避免模调度中 cache 代价的优化方法*

刘 利[1+], 李文龙[2], 郭振宇[1], 李胜梅[1], 汤志忠[1]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](Intel 中国研究中心 编译组,北京 100080)

Optimization to Prevent Cache Penalty in Modulo Scheduling

LIU Li[1+], LI Wen-Long[2], GUO Zhen-Yu[1], LI Sheng-Mei[1], TANG Zhi-Zhong[1]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Compiler Group, Intel China Research Center Ltd, Beijing 100080, China)

+ Corresponding author: Phn: +86-10-62774739, E-mail: liuli03@mails.tsinghua.edu.cn, http://www.tsinghua.edu.cn

Received 2005-01-06; Accepted 2005-05-12

Liu L, Li WL, Guo ZY, Li SM, Tang ZZ. Optimization to prevent cache penalty in modulo scheduling. *Journal of Software*, 2005,16(10):1842−1852. DOI: 10.1360/jos161842

Abstract: Software pipelining can speedup the loop execution. Modulo scheduling is a widely used heuristic for software pipelining. Cache hierarchies can improve memory systems, but probably all processor implementations introduce additional delays (cache penalty). This paper demonstrates possible cache penalty due to modulo scheduling, and presents an algorithm named Prevent Cache Penalty in Modulo Scheduling (PCPMS), which can prevent cache penalty due to modulo scheduling. Experimental results show that PCPMS can prevent cache penalty and improve the performance of programs.

Key words: software pipeline; modulo scheduling; memory optimization; cache penalty

摘 要: 软件流水能够加快循环的执行速度.模调度是一种被广泛采用的软件流水的启发式.为了改善存储系统,cache 使用了分级机制,但这也带来了额外的存储延迟-cache 代价.证明了模调度可能导致 cache 代价,并提出了一种可以避免模调度的 cache 代价的 PCPMS(prevent cache penalty in modulo scheduling)算法.实验结果表明,PCPMS 能够避免模调度中的 cache 代价,提高程序性能.

关键词: 软件流水;模调度;存储优化;cache 代价

中图法分类号: TP338 文献标识码: A

软件流水[1](software pipeline,简称 SWP)能够加快循环的执行速度.在软件流水中,相邻两个循环体的启动时刻差被称为启动间距(initiation interval,简称 II).模调度[2](modulo scheduling)是一种被广泛采用

* Supported by the National Natural Science Foundation of China under Grant No.60573100 (国家自然科学基金)

作者简介: 刘利(1981−),男,四川沐川人,硕士生,主要研究领域为指令级并行算法;李文龙(1977−),男,博士,研究员,主要研究领域为计算机体系结构,指令级并行算法;郭振宇(1981−),男,硕士生,主要研究领域为指令级并行算法;李胜梅(1981−),女,博士生,主要研究领域为指令级并行算法;汤志忠(1946−),男,教授,博士生导师,主要研究领域为计算机系统结构,指令级并行算法,并行编译技术.

的软件流水的启发式.

随着技术的发展,存储系统与处理机之间速度差距逐渐变大,为此,cache 使用了分级机制,但这会引入额外存储访问延迟[3,4].执行程序时,如果存取指令间出现了 bank 冲突,或者存储端口不够用,或者低级 cache 无法快速处理一些特殊情况,额外延迟就会产生,我们称这种额外延迟为 cache 代价(cache penalty,简称代价).

本文以安腾 2 为模型,证明了模调度可能导致代价,提出了能够避免模调度的代价的 PCPMS(prevent cache penalty in modulo scheduling)算法.安腾 2 是 IPF(Itanium processor family)[5,6]的第 2 代处理机,IPF 为软件流水提供了硬件支持.

本文第 1 节介绍模调度和 IPF 的特点及安腾 2 的代价[7].第 2 节证明模调度可能导致代价.第 3 节分析产生代价的条件,提出 3 种避免代价的方法,设计了 PCPMS 算法.第 4 节是实验结果和分析.最后是相关工作和总结.

1 相关知识

1.1 模调度

在模调度[8]中,一个循环体最晚与最早调度的指令间的调度时刻差被称为调度长度(scheduling length,简称 SL).II 的下限(MII)由资源限制和依赖限制决定,其中依赖限制根据数据依赖图(data dependence graph,简称 DDG)上的回路确定.在 DDG 上,如果节点 A 有边到 B,即 B 依赖于 A,本文就表示为 A→B,称 A 是 B 的前驱(parent),如果 A 有路径到达 B,则表示为 A→B.

1.2 IPF的体系结构特点

指令间存在着依赖,当处理机顺序执行时,依赖自然能得到保证.如果处理机支持乱序执行指令或同时发射多条指令,就需要用硬件保证指令间的依赖.IPF 支持显式并行指令集计算,在一个指令序列中,如果任意指令间都没有依赖,就称这一序列为指令组,因此处理器不必判断同一指令组中的指令间的依赖,但由于存取指令的延迟不固定,而且编译器难以准确分析出它们间的依赖,因此在执行存取指令时,存储系统需要检查它们的依赖关系.IPF 允许同时发射 6 条指令,但它们在执行顺序上存在先后.

IPF 为软件流水提供了硬件支持,其 cache 分为 3 级:L1,L2 和 L3.L1 分为指令 cache 与数据 cache(L1D),其中 L1D 只缓存整数的数据.安腾 2 的 L1D 的 cache 块长度是 64,其 L2 有 16 个大小为 16 字节的 bank,即两条地址差是 256 的倍数的存取指令会映射到相同的 L2 bank 中.

