迭代式全局指令调度*)

杨书鑫　薛丽萍　张兆庆

(中科院计算技术研究所先进编译技术研究组　北京 100080)

摘　要　基于非线性控制流图的全局指令调度由于非线性控制流的控制流图的复杂性不易计算出一条指令在其所在控制流图中的优先级,因此也不易判断来自不同基本块的指令的优先顺序,从而导致在决定一条指令何时被调度出该指令所在的基本块以及调度到哪儿时倾向于保守和随意。例如 D. Bernstein 的全局指令调度的启发性方法优先来自这些基本块的指令:调度器当前正在调度的基本块以及与当前基本块控制等价的基本块。然而,这种启发性方法往往导致处在关键路径上的指令被滞后。

本文提出的迭代式全局指令调度算法基于 D. Bernstein 的全局调度算法。它采用与 D. Bernstein 相同的启发性方法,但有选择地多次调度一个基本块使得处在关键路径上的指令被尽早调度。实验结果表明该算法以增加 10% 的调度时间开销提高调度器 8% 的性能。

关键词　全局指令调度,启发性方法,迭代式,非线性控制流图

The Iterative Global Instruction Scheduler

YANG Shu-Xin　XUE Li-Ping　ZHANG Zhao-Qing

(Research Group for Advanced Compile Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080)

Abstract　It is quite difficult for a non-linear control flow based acyclic global scheduler to determine an instruction's priority with respect to the control flow graph that scheduler deals with due to the complexity of the control flow. And hence, the approach to determine when an instruction should be moved across block boundary is ad hoc. Take D. Bernstein's global scheduling algorithm as an example, its heuristic favors the candidates come from the block that scheduler currently deals with and those blocks that are control equivalent with that block. Despite the straightforward nature of this scheme, it may postpone some instructions on critical path and hence lengthen the execution time. The iterative scheduling proposed in this paper is the extension to the D. Bernstein's algorithm. It employs the same heuristic approach as the D'Bernsteins'. However, it schedules an instruction on critical path as early as possible by selectively scheduling a block multiple times. The experiment indicates that the iterative scheduling can improve the performance of scheduler by as much as 8% at the cost the extra 10% scheduling time.

Keywords　Global instruction scheduler, Heuristic, Iterative, Non-linear control flow

1　引言

非科学计算的通用应用程序的分枝指令常常是平衡的,即它不偏袒分枝的任何一个出口。而基于线性控制流的调度器假定程序中有明显的高频率路径,而高频率路径之外的代码的性能是可牺牲的,因此这些调度不易在这些应用程序中取得较好的性能。调度算法应该扩展到非线性控制流图,权衡程序不同分枝上的指令的优先级,才能够取得理想的调度质量。

基于线性控制流的全局调度的代价模型(例如依赖高度,slack)是路径敏感的。线性控制流所包含的执行路径与控制流中的结点个数呈线性关系。因而调度可以综合一条指令在不同路径上的优先级构成它在整个控制流中的优先级。然而,在最坏情况下,非线性控制流图所包含的路径个数是其结点个数的指数函数,因此综合指令在每一条路径上的优先级来构成该指令在整个非线性控制流中的优先级是不现实的,从而导致它们对指令的优先级的评估比较粗糙。

以 Bernstein 的调度算法[5,17]为例:它自顶向下地调度区域中的每一个基本块;在调度一个基本块时,优先调度当前基本块以及与当前基本块控制等价的基本块中的指令,仅当目标处理机有空闲槽且这些基本块中又没有就绪指令时才将其它基本块的指令调度到当前基本块中。一个基本块 Y 与块 X 是控制等价的,当且仅当从控制流的任何一个入口到 Y 的任何一个路径必经过 X,且从 X 到控制流的任何一个出口的路径必经过 Y。事实上,Bernstein 的调度算法是用一个字典序(是否在当前基本块 X 或与 X 控制等价的基本块中,指令

