# Building an Optimizing Compiler 中文版

潘立丰陈畅

2023 年 10 月 08 日

# Contents

Bob Morgan 著

潘立丰陈畅译

CHAPTER 1

# 前言

自从 1940 年代晚期和 1950 年代早期数字计算机出现以来，构建编译器一直是一项具有挑战的活动。在那个时候，实现从数学家熟悉的算式到计算机指令的自动翻译是一项困难的任务。开发者需要解决如何将算术表达式翻译为指令，如何将数据存储在内存中，如何挑选指令来构建过程和函数。在 1950 年代晚期和 1960 年代早期，大多数计算机专家可以编写出简单的编译器，某种程度上实现了上述过程的自动化。事实上，翻译器采用”小型语言“是 UNIX 社区的基本理念。

从开始以来，人们一直要求翻译器生成高效的代码：翻译器必须充分利用计算机的性能。本来，有这种要求的原因是计算机内存小并且执行速率低。计算机硬件一代一代地更新，每一代都应用了新的架构理念。在每个阶段，也要求编译器不断改进，以更有效地利用这些新的机器。奇怪的是，权威人士总是预测，廉价而低效的翻译器将会胜任工作。他们争辩道，随着机器越来越快，内存越来越大，人们不再需要优化的编译器。不幸的是，人们购买了更大更快的机器，就想要处理更大或更复杂的问题，因此我们仍然需要优化的编译器。事实上，我们更加需要这样的编译器，因为新架构机器的性能对所生成代码的质量是敏感的。相比 1970 年代和 1980 年代的复杂指令集计算（CISC）机器，稍微改动指令的序列或者指令的选择，对这些机器性能的影响会大得多。

精简指令集计算（RISC）架构让计算机架构和编译器性能之间的相互影响变得合理。编译器和计算机架构有着一种相互的依赖关系，双方共同努力，让应用程序运行得更快。为了这个目的，通过暴露一些硬件操作的细节，简化了硬件，例如简单的 load-store 指令集和指令调度。这要求编译器处理这些新暴露的细节，生成执行更快的代码，比在 CISC 处理器上可能生成的代码都要快。

本书描述编译器中的优化和代码生成 phase 的设计。有很多书描述编程语言分析。它们强调词法分析、解析、语义分析等流程。有几本书描述针对向量和并行处理器的编译过程。本书描述针对单一超标量 RISC 处理器的高效程序的编译，包括算法的次序和结构和高效的数据结构。

本书是一份高层级设计文档。这有两个理由。起初，我试图写一本介绍所有可能方法的书，这样读者可以自

已选择使用哪些方法。这太庞大了，预计书的页数是几千页——太大了，不现实。在优化的编译器中有大量的不同算法和结构。选择是相互关联的，因此百科全书式的优化编译器方案不能解决一些最困难的问题。

其次，对大型软件流程，我想鼓励这种形式的设计。政府用一种三级的文档系统描述软件项目：A 层级文档是概述文档，描述项目的整体情况，列出各个部分。B 层级文档描述每个组件的操作，细节足够详细，让读者能够理解每个组件做什么，它怎么做的，而 C 层级文档是低层级文档，描述项目的每个细节。

作为一个开发者，我发现这种结构负担沉重，因为它退化为一种官僚主义的设施，用了大量纸张，表述极少内容。但是，其基本思想是合理的。本书将描述编译器的优化和代码生成组件，细节足够详细，让读者能够实现这些组件，如果他或者她认为合适。我会为每个组件描述一种方法，可以详细地考察组件之间的相互作用，这样所有的设计和实现都是清楚的。

每章会有一个描述其它可能的实现技术的小节。这个小节将包括目录信息，这样有兴趣的读者可以找到这些技术。

## 1.1 选择编译器技术的哲学

在书的开头，我想说一说我的设计哲学。当我刚开始编写编译器时（大约 1964 年），我注意到很多研究和开发工作是由文献描述的。虽然这些项目是基于不同的假设和需求的，但是有了文献之后，仿效和使用早前方法变得容易了，不必重新发明它们。因此在设计的时候，我会查阅文献和其它的实现，选择符合我的需求的技术。我的贡献是，选择技术，实现技术，让它适合其它组件，还有我所观察到的小的改进。

必须引入一条工程经验法则。决定使用已经发表的最新的技术是容易的。这种策略是危险的。任何优化或代码生成技术仅仅被使用了一段时间就选择它们会带来副作用。因此，我尝试避开那些不曾在原型或产品编译器中实现至少两次的技术。当我确定一项技术是合理的时候，我会破例一到两次，但是仅此而已。

在写作本书的过程中，我的观点在进化。它从记录已知的知识开始。我利用存在的技术设计并构建了几个编译器。随着书的文字越写越多，我学到很多集成这些算法的方法。它从串联不相关的想法开始，融合为更加集成的整体。它从简单描述工程规划开始，现在包含了一些崭新的想法。这大概就是任何智力工作的过程；但是，我发现它令人耳目一新，令人精神振奋。

## 1.2 如何使用本书

本书是为三个用途而设计的。第一个用途是描述优化编译器的结构，让读者可以实现它或它的变体（编译器编写者总是修改设计）。书的结构反应了这个用途。起始章描述了各个编译 phase 和它们之间的相互作用；后续章描述了每个编译 phase 涉及的算法。

本书也可以用作关于编译器优化技术的课本。它采用了一个样例，通过这个样例描述每个编译流程。学生分析不同的样例，而不是求解小的家庭作业问题。

实际上，本书的最大用途是满足求知欲。如果你和我一样，你捡起书本是因为你想学到某种知识。我希望你喜爱这本书，找到你想要的内容。开卷有益。

CHAPTER 2

# 第 1 章概述

什么是优化的编译器？我们为什么需要它们？它们从何而来？这些是本章要讨论的问题，还有如何使用这本书。在正式向读者呈现详细的编译器设计之前，本章作为导引，先说一下优化编译器开发的非正式历史，并给出一个可运行的例子，来说明编译器中的技术。本书的其余部分会一直使用这个例子。

## 2.1 1.1 什么是优化的编译器

一个程序员如何让他的应用程序获得期望的性能呢？起初程序员按照简单易懂的方式编写程序，如此，程序的正确性可以得到测试或证明。然后，通过剖析和测试，找出程序消耗时间和内存资源的热点，修改程序以改进其对资源的使用。在程序员作了所有合理的修改之后，进一步的性能改进只能来自编译器，就是如何最优地将编程语言翻译为目标机器的指令。

优化的编译器的目标，就是高效地使用目标机器的全部资源。编译器将源程序翻译为机器指令，这些指令使用各种各样的计算单元。理想的翻译能够得到这样的目标程序，它在指令执行的每个时钟周期让每个计算单元保持活跃，做着有用（非冗余）的工作。

当然，理想化的翻译并不总是可能的。源程序的计算需求集合可能是不平衡的。它可能做很多整数计算，很少浮点计算，或者反过来，或者算术计算很少，而载入和存储操作很多。针对这样的情况，编译器必须尽可能有效地利用过载的计算单元。

编译器必须想办法补偿不平衡的计算机系统。在理想状态下，处理器的速度匹配内存系统的速度，它们又都匹配输入输出（IO）系统的速度。在现代精简指令集计算机（RISC）系统上，这是不成立的：处理器比内存系统快得多。为了充分利用处理器的能力，编译器必须让生成的代码减少访问内存系统，通过将数值保持在寄存器中，或者通过组织代码使所需的数据驻留在高速缓存中。

一个附加的麻烦是获取指令。绝大部分内存引用碎片是对指令的引用。我们希望指令驻留在某个高速缓存中；然而，情况并不总是这样。当指令不容易驻留高速缓存的时候，编译器应该生成尽可能少的指令。当指令容易驻留高速缓存并且数据引用很多的时候，编译器允许自由地增加指令以减少对数据的等待。达到平衡不是一件容易的事。

总之，优化的编译器试图让应用程序在执行过程中尽可能有效地利用处理器和内存的全部资源。编译器必须转换程序以平衡对计算单元和内存的使用。它必须恰当地选择指令，使用尽可能少的指令，同时达到这样的平衡。当然，完全做到是不可能的，但是编译器必须尽其所能。

## 2.2 1.2 有所侧重的优化编译器的历史

编译器具有非凡的历史，但是常常被人忽视。重大的开发始于 19 世纪 50 年代。每隔一段时间，权威人士都会断言，编译器开发已经没什么可做的了。他们总是被证明错了。如今，随着处理器的速度越来越快，编译器技术需要得到大力的开发。在此，我列出若干个激励并影响我最深的编译器开发小组。对于在这个领域作出了重要贡献的其他小组，我无意轻视。

尽管有早期为解析和编译所作的工作，第一个重要的编译器是针对 IBM 704/709/7090/7094 的 Fortran 编译器（Backus）。这个项目标志着编译器开发的分水岭。为了得到程序员的认可，它所生成的代码必须类似于机器语言程序员手工编写的代码，因此这是一个高度优化的编译器。它必须编译一个完整的语言，尽管语言的设计对开发者是开放的。当时这个项目所需的技术并不存在，他们必须自己开发。这个团队非常成功，他们创造了近十年间最优秀的编译器之一。这个项目提出了编译器 pass 或者 phase 的概念。

之后，还是在 IBM，有个团队为 IBM 360/370 系列计算机开发了 Fortran/Level H 编译器。这些编译器同样是高度优化的编译器。他们的四元（quadruple）概念，和本书中介绍的设计所用到的抽象汇编语言的概念是类似的。Scarborough 和 Kolsky（1980）对编译器所作的后续改进，让此类编译器成为又一个十年间最优秀的编译器之一。

从 19 世纪 60 年代末期到 70 年代结束，有两个研究小组一直在开发上述编译器以之为基础的技术，也在开发新的技术。其中一个小组在 IBM，由 Fran Allen 领衔，另一个小组在纽约大学（NYU），由 Jack Schwartz 领衔。这两个小组首先提出了可达定义和位向量（bit-vector）的概念，用于描述程序转换条件。他们的大部分工作发表在文献上；如果你能得到 SETL 简报（NYU 1973）或者 SETL 项目相关报道的副本，就能了解其详情。

其他小组也在研究优化技术。William Wulf 定义了一种称为 Bliss 的语言（Wulf et al. 1975）。这是一种结构化的编程语言，在卡内基梅隆大学（MCU），Wulf 和他的团队为这种语言开发编译器优化技术。其中的一些技术只适用于结构化的语言，而另一些是通用的，适用于任何结构的语言。这个项目升级为具有产品质量级别的编译器的编译器（PQCC）项目，开发元编译器技术，以构造优化的编译器（Leverett, et al. 1979）。这些论文是一些最丰富又最少被采用的编译器开发技术的来源。

其他商业公司也在研究编译器技术。COMPASS 开发了基于 p-graph 技术的编译器（Karr 1975）。这种技术对于编译器优化比可达定义更优越，因为数据结构易于更新；然而，p-graph 的初始计算比可达定义慢得多。Reif（Reif 和 Lewis 1978）和 IBM Yorktown Heights 的后续开发者（Cytron et al. 1989）将 p-graph 转换成流图的静态单赋值形式（Static Single Assignment Form），这是当今编译器开发所采用的一种流图结构。

Ken Kennedy，纽约大学（NYU）的一个学生，在莱斯（Rice）大学创立了一个编译器小组，以继续他在编译器优化方面的工作。起初，这个小组专攻向量化技术。向量化需要好的标量优化，所以他们也一直研究标量优

化。在莱斯大学，由 Keith Cooper 领衔的小组作了分析多个过程（过程间分析）的部分最有效的工作（1988，1989）。本书大量采用了由 Cooper 领衔的大规模标量编译器项目和小组所设计的流图结构。

在 19 世纪 70 年代末期和 80 年代早期，随着超级计算机和 RISC 处理器的问世，新的编译器技术亟待开发。特别地，指令被管线化，数值在需要的时候是准备好的。指令必须重新排列，使得第一条指令的结果回来之前，启动一些别的指令。编译器编写者最初为如 Cray-1 这样的机器开发了这些技术。这方面工作的一个例子，是 Richard Sites 关于重排 Cray-1 汇编语言的论文（1978）。后来，IBM 801 项目（Auslander 和 Hopkins 1982）和卡内基梅隆大学的 Gross（1983）将这些技术应用于 RISC 处理器。这个领域的其它工作包括，描述 RS6000 编译器的论文（Golumbic 1990 和 Warren 1990），和威斯康辛（Wisconsin）大学关于指令调度的研究。

在 19 世纪 70 年代和 80 年代早期，寄存器分配是一个困难的问题：编译器应该如何将计算得到的数值指派给少量的物理寄存器，以最小化在寄存器和内存之间搬运数据的次数？Chaitin（1981，1982）将这个问题转换为图着色（graph-coloring）问题，并且开发出图着色启发式方法，这种方法在处理具有复杂流图的程序时表现良好。卡内基梅隆大学的 PQCC 项目开发了一种将它抽象为 bin-packing（装箱）问题的方法，这种方法在处理直线式或结构化过程时表现最好。本书中开发的技术是以上两种技术的综合，同时用到了部分 Laurie Hendron 在 McGill 大学所作的进一步工作。

## 2.3 1.3 所有这些技术给我们带来什么？

考虑以上历史，现代 RISC 处理器需要一个高效的编译器，建造这样的编译器所需的全部技术在大约 1972 年出现了，肯定地说是 1980 年。那么，更近期的研究有何价值呢？那个时候存在的技术可以做到，但是要付出很大的代价。更近期的关于优化编译器的研究带来了更高效更易于实现的技术。有两个例子能说明这个事实。在 19 世纪 60 年代后期和 70 年代早期，Fortran/Level H 编译器是其中一个最高效的优化编译器。它用到一个优化循环的算法，其基础是识别出嵌套的循环。在 19 世纪 70 年代后期，Etienne Morel 开发出一种称为消除部分冗余（Elimination of Partial Redundancies）的技术，执行更有效的代码移动，而不需要计算任何循环的信息（Morel 和 RenVoise 1979）。

类似地，静态单赋值形式的概念让很多转换算法更简单更直观。Killdall（1973）开发的常量传播（Constant propagation）算法令人觉得复杂。后来 Wegman 和 Zadeck（1985）作的建模让这项技术近乎直观。

新的技术让人们更易于建造优化的编译器。这是至关重要的。这些编译器是庞大的程序，大程序会遇到的问题，它都会遇到。如果我们简化了编译器的某个部分，就加快了开发进度，提高了编译速度，减少了编译器中存在的 bug（故障，缺陷）。这造就了更廉价更可靠的产品。

## 2.4 1.4 编译器后端的游戏规则

编译器后端有三个基本职能：生成能够忠实代表源程序本义的代码，有效率地分配机器的资源，尽其所能将程序改写为最有效率的形式。以上每个职能要遵守的一条根本规则是，必须忠实地表达源程序。

不幸的是，曾经编译器编写者认为，有必要让大部分程序保持正确，但不是全部程序。当程序员以不寻常的方式使用一些合法的特性时，编译器可能会生成程序的一个不正确的版本。这损害了优化编译器的名声。

如今人们意识到，编译器的代码生成和优化组件必须准确地体现程序的本义，既要符合源程序，也要符合编

程语言的语言参考手册。这并不意味着，当编译器打开或者关闭优化时，程序会给出完全相同的结果。有些程序以编译器未能识别的方式违背了语言的定义。经典的例子是在变量被赋值以前使用它。当关闭或者打开优化时，这些程序可能得到不同的结果。

幸运的是，标准组织在描述语言标准的时候，越来越明白编译器编写者想知道什么。现在，各个主要的语言标准都以某种方式描述编译器优化的限制。为此，有时候语言的某些方面被保留为未定义或者由实现决定。这样的措辞意味着，当编译器遇到语言的这些方面时，它可能会想怎么做就怎么做。然而，请慎重，对于编译器将怎么处理这些情况，用户社区会时常给出期望，而编译器最好尊重这些期望。

如果源程序在某个段落以一种超出编译器预期的方式使用一种语言特性，编译器会怎么做呢？它必须选择以保守的方式实现那个特性，甚至以程序的运行时性能为代价。即使作出了保守的选择，编译器也可能见机行事。举例来说，它可能以两种不同的方式编译相同的代码段落，并且生成代码去检查采用那个版本的代码是安全的。

## 2.5 1.5 标准测试和设计编译器

编译器编写者去哪里寻找一个优化的编译器所必须包含的改进呢？如何比较某种特定优化的两个变种并选择其中之一？编译器编写者利用目标机器的应用领域信息、源语言的应用领域信息、正确的判断力去选择一系列特定的优化，并且选择如何组织它们。

每个应用领域都有一套对其重要的标准程序。对于商业应用来说，排序和数据库是重要的。对于数值计算应用来说，线性代数和方程求解是重要的。其它程序对于仿真来说是重要的。编译器编写者会调查这些程序，并决定编译器该怎么做才能很好地翻译它们。与此同时，编译器编写者和他的用户会从这些程序中提取样例代码。这些样例代码会成为标准测试，用于检验编译器的质量是否达到要求。

他们还会调查被编译的源语言，决定必须支持的语言特性。在 Fortran 中，优化的编译器需要做强度减弱（strength reduction），因为程序员没有简化乘法运算的机制。在 C 中，强度减弱不太重要（尽管仍然有用）；然而，编译器需要很好地编译小的子函数，准确地计算出尽可能多的关于指针的信息。

有些标准优化是需要被实现的。消除冗余运算，将代码移出循环，这些优化对于命令式语言的优化编译器是必要的。事实上，这是第一原则的一部分，因为大多数应用程序员期望着这些优化。

编译器编写者必须小心谨慎。容易出现这样的事情，设计出来的编译器在编译标准测试程序时表现良好，编译普通程序时令人失望。Whetstone 标准测试包含一个代码 kernel，可以利用三角函数的一致性来优化它。SPEC92 标准测试有一个 kernel，EQNTOT，可以通过对整数指令的机智的向量化来优化它。

编译器编写者是否应该为处理这些反常的标准测试而加入特殊的代码呢？应该，也不应该。在竞争的世界，我们不得不加入特殊的代码，因为竞争对手是这样做的。然而，我们必须认识到，这并没有真正地建造出更好的编译器，除非大量的不同类别的程序证明这个特性是有用的。我们应该总是把标准测试看作关于编程的一般性检验。利用标准测试去找出通用的改进。总之，设计优化编译器的基本原则如下：

- 调查所关注应用领域的重要程序。选择对这些程序表现良好的编译技术。选择部分 kernel 作为标准测试。
- 调查被编译的源语言。从代码质量的视角发现它们的弱点。加入优化以补偿这些弱点。
- 保证编译器对标准测试程序表现良好，所采用的方法对其它程序来说是通用的。

## 2.6 1.6 本书概况

在设计开发一个编译器之前，开发者必须明白编译器的需求。这和编写编译器一样难以确定。我所找到的确定需求的方法是，手工编译几个经典的样例程序，假装你就是编译器。不是骗你！利用某种优化技术，编译器做不了的转换，你也做不了。

在第 2 章，我们就是这么做的，对一个特定的样例程序。对多个样例这么做太重复了。作为替代，我们会总结出若干个针对编译器的需求，这些需求体现在其它样例程序上。然后，我们去钻研设计。每章会描述编译器的后续 phase，给出 phase 涉及的理论，用高级别伪代码描述这个 phase。

我们假设读者能够根据这里给出的高层次描述写出详细的数据结构。也许，要想编写编译器，你必须对数据结构爱之如命。只有爱上复杂的数据结构，你才能享受编写编译器。

## 2.7 1.7 本书用作教科书

这本编译器设计可以被用作第二编译器课程的教科书。本书假设读者熟悉构建编译器前端和简单的代码生成技术，一个学期编译器课程所教授的内容。我考虑过加入一系列练习，让本书成为一本教科书。作为替代，采用了另一个方案，就是让学生直接参与到设计中来。

本书会一直使用图 1.1 中的样例函数，以启发设计，演示细节。如此，它会成为本书大部分阐述的中心。学生们应该把图 1.2 - 1.4 中的样例当作编译过程的运行示例。学生应该把每一章开发的技术应用到样例。本书也会时时给出这些样例的答案，如此学生就可以检查他/她的答案是否符合书本的答案。

图 1.2 是一个经典的矩阵相乘算法。它包含大量的浮点数计算，伴随着不平衡的内存访问。如图所示，里面的循环包含两个浮点运算，三个 load 操作，和一个 store 操作。问题在于，当发生的内存操作比计算更多时，如何从机器获得良好的性能。

图 1.3 计算向量 A 的最长单调序列的长度。这个过程用到了动态规划。数组 C(I) 记录了从位置 I 开始的最长的单调序列。它这样计算下一个元素：对于所有之前计算的序列，检查是否允许把 X(I) 添加到当前计算得到的序列的前端。这个样例几乎没有浮点运算。然而，它做了很多 load 和 store 操作，伴随着数量可观的条件分支判断。

图 1.4 是一个递归式的二分查找算法。学生可能会将它翻译为用指针操作二叉树的过程。此处的挑战是，如何优化内存访问，如何降低过程调用带来的时间消耗。我建议将课程的主要评分和项目关联起来，项目的内容是设计几个优化算法的原型。原型可以被快速地实现，审阅者也按照原型的标准去考察它。作为原型，不需要处理复杂的内存管理问题，而实际的优化编译器会遇到这样的问题。

```
SUBROUTINE MAXCOL(A,N,LARGE,VALUE)
   DOUBLE PRECISION VALUE(N), A(N,N)
   INTEGER N, LARGE(N)
   INTEGER I, J
   DO I = 1, N
      LARGE(I) = 1
      VALUE(I) = DABS(A(1,I))
      DO J = 2, N
         IF (DABS(A(J,I).GT.VALUE(I)) THEN
            VALUE(I) = DABS(A(J,I))
            LARGE(I) = J
         ENDIF
      ENDDO
   ENDDO
END
```

图 1: 图 1.1 Running Exercise Throughout Book

## 2.8 1.8 参考文献

Auslander, M., and M. Hopkins. 1982. An overview of the PL.8 compiler. Proceeding of the ACN SIGPLAN82 Conference on Programming Language Design and Implementation, Boston, MA. Backus, J. W., et al. 1957. The Fortran automatic coding system. Proceedings of AFIPS 1957 Western Joint Computing Conference (WJCC), 188-198.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. Proceedings of the SIGPLAN 82 Symposium on Compiler Construction, Boston, MA. Published as SIGPLAN Notices 17(6): 98-105.

Chaitin, G. J., et al. 1981. Register allocation via coloring. Computer Languages 6(1): 47-57.

Cooper, K., and K. Kennedy. 1988. Interprocedural side-effect analysis in linear time. Proceedings of the SIGPLAN 88 Symposium on Programming Language Design and Implementation, Altanta, GA. Published as SIGPLAN Notices 23(7).

Cytron, R., et al. 1989. An efficient method of computing static single assignment form. Conference Record of the 16th ACM SIGACT/SIGPLAN Symposium on Programming Languages, Austin, TX. 25-35.

Gross, T. 1983. Code optimization of pipeline constraints. (Stanford Technical Report CS 83-255.) Stanford University.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. A register allocation framework based on hierarchical cyclic interval graphs. (Technical report.) McGill University.

Karr, M. 1975. P-graphs. (Report CA-7501-1511.) Wakefield, MA: Massachusetts Computer Associates.

```
SUBROUTINE MATMUL(A,B,C,N)
   DOUBLE PRECISION A(N,N), B(N,N), C(N,N)
   INTEGER N, I, J, K
   DO I = 1, N
      DO J = 1, N
         C(I,J) = 0.0
      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
END
```

图 2: 图 1.2 Matrix Multiply Example

```
INTEGER FUNCTION MONOTONE(A,N)
   DOUBLE PRECISION A(N)
   INTEGER C(N), CMAX
   INTEGER I, J, N
   C(N) = 1
   CMAX = 1
   DO I = N - 1, 1, -1
      C(I) = 1
      DO J = I + 1, N
         IF ((X(I) <= X(J)).AND.(C(I) <= C(J)+1) THEN
            C(I) = C(J)+1
         ENDIF
      ENDDO
      IF (CMAX <= C(I)) THEN
         CMAX = C(I)
      ENDIF
   ENDDO
   MONOTONE = CMAX
END
```

图 3: 图 1.3 Computing the Maximum Monotone Subsequence

```
INTEGER FUNCTION BINARYSEARCH(A,N,L,U,KEY)
   DOUBLE PRECISION A(N), KEY
   INTEGER L, U, N
   INTEGER M
   IF (U < L) THEN
      BINARYSEARCH = 0
   ELSE
      M = (L+U)/2
      IF (A(M) = KEY) THEN
         BINARYSEARCH = M
      ELSIF (A(M) < KEY) THEN
         BINARYSEARCH = BINARYSEARCH(A,N,L,M-1,KEY)
      ELSE
         BINARYSEARCH=BINARYSEARCH(A,N,M+1,U,KEY)
      ENDIF
   ENDIF
END
```

图 4: 图 1.4 Recursive Version of a Binary Search

Kildall, G. A. 1973. A unified approach to global program optimization. Conference Proceedings of Principles of Programming Languages I, 194-206.

Leverett, B. W., et al. 1979. An overview of the Production-Quality Compiler-Compiler project. (Technical Report CMU-CS-79-105.) Pittsburgh, PA: Carnegie Mellon University.

Morel, E., and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. Communications of the ACM 22(2): 96-103.

New York University Computer Science Department. 1970-1976. SETL Newsletters.

Reif, J. H., and H. R. Lewis. 1978. Symbolic program analysis in almost linear time. Conference Proceedings of Principles of Programming Languages V, Association of Computing Machinery.

Scarborough, R. G., and H. G. Kolsky. 1980. Improved optimization of Fortran programs. IBM Journal of Research and Development 24: 660-676.

Sites, R. 1978. Instruction ordering for the CRAY-1 computer. (Technical Report 78-CS-023.) University of California at San Diego.

Wegman, M. N., and F. K. Zadeck. 1985. Constant propagation with conditional branches. Conference Proceedings of Principles of Programming Languages XII, 291-299.

Wulf, W., et al. 1975. The design of an optimizing compiler. New York: American Elsevier.

CHAPTER 3

# 第 2 章编译器的结构

为了决定编译器的结构，编译器编写者需要考虑什么？他需要考虑被编译的源语言，所要求的编译器的速度，所要求的目标机器代码的质量，用户社区的期望，和建造编译器的预算。本章将带你领略编译器编写者决定编译器结构所必须经历的过程。

考虑上述因素设计一个编译器的最好方法，是手工模拟编译过程，去编译由用户社区提供的程序。为了简短起见，本书会使用一个主要的样例程序。我们会用这个样例来决定所需的优化技术，同时决定各种转换的次序。

为了方便阐述，本章简化了这个过程。首先，我们会介绍基本的框架，包括编译器的主要组件和编译器内部单元的结构。然后，我们将手工模拟一个样例程序。

图 2.1 的样例是一个 Fortran 子函数。它找出矩阵中每列的最大的元素，同时保存这个最大元素的索引和绝对值。尽管它是用 Fortran 写的，选择什么源语言是不重要的。这个样例可以用任何常见的源语言编写。当然，有些优化的重要程度在不同的语言中是不一样的，但是所有语言都汇集到一组公共的特性，例如数组、指针、异常、函数等，它们在不同的语言中大同小异。然而，编译器必须很好地编译各个语言的特殊的特性。例如，C 语言具有丰富的索引数组或者描述动态存储的指针构造，Fortran 对形式参数具有特殊的规则，这些参数允许增强的优化。

```
SUBROUTINE MAXCOL(A,N,LARGE,VALUE)
   DOUBLE PRECISION VALUE(N), A(N,N)
   INTEGER N, LARGE(N)
   INTEGER I, J
   DO I = 1, N
      LARGE(I) = 1
      VALUE(I) = DABS(A(1,I))
      DO J = 2, N
         IF (DABS(A(J,I).GT.VALUE(I))
            VALUE(I) = DABS(A(J,I))
            LARGE(I) = J
         ENDIF
      ENDDO
   ENDDO
END
```

图 1: 图 2.1 Finding Largest Elements in a Column

## 3.1 2.1 编译器结构概述

本书是编译器设计过程的简化版本。从无到有设计一个编译器，必须经历反复迭代。首先，构思一种编译器结构（假说）。然后，用这种结构模拟编译过程。如果它如期望的那样工作得很好（它不会），那么这种设计是可接受的。在模拟编译的过程中，我们会发现要修改的地方，或者发现整个框架无法工作。于是，修改框架并再次模拟。重复这个过程，直到出现了令人满意的框架。如果这个框架真的无法工作，就废弃它并重新开始。

关于结构，需要决定两件主要的事情：程序是如何表示的，按照怎样的次序执行转换。编译器前端读入源程序，然后将它翻译为一种称为中间表示（IR）的形式，这是为了编译器后端在其上执行优化、寄存器分配、代码生成等。在 IR 上执行的各种各样的转换，称为 phase。

### 3.1.1 2.1.1 源程序的表示

在翻译的过程中，源程序必须存储在计算机上。这种存储在数据结构中的形式称为 IR。过去的经验告诉我们，这种表示应该满足三个条件：

- 1. 程序的中间形式应该被存储为一种接近机器语言的形式，只有适量的操作被保持为高层级（high-level）的形式，之后也被低层级化（lower）。这让每个 phase 能够操作程序中的所有指令。这样，每种优化算法可以应用于所有指令。如果高层级操作被保持在 IR 中，这些操作的子操作就不能被优

化，或者必须在此之后由专门优化器来优化。

- 2. 编译器的每个 phase 应该在 IR 中保留关于程序的所有信息。其中应该没有隐含的信息，这就是说，同样的信息，有的 phase 知道，有的 phase 不知道。这意味着，每个 phase 具有简单的接口，其输出可以由少量仿真器测试。这个要求暗示着，编译器的任何一个组件都不知道其它组件内部是如何实现的。这样，即使一个组件被修改或替换，也不会损害其它组件。
- 3. 编译器的每个 phase 必须能够独立地测试。这意味着我们必须编写能够读写 IR 样例的辅助程序。被写到文件的表示形式必须是一种二进制或文本的表示。

第二个要求规避了一种软件开发团队的自然倾向。当一个团队在实现组件 B 的时候，可能会以某种方式用到另一个团队实现组件 A 的信息。这可以正常工作，直到将来有一天第一个团队局部修改了他们的实现。突然，组件 B 不再正常工作了。如果第二个团队让第一个团队暗地里保存一些信息，让他们的组件 B 使用，情况就更糟糕了。现在，接口不再是程序的中间表示，而是中间表示加其它（可能无文档记录的）数据。避免这种问题的唯一办法，就是要求接口简单并且有文档记录。

优化的编译器是复杂的。经过数年的开发和维护之后，技术支持团队的大部分工作将是修复问题。几乎做不了进一步的开发，因为没有时间。这种情况之所以存在，是因为大部分编译器只能作为整体得到测试。编译一个测试程序，有些 phase 发生了错误（或者程序编译了，而运行的时候出错了）。编译器的问题在哪里？大概不在你观察到问题的地方。COMPASS 有一句精辟的话：Expletive runs downhill（上游排污，下游遭殃）。（当然，用了实际的 expletive。）其含义是，问题发生在编译器中前面的什么地方，一直没有被注意，直到后面的某个 phase，典型的如寄存器分配，或者构建目标对象（object）模块。做到下面几件事情可以避免这样的问题：

- 必须有程序测试中间表示的合法性。这些程序在编译时被临时调用，检查哪个 phase 产生了不正当的表示。
- 必须时常在 phase 中使用断言，检查要求成立的条件事实上是成立的。产品编译器经常这么做。
- 必须为每个 phase 创建回归测试集。这些测试包含 IR 的特殊版本，就是一个程序在这个 phase 之前被编译得到的 IR。这份 IR 是 phase 的输入，然后仿真 phase 的输出，检查结果程序是否正确运行。

怎么存储程序才能满足这些要求呢？根据经验作出选择，然后用之前讨论的手工模拟来检验它。在这个编译器中，每个函数都会被内部存储为一种类似通用 RISC 处理器的汇编语言的形式。

COMPASS 编译器引擎的经验告诉我们，一个操作计算得到的值必须具有通用的形式。值可能是向量，标量，或者结构化的值。在编译过程的早期，值的形式必须尽可能地接近源程序，这样程序才不会在被分析的过程中丢失信息。

这些观点几乎是自相矛盾的。我们要能够操作程序的最小片段，同时仍然能够恢复源程序中呈现的总体结构。这个矛盾导致了逐步低级化中间表示的想法。最初，LOAD 指令有一套完整的下标表达式，后来，这些专用的 load 指令被替换为机器级别的 load 指令。

汇编语言程序长什么样子？每一行一条机器指令。每条指令包含一个操作符，表明要执行的操作；一系列操作数；一系列存放结果的目标（target）。下面给出了中间表示的准确形式，除了其编码方法：

- 1. 指令编码为记录，多条指令构成记录的一个链表。
- 2. 操作符描述指令的动作，表示为所有操作的一个内建的枚举值。

- 3. 一系列常量操作数。有些指令可能包含常量操作数。它们不太可能得到优化，因此被直接存放在指令中。编译器起初不会使用很多常量操作数，因为这样减小了优化的机会。后来，很多常量被存放在指令中，而不是使用寄存器。
- 4. 一列寄存器代表指令的输入。大多数指令具有固定数量的输入，因此它们可以表示为一个小的数组。起初，假设寄存器的供应是无限的，它们是临时变量。
- 5. 一个寄存器作为指令的输出。

汇编语言也有程序 label，代表程序分支可以到达的地方。为了表示这个概念，中间表示被分割成 block（块），代表指令的直线序列。如果一个 block 中的一条指令被执行了，那么这个 block 中的所有指令都会被执行。每个 block 以一个 label 开始（或者前面有一条条件分支指令），以一条分支指令结束。程序中会加入冗余的分支指令，以保证在任何可能的条件下，在 block 的结尾都有一条分支指令。也就是说，不会 fall-through（直降）到下一个 block。

操作符的数量是巨大的。在目标机器上，每条指令都有一个独特的操作符。开始的时候，它们不会被用到；然而，随着编译的进行，低级化过程会把一系列机器无关的操作符翻译为目标机器的操作符。在此没必要列出所有的操作符。作为替代，图 2.2 列出了样例中用到的指令的子集。

现在，源程序被建模为一个有向图，其中的节点是 block。如果存在一条可能的从第一个 block 到第二个 block 的路径，这两个 block 之间就有一条有向的边。一个唯一的节点称为 Entry，代表源程序的入口点。在图中，入口节点没有前驱节点。类似地，一个唯一的节点称为 Exit，代表源程序的出口点，这个节点没有后继节点。在图 2.3 中，入口节点是 B0，出口节点是 B5。

```
iLDC   c => T        Load constant c into temporary T
i2i    T1 => T2      Copy integer temporary T1 into T2
iSLD   (T1) => T2    Load integer from memory location T1 into T2
iCMPGT T1,T2 => T3   Place true in T3 if and only if T1 > T2
iBCOND T,B1,B2       If T is true, branch to block B1; otherwise branch to block B2
iSUB   T1,T2 => T3   integer T3 = T1 - T2
iMUL   T1,T2 => T3   integer T3 = T1 * T2
iADD   T1,T2 => T3   integer T3 = T1 + T2
iSST   (T1),T2       Store value T2 into integer location T1
dSLD   (T1) => SF1   Load double precision value in address T1 into SF1
dSST   (T1),SF1      Store value in double precision temporary SF1 into address T1
dABS   SF1 => SF2    Place absolute value of value in SF1 into SF2
dCMPLE SF1,SF2 => SF3 Place true in SF3 if and only if SF1 <= SF2
dBCOND SF1,B1,B2     If SF1 is true, branch to B1; otherwise, branch to block B2
d2d    SF1 => SF2    Copy value in SF1 into SF2
BR     B             Unconditionally branch to block B
```

图 2: 图 2.2 Operation Codes Used In Examples

源程序的执行被建模为穿越图的一条路径。这条路径从入口节点开始，到出口节点终止。按照路径上节点的出现次序，路径中的每个节点内的计算得到执行。事实上，路径中的下一个节点是由节点中的计算决定的。在图 2.3 中，一条可能的路径是 B0，B1，B2，B4，B1，B3，B4，B5。这条执行路径意味着，执行 B0 内的所有计算，然后 B1 内的所有计算，然后 B2，等等。注意，B1 和 B4 内的计算被执行了两次。

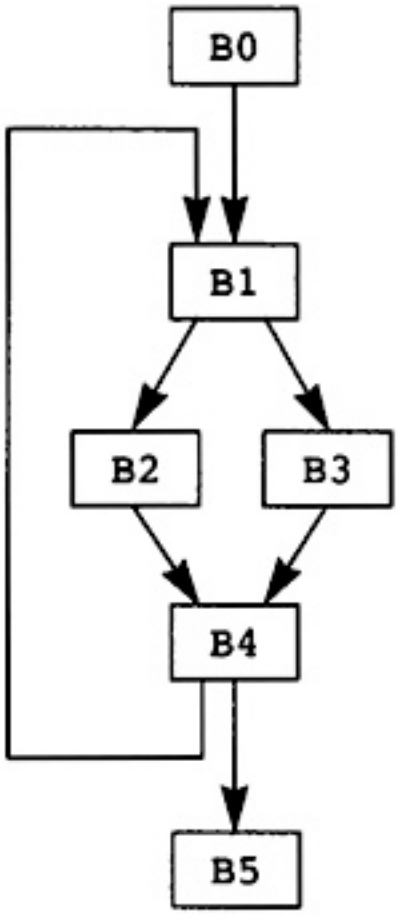


图 3: 图 2.3 Example Flow Graph

### 3.1.2 2.1.2 转换的次序

由于很难线性描述编译器结构，在此对它作个概括，本章的剩余部分再作回顾。本书的其余部分会给出详细内容。如图 2.4 所示，编译器被划分为多个单独的组件，称为 phase。下面，我们来介绍每个 phase 的概况。

编译器前端是语言特定的。它分析被编译的源文件，执行词法分析、解析、语义检查。它构建抽象语法树和符号表。肯定地说，我不会讨论编译器的这个部分，因为大多数教科书都会详细地讲解它。每种语言都有各自独特的前端，而不同语言可以共享编译器的其余部分，只要编译器能够处理好每种语言专用的特征。

在前端构建了抽象语法树之后，初始优化 phase 会构建流图，或者中间表示。由于中间表示看起来像抽象的机器语言，用标准的单 pass 代码生成技术可以构建流图，比如 1cc（Frazer and Hanson 1995）就是这么做的。尽管可以用模式匹配（pattern-matching）技术，由于流图足够简单，直接遍历抽象语法树，随时随地生成指令，就足以构建 IR。在构建流图的时候，可以对每个 block 内的指令作一些初步的优化。

支配者优化 phase 执行初步的全局优化。它可以识别如下情形：值是常量；两次计算会得出相同的值；指令不影响程序的结果。它识别并消除大部分冗余的计算。同时，它会再次应用已经在单个 block 内部执行的优化。它不会移动指令，从流图的一个点移动到另一个点。

过程间优化 phase 分析当前流图中的过程调用和整个程序中所有其它过程的流图。它找出哪些变量可能被过程调用修改，哪些变量和表达式可能引用相同的内存位置，哪些参数是已知的常量。它保存这些信息，为其它 phase 所用。

相关性优化试图优化 load 和 store 操作的执行时间。它是这样做的：分析数组和指针表达式，判断是否可以转换流图，使得在新的流图中 load 和 store 的数量变少了，或者 load 和 store 操作有希望击中 RISC 芯片上的高速缓存（cache）。这可能会交换或展开循环（loop）。

全局优化 phase 低级化流图，消除对数组表达式的符号化引用，将它们替换为线性地址表达式。这个过程会改造地址表达式，精心安排其操作数的位置，使得依赖于内层循环的表达式部分和不依赖于内层循环的操作数相分离。然后执行完整的一组全局优化，包括代码移动、强度减弱、死代码删除。

在全局优化之后，流图中的指令已经确定下来了。现在，编译器必须分配寄存器，并且重排指令以提高性能。在此之前，限制资源 phase 会转换流图，使得后续的 phase 可以正常工作并且更省力。限制资源 phase 会修改流图，减少所需寄存器的数量，以匹配可用的物理寄存器集。如果编译器知道所需的寄存器比可用的寄存器多得多，就会把一些临时变量保存到内存。它也会消除无用的临时变量副本。

接下来，尝试初次调度指令。寄存器分配和指令调度是矛盾的，所以编译器尝试调度指令。它依赖限制资源 phase 的效果，指望它保证寄存器分配能够顺利执行，而不需要进一步地把临时变量保存到内存。指令调度器同时重排来自几个 block 的指令，以减少执行最频繁的 block 所需要的执行时间。

在指令调度之后，寄存器分配 phase 用物理寄存器替换临时变量。这个过程分成三步，依次为下面三类临时变量分配寄存器：首先，在一个 block 中被计算而在另一个 block 中被使用的临时变量；其次，一个 block 中的临时变量，它可以和一个已分配寄存器的临时变量共享寄存器；最后，在单个 block 中被计算和使用的临时变量。这样的划分有赖于限制资源 phase 减小分配冲突的可能性：后面的分配和前面的分配发生冲突。

我们希望寄存器分配 phase 不需要插入将临时变量复制到内存的 store 和 load 操作。如果出现了这样的复制操作，指令调度 phase 就要再次执行。这时，调度器只会调度插入了新的指令的 block。

最终，IR 变成了表示汇编语言函数的形式。目标对象模块按照链接器（linker）要求的形式被写到文件。这是

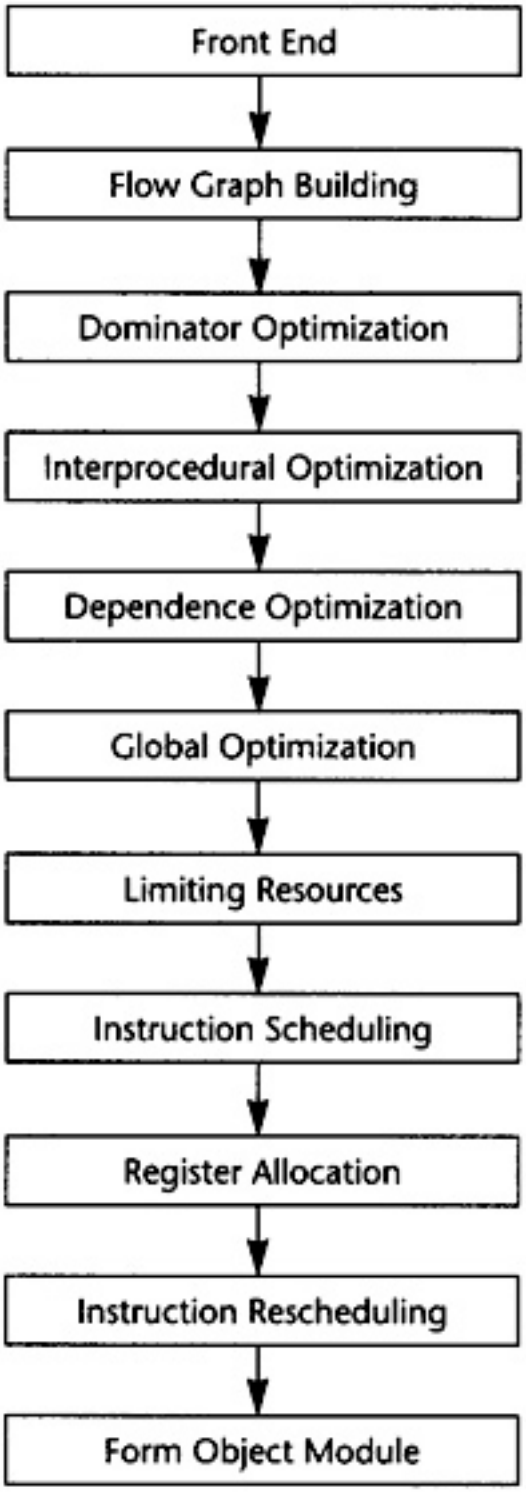


图 4: 图 2.4 Compiler Structure

一个困难的任务，因为关于目标对象模块格式的文档是不准确的，这是众所周知的。主要的工作在于发现正确的格式。之后的任务就是创建目标对象模块，这是琐碎（而又繁重）的工作。

## 3.2 2.2 编译器前端

为了理解每个 phase，我们会为每个 phase 逐行模拟编译图 2.1 中的标准样例，从前端开始。前端将源程序翻译为抽象语法树。正如早前提到的，我不会讨论前端的操作；但是，我们确实需要理解抽象语法树。图 2.1 中的程序的抽象语法树如图 2.5 所示。

每个函数都有一棵树，编码了所有函数结构。树是采用缩进的方式描绘的；每个节点的子树额外缩进一级。如此，节点的类别缩进一级，而子节点缩进得更多一点。我并不试图精确地描述抽象语法树。对于读者来说什么样的节点类别名字能自然地代表节点，就选择什么样的名字。例如，类别为 assign 的节点是赋值节点。

list 节点表示具有任意数量的子节点的节点，用在具有任意数量的部件的情形，例如由语句组成的 block。symbol 节点用一个文本参数表示变量的名字；当然，它实际上被表示为指向符号表的指针。

fetch 节点区分地址和值。这个编译器作了一个统一的关于表达式的假设：表达式总是表示值。因此，assign 节点以两个表达式作为操作数，一个代表位置的地址以接收结果，另一个代表赋值等式右边的值。fetch 节点表示从地址获取值。它有一个参数，代表位置的地址。fetch 节点的结果是存储在这个位置的值。

注意，这个树结构表示了程序完整的结构，指示了子函数的哪些部分包含在别的部分中。

## 3.3 2.3 构建流图

初级编译器书籍介绍了标准的代码生成技术，利用这种技术，抽象语法树被翻译为流图。有两种方法可以实现这种翻译。先进一点的方法是，对抽象语法树应用一种基于树的模式匹配算法来推导流图。在此不推荐这种技术，因为假设目标机器具有 RISC 特征。模式匹配流图会在后来生成复杂的指令。相反，抽象语法树应该被翻译为尽量简单的原子指令。这个函数能获得更多的优化机会。

因此，翻译应该实现为对抽象语法树的一次遍历。只要可能，就应该生成简单指令。一般的操作，像加法和乘法，可以低级化到一种层级，在这种层级中，程序流图中的每个记录代表一条简单指令。但是，此后需要被分析的操作（在源程序的等价层级上）被翻译为高层级的操作，它们等价于源程序的构造。这些操作后来被翻译为低层级操作，在需要分析这些操作的 phase 完成之后。以下四类操作应该被保留为高层级形式：

- 1. 变量下标形式的 fetch 或 store，A[i,j,k]，被保留为单个操作，它具有作为数组名字的操作数和作为下标表达式的操作数。以这种形式保留带下标的变量，而不是线性化下标表达式，使得后续的相关性分析可以求解一系列涉及下标的线性等式和不等式。
- 2. 一般的 load 和 store 操作也保留额外的信息。为了决定哪些 store 操作可以修改 load 操作将要读取的位置，这些信息是需要的。在涉及指针的语言中，这是尤其重要的。额外的分析，称为指针别名分析，被用于决定哪些存储位置被修改了。不会生成自动变量的 load 和 store，它们是在函数中声明的变量，其值在函数结尾时消失。反而，这些值会被当作程序流图中的临时变量一样处理。

```
Procedure
   symbol("MAXCOL")
   list
      symbol("a")
      symbol("n")
      symbol("large")
      symbol("value")
   list
      declare("a",double,array,parameter,bounds=("n","n"))
      declare("n",integer,scalar,parameter)
      declare("large",integer,array,parameter,bounds=("n"))
      declare("value",double,array,parameter,bounds=("n"))
      declare("i",integer,scalar,local)
      declare("j",integer,scalar,local)
   list
      doloop
         symbol("i")
         intconst(1)
         fetch
            symbol("n")
         list
            assign
               subscript
                  symbol("large")
                  list
                     fetch
                        symbol("i")
               intconst(1)
            assign
               subscript
                  symbol("value")
                  list
                     fetch
                        symbol("i")
               absvalue
                  fetch
                     subscript
                        symbol("a")
                     list
                        intconst(1)
                        fetch
                     symbol("i")
            doloop
               symbol("j")
               intconst(2)
               fetch
                  symbol("n")
               list
                  ifthen
                     greater_than
                        absvalue
                           fetch
                              subscript
                                 symbol("a")
                                 list
                                     fetch
                                       symbol("j")
                                     fetch
                                       symbol("i")
                        fetch
                           subscript
                              symbol("value")
                              list
                                 fetch
                                    symbol("i")
                     list
                        assign
                           subscript
                              symbol("value")
                              list
                                 fetch
                                    symbol("i")
                           absvalue
                              fetch
                                 subscript
                                    symbol("a")
                                 list
                                    fetch
                                       symbol("j")
                                    fetch
                                       symbol("i")
                        assign
                           subscript
                              symbol("large")
                              list
                                 fetch
                                    symbol("i")
                           fetch
                              symbol("j")
```