1.3 安腾2的代价的总结

在编译阶段,无法肯定指令执行时的准确延迟,因此可以把指令间的调度间隔作为它们的发射间隔.出现代价的两条存取指令必须具备一定的条件,可将这些条件分为 cache 条件和调度条件.cache 条件主要表现为两条存取指令访问存储系统的相同单元以及是否命中 cache 等.本文将硬件单元的大小称为代价长度(penalty size,S_P).调度条件主要表现为两条存取指令的调度间隔(cycle interval,I_C)和执行顺序.

表 1 列出了安腾 2 的所有代价类型,以及相应的代价长度、cache 条件和调度条件.

Table 1 The conditions of cache penalty on Itanium2

表 **1** 安腾 2 的代价的条件

Cache penalty kind	Penalty size (Bytes)	Conditions on cache	Condition on issue cycle interval (Cycles)
L1D load/store	64	Access the same L1D line and miss L1D	0
L1D store/load	64	Access the same L1Dline and hit L1D	0~3
L1D store/store	64	Access the same L1D line	0
L2 bank load/load	16	Access the same L2 bank	0
L2 bank store/store	16	Access the same L2 bank	0
L2 bank store/load	16	Access the same L2 bank	3

表 1 隐含了存取指令执行顺序条件.例如,对于 L1D load/store 代价,load 指令必须比 store 指令先执行.当调度间隔大于 0 时,执行顺序自然得到满足,而且只有当两条存取指令的存取类型不同时,才有必要考虑执行顺序,因此只有 L1D store/load 和 L1D load/store 代价受到执行顺序条件限制.L1D store/load 代价的额外延迟的大小

由 load 与 store 指令间的调度间隔决定,见表 2,其他类型的代价的额外延迟一般是几个 CPU 周期.

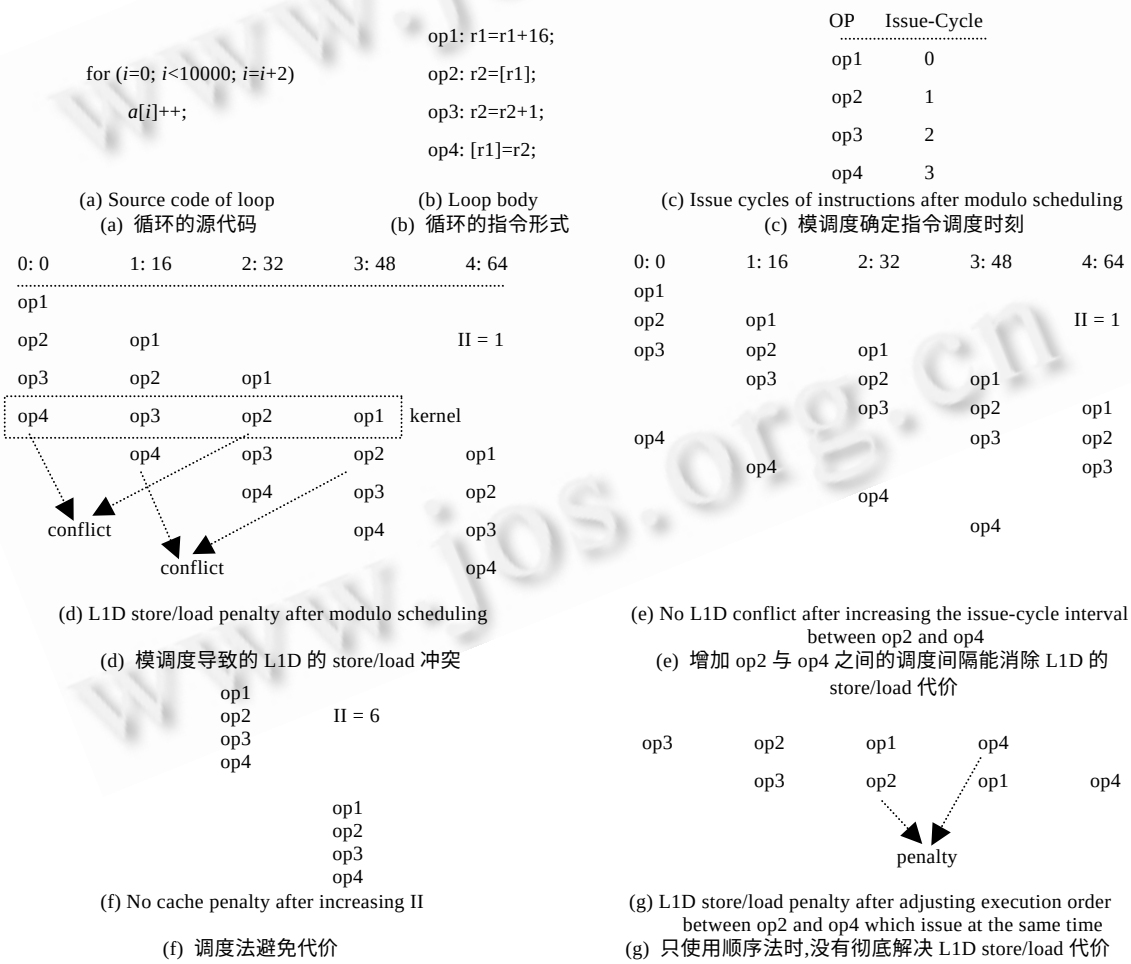
Table 2 L1D store/load penalty depends on how many cycles store precedes load by