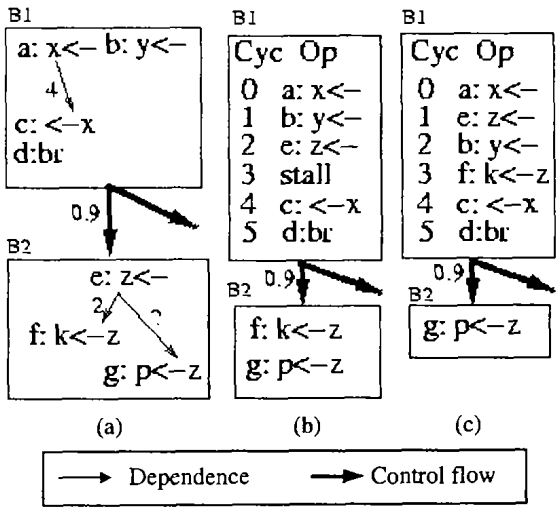

图 1

*)本研究受国家 863 计划(合同号 2001AA11061)和国家自然科学基金(批准号 69933020)的资助。

在它所在基本块范围内的优先级)构成一条指令在全局范围之内的优先级。尽管优先级的评估方法十分简单,但偏袒部分基本块的指令可能导致处在关键路径上的来自其它基本块的指令被滞后。例如在图 1(a)中,设处理机的宽度为 1,除了已标明的延迟外,其它指令的延迟均为 0。在图 1(a) 所示的控制流中,除了 B_1 自己外,没有其它基本块与它控制等价。设调度器当前正在调度 B_1,在上述的优先原则下,对 B_1 调度的结果如图 1(b)所示。因为指令 e 所在的基本块 B_2 不与 B_1 控制等价,所以它的优先级比指令 b 低。注意到指令 e 处在关键路径上,而 b 则可以滞后到第 4 拍(即 Cycle 3)发射。由于处在关键路径上的 e 被滞后,指令 f(或 g)无法被调度到 B_1 中,使得 B_2 需要 2 拍的执行时间。事实上,在同样的优先级机制下,我们可以对 B_1(在图 1(b)所示的基础上)再调度一遍就可以得到最优的结果,如图 1(c) 所示。对比图 1(b),B_2 只需要 1 拍的执行时间。

本文的主要内容就是讨论在 Bernstein 的调度算法基础上,如何通过有选择性地多次调度一个基本块减小"优先当前基本块及其控制等价基本块"启发性方法所带来的问题。本文第 2 节介绍相关工作,第 3 节介绍理解本文所必要的背景知识,第 4 节具体描述迭代式指令调度的动机和算法,在第 5 节报告实验结果及其诠释,最后总结全文。

2 相关工作

全局指令调度起源于微指令体系结构上的 microcode compaction。Trace scheduling[1]把调度区域从单个基本块扩展到了多入口多出口的线性控制流图,即 trace 上,同时把 list scheduling[11]作为启发性方法应用于调度中。尽管[1]调度过程较为简单,但是其代码补偿却十分复杂。基于 Superblock[3], Hyperblock[4] 调度算法派生于 trace scheduling[1],它通过尾复制取消旁入口(side entry)使得代码补偿过程变得简单。

事实上,即便是线性控制流也包含有多条执行路径。例如一个 Superblock 包含的执行路径等于它的出口数目(包括旁出口(side exit))。将 list scheduling 直接应用于基于线性控制流图的全局指令调度基于这样的假设:即控制流中的旁入口和旁出口的概率小得可以忽略不记。否则调度器将会过多地投机指令反而导致性能下降。对于不规则的非科学计算程序来说,上述的假定常常不成立。文[12]通过累加一条指令在不同路径的依赖高度与 Speculative Yield[2]的乘积来决定该指令的优先级。这种机制综合考虑了经过一条指令的所有执行路径以及路径的执行概率,对于基本线性控制流的全局调度来说,是一个比较理想的启发性方法。Speculative Hedge[13]不仅考虑了上述两个因素还考虑处理机的资源限制(例如处理机的宽度、功能部件的个数等)。

Treegion[14]调度通过尾复制(tail duplication)消除 joint 结点从而将一个任意无环控制流图转变成一棵树。尽管文[14]的调度区域依然是非线性的,然而由于调度区域是树,区域中所包含的路径数目与区域中的结点数目呈线性关系,调度器可以综合一条指令在不同路径上的优先级构成该指令在整个区域中的优先级。