图 5: 图 2.5 Abstract Syntax Tree for MAXCOL

- 3. 子函数调用被保留为代表函数名字的表达式和代表参数的表达式。并不展开传递参数的方法，例如，传值（call-by-value）和传引用（call-by-reference）。这使得后来编译器中的过程间分析 phase 可以作更细致的分析。
- 4. 库函数处理起来不同于其它函数调用。如果一个库函数已知是纯函数，就像处理算子一样处理它。这使得编译器可以应用库函数具有的一致性。程序的分析可能会用到别的函数调用特征，例如，malloc 已知会返回空指针或指向一块内存的指针，这块内存未被程序的其它部分引用。

图 2.6 给出的流图是直接翻译的结果。这里给出这个流图，是为了描述翻译的流程。这不是实际生成的，因为某些优化会在翻译的过程中被执行。注意，有两种不同的使用临时变量的方式。有些临时变量，例如 T5，用起来就像一个局部变量，随着程序的执行，它的值会被修改。其它临时变量，例如 T7，是纯函数的参数。对于 T7 来说，它总是存放常数 1。对于这些临时变量，总是用相同的临时变量存放相同操作的结果。因此，任何对常数 1 的 load 都会存放到 T7。翻译的过程必须保证，操作数在被使用之前求值。

为了保证表达式无论在何处计算都使用相同的临时变量，我们维护一个独立的表，称为形式临时变量表。它被下列实体索引：算子，操作数的临时变量，指令中包含的常量。对这个表的查询的结果，是临时变量的名字，它存放着操作的结果。图 2.7 给出了样例程序的形式临时变量表。有些条目是将在以后添加的，这里列出它们是为了将来引用它们。

在图 2.6 中，对于长长的指令队列，我们观察到的第一件事情是什么？考虑 block B1。常数 1 被加载了六次，表达式 I - 1 计算了三次。当流图构建之后，可以作大量简化：

如果有两个相同计算的实例，在它们之间没有操作修改它们的操作数，那么第二个计算实例是冗余的并且可以被删除，因为它计算得到的值总是跟第一个实例一样。

可以应用代数恒等式消除操作。例如，A * 0 可以被替换为 0。只有当计算 A 的副作用可以被忽略时，才可以这样做。有大量的代数恒等式可以被应用；然而，其中的小部分总是被用起来了，而我们明白，当发现新的代数恒等式可以改进程序的时候，就可以把它们加进来。

常量合并将表达式，例如 5 * 7 ，替换为结果数值，例如 35。这时常会触发其它简化。编译时的算术计算必须精确地模仿目标机器的算术行为。

这些转换通常会删除函数中大约 50% 的操作。因此，编译器中其余的分析会变快，因为在每项分析过程中，大约一半必须扫描的操作都已经被删除了。图 2.8 给出了这些简化的结果。

## 3.4 2.4 支配者优化

预备优化 phase 接收表示为流图的程序作为输入。它对程序应用全局优化技术，生成等价的程序流图作为输出。这些转换会考虑每个函数内可能的分支，就此意义而言，这些技术是全局的。

在编译器中有两种全局优化 phase。初步的 phase 执行尽可能多的全局优化而不移动流图中的计算。在执行了过程间分析和相关性优化 phase 之后，会执行一种更通用的全局优化 phase，以进一步清理和改进流图。下面介绍（此处）会用到的全局优化转换。

如果有计算 X * Y 的两个实例，并且第一个实例处在从 Entry 开始的所有路径上，那么可以删除第二个实例。这是通用冗余表达式删除的一个特殊的情形，即将在此后执行。这个简单的例子代表了最大数量的冗余表达式，因此在应用通用技术之前，有大量的工作要做。

```
B0:   PROLOG
      iLDC      1           => T7     /constant 1 always in T7
      i2i       T7          => T5     /I = 1, first copy into value
      iSLD      (T2)        => T8     /fetch value of N. T2 address of N
      iCMPGT    T5,T8       => T9     /is I > N
      iBCOND    T9,B5,B1              /no iterations to execute

B1:   iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I)), T3 address of LARGE
      iLDC      1           => T7     /constant 1
      i2i       T7          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iLDC      1           => T7     /constant 1 - compute address(A(1,I))
      iLDC      1           => T7     /constant 1
      iSUB      T7,T7       => T18    /1 - 1
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iADD      T19,T18     => T20    /N*(I-1) + (1-1)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T20     => T21    /8*(N*(I-1) + (1-1))
      iADD      T1,T21      => T22    /address(A(1,I))
      dSLD      (T22)       => SF2    /value A(1,I)
      dABS      SF2         => SF3    /DABS(A(1,I))
      d2d       SF3         => SF1    /copy into value of VALUE(I)
      dSST      (T17),SF1             /store the value of VALUE(I)
      iLDC      2           => T23    /constant 2 - Initialize index
      i2i       T23         => T6     /setup value of J
      iSLD      (T2)        => T8     /fetch N
      iCMPGT    T6,T8       => T24    /is J > N
      iBCOND    T24,B4,B2             /yes - no iterations

B2:   iLDC      1           => T7     /1
      iSUB      T6,T7       => T25    /J - 1
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iADD      T19,T25     => T26    /N*(I-1) + (J-1)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T26     => T27    /8*(N*(I-1) + (J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I-1
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      dSLD      (T17)       => SF1    /value of VALUE(I)
      dCMPLE    SF5,SF1     => SF6    /comparison
      dBCOND    SF6,B3,B6             /skip update of variables

B6:   iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I-1
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iLDC      1           => T7     /1
      iSUB      T6,T7       => T25    /J-1
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I-1
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iADD      T19,T25     => T26    /N*(I-1)+(J-1)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T26     => T27    /8*(N*(I-1)+(J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      d2d       SF5         => SF1    /update value of VALUE(I)
      dSST      (T17),SF1             /store it back in memory
      iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I-1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T6          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      BR        B3

B3:   iLDC      1           => T7     /increment by 1
      iADD      T6,T7       => T29    /J+1
      i2i       T29         => T6     /update value of J
      iSLD      (T2)        => T8     /fetch N
      iCMPGT    T6,T8       => T24    /is J > N
      iBCOND    T24,B4,B2

B4:   iLDC      1           => T7     /constant 1 to increment I
      iADD      T5,T7       => T30    /I+1
      i2i       T30         => T5     /first store into fetch location
      iSLD      (T2)        => T8     /get value of N
      iCMPGT    T5,T8       => T9     /is I > N
      iBCOND    T9,B5,B1              /more iterations to perform

B5:   EPILOG                          /return from subroutine
```

图 6: 图 2.6 Initial Program Representation

```
T1    Address(A)                    constant
T2    Address(N)                    constant
T3    Address(LARGE)                constant
T4    Address(VALUE)                constant
T5    Value(I)
T6    Value(J)
T7    Constant 1                    constant
T8    Value of N
T9    I > N
T10   I - 1
T11   Constant 4                    constant
T12   4*(I-1)
T13   Address(LARGE(I))
T14   Value(LARGE(I))
T15   Constant 8                    constant
T16   8*(I-1)
T17   Address(VALUE(I))
SF1   Value(VALUE(I))
T18   1 - 1
T19   N*(I-1)
T20   N*(I-1) + (1-1)
T21   8*(N*(I-1) + (1-1))
T22   Address(A(1,I))
SF2   Value(A(1,I))
SF3   DABS(A(1,I))
T23   Constant 2                    constant
T24   J > N
T25   J - 1
T26   N*(I-1) + (J-1)
T27   8*(N*(I-1) + (J-1))
T28   Address(A(J,I))
SF4   Value(A(J,I))
SF5   DABS(A(J,I))
SF6   DABS(A(J,I)) >= VALUE(I)
T29   J + 1
T30   I + 1
T31   0 > N                         added during value numbering
T32   8*N*(I-1)                     added during value numbering
T33   Address(A(1,I)) simplified    added during value numbering
T34   2 > N                         added during value numbering
T35   T28 + 8                       added during strength reduction
T36   T13 + 4                       added during strength reduction
T37   T17 + 8                       added during strength reduction
T38   8*N                           added during strength reduction
T39   T33 + 8*N                     added during strength reduction
```

图 7: 图 2.7 Initial Formal Temporary Table

复制传播和值传播会被执行。如果 X 是 Z 的副本，那么对 X 的使用可以替换为对 Z 的使用，只要在发生复制的点和发生使用的点之间，X 和 Y 都没有改变。这种转换可以改进编译器前端生成的程序流图。编译器会生成很多临时变量，例如循环计数，或数组附加信息的分量，它其实是复制操作。

常量传播找出已经被赋以常量值的变量，并且把对它们的使用替换为这个常量本身。如果程序中的分支条件用到了常量，就可以决定并删除不会被执行的分支。

当别的优化被应用的时候，作为局部优化，代数恒等式、窥孔优化、常量合并也会被执行。

（此处）编译器有意不应用下面的全局优化，因为它们会让之后的相关性分析更加困难。

强度减弱不应用。强度减弱将带常数的乘法（或循环不变量表达式）转换为重复的加法。更精确地说，如果在循环中有 I * 3 这样的表达式，每次循环 I 都增加 1，那么计算 I * 3 可以被替换为一个临时变量 T，每次循环这个变量都增加 3。

代码移动不应用。考察循环内的计算 X * Y，如果我们知道每次循环它都会被执行，并且在循环内其操作数不会改变，就可以把它移动到循环之前。这种转换会干扰循环交换，后者用于改善数据在高速缓存中的命中率，所以它被延迟到后面的全局优化 phase。

现在仔细观察流图，找出穿越流图的可能的路径，让你的手指沿着几条路径游走，从 block B0 开始，到 block B5 终止。每条路径重复计算了常量 1。代价更高的计算也被重复了。观察 block B2 和 B6。很多在 B6 中计算的表达式也在 B2 中计算了。因为 B2 处在通向 B6 的每条路径上，所以 B6 中的计算是不必要的。

什么样的技术可以廉价地删除这些计算呢？B2 是 B6 的支配者（待会给出精确的定义），意思是 B2 出现在从 B0 通向 B6 的每条路径上。有一系列应用于静态单赋值形式（待会给出定义）流图的算法可以删除重复的常量计算和表达式计算，这些计算已经出现在支配者 block 中。反正编译器要用到一些静态单赋值形式的算法，我们将使用这种形式来删除冗余的计算，就是说这些计算的副本已经出现在支配者 block 中。这是一种代价不高的广义化的局部优化，在构造流图期间使用它，得出图 2.9 中的结果。

重复这样的练习，即追踪穿过流图的路径。现在，流图中几乎没有明显的冗余表达式了。但是，仍然有一些。每次循环都执行的计算还没有被移动到循环之外。通常还有别的冗余表达式，不能由这个转换判定是冗余的，尽管在这个例子中没有出现。

大部分指令出现在哪里？它们在 block B2 中，计算一个地址，用于 load 数组元素。这些地址表达式在每次循环中都会改变，所以不能把它们移出由 block B2 开始的循环。它们有规律地改变，每次循环增加 8，所以后面的全局优化 phase 将应用强度减弱来删除其中的大部分指令。

## 3.5 2.5 过程间分析

在处理程序流图的时候，编译器的所有其它 phase 都是一次处理一个函数。每个 phase 接收作为输入的程序流图（或抽象语法树），生成作为结果的程序流图。而过程间分析 phase 为每个函数积累程序流图。它分析全部流图，向编译器的其余 phase 供给每个函数的程序流图，一次一个。它不是按照它们的原始次序给出函数的。假设没有递归调用，一个函数在别的函数调用它之前被输出给编译器的其余 phase。如此，随着编译过程的推进，更多信息被收集起来。

过程间分析 phase 为编译器的其它 phase 计算关于函数调用的信息。在编译器的局部和全局优化 phase 中，必须对函数调用的效果作出假设。如果函数调用的效果是未知的，那么优化 phase 必须假设，被调用函数知道

```
B0:   PROLOG
      iLDC      1           => T7     /constant 1 always in T7
      i2i       T7          => T5     /I = 1, first copy into value
      iSLD      (T2)        => T8     /fetch value of N
      iCMPGT    T7,T8       =>T31     /is I > N
      iBCOND    T31,B5,B1             /no iterations to execute

B1:   iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T7          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iMUL      T15,T19     => T32    /8*N*(I-1)
      iADD      T1,T32      => T33    /address(A(1,I))
      dSLD      (T33)       => SF2    /value A(1,I)
      dABS      SF2         => SF3    /DABS(A(1,I))
      d2d       SF3         => SF1    /copy into value of VALUE(I)
      dSST      (T17),SF1             /store value into VALUE(I)
      iLDC      2           => T23    /constant 2 - Initialize index
      i2i       T23         => T6     /setup value of J
      iSLD      (T2)        => T8     /fetch N
      iCMPGT    T23,T8      => T34    /is 2 > N
      iBCOND    T34,B4,B2             /yes - no iterations

B2:   iLDC      1           => T7     /1
      iSUB      T6,T7       => T25    /J - 1
      iSUB      T5,T7       => T10    /I - 1
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iADD      T19,T25     => T26    /N*(I-1)+(J-1)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T26     => T27    /8*(N*(I-1)+(J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      dSLD      (T17)       => SF1    /value of VALUE(I)
      dCMPLE    SF5,SF1     => SF6    /comparison
      dBCOND    SF6,B3,B6             /skip update of variables

B6:   iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iSUB      T6,T7       => T25    /J - 1
      iSLD      (T2)        => T8     /fetch N
      iMUL      T8,T10      => T19    /N*(I-1)
      iADD      T19,T25     => T26    /N*(I-1)+(J-1)
      iMUL      T15,T26     => T27    /8*(N*(I-1)+(J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      d2d       SF5         => SF1    /update value of VALUE(I)
      dSST      (T17),SF1             /store it back in memory
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T6          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      BR        B3

B3:   iLDC      1           => T7     /increment by 1
      iADD      T6,T7       => T29    /J + 1
      i2i       T29         => T6     /update value of J
      iSLD      (T2)        => T8     /fetch N
      iCMPGT    T6,T8       => T24    /is J > N
      iBCOND    T24,B4,B2

B4:   iLDC      1           => T7     /constant 1 to increment I
      iADD      T5,T7       => T30    /I + 1
      i2i       T30         => T5     /first store into fetch location
      iSLD      (T2)        => T8     /get value of N
      iCMPGT    T5,T8       => T9     /is I > N
      iBCOND    T9,B5,B1              /more iterations to perform

B5:   EPILOG                          /return from subroutine
```

图 8: 图 2.8 Flow Graph after Simplifications

```
B0:   PROLOG
      iLDC      1           => T7     /constant 1 always in T7
      i2i       T7          => T5     /I = 1, first copy into value
      iSLD      (T2)        => T8     /fetch value of N
      iCMPGT    T7,T8       => T31    /is I > N
      iBCOND    T31,B5,B1             /no iterations to execute
B1:   iSUB      T5,T7       => T10    /I - 1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T7          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iMUL      T8,T10      => T19    /N*(I-1)
      iMUL      T15,T19     => T32    /8*N*(I-1)
      iADD      T1,T32      => T33    /address(A(1,I))
      dSLD      (T33)       => SF2    /value A(1,I)
      dABS      SF2         => SF3    /DABS(A(1,I))
      d2d       SF3         => SF1    /copy into value of VALUE(I)
      dSST      (T17),SF1             /store value in VALUE(I)
      iLDC      2           => T23    /constant 2 - Initialize index
      i2i       T23         => T6     /setup value of J
      iCMPGT    T23,T8      => T34    /is 2 > N
      iBCOND    T34,B4,B2             /yes - no iterations
B2:   iSUB      T6,T7       => T25    /J - 1
      iADD      T19,T25     => T26    /N*(I-1) + (J-1)
      iMUL      T15,T26     => T27    /8*(N*(I-1) + (J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      dSLD      (T17)       => SF1    /value of VALUE(I)
      dCMPLE    SF5,SF1     => SF6    /comparison
      dBCOND    SF6,B3,B6             /skip update of variables
B6:   d2d       SF5         => SF1    /update value of VALUE(I)
      dSST      (T17),SF1             /store it back in memory
      i2i       T6          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      BR        B3
B3:   iADD      T6,T7       => T29    /J + 1
      i2i       T29         => T6     /update value of J
      iCMPGT    T6,T8       => T24    /is J > N
      iBCOND    T24,B4,B2
B4:   iADD      T5,T7       => T30    /I + 1
      i2i       T30         => T5     /first store into fetch location
      iCMPGT    T5,T8       => T9     /is I > N
      iBCOND    T9,B5,B1              /more iterations to perform
B5:   EPILOG                          /return from subroutine
```

图 9: 图 2.9 After Dominator Value Numbering

的值，和它可能调用的所有函数知道的值，都可能被这个函数调用改变或引用。现代语言鼓励用函数（或成员函数）构造程序，这个假设会引起麻烦。为了避免对函数调用作出保守的假设，这个 phase 为每个函数调用计算如下信息：

- MOD 这个函数调用可能修改的变量的集合。
- REF 这个函数调用可能引用的变量的集合。

过程间分析还计算一个函数的形式参数的值和它们之间关系的信息，包括：

- Alias 对于传引用（call-by-reference）的参数，计算哪些参数所引用的内存位置可能与其它参数或全局变量相同。
- Constant 在函数被调用的所有地方总是接受相同常量值的参数。此信息可用于改进已经发生的常量传播。

当涉及到数组引用的时候，过程间分析 phase 尝试找出数组的哪些部分已经被修改或引用了。存储这种信息必须作出粗略的估算，因为只有某种形状的存储引用模式才会被存储。当实际的形状不符合常见的引用模式时，必须作出保守的选择，将形状扩展到可选的形态。

## 3.6 2.6 相关性优化

对于 RISC 处理器来说，相关性优化的目的是减少对内存的引用，改善所发生的内存引用模式。

通过重构循环可以达成这个目标，使得在每次迭代中对内存的引用变少。程序经过转换，消除对内存的引用，如图 2.1 所示，其中用到了称为标量替换的转换，它保存了 A(I) 的值，这个值在循环的下次迭代中被用作 A(I-1)。经典的优化技术无法识别这种可能性，但是相关性优化技术可以做到。可以用一种更复杂的转换，称为展开和阻塞（unroll and jam），为嵌套的循环消除更多对内存的引用。

当对内存的引用不能被完全消除的时候，基于相关性的优化可以被用来改善被引用的值处在高速缓存中的可能性，如此提高对内存的读取速率。现代处理器的速率超过了其内存系统的速率。为了补偿这种的差异，人们引入了高速缓存系统，它保存近期被引用的内存位置的值。近期被引用的内存有可能被再次引用，硬件可以快速地返回保存在高速缓存中的值，而不是再次从内存位置读取出来。

```
DO I = 2, N                          IF (N > 1) THEN
   A(I) = A(I-1) + A(I)                 T = A(1)
ENDDO                                   DO I = 2, N
                                           T = T + A(I)
                                           A(I) = T
                                        ENDDO
                                     ENDIF
```

图 10: 图 2.10 Example of Scalar Replacement

考虑图 2.11 中的 Fortran 片段，它两次复制数组 A 到 B。在 Fortran 中，列的元素被存储在内存中相邻次序的位置。当硬件读取一个特定的元素的时候，会向高速缓存读入一整个缓存线（cache line）（典型地，32 字节到

```
DO I = 1, N                          DO J = 1, N
   DO J = 1, N                          DO I = 1, N
      B(I,J) = A(I,J)*2.0                  B(I,J) = A(I,J)*2.0
   ENDDO                                ENDDO
ENDDO                                ENDDO
```

图 11: 图 2.11 Striding Down the Columns

128 字节），但是下一个元素是行中的下一个元素，它不在这个缓存线中；相反，它可能处在内存中很远的位置。当内层循环结束的时候，外层循环的下次迭代开始执行，当前高速缓存中的元素可能已经被剔除出去了。

基于相关性的优化会将图 2.11 左边的程序转换为右边的程序。执行的计算不变，但是引用元素的次序变了。现在，来自 A 的下一个元素是列中的下一个元素，如此高速缓存就起作用了。

这个 phase 还会展开循环以改善以后的指令调度，如图 2.12 所示。左边的程序是原始的循环；右边的程序是展开的循环。在原始的循环中，编译器随后的 phase 将生成这样的指令，要求每次到 B 的 store 在每次后续迭代的从 A 的 load 之前执行。循环展开之后，从 A 的 load 可能和（到 B 的）store 交错出现，隐藏了引用内存的时间。后面会执行另一个优化，称为软件流水线，它增加更多这样的指令交错。

```
DO I = 1, N                          DO I = 1, N, 4
   B(I) = A(I)                          B(I) = A(I)
ENDDO                                   B(I+1) = A(I+1)
                                        B(I+2) = A(I+2)
                                        B(I+3) = A(I+3)
                                     ENDDO
                                     DO I = I, N
                                        B(I) = A(I)
                                     ENDDO
```

图 12: 图 2.12 Original (left) and Unrolled (right) Loop

## 3.7 2.7 全局优化

全局优化清理前面的 phase 转换得到的流图。此时，所有需要源代码层级信息的转换都已经被执行了，或者信息已经以编码的方式被程序流图吸收了。在执行通用算法之前，需要执行几个转换来简化流图。这些初始的转换都是建立在遍历支配者树和静态单赋值方法之上的。这包括原始的支配者优化和下面的优化。

- Lowering：指令被低级化，使得流图中的每个操作表示目标机器的一条单一指令。复杂的指令，例如数组下标引用，被替换为等价的基本机器指令的序列。或者，多条指令可能被合并成一条单一指令，当常量而不只是存放常量值的临时变量可以出现在指令中的时候。
- Reshaping：在应用全局优化技术之前，编译器会考虑程序的循环结构而转换程序。考虑出现在循环内

的表达式 I * J * K，其中 I 是最内层循环的索引，J 是下一层循环的索引，K 是循环不变量。程序语言典型的结合律会按 (I * J) * K 求值，而更好的方式是按 I * (J * K) 求值，因为在最内层循环中，计算 J * K 是不变量，可以移出这个循环。与此同时，我们会执行强度减弱、局部冗余表达式删除、代数恒等式等。

- Strength Reduction：考虑在一个循环的连续迭代期间按照规则的模式改变的计算。主要的例子是两个值相乘，其中一个值在循环中不改变，例如 I * J，其中 J 不改变，I 增加 1。这个乘法可以替换为一个临时变量，它在每次循环中增加 J。
- Elimination：为了协助强度减弱和流图整形，重复地执行支配者优化中的冗余表达式删除算法。

考虑我们的样例函数。每次内层循环都会计算表达式 address(A(J,I))。它可以被替换为一个临时变量，它初始化为 address(A(1,I))，每次循环迭代增加 8。

强度减弱首先在内层循环执行，然后在相邻的外层循环执行。这个流图有两个嵌套的循环。内层循环由 block B2、B6 和 B3 组成。在算术数列中变化的变量是 J，由符号寄存器 T6 表示。表达式 T25、T26、T27 和 T28 都随着 T6 线性变化，因此它们都是强度减弱的潜在对象；然而，T25、T26 和 T27 被用于计算 T28，因此我们要对 T28 作强度减弱。T28 在每次循环中增加 8。

为了有一个地方存放初始化 T28 的代码，我们在 block B1 和 B2 之间插入一个空的 block。为了方便记忆，我们称它为 B12，代表 B1 和 B2 之间的 block。编译器在循环中放入两组计算（如果不是已经存在的话）：

- 1. 初始化被强度减弱的变量的表达式，在这里变量是 T28。这需要复制所有涉及计算 T28 的表达式，并把它们插入到 block B12。
- 2. 强度减弱表达式的增量的表达式。在这里它是常数 8，已经存在了。

在向 B12 插入这些表达式的时候，编译器会执行冗余表达式删除、常量传播、常量合并等。在这个例子中，编译器知道在循环的入口处 J 的值是 2，因此会用常量值替换 J，也就是 T6。

图 2.13 中的代码表示了对内层循环应用了强度减弱之后的程序。T28 不再表示一个纯的表达式：现在它是一个编译器创建的局部变量。这并没有改变编译器如何处理涉及 T28 的 load 和 store 操作。由于它和之前的纯表达式具有相同的值，load 和 store 操作的副作用保存不变。

在编译器粗略的模拟过程中，我们发现编译器需要在强度减弱之前执行或多或少的冗余表达式删除、常量传播和常量合并。把强度减弱（和表达式整形）当作之前讨论的基于支配者的优化的一部分来执行，可以让我们得到此信息。

作为一个有用的假说，假定在对单入口循环执行强度减弱之前，先对循环入口点及其所有支配者树中的子节点执行基于支配者的转换。如果我们在这个点对循环执行强度减弱，就会获得三个优势。第一，强度减弱先应用于内层循环，后应用于外层循环。第二，循环体已经由基于支配者的算法简化了。还有第三，在循环入口点前面插入的 block 仍然可以得到关于可用表达式和常量的信息。

为了方便说明，在 block B3 中不再使用的计算已经被删除了。在实际中，它们将被后面的死代码删除 phase 删除。这样的次序让强度减弱更容易实现，因为编译器不需要顾虑是否有什么地方用到了一个将要被删除的计算。

现在考虑 block B12 的内容。我们知道 J 或者 T6 的值是 2。于是编译器对这个 block 应用值编码、常量传播、常量合并等优化。为了得到好的代码，需要执行一个别的优化。它执行了加减运算，然后乘以 8。应用整数乘法分发方法会产生更好的代码，因为 8 会被加到一个已经存在的值上，这样生成了图 2.14 中的代码。

```
        ...
        iBCOND    T34,B4,B12              /yes - no iterations
B12:    iSUB      T6,T7        => T25     /J - 1
        iADD      T19,T25      => T26     /N*(I-1) + (J-1)
        iMUL      T15,T26      => T27     /8*(N*(I-1) + (J-1))
        iADD      T1,T27       => T28     /address(A(J,I))
        BR        B2
B2:     dSLD      (T28)        => SF4     /value A(J,I)
        dABS      SF4          => SF5     /DABS(A(J,I))
        dSLD      (T17)        => SF1     /value of VALUE(I)
        dCMPLE    SF5,SF1      => SF6     /comparison
        dBCOND    SF6,B3,B6               /skip update of variables
B6:     d2d       SF5          => SF1     /update value of VALUE(I)
        dSST      (T17),SF1               /store it back in memory
        i2i       T6           => T14     /put in register for LARGE(I)
        iSST      (T13),T14               /store the value into LARGE(I)
        BR        B3
B3:     iADD      T6,T7        => T29     /J + 1
        i2i       T29          => T6      /update value of J
        iADD      T28,T15      => T35     /update value of address(A(J,I))
        i2I       T35          => T28
        iCMPGT    T6,T8        => T24     /is J > N
        iBCOND    T24,B4,B2
```

图 13: 图 2.13 Strength-Reduction Inner Loop

现在对外层循环执行强度减弱。有三个潜在的强度减弱对象：address(A(1,I)) 或 T33；address(VALUE(I)) 或 T17；address(LARGE(I)) 或 T13。再次，我在 block B0 和 B1 之间插入一个 block B01，用来为循环 B1, [B2, B6, B3], B4 存放初始化的值。这三个指针在 block B01 中被初始化，在 block B4 中被增加。

这种模拟过程的一个价值是观察在设计编译器时所想不到的情形。对于强度减弱，有两种这样的情形：

现在，load 常量 4 到 T11 发生得太早了。所有对它的使用已经被删除了，除了在循环的结尾更新指针。在这个例子中，这不是问题，因为后来这个常量会被合并到指令的立即数字段。更复杂的表达式可能在被使用之前早早地被计算出来了。对这个问题，没有容易的解决办法。

在 block B01 中对常量 8 的计算让 block B1 中的计算变得冗余。后面，代码移动算法应该识别这些情况并删除冗余的表达式。

对两个循环都作了强度减弱之后，编译器得到了图 2.15 中的流图。

```
B12:   iADD   T33,T15   => T28   /address(A(2,I)) = address(A(1,I))+8
       BR     B2
```

图 14: 图 2.14 Header Block after Optimization

检查流图，这个时刻正好。编译器已经创建了流图，简化了表达式，删除了大部分冗余表达式，应用了强度减弱，并执行了表达式整形。除了为强度减弱插入了一些专用的代码，没有移动任何表达式。代码移动将把代码移出循环。

这里提议一种代码移动技术，它的基础是 Etienne Morel 设计的称为部分冗余删除的技术（Morel and Renvoise, 1979）。抽象地说，这种技术试图在一些穿过流图的路径上插入表达式的副本，以增加冗余表达式的数目。它发挥作用的一个例子是循环。部分冗余删除算法会在循环前面插入循环不变量表达式的副本，使循环中原始的副本变成冗余。令人惊讶的是，这种技术不需要用到循环的知识。我们把三种别的技术和代码移动结合在一起：

- 1. 在代码移动中结合强度减弱的一种形式。这种技术实现起来代价不高，并且能在没有循环的地方应用强度减弱，这是一种优势。
- 2. 移动 load 和代码移动相结合。移动 load 操作可以被当作一个代码移动问题来处理，通过假装任何 store 操作实际上是一个 store 操作跟随相应的 load 操作。这样，关于一个表达式是否可用，store 操作可以被视作和 load 操作具有相同的效果。正如将要在这个例子中看到的那样，这会增加可以移动的 load 操作的数目。
- 3. 也可以移动 store 操作，通过向后查看流图，并且对反向流图应用一样的算法，这些算法是我们对正常的流图中的表达式所应用的算法。

在这个特定的例子中，代码移动只移除了对常量 4 和 8 的冗余的 load。对 VALUE(I) 的 load 被移出了内层循环。它不是循环不变量表达式，因为在循环中有一个对 VALUE(I) 的 store 操作。然而，一个 store 可以被视作一个 store 跟随一个 load，load 的结果寄存器和 store 的源寄存器相同，这个观点意味着，在使用 VALUE(I) 的每条路径上，都有一个 VALUE(I) 的 load，使得循环中的 load 是冗余的。这时得到了图 2.16 中的代码。

现在，利用部分冗余性，我们可以在反向程序流图上向前移动 store 操作，如图 2.17 所示。在循环中出现的到 VALUE(I) 和 LARGE(I) 的 store 可以被移动到 block B4。虽然我们以为这是移出循环的一次移动，但是分

```
B0:   PROLOG
      iLDC    1           => T7    /constant 1 always in T7
      i2i     T7          => T5    /I = 1, first copy into value
      iSLD    (T2)        => T8    /fetch value of N
      iCMPGT  T7,T8       => T31   /is I > N
      iBCOND  T31,B5,B01           /no iterations to execute

B01:  i2i     T3          => T13   /address(LARGE(I))
      iLDC    4           => T11   /increment for address(LARGE(I))
      i2i     T4          => T17   /address(VALUE(I))
      iLDC    8           => T15   /increment for address(VALUE(I))
      i2i     T1          => T33   /address(A(1,I))
      iMUL    T15,T8      => T38   /increment for address(A(1,I))
      BR      B1

B1:   iLDC    4           => T11   /constant 4
      i2i     T7          => T14   /put in register for LARGE(I)
      iSST    (T13),T14            /store the value into LARGE(I)
      iLDC    8           => T15   /constant 8
      dSLD    (T33)       => SF2   /value A(1,I)
      dABS    SF2         => SF3   /DABS(A(1,I))
      d2d     SF3         => SF1   /copy into value of VALUE(I)
      dSST    (T17),SF1            /store value in VALUE(I)
      iLDC    2           => T23   /constant 2 - Initialize index
      i2i     T23         => T6    /setup value of J
      iCMPGT  T23,T8      => T34   /is 2 > N
      iBCOND  T34,B4,B12           /yes - no iterations

B12:  iADD    T33,T15     => T28   /address(A(2,I)) = address(A(1,I)) + 8
      BR      B2

B2:   dSLD    (T28)       => SF4   /value A(J,I)
      dABS    SF4         => SF5   /DABS(A(J,I))
      dSLD    (T17)       => SF1   /value of VALUE(I)
      dCMPLE  SF5,SF1     => SF6   /comparison
      dBCOND  SF6,B3,B6            /skip update of variables

B6:   d2d     SF5         => SF1   /update value of VALUE(I)
      dSST    (T17),SF1            /store it back in memory
      i2i     T6          => T14   /put in register for LARGE(I)
      iSST    (T13),T14            /store the value into LARGE(I)
      BR      B3

B3:   iADD    T6,T7       => T29   /J + 1
      i2i     T29         => T6    /update value of J
      iADD    T28,T15     => T35   /update value of address(A(J,I))
      i2I     T35         => T28
      iCMPGT  T6,T8       => T24   /is J > N
      iBCOND  T24,B4,B2

B4:   iADD    T5,T7       => T30   /I + 1
      i2i     T30         => T5    /first store into fetch location
      iADD    T13,T11     => T36   /increment address(LARGE(I))
      i2i     T36         => T13   /put pointer back in place
      iADD    T17,T15     => T37   /increment address(VALUE(I))
      i2i     T37         => T17   /put pointer back in place
      iADD    T33,T38     => T39   /increment address(A(1,I))
      i2i     T39         => T33   /put pointer back in place
      iCMPGT  T5,T8       => T9    /is I > N
      iBCOMD  T9,B5,B1             /more iterations to perform

B5:   EPILOG                       /return from subroutine
```

图 15: 图 2.15 After Strength-Reducing Outer Loop

```
B0:  PROLOG
     iLDC     1           => T7   /constant 1 always in T7
     i2i      T7          => T5   /I = 1, first copy into value
     iSLD     (T2)        => T8   /fetch value of N
     iCMPGT   T7,T8       => T31  /is I > N
     iBCOND   T31,B5,B01          /no iterations to execute

B01: i2i      T3          => T13  /address(LARGE(I))
     iLDC     4           => T11  /increment for address(LARGE(I))
     i2i      T4          => T17  /address(VALUE(I))
     iLDC     8           => T15  /increment for address(VALUE(I))
     i2i      T1          => T33  /address(A(1,I))
     iMUL     T15,T8      => T38  /increment for address(A(1,I))
     iLDC     2           => T23  /constant 2 - Initialize index
     BR       B1

B1:  i2i      T7          => T14  /put in register for LARGE(I)
     iSST     (T13),T14            /store the value into LARGE(I)
     dSLD     (T33)       => SF2  /value A(1,I)
     dABS     SF2         => SF3  /DABS(A(1,I))
     d2d      SF3         => SF1  /copy into value of VALUE(I)
     dSST     (T17),SF1            /store value into VALUE(I)
     i2i      T23         => T6   /setup value of J
     iCMPGT   T23,T8      => T34  /is 2 > N
     iBCOND   T34,B4,B12          /yes - no iterations

B12: iADD     T33,T15     => T28  /address(A(2,I)) = address(A(1,I)) + 8
     BR       B2

B2:  dSLD     (T28)       => SF4  /value A(J,I)
     dABS     SF4         => SF5  /DABS(A(J,I))
     dCMPLE   SF5,SF1     => SF6  /comparison
     dBCOND   SF6,B3,B6           /skip update of variables

B6:  d2d      SF5         => SF1  /update value of VALUE(I)
     dSST     (T17),SF1            /store it back in memory
     i2i      T6          => T14  /put in register for LARGE(I)
     iSST     (T13),T14            /store the value into LARGE(I)
     BR       B3

B3:  iADD     T6,T7       => T29  /J+1
     i2i      T29         => T6   /update value of J
     iADD     T28,T15     => T35  /update value of address(A(J,I))
     i2i      T35         => T28
     iCMPGT   T6,T8       => T24  /is J>N
     iBCOND   T24,B4,B2

B4:  iADD     T5,T7       => T30  /I+1
     i2i      T30         => T5   /first store into fetch location
     iADD     T13,T11     => T36  /increment address(LARGE(I))
     i2i      T36         => T13  /put pointer back in place
     iADD     T17,T15     => T37  /increment address(VALUE(I))
     i2i      T37         => T17  /put pointer back in place
     iADD     T33,T38     => T39  /increment address(A(1,I))
     i2i      T39         => T33  /put pointer back in place
     iCMPGT   T5,T8       => T9   /is I > N
     iBCOND   T9,B5,B1            /more iterations to perform

B5:  EPILOG                       /return from subroutine
```

图 16: 图 2.16 After Code Motion

析过程跟循环没有关系。这取决于这样的事实：在每条到达 B4 的路径上都出现了这些 store 操作，这些重复的 store 确实出现在循环内。以上处理结合死代码删除，给我们带来这些优化 phase 的最终结果。

## 3.8 2.8 限制资源

现在，样例函数的程序流图已经被转换为一种适合为目标机器生成指令的形式。在程序流图的操作和目标机器的指令之间，有着一对一的对应关系。还有三件事情将决定结果程序。

- 窥孔优化：多条指令必须被结合成具有相同效果的单条指令。这包括经典的窥孔优化连同简化方法，后者包含将常量合并到可以使用常量的指令中去。
- 指令调度：必须找出最优的指令次序。通过重排指令，指令固有的超过一个机器时钟周期的延迟，可以被其它指令隐藏起来。
- 寄存器分配：程序流图中的临时变量必须被替换为物理寄存器。

不幸的是，指令调度和寄存器分配是相互制约的。如果编译器重排指令以减小执行时间，就会增加存放值所需的物理寄存器的数目。另一方面，如果编译器在指令调度前为临时变量分配物理寄存器，指令重排的尺度就会受到限制。这是一个已知的 phase 次序的问题。对于指令调度和寄存器分配，没有一个天然的次序。

限制资源 phase 执行这三个任务中的第一个，并且为指令调度和寄存器分配预处理代码。它试图这样解决这个问题：在指令调度之前求解部分寄存器分配问题，然后执行指令调度。随后执行寄存器分配，如果寄存器分配器生成了新的指令（spill code），就执行第二次指令调度。

在为指令调度和寄存器分配作准备之前，编译器将程序的中间表示低级化到效率最高的指令系列。这是最后的代码低级化 phase。

我们从修改流图开始，让每个操作对应目标机器的一个操作。由于我们选择了接近 RISC 处理器指令集的指令表示，大部分指令已经对应了目标机器指令。这个步骤通常被称为代码生成；然而，我们对代码生成的理解是更宽泛的。当我们建造流图的时候，我们开始代码生成；我们进一步作代码生成，去低级化流图；现在，我们以代码生成结束，实现流图指令和目标机器指令的对应关系。

为了说明代码低级化，我们假设目标机器具有这样的指令，它包含小常量的立即数操作数。例如，指令可以把小常量和一个立即数操作数相加。或者，load 和 store 操作可以接受一个常量，作为地址的正向偏移。目标机器也有这样的指令，它多次加一个寄存器的 4 或 8 倍，再加另一个寄存器，结果存放到目标寄存器。换句话说，我们考虑像 Alpha 那样的目标处理器。在低级化代码的时候，编译器还会执行下面的操作：

将流图中的指令替换为等价的目标机器指令。如果流图中的指令就是目标机器指令，编译器就让它保持原样。

去除寄存器到寄存器的复制操作。编译器不再接受这样的惯例，就是一个特定的表达式在一个固定的符号化寄存器中被计算出来。现在，全力去除寄存器到寄存器的复制。

在代码低级化的过程中，有些 block 会变成空的。编译器会删除它们。

在 Alpha 处理器上，下面这些重要的指令会简化这个特别的例子：

S4ADDQ 指令计算一个寄存器的 4 倍加另一个寄存器，简化了整数数组的地址算术运算。

S8ADDQ 指令计算一个寄存器的 8 倍加另一个寄存器，简化了双精度数组的地址算术运算。

```
B0:   PROLOG
      iLDC    1           => T7    /constant 1 always in T7
      i2i     T7          => T5    /I = 1, first copy into value
      iSLD    (T2)        => T8    /fetch value of N
      iCMPGT  T7,T8       => T31   /is I > N
      iBCOND  T31,B5,B01           /no iterations to execute

B01:  i2i     T3          => T13   /address(LARGE(I))
      iLDC    4           => T11   /increment for address(LARGE(I))
      i2i     T4          => T17   /address(VALUE(I))
      iLDC    8           => T15   /increment for address(VALUE(I))
      i2i     T1          => T33   /address(A(1,I))
      iMUL    T15,T8      => T38   /increment for address(A(1,I))
      iLDC    2           => T23   /constant 2 - Initialize index
      BR      B1

B1:   i2i     T7          => T14   /put in register for LARGE(I)
      dSLD    (T33)       => SF2   /value A(1,I)
      dABS    SF2         => SF3   /DABS(A(1,I))
      d2d     SF3         => SF1   /copy into value of VALUE(I)
      i2i     T23         => T6    /setup value of J
      iCMPGT  T23,T8      => T34   /is 2 > N
      iBCOND  T34,B4,B12           /yes - no iterations

B12:  iADD    T33,T15     => T28   /address(A(2,I)) = address(A(1,I))+8
      BR      B2

B2:   dSLD    (T28)       => SF4   /value A(J,I)
      dABS    SF4         => SF5   /DABS(A(J,I))
      dCMPLE  SF5,SF1     => SF6   /comparison
      dBCOND  SF6,B3,B6            /skip update of variables

B6:   d2d     SF5         => SF1   /update value of VALUE(I)
      i2i     T6          => T14   /put in register for LARGE(I)
      BR      B3

B3:   iADD    T6,T7       => T29   /J + 1
      i2i     T29         => T6    /update value of J
      iADD    T28,T15     => T35   /update value of address(A(J,I))
      i2i     T35         => T28
      iCMPGT  T6,T8       => T24   /is J > N
      iBCOND  T24,B4,B2

B4:   iSST    (T13),T14              /store the value into LARGE(I)
      dSST    (T17),SF1              /store the value into VALUE(I)
      iADD    T5,T7       => T30   /I + 1
      i2i     T30         => T5    /first store into fetch location
      iADD    T13,T11     => T36   /increment address(LARGE(I))
      i2i     T36         => T13   /put pointer back in place
      iADD    T17,T15     => T37   /increment address(VALUE(I))
      i2i     T37         => T17   /put pointer back in place
      iADD    T33,T38     => T39   /increment address(A(1,I))
      i2i     T39         => T33   /put pointer back in place
      iCMPGT  T5,T8       => T9    /is I > N
      iBCOND  T9,B5,B1             /more iterations to perform

B5:   EPILOG                       /return from subroutine
```

图 17: 图 2.17 After Store Motion

CPYS 指令，接受两个操作数，产生一个浮点值，它结合了一个操作数的符号和另一个操作数的绝对值。可以用它计算绝对值。

使用这些指令可能让其它指令变得不必要，例如，load 常量的指令，或者乘法或位移操作（其目标寄存器也是冗余的）。这些不必要的计算也必须被删除。部分可以由其它优化去做，或者由死代码删除去做。

编译器也会调整 block 的次序，于是条件分支的两个目标可能被替换为一个直降（fall-through）的目标; 但是，我们并不删除额外的分支部分，因为寄存器分配可能需要插入 block，而这样的删除会改变 block 的次序。图 2.18 中的代码展示了代码低级化的结果。在这个点，不要求相同的表达式总是在相同的寄存器中计算，因为这会增加不必要的指令。因此，用一条单一的 iADD 指令增加循环变量。注意，在循环中，用了 S8ADDQ 来增加引用 A 数组的指针。

在代码被低级化的同时，限制资源 phase 通过执行以下转换为指令调度和寄存器分配作准备。

- Rename：相同的临时变量被函数中不相关的两个部分使用，这样的情况很多。其来源是源程序为两个用途使用相同的自动变量，或者是编译器早前的 phase 所作的转换。现在，一部分临时变量的使用被重命名了，它们引用一个新的临时变量。通过使用不相关的名字，寄存器分配更有效率。图 2.19 说明了重命名。在左边的代码中，两个循环使用了相同的索引变量。在重命名之后，使用了两个不同的索引变量，如右边的代码所示。
- Coalesce：在程序流图中很多寄存器到寄存器的操作可以被删除。在 T1 = T2 的复制中，如果在从复制操作到使用 T1 的所有路径上，T1 和 T2 都没有变化，所有对 T1 的引用就可以被替换为对 T2 的引用，可以删除复制操作。删除一个复制操作可以暴露删除更多复制操作的可能性。如果知道一个临时变量和另一个已经被计算的临时变量相等，此编译器会用一种略微更通用的算法删除它。
- Pressure：在程序流图中点 p 处的寄存器压力，是在点 p 处存放临时变量所需的寄存器的数目，这些值是在 p 之前被计算出来，在 p 之后被使用的。为函数分配寄存器时，最大的寄存器压力，是对所需寄存器的最小数目的估值。这不是精确的下方估值，因为可能需要更多寄存器，由于函数中多条路径的相互作用。然而，如果寄存器压力大于可用的寄存器数目，那么在函数的部分区域，一些临时变量将被存储到内存中。这被称为寄存器溢出（spilling）。
- Spilling：限制资源 phase 会考虑寄存器压力超过物理寄存器数目的每个点。它会考虑包含那个点的每个封闭的循环，寻找一个临时变量，它在循环里面没有被使用，但是在后面被使用了（换句话说，它存放了一个跨越整个循环的值）。取一个对最外层循环具有以上属性的临时变量，在循环之前把它存储到内存，在循环之后在把它加载（reload）回来（在必要的地方）。这样，在循环内部的所有区域，寄存器压力减小 1。如果没有这样的跨越循环的临时变量，就选择一个在那个点所在 block 没有被使用的临时变量。如果不存在这样的临时变量，就选择一个在这个 block 中被使用或定义的临时变量。这整个过程会被反复执行，直到在函数中的任何位置，寄存器压力都已经被减小到可用寄存器的数目以下。

为了计算寄存器压力，编译器需要知道在流图的每个点，有哪些以后将被用到的临时变量，换句话说，程序中每个点的活跃的临时变量的集合。为了方便说明，每个临时变量活跃的点的集合被表示为一系列以两个数字表示的间隔，这些数字关联着图 2.18 中的每条指令。如果一个临时变量在间隔的第一条指令的开始处是活跃的，我们就用一个闭合的中括号表示它。如果它在一条指令的中间变得活跃了，我们就用一个开放得小括号表示它。图 2.20 显示了每个寄存器活跃的指令范围。

利用这个信息，我们可以计算程序中每个点所需寄存器的数目，也就是寄存器压力。如果所需寄存器的数目超过了可用的物理寄存器的数目，就无法给所有临时变量分派寄存器。图 2.21 给出了在子函数中每条指令前

```
0    B0:    PROLOG
1           iLDC     1           => T5    /I = 1
2           iSLD     (T2)        => T8    /fetch value of N
3           iBLE     T8,B5,B1             /no iterations if N <= 0

4    B1:    iLDC     1           => T14   /LARGE(I) = 1
5           dSLD     (T1)        => SF2   /value A(1,I)
6           CPYS     SF2         => SF1   /DABS(A(1,I))
7           iLDC     2           => T6
8           iCMPLE   T8,#2       => T34   /is 2 > N
9           iBCOND   T34,B4,B12           /yes - no iterations

10   B12:   iADD     T1,#8       => T28   /address(A(2,I)) - address(A(1,I))+8
11          BR       B2

12   B2:    dSLD     (T28)       => SF4   /value A(J,I)
13          CPYS     SF4         => SF4   /DABS(A(J,I))
14          dCMPLE   SF4,SF1     => SF6   /comparison
15          dBCOND   SF6,B3,B6            /skip update of variables

16   B6:    d2d      SF4         => SF1   /update value of VALUE(I)
17          i2i      T6          => T14   /put in register for LARGE(I)
18          BR       B3

19   B3:    iADD     T6,#1       => T6    /J + 1
20          iADD     T28,#8      => T28   /update value of address(A(J,I))
21          iCMPGT   T6,T8       => T24   /is J > N
22          iBCOND   T24,B4,B2

23   B4:    iSST     (T3),T14             /store the value into LARGE(I)
24          dSST     (T4),SF1             /store the value into VALUE(I)
25          iADD     T5,#1       => T5    /I + 1
26          iADD     T3,#4       => T3    /increment address(LARGE(I))
27          iADD     T4,#8       => T4    /increment address(VALUE(I))
28          S8ADDQ   T8,T1       => T1    /increment address(A(1,N))
29          iCMPGT   T5,T8       => T9    /compare fetch(I) ? fetch(N)
30          iBCOND   T9,B5,B1             /more iterations to perform

31   B5:    EPILOG                        /return from subroutine
```

图 18: 图 2.18 After Code Lowering

```
DO I = 1, N                    DO I1 = 1, N
   A(I) = B(I)                    A(I1) = B(I1)
ENDDO                          ENDDO
DO I = 1, M                    DO I = 1, M
   C(I) = D(I)                    C(I) = D(I)
ENDDO                          ENDDO
```

图 19: 图 2.19 Computing Right Number of Names

后活跃的寄存器。在这个特别的例子中，最大的寄存器压力出现在最内层循环中。这常常是成立的，但并不总是这样。

```
T1    [0,30]
T2    [0,2)
T3    [0,30]
T4    [0,30]
T5    (1,30]
T6    (7,22]
T8    (2,30]
T9    (29,30)
T14   (4,15],(17,23)
T24   (21,22)
T28   (10,22]
T34   (8,9)
SF1   (6,15],(16,24)
SF2   (5,6)
SF4   (12,16)
SF6   (14,15)
```