表 2 L1D store/load 代价与 load,store 指令间的调度间隔的对应关系

Store precedes load by (cycles)	Penalty (cycles)
0	17
1	3~5
2	3
3	1~3

2 模调度中潜在的代价

如图 1(b)所示,图 1(a)的循环体中有 4 条指令(省略了分支指令),其中 op2 和 op4 分别是整数 load 和 store 指令,它们在相同循环体中的地址相同.设 II 是 1,相同循环体中的指令调度时刻如图 1(c)所示.图 1(d)是前 5 个循环体的调度情况,并设 op1 和 op2 始终命中 cache,a[0]的地址是 0,则 op1 和 op2 在前 5 个循环体的存取地址是 0,16,32,48,64.用 op_*x* 表示第 *x* 循环体的 op,则 op4_0 和 op2_2 同时调度,且 op4_0 先执行,它们的存取地址 0 和 32 映射到相同 L1D 块,因此存在 L1D store/load 代价,同理,op4_1 与 op2_3 也存在代价,但 op4_3 与 op2_5 没有代价,因为它们存取地址 32 和 64 映射到不同 L1D 块.依此类推,每连续 4 个循环体出现 2 次调度间隔为 0 的 L1D store/load 代价.由此可见,模调度的结果中可能存在代价.



(a) Source code of loop (b) Loop body (c) Issue cycles of instructions after modulo scheduling

(a) 循环的源代码 (b) 循环的指令形式 (c) 模调度确定指令调度时刻

(d) L1D store/load penalty after modulo scheduling

(d) 模调度导致的 L1D 的 store/load 冲突

(e) No L1D conflict after increasing the issue-cycle interval between op2 and op4

(e) 增加 op2 与 op4 之间的调度间隔能消除 L1D 的 store/load 代价

(f) No cache penalty after increasing II

(f) 调度法避免代价

(g) L1D store/load penalty after adjusting execution order between op2 and op4 which issue at the same time

(g) 只使用顺序法时,没有彻底解决 L1D store/load 代价

Fig.1 The implicit store/load penalty in modulo scheduling and how to prevent it

图 1 模调度中潜在的 store/load 冲突和解决方法

根据上面的分析,引出几个定义:

定义 **1**(代价前者,代价后者). 对于一对存取指令,如果它们可能导致某种类型的代价,则称这个代价先执行的那条指令为代价前者,后执行的那条指令为代价后者.

定义 **2**(静态调度间隔,**static cycle interval**,I_{SC}). 代价后者与前者在相同循环体中的调度时刻差.

定义 **3**(动态调度间隔,**dynamic cycle interval**,I_{DC}). 代价前者与后者的最小非负调度时刻差.

定义 **4**(代价体差,**penalty iteration interval**,I_{PI}). 调度时刻差恰好是动态调度间隔时,用 $x1$ 表示代价后者所在循环体的序号,用 $x2$ 表示代价前者所在循环体的序号,把 $x1–x2$ 称为代价体差.

定义 **5**(静态地址间隔,**static address interval**,I_{SA}). 相同循环体的代价后者与前者的地址差称为静态地址间隔.

定义 **6**(动态地址间隔,**dynamic address interval**,I_{DA}). 调度间刻差恰好是动态调度间隔时,代价后者与前者的地址差称为动态地址间隔.

对于图 1(d),op2 与 op4 可能存在 L1D store/load 代价,代价前者是 op4,代价后者是 op2,静态调度间隔是–2,动态调度间隔是 0,静态地址间隔是 0,动态地址间隔是 32,代价循环体差是 2.

3 PCPMS 算法

本节分 6 小节介绍 PCPMS 算法.第 3.1 节介绍存取指令的地址和局部性分析,第 3.2 节介绍如何确定一对存取指令可能出现的代价类型,第 3.3 节对产生代价的 cache 条件进行定量分析,第 3.4 节对调度条件进行定量分析,第 3.5 节提出了 3 种避免代价的方法,第 3.6 节给出 PCPMS 算法的流程.

3.1 存取指令的分析

定义 **7**(地址增量). 一条存取指令在相邻两个循环体的地址差称为地址增量(address increment,I_A).

定义 **8**(规则存取指令). 地址增量固定不变的存取指令称为规则存取指令(regular memory instruction).

定义 **9**(不规则存取指令). 地址增量不固定的存取指令称为不规则存取指令(irregular memory instruction).

例如,在下面的指令集中,op1 规则,其地址增量是 8,op2 和 op4 不规则.

```
op1: ld r2 = [r1];
op2: ld r3 = [r2];
op3: r4=r2+8;
op4: ld r5=[r4];
op5: r1=r1+8;
```

由于规则存取指令的局部性较好,而且可以采用数据预取技术[9]将数据预取到 cache 中,因此在 PCPMS 算法认为规则存取指令总是命中 cache,而对于不规则存取指令,PCPMS 算法认为它们总是发生 cache 缺失.

3.2 一对存取指令可能出现的代价类型

在分析代价时,首先应该判断可能出现的代价类型,这样才能确定 cache 条件、调度条件和代价长度.表 3 列出了两条存取指令与代价的对应关系.

