是符号表达式，则当二者相同时，结果下标为同样的表达式；否则，结果下标的值为“*”。如果被合并的一个下标是表

达式，另一个下标是“*”，则结果下标为“*”。

一个剩余的问题是：这种表示的精确程度如何？实践证明这种表示方法过于简单了,因为这种方法不能表示区域有界的子数组。因此，在数组A(100, 100) 中，A(1:10, 1:10) 的最好的近似表示是A(*, *)。一种好得多的表示是有界规则区域，在这种表示方法中，允许指出每一维下标的上界和下界。它也可以被看作是Fortran 90中跨距为1的三元组记法。对这种表示的较为广泛使用的扩展是任意跨距的三元组记法和三角形子数组。

只要所用到的格具有有限递减链特性，那么我们就可以直接改写诸如MOD问题解法中过程间算法来处理格元素向量。由于形式参数被放入工作表的次数不可能超过k次(k是格的最大深度)，所以MOD算法中和绑定图相关的部分是收敛的。算法的可达性部分也是收敛的，因为算法每找到一个强连通区域，这个区域内的每个元素的元素向量被设置为这个区域内的最小格元素。

注意，到现在为止所讲述的四个副作用问题——MOD，REF，NKILL和USE中，我们总是想向高的方向估计所涉及到的数组区域，因为我们只有在确切知道一个变量不在调用点处的这些副作用问题的集合中时才做优化。考虑11.2.1节中的私有化的例子。我们将能把变量P变成循环私有的，如果我们发现对于循环开始处的调用点S_0有

$$P \in (KILL(S_0) \cap \neg USE(S_0))$$

换句话说，我们在

$$P \notin (NKILL(S_0) \cup USE(S_0))$$

时进行优化，所以我们必须向高的方向而不是向低的方向估计这些集合，只有这样才能保证不精确的结果只会导致失去优化的机会，而不会导致错误的代码。

11.2.9 调用图的构造

到现在为止，我们总是假设在一个精确的调用图上解决过程间数据流问题。乍一看，似
588 乎容易构造出这样一个调用图：只要简单地检查程序中的每个过程，对于每个调用点构造一条从调用过程到被调用过程的边。只要没有过程参数的存在，这种简单的方法就可以很好地工作。但是，如果有过程参数存在，我们将看到类似下面的代码:

```
      SUBROUTINE SUB1(X, Y, P)
        INTEGER X, Y
S0      CALL  P(X, Y)
        RETURN
      END
```

这里的问题是决定什么过程将在调用点S_0处被调用。如果对于SUB1只有一个调用序列，那么可以简单地反查调用链确定传给参数P的过程。但是，我们无法假设这里只有一条调用链存在，因为在语言中引入过程参数的目的是为了即使在同一个程序中，可以把多个不同的过程传给参数P。

为解决这个问题，对于每个过程参数P，我们必须能够确定可能被直接或间接地传递给P的过程的名字。但是，我们必须小心避免在多个过程参数被传递的情况下损失精确性。考虑如下的例子:

```
      SUBROUTINE SUB2(X, P, Q)
        INTEGER X
S1      CALL  P(X,Q)
```

```
        RETURN
      END
```

假设我们有两个调用

```
CALL SUB2(X, P1, Q1)
CALL SUB2(X, P2, Q2)
```

其中P1和P2只就它们的整型参数调用它们的过程参数，然后返回

```
      SUBROUTINE P1(X, Q)
        INTEGER X
S2      CALL Q(X)
        RETURN
      END
```

这样，在子程序SUB2中的调用点S_1处，我们可以把过程Q1传递给过程P1或者把过程Q2传递给过程P2。我们决不能把过程Q1传递给过程P2或者把过程Q2传递给过程P1。换句话说，在这个程序中，P1只能调用Q1，P2只能调用Q2。但是，简单的过程跟踪方案只维护能够传递给某个 589
给定参数的过程的列表，这个方案可以得到从P1到Q2和从P2到Q1的边，因为可能传给P1中的参数Q的过程列表包括Q1和Q2。

为了克服这个问题，一个精确的调用图构造算法必须跟踪可能被同时传给S_2中的过程形式参数的过程参数对。这就提出一个通用的过程调用图构造算法。

假设对于每个接受过程参数的过程p，我们收集一个过程各元组集合PROCPARMS(p)，这些过程各可以同时传递给p，元组中过程名的顺序同p的参数表中过程参数的顺序相同。这样，图11-17中的迭代算法就可以被用来确定在每个调用点传递的过程参数元组的正确集合。

```
procedure ComputeProcParms
  for each程序中的过程p do PROCPARMS(p) := Ø;
  W := Ø;
  for each程序中的调用点s do begin
    if此调用点向被调用过程的所有过程参数传递过程名字do begin
      令t = ⟨N1, N2, ..., Nk⟩ 是被传递的过程名字元组，其中过程名的顺序是其传递的顺序;
      W := W ∪ {⟨t, p⟩}，其中p是被调用的过程;
    end
  end
  while W ≠ Ø do begin
    令⟨t = ⟨N1, N2, ..., Nk⟩, p⟩ 是W中任一元素;
    W := W − {⟨t, p⟩};
    PROCPARMS[p] := PROCPARMS[p] ∪ {t};
    令⟨P1, P2, ..., Pk⟩ 是过程参数的集合，元组t = ⟨N1, N2, ..., Nk⟩ 的元素映射到这个集合中的元素;
    for each p中的过程调用点s,
      在此调用点，对应某些i(1 ≤ i ≤ k) 的Pi被作为过程参数传递do begin
      令u = ⟨M1, M2, ..., Mk⟩ 是被传递给在s处被调用过程q的过程名字的集合，
        其中每个Mi是在第i个位置的过程名字，
        或者是Nj（如果Pj在第i个位置被传递），不会有其他情况;
      if u ∉ PROCPARMS[q] then W := W ∪ {⟨u, q⟩};
    end
  end
end ComputeProcParms
```

图11-17 计算过程参数元组的算法

正确性 为了证明图11-17中的过程*ComputeProcParms*产生正确的结果，我们必须证明迭代可以终结，而且，对于程序中每个过程p，它计算出的PROCPARMS(p) 满足：当且仅当存在一条调用链把参数名$N_1, N_2, ..., N_k$依此顺序传递给p的相应位置的过程参数时，$\langle N_1, N_2, ..., N_k\rangle \in$ PROCPARMS(p)。

终结性：对于每个过程p，如果v_P是参数数目的上限，N_P是程序中作为参数传递的过程名的总数，那么程序中可能的元组的总数是

$$\sum_p N_p^{V_p} \tag{11-10}$$

这个数目显然是有限的。由于元组最多只会被放到工作表中一次，并且只有有限数目的元组，while循环中的每次迭代处理一个元组，所以这个算法必定是可终结的。

充分性：假设存在这样一条调用链$p_0, p_1, ..., p_l$：p_0是主过程，且$p_l = p$，沿着这条调用链把$N_1, N_2, ..., N_k$传递给p的过程参数，并且假设$\langle N_1, N_2, ..., N_k\rangle \notin$ PROCPARMS(p)。对于这条调用链的每个过程p_i，必有一个过程名的元组t_i。不失一般性，假设$p = p_l$是调用链中第一个输入元组不在PROCPARMS中的过程。换句话说，对于满足$0 \leqslant i < l$的所有i，$t_i \in$ PROCPARMS(p_i)。如果$l = 1$，那么参数$N_1, N_2, ..., N_k$就是主过程（主过程没有参数）中被显式传递给p的过程名。这种情况下，$\langle N_1, N_2, ..., N_k\rangle$ 由初始化循环加到PROCPARMS(p) 中。这和我们的假设矛盾，所以这必定是$l > 1$的情形。所以，$t_{l-1} = \langle M_1, M_2, ..., M_j\rangle$ 必定会在算法执行的某个时刻被从工作表中取出并加入到PROCPARMS(p_{l-1}) 中。在那个时刻，$\langle N_1, N_2, ..., N_k\rangle$ 必定会由于调用p的调用点而被放入工作表中，因为被传递给p的每个N_i必定是一个显式过程名或M_n，其中n是在第i个位置传递给p的p_{l-1}的过程参数索引。

必要性：假设$\langle N_1, N_2, ..., N_k\rangle \in$ PROCPARMS(p)。那么以下两种情况必有一种成立：一是$N_1, N_2, ..., N_k$都是显式的过程名字，这种情况下，它们都在某个调用点被直接传给p（可以证明这个必要条件）；二是表中至少有一个过程参数的名字。在后一种情况中，$\langle N_1, N_2, ..., N_k\rangle$ 必定是在 $\langle\langle N_1, N_2, ..., N_k\rangle, p\rangle$ 被从工作表W中取出时被加进PROCPARMS(p) 的。因为存在一个过程q以$N_1, N_2, ..., N_k$作为参数调用p，而且q被调用时，参数为$M_1, M_2, ..., M_j$且 $\langle M_1, M_2, ..., M_j\rangle \in$ PROCPARMS(q)，所以 $\langle\langle N_1, N_2, ..., N_k\rangle, p\rangle$ 必定会被放进工作表中。由于不同过程名的元组数目是有限的（$\leqslant N^\mu$），并且每个元组最多只可能被放进工作表一次，所以我们一定能沿调用链反向搜索找到相应的显式过程名。于是这个搜索过程中所访问的过程的反向序列就是我们所要的调用链。证明完毕。

590
~
591

复杂性 上面我们看出公式（11-10）给出元组的总数。令$v_{max} = \max_p(v_p)$，N_C表示至少有一个过程参数的过程的数目，N_P表示程序中某处被传给一个过程的过程名字的数目。那么，算法的运行时间近似为

$$\sum_p N_p^{V_p} = O(N_C N_p^{V_{max}}) \leqslant O(NN^{V_{max}}) = O(N^{V_{max}+1}) \tag{11-11}$$

其中N是程序中过程的数目。在程序中任何过程的过程参数不超过一个的特殊情况下，运行时间是$O(N^2)$。在Fortran的典型应用中，因为过程参数的使用是受限制的，所以算法的运行时间不是一个重要的因素。但是，对于那些允许更复杂的过程参数使用模式的语言来说，存在运行时间为线性调用图大小或接近线性调用图大小的逼近算法[133，130]。

11.3 过程间优化

在介绍分析整个程序的基础理论后，下面我们把这种分析用在过程间优化中。

11.3.1 内联替换

我们最熟悉的过程间优化是内联替换，它在调用点用一个子程序的过程体替换，在替换后的过程体中用实参数代替形式参数。

下面举一个内联替换的例子:

```
PROGRAM MAIN
   REAL A(100)
   CALL INPUT(A,N)
   DO I = 1, N
      CALL PROCESS(A,I)
   ENDDO
   CALL REPORT(A,N)
END
SUBROUTINE PROCESS(X,K)
   REAL X(*)
   X(K) = X(K) + K
   RETURN
END
```

如果我们内联子程序PROCESS，并用A和I分别替换X和K，得到如下代码: 592

```
PROGRAM MAIN
   REAL A(100)
   CALL INPUT(A,N)
   DO I = 1, N
      A(I) = A(I) + I
   ENDDO
   CALL REPORT(A,N)
END
```

这段代码说明内联替换的几个众所周知的好处:

(1) 过程调用的开销可以消除。

(2) 过程体代码被修改为适合调用点环境的形式。例如，这个例子中索引变量I就可以放在寄存器而不用放在内存中。

(3) 可以执行一些内联替换之前不可能进行的优化。在本例中，循环可以被向量化。

内联的好处如此令人信服，以至于许多人建议把内联作为一种通用方法来替代过程间分析的方法。如果程序中所有的过程都被内联，那么普通的单个过程分析方法就可以用来优化整个程序。

但是，过度地使用内联也会导致一系列的问题:

(1) 由于替换之后的过程大小可能会增长到无法控制的程度，过度使用单模块编译器的能力，因此寻求大量的替换可能导致编译系统的瘫痪。在一个很著名的例子中，一个从系统内联产生的程序花费95小时的编译时间[83]。

(2) 由内联程序产生的目标代码可能会运行得非常慢，因为优化编译器不能很好地处理系统内联的结果程序[83]。

(3) 在内联子程序中修改的任何代码都会导致它嵌入其中的每个过程的重新编译。极端的情况下，任何代码的修改都会造成对整个程序的重新编译。

(4) 一些子过程是很难被内联的，因为在用实参数替换形式参数的过程中会产生一些问

题。下面这段难看但合法的代码可以说明这一点:

```
PROGRAM MAIN
   REAL A(100, 100)
   ...
   CALL S(A(26,2),N)
   ...
END
SUBROUTINE S(X, M)
   REAL X(*)
   DO I = 1, M
      X(I) = X(I) + M
   ENDDO
   RETURN
END
```

593

这个例子中的困难是由于在子程序S中把二维实参数A当作一维数组而引起的。我们可以在主程序的数组A和一个一维数组之间引入一个等价语句，从而的到如下代码:

```
PROGRAM MAIN
   REAL A(100,100), a$(10000)
   EQUIVALENCE (A(1,1), a$(1))
   ...
      DO I = 1, N
      a$(I+125) = a$(I+125) +N
      ENDDO
      ...
END
```

这种方法的主要问题在于失去不同行和列之间的非依赖信息，而这一点对于依赖分析恰恰又是很重要的。进一步，如果其他过程调用了S，并且A是作为参数传给该过程的，那么这种方法就无能为力了。

代替系统内联，我们推荐一种选择性的、目标制导的内联方法，这种方法用全局程序分析来决定什么时候内联是有利的[44]。

11.3.2 过程克隆

通常内联的好处是可以获得一些特定的优化效益，这些优化对于那个过程的某些调用点是可行的，但不是对所有调用点都是可行的。考虑下面的例子:

```
PROCEDURE UPDATE (A, N, IS)
   REAL A(N)
   INTEGER N, IS
   DO I = 1, N
      A(I*IS-IS+1) = A(I*IS-IS+1) + PI
   ENDDO
END
```

594

初看起来，这段代码可以向量化，事实上如果不是由于有步长大小IS=0的可能性（这种情况下，计算就相当于把N*PI的结果加到A(1)上），这段代码原本是可以向量化的。

对这个问题的明显解决方案是基于IS的不同值，把代码改写成为不同的特定版本。虽然这可以通过一个运行时进行的测试实现，但是如果在编译时知道IS在每个调用上下文中的值，

我们就可以产生程序的两个版本，一个是IS≠0的情况，另一个是IS=0的情况。对于某些可以在编译时间确定IS值的UPDATE的调用点，编译器可以用一个对克隆后的版本的调用来代替对UPDATE的调用。

克隆是一个加强常数传播作用的非常有效的途径，对于那些调用发生时是不同常数值的参数，在原过程的不同的克隆版本中，这些参数可以被视为常数。Convex应用编译器是我们所知道的惟一运用这种优化的商用编译器，而这一点正是它使用克隆的目标[213]。

11.3.3 混合优化

有些情况下，涉及到多个过程的变换可以用来在不引入内联的缺点的情况下获得内联的效益。*循环嵌入*就是这样一个混合优化，这种优化可以把一个循环从一个过程移到另一个过程中[134]。

考虑11.3.1节中的针对内联的原始例子。如果把循环交换到子程序PROCESS中，我们可以得到如下代码，这段代码可以被向量化：

```
SUBROUTINE PROCESS(X,N)
   REAL X(*)
   DO K=1, N
      X(K) = X(K) +K
   ENDDO
   RETURN
END
```

注意子程序接口已经改变：循环的上界代替循环索引成为子程序的一个参数。

已经发现在一些例子中，类似这样的过程间优化是有用的，尽管至今只有有限的证据，不具有普遍性。

11.4 管理整个程序的编译

过程间编译引发的一个问题是编译管理方面的困难。在一个传统的编译系统中，任何一 595
个独立过程的目标代码仅仅是关于源代码中该过程的一个函数。但是在过程间编译系统中，一个过程的目标代码可能依赖于整个程序的源代码。这就意味着源代码中的一处修改可能会使得每个过程都重新编译。在一个大程序中，如果在每一次小的修改后都要重新编译全部的程序，那么用户会因此而感到不快。

我们可能希望能对程序的维护阶段所产生的修改的过程间效应有所限制。这样，可以检查过程间信息流效果的全局程序分析方法对于减少重新编译的次数是有用的。

我们首先把程序编译分成两个不同的阶段：一个阶段依赖于过程间信息，另一阶段不依赖于过程间信息。第一个阶段包括了许多通常的编译任务——词法分析、语法分析、语义分析等，我们称其为*局部分析*。在完成上述工作的同时，这个阶段也会检查被处理的过程得到一些局部信息作为过程间分析的输入，这些局部信息通常包括一些过程间分析中将要用到的集合，比如在MOD分析时要用到的IMOD集合。根据这样的划分，编译过程可以用图11-18中的结构表示。

虽然，这种结构实现了过程间编译，但重新编译的问题仍然没有被解决。但是，如果中间表示被保存下来，那么对于任何没有修改过的过程，局部分析阶段就不必重新执行了。

为了更直接地处理重新编译的问题，我们按照图11-19来组织编译系统。在这种组织方案中，词法分析的任务由一个单独的系统构件来完成，这个构件可以驻留在模块编辑器或者输

入工具内。源程序中每个过程的信息（包括语法分析之后的结果）保存在中间表示中。在这个系统中，程序通过*构件编辑器*由*程序构件*规范定义，构件编辑器可以看作是一个过程名列表的编辑器。注意：一个程序构件可能十分简单，例如像一个Unix命令*make*的输入文件一样简单。*程序编译器*是负责整个程序的编译的系统构件。它读入程序中过程的所有局部信息并且完成用户需要的各种过程间分析和优化。一旦有了所有过程间信息，模块编译器实现每个过程的优化变换，其中*模块编译器*可以看作是一个复杂的优化系统。注意通过把局部分析从优化中分离出来，我们消除了程序中各个过程之间的编译顺序的依赖关系。

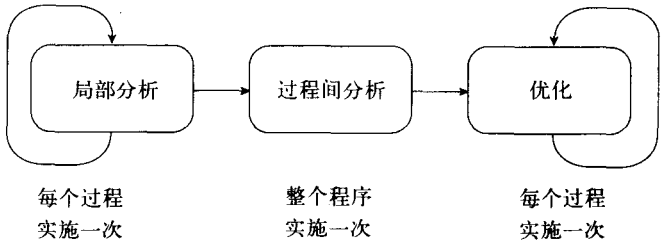


图11-18 完整的过程间编译过程

这个系统本身并没有解决重新编译的问题。它必须有一个从不同模块编译中的实施信息反馈的系统与之配合，这个信息反馈系统指出模块编译器依赖了哪些过程间分析要素。为了说明这个过程，考虑在图11-20中给出的过程间重编译MOD集的过程。

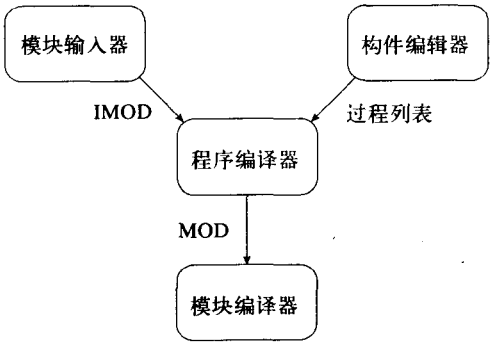


图11-19 过程间编译系统

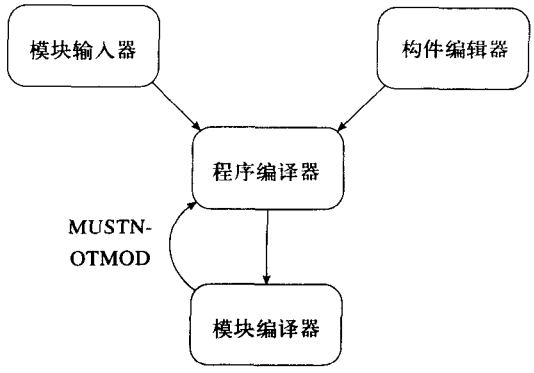


图11-20 重编译分析

我们定义集合MUSTNOTMOD(p, s)，集合中包括所有在保持过程p的代码正确的情况下，一定不会由于过程调用的副作用而在过程p中的调用点s被修改的变量。这样如果在调用点s处被调用的过程q的源代码被修改，从而在MUSTNOTMOD(p, s) 中的一个变量也被改变了，那么p必须被重新编译。我们把MUSTNOTMOD(p, s) 看做对过程p最后一次调用优化模块编译器时，这个优化模块编译器的动作的纪录。每当优化器利用“某个变量不会在调用点s处被修改”这个事实去做某种优化时，优化器就会把这个变量放入MUSTNOTMOD(p, s) 中。

在研究MUSTNOTMOD时，我们依赖于这样一个观察的结果：一个优化器只有当确信会使优化非法的事件不可能出现时，才会根据相关信息变换程序。因此，优化是依据不出现在MOD(s) 中的变量进行的，因为不出现在MOD中意味着这个变量不可能被作为调用的副作用所修改。同样，优化也将基于REF(s) 中不出现的某些变量而进行，所以MUSTNOTREF(p, s) 也将是重编译时

有用的集合。另一方面，我们将基于“一个变量必定会被一个过程调用注销”的事实进行私有化这样的优化，所以，MUSTKILL(p, s) 也将会被用于保存重编译信息。

在MOD的例子中，我们来考虑一下为了保证，程序中某个过程修改后的正确性，程序编译器必须做的事情。

（1）首先，程序编译器必须重新计算程序中每个过程调用点的MOD集合。

（2）对于程序中每个调用点$s \in p$，如果

$$\text{MUSTNOTMOD}(p, s) \cap \text{MOD}(s) \neq \emptyset$$

那么重编译过程p。

MUSTNOTMOD集合的计算依赖于模块编译器中所使用的优化，精确地收集这些信息可能会需要很大的代价。但是，在实践中，简单的近似的方法往往具有比较好的效果。下面是两个这样的计算MUSTNOTMOD的近似方法：

（1）MUSTNOTMOD(p, s) = ¬MOD(s)，其中¬MOD(s) 是p的最近一次编译的结果。当然，模块编译器不可能依赖于上次编译中任何不真的结果。 596 ~ 598

（2）MUSTNOTMOD(p, s) = ¬MOD(s)∩REF(p)。这个近似估计比上一个估计更进了一步，因为它不会因为仅仅通过过程p传递的变量的改变而重新编译。

R^{n}编程环境的设计者采用了第二个近似估计[63，50]。

11.5 小结

虽然，过程间分析在优化单处理器代码方面所起的作用有限，但是它在自动并行化系统中是十分重要的。为了充分发挥过程间分析的效果，多种分析问题必须被解决。这些问题分为两种类型：

（1）前向传播问题，这类问题确定给定过程被调用的上下文。

（2）后向传播问题，这类问题确定过程调用的副作用。

此外，过程间分析问题可以按照如下方法分类：

（1）流不敏感问题，这类问题不需要沿调用链跟踪过程的控制流图就可得到精确的解；

（2）流敏感问题，这类问题需要跟踪控制流才能得到精确的解。

我们已经证明了流不敏感的前向问题和后向问题都可以在$O((N+E)V)$ 的时间内解决，其中，N是程序中过程的数目，E是调用点的个数，V是程序中全局变量和参数的个数。

在最一般的情况下，流敏感问题已被证明是很难解决的。但是对于典型的编程语言，可以在调用图大小的多项式时间内得到较好的近似结果。在实践中，这些算法具有近似线性的复杂度。

单个变量的过程间问题通过自然的方式可以扩展为处理值域和符号值的形式。此外，还可以将其扩展用于分析数组变量的规则区域的副作用。在这两种情况下，运行时间将按一个因子扩大，这个因子和所采用的近似估计的网格的最大深度成比例。

为了实用，过程间分析系统必须尽可能减小由于一个局部的程序修改而必须被重新编译的过程的数目。用户将无法忍受对一个过程的简单修改就会造成对整个程序的重新编译的编译系统。要达到这一目的所使用的函数最好能嵌入在一个更通用的程序管理系统中。 599

11.6 实例研究

作者熟悉如下一些过程间编译系统：

（1）由Rice大学开发的向量化和并行化系统PFC[20]，使用了过程间常数传播和数组副作用分析以改进编译器的依赖分析。依赖分析的结果可以由一个称为PTOOL的特定浏览器展示出来。PFC被后来的包括ParaScope在内一些系统用作依赖测试服务程序，这些系统将在下面提到。

（2）Ardent Titan 编译器，这个编译器将在后面进一步讨论。

（3）同样由Rice大学开发的R"编程环境[63，89，90]是第一个被设计来支持过程间分析的实用编译器系统。在第11.4节中讨论的重编译分析在R"系统中得到首次尝试，其中实现了常数传播、ALIAS、MOD和REF。

（4）ParaScope[79]是R"的一个后继者，被设计为在程序并行化中使用过程间信息。FIAT过程间分析框架[135]原本是为ParaScope设计的，后来被SUIF编译器（见下文）所采用。虽然本身也会计算几个过程间问题的解，但ParaScope起初使用了PFC作为过程间数组区域分析的服务程序。

（5）D系统[6]，起初是基于ParaScope，它是为了支持“高性能Fortran”的编程而开发的。它包括一个FIAT的扩展框架，用于在用户正在给定程序的上下文进行编辑时实现过程间信息的交互式的重编译。除了传统的分析和优化，D系统还在过程间传播数组分布信息，支持“Fortran D”和“高性能Fortran”的编译[79]。

（6）Stanford的SUIF编译器的构造借助了过程间框架FIAT的帮助，FIAT是一个帮助快速建立过程间系统原型的工具[135]。这个编译器实现一个包括常数传播、数组注销分析和活跃分析在内的完整的过程间分析集合。在最近的实验中，编译器通过了来自NAS、Spec-92和Perfect基准测试程序包的程序测试，而不需要修改任何源程序。在这些总共27个程序中，16个借助利用过程间优化的结果得到了更多的并行循环。其中的4个程序仅仅是由于过程间的优
600 化而得到了非常明显的加速比[131]。

（7）Convex应用编译器[213]，它模仿了R"和ParaScope系统，是第一个实现对多个文件构成的整个程序进行过程间分析的商用系统。这个应用编译器执行了一整套的作业，包括为并行化所做的过程间分析和优化，计算过程间流敏感问题的解，基于过程间常数的克隆，以及应用数组区域分析。

除此之外，还有一系列的商用和研究系统使用过程间信息。

Ardent Titan编译器在一开始就是为过程间分析和优化而设计的。由于Titan I的主要目标应用是浮点科学计算和图形处理，所以编译器设计者感到某些形式的过程间分析是必不可少的。图形处理代码经常会使用一些小的C语言核心程序：3×3和4×4的矩阵乘法是经常用到且非常重要的。由于C语言中没有提供类似于Fortran中的对别名的限制（见第12章），所以为了能正确地向量化这些核心程序，优化器必须知道调用上下文信息。大型科学计算代码，尤其是那些可以向量化的代码，也经常是围绕一组为数不多的可向量化的Fortran核心程序而建立的；在早期编译器比较差的向量机上，如果编译器不能向量化这些核心程序，那么这些核心程序将被汇编代码所代替。

无疑Ardent Titan编译器可以向量化这些核心程序，但它能否有效地并行化这些代码有很多疑问。这些核心代码通常是一些单个循环，这意味着如果没有过程间分析，编译器必须并行化和向量化相同的循环。如果一个在向量状态下的处理器能够使内存总线饱和，那么给我们的感觉是这样的并行循环会比未并行版本运行更加慢（这一点后来被证明是正确的）。但是，许多这类核心程序是在循环中调用的——如果可以使用过程间信息，这些循环可以很容易并

且很有效地被并行化。

在确定了过程间分析是必要的之后，下一步就是要决定实现哪些形式的分析。编译器设计者选择了两种形式：

（1）过程的内联：正如上文所讨论到的，其中最直接的显然是谋求内联。

（2）用户输入：对于那些不适合被内联的并行子程序，编译器为用户提供了一组命令，用来声明带有相应存储控制的并行安全例程。在那些情况下（也仅仅在那些情况下），编译器可以并行化包含子程序调用的循环。

做内联这个决定具有很多方面的作用，其中一些是预料到的，还有一些是没有预料到的。其中主要的没有预料到的一件事是优化器中很多地方需要为内联做一些调整，尤其是当把一个C函数内联到一个Fortran函数中的时候。其中我们正确地预料到的一个作用是内联对于中间 601
表示的影响。在做内联或其他形式的过程间分析时，编译器必须多次访问编译单元（或过程）。每次访问时都扫描过程并作语法分析的做法显然是不理想的。一个更好的解决方案是使中间表示成为“永久的”不被干扰的形式，以便编译器就可以在中间代码的层次上迅速地集成其他过程的中间表示。

Ardent编译器支持一种可以把将要内联的过程做成库函数的工具。用户通常希望内联的一些过程，如图形处理中经常用到的LINPACK中的BLAS，可以用“预编译过”的形式存储在一个库中，这样即使在源程序不可用的情况下，它们也可以被很容易地内联。用户可以通过在命令行给出库文件名称的方式调用这些库函数。例如，考虑如下计算一个直角三角形斜边的简单例程：

```
REAL FUNCTION HYPOT(X,Y)
   HYPOT = X**2 + Y **2
   RETURN
END
```

如果这个例程已被编译到一个库中，并且在编译处理三角形数组循环

```
DO I = 1, N
   A(I) = HYPOT(B(I), C(I))
ENDDO
```

的命令行中给出这个库的名称，那么Ardent编译器将生成循环

```
DO I = 1, N
   A(I) = B(I)**2+C(I)**2
ENDDO
```

这个循环可以被Titan向量化和并行化，并得到近似于优化的性能。

11.7 历史评述与参考文献

过程间分析是由在20世纪70年代所发表的一系列工作中引入的。Allen给出了在没有递归的情况下如何进行程序的过程间数据流分析[13]。后来，在一篇没有发表的摘要中，Allen和Schwartz把这项技术扩展到存在递归的程序中。Spillman[252]描述了在IBM PL/I编译器中用 602
于单个文件中过程的过程间分析。这个实现的主要目标是分析指针所指向的内容。

Barth[38]是第一篇发表的讨论可能问题和必定问题的论文。Banning[37]介绍了流敏感和流不敏感问题的概念并且给出了解决这些问题的多项式时间算法。Myers[220]提出了一个针对流不敏感问题的通用算法，他证明了该算法在存在别名的情况下这些问题是NP完全补问题。

Myers[220]，Sharir和Pnueli[246]，Harrison[139]，Landi和Ryder[197]，Choi，Burke和Carini[74]，以及Hall，Murphy和Amarasinghe[136]等文献描述了流敏感分析的算法。11.2.3节介绍的由Cooper和Kennedy[85，86，87，88]提出的过程间流不敏感分析算法是11.2.4节介绍的别名分析算法[88]的原来形式。11.2.5节介绍的流敏感常数传播算法是基于Callahan，Cooper，Kennedy和Torczon[56]的工作。流敏感注销算法是在Callahan[53]提出的算法的基础上建立的，但这个特有的方程是新提出的。

过程间符号分析已由包括Haghighat和Polychronopoulos[129]，Irigoin，Jouvelot和Triolet[160]，以及Hall，Murphy和Amarasinghe[136]在内的若干研究人员讨论过。11.2.7中的处理方法依据Havlak[141]的做法。

11.2.8节讨论的数组区域分析是许多作者的研究工作的主题，包括Triolet，Irigoin和Feautrier[261]，Burke和Cytron[49]，Callahan和Kennedy[59，52]，Li和Yew[200]，Havlak和Kennedy[142]，Irigoin，Jouvelot和Triolet[160]，以及Hind等人[148]。若干研究人员提出了流敏感数组分析的算法，包括Irigoin[159]，Iitsuke[156]，Tu和Padua[263]，以及Hall，Murphy和Amarasinghe[136]。

Walter[267]，Weihl[274]，Spillman[252]，Burke[48]，以及Shivers[248，249，250]对调用图分析进行了研究。11.2.9节介绍的调用图构造算法是基于Ryder[240]中的一个算法。Callahan等人[54]证明这个算法在递归情况下收敛。Hall和Kennedy[133，130]给出精确度稍差的算法，但这个算法能在结果调用图大小的线性时间内得到近似的结果。

内联替换已有广泛的研究[101，234]。Cooper，Hall和Torczon[83]提到了内联替换的缺点。Ardent Titan编译器中的内联工具在Allen[17]中描述。Cooper，Hall和Kennedy[78，80，81]研究了过程克隆。Hall，Kennedy和McKinley[134]中研究了混合优化。这篇文章提到的与过程克隆紧密相关的若干优化技术和算法在以下论文中有研究：Wegman[271]关于过程内分析的论文，Bulyonkov[47]以及Ruf和Weise[239]关于部分计值的论文，Johnston[162]的关于APL的
603 动态编译的论文，以及Chambers和Ungar[70]关于SELF语言的论文。

11.4节介绍的程序管理的方法是在R"编程环境[63，86，87]中首先提出的。重编译分析由Burke，Cooper，Kennedy和Torczon[91，50]提出。

本章中的论述来自于Cooper等人[82]的讲座。

习题

11.1 比较过程调用图和绑定图。对于给定的程序，给出两种图大小的公式。

11.2 假设你在实现一种会改变数据数组布局的优化。下面例子说明数据变换的一个问题：

```
SUBROUTINE DATA1()
   INTEGER A(M,N)
   CALL    FOO(A)
END
SUBROUTINE DATA2()
   INTEGER B(M,N)
   CALL FOO(B)
END
SUBROUTINE FOO(T)
   INTEGER F(M,N)
   ...
END
```

如果我们决定把A的布局改为A(N,M)，但是数组B保持不变，那么我们将在子程序FOO中遇到问题，因为它的形式参数会有两种数据布局形式。你能设计一种过程间分析来检测这样的问题吗？

11.3 在类似于Pascal的程序设计语言中，作用域是可以嵌套的，其中一个函数可以在另一个函数内部定义，并且父函数的局部数据在子函数中是可见的。给出一个用于这样的语言的MOD分析算法。

11.4 Grove和Torczon[127]发现在MOD分析之后构造返回跳转函数更快。给出一个这个结果的可信的解释。 604

第12章

C语言和硬件设计中的依赖

12.1 引言

到现在为止，本书的重点在于运用依赖分析结果优化用Fortran编写的程序。虽然，这个重点是基本概念历史发展过程的自然产物，它隐含依赖仅对Fortran程序的应用是有帮助的。但是，这种含义严重地低估了依赖分析的能力，依赖分析可以运用在任何语言和任何数组和循环起作用的变换的上下文中。本章将对依赖分析的适用范围进行扩展，把它用于处理比Fortran更复杂的语言，以及运用到语言编译和优化之外的领域中。

从一开始，Fortran就被设计成可以进行高度优化的语言。Fortran最初设想的简单输入和问题领域，对优化的强烈需求，以及语言的早期发展过程，所有这些因素都导致Fortran缺乏很多更现代化的语言中常见的指针、结构等特征。本章第2节将介绍在把依赖分析扩展到处理更为现代的语言中的一些问题，例如C语言。

第3节介绍依赖分析在硬件设计中的一些应用。现代硬件设计通常是由一种基于语言的方案完成的，用类似于通用程序设计语言的高级语言来描述被设计的设备。然后，这些描述由工具（基本上是编译器）翻译成可以根据其进行实际芯片生产的低层描述。依赖分析是改善这个设计流中的工具的有用技术。

12.2 优化C语言

Fortran的设计者关注的是科学计算代码的运行时性能，在这样想法的指导下，只有当Fortran编译器生成的代码的性能是有经验程序员手写的并经过性能调整的汇编代码性能的两倍左右时，Fortran语言才会被用户所接受（见Backus[30]）。由于这个原因，Fortran语言更多地面向在科学计算程序中有用的且可以被自动优化的结构；这样，Fortran中就忽视了对于数组之外的其他数据结构的支持。

后来的语言通过引入对高级数据结构的支持扩展了程序设计应用领域的范围，但这样做的结果也增加了编译器的困难。同时，这些语言通过赋予用户更多的性能方面的控制能力而使得优化不再是程序的准备过程中所必需的。C语言就是这样的一个很好的例子。C起初就被设计成了“有类型的汇编语言”，它允许熟练的程序员可以在相对很高的层次指定想要的各个精确的硬件操作。例如，前增和后增在指令集中有很自然的对应，这一点在C语言的早期发展过程中是非常显著的。现在已经基本不用了的“寄存器”变量的声明在早期的C语言中实际上意味着“占用一个硬件寄存器”。

给定了C结构和硬件本身之间的对应关系，在很多情况下，C的原则是：不仅不需要优化，而且不希望进行优化。例如，操作系统是C语言很常用的一个领域。在操作系统中，经常通过读和写特殊的内存地址的方式来访问诸如键盘的外部设备，这样就导致类似于

```
while (! (t=*p)) ;
```

的代码。这样一个代码段可能被用来持续地轮询键盘，直到有一个字符被键入为止（其中p存

放的是对应于键盘的内存位置的地址)。由于p所指向的内容看起来是循环不变量(至少在没有声明成“volatile”的情况下是这样的),所以,优化器可能会把读取p所指的内容的操作移到循环之外。这样,系统将永远无法读取键盘的内容,导致系统挂起。类似这样的结构在C的早期应用中是非常普遍的。

由于C的这样的早期使用和原则,C在最初的目标更多地是针对可用性,而不是针对优化。C++继承了这样的趋势,在其中增加了若干简化程序开发的特征,但这是以损失优化为代价的。尽管在语言的方向上,这两种语言有着这样的趋势,但是优化对这两种语言起着至关重要的作用。机器的体系结构已经改变,指令集不再允许对C中结构的直接映射,而且,C和C++的应用领域也扩展了,例如用于不仅希望优化而且需要优化的技术计算领域。这样,尽管C的设计中没有考虑到运用优化,但是现在,优化已经成为任何一个成功的C语言编译器的
606 重要组成部分。

为了说明由于C引起的,在面向先进体系结构的优化中存在的一些挑战,考虑如下简单例程:

```
void vadd(double *a, double *b, double *c, int n) {
    while (n--)
        *a++ = * b++  + * c++;
}
```

这个例程是一个简单的向量加,如果把它写成等价的Fortran程序,是很容易向量化和优化的。但是,它的C版本的向量化或优化就很困难了,它包含如下一些问题:

(1) *指针*:利用C指针访问内存,很难分辨被访问的是哪些内存单元,尤其是在指针是作为子程序的参数进行传递的情况下。Fortran中等价的代码会使用很容易分析的数组。数组也可以在C版本中运用,但是,在书写这样的循环时,使用指针已经成为惯用的做法。

(2) *别名*:别名问题和指针问题有很密切的关系。Fortran标准保证数组以指针方式传给子程序时,不会相互覆盖,因而不会给向量化造成危害。换句话说,不同的参数表示不同的存储位置。C标准没有提供这样的保证。所以,C编译器即使能精确判断出每个不同指针所访问的内存单元,也不能直接向量化上面的循环。

(3) *副作用操作符*:在前面几章中提到,归纳变量替换是和常规的强度削减优化相反的过程。C中的副作用操作符,尤其是前增和后增操作,鼓励程序员用手工的方式削减访问多维数组所用到的计算强度。由此,C优化器就必须在类似于归纳变量替换这样的变换上进行额外的工作,以便在显式呈现数组寻址用到的下标计算。

(4) *循环*:Fortran中的DO-循环能精确地指出迭代的范围,这对于依赖分析来说是十分理想的。进而,还加上一些诸如不允许跳入循环这样的限制,从而可以使优化变得简单。C中没有类似于Fortran的对于循环的这类限制,也不容易得到依赖分析所需要的迭代范围。

这一节,我们概述C语言在高级优化中存在的这样那样问题。在大多数情况下,前面各章
607 介绍的用于Fortran的理论都可以很容易地扩展,用于解决C语言和其他类似语言所引入的问题。

12.2.1 指针

无疑,C语言对优化引入的最具挑战意义的问题是它不加限制地使用指针。指针引入了两个基本的问题:

(1) 一个指针变量可以在它的使用过程中指向不同的内存单元。确定在任何时刻通过一个指针间接访问的内存单元是一个难题。

（2）在任何给定的时间，一个给定的内存单元可以被多个指针变量访问，造成内存单元的别名。这不仅意味着不同的指针变量可以指向同一单元，而且意味着在检测C中的数组引用时，还必须检测所有可能为这个单元别名的指针变量。

换句话说，指针不仅能访问不同的变量，而且还允许同一变量被不同的名字访问。造成的后果就是依赖分析变成一个非常复杂和开销很大的过程。

依赖测试策略

在指针存在的情况下，一个C编译器如何使用第3章中介绍的依赖测试机制进行依赖测试？正如在引言部分所指出的，C程序中的习惯用法更倾向通过指针的间接引用，而不是通过数组下标表达式访问一个数组的不同元素。由于第3章中的依赖测试过程是检查带下标的数组引用对，所以编译器必须通过一些方法把每个类似“*p”这样的通过指针的间接访问转换为等价的下标数组引用。一种策略是给每个间接引用*p分配一个编译器生成的名字n，这个名字进入符号表，代表所有可以引用和*p同样的存储单元的指针引用。更明确地说，如果n是给*p分配的名字，那么每个通过p的间接引用可以被翻译成类似*(&n+e)的形式，其中e是编译时间知道或不知道的任意表达式。换句话说，我们希望把每个这样的引用写成等价的数组表达式

n[e]

注意，在p的声明作用域范围之内，名字n将伴随*p的每次出现，但是每次出现时都配以不同的索引表达式。如果p在可见程序范围内被以规则的方式更新，例如通过循环中的后增操作，那么我们可能会用类似n[a*i+c]的表达式代替*p，其中i是所在循环的循环索引变量，a是编译时的常数，c是某个在循环内不改变值的表达式。一旦所有*p的实例都有了相应的下标数组引用式，就可以按照第3章中的描述进行依赖测试了——测试每个伪数组n的引用对。 608

这个方案的问题在于可能有通过其他指针变量（例如q）的间接访问，编译器无法证明其不同于*p。换句话说，在没有其他信息的情况下，必须假设*p和*q会有重叠。每个这样的引用都必定与同*p相关的同一名字n联系。而且，我们也无法知道*p和*q相对于n的起始存储单元的偏移量之间的差，除非以编译器可见的形式，用包含其中一个赋值语句为给另一个定值。在最坏的情况下，如果指针的任何两个间接引用无法区分，那么依赖测试就会退化为标记任何引用对之间的依赖，这样，将排除所有的优化。下面几段介绍两种避免这种可能性的策略。

安全性断言

在一个没有过程间分析的分别编译系统中，指针问题是不可能解决的。因为不知道一个指针所有可能引用到的地址（只有在分析整个程序的情况下才能获得），编译器必须保守地假设通过两个不同指针的任何两个引用，或通过一个指针的引用和一个非局部数组访问，它们之间是相互依赖的。事实上，这个假设意味着所有的引用对都被假定存在依赖，所以无法进行任何优化。

在一个单独的编译系统中严格的自动分析是无法实现的，但是存在着一些有效的折衷方案。虽然，编译器必须假设指针可以不加限制地使用，但是在实际程序中，指针是以规范的、能为程序员理解的常规方式使用的（如果指针的用法过于复杂，程序员无法理解，那么这个程序很可能也无法工作）。有一个通用的但不是不可改变的规则：在一个时刻，通常只有一个指针指向一个数组，而作为函数参数传递的指针通常指向相互独立的存储单元。这类用法可以通过使用以下两个制导或编译器选项来指明：

（1）安全参数：一个函数的所有的指针参数保证指向不同的存储位置。

（2）安全指针：所有指针变量（无论是参数、局部变量或全局变量）都保证指向不同的存储单元。

这两个断言在多数程序中是成立的，并且当它们不成立时，程序员一般也知道这个情况。通过显式的断言，这两个程序制导使编译器可以区分作为参数传递的不同指针导致的间接引
609 用，这样就使编译器中有意义的依赖分析和优化成为可能。

整个程序的分析

指针的问题在单独的编译系统中无法解决，在对整个程序能够进行分析的情况下，要解决这个问题也存在困难。如果编译器能检查所有的调用过程，那么编译器就可以确定一个指针可能引用的全部的变量的集合。知道了这一点，通过两个指针的间接引用*p和*q在p和q能够指向的变量集合相交为空的情况下就不可能重叠。这就使问题就进入到可解决问题的范围，虽然现在所有的解决方案都不令人满意。

作这个分析的最简单方法使用类似于过程间常数传播的技术，尽可能地前向传播地址，以便建立给定例程中指针变量可以访问的变量集合。这样就提供了在最坏情况下可能访问的变量的边界，但是，这通常是不够的——一旦一个指针可以指向多个变量，那么由此导致的依赖通常会阻止所有的变换。人们进行了大量提高C程序中指针分析精确性的研究工作[144，206，197，274，152]，但是这个问题仍然没有一个完全满意的解决方案。结果，使用制导仍然是产品编译器中最常用的方法。

12.2.2 命名和结构

为了使用上一节中描述的框架，每个通过指针变量的间接访问必同一个“名字”或“桶”相关联，表示所有可能访问的存储单元。在Fortran语言中（或者至少在所有不包含指针语句的Fortran语言的变体中），一块内存区域可以用一个惟一的名字（数组名）所标识，这就提供了个容易计算依赖的基础。即使在使用EQUIVALENCE和COMMON语句时，数组引用也可以被静态地化简成对应于COMMON块或等价的变量集的基地址的一个偏移量。在C中，这样的化简并非是可能的，这就使建立依赖图的过程复杂化了。

考虑在C中将一个名字与通过指针引用的内存相关联的问题。这个过程必须保证任何可能访问同样内存单元的一对引用必须用同样的名字标注，以便依赖测试过程可以比较不同的引用。考虑C中如下结构的内存引用：

```
p;
*p;
**p;
*(p+4);
610 *(&p+4);
```

在第一种情况下，通过指针变量p引用的内存位置很清楚，也很容易命名——它引用程序内存中的一个单元，如果需要的话，我们可以精确地找出这个单元。

在第二种情况下，被引用的内存位置不清楚，难于命名。“*p”实际上是程序中被命名的或由其他变量生成的一些存储单元的别名。正如在12.2.1中讨论的，确定这个例子中变量p会是哪些变量的别名无疑是个必须解决的问题。但是，接下来的一个问题是使用依赖测试算法进行存储测试时用什么名字？为了提高效率，那些算法的基础都是把若干引用划分到若干个桶里（这些桶是基于变量名的）——假设排除别名，某个桶里的对象只能依赖同一桶里的对

象。“*p”不能放在p的桶里，因为p所代表的存储单元和通过这个存储单元中的地址引用的存储单元显然是不同的。假设在测试点无法惟一地确定p的值，因为如果是那样，我们就可以直接用这个值而不需要再用指针了。在无法确定惟一值的前提下，必须为测试“*p”创建一个新名字，因为测试的过程是由符号表驱动的，代表新名字的符号必须进入到符号表里，这样，通过这个指针的所有引用对就可以在一起测试。

在初次实现指针的依赖测试时，这个需求很容易被忽视，因为我们很容易假设放p的桶也可以用来同时放*p和p。同样的想法也适用于第三个例子：**p需要另一个名字来区别它所代表存储单元和p、*p所代表的存储单元。

值得注意的是，仅仅构造新的名字还不够；在第四个例子中，成功测试的关键是把对*p桶的引用和任何其他引用分开；这个引用需要对*p可能涉及的所有对象进行测试，而不需要测试其他对象。这说明我们在简化带*操作符的表达式时需要一些智能的做法，才能把它们分到合适的桶里。

处理第五个例子需要更加智能的做法：它应正确地简化为

```
p[1];
```

并且应当对p桶进行测试，而不测试任何其他对象。

由于这些复杂性，C的依赖测试框架需要比Fortran的更加强大。其中一个功能强大的表达式简化程序是十分关键的，它可以很好地理解带*的表达式的含义，把表达式转换成定义明确的规范形式。

C中的另一个复杂性来源是结构。本质上，结构的每个成员组成一个小小的数组，而且联合的存在还允许这个数组中有别名存在。这就又引入了一些关于命名的问题（“a.b”的名字是什么？）和其他一些测试中的问题。尤其是联合允许不同大小的对象的存储重叠，所以我 611
们就需要把这些对象的引用减小成可能的最小公共存储单元——字节组（bytes）或在依赖测试支持位域情况下的位组（bits）。如果一个结构成员，通过联合中的字节数组和整数数组都能访问到，那么，字节数组中的第四个元素和整数数组中的第一个元素会发生冲突，这一点是简单的测试系统所检测不到的。处理这个问题的最简单的方法是把整数引用拆分为四个单字节引用，并分别对每个进行测试。

12.2.3 循环

C中for-循环的使用限制比Fortran中的DO-循环要少得多。在C中，可以有跳入一个for循环体的跳转；如果存在归纳变量的话，归纳变量可以在循环体中被改变；循环的增量值也可以在循环内部被改变；控制循环初始化、增量和结束的条件本质上也没有任何形式上的限制。换句话说，如果有人希望在看到任意一个C的for-循环时，立刻就能找出循环变量的起始值、结束值和固定的增量，那么他会失望的。所以，尝试直接向量化C中的for-循环，通常是没有意义的。由于这样的原因，在中间语言中支持一个单独的for-循环的表示是不值得的。前端可以轻易地把输入的for-循环转换成while-循环，而且因为无论哪种循环都需要分析才能决定它是否能作为DO-循环，因此，单一的while-循环的形式可以简化分析和变换。

一个while-循环必须符合如下条件，才能被改写成DO-循环：

（1）必须具有一个明显可识别的归纳变量。

（2）在进入循环的所有路径上这个归纳变量的初值必须相同。

（3）归纳变量在循环中必须有且仅有一处增量语句。

（4）归纳变量的增量语句必须在循环的每个迭代都执行，而不应存在可能越过增量语句的路径。

（5）while-循环的结束条件必须符合DO-循环的要求。

（6）不存在可以从while-循环体外部跳到循环体内部的跳转。

显然，决定一个循环是否符合以上条件并不是可以通过简单的模式匹配来实现的，而需
612 要综合运用控制流分析和数据流分析。第2和第4条既需要数据流分析，以识别增量语句的位置，也需要控制流分析以保证赋值语句控制循环头的执行上。第6条需要简单检查控制流图。其他条件需要一些数据流分析来寻找合适的赋值语句，同时也需要模式匹配来保证找到的赋值语句满足要求。简单总结是：在建立程序的标量数据流信息之前，for-循环不能被转换成DO-循环。虽然有这些复杂因素，这一遍转换可以被相对容易地插入到前面描述的优化流中。

12.2.4 作用域和静态变量

C中的作用域规则和对与地址符号结合的文件静态变量的支持也引入必须处理的额外别名问题。作用域问题通常由前端处理，前端会为具有相同名字但出现在不同作用域的变量分别建立惟一的符号。如果采用这样的方案，那么后面的依赖测试框架不需要任何修改就能正确处理这些变量。但是，值得注意的是，用户经常错误地以“先使用后定义”的方式使用这些变量。在这种情况下，编译器会把它们视为没有定义的引用，从而会修改它们的值，于是感觉上是编译器的错误，而实际上是用户的问题。为用户解释这些问题是十分困难的，应当尽量避免。

同Fortran中的变量一样，静态变量可以被过程调用修改。但是，规则更为复杂，因为这些变量只能被可以看到它们的声明的过程所修改。编译器通常可以简单地根据作用域信息来确定静态变量是否可以修改，而这些信息应当在C的符号表里提供。

与地址操作符结合使用的静态变量会造成一些容易经常被编译器忽视的细微差别。当一个变量的地址被作为参数传递给一个过程调用时，这个过程可能在一个静态变量或外部变量中存入这个地址（在图像处理和视窗系统中很容易发生这种情况）。一旦这种情况发生，许多过程都可以借助静态变量的间接访问而修改原变量。这种情况很容易被优化器所忽视。

12.2.5 方言

在优化C语言程序的过程中，一个非常困难的问题是处理C的近20年的开发过程中形成的习惯用法的方言。使用C，也可以写出看起来很像Fortran程序的循环和数组引用，并且即使C中允许更大范围地使用别名，这些引用也很容易被优化和向量化。但是，历史上对C的使用，
613 没有人以这样的风格书写C代码。更常见的是用指针、副作用操作符和简约的代码，正如在12.2节开始给出的例子一样。优化C代码时一个主要的需求是把规则的、预期可以用Fortran改写的计算从杂乱的、质量低劣的限制优化潜力的应用中分离出来。

C的习惯用法引入的主要困难来自以下三种风格上的约定：

（1）*更愿意使用指针而不是数组*：历史上，C程序员就减少了使用数据引用的强度，他们更乐于通过指针来遍历数组，而不是通过规则的下标形式的引用。

（2）*使用副作用操作符*：副作用操作符可以减少录入上的工作量，而且在发明C的时候，这些操作符可以直接映射到普遍使用的机器的指令上。这些操作符能在表达式中间引入变量值的变化，使优化器的工作复杂化；此外，这些操作符还经常被用来把数组引用强度削减为指针。

(3) 使用地址操作符和间接引用操作符：从上两类约定自然就会使用这两种操作符。

在这三种结构中，副作用操作符最需要考虑。如果中间表示允许对副作用操作符施加注释，那么优化器的工作一定会复杂很多。在一个表达式被前向替换之前，必须检查它的副作用；在两个语句被调换位置之前，也必须检查副作用；等等。注意，做这样的检查是非常困难的，并且会引入令人讨厌的、难于发现的错误。如果中间表示不支持对副作用操作符施加这类注释，而是把副作用操作分解为几个原子操作，那么，我们将会得到更简单、更可靠的优化器。

这就是Titan C编译器所采用的方案[26]。它的前端在语法分析中消除了所有的副作用操作符，把诸如

```
x++;
```

这样的操作转换为

```
t = x;
x = t+1;
```

引入临时变量t是为了防止出现由易变变量引发的问题（见12.2.6节）。在语法分析器之后，中间表示中除了显式的赋值语句外，没有其他方式可以改变一个变量的值。

从中间表示中消除副作用操作符后，就可以用规范的形式来规范C标准的方言，但是仍然需要对一些变换做一些改进，包括：

(1) 常数传播：由于指针引用会引入模糊性，尽可能地删除这些含糊不清之处是十分必要的。所以，应当尽可能把地址操作符作为常数，并传播它们到可能的地方；把一个间接引用中的通用指针替换为真实的地址大大增加优化的机会。

(2) 表达式化简和识别：如12.2.2节中提到的，C程序中的方言使得在一个表达式中识别哪个变量是真正的“基变量”（base variable）更为困难。

(3) 转换成数组引用：本项作为前一项的一部分，在可能的情况下，直接把指针引用转换成数组引用是非常有用的。这个转换不仅需要很好的识别手段，而且需要更强的归纳变量替换手段。

(4) 归纳变量替换：在C的自然方言中，数组引用在进入优化器之前，已经通过强度削减转换成指针的间接引用——这样做已经被证明是妨碍依赖分析的。归纳变量替换必须被改进，以求对这些变量不进行强度削减。请注意，扩展副作用操作符时也需要改变归纳变量替换，因为必须识别这些不符合传统辅助归纳变量条件的形式并且删除掉。

这些改进都是很简单的，但是其中归纳变量删除需要在跟踪并且增量式更新副作用变量的使用-定义信息的方法上做一些重要的改进。因为C中的副作用操作符会造成一些归纳变量只有在一些其他归纳变量被删除之后才被“发现”，所以在这种情况下我们需要一个回溯算法。

614

12.2.6 其他问题

除了一些普遍性问题外，C中还存在一些由于历史原因和以往的惯用法引入的问题。而且，在存在这些问题的多数情况下，几乎不可能进行什么优化。

volatile（易变）

volatile（易变）变量的概念是C语言从操作系统中继承的结构之一。C程序中被声明为“volatile”的变量是不能被优化的；程序员已经声明了这个变量是特殊的变量，如本章前面提到的键盘接口，所以编译器不能通过任何途径优化对这个变量的使用。虽然这样的变量在先

进的优化系统中是可以被单独处理的，不过或许不值得这样做。优化整个函数而不影响单个
615 的volatile变量的过程是非常容易出错的（例如，由于两个volatile变量的引用顺序必须保持不变，这样在代码重排或调度时就很容易出现问题），而且，使用volatile变量的代码通常既不适合被优化，也不是理想的优化的目标。根据我们以前的经验，一个很突出的例子就是用于初始化机器的向量单元的代码。这些代码中有很多循环，优化器可以将它们很好地向量化。但是，由于代码的功能是初始化向量单元，使用向量单元来实现这段代码是不可取的。大多数使用volatile变量的代码都属于这类应用，因此更好的做法是让优化器直接跳过这些函数。

setjump和longjump

setjump和longjump是两个特殊的C库函数调用，通常由编译器直接实现，它们最初是被设计用来方便错误处理的。当调用setjump时，当前的上下文被存在一个缓冲区内。于是，longjump会被在调用链的更深的层次上调用；当用某种上下文调用longjump时，调用链上其他的调用将被跳过，动作结果好像对应的setjmp刚刚返回一样（这里要使用一个特殊的返回代码表示是longjump的返回而不是setjump的返回）。当错误的条件发生在调用链的深处时，可以用setjump-longjump对避免必须检查调用链上每个过程的返回代码。

setjump和longjump不仅会造成特殊的控制流，而且需要在setjump处保留计算的状态，而在执行longjump时恢复。当代码被优化且变量被保存在寄存器时，保存和恢复当时的计算状态是十分困难的。当代码没有被优化且变量被简单地保存在内存时，这个工作相对简单一些。所以，不优化包含setjump调用的例程是合理的、简单的和有效的做法。

varargs和stdargs

使用C和C++的一个唾手可得的便利是可以声明类似于“printf”这样的参数数目可变的函数，这些函数的参数数目依赖于调用点的环境。使用这个用法的接口是包含在头文件“varargs.h”（旧版本）或“stdargs.h”（较新的可移植性更好的版本）中的一组宏。在任何一个版本中，这些宏都会被扩展为一些制导，这些制导会告诉编译器把所有的寄存器参数保存到栈中事先设定的位置，并且使用一个指针变量通过栈来访问可变的参数表。很明显，这个指针变量是程序中若干不同参数的别名，这对于优化来说是很难处理的。因此，编译器不优化包含可变参数制导的程序是通用的一个较好的策略。

不幸的是，这个问题到这里并没有完。在C语言的早期使用过程中，流行的机器体系结构
616 中倾向使用可预测的栈结构，传递参数时只用到栈而不用寄存器。基于这种可预测性，程序员经常建立自己的可变参数表，他们取出一个传入参数的地址，并用这个地址访问其他参数——这本质上是一个自定义的varargs。如同正规的varargs一样，这样的用法造成的别名使优化无法进行，而通常最有效的方法是关掉这种过程的优化。如果向后兼容性十分重要，编译器还应当在这个过程的开始（prolog）中把所有寄存器参数保存到栈中。

需要指出的是，即使varargs过程使用栈布局的知识，在构造依赖图时正确地处理指针还是能够正确地优化varargs函数的。但是，结果依赖图很可能会十分庞大和复杂，并且限制大多数优化的应用，即使在最好的情况下，这样的函数也只能做少量的优化。所以，对于优化器来说，好的选择是忽略这些过程，而把时间和精力集中在通过较少的努力就获得更好优化效果的过程上。

12.3 硬件设计

随着硬件设计中许多设计规程的自动化，硬件设计在过去的20年中有了根本性的发展。

在20世纪80年代早期，硬件设计的典型做法是在门级或晶体管级上进行，由设计者用图形单元的形式显式地设计出门和连接这些门的线路。而今天，大多数硬件设计都是基于语言的。设计者使用类似于软件开发语言的语言通过文本的形式来描述硬件。描述的抽象级别可以在很大的范围内变化：在最低的级别，设计者仍然需要描述门以及门之间的连接（通常是通过一个网表格式列出所有的连接），而在最高的级别，设计者仅仅指出加法、乘法这样的操作，然后依赖工具将它们替换为合适的门电路。对于某种给定的设计，合适的抽象级别依赖于很多因素，包括对时间、面积和功耗的苛刻限制，设计者的设施等级，投入市场的时间上的考虑，以及用来制作设计的工艺过程，等等。但是，如果不考虑其他因素，设计的抽象级别总体上讲还是从细节实现向行为技术规程的方向发展。使这种趋势成为可能并且能持续下去的一个关键因素是编译技术能够支持高层技术规程的有效实现。

硬件设计一般可分为四级设计抽象：（1）*电路级*（现在通常称为*物理级*），（2）*逻辑级*，（3）*寄存器转换级*（RTL），（4）*系统级*[116]。在电路级，通常用示意图的形式来表达设计意图，示意图由晶体管、电容和电阻构成。物理上的布局信息在这个级别上也是很重要的。

在逻辑级，设计是用布尔等式的形式来表达的；实现层次上是门和触发器。虽然逻辑级 617
的设计是用门的形式给出的，在硅片上实现的门和设计时指定的门经常有所不同。造成这些不同的原因是：对于不同的技术，技术库不能实现一个公共的功能集合。这样，如果一个设计中指定一个AND门，但是所使用的技术库不提供AND门，那么，这个门就必须被转换成一个NAND门和一个NOT门（或者其他等价的门的集合）。*技术映射*就是实现这样转换的过程，且技术映射器也会为了时间、面积和功耗优化所得到的门。

RTL设计是依据算术部件、多路复用器（MUX）、寄存器[⊖]和存储器指定控制状态的转换和寄存器之间的数据传输。这些设计通常是指定一个控制一系列功能单元的执行的状态机，在不同周期之间保存值的若干寄存器，以及以时钟周期形式表示的每个状态的时序。*综合*是将一个RTL设计转换成等价的门和触发器的功能集合并用指定技术优化那些门的过程。

最后，在系统级，设计更多的是定义行为而不是定义实现。其中使用变量，但是这些变量不被绑定到寄存器或存储器；只对变量赋值执行顺序的等级指定时序。系统级设计通过*行为综合*被转换成可实现的设计。行为综合为实现设计而选择资源（算术部件——半加器、行波进位加法器、进位存储加法器等），把操作调度到资源当中，并给行为指定时序。

从上面的讨论中，我们可能已经比较清楚了：行为综合实际上是一个编译问题，而综合也接近于编译问题。编译器把一个问题描述从人的层次上的表示向下转换成一种在指定体系结构上能执行的表示。对于编译器来说，可用的资源有给定体系结构的处理器、寄存器和存储器；编译器使用机器的指令集把那些操作调度到上述资源中。除了没有事先给定一个目标体系结构和一组资源外，行为综合进行几乎同样的过程；它必须选择资源，并且要在额外的资源所需的空间、功耗和节省时间之间做出平衡。一旦资源被选定，两个领域内的分配和调度问题都是相似的。

不考虑设计的抽象级别，硬件开发总是涉及到两个基本的任务：验证和实现，或更加普遍的叫法是*模拟*和*综合*。*验证*是保证硬件行为描述确实是在做（设计者）希望它做的事情的过程。验证通常是通过在通用计算机上对硬件描述进行软件模拟来实现的。*实现*是自动把硬

⊖ 本节中用到的术语“寄存器”不是指处理器中能快速访问的寄存器，而是指能在一个时钟周期内保存一个值的存储单元，它和用于仅当被驱动时传递一个值的线是不同的。

618 件描述转换成可以掩模到硅片上的形式的过程。由于RTL是当今最常用的抽象层次，所以实现通常包含综合的过程。

事实上，模拟器和综合器（尤其是高层综合器）的核心基本上就是编译器。在两种情况下，优化都是这个过程的关键成分。由于模拟器包括执行一台计算机上一个硬件设备的软件来描述，所以它必定会比这个硬件设备本身慢（在数量级上）。和软件不同，硬件一旦制造出来，就不可能再变更；所以，我们不可能做到充分的模拟。同样，周期时间、面积、功耗等都是硬件设备的关键特性，并且，任何在实现上减少时间、面积或功耗的变换都会增加所得设备的价值。所以，优化就成为支持硬件设计的工具中必不可少的功能。

本节余下部分描述一些应用于硬件模拟和综合的依赖分析和高级优化技术。在描述任何特定的优化之前，我们将简单地展示一下硬件描述语言，为下面介绍优化提供必备的基础。其后，我们介绍模拟变换的基本思想以及用于高层综合的变换。

12.3.1 硬件描述语言

现今使用的硬件设计语言（HDL）主要有两种：Verilog[265]和VHDL[155]。Verilog在20世纪80年代初首先投入使用，并且，粗略地说，它可以被看作是C语言的扩展，在其中增加了描述硬件所必需的原语。虽然它最初被一家公司开发并推向市场，IEEE在1994年为这个语言制定了一个标准，这个标准现在得到多家厂商的支持。另一方面，VHDL和Ada非常类似，可以看作是对Ada语言的扩展，在其中增加描述硬件的原语。同Ada一样，VDHL由（专门的）委员会开发，并且从1984年它的推出之初至今得到了若干厂商的支持。

由于Verilog类似于C，所以，我们本节后面的例子将用Verilog来书写。在两种语言中，用于硬件描述的原语和扩展提供相同的基本功能。在Verilog中，包括如下扩展：

（1）*多值逻辑*：和C语言中每一位（bit）只能有0或1两个二进制值不同，Verilog中每一位可以有四个值：0，1，x和z。扩充的两个值是用来表示未知的或冲突的硬件状态："x"表示一个值正处于未知的状态（或者是0，或者是1），而"z"通常表示在驱动总线时的一个冲突。例如，在Verilog中被0除的结果用x表示。所有高层数据类型（例如Verilog中的"integer"，它和C中的"int"相对应）都是这样的多值位（称为"标量"）组成的向量。这样，模拟时就
619 不总是能够把算术操作直接映射到现有的机器指令上。例如，在Verilog中，两个整数相加，如果输入中有x，将使得结果也为x，所以就不能直接用简单的加法来执行这样的操作。

（2）*反应性*：硬件的一个特点是一个信号值的改变会自动地传播到与它连接的设备上，这与数据流系统的特点类似。这和依赖于严格的过程执行模型的C语言完全不同。在Verilog中，反应性行为基本上是用"always"语句和"@"操作符合起来描述的。always语句本质上是一个无限循环——它导致一个程序块的持续执行。@操作符将会阻塞执行，直至它的一个操作数的值发生指定变化。当二者结合起来，如在代码

```
always @(b or c)
  a = b +c;
```

中，无论何时任何一个输入改变，都导致（这段程序）计算出一个新的结果值。在这个例子中，b和c的任何变化将会自动引起以新的和更新a的值。在Verilog中还有其他的方法来表示这类行为，但是它们都能被规范化地归约为这种形式。

（3）*对象*：在C中，函数是抽象行为的主要机制之一，且参数是把信息传进和传出函数的主要方法。但不幸的是，这种结构还不足以充分地描述硬件。硬件中重要的对象是芯片中的

一部分；由于它是硅片上的特定区域，所以它有自己的寄存器和状态。这些语义都和函数调用的语义大不相同：函数调用代表一个单独的代码位置，可能为不同位置的调用保留状态。由于相同的硬件对象的每个实例都是芯片中完全分离的区域，在不同的对象之间没有任何状态可以保留（或者至少是不应该保留），每个对象在调用之间都保留自己的状态。由于这个差别，Verilog中数据封装的主要对象是模块，它和C++中的类在很多方面都很类似。

（4）连通性：除需要用模块表示芯片上不同的实例之外，硬件还需要实现对象之间的连通。C语言中函数的参数是将信息简单地传进和传出函数的瞬时机制，而硬件模块之间有物理的线连接，这些线可以持续地传入和传出信息。Verilog用模块的端口来支持这些功能：一个输入端口持续地从一个外部来源传入信息，而一个输出端口持续地向一个外部目标传出信息。这样，如果我们要将前面提到的加法器封装在一个模块中，我们应当做如下声明：

620

```
module add(a, b, c)
   output a;
   input b, c;
   integer a, b, c;
   always @(b or c)
      a = b + c;
endmodule
```

这个模块声明一个完成32位加法的对象（在Verilog中，整数总是32位宽）。与输入端口相连接的任何变量的修改都会自动地以新的和来更新与输出端口相连接的变量。

（5）实例化：声明对象是很少采用的，除非作为创建对象实例的一种方式。在Verilog中（同C++中一样），对象的新的实例由实例化过程产生。和C++不同，Verilog只允许静态的模块实例而不允许动态的模块实例。模块的实例化是通过声明一个新实例实现的：

```
integer x, y, z;
add adder1(x, y, z);
```

这个声明生成一个名为adder1的新加法器。变量x，y，z和模块add中端口a，b，c相连，所以，每当y和z的值有变化时，x都被隐式地更新为二者的新的和。模块add的每个实例都和硅片上不同的位置相对应，所以模块的内部变量（在Verilog中总是被声明为静态变量而不是自动变量）对于每个实例都是惟一的。因此模块的每个实例在变化之间都保留内部状态，但是模块的声明不保留任何状态。

（6）向量操作：由于Verilog是面向标量的，且把别的数据结构视为标量组成的向量或聚集类型，所以，Verilog支持简单的向量操作是很自然的。用位选择操作符（如A[1]）可以截取出单个的位；部分选择操作符（如A[3:4]——除1外不允许其他长度的跨距）可以截取出向量；向量可以连接在一起（{A[0], A[1:15]}的结果是A的最左边的16位）。

Verilog中还包含一些其他的原语和扩展，用以描述操作的定时和生成模块的上下文，但是上面描述的特征对于本章的讨论来说已经足够了。Verilog包括对所有级别的语言支持，从最低的电路级到最高的系统级。但是，它实际上通常用在门级和RTL级上。电路级对于大多数设计者来说需要太多细节，而对于较低层次的支持会影响到Verilog对于系统级的支持。例如，所有变量都有四种状态会使得系统级的模拟变慢，其中未知的值通常是不重要的。

621

从优化的观点来说，Verilog具有若干理想的特性。首先，其中本质上不存在别名。因为如果编译器要生成硬件并把它连接起来，那么编译器必须能够指明线的走向，所以语言定义无法使一个值有别名（除了一个相当明确的结构之外）。第二，在设计中，向量是被广泛使用

的，且语言限制了下标的形式，使得分析这些下标非常容易。最后，由于历史的原因，Verilog需要将整个硬件设计一次性地提交给编译器；不支持分别编译。所以，在Verilog的编译器中，有效的过程间分析是可能的。

但是，Verilog（或者更一般地讲是硬件设计）也给优化器造成了一些困难。第一个难题是由always块引入的非过程的持续的语义并且引入了时序。控制不再平滑地、可预测地从一个基本块流向另一个基本块。相反，一个变量的变化将激活另一个always块的活动，从而可能依次激活另一些always块，等等。另一个不同之处是没有循环。虽然Verilog在提供向量的同时也提供循环结构，但是，循环只用在系统级上。在较低的级别上，有循环结构执行，但是循环是通过always块和调度器而隐式表达的，而不是显式地表示在源级上。所以，在Verilog中识别可重复执行的计算比在过程化语言困难得多。最后是大小问题。硬件设计十分庞大，而且要求必须把整个硬件设计一次性提交给编译器，这样做有优点也有缺点。编译时间和存储空间的利用就成为这个领域内编译器和优化器考虑的关键因素。即使是对存储空间利用和编译时间非常注意的综合器，也需要用几天时间和几千兆字节空间来编译一个中等规模的硬件设计。在这些问题上有任何疏漏的综合器都是不能用的。

有了上述对Verilog和硬件设计的介绍，随后几小节将介绍一些在模拟和综合中要面对的挑战性问题，以及如何使用优化技术（例如依赖分析）来解决这些问题。

12.3.2 优化模拟

在模拟的过程中，编译器的目标是把硬件描述映射到用来模拟硬件设计的处理器的指令集上，好的模拟器事实上也就是这样。从这个层次的细节描述看，这个问题是个简单的编译问题，而只模拟这个层次的细节并不是一个困难的问题。但是，这种观点不仅完全没有看到模拟中真正的问题所在，而且会导致丧失很多优化机会。为了说明这一点，我们考虑如下的
622 行为级加法器：

```
module adder(a, b, c)
   input b[0:3], c[0:3];
   output a[0:3];
   always @(b or c)
      a = b + c;
endmodule
```

对于这样的输入，一个合理的编译器应当能识别出：只要不牵涉到未知的值，这样的加法可以在任何包含32位加法器的机器（当今所有机器实际如此）上作为单个指令执行——虽然结果中的28位最终不被使用，但是，实现这种加法的最快执行方式是首先进行检查，确保没有不确定的值，然后把操作映射到机器本身的加法指令。因为一旦通过复位条件的测试，硬件设计中一般不会出现不确定的值，那么这个操作将一直在稳定的计算状态下作为一个独立的加法执行。

但是，这种行为模式不是以上加法器可以采取的惟一形式。模拟器还很可能把加法器表示为

```
module adder(a, b, c)
   input b[0:3], c[0:3];
   output a[0:3];
   wire carry;
   add2 add_1(a[0:1], 0, b[0:1], c[0:1], carry);
```

```
    add2 add_r(a[2:3], carry, b[2:3], c[2:3], 0);
endmodule
module add2(sum, c_out, op1, op2, c_in)
    output sum[0:1], c_out;
    input op1[0:1], op2[0:1], c_in;
    wire carry;
    add1 add_r(sum[0], carry, op1[0], op2[0], c_in);
    add1 add_l(sum[1], c_out, op1[1], op2[1], carry);
endmodule
module add1(sum, c_out, op1, op2, c_in)
    input op1, op2, c_in;
    output sum c_out;
    always @(op1 or op2 or c_in) begin
        sum = op1 ^ op2 ^ c_in;
        c_out = (op1 & op2) | (op2 & c_in) | (c_in & op1);
    end
endmodule
```

623

这段代码实现与行为级加法器完全同样的功能。但是，如果把这段代码直接映射到典型的指令级上，那么会使这段代码的模拟要比第一个例子慢得多。原因在于涉及到的细节层次是不同的。行为级的例子可以直接映射到单个的加法指令；而本例处于更低细节层次上，显式地描述加法指令实现的所有细节（这些细节当然是硬件所需要的）。简单的模拟将只会以不同的机器指令执行所有的细节，而不会识别整个代码段是否能用一条机器指令来实现。

这两个例子说明优化硬件模拟的真正症结所在。除了要细致考虑良好调度的细节和在可能的情况下用二值逻辑代替四值逻辑外，模拟过程中的行为级代码的编译更像是一个标准的编译问题；其代码的层次更接近目标指令集的层次。所以，行为级设计的模拟是快速的——一般来说接近被编译代码的速度。但是，更低层次的设计由于插入了细节，会花费更多的模拟时间，而且会使其行为级的功能变得很模糊。所以，这样的模拟就很慢。优化模拟速度的关键是识别低层设计中实际的功能并把它映射成机器指令集中涉及细节较少的功能。

换一种说法，与其他因素相比，模拟速度和所模拟的设计的抽象级别更为相关——抽象级别越高，模拟速度就越快。由于在使用合理的优化技术后高层设计更易于有效地模拟，所以一个好的模拟器应当具有的一个主要的优化技术就是从低层设计中重新得到高层的功能性描述。这样，我们就能在所有的抽象级别上得到好的模拟性能。各个抽象级别的模拟性能很重要，因为即使所有的设计者都使用高层抽象，硬件工程师（情有可原的固执己见者）仍然会经常模拟综合的结果以确保综合工具不会出现错误。而且，硬件的特性经常会对设计强加一些低层的选择。例如，虽然一组门组合起来可以构成一个加法器，但是硬件最终要被实现成为单个位的门。所以类似于第二个例子中的硬件描述并不少见；使用这种表达形式，更易于用新技术替换其中的底层的模块。

幸运的是，前面几章介绍的技术为我们从低层设计中重新导出高层功能提供良好的基础，也提高了低层或高层设计的调度性能。下面的几小节将介绍如何运用这些技术提高模拟性能。

内联模块

从上面的一些例子中显然可见，模拟器需要的一个基本的优化是扩展模块内联的能力。

虽然数据封装是对程序员隐藏不必要细节的一种很好的编程技术，但是它具有对编译器隐藏
624 必需的优化信息的副作用。当我们的目标是重新导出一个被分割成若干模块的操作的功能时，这一点尤为明显。