图 20: 图 2.20 Table of Live Ranges

我们为每个寄存器集计算单独的寄存器压力：整数和浮点数。我们已经展示了整数寄存器的寄存器压力。图 2.21 没有给出浮点寄存器的寄存器压力，这样让这个表更容易理解；然而，程序中只有三个浮点寄存器，所以确定其寄存器压力是简单明了的。

现在，我们来计算每条语句开始处的寄存器压力。这是一对数字，表示在每条指令的开始处活跃的符号化寄存器或物理寄存器的数目，包括整数的和浮点数的。回想形式参数在程序的开始处是活跃的（如果程序在什么地方使用了它们），所以 T1、T2、T3 和 T4 在子函数的开始处是活跃的。

通常来说，小的流图不需要寄存器 spilling。最大的寄存器压力比寄存器的数目小得多。但是，让我们假装机器只有 8 个寄存器。在内层循环的末尾，寄存器压力是 9，在这个点有这么多活跃的符号化寄存器，我们无法把它们安置到可用的寄存器。在寄存器压力是 9 的这个点，符号化寄存器 T1、T3、T4、T5、T6、T8、T14、T24 和 T28 是活跃的；但是，T1、T3、T4、T5 和 T8 在内层循环没有被引用（定义或使用）。因此，其中一

个可以在循环前被挤出（spill），在循环后被再次加载进来。这样会让整个循环的寄存器压力减小 1。理想地，我们会尽可能选择被最少嵌套循环引用的寄存器。但是，这些临时变量在下一层循环都被引用了，因此我们随意地选择存储 T5，它是代表 I 的临时变量。

我们利用堆栈（SP 是一个专用的寄存器）将寄存器 spill 到内存。注意，寄存器压力在某个点达到了峰值，通过 spill 一个寄存器，我们减小了其它点处的寄存器压力。

插入的过程需要两个步骤。首先，在最外层循环开始的地方插入一个 store 操作，在这个循环内临时变量（T5）没有被引用，然后在循环结束的地方插入 load 操作，如果这个临时变量在循环后面是活跃的。其次，优化 load 和 store 的位置，把 load 移动到尽量远离程序开始的地方，把 store 移动到尽量远离程序结尾的地方。这样我们得到了图 2.22 中的代码。

在限制资源 phase 之后，编译器知道在程序流图的任意点其操作所引用的资源都是可用的。编译器剩余的 phase 都会保持这个不变属性，无论它们何时执行转换。

## 3.9 2.9 指令调度

现代 RISC 处理器的实现采用了所谓的流水线架构。这意味着每个操作被划分成多个阶段，执行每个阶段需要一个机器时钟周期。因为每个阶段需要一个时钟周期，所以每个时钟周期可能发出一条新的指令，但是发出之后过了几个时钟周期，这条指令可能还没有完成。不幸的是，大部分代码生成技术试图在一个值被发起计算之后尽快地使用它。在早期的机器上，这项技术是受欢迎的，因为它限制了所需的寄存器的数目。但是，这种次序减慢了 RISC 处理器的执行速率，因为值不是立即可用的。指令调度重排指令，更早地在指令的值被使用之前启动指令，这样处理器就不会被延迟了。

近期的 RISC 处理器可以同时发出若干条指令。这些指令必须是不相关的，并且使用不同的处理器功能单元。调度器必须建立这些指令的分组，称为 packet。一个 packet 中所有的指令可以被同时发出。

原始的指令调度器在单个 block 中调度指令，同时可能会考虑结束前驱 block 的指令。为了调度指令，调度器会创建称为指令依赖图的数据结构，图中的节点是指令，如果第一个指令必须在第二个指令之前执行，就在它们对应的节点之间连接有向的边。每条边标记一个数字，表示在两条指令之间必须等待的机器时钟周期间隔。调度器会对指令依赖图执行一种专用的拓扑排序算法，以找出所需时钟周期总数最小的指令次序。

限制于 block 的调度没有有效地利用 RISC 处理器的多指令同时发动的特性。block 通常是小的，其中的每条指令依赖于 block 中一些其它指令。将指令调度问题考虑为填充一个矩阵，其中的列代表可以被同时发动的指令，行代表执行这个 block 需要的机器时钟周期。block 调度会稀疏地填充这个矩阵：有很多空的槽，说明机器的多指令同时发动的特性没有被用起来。这是一个问题，尤其对于需要很多时钟周期的指令，像 load、store、乘法、除法或浮点运算。RISC 处理器的其它整数运算通常需要一个时钟周期。在编译器中，有若干中技术相互协作来改善这个问题：

- Unroll：编译器前期的 phase 已经执行了循环展开，这增加了 block 的长度，增加了 block 调度器调度指令的机会。
- Superblock：在循环中两条路径汇合的点，很难把指令从汇合点的后面移动到前面。当循环中的后继 block 是短的时候，编译器早前已经复制了这个 block，这样汇合的路径被替换为两个 block，它们只在循环的开头汇合。这个转换是在循环展开的同时被执行的。

```
Inst  Code                                          Live Registers                    Pressure
0     B0:   PROLOG                                  T1,T2,T3,T4                       4
1           iLDC    1           => T5               T1,T2,T3,T4                       4
2           iSLD    (T2)        => T8               T1,T2,T3,T4,T5                    5
3           iBLE    T8,B5,B1                        T1,T3,T4,T5,T8                    5
                                                    T1,T3,T4,T5,T8                    5
4     B1:   iLDC    1           => T14              T1,T3,T4,T5,T8                    5
5           dSLD    (T1)        => SF2              T1,T3,T4,T5,T8,T14                6
6           CPYS    SF2         => SF1              T1,T3,T4,T5,T8,T14                6
7           iLDC    2           => T6               T1,T3,T4,T5,T6,T8,T14             7
8           iCMPLE  T8,#2       => T34              T1,T3,T4,T5,T6,T8,T14             7
9           iBCOND  T34,B4,B12                      T1,T3,T4,T5,T6,T8,T14,T34         8
                                                    T1,T3,T4,T5,T6,T8,T14             7
10    B12:  iADD    T1,#8       => T28              T1,T3,T4,T5,T6,T8,T14             7
11          BR      B2                              T1,T3,T4,T5,T6,T8,T14,T28         8
                                                    T1,T3,T4,T5,T6,T8,T14,T28         8
12    B2:   dSLD    (T28)       => SF4              T1,T3,T4,T5,T6,T8,T14,T28         8
13          CPYS    SF4         => SF4              T1,T3,T4,T5,T6,T8,T14,T28         8
14          dCMPLE  SF4,SF1     => SF6              T1,T3,T4,T5,T6,T8,T14,T28         8
15          dBCOND  SF6,B3,B6                       T1,T3,T4,T5,T6,T8,T14,T28         8
                                                    T1,T3,T4,T5,T6,T8,T14,T28         8
16    B6:   d2d     SF4         => SF1              T1,T3,T4,T5,T6,T8,T28             7
17          i2i     T6          => T14              T1,T3,T4,T5,T6,T8,T28             7
18          BR      B3                              T1,T3,T4,T5,T6,T8,T14,T28         8
                                                    T1,T3,T4,T5,T6,T8,T14,T28         8
19    B3:   iADD    T6,#1       => T6               T1,T3,T4,T5,T6,T8,T14,T28         8
20          iADD    T28,#8      => T28              T1,T3,T4,T5,T6,T8,T14,T28         8
21          iCMPGT  T6,T8       => T24              T1,T3,T4,T5,T6,T8,T14,T28         8
22          iBCOND  T24,B4,B2                       T1,T3,T4,T5,T6,T8,T14,T24,T28     9
                                                    T1,T3,T4,T5,T6,T8,T14,T28         8
23    B4:   iSST    (T3),T14                        T1,T3,T4,T5,T8,T14                6
24          dSST    (T4),SF1                        T1,T3,T4,T5,T8                    5
25          iADD    T5,#1       => T5               T1,T3,T4,T5,T8                    5
26          iADD    T3,#4       => T3               T1,T3,T4,T5,T8                    5
27          iADD    T4,#8       => T4               T1,T3,T4,T5,T8                    5
28          S8ADDQ  T8,T1       => T1               T1,T3,T4,T5,T8                    5
29          iCMPGT  T5,T8       => T9               T1,T3,T4,T5,T8                    5
30          iBCOND  T9,B5,B1                        T1,T3,T4,T5,T8,T9                 6
                                                    T1,T3,T4,T5,T8                    5
31    B5:   EPILOG                                                                    0
```

图 21: 图 2.21 Live Registers and Register Pressure Before Instruction

```
Inst  Code                                          Live Registers                Pressure
0     B0:    PROLOG    (8 byte stack)               T1,T2,T3,T4                   4
1            iLDC      1            => T5           T1,T2,T3,T4                   4
             iSST      (SP),T5                      T1,T2,T3,T4,T5                5
2            iSLD      (T2)         => T8           T1,T3,T4                      3
3            iBLE      T8,B5,B1                     T1,T3,T4,T8                   4
                                                    T1,T3,T4,T8                   4
4     B1:    iLDC      1            => T14          T1,T3,T4,T8,T14               5
5            dSLD      (T1)         => SF2          T1,T3,T4,T8,T14               5
6            CPYS      SF2          => SF1          T1,T3,T4,T8,T14               5
7            iLDC      2            => T6           T1,T3,T4,T8,T14               5
8            iCMPLE    T8,#2        => T34          T1,T3,T4,T6,T8,T14            6
9            iBCOND    T34,B4,B12                   T1,T3,T4,T6,T8,T14,T34        7
                                                    T1,T3,T4,T6,T8,T14            6
10    B12:   iADD      T1,#8        => T28          T1,T3,T4,T6,T8,T14,T28        7
11           BR        B2                           T1,T3,T4,T6,T8,T14,T28        7
                                                    T1,T3,T4,T6,T8,T14,T28        7
12    B2:    dSLD      (T28)        => SF4          T1,T3,T4,T6,T8,T14,T28        7
13           CPYS      SF4          => SF4          T1,T3,T4,T6,T8,T14,T28        7
14           dCMPLE    SF4,SF1      => SF6          T1,T3,T4,T6,T8,T14,T28        7
15           dBCOND    SF6,B3,B6                    T1,T3,T4,T6,T8,T14,T28        7
                                                    T1,T3,T4,T6,T8,T14,T28        7
16    B6:    d2d       SF4          => SF1          T1,T3,T4,T6,T8,T28            6
17           i2i       T6           => T14          T1,T3,T4,T6,T8,T28            6
18           BR        B3                           T1,T3,T4,T6,T8,T14,T28        7
                                                    T1,T3,T4,T6,T8,T14,T28        7
19    B3:    iADD      T6,#1        => T6           T1,T3,T4,T6,T8,T14,T28        7
20           iADD      T28,#8       => T28          T1,T3,T4,T6,T8,T14,T28        7
21           iCMPGT    T6,T8        => T24          T1,T3,T4,T6,T8,T14,T28        7
22           iBCOND    T24,B4,B2                    T1,T3,T4,T6,T8,T14,T24,T28    8
                                                      T1,T3,T4,T6,T8,T14,T28      8
23    B4:    iSST      (T3),T14                       T1,T3,T4,T8                 4
24           dSST      (T4),SF1                       T1,T3,T4,T8                 4
             iSLD      (SP),T5                        T1,T3,T4,T8
25           iADD      T5,#1          => T5           T1,T3,T4,T5,T8              4
             iSST      (SP),T5                        T1,T3,T4,T5,T8
26           iADD      T3,#4          => T3           T1,T3,T4,T5,T8              5
27           iADD      T4,#8          => T4           T1,T3,T4,T5,T8              5
28           S8ADDQ    T8,T1          => T1           T1,T3,T4,T5,T8              5
29           iCMPGT    T5,T8          => T9           T1,T3,T4,T5,T8              5
30           iBCOND    T9,B5,B1                       T1,T3,T4,T5,T8              5
                                                      T1,T3,T4,T5,T8              5
31    B5:    EPILOG                                                               0
```

图 22: 图 2.22 Load and Store Operations for Spill

- Move：代码移动所用的普通优化技术试图为尽可能短的指令序列保持临时变量活跃。调度的时候，我们为每个 block 单独调度。对于执行频繁的 block，我们会重复代码移动算法，但是允许从一个 block 到另一个 block 的指令移动，即使不能减少指令的执行。
- Trace：考虑执行最频繁的 block B，不管怎么找到的，由启发式方法或者统计信息。找出包括 B 的最长的路径，它包含了路径上每个 block 的执行最频繁的前驱节点和后继节点。现在，假装将这条路径视作一个 block，对依赖图作一些修改，以保证条件分支会作出正确的动作。查看在这条路径上是否有任何指令可以被移动到前面的（或后面的）block。
- Software pipelining：在循环是单个 block 的特殊情形下，软件流水线可以给出良好的调度。软件流水线利用依赖图（不是指令依赖图）提供的依赖信息，将循环一次迭代的调度和后续迭代的调度交叠起来。这不会减小每个迭代占用的时间长度（可能会增加），但是让迭代能够更快地开始，从而减少整个循环的执行时间。在其它调度发生之前，提前识别这些可以作软件流水线的 block 和循环，并单独处理它们。

在指令调度期间，会发生一些窥孔优化。之前不相邻的指令，调度之后变得相邻了，于是可以作窥孔优化，例如调度之后出现这样的情形，一个 store 紧随一个对相同位置的 load。因此再次应用一些窥孔优化是有效的。

当指令调度完成的时候，指令的次序就固定了，不可以改变，除非再次执行指令调度。在那样的情况下，可能只需要重新运行 block 调度器。

我们已经缩小了寄存器需求，于是在流图中的每个点寄存器的值都可以被安置到物理寄存器中。现在，我们会重排指令以满足目标处理器的指令调度约束。我们假设一个像 Alpha 21164 这样的处理器，它在每个时钟周期可以发出四条指令。很多整数指令需要一个周期来完成。大部分浮点运算需要四个周期来完成。在任意给定的周期，我们可以发出一条或两条 load 指令，或者一条单一的 store 指令。不能像 load 指令那样在相同的周期内发出一条 store 指令。我们假设在必要时可以将其它整数操作填充进去。像整数乘法或浮点除法这样的指令需要大量时钟周期。

问题是怎么把指令分组为一到四个指令 packet，这样一个 packet 中的所有指令可以被同时发出。编译器也重排指令以尝试保证一个值在被使用之前是可用的，就是让程序不使用一个操作数（值）直到计算它的指令发出之后若干个时钟周期。load 和 store 操作需要的时间是可变的，取决于内存总线的负载，和值是否在高速缓存中。在 Alpha 21164 上，在处理器芯片上有两个高速缓存，而大部分系统在处理器主板上有一个更大的高速缓存。load 指令加载处理器最近的缓存需要 2 个周期，加载次近的缓存需要 8 个周期，加载板上缓存需要 20 个周期，如果数据是在内存中，就需要较长时间。此外，处理器包含优化加载连续内存位置的硬件。如果在两个连续的时钟周期发出两条对连续内存位置的 load 操作，处理器会优化对内存总线的使用。

在决定怎么调度的时候，重要的是至少不考虑无用的分支。在记录周期数的位置，用星号（*）标记它们。

利用硬件旁路，一个比较指令和一个分支指令可以在相同的时钟周期被发出。注意，对（B1 中的）SI9 的赋值可以向前移动，这样消除一个额外的 slot。还有，可以到达 B12 的 block 只有它的前驱节点，所有不需要插入 NOP。

现在，注意以 block B2 开始的内层循环包含三个 block。第一个 block 测试条件，第三个 block 更新迭代。在第三个 block 中，除了一条计算，其余的计算都可以被移动到第一个 block（hoisting），而剩余的指令可以被更有效地调度，通过生成这个迭代 block 的一个副本（superblock 调度）。

注意，在代码的中间插入了若干 NOP。机器一次挑选四条指令，它们对齐到 16 字节边界。在处理下一个 packet 之前，必须初始化当前 packet 中的所有四条指令。为了以最短的时间执行指令，我们必须最大化每个 packet 中不相关指令的数目。图 2.23 给出了指令调度的结果。

```
Number     Instruction                    Comment
0    B0:   PROLOG
1          iLDC     1         => T5      /(0) I = 1
2          iSLD     (T2)      => T8      /(0) fetch value of N
3          iBLE     T8,B5               /(1) no iterations if N <= 0
4          NOP                          /(1)

5    B1:   iLDC     1         => T14     /(0) LARGE(I) = 1
6          dSLD     (T1)      => SF2     /(0) value A(1,I)
7          CPYS     SF2       => SF1     /(1) DABS(A(1,I))
8          iLDC     2         => T6      /(1)
9          iCMPLT   T8,#2     => T34     /(2) is 2 > N
10         iBCOND   T34,B4              /(2) yes - no iterations

11   B12:  iADD     T1,#8     => T28     /(0) address(A(2,I)) = address(A(1,I)) + 8
12         NOP                          /(0) to line up loop entry

13   B2:   dSLD     (T28)     => SF4     /(0) value A(J,I)
14         iADD     T28,#8    => T28     /(0) schedule address update early
15         iCMPLE   T6,T8     => T24     /(1) is J > N
16         iADD     T6,#1     => T6      /(1) J + 1
17         CPYS     SF4       => SF4     /(3) DABS(A(J,I))
18         dCMPGT   SF4,SF1   => SF6     /(7) comparison
19         dBCOND   SF6, B6             /(11) skip update of variables
20         iBCOND   T24, B2             /(12)

21   B4:   iSST     (T3),T14            /(0) store the value into LARGE(I)
22         iADD     T3,#4     => T3      /(0) increment address(LARGE(I))
23         dSST     (T4),SF1            /(1) store the value into VALUE(I)
24         iADD     T4,#8     => T4      /(1) increment address(VALUE(I))
25         iADD     T5,#1     => T5      /(2) I + 1
26         S8ADDQ   T8,T1     => T1      /(2) increment address(A(1,N))
27         iCMPLE   T5,T8     => T9      /(3) compare fetch(I) ? fetch(N)
28         iBCOND   T9,B1               /(3) more iterations to perform

29   B5:   EPILOG                       /(0) return from subroutine

30   B6:   d2d      SF4       => SF1     /(0) Infrequently executed code moved
31         iSUB     T6,#1     => T14     /(0) put in register for LARGE(I)
32         iBCOND   T24,B2              /(1)
33         BR       B4                  /(1)
```

图 23: 图 2.23 Scheduled Instructions

## 3.10 2.10 寄存器分配

寄存器分配 phase 修改程序流图，用物理寄存器替换临时变量。对整个函数作寄存器分配的技术有若干种。有一种是基于图着色算法的。以每个临时变量作为节点构造一个图。如果两个节点（临时变量）不能占用相同的物理寄存器，就在它们之间连一条无向边。寄存器分配问题简化为着色这个图，不同的颜色代表不同的物理寄存器。

另一种寄存器分配方法是基于装箱（bin packing）问题的，每个箱子代表一个物理寄存器。如果在程序中不存在一个点，两个临时变量都需要存放值（同时活跃），那么它们可以被分配到相同的箱子中。

每种技术都有优势和劣势。在考虑条件分支的时候，图着色技术更胜一筹。因为装箱算法往往近似计算需要临时变量存放值的点的集合，利用一些易于求集合交集的数据结构，所以对于有分支的程序，装箱算法的表现不如图着色算法。

在考虑直线型代码的时候，装箱算法比图着色表现更好。因为装箱算法可以在它作分配的时候遍历 block，可以决定何时相同的寄存器可以被立即重用。它还可以利用程序中的操作和它们的次序的信息来决定将哪些临时变量存储到内存，当寄存器不够用的时候（即使已经执行了限制资源 phase，这也可能发生）。图着色没有引用局部性的概念。

这个编译器的寄存器分配器结合了这两种技术。因为限制资源 phase 已经运行过了，几乎不会发生寄存器 spilling 了。图着色被用来为在 block 的开始处存放值的临时变量分配寄存器，因为在这些地方图着色表现最好。一种由 Hendron（1993）建议的装箱算法的修改版被用来为每个 block 中的临时变量分配寄存器。

程序流图中有在 block 开始处活跃的临时变量（全局分配）和在 block 内部活跃的临时变量（局部分配），之前尝试分离它们，遇到了困难，因为无论先作全局分配还是先作局部分配，都会影响寄存器分配的质量。限制资源 phase 的存在解决了这个问题，在任何一个分配发生之前，它已经执行了全局临时变量 spilling。

注意，限制资源 phase 的出现消除了大部分寄存器分配期间的寄存器 spilling。它并没有消除全部 spilling。条件分支的副作用可以引起图着色或者装箱算法执行期间的寄存器 spilling。这种情况是不可避免的，因为最优的寄存器分配是 NP 完全问题。在发生寄存器 spilling 的地方，寄存器分配器会插入所需的 store 和 load 操作。

现在，我们对样例程序应用寄存器分配。首先，编译器必须重新为临时变量计算活跃的点，因为指令调度已经改变了这些点（见图 2.24）。注意，调度器引入了一个局部寄存器的重定义，因此我们需要更早地作 superblock 调度（当我们不知道它会带来惩罚），或者为正确数量的（临时变量）名字重新计算活跃的点，或者局部地为正确数量的（临时变量）名字重新计算活跃的点，当我们制造这些问题的时候。此处我们只处理整数寄存器；在这个例子中，浮点数寄存器是简单的，因为它们全部相互冲突，因此每个临时变量分配一个不同的寄存器。

在临时变量的生命时间信息被计算出来之后，编译器用图着色算法处理在一些 block 的开始处活跃的临时变量，或者处理直接被分派物理寄存器的临时变量。被分派物理寄存器的临时变量是被预先分配的；然而，此处必须考虑它们以避免任何意外的分派。物理寄存器会被命名为 $0、$1 等。注意形式参数对应的临时变量被分派到由目标机器的调用标准指定的物理寄存器。图 2.25 列出了被全局分派的寄存器，连同寄存器的类别。在这个例子中，所有需要的寄存器被称为 scratch 寄存器，这意味着，如果寄存器在函数中被使用了，就不需要保存和恢复寄存器中的值。

在所有 block 开始处活跃的寄存器被分配之后，我们可以分配只在单个 block 内部活跃的符号化寄存器。在这个小的样例中，只有几个。在实际的程序中，这些寄存器的数目比全局活跃寄存器的大得多。图 2.26 列出了这些局部寄存器。寄存器会尽可能地被重用，因为编译器希望最小化所用寄存器的数目。这避免了使用非

```
T1     [1,28],[30,33]   Global
T2     [1,2)            Global
T3     [1,28],[30,33]   Global
T4     [1,28],[30,33]   Global
T5     (1,28],[30,33]   Global
T6     (8,20],[30,33]   Global
T8     (2,28],[30,33]   Global
T9     (27,28)          Local
T14    (5,21),(31,33]   Global
T24    (15,20),[30,32)  Global
T28    (11,20],[30,33]  Global
T34    (9,10)           Local
SF1    (7,23),(30,33]   Global
SF2    (6,7)            Local
SF4    (13,18),[30,30)  Global
SF6    (18,19)          Local
```

图 24: 图 2.24 Live Ranges after Scheduling

```
T1               $16   Parameter
T2               $17   Parameter
T3               $18   Parameter
T4               $19   Parameter
T5               $22   Scratch Register
T6               $23   Scratch Register
T8               $24   Scratch Register
T14              $25   Scratch Register
T24              $20   Scratch Register
T28              $21   Scratch Register
SF1              $f10  Floating Scratch
SF4              $f11  Floating Scratch
Stack Pointer    $30   Stack Pointer
Return Address   $27   Return Address
```

图 25: 图 2.25 Global Register Assignments

scratch 寄存器的必要性，否则就需要在函数开头插入 store 操作以保存它的值，在结尾插入 load 操作以恢复它的值。

图 2.27 给出了结果汇编代码。所有临时变量都已经被替换为寄存器了。没有插入 spill 指令，因此指令序列没有改变。

| | | |
|---|---|---|
| T9 | $23 | Reuse of Register |
| T34 | $21 | New Register |
| SF2 | $f10 | Reuse of Register |
| SF6 | $f12 | New Register |

图 26: 图 2.26 Local Register Assignments

## 3.11 2.11 再次调度

下一个 phase 是再次调度 phase，只有当寄存器分配器更改了指令序列的时候，才会执行它。这是可以发生的，由于窥孔优化或者引入了 spill 代码。如果这些情况都没有发生，就不需要再次调度。

如果寄存器分配器生成了任何指令，就是说发生了寄存器 spill，就再次执行指令调度，但是仅仅针对插入了 load 或 store 操作的 block。

## 3.12 2.12 构建目标对象模块

最后，我们接近完成任务了。指令已经选择出来了；寄存器已经分派好了。剩余的任务是琐碎的，就是将这些信息和全局分配的数据信息翻译为目标对象模块。这项任务包括为调试器插入调试信息。由于我们的任务是漫长的，在此对最后这个 phase 点到为止。它涉及一点点错综复杂的技术。然而，它之所以复杂，是因为目标对象模块的结构是复杂的并且是无文档的。所有我看过的描述目标对象格式的文档都包含严重的错误。因此，这个项目涉及计算机科学试错，以确定链接器想要什么。这个 phase 还会为报表档案产生汇编语言清单，如果要求的话。

## 3.13 2.13 参考文献

Allen, R., and K. Kennedy. “;Advanced compilation for vector and parallel computers. San Mateo, CA: Morgan Kaufmann.

Frazer, C. W., and D. R. Hanson. 1995. A retargetable C compiler: Design and implementation. Redwood City, CA: Benjamin/Cummings.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. A register allocation framework based on hierarchical cyclic interval graphs. (Technical report.) McGill University.

```
Number     Instruction                    Comment
0    B0:   PROLOG
1          iLDC     1          => $22   /(0) I = 1
2          iSLD     ($17)      => $17   /(0) fetch value of N
3          iBLE     $17,B5              /(1) no iterations if N <= 0
4          NOP                          /(1)

5    B1:   iLDC     1          => $24   /(0) LARGE(I) = 1
6          dSLD     ($16)      => $f10  /(0) value A(1,I)
7          CPYS     $f10       => $f10  /(1) DABS(A(1,I))
8          iLDC     2          => $23   /(1)
9          iCMPLT   $17,#2     => $21   /(2) is 2 > N
10         iBCOND   $21,B4              /(2) yes - no iterations

11   B12:  iADD     $16,#8     => $20   /(0) address(A(2,I)) = address(A(1,I)) + 8
12         NOP                          /(0) to line up loop entry

13   B2:   dSLD     ($20)      => $f11  /(0) value A(J,I)
14         iADD     $20,#8     => $20   /(0) schedule address update early
15         iCMPLE   $23,$17    => $25   /(1) is J > N
16         iADD     $23,#1     => $23   /(1) J + 1
17         CPYS     $f11       => $f11  /(3) DABS(A(J,I))
18         dCMPGT   $f11,$f10  => $f12  /(7) comparison
19         dBCOND   $f12, B6            /(11) skip update of variables
20         iBCOND   $25, B2             /(12)

21   B4:   iSST     ($18),$24           /(0) store the value into LARGE(I)
22         iADD     $18,#4     => $18   /(0) increment address(LARGE(I))
23         dSST     ($19),$f10          /(1) store the value into VALUE(I)
24         iADD     $19,#8     => $19   /(1) increment address(VALUE(I))
25         iADD     $22,#1     => $22   /(2) I + 1
26         S8ADDQ   $17,$16    => $16   /(2) increment address(A(1,N))
27         iCMPLE   $22,$17    => $23   /(3) compare fetch(I) ? fetch(N)
28         iBCOND   $23,B1              /(3) more iterations to perform

29   B5:   EPILOG                       /(0) return from subroutine

30   B6:   d2d      $f11       => $f10  /(0) Infrequently executed code moved
31         iSUB     $23,#1     => $24   /(0) put in register for LARGE(I)
32         iBCOND   $25,B2              /(1)
33         BR       B4                  /(1)
```

图 27: 图 2.27 Code after Register Allocation

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. Register allocation using cyclic interval graphs: A new approach to an old problem. (Technical report.) McGill University.

Morel, E., and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. Communications of the ACM 22(2): 96-103.

Wolfe, M. 1996. High performance compilers for parallel computing. Reading, MA: Addison-Wesley.

CHAPTER 4

# 第 3 章图

要想成为编译器编写者，首先必须成为“数据结构迷”。他必须生活，呼吸，并且热爱数据结构，所以我们不会提供所有背景数学知识的完整列表，而通常来说它会出现在编译器书籍中。我们假设您随手可得任何一种数据结构或者编写编译器的书，例如 Lorho (1984) 或 Fischer and LeBlanc (1988)。本书假设您熟悉以下概念，具体含义详见所引用的数据结构书籍。

等价关系和分区。编译器经常计算等价关系或分区集合。等价关系经常被表示为一个分区：将所有相互等价的元素组织在一起，成为一组元素。因此，整个集合可以表示为一组不相交的元素集合。分区经常被实现为 UNION/FIND 数据结构。Tarjan (1975) 开创了这种方法。

集合上的部分有序关系。编译器包含很多显式和隐式的部分有序集合。例如，表达式的操作数必须在表达式之前计算。编译器必须能够表示这些关系。

本章讨论的主题是图。在编译器中，很多数据结构–例如控制流图和调用图–被表示为有向图。无向图用来表示寄存器分配算法中的干涉关系。因此，这里讨论这些主题，只是将这些理论用于实现编译器。讨论的主题如下：

- 用于实现有向图和无向图的数据结构
- 深度优先搜索和有向图中边的分类
- 支配者、后支配者和支配边界
- 计算图中的循环
- 表示集合

## 4.1 3.1 有向图

有向图由一组节点 N 和一组边 E 组成。每条边都有一个尾节点和一个头节点。有些书将边定义为尾节点和头节点的有序对。然而这使得编译器更加难以描述。可能有两条边具有相同的尾节点和头节点。在包含 C switch 语句或 Pascal case 语句的控制流图中，具有相同语句体的两种不同的选择将创建具有相同尾节点和头节点的两条边。

对于控制流图，有两个特殊的节点。Entry 是一个没有前驱的节点，代表程序开始的点。Exit 是一个没有后继的节点，代表程序结束的点。所有执行路径都从 Entry 开始；所有有限路径都在 Exit 处结束，代表一次完整的执行。请注意，无限长的路径是可能的，表示控制流图中的无限循环。

如果一个过程有多个入口点，这在 Fortran 中是可能的，就创建一个单独的不包含任何指令的 Entry 节点，在 Entry 节点和每个实际的入口点之间都有一条边。当指令发出时，程序入口代码被插入到每个入口点。单个 Entry 节点的存在保证了程序分析将得到正确执行。同样，如果有多个节点没有后继者，那么创建一个单独的 Exit 节点，每个原始出口节点和 Exit 节点之间都有一条边。

程序的每次执行都表示为从 Entry 到 Exit 的一条路径。不幸的是，反之则不然：有些从 Entry 到 Exit 的路径并不代表执行路径；例如，如果控制流图中有两个条件分支，它们以相同的条件表达式作为分支条件。在这种情况下，第二个条件分支只能向与第一个条件分支相同的方向跳转。向另一个方向跳转是不可能的。编译器无法识别这种情况，因此它假设所有路径都是可能的。这样的假设减少了优化的数量。

图 3.1 中的图表示运行示例的控制流图。节点 B0 是 Entry 节点。节点 B5 是 Exit 节点。程序中的任何执行路径都表示为 B0 和 B5 之间的一条路径。

有向图使用两种不同的技术实现。通常节点表示为某种数据结构，为了表示边，每个节点添加两个属性：节点的后继集合和前驱集合。X 的后继集是以 X 为尾节点的边的头节点 Y 的集合。类似地，X 的前驱集是以 X 为头节点的边的尾节点 P 的集合。因此在图 3.1 中，B3 的前驱是 B2 和 B6，而 B3 的后继是 B2 和 B4。请注意，任何节点 X 都满足这样的关系：X 是它的每个后继节点的前驱，X 也是它的每个前驱节点的后继。这些集合实现为链表，表示节点的数据结构包含这些链表的头。

另一种技术是为每个节点分配一个整数，并且将每条边表示为布尔矩阵中的位。如果节点 X 和 Y 之间有一条边，则在位置 EDGE[X,Y] 上该位被设置为真；否则设置为假。

后继/前驱表示的优点是扫描所有离开或进入节点的边是有效的。即使有向图是稀疏的，它也是空间有效的，就像大多数流图那样。矩阵方法在构建有向图时更有效，因为它更容易检查特定节点是否已经是后继节点。我们将在寄存器分配期间使用矩阵方法的一种派生方法；除此之外，将使用后继/前驱实现。

在无向图中，边没有方向。一条边不是在特定方向上从一个节点行进到另一个节点。相反，无向图代表了相邻的概念：两个节点相邻或不相邻。利用实现有向图的技术来实现无向图：对于无向图中的每条边 {X，Y}，构建两条边（Y，X）和（Y，X）。在矩阵形式中，这意味着矩阵是对称的，只需要存储一半的矩阵。

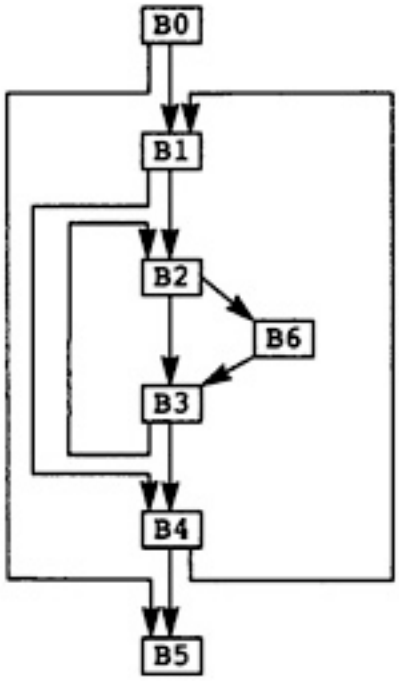


图 1: 图 3.1 Flow Graph for MAXCOL

## 4.2 3.2 深度优先搜索

在有向（或无向）图中访问节点没有自然顺序。编译器中的大多数算法以下列方式访问节点。编译器首先处理一些节点，如果它正在处理控制流图，通常是入口节点。

假设编译器正在处理某个节点 X，在处理 X 的某个时刻，编译器会处理 X 的后继节点。当然，编译器不想多次处理同一个节点，如果 X 的后继节点已经被处理过，它将不会再该后继节点。由于该算法是递归实现的，当 X 经过处理后，它将作为一个过程返回，以便开始处理 X 的前任可以继续处理。

如果有向图是一棵树，则深度优先搜索对应于一棵树的遍历。回想一下，在树的遍历中存在前序遍历的概念，其中一个节点在其后继节点被处理之前被处理；后序遍历，其中节点的子节点在节点上的实际工作完成之前被处理；和中序遍历，其中节点的工作在子节点的处理之间执行。有向图也有类似的想法。

在深度优先搜索期间，该算法可以按照访问节点的顺序为节点分配一个编号。这称为前序。如果按此顺序在节点上执行工作，则它对应于树的前序遍历。同样，按照完成的顺序为节点分配一个编号。这称为后序，对应于树中的后序遍历。一个重要的顺序是反向后序，因为它对应于在处理任何后继节点之前在节点上执行工作（可能除了循环之外）。

图 3.2 给出了深度优先遍历算法。此遍历将边分为四类。一条边 (n,S) 是一条树边，如果当 n 决定处理这个后继者时 S 还没有被处理。换句话说，这是第一次访问 S。由于每个节点只能有一个第一次访问它的前任，因此这些节点与树的边缘一起形成一棵树或树的森林，如图 3.3 所示。这种树结构很重要，因为它允许编译器使用树遍历的概念在控制流图中移动。

第二类包括后向边。这些是从一个节点到另一个已开始处理但尚未完成的节点的边。如果你看一下算法，这意味着边必须回到一个仍在被直接或递归调用它的过程处理的节点：在实现方面，边的头部是仍在堆栈上的

```
input:   Program Flow Graph G = (N,E,S)
         Edges are represented by sets of successors
output:  A classification of each edge and ordering of nodes

procedure DFS(n ∈ node)
   n.pre = preorder;
   preorder = preorder + 1;
   for each S ∈ Succ(n) do
      if S.pre = 0 then
         classify (n,S) as "tree edge";
         DFS(S);
      elseif S.rpost = 0 then          /* S is on Stack */
         classify (n,S) as "back edge";
      elseif n.pre < S.pre then
         classify (n,S) as "forward edge";
      else
         classify (n,S) as "cross edge";
      endif;
   endfor;
   n.rpost = rpostorder;
   rpostorder = rpostorder - 1;
end DFS;

preorder = 1;
postorder = |N|;
for each n ∈ N do
   n.pre = 0;
   n.rpost = 0;
endfor;
DFS(S);
```

图 2: 图 3.2 Basic Depth-First Search Algorithm

节点，并且该节点将是深度优先树中当前节点的祖先。这条边从一个节点到由树边组成的树中的一个祖先。

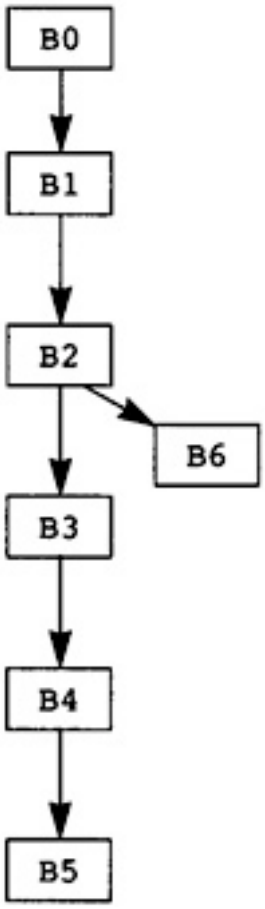


图 3: 图 3.3 Depth-First Search Tree for MAXCOL

与后向边相对的是前向边。从 n 到 S 的前向边是从节点到其后继节点的边；但是，已经处理了后继者。事实上，它是作为 n 的其他一些后继处理的结果进行处理的。所以这是深度优先搜索树中从祖先到后代的边。

没有其他边可以上树或下树，所以第四类边必须从一个子树到另一个子树。这些被称为交叉边。图 3.3 的边分类在表 3.1 中给出。

表 1: Table 3.1 图边的分类

| 树边 | 前向边 | 交叉边 | 后向边 |
|---|---|---|---|
| B0 -> B1 | B0 -> B5 | B6 -> B3 | B3 -> B2 |
| B1 -> B2 | B1 -> B4 | B4 -> B1 | |
| B2 -> B3 | | | |
| B2 -> B6 | | | |
| B3 -> B4 | | | |
| B4 -> B5 | | | |

有一个涉及深度优先搜索的基本原则。考虑从某个节点 n 开始的深度优先搜索。深度优先搜索将访问的节点

集正是离开 n 的某个路径上的节点集。为什么？显然，深度优先搜索遍历所访问的任何节点都在某个路径上，因为树的边缘形成了一条路径。相反，考虑从 n 开始的任何有限路径。下一个节点是 n 的后继节点。在深度优先搜索中，节点的每个后继节点要么从该节点访问过，要么已经被访问过。因为我们从 n 开始，所以从 n 访问这个后继。从 n 到该后继者的边可以被从 n 到后继者的树节点路径替换。现在考虑下一个节点：它要么是从路径上的第二个节点访问过的，要么已经从第一个节点访问过。再次，可以拼接树节点的路径以创建从 n 到第二个节点的路径。这个过程可以一直持续到到达路径上的最后一个节点，此时我们有一条从 n 到结束节点的树边路径，表明通过深度优先搜索到达了结束节点。

我建议您熟悉深度优先搜索。它是编译器中所有其他算法的基础。

## 4.3 3.3 支配关系

由于程序控制流图用于描述程序的执行路径，而优化是一种避免重复已经完成的工作的技术，我们需要一些概念，即在所有执行路径上一个块总是在另一个块之前。这个概念被称为支配地位。

**定义**

*支配者*：考虑一个程序流图（N，入口，出口），当且仅当从入口到 B2 的每条路径都包含 B1 时，块 B1 支配块 B2。

支配者的大部分属性由两种论据决定，每一种都基于支配的定义。第一种论证形式是考虑从入口到块 B 的所有路径。由于支配者在所有这些路径上，可以确定支配者的属性。第二种论证形式是通过剪切和粘贴路径来推理的。考虑一条从 Entry 到 B 的路径，它不包含特定的块 D。通过在末尾添加一条边，可以将该路径扩展为到另一个块的路径；新的路径仍然没有经过 D 。

*引理 D1*：

每个块 B 支配自己，因为 B 在从 S 到 B 的每条路径上。

*引理 D2*：

如果 B2 支配 B1，B1 支配 B，那么 B2 支配 B。

**证明**

考虑从 S 到 B 的每条路径。根据优势的定义，B1 在每条路径上。考虑从 S 到 B1 的子路径。根据支配地位的定义，B2 在这条路上；因此，B2 在从 S 到 B 的每条路径上。也就是说，B2 支配 B。

*引理 D3*：

如果 B2 支配 B，而 B1 支配 B，那么要么 B2 支配 B1，要么 B1 支配 B2。换句话说，B 的支配者形成了一个线性有序的序列。此列表中 B 之后的支配者称为 B 的直接支配者，写作 idom(B)。

**证明**

考虑从 Entry 到 B 的任何路径。如果路径不简单，则丢弃所有循环，在路径中制作一个简单的路径。由于 B2 和 B1 都支配 B，因此它们都在路径上。考虑 B2 在路径上跟随 B1 的情况（B1 跟随 B2 的情况是对称的）。我们声称 B1 支配 B2。为了证明矛盾，假设 B1 不支配 B2。那么一定有一条从 S 到 B2 的路径不包含 B1。用从

S 到 B2 的新路径替换从 S 到 B 的原始路径的第一部分。我们现在有一条不包含 B1 的通往 B 的路径，这与 B1 支配 B 的假设相矛盾。

引理 D3 意味着支配关系可以表示为一棵树，其中每个块的父级是其直接支配者。我们在图 3.4 中为程序 MAXCOL 展示了这棵树。请注意，入口节点 B0 没有直接支配者，因此它是树的根。任何只有一个前驱的节点都将前驱作为其支配者，因为每条路径都必须经过前驱。因此，B2 是 B6 的直接支配者。

计算支配关系的历史很有趣。早期的算法很慢。Purdom (1972) 设计了最早的实用算法之一。为了计算由 B 支配的块，他假装 B 不在图中。然后他执行了深度优先搜索。无法到达的区块只能通过 B 才能到达，因此 B 必须支配它们。在图 3.1 的程序控制流图中，如果我们假设 B2 不在控制流图中，那么块 B2、B3 和 B6 是不可达，所以 B2 支配这三个节点。B2 不支配 B4，因为从 B1 到 B4 的替代路径避开了 B2 。

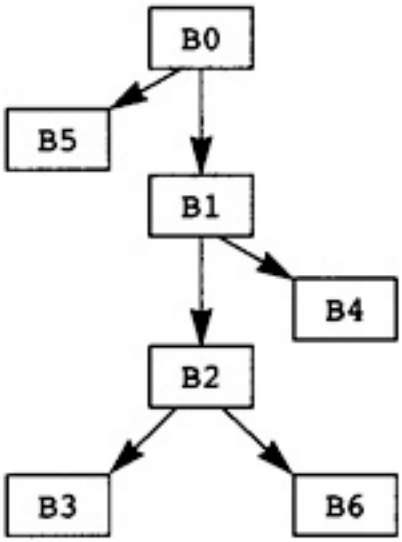


图 4: 图 3.4 Dominator Tree for MAXCOL

当前用于计算直接支配树的算法是由 Lengauer 和 Tarjan (1979) 开发的。该算法有两种形式，运行时复杂度为 O(|N||n|N|) 或 O(|N| (|N|))，具体取决于实现的复杂度。我没有在这里说明算法，因为它太复杂，无法在可用空间中准确描述。相反，我将给出算法的合理化，然后是 Purdom 的更简单的算法，易于理解。

Tarjan 使用在程序控制流图的深度优先搜索期间收集的信息来计算支配者。请注意，B 的支配者是任何深度优先搜索树中 B 的祖先。通常它将是深度优先搜索树中的直接父级。什么时候不会这样？当在深度优先搜索树中有一条不是树边的边进入 B 时。这样的边缘意味着除了树中的路径之外，还有另一种方法可以到达 B。在这种情况下，可以成为 B 支配者的最近块是 B 树中的共同祖先和边缘的尾部。但是现在事情变得复杂了，因为该块可能不是支配者，因为另一条边进入了其中一个块。

为了解决这些问题并存储我们一直在讨论的信息，Tarjan 定义了一个称为半支配数的量，并在深度优先搜索树的自下而上遍历中计算这些值。有了这些值，他可以很容易地计算出实际的支配者。

编译器将支配信息存储为树。树的节点是控制流图中的块；但是，树边不一定是控制流图边。树中任何节点的父节点都是它的直接支配者。对于每个块 B，编译器保留两个存储支配信息的属性：

- idom(B) 是 B 的直接支配者。
- children(B) 是块的集合，其中 B 是直接支配者。从逻辑上讲，这个信息是一个集合；但是，将信息存储

为链表是有用的，由 B 支配的 B 的后继者在列表中排在第一位。这将使后面的一些优化算法更有效地工作。

此树结构导致运行示例的树如图 3.4 所示。

编译器还需要知道一组块的共同支配者。共同支配者是支配块集合的每个元素的块并且被支配集合中的每个块的所有其他块支配。这个共同支配者可以如图 3.5 所示计算。该算法的工作原理是，如果 Z 不支配 B，并且 B 不支配 Z，那么人们可以从其中一个沿着支配树向上走，找到一个支配两者的块。

尽管它计算一对的共同支配者，但该算法适用于任何一组块，因为可以通过成对计算块的共同支配者来找到共同支配者。

这是一个计算支配者的简单算法。回忆一下深度优先搜索的基本原理。访问节点 n 的深度优先搜索也会访问从 n 可达的所有节点。现在通过假设进入 n 的边不存在并且 n 不存在来假设 n 不在图中。在这个残缺图上从 Entry 开始执行深度优先搜索。哪些节点无法从之前可访问的 Entry 访问？如果没有通向它的路径，则该节点不可到达。如果它以前是可达的，这意味着 n 在这些不可达节点的每条路径上。换句话说，n 是所有这些不可达节点的支配者。因此，该算法包括执行单个深度优先搜索以确定所有可到达节点。丢弃无法访问的节点。现在对于控制流图中的每个节点 n，假设 n 不在图中，并从 Entry 开始重复深度优先搜索。不可达的节点是 n 支配的节点。

```
function CommonDominator(Z: block, B: block) returns block;
   if B dominates Z then
      return B;
   endif;
   while Z does not dominate B do
      Z = idom(Z);
   endwhile;
   return Z;
endfunction CommonDominator;
```

图 5: 图 3.5 Computing the Common Dominator

## 4.4 3.4 后支配者

如果编译器将计算移动到控制流图中的较早点，则支配信息会给出控制流图中将计算移动到的安全位置。编译器可以将计算移至当前块的每条路径上的较早块。相反的信息也很有用。如果编译器想要将计算移到稍后的位置，它可以移到哪里？这个问题引出了 postdominance 的想法，它与 dominance 具有相似的特征，除了路径是从 B 到 Exit 而不是从 Entry 到 B，并且使用后继块而不是前驱块。

**定义**

后支配：当且仅当从 B 到 Exit 的每条路径都包含块 X 时，块 X 后支配块 B。

支配的相应性质成立。实际上，后支配只是反向图上的支配关系，其中后继者被前辈取代，反之亦然。通过计算反向图上的支配，可以使用相同的算法来计算后支配。信息可以存储为树，如图 3.6 所示。后支配的属

性如下：

- pdom(B) 表示 B 的直接后支配者，并表示 B 在后支配者树中的父级。
- pchildren(B) 表示立即由 B 后支配的块的集合。这再次表示为实现为链表的集合，其中 B 的前辈也由 B 支配，在列表中首先出现。

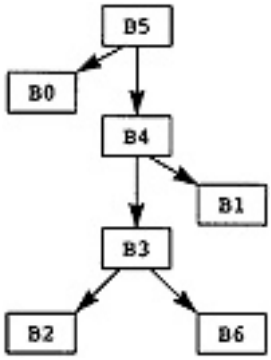


图 6: 图 3.6 Postdominator Tree for MAXCOL

## 4.5 3.5 支配边界

考虑任何离开块 B 的路径。最初路径上的块由 B 支配。最终到达一个不由 B 支配的块。除非路径返回到 B，否则之后的所有块都不受 B 支配。不被 B 支配的第一个块是重要的，因为它指示了 B 支配的块的范围，并使用有关 B 中的计算的信息指示了优化的限制。考虑到所有路径，拥有该特征的块的集合称为支配 B 的边界。

**定义**

支配边界：块 B 的支配边界 DF(B) 是所有块 C 的集合，使得 B 支配 C 的前任，但 B 等于 C 或 B 不支配 C。

该定义是对前述动机的重述。如果 C 是一个块，其前任由 B 支配而 C 不是，则存在从 B 到前任的路径。将该前任的边添加到 C 中，并且一条路径与动机相匹配。显然，与动机相匹配的路径会在支配边界中引入块。

请注意，块 B 是特殊处理的。从 B 开始的循环，经过由 B 支配的块并返回到 B，将 B 引入支配边界。

可视化支配边界的一种方法是考虑以 B 为根的支配树的子树。从该子树中的一个块到子树外部块的控制流图边将子树外部的块引入支配边界。为了便于讨论，B 被认为在子树之外。

这为计算支配边界提供了一个简单的算法。自下而上遍历支配树，在父级之前计算子级的支配边界。在考虑块 B 时，有两种情况：

离开 B 且不导致 B 在支配树中的子节点的流图边必须到达等于 B 或不被 B 支配的块。（如果块由 B 支配，则 B 必须是它的直接支配者，所以它是一个孩子。）这样的块属于 B 的支配边界。

考虑支配树中 B 的一个孩子 C 的支配边界中的块 X。如果 X 不等于 B 并且不被 B 支配，那么它在 B 的支配边界。如果 X 被 B 支配，那么 B 必须是它的直接支配者，因为它不被 C 支配。由于 B 不是它自己的直接支配者，这两个条件可以结合起来给出图 3.7 所示的算法。

```
procedure Calculate_Dominance_Frontier(B: Block)
   foreach C ∈ children(B) do
      call Calculate_Dominance_Frontier(C);
   endfor;
   DF(B) = ∅
   foreach X ∈ Succ(B) do
      if idom(X) ≠ B then
         add X to DF(B);
      endif;
   endfor;
   foreach C ∈ Children(B) do
      foreach X ∈ DF(C) do
         if idom(X) ≠ B then
            add X to DF(B);
         endif;
      endfor;
   endfor;
end Calculate_Dominance_Frontier;
```

图 7: 图 3.7 Computing the Dominance Frontier

表 2: Table 3.2 支配边界

| 块 | 支配边界 |
|---|---|
| B3 | B2 B4 |
| B6 | B3 |
| B2 | B2 B4 |
| B4 | B1 B5 |
| B1 | B1 B5 |
| B5 | Ø |
| B0 | Ø |

考虑图 3.1 中支配树的运行示例。自下而上的支配树遍历首先访问块 B3、B6、B2、B4、B1、B5，然后是 B0。随着行走的进行，支配边界被计算出来（见表 3.2）。在支配边界的计算中，B3 发现 B2 和 B4 在其支配边界中，因为它们是继任者，不受 B3 的支配。类似地，B6 在其支配边界中找到 B3。在计算 B2 的支配边界期间，B3 不会处于其支配边界，因为 B2 支配 B3。但是，B2 处于 B2 的支配边界。

## 4.6 3.6 控制依赖

编译器需要知道一个块的执行导致另一个块的执行的条件。这里描述的想法来自 Cytron（1987、1990 和 1991）。考虑两个块 B 和 X。B 何时控制 X 的执行?