对于基于线性控制流的调度器来说,完全忽略调度区域之外的代码的质量来提高当前调度区域的代码的性能可能反而会降低性能。尾复制是有限的(否则将导致代码膨胀),这个约束限制了 Treegion 调度区域的大小。调度器应该能够作用于含有 joint 结点的任意无环控制流图才能充分挖掘硬件资源丰富的现代处理机的潜力。这样的调度算法有许多。在这些调度算法中,有些并未提及如何决定指令的优先级而介绍其调度框架,如文[16];有些对指令优先级作一定程度的逼近但没有数据表明这样的逼近能够在多大程度上接近最优情况,例如文[2];有些类似于 Bernstein 的算法,即优先"当前基本块及其控制等价基本块"中的指令,仅当目标处理机有空闲槽且这些基本块又没有就绪指令时才将其它基本块的指令调度到"当前基本块"中。

3 背景知识

在这一节中,我们简单介绍 D. Bernstein 全局调度框架及其启发性方法,先介绍几个概念:

• $SISS(M,S)$:设控制流图 G 只有唯一的入口结点 R,G 所包含的基本块的集合为 N,块 M 和 S 是 N 的元素,其中 M 可达 S。$SISS(M,S)$是包含 M 的 N 的一个子集,且从 R 到 S 的任何一条路径必经过 $SISS(M,S)$的一个元素。如果$|SISS(M,S)|>1$,那么 $SISS(M,S)$中的任何两个元素在 G 上互相不可达。

• *M-ready*:我们称块 S 中的指令 i 相对于块 T *M-Ready*,如果任何一条 i 所数据依赖的指令 x 要么已经被调度要么从 T 不可达 x 所在的块(我们认为任何基本块可达它自己)。如果指令 i 相对于块 T *M-ready*,我们称 i 是 T 的 *M-ready* 候选指令。

• $CE(B)$:在本文中表示集合$\{x|x$ 是基本块 $\wedge\ x$ 与基本块 B 控制等价$\}$。注意,B 本身是 $CE(B)$的元素。

给定一个单入口的区域 R,D. Bernstein 通过以下步骤对 R 进行全局指令调度:

(1)通过 edge-splitting 消除所有的 JS 边(即源于出度大于 1 的结点且指向入度大于 1 的结点的边)

(2)创建 PDG[7]图,通过它统一地表示程序中的控制和数据依赖关系

(3)以自顶向下的拓扑顺序访问 R 中的每一个结点 N(基本块)并对其作用以下动作:

3.1)从 N 的所有可达的基本块(局限于 R 中)中找出所有相对于 N *M-Ready* 的候选指令,并由候选指令列表予以管理。

3.2)重复执行如下动作直到 N 中的所有指令均已调度:根据一定的启发性方法(见下文),从候选指令列表选出"最优"的候选指令 i 并调度之,如果 $S=SISS(N,i$ 所在的基本块$)-\{N\}$非空,复制 i 到 S 中的每一个基本块末尾(插在分支指令之前);更新候选指令列表。

(4)删除 R 中的空基本块

设调度器当前正在调度基本块 T,"最优"的候选指令是这样找到的:首先从考虑处在 $CE(T)$中的且能够在当前拍发射的(不会引起流水线暂停)相对于 T *M-ready* 的具有"最高优先级"(见下文)的指令,如果这样的指令不存在,考虑不在 $CE(T)$中的 *M-ready* 候选指令(由于处理机的结构相关或数据相关的缘故,"最优"指令可能不存在。调度器内部维护一个拍,每当上述情况发生,拍数增加 1。如果候选指令队列非空,当拍数增加到一定程度的时候,至少存在一条指令能够被调度)。至于在一个基本块内具有"最高优先级"的指令可以有多种方法衡量,常见的方法是通过依赖高度,本文以及文[5,17]亦采用此法,不再赘述。