幸运的是，HDL中有两个特性使模块的内联成为一个相对简单的过程：

（1）整个设计需要一次性地提交给模拟器。这意味着所有模块的源代码总是能够找到。

（2）不允许递归（递归硬件是有点令人生畏的想法）。所以，在实例图（相当于调用图的“模块”）上总可以加进一个拓扑排序，而且可以在线性时间内完成内联而且不必担心无限的递归。

决定何时结束内联需要很仔细的考虑，因为在功能单元的级别上内联通常是没用的。

我们重新回到上面的第二个例子，内联所有的模块之后的结果是：

```
module adder(a, b, c)
     input b[0:3], c[0:3];
     output a[0:3];
     wire carry, temp, temp1;
     always @(b[1] or c[1] or carry) begin
        a[1] = b[1] ^ c[1] ^ carry;
        temp = (b[1] & c[1]) | (c[1] & carry) | (carry & b[1]);
     end
     always @(b[0] or c[0] or temp) begin
        a[0] = b[0] ^ c[0] ^ temp;
        0 = (b[0] & c[0]) | (c[0] & temp) | (temp & b[0]);
     end
     always @(b[3] or c[3] or 0) begin
        a[3] = b[3] ^ c[3] ^ 0;
        temp1 = (b[3] & c[3]) | (c[3] & 0) | (0 & b[3]);
     end
     always @(b[2] or c[2] or temp1) begin
        a[2] = b[2] ^ c[2] ^ temp1;
        carry = (b[2] & c[2]) | (c[2] & temp1) | (temp1 & b[2]);
     end
endmodule
```

625 注意上面的一个结果被映射成常数0，这是硬件设计中一种抛弃一个结果的方法。

执行顺序

本书中已经反复说明，语句执行顺序对语句执行的效果具有巨大影响。如同在其他领域上一样，这个命题在硬件设计也是正确的。例如，考虑前一节中被内联的例子，假设c的值从0变为1，b的二进制值为1111。显然，所有的结果位都会连锁地从1变为0。这个简单加法器的硬件将会有效地引起这样的变化：c的改变激活第三个always块。第三个always块的执行使得temp1的值从0变为1。接下来，这个变化激活第四个块，从而使得carry的值从0变到1。这个过程继续下去，然后被激活的是第一块，最后是第二块。在真实的硬件中电流的触发和设想的激活模型是非常相符的。

结果硬件可以高效地执行，但是硬件的软件模拟执行可能不是如此高效。硬件的效率来自于每个独立的位变化的触发，只要有连接加法器的门的线路，这些触发就会很自然地发生。但是，软件就无法总是很方便地跟踪单独的位，因为这样做需要非常多的内存。相反，软件模拟

器在某些情况下不得不把每一个“变化”位和一个实体结合起来。例如，在上面的例子中，模拟器用一位来表示a中任何的变化，而不是用4个单独的位表示a[0]，a[1]，a[2]和a[3]的变化，这样做忽略了具体的发生了变化的位。这种情形和编译器数据流分析相同，数据流分析中数组不会“注销”，数组中任何元素的值发生变化都意味着数组所有元素发生变化。

在不可能有单独变化位的情况下，执行次序就成为影响上面加法器的模拟效率的主要因素。在只对数组实体而不是对它们的单个元素有变化位的条件下，考虑输入变化与以前相同时的执行过程。c中的变化（即使是只有一位发生变化）将引起所有always块被调度。假设它们以词法顺序执行，第一个和第二个always块将执行，但输出结果没有变化。当第三个块执行时，它将使得a[3]从1变到0，并且进位位temp1被置为1，激活基于它的变化。依赖于temp1的惟一的一个块是块4，它已被起始变化引起的执行调度。于是，块4的执行使a[2]和carry发生变化，这样又重激活块1。块1再通过块2引起carry的变化，然后，结果就会稳定下来。这样的次序显然不如下层硬件有效，在最坏的情况下（尤其是当carry的位被表示为一个数组而不是单个变量的情况下），其开销可能是平方级的。

问题的解决方法很明显：以基于单个数组元素的依赖图的拓扑顺序来执行各块。当这些块被拓扑重排序后：

626

```
module adder(a, b, c)
    input b[0:3], c[0:3];
    output a[0:3];
    wire carry, temp, temp1;
    always @(b[3] or c[3] or 0) begin
    a[3] = b[3] ^ c[3] ^ 0;
    temp1 = (b[3] & c[3]) | (c[3] & 0) | (0 & b[3]);
    end
    always @(b[2] or c[2] or temp1) begin
    a[2] = b[2] ^ c[2] ^ temp1;
    carry = (b[2] & c[2]) | (c[2] & temp1) | (temp1 & b[2]);
    end
    always @(b[1] or c[1] or carry) begin
    a[1] = b[1] ^ c[1] ^ carry;
    temp = (b[1] & c[1]) | (c[1] & carry) | (carry & b[1]);
    end
    always @(b[0] or c[0] or temp) begin
    a[0] = b[0] ^ c[0] ^ temp;
    0 = (b[0] & c[0]) | (c[0] & temp) | (temp & b[0]);
    end
endmodule
```

数据的变化以和词法执行同样的方向通过各块。结果是模拟过程像有单独的变化位一样高效，但没有内存的开销。

由于在这个级别的模拟中不存在显式的循环，而且也没有别名，所以基于索引的值计算依赖图是一个很简单的任务。但是拓扑排序完全不是直接就能得到的，因为依赖图中可能有环。和在Fortran的调用图中一样，这些环只能是静态的（动态环意味着无法处理的硬件）。所以，我们需要一个类似于并行*codegen*算法（图2-2）的过程来给各块排序。但是，这个过程会更复杂一些。在*codegen*中，循环携带依赖的特性保证了递归调用会打破所有的环。对于硬件没有类似的概念。所以，选择合适的边来打破环就是一个复杂的过程，是算法的关键成分。

动态调度和静态调度

627 前面一段已经显示出调度是硬件模拟中的一个主要问题。在确定性软件中有效地仿真硬件的数据流执行，这个过程包含有难于权衡先后的困难的抉择，其中最困难的是动态调度和静态调度之间的抉择。

最自然的模拟硬件的数据流执行的办法就是动态跟踪值的变化并且把这些变化沿连接的线路传递给受影响的计算单元。这种方法称为*动态调度*。如果采用这种模型，那么每当模拟器给一个对象计算一个值的时候，必须把这个新值和以前的值进行比较。如果这个对象的值发生了变化，模拟器将会调度所有与之连接的部件以使它们根据新的值进行更新。如果对象的值没有变化，模拟器就不需要做什么。同前面的例子一样明显，这样的方案（只求在“位”一级计算变化）的优点是它能精确地模仿硬件的执行，并且只计算变化了的值。缺点就是检查有无变化发生的开销很大。这个开销很可能成为整个计算时间的主要开销，当检查是在每一位都要进行时尤其如此。

另一种可以避免变化检查的开销的方案是静态调度模拟（*静态调度*），这种方案是建立在上一段引入的拓扑模型的基础上的。在这个模型之下，模拟器不需要检查对象的值是否发生了变化；相反，它会盲目地（或者用更常用的技术词汇说是*健忘地*）扫描所有的对象，并且不管是否有变化传播，都会计算所有对象的值。这样做的优点是避免所有的变化检查。纯静态调度只能用于模拟依赖图上没有环的设计。这个限制致使一些设计无法考虑（用此方法模拟），它所支持的设计风格是硬件设计师一般支持的风格。

决定动态调度和静态调度哪一个更有效的关键在于电路行为的层次。如果每个时间步内，一个电路的变化只和少数几个元素相关，那么，检查变化的动态调度可以很快地定位变化的元素，并且使电路静止下来，而不需要计算电路中没有变化的区域。另一方面，如果电路是非常活跃的，那么笼统地更新电路的所有部分会更加有效，因为这时变化检查的结果通常都为真。对于一个特定的测试向量集来说，很难事先知道一个电路（或电路的一个部分）是高度活跃的还是暂时不活跃的，所以，事先决定静态调度和动态调度哪一个更有效就更为困难。但是，使用静态分析来改进动态调度的次序总是可以提高模拟的性能的，通常能提高4到5倍。

我们的经验显示在硬件模拟中（和在并行执行中一样）使用有静态分析指导的动态调度可以得到最好的结果。本节剩余部分都假设采用这种调度策略。

合并always块

我们已知动态调度的主要开销在于变化检查的计算和传播，所以合并具有相同触发条件
628 的always块是可以实现的最简单的优化之一。合并always块可以增加每个开销单元完成的计算量，这样就可以使模拟更有效。

在*同步设计*中，合并always块尤其重要，因为同步设计中事件是由系统时钟的变化而触发的。在同步设计中，大多数模块类似于以下形式：

```
module add1(sum, c_out, op1, op2, c_in, clk)
    input op1, op2, c_in, clk;
    output sum c_out;
    always @(posedge(clk)) begin
       sum = op1 ^ op2 ^ c_in;
       c_out = (op1 & op2) | (op2 & c_in) | (c_in & op1);
    end
endmodule
```

不管输入值有无变化，时钟的上升沿都被用作更新累加和。当这个同步加法器用在前面介绍的异步加法器（只被输入值触发，和系统时钟无关）的位置上构造一个4位加法器时，在模块内联之后得到如下结果：

```
module adder(a, b, c, clk)
     input b[0:3], c[0:3], clk;
     output a[0:3];
     wire carry, temp, temp1;
     always @(posedge(clk)) begin
        a[3] = b[3] ^ c[3] ^ 0;
        temp1 = (b[3] & c[3]) | (c[3] & 0) | (0 & b[3]);
     end
     always @(posedge(clk)) begin
        a[2] = b[2] ^ c[2] ^ temp1;
        carry = (b[2] & c[2]) | (c[2] & temp1) | (temp1 & b[2]);
     end
     always @(posedge(clk)) begin
        a[1] = b[1] ^ c[1] ^ carry;
        temp = (b[1] & c[1]) | (c[1] & carry) | (carry & b[1]);
     end
     always @(posedge(clk)) begin
        a[0] = b[0] ^ c[0] ^ temp;
        0 = (b[0] & c[0]) | (c[0] & temp) | (temp & b[0]);
     end
endmodule
```

629

由于所有的块都触发同样的条件，所以代码段

```
module adder(a, b, c, clk)
     input b[0:3], c[0:3], clk;
     output a[0:3];
     wire carry, temp, temp1;
     always @(posedge(clk)) begin
        a[3] = b[3] ^ c[3] ^ 0;
        temp1 = (b[3] & c[3]) | (c[3] & 0) | (0 & b[3]);
        a[2] = b[2] ^ c[2] ^ temp1;
        carry = (b[2] & c[2]) | (c[2] & temp1) | (temp1 & b[2]);
        a[1] = b[1] ^ c[1] ^ carry;
        temp = (b[1] & c[1]) | (c[1] & carry) | (carry & b[1]);
        a[0] = b[0] ^ c[0] ^ temp;
        0 = (b[0] & c[0]) | (c[0] & temp) | (temp & b[0]);
     end
endmodule
```

的执行会更有效，因为只引起一个块的一次调度开销而不是四次调度开销。虽然调度一个块的开销不是特别高，但是如果省掉足够多这样的开销，仍然会节省很多模拟时间。

一般说来，当一个always块的前趋块的执行控制条件隐含着它本身的控制条件为真时，这个always块就可以合并到它的前趋块中。这里假设前趋块的条件用来控制被合并块的执行，并且允许如下常见例子中的合并：

```
always @(posedge(clk)) begin
```

```
    blk1
end
always @(posedge(reset) or posedge(clk)) begin
    blk2
end
```

这种情况下，合并时需要复制一些代码，得到如下的合并结果：

```
always @(posedge(clk)) begin
    blk1
    blk2
end
always  @(posedge(reset)) begin
    blk2
end
```

630

虽然把具有相同控制条件或具有隐含控制条件的always块合并是一个简单的优化，但是它并不像最初看上去那么简单。主要的问题涉及到并行语言中固有的不确定性。合并always块可能会改变设计的输出。虽然在语言的语义中这样的改变是合法的，但是在优化之后输出发生了变化通常是不会使设计者感到高兴的。很多输出上的如此改变显示设计存在有待改正的问题，但是，另外一些输出改变的麻烦对于设计者而言是没有问题的。模拟器无法分清这两种情况，保证安全的办法就是避免所有的输出改变。

为了说明块的合并是怎样改变设计的输出的，再来考虑前面同步加法器的例子，其中所有的1位加法器都作内联。如果没有合并，最后的结果依赖于always块的触发顺序。如果这些块是以程序中列出的词法顺序被触发的，模拟就会给出预期的结果，合并不会改变输出。但如果它们被触发的是以词法顺序的逆序（Verilog不保证这种情况下以这样的顺序触发，这种情况的设计是错误的），那么在原来的设计中，一个加法器将在输入改变后的3个时钟周期内不会得到正确的结果，因为把进位从第一位传播到最后一位需要很长时间。但是，块的合并会改变这个结果，因为合并后不同的块（执行顺序不确定）被合并成为一个块（执行顺序确定）。借助于不合并相互之间存在真依赖边的块，以及不合并会在一块内沿进入它的依赖边移动的那些块，上述改变是可以避免的。

同步逻辑中通常包含一个控制这个always块的时钟沿以及一个复位条件。一个非常常见的模板是

```
always @(posedge(clk) or reset) begin
    if reset then
        // 实现硬件复位的代码
    else
        // 本块真正的功能
end
```

631

由于复位代码很少被执行，所以这个块几乎总是执行posedge这个条件。对于类似这样的情况，把块分裂成为两块

```
always @(posedge(clk)) begin
    if reset then
        // 实现硬件复位的代码
    else
        // 本块真正的功能
```

```
end
always @(reset) begin
   if reset then
      // 实现硬件复位的代码
   else
      // 本块真正的功能
end
```

总是有好处的，然后识别变化的控制总是意味着每个块中的只有一个分支被执行，

```
always @(posedge(clk)) begin
   // 本块真正的功能
end
always @(reset) begin
   //实现硬件复位的代码
end
```

如此的块合并就可以把所有复合的代码放到一个很少执行到的块中，而把所有其他典型的代码移到一个经常执行的块中。

向量化always块

对于门级模拟来说，一个主要目标是重新得到高层抽象，因为高层抽象是这些门的模拟的原始意图。已知硬件设计工具和硬件设计者所做的一个主要变换是把抽象的操作分解成位操作，所以一个硬件模拟器的主要变换是把这些位操作重新组合成抽象操作。这个重新组合的过程相当于向量化。

向量化是硬件模拟中一种重要的优化，但实现的方法和编译器中的方法是不同的。其原因与硬件描述和输入语言的性质有关。我们仍然以前面给出的内联后的异步加法器为例:

```
module adder(a, b, c)
     input b[0:3], c[0:3];
     output a[0:3];
     wire carry, temp, temp1;
     always @(b[3] or c[3] or 0) begin
        a[3] = b[3] ^ c[3] ^ 0;
        temp1 = (b[3] & c[3]) | (c[3] & 0) | (0 & b[3]);
     end
     always @(b[2] or c[2] or temp1) begin
        a[2] = b[2] ^ c[2] ^ temp1;
        carry = (b[2] & c[2]) | (c[2] & temp1) | (temp1 & b[2]);
     end
     always @(b[1] or c[1] or carry) begin
        a[1] = b[1] ^ c[1] ^ carry;
        temp = (b[1] & c[1]) | (c[1] & carry) | (carry & b[1]);
     end
     always @(b[0] or c[0] or temp) begin
        a[0] = b[0] ^ c[0] ^ temp;
        0 = (b[0] & c[0]) | (c[0] & temp) | (temp & b[0]);
     end
endmodule
```

632

这个版本做了一点优化：传播进位输入值0并且标注没有使用进位输出位。如果我们去掉

所做的优化，并且对表示进位的元素做标量扩展：

```
module adder(a, b, c)
    input b[0:3], c[0:3];
    output a[0:3];
    wire carry[0:3]
    always @(b[3] or c[3] or carry[3]) begin
       a[3] = b[3] ^ c[3] ^ carry[3];
       carry[2] = (b[3] & c[3]) | (c[3] & carry[3]) | (carry[3] & b[3]);
    end
    always @(b[2] or c[2] or carry[2]) begin
       a[2] = b[2] ^ c[2] ^ carry[2];
       carry[1] = (b[2] & c[2]) | (c[2] & carry[2]) | (carry[2] & b[2]);
    end
    always @(b[1] or c[1] or carry[1]) begin
       a[1] = b[1] ^ c[1] ^ carry[1];
       carry[0] = (b[1] & c[1]) | (c[1] & carry[1]) | (carry[1] & b[1]);
    end
    always @(b[0] or c[0] or carry[0]) begin
       a[0] = b[0] ^ c[0] ^ carry[0];
       cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
    end
endmodule
```

633

现在，每块之间除了参数有细微不同外，代码都是相同的。这正是我们想要得到的，因为这些块本来就是把同一个模块内联到不同的上下文中得到的。尽管块之间有依赖，这些块可以合并为：

```
always @(b[3] or c[3] or carry[3]
         or b[2] or c[2] or carry[2]
         or b[1] or c[1] or carry[1]
         or b[0] or c[0] or carry[0]) begin
   a[3] = b[3] ^ c[3] ^ carry[3];
   carry[2] = (b[3] & c[3]) | (c[3] & carry[3]) | (carry[3] & b[3]);
   a[2] = b[2] ^ c[2] ^ carry[2];
   carry[1] = (b[2] & c[2]) | (c[2] & carry[2]) | (carry[2] & b[2]);
   a[1] = b[1] ^ c[1] ^ carry[1];
   carry[0] = (b[1] & c[1]) | (c[1] & carry[1]) | (carry[1] & b[1]);
   a[0] = b[0] ^ c[0] ^ carry[0];
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
end
```

根据依赖关系，代码可以重组为

```
always @(b[3] or c[3] or carry[3]
         or b[2] or c[2] or carry[2]
         or b[1] or c[1] or carry[1]
         or b[0] or c[0] or carry[0]) begin
   carry[2] = (b[3] & c[3]) | (c[3] & carry[3]) | (carry[3] & b[3]);
   carry[1] = (b[2] & c[2]) | (c[2] & carry[2]) | (carry[2] & b[2]);
   carry[0] = (b[1] & c[1]) | (c[1] & carry[1]) | (carry[1] & b[1]);
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
```

```
    a[3] = b[3] ^ c[3] ^ carry[3];
    a[2] = b[2] ^ c[2] ^ carry[2];
    a[1] = b[1] ^ c[1] ^ carry[1];
    a[0] = b[0] ^ c[0] ^ carry[0];
end
```

由于最后4个语句之间没有依赖，可以进一步简化为 634

```
always @(b[3] or c[3] or carry[3]
         or b[2] or c[2] or carry[2]
         or b[1] or c[1] or carry[1]
         or b[0] or c[0] or carry[0]) begin
   carry[2] = (b[3] & c[3]) | (c[3] & carry[3]) | (carry[3] & b[3]);
   carry[1] = (b[2] & c[2]) | (c[2] & carry[2]) | (carry[2] & b[2]);
   carry[0] = (b[1] & c[1]) | (c[1] & carry[1]) | (carry[1] & b[1]);
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
   a = b ^ c ^ carry;
end
```

这时已经不能完全恢复原来的加法，但是产生和的异或操作可以由一条不带掩码的机器指令实现，而不再需要一系列带掩码的单独指令。这段代码将可以很好模拟（加法器），因为一旦进入这个块，就会从块的头部一直扫描到块的尾部；在扫描的结尾，所有变量都得到了正确的更新值，而不需要其他的扫描。但是，计算进位数组的位操作会比较慢。从平均的执行情况看，向量化进位操作可能可以提高性能：

```
always @(b or c or carry) begin
   carry[0:2] = (b[1:3] & c[1:3]) | (c[1:3] & carry[1:3]) | (carry[1:3] & b[1:3]);
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
   a = b ^ c ^ carry;
end
```

从严格的语义的观点来看，这样的向量化是不对的。但是，在此处的上下文中，它是正确的，因为进位变化的传播会使块被重新激活直到进位达到一个定点值为止。在调度中使用一些静态的控制，代码可以被有效地转换成

```
always @(b or c) begin
   carry[0:2] = (b[1:3] & c[1:3]) | (c[1:3] & carry[1:3]) | (carry[1:3] & b[1:3]);
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
end                                                                    635
always @(carry) begin
   carry[0:2] = (b[1:3] & c[1:3]) | (c[1:3] & carry[1:3]) | (carry[1:3] & b[1:3]);
   cout = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
end
always @(b or c or carry) begin
   a = b ^ c ^ carry;
end
```

假设调度的结果是这样的：第一块是被变化触发的第一块，然后第二块也被触发，直到进位稳定，然后才是最后一块执行。这样的调度需要的计算量最小。另外，用依赖分析可以判定进位环将在最多三次迭代内收敛，所以进位环块可以被静态地展开，以避免对变化检查的需求。不考虑这个块是被静态调度还是被动态调度，那么关键的变换是通过分布变换把环放在一块内，而无环部分放在另一块中。

最后需要说明，一旦这些块被向量化，且在实例化中参数被前向替换成

```
always @(b or c or carry) begin
   carry[0:2] = (b[1:3] & c[1:3]) | (c[1:3] & { carry[1:2], 0 }) |
                ({ carry[1:2], 0 } & b[1:3]);
   0 = (b[0] & c[0]) | (c[0] & carry[0]) | (carry[0] & b[0]);
   a = b ^ c ^ carry;
end
```

然后可以被简化为

```
always @(b or c or carry) begin
   carry[0:2] = (b[1:3] & c[1:3]) | (c[1:3] & { carry[1:2], 0 }) |
                ({ carry[1:2], 0 } & b[1:3]);
   a = b ^ c ^ carry;
end
```

这样，用模式匹配的方法很容易就能识别这个操作序列是一个进位输入为0的简单加法。由于
636 从这段代码进位输出位和进位中间位都没有在其他地方再被使用，所以代码可以简单地改写为

```
always @(b or c) begin
   a = b + c;
end
```

重新得到设计者的原始想法。代码不能完全模拟一条单独的加法指令，因为需要一些掩码把32位结果化简为4位结果，但是它将能模拟所设计的硬件，所需时间为实际时间的倍数，而用门级表示的模拟时间要慢几个数量级。

二值逻辑和四值逻辑

和二值逻辑相比，四值逻辑会引入额外的开销，并且几乎没有人愿意买到会进入未知状态的硬件，所以模拟性能的一个很明显的改进方法是尽可能地使用二值逻辑。一个四值逻辑操作的最快实现是查表，大约需要五条指令和一次高速缓存不命中的可能。二值逻辑的模拟只需要一条指令。然而，二值逻辑的模拟被向量化后，一个这样的指令可以完成32个操作。所以，使用二值逻辑的模拟比用四值逻辑的模拟快3-5倍。

但不幸的是，使用二值逻辑不是很简单的事。尽管，除了在少数竞争总线或高速缓存初始化这类极少见的情况下，在加电之后，未知状态是不应出现的状态，但是在一个设计中分离出只能看到二值逻辑的区域也是很困难的。所有的源程序都存在并且可以使用过程间分析，这一点为发现没有未知状态的区域提供了理论基础，但是，未知状态事实上可能由任何操作引起，这通常会严重限制可以完全用二值逻辑执行的区域的数量。

但是，由于二值逻辑比四值逻辑高效很多，加之未知状态的检测只需要两条或三条指令，所以，合理的方案是检查未知状态，但是快速默认执行的二值逻辑。这种方案在所有的情况下都能得到适当的加速比，而且不失普遍性。

改写块条件

同步块的语义是：所有变化都按系统时钟信号的变化规则地更新。Verilog中描述同步块的方法使这些块看起来像在每个时钟信号都在做重复的计算。例如，前面描述的同步加法器被写成

```
always @(posedge(clk)) begin
   sum = op1 ^ op2 ^ c_in;
```

```
    c_out = (op1 & op2) | (op2 & c_in) | (c_in & op1);
end
```

637

其中，sum和c_out在时钟的每个上升沿被重新计算。这种计算是多余的模拟，在实际的硬件中不被执行。硬件的计算部件只有当输入值改变时才改变结果；时钟只是限制结果通过某个寄存器的手段，使模块外可以使用这些结果。如果输入操作数在两个时钟周期之间没有改变，那么不需要做任何重新计算；同样的结果会继续流过那个寄存器。

把以上的块改写为如下形式，在模拟器中可以具有同样的行为，从而可以节省模拟器的开销：

```
always @(op1 or op2 or c_in) begin
    t_sum = op1 ^ op2 ^ c_in;
    t_c_out = (op1 & op2) | (op2 & c_in) | (c_in & op1);
end
always @(posedge(clk)) begin
    sum = t_sum;
    c_out = t_c_out;
end
```

只要输入的值发生变化（大概比时钟信号频率低得多），计算会被执行，而且结果被输出到时钟信号的输出寄存器。大多数硬件设计者都会按照这样的风格设计，但是，当设计者不采用这种风格时，这个变换可以提高模拟的性能。

基本优化

虽然对模拟器的最有效的优化是那些试图重构高层设计意图的变换，因此而提高抽象的层次，但是本书描述的许多编译器优化也能提高模拟的性能。例如，有利于发现高层设计意图的优化对于用高层形式表达的设计就没什么作用。但是，对于无论用什么层次的设计，标准的编译器优化可以提高其高层设计的性能，尽管它们提高的程度不如重构高层抽象的优化那么明显。

除了模块内联之外（模块内联实际上是与过程内联等价的优化），全局编译器优化对于提高模拟性能不是有效的。原因很容易理解。对于always块的异步的、按变化激活的语义来说，设计的控制流图内充斥着大量的代表从一个块到另一块的潜在的控制流边。所以把重点放在always块内的优化会是最有效的策略。如果某些块已经被合并和向量化，那么所得到的结果块必定包含了大量的计算。

高层设计中有时会用到循环，所以向量化在模拟器中是很有价值的优化。由于硬件描述语言所支持的下标形式是十分有限的，这就使循环的向量化处理十分简单。其他的有用的优化包括常数传播和死代码消除。它们的作用都来自于硬件的重用：设计者会在可能的情况下重用模块和构件，这样就造成大量通用的模块在实例化的地点被剪裁。例如，常数传播可以识别出进位输入位为0的这类情况，死代码消除可以消除无用的进位输出位。

638

但是，可能最有利的优化是公共子表达式消除。即使在高层设计中，也通常会直接访问向量的某些位或者一部分，以及给这些位赋值。这就导致即使在简单的代码段中也存在大量冗余的屏蔽和移位操作。公共子表达式消除可以检测到并删除这些冗余。

12.3.3 综合优化

设计者在设计硬件使其完成某些特定功能的过程中会插入很多细节，在模拟中的目标是去掉这些细节。高层综合的目标正好与此相反；它将自动为设计者插入一些细节，这样可以

使设计者专注于功能，而对于细节的关注由综合器来完成。这个目标和标准编译器类似，因为在程序开发中程序员只需要描述他们想要的功能，而从功能到底层机器体系结构的映射则由编译器来完成。所以，很自然地期望综合的过程严重依赖于编译技术。

虽然上面的期望在很多方面是正确的，但是，综合中也面临着一个比任何标准编译器中的问题都困难的问题。按照常规（可重定目标的编译器除外），编译器是面向一个固定的充分了解的体系结构，而且通常针对获得最小执行时间这个单一目标。另一方面，高层综合面对的却不是一个固定的被充分了解的体系结构，因为高层综合的部分目标是决定适合问题的体系结构的范围以及这样的体系结构性能如何。类似地，综合的目标不是单一的。大多数情况下，得到最小的周期时间（指硬件执行的速度）是硬件设计的一个重要考虑因素，但是实现结果的面积或大小也是很重要的（比一个编译得到的可执行文件的内存覆盖区域重要得多），在某些应用中，功耗也是非常关键的。换句话说，综合器更像是一个没有确定目标体系结构的编译器，而且它具有一系列在周期时间、面积、功耗之间游离的目标。没有明确的关于目标的定义使综合成为一个很复杂的过程。相当于在编译器进行寄存器分配时，必须决定拥有的寄存器的数量，使其在不超过面积和功耗限制的条件下达到最好的性能。在固定寄存器数量的情况下分配寄存器是非常困难的。

尽管这种复杂性是综合引进的，但是仍然是一个编译器问题，通过一些确定的方式，本
639 书中所描述的技术可以为综合技术提供一个坚实的基础。这些技术不可能在近期内使高层综合完全自动化，因为优化标准没有很好的定义和范围广泛，这就需要人的指导。但是，按照时间、面积或功耗的任何度量标准，即使在没有精确测量这些因素之间相互作用的情况下，这些技术仍然能优化一种实现结果。而且，设计者希望用于综合的若干关键算法就是本书中介绍的非常适用于分析的那些算法，如矩阵乘、点积和其他向量计算。

高层综合的当前状态在很多方面反映的是并行体系结构编译器在上世纪70年代后期的状态。现在，高层综合非常关注在不考虑跨循环迭代数据流的情况下，如何在一个基本块内最大程度地开发并行性。当前综合器处理具有的增加可行的并行性的主要技术是增加基本块的大小。循环展开是处理循环的一个主要的方法，并且也是当前惟一的方法。这样的方案能为迭代次数少的循环产生很好的硬件，但是，这样的方案对于迭代次数多的循环不能获得好的结果。同在标准编译中一样，有效处理大量的循环需要在真实的循环中调度计算，在分析跨越循环迭代的值的基础上对计算做软流水调度。当然，这里还需要数据依赖分析。

正如读者所期待的，本书中介绍的优化技术都可以直接应用在面向循环计算的硬件综合中。本节的剩余部分将详细介绍编译器中的技术和优化如何应用于面向循环的算法的硬件综合中。“硬件综合”是一个包含若干领域的技术的概念。在较低的层次，设计者用门来声明计算，综合器为设计技术重新映射这些门并优化它们。在中间层次，设计者用时钟和周期时间来表示计算，并允许综合器优化这些约束条件。在最高的层次，设计纯粹用功能行为来描述，把时钟周期和周期时间的控制全部留给综合器。这也是本节剩余部分考虑的层次。在这个层次，由于设计者只指出计算的行为，所以我们用C作为表述大多数例子的语言。

本节仅试图介绍一些关于如何将优化技术用于高层综合的想法，并不涉及实现高层综合的精确算法。高层综合算法本身可以作为整本书的主题。

基本框架

在基础层次，硬件综合的问题是把诸如点积

```
for (i = 0; i<100; i++)
    t = t + a[i] * b[i];
```

640

这样的简单计算归约成一系列的门，如AND, OR, NOT等。解决这个问题的最简单的方法也很直观：把乘法转换成门，把加法转换成门，然后再尝试优化所得的结果以减小面积、缩短时间以及可能降低功耗。在理想状态下，这个方案可以实现；但在实际情况下，它是行不通的。原因是把类似乘法或加法这样的操作转换成门的处理，将会在结果中引入一些假设和限制，这些假设和限制在以后的优化中是不容易去掉的。

例如，乘法器的一种做法是跨越周期的边界把它们排成流水线，这样一个乘法就可以用若干时钟周期完成。如果乘法被转换成一个流水线乘法器，那么作为优化的一部分，后端的处理可能需要减少或者完全消除流水线，这个过程是很难自动完成的。同样，如果插入一个非流水线的乘法器，也可能会需要使它变为流水线形式。在一个乘法器中，有大量的门，可能的流水形式非常多，难以自动开发利用。所以，这个参数字也难以自动得到。流水线仅仅是一个部件中可以考虑的一个方面，还有其他很多方面。例如，加法器可以是先行进位加法器、逐位进位加法器、保留进位加法器、半加器、全加器或其他若干变形。每种在功能上都有细微的差异，这些差异是根据不同的环境优化的。自动地将一种形式的门级表示转换为另一种形式是非常困难的，因为每种形式在语义上都细微的差别，这些差别都包括加法作为一个个方面。当我们同时考虑多个部件时，复杂度就会指数增长：把加法器和乘法器合并成高效的乘法-累加部件（MAC）就是这样一个例子。虽然理想的情况是：输出的大量的门很可能被自动地优化成所有可能的变体的最合适的形式，但是，实际上这个搜索空间太大了，尤其是在多数部件都由几百个或几千个门组成的情况下。

所以，优化一个简单转换为门的策略在高层综合中是低效的。这说明得到一个好的综合结果的一个关键方面是选择部件。得到一个好的选择需要两个条件：要有一个可供选择的丰富的高度协调的部件库，以及为每个操作符或每组操作符根据它们所在的上下文选择最优的部件。对于一个有经验的设计者来说，借助于一个好的低层综合工具，构造一个好的部件库是比较容易的。但是，选择部件的最优集合却比较困难。

为了说明其中可能遇到的一些困难，请考虑在本段开始给出的点积的例子可能发生的情况。如果暂时忽略流水线部件，并且假设在每次迭代中，两个数组引用都能在内存中得到，那么最简单的方案是用一个乘法器做乘法，并和一个全加器链接完成计算。如果使用函数调用的形式来表示功能部件，那么所得到的结果是

```
t = ADD(t, MULTIPLY(LOAD(a[i], LOAD(b[i])));
```

641

假设所有的功能部件都能在输入值准备好后的1个周期内得到结果，这个简单的方案需要3个周期把一个新值加到t上（不考虑明显的流水线的可能）。但是，设计MAC部件的目的就是通过把乘法的结果直接送给加法器来加速这样的计算。使用MAC，可得到

```
t = MAC(t, LOAD(a[i], LOAD(b[i]));
```

这样，每次简单的调度更新只需要2个周期。换句话说，这个简单的计算至少有两种不同的可行选择。这个最简单的形式很可能不会被使用，主要原因是它没有采用显式的流水线调度，并且它假设从内存同时取两个操作数的能力。尤其内存的带宽是不能忽视的问题。

很有可能，部件的选择会基于类似于如下的循环展开的形式：

```
for (i=0; i < 100; i = i+4)
    t = t + a[i]*b[i] + a[i+1]*b[i+1]+ a[i+2]*b[i+2]+ a[i+3]*b[i+3];
```

这样，马上出现了很多的可能性。每次循环迭代中有5个加法和4个乘法；一个很明显的选择是使用5个加法器和4个乘法器。4个MAC也可以同样好地完成计算。但是经常使用的是多个输入的MAC，所以另一种选择是使用一个4-输入的MAC，或者使用由一个加法器连接起来的两个2-输入的MAC。如果加法可以是可再结合的（这对于定点计算来说是确实的），那么表达式可以改写为使用半加器的形式，这样比使用全加器节省时间和空间。MAC在那些空闲的周期中也可以被用于作加法或乘法。换句话说，在这个例子中以及在大多数情况下部件的选择是一个复杂的过程。

虽然这个过程是复杂的，但在编译器有一项非常类似的技术，这项技术已经得到很深入的研究。这个类似的技术就是“复杂指令集计算机”（CISC）的指令选择。在上世纪60年代和70年代，CISC体系结构是占主导地位的体系结构，它具有复杂的指令完成内存和内存之间的算术操作以及多个算术操作。在复杂指令集的体系结构中，指令选择对于性能来说远比指令调度要重要得多，所以，在编译器领域内，指令选择得到了广泛深入的研究。这些研究的结果是大量树匹配的算法，这个算法对于合理的限制和优化标准，可以产生最优的指令选择结果。这些算法可以被直接用于部件的选择[46，11，257]。

快速有效的树匹配算法的存在为高层综合和优化提供一个简单的框架。粗略地说，高层
642 综合可以被描述为选择部件，把操作分配到部件上（资源分配），然后当操作出现时调度操作。后两个步骤（像在常规编译器中那样）虽然是分别描述的，但是实际上这两个步骤是紧密结合在一起的，因为把操作绑定到部件上时就限制这个操作何时被执行。本书中介绍的类似的编译器框架执行高层优化，然后转换成低层表示、寄存器分配和指令调度。理想情况下，高层优化在综合中的位置和在编译器中是一样的：作为寄存器分配和调度的前驱提高性能并提供有用的信息。但是对于综合来说，这样的布局不是那么显而易见的，因为（1）在综合中，优化标准比编译器中多，编译器中惟一的优化标准就是执行时间，（2）综合时没有一个明确定义的目标，如像机器的指令集或寄存器。好在快速树匹配算法缓解了第二个问题，因为这些算法能快速地改为当前最优的体系结构。另外，还有一些为简化问题所做的假设，这些假设可以消除另一个问题，使得在综合中可以使用这样的布局。

需要的简化假设涉及到一些限制。低层综合有一些限制，如信号沿特定路径传播所允许的最大时间，以及特定部件所允许的面积大小等。这样的限制很明显不符合高层综合的要求，因为高层综合中只定义功能。但是，在行为级，一些形式的限制也是必需的，因为如果没有这些限制，综合的结果很快就会转向一个完全没有用的全局最小值的。例如，如果没有对空间使用上界的限制，只是一味地减少时间，那么，高层综合工具将会为计算中可以并行执行的每个操作生成一个单独的功能部件——使并行性最大化。所得到的庞大的硬件速度会很快，但是它也是浪费的且在很大程度上是没有用的。同样，如果没有时间上的最大限制而一味地减小空间，高层综合将会为每个单独的操作只生成一个功能部件，尽可能使计算串行化。这又是一个浪费的极端。最有用的结果介于这二者之间，但是如果没有某种形式的限制把优化器限制在这个中间的范围内，那么只可能返回这些极端的方案。

支持这个意图的高层限制的形式是在所需的功能部件的数量和类型上的限制——这在本质
643 上是空间的限制。功能部件的数量上的限制已经被高层设计者所理解，并且在限制可用功能

单元的条件下与最大限度地减少整体时间这个目标相结合，就能合理利用设计空间。并且，这种限制把高层优化器推回到它所熟悉的设计空间：在固定的功能资源集合上最小化执行时间，然后最小化资源的使用（如寄存器和内存访问等）。在这个空间内，高层优化之后接着部件选择、分配和调度这样的步骤是有意义的。可以预测，基于循环的变换以建立新的功能单元的需求为代价，可以单调减少执行时间。但是，这些功能单元只是当前循环中用到的同样类型的功能单元，所以，在最坏的情况下，数量上的限制只导致操作在现有功能单元的顶端单元被调度，而不会减少执行时间。

