软件流水中隐藏存储延迟的方法*

刘 利[1+], 李文龙[2], 陈 彧[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)

Hiding Memory Access Latency in Software Pipelining

LIU Li[1+], LI Wen-Long[2], CHEN 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, Beijing 100080, China)

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

Received 2004-09-30; Accepted 2005-06-02

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. DOI: 10.1360/jos161833

Abstract: Software pipelining tries to improve the performance of a loop by overlapping the execution of several successive iterations. As processor gets much higher speed, the memory access latency becomes a bottleneck that restricts higher performance. Software pipelining has been combined with several memory optimization technologies for higher performance by hiding memory access latency. This paper presents a foresighted latency modulo scheduling (FLMS) algorithm which determines the latency of load instructions according to the feature of the loop. Experimental results show that FLMS decreases the stall time and improves the performance of programs.

Key words: software pipeline; modulo scheduling; memory access latency; FLMS (foresighted latency modulo scheduling)

摘 要: 软件流水是一种重要的指令调度技术,它通过同时执行来自不同循环体的指令来加快循环的执行速度.随着处理机运行速度的逐渐提高,存储访问延迟成为性能提高的瓶颈.为了减轻存储系统影响,软件流水结合了一些存储优化技术,通过隐藏存储延迟来提高性能.提出了一种延迟可预测的模调度算法(foresighted latency modulo scheduling,简称 FLMS),它根据循环的特点来确定 load 指令延迟.实验结果表明,FLMS 算法减少了阻塞时间,提高了程序性能.

关键词: 软件流水;模调度;存储延迟;FLMS(foresighted latency modulo scheduling)

中图法分类号: TP338 文献标识码: A

软件流水[1](software pipeline,简称 SWP)是一种指令调度技术,它通过同时执行来自不同循环体的

* 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;陈彧(1981－),男,博士生,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.

指令来加快循环的执行速度.在软件流水中,一个循环体启动于上一循环体结束之前,相邻两个循环体的启动时刻差称为启动间距(initiation interval,简称 II).模调度[2](modulo scheduling)是被广泛采用的软件流水的启发式,在模调度中,所有循环体的调度结果相同,而且按照一个固定的 II 依次启动.

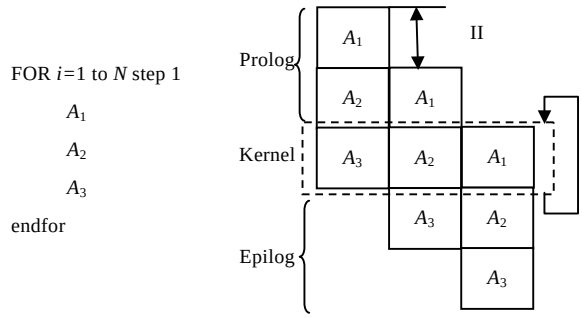
在过去,存储系统的速度与处理机的速度接近,传统模调度以 cache 命中的延迟调度 load 指令.随着存储系统与处理机之间速度差距逐渐变大,存储访问延迟已成为性能提高的瓶颈.为了减轻存储延迟的影响,传统模调度结合了一些存储优化技术,例如数据预取(data prefetches)[3]和数据猜测(data speculation)[4].这些技术把数据读取过程与指令执行过程重叠起来,从而隐藏存储延迟,但它们有一定的开销,同时,在数据预取技术中,预取指令与对应存取指令的调度间隔、预取指令的冗余性等,限制了数据预取的有效性.在数据猜测技术中,load 指令所能提前调度的量、store 指令与 load 指令之间的发生地址重叠的概率、执行恢复代码的开销等,限制了数据猜测的有效性,因此应用数据预取和数据猜测时,都必须非常谨慎.Sánchez[5]提出了一种能够隐藏存储延迟的模调度算法——CSMS(cache sensitive modulo scheduling)算法,它依据局部性和依赖关系来确定 load 指令以 cache 命中的延迟或以 cache 缺失的延迟进行调度,与传统模调度相比,CSMS 算法会使 II 变大,尽管 load 指令的延迟更接近于执行时所需的时间,但只有两种选择,这有一定的局限性.

本文提出了一种延迟可预测的模调度算法(foresighted latency modulo scheduling,简称 FLMS).它以传统模调度算法为基准,在不增大 II 的前提下,根据局部性、存储系统和循环的特点来确定 load 指令的延迟.本文第 1 节介绍模调度、数据预取和数据猜测等相关知识.第 2 节提出并分析了 FLMS 算法.第 3 节是实验结果和分析.最后总结全文.

1 相关知识

1.1 模调度

模调度的结果由装入(prolog)、核心(kernel)和排空(epilog)3 部分组成.在装入部分,前若干个循环体以 II 为间隔依次启动,随之出现一个重复模式,即核心,其长度等于 II,核心反复执行直到所有循环体都被启动,离开核心后进入排空部分,直到循环执行完毕.对于图 1(a)中的循环,经过模调度后,所得调度结果如图 1(b)所示.每个循环体的调度长度(scheduling length,简称 SL)为 $3\times II$.虚框内是核心.