Table 3 Implicit cache penalty kind according to pair of memory instructions

表 **3** 一对存取指令可能存在的代价类型

Pair of memory instructions	Characters of two memory instructions	Penalty kind
Load, load	Integer and regular	No penalty
Load, load	Float or irregular	L2 bank load/load
Load, store	Integer and regular	L1D store/load
Load, store	Integer and irregular	L1D load/store or L2 bank store/load
Load, store	Float	L2 bank store/load
Store, store	Integer	L1D store/store
Store, store	Float	L2 bank store/store

对于两条整数 load 指令,如果它们规则,则命中 L1D,也就没有代价;如果它们不规则,则发生 L1D 缺失,数据请求到达 L2,因此代价发生在 L2 bank.对于整数 load 和 store 指令,如果它们规则,数据请求不会到达 L2,则代价

是 L1D store/load,否则代价是 L1D load/store 或 L2 bank store/load.由于浮点数不能缓存到 L1D 中,因此,一对浮点类型的存取指令的代价只能发生在 L2 bank.对于其他组合方式的两条存取指令,不太可能产生代价,这将在第 3.3 节中加以说明.

3.3 对cache条件的分析

存取指令在执行时的存取地址只能在程序运行时得到,但编译器能够确定存取指令间的地址差,因此可以根据动态地址间隔来判断两条存取指令是否会访问相同的 cache 单元.这需要先求出静态地址间隔.

3.3.1 静态地址间隔

静态地址间隔是由循环体本身决定的,可以根据编译的相关信息直接得到静态地址间隔.在 PCPMS 算法中,如果两条存取指令间出现代价的可能性很小,或者难以分析出它们的地址间隔,就认为它们之间没有代价,并把静态地址间隔标记为∞,这主要包括以下几种情况:

- 如果不能得到两条不规则存取指令的地址差,静态地址间隔就是∞.例如,下面指令集中的 op3 和 op4:

op1: ld r3=[r1];

op2: ld r4=[r2];

op3: st [r3]=1;

op4: ld r5=[r4].

- 如果两条存取指令分别是规则与不规则的,则静态地址间隔是∞.
- 浮点数存取指令与整数存取指令间的静态地址间隔是∞.
- 如果两条规则整数存取指令的地址增量不同或者它们存取的数据属于不同的数组,则它们出现访问相同 L1D 块的概率几乎为 0,这样,它们的静态地址间隔是∞.
- 如果两条规则浮点数存取指令的地址增量相同,且存取的数据属于不同数组,则它们访问相同 L2 bank 的概率与这两个数组基地址差有关,那么就可以利用数组数据分布[10]来降低出现代价的可能性,因此,在多数情况下可以认为这两条指令的静态地址间隔是∞.
- 如果两条规则浮点数存取指令的地址增量不同,它们会以一定的频率访问相同的 L2 bank.例如,设两条规则浮点数存取指令的地址增量分别是 8 和 16,且它们在第 1 循环体访问相同的 L2 bank,则它们在第 32 循环体也访问相同 L2 bank,频率是 1/32.频率的计算公式如下:

$$F_{same\_bank}=\left\lceil\frac{bank\_size}{\mathrm{Max}(I_{A1},I_{A2})}\right\rceil\times\frac{|I_{A1}-I_{A2}|}{256},$$

其中 I_{A1} 和 I_{A2} 分别表示这样的两条存取指令的地址增量.可以设定一个频率域值,当地址增量不同的两条规则浮点数存取指令访问相同 L2 bank 的频率低于域值时,它们的静态地址间隔就是∞.

- 如果两条规则存取指令的地址间隔是编译时未知的,就认为它们的静态地址间隔是∞.

静态地址间隔是∞的两条存取指令不太可能出现代价,因此它们对应的组合就没有在表 3 中列出.

3.3.2 动态地址间隔

静态地址间隔为∞的两条存取指令的动态地址间隔也是∞.不规则存取指令的地址变化规律难以分析,对于第 3.1.1 节的指令集中的 op2 与 op4,编译阶段只能得到它们在相同循环体地址差,无法推算出 op4 与其他循环体的 op2 之间的地址差,因此,PCPMS 算法规定不规则存取指令间的代价体差是 0.

一对规则存取指令间的代价体差可能不是 0,当静态调度间隔小于 0 时,

$$I_{PI}=\lceil -I_{SI}/II\rceil \tag{1}$$

否则

$$I_{PI}=-\lfloor I_{SI}/II\rfloor \tag{2}$$

则动态地址间隔是

$$I_{DA}=I_{SA}+I_A\times I_{PI} \tag{3}$$

对于 L2 bank 代价,如果 I_{DA}%256<16 或 I_{DA}%256>240,两条存取指令就会访问相同的 L2 bank.为了与 L1D 代价

统一起来,需要转化 L2 bank 代价的动态地址间隔:

如果 I'_{DA}%256<241,

$$I_{DA} = I'_{DA}\%256;$$

如果 I'_{DA}%256>240,

$$I_{DA} = I'_{DA}\%256 - 256,$$

其中 I'_{DA} 表示由式(3)计算得到的动态地址间隔.

3.4 对调度限制的分析

在分析调度限制时,需要判断两条存取指令是否符合调度间隔条件,这表现在对动态调度间隔的分析上.在求得动态调度间隔之前,需要已知静态调度间隔.

3.4.1 静态调度间隔

在模调度过程中,各条指令的调度时刻是按照一定的优先级依次确定的,调度时刻已确定的指令的状态被标记为已调度.如果两条存取指令都被标记为已调度,就能得到它们的静态调度间隔和代价体差.