而且，综合器相对于编译器在编译时间需求上的差别，使得可以用一个更强的方案。理想情况下，编译器将优化并产生代码，然后用一些合理的输入去执行这些代码，显示结果，并用其中的反馈重复优化和产生代码的过程。但是，这个理想的模型在实际开发过程中无法使用，主要是由于所需的周转时间。但是，类似的模型在综合的框架中是可能被使用的。虽然真正“执行”结果门电路是不可能的，但是，可以优化计算，选择部件，分配和调度，这样就可以得到要求的执行时间和面积的反馈信息，然后重复优化的过程。基本上，这样的方案可以用来研究问题映射到不同体系结构上的效果。在高层综合中，这样的方案是可行的，因为在综合的过程中可以产生硬件，这就缓解了对编译一次接一次的需求。

本节的剩余部分假设采用上面描述的优化框架：在对功能单元限制的条件下执行针对减少执行时间的高层优化，然后是部件选择，分配和调度。但是，这个假设的基础是树匹配算法足够快，使这个过程在需要时可以重复以决定拒绝或考察可选的解决方案。

循环变换

正如执行顺序上的变化可能影响执行速度一样，这样的变化也可以影响功能部件的使用情况，进而影响到被综合的硬件完成一个计算所需要的时间。换句话说，循环交换，循环分布等一些基于循环的变换可以影响被综合的硬件的效率。

为了说明这个问题，我们考虑下面这个颇为常见的例子：

```
for (i = 0; i < 100; i++) {
   t[i] = 0;
   for (j = 0; j < 3; j++)
      t[i] = t[i] + (a[i - j] >> 2);
}
for (i = 0; i < 100; i++) {
   o[i] = 0;
   for (j = 0; j < 100; j++)
      o[i] = o[i] + m[i][j] * t[j];
}
```

这个代码段的输入数据集合是a，用一个FIR滤波器使之平滑后存入临时数组t，然后通过矩阵m把它重新映射到输出向量o中。在数字信号处理（DSP）领域内，这样的变换是十分常见的。假设有足够的功能单元可以一次完成加法、移位和乘法-累加计算，那么需要10 300个周期来 644
完成上述计算。

如果把循环分布，我们得到

```
for (i = 0; i < 100; i++)
   t[i] = 0;
for (i = 0; i < 100; i++)
```

```
      for (j = 0; j < 3; j++)
         t[i] = t[i] + (a[i - j] >> 2);
   for (i = 0; i < 100; i++)
      o[i] = 0;
   for (i = 0; i < 100; i++)
      for (j = 0; j < 100; j++)
         o[i] = o[i] + m[i][j] * t[j];
```

然后，重新排序循环的拓扑顺序，让没有依赖的计算首先执行，从而把初始化计算移到最开始处：

```
for (i = 0; i < 100; i++)
   t[i] = 0;
for (i = 0; i < 100; i++)
   o[i] = 0;
for (i = 0; i < 100; i++)
   for (j = 0; j < 3; j++)
      t[i] = t[i] + (a[i - j] >> 2);
for (i = 0; i < 100; i++)
   for (j = 0; j < 100; j++)
      o[i] = o[i] + m[i][j] * t[j];
```

到现在为止，还没有改善执行时间。这里存在几个循环合并的机会，只要不会增加功能单元，循环合并通常来说都是有用的。如果两个主要的循环被合并了，可以实现执行时间的改善。在当前的状态下，不可能合并这两个循环，因为合并会导致后一个循环得到错误的t[j]的值。但是，如果把后一个循环的内外层交换：

```
for (i = 0; i < 100; i++)
   t[i] = 0;
for (i = 0; i < 100; i++)
   o[i] = 0;
for (i = 0; i < 100; i++)
   for (j = 0; j < 3; j++)
      t[i] = t[i] + (a[i - j] >> 2);
for (j = 0; j < 100; j++)
   for (i = 0; i < 100; i++)
645      o[i] = o[i] + m[i][j] * t[j];
```

那么，就可以执行循环合并了，结果是

```
for (i = 0; i < 100; i++)
   o[i] = 0;
for (i = 0; i < 100; i++) {
   t[i] = 0;
   for (j = 0; j < 3; j++)
      t[i] = t[i] + (a[i - j] >> 2);
   for (j = 0; j < 100; j++)
      o[j] = o[j] + m[j][i] * t[i];
}
```

对t实施简单的标量替换得到

```
for (i = 0; i < 100; i++)
   o[i] = 0;
```

```
for (i = 0; i < 100; i++) {
   t = 0;
   for (j = 0; j < 3; j++)
      t = t + (a[i - j] >> 2);
   for (j = 0; j < 100; j++)
      o[j] = o[j] + m[j][i] * t;
}
```

然后，利用由i-循环携带的a的输入依赖得到

```
for (i = 0; i < 100; i++)
   o[i] = 0;
a0 = a[0];
a1 = a[-1];
a2 = a[-2];
a3 = a[-3];
for (i = 0; i < 100; i++) {
   t = 0;
   t = t + (a0 >> 2) + (a1 >> 2) + (a2 >> 2) + (a3 >> 2);
   a3 = a2;
   a2 = a1;
   a1 = a0;
   a0 = a[i + 1];
   for (j = 0; j < 100; j++)
      o[j] = o[j] + m[j][i] * t;
}
```

646

由于对内存和操作单元的需求大大减少了，所以以上的形式很容易在10 000周期内完成。如果使用任何合理的跨循环迭代的流水线，那么，所需的时间会大幅度下降。

对于提高执行效率有意义的基于循环的变换包括如下几种:

(1) *循环合并*: 对于两个不用相同功能单元执行不同的操作的循环，可以通过合并来实现这两个循环之间的流水线，从而第二个循环就不必等待第一个循环执行完再开始。

(2) *循环分布*: 循环分布可以用来把使用相同功能部件的操作分离开来。由于允许在每个分布后的循环中没有冲突的不同操作之间的重叠，所以，这样做可能会增加并行度。

(3) *向量化*: 当知道一个给定的功能单元要流水线执行（可以通过实际的部件选择达到）时，一种保证这个部件能被充分利用的最简单的方法就是将使用这个部件的操作向量化。上面例子中利用i-循环的输入依赖本质上是向量化j-循环。

(4) *循环交换*: 通过制造新的类似MAC的配对计算的机会，循环交换可以直接增加并行度。但是，增加并行度更有效的方式或许是增加其他变换的机会，例如上例中所用的增加循环合并。

表12-1更具体地说明使用循环变换可能改善的性能的种类。表中包含一对组合的矩阵乘法所需要的晶体管数量和以纳秒计量的执行时间。这些数据提供离散余弦变换模块（Discrete Cosine Transformation，DCT）的核心，离散余弦变换模块是MPEG和JPEG中计算密集的部件之一。本书中用到的基本循环变换和合理的调度技术相结合，可以在空间增加不大的情况下提供近10倍的速度提高。其中，最大的提高来自跨迭代的流水线——这个变换在下面“流水线和调度”一段中会进一步讨论。

表12-1 循环优化在DCT核心上的效果①

设 计	时延（ns）	面积（晶体管）
串行	37 684	25K
循环展开一次	23 552	44K
循环展开/链接	18 842	41K
循环展开/多周期	14 131	42K
流水线处理	18 842	50K
循环合并/流水线	9 476	26K
循环和FU流水线	4 040	30K

① 引自："Exploring DCT Implementations" by G. Aggarwal and D. Gajski, Technical Report UCS-ICS 98-10, Department of Information and Computer Science, University of California, Irvine, CA, March 1998, page 23。

控制流和数据流

除树匹配之外，部件选择时还需要考虑的因素之一是控制流和数据流之间的差别。在冯·诺伊曼体系结构中，数据流涉及到内存和寄存器之间的数据移动，而控制流是指由顺序执行和分支执行造成的程序计数器的变化。但是，在综合后的硬件中，数据流涉及到数据在不同功能部件之间的移动，而控制流是指在任何给定的时间步，指定哪些功能部件应当工作在什么数据上。这种情况下的数据流更为复杂：在每个时间步，必须保证正确的数据到达合适的功能单元，而那个功能单元能根据需要或者被激活或者被停止。这样的额外控制需要一个状态机，其中的状态是原始计算中的基本块，状态转换由基本块之间的控制流控制。除了引入这个状态机之外，控制流和数据流分析在硬件中和在编译器中是一样的。

但是，对于硬件中经常使用的部件，也还存在一些小问题。考虑下面简单的代码段：

```
if (a)
  o = b;
else
  o = c;
```

如果a为非0值，那么输出o被设为b的值；否则，输出o被设为c的值。这段代码应当被作为多个基本块来处理，并将其嵌入到状态机中。但是，这样的选择是常见的，有一个标准部件可以完成这个功能——mux或选择器。mux是一个3（或更多）输入的功能单元，它使用其中一个输入的值来决定剩余的输入中那个输入会通过选择器。这样，这段代码就被作为单个基本块来处理了。

mux和SSA中的选择函数φ非常相似：它们具有相同的功能。当不同的值通过不同的控制流路径汇合到一起时，φ函数用来表示对不同的值的选择；在硬件中，mux也需要同样的功能。所以，使用SSA作为内部表示具有的附带好处是容易检测到并插入mux部件。

647
~
648

还有一些其他特殊的硬件结构也需要调整。时钟的出现把时间引入到计算中，使硬件变得复杂。这样就引入四类不同的变量：

（1）*连线*：线表示功能单元之间的实际的硬件连接。从门传到线上的结果立即可见。线在时钟周期之间不需要保存值；这一点使它成为编译器中临时变量的硬件等价物。

（2）*锁存器*：锁存器表示在一个时钟周期内保持不变的值，但不能保存更长时间。锁存器可以被看成线，它的输入在周期中间可能会变化，但是它对它所驱动的功能单元的输出必须保持不变，至少保持到周期结束。C语言中没有和锁存器直接类似的成分。

（3）寄存器：寄存器对应于C中的静态变量；它们能在一个或多个周期间保留值。当一个变量的值来自其他状态时，需要用寄存器保存它；通过察看向上暴露的使用点，很容易检测到这种情况。

（4）内存：内存是寄存器的特例，它们非常大（因此不能像访问标量一样访问）而且有很长的不确定的生存期。内存对应于C语言中的数组。

线和寄存器很容易通过活跃分析检测；所用的算法类似于检测循环迭代之间活跃标量的算法。锁存器不会在一个严格的行为描述中出现，内存同数组引用一样很容易检测。数据流分析必须检测并标记每个变量的类，因为每类需要不同的硬件处理方法。

除了识别特殊的硬件结构和将控制流图转换为状态机外，硬件中的控制流分析和数据流分析和软件中的分析是相同的。在硬件中，经常会把控制流图和数据流图合并成一个单独的控制数据流图（CDFG），但是这样做并没有什么实际的好处。事实上，当这两种图分开时，状态机的生成通常会更简单。

流水线和调度

选择部件（包括mux）之后的步骤是调度这些选择的部件。像编译器中的调度和寄存器分配一样，部件选择和调度的顺序也不是明显的。事实上，硬件的调度可以很清晰地分为两个不同的任务：使用依赖信息使循环实现流水线（这可以在任何部件选择前容易地完成），以及使用类似表调度的简单调度技术做最后的调度（必须在部件选择之后完成）。

硬件流水线和软件流水线相同，类似软流水的循环处理方法可以直接用在硬件调度上。在基于循环的算法中，关键的想法是跨循环迭代调度计算以发现不同迭代的计算之间的并行性，而不是把循环体作为一个基本块来调度。这类调度可以在源级表示出来：使用所选部件的近似表示，初步给出循环中自然存在的流水线。 649

在部件选定之后，可以再次进行调度。如果从一个更早的调度中在源级实现流水线，那么可以用简单的调度把部件组合起来。如果调度过程没有充分利用能使用的所有部件，那么在允许使用的面积（即部件的数量）上施加更严格的限制后进行重新调度，以减少所用功能单元的数量。

减少访存

像在编译中一样，得到最佳执行速度的关键因素之一是把操作数以一定的速率提供给执行单元，这个速率必须足够保持执行单元全速运行。这一点在编译器为处理器产生代码时非常重要，同样，对于综合器来说，当馈给外部存储器中定制的功能单元供数时，这一点也非常重要。同执行单元的速度相比，访问存储器的速度一般是很慢的，所以最大限度减少访问存储器的次数和数量是很重要的。

幸运的是，这个问题完全同编译器的问题对应，已在第8章和第9章中透彻地讨论过。那两章中描述的用于解决这个问题的技术包括：

（1）循环交换：循环交换可以调整执行的顺序，使操作数可被重用，这样就可以用寄存器来保存被重用的操作数而不必把它们存入内存。

（2）循环合并：循环合并可以用来增加操作数在一个循环中的重用次数，由此减少取数的次数。

（3）标量替换：标量替换与其说是一项实际的技术，不如说它是一个源级变换，标量替换使编译器很容易识别出循环中被重用的内存位置，所以可以把它们保存在寄存器，而不是

存入内存。

（4）*循环分段*：内存单元可以一次返回多于一个的值——通常情况下，一个高速缓存行是能从内存中取得的最小数据单元。这就意味着循环分段、分块甚至简单的向量化都能用于增加“免费”得到的操作数的重用。

（5）*循环展开和压紧*：循环的展开和压紧可以减少内存流量，对软件和硬件都有好处。

（6）*预取*：如果可以预测取数的范围，则在使用操作数前预取可以减少取数次数。

650 这些变换以及其他一些变换在Panda, Dutt和Nicolau[223]中有很详细的描述。

12.4 小结

本章讨论了三种不同情况下的依赖信息的应用：

- *C编译*：C为高级优化提出了很多挑战，包括使用很多小过程，标准的习惯性方言法，缺乏对别名的限制，以及副作用操作符等。幸运的是，本书中介绍的分析框架可以成功地改用于C语言，这一点已在包括Ardent Titan编译器在内的几种商用编译器中都得到验证[26]。
- *用硬件描述语言模拟硬件*：在模拟过程中，一个压倒性的目标是最小化模拟时间。实现这个目标的主要策略是在可能的情况下，恢复设计者的原有意图，所以模拟的是设计意图而不是设计细节。由于硬件设计被分解成位操作，所以向量化是恢复原始抽象设计意图的一种很自然的方法。但是，always块中隐含的循环给优化环节引入一种不同的设计结构，这种结构用循环范围内的依赖图不能优化，而必须使用面向全局的依赖图。
- *从高层描述综合硬件*：综合是把用高层抽象的方式书写的硬件设计翻译成可以用低层硬件实现的过程。从基本上讲，高层综合是一个编译问题，尽管由于没有固定的目标体系结构而且优化准则多变的原因，高层综合会更加复杂。部件选择大大简化了这个问题，并且可以用类似于CISC处理器中选择代码的树匹配技术来实现。用跨循环迭代调度取代仅仅在单个块内调度是使用依赖分析的主要优点，但是还有其他优点，包括得到更好的周期次数，增加内存的效率，以及减少临时变量的需求等。在这个领域，还有许多困难的问题需要解决，但是，高级优化已经在一些关键的面向循环的基准测试程序中显示出对性能的显著的提高。

这里给出的决不是依赖分析所有可应用领域的概要，但是，以上的内容至少说明在并行机和
651 向量机的Fortran应用的优化之外，依赖分析在很多情况下还是非常有用的。

12.5 实例研究

12.2节所介绍的优化C程序的方案被Ardent Titan编译器采用[26]，这个编译器中，在C和Fortran的前端之间共用中间表示，优化器，代码生成器，允许这两种语言完全访问Titan的并行和向量部件。优化器被设计成可重定目标的，可以为不同的体系结构产生代码。这个编译器不仅在操作系统核心程序和视窗应用程序领域取得成功，而且在科学计算和图形处理领域也很成功。这个编译器采用了和以往大多数C编译器非常不同的方案，它开发了高层中间表示，并对中间表示进行分析以寻找优化机会。但是，尽管存在着这些不同，把现存的应用程序引入到这个编译器上仍然是比较容易的。这个编译器的代码必须经常调整的一个主要领域是使用副作用操作符时，按照语言定义操作的结果是未定义的（例如`a[i]=i++;`）。如果不考虑结果未定义，用户已经逐渐把PCC（Portable C Compiler）[161]的输出视为正确结果，而且希望所有的编译器也能得到同样的结果。由于Titan的作者之一Steve Johnson也是PCC的作者，所

以这个问题经常引起激烈的争论。

基于语言的硬件开发领域，尤其是在高层综合领域，一直是一个研究课题，还没有一个被普遍接受的方法。本章中提到的方法可提高由高层描述产生的硬件的质量，但是并没有提供一个全面的解决方案。然而，实际上似乎可以肯定，最终会证实获得成功的将是面向循环的方案而不是面向单个基本块的方案。

12.6 历史评述与参考文献

在上世纪80年代，出现了第一批商用向量化C编译器——Convex向量化C编译器[212]和Ardent Titan C编译器[24]。在这之前，虽然有少数研究已经涉足了C的向量化和并行化研究，但绝大多数研究还是集中在Fortran上面。由于这两个C编译器的出现，使得很多研究人员把研究的重点集中在C的指针和别名上面[144，206，197，274，152]，但是，到现在为止，仍然没有各个方面都令人满意的解决方案。 652

习题

12.1 为下面的循环构造依赖图：

```
for (i = 0; i<100; i++)
   a[i] = *p++;
```

12.2 下面的C循环能够向量化吗？说明能或不能向量化的原因。

```
for (i = 0; i< 100; i++)
   a[i]= b[i++];
```

12.3 在下面代码段中，给出a[0]和a[1]的正确值：

```
i = 0;
a[0] =0;
a[1]=0;
a[i]=i++;
```

12.4 事实上，所有的体系结构都支持寄存器的左移和右移操作。但是，有些体系结构并不是直接支持这些操作，而是支持“extract”和“deposit”操作，这两个操作分别表示从一个寄存器中取出一组连续的位，和把一组连续的位插入到一个寄存器中。例如，extract(a,1,3)会恰当地移位以返回a中的第1位到第3位（a[1:3]），而deposit(a,0,2,extract(a,1,3))将会把a的第0位到第2位设为指定的值，这样实质上就是执行一个左移操作。用向量Verilog的表示法，如何写出deposit(a,0,2,extract(a,1,3))的等价形式？

12.5 给定一个小于等于4的正值s，写出执行4－位变量的左移操作的Verilog代码。这个代码可以被分解成1位的操作吗？

12.6 给定一个大于等于－4的负数s，写出执行4－位变量的右移操作的Verilog代码。这个代码可以被分解成1位的操作吗？

12.7 你能把习题12.5和12.6中的结果合并为一个既能执行左移操作又能执行右移操作的移位器吗？ 653

第13章

编译数组赋值

13.1 引言

由于Fortran 90的出现，数组赋值和数组表达式成为重要的语言结构。虽然Fortran 90的定义历经了13年，而且是以一种缓慢的速度被广泛接受的，但是现在它已经开始取代Fortran 77，成为科学计算编程所选择的语言。

科学计算界没有立即接受Fortran 90的一个原因是成熟的编译技术形成缓慢。这个语言增加了许多新的特性，给实现者带来了现实的问题。因此，为了得到与Fortran 77编译器水平相当的性能，Fortran 90编译器必须做更多的工作。

单单是实现一个Fortran 90的新特性——数组赋值语句的编译工作就是复杂的，本章将对此作简单介绍。虽然该语句最初是为了提供一种直接描述并行操作或向量操作的机制，但是它的实现必须仔细考虑用于执行该语句的特定硬件。特别在单处理器上实现数组赋值时，这个语句必须被转换为一个标量循环。这个被称为标量化（scalarization）的转换必须以一种可以有效使用存储层次结构的方式实现，而不管最终的目标体系结构是什么。

乍看起来，标量化好像是简单的——简单地用一个DO-循环替换每个数组赋值，循环的索引变量在数组下标的某个分量的范围内进行迭代。但是，两个原因使得问题复杂化。首先，我们希望避免大量的存储空间需求和数组拷贝操作，当我们必须使用一个由编译器生成的大小与被赋值的数组元素空间大小一样的临时数组时，这种操作是必要的。而且，上述简单的标量化策略会为每个数组赋值语句产生一个单语句循环，所以一个重要的优化是尽可能多地将这些循环进行合并，以提高寄存器重用的数量。如同我们将看到的，依赖理论在实现这些策略中扮演一个关键角色。 655

总之，一个良好的标量化方案除了产生一个正确的变换外，有两个基本的目标：

（1）避免需要生成大小无界的临时数组

（2）生成的标量化循环可以用第8章和第9章中介绍的技术进行优化，以得到良好的存储层次结构的性能

下一节介绍正确的标量化方法和由这个处理过程引入的临时数组最小化导致的问题。虽然这些结论是从对标量机器编译Fortran 90得到的，但如同13.5节中介绍的，这些结论可以方便地更改到向量寄存器机器上。

13.2 简单的标量化

标量化的一个显而易见的策略是将任何单独的向量语句用一个在向量的元素上迭代的循环替代。考虑下面的循环：

```
A(1:200) = 2.0 * A(1:200)
```

一个标量的实现如下：

```
S1   DO i = 1, 200
```

```
S2      A(i) = 2.0 * A(i)
    ENDDO
```

此处的S_1这个循环，从1到完成操作的长度进行迭代。标量语句S_2在任何一个标量机器上都可以用一个单元素的操作实现。请注意，从向量操作的迭代空间到结果循环的迭代空间之间有一个直接变换。

回想Fortran 90的语义：向量语句的执行要保证对语句的所有输入的读取必须在语句的任何结果被写回之前完成。这个要求不仅是自然的，而且也确切反映向量寄存器机器上的向量部件特点，即数组片段在这些部件上的读取或写回通过单个操作完成。但是，这一点与串行循环的语义是不一样的，串行循环是一个迭代接着一个迭代执行的，在此基础上，读/写操作是交替进行的。当与原例下述变形类似的数组语句进行标量化时，上述的不同是会导致问题
656 的：

```
A(2:201) = 2.0 * A(1:200)
```

此处，简单的变换策略生成

```
DO i = 1, 200
   A(i + 1) = 2.0 * A(i)
ENDDO
```

这个版本的程序得到的结果与原始程序不同，因为每个迭代中计算结果的值在下一次迭代中被使用，而在向量语句的实现中，赋值号右端引用的值必须是任何写操作发生之前原有的值。

这个例子显示在标量硬件上实现向量语句不是轻而易举的。问题的缘起是因为寄存器只能保留有单个的值而不是在赋值号右端所需的所有值。这个例子中显示的这类错误称为标量化失效，即语句的含义在标量化过程中被改变了。

图13-1给出实现从一个一维数组赋值语句到一个简单的标量循环的简单变换过程*Simple Scalarize*的非形式说明。在输入程序中，一个向量操作的范围由语句的下标中见到的三元组或向量说明符说明，其形式为（下界：上界：增量）。标量化是通过检查赋值左端的三元组决定的。

```
procedure SimpleScalarize(S)
  // S是要被标量化的语句
  令V0=(L0:U0:D0)是S左端的向量迭代说明符;
  // 生成标量化循环
  令i是新循环的索引变量;
  生成语句 DO i=L0, U0, D0;
  for each S中的向量说明符V=(L:U:D) do
    用 (i+L-L0)代替V;
  生成一个ENDDO语句;
end SimpleScalarize
```

图13-1 单个向量操作的标量化

图13-1中介绍的方法如何导致标量化失效是显而易见的。由于原始的语句要求所有写操作要在所有右端的值被读取后再执行，而标量操作的顺序可能是交替完成读取操作和写操作的，所以标量操作的顺序可能不能保证原来的向量语句的语义。因此，如果一个标量操作要

写一个在稍后的操作中要读取的值，将产生一个标量化失效。幸运的是，借助依赖分析，这种情况可以被精确地检测到。

原理13.1 一个向量赋值语句产生标量化失效，当且仅当标量化的循环携带一个真依赖。

为了说明这个原理的正确性，请回忆，数组赋值的执行是在完成这个语句左端的任何写操作前读入所有的右端的值。于是仅有的可能失效是标量化的循环中某次迭代写了一个内存单元，而这个单元的值在后面的迭代中被读取。这是一个精确的循环携带的真依赖的定义。

这个原理提供一个精确的机制，将导致标量化失效的向量语句从其他不导致标量化失效的向量语句中区分出来。由于由标量化循环携带的真依赖导致这类失效，这种依赖被称为标量化依赖。

为了保证程序的正确执行，编译器必须不产生标量化依赖。通过使用临时存储这种简单的方法就可以达到上述的目标[279]，如下面的例子中显示的:

```
A(2:201) = 2.0 * A(1:200)
```

这个语句可以被分裂成两个向量语句

```
T(1:200) = 2.0 * A(1:200)
A(2:201) = T(1:200)
```

（其中，T是编译器生成的临时数组）然后利用图13-1的*SimpleScalarize*进行标量化，得到下面的代码:

```
DO i = 1, 200
   T(i) = 2.0 * A(i)
ENDDO
DO i = 2, 201
   A(i) = T(i - 1)
ENDDO
```

由于T是一个新的数组，它的存储空间不会和程序的数据数组重叠，所以这个标量化的方法确保避免一个标量化依赖。换句话说，由于整个结果向量存放在一个临时数组中，然后只在所有右端的操作（包括读操作）完成之后才将结果向量拷贝到结果数组中，所以标量化失效被避免了。

657 ~ 658

不幸的是，由于两个原因，这个方法的代价是高昂的:

- 由于T（它的大小可能在编译时刻难以预测）必须在内存中，此方法需要大量增加内存需求。而且，动态分配数组T的开销可能是很大的。
- 该方法需要对结果向量中的每个元素进行存储，然后读入，然后再存储。与不需要临时数组的访存相比，这对于每个向量元素增加两次额外的存储操作。如果可以将临时数组元素保留在寄存器中，则会取得好得多的效率。

由于这些缺点，使用完全长度的临时数组存储最好作为最后一个手段。

请注意，在可能的时候，临时数组存储的缺点可以通过后面的“合并”循环避免[279]。两个循环可以安全地合并的条件与前面的原理中指出的单个语句标量化安全的条件完全一样——也就是说，如果两个循环可以被合法地合并，则原来语句应该已经被安全地标量化了。所以，我们更喜欢用一种直接的测试以便在可能的情况下避免生成临时数组。注意，循环合并作为一

种标量化优化仍然是有效的。这个题目将在13.6节讨论。

图13-2给出一个改进的标量化过程*SafeScalarize*，通过使用原理13.1中的测试避免产生标量化失效。首先，对于每一个向量语句计算出一个完整的依赖图（不仅是对标量化的循环）。如果这个语句本身没有标量化依赖，则用*SimpleScalarize*对它标量化；否则，使用临时存储以保证安全。虽然从图13-2中不能明显看出为什么需要构造完全的依赖图（且不仅是惟一可能导致标量化失效的真依赖），但后面几节将清楚说明这一点。

```
procedure SafeScalarize(S)
  if S没有标量化依赖
  then SimpleScalarize(S);
  else begin
    令T是一个新的临时数组，其长度足以容纳由S计算得到的所有元素;
    借助T将S分裂为S1和S2，使得
      在S1中，T的值由S的右端赋值，
      而在S2中，T的值赋给S的左端
    SimpleScalarize(S1);
    SimpleScalarize(S2);
  end
end SafeScalarize
```

图13-2 安全的标量化

*SafeScalarize*保证对每一个向量语句生成正确的标量化程序。但是，由于使用大的临时数组而降低性能，所以在可能的情况下最好避免执行else子句。下一节介绍能够不使用临时存储就可以消除标量化失效的变换。

13.3 标量化变换

只要简单标量化的结果会产生一个由标量化循环携带的真依赖，图13-2中的*SafeScalarize*过程采取的就是代价很高的解决方案。这种代价通常可以通过代码变换消除这种依赖而避免。下面的几小节介绍几种可以用于消除标量化依赖的变换。虽然得到正确的程序意味着消除程序中真依赖看起来可能是违反直觉的，但是请记住这个问题中的真依赖是由标量化算法不正确地引入的。在实施这些变换中，我们仅仅是恢复程序原来的含义。

13.3.1 循环反转

考虑这个向量语句

```
A(2:256) = A(1:255) + 1.0
```

图13-1中的*SimpleScalarize*对于这个例子将产生一个标量化失效，这是因为每个输出将被存储在这个标量循环的下一个迭代中要输入的位置。但是，在解决这个问题时可以容易看出一个简单而优雅的方法——我们简单地从后向前执行这个标量化的循环，消除循环携带的真依赖。

```
DO i = 256, 2, -1
  A(i) = A(i - 1) + 1.0
ENDDO
```

在这个例子中，在左端存储的数组元素改写已被读出的数组元素——这是我们希望的结果。这个称为*循环反转*[270]的常见的变换在第6章中作过介绍。

为了将循环反转应用于标量化，一个向量语句首先被标量化，然后标量化的循环被反转。从上面的讨论中，容易看出循环反转将一个标量化的循环携带的真依赖转换为反依赖。所以，它对于标量化看起来是理想的。

659 ~ 660

可惜这个变换将反依赖也映射为真依赖。结果，当标量化的循环也携带有一个反依赖时，循环反转将不能更正一个标量化失效，如同向量语句

```
A(2:257) = (A(1:256) + A(3:258)) / 2.0
```

生成一个直接的标量化循环

```
DO i = 2, 257
   A(i) = (A(i - 1) + A(i + 1)) / 2.0
ENDDO
```

这个循环既携带有一个真依赖（由右端的第一个操作数引入的），又有一个反依赖（由第二个操作数引入的）。如果标量化循环被反转，

```
DO i = 257, 2, -1
   A(i) = (A(i - 1) + A(i + 1)) / 2.0
ENDDO
```

现在第一个操作数引入一个携带的反依赖，第二个操作数引入一个携带的真依赖。通过简单的观察，从这个例子得到的教训可以总结为：循环反转可以消除由一个循环携带的标量化失效，当且仅当循环不携带反依赖。这个结论的证明留给读者作为一个习题。

这个观察的含义是，当标量化循环同时携带有真依赖和反依赖两种依赖时，需要更复杂的变换技术。输入预取是这类变换的一种。

13.3.2 输入预取

让我们更细致地考虑前一小节中的例子：

```
A(2:257) = (A(1:256) + A(3:258)) / 2.0
```

由*SimpleScalarize*生成的标量化循环同时含有一个真依赖和一个反依赖，所以无论循环迭代采用何种执行次序，都有一个标量化失效。问题是输入向量与输出向量向重叠，而且输入向量间相互重叠。这导致简单的标量化循环

```
DO i = 2, 257
   A(i) = (A(i - 1) + A(i + 1)) / 2.0
ENDDO
```

存储到左端的第一个元素成为对下一次迭代的输入。

661

由于这个标量化依赖的范围是一次迭代，我们可以设法将这个结果保存到一个临时存储空间中，直至下一次迭代的输入读取完成。令T1是用来保存第一个输入数组的一个元素，并令T2是用来保存输出数组的一个元素。在这个上下文中对T1和T2的简单使用如下所示：

```
DO i = 2, 257
   T1 = A(i - 1)
   T2 = (T1 + A(i + 1)) / 2.0
   A(i) = T2
ENDDO
```

虽然这个循环也有同样的标量化问题，我们现在可以通过把对T1的赋值正好移到前一次迭代中T2存储位置的前面而改正。重叠于是被消除。

```
T1 = A(1)
DO i = 2, 256
   T2 = (T1 + A(i + 1)) / 2.0
   T1 = A(i)
   A(i) = T2
ENDDO
T2 = (T1 + A(258)) / 2.0
A(257) = T2
```

注意，在循环前插入了一个初始化语句，而对T1的赋值语句也被改写成下一次迭代的读出值。另外，我们将最后一次迭代剥离，以避免对A的多余访问。

这种方法称为输入预取，是对前述的简单使用临时存储的改进，因为它只需要长度为1的临时变量。由于在一个好的标量寄存器分配方案中，临时变量将被指派给寄存器，结果代码避免完全长度临时数组的所有缺点。

输入预取是比循环反转更为常用的方法，因为很多情况下可以用于许多循环反转无法消除所有标量化依赖的场合。现在遇到的问题是：它的通用程度如何？输入预取使得下一次迭代的输入在当前迭代的输出被写回之前被读取，从而避免输入的值被那些输出的值改写。但是如果像下面的例子那样，输出改写的是再下一次迭代的输入，结果又如何呢？

```
DO i = 2, 257
   A(i + 2) = A(i) + 1.0
ENDDO
```

662

在此例中，由于我们现在必须提前两次迭代进行预取，输入预取变得更复杂了。这意味着我们必须总是利用临时变量保留输入的两个副本。假设我们如下定义变量T1，T2和T3：在对当前迭代的输出进行写操作的点，T1保存下一次迭代的输入，T2保存再下一次迭代的输入，T3保存需要写回的输出。这样，可用下述代码消除标量化依赖：

```
T1 = A(1)
T2 = A(2)
DO i = 2, 255
   T3 = T1 + 1.0                    ! 计算
   T1 = T2                          ! 拷贝下一次迭代的输入
   T2 = A(i + 2)                    ! 对再下一次迭代的读取
   A(i + 2) = T3                    ! 现在写回安全的
ENDDO
T3 = T1 + 1.0
T1 = T2
A(258) = T3
T3 = T1 + 1.0
A(259) = T3
```

虽然这种形式的代码比用图13-2中的*SafeScalarize*过程生成的代码访问主存的次数少，它还是有缺点的——它产生一次寄存器到寄存器的拷贝，并且需要大量的标量寄存器。但是，如同我们在8.3节中看到的那样，可以通过展开标量循环来消除拷贝操作，且不需要增加标量寄存器的代价。

通常，输入预取可以用于消除任何在编译时刻已知其依赖阈值的标量化依赖。对于第一次迭代之外的每一次迭代，该方法需要一个寄存器临时变量和一个额外的拷贝操作。有了这个限制，输入预取的适用条件就可以简单地陈述了。

原理13.2 任何一个在编译时刻已知其依赖阈值的标量化依赖，可以通过输入预取而修正。

从依赖阈值的定义可以直接得到这个原理。由于依赖阈值为1表明发生预取的迭代是紧邻写操作迭代的下一个迭代，并且由于输入预取在当前迭代的写操作之前插入对下一次迭代的读取，这种方法必然可以修正以前的缺点。

图13-3中的输入预取算法对长度为1的预取距离是有效的。通过使用与8.2节中类似的方法可以把算法推广到更大的阈值。

```
procedure InputPrefetch(S_loop, D)
  // S_loop = DO i = L, U, INC; S; ENDDO是标量化循环
  // 假设预取的需要已经确定，且这个语句已经用SimpleScalarize标量化。
  // D是表示标量语句标量化依赖的依赖图
  // D中的每条边e映射一个区域引用source(e) 到另一个引用target(e)。依赖阈值总是1
  // 只对真依赖使用输入预取，所以source(e) 总是S的左边
for each边e ∈ D do begin
  创建一个称为T0的临时变量; // 可假定为一个寄存器变量
  // 必须生成两个赋值语句:
  //  一个在标量化循环之前的初始化语句，一个在向量语句S后的下一次迭代输入的预取语句
  生成初始化语句T0 = init_reference，其中
    init_reference可以通过将target(e) 中标量化循环变量i的所有实例用L替换而得到，
    L是原来的标量化循环的下界;
  在标量化循环中，将更新语句T0 = update_reference插入在S之后，
    其中update_reference可以通过将target(e) 中标量化循环变量i的所有实例用i+INC替换而得到;
  将S中的target(e) 用T0代替;
  end
  // 现在，我们根据相应赋值语句生成保存当前迭代输出的临时变量
  创建一个新的临时变量T1;
  在标量化循环的结尾插入赋值语句source(e) = T1;
  用T1代替S中左端的source(e);
end InputPrefetch
```

图13-3 输入预取算法

*InputPrefetch*是一个消除依赖阈值为1的标量化依赖的过程。这个算法直接配合前述注解，具有标量化依赖边数的线性时间复杂性。

虽然预取算法可以被扩充为适用于依赖阈值大于1的情形，但是实现的代价和需要的寄存器数目增长非常快。另一个可选择的方法是利用循环分裂来降低复杂性。

663 ~ 664

在许多的情形下，标量化依赖的阈值不是常数，但是输入预取对于这些情形仍然是有用的。为了说明这种情形，考虑Fortran 90语句

```
A(1:N) = A(1:N) / A(1)
```

该语句可以简单地标量化为

```
DO i = 1, N
   A(i) = A(i) / A(1)
```

```
ENDDO
```

这个循环中，从第一次迭代到其自身有一个反依赖，而从第一次迭代到所有其他的迭代有一个携带的依赖，所有的这些依赖都是由于对引用A(1)的使用。在Fortran 90的语义中，A(1)的值应该是这个循环的任何迭代开始执行前就存在的值。所以，通过对进入标量化循环前需要的单个值进行预取，可以处理这个例子。

```
tA1 = A(1)
DO i = 1, N
   A(i) = A(i) / tA1
ENDDO
```

即便是这个单个的常数值不是在第一次迭代中使用，这个策略也可以正确工作。下面给出的第二个例子显示这种情形更复杂的版本。

```
A(1:N, 1:M) = A(1:N, 1:M) / SPREAD(X(1:M), 1, N)
```

内在过程SPREAD将向量X(1:M)复制N个拷贝，以得到一个相应大小的矩阵。这段代码的简单标量化版本如下:

```
DO j = 1, M
   DO i = 1, N
      A(i,j) = A(i,j) / X(j)
   ENDDO
ENDDO
```

在这个例子中，应该在恰好在内层循环之前的位置实施输入预取。

```
DO j = 1, M
   tX = X(j)
   DO i = 1, N
      A(i,j) = A(i,j) / tX
   ENDDO
ENDDO
```

665

13.3.3 循环分裂

当标量化依赖的距离阈值大于1时，对其使用*InputPrefetch*的基本问题是必须把标量化循环中（从依赖源点所在迭代开始）直到阈值的迭代中临时变量的值保存起来。如果这一点可以避免，预取就更加实用了。循环分裂（图13-4）是一种完成这种优化的方法。考虑把下面的向量语句进行标量化的问题:

```
A(3:6) = (A(1:4) + A(5:8)) / 2.0
```

