

ORC 的全局指令调度技术

杨书鑫 张兆庆

(中国科学院计算技术研究所 北京 100080)

摘要 IA-64 是一种崭新的体系结构, 它为挖掘程序中潜在的指令级并行提供了丰富的硬件支持, 例如: 大寄存器组、(控制/数据)投机、谓词等。Itanium 是 IA-64 的一个具体实现。该文作者将 Bernstein 的基于超标量处理机的全局指令调度算法应用于显式并行 (EPIC) 的 Itanium 处理机上。在结合 Itanium 处理机特性的同时, 作者对 Bernstein 的算法有以下两点创新: (1) 应用层次化区域。相对于传统的扁平区域, 这样的区域具有很强的灵活性并提供了调度器大小合适的调度范围, 使其既能充分利用硬件资源又能够有效地控制调度的时间和空间开销。(2) 集成 P-Ready 指令调度。P-Ready 是在与 Bernstein 算法框架差异很大的上下文中提出的。P-Ready 指令调度能够把优先级高的指令尽早调度即使这条指令并没有在所有经过它的执行路径上解除数据依赖。集成 P-Ready 指令调度到 Bernstein 的算法框架上是十分有意义的。

作者在“基于 Itanium 处理机的开放源码编译器 ORC”中实现了该文介绍的算法, 实验结果显示全局指令调度器对 CPU2000int 基准测试例平均有 8.4% 的运行时加速比。作为应用层次化区域的优越性的一个反映, 调度指令跨越嵌套循环最高可取得 12.9% 的运行时加速比。此外, P-Ready 指令调度对 CPU2000int 的测试例平均有 1.37% 的运行时加速比, 最高可达 7.6%。

关键词 IA-64; Itanium; 全局指令调度; 层次化区域; P-Ready

中图法分类号 TP302

Global Instruction Scheduling Technique in ORC

YANG Shu-Xin ZHANG Zhao-Qing

(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080)

Abstract IA-64 is a novel architecture that provides ample hardware support to exploit instruction level parallelism, including large register file, data/control speculation, predicate etc. And it is exemplified by the Itanium processor. Authors adapt D. Bernstein's algorithm, which is targeted for superscalar processor, to EPIC processor Itanium. And authors take full advantage of Itanium specific details, and have two improvements to D. Bernstein's algorithm: (1) Apply hierarchical region structure into this algorithm. Comparing with the traditional flat structured scheduling region, hierarchical region structure is not only flexible but also has a capability to provide scheduler with appropriate sized scheduling scope. Hence, it renders possible for scheduler to take full advantage of ample hardware resource and effectively harness the compilation time and space. (2) Integrate P-Ready instruction scheduling. P-Ready is proposed under a context that is totally different with D. Bernstein's. Its usefulness stems from the fact that it can schedule a critical instruction as early as possible even if it is not ready on all execution paths. It is meaningful to integrate P-ready instruction scheduling into D. Bernstein's framework.

The algorithm proposed in this paper has been implemented in the open research compiler ORC which is a co-project between Intel Corp. and Institute of Computing Technology, CAS.

Authors' results show that the global scheduler achieves 8.4% runtime speedup on CPU2000int benchmarks. As an indicator of the advantage of applying hierarchical region structure, the technique that moves instruction across nested loops can obtain up to 12.9% runtime speedup. The integrated P-Ready instruction scheduling can obtain up to 7.6% speedup and 1.37% speedup on average for all CPU2000int benchmarks.

Keywords IA-64; Itanium; global instruction scheduling; hierarchically structured region; P-Ready

1 引言

指令调度是现代高性能编译器的关键技术。现代处理器一般具有丰富的硬件资源，然而应用程序（尤其是非科学计算的通用应用程序）的基本块通常很小，调度器只有调度指令跨越基本块的边界才能充分地发挥处理器潜在的指令级并行的能力。全局指令调度得到了广泛的研究。早期的全局指令调度局限于线性控制流图，例如文献[1~3]。这些基于线性控制流图的全局指令调度有一些共同的显著缺点：它们以牺牲较低频率的执行路径的性能来提高一条高频率执行路径的性能，但是由于缺少高频率和低频率执行路径性能得失的权衡，往往过犹不及，导致较低频率执行路径的性能损失过多而降低整体性能。相反地，由于基于非线性控制流的全局调度^[4~7]能够在更大的控制流上进行指令调度同时又能权衡不同路径上代码的性能，因而在理论和实践上能够取得比基于线性控制流的全局调度更好的性能。

文献[5]介绍的是一个基于非线性控制流的全局调度算法，该算法十分简洁。尽管文献[5]针对超标量处理器，但是它能够很方便地被改造为针对VLIW或EPIC^[8]处理器的算法。我们在ORC^[9]中所实现的全局调度的算法基本以文献[5]为框架。在充分利用目标处理器特性的前提下，我们还对文献[5]做了不少改进。本文主要介绍其中的两点：

(1) 应用层次化区域

传统的全局指令调度的区域结构是“扁平”的，即一个编译单元的所有调度区域构成全局控制流的一个划分。调度算法本身也对调度区域有一定的要求，例如文献[5]要求控制流只能从一个节点转移到该区域中。然而，满足调度算法要求同时又是“扁平”的区域往往偏小，不利于调度器挖掘程序中潜在的指令级并行性。例如在图1(a)中，由于无环调度不能跨越循环边界，{块3}单独构成一个区域 R_2 。剩下

的控制流{块1,块2,块4,块5,块6}无法构成文献[5]所要求的区域，这是因为控制流可以从多个基本块(即块1,5和6)到达这个区域。因此，调度器只能将剩下的区域划分为两个小的区域分别调度：即区域{块1,2,4,5}和区域{块6}。但是层次化区域^[10]通过将嵌套的区域 R_2 抽象成一个节点后就可以拥有较大的调度区域 R_1 ，如图1(b)所示。