如果 B 只有一个后继块，它不控制任何事情的执行。一旦 B 开始执行，它就完成执行并进入下一个块。因此 B 必须有多个后继者才能被视为控制 X 执行的块。

B 必须有一些离开它的路径通向 Exit 块并避开 X。如果这不是真的，那么 B 的执行将始终导致 X 的执行。换句话说，B 不能被 X 后支配。

B 必须有一些离开它的路径通向 X。同样，这个条件的失败将违反控制的概念。因此 B 可以被视为一个开关：某条出路通向 X，另一条出路避开 X。

B 应该是具有此特性的最新的块。确实，较早的块可能同样控制 X 的执行；但是，该块可以被视为控制 B 的执行，然后 B 控制 X 的执行。

所有这些条件都可以概括为以下定义。

**定义**

*控制依赖*：当且仅当存在从 B 到 X 的非空路径使得 X 后支配除了 B 之外的路径上的每个块时，块 X 是依赖于块 B 的控制。X 与 B 相同，或者 X 不后支配 B 。

第一个条件总结了 B 是具有到 X 的路径的最新块的想法。如果有满足另一个条件的较晚块，则 X 不会在路径上的所有块中占后支配地位。第二个条件与第一个条件中路径的存在一起给出了切换条件。有一种方式通过 B 可以避开 X，而另一种方式必须通向 X。

需要对控制依赖进行更精确的定义，因为编译器需要了解切换机制，从 B 中出的边必须导致 X。这涉及对记录所涉及边的定义的补充。

**定义**

*控制依赖*：块 X 控制依赖于边 (B,S) 当且仅当有一条从 B 到 X 的非空路径从边 (B,S) 开始，使得 X 后支配路径上除 B 之外的每个块。X 要么与 B 相同，要么 X 不后支配 B。

不幸的是，该定义使用了一些未知的路径。为了有一个计算控制依赖的有效方法，编译器需要一个更一般的条件。幸运的是，条件与 X 后支配 S 相同。

**观察**

如果 B 和 X 是控制流图中的块，其中存在从每个块到 Exit 的路径，则 X 后支配 B 的后继 S 当且仅当存在从 B 到 X 通过 S 的非空路径使得 X 后支配该路径上 B 之后的每个节点。

**证明**

假设路径存在。因为 S 在路径上，所以 S 被 X 后支配。相反，假设 S 被 X 后支配。从 S 到 Exit 有一些路径。由于 S 受 X 后支配，因此 X 在这条路径上。在 X 处剪短路径并将 B 和从 B 到 S 的边添加到路径的开头。这给出了从 B 到 X 的路径。路径上除 B 之外的每个节点都由 X 后支配。如果不是，则有一条从它到 Exit 的路径，通过剪切原始路径并粘贴到新路径中，一个可以创建一条从 S 到 Exit 的路径，从而避免 X，这是一个矛盾。因此我们有该路径。

**观察**

如果 S 是 B 的后继者，则要么 S 是 B 的后支配者，要么 pdom(S) 被 pdom(B) 后支配。

**证明**

假设 S 不是 B 的后支配者。考虑从 S 到 Exit 的任何路径。有可能延伸到从 B 到 Exit 的路径。因此，pdom(B) 在这条路径上。因此 pdom(B) 不等于 S 并且在从 S 到 Exit 的每条路径上，因此它是 S 的后支配者。因此它必须后支配 pdom(S)。

现在我们可以给出一个计算控制依赖关系的算法。看定义：边 (B,S) 是给定的。哪些块是控制依赖于这条边的？任何后支配 S 并且不后支配 B 的块。这些是后支配树中的节点，从 S、pdom(S)、pdom(pdom(S)) 开始，并在但不包括 pdom(B) 处停止。第二个观察表明，通过父母（后支配者）向上遍历树，算法必须最终到达 pdom(B)。

图 3.8 中的算法可以应用于每条边。实际上，它需要应用于留下具有多个后继者的块的每条边，因为具有单个后继者的块可以没有依赖于它的块控制。对于我们的运行示例，这给出了表 3.3 中的结果。有时编译器需要转置这些信息：对于每个块，它依赖于哪些块。在这种情况下，使用相同的算法；然而，信息是由依赖块而不是由导致依赖的边索引存储的。

```
function Find_Control_Dependence((B,S): edge)
   depends = ∅;
   X = S;
   while X ≠ (pdom(B) do
      add X to depends;
      X = pdom(X);
   endwhile;
   return depends;
endfunction Find_Control_Dependence;
```

图 8: 图 3.8 Calculating Control Dependence

表 3: 表 3.3 示例程序的控制依赖关系

| 边 (B,S) | 控制依赖于 (B,S) 的块 |
|---|---|
| (B0,B5) | ∅ |
| (B0,B1) | B1, B4 |
| (B1,B4) | ∅ |
| (B1,B2) | B2, B3 |
| (B2,B3) | ∅ |
| (B2,B6) | B6 |
| (B3,B2) | ∅ |
| (B3,B4) | ∅ |
| (B4,B1) | B1, B4 |
| (B4,B5) | ∅ |

## 4.7 3.7 循环和循环树

优化编译器试图减少程序执行期间发生的计算次数。因此，编译器需要确定程序中最常执行的区域并集中精力改进它们。在编译时确定频繁执行的区域是不切实际或不可能的。然而，重复执行的程序部分，即循环，是最佳候选者。因此编译器构建了一个数据结构来表示有关循环的信息。

**定义**

环形：循环是一组块 L，如果 B0，B1 ϵ L，则存在从 B0 到 B1 的路径和从 B1 到 B0 的路径。如果 B 有一个不在 L 中的前任，则块 B ϵ L 是入口块。如果 B 有不在 L 中的后继者，则块 B ϵ L 是退出块。

换句话说，循环是程序的一个区域，其中执行路径可以重复地从一个块循环到另一个块。入口块是执行可以进入循环的块，退出块是执行可以离开循环的块。由于我们假设从 Entry 到任何块都有一些执行路径，因此每个循环必须至少有一个入口块。有趣的循环是具有单个入口块的循环，或单入口循环。对于这样的循环，入口块必须支配循环中的所有其他块。如果存在避开入口块的路径，则路径上的循环中必须存在第一个块，并且该块将是另一个入口。

图 3.9 给出了计算单入口循环的循环块的算法。考虑任何块 B。它可以成为单入口循环的入口块的唯一方法是在控制流图的某个深度优先搜索行走中是否存在后边。考虑替代方案：循环中的入口块必须包含在循环路径中，并且是循环中到达的第一个块。因此，循环中的所有块都将是 B 在遍历中的后代，并且通向 B 的边是后边。

该算法背后的想法是向后走循环。考虑 B 的每个前驱来自后边。从这些前驱向后走图。最终走回到 B，并且循环中的所有块都将被访问。该算法使用工作列表算法实现了这个想法。集合队列包含所有已知在循环中但其前驱尚未处理的块。每个块最多被插入一次队列，因为队列 ∴ 循环并且插入仅在块尚未在循环中时发生。

稍后我们将推广这个算法来处理多入口循环，并用它来计算循环的嵌套结构。编译器不仅需要知道循环，还需要知道哪些循环包含在其他循环中。请注意，编译器计算循环的方式将确保所标识的循环是不相交的（没有共同的块）或嵌套的（一个循环是另一个循环的子集）。嵌套结构用于三个目的：

1. 编译器在基于依赖的优化期间使用循环嵌套，因为这些阶段转换循环以提高程序性能。
2. 循环嵌套用于执行一种强度降低。在循环的每次迭代期间以常规方式修改的值可以以更有效的方式计算；例如，乘法可以用重复的加法代替。
3. 在寄存器分配期间使用循环嵌套来查找程序中可以存储值或从内存中加载值的点。

### 4.7.1 3.7.1 无限循环

一个循环可能没有退出块，在这种情况下它是一个无限循环。这样的循环可能发生在实际程序中。考虑一个程序，它使用硬件中断或信号机制来执行所有操作，而主程序仍处于循环中。程序员可以把这个循环写成一个无限循环。这些是结构性的无限循环。由于程序执行期间发生的实际计算，可能存在编译器无法确定的其他无限循环。

当存在这些结构性无限循环时，许多全局优化算法会给出错误的结果。这些算法都是基于减少从入口到出口的路径计算数量的想法。如果有一个没有这样的路径的块，算法可能会以意想不到的方式执行。

```
procedure FIND_LOOP(B: block)
      Loop = ∅;
      Queue = ∅;
      foreach P ∈ Pred(B) do
         if (P,B) is a backedge then
            if (P ∉ Loop) ∧ (P ≠ B)then
               add P to Queue;
               add P to Loop;
            endif;
         endif
      endfor;
      while Queue ≠ ∅ do
         take X from Queue;
         delete X from Queue;
         foreach P ∈ Pred(X) do
            if (P ≠ B) ∧ (P ∉ Loop)then
               add P to Queue;
               add P to Loop;
            endif
         endfor;
      endwhile;
   add B to Loop;
endprocedure FIND_LOOP;
```

图 9: 图 3.9 Template of Code for Finding a Loop

一个简单的设备消除了这些结构性的无限循环：从循环中的一个块插入一条边到退出。当然，这条边永远不会被遍历，因为块中没有指令可以使程序沿着这条边流动。但是，优化算法现在将正常执行。

编译器如何识别这些无限循环？如果没有从它到 Exit 的路径，则该块处于无限循环中。因此，在控制流图的反向上执行深度优先搜索（将前驱视为后继，反之亦然）。未被访问的块是无限循环中的块。在深度优先搜索之后，选择一个未被访问的块，在它和 Exit 之间创建一条边，然后尝试使用这条边继续进行深度优先搜索。图 3.10 描述了这个算法。

```
procedure FIND_INFINITE_DFS(B: block)
   if B ∉ Visited then
      add B to Visited;
      foreach P ∈ Pred(B) do
         call FIND_INFINITE_DFS(P);
      endfor;
   endif;
endprocedure FIND_INFINITE_DFS;

procedure FIND_INFINITE;
   Visited = ∅;
   call FIND_INFINITE(Exit);
   while Visited ≠ All_Nodes do
      take B from All_Nodes Visited;
      add (B,Exit) as edge in flow graph;
      call FIND_INFINITE_DFS(B);
   endwhile;
end;
```

图 10: 图 3.10 Eliminating Infinite Loops

### 4.7.2 3.7.2 单入口和多入口循环

如前所述，循环可以按入口块的数量进行分类。没有入口块的循环是不可达的：指令无法执行，因此这些循环已经被消除。单入口循环对于优化器来说是最有趣的。必须处理多入口循环，因为它们可能出现在程序中；但是，优化技术不会那么有效。许多优化技术仅适用于单入口循环。[1]

编译器如何识别多入口循环？循环是循环路径的联合。考虑这些循环路径之一。在深度优先搜索期间，访问的路径上有第一个块 B。循环上的所有其他块都是 B 的后代，进入 B 的循环边是后边。因此，通过考虑这些前辈并向后走循环，可以找到如图 3.9 所示的带有条目 B 的循环。多入口循环的问题在于，此遍历可以从循环中逃脱（向后遍历其他入口之一）并最终一路返回入口。这意味着 B 不支配这些前辈。考虑图 3.11 中的多入口循环 {C,D} 。如果深度优先搜索按 {A,C,D,E,B} 的顺序访问块，则 C 是循环中被访问的第一个块。边 (D,C) 是后边。当从 D 向后走时，访问 {D,C,B,A}。

[1] 单入口循环通常称为可约循环。多入口循环称为不可约循环。该编译器使用优化单入口循环的技术。识别多入口循环以确保不发生不正确的翻译。

为了避免这个问题，必须修改算法以停止后向行走。但是步行应该停在哪里？编译器想要一个单入口区域，即使它不是循环。因此，在最靠近循环且支配循环中所有块的块处停止步行。这将是控制标头 B 和 B 的所有前辈通过后缘到达 B 的块。回想一下，B 支配自己。使用这些信息，图 3.9 中的算法被修改为图 3.12 中的算法。

该算法实现了我们刚刚讨论的想法。请注意，当遇到多入口循环时，此时不会计算循环体。相反，导致循环体的块集记录在称为生成器的属性中。在开始循环识别之前，该集合将被初始化为空。具有非空生成器集的块是多入口循环的直接支配者。由于以下原因，无法立即识别循环体：

我们很快就会看到，整个过程都嵌入在深度优先搜索中，在该搜索中，从一个块开始的循环会在遍历的所有块都处理完之后才被识别。记录生成器集允许这对于多入口循环也是如此。

不止一个多入口循环可以有相同的直接支配者。对于形成循环嵌套的过程，聚合将被视为一个循环。

我们将能够更有效地处理此循环中包含的循环。考虑一个多入口循环，入口块 B1 和 B2 具有公分母 C。通过延迟循环的识别直到所有后继者都被识别，将处理发生在 C 和 B1 或 C 和 B2 之间的路径上的循环作为嵌套循环。如果此子循环是单入口循环，则可以对其应用整套优化。如果在处理 B1 或 B2 时创建了多入口循环的主体，则这些子循环不会被视为单独的循环。

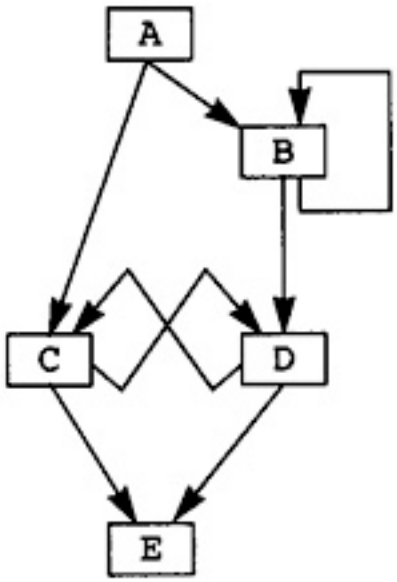


图 11: 图 3.11 Example Multiple-Entry Loop

我们将对 FIND_LOOP 稍作修改，以构建测试循环树，但这是基本算法。当找到一个单入口循环时，循环体被识别。当找到一个多入口循环时，循环体的识别被延迟到块 Z 的处理。这个循环体由非空生成器（Z）集合的存在来识别。

后面的描述将 FIND_LOOP 分为两个过程：第一个找到生成器，第二个找到循环体。该过程被拆分，以便查找多入口循环的主体可以使用与单入口循环相同的代码。

```
function FIND_LOOP(B: block) returns boolean;
   Z = B;                        //Entry block
   Loop = ∅;                     //Blocks in the loop
   Queue = ∅;                    //Unprocessed blocks
   foreach P ∈ Pred(B) do
      if (P,B) is a backedge then
         if (P ∉ Loop) ∧ (P ≠ B) then
            Z = CommonDominator(Z,P);
            add P to Queue;
            add P to Loop;
         endif;
      endif
   endfor;
   if Z ≠ B then
      generators(Z) = generators(Z) ∨ Loop;
      return false;              //False means multiple-entry loop
   endif;
   while Queue ≠ ∅  do
      take X from Queue;
      delete X from Queue;
      foreach P ∈ Pred(X) do
         if P ≠ Z then
            if P ∉ Loop then
               add P to Queue;
               add P to Loop;
            endif
         endif
      endfor;
   endwhile;
   add Z to Loop;
   return true;                  //True means single-entry loop
endprocedure FIND_LOOP;
```

图 12: 图 3.12 Identifying a General Loop

### 4.7.3 3.7.3 计算循环树

编译器需要完整的循环集和循环之间的关系。此信息存储为树。循环 L1 是 L2 的孩子当且仅当 L1 是 L2 的子集并且不包含在 L2 中包含的任何其他循环中。用于计算循环的算法找到具有特定标头块的最大循环。这确保了两个循环要么不相交，要么一个包含在另一个循环中，这种条件允许将循环组织成一棵称为循环树的树。循环树中有四种节点：

1. 树的叶子是控制流图中的块。
2. 单入口循环是树中内部节点的一种形式。
3. 组织为单入口区域的多入口循环是内部节点的另一种形式。回想一下，多入口循环包括循环以及从循环返回到循环中所有块的公共支配者的所有树节点。
4. 树的根是代表整个流图的特殊节点。它不会是循环或块，因为控制流图包括两个块：没有前驱的入口和没有后继的出口。这些块不能参与循环并且不是单个块。

为了记录树结构，将属性添加到块和循环树中的其他节点：

LoopParent(X) 是一个属性，指示该节点是树中哪个节点的子节点。它还指示循环或块包含在哪个循环中。LoopParent(X) 也可以是根，指示此块或循环不包含在另一个循环中。根的 LoopParent 为 NIL。

LoopContains(X) 是由 X 表示的区域中的节点集合。对于一个块，它是 NIL。对于循环或根，它是树中 X 的子集的集合，与直接包含在该区域中的循环或块的集合相同。

LoopEntry(X) 是作为该区域入口的块。

这些属性允许循环树周围的自由时刻，完全了解哪些块和循环包含在其他块和循环中。

随着循环树的建立，每个循环都被识别并输入到树中。一旦它被输入到树中，它就会被作为一个单一的实体来处理。在构造过程中，它的内部结构不再被查看。算法 FIND_LOOP 被修改以处理树节点并被扩充为完整构造过程的一部分。为了形成这棵树，我们需要对算法进行两处修改：

1. 按后序考虑图中的块。由于深度优先搜索的结构，包含在另一个单入口循环中的单入口循环具有具有较小后序号的入口块。因此，通过后序访问块，内部循环在外部循环之前被识别。
2. 一旦确定，将每个循环当作一个单独的块来处理。这是通过为每个块或循环保留一个数据来完成的，该数据指示它包含在哪个块或循环中（如果有的话）。当一个人找到一个块时，使用这个数据向外扫描到包含这个块的最外面的识别循环。

编译器现在拥有完整的算法。在图 3.13 中，我们有 FIND_LOOP 的最终版本，它计算块，称为生成器，确定循环中的所有其他块。如果是单入口循环，则 FIND_LOOP 继续并使用 FIND_BODY 在循环树中构建节点。

FIND_BODY 通过从生成循环的块向后移动到头部来计算循环体中的节点集（参见图 3.14）。中间的所有块都在循环中。它在循环树中构建节点并填充所有属性。必须注意确保块和已经计算的循环之间的区别。循环头和前驱总是块。在将节点插入循环树之前，编译器必须找到已计算的最大封闭循环。这是由 LoopAncestor 完成的，如图 3.15 所示。

LoopAncestor 通过向上扫描 LoopParent 属性来查找包含当前循环或块的最外层已处理循环，直到找到具有空条目的节点。由于一旦识别出封闭循环，该属性就会由 FIND_BODY 更新为非空条目，因此该算法给出了最外层的现有循环。

```
procedure FIND_LOOP(B: block);
   Z = B;                                  //Entry block
   Loop = ∅;                               //Blocks in the loop
   foreach P ∈ Pred(B) do
      if (P,B) is a backedge then
         Z = CommonDominator(Z,P);         //Find header block
         if (P ∉ Loop) ∧ (P ≠ B) then
            add P to Loop;
         endif;
      endif
   endfor;
   if Z ≠ B then                           //Multiple-entry loop
      generators(Z) = generators(Z) ∨ Loop;
   else                                    //Single-entry loop
      call FIND_BODY(Loop,Z,single_entry_loop);
   endif;
endprocedure FIND_LOOP;
```

图 13: 图 3.13 Computing Generators of a Loop

最后，可以描述计算循环的主要过程（见图 3.16）。Calculate_Loop_Tree 首先执行深度优先搜索以计算每个节点和后边缘的后序数。该实现可以在计算算法的其余部分的同时执行这种深度优先遍历，只需在访问节点后将计算嵌入到递归深度优先搜索过程中。

首先 Calculate_Loop_Tree 初始化块的所有属性。这些可以在创建块时初始化；但是，为了完整起见，此处描述了该步骤。然后程序按后序访问块。如果生成器集非空，则该块是多入口循环的头部，因此构建该循环。然后该过程检查该块是否是单入口循环的头部。请注意，一个块可能是多入口循环和单入口循环的头部。在这种情况下，编译器构建两个循环的嵌套：多入口循环是最内层循环，单入口循环是外层循环。图 3.17 给出了我们常设示例的循环树。

## 4.8 3.8 实现整数集

在整个编译器中，需要整数集。我们已经看到了一个例子：在深度优先搜索期间访问的一组节点。有多种方法可以实现这些集合，具体取决于计算和使用它们的要求。

一种形式的集合由节点组成，其中构造算法保证我们不会尝试两次添加相同的节点，或者集合很小，因此通过集合的搜索时间很短。在这种情况下，集合可以实现为链表。插入包括将元素添加到列表的开头或结尾。删除包括从链表中删除元素，搜索包括扫描链表。这种形式的集合对于扫描集合中的所有元素是有效的，但对于插入或删除来说效率不高。

另一种方法是使用位向量来表示集合。为值域中的每个可能元素分配一个唯一的整数值，从 0 开始。然后将任何集合表示为长度为分配的最大数加 1 的位数组。这种技术提供了插入的有效实现（查找的索引位并设置它），删除（索引以找到该位并将其清除），并集，交集和搜索（索引以查找该位并检查它是否为 1）。如果集合不是稀疏的，那么这种方法在空间上非常有效。但是，扫描集合中的所有元素效率不高。不幸的是，扫描

```
procedure FIND_BODY(Generators: set of block; Head: block, kind: LoopKind)
   Loop = ∅;
   Queue = ∅;
   foreach B ∈ Generators do          //Copy generators to body and Queue
      L = LoopAncestor(B);            //But use outermost containing loop
      if L ∉ Loop then                //Only enter each loop once
         add L to Loop;
         add L to Queue;
      endif
   endfor;
   while Queue ≠ ∅ do                 //Work-list algorithm
      take B from Queue;              //Remove arbitrary entry
      delete B from Queue;
      foreach P ∈ Pred(LoopEntry(B)) do
         if P ≠ Head then             //Add in predecessors
            L = LoopAncestor(P);
            if L ∉ Loop then
               add L to Queue;
               add L to Loop;
            endif
         endif
      endfor;
   endwhile;
   add Head to Loop;
   X = new Loop Tree node of type kind;
   LoopContains(X) = Loop;            //Set of attributes of new node
   LoopEntry(X) = Z;
   LoopParent(X) = NIL;
   foreach B ∈ Loop do                //Update parents for components
      LoopParent(B) = X;
   endfor;
endprocedure FIND_LOOP;
```

图 14: 图 3.14 Computing the Body of a Loop

```
function LoopAncestor(B: block) returns LoopNode;
   while LoopParent(B) ≠ NIL do
      B = LoopParent(B);
   endwhile
   return B;
endfunction LoopAncestor;
```

图 15: 图 3.15 Finding the Outermost Processed Loop

```
procedure Calculate_Loop_Tree;
   perform a depth first search recording post-order and back edges;
   foreach B ∈ G do                    //Initialize block information
      LoopParent(B) = NIL;
      generators(B) = ∅;
      LoopEntry(B) = B;
      LoopContains(B) = B;
   endfor;
   foreach B ∈ G in post order do
      if generators(B) ≠ ∅ then
         call FIND_BODY(generators(B),B,multiple_entry_loop);
      endif;
      call FIND_LOOP(B);
   endfor;
   call FIND_BODY({Exit},Entry,loop_tree_root);
endprocedure Calculate_Loop_Tree;
```

图 16: 图 3.16 Computing the Complete Loop Tree

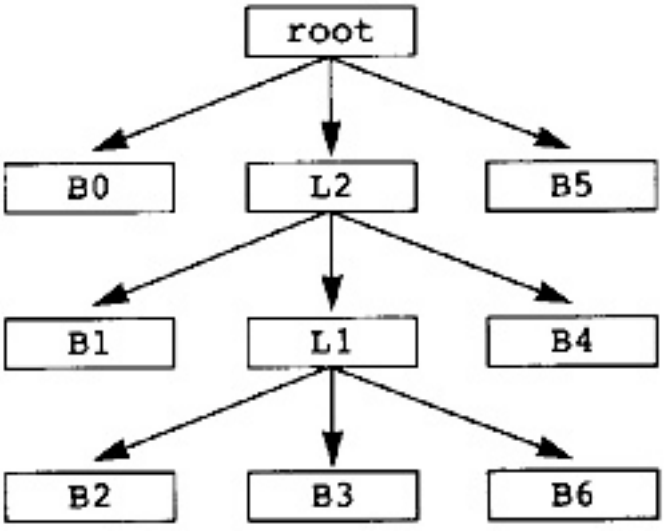


图 17: 图 3.17 Loop Tree for Example Program

是编译器中的常见活动。

Preston Briggs (1993) 基于 Aho、Hopcroft 和 Ullman (1974) 中的提示开发了另一种技术。这种技术在所有领域都非常有效操作；但是，它比位向量占用的空间多一个数量级，因此如果需要大量集合，则不希望使用它。

考虑我们的整数世界，编号从 0 到 MAX。分配具有初始 INDEX[0:MAX] 和 VALUE[0:MAX] 以及单个整数变量 NEXTPLACTE 的两个 MAX + 1 个元素的数组。

算法背后的想法（图 3.18）是集合的元素存储在 VALUE 中，从底部开始，并将它们堆积在相邻的槽中。当元素 X 添加到 VALUE 时，VALUE 中存储它的索引将放置在 INDEX(X) 中。否则 INDEX 的值不会被初始化。奇怪的是，该算法正在处理未初始化的数据。

算法如何知道值何时在集合中？它检查相应的 INDEX(X)。该信息可能未初始化，因此首先检查该值是否在范围内。如果不是，则该元素不在集合中。如果值在范围内，它仍然可以未初始化，因此它检查 VALUE 数组中的相应值。如果值匹配，则算法知道该元素在集合中。

从集合中删除一个元素有点棘手。该算法必须在恒定时间内运行，因此它不能删除一个元素并将其他元素向下移动。相反，它将集合中的最后一个元素向下移动到正在腾出的位置。同时它调整它的 INDEX 值并减少计数器 NEXTPLACE。

基本操作发生在 O(1) 时间内，扫描集合中的元素与集合中的实际元素成正比。不过，它确实需要更多空间。考虑一个元素由 16 位数字表示的实现。因此，每个元素有 32 位，表明这种表示法占用的空间是位向量方法的 32 倍。因此，当只需要少量集合（通常是一两个）时，这种表示就可以很好地工作。

## 4.9 3.9 参考文献

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974. The design and analysis of computer algorithms. Reading, MA: Addison-Wesley.

Briggs, P., and L. Torczon. 1993. An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems 2(1-4): 59-69.

Cytron, R., and J. Ferrante. 1987. An improved control dependence algorithm. (Technical Report RC 13291.) White Plains, NY: International Business Machines, Thomas J. Watson Research Center.

Cytron, R., J. Ferrante, and V. Sarkar. 1990. Compact representations for control dependence. Proceedings of the SIGPLAN ‘90 Symposium on Programming Language Design and Implementation. White Plains, NY. 241-255. In SIGPLAN Notices 25(6).

Cytron, R., J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4): 451-490.

Fischer, C. N., and R. J. LeBlanc, Jr. 1988. Crafting a compiler. Redwood City, CA: Benjamin/Cummings.

Lengauer, T., and R. E. Tarjan. 1979. A fast algorithm for finding dominators in a flow graph. Transactions on Programming Languages and Systems 1(1): 121-141.

Lorho, B. 1984. Methods and tools for compiler construction: An advanced course. Cambridge University Press.

```
procedure MAKE_SET_EMPTY;
   NEXTPLACE = 0;
endprocedure MAKE_SET_EMPTY;

function FIND_ELEMENT(X: int) returns boolean;
   if (INDEX(X)<NEXTPLACE)∧(INDEX(X)≥0) then
      if VALUE(INDEX(X)) = X then return true;
   endif;
   return false;
endfunction FIND_ELEMENT;

procedure INSERT_ELEMENT(X: int);
   if ¬FIND_ELEMENT(X) then
      NEXTPLACE = NEXTPLACE + 1;
      VALUE[NEXTPLACE] = X;
      INDEX[X] = NEXTPLACE;
   endif;
endprocedure INSERT_ELEMENT;

procedure DELETE_ELEMENT(X: int);
   if FIND_ELEMENT(X) then
      VALUE(INDEX(X)) = VALUE(NEXTPLACE);
      INDEX(VALUE(NEXTPLACE)) = INDEX(X);
      NEXTPLACE = NEXTPLACE - 1;
   endif;
endprocedure DELETE_ELEMENT;

Scanning elements in set performed by the macro
   I = 0
   while I < NEXTPLACE do
      X = VALUE(I);
      process X;
   endwhile;
```

图 18: 图 3.18 Efficient Set Algorithm

Purdom, P. W., and E. F. Moore. 1972. Immediate predominators in a directed graph. Communications of the ACM 8(1): 777-778.

Tarjan, R. E. 1975. Efficiency of a good but not linear set of union algorithm. Journal of ACM 22(2): 215-225.

CHAPTER 5

# 第 11 章限制资源

现在，编译器已经准备好为每条指令分配机器资源（例如寄存器、条件码等）。有若干种可用的资源分配算法，各自擅长处理特定类型的问题。当可用的物理寄存器比所需的物理寄存器数目大得多时，每种寄存器分配算法都表现良好。反之，则每种算法都表现糟糕。

这里给出的算法是多种分配算法的结合，尝试吸收它们各自的长处。它的组织结构是一个算法序列，每个子算法做一部分分配工作。先前以这种方式组织的算法受到 phase 次序问题的困扰：分配一个临时寄存器集，导致分配其它寄存器集变得更困难。编译器通过限制资源来缓减这个问题。

因此，第一件必须做的事情是减少所需寄存器的数目。在寄存器分配 phase 和指令调度 phase 之前做这件事情，之后它们就可以假设存在充足的可用物理寄存器。如图 11.1 所示，按照下面的次序执行这些功能：

1. 限制资源 phase 竭尽所能减少对机器资源的需求，同时不减小程序执行的速率。它做如下工作：
    a. 执行窥孔优化，生成确定的目标指令序列。
    b. 执行寄存器合并（coalesce）和寄存器重命名（rename），尽可能消除复制操作，将流图中多个独立的部分用到的每个临时变量替换为不同的临时变量。
    c. 减小寄存器压力，在程序的每个点限制所需寄存器的数目，以匹配可用物理寄存器的数目。
2. 指令调度 phase 重新组织目标指令，以减少执行流图所需的机器时钟周期数。与此同时，避免寄存器压力增加到超出可用寄存器集。
3. 寄存器分配 phase 为临时变量指派物理寄存器。这是分三步做到的。
    a. 首先，为跨 block 活跃的（全局）临时变量指派寄存器。
    b. 在一个 block 内，为可以和全局临时变量共享寄存器的临时变量指派寄存器。
    c. 在一个 block 内，为活跃且未分配寄存器的临时变量指派寄存器。

4. 再次执行指令调度 phase，当且仅当寄存器分配器插入了 load 和 store 操作。

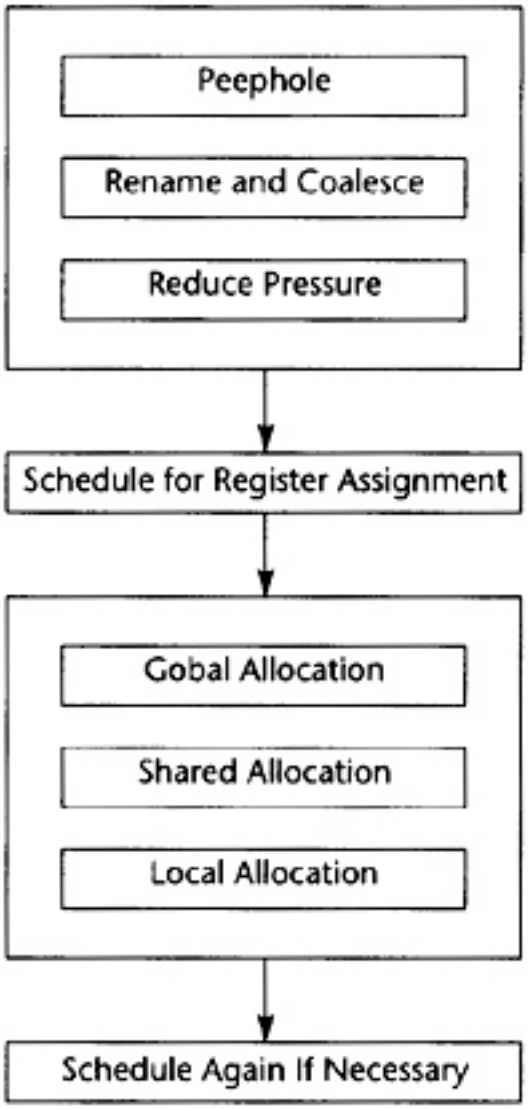


图 1: 图 11.1 Back-End Structure

寄存器分配 phase 必须高效地利用机器资源。这意味着使用尽量少的寄存器。当寄存器不够用的时候，寄存器分配器插入尽量少的 load 和 store 操作。使用最少的寄存器并且插入最少的 load 和 store 操作是不切实际的，因为这是一个 NP 完全问题。作为替代，我们会采用启发式方法尽可能地做好工作。

## 5.1 11.1 限制资源的设计

限制资源 phase 执行四个动作来减少指令数目和所用的寄存器。它们被分组为两个单独的子 phase。首先，执行窥孔优化、寄存器重命名（rename）、寄存器合并（coalesc）。然后，减小所需的寄存器数目（减小寄存器压力），这样在流图的任意点，所需的寄存器不多于机器上可用的寄存器。

如上面提到的那样，本编译器将窥孔优化、寄存器重命名、寄存器合并结合为一个单一的算法。为什么？如果单独实现它们，算法就需要占用大量空间和时间。考虑每个部分：

寄存器重命名涉及将单一临时变量分解为多个临时变量，假如它在流图的多个部分被独立地使用。这要求了解一个临时变量的哪些求值可能被使用。流图的静态单赋值（SSA）形式可以提供此信息。

寄存器合并利用称为冲突图（conflict graph）的数据结构删除寄存器到寄存器的复制。这个冲突图大概是编译器中最大的数据结构。我们观察到，很多重命名可以在流图的静态单赋值形式上进行，而不需要冲突图。只要为小部分临时变量构建冲突图，从而减小它的尺寸。当编译器可以查看每条指令的操作数的定义时，窥孔优化表现最好。静态单赋值形式给我们提供了此信息，因此可以在此时执行窥孔优化。肯定地，要在指令调度之前执行它。

作为一个清理 phase，必须删除死代码。再次地，这个算法在流图的静态单赋值形式上执行。

由于这些算法都在静态单赋值形式上运行，它们可以相继执行；但是，它们也可以被结合。在寄存器合并删除寄存器复制的时候，可以同时执行窥孔优化。不久我们会看到，寄存器重命名和寄存器合并可以结合为一个算法，它为了改造正常的流图而计算临时变量的一个划分。

这些基于静态单赋值形式的算法执行之后，算法处理正常的流图和循环结构，插入 load 和 store 操作，以减少所需的寄存器数目，匹配目标机器上可用的寄存器。如此，限制资源的主要过程具有如图 11.2 的形式。

如果流图中不存在非正常边，这些算法可以进一步结合。[1] 可以同时做窥孔优化、局部合并（coalesce）、构建静态单赋值形式。由于编译器必须避免非正常边的复制操作，必须在执行合并或窥孔优化之前找出这些边和相应的 φ 节点。可以在构建静态单赋值形式期间找出它们；但是，简单起见，我们单独描述它。

```
procedure LIMIT;
    call CALCULATE_SINGLE_ASSIGNMENT_FORM;
    call LIMIT_PROCESSING_FOR_ABNORMAL_EDGES;
    call PEEPHOLE_AND_LOCAL_COALESCE;
    call RENAME_AND_GLOBAL_COALESCE;
    call DEAD_CODE_ELIMINATION;
    call CALCULATE_NORMAL_FORM;
    call REDUCE_PRESSURE;
endprocedure LIMIT;
```

图 2: 图 11.2 LIMIT Main Procedure

[1] 现在，你应该明白了非正常边是编译器编写者存在的根源。

```
procedure LIMIT_PROCESSING_FOR_ABNORMAL_EDGES;
   Occurs_In_Abnormal_φ_node = ∅;
   Pairs_In_Abnormal_φ_node = ∅;
   foreach B ∈ N do
      if Φ(B) ≠ ∅ then
         for P ∈ PRED(B) do
            if (P,B) is abnormal edge then
               let P be the ith predecessor of B;
               foreach T0 = φ(...,Ti,...) ∈ Φ(B) do
                  add T0 to Occurs_In_Abnormal_φ_node;
                  add Ti to Occurs_In_Abnormal_φ_node;
                  add (T0,Ti) to Pairs_In_Abnormal_φ_node;
               endfor;
            endif;
         endfor;
      endif
   endfor;
endprocedure LIMIT_PROCESSING_FOR_ABNORMAL_EDGES;
```

图 3: 图 11.3 Effect of Abnormal Edges

必须删除所有非正常边上由 phi 节点导致的复制。因此，对于这些 phi 节点中的临时变量，什么也做不了。不改变临时变量求值或使用的点就可以做到。实际上，使用可以被删除，但不能被增加。为了找出这些临时变量，执行图 11.3 中的算法，它计算两个集合：Occurs_In_Abnormal_φ_node，这是这些边所包含的所有临时变量的集合；Pairs_In_AbnormaL_φ_node，这是临时变量对的集合，如果编译器不仔细就可能导致它们之间的复制。这个算法简单地查看所有包含任何 φ 节点的 block，检查它和每个前驱 block 之间的边是否为非正常边。如果是的话，就扫描 φ 节点的每个临时变量，将它们添加到这些集合。

## 5.2 11.2 窥孔优化和局部合并

窥孔优化是一种具有机器无关形式的目标机器相关的优化（见图 11.4）。不论一个优化编译器如何优秀，对生成的代码执行简单的模式匹配，都有改进的机会：

一个对 X 的 load 可能跟着一个对 X 的 store。编译器已经试着消除它们；但是，存在 phase 次序问题会限制这样的尝试。一个 load 操作所使用的地址可能是一个临时变量和一个常量的和。在目标机器上，如果这个常量是小的，就可以把它合并到 LOAD 指令的（地址）偏移中。STORE 指令也是如此。

目标机器可能有专用的指令，例如 Alpha 处理器有 S4ADDQ 指令。这条指令对一个操作数乘以 4，再加第二个操作数。它比乘法指令快，因为一个操作数的位被直接位移后写入目标寄存器。

局部寄存器合并在消除复制指令的时候，让使用其目标操作数的指令改为使用其源操作数，这是一种机器无关形式的窥孔优化。

编译器可以逐条扫描指令，象征性地执行它们，记住尽可能多的信息。当一个常量是 0 的时候，和这个常量的逻辑 AND 产生结果 0，各个位都是。如果后面的指令尝试修改或查询这些位，编译器就可以改变指令序列来生成更好的代码。

```
procedure PEEPHOLE_AND_LOCAL_COALESCE;
   change = false;
   initialize information for temporaries to nothing;
   repeat
      Occurs_in_Copy = ∅;
      call PEEPHOLE_BLOCK(Entry);
   until ¬change;
endprocedure PEEPHOLE_AND_LOCAL_COALESCE;
```

图 4: 图 11.4 Driver for Peephole Optimization

正常来说，窥孔优化按照执行顺序逐条扫描每个 block 中的指令。本编译器按照流图的支配者线路访问每个 block 和其中的指令。这意味着每个操作数在其它指令使用它们之前已经被求值。当然，对于 φ 节点是不成立的。对这种情况，编译器必须作最坏的假设：它不知道临时变量的值。对于其它临时变量，编译器记录如下信息：

- 如果临时变量容纳一个常量值，就记录这个常量值。

- 如果临时变量不是一个常量值，编译器知道这个临时变量某些位的信息吗？它是正数吗？已知某些位是 0 或者 1 吗？

编译器包含一系列在指令流中识别模式的函数：每种指令类型有一个函数。这些函数识别所有以特定指令结尾的模式，为生成更好的代码序列执行所需的转换。它还记录每个临时变量的信息，它们是那种类型指令的目标。如果指令序列改变了，就重新从转换后的序列的第一条指令开始模式匹配。因此，相同的指令可能应用多个模式。

尽管窥孔优化从第一条被转换的指令重新开始扫描使得它能够识别多个模式，有些模式还是不能被识别。重复整个窥孔优化 phase，直到没有模式被匹配。对于每个临时变量，上次迭代收集的信息还是有效的。φ 节点可以利用这些信息在后续迭代中获得更好的信息；然而，不应该仅仅为 φ 节点获得更好的信息而重复整个窥孔优化 phase，从中并不能获得足够的信息。

必须避免这样的转换：在 Occurs_In_Abnormal_φ_node 中增加一个临时变量的使用，或者移动这样的临时变量被求值的点。这样一来，编译器就保证后面不会在非正常边上生成复制操作。

图 11.5 中的算法描述了处理一个 block 的过程。在一个 block 中，按照执行的顺序去操作。首先，处理 φ 节点。只有少量转换可能消除 φ 节点；但是，可以从已知的关于操作数的信息获得关于结果的值的信息。

处理了 φ 节点之后，编译器模拟 block 的执行。在这个过程中，对于列表中的每条指令，调用窥孔优化函数。这个函数会执行任意转换。如果一个转换发生了，就返回值真。下面是窥孔优化的诀窍。如果没有发生转换，编译器会继续处理下一条指令。如果发生了转换，编译器会再次处理被转换的指令，现在这条指令可能不同于原始的指令。必须小心从事，避免跳过一条指令，试图再次处理删除的指令，或者发生通常的崩溃。

处理了这个 block 之后，继续遍历支配者树，处理支配者树中这个 block 的子节点。

这里，我们不会描述所有的函数，因为它们的数目和模式取决于目标机器。我们只描述对 φ 节点、复制指令、和整数乘法的处理。读者可以推演针对所有机器的结构。

为任意指令创建函数的时候，首先考虑可以应用的转换。对于 φ 节点，当它具有 T0 = φ(T1, ⋯, Tm) 的形式时，下面的转换是可能的：

- 如果 T1 到 Tm 都是相同的临时变量，这个 φ 节点可以被改写为单个复制操作，T0 = T1。如果所有这些临时变量都不涉及非正常边，就可以删除这个复制。
- 如果 T1 到 Tm 除了一个之外都是相同的临时变量，而那个临时变量和 T0 相同，那么这个 φ 节点也可以被改写为一个复制操作，并且可能被删除。

因此，处理 φ 节点，首先识别以上两种可能的情况，并作转换。然后，找出操作数共有的特征，把这些特征赋予目标操作数（见图 11.6）。

作为正常指令的一个例子，考虑整数乘法指令。窥孔优化怎么处理它？如果它是和一个常量的乘法，它已经被转为位移和加法操作。有必要再次检查一些简单的案例，以防万一它们被漏过或者产生于替换之后。[2] 图 11.7 给出了这个函数的片段。注意，这里并没有考虑专用指令，例如 Alpha 的 S4ADDQ。整数加法函数会考虑它，因为它是最后的操作。

这里，另一类要考虑的指令是 i2i，流图中的整数复制操作。这里只有一种转换。如果源和目标没有涉及非正

[2] 似乎总是发生这样的案例。编译器是精心设计的，使得特定指令的所有实例在编译器中的一个单一的位置被转换；但是，后面的转换可能生成同样的案例。因此，如果代价不高的话，应该去检查这样的案例没有发生。

```
procedure PEEPHOLE_BLOCK(B: Block);
   foreach T0 = φ(T1,...,Tm) ∈ Φ(B) do
      call PEEPHOLE_φ_NODE(T0 = φ(T1,...,Tm));
   endfor;
   while more instructions I in B do
      applied = false;
      case opcode(I) of
         ...
      iMUL:
         applied = PEEPHOLE_INTEGER_MUL(I);
      i2i:
         applied = PEEPHOLE_INTEGER_COPY(I);
      endcase;
      if applied > 0 then
         change = true;
         repeat processing the last applied instructions in B;
      else
         go on to next instruction in execution order;
      endif;
   endwhile;
   foreach C ∈ Children(B) do
      call PEEPHOLE_BLOCK(C);
   endfor;
endprocedure PEEPHOLE_BLOCK;
```

图 5: 图 11.5 Peephole Optimization of a Block

常边，使用目标临时变量的地方都可以替换为源临时变量，完全消除前者。图 11.8 对此作了解释。这个函数检查临时变量是否出现在非正常边上；如果不是，就修改所有使用目标临时变量的指令。

在为窥孔优化作扫描时，编译器预先计算出现在复制操作或 φ 节点中的临时变量的集合。之后只为这些临时变量计算冲突图，减少图的尺寸，提高编译速度。集合 Occurs_in_Copy 存放出现在复制操作或 φ 节点中的临时变量。注意，窥孔优化的每个 pass 会重新计算这个集合，因为对复制的处理可能会改变出现在复制操作中的临时变量的集合（图 11.8）。

```
procedure PEEPHOLE_φ_NODE(T0 = φ(T1,...,Tm));
   if T1 through Tm are all the same temporary T then
      delete the φ-node;
      insert copy T0 = T1 at beginning of block;
      change = true;
   else if T1 through Tm are same temporary T except one T' then
      if T0 = T' then
         delete the φ-node;
         insert copy T0 = T at beginning of block;
         change = true;
      endif;
   endif;
   Occurs_in_Copy = Occurs_in_Copy ∪ {T0,T1,...,Tm};
   find information common to T1 through Tm;
   assign that information to T0;
endprocedure PEEPHOLE_φ_NODE;
```

图 6: 图 11.6 Peephole Optimizing φ-nodes

## 5.3 11.3 计算冲突图

寄存器重命名和寄存器合并算法需要一个称为冲突图（conflict graph）的数据结构。[3] 它表示两个临时变量在流图中一些共同的点具有不同的值。

**定义**

冲突图：给定一个临时变量的集合 R，R 的冲突图是由节点和边构成的无向图，其节点是 R 中的临时变量，在临时变量 T1, T2 ∈ R 之间连一条边，如果流图中存在任意的点 p 满足下面的条件：