4 迭代式指令调度

上文已经提到优先 CE("当前基本块")中指令的启发性方法会延迟关键路径上的指令因而增加了目标代码的执行时

间。然而，如果保持这个简单的启发性方法不变，通过对一个基本块的多次调度就可以减小这个启发方法的不足。我们称对基本块的的第一次调度为基本调度，之后的调度分别称为第一次迭代调度、第二次迭代调度等等，并统称为迭代调度。本节先讨论在什么情况下需要迭代调度，接着讨论如何进行迭代调度。

4.1 哪些情况需要迭代调度

在 Bernstein 的算法框架内，当一个基本块的分枝指令（假设目标体系结构没有采用 delay slot 技术减小分枝指令延迟的影响，下同）或该基本块的最后一条指令被调度后，调度器将不再分配新的"拍"以容纳来自其它基本块的指令。对于超标量体系结构来说，一个基本块的分枝指令或该基本块的最后一条指令被调度意味着对这个基本块调度的结束。VLIW 体系结构则不然，上述的"分枝指令或最后一条指令"所在的超长指令可能还有空闲的槽，而 VLIW 处理机不会自动地利用这些空闲槽对应的功能部件执行后续超长指令字中的指令，从而造成硬件资源的浪费。为此，在 VLIW 体系结构上，结束一个基本块调度的条件是当一个基本块的分枝指令或该基本块的最后一条指令被调度，并且在该基本块的最后一条超长指令字没有空闲的槽发射就绪的指令。为简洁记，下文不再区分在两种不同的体系结构上结束一个基本块调度的条件。

调度器不应该调度（相对于当前基本块）执行频率很低的 *M-ready* 候选指令即使目标处理机有空闲的功能部件。否则，指令调度将会引起过大的寄存器压力（若调度在寄存器分配之前进行）、增加 I-cache miss 等负面影响。在本节中，"*M-ready*"候选指令指的是对当前基本块"有益"的那些 *M-ready* 候选指令。

在优先来自 CE("当前基本块")指令的条件下，有两种情况意味着通过迭代调度可能可以提高调度质量。

情况 1 如图 1(a)所示，假设目标机的宽度为 1，指令 a、e 的延迟分别为 4 和 2，其它指令的延迟均为 0，$CE(B_1)=\{B_1\}$。设调度器当前正在调度 B_1。在优先 CE("当前基本块")指令的条件下，指令 b 被过早调度从而导致具有更高优先级的指令 e 被滞后，间接地导致了指令 f（或 g）不能被调度到 B_1 中。观察到在结束 B_1 调度时，尚有两条指令（即指令 f 和 g）相对于 B_1 *M-ready*，这意味着，对 B_1 的调度可能还有提高的潜力。事实上，在块 B_1 调度完毕后，指令 e 较 b 在 B_1 中有更高的优先级（两者在块 B_1 中均没有依赖后继，但 e 的延迟较 b 大），如果我们对块 B_1 再调度一次，那么 e 将较 b 早调度。如图 1(c)所示。这次调度把指令 f 提到 B_1 块中，从而块 B_2 只需要 1 拍的执行时间。

设基本调度结束后，相对于当前基本块 B *M-ready* 的指令集合为 I，且 I 非空。在决定是否进行迭代调度时，我们应该忽略 I' 中的指令，其中 $I'=\{i \mid i$ 来自 $CE(B)\}$。我们的机制已经优先来自 $CE(B)$的指令，因此下一次调度也不可能将它们调度到块 B 中。综合上面讨论，我们有：

规则 1 当结束一个基本块 B 的调度（基本调度或一次迭代调度）后，如果集合 $S=\{i \mid i$ 尚未被调度 $\wedge\, i$ 相对于 B *M-ready* $\wedge\, i$ 不是来自 $CE(B)\}$非空，那么对该基本块再进行一次调度可能会提高调度质量。

情况 2 如图 2(a)所示，类似于情况 1 的讨论，我们假设目标机的宽度为 1；指令 a 和 e 的延迟分别为 4 和 5；其它指令的延迟均为 0。在"优先当前基本块"指令的情况下，指令 e 最早也只能在拍 2 发射，由于 e 的延迟为 5，从而有 1 拍的延迟延伸到块 1 的外面。从而导致了指令 f 要被滞后 1 拍后发射。事实上，在 B_1 的基本调度结束（图 2(b)）后，对其进行一遍迭代调度即可消除延伸到块外的延迟，如图 2(c)所示。