表 3 中曾提到,不规则的 load 和 store 指令可能出现两种代价,但是这两种代价的调度间隔条件是不同的.由于不规则存取指令间的代价体差是 0,因此可以利用静态调度间隔加以区别,如果静态调度间隔是 0,不规则的 load 和 store 指令可能出现的代价是 L1D load/store,否则是 L2 bank store/load.

L2 bank store/load 代价的调度间隔限制是 3,在分析代价时,为了将它与其他类型的代价统一起来,就认为它的代价前者延迟 3 个周期调度,即让静态调度间隔减小 3.这样就转化了调度间隔限制,即求得新的静态调度间隔后,L2 bank store/load 代价调度间隔限制变成了 0.

3.4.2 动态调度间隔

动态调度间隔与代价体差是互相对应的,因此,

$$I_{DC} = I_{SC} + II \times I_{PI},$$

对于一对不规则存取指令,由于它们的代价循环体差是 0,因此,

$$I_{DC} = I_{SC} \tag{4}$$

而对于一对规则存取指令,如果静态调度间隔小于 0,由公式(1)可得,

$$I_{DC} = I_{SC} + II \times \lceil -I_{SC} / II \rceil \tag{5}$$

否则,由公式(2)可得,

$$I_{DC} = I_{SC} - II \times \lfloor I_{SC} / II \rfloor \tag{6}$$

3.4.3 代价的存取指令执行顺序

根据第 1.3 节,当动态调度间隔是 0,且代价是 L1D store/load 或 L1D load/store 时,才有必要考虑执行顺序条件.

3.5 避免代价的方法

可以通过改变动态地址间隔来避免代价,这种方法称为地址法;也可以通过改变动态调度间隔来避免代价,这种方法称为调度法;还可以通过改变执行顺序来避免代价,这种方法称为顺序法.

3.5.1 地址法

地址法的任务是使动态地址间隔的绝对值变得比代价长度要大.由公式(3)可得,地址法的途径是改变代价体差,由于不规则存取指令的代价体差是 0,因此地址法只适用于规则存取指令.再由公式(1)和(2)可得,地址法的实质是改变静态调度间隔,且改变量是 II 的倍数.在改变静态调度间隔时,保持一条指令的调度时刻不变,而增加另一条的调度时刻,称调度时刻增量为无代价调度增量(no penalty cycle increment,I_{NPC}).如果动态地址间隔的绝对值不小于代价长度,无代价调度增量就是 0;否则,当调度时刻不变的是代价前者时,

如果 I_A>0,

$$I_{NPC} = (\lceil (S_P + I_{SA}) / I_A \rceil + I_P) \times II,$$

如果 I_A<0,

$$I_{NPC}=(\lceil -(S_P-I_{SA})/I_A\rceil+I_P)\times II\ .$$

当调度时刻不变的是代价后者时,

如果 I_A>0,

$$I_{NPC}=(\lceil (S_P-I_{SA})/I_A\rceil-I_P)\times II\ ,$$

如果 I_A<0,

$$I_{NPC}=(\lceil -(S_P+I_{SA})/I_A\rceil-I_P)\times II\ .$$

如图 1(e)所示,在增加 op4 与 op2 的静态调度间隔后,避免了 op4 与 op2 的代价,此时 II 没有变大.由此可见,地址法不会影响 II,但可能会较大地增加模调度的调度长度,这会增加寄存器需求[11].在模调度过程中,II 没有确定,因此地址法实现于模调度结束之后.无代价调度增量是 II 的倍数,因此地址法不会导致资源冲突,只需保证所有指令间的依赖关系.地址法的实现细节,将在第 3.5.4 节中加以介绍.

3.5.2 调度法

调度法的任务就是使得动态调度间隔不满足代价的调度间隔条件.由公式(4)~(6)可得,调度法的实质也是改变静态调度间隔,但与地址法不同的是,静态调度间隔的改变量不需要是 II 的倍数,而且对规则存取指令来说,II 限制了动态调度间隔的候选值,例如当 II=1 时,候选值只有 0,依此类推,当 II=n 时,候选值的范围是[0,n−1],因此动态调度间隔的改变可能会导致 II 变大.如图 1(f)所示,使用调度法,将 II 增大为 6 后,避免了 L1D load/store 代价.调度法对静态调度间隔的调整比较细微,而且还可能使 II 变大,因此它实现于模调度过程中.模调度在调度某条指令时,首先根据依赖确定调度时刻的范围,然后按照一定的顺序,在这个范围中找到一个不会与已调度的指令发生资源冲突的调度时刻,最后把这条指令的状态改成已调度.实现调度法时,只需要对模调度算法进行简单的扩展,即在选择存取指令的调度时刻时,还要考虑代价,如果这条指令在某个时刻调度时,会与某条已调度的存取指令出现代价,这个时刻就不能是它的调度时刻.

3.5.3 顺序法

只有当动态调度间隔为 0,且代价是 L1D load/store 或 L1D store/load 时,才能使用顺序法.单独使用顺序法时,可能无法彻底避免 L1D store/load 代价.例如,在图 1(g)中,改变同时调度的 op4 与 op2 的执行顺序后,并没有彻底避免代价,这是因为 II 较小,而且产生 L1D store/load 代价的两条指令的代价体差不为 0,此时只要再结合地址法,就能避免代价.例如,将图 1(d)的 op4 与 op2 的静态调度间隔增大 1 后,再使用顺序法,就能完全避免 L1D store/load 代价.不规则的整数 load 和 store 指令的代价体差是 0,因此顺序法能够避免 L1D load/store 代价.顺序法在调整同时调度的指令的执行顺序时,需要保证它们之间的依赖关系.