标量化循环携带一个真依赖，同时携带一个反依赖，每一个依赖的阈值为2。

```
DO i = 3, 6
   A(i) = (A(i - 2) + A(i + 2)) / 2.0
ENDDO
```

对*InputPrefetch*进行扩展来处理这个例子，A的两个元素将需要临时变量。但是，由于无论是真依赖，还是反依赖，其阈值都是2，我们可以将标量化循环改写为两个相互独立的循环，二者之间不会相互作用。

```
DO i = 3, 5, 2
   A(i) = (A(i - 2) + A(i + 2)) / 2.0
```

```
ENDDO
DO i = 4, 6, 2
   A(i) = (A(i - 2) + A(i + 2)) / 2.0
ENDDO
```

```
procedure SplitLoop(S_loop, D)
  // S_loop = DO i = L, U, INC; S; ENDDO是应被分裂的循环
  // D是包含这个循环中所有标量化依赖的图
  令e是D中的任何一条真依赖的边;
  T := e的阈值;
  all_the_same := true;
  for each在D中的依赖边e while all_the_same do
    if e表示了一个真依赖and e的阈值 ≠ T
      or e表示一个反依赖
      and T不能整除e的阈值
    then all_the_same := false;
  if all_the_same then begin
    将标量化循环用一个新的循环嵌套替代:
    S1  DO i1 = L, L + T * INC, INC
    S2      DO i2 = i1, U, T*INC
              S
            ENDDO
          ENDDO;
    令D2是循环S2的标量化依赖;
    InputPrefetch(S, D2);
  end
end SplitLoop
```

图13-4 循环分裂算法

这些循环中的每一个的标量化依赖都只跨越了一个迭代，所以可以用简单的输入预取方法消除这些依赖。请注意，如果反依赖具有的阈值不是2的倍数，则上述分裂将产生错误的结果，因为这样的话，第一个循环可能对要被第二个循环读取的地址（如A(i+2)）进行写操作。作为一种风格上的约定，我们将总是把循环分裂写成两个循环嵌套:

```
DO i1 = 3, 4
   DO i2 = i1, 6, 2
      A(i2) = (A(i2 - 2) + A(i2 + 2)) / 2.0
   ENDDO
ENDDO
```

在分裂形式中，内层循环携带一个阈值为1的标量化依赖，而外层循环没有携带任何依赖。现在可以直接对内层循环使用输入预取，得到

```
DO i1 = 3, 4
   T1 = A(i1 - 2)
   DO i2 = i1, 6, 2
      T2 = (T1 + A(i2 + 2)) / 2.0
      T1 = A(i2)
```

```
      A(i2) = T2
    ENDDO
ENDDO
```

通过创建一个只有单一的标量化依赖的循环，循环分裂减少实现预取所需的寄存器数目。

虽然对一个携带有多个依赖且每个依赖具有不同的依赖阈值的循环可以使用循环分裂[221，16]，但是除非阈值是相同的，否则循环分裂不能产生一个循环，且这个循环的所有依赖具有的阈值都为1。所以，如像

666 ~ 667

```
A(128:392) = A(1:191) + A(65:255) + &
                A(256:512) + A(392:512)
```

这样的向量操作中，无论循环按哪个方向进行迭代，总有阈值为常数1或2的真依赖，语句无法容易地用这个方法进行标量化。另外，依赖不具有常数的阈值时，循环分裂也是不能采用的。

输入预取和循环分裂的价值可以如下总结：

原理13.3 任何一个标量化循环，其所有的真依赖都有一个相同的常数阈值T，且所有的反依赖可以被T整除，则这个循环可以用输入预取和循环分裂进行变换，使得所有标量化依赖被消除。

在Allen的学位论文[16]中可以找到这个原理的证明。

如果图13-2中的*SafeScalarize*过程需要大量的临时数组变量，循环反转、循环对齐和循环分裂提供代价相对低廉的方案。图13-5给出一个改进的称为*FullScalarize*的标量化算法，与这些变换混合使用。虽然不使用临时存储空间时*FullScalarize*不能区分所有可能的Fortran 90语句，但它能够成功地区分在实践中会遇到的大部分语句。

```
procedure FullScalarize
  for each向量语句S do begin
    假设S已经被标量化，计算S本身的依赖;
    if S没有对自身的标量化依赖
    then SimpleScalarize(S);
    else if S有标量化依赖，但是没有自身的反依赖
    then begin
      SimpleScalarize(S);
      反转标量化循环;
    end
    else if所有标量化依赖的阈值为1
    then begin
      SimpleScalarize(S);
      InputPrefetch(S);
    end
    else if S的所有标量化依赖有相同的常数阈值T
      and所有的反依赖都有被T整除的阈值
      then SplitLoop(S);
```

图13-5 修改的标量化算法

```
        else if S的所反依赖有相同的常数阈值T
          and所有的真依赖都有被T整除的阈值
        then begin
          反转该循环;
          SplitLoop(S);
        end
        else SafeScalarize(S);
      end
    end FullScalarize
```

图 13-5（续）

*FullScalarize*对所有含有一维向量的语句都可以有效地使用。但是，Fortran 90允许多维的向量。多个向量维为新的策略提供了机会，这一点将在13.4节中讨论。

13.4 多维标量化

到目前为止，所有介绍过的处理方法都是针对一维向量的。但是，Fortran 90中的向量语句不是局限为一维向量的。例如，语句

```
A(1:100, 1:100) = A(1:100, 1:100) + 1.0
```

在向量赋值的左端和右端都有多于一个下标的向量迭代。在此种情况下，假设维是从左到右处理的。例如，语句

```
A(1:100, 1:100) = B(1:100, 1, 1:100)
```

与下面的循环具有同样的作用:

```
DO J = 1, 100
   A(1:100, J) = B(1:100, 1, J)
ENDDO
```

多维向量的引入，在标量化过程中创造了很多新的机会，同时也带来了很多新的问题。在最简单的情况下，多个维可以坍缩为一个向量操作。复杂的例子中可能需要复杂的变换，例如利用循环交换得到一个安全的标量化。下面几小节讨论这些变换，以及将它们应用于对多维向量语句的标量化中。

13.4.1 多维中的简单标量化

多维的标量化意味着简单地将每个维的迭代算符转换为一个对应的循环。例如，在语句

```
A(1:100, 1:100) = 2.0 * A(1:100, 1:100)
```

中，利用这个方法将得到

```
DO j = 1, 100
   DO i = 1, 100
      A(i,j) = 2.0 * A(i,j)
   ENDDO
ENDDO
```

在审视这种风格的代码时，安全地将多个向量维标量化的中心问题是清楚的——当每一个标量化循环对应一个不同的向量迭代运算符时，任何一个携带一个真依赖的标量化循环，就改变了原来语句的语义。所以，可以把修正单个向量操作的同样变换（循环反转、循环对齐

等）用于每个标量化循环，虽然其开销可能比在一维的情况要高得多。

一个显而易见的问题是：标量化后循环的次序应该如何？在通常情况下，这个问题的答案依赖于目标机器。例如，对于大多数的机器，最重要的目标是在最内层循环获得跨距为1的访问。但是，在一个类似于Cray T90的向量机上，最内层循环很可能会被向量化，且因为对于大多数的跨距，向量读操作会由于交错存取存储器[⊖]的原因而获得好的性能，所以最内层循环应该是可能被最有效向量化的循环。

虽然可以定义一个目标函数*score*来反映对机器的依赖，这个函数指明一个给定的向量操作在特定硬件上执行的速度，但是一个更为简单的模型可以适用于本章的讨论。我们将假设更短的跨距总是更好一些，所以对最内层位置的最优选择是沿着跨距为1的列执行。这样，在一组多迭代算符中最左端的向量迭代算符应与最内层循环相对应。

668 ~ 670

在这些假设的基础上，将13.3节中的结果扩展到多维是简单而直接的。每个向量迭代算符可以被独立地标量化。对任何引用的最左端向量迭代算符的标量化循环（按照我们的假设是最有价值的）成为最内层循环；对于其余的迭代算符，从左到右进行标量化，从次最内层位置开始，向外推进。

一旦得到了一个初始的循环次序，按照从最外层到最内层的次序对每个标量化循环做如下的处理：

（1）测试循环是否携带标量化依赖。如果没有，则处理下一层循环。

（2）如果标量化循环只携带真依赖，对循环进行反转并处理下一层循环。

（3）应用输入预取以消除其针对的依赖，在需要的情况下使用循环分裂方法。在一开始，这种处理可能不希望越出内层循环，因为寄存器将很快会被用完。但请注意，在外层循环中，预取的是对一个子矩阵（剩余的维）进行的。这总比下一节中介绍的为整个矩阵生成一个临时存储要好。

（4）否则，循环携带有标量化失效，需要临时存储空间。对这个循环生成一个使用临时存储空间的标量化结果，并终止标量化测试，因为临时存储空间将消除所有的标量化失效。

虽然在大多数的情况中，这个简单的策略可以相当好地运用，它仍然没有利用好一个重要的机会。当有多个标量化循环时，它们被执行的次序不仅会影响到标量化依赖，而且也会影响到其他变换的代价。

13.4.2 外层循环预取

在外层循环中输入预取的有效性可以通过下面的例子显示出来：

```
A(1:N, 1:N) = (A(0:N - 1, 2:N + 1) + A(2:N + 1, 0:N - 1)) / 2.0
```

如果我们试图将列访问的长度划分为向量寄存器长度，会生成两个标量化依赖。首先，有一个方向向量为“(<,>)”的标量化真依赖，包括右端的第二个输入。其次，有一个方向向量为“(>,<)”的反依赖，包括右端的第一个输入。因此，我们不能对外层循环使用循环反转变换。

671

但是，可能对外层循环使用输入预取。如果我们使用这种方法，临时变量将不是一些单变量，而是像如下的输出代码中的临时数组：

```
T0(1:N) = A(2:N + 1, 0)
  DO j = 1, N - 1
```

⊖ 特定的向量跨距可能导致存储的体冲突，降低性能。

```
      T1(1:N) = (A(0:N - 1, j + 1) + T0(1:N)) / 2.0
      T0(1:N) = A(2:N + 1, j)
      A(1:N, j) = T1(1:N)
   ENDDO
   T1(1:N) = (A(0:N - 1, N + 1) + T0(1:N)) / 2.0
   A(1:N, N) = T1(1:N)
```

需要的临时空间总量与原始矩阵的两行相同，这比整个结果矩阵的一个拷贝要求的存储空间要小得多。但是，乍看起来，与利用简单方法需要的读/写操作总数相比，这个例子中的读操作和写操作的总数似乎要多。这是因为j-循环的每一次迭代将输入的一行拷贝到T_0中，将输出的一行拷贝到T_1中，将T_1中的一行拷贝到输出矩阵中——与直接将这些向量操作标量化相比每个元素多两次读操作和两次写操作，与使用临时变量的简单标量化相比，每个元素多一次读操作和一次写操作。

但是，这个分析是使人迷惑的。当我们标量化内层循环时，考虑会发生什么情况。

```
DO i = 1, N
   T0(i) = A(i + 1, 0)
ENDDO
DO j = 1, N - 1
   DO i = 1, N
      T1(i) = (A(i - 1, j + 1) + T0(i)) / 2.0
   ENDDO
   DO i = 1, N
      T0(i) = A(i + 1, j)
   ENDDO
   DO i = 1, N
      A(i, j) = T1(i)
   ENDDO
ENDDO
DO i = 1, N
   T1(i) = (A(i - 1, N + 1) + T0(i)) / 2.0
ENDDO
DO i = 1, N
   A(i, N) = T1(i)
ENDDO
```

672

在任何一个内层循环中都没有任何的标量化依赖，并且如同我们将在13.6节中看到的，可以使用循环合并变换将所有三个循环合并成一个标量化循环。

```
DO i = 1, N
   T0(i) = A(i + 1, 0)
ENDDO
DO j = 1, N - 1
   DO i = 1, N
      T1(i) = (A(i - 1, j + 1) + T0(i)) / 2.0
      T0(i) = A(i + 1, j)
      A(i, j) = T1(i)
   ENDDO
ENDDO
```

```
DO i = 1, N
   T1(i) = (A(i - 1, N + 1) + T0(i)) / 2.0
   A(i, N) = T1(i)
ENDDO
```

现在应当清楚临时变量T_1不会携带有用的值跨越循环边界而且可以由一个寄存器代替。因此，最终的代码与下面的代码看起来类似:

```
DO i = 1, N
   T0(i) = A(i + 1, 0)
ENDDO
DO j = 1, N - 1
   DO i = 1, N
      R1 = (A(i - 1, j + 1) + T0(i)) / 2.0
      T0(i) = A(i + 1, j)
      A(i, N) = R1
   ENDDO
ENDDO
DO i = 1, N
   R1 = (A(i - 1, N + 1) + T0(i)) / 2.0
   A(i, N) = R1
ENDDO
```

这段代码中用到的读操作和写操作的个数与简单的标量化版本是一样的，而使用的临时存储空间要少得多（A的一行）。

一旦我们使用了外层循环的预取来消除这两个标量化依赖，这些依赖不会再出现在内层循环中。因此，它可以被容易地标量化。通常，使用外层循环的预取将只会考虑消除被外层
673 循环携带的依赖。仍然可能需要内层循环的预取以消除由该循环携带的依赖。

13.4.3 用于标量化的循环交换

虽然从执行的观点看，选择的循环次序可能是最优的，但从标量化的观点看，这个次序不总是最优的。例如，考虑

```
A(2:100, 3:101) = A(3:101, 1:201:2)
```

按照规定的次序标量化，将得到

```
DO i = 3, 101
   DO 100 j = 2, 100
      A(j, i) = A(j + 1, 2 * i - 5)
   ENDDO
ENDDO
```

外层循环携带一个真依赖，因此有一个标量化失效。由于这个循环还携带有多个反依赖，且这些依赖的阈值不是规则的，所以无论是循环输入预取还是循环反转，都不能消除这个失效。因此，这个语句对其自身的依赖实际是两个依赖，方向向量分别为“(<,>)”（当i=3时循环写入A(*,3)而当i=4时循环将从同一列中读取）和“(>,>)”（当i=6时循环读取A(*,7)且随后当i=6时，循环将写入同一列）。

但是，如果将这两个循环交换，我们得到方向向量“(>,<)”和“(>,>)”。现在，外层循环携带两个反依赖，且可以被容易地标量化，其长度为1。这样消除由于内层循环的两个依赖，

并且内层循环也可以容易地向量化。

```
DO i = 2, 100
   DO j = 3, 101
      A(j, i) = A(j + 1, 2 * i - 5)
   ENDDO
ENDDO
```

虽然这个次序可能没有最优地使用跨距的存储访问，但它比使用临时存储空间高效多了。

初看起来，标量化循环的循环交换似乎应该是不合法的，因为一个标量化依赖从真依赖被转换为反依赖。但是请记住，我们寻找的是一种标量化的方法，使得标量化循环不携带标量化真依赖。任何带有这种依赖的标量化都是不正确的。根据这个判断标准，原来的标量化是不正确的，但是最后的标量化是正确的。通常，我们可以自由地采用任何标量化循环的次序，以及每个循环的任何方向，只要没有任何一个结果的循环携带一个标量化依赖。我们将使用这个原理导出一个通用的标量化算法。 674

即便是当循环交换不能消除一个标量化失效时，循环交换仍然是一种可以减小临时空间大小的有用变换。假设我们希望切分下面的语句，使得列迭代算符与内层循环相对应:

```
A(1:128, 1:128) = A(1:128, 0:127)
```

在外层循环的简单标量化

```
DO i = 1, 128
   A(1:128, i) = A(1:128, i - 1)
ENDDO
```

中，外层循环携带一个真依赖，这个依赖可以通过外层循环的预取来消除:

```
T0(1:128) = A(1:127, 0)
DO i = 1, 127
   T1(1:128) = T0(1:128)
   T0(1:128) = A(1:128, i)
   A(1:128, i) = T1(1:128)
ENDDO
T1(1:128) = T0(1:128)
A(1:128, 128) = T1(1:128)
```

对列迭代运算符标量化，并在可能的地方合并语句后（见13.6节），这个例子变成

```
DO j = 1, 128
   T0(j) = A(j, 0)
ENDDO
DO i = 1, 127
   DO j = 1, 128
      T1(j) = T0(j)
      T0(j) = A(j, i)
      A(j, i) = T1(j)
   ENDDO
ENDDO
DO j = 1, 128
   T1(j) = T0(j)
   A(j, 128) = T1(j)
ENDDO
```

由于循环中现在对A是从列i读取的，仅有的包含A的依赖具有方向向量“(=,=)”。所以可
675 以交换这些循环，得到

```
DO j = 1, 128
   T0(j) = A(j, 0)
ENDDO
DO j = 1, 128
   DO i = 1, 127
      T1(j) = T0(j)
      T0(j) = A(j, i)
      A(j, i) = T1(j)
   ENDDO
ENDDO
DO j = 1, 128
   T1(j) = T0(j)
   A(j, 128) = T1(j)
ENDDO
```

现在可以合并得到的外层循环，产生

```
DO j = 1, 128
   T0(j) = A(j, 0)
   DO i = 1, 127
      T1(j) = T0(j)
      T0(j) = A(j, i)
      A(j, 128) = T1(j)
   ENDDO
   T1(j) = T0(j)
   A(j, 128) = T1(j)
ENDDO
```

由于T_0和T_1的值绝不会跨越j-循环的不同迭代被重用，并且一个好的标量替换算法将使它们简化为标量变量。

```
DO j = 1, 128
   T0 = A(j, 0)
   DO i = 1, 127
      T1 = T0
      T0 = A(j, i)
      A(j, i) = T1
   ENDDO
   T1 = T0
   A(j, 128) = T1
ENDDO
```

676 通过交换循环，我们消除了所有临时存储空间和所有多余的读操作和写操作。

注意，当标量化的次序确定之后，我们可能已经决定将输入预取操作移动到内层循环。通常，通过这样的循环交换把输入预取应用于最内层循环总会更好，因为这样可以使需要的临时存储空间达到最小，同时由于保证临时变量可以存储在寄存器而不需要写回内存，使性能得到提高。当然，这些工作必然大大增加i-循环中对A的非单位跨距访问的代价。

通过循环交换和循环合并提高寄存器使用效果的通用方法是13.6节的讨论主题。

13.4.4 通用的多维标量化

我们针对多维标量化的通用方法将包括构造一个用于标量化循环嵌套的称为标量化方向矩阵的特殊数据结构。假设我们希望切分一个具有m个向量维的语句。假设 $(l_1, l_2, ..., l_m)$ 代表通过某种代价-效益分析确定的标量化循环从最外层到最内层的理想次序。令 $(d_1, d_2, ..., d_n)$ 是由循环嵌套中某个循环所携带的这个语句对自身的所有真依赖和反依赖的方向向量，其中每个反依赖的方向向量中的方向都被反转。这个语句的标量化方向矩阵是一个$n \times m$矩阵，对所有的k，$1 \leqslant k \leqslant n$，它的第$k$行元素“<”，“>”或“=”由$d_k$构成。例如，在语句

```
A(1:N, 1:N, 1:N) = A(0:N - 1, 1:N, 2:N + 1) + &
        A(1:N, 2:N + 1, 0:N - 1)
```

中，如果选择循环次序使得最内层循环在一个列上迭代，我们得到下面的标量化方向矩阵:

$$\begin{bmatrix} > & = & < \\ < & > & = \end{bmatrix}$$

如果检查这个方向矩阵的任意列，我们可以直接发现是否对应的循环可以被安全地标量化为循环嵌套的最外层循环:

- 如果一列中所有的项都是“=”或“>”，循环可以被安全地标量化为最外层循环，且不需要循环反转。
- 如果一列中所有的项都是“<”或“=”，循环可以被安全地标量化为最外层循环，但要用循环反转。
- 如果一列中所有的项都是“<”和“>”，循环不能用简单的方法标量化。 677

在前面的例子中，建议的外层循环不能用简单的方法标量化，但是如果将其他两个循环中的任一个移到最外层的位置，它们都可以用简单的方法标量化。

一旦选中了一个循环做标量化，由这个循环携带的依赖——任何其方向向量中与被选择的循环对应的位置不含有“=”的依赖——可以在进一步的考虑中被消除。因此内层循环的标量化可以限制为对方向矩阵的一个子矩阵的分析，这个子矩阵是将原始矩阵中选择的列和该列中符号不是“=”的所有行删除得到。在我们的例子中，我们可以选择第二列作为标量化的外层循环。如果我们将这一列向外移动，标量化矩阵变为

$$\begin{bmatrix} = & > & < \\ > & < & = \end{bmatrix}$$

采用这种方法的标量化将在进一步的考虑中删除第二行，把标量化矩阵简化为

$$[> <]$$

最左端的列与先前的外层维对应，它现在可以不进行反转而被标量化。一旦这一步完成，所有的依赖都消除了，且内层的维很容易被标量化。在这个例子中，得到如下的代码:

```
DO j = 1, N
   DO k = 1, N
      DO i = 1, N
         A(i, j, k) = A(i - 1, j, k + 1) + A(i, j + 1, k - 1)
      ENDDO
   ENDDO
ENDDO
```

图13-6中的算法是对这种方法的更形式化的描述。这个算法假设标量操作在一开始已经按照循环表*loop_list*中最优的执行次序从最外层到最内层进行了排序。在可能不使用临时存储空间的情况下，该算法试图保持这个次序。

*CompleteScalarize*的变形可以适用于不同的机器类型。对于临时存储空间的减少不比按跨距为1的维进行访问有价值的机器，这个算法可以调整为排除选择内层的循环，直到所有其他循环已经被标量化。

这个算法的时间复杂性不会比$O(m^2n)$更差，其中m是循环的个数，而n是依赖的个数。为
678 了证明这一点，考察对下一个可以被标量化的列的搜索，它不会耗费超过$O(mn)$的时间，因为它对每个元素的检查一定不会超过一次。由于有m个循环，我们需要这样做的次数不超过m次。

```
procedure CompleteScalarize(S, loop_list)
  // CompleteScalarize试图在不使用临时存储空间的前提下切分一个向量操作。
  // S是要被标量化的语句;
  // loop_list是按“最优”执行顺序排序的一个循环表。
  令M是从将S标量化为loop_list得到的标量化方向矩阵
  while有其余可被标量化的循环do begin
    令l是循环表中第一个可以简单标量化的循环，无论是否使用循环反转
      （通过从左到右检查M中的列确定这个循环）;
    if不存在这样的l then begin
      令l是loop_list中的第一个循环;
      通过输入预取的方法切分l;
      if前面一步失败
      then利用简单的临时存储方法切分S并退出;
    end
    else // 将l变成最外层循环
      根据M中对应l的项，直接切分l,
        或者借助使用循环反转切分l;
    将l从loop_list中删除;
    令M'是在M中删除了与l对应的列以及在那一列
      对应的位置是非“=”项的行后得到的;
    M := M';
  end
end CompleteScalarize
```

图13-6 一个完全的标量化算法

对于一个给定的语句S和一个循环表，*CompleteScalarize*生成一个正确的标量化结果，并有如下的性质:

（1）对可能的最内层循环实施输入预取。

679 （2）标量化循环的次序是可能的最接近于满足性质(1)的标量化中输入预取指定的次序。

从方向矩阵的定义得到正确性。由于可以自由选择任何一个不会导致标量化失效的循环作为最外层的标量化循环，我们只需要寻找其携带的所有依赖都是反依赖或真依赖的某个循环。在第一种情形中，标量化循环是按正常次序迭代的，而在第二种情形中，标量化循环是按反转次序迭代的。

一旦选定最外层的标量化循环，这个循环携带的依赖就不会是某个内层循环的标量化依赖，因为在这样一个依赖中，依赖源和依赖目标中外层循环索引标量的值必定是不等的。对于一个发生在内层循环的标量化失效，它的发生必然是因为对于外层循环索引变量的同样的值的情况下，内层循环索引变量的两个不同的值访问了同一个内存单元。只有在依赖对应于外层循环的位置是一个“=”号时，这种情况才是可能的。因此，消除被外层循环携带的依赖是正确的。

一旦矩阵中只有“<”和“>”都出现的列，就使用输入预取。一旦使用了这种方法，则由使用这种方法的循环携带的依赖可以被消除（根据上面的论证），且继续进行处理。

为了证明性质（1），我们必须说明不存在一个与本算法选择的序列不同的一个循环序列的选择，在这个选择中要求输入预取只能发生在比*CompleteScalarize*选定的序列中更深的嵌套层次。换句话说，在构造一个标量化循环嵌套的某一步，对于下一个外层循环的选择可能多于一个。有可能出现错误的选择会允许在输入预取前有一个另外的循环选择吗？如果循环选择的过程具有有限Church-Rosser性质[12，245]，答案是否定的，此性质在任何序列达到相同的界限时成立。在循环选择系统中，只有有限个循环，且每一步只选择一个循环，这一事实决定了有限性。所以，每个选择序列的终结是循环表处理完结，或是达到需要输入预取的状态。

为了证明这种选择总可以达到相同的界限，我们必须定义此处的“相同”指的是什么。从我们的目的考虑，一个问题的构型是由方向矩阵定义的。两个构型是相同的，如果它们有相同数量的列（存留的循环）和行，但是可能有置换。Sethi已经证明如果一个置换系统满足两个特性，则它有有限Church-Rosser性质，这两个特性称为P1和P3[245]:

- *P*1: 如果完成任何变换（循环选择）步骤，则有一步骤序列使得初始的构型达到一个界限，并有一步骤序列使新的构型达到同样的界限。
- *P*3: 如果从一个初始的构型完成两种不同的变换步骤，则存在多个变换序列使得得到的每个构型达到同样的界限。

680

特性*P*1在我们的系统中是平常的，因为我们可以将新的构型作为界限，这个界限可以通过选择相同的循环而对初始构型进行变换这样一个单独的步骤达到。特性*P*3几乎是平常的。假设我们在一种情况中从构型c_0开始，选择循环l_1，达到c_1，而在另一种情况中选择l_2，达到c_2。由于l_2是c_1中的符合条件的选择，我们现在可以选择它，达到c_3。类似地，我们可以在c_2的构型中选择l_1，达到c_4。现在，面向c_3的方向矩阵与c_0的方向矩阵不同之处是前者只含有在列l_1或l_2的位置含有一个“=”的行，且消除那些列。但是c_4的方向矩阵刚好含有相同的行和列。所以，这两个构型是相同的，证明了*P*3和有限Church-Rosser性质。

所以，无论如何选择符合条件的列，我们总是达到同样的界限。因此，在*CompleteScalarize*中的选择顺序不能再改进了，证明了性质（1）。但由于第一个（最外层）符合条件的循环总是被选择，所得的标量化循环次序是最可能接近理想次序的一个，证明了性质（2）。

虽然*CompleteScalarize*提供将一个循环嵌套标量化，其结果可以称为在特定的限制意义上最优的，仍然可以通过将标量化循环和围绕原始向量语句的循环进行交换，并将邻近的向量语句对应的标量化循环进行合并，从而提高寄存器的重用性。这是13.6节讨论的主题。

13.4.5 一个标量化的例子

前面几小节已经讨论了一系列基于依赖的技术将向量代码标量化。在这一小节，我们用科学计算中常用的一个例子——微分方程组的有限差分松弛求解法——说明这些变换的有效

性。这个算法的一个Fortran 90版本是

```
DO J = 2, N - 1
   A(2:N - 1, J) = (A(1:N - 2, J) + A(3:N, J) + &
            A(2:N - 1, J - 1) + A(2:N - 1, J + 1)) / 4
ENDDO
```

概略地说，这段代码将每一点的值置为周围四个点的平均值。结果，在内存的单元间存在大量的冲突，导致一个集中的依赖图。图13-7给出这段代码的依赖，其中假设向量操作借助一个索引变量为i的标量循环完成。

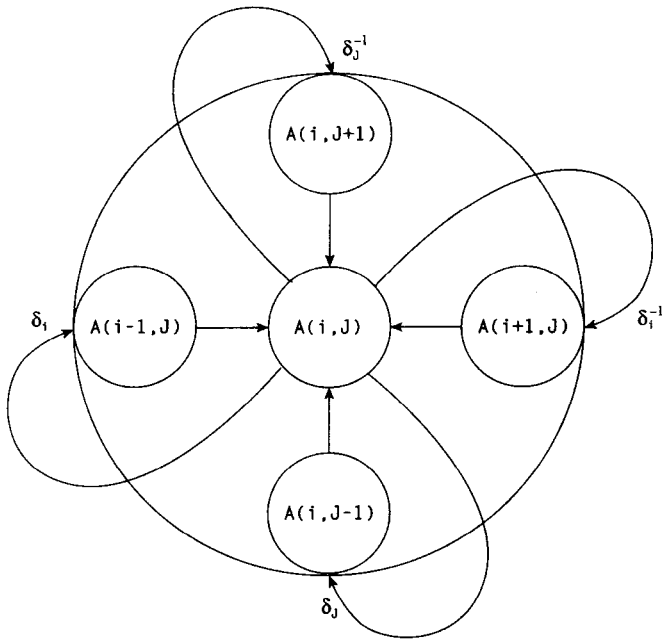


图13-7 有限差分循环的依赖图

这个例子包含一个标量化依赖（在标量化循环δ_i上的真依赖）以及一个阻止循环反转的依赖（在标量化循环δ_i^{-1}上的反依赖）。结果，一个简单的编译器将使用临时存储空间正确地对这个语句标量化，得到下面的程序:

```
DO J = 2, N - 1
   DO i = 2, N - 1
      T(i - 1) = (A(i - 1, J) + A(i + 1, J) + A(i, J - 1) + A(i, J + 1)) / 4
   ENDDO
   DO i = 2, N - 1
      A(i, J) = T(i - 1)
   ENDDO
ENDDO
```

对T的引用需要$2(N-2)^2$次对内存的访问（一半是写入操作，另一半是读取操作）。

681 ~ 682

因为被标量化循环携带的依赖是拥有常数阈值1的规则依赖，13.3.2小节中介绍的分析揭示这个语句是输入预取的一个候选者。直接使用输入预取将得到如下的代码:

```
DO J = 2, N - 1
```

```
      tA0 = A(1, J)
      DO i = 2, N - 2
         tA1 = (tA0 + A(i + 1, J) + A(i, J - 1) + A(i, J + 1)) / 4
         tA0 = A(i - 1, J)
         A(i, J) = tA1
      ENDDO
      tA1 = (tA0 + A(N, J) + A(N - 1, J - 1) + A(N - 1, J + 1)) / 4
      A(N - 1, J) = tA1
   ENDDO
```

如果临时变量被分配到寄存器中，这个版本同原来的Fortran 90程序相比，不需要额外的内存访问。代价是在执行这个循环时需要分配两个寄存器。鉴于寄存器操作比内存访问的代价小，变换后的代码在大多数机器上执行应该好得多。

13.5 对向量机器的考虑

当为一台向量机生成代码时，编译器应当使用不同的标量化方法，使得在内层循环生成向量操作与目标机器的向量寄存器的大小相匹配。通常，实现这个处理是简单直接的，但是可能需要一些额外的代码以保证正确计算出向量长度。为了说明其中的问题，我们再次考察一个简单的标量化例子，其边界是未知的:

```
A(1:N) = A(1:N) + 1.0
```

如同我们在13.2节中看到的，这个循环的标量化是简便的。但是，如果生成的操作的长度要与目标机器的内在向量长度（譬如说64个元素）相匹配，我们必须小心正确地处理N不是64的整倍数的情形。我们推荐的策略是以一个短向量操作开始，这个向量的长度是mod (N, 64)，这个策略可以保证所有剩余的向量片断可以长度恰好为64。

```
VL = MOD(N, 64)
IF (VL .GT. 0) A(1:VL) = A(1:VL) + 1.0
DO I = VL + 1, N, 64
   A(I:I + 63) = A(I:I + 63) + 1.0
ENDDO
```

通过这些修改，本节中介绍的所有标量化策略都可以对向量机扩展。 683

13.6 标量化后的循环交换和合并

对于一个单独的语句，本章中介绍的标量化策略为其生成代码作了很好的工作。但是，标量化通常生成很多单独的循环。此外，由于这些循环不携带依赖，寄存器的大量重用是不常见的。为了解决这些问题，有必要对标量化的结果使用第8章中介绍的循环交换、循环合并、循环展开和压紧以及标量替换等技术。

下面的从一个线性方程求解程序中抽象出来的例子说明其中的一些问题:

```
      DO K = 1, N
S1       M(K, K:N + 1) = M(K, K:N + 1) / M(K, K)
S2       M(K + 1:N, K + 1:N + 1) = M(K + 1:N, K + 1:N + 1) -
            SPREAD(M(K, K + 1:N + 1), 1, N - K) * &
            SPREAD(M(K + 1:N, K), 2, N - K + 1)
      ENDDO
```

我们已经在13.3.2小节中看到了内部过程SPREAD。为了正确地标量化语句S_1，我们注意到即使

有一个包含M(K,K)的标量化依赖存在，但它可以通过输入预取的退化形式被消除。标量化循环变成

```
tM = M(K, K)
DO j = K + 1, N
   M(K, j) = M(K, j) / tM
ENDDO
```

语句S_2是复杂的，因为它是一个使用SPREAD内部过程的二维数组语句，这个内部过程只是简单地将第一个参数给定的数组，按照第三个参数指明的拷贝次数，在第二个参数指定的维上进行复制。在上例中应用的作用是简单地获得由SPREAD的第一个参数指明的两个向量的外积。这个语句的简单标量化将是

```
DO j = K + 1, N + 1
   DO i = K + 1, N
      M(i, j) = M(i, j) - M(K, j) * M(i, K)
   ENDDO
ENDDO
```

但是，由于M(K,j)在右端的使用，i-循环携带有一个标量化依赖，M(K,j)错误地使用在
684 第一次迭代中计算的M的值。这也可以用输入预取消除。

```
DO j = K + 1, N + 1
   tMKj = M(K, j)
   DO i = K + 1, N
     M(i, j) = M(i, j) - tMKj * M(i, K)
   ENDDO
ENDDO
```

注意，对M(i,K)的引用是没有问题的，因为j-循环从迭代K+1开始。现在我们给出完整的标量化结果。

```
DO K = 1, N
   tM = M(K, K)
   DO j = K, N + 1
      M(K, j) = M(K, j) / tM
   ENDDO
   DO j = K + 1, N + 1
      tMKj = M(K, j)
      DO i = K + 1, N
         M(i, j) = M(i, j) - tMKj * M(i, K)
      ENDDO
   ENDDO
ENDDO
```

由于循环交换在这个循环嵌套中被排除，首先尝试循环合并。在循环对齐后，两个j-循环可以被合并，得到

```
DO K = 1, N
   tM = M(K, K)
   M(K, K) = tM / tM
   DO j = K + 1, N + 1
      M(K, j) = M(K, j) / tM
      tMKj = M(K, j)
```

```
      DO i = K + 1, N
         M(i, j) = M(i, j) - tMKj * M(i, K)
      ENDDO
   ENDDO
ENDDO
```

下一步，对j-循环使用循环展开和压紧，以利用重用M(i,K)的机会。在这个例子中，我们使用的展开因子是2，但使用更大的因子也是有可能的。在内层循环重排和合并后，这个例子变成

685

```
DO K = 1, N
   tM = M(K, K)
   M(K, K) = tM / tM
   DO j = K + 1, N + 1, 2
      M(K, j) = M(K, j) / tM
      tMKj = M(K, j)
      M(K, j + 1) = M(K, j + 1) / tM
      tMKj1 = M(K, j + 1)
      DO i = K, N + 1
         M(i, j) = M(i, j) - tMKj * M(i, K)
         M(i, j + 1) = M(i, j + 1) - tMKj1 * M(i, K)
      ENDDO
   ENDDO
ENDDO
```

最后，我们使用标量替换及某种进一步的重排，得到如下结果:

```
DO K = 1, N
   tM = M(K, K)
   M(K, K) = tM / tM
   DO j = K + 1, N + 1, 2
      tMKj = M(K, j) / tM
      M(K, j) = tMKj
      tMKj1 = M(K, j + 1) / tM
      M(K, j + 1) = tMKj1
      tMiK = M(i, K)
      DO i = K, N + 1
         M(i, j) = M(i, j) - tMKj * tMik
         M(i, j + 1) = M(i, j + 1) - tMKj1 * tMiK
      ENDDO
   ENDDO
ENDDO
```

这个形式的代码每5次访存操作可以完成4次浮点运算。借助另外的循环展开和压紧，这段代码可以被改为接近每次访存操作完成一次浮点运算的水平。

13.7 小结

本章陈述了为Fortran 90的数组赋值生成良好代码的问题。这个处理过程的基本策略是将数组赋值转换为可以正确实现原来语句语义的串行循环嵌套。如果这个处理过程的目标不是为了避免使用大的由编译器生成的临时数组的话，这种处理是容易的。

686

本章提供一系列不同的用于减少需要的临时存储空间数量的策略，包括循环反转，输入预取和循环分裂。对于多维数组语句，精心选择正确的循环次序也可能用于避免临时存储的

生成。

注意这些技术对在向量机上高效实现Fortran 90也是需要的，这一点非常重要。在向量机上，最内层循环应该被按照内在的向量操作的长度进行分段。

陈述的第二个问题是在生成标量化代码后，如何提高它的性能。已知如果能够明智的使用第8章中给出的标量寄存器分配技术，是可能生成性能最接近于手工优化的代码的。

13.8 实例研究

无论PFC或是Ardent Titan都没有实现过Fortran 90，所以这里介绍的标量化策略不是必需的。最近，Yuan Zhao在Rice大学的dHPF编译器的环境下实现了这些策略。除本章介绍的这些策略之外，Zhao的实现包括了一个主要的改进——在实现中综合使用了对齐和临时数组的坍缩技术，进一步减少由于Fortran 90生成的循环中需要的临时数组空间的数量。虽然这好像只是较小的改进，但是它深刻影响了结果程序中的高速缓存行为。对于一个用9点模板写的一个HPF程序的SOR例子，在一个单独的处理器的SGI机器上使用本机Fortran 90编译器得到的代
687 码，Zhao的对齐策略在代码性能上得到整数倍的加速比。由这项工作产生的技术报告也指出，在某些情形中，为了减少从Fortran 90语句生成标量化循环时需要的临时存储空间，也可以使用循环倾斜技术[287]。