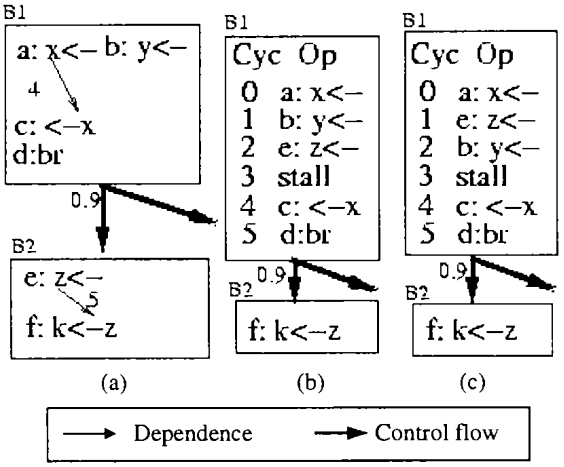

图 2

在此例中，假设指令 c 的延迟大于 2，那么在图 1(b)中，c 的延迟也要延伸到 B_1 之外。可是 c 已经被调度到尽可能早的地方了，无论再进行多少次迭代调度也不可能隐藏它的延迟。所以，在决定是否进行迭代调度时，我们应忽略在最近一次调度中，来自 CE("当前基本块")的指令。至此，我们得到规则 2。

规则 2 设在结束一个基本块 B 的调度（基本调度或一次迭代调度）后，如果集合 $S=\{i \mid i$ 在最近一次调度中不是来自 $CE(B) \wedge i$ 的延迟延伸到 B 之外$\}$非空，那么对该基本块再进行一次调度可能会提高调度质量。

4.2 哪些情况不能迭代调度

规则 1 和 2 仅仅乐观地估计再进行一次迭代调度可能会提高调度质量，但是并不断言一定会提高质量。所以当情况 1 或/与情况 2 出现的时候，不能盲目地启动下一个迭代调度，否则会陷入死循环。例在图 1(c)中，尽管块 1 调度结束之后还有相对于 B_1 *M-ready* 的指令 g，但是无论对 B_1 进行多少次的迭代调度也无法将指令 g 调度到 B_1 中。所以，我们有：

约束 1 如果对块 B 一次迭代调度没有将来自非 $CE(B)$的指令调度到 B 中，迭代调度必须终止。此外，如果迭代调度次数过多，将会大大降低编译速度，所以：

约束 2 当迭代调度次数超过某个（事先指定的）上限，迭代调度必须终止。

"某个（事先指定的）上限"因目标而定，上限越大，性能提高也越多，但是付出的时间开销也越大。第 5 节显示，上限为 1 是比较合理的，因为它平衡了性能和时间开销。

4.2 迭代调度

算法 1 综合了上述讨论。其中的 Sched_BB()是对一个基本块的一次调度（既可能是基本调度也可能是迭代调度）。User_defined_threshold 为调度器预定的迭代调度的次数上限。

算法 1 迭代调度框架

```
Iterative_Sched_BB (BASIC_BLOCK * bb) {
计算 bb 中每一条指令的依赖高度;
找出所有相对于 bb M-ready 的候选指令;
Sched_BB (bb); /* 基本调度 */
  INT iterative_sched_num=0;
  while ((满足规则 1 or 满足规则 2) and 满足约束 2) ) {
    标志 bb 中的所有指令为"未调度过";
    计算 bb 中每一条指令的依赖高度;
   * 用简捷的方法找出所有相对于 bb M-ready 的候选指令;
    Basic_Sched (bb); /* 一次迭代调度 */.
    if (满足约束 1) {
```

```
        iterative_sched_num = iterative_sched_num + 1;
    } else {
        break;
    }
} /* end of while */
} /* end of Iterative_Sched_BB() */
```