3.5.4 合理选择避免代价的方法

顺序法对模调度的影响最小,而且它可以配合其他两种方法,应该尽量使用,但它有严格的应用条件.调度法能够消除任意代价,而地址法只适用于规则存取指令,且它们对 II 和调度长度的影响程度不同,因此应该合理选择这两种方法,使得 II 和调度长度不会过分增大.可以在模调度的过程中决定使用调度法还是地址法,但此时难以预见调度法是否会使 II 变大,因此 PCPMS 算法采用的策略是在模调度之前进行抉择.

定义 **10**(安全 **II**(**Safe II**),简称 **SII**). 只使用调度法避免代价时,II 必须满足的最小值.

不规则存取指令不会影响 SII.为了估算出 SII,可以将循环体中的所有规则存取指令分成若干个代价组.代价组满足的条件是,如果组中有多条指令,任意两条指令都有可能出现代价.代价组的划分在模调度之前,此时还不知道循环体中指令的调度时刻,因此不需要考虑调度条件,这时采用的近似方法是将静态地址间隔不是∞规则存取的指令划分到同一个代价组中.

每一个代价组都有自己的 SII,如果代价组中的可能代价类型不包括 L1D store/load,则

$$SII=n\ ;$$

否则,

$$SII=n+3\ .$$

如果代价组的 SII 不大于 MII,则对这个组中的代价,不采用地址法消除;否则,应该把代价组分拆成若干个子代价组,使得所有子代价组的 SII 不大于 MII.对于每个子代价组内的代价,仍然不采用地址法;而对于子代价

组间的代价,在模调度结束后,采用地址法和顺序法消除.分拆代价组时,应该满足以下原则:

- 如果代价组有多条指令属于相同依赖回路,则它们必须分拆到相同子代价组中.
- 将子代价组表示为一个节点,对于两个子代价组 A 和 B,设 a 和 b 分别是 A 和 B 的任意一条指令,如果在数据依赖图上,a→b,画一条 A 到 B 的依赖边,这样就形成了子代价组依赖图(penalty subgroup dependence graph).分拆后,子代价组依赖图上不能有回路,例如,设 op1,op2,op3,op4 是代价组中的 4 条指令,其中 op1→op3,op2→op4,则分拆的结果不能是{op1,op4},{op2,op3}.
- 设指令 a 和 b 是代价组的两条指令,如果 a→b,则尽量把 a 和 b 分拆到不同子代价组中.
- 尽量把同类型的指令分拆到相同的子代价组中.例如,子代价组中只有 load 指令或只有 store 指令.

在模调度结束后,地址法的具体实现是:

- 同一个子代价组的所有指令的调度时刻增量相同,这样就能保证子代价组中不会出现新的代价,同时,还需要保证依赖关系,因此,同一强连通块的所有指令的调度时刻增量也必须相同.在数据依赖图上,将调度时刻增量需要保持相同的所有指令看成是一个节点,把所得的新有向图称为牵连依赖图(implication dependence graph),将各个节点的调度时刻增量称为调度增量(cycle increament),每个节点的所有指令的调度时刻增量都是这个节点的调度增量.
- 将牵连依赖图中的所有节点标记为未调整.记录每个节点的调度增量.
- 按照牵连依赖图的拓扑排序的次序来调整各个节点的调度时刻.设当前处理的节点是 A.首先,计算出 A 在牵连依赖图中的所有前驱的调度增量的最大值(称为最大前驱调度增量,max parent cycle increment,I_{MPC},没有前驱时,值是 0),并把 A 的所有指令的调度时刻增大 I_{MPC}.然后,如果 A 的指令与已调整的节点的指令之间有代价,就计算出相应的无代价调度增量,并求出这些无代价调度增量的最大值(没有代价时,值是 0),用 I_{MNPC} 表示,再把 A 中所有指令的调度时刻增大 I_{MNPC}.最后把 A 标记为已调整.因此,A 的调度增量是 $I_{MIC}+I_{MNPC}$.

3.6 算法流程

调度法和地址法都需要判断两条存取指令间是否有代价,这由 Has_cache_penalty2 函数实现.Has_cache_penalty2 函数的伪代码如图 2 右上部分所示.它首先确定代价类型,然后判断动态地址间隔是否小于代价长度,最后判断调度条件是否满足.Has_cache_penalty2 函数在第 6 步实现了顺序法,此时首先判断依赖是否允许调整同时调度的指令的执行顺序,然后调整执行顺序,最后判断顺序法是否能够消除代价.在函数 Has_cache_penalty2 中实现顺序法,便于将顺序法与其他两种方法结合起来.

调度法需要在模调度过程中判断当前调度的存取指令是否与任意已调度存取指令之间有代价,这由函数 Has_cache_penalty1 来实现.Has_cache_penalty1 函数的伪代码如图 2 左上部分所示,在主循环中,它首先判断 memop2 的调度时刻是否已经确定,然后,当 memop1 是规则存取指令时,判断 memop1 和 memop2 是否在相同的代价组或子代价组中(如果代价组进行了分拆),最后调用了函数 Has_cache_penalty2.