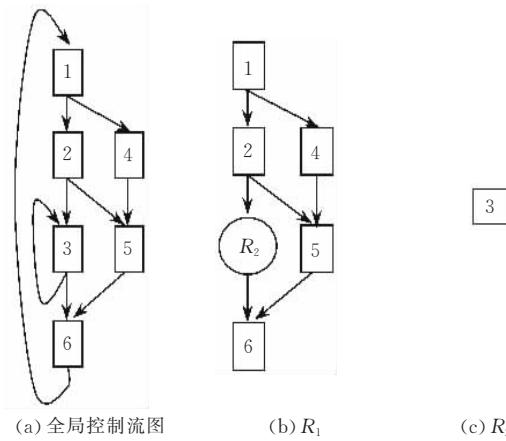


图 1 层次化区域

另一方面，如果区域较大的话，层次化调度区域还能够把大区域分割成若干大小合适的小区域（分割出去的小区域被层次化地嵌套到分割前的大区域中）。这一点对全局调度来说也很重要，因为，全局调度的算法复杂度一般都不是线性的，太大的调度区域必将带来过大的编译时间和空间开销。

由于层次化区域能够提供给调度器大小合适的调度范围，应用层次化区域到全局指令调度中是很有意义的。本文将讨论如何把层次化区域应用到文献[5]的调度框架中。

(2) 集成 P-Ready 指令调度

P-Ready 指令调度^[6]是一种指令级上的尾复制，并且被证明是十分有效的优化手段。它能够让优先级高的指令尽早地被调度即使这条指令并没有在所有经过它的执行路径上解除数据依赖关系。P-Ready 指令调度是在“滞后补偿”(differed compensation)

和“基于执行路径的依赖关系表示”的上下文中提出的,而文献[5]的代码补偿发生在指令调度的同时,其数据流和控制流的表示方法与文献[6]相去甚远,为此,本文介绍如何在文献[5]的框架上进行P-Ready候选指令的识别和调度。

本文第2节介绍理解该文必要的背景知识;第3节介绍如何把层次化区域应用到文献[5]的算法框架中;第4节描述了在文献[5]的算法框架内的P-Ready候选指令的识别与调度,受P-Ready概念的启发,作者还提出了非正交割集的其它两个应用;第5节简单介绍与ORC的全局调度相关的工作;第6节报告实验结果与分析;第7节总结全文。

2 背景知识

2.1 Itanium 处理机

Itanium 处理机^[1]是 IA-64 体系结构的具体实现。它通过显式并行(EPIC)及丰富的硬件资源支持高度的指令级并行能力。数据投机和控制投机的硬件支持帮助编译器突破控制依赖和存储依赖以减少内存延迟的影响。数据投机支持可以让 load 指令提前到与其 alias 的 store 指令之前执行。借助控制投机 unsafe load 也能够被控制投机而不会触发异常。不论是数据投机还是 unsafe load 的控制投机,编译器需要在恰当的地方插入 chk 指令以检查投机失败与否,如果投机是失败的,chk 指令跳转到恢复块重新执行 load 指令以及部分直接或间接流依赖于该指令的指令,然后再跳转到 chk 指令的下一条指令开始执行。从行为上看 chk 指令是一种分支指令。谓词(predicate)使指令可以条件执行,编译器可借助谓词通过 If-conversion 将控制依赖转变为数据依赖,从而消除部分分支指令。允许在一个机器周期内同时发射多条分支指令的 Multiway branch 技术可以用来减小关键路径的长度。Itanium 有很大寄存器组,这使得调度器可以在较大的区域内进行全局指令调度而不会导致过多的寄存器溢出(spill)。

相邻的停止位(stop-bit)之间的指令序列是一个指令组(instruction group),Itanium 处理机在一拍内同时执行一个指令组中的所有指令。指令组包含的指令个数是不固定的。3 条连续的指令被拼装成一个指令束(bundle)其类型必须是既定的一组模板(template)之一。为了拼装指令束,调度器有时要在指令之间插入 nop。停止位可处在指令束中相邻

的指令之间或最后一条指令的末尾。

2.2 术语

为了说明方便,引入两个定义。

定义 1. 设单入口控制流图为 $G = \langle N, E \rangle$, 其中 N, E 分别为结点集和边集, $x, y \in N$, x 可达 y , N' 是 N 的子集。如果从 G 中去除 N' 中的所有结点以及与这些结点相连接的边, x 不再可达 y , 则称 N' 是 x 和 y 之间的割集, 记作 $CS(x, y)$ 。

定义 2. 设单入口控制流 $G = \langle N, E \rangle$, N 和 E 分别是结点集和边集, R 是根结点。 $s \in N$, N', N'' 均是 N 的子集。对于 N' 中任何一个可达 s 的元素 p , N'' 是 p 与 s 的割集, 则称 N'' 是 N' 与 s 之间的集割集, 记作 $SCS(N', s)$ 。

满足如下条件的割集 $CS(x, y)$ (集割集 $SCS(N', s)$) L 分别称作正交的和极小的:(1) L 中的任意两个元素 $p, q (p \neq q)$, p 和 q 相互不可达。(2)对于 L 中的任意一个元素 p , $L - \{p\}$ 不再是 x 和 y 之间的割集(不再是 N' 和 s 之间的集割集)。

根结点与结点 s 之间的且包含结点 m 的极小正交割集即为文献[5]提出的 $SISS(m, s)$ 。在控制流中没有 JS 边(即源于出度大于 1 的结点且指向入度大于 1 的结点的有向边,作者认为 critical edge 更标准些)的情况下,若 m 可以到达 s , $SISS(m, s)$ 一定是存在的。 $SISS(m, s)$ 可能有多个实例,文献[5]的调度算法要求具体调度所用的割集必须是 $SISS(m, s)$ 所有实例中离 s “最近”的一个,本文亦然。关于“最近”可以形式定义为:在 $SISS(m, s)$ 的多个实例中,如果实例 x 是任何其它实例与 s 之间的集割集,则 x 是距离 S “最近”的一个实例。