(一次)迭代调度和基本调度的差别仅仅体现在调度之前的搜寻 *M-ready* 候选指令上(在算法1中表现在用 * 标注的语句)。假设当前调度的基本块为 B。基本调度在调度之前建立初始的 *M-ready* 候选指令集合的时候需要判断 B 以及 B 所能到达的所有基本块中的所有指令是否相对于 B *M-ready*。而迭代调度则可以简单得多,它只需要检查 B 中的每一条指令以及上一次调度(基本调度或迭代调度)结束时相对于 B *M-ready* 的指令(很显然,在迭代调度开始的时候,这些指令之外的指令均不相对于 B *M-ready*)。

5 实验数据

5.1 试验平台

我们在基于 Itanium 处理机[8]的开放源码编译器 ORC v2.1[10]上完成所有试验。ORC 是英特尔和计算所的合作项目,它具有极其丰富的优化阶段。在 ORC 中全局指令调度发生在寄存器分配之前。寄存器分配过程中可能会对增加或删除一些基本块中的指令,对于这些基本块,ORC 会在寄存器分配之后,调用局部指令调度器对其重新调度。有些循环在全局指令调度之前已经被软流水调度器(Software pipeling)调度过,全局指令调度器忽略对这些循环的调度。

SPEC CPU2000[15]的整型基准测试程序被用来评估全局调度器的性能。运行基准测试程序的 Itanium-I 工作站频率为 733Mhz,具有2MB L3 cache 和1GB 的内存。运行的操作系统为 Red Hat 7.2。

5.2 迭代调度的有效性

全局调度的指令不仅和调度算法有关,还和调度区域的大小有一定的关系(区域越大,调度器可选的候选指令越多,因而调度质量越好)。区域的大小如表1所示。这些数据是在调度开始还没有通过 edge-splitting 消除 critical-edge 的时候统计的。第2列至第4列(从左往右)分别表示在对应的测试例中,调度区域包含基本块的个数的最大值、平均值以及区域包含指令个数的最大值和平均值。

表1 调度区域的大小

Bench-mark	Max-BB#	Avg-BB#	Max-OP#	Avg-OP#
bzip2	29	6.81	149	33.86
crafty	39	6.27	380	39.8
eon	65	5.22	576	30.33
gap	75	7.93	443	40.3
gcc	61	7.16	612	31.13
gzip	30	6.74	131	33.07
mcf	18	6.56	154	34.83
parser	46	6.15	125	24.56
perlbmk	44	6.99	782	32.38
twolf	37	7.15	239	44.55
vortex	39	8.68	592	48.19
vpr	37	7.26	207	40.54

迭代调度相对于基本调度在运行时(runtime)的加速比如表2所示。Nogs 和 Base 列分别为没有全局指令调度和没有迭代调度(即只有基本调度)的运行时间。两者之差(即 Delta 列)作为评估迭代调度的基准。基本调度的加速比(SPbase 列)的计算公式为(Nogs-Base)/ Nogs * 100%。Iter1列和 iter4列分别为在最大迭代调度为1和4的条件下,测试例的运行时间。它们相对于基本调度(Base 列)的加速比分别罗列于 SP1 和 SP4列,其中 SP1=(Base-Iter1) Delta * 100%;SP4=(Base-iter4) Delta * 100%。

表2 基本调度和迭代调度在运行时的加速比

Bench-mark	Nogs	Base	Delta	SPbase	Iter1	SP1	Iter4	SP4
bzip2	623	597	26	0.04	597	0.00	597	0.00
gzip	641	515	126	0.20	512	0.02	512	0.02
vpr	655	619	36	0.05	615	0.11	615	0.11
crafty	395	358	37	0.09	350	0.22	348	0.27
mcf	840	819	21	0.03	819	0.00	819	0.00
parser	980	911	69	0.07	907	0.06	900	0.16
gap	577	515	62	0.11	513	0.03	513	0.03
perlbmk	987	670	317	0.32	659	0.03	658	0.04
eon	550	534	16	0.03	529	0.31	533	0.06
vortex	697	634	63	0.09	621	0.21	621	0.21
twolf	1180	1124	56	0.05	1124	0.00	1124	0.00
gcc	394	377	17	0.04	377	0.00	372	0.29
平均						0.083		0.10