- T1 和 T2 可能具有不同的值。
- T1 和 T2 在点 p 同时活跃。这意味着，有一条从对 T1 的求值到使用的路径包含点 p，并且有一条从对 T2 的求值到使用的路径包含点 p。注意，如果其中一个临时变量未初始化，就不需要边。

[3] 通常这个数据结构称为 interference graph，重用指令调度期间所构建的数据结构的名字。于是，我选择了卡内基梅隆大学的 PQCC 项目（Leverett et al. 1979）所采用的名字。

```
function PEEPHOLE_INTEGER_MUL(I) returns integer;
   let the instruction be T0 = T1 * T2;
   if T1 and T2 are both constants then
      calculate value V in the compiler for T1 * T2;
      replace instruction by iLDC V => T0;
      return 1;
   else if T1 is constant 1 then
      replace instruction by i2i T2 => T0
      return 1;
   else if T2 is constant 1 then
      replace instruction by i2i T1 = >T0
      return 1;
   else if T1 is a power of 2 then
      replace instruction by shift by log(T1);
      return 1;
   ...
   endif;
endfunction PEEPHOLE_INTEGER_MUL;
```

图 7: 图 11.7 Peephole Optimization for Integer Multiplication

```
function PEEPHOLE_INTEGER_COPY(I: Instruction) returns boolean;
   let I be the instruction T0 = T1;
   if T0 ∉ Occurs_In_Abnormal_φ_node then
      if T1 ∉ Occurs_In_Abnormal_φ_node then
         if T0 ≠ T1 then
            foreach J ∈ Uses(T0) do
               replace uses of T0 by uses of T1 in J;
            endfor;
         endif;
         delete instruction I;
         return 1;
      endif;
   endif;
   Occurs_in_Copy = Occurs_in_Copy ∪ {T0,T1};
   return ∅;
endfunction PEEPHOLE_INTEGER_COPY;
```

图 8: 图 11.8 Peephole Optimizing Copy Operations

怎么表示这个数据结构？文献上描述了两种表示方式，本编译器将它们合而为一。鉴于临时变量表示为小的整数，将冲突矩阵表示为一个对称的位矩阵，其中 C[i,j] 为真，当且仅当临时变量 Ti 和 Tj 冲突。这使得访问矩阵检查一个冲突非常快；但是，找出和一个临时变量冲突的所有临时变量比较慢。作为替代，一个冲突图可以这样表示，每个临时变量有一个列表，记录所有和它冲突的邻居临时变量。这使得找出和一个临时变量冲突的临时变量容易了；但是，查明一个特定冲突的存在是费时的。

不幸的是，算法必须执行这两种检查，因为在图的构建期间，它需要检查冲突的存在，而之后它需要知道和一个特定临时变量冲突的临时变量。有些冲突图的实现首先创建位矩阵表示，然后将它翻译为邻居列表。这种转换消耗大量时间。其它的实现同时持有两种数据结构，针对特定的操作，哪种数据结构用起来更有效率就用哪种。这让编译器消耗更多内存。

我们的编译器以两种方式优化冲突图的构建。首先，只为编译器预先确定的临时变量的一个子集构建冲突图。保持小的临时变量集合，就节省了时间和空间。其次，编译器将冲突图实现为一个结合的哈希表和表示冲突邻居的列表。哈希表和图表示共享数据结构，避免额外的内存消耗。

### 5.3.1 11.3.1 冲突矩阵的表示

本编译器结合了两种表示方式，使用一个哈希表和一个表示无向图的链表。为此，在表中将每条边表示为一个条目。三个不同的链表持有这个条目：

哈希表表示为一个链接的哈希表，因此条目中有一个字段，称为 hashnext，存储指向哈希表的这个链中的下一个条目的指针。

小编号临时变量的邻居保存在一个列表中。在冲突邻居列表中，针对小编号节点，字段 smallnext 代表指向下一个邻居的指针。

相应地，大编号临时变量的邻居保存在一个列表中。在冲突邻居列表中，针对大编号节点，字段 largenext 代表指向下一个邻居的指针。

对于冲突图来说，不存在值表示和自己冲突的临时变量；因此，一条边连接着严格的小编号临时变量和严格的大编号临时变量。

在条目中还有两个针对边的字段：

- 字段 smaller 记录小编号临时变量的数目。
- 字段 larger 记录大编号临时变量的数目。

注意，边不存储数据。对算法来说，边的存在是重要的事情。于是，边的数据结构的样子看起来如图 11.9 所示。

为了检查特定冲突的存在，编译器使用一个链接的哈希表，ConflictHash，它的尺寸大约是 HASHSIZE，它可以是 2 的幂，因为用了简单的哈希函数。设 Ti 是由整数 i 表示的临时变量，相应地，设 Tj 是由整数 j 表示的临时变量。由于我们对临时变量的频率和相互关系一无所知，哈希函数线性化相应对称位矩阵中的条目，并除以表的尺寸。换句话说，哈希函数生成一个索引，去索引哈希表中的一个链表。当然，根据 hashnext 向下扫描这个链表，直到找到匹配的 smaller 和 larger，表明找到了一条边。

```
Conflict(Ti, Tj) =
(if i < j then
    j(j - 1)/2 + i
 else
    i(i - 1)/2 + j) mod HASHSIZE
```

```
type ConflictEntry =
        record
            hashnext:       pointer to ConflictEntry;
            smallnext:      pointer to ConflictEntry;
            largenext:      pointer to ConflictEntry;
            smaller:        temporary;
            larger:         temporary;
        endrecord;
```

图 9: 图 11.9 Structure of a Conflict Entry

在插入边的时候，新的边被添加到链表的头部，因为局部性表明，一旦发生了一次插入，很可能很快会尝试相同的插入。

其它操作是找到一个临时变量所有的邻居。设 Ti 是整数 i 相应的临时变量。利用一个类似图 11.10 的算法，向下扫描和 Ti 冲突的临时变量的列表。

编译器还会记录一个临时变量的邻居数目。为此，给临时变量增加一个属性，称为 NumNeighbors，它初始化为 0，并且每次添加一个冲突就加 1。

```
p = Neighbors(Ti);
while p ≠ NULL do
   there is conflict between smaller(p) and larger(p);
   if i < larger(p) then
      p = smallnext(p);
   else
      p = largenext(p);
   endif;
endwhile;
```

图 10: 图 11.10 Schema for Referencing Neighbors of Ti

### 5.3.2 11.3.2 构建冲突图

定义给出了计算冲突图的基本技术。考虑流图中的每个点。如果两个临时变量在某个点是活跃的，并且不知道它们是否具有相同的值，就在它们之间生成一条边。这意味着，编译器需要知道每个点活跃的临时变量的集合。在活跃或死亡分析之后，编译器只知道在每个 block 末尾活跃的临时变量。找出 block 内部任意点的活跃临时变量的方法是，向后扫描 block，应用活跃变量定义来更新其集合，如下概述的那样：

1. 向后扫描指令，首先将作为当前指令的目标的临时变量标记为死亡。
2. 将作为操作数的临时变量标记为活跃。
3. 对于在一个特定的点活跃的 (T1, T2) 对，在冲突图中创建一条连接 T1 和 T2 的边。

这个方法是低效的，因为通常两个临时变量在大量的点是活跃的。算法会尝试在每个这样的点插入一个冲突。当然，编译器会发现这个冲突已经存在了，不会插入它。但是，尝试这些无用的插入会消耗大量时间。作为替代，我们会利用 Chaitin（1981）所作的观察去减少工作量。

观察（*Observation*）

考虑从入口点到 T1 和 T2 活跃的点 p 的任意路径。下面的条件之一是成立的：

1. 在路径上对 T2 求值的某条指令处 T1 是活跃的。
2. 在路径上对 T1 求值的某条指令处 T2 是活跃的。
3. 在路径上点 p 之前 T1 或 T2 没有被求值，则编译器可以忽略这个冲突。[4]

**证明（Proof）**

给定一个路径，沿着路径向着入口方向向后行走。在开始行走时，T1 和 T2 都是活跃的。当其中之一变为不活跃的第一条指令处停下来。下面是几种可能：

[4] 一个不存放值的临时变量可以和任何其它临时变量共享一个寄存器。我们可以将其它临时变量中的值赋给它，因为它有什么样的值是无所谓的。

没有指令变为不活跃。这种情况下，在 p 之前的路径上，没有指令对临时变量求值，因此它们都包含未初始化的数据，于是出现上面的第三种情况。

其中一个临时变量变为不活跃，因为它是一条指令的目标。由于我们在临时变量变为不活跃的第一条指令处停下来，另一个临时变量还是活跃的，因此这是前面两种情况的其中之一。

其中一个临时变量变为不活跃，因为从入口点到当前点的任意路径上，没有对这个临时变量求值。这种情况下，此路径没有对这个临时变量求值，因此这是第三种情况。

根据活跃和不活跃的定义，只会出现这些情况，因此我们证明了这个观察结论。

这个观察意味着，我们不必为在一个点活跃的每一对临时变量创建冲突。编译器只需要在它们之间创建冲突，就是在一个点被求值的临时变量和在这个点活跃的其它临时变量。这得出了图 11.11 中的算法。它按照活跃/死亡分析一样的方法，为节点中的临时变量计算生命期信息，然后利用这些信息和最后的观察将冲突添加到冲突图中。

<Figure 11.11 Computing a Partial Conflict Graph>

作为一个例子，考虑图 11.12 中的直线型代码片段。假设 T5 是代码结尾处唯一活跃的寄存器，T0 和 T2 是代码开始处活跃的寄存器。向后扫描指令，我们得到图中第二列列出的冲突，这些是由指令建立的冲突。

在编译器中有两个地方会用到这个算法。首先，寄存器重命名和寄存器合并算法会用到它。为了那个目的，它需要作如下描述的修改。之后，全局寄存器分配会按这里陈述的样子使用它。

在寄存器重命名和寄存器合并期间，编译器计算临时变量的一个划分：当流图被翻译回正常形式的时候，属于相同划分的两个临时变量将被赋予相同的名字。编译器需要两个划分之间的冲突的概念：两个划分是冲突的，如果存在任意的点，在那个点两个划分都有元素是活跃的，并且无法知道它们存放相同的值。话句话说，一个划分在它的元素活跃的点的交集上是活跃的。构建划分的冲突图的算法和临时变量的是一样的；然而，边是在 (FIND(T1), FIND(T2)) 之间构造的，而不是在 (T1, T2) 之间，其中划分是由 UNION/FIND 算法表示的。

```
iSLD    0(T0)    => T1      (T0,T1), (T2,T1)
iADD    T1,#1    => T3      (T0,T3), (T1,T3), (T2,T3)
iADD    T2,T1    => T4      (T0,T4)
iADD    T4,T0    => T5
```

图 11: 图 11.12 Example Conflict Graph

## 5.4 11.4 结合的寄存器重命名和寄存器合并

限制资源 phase 为寄存器重命名、窥孔优化和部分寄存器合并实现了一个结合的算法。结合是基于这样的观察的，就是这些算法都计算临时变量的一个划分，在翻译回正常形式期间使用这个划分。起初建立静态单赋值形式超出了寄存器重命名的要求；它指派太多的新寄存器名字，插入复制操作在它们之间复制值。寄存器重命名创建最小的划分，删除所有这些插入的复制操作。不是直接删除它们，而是将它和寄存器合并中的删除复制操作相结合。

### 5.4.1 11.4.1 寄存器重命名

寄存器重命名消除这样的情形，就是流图的不同部分使用了相同的临时变量来存放不同的值。静态单赋值形式为寄存器重命名提供了一个基础。回想，静态单赋值形式为值的每次定义生成一个新的临时变量名字。当翻译回正常形式时，这些名字被重新结合来消除由 φ 节点隐含的复制操作。回想，翻译回正常形式是由临时变量之间的关系控制的。在正常形式的流图中，两个相关的临时变量共享相同的名字。

在实现寄存器重命名的时候，构建消除所有来自 φ 节点的复制的最小关系。这个关系是一个条件的传递闭包，这个条件就是，两个临时变量是相关的，如果一个是 φ 节点的操作数，另一个是相同 φ 节点的目标。关系是这样实现的，就是利用 UNION/FIND 算法创建所有临时变量的一个划分。因此，算法包括翻译为最小的 SSA 形式，通过声明每个 φ 节点的操作数和目标是相关的来构建划分，还有翻译回正常的形式。

### 5.4.2 11.4.2 寄存器合并

寄存器合并删除尽可能多的复制操作。很多复制操作已经在窥孔优化期间被删除了，所有复制操作，除了 φ 节点隐含的和涉及非正常边上关联 φ 节点的临时变量的复制操作，都被它删除了。最大比例的复制操作是这样被删除的。对于剩余的复制操作，利用 Chaitin（1981）的观察删除它们：如果一个复制操作的源和目标不相冲突，那么它们可以结合为一个寄存器。一旦两个临时变量被结合了，此算法可再次应用于另一个复制操作。此观察创建了临时变量的一个划分：如果两个临时变量在寄存器合并期间被结合了，它们就属于相同的分组。

SSA 形式的寄存器重命名算法会在流图中生成非正常边关联的 φ 节点。当流图被翻译回正常形式时，必须不让这些 φ 节点生成复制操作。因此，算法必须避免删除那些会导致复制操作出现在非正常边上的复制操作。照常来说，非可能边是没关系的，因为反正其上的代码绝不会被执行。

此算法包括利用 SSA 形式消除大部分复制操作。初始地，这样划分临时变量，每个临时变量自身构成一个分组。然后，调查每个 φ 节点和复制指令。如果一个操作数和目标临时变量不相冲突，就把它们放入相同的分组。然后，将流图翻译回正常形式。

注意寄存器合并和寄存器重命名之间的相似性。它们都创建了一个划分，用来消除 φ 节点处的复制操作。

### 5.4.3 11.4.3 集成算法

集成寄存器重命名和寄存器合并是简单明了的。它们都建立临时变量的一个划分，为了重构流图的正常形式。寄存器合并建立最小的划分，寄存器重命名会无偿发生。

驱动程序如图 11.13 所示。流图已经是静态单赋值形式。首先，计算全局值编码，这样编译器就知道哪些临时变量可能具有相同的值：这用来计算冲突图。初始地，每个临时变量自身被放入划分的一个单独分组。然后，对于一对临时变量，如果它们出现在非正常边上的复制操作中，就合并它们的分组，这样就不会出现涉及它们的复制操作。我们已经约束了窥孔优化，因此这是合法的。

<Figure 11.13 Coalescing and Renaming>

现在利用 Chaitin 的观察合并划分集合，这和这样的重命名是一样的，就是重命名一个临时变量，让它和另一个临时变量一样。利用一个 UNION/FIND 算法实现划分，划分中分组的 FIND 用作临时变量代表。如果两个

临时变量不相冲突，就可以合并为一个。在这个点，编译器只关系合并那些作为复制操作或 φ 节点的源和目标的临时变量。之后在全局变量分配期间，会利用相同的观察来分配寄存器。

在研究 COALESCE_TEMPORARIES 的时候，我们会发现，当我们合并两个临时变量时，需要更新冲突图。然而，更新是保守的，是不精确的，因此重新计算冲突图并重复合并，直到没有更多的复制操作可消除。

图 11.14 中的 COALESCE_TEMPORARIES 遍历流图，检查所有复制操作。如上所述，存在两种形式的复制：来自中间表示的显式复制和 φ 节点中的隐式复制。鉴于一些复制的删除可能会阻碍另一些复制的删除，遍历流图的时候，首先处理执行最频繁的 block。如果不能通过统计或静态估计获得此信息，就先处理循环最里面的 block。这个信息也没有，就按任意次序遍历 block。

```
procedure COALESCE_TEMPORARIES;
   foreach B ∈ N in highest frequency of execution first do
      foreach I ∈ B do
         if I is a copy operation T0 = T1 then
            call CHECK_COALESCE(T0,T1);
         endif;
      endfor;
      foreach S ∈ SUCC(B) do
         let B be the ith predecessor of S;
         foreach T0 = φ(T1,...,Tm) ∈ Φ(S) do
            call CHECK_COALESCE(T0,Ti);
         endfor;
      endfor;
   endfor;
endprocedure COALESCE_TEMPORARIES;
```

图 12: 图 11.14 Walking the Graph and Checking Coalescing

最后，图 11.15 中的 CHECK_COALESCE 作真正的事情。分组的冲突信息存储为临时变量代表的冲突信息，因此首先找出临时变量代表。如果它们是相同的代表，那么临时变量已经被直接或间接地合并了。其次，检查它们是否冲突。如果是冲突的，就不做什么；否则，用 UNION 方法合并这两个分组，将原来分组的冲突信息的联合赋予新的临时变量代表。

UNION/FIND 算法正常的实现让 T0 或 T1 作为新的临时变量代表。？这样的话，其中一个循环可以省去。在这个 pass 中，一旦消除了一个复制操作，就标记发生改变了。如果余下没有复制操作了，算法也可以停止。

这项技术的优势是什么？如早前所述，局部合并消除大部分复制操作，而不使用冲突图。其次，全局值编码允许消除级联的复制，而不用重复创建冲突图。第三，算法只为那些有机会合并的临时变量计算冲突图。

有些其它目标架构要求一种隐含的合并。如果目标机器不是 RISC 处理器，那么它可能有这样的指令，指令结果被存放到一个操作数中。中间表示模仿了 RISC 处理器，寄存器分配器希望让尽可能多的目标和操作数之一相同。为此，用两条目标机器指令替换一条 RISC 指令：从一个操作数到目标的复制指令和具有相同目标和（隐含）操作数的目标机器指令。利用合并消除这个复制指令，也就是说，让操作数和目标为相同的临时变量。

```
procedure CHECK_COALESCE(T0,T1: Temporaries);
    T0' = FIND(T0);
    T1' = FIND(T1);
    if T0' ≠ T1' ∧ T0' does not conflict with T1' then
        call UNION(T0',T1');
        T = FIND(T0')
        foreach S ∈ Conflicts(T1') do
            add S to Conflicts(T);
        endfor;
        foreach S ∈ Conflicts(T0') do
            add S to Conflicts(T);
        endfor;
        change = true;
    endif;
endprocedure CHECK_COALESCE;
```

图 13: 图 11.15 Coalescing Two Temporaries

## 5.5 11.5 计算寄存器压力

编译器已经尽可能地减小了在用临时变量的数目。现在编译器需要决定每个临时变量在流图中什么地方被赋予寄存器。无论何时一个临时变量在使用中，它是在寄存器中；但是，在使用之间，它可能被挤出（spill）到临时内存位置。我们把所需寄存器数目的粗略估算称作寄存器压力（register pressure），所以编译器必须首先计算寄存器压力或者每个点活跃寄存器的数目。如果有多个寄存器集，例如不同的整数和浮点数寄存器，那么单独为每个寄存器集计算寄存器压力。

**定义**

*寄存器压力*：给定流图中的一个点 p，寄存器压力是在 p 处活跃临时变量的数目。如果有分开的寄存器集，那么每个集的寄存器压力是单独计算的。

通过计算每个 block 末尾活跃的临时变量集合，可以确定寄存器压力。这个集合的尺寸给出了 block 中最后一条指令后面的寄存器压力。然后，编译器向后遍历每个 block，追踪每个点哪些寄存器是活跃的。集合的尺寸就是寄存器压力。在每条指令处，编译器将执行下面的步骤：

1. 首先，对于一条指令，将存放其（结果）值的临时变量标记为不活跃，并将它移出活跃寄存器集合。如果这个临时变量在此之前是不活跃的，那么可以删除这条指令。
2. 接着，将作为指令操作数的临时变量标记为活跃。
3. 该指令之前的寄存器压力是处理该指令之后活跃寄存器集合的尺寸。记住，我们按照逆向执行顺序处

理指令。

除了每条指令处的寄存器压力，算法需要知道每个 block 和每个循环的最大寄存器压力。为此，编译器利用循环树（loop tree）。一次遍历这棵循环树就可以计算得到所有关于寄存器压力的信息，如图 11.16 所描述那样。

寄存器压力是循环树的一个综合属性。其中每个节点的寄存器压力，是其子节点的寄存器压力的最大值。因此，计算一个循环的寄存器压力，就是找出封闭的循环和 block 的最大寄存器压力，如图 11.17 所示。

```
procedure CALCULATE_PRESSURE;
   call CALCULATE_LIVE;
   call COMPUTE_LOOP_TREE;
   foreach L ∈ LoopContains(Root_of_Loop_Tree) do
      call CALCULATE_PRESSURE_LOOP(L);
   endfor;
endprocedure CALCULATE_PRESSURE;
```

图 14: 图 11.16 Finding Register Pressure In Flow Graph

```
procedure CALCULATE_PRESSURE_LOOP(L: Loop_Node);
   case NodeKind(L) of
   block:
      call CALCULATE_PRESSURE_BLOCK(L);
   single_entry_loop, multi_entry_loop;
      MaxPressure = 0;
      Occur_in_Loop = ∅;
      foreach S ∈ LoopContains(L) do
         call CALCULATE_PRESSURE_LOOP(S);
         MaxPressure = Max(MaxPressure,Pressure(S));
         Occur_in_Loop = Occur_in_Loop ∪ Occurs(S);
      endfor;
      Pressure(L) = MaxPressure;
      Occurs(L) = Occurs_in_Loop;
   endcase;
endprocedure CALCULATE_PRESSURE_LOOP;
```

图 15: 图 11.17 Finding Pressure in a Loop

计算一个 block 的寄存器压力如图 11.18 所示。这个结构模仿了活跃/死亡分析所采用的计算局部生命期信息的方法。按照逆向执行顺序扫描 block，按照向后的顺序执行每条指令。当发现一个定义时，其临时变量变为

不活跃；当发现一个使用时，其临时变量变为活跃，除非它已经是活跃的。寄存器压力是每对指令之间活跃寄存器的数目。

<Figure 11.18 Computing Pressure in a Block>

有些处理器，例如 INTEL i860，包含这样的指令，它们在使用操作数之前定义目标寄存器。在这种情况下，必须改变代码以符合硬件的要求。对于这些特定的指令，会按照向后执行顺序，首先引用其操作数，然后修改其目标。

## 5.6 11.6 减小寄存器压力

现在，编译器将通过减小流图中每个点的寄存器压力，使它不大于可用的物理寄存器数目，来简化寄存器分配问题。如果存在多个寄存器集，则单独处理每个集。编译器找出寄存器压力太大的点。它将一个临时变量在这个点之前存储到内存，又将它在这个点之后加载回来。每次使用临时变量，它必须在寄存器中。在存储它的 STORE 和加载它的 LOAD 指令之间，这个临时变量不再活跃，于是寄存器压力减小了。

归纳起来说，假设流图中点 p 处的寄存器压力太高了，一个临时变量 T 将被挤出（spill）到内存。必须指派一个内存位置 MEMORY(T) 存放 T 的值。然后，必须向程序添加指令在 T 和内存位置之间搬运数据。如果 T 在程序中点 p 处是活跃的，而编译器想在那个点再利用存放 T 的寄存器，那么

- 在 T 被求值处到 p 的每条路径上，插入 store 操作，将数据从 T 搬到 MEMORY(T)。
- 在 p 到可能把 T 用作操作数的任何指令处的每条路径上，插入 load 操作，将数据从 MEMORY(T) 搬到 T。

满足这些条件并不难。编译器可以在每条计算 T 值的指令之后插入一条 store 操作，在每条使用 T 值的指令之前插入一条 load 操作。问题在于，这会生成太多内存引用指令。在现代处理器上，内存引用是最昂贵的操作之一，因此编译器要减小这类指令的数目。这些指令还消耗指令缓存空间，进一步降低性能。

如果程序中存在一个点，其寄存器压力超过可用寄存器的数目，那么编译器会挤出（spill）一个临时变量以减小寄存器压力。[5] 因为编译器想要减小被执行的 load 和 store 操作的数目，它从程序中执行频度最高的点开始作临时变量挤出，尝试在执行频度较小的点插入 load 和 store 操作。为此，在函数中寄存器压力最大的点执行三个步骤：

1. 找出包含 p 的最大循环（最外层循环），在 p 处存在一些这样的临时变量 T，它们跨越循环是活跃的，在循环中没有被使用。T 中存放的值向下传递，穿过循环。在循环开始处，插入一个 store 操作，把 T 存储到 MEMORY(T)，在 T 活跃的每个循环出口处，插入一个 load 操作，从 MEMORY(T) 载入 T。尝试最大限度向函数入口处移动 store 操作，而不增加它们被执行的次数。尝试最大限度向函数出口处移动 load 操作，而不增加它们被执行的次数。这可能减小其它点的寄存器压力。
2. 如果找不到这样的循环和临时变量 T，就想办法处理寄存器压力太高的单个 block。找出一个这样的临时变量 T，它在整个 block 是活跃的，在 block 中没有被使用。如果 T 在 block 后面是活跃的，就在 block

[5] 存在这样的情形，寄存器压力不是所需寄存器数目的准确度量。有时候，需要更多的寄存器，由于复杂交织的寄存器使用模式。有些穿过流图的路径不会被执行，有的点存在未初始化临时变量，这些地方所需的寄存器可能较少。然而，通常寄存器压力非常接近所需的寄存器数目。

之前插入 store 操作，在 block 之后插入 load 操作。同样地，尝试向函数入口 block 移动 store 操作，向出口 block 移动 load 操作。

3. 如果以上方法都无法减小寄存器压力，就得在寄存器压力太高得 block 内部插入 load 和 store 操作。选择这样一个临时变量 T，它在点 p 处是活跃的，此点之后的大量指令没有使用它。在 T 的定义之后（或者在 block 的开始处，如果它在 block 内没有定义的话），插入一个 store 操作。在 T 的下一次使用之前（或者在 block 的末尾，如果它在 block 内没有被使用的话），插入一个 load 操作。如果 load 出现在 block 的开始处，就尝试最大限度让它远离函数入口，而不增加执行的频度。类似地，最大限度让 store 远离函数出口。

一旦编译器插入了 load 和 store 操作，它就利用部分冗余消除技术让 load 远离入口 block，让 store 远离出口 block。利用 EARLIEST 算法，尽可能远地移动这些操作。

记得寄存器分配是 NP-完全问题，因此不存在一个对所有情形都表现良好的算法。这意味着，实现者（和作者）必须拒绝太复杂的分配机制：过去的经验表明，它们给不了对等的回报。

这样做更有效率，就是为每个循环计算可以挤出（spill）的临时变量，然后扫描各个循环，从外层到内层，如果寄存器压力太高，就挤出（spill）临时变量。为流图中的每个循环和 block 计算一个属性，Through(L)。图 11.19 和 11.20 给出了算法。

函数 COMPUTE_THROUGH 开始递归遍历循环树。只有那些寄存器压力高的循环才需要它，因而不用为不太复杂的循环计算这个属性。这会节省一点时间。注意，对于包含其它循环的循环，这不成立。如果外层循环具有高的寄存器压力，那么即使内层循环不太复杂，也会计算它的寄存器压力。避免不需要的计算让事情变得太复杂。

函数 COMPUTE_THROUGH_LOOP 单独处理来自循环的 block。对于一个 block，一个临时变量在整个 block 是活跃的，在 block 中没有引用，当且仅当它在 block 的开始处是活跃的且没有引用。警告：如果一个临时变量在 block 的开头和末尾都是活跃的，则未必它在整个 block 是活跃的，因为它可能在 block 中变为不活跃，后来再变为活跃。当然，如果临时变量在 block 中没有引用，就不会发生这种情况。

```
procedure COMPUTE_THROUGH;
   call CALCULATE_PRESSURE;
   foreach L ∈ LoopContains(Root_of_Loop_Tree) do
      if Pressure(L) > Max_Physical_Registers then
         call COMPUTE_THROUGH_LOOP(L);
      endif;
   endfor;
endprocedure COMPUTE_THROUGH;
```

图 16: 图 11.19 Computing Transparent Temporaries

那些在整个循环中活跃而没有引用的临时变量的集合，是循环的每个组件的相应集合的交集。函数 COMPUTE_THROUGH_LOOP 计算这个交集。编译器只关注最外层循环，在其中一个临时变量是活跃的而没有引用，因此计算循环的 Through 集合之后，它删除内层循环中对这些临时变量的引用。

```
procedure COMPUTE_THROUGH_LOOP(L: Loop_Node);
   case NodeKind(L) of
   block:
      Through(L) = LiveIn(L) - Occurs(B);
   single_entry_loop, multi_entry_loop:
      Through(L) = Temporaries;
      foreach S ∈ LoopContains(L) do
         call COMPUTE_THROUGH_LOOP(S);
         Through(L) = Through(L) ∩ Through(S);
      endfor;
      foreach S ∈ LoopContains(L) do
         Through(S) = Through(S) - Through(L);
      endfor;
   endcase;
endprocedure COMPUTE_THROUGH_LOOP;
```

图 17: 图 11.20 Main Through Calculation

对于单入口循环，计算 Through 属性有更简单的办法。对于单入口循环，一个临时变量在整个循环中活跃而没有引用，当且仅当它在入口 block 的开头是活跃的且在循环中没有引用。这是成立的，因为在循环中从一个 block 到其它 block 都有一条路径。对于多入口循环，这是不成立的，因为编译器在循环的开始处添加了若干 block，创建流图的单入口区域。在这些添加的 block 中，从一个 block 到其它 block 不存在路径。

## 5.7 11.7 计算寄存器 Spill 点

算法是这样被描述的，编译器找到一个寄存器压力太高的点，又找到一个跨越循环占用寄存器的临时变量，挤出（spill）这个临时变量。向下遍历循环树是一个更简单的方法。考虑每个循环的寄存器压力。如果压力太高，就挤出一个这样的临时变量，它在整个循环是活跃的，在循环中没有被引用。一直这样做，直到寄存器压力降低了。

这可能是无效率的，因为算法在一个循环中选择一个临时变量并挤出它，在另一个循环中选择不同的临时变量并挤出它；这样，可能会在两个循环之间插入大量的 load 和 store 操作，尽管可以为两个循环挤出一组临时变量，避免在它们之间插入 load 和 store。编译器尝试这样避免这个问题，就是基于一个循环的所有子循环选择挤出的临时变量。这只是一个启发式方法，因为选择挤出哪些临时变量，获得最优的方案，是一个 NP-完全问题。

在图 11.21 中，算法以驱动函数开始，它只计算寄存器压力和包含如此临时变量的 Through 集合，这些临时变量在每个循环中是活跃的，但是没有被引用。然后这个函数开始遍历循环树。当函数遇到压力小于寄存器数目的 block 或节点时，停止遍历。最终，它为指令调度重新计算压力。

这个算法有两个基本的函数：一个减小循环中的压力（见图 11.22），另一个利用一个不同的算法减小 block 中的压力（稍后在小节 11.7.1 中描述）。我们已经讨论了在循环中减小压力的算法。减小 block 内的压力是最后的措施，只有不存在如此临时变量时才会被执行，它们在整个 block 活跃并且没有被使用。

现在，我们来讨论减小循环中的压力，如图 11.22 描述的那样。算法的描述比实际想法更胆怯。计算循环或 block 的集合，High_Pressure，它们内部的寄存器压力太高。编译器要挤出一个在这些循环中都活跃的临时变量，如果可能的话。到最后，计算一个优先级队列，Excess_Pressure，由 High_Pressure 所包含的循环或 block 组成。优先级由寄存器压力的超额给定。算法选择一个待挤出的临时变量（很快会描述），然后挤出它（很快也会描述）。当在循环中挤出了尽可能多的临时变量，如果必要的话，才在子循环和 block 中挤出临时变量。

```
procedure REDUCE_PRESSURE;
   call COMPUTE_THROUGH;
   if Pressure(Root_of_Loop_Tree) > Max_Physical_Registers then
      call REDUCE_PRESSURE_LOOP(Root_of_Loop_Tree);
   endif;
   call CALCULATE_PRESSURE;
endprocedure REDUCE_PRESSURE;
```

图 18: 图 11.21 Driver for Reducing the Pressure

怎么选择待挤出的临时变量呢？考虑图 11.23 中的算法。选择压力超额最多的循环（或 block）。这个循环的 Through 集合中的每个临时变量都是挤出候选者。被选的临时变量也是大多数其它需要挤出临时变量的循环

```
procedure REDUCE_PRESSURE_LOOP(L: Loop_Node);
   case NodeKind(L) of
   block:
      call REDUCE_PRESSURE_BLOCK(L);
   otherwise:
      Exceed_Pressure = ∅;
      High_Pressure
         = {C ∈ LoopContains(L)|Pressure(C) > Max_Physical_Registers};
      foreach C ∈ High_Pressure do
         Excess_Temporaries = Pressure(C) - Max_Physical_Registers;
         if Excess_Temporaries > 0 ∧ Through(C) ≠ ∅ then
            add C to Exceed_Pressure priority Excess_Temporaries;
         endif;
      endfor;
         while Exceed_Pressure ≠ ∅ do
            choose C from Exceed_Pressure with highest priority;
            T = CHOOSE_SPILL;
            call PERFORM_SPILL;
            call OPTIMIZE_SPILL_PLACEMENT(T);
         endwhile;
         foreach C ∈ High_Pressure do
            if Pressure(C) > Max_Physical_Registers then
               call REDUCE_PRESSURE_LOOP(C);
            endif;
         endfor;
      endcase;
   endif;
endprocedure REDUCE_PRESSURE_LOOP;
```

图 19: 图 11.22 Spilling Temporaries in a Loop

的挤出候选者。这让优化安置 load 和 store 操作的算法获得最大的机会来避免一些 load 和 store 操作。

图 11.24 中的算法描述了如何插入 load 和 store 操作。首先，必须有一个内存位置存放这个值。所有对相同临时变量的引用必须使用相同的内存位置。在循环入口之前插入 store 操作，如果临时变量在循环出口点仍然活跃，就在那里插入 load 操作。循环内部没有引用这个临时变量，这保证了新程序和原始程序具有完全相同的计算效果。然后更新数据结构。如果循环不再有超限的寄存器压力，就把它移出 Excess_Pressure 和 High_Pressure。如果它仍然有超限的寄存器压力，那么优先级减一。

```
function CHOOSE_SPILL returns Temporary;
   Spill_Candidates = Through(C);   //Possible temporaries to spill
   foreach T ∈ Spill_Candidates do  //Can spill elsewhere?
      Quiet_Regions(T) = {C};
      foreach C' ∈ Exceed_Pressure do
         if T ∈ Through(C') then
            add C' to Quiet_Regions(T);
         endif;
      endfor;
   endfor;
   choose T such that |Quiet_Regions(T)| largest;
   return T;
endfunction CHOOSE_SPILL;
```

图 20: 图 11.23 Choosing which Loop Temporary to Spill

更新寄存器压力是代价最高的动作，因此编译器采用近似的办法减小预先选择的循环和 block 的压力。优化安置 load 和 store 操作的算法可能在其它地方减小压力。然而，在一个循环内的很多必要的地方挤出相同的临时变量，让近似方法表现更好。所有这样的循环和 block 确实挤出了临时变量，调整了压力，原来它们的压力是高的，并且有临时变量可以被挤出。那些压力不太高的循环或者 block，其压力得不到调整。因此，算法会遍历循环树，所记录的压力减一。当到达压力低的叶子或循环时，就停下来。在图 11.25 中，算法被描述为简单地向下遍历这棵树，修正属性 Pressure 的值。

### 5.7.1 11.7.1 减小 block 的压力

在单 pass 寄存器分配中，经典的 spilling 算法被用来在 block 中挤出（spill）临时变量。按照执行顺序扫描整个 block。当到达寄存器压力太高的点时，选择一个这样的临时变量，它在这个点是活跃的（因此压力将会减小），并且将来它下次被用作一个指令的操作数的位置是最远的。挤出这个临时变量将最大化 block 的指令序列，那里的压力减小了。为了选择这个临时变量，如果一个活跃的临时变量在这个 block 中不再被使用，就假设它在 block 末尾后面有虚假的使用。

这个算法实现为两个 pass。第一个 pass 向后扫描整个 block，组建一列这样的指令，它们使用了出现在这个 block 中的每个临时变量，然后计算每条指令之前的寄存器压力（图 11.26）。它模仿我们之前用来计算活跃/死亡信息和寄存器压力的代码。注意，寄存器合并和寄存器重命名保证了在 block 中一个临时变量只有一次求值。因此，这列指令的开始时一个临时变量变为活跃的第一个点。

第二个 pass 向前扫描整个 block（图 11.27）。经过每条指令的时候，将它移出之前组建的列表，使得列表总

```
procedure PERFORM_SPILL(T: Temporary);
   if T does not have associated spill location MEMORY(T) then
      allocate a spill location MEMORY(T) for T
   endif;
   foreach C' ∈ Quiet_Regions(T) do //Spill outside these regions
      add STORE T,MEMORY(T) on entry edge before LoopEntry(C');
      for each block B ∉ C' where B has predecessor P ∈ C' do
         if T is live on entry to B then
            add LOAD MEMORY(T) => T on edge (P,B);
         endif;
      endfor;
      remove T from Through(C');
      if Through(C') = ∅ then
         remove C' from Exceed_Pressure;
      endif;
      call UPDATE_PRESSURE(C');
      Excess_Temporaries = Pressure(C') - Max_Physical_Registers;
      if Excess_Temporaries > 0 then
         change priority of C' to Excess_Temporaries in
               Exceed_Pressure;
      else
         remove C' from Excess_Pressure;
         remove C' from High_Pressure;
      endif;
   endfor;
endprocedure PERFORM_SPILL;
```

图 21: 图 11.24 Inserting Spilled Loads and Stores

```
procedure UPDATE_PRESSURE(L: LoopNode);
   Pressure(L) = Pressure(L) - 1;
   if Pressure(L) > Max_Physical_Registers then
      foreach C ∈ LoopContains(L) do
         call UPDATE_PRESSURE(C);
      endfor;
   endif;
endprocedure UPDATE_PRESSURE;
```

图 22: 图 11.25 Updating Pressure

是存放 block 中余下的临时变量引用。在向前扫描的过程中，维护一个所有活跃临时变量的集合。当寄存器压力超过寄存器数目时，其中一个临时变量被存储到内存，在下次使用之处把它加载回来。为了在 block 的开始处跟踪活跃临时变量的集合，利用了起始 pass 计算的 Live 集合，随着编译器向前扫描整个 block，对临时变量执行反向动作。

**Figure 11.26 List of Uses for Reducing Pressure**

```
procedure REDUCE_PRESSURE_FIND_USES(B: Block);
   Live = LiveOut(B);
   foreach T ∈ Live do
      create list Reduce_Uses(T) containing end_of_block;
   endfor;
   foreach I ∈ B in reverse execution order do
      foreach T ∈ Targets(I) do
         delete T from Live;
      endfor;
      foreach T ∈ Operands(I) do
         if T ∉ Live then
            create empty list Reduce_Uses(T);
         endif;
         add T to Live;
         add I to head of the list Reduce_Uses(T);
      endfor;
   endfor;
endprocedure REDUCE_PRESSURE_FIND_USES;
```

图 23: 图 11.26 List of Uses for Reducing Pressure

应该存储哪个临时变量呢？那个将来下一次使用位置最远的临时变量。换句话说，扫描活跃临时变量集合，选择其使用列表的下一个条目最近的一个临时变量。这是单 pass 寄存器分配器使用的经典启发式方法，它尽可能让一个寄存器保持长时间可用。

在指令中实际的寄存器压力太高的点，是在使用操作数（这可能减小寄存器压力）和目标写入值（这可能增加寄存器压力）之间。如果压力太高，就在这条指令之前存储待挤出（spill）的临时变量（临时变量必须是该指令的一个操作数，或者是该指令没有用到而活跃的另一个临时变量）。下次使用之前，必须再次加载这个值。如果临时变量是活跃的，而 block 不再使用它，就在临时变量活跃的每个出口插入 load 操作，并调用优化安置挤出操作的算法。类似地，如果在 block 的开始处插入 load 操作，就必须调用优化算法来改善挤出操作的位置（见图 11.28）。

**Figure 11.27 Reducing Pressure in a Block**

```
procedure REDUCE_PRESSURE_BLOCK(B: Block);
   call REDUCE_PRESSURE_FIND_USES(B);
   if |Live| ≥ Max_Physical_Registers then
         call REDUCE_SPILL_LOCAL (begin_block)
   endif
   foreach I ∈ B in execution order do
      foreach T ∈ Operands(I) do
         remove I from Reduce_Uses(T);
         if Reduce_Uses(T) = ∅ then
            remove T from Live;
         endif;
      endfor;
      foreach T ∈ Targets(I) do
         while |Live| ≥ Max_Physical_Registers do
            call REDUCE_SPILL_LOCAL(I);
         endwhile;
         add T to Live;
         if Reduce_Uses(T) = ∅ then
            remove T from Live;
         endif;
      endfor;
   endfor;
endprocedure REDUCE_PRESSURE_BLOCK;
```

图 24: 图 11.27 Reducing Pressure in a Block

**Figure 11.28 Inserting a Spill within a Block**

```
procedure REDUCE_SPILL_LOCAL(I: Instruction);
   perform_optimization = false;
   Choice = {T ∈ Live| Reduce_Uses(T) most future instruction};
   if MEMORY(Choice) is not allocated then
      allocate memory location for MEMORY(Choice);
   endif;
   insert STORE Choice,MEMORY(Choice) before I;
   remove Choice from Live;
   if I is the first instruction in the block then
      perform_optimization = true;
   Next = Instruction in Reduce_Uses(Choice);
   if Next = end_of_block then
      insert LOAD_MEMORY(Choice) => Choice on exit edges;
      perform_optimization = true;
   else
      insert LOAD_MEMORY(Choice) => Choice before Next;
   endif;
   if perform_optimization then
      call OPTIMIZE_SPILL_PLACEMENT(Choice);
   endif;
endprocedure REDUCE_SPILL_LOCAL;
```

图 25: 图 11.28 Inserting a Spill within a Block

## 5.8 11.8 优化 Spill 指令的位置

一旦存储和载入操作的初始位置确定之后，编译器着手优化这些 STORE 和 LOAD 指令的位置，把它们移动到执行频度较低的点。移动它们的动作减小了所越过点的寄存器压力，让压力减小算法的后程变得容易。

为了优化安置这些存储和载入操作，编译器为每个挤出（spill）的临时变量建立下面的集合。需要在整个挤出过程和整个流图中维护这些集合，因为在流图的一个区域挤出一个临时变量，可能会改变流图其它部分的存储和载入操作的位置。

- STORE_IN(T) 是在 block 的开头有 STORE 指令的 block 的集合，指令将 T 存储到 MEMORY(T)。
- STORE_OUT(T) 是在 block 的末尾有 STORE 指令的 block 的集合，指令将 T 存储到 MEMORY(T)。
- LOAD_IN(T) 是在 block 开头有 LOAD 指令的 block 的集合，指令从 MEMORY(T) 载入 T。
- LOAD_OUT(T) 是在 block 末尾有 LOAD 指令的 block 的集合，指令从 MEMORY(T) 载入 T。

这节描述改善安置这些载入和存储操作的算法。一旦基于循环的算法确定了循环外部指令的位置，编译器就为这些载入和存储操作连同之前相同临时变量的载入和存储操作寻找更好的位置。所用的算法是部分冗余删除的 EARLIEST 算法。

### 5.8.1 11.8.1 优化 Store 操作

考虑将 T 挤出到 MEMORY(T) 的存储操作。这些指令只依赖 T，可视为一元（unary）操作。我们一旦定义了评估存储操作意味什么，定义了什么操作会杀死它们，就可以像任何其它指令那样优化它们。（注：杀死的含义是让它失效。）

什么指令会评估存储操作？它们是那些保证其执行之后内存中的值和 T 中的值是一样的指令。显然，编译器插入的存储操作满足这个条件。然而，从 MEMORY(T) 到 T 的载入操作也满足这个条件。因此，评估存储操作的指令包括存储和载入指令。

什么指令会杀死存储操作？它们是破坏如此条件的指令，这个条件是 T 中的值和 MEMORY(T) 中的值一样，它们是修改 T 的任何指令。注意，LOAD 指令首先杀死 T，然后产生评估存储操作的效果。

注意，T 被用作操作数不影响存储操作的安置。向入口 block 移动存储操作，它们永远不会改变 T 的值，因此一个存储操作可以越过 T 的使用，而不影响任何寄存器的值。这样我们得到了预期和可用的如下定义：

- STORE_ANTLOC(I) = STORE_IN(T)
- STORE_AVLOC(I) = STORE_OUT(T) ∪ LOAD_OUT(T)
- STORE_TRANSP(B) = {T | B 中的指令不修改 T}

现在，可以用这些集合计算 STORE_ANTIN、STORE_ANTOUT、STORE_AVIN 和 STORE_AVOUT。然后，可以用 EARLIEST 方程计算 STORE_EARLIEST。这指示了在哪些点插入新的 STORE 指令，在哪些点删除旧的 STORE 指令的实例。

相比表达式全局优化的情形，STORE 指令应该尽可能向远处移动。这可能减小流图其它部分的寄存器压力，避免否则发现不了的进一步挤出（spill）。因此，使用了 EARLIEST 算法，而不是 LATEST 算法。

STORE_EARLIEST 的计算利用了 EARLIEST 方程，将存储操作的预期集合和可用集合替换为这里描述的相应集合。插入和删除 STORE 指令的算法还用到了 EARLIEST 中描述插入和删除的方程。

需要进一步优化以减小流图中存储操作的数目。EARLIEST 方程可以描述在所有通向一个 block 的边上插入相同的计算。这时，应该在 block 的开头插入计算，而不是一条边上。还有，如果算法描述了在同一个 block 的开头插入又删除一个存储操作，就不要插入或删除。当不能移动一个存储操作时，EARLIEST 会发生这种情况：算法描述了在通向 block 的每条边上插入一个存储操作，又在 block 中删除它。

在这个算法中，非正常边遇到了和部分冗余相同的问题，而解决方法是一样的。如果算法试图在非正常边上插入一个存储操作，编译器就假装在边的开头有一条修改 T 的指令。这样，T 不是预期的，在边上不会发生插入。添加了这条指令之后，再次执行算法，计算在哪些点插入存储操作，得到这些点的新的集合。图 11.29 描述了完整的算法。

### 5.8.2 11.8.2 优化 LOAD 的位置

可用相同的技术移动载入操作，除了编译器需要向出口方向移动它们。我们在反向图上沿着前驱节点应用部分冗余算法，类似正常的 EARLIEST 算法沿着后继节点遍历正向图。

为此，编译器必须知道哪些指令评估一条 LOAD 指令，哪些指令杀死它。一条指令评估一条 LOAD 指令，如果它保证临时变量中的值和内存中的值相同。明显地，一个 LOAD 指令评估一个 LOAD 指令，一个 STORE 指令也评估一个 LOAD 指令。

哪些指令杀死一个 LOAD 指令？临时变量的一次使用或求值杀死一个 LOAD 指令。使用会杀死它，因为越过这个使用移动 LOAD 将破坏这个使用的值。临时变量的求值会杀死它，因为它会生成一个值，它不同于内存中的值。

到出口的一些路径可能不再使用临时变量 T，这时可以进一步优化。如果在插入 LOAD 指令的点 T 不活跃，就可以取消这次插入。

## 5.9 11.9 参考文献

Chaitin, G. J., et al. 1981. Register allocation via coloring. Computer Languages 6(1):47-57.

Leverett, B. W., et al. 1979. An overview of the Production-Quality Compiler-Compiler project. (Technical Report CMU-CS-79-105.) Pittsburgh, PA: Carnegie Mellon University.