此外,为了让本文尽量“自包含”,引用两个定义。

定义 3. 若指令 i 所数据依赖的任何一条指令 j 要么已经被调度要么从基本块 T 不可达 j 所在的基本块(任何基本块可达它自己),则指令 i 相对于基本块 T M-Ready^[5]。

定义 4. 如果基本块 S 中的指令 i 不相对于基本块 T M-ready, 并且至少存在一条从 T 到 S 的路径 p , i 所数据依赖的且尚未被调度的指令不在 p 上的任何一个基本块中, 则指令 i 相对于基本块 T P-Ready^[6]。

2.3 Bernstein 全局调度框架

Bernstein 全局调度框架如下:

给定一个单入口多出口区域 R , Bernstein 通过以下步骤对 R 进行全局指令调度:

1. 通过 edge-splitting 消除 R 中所有的 JS 边;
2. 创建 PDG^[12] 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区域中的每一个结点 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, 从候选指令列表中选出“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. 消除空基本块(通过 edge-splitting 产生的).

3 应用层次化调度区域

3.1 概述

传统的全局指令调度的区域结构是扁平的, 即一个编译单元的所有调度区域构成全局控制流图的一个划分. 然而, 为了使区域之间不重叠, 同时又要满足调度算法对区域形状的特定要求(例如文献[5]要求区域必须是单入口的), 区域的面积往往偏小. 另一方面, 全局调度算法的复制性往往是非线性的, 如果调度的区域太大那么编译时间和空间将十分可观.

层次化区域^[10]能够有效地控制调度区域的大小. 它除了顶层区域外(它代表整个编译单元的全局控制流), 任何一个区域嵌套并且仅嵌套于另一个区域(称为该区域的直接外层区域). 循环被表示成一个区域(该区域可能包含若干个嵌套区域)并且忽略其回边. 嵌套的循环区域在其直接外层区域中表示成一个抽象结点. 在图 1 中, 图 1(a) 的内层循环被抽象成区域 R_2 且在其直接外层区域 R_1 中表示为一个抽象结点(用圆表示). 所有循环的回边均被忽略.

在本文中, 我们用“区域内”指一个区域所包围的控制流图, 而用“区域上”指一个区域所包围的控制流图但是不包括它所有的嵌套区域所包围的控制流图. 例如在图 1(b) 中, “ R_1 上”有 5 个基本块, “ R_1 内”则有 6 个.

关于层次化区域构造的细节详见文献[10]. 本节仅从调度的角度出发介绍如何把层次化区域应用到以文献[5]为框架的调度算法中. 作者认为层次化

区域不仅能应用于文献[5]的情况, 对许多全局调度算法都具有普遍意义.

区域的构造可以由一个独立的阶段予以实现, 而不是在指令调度的同时进行(例如文献[1]的情况). 这样既减小了调度器的复杂性也提高了编译器的模块化.

3.2 调度区域的控制流变形

本节讨论把层次化区域应用于文献[5]算法中, 需要对调度区域作用哪些变形.

除了需要通过 edge-splitting 消除 JS 边外^[5], 在调度之前, 还要对层次区域作用以下变形.

(1) 块 S 中的指令 i 被向上调度到块 T 后, 还要将补偿代码放置 $CS(S, T) - \{T\}$ 中的每一个基本块末尾以保持语义. 由于嵌套区域是一个抽象结点, 补偿代码不能放置到嵌套区域中, 因此, 我们应该确保 $CS(S, T) - \{T\}$ 中没有元素为嵌套区域. 为此, 调度之前调度器应通过 edge-splitting 消除源于嵌套区域而指向入度大于 1 的基本块的有向边. 于是, 若 $CS(S, T) - \{T\}$ 中存在一个元素为嵌套区域, 设为 R , 直接用 R 的所有直接后继代替之, 直到 $CS(S, T) - \{T\}$ 不再有元素为嵌套区域. 例如在图 1(b) 中, 嵌套区域 R_2 和块 6 之间应该插入一个空基本块(设为块 7), 于是若嵌套区域 $R_2 \in CS(S, T)$, 我们可以用 $CS'(S, T) = (CS(S, T) - \{ \text{嵌套区域 } R_2 \}) \cup \{\text{块 } 7\}$ 替换之.

(2) 向上指令调度的补偿代码是放置到基本块的“末尾”的: 如果基本块没有分支指令, 则补偿代码作为该基本块的最后一条指令, 否则把它插在分支指令之前. 注意到补偿代码与分支指令在调度前后位置是颠倒的, 为此调度器必须维护两者之间的数据依赖关系. 在 Itanium 处理机上条件转移指令以谓词变量作为其操作数, 而补偿代码可能定义同名的操作数, 于是两者之间存在反依赖. 这种情况出现得较少, 尤其在寄存器分配之前. 调度器可以通过 Renaming 等手段解决上述问题. 函数调用指令通常被视为分支指令, 且标志着一个基本块的结束^[13]. 因此如果分支指令为函数调用, 补偿代码也应该插在函数调用之前, 然而这个函数可能和补偿代码 alias, 这将导致错误的结果. 类似于(1), 我们在以函数调用结束的基本块及其入度大于 1 直接后继之间插入一个空基本块. 例如在图 1(b) 中若块 4 以函数调用结束, 调度器应在块 4 和 5 之间插入一空块.

在指令调度的过程中, 调度器不再对控制流进

行变形以避免增量式的控制流信息维护所带来的巨大困难.