试验结果表明,本文提出的迭代调度算法比 D. Bernstein 的算法有显著提高。表2显示最大迭代调度为1和4的迭代调度相对于基本调度(D. Bernstein 的算法)的加速比分别为8.29%和9.98%。

5.3 试验分析

迭代调度对性能的提高的原因是把来自非 CE("当前基本块")的处在关键路径上的指令尽早地调度,具体表现为更多的指令被调度跨越基本块的边界。表3统计了这种情况。在我们的统计中,不管是全局调度之前就存在的指令,还是被复制出来的作为补偿代码的指令,只要调度器调度该指令跨越基本块的边界,那么它就在统计之列。在表3中,AcrossBase、Across1和 Across4分别表示在基本调度以及迭代调度次数上限分别为1和4的调度中,跨越基本块边界的指令个数。而 Inc1和 Inc4分别表示(Across1-AcrossBase)/AcrossBase * 100%以及(Across4-AcrossBase)/AcrossBase * 100%。表3显示上限为1的迭代调度能够(先对于基本调度)普遍而且显著地增加跨越基本块边界的指令的数目;相对于上限为1的迭代调度来说,上限为4的迭代调度虽然能够普遍地增加跨越基本块边界的指令的数目但是只有个别测试例比较显著。这个统计结果和两个调度相对于基本调度运行时的提高相一致。

表3 跨越基本块边界的指令数目

Bench-Mark	Across_base	Across_1	Inc1	Across_4	Inc4
bzip2	1081	1115	0.03	1115	0.03
Crafty	5811	6282	0.08	6481	0.12
Eon	13293	13710	0.03	13739	0.03
Gap	22403	23203	0.04	23336	0.04
Gcc	42150	43975	0.04	44155	0.05
Gzip	1671	1727	0.03	1727	0.03
Mcf	302	305	0.01	309	0.02
Parser	3311	3398	0.03	3410	0.03
Perlbmk	19373	20626	0.06	20845	0.08
Twolf	7121	7299	0.02	7345	0.03
Vortex	18081	18588	0.03	18700	0.03
Vpr	4119	4170	0.01	4183	0.02
平均			0.034		0.043

5.4 编译时间

迭代调度的代价是需要对某些基本块重新调度。表4中 Total 列为对应测试例中全局指令调度器调度过的基本块的

个数;Resched 列表示需要在基本调度之后至少执行一次迭代调度的基本块的个数。两者的比值(Resched/Total)列于最右一列。统计结果显示约有13%的基本块需要在基本调度之后至少执行一次迭代调度。

表4 需要迭代调度的基本块数目

Bench-Mark	Total	Resched	Total/Resched
bzip2	1032	147	0.14
crafty	4640	760	0.16
eon	18563	1358	0.07
gap	21371	3924	0.18
gcc	67362	6416	0.1
gzip	1430	244	0.17
mcf	293	45	0.15
parser	4577	530	0.12
perlbmk	24516	2655	0.11
twolf	7251	852	0.12
vortex	22334	2637	0.12
vpr	4786	448	0.09
平均			0.13

在表5中,我们分别统计了基本调度以及在最大迭代次数为1和4的情况下的调度时间(分别对应于 Base,Iter1和 Iter4列)。两个迭代调度相对基本调度增加的时间开销(以百分比的形式)分别由 Inc1和 Inc4列显示。统计结果表明:在迭代上限为1和4的情况下,调度时间分别增加10%和14%。综合两者对运行时的加速比,我们认为迭代上限为1的迭代调度是比较实用的。

表5 迭代调度时间的时间开销

Bench-mark	Base	Iter1	Inc1	Iter4	Inc4
Bzip2	1.93	2.02	0.05	2.15	0.11
Crafty	10.54	11.95	0.13	12.79	0.21
Eon	39.57	44.44	0.12	47.61	0.20
Gap	41.45	45.53	0.10	47.46	0.14
Gcc	103.84	112.27	0.08	116.2	0.12
Gzip	2.63	2.78	0.06	3.02	0.15
Mcf	0.75	0.9	0.20	0.84	0.12
parser	6.75	7.41	0.10	7.42	0.10
perlbmk	41.26	44.51	0.08	46.94	0.18
Twolf	14.51	15.77	0.09	16.39	0.13
vortex	37.92	41.64	0.10	44.07	0.16
Vpr	7.88	8.68	0.10	8.67	0.10
平均			0.10		0.14