13.9 历史评述与参考文献

Wolfe引入了标量化的概念，并说明了简单的算法（图13-2）总是可以应用的[279]。他发现在循环合并是合法的场合下，循环合并可以用于消除临时存储空间的使用。他还发现在循环中将循环不变向量保留在寄存器中的重要性，例如矩阵乘法中的结果向量。

本章中的标量化算法取自Allen和Kennedy的一篇文章[22]以及Allen的学位论文[16]。最近，Zhao和Kennedy利用对齐和循环倾斜变换技术对这些策略作了改进[287]。

Sarkar[241]发现在并行化和使标量化中的临时变量空间最小化间需要一个权衡。他给出一个算法，将标量化和并行化结合起来以求得到两方面的好处。

Roth在其学位论文和相关的文章中[238，179，178，180]提出了一种编译策略，在整个数组的层次上分析Fortran 90程序并运用各种初等变换，减少在特定的并行计算机上实现这个策略的代价。对于一个用CM Fortran写的面向Thinking Machines CM-5的模板程序，其中包含了许多对同一个数组的不同的移位操作，Roth的策略可以非常有效地减少程序中通信量。

习题

688

13.1 什么是标量化失效？一个标量化失效总是可以消除的吗？

13.2 证明循环反转技术能够消除一个循环携带的标量化失效，当且仅当这个循环不携带反依赖。为什么这个命题中没有提及输出依赖？输出依赖在标量化中是一个问题吗？

13.3 证明下述命题：任何一个标量化循环，如果其中的所有真依赖都具有相同的常数距离阈值T，且所有的反依赖的阈值可以被T整除，则这个循环可以用输入预取和循环分裂技术进行变换，从而消除所有的标量化依赖。

13.4 将数组语法与第1章中介绍的PARALLEL DO循环类型作比较。它们有精确一致的含义吗？也就是说，一个符合数组语法的语句总是能够被翻译为一个PARALLEL DO循环，而一个单语句的PARALLEL DO循环总是能够被翻译为数组语法语句吗？

第14章

编译高性能Fortran

14.1 引言

在20世纪80年代末，高端并行性发生了变化，由基于总线的共享存储系统（这类系统在今天称为*对称多处理机*（SMP））转变为可扩展的计算机系统，这类系统的存储器被分开，同单独的处理器组装在一起。早期的这类系统的典型设计是超立方体系统，它是由Caltech公司提出，并由Intel和其他公司推广普及。后来，这类系统的设计虽然采用了不同的网络体系结构，但仍然保持分布式存储的设计。这类机器被称为*分布式存储多处理机*。

由于早期分布式存储系统中的单个处理器采用32位寻址方式，单个处理器无法访问一个庞大配置的聚合存储器的每个字。所以，大部分这类机器采用了某种形式的“消息传递”在处理器间实现数据通信。当一个处理器需要位于其他处理器存储中的一组数据时，数据的拥有者将显式地向需要这些数据的处理器*发送*数据，而使用数据的处理器执行*接收*操作，取出这些数据。这类发送和接收的操作在程序中通常是以Fortran或C语言系统库的调用形式实现。这种库的缺点是每一个库只能针对具体的机器实现，使得代码难于在系统间进行移植。*消息传递接口*（MPI）的发布解决了这个问题。MPI是一个可以被高级语言调用的实现消息传递的标准接口。今天，MPI被多数面向可扩展并行机的程序使用。

然而，MPI和它的同类机制存在一个严重的缺点——它们难以被掌握和使用。为编写一个MPI程序，用户通常必须将一个应用程序重写为*单程序流多数据流*（SPMD）的形式。为了说明这个问题，我们首先以一个简单的求和归约计算为例，以Fortran 90中的Fortran 77子方言书写的程序如下：


```
PROGRAM SUM
   REAL A(10000)
   READ (9) A
   SUM = 0.0
   DO I = 1, 10000
      SUM = SUM + A(I)
   ENDDO
   PRINT SUM
END
```

为了在一个并行的消息传递机器上执行这个计算，我们必须将这个程序转换为SPMD形式，在这种形式的程序中，每个处理器在数据空间的不同子集上执行完全相同的程序。一个处理器通过询问处理器专用的环境变量来决定其在哪些数据上进行操作。下面是一个简单而效率低下的SPMD程序：读入一系列数据项，将应存储在不同的处理器中的数据项子集发送出去，计算和，并打印结果。

```
PROGRAM SUM
   REAL A(100), BUFF(100)
   IF (PID == 0) THEN
```

```
      DO IP = 0, 99
         READ (9) BUFF(1:100)
         IF (IP == 0) A(1:100) = BUFF(1:100)
         ELSE SEND(IP, BUFF, 100) ! 100 words to Proc I
      ENDDO
   ELSE
      RECV(0, A, 100) ! 100 words from proc 0 into A
   ENDIF
   SUM = 0.0
   DO I = 1, 100
      SUM = SUM + A(I)
   ENDDO
   IF (PID == 0) SEND(1, SUM, 1)
   IF (PID > 0) RECV(PID-1, T, 1)
      SUM = SUM + T
      IF (PID < 99) SEND(PID+1, SUM, 1)
      ELSE SEND(0, SUM, 1)
   ENDIF
   IF (PID == 0) THEN; RECV(99, SUM, 1); PRINT SUM; ENDIF
END
```

690

虽然这个程序在计算100个元素的局部部分和时实现了重叠计算，在由部分和计算总和时，计算依旧是串行的。所以，这个程序并未获得最全面的并行性。但是，这个程序显示了利用消息传递机制编写SPMD程序的某些困难。在这个程序中，程序员必须管理数据的布局、移动和同步，这是它与Fortran 90程序的主要不同之处。为了完成这些工作，程序员必须手工对代码进行循环分段，产生SPMD代码。

在上世纪90年代初，经过一个非正式的标准化过程推出的高性能Fortran（HPF）是对Fortran 90的扩充。推出它的原意是将数据管理的大部分细节自动化。在HPF中，程序员最主要的智力工作是决定数据以何种布局存放在并行机器的处理器存储系统中。HPF有3种制导用于实现上述功能，制导以注释的形式出现：

- TEMPLATE制导：提供一种机制，使得用户可以声明一个细粒度的虚拟处理器组，表示对解决问题可能有用的最大并行性。对于上面的求和问题，下述制导满足需求：

```
!HPF$ TEMPLATE T(10000)
```

- ALIGN制导：提供一种将数组与一个TEMPLATE对齐或数组相互对齐的方式。对于上面的求和问题，可以用

```
!HPF$ ALIGN A(:) WITH T(:)
```

- DISTRIBUTE制导：描述如何将一个虚拟处理器组或一个数据数组分布到一个实际的并行机器的各个存储器中，它是与机器无关的。例如，在上面例子将数据在100个处理器间进行分布。在HPF中这可以用

```
!HPF$ DISTRIBUTE T(BLOCK)
```

实现。

另一种方案是上述的三个制导也可以用如下的制导代替：

```
!HPF$ DISTRIBUTE A(BLOCK)
```

这个制导直接将一个数据数组分布到实际的处理器上。

利用这些制导，归约求和程序可以写成如下形式：

```
PROGRAM SUM
   REAL A(10000)
   !HPF$ DISTRIBUTE A(BLOCK)
   READ (9) A
   SUM = 0.0
   DO I = 1, 10000
      SUM = SUM + A(I)
   ENDDO
   PRINT SUM
END
```

691

此程序明显比消息传递程序简单，但它具有弱点——编译器必须作大量的工作才能产生比较有效的程序。特别地，它必须识别出这个题目的主要计算是求和归约，并将SUM的值在各个处理器上复制。最后它必须生成最终并行求和的代码。

除数据分布制导外，HPF还有一些用于辅助并行性识别的特殊制导。制导

```
!HPF$ INDEPENDENT
```

指明后面紧跟的一个循环可以被并行化，而不必考虑通信或同步。虽然许多编译器可以自动识别并行性，但这个制导可以确保接受这个程序的各个编译器将并行执行这个循环。

在前面的例子中，由于是求和归约计算，因此这个制导是不适用的。然而，归约运算在科学计算程序中是常用的，因此HPF允许用一个特别的限定词说明一个特定变量是一个归约求和的目标：

```
!HPF$ INDEPENDENT, REDUCTION(SUM)
```

利用这个制导，上面的例子可以写成如下的HPF程序：

```
PROGRAM SUM
   REAL A(10000)
   !HPF$ DISTRIBUTE A(BLOCK)
   READ (9) A
   SUM = 0.0
   !HPF$ INDEPENDENT, REDUCTION(SUM)
   DO I = 1, 10000
      SUM = SUM + A(I)
   ENDDO
   PRINT SUM
END
```

编译器可以很容易处理这个版本的程序，生成一个有效的消息传递程序。

作为最后的一个例子，我们给出一个简单的HPF代码段，它的作用是模仿多网格处理方法。在这个例子中，用TEMPLATE实现粗粒度网格APRIME和细粒度网格A的对齐。INDEPENDENT制导用于保证编译器之间的可移植性，编译器不识别计算循环的并行性。

692

```
REAL A(1023, 1023), B(1023, 1023), APRIME(511, 511)
!HPF$ TEMPLATE T(1024, 1024)
!HPF$ ALIGN A(I, J) WITH T(I, J)
!HPF$ ALIGN B(I, J) WITH T(I, J)
```

```
!HPF$ ALIGN APRIME(I, J) WITH T(2 * I - 1, 2 * J - 1)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK)
!HPF$ INDEPENDENT, NEW(I)
DO J = 2, 1022 ! Multigrid Smoothing (Red-Black)
   !HPF$ INDEPENDENT
   DO I = MOD(J, 2), 1022, 2
      A(I, J) = 0.25 * (A(I + 1, J) + A(I + 1, J) + A(I, J - 1) & + A(I, J + 1)) + B(I, J)
   ENDDO
ENDDO
!HPF$ INDEPENDENT, NEW(I)
DO J = 2, 510 ! Multigrid Restriction
   !HPF$ INDEPENDENT
   DO I = 2, 510
      APRIME(I, J) = 0.05 * (A(2 * I - 2, 2 * J - 2) + &
                   4 * A(2 * I - 2, 2 * J - 1) + A(2 * I - 2, 2 * J) + &
                   4*A(2 * I - 1, 2 * J - 2) + 4 * A(2 * I - 1, 2 * J) + &
                   A(2 * I, 2 * J - 2) + 4 * A(2 * I, 2 * J - 1) +  &
                   A(2 * I, 2 * J))
   ENDDO
ENDDO
! Multigrid convergence test
ERR = MAXVAL(ABS(A(:, :) - B(:, :)))
```

在这个例子中，外层循环的INDEPENDENT制导中的限定词NEW(I)，用于保证内层循环的归纳变量I在执行外层循环不同迭代的每一组处理器上被复制。它的作用大致同其他并行语言中的PRIVATE制导等价。

在本章的余下部分，我们将介绍一些针对本书前面介绍的机器的编译策略，使得HPF程序编译后获得相同有效的消息传递程序。此项工作的一个目标是获得接近于最好的手写消息
693 传递程序的性能。

14.2 HPF编译器概览

在这一节中，我们将讨论HPF编译器的典型结构，并用一个简单的例子说明不同的编译阶段。通常，HPF编译的目标程序是含有实现通信的MPI调用的Fortran 77或Fortran 90程序。考虑到本章的阐述目的，我们把目标语言定为Fortran 90语言的Fortran 77子方言再加上一些简单的通信调用，这些调用我们将陆续介绍。我们将假设所有的数组赋值语句已经利用第13章中介绍的方法变换为串行循环（标量化）。考虑到这种处理的目的，我们还假设在计算中用到的处理器的个数是固定的，并且是在编译时刻已知的，当然这个假设是可以轻易地放宽的。

在绝大多数HPF的实现中计算划分的原则是*拥有者计算规则*，这条原则指出，每个赋值语句的左端的拥有者必须计算右端的表达式。虽然这条原则在标准中是隐含的，但并不是必需的。事实上，编译器可以自由地根据需要以任意的方式放宽拥有者计算的规则，但是，编译器应该在为了获得明显的性能提高时才放宽上述规则，因为用户希望根据拥有者计算规则分布计算，所以他/她可能已经有了以此分布数据的考虑。在本章的后面，我们将探讨放宽这条规则的优化。

HPF编译过程包括以下几个阶段：

（1）*依赖分析*：当判断在最终的程序中是否需要加入通信以及在何地加入通信时，需要建立完全的依赖集。

（2）分布分析：数据分布分析的目的是决定在程序中的每一点对每一个数据结构采用何种数据分布方式。虽然在数据达到程序中的一个给定点，同一个数据结构可能有多种分布方式，但是这是不希望的，所以我们在此不讨论这种情况。

（3）划分：一旦程序中任何部分的分布信息已知，计算划分就确定了。也就是说，我们必须对程序中的每个语句实例确定一个执行它的处理器，其中一个语句实例是通过控制该语句执行的循环索引变量对其参数化的。在我们此处讨论的编译器中，这个工作相当于对每条语句用ON HOME制导注释，这条制导将在下面例子的改进版中描述。

（4）通信分析和定位：确定需要通信的位置是必需的。这需要检查程序中存在于语句间的依赖和语句内的依赖。

（5）程序优化：下一步，编译器实施程序变换以提高所获得程序的性能。在很大程度上，这意味着构造这样的通信和同步，用最小的代价获得最大的并行性。 694

（6）代码生成：生成一个SPMD程序通常包括三个任务。首先，实际的SPMD程序的代码必须由一个循环分段和增加条件掩码的过程生成。其次，通信必须最后定案。最后，必须完成每个处理器上管理所需存储的程序变换。

我们用一个模拟松弛计算的简单HPF代码段为例介绍上述步骤。在这段代码中删去了输入-输出和子程序调用，这样可以将注意力集中到最重要的问题上。

```
      REAL A(10000), B(10000)
      !HPF$ DISTRIBUTE A(BLOCK), B(BLOCK)
      DO J = 1, 10000
         DO I = 2, 10000
S1          A(I) = B(I - 1) + C
         ENDDO
         DO I = 1, 10000
S2          B(I) = A(I)
         ENDDO
      ENDDO
```

图14-1给出这个例子中的依赖。由于在每个内层循环中用到不同的数组，所以每个循环都没有携带依赖。由于语句S_1的输出在包围语句S_2的循环的两次不同迭代中作为输入使用，所以存在一个从S_1到S_2的循环无关的真依赖。同样，还存在一个从S_2到S_1的由外层循环携带的真依赖。从语句S_1到其自身以及从语句S_2到其自身各存在一个外层循环携带的输出依赖，从语句S_1到S_2以及从S_2到S_1各存在一个由外层循环携带的反依赖。我们很快就会看到这些依赖对通信分析的作用。但是，重要的一点是关于I的循环不携带依赖。

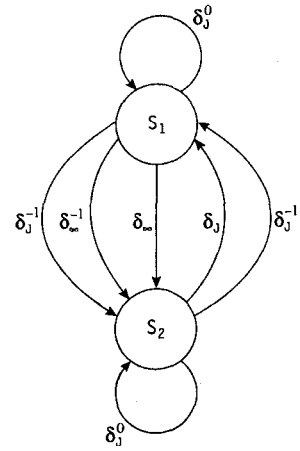


图14-1 例子中的依赖

这个例子的分布分析是简单的——在这个代码段中的任何位置，数组A和数组B都是块分布方式。这表明如果遵守拥有者计算的原则，在整个数组上进行迭代的任何循环可以平均分布到所有的处理器上。基于这样的认识，理想的划分显然是分布I-循环的计算。

通信分析应确定通信插入的位置。在开始，通信被放在需要通信的数据访问附近。随后，

通信将根据依赖关系进行移动。通过对循环中不同引用数据足迹的检测可以确定对通信的需求。例如在包围语句S_1的循环中，每个处理器将对A(L:L+99)计算其结果，此处A(L:L+99)是该处理器拥有的数组A中的位置范围（在处理器0上，A(1)的结果没有计算）。另一方面，在赋值语句右端将访问B(L-1:L+98)。注意对B(L-1)的引用，执行语句的处理器不是这个数组元素的拥有者。所以，这是一个非本地处理器引用。我们把得到上述结论的这种处理称为数据足迹分析（*footprint analysis*）。通过类似的数据足迹分析，我们可以看到包围语句S_2的循环不含有非本地处理器引用。当这个分析完成时，我们看到只有第一个I-循环的一次迭代需要通信。如果我们现在实施循环分段，则结果的代码段如下：

```
        ! Shadow location B(0) receives data
        REAL A(1, 100), B(0:100)
        !HPF$ DISTRIBUTE A(BLOCK), B(BLOCK)
        DO J = 1, 10000
I1          IF (PID /= 99) SEND(PID + 1, B(100), 1)
I2          IF (PID /= 0) THEN
               RECV(PID - 1, B(0), 1)
               A(1) = B(0) + C
            ENDIF
            DO I = 2, 100
S1             A(I) = B(I - 1) + C
            ENDDO
            DO I = 1, 100
S2             B(I) = A(I)
            ENDDO
        ENDDO
```

注意，在每个循环中的数据接收条件已经和对接收数据的使用合并在一起。虽然这只是一个较小的优化，但是仍然可以使我们从条件结构中得到双倍的好处：不仅是避免等待永远不会发送的数据，而且可以避免在处理器0上执行第一次迭代。

695 ~ 696

这个例子中的另一个有意思的特性是在每个处理器上对额外分配的存储的使用，即本例中的B(0)单元，这个单元用于存放从相邻处理器通信得到的数据。这些额外分配的单元通常统称为重叠区或阴影区。

然后，编译器必须确定通信的正确插入位置。显然，我们需要在使用数据前接收这些数据，但是我们提前多长时间发送这些数据呢？例如，我们可以在J-循环外发送这些数据吗？本例中这种做法是不可以的，因为关于数组A的值的依赖是由J-循环携带的。由于在本循环中计算的值在同一个循环的后面使用，我们必须使通信维持在此循环之中。我们可以试图移动I_1中的发送，使它与J-循环上一次迭代最后计算B(100)的值靠近。然而，如果我们要实现上述移动，就必须插入特殊的代码保证第一个迭代接收到所需的数据。由于这个复杂的变换不能明显提高整体性能，我们保持原来的循环形式。

下一步骤是优化。通信优化采用三种通用的策略：

（1）将在同一对处理器间发送的数据聚集成一个消息包，从而减少消息发送启动的代价。这种优化对本例的程序没有太大的意义，因为没有大量数据可以聚集——基本上，每一步我们在一对处理器间的每个方向上发送一个字。

（2）通信和计算的重叠。随后我们会看到，这种优化对于本例是非常有效的。

（3）特殊通信模式的识别。特殊通信模式可以替换为快速的系统调用以实现集合通信。

这种优化的一个主要例子是求和归约的识别，大多数机器上都有实现求和归约的快速系统例程。另一个例子是广播通信。

对于这个例子，最有效的优化是通信和计算的重叠。如果仔细观察这段代码，可以发现除了一次迭代外，包围语句S_1的循环的所有其他迭代都纯粹是局部的——根本不需要通信。如果将这段代码放在数据的一次发送和一次接收之间，则可以获得显著的通信和计算重叠。将数据接收和计算的代码（在I_2的条件语句中的代码）移动到包围S_1的循环的后面的某个位置即可实现上述功能：

```
      ! Shadow location B(0) receives data
      REAL A(1, 100), B(0:100)
      !HPF$ DISTRIBUTE A(BLOCK), B(BLOCK)
      DO J = 1, 10000
I1       IF (PID /= 99) SEND(PID + 1, B(100), 1)
         DO I = 2, 100
S1          A(I) = B(I - 1) + C
         ENDDO
I2       IF (PID /= 0) THEN
            RECV (PID - 1, B(0), 1)
            A(1) = B(0) + C
         ENDIF
         DO I = 1, 100
S2          B(I) = A(I)
         ENDDO
      ENDDO
```

至此这个循环有很好的形状，由此生成的最终版本的SPMD代码与上述代码很相像。在生成SPMD代码时，编译器必须对分布的数据区进行分段，使得数据区的大小适合于一个处理器和附加的某个重叠区或阴影区存储（如上述的B(0)）。

在后面几个小节中，我们将介绍实现上述某些优化和在本例中没有显现的优化算法。如果完整地讨论HPF编译技术，则需要一本书来陈述，所以我们不会讨论全部的HPF编译技术。我们的处理方式是力求告诉读者如何将本书中给出的概念和原则应用到对HPF语言的处理中。

14.3 基本循环的编译技术

本节我们将详细讨论实现HPF中循环嵌套的方法。概括而言，编译策略可以看成是先进行初步分析，然后对整个循环嵌套进行检查，并将其变换为等价的SPMD循环嵌套格式，其中要在适当的位置插入通信。

14.3.1 分布信息的传播和分析

第一步是确定在程序中的一个特定点，一个给定的数组可能具有何种分布，以便生成访问这个数组的代码。由于以下两个原因这不是一个简单问题：

（1）HPF支持动态重分布，这是一个有效的语言扩展。确切地说，用户可以在到达特定点的不同控制流路径上插入REALIGN和REDISTRIBUTE语句（这两个语句都是可执行的）。

（2）HPF提供一种机制，使得一个子程序的形式参数可以从调用程序继承分布模式。如果在不同的调用点将不同的分布模式传递给同一个形式参数，则同一个子程序可能应用多种分布信息。

显然，这类二义性最好予以避免，因为在代码中它是不可能被消除的，构造代码的惟一途径是通过对运行时系统的查询得到精确的分布信息，但这样会严重增加代码或存储空间的开销，或同时增加两方面的开销。然而，首先必须确定是否存在任何二义性。

在局部范围内，通过解一个称为*到达分解*的数据流分析问题可以实现分布分析，这与4.4节中讨论的到达定值问题直接类同。事实上，当把任何REALIGN和REDISTRIBUTE语句（或语句对）作为一个定值时，到达分解问题就是一个精确的到达定值问题。此分析也要为程序的开始点设定初始分解。当跨越过程时，必须解决过程间版本的到达分解问题。这是一个流敏感问题，类似于一个简单的常数传播的形式。这个问题将在14.4.7小节中讨论。对于那些有多个数据分布模式到达一个程序点的例子，可以基于运行时测试数据分布信息的方法，实现运行时刻的代码复制，从而消除数据分布信息的二义性。由不同的调用链导致的二义性通常可以通过过程克隆来消除，这种方法也将在14.4.7小节中讨论。

在本章的余下部分，我们假设对于每个数组，在程序中的每一点只有一个可用的数据分布模式。事实上处于简化的考虑，我们假设在一个子程序中，同一个数组的所有引用具有的都是一个相同的数据分布模式。在此基础上，我们可以构造映射，实现从数组的全局下标空间到一个处理器及在这个处理器内的局部下标空间的变换：

$$\mu_A(i)=(\rho_A(i),\delta_A(i))=(p,j) \tag{14-1}$$

式中ρ_A将一个多维数组的全局下标i映射到一个处理器上，这个处理器拥有以i为索引的数组元素，而δ_A将全局下标映射到拥有该数组元素的处理器的局部下标j。例如，假定声明在一组编号为0至$p-1$的所有处理器上以BLOCK方式分布一个一维数组A，如果数组A声明有N个元素，下标从元素1开始，则这个数组分布块的大小由

$$B_A=\lceil N/p\rceil \tag{14-2}$$

给出，此公式保证除了最后一个处理器上的数据块未被填满外，其他所有数据块的大小是一样的。有了这个定义，我们就能够计算映射函数的值：

$$\rho_A(i)=\lceil i/B_A\rceil-1 \tag{14-3}$$

$$\delta_A(i)=(i-1)\bmod B_A+1=i-\rho_A(i)B_A \tag{14-4}$$

699 这些映射函数将是下一小节介绍的循环分析的基本内容。

一旦我们知道了一个循环嵌套中不同数组所具有的分布模式，我们就完成了确定对这个循环进行计算划分和插入通信的准备工作。我们从计算划分开始。

14.3.2 迭代的划分

大多数HPF编译器采用（左端）拥有者计算的原则进行计算划分。这条原则表明的是，对于一个单语句的循环，由迭代的赋值语句左部数据项的拥有者执行每个迭代。所以，在循环

```
REAL A(10000), B(10000)
!HPF$ DISTRIBUTE A(BLOCK), B(Block)
DO I = 1, 10000
A(I) = B(I) + C

ENDDO
```

中，迭代I在A(I)的拥有者上执行。如果有100个处理器参加运算，则开始的100次迭代在处理器0上执行，下一个100次迭代在处理器1上执行，以此类推。

确定了左端拥有者计算的原则，我们应该如何处理多语句的循环呢？一个简单的策略是利用循环分布，将循环分解为一系列的单语句循环。稍后，这些循环将通过第6章介绍的带类型合并技术重新合并在一起。除了不能再进一步分布的依赖环的例子外，这个方法对其他循环都有效。

对于依赖环情形，编译器必须选择在依赖环中的某个语句的左端的拥有者上执行所有计算。我们称这种引用为划分引用。在依赖环循环中选择一个划分引用应该采用如下的启发式策略：它的目标是使语句集合中的携带依赖数目减到最少。如同我们在14.4.3小节中将看到的，通过对齐优化（此优化技术与在6.2.3小节中介绍的技术类似）可以减少总的携带依赖的数目。

一旦选定一个划分引用，就必须实现循环划分。这项工作的主要结果是确定给定的循环从全局迭代到执行迭代的处理器的映射。对于一个循环头中索引变量为I的给定的循环

```
DO I=1, N
```

假设划分引用是

```
A(α(I))
```

700

其中α是一个函数，它也可能包括和处理中的循环相互嵌套的其他外层或内层循环的索引。我们暂时假设α是一个索引的线性函数或某种其他的一对一的映射。然后，当一个给定循环L的划分引用为A(α(I))时，拥有迭代I的处理器（即负责执行迭代I的计算的处理器）由下式给出：

$$\theta_L(I)=\rho_A(\alpha(I)) \tag{14-5}$$

在给定的处理器p上执行的索引集合可用表达式

$$\{I\text{满足}1\leqslant I\leqslant N\text{且}\theta_L(I)=\rho_A(\alpha(I))==p\} \tag{14-6}$$

确定。这个集合可以定义为更简单的形式：

$$\alpha^{-1}(\rho_A^{-1}(\{p\}))\cap[1{:}N] \tag{14-7}$$

这个迭代集合的问题是它是全局迭代集合的一个子集。为了有效性，我们把它转换为局部迭代集合。为此目的，我们需要计算从全局迭代到局部迭代的映射。对于一个给定的循环L，我们称这个映射为Δ_L。

对于局部迭代集合，它可以方便地表示从1到某个上界的一次遍历，所以我们将使

$$\alpha^{-1}(\rho_A^{-1}(\{p\}))$$

中的最小值映射为索引1。对于拥有迭代1和N中的划分引用的处理器必须作为特别情况处理，我们将马上讨论到这种情况。

考虑下面循环的例子：

```
DO I = 1, N
   A(I+1) = B(I) + C
ENDDO
```

引用A(I+1)是划分引用。如果A与在前面的例子中声明的一样，

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
```

并且有100个处理器参加计算，分布块的大小是100，并且

701

$$\rho_A^{-1}(\{p\}) = [100p+1 : 100p+100]$$

由于$\alpha(i)=i+1$，反减1:

$$\alpha^{-1}(\rho_A^{-1}(\{p\})) = [100p : 100p+99]$$

由于$100p$是这个迭代范围的全局索引的最小值，我们将为每个处理器通过映射给出新的迭代变量i，它将从1开始迭代到100。特别规定，

```
i = I - 100 * PID + 1
```
(14-8)

其中i是每个处理器的局部变量。此式可以改写为计算I的值:

```
I = i + 100 * PID - 1
```
(14-9)

根据这个例子，我们可以定义从全局迭代空间到处理器p的局部迭代空间的抽象的映射$\Delta_L(I,p)$如下:

$$\Delta_L(I,p) = I - \min(\alpha^{-1}(\rho_A^{-1}(\{p\}))) + 1 \tag{14-10}$$

在我们的例子中，该映射是

```
ΔL(I,PID) = I - 100 * PID + 1
```
(14-11)

下一步工作是调整数组的下标，使其与对局部索引集合的映射匹配。考虑如下形式的数组B的引用:

$$B(\beta(I))$$

我们的目标是把这个引用映射为一个对局部B的引用

$$B(\gamma(i))$$

此处的$\gamma(i)$将局部索引i映射到局部B的正确的位置。换言之，为了生成正确的代码，我们必须计算一个局部索引函数γ，用以替代数组引用下标中的β。预期的关系由图14-2确定。在这个图示中，GL代表全局循环迭代，GI代表全局数组下标，LL代表局部循环索引，LI代表局部数组下标。映射β是全局数组下标的函数。前面描述的函数Δ和前一小节描述的函数δ实现全局信息到局部数组下标的映射。

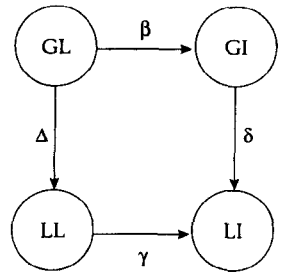


图14-2 索引集合和迭代集合间的映射

就图14-2而言，显而易见，对于一个给定的处理器p，希望的局部下标的函数由下式给出:

$$\gamma(i) = \delta(\beta(\Delta^{-1}(i))) \tag{14-12}$$

在我们的例子中，函数δ由

```
δB(K) = K - 100 * PID
```
(14-13)

定义，而Δ_L（从公式（14-11）得到）的逆由

```
ΔL^-1(i,PID) = i + 100 * PID -1
```
(14-14)

给出。由于β是恒等式，我们可以简单地应用从公式（14-11）中得到的Δ_L，生成

```
γ(i) = i + 100 * PID -1 - 100 * PID = i - 1
```
(14-15)

由于下标函数α对其参数加1，所以对数组A的映射与此映射只有微小的不同。因此对于局

部循环，引用A(I+1)被改写成A(i)。

在考虑了所有这些之后，对于内部的处理器，循环变成

```
DO i = 1, 100
   A(i) = B(i - 1) + C
ENDDO
```

然而，如何处理全局循环的上界和下界的细节还没有讨论。这些讨论只对于拥有迭代L（循环下界）和N的处理器是重要的。对于拥有循环L的处理器，我们必须保证循环的开始不早于迭代

$$\Delta_L(L,\ p)=L+1-\min(\alpha^{-1}(\rho_A^{-1}(\{p\}))) \tag{14-16}$$

702 ↓ 703

同样，在拥有迭代N的处理器上，我们必须保证迭代的执行不超过

$$\Delta_L(N,\ p)=N+1-\min(\alpha^{-1}(\rho_A^{-1}(\{p\}))) \tag{14-17}$$

注意，一个给定的迭代K的拥有者由公式$\theta_L(K)=\rho_A(\alpha(K))$得到。所以，我们可以通过对上述等式的测试确定哪个处理器是拥有者。这些要点可以通过上述例子的推演显示。借助于公式（14-16）的使用，我们发现0号处理器拥有全局迭代1，所以在这个处理器上对应的迭代下界是2。对于循环上界N，拥有这个迭代的处理器是

$$p_N=\lceil(N+1)/100\rceil-1=\lfloor N/100\rfloor$$

利用公式（14-17），对应的局部迭代由

$$N+1-PID\ *\ 100=N+1-100\ *\ (\lfloor N/100\rfloor)=N\ \bmod 100+1$$

给出。因此，例子循环得到的最后形式如下：

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
IF (PID == CEIL((N + 1 / 100) - 1) hi = MOD(N, 100) + 1
DO i = lo, hi
   A (i) = B(i - 1) + C
ENDDO
```

至此，我们已经能够解决如何转换为局部索引和局部下标表达式的问题。当处理器计算一个并不为它所拥有的表达式时，我们仍然存在应该如何做的问题。这也是下一节讨论的通信生成的目标。

14.3.3 通信生成

循环编译过程的最后一步是生成这个循环需要的通信。正如我们前面指出的，这个工作可以通过分析每个引用的足迹来完成。足迹是指引用在局部的迭代空间上处理的迭代的集合。

假设我们有一个数组引用，它出现在一个被分布的循环中赋值语句的右端，这个循环的索引变量是I，数组引用的下标表达式是β(I)：

$$B(\beta(I))$$

索引变量值的集合由

$$\beta^{-1}(\rho_B^{-1}(\{p\})) \tag{14-18}$$

704

给出，其中这个引用引用了B的一个元素，而这个元素是局部于处理器p的。

如果这个引用是语句右端惟一的一个引用，则不需要通信的迭代集合由

$$\alpha^{-1}(\rho_A^{-1}(\{p\})) \cap \beta^{-1}(\rho_B^{-1}(\{p\})) \cap [1:N] \tag{14-19}$$

给出。其他任何迭代将需要从某个其他的处理器接收数据。也就是说，如果

$$I \in (\alpha^{-1}(\rho_A^{-1}(\{p\})) - \beta^{-1}(\rho_B^{-1}(\{p\}))) \cap [1:N] \tag{14-20}$$

则在迭代I中，我们必须为了语句右端的这个引用而从拥有这个元素的处理器接收数据。

在一些迭代中，我们需要将数据发送到其他的处理器。在任何迭代中，当处理器p拥有数据但是不执行进行这个计算的语句时，则发生这种情况。在我们的集合表示法中，发送发生在满足

$$I \in (\beta^{-1}(\rho_B^{-1}(\{p\})) - \alpha^{-1}(\rho_A^{-1}(\{p\}))) \cap [1:N] \tag{14-21}$$

的时候。

让我们现在回到例子，探讨这些公式的含义。

```
REAL A(10000), B(10000)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK)
   ...
DO I = 1, N
   A(I + 1) = B(I) + C
ENDDO
```

我们已经知道

$$\alpha^{-1}(\rho_A^{-1}(\{p\})) = [100p : 100p+99]$$

语句右端的迭代的集合由

$$\beta^{-1}(\rho_B^{-1}(\{p\})) = [100p+1 : 100p+100]$$

给出，其中的数据是局部的。根据公式（14-19），在处理器p上不需要通信的迭代集合是

$$[100p : 100p+99] \cap [100p+1 : 100p+100]$$
$$= [100p+1 : 100p+99]$$

705

按照公式（14-20），其数据必须从其他的处理器接受的迭代集合由

$$[100p : 100p+99] - [100p+1 : 100p+100] = [100p : 100p]$$

给出，而按照公式（14-21）在计算开始前必须向一个相邻处理器发送数据的迭代的集合是

$$[100p+1 : 100p+100] - [100p : 100p+99]$$
$$= [100p+100 : 100p+100]$$

在为这些局部迭代生成代码之前，我们必须决定局部存储分配应该是什么样的。在这个例子中，我们需要在每一个处理器上存放A的100个元素。另外，我们需要足够的空间存放B的100个元素，以及一个额外的存储单元用于存放从左边紧邻的处理器传送来的B的一个元素的值。因此，让我们假设我们对局部值的存储空间是A(1:100)和B(0:100)。

在探讨要生成的代码前，我们需要一种方式把那些需要进行通信的迭代映射为局部的迭代集合。让我们从“接收”开始，因为显然我们必须在一个迭代中接收对其执行计算的数据。这个循环的局部迭代是

$$\Delta_L(100 * PID, PID) = 100 * PID - 100 * PID + 1 = 1 \tag{14-22}$$

所以我们必须在迭代1中接收。在调整后包含接收数据的循环代码如像下面的形式:

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
IF (PID == CEIL((N + 1) / 100) - 1) hi = MOD(N, 100) + 1
DO i = lo, hi
   IF (i == 1 .AND. PID /= 0) RECV(PID - 1, B(0), 1)
   A(i) = B(i - 1) + C
ENDDO
```

上面的接收操作实际上并不需要保证PID≠0的条件，因为在那个处理器上lo将永远不等于1，但是出于完整性的考虑，我们保留这个条件。

发送操作必然发生在如下的局部迭代中:

$$\Delta_L(100 * PID + 100, PID) = 100 * PID - 100 * PID + 101 = 101 \quad (14\text{-}23)$$ 706

但是，请注意这个迭代不是在计算的正常迭代范围之内。所以，我们必须要么为了发送而把迭代范围扩展到包含这次迭代，要么在循环外生成执行发送的特殊代码。此外，我们必须保证不会在循环的最后一个迭代发送，因为不会有接收发生。如果选择第一种方案，我们需要在循环中加入另一个条件语句，控制每一个迭代的执行:

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP) hi = MOD(N, 100) + 1
DO i = lo, hi + 1
   IF (i == 1 .AND. PID /= 0) RECV(PID - 1, B(0), 1)
   IF (i <= hi) THEN
      A(i) = B(i - 1) + C
   ENDIF
   IF (i == hi + 1 .AND. PID /= lastP) &
      SEND(PID + 1, B(100), 1)
ENDDO
```

将发送移动到循环外的位置，可以大大简化这段代码:

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP) hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   DO i = lo, hi
      IF (i == 1 .AND. PID /= 0) RECV(PID - 1, B(0), 1)
      A(i) = B(i - 1) + C
   ENDDO
   IF (PID /= lastP) SEND(PID + 1, B(100), 1)
ENDIF
```

我们注意到同样可以通过将接收移动到循环外而进一步简化这段代码。基于后面将说明的原因，我们将选择从主计算循环中剥离接收迭代的方式实现上述需求: 707

```
lo = 1
```

```
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP)   hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   IF (lo == 1) THEN
      RECV(PID - 1, B(0), 1)
      A(1) = B(0) + C
   ENDIF
   ! lo = MAX(lo, 1 + 1) == 2
   DO i = 2, hi
      A(i) = B(i - 1) + C
   ENDDO
   IF (PID /= lastP) SEND(PID + 1, B(100), 1)
ENDIF
```

这是你希望编译器能够生成的一类代码。但是，这段代码仍然有一个问题。如像写出的那样，循环将在处理器组中的处理器间以串行方式执行。也就是说，处理器0在处理器1开始执行前完成循环的所有操作，因为发送是放在循环的最后，而接收是放在开始的位置。如果这段代码能够安全地重排，以便首先完成所有的发送，则并行性可以显著地提高。新的代码是

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP)   hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   IF (PID /= lastP)  &
      SEND(PID + 1, B(100), 1) ! 从最后移动过来
   IF (lo == 1) THEN
      RECV(PID - 1, B(0), 1)
      A(1) = B(0) + C
   ENDIF
   DO i = 2, hi
      A(i) = B(i - 1) + C
   ENDDO
ENDIF
```