3.3 调度指令跨越嵌套区域

文献[5]通过不停执行以下过程完成对整个区域的调度:识别 M-Ready 候选指令,调度 M-Ready 候选指令,维护 M-Ready 候选指令队列.本节要回答的问题是:当一个区域中包含有嵌套区域时,如何执行上述过程?

(1) 区域概要信息

首先,我们必须在包含有嵌套区域的情况下识别出 M-Ready 候选指令.一个比较直观的做法是将嵌套区域 R_x 想象成一个仅含一条指令的基本块,而这条指令:

①引用 R_x 内所有指令所引用的寄存器变量;

②定义 R_x 内所有指令所定义的寄存器变量;

③读访问 R_x 内所有 load 指令所访问的内存单元;

④写访问 R_x 内所有 store 指令所访问的内存单元;

⑤它可能的副作用(side effect)是 R_x 内所有指令副作用的集合.

上述的“想象”“消除”了区域中的嵌套区域,从而可以直接利用 M-Ready 的定义在含有嵌套区域的区域中识别出 M-Ready 候选指令.

在具体实现上,我们将上述“想象”的指令所读写的寄存器和内存单元以及副作用的情况用一个数据结构——区域概要信息(summary)予以描述.注意到我们的调度器是按从内层到外层的顺序调度区域的.当一个区域 R_x 调度完毕之后立即收集它的概要信息,在调度 R_x 外层区域时嵌套区域的概要信息已经就绪.此外,为了实现上的统一和方便,上述“想象”的指令可以用一条哑(dummy)的物理指令在控制流上表示出来.同时它和其它指令的依赖关系也同步地在数据依赖关系图(DDG)上表示出来.哑指令永远不会被调度,并且在全局调度完成后被删除.

区域概要信息除了用来描述对应区域内的数据流情况,还可以携带更多的信息描述对应区域其它方面的特性.例如,描述区域的入口和出口处指令的未竟(dangling)延迟的情况,从而在调度外层区域的时候能够有效地防止调度器将一些指令调度到离嵌套区域太近的地方导致流水线在嵌套区域的边界处暂停,详见文献[4].

(2) 控制寄存器压力

由于我们的全局调度发生在寄存器分配之前,

所以盲目地调度指令跨越一个“大”的嵌套区域可能会显著地增加寄存器压力,从而导致不必要的 spill 和 restore 代码反而降低了性能.

以下规则可以避免这种情况发生:设有 n 个寄存器变量被指令 i 所引用并且在 i 之后不再活跃,另设 i 定义的寄存器变量个数为 m (在这里,我们假定在全局调度的时候,程序中不再有死代码,即每一个变量都会被引用),

①仅当 $m \leq n$ 才允许 i 向上(即逆着控制流方向)跨越过“大”嵌套区域.

②仅当 $n \leq m$ 才允许 i 向下(顺着控制流方向)跨越过“大”的嵌套区域.

③若区域并不“大”,不考虑寄存器压力.

不难得知,在规则①,②下,调度指令跨越“大”区域前后,相互干涉的 live range 的个数没有增加,因而也不会增加寄存器压力.由于不“大”的嵌套区域不会引起严重的寄存器压力问题,因此不考虑寄存器压力因素反而可以增加可以调度的指令个数,提高了调度质量.

另一方面,即使违反规则①,②,有时也不会增加寄存器压力.这是由于随着调度的进行,跨越嵌套区域的 live range 的个数可能会减少.以向上调度为例,设指令 j 引用 $m' = m - n + 1$ 个 i 所定义的寄存器变量,另设 j 继 i 之后被调度跨越嵌套区域.于是,在调度 j 之后,由调度 i 所引起的跨越嵌套区域的 live range 个数增加为 $m - m' - n = -1$.在这个例子中,违反规则①反而降低了寄存器压力.

对于“大”嵌套区域我们比较保守,即确保每一次调度都不会(在调度那一个时刻)增加跨越嵌套区域的 live range 的个数,而不期望这个数目会随着调度的进行随之减小.

至于嵌套区域的“大小”可以用该区域所包含的相互干涉的 live range 个数量化地定义,但是这样不仅大大增加了实现的难度,而且大大增加了调度的时空开销.我们可以用区域中指令中的个数近似地表示其“大小”,并通过实验确定“大区域”与“小区域”的边界.嵌套区域“大小”的信息纪录在它的区域概要信息数据结构上.

(3) 调度指令跨越嵌套区域

区域概要信息把嵌套区域所蕴含的数据依赖转化成显式的指令之间的数据依赖.3.2 节介绍的控制流变形确保了用于代码补偿的每一个结点都是基本块.通过上述两个手段,嵌套区域对调度器来说变得“透明”,调度器可以完全像文献[5]所介绍的那样

调度 M-ready 指令.

4 集成 P-Ready 调度

4.1 P-Ready 候选指令的识别和调度

P-Ready 调度被视为指令级尾复制并且被证明是十分有效的^[6]. 文献[6]的算法框架和本文所基于的算法框架^[5]相去甚远:前者以 Wavefront 为一个基本调度单位而后者逐个调度区域中的基本块;前者“滞后补偿”(differed compensation)而后的代码补偿发生在指令调度的同时;前者用执行路径位向量表示数据依赖和控制依赖而后者则采用 PDG^[12]. 不言而喻,在不同的算法框架内 P-Ready 候选指令的识别和调度是不同的(两者统称为 P-Ready 调度,下同,不再赘述). 下文将介绍在文献[5]的算法框架内的 P-Ready 调度算法.