结论 基于非线性控制流的全局指令调度比基于线性控制流的全局指令调度具有明显的优越性,因为它能够在更大的范围内进行指令调度并且能够权衡不同控制流不同分支上的指令的重要性。然而,由于非线性控制流存在 joint 结点,控制流包含的路径的数目可能十分庞大。全局调度器无法综合一条指令在不同路径上的优先级(例如依赖高度)来构成该指令在整个控制流中的优先级,而整个控制流中的优先级是启发性方法的依据。Bernstein 全局调度的启发性方法组合了"优先当前基本块及其控制等价基本块"和指令在它所在基本块优先级,然而这种偏袒来自部分基本块指令的启发性方法会延长关键路径。本文提出的迭代式指令调度算法,就像它名字所预示,一步一步地将被滞后的处在关键路径的指令尽量提前发射。实验表明迭代能以较少的额外编译时间开销显著地提高调度质量。迭代上限为1的调度是比较实用的,调度器8.3%的性能而只增加10%的调度时间。

参考文献

1 Fisher J A. Trace scheduling: A technique for global microcode compaction. IEEE Trans. Comps., 1981, C-30(7): 478~490

2 Fisher J A. Global code generation for instruction-level parallelism: Trace scheduling-2: [Technical Report HPL-93-43]. Hewlett-Packard Laboratories, June 1993

3 Hwu W W, et al. The superblock: An effective technique for VLIW and superscalar compilation. J Supercomputing, 1993

4 Mahlke S A, Lin D C, Chen W Y, Hank R E, Bringmann R A. Effective compiler support for predicated execution using the hyperblock. In: Proc. of the 25th Annual Intl. Symposium on Microarchitecture, June 1992. 45~54

5 Bernstein D, Cohen D, Krawczyk H. Code Duplication: An Assist for Global Instruction Scheduling. In: Proc. 24 th Ann. Int'l Symp. Microarchitecture (MICRO24), 1991

6 Bharadwaj J, Menezes K, McKinsey C. Wavefront scheduling: Path based data representation and scheduling of subgraphs. In: Proc. Int. Symp. Microarchitecture (MICRO32), Israel, Nov. 1999. 262~271

7 Ferrante J, Ottenstein K J, Warren J D. The Program Dependence Graph and its Use in Optimization. ACM Transation of Programming Languages and Systems, 1987, 9(3): 319~349

8 Intel Inc. http://www.intel.com/design/itanium/itanium/index.htm.

9 SPEC2000

10 ORC team, ORC v2.0 suite. http://sourceforge.net/projects/ipf-orc, 2001-2003

11 Coffman Jr E G. Computer and Job-Shop Scheduling Theory. John Wiley and Sons, New York, 1976

12 Bringmann R A. Enhancing ILP Through Compiler-Controlled Speculation: [PhD thesis]. Department of Computer Science, University of Illinois, Urbana, IL, 1995

13 Deitrich B L, Hwu W W. Speculative hedge: Regulating compile-time speculation against profile variations. In: Proc. of the 29th Intl. Symposium on Microarchitecture, 1996. 70~79

14 Banerjia S, et al. Treegion Scheduling for Highly Parallel Processors. In: Proc. of Euro-Par'97, Passau, Germany, Aug. 1997

15 Stan Performance Evaluation Corporation. http://www.spec.org/cpu2000, 2003

16 Bharadwaj J, Menezes K N, McKinsey C. Wavefront Scheduling: Path-Based Data Representation and Scheduling of Subgraphs. In: Proc. of the 32nd Annual Intl. Symposium on Microarchitecture (MICRO32), Haifa, Israel, Dec. 1999

17 Bernstein D, Rodeh M. Global Instruction Scheduling for Superscalar Machines. In: Proc of the SIGPLAN Annual Symp., June 1991