这导致一个问题，这样的一个重排序何时是合法的？为了回答这个问题，我们必须有某种
708 机制，以便计算包含有通信的循环中的依赖，如上面给出的例子。诀窍是将通信操作看成是内存访问，这样就可以直接使用我们已经建立起来的依赖机制。解决方案是接受这样一个约定：一个接收只不过是将A从全局存储位置拷贝到一个局部位置。类似地，一个发送是一个将A从局部位置拷贝到全局位置。借助这些方法，上面的初始版本的循环看起来类似于下面的代码：

```
      IF (PID <= lastP) THEN
S1       IF (lo == 1 .AND. PID /= 0) THEN
            B(0) = B0(0) ! 接收
            A(1) = B(0) + C
         ENDIF
         DO i = 2, hi
            A(i) = B(i - 1) + C
```

```
        ENDDO
S2      IF (PID /= lastp) Bg(100) = B(100)   !发送
    ENDIF
```

如果我们当初对这段代码实施的是依赖分析，将发现不存在导致从S_1到S_2的依赖链。因此，在包含的IF语句内部，这些语句是可以重排序的。

另一方面，如果初始代码是

```
REAL A(10000), B(10000)
!HPF$ DISTRIBUTE  A(BLOCK), B(BLOCK)
   ...
DO I = 1, N
   A(I + 1) = A(I) + C
ENDDO
```

循环将被改写为

```
    IF (PID <= lastP) THEN
S1      IF (lo == 1 .AND. PID /= 0) THEN
           A(0) = Ag(0) ! 接收
           A(1) = A(0) + C
        ENDIF
        DO i = 2, hi
           A(i) = A(i - 1) + C
        ENDDO
S2      IF (PID /= lastP) Ag(100) = A(100) ! 发送
    ENDIF
```

此例中有一个依赖，必须以串行方式计算。因此，重排序将不是正确的。 709

14.4 优化

一旦确定循环生成代码的基本框架，我们就能够看到可以实施很多优化。这些优化的目的是提高通信的性能或通过通信和计算的重叠而隐藏通信。另外，通过流水线技术，可以使一些不能完全并行的计算获得部分并行化。最后，我们将考虑与存储管理相关的问题，以及这些问题可能如何使通信优化问题复杂化。

14.4.1 通信向量化

在大多数消息传递机器上，发送一个消息的代价包括两部分：启动代价和每个单元传输的代价。一般而言，启动的代价是巨大的，所以将在同一对处理器间传输的多个消息合并成一个消息是有好处的。这是通信向量化的目标。通过将我们已经考虑过的例子扩展为两个循环

```
REAL A(10000, 100), B(10000, 100)
!HPF$ DISTRIBUTE  A(BLOCK, *), B(BLOCK, *)
DO J = 1, M
   DO I = 1, N
      A(I + 1, J) = B(I, J) + C
   ENDDO
ENDDO
```

可以说明这个变换。

如果我们应用前节中的代码生成过程，将得到如下形式的循环：

```
DO J = 1, M
   lo = 1
   IF (PID == 0) lo = 2
   hi = 100
   lastP = CEIL((N + 1) / 100) - 1
   IF (PID == lastP) hi = MOD(N, 100) + 1
   IF (PID <= lastP) THEN
      IF (PID /= lastP)  &
         SEND(PID + 1, B(100, J), 1)
      IF (lo == 1) THEN
         RECV(PID - 1, B(0, J), 1)
         A(1, J) = B(0, J) + C
      ENDIF
710   DO i = 2, hi
         A(i, J) = B(i - 1, J) + C
      ENDDO
   ENDIF
ENDDO
```

如果可以确定能够将J-循环围绕代码中的三个组成部分分布为

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP) hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   DO J = 1, M
      IF (PID /= lastP) &
         SEND(PID + 1, B(100, J), 1)
   ENDDO
   DO J = 1, M
      IF (lo == 1) THEN
         RECV(PID - 1, B(0, J), 1)
         A(1, J) = B(0, J) + C
      ENDIF
   ENDDO
   DO J = 1, M
      DO i = 2, hi
         A(i, J) = B(i - 1, J) + C
      ENDDO
   ENDDO
ENDIF
```

那么我们可以将代码中所有的接收操作向量化，产生长得多的消息:

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP) hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
```

```
      IF (lo == 1) THEN
         RECV(PID - 1, B(0, 1:M), M)
         DO J = 1, M                                          711
            A(1, J) = B(0, J) + C
         ENDDO
      ENDIF
      DO J = 1, M
         DO i = 2, hi
            A(i, J) = B(i - 1, J) + C
         ENDDO
      ENDDO
      IF (PID /= lastP) &
         SEND(PID + 1, B(100, 1:M), M)
   ENDIF
```

那么实现这样一种分布的条件是什么呢？换句话说，我们何时可以将发送和接收语句移动到包含它们的循环（譬如J-循环）之外？为了理解这个问题，我们使用在前一节中用过的依赖计算策略——将通信语句看作是在局部数据与全局数据之间的拷贝。在第一个代码生成步骤之后，循环变成

```
      DO J = 1, M
         lo = 1
         IF (PID == 0) lo = 2
         hi = 100
         lastP = CEIL((N + 1) / 100) - 1
         IF (PID == lastP)   hi = MOD(N, 100) + 1
         IF (PID <= lastP) THEN
S1          IF (PID /= lastP) B_g(100, J) = B(100, J)
            ! SEND(PID + 1, B(100, J), 1)
            IF (lo == 1) THEN
S2             B(0, J) = B_g(0, J) ! RECV(PID - 1, B(0, J), 1)
S3             A(1, J) = B(0, J) + C
            ENDIF
            DO i = 2, hi
S4             A(i, J) = B(i - 1, J) + C
            ENDDO
         ENDIF
      ENDDO
```

假设J-循环可以和IF语句进行交换，问题在于外层循环能否围绕IF语句中的语句分布。如果J-循环没有携带包含通信语句的依赖环，这种分布是可能的。在前面的例子中，带标记的语句间惟一的依赖是S_2和S_3间的依赖。因此，没有依赖环包含J-循环，从而所有的通信都可以向量化。

我们现在可以将这些思想概括为通信向量化的规则。 712

原理14.1 如果由内层循环生成的通信语句不包含在外层循环携带的依赖环中，这些通信语句相对于外层循环是可以向量化的。

为了说明这个原理的合法性，请回忆我们知道循环分布是可能的，只要对其实施分布的语句

不构成一个由被分布的循环携带的依赖环（见2.4.2节）。如果存在这样一个依赖环，则分布不能被实施，且通信向量化是被禁止的。

下面是另外一个例子，与我们原来的例子稍有不同:

```
REAL A(10000, 100), B(10000, 100)
!HPF$ DISTRIBUTE  A(BLOCK, *), B(BLOCK, *)
DO J = 1, M
   DO I = 1, N
      A(I + 1, J + 1) = A(I + 1, J) + B(I, J)
   ENDDO
ENDDO
```

与前面的例子类似，由于对I-循环的划分，我们需要对数组B的通信。另一方面，对A的引用不需要通信，因为这些引用与I-循环的同一迭代是对齐的。因此这个循环将有一个与前面例子相同的通信依赖模式，所以此处的通信也是可以被向量化的。

另一方面，假设我们造成了一种情景，其中需要对被计算的变量A进行通信。

```
REAL A(10000, 100), B(10000, 100)
!HPF$ DISTRIBUTE  A(BLOCK, *), B(BLOCK, *)
DO J = 1, M
   DO I = 1, N
      A(I + 1, J) = A(I, J) + B(I, J)
   ENDDO
ENDDO
```

在最初的代码生成步骤后，得到的代码看起来如像下面的形式，其中的通信同前面一样已经用拷贝替换:

```
      DO J = 1, M
         lo = 1
         IF (PID == 0) lo = 2
         hi = 100
         lastP = CEIL((N + 1) / 100) - 1
713      IF (PID == lastP)   hi = MOD(N, 100) + 1
         IF (PID <= lastP) THEN
            IF (lo == 1) THEN
               B(0, J) = B_q(0, J) ! RECV(PID - 1, B(0, J), 1)
S1             A(0, J) = A_q(0, J) ! RECV(PID - 1, A(0, J), 1)
S2             A(1, J) = A(0, J) + B(0, J)
            ENDIF
            DO i = 2, hi
S3             A(i, J) = A(i - 1, J) + B(i - 1, J)
            ENDDO
S4          IF (PID /= lastP) THEN
               B_q(100, J) = B(100, J) ! SEND(PID + 1, B(100, J))
               A_q(100, J) = A(100, J) ! SEND(PID + 1, A(100, J))
            ENDIF
         ENDIF
      ENDDO
```

虽然发送B的值的语句可以移动到开始的位置，但此处由I-循环携带的依赖阻止我们重排发送语句和接收语句。然而，J-循环不携带任何依赖，所以通信仍然可以被向量化:

```
      lo = 1
      IF (PID == 0) lo = 2
      hi = 100
      lastP = CEIL((N + 1) / 100) - 1
      IF (PID == lastP)   hi = MOD(N, 100) + 1
      IF (PID <= lastP) THEN
         IF (lo == 1) THEN
            RECV(PID - 1, B(0, 1:M), M)
S1          RECV(PID - 1, A(0, 1:M), M)
            DO J = 1, M
S2             A(1, J) = A(0, J) + B(0, J)
            ENDDO
         ENDIF
         DO J = 1, M
            DO i = 2, hi
S3             A(i, J) = A(i - 1, J) + B(i - 1, J)
            ENDDO
         ENDDO
S4       IF (PID /= lastP) THEN
            SEND(PID + 1, B(100, 1:M), M)
            SEND(PID + 1, A(100, 1:M), M)
         ENDIF
      ENDIF
```

714

最后，我们给出一个不可能向量化的例子。

```
REAL A(10000, 100)
!HPF$ DISTRIBUTE  A(BLOCK, *)
DO J = 1, M
   DO I = 1, N
      A(I + 1, J + 1) = A(I, J) + C
   ENDDO
ENDDO
```

在对I-循环实施划分并插入通信后，我们得到

```
      DO J = 1, M
         lo = 1
         IF (PID == 0) lo = 2
         hi = 100
         lastP = CEIL((N + 1) / 100) - 1
         IF (PID == lastP)   hi = MOD(N, 100) + 1
         IF (PID <= lastP) THEN
            IF (lo == 1) THEN
S0             A(0, J) = Aq(0, J) ! RECV(PID - 1, B(0, J), 1)
S1             A(1, J + 1) = A(0, J) + C
            ENDIF
            DO i = 2, hi
S2             A(i, J + 1) = A(i - 1, J) + C
            ENDDO
            IF (PID /= lastp) THEN !SEND(PID + 1, A(100, J + 1))
S3             Aq(100, J + 1) = A(100, J + 1)
```

```
          ENDIF
        ENDIF
      ENDDO
```

此处的通信语句不能被向量化，因为存在一个由J-循环携带的依赖环。请注意依赖环的距离与数组A的块长是相等的。因此，我们可以通过循环分段得到部分的通信向量化，如下面代码所示：

```
      DO J = 1, M, 100
        lo = 1
        IF (PID == 0) lo = 2
        hi = 100
        lastP = CEIL((N + 1) / 100) - 1
        IF (PID == lastP)   hi = MOD(N, 100) + 1
        IF (PID <= lastP) THEN
715       IF (lo == 1) THEN
S0          RECV(PID - 1, A(0, J:J + 99), 100)
            DO j = J, J + 99
S1            A(1, j + 1) = A(0, j) + C
            ENDDO
          ENDIF
          DO j = J, J + 99
            DO i = 2, hi
S2            A(i, j + 1) = A(i - 1, j) + C
            ENDDO
          ENDDO
          IF (PID /= lastp) THEN
S3          SEND(PID + 1, A(100, J:J + 99), 100)
          ENDIF
        ENDIF
      ENDDO
```

14.4.2 重叠通信和计算

通过将通信与计算重叠，通常可以隐藏通信的代价。为了说明这种情况是如何发生的，让我们回到前小节中用到的例子。

```
REAL A(10000), B(10000)
!HPF$ DISTRIBUTE  A(BLOCK), B(BLOCK)
   ...
DO I = 1, N
  A(I + 1) = B(I) + C
ENDDO
```

我们已经看到这个例子可以得到下述局部代码：

```
    lo = 1
    IF (PID == 0) lo = 2
    hi = 100
    lastP = CEIL((N + 1) / 100) - 1
    IF (PID == lastP)   hi = MOD(N, 100) + 1
    IF (PID <= lastP) THEN
S0    IF (PID /= lastP) SEND(PID + 1, B(100), 1)
```

```
S1      IF (lo == 1) THEN
           RECV(PID - 1, B(0), 1)
           A(1) = B(0) + C
        ENDIF
L1      DO i = 2, hi
           A(i) = B(i - 1) + C
        ENDDO
     ENDIF
```

716

请注意循环L_1中的代码都不依赖于同其他处理器通信得到的值。所以，我们可以对代码重排序，使得这段计算出现在发送和接收之间。

```
     lo = 1
     IF (PID == 0) lo = 2
     hi = 100
     lastP = CEIL((N + 1) / 100) - 1
     IF (PID == lastP)   hi = MOD(N, 100) + 1
     IF (PID <= lastP) THEN
S0      IF (PID /= lastP) SEND(PID + 1, B(100), 1)
L1      DO i = 2, hi
           A(i) = B(i - 1) + C
        ENDDO
S1      IF (lo == 1) THEN
           RECV(PID - 1, B(0), 1)
           A(1) = B(0) + C
        ENDIF
     ENDIF
```

这个被称为*迭代重排序*的变换在发送和接收之间隐藏尽可能多的通信延迟。如果所有的处理器是同时发送的，则计算的时间可以从实际的通信时间中减去，显著地降低通信的代价。

迭代重排序总是可以用于原来的循环不携带任何依赖的地方——也就是，只要在生成的代码中没有依赖强制接收操作必须先于循环体中的计算。

通信优化的另一种方法，是在控制流图内将通信移动到程序中可能的最早点。通过建立一组数据流方程，并利用与部分冗余判定类似的计算过程求解这组方程，即可以确定正确的插入位置[137]。

14.4.3 对齐和复制

有了14.3节中介绍的将一个循环的迭代划分的方案，我们不需要使用使得每个循环尽可能小的循环分布技术。我们可以采用另外一种方式，在一个大循环中选择一个划分引用，并将整个循环一起划分。这种方案的缺点是这个循环可能携带了许多依赖，每一个依赖都要求通信。在本小节中，我们介绍如何改善这个问题。

717

假设我们为一个循环生成代码，这个循环是在一个分布的维上进行迭代——也就是说，循环的迭代将在不同的处理器上执行。那么，这个循环携带的每个真依赖都要求通信。因此我们如果可以消除这些携带的依赖，总的通信量将会减少。

我们暂且假设循环中没有依赖环。那么我们从6.2节知道借助循环对齐和代码复制，所有携带的依赖都可以消除。考虑一下下面的代码:

```
DO I = 1, N
   A(I + 1) = B(I) + C(I)
   D(I) = A(I + 1) + A(I)
ENDDO
```

这个循环中有一个包含A的阈值为1的携带依赖，还有一个包含A的不能单独用对齐求解问题的循环无关依赖。使用循环对齐和代码复制，这个循环被变换为

```
DO I = 1, N
   T0 = B(I) + C(I)
   A(I + 1) = T0
   IF (I == 1) THEN
      T1 = A(I)
   ELSE
      T1 = B(I - 1) + C(I - 1)
   ENDIF
   D(I) = T0 + T1
ENDDO
```

此后，代码生成过程可以用于这个结果，得到一个完全并行循环，虽然在每个迭代中需要少许额外的计算。

在存在依赖环的情形，这个策略可以用来将依赖环执行中需要的总通信数量减到最少。基本思想是通过排列代码，使得存在的后向依赖数目最小，然后对齐循环体，从而消除前向携带的依赖。

718 这个策略将可以把流水线处理的重点转向只关心后向携带的依赖。

14.4.4 流水

当一个循环嵌套的内层循环必定是串行化的，如果外层循环不含有跨越处理器的依赖，有可能达到某种并行性。下面是一个简单的例子:

```
REAL A(10000, 100)
!HPF$ DISTRIBUTE A(BLOCK, *)
DO J = 1, M
   DO I = 1, N
      A(I + 1, J) = A(I, J) + C
   ENDDO
ENDDO
```

循环I的初始代码生成将产生

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP)   hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   DO J = 1, M
      IF (lo == 1) THEN
         RECV(PID - 1, A(0, J), 1)
         A(1, J) = A(0, J) + C
      ENDIF
      DO i = 2, hi
```

```
            A(i, J) = A(i - 1, J) + C
         ENDDO
         IF (PID /= lastP) &
            SEND(PID + 1, A(100, J), 1)
      ENDDO
   ENDIF
```

虽然在这个例子中，通信操作可以被完全向量化，但这样做是不明智的，因为这样就放弃了并行性。事实上，上面形式的循环比一个向量化循环有更多的并行性，因为一旦处理器0完成J-循环的第一次迭代，处理器1就可以开始计算其局部的A(1:100,1)。换句话说，J-循环的迭代可以被重迭。这可以通过图14-3说明。

这个方案的问题是每次通信一个字的开销将超过并行性获得的收益。图14-3是容易令人误解的，因为它没有显示出任何通信启动或实现的开销。图14-4给出了一个更接近现实的图示，它显示出为实现一次发送而在处理器和通信延迟两方面的开销。 719

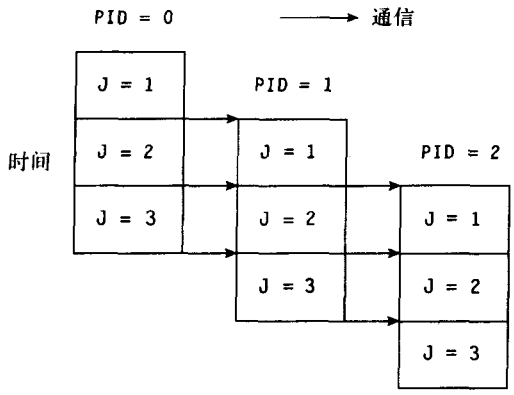


图14-3 伴有通信的流水线并行性

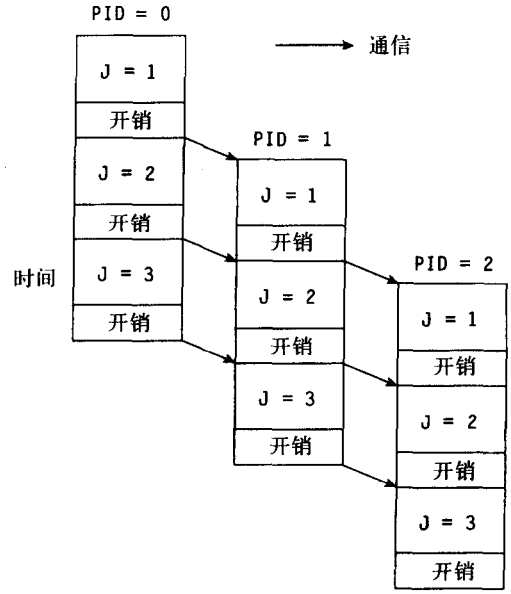


图14-4 伴有通信开销的流水线并行性 720

由于无论有多少个字要发送，大多数开销的代价是同样的，所以通过在传送数值前计算J-循环的几次迭代从而增加流水的粒度通常是有利的，如下面代码中那样:

```
lo = 1
IF (PID == 0) lo = 2
hi = 100
lastP = CEIL((N + 1) / 100) - 1
IF (PID == lastP)   hi = MOD(N, 100) + 1
IF (PID <= lastP) THEN
   DO J = 1, M, K
      IF (lo == 1) THEN
         RECV(PID - 1, A(0, J:J + K - 1), K)
         DO j = J, J + K - 1
```

```
            A(1, j) = A(0, j) + C
         ENDDO
      ENDIF
      DO j = J, J + K - 1
         DO i = 2, hi
            A(i, j) = A(i - 1, j) + C
         ENDDO
      ENDDO
      IF (PID /= lastP) &
         SEND(PID + 1, A(100, J:J + K - 1), K)
   ENDDO
ENDIF
```

此处的块因子K是一个可以根据机器和循环中的计算数量两方面因素调整的参数。可以使用多种不同的策略确定这个参数。例如，在每个机器上一次试验性的运行可以为短的循环决定最优的分块。然后，这个同样的分块可以用于更大的循环。

14.4.5 一般依赖环的识别

在并行计算机上实现的许多计算包括重复从在多个内存中分布的数组取值（作为输入）的操作。大多数这样的依赖环是标准的模式，如求和归约。这些模式通常在每台机器上有一个快速的库实现。一个好的HPF编译器应该能够借助使用这样的库例程来识别和替换这种原始的计算。作为一个例子，假设我们有如下的程序:

```
REAL A(10000), B(10000)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK)
   ...
S = 0.0
DO I = 1, N
   S = S + A(I) * B(I)
ENDDO
```

721

假设此处的变量S在每一个处理器上被复制，结果是在每个处理器上有S的一个值，这个值等于A和B之积的和。编译器必须首先将这个计算分解为局部和全局的成分。假设有100个处理器，因此块长是100。

```
lo = 1
hi = 100
lastP = CEIL(N / 100) -1
IF (PID == lastP)   hi = MOD(N - 1, 100) + 1
Slocal = 0.0
IF (PID <= lastP) THEN
   DO i = lo, hi
      Slocal = Slocal + A(i) * B(i)
   ENDDO
ENDIF
```

一旦这一步完成，最终的值就可以通过调用库例程

```
S = GLOBAL_SUM(Slocal)
```

计算出来，其他标准的依赖环可以用同样的方式处理。

14.4.6 存储管理

一个好的HPF编译器必须采取步骤保证它生成的代码不会要求超过可能的临时存储空间。通常，一个计算应该不会要求获得比每个处理器拥有的数组更多的临时存储空间。换句话说，我们将假设在每个处理器上不会有超过一半的内存被用来保存通信的值。

为了遵守这些约定，可能要求编译器对计算做明显的重组。为了说明这一点，我们将考虑矩阵乘法的例子，如下面的代码所示：

```
!HPF$ ALIGN  A(I, J), B(I, J), C(I, J) WITH T(I, J)
!HPF$ DISTRIBUTE  T(BLOCK, BLOCK) ONTO P(4, 4)
DO K = 1, N
   DO J = 1, N
      DO I = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

722

应该指出，在这个循环中，通信可以移动到所有循环之外，这是编译器容易确定的一个事实。但是应该实施这个移动吗？在不同的代码位置，存储的需求是什么？如同我们将看到的，将通信放置在最外层循环之内可以获得对存储需求的显著减少。

首先，如果试图将这些代码完全移动到最外层循环之外，让我们检查会发生什么。如果做了代码移动，生成的代码看起来可能像下面形式，其中我们假设局部存储保存A，B和C的大小$n \times n$的块：

```
! 与资源无关的放置
   DO pR = 0, nRows - 1
      IF (pR /= myR) &
         SEND((pR, myC), B(n * myR + 1:n * myR + n, 1:n), n * n)
   ENDDO
   DO pC = 0, nCols - 1
      IF (pC /= myC) &
         SEND((myR, pC), A(1:n, myC * n + 1:myC * n + n), n * n)
   ENDDO
   DO pR = 0, nRows - 1
      IF (pR /= myR) &
         RECV((pR, myC), B(n * pR + 1:n * pR + n, 1:n), n * n)
   ENDDO
   DO pC = 0, nCols - 1
      IF (pC /= myC) &
         RECV((myR, pC), A(1:n, n * pC + 1:n * pC + n), n * n)
   ENDDO
   DO K = 1, n * 4 ! = N
      DO J = 1, n
         DO I = 1, n
            C(I, J) = C(I, J) + A(I, K) * B(K, J)
         ENDDO
      ENDDO
   ENDDO
```

请注意特殊变量myR和myC指明执行计算的处理器的行和列，因此数据对(myR,myC)等价于当

前的PID。

这段代码的问题是，它需要数组A的4块存储空间，每块的大小是n^2，数组B同样大小的4块存储空间，以及数组C的1块存储空间。因此需要的总存储空间是9个大小为n^2的块。这些块中只有3块是局部的。所以，在这个通信模式中，每个处理器为了与其他处理器通信需要$6n^2$的存储单元——是局部变量需要的存储空间的两倍。这个问题在图14-5中显示。

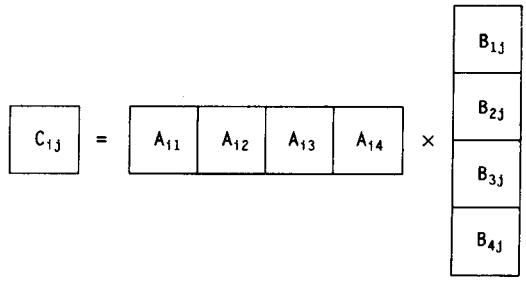


图14-5 矩阵乘法需要的存储空间

在这里，每个处理器需要的存储空间是计算块C_{1j}时A的每个块和B的每个块需要的存储空间。

如果将通信留在最外层循环之内，我们得到非常适度的存储空间需求，如下面的代码所示：

```
! 基于资源的放置
  DO K = 1, n * 4 ! = N
     kP = CEIL(K / n) - 1
     kloc = MOD(K - 1, n) + 1
     sR = MOD(myR + kP, 4); sC = MOD(myC + kP, 4)
     rR = MOD(myR - kP, 4); rC = MOD(myC - kp, 4)
     IF (sR /= myR) SEND((sR, myC), B(kloc, 1:n), n)
     IF (sC /= myC) SEND((myR, sC), A(1:n, kloc), n)
     IF (rR /= myR) THEN
        kR = n + 1
        RECV((rR, myC), B(kR, 1:n), n)
     ELSE
        kR = kloc
     ENDIF
     IF (rC /= myC) THEN
        kC = n + 1
        RECV((myR, rC), A(1:n, kC), n)
     ELSE
        kC = kloc
     ENDIF
     DO J = 1, n
        DO I = 1, n
           C(I, J) = C(I, J) + A(I, kC) * B(kR, J)
        ENDDO
     ENDDO
  ENDDO
```

723
~
724

这个方案需要的存储空间是非常小的——每个处理器只需要$2n$个额外的存储单元。这些单元是被分配到数组A的第$n+1$列和数组B的第$n+1$行。这个数量的存储单元比局部数组要求的$3n^2$个存储单元小得多。所以这是通信插入的最好位置。但是，如果我们对K-循环进行如下分块，可以获得更好的性能：

```
! 基于资源的放置
DO kP = 0, 3
```

```
      sR = MOD(myR + kP, 4); sC = MOD(myC + kP, 4)
      rR = MOD(myR - kP, 4); rC = MOD(myC - kp, 4)
      IF (sR /= myR) SEND((sR, myC), B(1:n, 1:n), n * n)
      IF (sC /= myC) SEND((myR, sC), A(1:n, 1:n), n * n)
      IF (rR /= myR) THEN
         kR = n
         RECV ((rR, myC), B(n + 1:n + n, 1:n), n * n)
      ELSE
         kR = 0
      ENDIF
      IF (rC /= myC) THEN
         kC = n
         RECV ((myR, rC), A(1:n, n + 1:n + n), n * n)
      ELSE
         kC = 0
      ENDIF
      DO K = 1, n
         DO J = 1, n
            DO I = 1, n
               C(I, J) = C(I, J) + A(I, k + kC) * B(k + kR, J)
            ENDDO
         ENDDO
      ENDDO
   ENDDO
```

在局部存储之外，这个例子要求两个额外的存储块，每块的大小是n^2。结果通信需要的存储大约是局部存储的2/3。应该指出，在许多并行系统上这个程序是实现矩阵乘法的最优方式。

归结起来，一个HPF编译器必须分析一个循环嵌套中每一种通信安排所要求的缓冲区资源。然后，应该选择需要的存储空间不超过局部变量所需存储空间的安排。在一些例子中，可以使用循环分段技术减少通信频度，其代价是需要不超限定的某些存储空间，如前面的例子所示。 725

14.4.7 处理多个维

我们以两维通信的讨论作为本节的结束。假设给定循环

```
REAL A(1000, 1000) B(1000, 1000)
!HPF$ DISTRIBUTE A(BLOCK, BLOCK), B(BLOCK, BLOCK)
   ...
DO J = 1, M
   DO I = 1, N
      A(I + 1, J + 1) = B(I, J + 1) + B(I + 1, J)
   ENDDO
ENDDO
```

并假定我们有100个处理器，排列成一个10 × 10的网格。按照与前面例子类似的处理，对内层循环生成看起来可能像下面形式的代码:

```
ilo = 1
IF (myR == 0) ilo = 2
ihi = 100
ilastP = CEIL((N + 1) / 100) - 1
```

```
IF (myR == ilastP)   ihi = MOD(N, 100) + 1
IF (ilo == 1) THEN
   RECV((myR - 1, myC), B(0, J + 1), 1)
ENDIF
DO i = ilo, ihi
   A(i, J + 1) = B(i - 1, J + 1) + B(i, J)
ENDDO
IF (myR /= ilastP) &
   SEND((myR + 1, myC), B(100, J + 1), 1)
```

如果我们对外层循环进行同样的处理，在适当的地方实施分布，我们得到

```
! 计算局部循环的上下界
jlo = 1
IF (myC == 0) jlo = 2
jhi = 100
jlastP = CEIL((M + 1) / 100) - 1
IF (myC == jlastP) jhi = MOD(M + 1, 100) + 1
ilo = 1
IF (myR == 0) ilo = 2
ihi = 100
ilastP = CEIL((N + 1) / 100) - 1
IF (myR == ilastP) ihi = MOD(N + 1, 100) + 1
! 接收数据
IF (jlo == 1) THEN
   DO i = ilo, ihi
      RECV((myR, myC - 1), B(i, 0), 1)
   ENDDO
ENDIF
IF (ilo == 1) THEN
   DO j = jlo, jhi
      RECV((myR - 1, myC), B(0, j), 1)
   ENDDO
ENDIF
! 计算
DO j = jlo, jhi
   DO i = ilo, ihi
      A(i, j) = B(i - 1, j) + B(i, j - 1)
   ENDDO
ENDDO
! 发送数据
IF (myR /= ilastP) THEN
   DO j = jlo, jhi
      SEND((myR + 1, myC), B(100, j), 1)
   ENDDO
ENDIF
IF (myC /= jlastP) THEN
   DO i = ilo, ihi
      SEND((myR, myC + 1), B(i, 100), 1)
   ENDDO
ENDIF
```

726

我们注意到通信是可以被向量化的，且计算和通信是可以重叠的，所以在优化后，代码变成

```
! 计算局部循环的上下界
jlo = 1
IF (myC == 0) jlo = 2
jhi = 100
jlastP = CEIL((M + 1) / 100) - 1
IF (myC == jlastP) jhi = MOD(M, 100) + 1
nJ = jhi - jlo + 1
ilo = 1
IF (myR == 0) ilo = 2
ihi = 100
ilastP = CEIL((N + 1) / 100) - 1
IF (myR == ilastP)   ihi = MOD(N, 100) + 1
nI = ihi - ilo + 1
! 发送数据
IF (myR /= ilastP) THEN
   SEND((myR + 1, myC), B(100, jlo:jhi), nJ)
ENDIF
IF (myC /= jlastP) THEN
   SEND((myR, myC + 1), B(ilo:ihi, 100), nI)
ENDIF
! 计算
DO j = MAX(2, jlo), jhi
   DO i = MAX(2, ilo), ihi
      A(i, j) = B(i - 1, j) + B(i, j - 1)
   ENDDO
ENDDO
! 接收数据和计算
IF (jlo == 1) THEN
   RECV((myR, myC - 1), B(ilo:ihi, 0), nI)
   DO i = MAX(2, ilo), ihi
      A(i, 1) = B(i - 1, 1) + B(i, 0)
   ENDDO
ENDIF
IF (ilo == 1) THEN
   RECV((myR - 1, myC), B(0, jlo:jhi), nJ)
   DO j = jlo, jhi
      A(1, j) = B(0, j) + B(1, j - 1)
   ENDDO
ENDIF
```

727

注意在最后这个版本中，我们已经将依赖于从其他处理器上接收数据的计算的执行剥离出来，所以允许将通信和只包含计算的循环重叠起来。

14.5 HPF的过程间优化

虽然HPF编译器可以从许多过程间分析和优化技术中获得好处，但其中的两个处理技术是特别重要的。首先，如同我们早先提到的，实施过程间的数据分布信息的追踪，以尽力保证在程序中的每一点只有一个到达的数据分布模式。在那些数据分布模式在编译时刻已知的

728

程序点，由于不需要进行运行时的测试，所以可以生成非常高效的代码。

到达分布模式的问题可以归结为一个与过程间常数传播的直接模拟。在这个抽象过程中，已知的分布模式作为常数值，而变量就是与分布数组相关联的未知的分布模式。如同在常数传播中一样，过程克隆技术可以被用来减少到达一个给定的子程序的不同分布模式个数。如果两个不同的调用链会向同一个过程传送不同的分布模式，则将这个过程克隆为两个拷贝，每个调用链对应一个拷贝，消除不确定性。

第二个重要的过程间变换是通信生成。考虑下面的循环:

```
!HPF$ ALIGN A(I, J), B(I, J) WITH T(I, J)
!HPF$ DISTRIBUTE T(BLOCK, *) ONTO P(16)
   ...
   DO J = 1, N
      CAll SUB(A(*, J), B(*, J))
   ENDDO
   ...
SUBROUTINE SUB(X, Y)
   !HPF$ INHERIT X, Y;
   DIMENSION X(*), Y(*)
   DO I = LBOUND(X), UBOUND(X)
      X(I) = Y(I + 1) + C
   ENDDO
END
```

在这段代码中，过程间的数据分布分析决定子程序内X和Y在循环的迭代间采用BLOCK方式分布。但是，分布的数组包含了一个与I-循环相关的通信需求。这个通信可以被移动到循环之外。但是，对于大多数高效的操作，我们应该将通信移动到子程序调用之外，在那里这个通信可以被向量化。这种形式的变换技术称为*过程间通信优化*。

14.6 小结

我们已经证明本书前面提出的那些原理对于编译和优化高性能Fortran（HPF）是有效的。
729 基本的编译算法由以下的内容组成:

(1) 分布分析和传播

(2) 迭代的划分

(3) 通信生成

很多优化技术可以提高由编译器生成的通信的性能，且保证不会违反资源的限制。这些优化技术包括通信向量化，通信和计算的重叠，对齐和复制，流水线，归约识别，以及存储分析和管理。

14.7 实例研究

本书的作者曾经参加了两个具有HPF语言特征的编译系统的实现。由Hiranandani，Kennedy和Tseng领导的最初的Fortran D编译器项目[149，150，151]实现了本章中介绍的大多数分布特性，包括根据拥有者计算原则实施计算划分、通信设置和特殊情形的识别（例如归约识别）。主要的优化工作是关注于提高通信的性能。该项目使用了通信向量化，计算和通信的重叠，

以及粗粒度流水技术。对包括线性代数和松弛计算这样几个核心程序和小型应用程序的试验，说明这种方法是有前途的。表14-1总结用于试验的核心程序和应用程序。前三行是众所周知的核心程序，包括一个求和归约和两个模板核心程序。其后的两行是需要通过流水线获得更高并行性的核心程序。最后有四个应用程序，包括一个高斯消去法基准测试程序和一个简单的浅水气候模型。

表14-1 HPF核心程序和应用程序

程 序 名	程序类型	数据大小
Livermore 3: Inner Product	归约核心程序	1024k
Jacobi Iteration	核心程序	2k × 2k
Livermore 18: Explicit Hydrodynamics	核心程序	512 × 512
Successive Over Relaxation	流水线核心程序	2k × 2k
Livermore 23: Implicit Hydrodynamics	流水线核心程序	1k × 1k
Shallow	应用程序	1k × 1k
Disper	应用程序	256 × 8 × 8 × 4
DGEFA	线性代数基准测试程序	2k × 2k
Erlebacher	应用程序	128 × 128 × 128

730

所有的性能测量都是在一个有32个处理器的Intel iPSC 860上进行的。HPF编译器版本是从Fortran D编译器发展而来，结果程序的运行时间是与使用Intel通信库开发的手写消息传递版本的程序相比较（图14-6）。加速比超过了程序串行执行时间。在一些例子中得到了超线性加速比，这是因为并行版本的程序适合于一个单独的处理器的高速缓存。在流水核心程序中，Fortran D编译器（产生的代码）胜过了手写版本的程序，这可能是由于i860节点编译器的异常行为。Tseng的学位论文报告了这个研究中得到的实验结果的细节[262]。

虽然实验被限定在相对较小的程序上，这些结果表明一个好的编译器可以为HPF程序生成代码，使这个代码的性能与手工使用消息传递库开发的代码性能相当。结果，Fortran D的成功对于早期的商用编译器项目有着重要的影响。

在20世纪90年代中期，被称为dHPF的第二个实现HPF的项目是在Rice大学发起的，由John Mellor-Crummey和Vikram Adve领导。这个项目试图借助使用更积极的策略实现计算划分和通信的优化，从而克服Fortran D和商用HPF编译器的限制。在原来的Fortran D工作的基础上作了两个重要的改进。首先，拥有者计算原则被放松，允许使通信达到最小化的计算划分。这种方法与14.4.3节中讨论的对齐策略类似。第二，这个项目实现了贯穿本章讨论的集合操作，其中使用了由Maryland大学开发的Omega系统[166，228]。这个系统提供了一种便利的方法，实现HPF要求的索引集分裂操作。当与对齐结合使用时，这种方法允许使用计算的部分重复而减少通信代价。Omega也被用于代码生成，因为它可以生成循环集合，处理所有集合分析需要的特殊情形。实验显示出这些改进在处理真实代码时获得了很好的回报。特别，对于编码风格没有对编译器的能力和缺点让步的NAS并行基准测试程序SP和BT，HPF编译器生成程序的性能可以得到由NAS手工开发的程序的性能90%以上[73]。