为了说明方便,先定义几个函数:

① 定义 $H(i)$ 为指令 i 所在的基本块;

② 定义 $UOPs(i, T)$ 为集合 $\{p \mid p \text{ 尚未被调度} \wedge \text{基本块 } T \text{ 可达 } H(p) \wedge i \text{ 数据依赖于 } p\}$;

③ 定义 $HS(i, T)$ 为集合 $\{H(p) \mid p \in UOPs(i, T)\}$;

$UOPs(i, T)$ 可以形象地理解为:指令 i 逆着块 T 到 $H(i)$ 的所有执行路径向上运动,在这运动过程中所遇到的 i 所数据依赖的指令的集合.

设在控制流图 G 中块 S 中的指令 i 不相对于块 T M-Ready,于是 $UOPs(i, T)$ 和 $HS(i, T)$ 都不是空集. 如果 $T \in HS(i, T)$ 或 $HS(i, T)$ 是 T 和 $H(i)$ 之间的割集,那么指令 i 不相对于块 T P-Ready(因为任何一条从 T 到 $H(i)$ 的路径上都有指令 i 所数据依赖的且尚未调度的指令).

否则,根据定义 4,指令 i 相对于块 T P-Ready. 对于 P-Ready 候选指令,调度器首先需要决定调度该指令是否“有益”,如果回答是肯定的才决定如何调度它. ORC 全局调度所采用启发性方法与文献[5]相差很大,我们将另文介绍. 限于篇幅,在此我们仅讨论如何调度一条 P-Ready 指令.

关于调度一条 P-Ready 指令只需要决定补偿代码的放置位置就行了. 我们把放置 P-Ready 候选指令的补偿代码的基本块划分成两类:

(1) S_1 . 为了保持 i 和 $UOPs(i, T)$ 中的指令之间的数据依赖,必须把代码补偿到 S_1 中每一个基本块末尾. 其中, S_1 是 $HS(i, T)$ 和 $H(i)$ 之间的集割集.

(2) S_2 . 为了保证从根结点到 $H(i)$ 的每一条路径上指令 i 至少被执行一次,必须放置补偿代码于

S_2 中的每一个基本块的末尾.

S_1 必须满足如下 3 个条件:(1) 它不应该是 T 和 $H(i)$ 之间的割集;(2) 我们还要求 S_1 是正交的以减小复杂性;(3) S_1 应该是极小的,这样使得被调度到 T 的指令 i (定值) 到达 $H(i)$ 的概率较大,从而调度 i 的“收益”也较大.

S_1 一定存在,并且可能有多个实例(证略). 我们构造 S_1 的过程十分简单:首先,从 $H(i)$ 逆着控制流方向遍历控制流图直到访问到一个 joint 结点 J (如果 $H(i)$ 本身就是 joint 结点,遍历终止);接着,试图从集合 $P = \{J \text{ 的直接前驱}\}$ 中找出一个 S_1 实例——它是 P 的真子集(如果寻找的结果是失败的,则简单地认为指令 i 不相对于 T P-Ready). 由于 JS 边在调度之前已经被消除, P 中的任意两个结点必然互相不可达(证略),所以若 S_1 是 P 的真子集,它一定是正交的. 为了满足上述的 S_1 的条件(1)和(3),在满足 S_1 的 3 个条件下,构造过程不停从中删除一些结点,直到不能再删除为止.

从文献[5]可知,放置补偿代码到 $S = SISS(T, H(i)) - \{T\}$ 可确保从根结点到 $H(i)$ 的每一条路径上指令 i 至少被执行一次. 因为调度器将把代码补偿到 S_1 中的每一个基本块中,所以没有必要再把代码补偿到这些基本块中: $\{x \mid x \in SISS(T, H(i)) \wedge S_1 \text{ 是 } x \text{ 和 } H(i) \text{ 之间的割集}\}$. 于是,我们有

$$S_2 = SISS(T, H(i)) - \{T\} - \{x \mid x \in SISS(T, H(i)) \wedge S_1 \text{ 是 } x \text{ 和 } H(i) \text{ 之间的割集}\} \quad (1)$$

以图 2 为例,设 T 为基本块 1. 指令 ② 相对于 T P-Ready. $UOPs(\text{②}, \text{块 } 1) = \{\text{①}\}$. $HS(\text{②}, \text{块 } 1) = \{\text{块 } 4\}$. 构造 S_1 的过程是这样的:由于离 $H(\text{②})$ 最近的 joint 结点是块 5,所以 $P = \{\text{块 } 5 \text{ 的直接前驱}\} = \{\text{块 } 3, \text{块 } 4\}$;构造过程试图从 P 中找出一个真子集作为 S_1 ,显然 $S_1 = \{\text{块 } 4\}$ 即为所求.

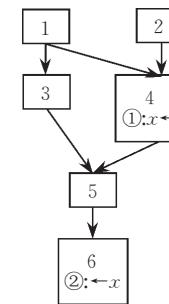


图 2 S_1 构造示意(方框表示基本块,方框内的数码表示基本块标识. 指令 ① 和 ② 分别定义和使用变量 x)

此外,由于 $SISS(\text{块 } 1, \text{块 } 6) = \{\text{块 } 1, \text{块 } 2\}$,且由于 S_1 是块 2 和块 6 之间的割集,所以,根据式(1)

即可求出 $S_2 = SISS(\text{块 } 1, \text{块 } 6) - \{\text{块 } 1\} - \{\text{块 } 2\}$, 即空集. 于是调度器这样调度 P-Ready 指令②: 将②移到块 1 中, 同时, 作为补偿代码, 将②复制到 $S_1 \cup S_2$ (即集合{块 4})中每一个基本块的末尾.

P-Ready 候选指令的识别如算法 1 所示. 算法 1 判断指令 i 是否相对于区域 R 中的基本块 T P-Ready. 若答案是肯定的, 算法 1 返回补偿代码所在基本块的集合(即 $S_1 \cup S_2$), 否则返回空集.