**Figure 11.29 Inserting and Deleting Spilled STOREs**

```
procedure MOVE_STORES(T: Temporary);
   call CALCULATE_STORE_LOCAL(T);
   call CALCULATE_STORE_ANTICIPATION(T);
   call CALCULATE_STORE_AVAILABLE(T);
   foreach B ∈ N do
      on_all_edges = ∩ {STORE_EARLIEST_P,B | P ∈ Pred(B)};
      if on_all_edges then
         if B ∉ STORE_IN(T) then
            insert STORE T,MEMORY(T) at beginning of B;
            add B to STORE_IN(T);
         endif;
      else
         foreach P ∈ Pred(B) do
            if STORE_EARLIEST_P,B then
               if |Succ(P)| = 1 then
                  insert STORE T,MEMORY(T) at end of P;
                  add P to STORE_OUT(T);
               else
                  Recompute placement of STOREs assuming
                  that there is an instruction at beginning of
                  B that kills the STORE T,MEMORY(T).
               endif
            endif;
         endfor;
         if B ∈ STORE_IN(T) then
            delete STORE T,MEMORY(T) at beginning of B;
         endif
      endif;
   endfor;
endprocedure MOVE_STORES;
```

图 26: 图 11.29 Inserting and Deleting Spilled STOREs

CHAPTER 6

# 第 12 章指令调度和再次调度

精简指令集计算（RISC）处理器采用简单的指令（这样保持硬件简单）来提高执行速率，还采用若干其它器件，来增加在固定时间间隔内可以执行的指令数目。

处理器是流水线化的。这意味着，单个指令在执行的时候被分解为若干个大约尺寸相同的小任务。一条指令的单个任务在一个装配线（称为流水线）上执行。流水线化提高了装配线的效率。当第一条指令完成第一个阶段时，第二条指令就可以开始执行了，或者被发出。在每个流水线上，每个指令周期可以发出一条指令。但是，一条指令发出之后，经过一到几个周期，也许没有完成求值。如果后面的指令试图使用前面指令计算的值，而前面的指令还没有完成所有流水线阶段，那么处理器会停顿，或者延迟发出第二条指令，直到第一条指令已经结束。为了提高性能，编译器会重排指令，以避免处理器停顿。[1]

处理器在相同时间会发出多条指令。处理器会加载一小组指令，称为 packet，分析这些指令之间的关系。如果指令不使用或修改小组中其它指令的操作数，那么可能在相同时间发出这些指令。这提高了性能，如果处理器有多个计算单元，例如整数算术单元和浮动算术单元。

处理器可能有多个整数单元和多个浮点数单元。这时，预取指令的 packet 容量更大，可能同时发出多条算术指令。不是所有算术单元是相同的。这时，编译器会重排指令，这样每个 packet 会有指令可以在不同的算术单元上执行。

具有这三个特征的处理器称为超标量处理器。今天大多数在用的处理器是超标量的。很多处理器有一个额外的特性，称为乱序（out-of-order）执行。这样的处理器会像上面描述的那样工作，而且当 packet 中前面的指令由于受限而不能执行时，允许执行 packet 中后面的指令。这里我们不讨论这个特性，因为编译器几乎不能做什么事情来加强处理器乱序执行，对于普通的超标量处理器来说，这是不重要的。

[1] 当一个值还没有准备好时，早期的 RISC 处理器不会停顿。相反地，它们以垃圾数据为输入执行指令。编译器负责保证这样的执行不会发生。近期的处理器在等待操作数时都会停顿，因为有些指令不能确定等待的时间，特别是 LOAD、乘法和除法指令，让调度变得困难。

Digital Alpha 21164 处理器是超标量处理器的一个例子。考虑它是怎么匹配上述标准的。首先，Alpha 不是一个乱序执行的处理器。所有指令都是按照次序执行的；如果一条指令被延迟执行，跟在它后面的所有指令也会被延迟执行。

Alpha 是管线化的。大多数整数算术单元的指令需要一个时钟周期；浮动指令需要四个时钟周期。以上法则的一些例外是条件赋值、LOAD、乘法和浮点除法指令。Alpha 会尝试在每个时钟周期发出四条四字节的指令。指令的 block 必须对齐到一个 16 倍数的地址。如果地址不是 16 倍数，那么获取包含当前指令的 packet，会忽略 packet 中开头的指令，这样减少了这个时钟周期内可以发出的指令数目。如果 packet 中的指令具有依赖关系，它们不能全部发出，就会发出 packet 开头部分的指令，直到不能被立即发出的第一条指令。

Alpha 包含两个整数算术单元和两个浮点算术单元。两个整数算术单元可以执行大多数整数指令。例外如下，一个单元做移位操作，另一个单元做跳转操作。有一个浮点乘法单元和一个浮点加法单元。有些指令被两个单元共享。

理想地，Alpha 在一个时钟周期发出四条指令。其中两条指令由整数算术单元执行。另外两条指令，其中一条由浮点乘法单元执行，最后一条指令由浮点加法单元执行。这些指令在 packet 中出现的次序是任意的。

Alpha 还有别的特征，调度器必须予以考虑。考虑 load 操作。它们由整数算术单元执行；但是，从内存获取数据所需的时间取决于当前哪个高速缓存或内存单元包含此数据。

Alpha 包含三个片上高速缓存：一个数据缓存，一个指令缓存，一个更大的辅助缓存。基本缓存容纳 8K 数据，组织为直接内存映射缓存，每个 cache line 的长度是 32 字节。如果数据在这个高速缓存中，一个 load 操作需要两个周期。指令缓存也是 8K 的片上缓存；但是，这里不讨论它。

辅助片上缓存存放指令和数据。它存放 96K 信息，组织为三路组相联高速缓存，每个 cache line 包含 64 字节数据。如果数据在这个高速缓存中，一个 load 操作需要九个周期，包括把数据搬运到基本缓存。

在 Alpha 系统上，通常有一个大的主板高速缓存。这个缓存容纳若干兆数据，组织为直接内存映射缓存，每个 cache line 容纳 64 字节数据。如果数据在这个缓存中，一个 load 操作需要二十个周期，包括把数据搬运到两个更高层级的缓存。

如果数据在内存中而不在缓存中，load 操作就需要更长的时间：有时在一百个周期范围内，取决于涉及的系统。这个时间太长了，使得调度器模拟其准确的时间没有意义。编译器可以用两种方法处理它。编译器可以乐观地假设数据在其中一个缓存中，针对这种情形调度指令，或者编译器意识到这些 load 操作会消耗大量时间，尝试尽量把它们移动到早前的位置。

程序执行管线化的效果是这样的，程序中的指令在某个点发出，经过一定数量的时钟周期才得到结果。由于硬件优化，经过多少周期而获得结果值，可能取决于它怎么被使用，因此延迟（delay/latency）是计算一个值的指令和使用这个值的指令的函数。

指令调度 phase 重排被编译函数中的指令，消除尽可能多的停顿。有三种不同类型的调度器，根据它们尝试在多大的函数区域重排指令：

Block 调度器在单个 block 内重排指令。程序流图的形式不会改变。一个 block 重排指令，和另一个 block 重排指令不相关，可能的例外是在 block 末尾计算的值（或在 block 开头使用的值）。

Trace 调度器在一条简单路径上的 block 内重排指令。选择程序中执行最频繁的路径重排指令。指令可能从一个 block 移动到另一个 block。事实上，指令可以移动到所计算的值保证未被使用的地方（推测执行）。重排这些更大的指令序列，可以发现更多机会来消除停顿。

软件流水线器重排并复制循环中的指令以消除停顿。软件流水线的结果是一个新的循环，原始循环中的多次迭代被合并成一次迭代同时计算。

对于超标量处理器来说，block 调度是不够的。如果机器在每个时钟周期可以发出四条指令，那么一个周期的延迟意味着没法执行四次潜在的计算。编译器必须想办法把计算指令移动到一起，这样它们可以同时执行。大多数 block 是小的，编译器必须组合来自多个 block 或一个循环的多次迭代的计算指令。换句话说，编译器必须执行某种形式的 trace 调度和软件流水线。

到此刻为止，我们忽略了执行指令所需的时间和处理器发出指令的方式。指令调度 phase 重排指令以避免停顿。图 12.1 的 Alpha 指令序列计算表达式 A = (B + C) + D * E，如果按照源语言所描述的次序执行指令，就会浪费两个周期。初始的 load 操作需要两个周期，从数据缓存中获取数据。可以利用这些周期执行后面的 load 操作，允许 load 和后续的乘法和加法重叠。

```
0     ldt     f0,B              0     ldt     f0,B
0     ldt     f1,C              0     ldt     f1,C
2     addt    f0,f1,f2          1     ldt     f3,D
2     ldt     f3,D              1     ldt     f4,E
3     ldt     f4,E              2     addt    f0,f1,f2
5     mult    f3,f4,f5          3     mult    f3,f4,f5
9     addt    f2,f5,f6          7     addt    f2,f5,f6
13    stt     f6,A              11    stt     f6,A
```

图 1: 图 12.1 Instructions Before (left) and After (right) Scheduling

图 12.1 间接显示出三个顾虑。首先，原本的指令序列只用三个寄存器就可以执行。重排后的序列需要四个寄存器。重排指令可能增加所需寄存器的数目，让寄存器分配更困难。也有这样的情形，指令调度减小了所需寄存器的数目；但是，这很少见。总的来说，指令调度让寄存器分配更困难。

第二个问题是，当寄存器分配无法为所有临时变量分配物理寄存器时，会发生什么。寄存器分配器会插入 store 和 load 操作，把数据写到内存或者从内存读取数据。这些指令破坏了原始的指令调度，在寄存器分配之后可能需要重复指令调度。在第二次指令调度时，大多数临时变量已经赋予了物理寄存器，因此指令可移动的范围变小了。

第三个问题在上面的例子中是隐晦的。Alpha 处理器同时可以发出四条指令。在这个例子中，可发出的指令总是不超过两条。很多周期没有指令可以发出，这样浪费了很多启动指令的机会（周期）。编译器怎样变形程序才能获得更多可发出指令？我们已经讨论了一种方法：循环展开。如果图 12.1 的代码是一个循环中的数组引用，那么循环的四次迭代可以同时执行，利用很多浪费的时钟周期。

## 6.1 12.1 指令调度 phase 的结构

编译器按组调度 block：每个小组是支配者树的一条路径。注意，这是扩展的 block 的一般化。每个扩展的 block 是支配者树的子树，扩展的 block 构成的路径是支配者树的一条路径。一种 trace 调度方法在扩展的 block 上调度指令，因此这是那种技术的一般化。在调度 block 的时候，依次执行下面的操作：

1. 创建支配者树。支配者树是全局调度的基础数据结构。编译器用它找出可统一调度的路径。在调度指令的时候，它还会对指令执行值编码算法，在将被调度的路径上消除无用的计算。
2. 找出所有单 block 循环。对这些循环执行软件流水线。这涉及创建若干个 block，以存放开始循环的指令、结束循环的指令和循环的副本，仅当执行的迭代很少时存放循环的副本。把这些 block 插入到支配者树中，原来的 block 标记为已调度单 block 路径的一个成员。
3. 为了执行指令调度，编译器需要知道在 block 和它支配的 block 之间有哪些临时变量被使用或定义。它们被称为 IDEES 和 IUSE 集合（不久我们会讨论如何计算它们）。编译器在开始调度之前计算它们。
4. 计算贯穿支配者树的路径或者 trace，其上的 block 将被统一调度。有两种截然不同的组成 trace 的准则。如果执行频率是已知的，那么首先选择执行频率最高的路径，然后选择执行频率较小的路径。如果两个 block 具有相似的执行频率，那么更有可能与其前驱或后继合并的 block 被添加到路径中。
5. 现在对支配者树执行深度优先搜索。当到达开始一个 trace 的 block 时，调度这个 trace。随着编译器遍历支配者树，它执行值编码算法，以找出重复的指令。重复的指令可以被删除，不需要调度。调度分成两个部分：计算指令之间的冲突图（interference graph），然后调整指令次序，让尽可能多的指令重叠执行。

和其它 trace 调度的方法不同的是，这个算法处理流图中的临时变量。后面会分配寄存器。在寄存器分配之后执行指令调度，每个 trace 是一个 block，因此指令调度只发生在一个 block 之内。这时，只有那些插入了新指令的 block 需要再次调度。

## 6.2 12.2 Phase 次序

调度器执行两次。它跟在限制资源 phase 之后，所以我们知道，在程序中任意的点，有足够的寄存器存放所有的值。它在寄存器分配之前，所以还存在临时变量。它重排程序中的指令，生成相同程序的一个正确表示。在寄存器分配之后，调度器可能被再次调用，如果程序中插入了挤出（spill）操作。注意，窥孔优化在调度前执行。这样，指令调度 phase 附近的 phase 执行序列如图 12.2 所示。

指令调度发生在临时变量被绑定物理寄存器之前，前后移动指令的自由度相对更大。这可能会增加程序中每个点的活跃寄存器的数目。因此，必须约束调度器，在程序中的每个点，不让所需寄存器的数目增加到超过可用寄存器的数目。如果这对指令调度器限制得太多，我们将改变限制资源 phase，进一步减小寄存器压力。当我们测试真实程序时，可以实验性地做这件事。

指令调度可以为窥孔优化创造机会。对于访问相同位置的 load 和 store 操作，它可以移动它们，让它们相邻。因此，当调度器调度指令的时候，它必须准备好作一些形式受限的窥孔优化。执行寄存器分配之后，可以再次调用指令调度器，如果寄存器分配器生成了新的指令。如果分配寄存器的时候没有发生寄存器挤出（spilling），就没有必要执行第二次指令调度。

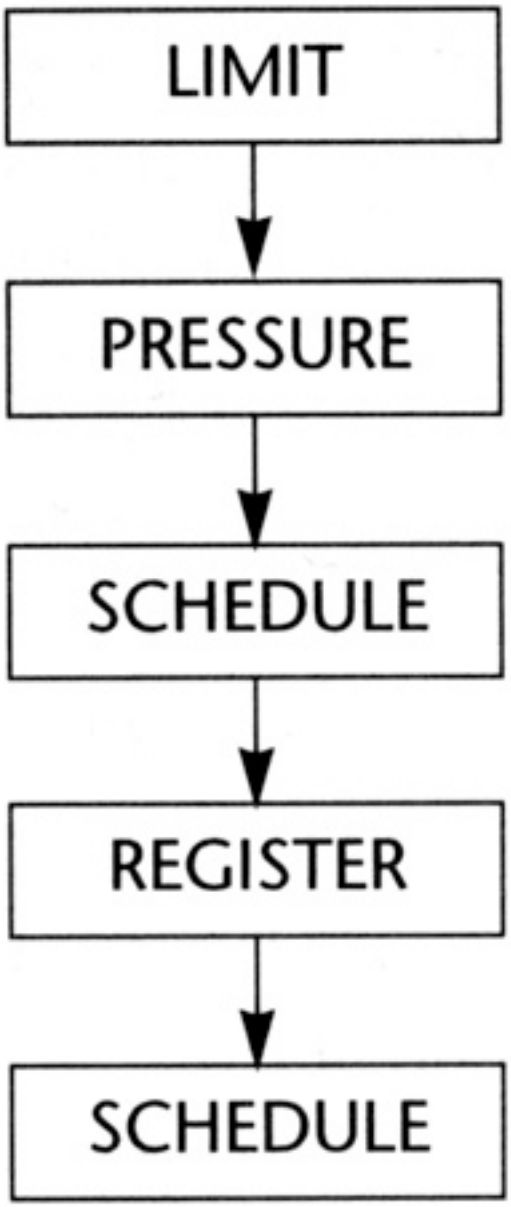


图 2: 图 12.2 Sequence of Phases Involving Scheduling

## 6.3 12.3 例子

这里给出两个例子，来说明指令调度。首先，图 12.3 是可运行例子的内层循环。我们会调度这个循环体，提高一些性能，即使循环包含的指令不多。这是真实程序中很多循环的典型情况。

图 12.4 给出了相应的调度后的流图片段。循环中的所有 store 操作被移走了，superblock[2] 转换把指令复制到了循环的末尾，以改善指令调度。

图 12.5 的例子有两个用途。编译器会软件流水线化这个循环，重叠多次迭代的执行。除了展示软件流水线，我们会用这个例子说明编译器将如何编译一个未作软件流水线化的循环。这样的循环可能会被展开，为了增加可以调度的指令。

```
DO J = 2, N
   IF (DABS(A(J,I) .GT. VALUE(I))
      VALUE(I) = DABS(A(J,I))
      LARGE(I) = J
   ENDIF
ENDDO
```

图 3: 图 12.3 Inner Loop of Example

```
12 B2:   dSLD     (T28)       => SF4   /value A(J,I)
13       CPYS     SF4         => SF4   /DABS(A(J,I))
14       dCMPLE   SF4,SF1     => SF6   /comparison
15       dBCOND   SF6,B3,B6            /skip update of variables

16 B6:   d2d      SF4         => SF1   /update value of VALUE(I)
17       i2i      T6          => T14   /put in register for LARGE(I)
18       BR       B3

19 B3:   iADD     T6,#1       => T6    /J + 1
20       iADD     T28,#8      => T28   /update value of address(A(J,I))
21       iCMPGT   T6,T8       => T24   /is J > N
22       iBCOND   T24,B4,B2
```

图 4: 图 12.4 Instructions in the inner Loop

图 12.6 给出了循环被软件流水线化时前面的编译器 phase 产生的指令。循环体包含一次循环迭代的指令。图 12.7 给出了假设循环不会被软件流水线化时所生成的指令。循环按照四次被展开，于是有些计算可以重叠。在这个例子中，编译器可能实际上按照四次以上展开循环；但是，作为一个例子，展开得更多没有意义。

在描述调度算法本身之前，我们来讨论五个话题，它们构成了调度的基础：

[2] Superblock 调度在附录 B 中讨论。

```
DO I = 1,20
   A(I) = B(I) + C(I)
ENDDO
```

图 5: 图 12.5 Vectorizable Loop

```
B2: dSLD     (T1)       => SF1    /LOAD B(I)
    dSLD     (T2)       => SF2    /LOAD C(I)
    dADD     SF1,SF2    => SF3    /B(I) + C(I)
    dSST     (T3),SF3             /A(I) = B(I) + C(I)
    iADD     T1,+8      => T1     /increment address of B(I)
    iADD     T2,#8      => T2     /increment address of C(I)
    iADD     T3,#8      => T3     /increment address of A(I)
    iADD     T4,#1      => T4     /increment loop count
    iCMPGT   T4,T5      => T6     /check end of loop
    iBCOND   T6,B2,B3
```

图 6: 图 12.6 Instructions for Vectorizable Loop

编译器不是在单个 block 中调度指令，而是在一组 block 中调度指令，这组 block 称为 trace。首先编译器必须计算 trace。然后调度 trace 中的指令，就像这些指令来自单个 block。

如同一会你将看到的，trace 不必是流图中顺序相邻的 block。当它们不相邻时，编译器必须计算这样的临时变量，它们在 trace 中的多个 block 之间被使用或定义。

当 trace 和 block 间的信息是已知的时候，编译器会计算一种称为冲突图（interference graph）的数据结构，它描述哪些指令必须在其它指令之前求值，必须提前多长时间发出这些优先的指令。

就在指令被调度之前，编译器必须为每条指令估算从它开始执行到 trace 的末尾需要多少个时钟周期。这被称为关键路径信息，调度指令时会根据该信息选择指令。

在调度指令的时候，编译器模拟指令的执行，在每个执行周期跟踪记录处理器的哪些功能单元是忙碌的。跟踪的方法是维护一组状态信息，在每个周期更新它。预先计算功能单元所能达到的所有状态，将此表达为一个有限状态机，这样做更高效。更新状态，然后规约到一个状态转移。

我们将依次讨论这些话题，然后在结束的时候给出调度算法。

```
B2:   dSLD     (T1)          => SF1    /LOAD B(I)
      dSLD     (T2)          => SF2    /LOAD C(I)
      dADD     SF1,SF2       => SF3    /B(I) + C(I)
      dSST     (T3), SF3               /A(I) = B(I) + C(I)
      dSLD     8(T1)         => SF4    /LOAD B(I+1)
      dSLD     8(T2)         => SF5    /LOAD C(I+1)
      dADD     SF4,SF5       => SF6    /B(I+1) + C(I+1)
      dSST     8(T3),SF6               /A(I+1) = B(I+1) + C(I+1)
      dSLD     16(T1)        => SF7    /LOAD B(I+2)
      dSLD     16(T2)        => SF8    /LOAD C(I+2)
      dADD     SF7,SF8       => SF9    /B(I+2) + C(I+2)
      dSST     16(T3),SF9              /A(I+2) = B(I+2) + C(I+2)
      dSLD     24(T1)        => SF10   /LOAD B(I+3)
      dSLD     24(T2)        => SF11   /LOAD C(I+3)
      dADD     SF10,SF11     => SF12   /B(I+3) + C(I+3)
      dSST     24(T3),SF12             /A(I+3) = B(I+3) + C(I+3)
      iADD     T1,#32        => T1     /increment address of B(I)
      iADD     T2,#32        => T2     /increment address of C(I)
      iADD     T3,#32        => T3     /increment address of A(I)
      iADD     T4,#4         => T4     /increment loop count
      iCMPGT   T4,T5         => T6     /check end of loop
      iBCOND   T6,B2,B3
B3:
```

图 7: 图 12.7 Unrolled Loop

## 6.4 12.4 计算 trace

同时调度多个 block 的想法以 trace 调度的形式流行开来，trace 调度是 Fisher（1981）提出的。他注意到，大多数程序的一部分 block 比其它 block 更频繁地被执行。如果我们选择一个这样的 block 然后扩展它，添加它前面的 block 和后面的 block，形成一条 block 的路径，那么我们可以一起调度这些 block 中的所有指令。当然，编译器必须插入指令以修复跳转指令的效果，包括跳入和跳出这条路径。

Trace 调度表现良好，但是它有一个严重的弱点。这些为了修复跳入和跳出 trace 而插入的指令称为补偿代码，它们的数量可以很大，它们本身可能没有经过良好的调度。因此，具有单个主导 trace 的流图会得到良好的调度（大部分时间消耗在单个 trace 中）。但是，如果流图有多个重要的 trace，或者找不出单个主导 trace，调度的效果就没有那么好，因为补偿代码会让程序变慢。

Freudenberger, Gross 和 Lowney（1994）注意到，如果选择这样一个 trace，不存在从 trace 外的 block 到 trace 的跳转分支，trace 中的 block 是 trace 中唯一的另一个 block 的后继节点，那么可以消除大部分补偿代码。消除了大部分补偿代码，这个算法给出的性能和通用的 trace 算法几乎一样好。这些 trace 的另一个名字是扩展的 block（extended block）。

**定义**

*扩展的 block*：

流图中的扩展的 block 是满足下列条件的一组 block：

在扩展的 block 中存在单个 block B0，它在扩展的 block 中没有前驱节点。它所有的前驱节点出现在扩展的 block 之外。

在扩展的 block 中，每个除 B0 之外的 block B 都有单个前驱节点，这个前驱节点是扩展的 block 的成员。

换句话说，扩展的 block 是流图中的一棵 block 的树。Lowney 建议，trace 取扩展的 block 中的一条路径。

本调度器是基于这种想法的一般化，Sweany 和 Beaty（1992）提出了这种想法，后来 Huber（1995）改进了它。Sweany 选择支配者树中的路径作为 trace。一个 trace 由一个 block 序列组成，其中每个 block 是下一个 block 的直接支配者。然后，调度这个 trace，就像调度一个 block 的指令。在这个 trace 中向前或向后移动指令，这样有些指令可能被移动到执行频度较小的点，或者被移动到其执行时间可以被隐藏的位置。

将 Sweany 的准则应用于扩展的 block。扩展的 block 中的每个 block 要么是入口 block，要么在扩展的 block 中受它的前驱节点支配。然而，Sweany 的 trace 定义允许其它可能的 trace。考虑程序中的一个结构化的 if 语句。如果两个可选的分支具有几乎相等的频度，那么构建这样一个 trace 可能更好，它由开头的分支语句和末尾的汇合语句组成。

这种指令移动和优化器中的代码移动有何不同？优化器移动指令是受限的，它不能把指令移动到任意远的地方：它仅仅把指令移动到一个稍后总是将被使用的点，它不能把指令移动到这样一个点，在那里执行频度可能会增加，计算和使用之间的指令序列被最小化。指令调度器不会受到这样的限制。它可以把指令移动到一个不保证被使用的点，只要指令没有代价，寄存器压力不过量。

### 定义

*Trace*:

一个 trace 是这样的 block 序列 B1, B2, ⋯, Bn，其中对于每个 1 < i <= n，block Bi-1 直接支配 Bi。就是说，一个 trace 是支配者树中的一条路径。

编译器会把流图划分成一个个分离的 trace。第一个被构造的 trace 应该代表执行最频繁的 block，按某种方式被展开以改善这些 block 的执行。下一个最重要的 trace 从余下的 block 中构造，依此类推。编译器使用怎样的标准来选择 trace 的 block 呢？它需要考虑下面的因素：

这个 trace 应该包含还不属于任何 trace 的执行最频繁的 block B。选择是基于频度信息的。有三种方法可收集此信息：统计，静态频度信息估计（如 Ball 和 Larus（1992）的方法），和最内层循环粗略估计（考虑最内层循环执行频度最高，分支具有相等的可能）。这个 block B 被称为 trace 的锚点，因为 trace 完全由这个元素的选择决定。我们很快会看到，锚点不是 trace 的入口点。

考虑 B 的后继节点 S，B 是它们的唯一前驱节点，它们不属于别的 trace。选择 S 中执行频度最高的节点。必然地，这个执行频度小于 B 的执行频度。将 S 纳入 trace，对 S 的后继节点递归重复这个过程。这个过程的效果是将从 B 开始的扩展的 block 中执行最频繁的路径纳入 trace。

再考虑锚点 B 的直接支配节点 D。如果它还不属于一个 trace，也不嵌套于跟 B 不相关的一个循环，就把 D 纳入 trace。由于 B 具有最高频度，D 的频度不会高于 B；但是，它可能嵌入于一个不包含 B 的循环。这时不要添加 D。对 D 的直接支配节点重复这个过程，以此类推。

如果在 B 处没有后继节点可用于扩展 trace，支配者树中 B 的一个孩子节点也是 B 的后支配节点，并且它的执行频率和 B 一样，就把这个后支配节点也包含进来。

如果 trace 超过（实验决定的）一个固定的尺寸，就终止它。这个尺寸应该按照指令数量统计。有些调度算法不与 trace 的尺寸成线性，因此避免生成太长的 trace。反过来，一个长的 trace 已经存在显著数量的指令重叠，因此再增加 trace 的尺寸，几乎不会带来益处。

给定这些条件，计算 trace 的算法是简单明了的，如图 12.8 所示。构造一个按执行频率排序的 block 优先级队列。利用这个队列找出 trace 的锚点，然后依照上面提到的规则扩展它。向后扫描，包含支配者节点，直到必须停止 trace。这给出了入口点。现在从锚点开始向前扫描，包含扩展的 block 的一条路径，或者一个后支配者节点。这些规则是灵活的。trace 的最优选择取决于用户的编程风格和源语言的最优编程风格，因此准备好修改此代码，以满足这些需求。

编译器需要一种命名 trace 的方法。编译器把 trace 的入口 block 用作名字。每个 block 有一个属性 trace(B)，它要么是 NULL，由于 block 还未插入到一个 trace，要么是 trace 的入口 block。有了这个属性，就能轻松找出 trace 中的所有 block。trace 由一组 block 组成，它们构成支配者树中的从 trace 入口 block 开始的一条路径。简单地向下扫描这棵树，查看每个孩子节点。如果一个孩子节点的属性值和 trace 相同，那么 trace 包含这个孩子。如果没有孩子节点的属性值和它的父亲节点相同，那么 trace 终止了。

注意，我们用双竖线，|B|，表示 B 中的指令数量。这种表示法是有道理的，因为在数学中双竖线用于表示基数。

图 12.9 给出了将锚点的支配者添加到 trace 的决定过程。任意支配者（编译器必须在树的根停下来），如果它们不在 trace 中，就添加它们。如果 trace 太长了，就终止它。编译器还要检查支配者是否在一个循环中，而

```
procedure CALCULATE_TRACES;
   Queue = ∅;
   foreach B ∈ N do
      trace(B) = NULL;
      add B to Queue with priority frequency(B);
   endfor;
   while Queue ≠ ∅ do
      take B from Queue;                    //The anchor
      NumberInstructions = |B|;
      C = B;                                //This will become entry
      D = IDOM(C);
      while CAN_ADD_DOMINATOR(D,C) do
         NumberInstructions = NumberInstructions + |D|;
         C = D;
         D = IDOM(C);
      endwhile;
      D = B;                                //Now have entry C
      while D ≠ C do                        //Include blocks in trace
         trace(D) = C;
         remove D from Queue;
         if D = C then break;
         endif;
         D = IDOM(D);
      endwhile;
      D = CAN_ADD_SUCCESSOR(B);
      if D ≠ NULL then
         repeat
            trace(D) = C;
            NumberInstructions = NumberInstructions + |D|;
            D = CAN_ADD_SUCCESSOR(D);
         until D = NULL;
      else if NumberInstructions < MaxInstructions then
         P = PDOM(B);                       //Postdominator
         if P ≠ NULL ∧ (P is child of B) ∧ frequency(B) = frequency(D) then
            trace(P) = C;
            NumberInstructions = NumberInstructions + |P|;
         endif;
      endif;
   endwhile;
endprocedure CALCULATE_TRACES;
```

图 8: 图 12.8 Calculating Traces

这个循环不直接或间接包含锚点。支配者位于外层循环是适合的，而位于不直接或间接包含锚点的循环是不适合的。

```
function CAN_ADD_DOMINATOR(D: Block, B: Block) returns boolean;
   if D = NULL then
      return false;
   endif;
   if trace(D) ≠ NULL ∨ NumberInstructions ≥ MaxInstructions then
      return false;
   endif;
   L = LoopParent(D);    //Uses loop tree
   if L is ancestor of B in loop tree then
      return true;
   else
      return false;
   endif;
endfunction CAN_ADD_DOMINATOR;
```

图 9: 图 12.9 Determining Whether Dominators Can Be Added to a Trace

图 12.10 的算法用于扩展 trace，从锚点开始扩展为扩展的 block。找到一个后继节点，它只有一个前驱节点。选择执行频率最高的后继节点，它就是下一个添加到 trace 的 block。

现在考虑我们在本书中一直使用的程序例子。我们使用流图，但是不构造超级 block。构造超级 block 能生成更好的 trace，那是将来讨论的话题。假设每个循环被执行 100 次，那么内层循环实际上被执行了近 10000 次。假设在每个循环中最大值改变了大约 10 次，因此 block B6 的执行次数是 1000 次（见表 12.1）。

编译器构造 block 的优先级队列，选择其中一个执行最频繁的 block。这里的选择不是唯一的。一种可能是会首先选择 block B3。然后扫描这个 block 的直接支配节点，得到第一个 trace {B0, B1, B2, B3}。下一个 trace 将是单个 block {B6}。然后 block {B4} 构成一个 trace，{B5} 是最后一个 trace。

<Table 12.1 Hypothetical Frequencies>

另一种可能是选择 block B2 作为锚点来构造第一个 trace。添加支配节点，添加扩展的 block 的后继节点。这给出了第一个 trace {B0, B1, B2, B6}。然后 {B3} 自身会构成一个 trace，{B4} 和 {B5} 也是。

## 6.5 12.5 预计算资源信息

此调度器处理 trace，它们是穿过支配者树的路径。在一个 block 和它的支配者之间，可能有多个 block。编译器必须知道哪些临时变量和内存位置在这些 block 中被使用或修改。

```
function CAN_ADD_SUCCESSOR(B: Block) returns Block;
   BestS = NULL;
   foreach S ∈ SUCC(B) do
      if trace(S) = NULL ∧ |PRED(S)| = 1 then
         if frequency(S) > frequency(BestS) then
            BestS = S;
         endif;
      endif;
   endfor;
   return BestS;
endfunction CAN_ADD_SUCCESSOR;
```

图 10: 图 12.10 Determining Whether a Successor Can Be Added to a Trace

### 6.5.1 12.5.1 定义和使用信息

调度器选择一个 block 序列 B1, B2, ···, BN，其中每个 block 是它的后继节点的直接支配者。然后，一起调度这些 block，可能将某个计算从一个 block 移动到前面的或者后面的 block。为此，编译器必须知道哪些临时变量在这两个 block 之间被修改或使用。这里所用的算法以 Reif 和 Lewis（1978）的算法为基础，Sweany 和 Beaty（1992）为指令调度改造了它。

**定义**

*OUT*:

对于每个 block B，OUT(B) 是执行 B 过程中被修改的临时变量的集合。

**定义**

*IDEFS*:

对于每个 block B，IDEFS(B) 是从 IDOM(B) 到 B 的某条路径上被定义的临时变量的集合。这不包括发生在 B 或 IDOM(B) 中的定义。

在图 12.11 中，IDEFS(B4) 包括 T2 和 T3，但是不包括 T1 和 T4。它包括 T2 和 T3，因为它们是在从 B1 到 B4 的路径上被定义的，而 B1 是 B4 的直接支配者。

除了定义，使用也存在类似的信息集合。思想是相同的，后面我们会看到的计算方法也是相同的。唯一不同的是，被检测的是作为操作数的临时变量和变量的使用，而不是指令的结果。

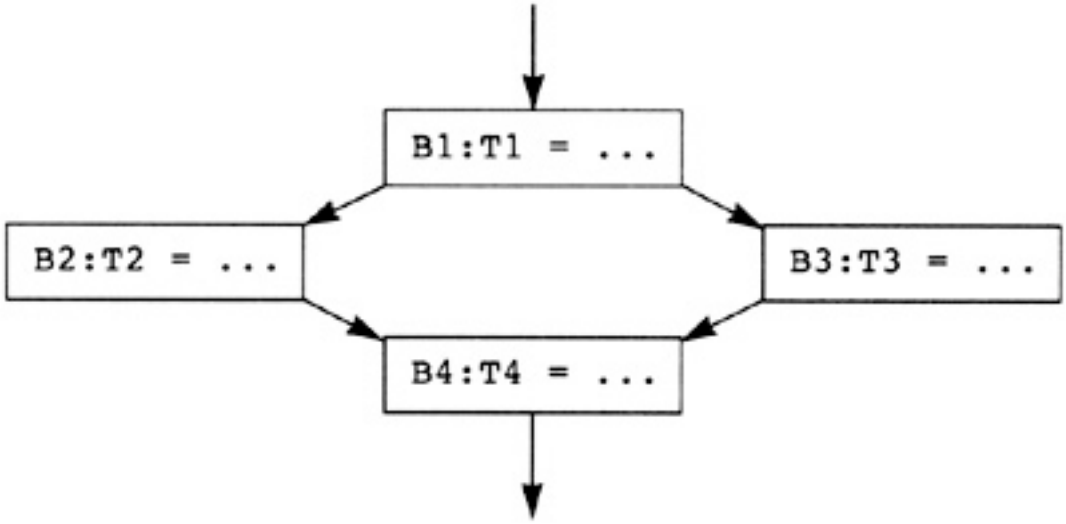


图 11: 图 12.11 Flow Graph for IDEFS Compuation

**定义**

*IUSE*:

对于每个 block B，IUSE(B) 是从 IDOM(B) 到 B 的某条路径上作为操作数被使用的临时变量的集合。这不包括出现在 B 或 IDOM(B) 中的使用。

### 6.5.2 12.5.2 计算指令干涉信息

两个观察（observation）和一个数据结构说明了计算 IDEFS 和 IUSE 集合的技术。考虑从 B 的支配者到 B 的任意路径，IDOM(B) = B0, ⋯, Bn = B。注意每个 Bi 都受 IDOM(B) 的支配。

开始遍历以 IDOM(B) 开始的路径。在流图中 B1 必须是 IDOM(B) 的后继节点。这意味着 IDOM(B) 是 B1 的直接支配者。将 B1 标记为 block Z1。继续遍历路径。起初，block（可能为空集）受 Z1 支配，但是最终要么到达了路径的末尾，要么找到了一个不受 Z1 支配的 block。将该 block 称为 Z2。断言 IDOM(B) 也是 Z2 的直接支配者。它受 IDOM(B) 支配，而不受路径上 IDOM(B) 之后任意其它 block 支配，因此它的直接支配者肯定是 IDOM(B)。继续遍历，直到找到一个不受 Z2 支配的 block，将它称为 Z3。完成整个过程，找到该路径上的一个 block 序列 Z1, ⋯, Zm，其中每个 block 在支配者树中是 IDOM(B) 的一个孩子节点。我们所要做的是找出每个在 Zi 和 Zi+1 之间的程序片段被修改的临时变量。我们即将看到，根据此信息，我们能计算出 IDEFS(Zi) 集合。

另一个观察告诉我们如何计算在 Zi 和 Zi+1 之间被修改的临时变量。考虑 Zi+1。在流图中，我们知道它的每个前驱节点。其中一个前驱节点是路径上 Zi+1 之前的 block。这个前驱节点受 Zi 支配。如果编译器知道受 Zi 支配的所有 block 的 IDEFS 信息，它就能够计算出从 Zi 到这个前驱节点的任意路径上被修改的临时变量集合（然后结合前驱节点的 OUT 信息，得到从 Zi 到 Zi+1 的信息）。

在描述这个计算方法之前，编译器需要一个这样的公式，它将 IDEFS 和在两个 block P0 和 Pr 之间可能被修改的临时变量的集合关联起来，其中 P0 支配 Pr。考虑 block 序列 P0, ⋯, Pr，其中 Pi+1 是 Pi 的直接支配者。

在 P0 到 Pr 之间的任意路径包括所有这些 block，而 IDEFS 的定义表明，在它们之间的任意路径上可能被修改的临时变量的集合，DEFS，必须满足下面的等式：

DEFS(P0, Pr) = IDEFS(Pr) . IDEFS(Pi) . OUT(Pi)

我们有基本的信息。编译器如何将基本的信息组织成一个算法？首先，编译器必须按照支配者树自底向上计算这些信息：为了计算支配者 block 的信息，需要被支配的 block 的信息。由于 IDEFS 的定义方式和前一个观察，这个观察就是支配者树中一个节点的孩子的信息可以影响其它孩子的信息，一个节点的所有孩子的信息是同时计算的。

编译器需要知道 DEFS(Zi, P)，其中 P 是 Zi+1 的前驱节点。此信息难于高效地存储。存储使用 UNION/FIND 算法。考虑一个 block B0，它是当前正在处理的 block。假设 Z1 到 Zn 是支配者树中 B0 的孩子。这样，每个受 B0 支配的 block，是以 Zi 为根节点的子树的一个成员。如果有一个 block P，它是同一路径上 Zi+1 的前驱节点，就可以从 P 开始沿着支配者树向上走，到达 B0 相应的孩子，它是树的根节点。在此遍历过程中，我们可以利用上面的公式计算 DEFS(Zi, P)。结合 OUT(P)，我们就可以计算在 Zi 和 Zi+1 之间可能被修改的临时变量。这是我们需要的信息。

但是，这样的树遍历是低效的。于是，创建一个影子数据结构，在遍历树的时候它包含相同的信息。在遍历过程中，此数据结构被折叠（collapse）。此数据结构基于 UNION/FIND 树，添加 EVAL 操作以计算集合。下面介绍它是如何被构造的。当处理一个 block 的时候，把它添加到 UNION/FIND 结构，其中这个划分（partition）的代表是已经处理的子树的根节点 block，子树中所有的 block 都受这个代表支配。当然，会发生标准的 UNION/FIND 折叠，使得该树比实际的支配者树更薄。此 UNION/FIND 结构中的边关联此结构中父节点和子节点之间的 DEFS。当折叠发生时，DEFS 集合被更新，以表示新的父亲和孩子。当 EVAL 被调用时，发生折叠，结果 DEFS 作为值被返回。

现在我们的算法差不多成形了，除了为特定节点的孩子计算 IDEFS。之前的讨论告诉我们什么？我们可以把 B0 的孩子看作一个新的图，其中在两个孩子之间有一条边，如果有一条不经过父节点的边，从一个孩子到另一个孩子。给定这个新的图，IDEFS 中临时变量的集合变成从图的根节点（它们是父节点的直接后继孩子节点）到某个节点的任意路径上被修改的临时变量的集合。为此，可以按照拓扑排序这些孩子。当然，会有强连通区域。这意味着任意的路径会穿过强连通区域，因此必须计算在一个强连通区域内被修改的所有临时变量的联合。

图 12.12 给出了执行上述计算的算法。查看每个孩子的前驱节点，找出支配这个前驱节点的别的孩子节点，这样构造孩子节点 Zs 的图。这确定了两个孩子节点之间的边。如之前指出的那样，向上遍历支配者树可以做到这个事情。相反，这个事情是由 UNION/FIND 算法做到的，因此路径可能被折叠。然后计算强连通分量，按照反向后序来排序其中的节点。这样达到了拓扑排序的效果。前驱节点出现在后继节点之前，除了强连通区域。

由于一条路径可以经过强连通区域任意次，一个强连通区域的作用是其中的 block 的作用的联合。对于单个 block，前驱节点和当前 block 之间没有作用。已经计算了概要的作用，此信息被添加到已经为前驱节点计算的信息中，以指示在这样的路径上可以计算什么，即从直接支配节点开始穿过一个它的后继节点的路径，这个后继节点也是当前节点的一个孩子（或根）。然后，此信息被添加到支配者树以存储结果。

图 12.13 给出了实现 UNION/FIND 和 EVAL 所需的支持函数。因为文献中几乎不使用 EVAL 操作，所以把它们包括进来了。实现它们需要两个属性。DEFS 表示在父节点和孩子节点之间被改变的临时变量的集合；此信息存储在孩子节点那里。FindParent 给出一个 block 的父节点。如果它是空，那么这是当前树的根。

```
procedure CALCULATE_IDEFS;
   call IDEFS_INITIALIZE;
   foreach B ∈ N in postorder on the dominator tree do
      let Z1,...,Zn be the children of B in the dominator tree;
      Edges = ∅;
      foreach Zi do      //Form Graph of the Zs
         foreach P ∈ Pred(Zi) do
            add (IDEFS_FIND(P),Zi) to Edges;
         endfor;
      endfor;
      Now consider the Zs with edges Edges as directed graph W;
      compute strongly connected regions S of W;
      foreach S in reverse postorder do
         CurrentDEFS = ∅;
         if S is strongly connected then
            foreach C ∈ S do
               CurrentDEFS = CurrentDEFS ∪ OUT(C);
            endfor;
         endif;
         foreach C ∈ S do
            foreach P ∈ Pred(C) do
               CurrentDEFS = CurrentDEFS ∪ IDEFS_EVAL(C) ∪
                             OUT(C)
            endfor;
         endfor;
         foreach C ∈ S do
            call IDEFS_UNION(B,C,CurrentDEFS);
         endfor;
      endfor;
   endfor;
endprocedure CALCULATE_IDEFS;
```

图 12: 图 12.12 Algorithm for IDEFS

```
procedure IDEFS_INITIALIZE;
   foreach B ∈ N do
      FindParent(B) = NULL;
   endfor;
endprocedure IDEFS_INITIALIZE;

function IDEFS_FIND(B: Block) returns Block;
   Parent = B;
   while FindParent(Parent) ≠ NULL do
      Parent = FindParent(Parent);
   endwhile;
   call IDEFS_COLLAPSE(Parent,B);
   return Parent;
endfunction IDEFS_FIND;

procedure IDEFS_UNION(Parent, Child: Block, Info: Set of Temporary);
   FindParent(Child) = Parent;
   DEFS(Child) = Info;
endprocedure IDEFS_UNION;

function IDEFS_EVAL(B: Block) return set of Temporary;
   Parent = IDEFS_FIND(B);
   return DEFS(B);
endfunction IDEFS_EVAL;

procedure IDEFS_COLLAPSE(Root, Child: Block);
   Parent = FindParent(Child);
   if Parent ≠ Root then
      call IDEFS_COLLAPSE(Root,Parent);
      DEFS(Child) = DEFS(Parent) ∪ OUT(Parent) ∪ DEFS(Child);
      FindParent(Child) = Root;
   endif;
endprocedure IDEFS_COLLAPSE
```

图 13: 图 12.13 Algorithms for UNION/FIND/EVAL

初始化简单地将所有 FindParent 属性设置为空。DEFS 属性不需要初始化，因为它只有在被设置之后才会被使用。FIND 操作向上遍历树，找出树的根。此事一旦发生，就利用折叠函数折叠这棵树，以缩短将来的遍历过程。

UNION 操作有一个固定的作为父节点的 block。保证输入给它的两个 block 的 FindParent 属性是空，因此不会发生折叠。其它属性是在父节点和子节点之间被修改的 block 集合，被简单地存储在数据结构中。

EVAL 操作利用 FIND 找出根节点。此时会发生一次折叠（在 FIND 中）。因此，EVAL 简单地返回存储的数据，此数据已经被更新为在根（现在为父节点）和当前 block 之间。

上述内容如此复杂，有必要给出一个例子。考虑一直在用的例子（回顾图 2.1），考虑它普通的流图。我们将处理单个临时变量。在这个案例中，我们可以把它看作布尔值而不是集合：如果临时变量在集合中，那么值为真。注意，block B1 支配 block B2 和 block B4。假设一个临时变量在 block B6 中被修改。IDEFS(B4) 是什么?