PCPMS 算法的主函数如图 2 的下半部分所示.函数首先分析了循环体中的存取指令,然后将规则存取指令分成若干代价组,并将有的代价组拆分成若干子代价组.在模调度结束后,就采用地址法来消除子代价组间的代价.PCPMS 函数中的第 4 步和第 5 步就是地址法,这与第 3.5.4 节描述的地址法的实现是一致的,在此不作具体说明.在软件流水中,用 PCPMS 函数替换掉起初的模调度函数,这样就能避免模调度的代价.

4 实验数据分析

我们已经在 ORC[12] 2.1 中实现了 PCPMS 算法,ORC 是一个开放源码的编译器,其软件流水模块采用了 Huff 的模调度算法[11].本节将以 NAS kernel benchmarks 和 SPECfp2000 为例,对 PCPMS 算法进行测试和分析,对比的基准(base)使用的编译选项是–O3.编译的结果在安腾 2 上执行.ORC2.1 还存在着一些问题,IS 的编译结果会在运行过程中出错,因此在第 4.2 节中没有分析 IS.

4.1 模调度结果的分析

表 4 列出了 NAS kernel benchmarks 在使用 PCPMS 算法前后,模调度的 II 和 SL,由于测试程序中有若干循

环做了模调度,因此表中的数据是平均值.平均看来,PCPMS 算法使调度长度变大了,这与前面的分析相一致,而且 II 也有所增大,这是因为调度法增加了存取指令的调度限制,尽管在模调度之前采用了启发式以选择调度法与地址法,但是那时也无法准确估计出调度法对调度限制的影响程度,以及是否会使 II 变大.从编译结果的角度来看,PCPMS 算法使模调度变差了.

```
bool Has_cache_penalty1 (memop1) {
  For (every memory instruction, specified as memop2) {
1   If (memop1 == memop2)
        continue;
2   If (memop2 has not issued)
        continue;
3    If (memop1 is regular &&
      two memops are not in same penalty subgroup)
        continue;
4   I_SA = static address interval between the two memops;
5   If (I_SA == ∞ )
        continue;
6   If (Has_cache_penalty2 (memop1, memop2))
        return true;
  }
  return false;
}
```

```
bool Has_cache_penalty2 (memop1, memop2){
1  Determine the penalty kind of the two memops;
2  S_P = penalty size;
3  I_DA = dynamic address interval between the two memops;
4  If (I_DA >= S_P) return false;
5  I_DC = dynamic cycle interval between the two memops;
6  If (penalty is L1D store/load or load/store && I_DC == 0) {
6.1    if (adjusting execution order will obey dependence)
           return true;
6.2    adjusting execution order;
6.3    if (penalty is store/load && II < 3 && I_DA + I_A < S_P)
           return true;
6.4    return false;
   }
7  If (penalty is L1D store/load && I_DC <= 3) return true;
8  If (I_DC == 0) return true;
9  return false;
}
```

```
void PCPMS (loop body) {
1   Analyze the address of every memory instruction in loop body;
2   Divide regular memory instructions into penalty groups and divide penalty group into subgroups;
3   Modulo scheduling;
4   Construct implication-dependence-graph, mark all nodes in implication-dependence-graph as unadjusted;
5   For (every node in implication-dependence-graph, in the topological order) {
        Specify current node as A;
        I_MPC = max-parent-cycle-increment of A;
        If (I_MPC > 0)    Increase the issue cycles of all instructions in A by I_MPC;
        I_MNPC = the max of no-penalty-cycle-increment of all memory instructions in A;
        If (I_MNPC > 0)    Increase the issue cycles of all instructions in A by I_MNPC;
        Mark A as adjusted;
```

Fig.2 Algorithm of PCPMS

图 2 PCPMS 算法流程

Table 4 Results of modulo scheduling on average

表 **4** 模调度的平均结果

Benchmark	kind	II	SL	Benchmark	Kind	II	SL
BT	Base	11.14	39.16	CG	Base	6.88	16.28
	PCPMS	11.26	39.42		PCPMS	7.25	16.59
EP	Base	11.33	57.33	FT	Base	9.06	21.88
	PCPMS	11.33	57.45		PCPMS	9.24	23.82
IS	Base	7.4	11.6	LU	Base	11.35	45.04
	PCPMS	7.4	12.2		PCPMS	11.67	46.16
MG	Base	9.86	24.77	SP	Base	12.40	35.33
	PCPMS	9.86	25.12		PCPMS	13.38	38.14

4.2 性能分析

表 5 是 NAS kernel benchmarks 在使用 PCPMS 算法后,L2 bank 冲突的变化情况,可以看出,PCPMS 算法有效地降低了出现代价的次数.之所以没有完全消除 L2 bank 冲突,是因为 PCPMS 算法只针对做了模调度的循环进行了优化.表 6 是 PCPMS 算法为 NAS kernel benchmarks 带来的性能加速比图,其中 CG 和 FT 的性能有较大的提高,因为它们的代价都大幅度地减少了,这可以从表 5 看出,7 个测试程序的平均性能提高了 8.7%.表 7 是 PCPMS 算法为 SPECfp2000 带来的性能加速比图,平均性能提高了 2.7%.Sixtract 的性能有所降低,这是因为避免代价带来的性能提高不足以弥补模调度的性能降低.从整体看来,尽管 PCPMS 算法使模调度变差了,但是它有效地避免了模调度中的代价,实现了存储优化,最终提高了编译器的性能.