算法 1. BBS.

```
Is_P-Ready(INSTRUCTION i, BB T, REGION r)
输入:  $i$ : 指令;  $T$ : 基本块;  $r$ : 调度区域, 块  $T$  和指令  $i$  都在  $r$  上
输出: 如果指令  $i$  相对于块  $T$  P-Ready, 返回补偿代码所在基本块的集合, 否则返回空集
```

```
{
     $b \leftarrow H(i);$ 
    while( $b$  只有一个前驱) {if( $b = T$ ) then {return 空集}
    else { $b \leftarrow b$  的前驱};}
    if( $T$  是  $b$  的直接前驱) then{return 空集;}
     $K \leftarrow \{b\}$  的直接前驱结点;
    for each  $x \in K$  {
        if( $K - \{x\}$  是  $HS(i, T)$  和  $H(i)$  之间的集割集)
            then { $K \leftarrow K - \{x\}$ ;}
    } /* end of for */
    /* 现在  $K$  即为  $S_1$  */
    if( $K$  不是  $T$  和  $H(i)$  之间的集割集) then{
         $K' \leftarrow K;$ 
        for each  $y \in SISS(T, H(i)) - \{T\}$  {
            if( $K'$  不是  $y$  和  $H(i)$  之间的集割集) then
                { $K \leftarrow K \cup \{y\}$ ;}
        } /* end of for */
    } else { $K \leftarrow \emptyset$ }
    return  $K$ ;
}
```

4.2 应用非正交割集的其它优化

从定义上看, P-Ready 候选指令一定不是 M-Ready 的, 反之亦然. 但有些情况下如果把 M-Ready 候选指令“像 P-Ready 候选指令一样被调度”会收到很好的效果. 这里所谓的“像 P-Ready 候选指令一样被调度”指的是来自块 S 的候选指令 I 的补偿代码所在的基本块和目标基本块构成根结点和 S 之间的非正交割集.

(1) 减小寄存器压力. 如图 3(a)所示, 设 R_1 是一个很大的嵌套区域, 块 3 中的指令 i 相对于块 1 M-Ready. 由于 $SISS(\text{块 } 1, \text{块 } 3) = \{\text{块 } 1\}$, 调度 i 可以不需要代码补偿. 但是 i 定义的 live range 和穿越

或源于 R_1 的所有 live range 干涉, 而 R_1 又是一个很大的区域, 调度 i 将引起较大的寄存器压力. 我们可以“像调度 P-Ready 候选指令”一样调度 i : 即把它移到块 1 中, 同时复制一份补偿代码到块 2. 上述调度方法使 i 尽早得到调度且避免了跨过大嵌套区域所带来的寄存器压力.

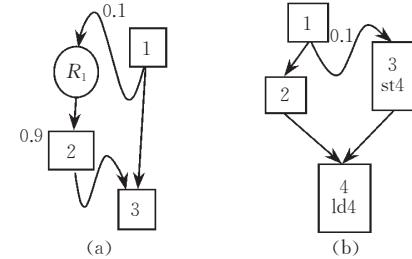


图 3 调度过程示意(方框表示基本块, 圈表示嵌套区域, 方框内的数码表示基本块标识, 其它文字为基本块内的指令)

(2) 避免插入 chk 指令. 在图 3(b)中, 尽管 ld4 和 st4 可能 alias, 在数据投机的硬件支持下调度器仍然可以视 ld4 相对于块 1 M-Ready, 并将它数据投机到块 1(改变其形式为 ld4.a)同时在 ld4 原来的位置上插入 chk 指令. 考虑到路径 1→2→4 执行的频率较高(块 1 到块 2 的概率为 0.9)而另一条路径的性能可以牺牲, 调度器可将 ld4 调度到块 1(不改变其形式)同时补偿到块 3 的末尾, 这样就可以不用插入 chk.

5 相关工作

调度器除了要维持指令之间的数据依赖以及控制依赖以外还要保证不能违反结构依赖. 传统的做法是通过 Resource Reservation Table^[15] 管理资源占用情况, 这种做法往往将硬件细节分散在调度器的各个地方, 给代码移植带来困难. 此外, 由于 Itanium 处理机硬件复杂性要求更先进的资源管理. 关于 ORC 中的全局指令调度的资源管理方法, 读者可参阅文献[16].

对 load 指令的数据投机和控制投机是 Itanium 处理机的一个重要特色. ORC 的调度器采用文献[17]提出的算法框架. 文献[17]使得恢复块的生成可以在调度之后离线进行. 从而避免了调度器在调度的同时维护控制流及其相关信息(例如 dominate 和 post-dominate).

全局调度前后, 控制流差别十分巨大: 调度器不仅在调度之前通过 edge-splitting 增加了许多空基本

块,在调度的同时会将一些本来非空的基本块全部(或只留下分枝指令)调度出该基本块;此外,数据投机和控制投机也要求在控制流中插入一些“恢复块”. 我们并不像文献[5]一样仅仅在调度完一个区域后消除空基本块,而是在整个编译单元都调度完后,用一个独立的控制流优化器优化控制流. 控制流优化器的一个重要工作是重新安排基本块的位置. 基本块放置的算法以文献[18]为原型.

6 实验结果

6.1 实验方法

作者在中科院计算技术研究所和英特尔公司的合作项目——基于 Itanium 处理机的开放源码编译器 ORC^[9]中实现了本文介绍的算法. ORC 采用的一系列优化技术充分发挥了 IA-64 体系结构的指令级并行的能力. 它派生于 SGI 公司的开放源码编译器 Pro64, 而后者是从 SGI 公司的产品编译器 MIP-Spro 移植来的. ORC 前端支持 C, C++, Fortran 77/95, 其中 C 和 C++ 的前端支持 GNU C 的方言; 中端包括①过程间分析与优化, ②循环嵌套优化(LNO) 和 ③全局标量优化; 后端包括代码生成(CG). 基于符号表的中间表示 WHIRL 作为从前端到后端各个阶段通信的“载体”.