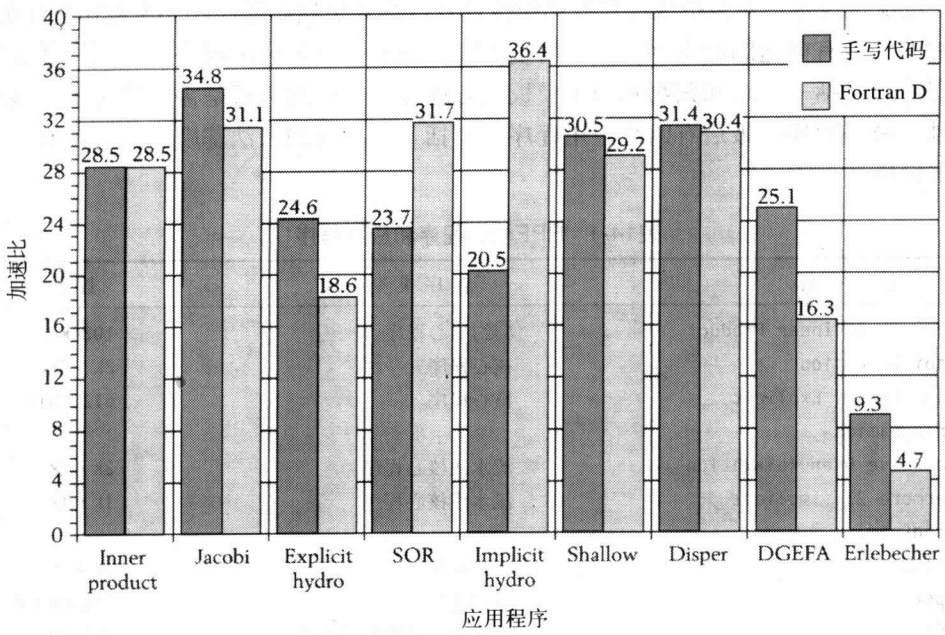


图14-6 HPF核心程序和应用程序与手写程序性能对比

14.8 历史评述与参考文献

HPF语言的根基是Rice大学开发的Fortran D项目[115]和Vienna大学开发的Vienna Fortran项目[71，289]。这两个项目是基于分布计算的编译系统中最常见的系统。高性能Fortran自身在几年后由Kennedy领导的高性能Fortran论坛标准化[146，147]。

许多研究者已经发表了关于分布式存储的编译技术的文章，这些编译技术是基于分布说明的。两篇最早的关于分布式存储策略编译技术的文章分别由Callahan和Kennedy[60]及Zima，Bast和Gerndt[288]发表。本章是基于Hiranandani，Kennedy和Tseng的工作的，他们在Rice大学发布了一个实验性的HPF编译器[149，150，151]。一些重要的编译策略也是由Hatcher，Quinn等人[140]，Koelbel和Mehrotra[187]及Rogers和Pingali[235]给出的。由Fox领导的一个小组发展了另一种HPF实现策略，基于广泛使用包通信（packaged communication）和计算库[42]。最近，由John Mellor-Crummey和Vikram Adve领导的Rice大学的dHPF计划采用了更加主动的优化策略[8，7]，这些策略是构建在强有力的整数集合处理软件基础上的，这个软件是Maryland大学的Pugh及其同事开发的[166，228]。Kennedy和Sethi开发了此处给出的基于资源的通信安排方法[181]。

731 ~ 732

习题

14.1 与对串行Fortran程序的编译相比，编译器为一个HPF程序生成代码需要哪些额外的步骤？

14.2 如果下面的程序在一个4处理器的机器上执行，计算每个处理器通信的总字节数。你能够改写数据分布方式，以使得本程序没有通信吗？

```
REAL A(N), B(N)
```

```
!HPF$ TEMPLATE T(N)
!HPF$ ALIGN A(I) WITH T(I)
!HPF$ ALIGN B(I) WITH T(I)
!HPF$ DISTRIBUTE T(BLOCK)
DO I = 1, N - 1
   A(I) = B(I + 1)
ENDDO
```

14.3 如果下面的程序在一个4处理器的机器上执行，计算每个处理器通信的总字节数。假设数据分布的方式已选定，你能够给出另外一种方法使得迭代的划分是均匀的而通信量减少一半吗？

```
REAL A(N), B(N)
!HPF$ TEMPLATE T(N)
!HPF$ ALIGN A(I) WITH T(I)
!HPF$ ALIGN B(I) WITH T(I)
!HPF$ DISTRIBUTE T(BLOCK)
DO I = 1, N - 1
   A(I) = A(I + 1) + B(I + 1)
ENDDO
```

14.4 当下面的循环嵌套在4个处理器上执行时，通信消息的次数是多少？你能够改写该程序把消息的次数减少到原来的一半吗？你能够将总的消息次数减少到3次吗？你的优化导致了任何负面效果吗？

```
REAL A(N, N)
!HPF$ TEMPLATE T(N, N)
!HPF$ ALIGN A(I, J) WITH T(I, J)
!HPF$ DISTRIBUTE T(*, BLOCK)
DO I = 1, N
   DO J = 1, N - 1
      A(I, J) = A(I - 1, J) + 1.0
   ENDDO
ENDDO
```

733

14.5（承接上一个问题）减少消息次数的方法是对I-循环展开（展开次数由k因子决定），并和J-循环交换，如下面的程序所示。这种变换称为粗粒度的流水变换。这种变换总是合法的吗？（提示：这种变换遵循与展开与压紧一样的合法性条件。）

```
REAL A(N, N)
!HPF$ TEMPLATE T(N, N)
!HPF$ ALIGN A(I, J) WITH T(I, J)
!HPF$ DISTRIBUTE T(*, BLOCK)
DO II = 1, N, K
   DO J = 1, N - 1
      DO I = II, II + K - 1
         A(I, J) = A(I - 1, J) + 1.0
      ENDDO
   ENDDO
ENDDO
```

14.6 *编程实习* 对习题14.4和14.5中的例子生成使用PVM或MPI的并行代码。评估粗粒度流水线变换获得的性能。你能够给出最优的循环展开因子吗？测试你的方法。

附 录

Fortran 90基础

引言

在本附录中我们对Fortran 90（以及较新的标准Fortran 95）的特征提供一个简要的介绍，它们关系到理解本书的内容。此介绍仅是语言特性的很小子集。有兴趣的读者可以参考在附录末尾“补充读物”中列举的文献，了解更多的语言特性。

词法特性

Fortran 90引入了称为*自由源码形式*的新词法输入格式，在这种形式中老的面向列的形式被面向行的形式所替代，它允许在一行上有多个语句。因为本书全部使用自由源码形式，我们集中关注对于理解正文最相关的特征。

如同在老的形式中一样，假设语句是由行的结尾终止的。但是如果行中包含一个不在注释中的&符号，则语句在下一行上继续。如果关键字或字符串被跨行分开，那么续行的开始必须也是一个&符号。

多个语句可以出现在一行中，它们用分号（；）隔开。在一行结尾前面的最后一个语句不需要用分号终止。

734
~
735

如果惊叹号（！）出现在带引号的字符串或其他注释外则惊叹号用于开始注释行。

空白符是有意义的，并且不能出现在关键字、名字或数中。例外情况是关键字本身确实是由多个字组成的短语，如“END DO”。

与以前的Fortran版本一样，标号是数字型的，但是它们可以出现在语句前的任何地方，甚至在第6列之后。另外，可以用*结构名*去标记一个结构，如DO-循环。结构名是一个变量后面跟一个冒号。结构名在同一程序中不可以用作变量名。结构名可以用在EXIT和CYCLE语句中，作为特定结构的出口，或在循环的情况下，转到结构的下一次迭代。

Fortran 90引入新的比较操作符（==，/=，< ，>，<= ，>=）用于替代老的带点的相应记法。

下面的例子说明其中的某些概念：

```
    PROGRAM MAIN
        REAL A(10000), B(10000) &
             C(10000,10000) ! this is the big data array
        CALL GETDATA(A,B,C);CALL CHECKDATA(A,B,C)
OUTER:  DO I = 1, 10000
INNER:     DO J = 1, 10000
              IF(A(J) == 0.0) EXIT OUTER
50            C(I,J) = C(I,J) + A(I) * B(J)
           END DO INNER
        END DO OUTER
        CALL WRITEDATA(C)
    END PROGRAM
```

数组赋值

因为Fortran 90的主要目的之一是提供对向量和并行硬件的支持，很自然需要Fortran 90包含向量和并行操作。实现的一种方法是将向量和数组作为赋值语句中的聚合处理。如果X和Y是两个同维数组，则

```
X = Y
```

一个元素接一个元素地将Y复制到X中，此赋值等价于语句序列

```
X(1) = Y(1)
X(2) = Y(2)
...
X(N) = Y(N)
```

736

使用下面的约定标量值可以与向量值混用，即在实行运算前将标量扩展为相应维数的向量。这样，

```
X = X + 5.0
```

将常数5.0加到数组X的每个元素上。

在Fortran 90中数组赋值是看作为同时执行的；即必须将赋值处理成使得所有输入操作数的取值是在任何存入输出值之前。例如，考虑

```
X = X/X(2)
```

即使X(2)的值被该语句改变，也要始终使用X(2)的原始值，这样结果如同

```
T = X(2)
X(1) = X(1)/T
X(2) = X(2)/T
...
X(N) = X(N)/T
```

此语义与1.3节介绍的向量硬件语义相匹配，只是向量长度是无界的。

三元组记法

数组片断，包括单独的行和列，可以用*三元组记法*来指定。在Fortran 90中，用冒号分开的三个整型表达式的三元组可以用在任何数组赋值的下标中。三元组的成份是

(initial:final:stride)

其中整型表达式initial指明向量迭代中的第一个下标，final指明向量迭代中的最后一个下标；stride指明下标增量。在大多数常见用法中，最后一个冒号和跨距（stride）是省略的，此时增量默认为1。

三元组记法可以用例子做最好的解释。例如，如果A和B是100 × 100矩阵，则

```
A(1:100,I) = B(J,1:100)
```

737

将B的第J行赋给A的第I列。此时由于是赋值一整行或整列，我们可以省略初始和终止下标表达式，这时默认为声明的界。这样

```
A(:,I) = B(J,:)
```

与前面的例子有相同的效果。当向量运算的迭代区域小于整个行或列时三元组记法的实际价值就显现出来了。下面的赋值可用来将B的第J行的前M个元素赋给A的第I列的前M个元素。

```
A(1:M,I) = B(J,1:M)
```

此语句具有下列赋值语句的作用:

```
A(1,I) = B(J,1)
A(2,I) = B(J,2)
...
A(M,I) = B(J,M)
```

其中M的值可以比它出现的那一维的数组的实际上界小得多。

出现第三个成分时，指明在该下标位置中索引向量的“跨距”(stride)。例如，可以用下述语句将B的第J行的前M个元素赋给A的第I列在奇数下标位置上的前M个元素:

```
A(1:M*2-1:2,I) = B(J,1:M)
```

在处理含有移位片断运算时三元组记法也是有用的。赋值语句

```
A(I,1:M) = B(1:M,J) + C(I,3:M+2)
```

有下述语句的作用:

```
A(I,1) = B(1,J) + C(I,3)
A(I,2) = B(2,J) + C(I,4)
...
A(I,M) = B(M,J) + C(I,M+2)
```

738

条件赋值

Fortran WHERE语句允许一个数组赋值受一个条件掩码数组控制。例如，

```
WHERE (A<0.0) A = A + B
```

指明构成A和B的向量和，但是回存到A的元素只是A中相应位置原始值小于零的元素。此语句的语义要求它的行为就像仅对相应控制条件位置为真的成分才进行计算。

语句的特例如

```
WHERE (A/=0.0) B = B/A
```

此处的语义要求作为右边计算结果产生的除法检查不影响程序行为；代码必须向用户隐蔽此种错误。换句话说，忽略控制向量为假的位置上右端计算结果可能出现的错误的副作用。

多维数组赋值

Fortran 90允许整个矩阵甚至大维数组在一个语句中赋值。例如，

```
A(1:N,1:M) = A(1:N,1:M)/X(1:N,1:M)
```

A的1到N行和1到M列的每个元素除以X的相应元素并把结果回存到A。按照Fortran 90标准，像这样的数组赋值是合法的，每个数组在每一维中的子表达式必须与左端在排列（维数）和长度（大小）匹配。上面例子的各维从左到右总是匹配的，所以

```
A(1:N,1:M) = A(1:N,1:M)/X(1:M,1:N)
```

是非法的，尽管通过转置X可以使各维匹配。这就要求显式给出转置运算:

```
A(1:N,1:M) = A(1:N,1:M)/TRANSPOSE(X(1:M,1:N))
```

标量可以在数组赋值中自由使用，因为意图总是清楚的。事实上，每个标量自动扩展到一个排列和长度正确的数组，其中每个元素等于原始的标量。这样

739

```
A(1:N,1:M) = A(1:N,1:M)/X(1, 1)
```

A的每个元素除以X(1,1)。

有一些实例要求写出这样的表达式，其中不同的成分数组有不同的排列。例如，假设我们要求矩阵A的每一行被向量X除。则我们需要使用SPREAD内部函数去扩展向量为两维:

```
A(1:N,1:M) = A(1:N,1:M)/SPREAD(X(1:M),1,N)
```

内部函数SPREAD复制N个向量X(1:M)产生一个适当大小的矩阵，矩阵的每行是原始X的拷贝。如果需要复制到列，则SPREAD的第二个参数是2。

分散运算和聚集运算

Fortran 90允许在下标位置中使用索引数组。例如，

```
A(1:N) = A(1:N) + B(IX(1:N))
```

取向量IX并用它的元素去选择用在向量表达式中使用的B的元素。

FORALL 语句

虽然在最后一刻将它从Fortran 90标准中删除了，Fortran 95标准提供了一种称为FORALL的特殊的向量运算。因为这对于表示不寻常的向量概念是一个有用的结构，我们将它包括在这里。FORALL语句有两种形式。第一种形式在一行上包含单个语句:

```
FORALL (I = 1,N) A(I) = B(I) + C(I)
```

第二种形式看上去像一个包含有一个或多个Fortran 90语句的循环。例如，

```
FORALL (I = 1,N)
   A(I) = B(I) + C(I)
END FORALL
```

FORALL语句解释为一个向量赋值语句，FORALL语句中的索引指出向量迭代的维。这样，
740 上面的FORALL例子与下述简单数组赋值有相同的含义:

```
A(1: N) = B(1;N) + C(1:N)
```

因为FORALL语句有相同的含义，这意味着它的行为必须在存入任何值之前从存储器中取出右边使用的所有量的值。所以

```
FORALL (I = 1,N) A(I) = B(I) / A(2)
```

对循环的每次迭代使用A(2)的老值。

迄今为止的例子没有提供不利用带三元组记法的数组赋值的功能。但是FORALL语句可以写出某些向量语句，它不能用简单的三元组记法表示。例如，假设我们想要将向量B赋给矩阵A的对角线元素。它可用下面FORALL语句完成:

```
FORALL (I = 1,N) A(I,I) = B(I)
```

虽然写一个复杂的条件赋值语句可能得到相同的效果，但是那种语句的性能将不会很好。因此，对于写有技巧性的向量赋值，FORALL语句是有用的方法，对向量化编译器的输出也是有用的。

最后一个例子显示出FORALL语句可以如何写成另外的结构，它不用FORALL语句是很难应付的。下面的循环是矩阵A的行乘矩阵B的列:

```
FORALL (I = 1,N) A(I,1:N) = A(I,1:N)*B(1:N,I)
```

若不用FORALL表示，你就需要写出

```
A(1:N,1:N) = A(1:N,1:N) * TRANSPOSE(B(1:N,1:N))
```

它足够清楚，但是如果Fortran 90编译器不能识别转置能够用索引交换来替代，就可能产生不必要的数据移动。

库函数

数学库函数，如SQRT和SIN，以元素为基础扩展到向量和数组。另外，提供新的内部函数，如矩阵内积（DOTPRODUCT）和转置（TRANSPOSE）。特别函数SEQ(1,N)返回从1到N的索引向量。 741
也提供归约函数（类似在APL中）。例如，SUM应用于一个向量上，返回向量中所有元素的和，而PRODUCT返回这些元素值的乘积。

Fortran 90还包括一些有用的数组运算。循环移位CSHIFT可用来实行数组任何维中的循环移位，而截尾移位EOSHIFT移位并截尾数组，用指定的边界值填充空位。最后MAXLOC和MINLOC返回值分别是一个数组的最大值和最小值的位置，即按大小等于自变量数组的一维数组形式中的元素顺序。

Fortran 90和95库函数的完整列表可以在ISO Fortran标准文档中找到[157，158]。

补充读物

Fortran 90和95的补充读物，我们推荐如下：

（1）正式标准，ISO 1539:1991，*Fortran standard* [157]，是Fortran 90特性精确定义的最好来源。

（2）正式Fortran 95标准，ISO/IEC 1539-1:1997，*Information Technology-Programming Languages-Fortran* [158]，具有与Fortran 95同样的作用。

（3）Adams et al.，*Fortran Top 90* [3]，是专门讨论新语言特性的汇集。就此而言，这是一本介绍对Fortran 77改变的优秀导引。

（4）Adams et al.，*Fortran 90 Handbook* [4]与*Fortran 95 Handbook* [5]，提供对整个语言的详尽综述。 742

参考文献

[1] W. Abu-Sufah. Improving the performance of virtual memory computers. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[2] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12): 685–690, December 1974.

[3] J. C. Adams, W. S. Brainerd, J. T. Martin, and B. T. Smith. *Fortran Top 90*. Unicomp, San Bernadino, CA, 1994.

[4] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, 1992.

[5] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 95 Handbook*. The MIT Press, Cambridge, MA, 1997.

[6] V. Adve, A. Carle, E. Granston, S. Hiranandani, C. Koelbel, J. Mellor-Crummey, C. Tseng, and S. K. Warren. The D System: Support for data-parallel programming. Technical Report CRPC-TR94-378, Rice University, Center for Research on Parallel Computation, January 1994.

[7] V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Lanaguage Design and Implementation*, June 1998.

[8] V. Adve and J. Mellor-Crummey. Advanced code generation for High Performance Fortran. In *Languages, Compilation Techniques, and Run-Time Systems for Scalable Parallel Systems*, Springer-Verlag, New York (to appear).

[9] G. Aggarwal and D. Gajski. Exploring DCT Implementations. Technical Report ICS-TR-98-10, University of California, Irvine, March 1988.

[10] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms*. Exercise 2.12. Addison-Wesley, Reading, MA, 1974.

[11] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. In *Seventh Annual ACM Symposium on Theory of Computing*, pages 207–217, May 1975.

[12] A.V. Aho, R. Sethi, and J. D. Ullman. Code optimization and finite Church-Rosser systems. In R. Rustin, editor, *Design and Optimization of Compilers*, Prentice Hall, Upper Saddle River, NJ, 1972.

[13] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress 1974*, pages 398–402, North Holland, Amsterdam, 1974.

[14] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. In *Proceedings of the 1968 ACM International Conference on Supercomputing*, pages 207–215, July 1988.

[15] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[16] J. R. Allen. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. thesis, Department of Mathematical Sciences, Rice University, May, 1983.

[17] J. R. Allen. Unifying vectorization, parallelization, and optimization: The Ardent compiler. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Third International Conference on Supercomputing*, 1988.

[18] J. R. Allen, D. Bäumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing*, IEEE Computer Society Press, Silver Spring, MD, August 1986.

[19] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[20] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[21] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[22] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[23] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Languages*, January 1983.

[24] R. Allen. Unifying vectorization, parallelization, and optimization: The Ardent compiler. In *Third International Conference on Supercomputing*, 2:176–185, 1988.

[25] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, January 1987.

[26] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 241–249, June 1988.

[27] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, 1988.

[28] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[29] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*, second edition. SIAM Publications, Philadelphia, 1995.

[30] J. Backus. The history of FORTRAN I, II, and III. *ACM SIGPLAN Notices* 13(8):165–180, August 1978.

[31] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, November 1989.

[32] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1976. Report No. 76-837.

[33] U. Banerjee. Speedup of ordinary programs. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

[34] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, 1988.

[35] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

[36] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Computing.* The MIT Press, Cambridge, MA, August 1990.

[37] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, January 1979.

[38] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[39] W. Baxter and H. R. Bauer III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989.

[40] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[41] K. V. Besaw. Advanced techniques for vectorizing dusty decks. In *Proceedings of the Second International Conference on Supercomputing*, May 1987.

[42] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[43] T. Brandes. The importance of direct dependences for automatic parallelization. In *Proceedings of the Second International Conference on Supercomputing*, July 1988.

[44] P. Briggs, K. D. Cooper, M. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-148, Department of Computer Science, Rice University, November 1990.

[45] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):275–284, July 1989.

[46] J. Bruno and R. Sethi. Code generation for a one-register machine. In *Journal of the Association for Computing Machinery*, 23(3):502–510, July 1976.

[47] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[48] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. Research Report RC 12702, IBM, Yorktown Heights, NY, September 1987.

[49] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[50] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[51] D. Callahan. Dependence testing in PFC: Weak separability. *Supercomputer Software Newsletter 2*, Department of Computer Science, Rice University, August 1986.

[52] D. Callahan. A global approach to the detection of parallelism, Ph.D. thesis, Department of Computer Science, Rice University, March 1987.

[53] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):47–56, July 1988.

[54] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, SE-16(4): 483–487, April 1990.

[55] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[56] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7):152–161, July 1986.

[57] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing '88*, November 1988.

[58] D. Callahan and M. Kalem. Control dependences. *Supercomputer Software Newsletter 15*, Department of Computer Science, Rice University, October 1987.

[59] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Springer-Verlag, New York, June 1987.

[60] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 3:151–169, October 1988.

[61] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices*, 26(4): 40–52, April 1991.

[62] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[63] A. Carle, K. D. Cooper, R.T. Hood, K. Kennedy, L. Torczon, and S. K. Warren. A practical environment for Fortran programming. *IEEE Computer*, 20(11): 75–89, November 1987.

[64] S. Carr. Memory hierarchy management. Ph.D. thesis, Department of Computer Science, Rice University, September 1992.

[65] S. Carr, D. Callahan, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.

[66] S. Carr and K. Kennedy. Compiler Blockability of Numerical Algorithms. In *Proceedings of Supercomputing '92*, November 1992.

[67] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice & Experience*, 24(1), January 1994.

[68] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of SIGPLAN 82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6):98–105, June 1982.

[69] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[70] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):146–160, July 1989.

[71] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed-Memory Machines*, North-Holland, Amsterdam, 1992.

[72] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[73] D. Chavarria-Miranda, J. Mellor-Crummey, and T. Sarang. Data-parallel compiler support for multipartitioning. In *Proceedings of the European Conference on Parallel Computing* (Euro-Par), August 2001.

[74] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, January 1993.

[75] F. Chow. A portable machine-independent global optimizer—design and measurements. TR 83-254, Department of Electrical Engineering and Computer Science, Stanford University, December 1983.

[76] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):222–232, June 1984.

[77] W. Cohagan. Vector optimization for the ASC. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems*, March 1973.

[78] K. D. Cooper. Interprocedural data flow analysis in a programming environment. Ph.D. thesis, Computer Science Department, Rice University, April 1983.

[79] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[80] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 96–105, April 1992.

[81] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), 1993.

[82] K. D. Cooper, M. W. Hall, K. Kennedy, and L. Torczon. Interprocedural analysis and optimization. To appear in *Communications in Pure and Applied Mathematics*.

[83] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[84] K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Draft, Computer Science Department, Rice University, November 2000.

[85] K. D. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):247–258, June 1984.

[86] K. D. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information—a correction. *SIGPLAN Notices*, 23(4):35–42, April 1988.

[87] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):57–66, July 1988.

[88] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[89] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In *Proceedings of the SIGPLAN '85 Symposium on Compiler Construction*, June 1985.

[90] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[91] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86*

Symposium on Compiler Construction, SIGPLAN Notices, 21(7):58–67, July 1986.

[92] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, June 1990.

[93] R. Cytron. Compile-Time Scheduling and Optimization for Asynchronous Machines. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1984.

[94] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[95] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference of Parallel Processing*, August 1987.

[96] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages*, June 1989.

[97] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 452–490, October 1991.

[98] E. S. Davidson. Design and control of pipelined function generators. In *Proceedings of the 1971 International IEEE Conference on Systems, Networks, and Computers*, pages 19–21, January 1971.

[99] E. S. Davidson. Scheduling for pipelined processors. In *Proceedings of the 7th Hawaii Conference on Systems Sciences*, pages 58–60, 1974.

[100] E. S. Davidson, D. Landskov, B. D. Schriver, and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, 1981.

[101] J. Davidson and A. Hollar. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[102] H. Dietz. Finding large-grain parallelism in loops with serial control dependences. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[103] C. Ding. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. Ph.D. thesis, Department of Computer Science, Rice University, January 2000.

[104] C. Ding and K. Kennedy. Resource-constrained loop fusion. Technical report, Computer Science Department, Rice University, November 2000.

[105] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium*, April 2001.

[106] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[107] C. Dulong, R. Krishnayer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr.

An overview of the Intel IA-64 compiler. *Intel Technology Journal*, Q4, 1999.

[108] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Fourth International Workshop, Springer-Verlag, Berlin, August 1991.

[109] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. Ph.D. thesis, Department of Computer Science, Yale University, 1985.

[110] J. Ferrante and M. Mace. On linearizing parallel code. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, January 1985.

[111] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *Proceedings of the Second International Conference on Supercomputing*, July 1988.

[112] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[113] J. A. Fisher. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources. Ph.D. thesis, Department of Computer Science, New York University, 1979.

[114] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[115] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[116] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, 1992.

[117] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computation*, 5(5):587–616, October 1998.

[118] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[119] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, 1979.

[120] P. A. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined processor. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, July 1986.

[121] M. Girkar and C. Polychronopoulos. Compiling issues for supercomputers. In *Proceedings of Supercomputing '88*, November 1988.

[122] G. Goff. Practical techniques to augment dependence analysis in the presence of symbolic terms. Technical Report TR92-194, Department of Computer Science, Rice University, October 1992.

[123] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In

Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991.

[124] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 228–234, August 1984.

[125] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 32(1):172–202, January 1976.

[126] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2): 133–155, February 1990.

[127] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

[128] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances in Languages and Compilers for Parallel Computing*, The MIT Press, Cambridge, MA, August 1990.

[129] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[130] M. W. Hall. Managing interprocedural optimization. Ph.D. thesis, Department of Computer Science, Rice University, April 1991.

[131] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, and M. S. Lam. Interprocedural analysis for parallelization: Preliminary results. Stanford Computer Systems Laboratory Technical Report, CSL-TR-95-665, March 1995.

[132] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: An interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[133] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), September 1992.

[134] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, November 1991.

[135] M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[136] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. Interprocedural analysis for parallelization: A case study. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[137] R. V. Hanxleden and K. Kennedy. Give-N-Take—a balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[138] D. Harel. A linear-time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[139] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[140] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[141] P. Havlak. Interprocedural symbolic analysis. Ph.D. thesis, Department of Computer Science, Rice University, May 1994.

[142] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[143] M. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis of programs. *SIAM Journal of Computing*, 4: 519–532.

[144] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, 1992.

[145] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, second edition. Morgan Kaufmann, San Francisco, 1996.

[146] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.

[147] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1997.

[148] M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[149] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, November 1991.

[150] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed-Memory Machines*, North-Holland, Amsterdam, 1992.

[151] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, November 1993.

[152] S. Horowitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[153] P. Y. T. Hsu. Highly concurrent scalar processing. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1986.

[154] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Pro-*

ceedings of the Thirteenth Annual International Symposium on Computer Architecture, pages 386–395, 1986.

[155] IEEE. *Standard VHDL Language Reference Manual*. 1988.

[156] T. Iitsuka. Flow-sensitive interprocedural analysis method for parallelization. In *IFIP TC10/WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993.

[157] International Standards Organization. *ISO 1539: 1991, Fortran Standard*. 1991.

[158] International Standards Organization. *ISO/IEC 1539-1: 1997, Information Technology–Programming Languages–Fortran*. 1997.

[159] F. Irigoin. Interprocedural analyses for programming environments. In *NSF-CNRS Workshop on Environments and Tools for Parallel Scientific Programming*, September 1992.

[160] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, June 1991.

[161] S. C. Johnson. A portable compiler: Theory and practice. In *SIGPLAN '78 Symposium on Principles of Programming Languages*, pages 97–104, 1978.

[162] R. L. Johnston. The dynamic incremental compiler of APL\3000. In *Proceedings of the APL '79 Conference*, pages 82–87, June 1979.

[163] D.-C. Ju, C.-L. Wu, and P. Carini. The classification, fusion, and parallelization of array language primitives. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1113–1120, October 1994.

[164] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 32(1):158–171, January 1976.

[165] J. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3): 305–318, 1977.

[166] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical report, Department of Computer Science, University of Maryland, College Park, April 1996.

[167] K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics*, Gordon and Breach, Section A, 3:117–130, 1972.

[168] K. Kennedy. Automatic translation of Fortran programs to vector form. Technical Report 476-029-4, Department of Mathematical Sciences, Rice University, October 1980.

[169] K. Kennedy. A survey of data-flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 1–51. Prentice Hall, Upper Saddle River, NJ, 1981.

[170] K. Kennedy. Triangular Banerjee inequality. *Supercomputer Software Newsletter 8*, Department of Computer Science, Rice University, October 1986.

[171] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, May 2000.

[172] K. Kennedy. Fast greedy weighted fusion. *International Journal of Parallel Processing*, 29:5, October 2001.

[173] K. Kennedy, C. Koelbel, and M. Paleczny. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium of the Frontiers of Massively Parallel Computation*, February 1995.

[174] K. Kennedy and K. McKinley. Loop distribution with arbitrary control flow. *Proceedings: Supercomputing '90*, pages 407–416, November 1990.

[175] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, July 1992.

[176] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Number 768, pages 301–320, Springer-Verlag, Berlin, 1993.

[177] K. Kennedy and K. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report CRPC-TR94646, Center for Research on Parallel Computation, Rice University, 1994.

[178] K. Kennedy, J. Mellor-Crummey, and G. Roth. Optimizing Fortran 90 shift operations on distributed-memory multicomputers. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC '95)*, August 1995.

[179] K. Kennedy and G. Roth. Context optimization for SIMD execution. In *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[180] K. Kennedy and G. Roth. Dependence analysis of Fortran 90 array syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, August 1996.

[181] K. Kennedy and A. Sethi. Resource-based communication placement analysis. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, New York, August 1996.

[182] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on the Principles of Programming Languages*, pages 194–206, October 1973.

[183] D. Klappholz and X. Kong. Extending the Banerjee-Wolfe test to handle execution conditions. Technical Report 9101, Department of EE/CS, Stevens Institute of Technology, 1991.

[184] D. Klappholz, K. Psarris, and X. Kong. On the perfect accuracy of an approximate subscript analysis test. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, June 1990.

[185] D. E. Knuth. *Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.

[186] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[187] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[188] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[189] D. Kuck. *The Structure of Computers and Computations*, Volume 1. John Wiley and Sons, New York, 1978.

[190] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, January 1981.

[191] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[192] R. Kuhn. Optimization and interconnection complexity for parallel processors, single-stage networks, and decision trees. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1980.

[193] M. Lam. A systolic array optimizing compiler. Ph.D. thesis, Carnegie Mellon University, 1987.

[194] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[195] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[196] L. Lamport. The coordinate method for the parallel execution of iterative DO loops. Technical Report CA-7608-0221, SRI, Menlo Park, CA, August 1976, revised October 1981.

[197] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 27(7):235–248, July 1992.

[198] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[199] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, July 1992.

[200] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, July 1988.

[201] Z. Li and P. Yew. Some results on exact data dependence analysis. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, Cambridge, MA, 1990.

[202] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, June 1989.

[203] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proceedings of the Second International Conference on Supercomputing*, July 1988.

[204] D. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 17(2):121–145, January 1977.

[205] L. Lu and M. Chen. Subdomain dependence test for massive parallelism. In *Proceedings of Supercomputing '90*, November 1990.

[206] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.

[207] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[208] E. J. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal* 35(5):1417–1444, November 1956.

[209] K. S. McKinley. Automatic and interactive parallelization. Ph.D. thesis, Department of Computer Science, Rice University, April 1992.

[210] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4), July 1996.

[211] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[212] R. Mercer. The Convex Fortran 5.0 compiler. In *Third International Conference on Supercomputing*, 2:164–175, 1988.

[213] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 1(3), December 1992.

[214] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computer Systems*, C-36(12):1485–1495, December 1987.

[215] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[216] T. C. Mowry. Tolerating latency through software-controlled data prefetching. Ph.D. thesis, Department of Electrical Engineering, Stanford University, March 1994.

[217] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[218] S. S. Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[219] Y. Muraoka. Parallelism exposure and exploitation in programs. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[220] E. Myers. A precise inter-procedural data flow algorithm. In *Conference*

Record of the Eighth Annual Symposium on Principles of Programming Languages, January 1981.

[221] D. A. Padua. Multiprocessors: Discussion of some theoretical and practical problems. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1979. Report 79-990.

[222] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[223] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems—On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, 1999.

[224] K. Pieper. Parallelizing compilers: Implementation and effectiveness. Ph.D. thesis, Stanford Computer Systems Laboratory, June 1993.

[225] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell, MA, 1988.

[226] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, 1987.

[227] A. Porterfield. Software methods for improving cache performance on supercomputer applications. Ph.D. thesis, Department of Computer Science, Rice University, 1989.

[228] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[229] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[230] W. V. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59(8):521–531, October 1952.

[231] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7: 9–50, 1993.

[232] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the Fourteenth Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[233] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures. In *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, pages 131–139, April 1982.

[234] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 321(3), August 1989.

[235] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[236] B. K. Rosen, M. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming*, 1988.

[237] C. Rosene. Incremental dependence analysis. Ph.D. thesis, Department of Computer Science, Rice University, May 1990.

[238] G. Roth. Optimizing Fortran 90D/HPF for distributed-memory computers. Ph.D. thesis, Department of Computer Science, Rice University, 1997.

[239] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the PEPM '91 Symposium on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Notices*, 26(9):321–333, September 1991.

[240] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, 1979.

[241] V. Sarkar. Optimized execution of Fortran 90 array language on symmetric shared-memory multiprocessors. In *Proceedings of LCPC '98*, pages 131–147, Springer-Verlag, Heidelberg, 1999.

[242] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, June 1991.

[243] R. G. Scarborough and H. G. Kolsky. A vectorizing FORTRAN compiler. *IBM Journal of Research and Development*, March 1986.

[244] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.

[245] R. Sethi. Testing for the Church-Rosser property. *Journal of the ACM*, 21(4), October 1974.

[246] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice Hall, Upper Saddle River, NJ, 1981.

[247] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[248] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):164–174, July 1988.

[249] O. Shivers. Control-flow analysis of higher-order languages. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[250] O. Shivers. The semantics of scheme control flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Notices*, 26(9):190–198, September 1991.

[251] Silicon Graphics, Inc. Performance tuning optimization for Origin2000 and Onyx. Technical report. *techpubs.sgi.com/library/manuals/3000/007-3511-001/html*.

[252] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the IFIP Congress 1971*, pages 376–381, North Holland, Amsterdam, 1971.

[253] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, pages 62–73, January 1995.

[254] Standards Performance Evaluation Corporation, SPEC release 1.2, September 1990.

[255] B. Su and J. Wang. GUPR*: A new global software pipelining algorithm. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 212–216, November 1991.

[256] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing* 1(2):146–160, 1972.

[257] S. Tjiang. *Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers*. Stanford University, 1993.

[258] R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[259] R. A. Towle. Control and data dependence for program transformation. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1976.

[260] R. Triolet. Interprocedural analysis for program restructuring with Parafrase. CSRD Report No. 538, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1985.

[261] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7):176–185, July 1986.

[262] C.-W. Tseng. An optimizing Fortran D compiler for MIMD distributed-memory machines. Ph.D. thesis, Department of Computer Science, Rice University, January 1993.

[263] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[264] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, July 1973.

[265] *Verilog Hardware Description Language Reference Manual (LRM)*. Open Verilog International, Los Gatos, CA, 1995.

[266] D. Wallace. Dependence of multi-dimensional array references. In *Proceedings of the Second International Conference on Supercomputing*, July 1988.

[267] K. Walter. Recursion analysis for compiler optimization. *Communications of the ACM*, 19(9):514–516, 1976.

[268] H. S. Warren. Instruction scheduling for the IBM RISC System/6000. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.

[269] J. Warren. A hierarchical basis for program transformations. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, January 1984.

[270] D. Wedel. FORTRAN for the Texas Instruments ASC system. *SIGPLAN Notices*, 10(3):119–132, March 1975.

[271] M. Wegman. General and efficient methods for global code improvement. Ph.D. thesis, University of California, Berkeley, December 1981.

[272] M. Wegman and K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 291–299, January 1985.

[273] M. Wegman and K. Zadeck. Constant propagation with conditional branches.Technical Report CS-89-36, Department of Computer Science, Brown University, May 1989.

[274] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Symposium on Principles of Programming Languages*, January 1980.

[275] M. E. Wolf. Improving locality and parallelism in nested loops. Ph.D. thesis, Department of Computer Science, Stanford University, August 1992.

[276] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[277] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[278] M. J. Wolfe. Techniques for improving the inherent parallelism in programs. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1978.

[279] M. Wolfe. Optimizing supercompilers for supercomputers. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1982.

[280] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[281] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15(4):279–293, August 1986.

[282] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[283] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[284] M. Wolfe. Beyond induction variables. In *Proceedings of The ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, June 1992.

[285] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[286] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[287] Y. Zhao and K. Kennedy. Scalarizing Fortran 90 array syntax. Technical Report TR01-373, Department of Computer Science, Rice University, March 2001.

[288] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[289] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. *Vienna Fortran—A Language Specification, Version 1.1*. Interim Report 21, ICASE, Hampton, VA, March 1992.

索引

索引中的页码为英文原书页码，与书中边栏页码一致。

B

C

D

E

F

G

H

J

K

L

M

N

O

S

T

U

W, X, Y

Z