在处理 block B1 之前（它计算 IDEFS(B4) 的值），此算法主要处理 B2（它计算 IDEFS(B3) 和 IDEFS(B6) 的值）。block B3 和 B6 构成一个图，B6 处在 B3 的上游。当应用此算法的时候，OUT(B6) 的值被添加到 IDEFS(B3) 中，因此 IDEFS(B3) 为真。

现在，对 B1 应用此算法，计算 IDEFS(B4) 和 IDEFS(B2) 的值。B4 的一个前驱节点是 B3，它受 B2 支配，因此在子节点构成的图中，B2 处在 B4 的上游。在为 B4 计算 IDEFS 集合时，检查它的前驱节点 B3，我们发现 IDEFS(B3) 为真，所以 IDEFS(B4) 为真。

此博弈算法可用于计算 IUSE 集合，利用被用作操作数的临时变量的 IN 集合，而不是被修改的临时变量的 OUT 集合。

## 6.6 12.6 指令干涉图

现在，编译器已经确定了要调度的指令集合，构建了调度中用到的数据结构。[3] 指令干涉图记录了排序指令的限制。为每个 trace 构建干涉图，记录哪些指令必须在其它指令之前发出，必须提前多少时钟周期发出，这样当它们被后面的指令使用时，其值是可用的（见图 12.14）。

**定义**

干涉图：

给定一个 trace，它包含 block {B0, ⋯, Bn}，它的指令干涉图是一个有向无环图。在该图中存在三种不同类型的节点：

- trace 中的每条指令是图中的一个节点。它们是干涉图的基本元素。
- trace 中的每个 block B 有一个 Block Start 节点，它们被引用为 Block_Start(B)。这个节点被用来决定一个 block 从何处开始。它还携带必要的依赖信息，用以阻止这样的指令排序，它可能导致以后将被使用的数据被破坏掉。

[3] 注意我说了”用到的“而非”需要的“。不构建干涉图而作指令调度是可能的。反过来，跟踪指令计算操作数，跟踪它们的位置，这样可以隐式地构建干涉图。构建干涉图会更容易更有效，尽管它消耗时间和空间。

```
IDG = ∅;
for R ∈ Resources do
   LastWrite(R) = ∅;
   LastRead(R) = ∅;
endfor;
foreach B ∈ Trace in dominator order do
   foreach I ∈ B in execution order do
      add I to IDG;
      foreach T ∈ Operands(T) do
         if LastWrite(R) ≠ ∅ then
            choose J from LastWrite(R);
            if I ∉ Succ(J) then
               add I to Succ(J);
               add J to Pred(I);
               delay(J,I) = MachineDelay(J,I);
            endif;
         endif;
         add I to LastRead(R);
      endfor;
      foreach T ∈ Targets(I) do
         for J ∈ LastRead(R) do
            if I ∉ Succ(J) then
               add I to Succ(J);
               add J to Pred(I);
               delay(J,I) = AntiDependenceDelay(J,I);
            endif;
         endfor;
         if LastWrite(R) ≠ ∅ then
            choose J from LastWrite(R);
            if I ∉ Succ(J) then
               add I to Succ(J);
               add J to Pred(I);
               delay(J,I) = 1;
            endif;
         endif;
         LastWrite(R) = {I};
         LastRead(R) = ∅;
      endfor;
   endfor;
endfor;
```

图 14: 图 12.14 Computing the Interference Graph

- trace 中的每个 block B 有一个 Block End 节点，它们被引用为 Block_End(B)。结合 Block Start 节点，它被用来决定一个 block 有哪些指令，并且携带必要的依赖信息，用以阻止错误的指令排序。

在两个节点之间的边 (Tail, Head) 表明在最终的指令次序中 Tail 必须处在 Head 的上游。在两个节点之间没有边意味着它们的次序是任意的。每条边会标注一个整数 *delay((Tail), Head)*，指示从 Tail 发出到 Head 发出之间至少间隔多少个时钟周期。如果延迟是 1，那么 Head 可能在 Tail 随后的时钟周期发出。延迟为 0 是可能的。这通常意味着存在专用的硬件，相比正常的管线时序，它让一条指令的值更快地可被另一条指令使用。

何时在两个节点之间会有一条边？有两个必要的条件。在原始的指令次序中，Tail 必须处在 Head 的上游；就是说，Tail 在 Head 之前被执行。第二，两条指令必须使用或者定义相同的资源。有四种情况：

- 真依赖：如果 Tail 修改了某个资源，后面 Head 会使用这个资源，那么这是一个真依赖。在两个节点之间存在一条边，它的延迟数值指示 Tail 完成修改资源所需的时间长度。延迟的长度依赖于 Tail 和 Head，因为对于不同的指令对，资源变为可用所需的时间是不同的。
- 反依赖：如果 Tail 使用了某个资源，后面 Head 会修改这个资源，那么这是一个反依赖。不允许改变指令的次序：如果 Head 在 Tail 前面发出，那么 Tail 需要的值会被破坏掉。通常延迟是 1，表明仅仅存储和载入的次序是重要的；然而，架构可能给定一个不同的延迟。在 Alpha 21164 上，在一个 STORE 和一个 LOAD 指令之间的一条反依赖边的延迟是 3，因为访问刚刚被存储的数据是更困难的。
- 输出依赖：如果 Tail 和 Head 修改相同的资源，那么必须保持原始的次序，这样后面的节点会得到该资源被 Head 修改后的值。通常延迟是 1，表明仅仅次序是有关的。
- 输入依赖：如果 Tail 和 Head 都使用某个资源而不修改它，那么它们的次序是无限制的。因为指令的次序是任意的，所以不会创建边。

一个资源是任意表达程序执行状态改变的量。因此，每个临时变量是一个资源。于是，从求值一个临时变量的指令到使用这个临时变量的每条指令之间，会有一条边。从求值一个临时变量的指令到求值相同临时变量的下一条指令之间，会有一条边。从使用一个临时变量的每条指令到求值相同临时变量的下一条指令之间，会有一条边。

如果目标机器具有条件码，那么条件码是一个资源。处理它们，像处理临时变量。如果指令集普遍地设置条件码，像有些复杂指令集计算（CISC）架构，那么应该特殊处理条件码，因为干涉图的尺寸会特别大。在大多数 RISC 架构中，只有一些指令设置条件码（如果存在条件码），一些指令读取条件码。这时，将条件码处理为指令的隐式操作数或结果，就像临时变量处理为实际的参数那样。

根据 LOAD 和 STORE 指令能够引用的内存区域为它们计算干涉信息。编译器可以识别的每个内存区域是一个资源；因此之前别名分析中用到的标签指示了单独的资源。载入和存储操作的边匹配出现的依赖类型：

在每个存储操作和每个后续对相同内存区域的载入操作之间，有一条边。如果编译器能够确定它们引用的内存区域不重叠，那么边是不必要的。编译器能够确定内存区域是否不同，如果地址是已知不同的（例如，地址是不同的常数），或者如果依赖分析器留下的信息表明存储和载入操作不会引用相同的内存位置。

在每个存储操作和每个后续存储操作之间，有一条边。像考虑存储和载入操作那样考虑这种情形。

在每个载入操作和后续相同内存区域的下一个存储操作之间，有一条边。当然，如果地址已知是不同的，那么边是不需要插入的。

不是所有的边都需要插入到图中。假设编译器在创建一条边 (Tail, Head)，而图中已经存在两条边 (Tail, Middle) 和 (Middle, Head)，且

delay((Tail, Head)) <= delay((Tail, Middle)) + delay((Middle, Head))

那么，新的边是不必要的。图中已经存在的边比新的边对指令次序施加更强的约束。容易识别下列三种此类情况：

- 考虑一个节点 Head 使用一个资源 R。根据定义，肯定有这样一条边，它从每个修改 R 的上游节点到 Head。编译器只需要记录从上一个修改 R 的上游节点到 Head 的边。修改 R 的节点集合在图中构成一组边，因为在每个这样的节点到下一个节点之间存在输出依赖。
- 输出依赖存在相似的情形。如果 Head 修改资源 R，那么只需要一条从上游修改 R 的节点到 Head 的输出依赖边。
- 考虑一个节点 Tail 使用一个资源 R。从 Tail 到下一个修改 R 的节点有一条边，记录一个反依赖；然而，不需要记录它和后面修改 R 的节点之间的反依赖，因为初始的反依赖和随后修改 R 的节点之间的输出依赖包含了此反依赖。

BlockStart(B) 和 BlockEnd(B) 的干涉条件是什么？这些节点代表每个 block 的边界，因此编译器必须确保 BlockStart 节点出现在 BlockEnd 之前，支配者的 BlockEnd 节点出现在受支配 block 的 BlockStart 之前。另一种观察 BlockStart 节点的方式是这样的，它代表了出现在 block 之前且在支配者之后的所有指令。这些思想给出了 BlockStart 和 BlockEnd 的干涉条件：

在 BlockStart(B) 和 BlockEnd(B) 之间有一条干涉边，在 BlockEnd(IDOM(B)) 和 BlockStart(B) 之间也有一条干涉边。这样，BlockStart 和 BlockEnd 节点构成了图中的一个链表。实现它的时候，要么强制这些边存在，要么引入一个虚假的资源，让每个 BlockStart 节点写这个资源，让每个 BlockEnd 节点读这个资源。这会创建如上面提到的相同的边。

假装 BlockStart(B) 会读在 B 和 IDOM(B) 之间的指令读取的每个资源，假装它会写在 B 和 IDOM(B) 之间定义的每个资源。换句话说，让 IUSE(B) 作为 BlockStart(B) 使用的资源的集合，让 IDEFS(B) 作为 BlockStart(B) 定义的资源的集合。

## 6.7 12.7 计算指令优先级

接下来，编译器会计算每条指令的优先级，换句话说，在调度 trace 中的指令时，优先级表示指令对于整体调度的重要程度。如果编译器延迟调度一些指令，所谓的关键指令，那么整个 trace 的执行时间会变长。其它指令在被调度时有更大的自由度。

一条指令的优先级，是指令的最小执行时间，从它调度后的位置到 trace 的末尾。考虑一条未调度的指令，从它将来调度后所处的点到 trace 的末尾，其时间间隔最长。如果我们延迟调度这条指令一个周期，整个 trace 的执行长度就增长了一个周期。因此，对调度来说最重要的指令是时间间隔最长的指令，从它开始执行到 trace 的末尾。编译器会计算冲突图（interference graph）中从指令到冲突图的叶子最长的路径，以此估算一条指令从发出到 trace 的末尾所需的时间间隔。

为什么这是一种估算？这个数字可能不准确，有两个主要的原因。求出最长的路径，作为时间的长度，这个方法假设有足够的功能单元，这样每条指令在任意时钟周期都可以被调度出去。它还假设每个功能单元在每个时钟周期是可用的。如果没有足够可用的功能单元，那么有些指令必须延迟一个周期。在有些 Alpha 处理器上，每四个周期只能发出一条乘法指令。

冲突图是无环的，最长路径可以被高效地计算出来，同时可以改进估算，以部分补偿这两个状况。编译器必须为每条指令计算属性 priority(I)。可以用这样的方法计算这个属性，深度优先遍历整个冲突图，对于一个节点，先计算其后继的优先级，再计算它的优先级：

priority(J) = max {delay(J, I) + priority(I) | I ∈ Succ(J)}

不是所有指令都实现为简单管线化形式，因此必须用更复杂的公式。作为代表，考虑下面两个 Alpha 21164 中的情况：

- 整数乘法指令发出的频度不能超过每四到十二个周期一次，具体频度取决于指令和源操作数。每条乘法指令的延迟是八到十六个周期，因此乘法指令是部分管线化的。
- 在上一条浮点除法指令的结果出来之前，不能发出另一条浮点除法指令。

为了计算出一个更准确的优先级值，编译器必须计算由这些类型的指令导致的总的延迟。优先级不会小于这些值的其中一个。

编译器在计算这些总的值的时候，为指令依赖图中的每个节点维护临时变量属性 multiply_latency 和 divide_latency。这些属性只用来计算优先级，计算优先级之后可以丢弃它们。

图 12.15 描述了这个算法。它是前面的讨论的一个直接的实现。这个算法的形式是一个深度优先搜索，先处理后继节点，再处理当前节点。利用我们讨论过的方法计算到达 block 末尾所需的最长时间。如果有其它应该包括的信息，也可以添加到这个算法中。

## 6.8 12.8 模拟硬件

一种观点将指令调度视作编译器模拟硬件，在每个时钟周期跟踪哪些功能单元在使用。然后它选择一条待发出的指令，根据当前哪些功能单元不在使用，并且在这条指令将来执行期间也不在使用。

为了进行这样的模拟，编译器需要一种追踪当前在使用功能单元的机制。编译器需要这样的一种高效的机制，最好一次简单的载入操作就能查明所有功能单元的当前状态。

历史上，功能单元的状态建模为一个布尔矩阵。每列代表一个时钟周期，其中第一列是当前时钟周期。每一行代表一个功能单元，如果它在任意列（也就是时钟周期）的值是真，那么这个功能单元在相应的时钟周期是在使用中。类似地，每条指令建模为一个相同形式的矩阵（时钟周期表示为列，功能单元表示为行）。如果一条指令在随后的周期不使用已经在使用的任意功能单元，换句话说，如果两个矩阵元素对元素相与（AND）的结果是一个零矩阵，就可以调度（选择发出）这条指令。如果可以调度这条指令，就可以这样更新状态，将之前的状态和被调度指令的状态相或（OR），得到新的状态。

Table 12.2 Hypothetical Machine State

最终，将没有指令能够被调度，因为功能单元是忙碌的，或者所依赖的前面的指令还在执行。这时，编译器将调度推进到下一个机器周期。这包括平移状态矩阵，这样第二列变成第一列，第三列变为第二列，等等。

为了解释这个方法，考虑一个假设的机器，它有一个整数功能单元、一个浮点加法单元和一个浮点乘法单元。假设我们在调度一个机器周期的中间，如表 12.2 中的机器状态所示。这个状态表明，我们已经调度了什么指令，它在使用整数单元。

```
procedure COMPUTE_PRIORITY(instruction J)
   add J to Visited;
   multiply_latency(J) = Time_for_Multiply(J);
   divide_latency(J) = Time_for_Divide(J);
   priority(J) = 0;
   for I ∈ Succ(J) do
      if I ∉ Visited then
         call COMPUTE_PRIORITY(I);
      endif;
      multiply_latency(J) = multiply_latency(J) + multiply_latency(I);
      divide_latency(J) = divide_latency(J) + divide_latency(I);
      priority(J) = max(priority(J), priority(I) + delay(J,I));
   endfor;
   priority(J) =
            max(priority(J),multiply_latency(J),divide_latency(J));
endprocedure COMPUTE_PRIORITY;

Visited = ∅;
create priority;
create multiply_latency;
create divide_latency;
for J ∈ IDG do
   if J ∉ Visited then
      call COMPUTE_PRIORITY(J);
   endif;
endfor;
destroy multiply_latency;
destroy divide_latency;
```

图 15: 图 12.15 Computing Instruction Priority

表 12.3-12.5 代表单个指令类型的资源矩阵。多个指令可能共享功能用途的相同模式，因此它们可能结合在一起，让数据结构变小。

Table 12.3 Resource Matrix for Integer Operations

Table 12.4 Resource Matrix for Floating Add

Table 12.5 Resource Matrix for Floating Multiply

整数类型在一个周期完成任何运算，因此它在执行期间占用功能单元一个周期。浮点加法指令使用浮点单元两个周期，因此它不是完全管线化的。它只能间隔一个周期启动一条浮点指令。浮点乘法指令是完全管线化的。实际上，它应该被表示为多个功能单元在每个周期执行一个阶段；但是，只有浮点乘法器在使用这些功能单元，它们完全取决于管道中的第一个阶段，因此机器模型可以简化为只显示第一个阶段。

如果调度器首先调度一个浮点加法指令，然后在相同周期调度一个浮点乘法指令，那么机器状态看起来像表 12.6 那样。在这个周期，无法调度更多指令。

Table 12.6 End of One Cycle

Table 12.7 Machine State at Start of Next Cycle

为了开始下一个周期，机器将所有列向左平移一格，表明当前周期已经结束，下一个周期变成了当前时钟周期。这生成了表 12.7 中的状态。注意，机器可以发出一条整数指令或者一条浮点乘法。但是，不能发出浮点加法指令，因为相应的功能单元还是忙碌的。回想起浮点加法单元使用相同资源两次。

上面的描述是一种简化版本。有更多功能单元，不是所有功能单元都直接对应指令类型。例如，一个整数寄存器写的功能单元，将结果数据写到寄存器堆。还有，一些指令会使用多个主要功能单元：一个复制整数到浮点数寄存器的指令，会涉及一些整数功能单元和一些浮点功能单元。

有这样一个问题，以这种方式计算机器的状态太费时间了，要求调度器使用专用的代码。本编译器使用一种 Bala 和 Rubin（1996）提出的技术，来简化和加快处理状态。

### 6.8.1 12.8.1 预先计算机器状态机

主意是简单的。将机器状态表示为一个有限状态机。仔细看上面给出的描述。将每个机器状态矩阵视作有限状态机的一个状态。将每种指令类型视作词汇表的一个词，在词的作用下，一个状态转移到下一个状态，表示为矩阵相或（OR），如上面提到的那样。这给出了一个非确定性有限状态机。构建和它相关联的确定性有限状态机，我们就可以使用这个状态机而不是矩阵。这样，所有状态转移被简化为查询一个矩阵。

这个想法有一个问题。状态的数量可能很大：成千上万。这使得有限状态机需要巨大的存储。然而，Bala 和 Rubin 注意到，处理器有着非常规则的结构。整数单元几乎和单个浮点单元无关。是时候审视有限状态机的向量积了。考虑有两个状态机，其状态是 S1 和 S2，那么我们可以建立向量积有限状态机 (S1, S2)，它是由 S1 和 S2 构成的有序的状态对。执行从一个状态到另一个状态的转移，等价于有序对的每个元素执行转移，取转移结果的有序对：也就是，$\tau(S1,S2) = (\tau(S1),\tau(S2))$。

方案是这样的。将处理器划分成几个主要功能元素。每个部分构成一个机器。注意，所有指令是每个机器词汇表的一部分；整数指令很少改变浮点机器的状态，反过来也是。存储两个机器的状态，利用两个矩阵执行查找。每个主要功能部分的机器有数百个状态，而你在存储两个机器的状态。因此，状态可以表示为一对 16 位的数字。

注意，有限状态机可能是非确定性的。为什么？我们之前描述的构造不是确定性的吗？如果每个功能是单个功能单元，那么答案是肯定的。如果相同操作有多个单元（例如，多个整数功能单元），就会有相同指令类型到不同状态的多个转移。

这个机器的开始状态是什么？显然，一种开始状态是表示为值都是 false 的矩阵的状态；但是，还有两种其它状态类型：

- 当一个机器周期完成时，必须为下一个周期初始化机器状态。这要求将矩阵左移一列。因此，我们需要一个函数 STATE_SHIFT(S)，它读取一个状态 S，给出一个这样的状态，它的矩阵是将所有值都左移一列。这个函数的区间必须被考虑为调度下一个周期的开始状态。在内部，这个函数被表示为由 S 索引的一个向量，为下一个周期开始处的状态给出状态号。为了减小开始状态的数目，如果一个函数单元在一个给定的周期没有要调度的指令，就让调度器发出一条 NOP 指令。这意味着，所有初始函数单元将达到这样一种状态，它完成一个周期，而我们不需要为中间状态执行移位操作。
- 在 block 的开始处，编译器执行一个它的前驱 block 之后，必须估算一次机器的状态。这不需要准确：计算越精确，发生的停顿就越少。因为编译器不知道哪个 block 实际上是前驱 block，它通过将多个状态矩阵或起来，根据每个前驱 block 结束处的状态构造一个状态。实际上，我们只需要考虑两个前驱 block，因为我们可以连续地对剩余的前驱 block 成对地执行这个过程。因此，我们必须构造结束 block 的任意两个状态的或，由它们生成一个新的开始状态。我们需要一个函数 COMBINE_PRED(S1, S2)，它接受两个矩阵的或作为参数，返回移位后的结果，作为 block 的第一条指令的开始状态。

我们已经概述了程序。所有的计算都是在编译器构建期间做的，这样机器中的代码包括代表转移函数的矩阵、COMBINE PRED 函数、和一个代表 STATE_SHIFT 机器的向量。这非常像 LEX 和 YACC 中用到的表。

图 12.16 给出了算法的梗概。起初，机器的开始状态是完全空闲的。这个算法是按照矩阵编写的；但是，一个状态的矩阵存储一次，使用一个唯一的表示状态的整数来表示所有表中的矩阵，这些表被产生出来为编译器所用。

有一个等待列表，称为 StateQueue，每个状态自创建之后就存放在那里。每个状态只进入队列一次，因为它同时进入 StateTable 和 StateQueue，而且不会从 StateTable 移出。当一个状态被处理的时候，生成器尝试为每种可能的指令类别创建一个转移。

如果没有生成转移，那么对当前时钟周期来说机器满了，编译器必须生成一个转移，为下一个周期生成一个新的开始状态。为此，操作那个状态的矩阵，然后查看是否已经存在一个相应的状态。如果没有，也把它添加到状态集合中。

继续整个处理过程，直到所有状态已经被处理了，这样所有转移是已知的。算法执行结束后，一定找到了等价的确定性有限状态机。

### 6.8.2 12.8.2 向后查看已调度指令序列

针对有些调度优化和软件流水线，编译器有时想要向后扫描指令，为了向一个已调度列表插入指令。记录在资源矩阵中的机器状态和我们刚刚计算得到的状态告诉我们，是否存在一个空的位置，在那里可以插入一条指令。它并没有告诉我们，在那里插入一条指令是否会干涉后面某条已经被调度的指令。为此，我们需要反向有限状态机。

```
procedure BUILD_SCHEDULE_FSM;
   TransitionTable = ∅;
   StateTable = ∅;
   StateQueue = ∅;
   let FM be matrix with all false;    //Enter start state
   add FM to StateTable;
   add FM to StateQueue;
   while StateQueue ≠ ∅ do
      take F from StateQueue;
      get matrix FM for F;
      foreach instruction class I do
         get matrix IM for I;
         if FM ∩ IM = ∅ then
            D = FM ∪ IM;
            if D ∉ StateTable then
               add D to StateTable;
               add D to StateQueue;
            endif;
            add transition from F to D under I to TransitionTable;
         endif;
      endfor;
      if no transitions out of F then
         let D be FM shifted left by one column;
         if D ∉ StateTable then
            add D to StateTable;
            add D to StateQueue;
         endif;
         record STATE_SHIFT(FM) = D;
      endif;
   endwhile;
endprocedure BUILD_SCHEDULE_FSM
```

图 16: 图 12.16 Generating State Machine

考虑相同的状态集合，但是按照反方向构建转移。这样我们得到一个十足的非确定性有限状态机，由此我们可以构建一个确定性有限状态机。调度一个 block 之后，我们对 block 运行反向状态机，赋予每条指令一对状态数字。前向状态数字指示将来可以出现的合法指令，后向状态数字指示过去可以出现的合法指令。

现在，我们有了在指令执行之前对机器状态的表示和在指令执行之后对机器状态的表示。我们为每条指令存储此信息。在指令调度和寄存器分配期间，为每条指令记录两个临时属性。ForwardState(I) 是指令 I 执行之前机器的状态。BackwardState(I) 是指令 I 之后其余指令的状态。

### 6.8.3 12.8.3 在调度时替换指令

正常的指令调度只需要 ForwardState(I) 执行表调度（list scheduling）。事实上，不需要将它存储为一个属性，因为编译器只需要当前的状态，它可以存储为一个全局变量。调度指令打乱原始顺序，有三种实例：

正常调度指令的时候，我们在一个时钟周期调度一条关键指令，必须确保 block 的最小长度。此后，能够在它之后调度的关键指令变少了，只要它们不会延迟这条关键指令的调度。可以这样调度它们，先调度下一条关键指令，然后在它之前插入其它指令。

在执行软件流水线时，编译器调度一条指令，会假装实际上在均匀间隔的后续周期调度相同指令的影子版本。编译器必须记录这样的事实，影子指令被安排在后面固定的点。这样，有些后面的指令必须在下一条当前指令之前被调度。

在寄存器分配期间，指令极少会挤出（spill）到内存。这需要插入载入和存储操作。为此，最好的办法是在调度好的指令序列中找出可以放置 LOAD 或者 STORE 指令的空的位置，然后直接在那里放置这些指令。

因此，我们需要知道在什么条件下一条指令可以被另一条指令替换。这包括在已调度序列的一个空位插入一条指令的可能性。

假设已调度序列的每个位置具有状态 ForwardStarte(I) 和 BackwardState(I)，不管位置上有没有指令。于是，这个已调度序列可以被实现为一个足够大的数组，每条指令占据一个位置。开始时，将 ForwardState 和 BackwardState 属性初始化为每个机器的开始状态，指示所有资源矩阵都是空的。

下面考虑指令 I 可以被插入到位置 IS 的条件。能够在那个位置插入指令，意味着该指令和之前已经调度的所有指令都不冲突。这等同于有一个 ForwardState(IS) 的输出转移，因为只有在无冲突时我们才会创建转移。BackwardState(IS) 属性指示是否存在已经调度的后续指令和 I 冲突。如果不存在后续指令和 I 冲突，那么在 I 处有一个合法的 BackwardState(IS) 的输出转移。

如果指令 I 可以被放置在位置 IS 处，那么必须更新位置的 ForwardState 和 BackwardState 属性。这涉及从位置 IS 向前重新计算 ForwardState 属性，从位置 IS 向后重新计算 BackwardSate 属性。这没感觉上那么耗时。因为我们在处理有限状态机，只要新计算的状态不同于之前存储的状态，我们只需要向前（或向后）扫描。

只有少量位置重新计算状态会出现不同。为什么？回想有限状态机的构建，它涉及资源矩阵和列位移。一旦已经向左移动了当前指令涉及的所有列，当前指令在状态机中是不可见的。换句话说，只会出现少量的位移（矩阵的列的最大数量）。在实践中，只需要少量迭代。

图 12.17 给出的伪代码概述了这个插入算法。它详细描述了上面的讨论。如果指令无法插入，就返回 false。反之，插入指令并更新状态。

```
function INSERT_INSTRUCTION(I:Instruction, IS:slot) returns boolean;
   if τ(ForwardState(IS),I) ≠ Error ∧ τ(BackwardState(IS),I) ≠ Error then
      place I in slot IS;
      J = IS;
      State = ForwardState(IS);
      repeat                    //Fix up forward direction
         State = τ(State,Instruction_in_slot(J));
         J = J + 1;
      until State = ForwardState(J);
      J = IS
      Bstate = BackwardState(IS);
      repeat                    //Fix up backward direction
         BState = τ(BState,Instruction_in_slot(J));
         J = J - 1;
      until Bstate = BackwardState(J);
      return true;
   else
      return false;
   endif;
endfunction INSERT_INSTRUCTION;
```

图 17: 图 12.17 Inserting Instructions in Slots

## 6.9 12.9 调度算法

调度器将 trace 中的指令打包为 packet。每个 packet 中的指令可以在相同的时钟周期发出。下一个 packet 中的指令可以在下一个时钟周期发出，依此类推。对于 Digital Alpha 21164 来说，调度器会尝试发出四条指令：两个整数操作，一个浮点加法，和一个浮点乘法。在一个特定的时钟周期，如果不存在更多可发出的指令，调度器就会向每个不用的功能单元发出 NOP 操作。这样，每个 packet 总是满的；然而，可能包含 NOP 指令。之后编译器会结合多个 packet 以消除 NOP 操作。这不会直接加速处理器的执行；但是，这会减少指令的数目，从而提高指令缓存（cache）的效率。

这样，指令调度 phase 尝试将指令划分成多个 packet，每个 packet 中的指令可以被同时发出。为此，它必须将指令分组为多个 packet，使得一个 packet 中的指令不相互冲突。

如之前提到的那样，本调度器使用这样一个概念，就是基于支配者树的 trace。第一件要做的事情是计算辅助信息：trace，IDEFS，和 IUSE 集合。然后开始遍历支配者树，如图 12.18 描述的那样。它的基本结构是这样的，选择一个 trace，调度它，然后从 trace 中的 block 出发向下走，针对不在 trace 中的别的子节点执行一个 trace。与此同时，我们利用对支配者树的值编号来跟踪已经被调度的指令。用操作码（opcode）和操作数的值编号去索引值表。当一个临时变量被修改时，要么是表迎来了新的操作码和操作数，要么迎来了非已知的指令，但是有一个新的值编号（来自 IDEFS 计算）。

这里为什么使用值编号？所有冗余表达式不是被消除了吗？不是！指令调度可能引入冗余的表达式。考虑图 12.19 中的源语句。如果其中一个分支和开头的条件表达式属于同一个 trace，那么相当有可能 A*B 会被调度

```
procedure Schedule;
   call Calculate_Traces;
   call Calculate_IDEFS;
   call Calculate_IUSE;
   Initialize value numbering tables to empty;
   call ScheduleTrace(Root);
endprocedure Schedule;
```

图 18: 图 12.18 Driver for the Scheduler

```
if X <= 0 then
   G = A * B + 1;
else
   G = A * B + 2;
endif;
```

图 19: 图 12.19 Example of Hoistable Instruction

到条件转移之前。于是，它在别的 trace 的开头是可用的。

图 12.20 给出了遍历支配者树的实际算法。首先确定 trace，如之前描述的那样。它以一个 block 为开头，Trace(B)=B。在支配者树中最多一个子节点具有相同的 trace 的值，沿着树向下走，直到不存在子节点具有 trace 的 B 的值。然后调用 SchedulePackets 以调度这个 trace。调度 trace 之后，一次跟踪一条指令，将指令输入值表。当到达一个 block 的边界时，接着处理 trace 中的子节点；但是，得在那之前调度每个其它子节点的指令，因为这样的 block 肯定是一个 trace 的开头。

```
procedure ScheduleTrace(B: Block);
   let Trace = {B0,...,Bt} be the trace rooted at B;
   call SchedulePackets(Trace);
   foreach C ∈ Trace in dominator order do
      foreach I ∈ C after scheduling do
         enter I in value number table keyed by operands, operator
         if unknown modification occurs, give new value number
      endfor;
      foreach D ∈ Children(C) do
         if Trace(D) = D then
            call ScheduleTrace(D);
         endif;
      endfor;
      remove value number table information for C;
   endfor;
endprocedure ScheduleTrace;
```

图 20: 图 12.20 Determining the Trace and Walking It

图 12.21 开始真正的工作。SchedulePackets（注意复数形式）首先计算干涉图。这时，初始化属性 Ready(I) 和 PredLeft(I)，前者是指令可以被调度而不造成停顿的第一个时钟周期，后者是还没有被调度的前驱节点的数目。PredLeft(I) 是许多拓扑排序算法用到的属性，用以控制拓扑排序。总之，指令调度是干涉图的拓扑排序。Ready(I) 是操作数可用的最大次数。指令被调度，且指令对所关联的延迟已经发生，这时其操作数是可用的。由于它是一个最大值，我们把 Ready(I) 初始化为 0，每当我们发现一个给出更大值的操作数，就增加它。

在调度指令之前，算法会检查冲突图根节点处的指令是否在 trace 之外可用。如果是，就用一个 COPY 指令替换它。我们想做得更好，但是这里存在一个 phase 次序问题。寄存器合并（coalescing）已经发生了。我们试图让寄存器分配器为它们分配相同的寄存器；但是，这无法保证，因此必须使用复制指令，它会阻止其它优化。

集合 Ready 包含所有在这个周期就可以调度而无需延迟的指令。集合 Available 包含在这个周期或将来某个周期可调度的指令。换句话说，和那个集合中的成员相干涉的所有指令已经被调度了。为了计算这个集合，我们为每条指令记录一个属性，称为 PredLeft(I)，它是冲突图中还没有被调度的前驱节点的数目。当这个属性变为 0 时，指令被添加到 Available 集合。

有了以上这些设施，图 12.22 中的 Schedule Packet 程序从 Ready 选择可调度的指令。首先选择最重要的指令，只选择那些和已经调度的指令不相冲突的指令。所有指令被调度之后，Available 集合得到更新。packet 中一条指令的每个后继节点的 PredLeft 属性被减小。当它变为 0 时，其指令被添加到 Available 集合中。

```
procedure SchedulePackets(Trace);
   call Calculate_Interference_Graph(Trace);
   foreach I ∈ Trace do
      Ready(I) = 0;
      PredLeft(I) = |Interfere_Pred(I)|;
   endfor;
   let Available = {I|PredLeft(I) = ∅};
   foreach I ∈ Available do
      if I matches entry J in value table then
         replace I by copy from destination of J to
            destination of I;
      endif;
   endfor;
   IS = 0;
   State = Initial_State
   while Available ≠ ∅ do
      call SchedulePacket;
      IS = IS + 1;
   endwhile;
endprocedure SchedulePackets;
```

图 21: 图 12.21 Starting Trace and Scheduling Packets

```
procedure SchedulePacket;
   Packet = ∅;
   Ready = {I ∈ Available |IS ≥ Ready(I)};
   foreach I ∈ Ready in decrease value of Schedule_Importance(I) do
      if τ(State,I) ≠ Error then
         remove I from Available;
         add I to Packet;
         State = τ(State,I);
      endif;
   endfor;
   add function unit NOP operations until full packet;
   place Packet in slot IS in schedule;
   foreach I ∈ Packet do
      foreach J ∈ Interfere_Succ(I) do
         if IS + delay(I,J) ≥ Ready(J) then
            Ready(J) = IS + delay(I,J);
         endif;
         PredLeft(J) = PredLeft(J) - 1;
         if PredLeft(J) = 0 then
            add J to Available;
         endfor;
      endfor;
   endfor;
endprocedure SchedulePacket;
```

图 22: 图 12.22 Scheduling a Packet

什么是 Schedule_Importance？它决定 Ready 集合中哪些指令首先被调度。它对 Ready 集合中的指令作 lexographic 排序，它的思想基于 Warren（1990）描述的 RS600 指令调度器。针对每个主要的功能单元，指令被分别排序，按照下面的次序。

考虑具有最大 Priority(I) 的指令子集。这些指令比其它指令更重要，所以首先调度它们。Ready 已经包含其操作数已经在寄存器中的指令。首先计算决定执行序列长度的指令。

在此更小的指令集合中，减小寄存器压力的指令比增加寄存器压力的指令更重要。寄存器压力增大是指令调度的危险之一。事实上，如果追踪寄存器压力，就避免让它超过可用的寄存器数目。

在此更小的指令集合中，在干涉图中，后继节点多的指令比后继节点少的指令更重要。调度后继节点多的指令，能更快地增加 Available 集合的尺寸，因此似乎在不久的将来会有更多可被调度的指令。

如果还没有一个最好的选择对象，选择在原始 trace 中最靠前的指令。

当然这是一种启发式排序。通常来说，调度是一个 NP-完全问题。可以为特定处理器添加其它准则。例如，新的 Alpha 处理器能够对相邻连续的内存位置作多次存储，这是一种优势。这可以添加为一个准则。如果上一个周期有一条存储指令，Ready 集合中有一条存储指令，而后者指向的内存位置紧跟着前者指向的内存位置，就优先调度后者。

### 6.9.1 12.9.1 改进

针对这个调度算法，有两个可能的改进。它依赖处理器和被典型调度的程序集合。第一个改进发生在 Schedule_Importance。如果有一条关键的指令要调度，就调度它；但是，一条更早的指令不是关键指令，它被调度在一个更早的位置，它可能阻止关键指令的调度。怎么修改调度器可防止这种情形呢？（有时想到这是 NP-完全的）

考虑集合 Available，它包含所有这样的指令，其操作数开始被求值。选择这个集合中 Priority 最大的指令。计算指令位置，在那里指令的操作数是可用的。然后，在执行正常的调度过程之前，将这个关键指令调度到这个位置。

这对调度算法作了大的改动；但是，在有些处理器上它可能有用。此算法不再按顺序调度指令，这样我们只需要跟踪 ForwardState。现在，指令调度过程被视作一个大的指令数组，起初它是空的，每个空的指令位置的 ForwardState 和 BackwardState 为初始状态。在调度过程中插入一条指令必须使用替换算法而不是简单的插入。对于具有复杂架构的处理器来说，这个改动是值得的。

对调度算法别的改动是向后调度指令。换句话说，首先调度最后一个 packet，然后前面的 packet，依此类推，直到第一个 packet。为此，编译器必须建立干涉图的反向遍历，计算从一条指令到 block 开头的周期数，而不是从它到 block 结尾的周期数。除此之外，算法是相同的。

向后调度 trace 有两个优势。首先，调度器可以跟踪准确的寄存器压力。如我们之前看到的那样，向后追踪指令序列，能够让编译器看到哪条指令是最后一次使用，因此编译器知道何时一个临时变量是活跃的或不活跃的。

向后调度的另一个优势更加微妙。当向前调度指令时，会出现这样的点，在那里没有重要的指令可调度；但是，可能有的指令原本可以晚些被调度，它们被提早调度了，因为除此之外无事可做。这无故地增加了寄存器压力。通过反向调度指令，编译器会在几乎可能最近的时间调度一条指令，让其值在需要的时候可用。向

后调度 trace 的弱势是一个未知数。Trace 调度典型地按前向次序调度指令。后向调度在多大程度上适用需要一些实验。有人喜欢这样做，把指令从一个 trace 中执行最频繁的 block 调度出去。这怎么做?

可以作调度算法的最终改进。如果 trace 的开头的一些前驱节点已经被调度了，那么开始状态不是有限状态机的初始开始状态。相反，有些功能单元可能是忙碌的。建立有些状态机的时候，我们计算两个状态的汇合（Join）。这可用于前驱节点计算初始状态。如果一个前驱节点还没有被处理，就在生成汇合（Join）时忽略它。

### 6.9.2 12.9.2 Block_Start 和 Block_End

在讨论调度算法的时候，我们未曾讨论干涉图中的 Block_Start 和 Block_End 节点，插入它们是为了标记 block 的边界，确保调度过程是合法的。在调度过程中怎么安置它们呢?

就像指令一样处理 Block_Start 和 Block_End。它们是仅有的引用虚构功能单元的指令。在每个 packet 中，也有一个虚构的位置，在那里可以存放一个这样的伪指令。就按照算法设计的那样执行调度。包含 Block_Start 伪指令的 packet 表示 block 的开头，包含 Block_End 伪指令的 packet 表示 block 的结尾。这样，指令被调度之后，可以解析得到它们所属的原始 block。

## 6.10 12.10 软件流水线

有一种专门为简单循环设计的调度方法。如果按照上述方法调度循环体，那么循环的开头什么事情也不做。在循环体中，功能单元变得活跃，而在循环体的结尾处，功能单元再次无事可做。这样使用功能单元是低效的。

缓减这个问题有两种方法。首先，编译器可以按若干次展开循环，然后调度展开的循环体。这减缓了问题，因为展开的循环中包含多个循环体的副本，这样功能单元可能会保持忙碌；但是，在展开的循环的前端和后端，问题仍然存在。此外，循环体可能变得很大，引起指令高速缓存（cache）的问题。

调度循环的另一种方法是软件流水线。考虑有一个循环 L，编译器预先知道将被执行的迭代的次数。如果编译器能够让第二次迭代在第一次迭代之后立刻执行，后续迭代依此类推，那么在每次迭代的结尾处，功能单元会保持忙碌。

当然，上面陈述的是不可能的。只有一个指令流。但是编译器能够为各个迭代生成单独的指令流，然后尝试将它们交织在一起，形成一个指令流。实际上，图 12.23 的右侧列为此建立了模型，更加恰当地说明了编译器是怎么做的。编译器决定一个数字，将它引用为//，或者初始间隔。第一个循环迭代在第二个循环迭代的//周期之前开始执行，第二个循环迭代在第三个循环迭代的//周期之前开始执行，依此类推。得到的代码由三个小节组成。序曲部分代码为循环的执行作准备。它包含大部分第一个循环迭代的代码，少量第二个循环迭代的代码，依此类推。这样做的目的是让软件流水线后的循环周期性连续地执行计算。

软件流水线化的循环是重要的概念。循环的多次迭代相互交叠。第一次穿过软件流水线化的循环时，编译器完成第一次迭代的最后一条指令，完成第二次迭代的前部指令，等等。处理下一个迭代时，第一个迭代已经完成。第二次迭代所执行的指令，和之前循环体执行期间的第一次迭代相同，除了它们是为了后面的迭代。

软件流水线化的循环包含循环多次迭代的指令。我们稍后会讨论怎么知道迭代的数目。也会按某种程度展开循环，重命名临时变量，使得物理寄存器被正确使用。重要的是，原始循环中的每条指令在软件流水线化的循环中出现一次（如果循环被展开了，那么每条指令出现的次数是循环展开的次数）。

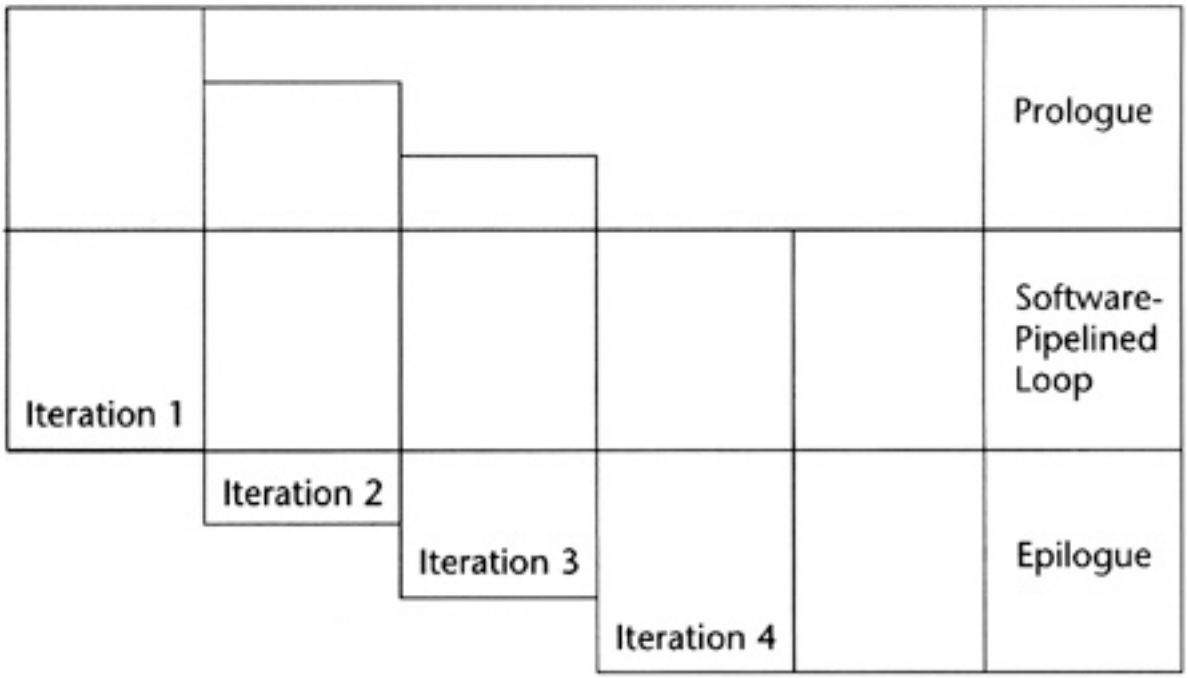


图 23: 图 12.23 Schematic of Pipelined Loop

这有什么好处？当循环单独的迭代引用独立的数据时，软件流水线是有效的。在这种情况下，一个迭代的计算和另一个迭代的计算不相关，因此多次迭代的指令可以被更紧密地排列（通常紧密得多）。

软件流水线化的循环一直执行，直到完成几乎所有的迭代。然后退出而进入尾声代码，完成循环最后的迭代。

如果原始循环的迭代数目足够小，软件流水线就没有优势。事实上，这让实现软件流水线更困难。如果循环的迭代数目是某个数字的倍数（后面再说怎么确定这个数字），那么实现软件流水线也会更简单。生成循环的两个副本，这样可以结合以上两个认知：一个是一模一样的副本，另一个是软件流水线化的副本。编译器如图 12.24 所示那样处理代码。在构建软件流水线化的循环期间决定常数 D 和常数 S，前者是软件流水线之前的迭代次数，后者表示循环的迭代次数。

编译器必须生成序曲（prologue）、尾声（epilogue）和软件流水线化的循环。实际上，首先会生成软件流水线化的循环，而所有其它的计算由循环决定。具体过程如下：

1. 调度循环的单次迭代，使得它可以和自身合并。想象卷起一张透明的纸，纸上有一些标记，卷起过程中，要求标记不重叠，标记均匀地分布。
2. 对于这个单次迭代的指令序列，找出循环中被赋值的一个变量活跃的最大时钟周期数。这将决定 S，并且和调度一起，将决定//和软件流水线化的循环。
3. 然后重叠循环的开头几次迭代的多次执行，生成顺序的代码（而不是循环），作为序曲。
4. 以同样的方法生成尾声。就是重叠循环最后的几次迭代，生成顺序的代码。

要开始这个过程，我们需要估计初始间隔。这是软件流水线化的循环的长度。这是一个初始的估计，在决定循环的过程中，有几个因素会导致它改变。

```
DO I = L,H
   Body
ENDDO
```

```
I = L
IF (H >= L + D) THEN
   WHILE I <= H - S DO
      Body 1
      Body 2
      ...
      Body S
      I = I + S
   ENDWHILE
ENDIF
WHILE I <= H DO
   Body
   I = I + 1
ENDWHILE
```

图 24: 图 12.24 Combining Unrolling and Start

### 6.10.1 12.10.1 估计初始间隔和限制条件

应用软件流水线，必须满足一些条件。我们给出一个简化的假设，即循环的每次迭代与其它迭代不相关。可以用多种方法检查这个条件：

如果编译器包含循环转换数据依赖分析器，那么可以用它检查循环是否存在循环递进（loop-carried）依赖。[4] 这是最好的技术，将发现更多适合软件流水线的循环。

如果循环中所有赋值语句左侧所写的数组和右侧的数组不同，那么迭代是不相关的。一个例外是，右侧出现的 load 操作所访问的位置被写了数据。这是最小的条件。它会找出大量适合软件流水线的循环，但是在线性代数代码中则不会。

如果循环中所有 load 和 store 操作通过临时变量被引用，而这些临时变量在每次迭代中改变的数据量相同，并且知道这些操作所引用的内存区域是不同的，那么这个循环可以被软件流水线化。令人惊讶的是，这是专用的也是通用的技术。它是专用的，因为它只能发现少量这样的情形。但是，如果用它生成两份循环的副本，一个是顺序执行，一个是流水线执行，就可以在循环的开头通过比较指针来选择它们。

本书实现一种软件流水线的有限形式。存在循环递进依赖时实现软件流水线是可能的。应用此处我们所讨论的一样的技术，结合干涉图中的依赖关系。相比简单的循环展开，此技术表现更好的情形是有限的。

说了这么多，到底什么是初始间隔//的初始估计呢？考虑循环 L。它由很多指令组成。每条指令必须在软件流水线化的循环中执行。将这些指令分类放到 bucket 中，每个功能单元类别一个 bucket：各种浮点单元，整数单元，和 load/store 单元。每个 bucket 中的元素数目除以那种类别的单元的数目。在 Alpha 上，有两个整数单元，因此整数指令的数目除以 2。[5] 这些系数的最大值是对初始间隔//的估计。

[4] 循环递进依赖指的是，循环的一个迭代存储一个值，而另一个迭代可能会加载这个值，或者一个迭代有一个 load 操作，而另一个迭代可能有一个针对相同位置的 store 操作。store 操作对 store 操作是类似的。

[5] 是的，我知道它们不是相同的单元。这个过程得到一个近似值。如果它不符合要求，后面会增加它。

简单来说，这个估计说明 packet 必须有足够的位置来存放所有指令。因此，当 packet 有充足的位置时，//取最小的值。

### 6.10.2 12.10.2 调度单次迭代

为了构造软件流水线化的循环，我们首先确定循环单次迭代的调度，对其自身滚动重复，以建造软件流水线化的循环。在讨论 trace 指令调度时，用到了相同的技术。但是，有两个主要的不同之处：第一，我们在处理构成循环的单个 block；第二，我们不会按顺序调度指令。

这意味着，我们将使用一种替换指令的算法，在状态机小节中我们讨论过该算法。起初，指令序列被安排为一个大的 packet 数组，每个 packet 中的位置都是空的。初始化 ForwardState 和 BackwardState 属性，以表明没有功能单元是忙碌的。现在计算冲突图，如我们为 trace 所做的那样。不需要包含 Block_End 和 Block_Start 伪指令。

现在，按照调度 trace 所描述的方法调度指令，但是有一个修改。当一条指令被放置到一个 packet 中的空位时，在//周期后的 packet 中的相同空位，在 2 * //周期后的 packet 中的相同空位，等等，依次插入这条指令的一个副本，在//周期前的 packet 中的相同位置，在 2 * //周期前的 packet 中的相同空位，等等，依次插入这条指令的一个副本。换句话说，在这些时钟周期为//的倍数的 packet 中的相同位置，都会有这条指令的一个副本。

因为我在这些空位中放置相同指令的副本是同步的，编译器没必要检查每个空位，以确认可以插入指令。每个空位的 ForwardState 和 BackwardState 和它的副本相同。如果它们不同，那么初始间隔太小了，则使用一个更大的初始间隔重新开始这个过程。

有可能不能实现这样的调度。需要插入一条指令，但是没有空位。这时，停下来，增加初始间隔，再次尝试。

重复这个过程，直到得到这样一个指令序列，每间隔//周期有相同指令的一个副本。注意这个过程必须终止。如果初始间隔//的值和一个 block 的指令序列的长度相同，那么我们会得到相同的指令序列而没有冲突。但是，软件流水线的性能正比于指令序列的原始长度除以//的值，因此当//接近于贯穿干涉图的最长路径的长度的时候，应该停止整个过程，改为使用循环展开。

想法是滚动重复这个指令序列，得到一个循环，使得它的长度是//个 packet。这不是工作的结尾，由于临时变量和物理寄存器。如果我们完全滚动重复循环体，那么每个临时变量经历一次循环就会被破坏，尽管在顺序执行指令时，从求值点到使用点的时间延迟可能变得更长了。举例来说，当求值和使用之间的时间是四个周期时，初始间隔可以取两个周期。因此，为了避免这个问题，我们必须按照所需的最小尺度展开循环。

### 6.10.3 12.10.3 展开循环和重命名临时变量

在这个时候，忽略指令序列中指令的副本。只考虑为一次迭代插入的指令。计算临时变量活跃的最大周期数 TL。利用常用的方法计算这个值，在编译器中我们一直在用这样的方法。向后扫描指令序列，记录何时临时变量变为活跃又变为不活跃。最大的长度就是最长的生命期。

现在，用上面确定的指令序列（包含指令副本）的最后//个 packet 构建一个指令序列，成为软件流水线化的核心版本。按照 S = (TL / //) 次展开这个循环，以保证定义和使用之间有足够的距离。

接下来，我们必须重命名临时变量使得使用和定义的关系符合（插入副本和展开循环之前）原始的指令序列。为此，考虑循环内部计算的所有临时变量：{T1, ···Tk}。在展开的循环中，生成这些寄存器的 S 份不同的副本：循环的每个迭代各有一份。如果你喜欢，原始的寄存器可以用作其中一份。

现在，同步地遍历原始的指令序列和展开的指令序列，修改新的循环中的临时变量，使得属于原始循环一次迭代的临时变量是同一组临时变量。为此，考察回滚的指令序列中的每条指令，同时考察原始序列中相同的指令。对于指令定义的临时变量，考虑所有的使用。展开的循环中的指令要使用相同的临时变量。在展开的循环中找得到这些指令，因为它们到原始指令的距离和原始的指令序列是一样的（考虑循环末尾的包裹）。

现在我们得到了软件流水线化的循环。它由核心循环体的 S 份副本组成，重命名临时变量使得值的定义和使用的关系是正确的。但是，事情还没结束。现在应该计算一下寄存器压力。如果寄存器压力太高，要挤出（spill）寄存器，就得增大初始间隔，重复整个过程，直到寄存器压力降下来。寄存器挤出将完全抵消软件流水线的优势。

### 6.10.4 12.10.4 生成序曲

为了生成序曲，我们来考虑软件流水线化的循环。假设序曲已经生成了。循环自身代表真实循环的 S 次迭代，其长度是// * S。当程序已经执行核心循环体的第一个//时钟周期时，我们完成了原始循环的第一次迭代。由于核心副本由原始指令序列的最后//条指令组成，序曲可以初始化为原始指令序列除了最后//条指令以外的所有其它指令，并且重命名其临时变量以匹配第一次迭代的临时变量。

当编译器结束第二个//时钟周期时，我们已经完成原始循环的第二次迭代。那个指令序列的最后//个 packet 在这个核中执行，前面//个 packet 在前面的核中执行。因此序曲包含所添加的一次迭代的除了最后//个 packet 的所有指令，之后移位//个 packet，并且重命名临时变量以使用来自第二次迭代的临时变量。继续这个过程直到再也没有指令可以添加。

尽管序曲中的这些指令构成了正确的指令序列，应该将它和其它周围的代码合并起来，更好地调度它们。这样，序曲应该作为包含循环开头的 trace 的一部分而被调度。

注意在展开的循环的末尾，我们执行了原始循环的 S 次迭代。于是，完整执行展开的循环，意味着执行了若干倍数原始循环的 S 次迭代。

### 6.10.5 12.10.5 生成尾声

按照生成序曲那样的方式生成尾声。当展开的循环被执行时，一次迭代（它是 S 的倍数）已经完成。有 [序列长度 / //] 次迭代还在执行中。可以这样构建尾声，将它初始化为原始指令序列的最后//条指令，重命名临时变量，使它们匹配展开的循环的下一个直到最后一个循环体。然后，添加原始指令序列中最后 2 * //个 packet，重命名临时变量，使它们匹配展开的循环中的前面的核的副本，等等。

注意这个迭代的数量可能大于 S，因此这个过程可能向后重复直到展开的循环中的最后一个循环核。如果循环中的所有指令都占用单个时钟周期，那么软件流水线一点好处也没有。可获利的循环是那些包含浮点数运算或者载入操作的循环。在 Alpha 上，一个浮点操作占用四个时钟周期。软件流水线隐藏了这个延迟，因此有可能实现四倍的加速。当然，如果已经有部分运算可以并行执行，这样大幅度的改善可能不会出现。载入操作可以带来更好的收益。在 Alpha 上，从 S-Cache 载入数据需要八到九个时钟周期。软件流水线可以隐藏很多这样的延迟；但是，更多的延迟意味着更多寄存器，这限制了软件流水线。

现在我们得到了整个循环，包括序曲、流水线化的循环、尾声。软件流水线可以获得的最大收益是多少？理想情况下，每个功能单元在每个时钟周期执行一条指令。最坏情况下，每个功能单元最多一个时刻执行一条指令，因此最大可能的加速倍数是流水线的长度。目标收益在于载入操作，它可能占用大量时钟周期。

### 6.10.6 12.10.6 乱序执行

新近的 RISC 处理器支持乱序（out-of-order）执行。这意味着处理器有一个指令缓冲区，其中的指令已准备好发出。处理器获取指令之后将它们存放在这个缓冲区，当一条指令的操作数可用时发出这条指令。如果一条特定的指令的操作数不可用，就等待。有可能后面的指令满足了操作数可用的条件，在前面的指令之前执行，正如名字表述的那样。

编译器如何模拟乱序执行？在文献中这个问题未得到解决。下面是我关于调度乱序执行的观点。

假装编译器能够做到完美的调度，这样每条指令恰好在操作数可用时准备好去执行。那么不会有延迟或停顿，处理器会全速运行。指令缓冲区的大小无关紧要。指令恰好在需要它们时是可获得以执行的。实际上缓冲区无限大：它永远不会溢出将要执行的指令。

有两个原因使得完美调度是不可能的。有些指令，例如 LOAD 指令，执行一段时间时钟周期，具体多少周期是无法计算的。编译器只能猜测这些指令需要多少时间。第二，在程序分支的地方，编译器根据分支条件猜测它会执行哪条路径。如果猜测错误，处理器必须回退并且再次执行指令。

我将乱序执行视作处理这些不可计算事件：载入和分支预测。编译器应该调度指令仿佛处理器不是乱序执行处理器。这种调度越有效，实际指令缓冲区的尺寸越大。乱序组件的角色是为了处理不可预测事件。换句话说，编译器利用指令缓冲区存放受限于这些不可预测事件和缓冲区的指令，减小由它们造成的时间损失。

调度指令仿佛它是顺序执行处理器，让硬件能够处理不可预测事件。为这样的处理器调度指令，这是一种合理的初始设想。将来会不会出现更好的调度方法，我们拭目以待。

## 6.11 12.12 参考文献

Bala, V., and N. Rubin. 1996. Efficient instruction scheduling using finite state automata. Unpublished memo, available from authors. (Rubin is with Digital Equipment Corp.)