指令调度和寄存器分配是 ORC 代码生成阶段最重要的阶段. 在 ORC 中, 指令调度有三种: 即软流水调度(software pipeling)和本文介绍的无环全局调度(GLOS)和局部调度(LOCS).

软流水调度安排在全局调度之前, 目前它只能调度单基本块的循环体. GLOS 安排在软流水调度之后, 寄存器分配之前. GLOS 忽略被软流水调度过的循环(所对应的区域). 在寄存器分配之后, ORC 通过 LOCS 对那些在 GLOS 之后, 指令发生变化(主要是由于寄存器分配时产生的 spill 和 restore 代码)的基本块重新进行调度. 在 ORC 中 LOCS 被看作是 GLOS 特例, 它事实上和 GLOS 共享相同的代码.

SPEC2000 的整型基准测试程序^①被用来评估全局调度器的性能. 运行基准测试程序的 Itanium 工作站频率为 733Mhz, 具有 2MB L3 cache 和 1GB 的内存, 运行的操作系统是 RedHat 7.2/IA-64.

我们用于评估某一项优化 x (例如 P-Ready 指令调度)对测试例 c 的加速比是这样计算的:

(没有作用 x 时 c 的运行时间 - 作用 x 后 c 的运行时间)/没有作用 x 时 c 的运行时间 $\times 100\%$.

6.2 实验结果和实验分析

(1) 全局调度器整体的有效性

表 1 的第 2 列罗列了整个全局调度器加速比. 全局调度器基本对所有的例子的性能都有显著的提高, 除了 eon 和 mcf 以外. eon 类似于用于科学计算的浮点程序: 它的基本块十分大(大多基本块具有几十到近 300 条指令), 因此基本块内部就有较高的指令级并行度, 全局调度对它的性能提高毫无帮助. 从静态的汇编代码上看, 全局调度对 mcf 也有比较明显的提高. 但实际上严重的 D-cache miss 破坏了全局调度所可能取得的指令级并行上的提高. 已有研究^[19]表明, 通过 stride prefetch 减小 D-cache miss 可将 mcf 的性能提高 65%.

调度器对 gzip 是十分有效的, 原因在于 gzip 有一个执行频率很高的热点(hot spot)函数 longest-match(), 在该函数中有一个执行频率很高的循环, 而这个循环不适合软流水调度并且这个循环包含的基本块个数较多. 全局调度能够比较容易地在这个函数中取得比较明显的加速比, 从而对整个测试例有比较明显的加速比.

表 1 运行时加速比

测试例	全局调度 加速比(%)	跨越嵌套循环 加速比(%)	P-Ready 加速比(%)
bzip2	1.4	0.2	-0.91
gzip	23.31	12.90	7.6
vpr	6.35	1.07	0.80
crafty	16.61	-0.7	2.51
Parser	5.32	0.5	1.1
mcf	0.2	0	0
Vortex	6.8	0.4	2.5
Gap	10.94	0.53	-0.56
Twolf	8.45	0.2	0
eon	-0.51	-0.43	-1.27
perlbench	13.42	0.49	1.45
Gcc	8.75	1.87	1.49
平均	8.403	1.41	1.37

(2) 应用层次化调度区域的有效性

基于层次化区域的全局调度的性能在很大程度上依赖于基于层次化区域构造的质量. 关于基于层次化区域构造的方法读者可以参考文献[10]. 表 2 显示作为全局调度器输入的层次化区域上包含的基本块和指令的平均数. 从表 2 可以看出每一个测试例的层次化区域的大小比较接近.

^① Stan Performance Evaluation Corporation. <http://www.spec.org/cpu2000>, 2004

表 2 调度区域的大小

测试例	平均基本块#	平均指令#
bzip2	6.81	33.86
Crafty	6.27	39.8
eon	5.22	30.33
Gap	7.93	40.3
Gcc	7.16	31.13
gzip	6.74	33.07
mcf	6.56	34.83
Parser	6.15	24.56
perlbench	6.99	32.38
Twolf	7.15	44.55
Vortex	8.68	48.19
vpr	7.26	40.54

检验应用层次化区域有效性最可信的方法是将基于层次化区域与基于扁平区域的两种调度器的质量进行比较。由于时间和资源所限,作者尚未实现后者。在这里我们用“调度指令跨越嵌套区域”这项技术的有效性从一个侧面反映应用层次化调度区域的有效性。上文提到过,层次化区域中的“区域”有可能对应一个循环,有些不是。这些不对应循环的区域有时是从大的区域中分割出来,用来控制编译时间和空间开销的。而扁平调度区域中虽然无法包含循环,但是它可能覆盖一个尽可能大的无环控制流图。为此,我们只评估“调度指令跨越嵌套循环区域”而不是“调度指令跨越任意嵌套区域”。

该项技术的加速比如表 1 第 3 列所示。

这项技术除了对 vpr 和 gzip 以外,基本对所有的测试例都没有比较明显的帮助。Vpr 充斥着小循环,区域中的基本块常常被对应于小循环的嵌套区域所分割,所以“调度指令跨越嵌套区域”能够比较明显地加大区域中小循环两边的基本块的并行度从而提高了 vpr 的性能。上文提到 gzip 的热点函数 *longest_match()* 有一个执行频率很高的循环。这个循环还包含一个嵌套循环。调度器正是因为在外层循环中调度指令跨越它的嵌套循环才取得显著的加速比的。

(3) P-Ready 指令调度的有效性

如表 1 第 4 列所示,P-Ready 指令调度对大多数测试例都有比较明显的性能提高。其中对 gzip 的加速比最为显著,这是因为在 gzip 的热点函数中有 P-Ready 指令调度的机会。但是也有个别例子的性能反而下降了。