Table 5 The ratio of number of L2 bank conflicts due to PCPMS compared to basic number of L2 bank conflicts

表 5 使用 PCPMS 后,L2bank 冲突次数的变化

Benchmark	BT	LU	CG	MG	EP	SP	FT
Ratio	0.642	0.956	0.201	0.951	0.879	0.587	0.130

Table 6 Speedup of NAS kernel benchmarks due to PCPMS

表 6 使用 PCPMS 后,NAS 的性能加速比

Benchmark	BT	LU	CG	MG	EP	SP	FT	AV
Speedup	1.039	1.032	1.246	1.037	1.009	1.021	1.228	1.087

Table 7 speedup of SPECfp2000 due to PCPMS

表 7 使用 PCPMS 后,SPECfp2000 的性能加速比

Benchmark	Speedup	Benchmark	Speedup	Benchmark	Speedup	Benchmark	Speedup
Wupwise	1.012	Swim	1.063	Mgrid	1.050	Applu	1.023
Mesa	1.010	Galgel	1.043	Art	1.087	Equake	1.020
Facerec	1.011	Ammp	1.028	Lucas	1.022	Fma3d	1.006
Sixtrack	0.990	Apsi	1.013	AV	1.027		

5 相关工作和结论

存储优化的目的是降低存储延迟对程序性能的影响.存储延迟可以分为两类:cache 缺失的延迟和额外的延迟(即代价).提高数据局部性[13]能够减少 cache 缺失的次数,数据预取[14]和数据猜测[15]等技术能够隐藏 load 指令的存储延迟,FLMS 算法[8]能够隐藏模调度中的 load 指令的存储延迟,从而降低 cache 缺失的延迟对程序性能的影响.代价是由 cache 的分级机制导致的,因此避免代价是存储优化的重要内容.文献[7]以安腾 2 为例,揭示了 cache 分机机制导致代价的原因,列举了安腾 2 的各种代价,并提出了一种能够避免代价的调度算法,从实验数据可以看出,避免代价能提高程序性能.循环在程序的执行时间中占有较大比例,循环的调度方式可分为无环调度和有环调度,其中模调度属于有环调度,当 SL 和 II 相等时,对应的模调度就与无环调度没有区别了,因此可以把无环调度看作特殊的模调度.文献[7]中的调度算法主要针对无环调度,而本文提出的 PCPMS 算法主要针对循环调度.与以前的算法相比,PCPMS 算法的特点是:

- 适用于模调度和循环的无环调度.
- 采用了 3 种避免代价的方法:调度法、地址法和顺序法,并给出了选择它们的启发式.
- 通过代价长度和调度间隔条件等,将各种类型的代价统一起来.
- 利用代价体差、动态调度间隔和动态地址间隔,将相同循环体的代价与来自不同循环体的存取指令间的代价统一起来.
- 尽管调度法和地址法实现的时机不同,但是它们都用相同的方法来检测代价.

实验证明,PCPMS 算法能够避免模调度中的代价,并提高编译器的性能.

References:

[1] Allan VH, Jones RB, Lee RM, Allan SJ. Software pipelining. ACM Computing Surveys, 1995,27(3):367−432.

[2] Rau BR. Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proc. of the 27th Annual Int'l Symp. on Microarchitecture. New York: ACM Press, 1994. 63−74.

[3] Intel Corp. Intel Itanium 2 Processor Reference Manual for Software Development and Optimization. Intel Corporation, 2004.

[4] Intel Corp. Intel Pentium 4 and Intel Xeon Processor Optimization. Reference Manual. Intel Corporation, 1999.

[5] Intel Corp. Intel Itanium Architecture Software Developer's Manual. Volume 1-4, Revision 1.1. Intel Corporation, 2000.

[6] Huck J, Morris D, Ross J, Knies A, Mulder H, Zahir R. Introducing the IA-64 architecture. IEEE Micro, 2000,20(5):12−23.

[7] Collard J-F, Lavery D. Optimizations to prevent cache penalties for the Intel® Itanium® 2 processor. In: Int'l Symp. on Code Generation and Optimization, 2003. 105−114.

[8] Liu L, Li WL, Chen Y, Li SM, Tang ZZ. Hiding memory access latency in software pipelining. Journal of Software, 2005,16(10):1833−1841 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/1833.htm

[9] Callahan D, Kennedy K, Porterfield A. Software prefetching. In: Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 1991. 40−52.

[10] Rubin S, Bodík R, Chilimbi T. An efficient profile-analysis framework for data-layout optimizations. ACM SIGPLAN Notices, 2002,37(1):140−153.

[11] Huff RA. Lifetime-Sensitive modulo scheduling. In: Budd TA, ed. Proc. of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1993. 258−267.

[12] Roy J, Sun C, Wu CY. Tutorial: Open research compiler for itanium processor family (IPF). In: Proc. of the 34th Annual Int'l Symp. on Microarchitecture. New York: ACM Press, 2001.

[13] Song YH, Xu R, Wang C, Li ZY. Improving data locality by array contraction. IEEE Trans. on Computers, 2004,53(9):1073−1084.

[14] Doshi G, Krishnaiyer R, Muthukumar K. Optimizing software data prefetches with rotating registers. In: Hurson AR, ed. Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques. IEEE Press, 2001. 257−267.

[15] Mahadevan U, Nomura K, Ju RD, Hank R. Applying data speculation in modulo scheduled loop. In: Hurson AR, ed. Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques. IEEE Press, 2000. 169−176.

附中文参考文献:

[8] 刘利,李文龙,陈彧,李胜梅,汤志忠.软件流水中隐藏存储延迟的方法.软件学报,2005,16(10):1833−1841.http://www.jos.org.cn/1000-9825/16/1833.htm