Ball, T., and J. R. Larus. 1992. Optimally profiling and tracing programs. Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL92, Albuquerque, NM. 59-70.

Fisher, J. A. 1981. Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers C-30(7): 478-490.

Freudenberger, S. M., T. R. Gross, and P. G. Lowney. 1994. Avoidance and suppression of compensation code in a trace scheduling compiler. ACM Transactions on Programming Languages and Systems 16(4):1156-1214.

Huber, B. L. 1995. Path-selection heuristics for dominator-path scheduling. Master of Science thesis, Michigan Technical University.

Reif, J. H., and H. R. Lewis. 1978. Symbolic program analysis in almost linear time. Conference Proceedings of Principles of Programming Languages V, Association of Computing Machinery.

Sweany, P. H., and S. Beaty. 1992. Dominator-path scheduling: A global scheduling method. Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25), 260-263.

Warren, H. S. 1990. Instruction scheduling for the IBM RISC System/6000 processor. IBM Journal of Research and Development 34(1).

CHAPTER 7

# 第 13 章寄存器分配

编译器已经执行了寄存器重命名和寄存器合并，减小了寄存器压力，调度了指令。是时候给每个临时变量指派寄存器了。寄存器分配器必须满足下列按重要程度排序的约束条件：

- 正确性：编译器必须为不同的临时变量指派不同的寄存器，如果流图中存在一个点，在那里两个临时变量可能存放不同值，并且之后都将被使用。如果其中之一在那个点是未初始化的，那么编译器可以假设两个临时变量的值相同。
- 避免挤出（spill）：编译器应该尽其所能为临时变量指派寄存器，使得寄存器分配器在程序执行路径上插入的 LOAD 和 STORE 指令尽量少。
- 使用少量寄存器：编译器应该使用寄存器集尽量少的寄存器。应该先用函数调用者保存的寄存器，后用被调函数保存的寄存器。

很多编译器以简化的方式审视寄存器分配问题。它们将它描述为某种算法问题，例如图着色（graph coloring）或者装箱子（bin packing），然后利用启发式方法求解特定的方程式。这样的寄存器分配器对于寄存器需求量小的问题表现良好；但是，所需的寄存器数目远远超过可用的寄存器数目，这些寄存器分配方法都会生成大量的挤出（spill）指令，即寄存器分配器生成的载入和存储内存的指令。

这些寄存器分配技术都只利用了两种可用信息类型的其中之一，这是问题所在。图着色分配器用到了干扰图（interference graph）或冲突图（conflict graph）的概念。冲突图不表达哪些指令是相互邻近的，因此这种方法对于 block 表现糟糕。装箱子寄存器分配器对于 block 表现良好，但是它不得不利用近似方法处理控制流。有些情形一种算法表现更好，另一些情形另一种算法表现更好。这里的寄存器分配方法采纳了每种算法的长处。

本编译器结合以上两种算法。记得编译器已经插入了挤出（spill）指令，以减小寄存器压力，让它小于或等于可用寄存器数目。现在，编译器将利用三种不同的分配算法来分配寄存器：

编译器利用 Preston Briggs（1992）提出的一种派生的图着色寄存器分配算法，为跨越 block 边界活跃的临时

变量分配寄存器。

编译器利用 Laurie Hendron（1993）提出的一种派生的 FAT 算法，为那些可以和全局变量共享寄存器的临时变量分配寄存器。

编译器利用标准的单 pass 寄存器分配算法，为那些只在单个 block 中活跃的临时变量分配寄存器。这是一种装箱子（bin-packing）算法，按照反向执行顺序遍历 block，逐个为局部临时变量分配寄存器。

分开处理局部和全局临时变量，编译器可能会遇到 phase 次序问题：为全局临时变量分配寄存器，可能会阻碍为局部临时变量分配寄存器。这是不可避免的，最优寄存器分配是一个 NP-完全问题。从设计的角度恰当选择算法，尽可能避免这样的问题。

为了说明全局和局部寄存器分配的相互作用，考虑图 13.1 中一个 block 的图示。使用活跃临时变量的指令集合表示为水平间距。每个临时变量表示为不同的行。全局寄存器分配器会创建像 R1、R2 和 R4 那样的情形。R1 在另一个 block 被赋予一个值，其值在这个 block 中被最后一次使用。R2 在这个 block 中被赋予一个值，其值在另一个 block 中被使用。R4 结合两种情况：R4 在这个 block 中被赋予一个值，控制流离开又回到这个 block，在这个 block 的较早处使用这个值。R3 是典型的局部临时变量。它在 block 中被赋予一个值，之后其值在这个 block 中被最后一次使用。在大型函数中，这是最常见的临时变量。全局分配器为 R1、R2 和 R4 分配寄存器。FAT 算法会让 R2 和 R4 与其它局部临时变量结合。局部分配器为 R3 分配寄存器。

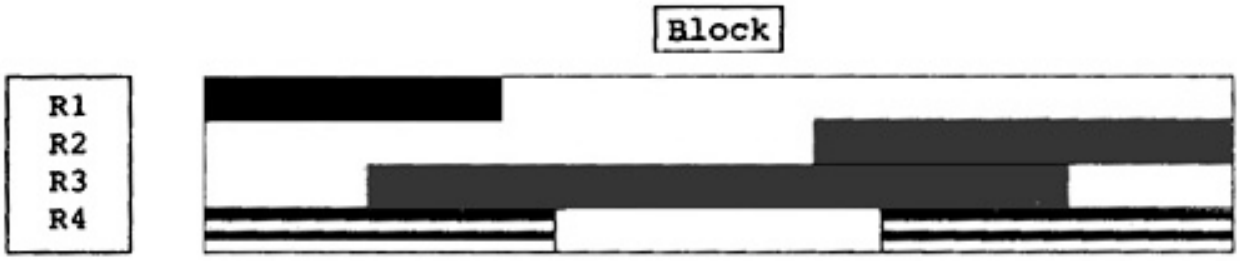


图 1: 图 13.1 Pictorial Representation of Block

```
procedure REGISTER_ALLOCATION;
   call GlobalAllocate;
   foreach B ∈ N from innermost loop outward do
      call LocalAllocate(B);
   endfor;
endprocedure REGISTER_ALLOCATION;
```

图 2: 图 13.2 Driver for Register Allocation

记得所有这些算法都是近似最优分配。通过求解整数规划问题可以达到最优分配；但是，对于产品编译器来说，这种技术代价太高。

寄存器分配的主体算法是依次执行每种形式的分配。FAT 算法创建数据结构，局部寄存器分配器使用此数据结构，两者一起工作。图 13.2 给出了调用结构。首先执行全局寄存器分配，然后对每个 block 应用 FAT 算法和局部寄存器分配算法。

## 7.1 13.1 全局寄存器分配

首先，编译器为全局临时变量分配寄存器，它们存放的值跨越 block 边界有效。通常，这意味着临时变量在一个 block 被求值，在另一个 block 被使用；但是，可能出现如下情形：临时变量在这个 block 中被求值，控制流离开 block，后来又返回，它的值在相同 block 的较早处被使用。

这个全局分配器基于 Preston Briggs（Briggs，Cooper，Torczon 1994）对 Chaitin（1981）图着色寄存器算法的修改。它使用冲突图和早前限制资源 phase 引入的干扰或冲突的概念。两个临时变量可以被分配相同的物理寄存器，如果它们不冲突，也就是说在流图中不存在这样的点，在那里它们存放着可能不同的值，其值之后会被使用。

分配问题是为每个节点（临时变量）分配一个寄存器，由边相连的两个节点不会被赋予相同的寄存器。这是图着色问题，其中寄存器集合是颜色集合。不幸的是，图着色是一个 NP-完全问题，不存在已知的好的求解算法。

Chaitin 为图着色重新应用一种启发式方法，它对复杂控制流是有效的。[1] 在冲突图中，邻居数小于颜色（寄存器）数的节点总是可以被着色，因为它可以被赋予除邻居颜色之外的任意颜色。这时，这个节点可以从图中移出，后面它将着色，在它的邻居都已经着色之后。重复这个过程，直到所有节点都已经从图中移出（如果可能的话）。然后，反转这个过程。将每个节点添加回图中，给它一个颜色，该颜色不同于当前图中其邻居的颜色。

经常地，利用这个启发式方法，所有节点都可以被移出图。这时，上述观察给出了着色该图的完整的算法。Chaitin 原本提出了一个算法，在没有邻居数小于颜色数的节点时停下来。算法然后选择一个节点挤出（spill）内存。

图 13.3 中的冲突图说明了启发式方法和一个较新的改进。有四个临时变量，边表示冲突。S3 有一个邻居，可以把它移出图，留下 S0、S1 和 S2。移出 S3 之后，它们各自有两个邻居，因此接着可以移出它们中的任意一个，比如说 S0，然后 S1，接着 S2。最后，我们得到一个序列（S2，S1，S0，S3），它们需要被赋予寄存器。S2 是第一个。把它放回到图中，赋予它任意寄存器，比如 R0。S1 是下一个：把它放回图中，赋予它一个寄存器，任意除了赋予 S2 的寄存器，比如说 R1。类似地，对 S0，赋予它 R2。最后，要给 S3 赋予一个寄存器。它只和 S2 冲突（S2 被赋予 R0），可以被赋予 1 或 R2。如此，这个算法可以分配寄存器，尽管 S2 有三个邻居。

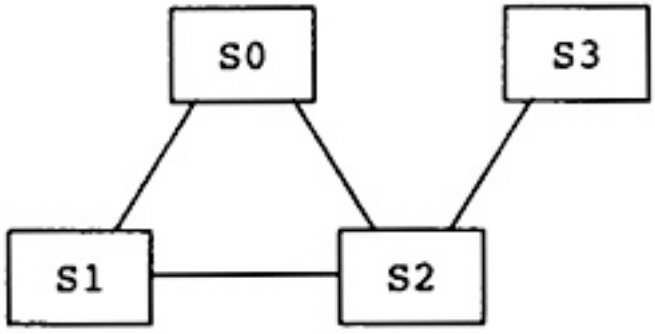


图 3: 图 13.3 Example Conflict Graph

[1] 如 Chaitin 同样注意到的，人们可以构建一个程序，其冲突图是任意的无向图，因此可以出现非常一般的图。然而，大多数图是简单的。例如，大多数临时变量只有一个定义点，大多数程序是结构化的，临时变量之间的相互作用更加受限。

纵然算法是借用序列描述的，我们看到，节点移出栈的次序和它们被赋予寄存器的次序是相反的。如此，当节点被移出冲突图时，它被压入栈，当它被再次插入冲突图时，它被弹出栈。

当然，节点本身并不移出冲突图。当节点被移出时，算法所引用的全部是它在冲突图中的邻居数。因此，算法必须记录仍然在图中的邻居的数目，当节点被移出时更新它。

### 7.1.1 13.1.1 何时启发式方法行不通

描述的启发式方法不是一个完整的算法。有可能冲突图中没有这样的节点，其邻居数小于颜色数。在当前这个编译器中，这是很少发生的，因为此前减小了寄存器压力，每个点活跃的临时变量数目减小了，冲突也减少了。然而，这是可能的。

Chaitin 原本建议，选择并挤出这样一个临时变量，它最不重要，它的邻居最多。在临时变量的每次定义之后，插入存储操作，存储到内存位置，在每次被使用之前，插入载入操作，载入到临时变量。这会从冲突图中去掉所有通向这个节点的边，算法可以继续去掉边，有希望从冲突图中移出更多节点。将节点弹出栈，赋予它一个颜色，此时总是有一个可用的颜色。

Preston Briggs 对 Chaitin 的原始算法提出了一个改进，使它的效果更好。Chaitin 方法的问题是着色启发式方法太粗糙了。它假设临时变量的每个邻居会被赋予不同的寄存器，于是其它临时变量所需的寄存器数目是邻居的数目。实际上，几个邻居可能赋予相同的颜色。如果这样的话，Chaitin 方法插入了不必要的存储和载入操作。

和 Chaitin 一样，Briggs 也建议选择最不重要的临时变量；然而，不是立即插入存储和载入操作，而是简单地将临时变量压入正在构建的栈中。现在，当从栈中弹出节点并赋予它颜色时，会出现没有颜色可用的情形。这时，如 Chaitin 的算法那样挤出（spill）那个临时变量。

Briggs 和 Chaitin 都在一个循环中重复寄存器分配，直到所有临时变量都被赋予了寄存器。当发生寄存器挤出时，寄存器分配器完成一趟完整的 pass，而有些临时变量没有被赋予物理寄存器。在挤出的值就要被使用和被存储之前，需要分配寄存器来存放它。寄存器分配 pass 在中途之前不能分配这些寄存器，因此处理它们的最有效方法是重复着色算法，利用完整的寄存器集合。

这里提出的寄存器分配器不需要重复图着色算法。挤出（spill）引入的新临时变量在单个 block 中被存储和载入，后面局部调度器可以处理它们。这隐含着，在局部寄存器分配期间，寄存器压力可能超过物理寄存器的数目。总结来说，给无法着色的寄存器赋予内存挤出位置，跟早前限制资源 phase 赋予挤出位置完全一样。之后在局部寄存器分配期间，为这些临时变量确定存储和载入的内存位置，赋予寄存器。为此，当临时变量 T 被挤出时，全局分配器执行如下转换：

- 为挤出的临时变量分配一个内存位置，MEMORY(T)，如果还没有分配的话。
- 将这个临时变量添加到 SpillRegisters 集合，指示局部寄存器分配器，应该在它首次使用前插入 LOAD 指令（如果前面没有定义的话），在它末次定义之后插入 STORE 指令（除非临时变量不再活跃）。

注意，这是资源限制 phase 挤出操作的角色反转。在限制资源 phase 中，编译器假设临时变量在寄存器中，只有在真正必要时，才将临时变量搬运到内存。这里假设临时变量在内存中，在需要时将它搬运到寄存器。因此，载入操作出现在 block 之前，存储操作出现在 block 之后。向后移动载入操作，向前移动存储操作，必然影响其它已经分配的临时变量。这样，对于这些操作，不把它们移到 block 内，就不能改善它们的位置。

### 7.1.2 13.1.2 总体算法

本编译器将这些想法结合成一个算法（见图 13.4）。首先，编译器为那些在任意 block 开始处活跃的临时变量重新计算冲突矩阵。冲突图的每个节点（也就是临时变量）关联一个计数，NeighborsLeft。将它初始化为等于这个节点的邻居数。在初始化 NeighborsLeft 的同时，这些节点按照 bucket 排序放入 bucket。同一个 bucket 中的所有节点具有相同的邻居数。

### 7.1.3 13.1.3 建立待着色临时变量的栈

然后，利用启发式方法从冲突图中移除节点，把它们压入到寄存器（临时变量）栈中。节点是按照 bucket 排序的，编译器只需要查看其中一个 bucket。

```
procedure GLOBAL_ALLOCATION;
   call Compute_Conflict_Graph(GlobalVars);
   MaxNeighbors = MAX{|Conflict(T)| where T ∈ GlobalVars};
   for i = 0 to MaxNeighbors do          //Initialize buckets
      Bucket(i) = ∅;
   endfor;
   foreach T ∈ GlobalVars do             //Bucket sort temporaries
      add T to Bucket(|Conflict(T)|);
      NeighborsLeft(T) = |Conflict(T)|; //Initialize for stacking
      InGraph(T) = true;                //Initialize for coloring
   endfor;
   call BUILD_ALLOCATION_STACK;
   call ASSIGN_GLOBAL_REGISTERS;
endprocedure GLOBAL_ALLOCATION;
```

图 4: 图 13.4 Driver Procedure for Global Allocation

应该首先检查哪些 bucket? 是所包含的节点具有最多边的 bucket，还是所包含的节点具有最少边的 bucket? 对作者来说，这是不明确的。如果首先查看边最多的节点，那么被移除的每个节点的边的总数更大，很可能更多节点的边的数目小于寄存器的数目。如果首先查看邻居较少（边较少）的节点，那么邻居数较小的节点将最后被着色，那时着色的自由度更小。当可用的寄存器较多时，将首先着色邻居数较大的节点。这个问题没有明确的答案。本书的设计首先查看边较少的节点，因为这样伪代码更简单。想要试验不同的次序，只需修改循环中引用 bucket 的地方。[2]

如图 13.5 所描述的栈操作的算法，可以作一些优化，选择合理的数据结构。这里有一些注意点。栈可以实现为预先分配的数组。它的尺寸不可能大于全局临时变量的数目。

编译器必须能够删除 bucket 中的任意节点。bucket 可以实现为双向链表。向 bucket 插入节点时，总是可以在链表的开头插入。

[2] 莱斯大学的 Keith Cooper 评论道，只有实验才能验证任何对寄存器分配算法的似乎合理的改进。从我的经验来说，有很多对算法的改变，在理论上应该只会提高分配器的性能，却降低了分配器的性能。这是 NP-完全问题的基本特征。

算法被写成尽可能简单地控制 i。我们可以试验选择节点的次序。我们也可以减小增长的数目。考虑所陈述的算法。如果当前节点在 Bucket(i) 中，那么下一个节点肯定在 Bucket(j) 中，其中 j >= i - 1，因此可以从那个点开始循环，而不是从 0 开始。

### 7.1.4 13.1.4 为栈中的临时变量赋予寄存器

临时变量被压入了栈中，易于分配的临时变量在栈的底部，难于分配的临时变量在顶部，之后，图 13.6 中的算法遍历整个栈，为临时变量赋予颜色。每个临时变量必须被赋予一个不同于其邻居的颜色。

```
procedure BUILD_ALLOCATION_STACK;
   ComputedPriority = false;              //No spill information yet
   Nodes = GlobalVars;                    //Nodes yet to pushed
   Stack = ∅;                             //Stack for coloring
   while Nodes ≠ ∅  do
      i = 0;                              //Find non-empty bucket
      while Bucket(i) = ∅ do
         i = i + 1;
      endwhile;
      if i > MaxPhysicalRegisters then //Satisfy heuristic?
         if ¬ComputedPriority then     //Have we computed spill info?
            ComputedPriority = true;
            call ComputePriority;
         endif;
         choose T from {U|Priority(T)/NeighborsLeft(T) is minimum};
         delete T from Bucket(NeighborsLeft(T));
      else
         choose T from Bucket(i);
         delete T from Bucket(i);
      endif;
      delete T from Nodes;
      push T on Stack;
      InGraph(T) = false;
      foreach U ∈ Conflict(T) do         //Update neighbors
         if InGraph(U) then
            delete U from Bucket(NeighborsLeft(U));
            NeigborsLeft(U) = NeighborsLeft(U) - 1;
            add U to Bucket(NeighborsLeft(U));
         endif
      endfor;
   endwhile;
endprocedure BUILD_ALLOCATION_STACK;
```

图 5: 图 13.5 Building Stack of Temporaries to Allocate

注意，算法不会试图更新返回到图中的邻居的数目。它不会更新属性 InGraph，因为它是用来告知已经着色了一个临时变量。

如果查看所有邻居之后，发现没有剩余的寄存器，就挤出（spill）这个临时变量。这包括，设置 InGraph 属性为假，指示它没有关联的物理寄存器，将这个临时变量添加到 SpillRegisters。局部寄存器分配器会想办法插入载入和存储操作，实现临时变量挤出。

```
procedure ASSIGN_GLOBAL_REGISTERS;
   AlreadyAssigned = ∅;        //Assigned physical registers
   SpillRegisters = ∅;         //Registers to spill
   while Stack ≠ ∅ do
      pop T from Stack;
      UnavailableRegisters = ∅;
      foreach U ∈ Conflict(T) do
         if InGraph(U) then
            add Color(U) to UnavailableRegisters;
         endif;
      endfor;
      if |UnavailableRegisters| = MaxPhysicalRegisters then
         if MEMORY(T) is not allocated then
            allocate location for MEMORY(T);
         endif;
         add T to SpillRegisters;
      else
         Color(T) = ChooseRegister(T);
         add Color(T) to AlreadyAssigned;
         InGraph(T) = true;
      endif;
   endwhile;
endprocedure ASSIGN_GLOBAL_REGISTERS;
```

图 6: 图 13.6 Register Coloring Algorithm

### 7.1.5 13.1.5 选择实际的物理寄存器

任何没有赋予给邻居临时变量的物理寄存器，大约都可以赋予给当前临时变量；但是，选择某些物理寄存器可能改善最终的结果。如果有一个物理寄存器，在函数别的地方已经使用了它，那么优先选择这个寄存器。如果只有未使用的寄存器可用，那么编译器必须斟酌处理器的调用规范。有些寄存器由调用函数保存和恢复。这些寄存器是临时变量寄存器，当前函数可以使用它们，而不带来额外代价。其它寄存器必须由被调函数保存和恢复。在函数内部第一次使用它们的时候，必须在函数序曲和尾声处插入代码以保存和恢复这些寄存器。

图 13.7 中的算法实现了这些想法，还附加了一个想法。考虑临时变量 T，正在为它分配寄存器。它的有些邻居（其 InGraph 属性为假），不妨称其中之一为 U，还没有分配寄存器。如果可以为 T 分配一个寄存器，相同于其它和 U 冲突的临时变量之一的寄存器，那么到时候为 U 分配寄存器可能更容易。

```
function ChooseRegister(T: Temporary) return PhysicalRegister;
   foreach U ∈ Conflict(T) do
      if ¬InGraph(U) then
         foreach Z ∈ Conflict(U) do
            if InGraph(Z) ∧ T ∉ Conflict(Z) then
               return Color(Z);
            endif;
         endfor;
      endif;
   endfor;
   foreach U ∈ AlreadyAssigned do
      if U ∉ UnavailableRegisters then
         return Color(U);
      endif;
   endfor;
   foreach U ∈ CallerSave do
      if U ∉ UnavailableRegisters then
         return Color(U);
      endif;
   endfor;
   foreach U ∈ CalleeSave do
      if U ∉ UnavailableRegisters then
         if MEMORY(U) not allocated then
            allocate memory for MEMORY(U);
         endif;
         insert STORE U,MEMORY(U) in prologue;
         insert LOAD MEMORY(U) => U in epilogue;
      endif;
   endfor;
   endfunction ChooseRegister;
```

图 7: 图 13.7 Choosing the Register

如果这个启发式方法行不通，就尝试给 T 赋予一个已经被使用的物理寄存器。这会降低已用寄存器的数目。记得指令调度已经发生，编译器已经重排指令，使用更多寄存器不会带来任何好处。

如果没有可用的已使用寄存器，就用一个 CallerSave 寄存器，因为保存和恢复它们没有代价。这也失败了，就用一个 CalleeSave 寄存器；然而，必须在流图的序曲和尾声插入代码以保存和恢复物理寄存器。

### 7.1.6 13.1.6 实现挤出（Spilling）

尽管伪代码有所描述，我们不曾讨论在选择临时变量压入栈的时候，没有临时变量满足启发式方法的情形。我们讨论了在指派寄存器的时候，没有寄存器可用该怎么办。这时，临时变量被放入集合 SpillRegisters，延迟挤出（spilling）操作直到局部寄存器分配。

本编译器利用 Chaitin 的方法选择临时变量，压入栈中（Chaitin 1982）。最近提出了更复杂的技术；然而，在当前的设计中它们的价值是不确定的。更复杂的技术看起来对于直线型代码和寄存器压力很大的情形表现更好；然而，我们用不同的方法处理这些情形。

选择临时变量压入栈时，有两个因素。寄存器着色的次序，和它们被放入栈的次序相反，编译器应该将最不重要的临时变量压入栈中。其次，编译器应该压入一个临时变量，它和大量不在栈中的临时变量冲突。这会减小冲突图中边的数目，使得更多节点更有可能满足着色启发式方法。编译器必须把这两个条件拼合在一起，形成单个算法或方程，来描述节点的优先级。很多方程可以做到；我们使用 Chaitin 的方程，它选择值最小的临时变量：

$$\frac{\Sigma\{frequency(p) | p \text{ is a point where } T \text{ is used or defined}\}}{NumberLeft(T)}$$

不幸的是，编译器无法预先计算以上信息，为可能发生挤出的地方保存起来，因为在临时变量压入栈的过程中，属性 NumberLeft(T) 在不断地变化。作为替代，编译器预先计算下面的方程，然后在需要挤出的时候执行除法：

```
Priority(T) = ∑{frequency(p) |p is a point where T is used or defined}
```

就代码而言，子函数 ComputePriority[3] 遍历流图，找出涉及临时变量的载入和存储操作，计算这个表达式的分子。将它保存为属性 Priority(T)。之后，当要选择一个临时变量压入栈的时候，除以分母，选择结果值最小的那个。

[3] 伪代码不包含 ComputePriority 的代码。它琐碎地遍历流图，利用存储在 block 中的频度信息，查看出现的载入和存储操作，累积优先级信息。

## 7.2 13.2 局部寄存器分配

全局寄存器分配完成了。现在，我们必须分配在 block 中活跃的寄存器。这个分配器有着不同的结构，因为在函数中临时变量活跃的区域更加规则。在 block 中可以按照指令被执行的次序枚举它们。如果没有已分配寄存器的全局临时变量，针对直线型代码，有简单的算法可以做到良好的局部分配。本书的编译器最后肯定会利用这些想法，但是必须首先处理已分配寄存器的全局临时变量，这样它们不至于破坏简单的直线型算法（图 13.8）。

在局部寄存器分配之前，编译器必须处理那些全局寄存器分配器没有给它们分配寄存器的全局临时变量。它们是集合 SpillRegisters 中的临时变量。编译器必须检查整个 block，执行三个任务。首先，在这些临时变量最后一次被赋值之后，必须插入一个 STORE 指令，把值写到内存。其次，在这些临时变量第一次被使用之前，必须插入一个 LOAD 指令，从内存读取值，如果这个使用的前面不是对临时变量的赋值的话。最后，在这个 block 内，必须给予这个临时变量一个新的名字。每个临时变量关联着一个单一的名字，每当编译器把临时变量引用分割为单独分配的部分时，必须为它创建一个新的名字。临时变量有了新的名字，它在不同的 block 里就可以被分配为不同的寄存器。

```
procedure LOCAL_ALLOCATE(B: Block);
   if SpillRegisters ≠ ∅ then call INSERT_GLOBAL_SPILL(B);
   call endif;
   call LOCAL_CLASSIFY(B);
   call BUILD_LOCAL_CONFLICT(B);
   call BUILD_LOCAL_BUCKETS;
   Stack = ∅;
   NumRegisters = MaxPressure;
   i = 0;
   while LiveStart ≠ ∅ do
      call ADD_TO_LOCAL_STACK;
      take T from LiveStart;
      delete T from LiveStart;
      call ALLOCATE_WITH_GLOBAL(T);
   endwhile;
   call ADD_TO_LOCAL_STACK;
   if any unstacked temporaries then call ONE_PASS_ALLOCATE(B);
   call endif;
   call GIVE_STACKED_TEMPORARIES_COLOR;
endprocedure LOCAL_ALLOCATE;
```

图 8: 图 13.8 Main Local Allocation Procedure

图 13.9 中的算法分两步执行了这三个任务。第一个 pass 反向遍历指令，对于这些临时变量的每一个，找出为其赋值的最后一条指令。在这些指令后面插入一个存储操作。同时，确定哪些临时变量前面需要插入一个载入操作。它一开始假设载入操作是需要的，如果发现了早前对临时变量的赋值，就否定这个假设。

第二个 pass 是前向 pass，利用属性 NewName 为挤出的临时变量存放局部的名字，在第一次使用临时变量名字前插入载入操作。

挤出（spill）全局临时变量之后，局部寄存器分配器分类出现在 block 中的临时变量。在描述分类之前，读者应该明白，寄存器分配器遍历指令的过程模仿了计算活跃信息的过程。事实上，经常计算活跃信息。总是按照逆向执行顺序遍历流图，隐式或显式计算活跃信息，同时执行某种处理。分类临时变量的时候，所收集的信息是一系列临时变量集合和最大寄存器压力，就是在任何时间点最大的活跃临时变量数。下面列出了这些集合：

- LiveThrough：这些临时变量在 block 中每个点都活跃。它们可能在 block 中被引用，也可能被修改；然而，它们在指令之间的任意点都是活跃的。因此，在整个 block 中，它们中的每一个都占据一个物理寄存器，使得这些物理寄存器不能用于局部分配。
- LiveStart：这些临时变量在 block 开头活跃，而在 block 中若干指令之后变为不活跃。这些全局临时变量给局部寄存器分配器带来麻烦。这个局部寄存器分配器向后遍历 block 中的指令（记得模拟计算活跃信息），为临时变量分配寄存器，必须小心从事，不让所分配的寄存器和已分配给 LiveStart 中的临时变量的物理寄存器重叠。分配器使用了 FAT 启发式方法。
- LiveEnd：这些临时变量在 block 的某条指令处变为活跃，且在 block 的末尾处活跃。它们不会给局部寄存器分配器带来麻烦。实际上，这些是预先分配的局部临时变量，为了在这个 block 中为它们分配寄存器。
- LiveTransparent：这些临时变量跨越 block 活跃，而在这个 block 中没有引用。像 LiveThrough 一样，这些临时变量占据一个物理寄存器，跨越这个 block。然而，当寄存器压力太高时，它们是有用的，因为可以在这个 block 之前和之后挤出（spill）它们，如限制资源 phase 所做的那样。
- LocalRegisters：这些局部临时变量在 block 中变为活跃，后来在 block 中变为不活跃。在计算密集的程序中，这是数量最大的一类临时变量。为这些临时变量分配物理寄存器是本节的重点。注意，挤出的临时变量所关联的新建临时变量属于这一类。

图 13.10 中的算法在 block 内精确地重新计算活跃信息，按照上面的定义，利用该活跃信息分类所有临时变量。举例来说，LiveTransparent 中的临时变量在 block 的出口是活跃的，在 block 内没有对它的引用。因此，LiveTransparent 初始化为出口处活跃的临时变量集合，然后移除被引用的临时变量。其它集合处理起来是类似的。

分类了临时变量之后，是时候准备寄存器分配了。令人惊奇的是，编译器为 block 计算冲突图。尽管这个分配器不以图着色为基础，但是图着色启发式方法提供了有用的信息：那些邻居少于可用颜色的临时变量是容易着色的，因此可以放在一边。这样重复这个过程，将所有容易的临时变量都放在一边，只剩下那些着色困难的临时变量，以专门的方式处理它们。事实上，容易的临时变量是琐碎的，移除它们之后，只对困难的临时变量做困难的决定。

编译器的局部寄存器分配器计算两种数据结构（见图 13.11）。第一种是局部冲突图，图中出现的临时变量只有当前 block 的临时变量。我们希望，建立的图是一个小的图。有这样的情形，函数是一个大的 block（几千行代码）。这时，全局冲突图是小的，而局部冲突图是大的。[4]

[4] 编译器编写者经常忘记有两类程序员。人类程序员更容易应付。编译器可以估算使用的模式。程序编写的程序更难处理，它们包含不友好的结构。

```
procedure INSERT_GLOBAL_SPILL(B: Block);
   InsertSTORE = SpillRegisters; InsertLOAD = ∅;
   foreach I ∈ B in reverse execution order do
      foreach T ∈ Targets(I) do
         if T ∈ SpillRegisters then
            if T ∈ InsertSTORE then
               delete T from InsertSTORE;
               insert STORE T,MEMORY(T) after I;
            endif;
            delete T from InsertLOAD;
         endif;
      endfor;
      foreach T ∈ Operands(I) do
         if T ∈ SpillRegisters then add T to InsertLOAD; endif;
      endfor;
   endfor;
   NewName = ∅;
   foreach I ∈ B in execution order do
      foreach T ∈ Operands(I) do
         if T ∈ SpillRegisters then
            if NewName(T) = NULL then
               NewName(T) = new temporary name;
            endif;
            if T ∈ InsertLOAD then
               delete T from InsertLOAD;
               add LOAD MEMORY(T) => NewName(T) before I;
            endif;
            replace T by NewName(T) in I;
         endif;
      endfor;
      foreach T ∈ Targets(I) do
         if T ∈ SpillRegisters then
            if NewName(T) = NULL then
               NewName(T) = new temporary name;
            endif;
            replace T by NewName(T) in I;
         endif;
      endfor;
   endfor;
endprocedure INSERT_GLOBAL_SPILL;
```

图 9: 图 13.9 Spilling and Classifying Temporaries

```
procedure LOCAL_CLASSIFY(B: Block);
   LiveTransparent = LiveOut(B);
   LiveThrough = LiveOut(B);
   LiveStart = ∅;
   LocalRegisters = ∅;
   Live = LiveOut(B);
   MaxPressure = |Live|;
   foreach I ∈ B in reverse execution order do
      foreach T ∈ Targets(I) do
         delete T from Live;
         delete T from LiveTransparent;
         delete T from LiveStart;
         if T ∉ Operands(I) then
            delete T from LiveThrough;
         endif;
         if Color(T) = NULL then
            add T to LocalRegisters;
         endif;
      endfor;
      foreach T ∈ Operands(I) do
         add T to Live;
         delete T from LiveTransparent;
         add T to LiveStart;
         if Color(T) = NULL then
            add T to LocalRegisters;
         endif;
      endfor;
      Pressure(I) = |Live|;
      MaxPressure = max(MaxPressure,Pressure(I));
   endfor;
   LiveEnd = LiveOut(B) - LiveThrough;
endprocedure LOCAL_CLASSIFY;
```

图 10: 图 13.10 Classifying Temporaries in a Block

算法还计算临时变量的活跃范围。FAT 算法需要该信息。为了记录该信息，赋予每条指令两个数字。从 block 的末尾开始，数字为 0，向着 block 的开始处，数字递增。数字对中小的那个代表修改寄存器的指令部分。大的那个代表获取操作数的指令部分。

```
procedure BUILD_LOCAL_CONFLICT(B: Block);
   LocalConflict = ∅;
   Live = LiveOut(B) - SpillRegisters;
   TimeCount = 0;
   foreach T ∈ Live do
      EndTime(I) = TimeCount;
   endfor;
   foreach I ∈ B in reverse execution order do
      TimeCount = TimeCount + 1;
      EndTime(I) = TimeCount;
      foreach T ∈ Targets(I) do
         StartTime(T) = TimeCount;
         delete T from Live;
         foreach U ∈ Live do
            create entry (T,U) in LocalConflict;
         endfor;
      endfor;
      TimeCount = TimeCount + 1;
      StartTime(I) = TimeCount;
      foreach T ∈ Operands(T) do
         EndTime(T) = TimeCount;
         add T to Live;
      endfor;
   endfor;
   TimeCount = TimeCount + 1;
   foreach T ∈ Live do
      StartTime(T) = TimeCount;
      foreach U ∈ Live - {T} do
         create entry (T,U) in LocalConflict;
      endfor;
   endfor;
endprocedure BUILD_LOCAL_CONFLICT;
```

图 11: 图 13.11 Building Lifetimes and Local Conflict Graph

每个临时变量关联两个属性。StartTime(T) 是关联写临时变量的指令的计数。如果临时变量在 block 的开始是活跃的，那么它引用一个在 block 前的值。EndTime(T) 是引用临时变量的最后一条指令的计数。如果临时变量在 block 末尾是活跃的，那么这个属性指代 block 的末尾。一次遍历 block，模拟计算活跃信息，计算得到这些属性，临时变量第一次变为活跃时，赋值 EndTime，第一次变为不活跃时，赋值 StartTime。

计算得到冲突信息和生命期信息之后，寄存器分配器准备执行标准的图着色启发式方法，移除容易的临时变

量。如同全局分配器，临时变量按照 bucket 排序（见图 13.12）。像全局寄存器分配器那样，采用相同的方法建立相同的属性。

现在，为了方便理解，我们以灵活的方式描述算法。我们要做的是，遍历整个 block，给临时变量赋予物理寄存器。后面，图 13.15 描述了这个算法。在分配开始之前，所有物理寄存器都存放在一个称为 FreeRegisters 的集合中，它们是可用的寄存器。我们扫描 block（还是按照反向的顺序，模拟活跃信息计算），当一个临时变量第一次变为活跃的时候（就是说，我们找到了临时变量的最后一次使用），把 FreeRegisters 中的一个寄存器赋予给它。在一个临时变量被定义的点（如果它不是同时被用作操作数），我们把分配给它的物理寄存器返还给 FreeRegiters。

问题是，如果在 block 的另一端，有全局临时变量已经分配了物理寄存器，这个方法会行不通。我们可能从 FreeRegisters 取出一个物理寄存器，赋予给一个临时变量，它的生命期重叠一个全局临时变量，而后者已经在使用那个寄存器。

解决方法是，预先处理在 block 另一端活跃的全局临时变量（这里是 block 的开头，因为我们在向后遍历 block）。这是 FAT 启发式方法。取这些临时变量的其中之一，称之为 T。FAT 启发式方法执行下面的操作：

1. 扫描整个 block，找出寄存器压力达到最大值的所有点。这些点称为 FAT 点。
2. 对于每个 FAT 点，选择一个在这个点活跃的局部临时变量。我们说，这个临时变量覆盖这个 FAT 点。这样选择临时变量，使得每个 FAT 点被覆盖，并且任意所选择的两个临时变量的生命期不重叠，和 T 的生命期也不重叠。这可能做不到；那时，将会有进一步挤出（spilling）。毕竟，这是一个启发式方法，不是算法。
3. 每个覆盖 FAT 点的这些临时变量都赋予和 T 相同的物理寄存器。
4. 在后续的分配中，不考虑那些 T 和覆盖 FAT 点的临时变量所关联的物理寄存器。在覆盖 FAT 点的临时变量之一活跃的每条指令处，寄存器压力减小 1。换句话说，我们忽略这些物理寄存器，T，和覆盖 FAT 点的临时变量。
5. 现在重复这个过程，处理在 block 开头活跃的其它全局临时变量，直到它们全部处理完毕。

在这个时刻，已经没有我们所关心的在 block 开头活跃的临时变量，于是我们可以应用单 pass 局部寄存器分配器，如上面描述的那样。

这是我们所用的算法。唯一的修改是，在每次处理这些临时变量时，编译器应用着色启发式方法，移除容易的寄存器。这是我们在图 13.8 中描述的算法。现在我们描述支持函数（support procedure）。

图着色启发式方法实现为两个函数，ADD_TO_LOCAL_STACK（见图 13.13）和 GIVE_STACKED_TEMPORARIES_COLOR（见图 13.14）。它们是全局分配算法的副本，这里不进一步描述它们。注意，变量 NumberRegisters 开始时等于常量 MaxPhysicalRegisters，在 FAT 算法执行过程中，它不断地递减。

注意，应用着色启发式方法的时候，应该不会涉及挤出（spilling）。当临时变量的邻居数小于颜色数时，将它压入栈中。如果条件不成立，就不能压入栈中。应用 FAT 启发式方法的时候，一个物理寄存器被放到一边，不再参与其中，因此允许的邻居数减小 1。这不影响之前压入栈中的任意临时变量。

图 13.15 描述了单 pass 寄存器分配器。它是一个单一的 pass，模拟活跃信息计算（所以它知道一个临时变量何时变为活跃），当一个临时变量变为活跃时，分配空闲的物理寄存器。如果一个临时变量已经有一个颜色了，就不需要给它赋予一个。可能需要在 block 内挤出（spill）临时变量，由于 FAT 启发式方法的失败。

```
procedure BUILD_LOCAL_BUCKETS;
   MaxNeighbors = MAX{|Conflict(T)| where T ∈ LocalRegisters};
   for i = 0 to MaxNeighbors do
      Bucket(i) = ∅;
   endfor;
   foreach T ∈ LocalRegisters do
      add T to Bucket(|Conflict(T)|);
      NeighborsLeft(T) = |Conflict(T)|;
      InGraph(T) = true;            //Initialize for coloring
   endfor;
endprocedure BUILD_LOCAL_BUCKETS;
```

图 12: 图 13.12 Build Buckets for Local Coloring

```
procedure ADD_TO_LOCAL_STACK;
   i = ∅;
   while i < NumberRegisters do
      if bucket(i) ≠ ∅ then
         take T from bucket(i);
         push T on Stack;
         foreach U ∈ Conflict(T) do
            if InGraph(U) then
               InGraph(U) = false;
               delete U from bucket(NeighborsLeft(U));
               NeighborsLeft(U) = NeighborsLeft(U) - 1;
               add U to bucket(NeighborsLeft(U));
               if i > NeighborsLeft(U) then i = NeighborsLeft(U);
               endif;
            endif;
         endfor;
      else
         i = i + 1;
      endif;
   endwhile;
endprocedure ADD_TO_LOCAL_STACK;
```

图 13: 图 13.13 Building Local Graph-Coloring Stack

```
procedure GIVE_STACKED_TEMPORARIES_COLOR;
   while Stack ≠ ∅ do          //No spilling possible
      pop T from Stack;
      UnavailableRegisters = ∅;
      foreach U ∈ Conflict(T) do
         if InGraph(U) then
            add Color(U) to UnavailableRegisters;
         endif;
      endfor;
      Color(T) = ChooseRegister(T);
      add Color(T) to AlreadyAssigned;
      InGraph(T) = true;
   endwhile;
endprocedure GIVE_STACKED_TEMPORARIES_COLOR;
```

图 14: 图 13.14 Coloring the Easy Local Temporaries

图 13.16 中的 FAT 启发式方法是对原始描述的直接实现。利用 FinishTime 局部变量，选择非重叠的生命期。按照逆向执行顺序遍历，这个变量指示了这样的点，在那里最近添加到覆盖集合中的临时变量再次变为不活跃。属性 BeginTime 指示了这样的点，在那里一个全局临时变量变为不活跃，它将要和所有这些临时变量共享一个物理寄存器。因此，被选择的下一个临时变量应该在最大压力点活跃，并且它的生命期不和开头的全局变量或覆盖集合中前面的临时变量重叠。

当需要挤出（spilling）的时候，使用经典的挤出启发式方法（图 13.17）。在寄存器分配的过程中，考虑一条指令 I，它有一个操作数需要一个赋予物理寄存器的临时变量。没有足够的物理寄存器，于是选择一个临时变量，它前面的使用是最远的。在 I 之后插入一个载入操作，在临时变量的上一次定义之后插入一个存储操作，这样一个寄存器被释放了，可用于 block 中可能最长的一段时间。

## 7.3 13.3 参考文献

Briggs, P., K. D. Cooper, and L. Torczon. 1992. Coloring register pairs. ACM Letters on Programming Languages and Systems 1(1): 3-13.

Briggs, P., K. D. Cooper, and L. Torczon. 1994. Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems 16(3): 428-455.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, MA. Published as SIGPLAN Notices 17(6): 98-105.

Chaitin, G. J., et al. 1981. Register allocation via coloring. Computer Languages 6(1): 47-57.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. Register allocation using cyclic interval graphs: A new approach to old problem.(Technical report.) McGill University.

```
procedure ONE_PASS_ALLOCATE(B: Block);
   GlobalRegisters = {Color(T)|T ∈ LiveStart}
   FreeRegisters = All physical registers;
   for T ∈ LiveEnd do
      delete Color(T) from FreeRegisters;
      InGraph(T) = true;
   endfor;
   FreeRegisters = FreeRegisters - GlobalRegisters;
   Live = LiveEnd;
   for I ∈ B in reverse execution order do
      foreach T ∈ Targets(I) do
         delete T from Live;
         if (T ∉ Operands(I)) ∧ (Color(T) ∉ GlobalRegisters) then
            add Color(T) to FreeRegisters;
         endif;
      endfor;
      foreach T ∈ Operands(I) do
         if T ∉ Live then
            add T to Live;
            if Color(T) = NULL then
               if FreeRegisters = ∅ then
                  call LOCAL_SPILL_REGISTER(I,B);
               endif;
               choose S from FreeRegisters;
               delete S from FreeRegisters;
               Color(T) = S;
               InGraph(T) = true;
            endif;
         endif
      endfor;
   endfor;
endprocedure ONE_PASS_ALLOCATE;
```

图 15: 图 13.15 One-Pass Register Allocation

```
procedure ALLOCATE_WITH_GLOBAL(B: Block, G: Temporary);
   BeginTime = EndTime(G);
   FinishTime = 0;                  //The end of the block
   foreach T ∈ LiveEnd do
      if Color(T) = Color(G) then
         FinishTime = StartTime(U);
         break;
      endif;
   endfor;
   Live = LiveOut(B);
   foreach I ∈ B in reverse execution order do
      foreach T ∈ Targets(I) do
         delete T from Live;
      endfor;
      foreach T ∉ Operands(I) do
         add T to Live;
      endfor;
      if I precedes FinishTime then
         if |Live| ≥ NumberRegisters + |LiveThrough| then
            while T ∈ Live do
               if EndTime(T) is later than FinishTime then
                  next iteration of loop;
               endif;
               if StartTime(T) precedes BeginTime then
                  next iteration of loop;
               endif;
               Color(T) = Color(G);
               FinishTime = StartTime(T);
               break;
            endwhile;
         endif;
      endif;
   endfor;
   NumberRegisters = NumberRegisters - 1;
endprocedure ALLOCATE_WITH_GLOBAL;
```

图 16: 图 13.16 FAT Heuristic

```
procedure LOCAL_SPILL_REGISTER(I: Instruction, B: Block);
   Scan the block in execution order finding the T ∈ Live with
      earliest previous use or definition;
   if MEMORY(T) is not allocated then
      allocate MEMORY(T);
   endif;
   insert LOAD MEMORY(T) => T after I;
   Insert a STORE T,MEMORY(T) after the previous use;
   Get a new temporary name;
   Scan from the previous use back to the beginning of the
      block renaming all references to T with this new name;
   delete Color(T) from FreeRegisters;
endprocedure LOCAL_SPILL_REGISTER;
```

图 17: 图 13.17 Spilling within the Block

CHAPTER 8

# Indices and tables

- genindex
- modindex
- search