利用动态运行时间作为测量加速比的手段(而不是从汇编代码为基础静态地计算加速比)有一个非常显著的优点:它能够比较客观且综合地评价一个优化手段的机会和效果。但是它也有显著的缺点

在于它容易受其它因素的干扰,我们看到有些测试例的性能反而下降了。整体地说,P-Ready 调度对大多数测试例有普遍而显著的提高,而跨越嵌套区域的调度仅对个别例子有十分显著的提高。

7 结 论

作为 EPIC 体系结构实例的 Itanium 处理机为编译器提供了丰富的硬件支持。全局调度器必须在比较大的区域内调度指令才能够充分利用硬件资源,缩短执行路径从而提高调度质量。对于全局调度来说,层次化区域相对于传统的扁平区域具有明显的优点:它不仅能够提供比较大的范围以利于调度器挖掘程序潜在的指令级并行性,它还能够有效地防止区域过大引起不合理的编译时空开销。因此,应用层次化区域到全局调度中是很有意义的。调度指令跨越嵌套循环是这个意义的一个反映。我们的实验表明调度指令跨越嵌套循环对 gzip 有 12.9% 加速比。P-Ready 调度是指令级的尾复制。它可以让优先级高的指令尽早被调度尽管这条指令并没有在所有执行路径上解除数据依赖。实验显示作者提出的基于文献[5]的 P-Ready 调度对 SPEC2000int 测试例平均有 1.37% 的运行时加速比,特别地,对 gzip 有 7.6% 的加速比。

参 考 文 献

- 1 Fisher J. A.. Trace scheduling: A technique for global micro-code compaction. IEEE Transaction on Computers, 1981, 30 (7): 478~490
- 2 Hwu W. W. et al.. The superblock: An effective technique for VLIW and superscalar compilation. Journal of Supercomputing, 1993, 7(1): 229~248
- 3 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: Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, 1992, 45~54
- 4 Bernstein D. , Rodeh M.. Global instruction scheduling for superscalar machines. In: Proceedings of the SIGPLAN Annual Symposium, Toronto, Ontario, 1991, 241~255
- 5 Bernstein D. , Cohen D. , Krawczyk H.. Code duplication: An assist for global instruction scheduling. In: Proceedings of the 24th Annual International Symposium Microarchitecture (MICRO24), Albuquerque, New Mexico, Puerto Rico, 1991, 103~113
- 6 Bharadwaj J. , Menezes K. , McKinsey C.. Wavefront schedu-

- ling: Path based data representation and scheduling of subgraphs. In: Proceedings of the International Symposium Micro-architecture (MICRO32), Haifa, 1999, 262~271
- 7 Fisher J. A.. Global code generation for instruction-level parallelism: Trace scheduling-2, Hewlett-Packard Laboratories, Palo Alto, USA: Technical Report HPL-93-43, 1993
- 8 Schlansker M. S., Rau M. S.. EPIC: Explicitly parallel instruction computing. IEEE Computer, 2000, 33(2):37~45
- 9 ORC team, ORC suite. <http://sourceforge.net/projects/ipf-orc>, 2001~2004
- 10 Liu Yang, Zhang Zhao-Qing, Qiao Ru-Liang. A region based compilation framework. Chinese Journal of Computers, 2003, 26(2): 188~194(in Chinese)
(刘 畅, 张兆庆, 乔如良. 基于域的编译框架. 计算机学报, 2003, 26(2): 188~194)
- 11 Intel Corp. <http://www.intel.com/design/itanium/family/>, 2004
- 12 Ferrante J., Ottenstein K. J., Warren J. D.. The program dependence graph and its use in optimization. ACM Transaction of Programming Languages and Systems, 1987, 9(3): 319~349
- 13 Muchnick S. S.. Advanced Compiler Design and Implementation. San Francisco, California: Morgan Kaufmann Publishers, 1997, 174
- 14 Abraham S. G. et al.. Meld scheduling: Relaxing scheduling constraints across boundaries. In: Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, France, 1996, 308~321
- 15 Eichenberger A., Davidson E.. A reduced multipipeline machine description that preserves scheduling constraints. In: Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation, Philadelphia, Pennsylvania, 1996, 12~22
- 16 Chen Dong-Yuan et al.. Efficient resource management during instruction scheduling for the EPIC architecture. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, Louisiana, 2003, 36~45
- 17 Ju Roy et al.. A unified compiler framework for control and data speculation. In: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, Philadelphia, Pennsylvania, 2000, 157~168
- 18 Pettis K., Hansen R. C.. Profile guided code positioning. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, New York, 1990, 16~27
- 19 Wu You-Feng. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, 2002, 210~221



YANG Shu-Xin, born in 1975, Ph. D. candidate. His research interest is in advanced compilation technology.

Background

This paper is a part of work of Open Research Compiler (ORC) project. The objective of this project is to provide a leading open source ItaniumTM Processor Family (IA-64) compiler infrastructure to the compiler and architecture research community.

The design of ORC stresses on the following aspects: compatibility to other open source tools, robustness of the entire infrastructure, flexibility and modularity for quick prototyping of novel ideas, and leading performance among Ita-

ZHANG Zhao-Qing, born in 1938, professor, Ph. D. supervisor. Her research interests include parallel compilation, environment for parallel programming and automatic parallelization.

niumTM Processor Family open source compilers.

ORC is a collaboration between Intel Corp. and Advanced Compilation Group of Institute of Computing Technology of Chinese Academy of Sciences. The former is a industry giant and the later is a leading compilation technology research group in China.

This paper describes the global acyclic instruction scheduling in ORC compiler. The experiment indicates global scheduler in ORC compiler has very good performance.