(a) A loop (b) Modulo scheduling

(a) 一个单重循环 (b) 模调度

Fig.1 The modulo scheduling of a loop

图 1 单重循环模调度

II 是衡量模调度的重要指标,它的下限(MII)由两个因素决定:资源限制(ResMII)和依赖限制(RecMII),计算公式如下:

$$ResMII=\max_{\forall 资源r}\frac{使用r类资源的操作个数总和}{目标处理机中资源r的总数},\ RecMII=\max_{\forall 相关回路c}\frac{依赖回路c上的所有依赖延迟之和}{依赖回路c上的所有依赖体差之和}.$$

其中,依赖回路主要表现为数据依赖图(data dependence graph,简称 DDG)上的回路.在模调度算法中,RecMII 的计算是以强连通块(strongly connected component,简称 SCC)为单位来计算的,因为一个回路的所有指令都在同

一 SCC 中,这样可降低计算的复杂性.SCC 的 RecMII 是其包含的所有依赖回路的 RecMII 的最大值.

程序的运行时间($T_{execute}$)可分为计算时间($T_{compute}$,执行指令的时间)和阻塞时间(T_{stall},处理机等待的时间,包括存储阻塞时间、寄存器阻塞时间等).

$$T_{execute} = T_{compute} + T_{stall} \tag{1}$$

在模调度中,

$$T_{compute} = (trip\_count + \lceil SL / II \rceil - 1) \times II \approx (trip\_count - 1) \times II + SL \tag{2}$$

其中 *trip_count* 表示循环的执行次数.

1.2 数据预取

数据预取技术把将要使用的数据提前装入 cache,从而隐藏存储延迟,提高程序性能.但是,数据预取会增加存储系统的压力,将其应用于模调度时,存在以下问题:

- 数据预取技术需要在编译阶段加入预取指令,这会增加代码的长度,还可能使 II 变大.
- 预取指令属于存储指令,它会给存储系统带来压力.
- 可能会执行冗余的预取指令.可以使用条件执行预取指令、循环展开、旋转寄存器优化[6]等方法来避免执行冗余的预取指令.但这些技术有自己的适用条件,也会带来其他开销.
- 可能出现极端情况:被预取到 cache 中的数据根本就未被使用,或者在使用前就被替换出 cache 了.

1.3 数据猜测

在编译过程中,如果无法确定 store 与 load 指令是否会发生地址重叠,它们之间就存在着模糊的依赖.此时需要在 DDG 上构造一条模糊依赖边,以保证 store 比 load 指令先调度.在体系结构的支持下,可以打破模糊依赖的限制,使 load 指令在 store 指令前调度,而且在保证程序语义的条件下,还能将依赖于 load 指令的其他指令提前到 store 指令之前调度,这样就形成了数据猜测.如果运行时 store 指令与 load 指令发生了地址重叠,数据猜测就失败了,此时需要执行恢复代码,使程序恢复到正确的状态.

数据猜测打破了模糊依赖对 load 指令的限制,使 load 指令能够提前调度,从而隐藏存储延迟.数据猜测也有一定的开销,它会带来代码膨胀,也可能使 II 变大,而且过多的猜测失败会导致性能降低.

2 FLMS 算法

FLMS 算法以传统模调度算法为基准,在不增大 II 的前提下,根据局部性、存储系统和循环的特点来确定 load 指令的延迟.本节先介绍 FLMS 算法如何确定 load 指令的延迟,然后分析 FLMS 算法的原理.

2.1 确定load指令的延迟

2.1.1 局部性和循环特点的分析

定义 **1**(地址增量). 对于循环体(做软件流水的循环指令的集合)中的存取指令,其在相邻两个循环体的地址差称为地址增量(update value,简称 UV).

定义 **2**(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).

对于规则存取指令,其地址在每个循环体都会增加固定的值.例如,下面代码中的 op1,它的 UV 是 8.

op1: ld r2=[r1]

op2: r1=r1+8

在绝大多数情况下,规则存取指令的 UV 是编译时能得到的常数.当 UV 不是常数时,可以认为它是一个较小的值(例如 8),或者根据概率来判断.

定义 **3**(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).

不规则存取指令的地址是循环体动态运行的结果.例如,下面代码中的 op2,它的地址是 op1 读取的数据,只有当 op1 执行完毕,才能知道 op2 的地址.

op1: ld r2=[r1]

op2: ld r3=[r2]

对于规则存取指令,可以计算出它在相邻两个循环体的地址映射到不同 cache 块的概率(称为缺失概率),如果 UV 的绝对值小于 cache 块的长度,则

$$\text{缺失概率}=\frac{|UV|}{\text{cache块的长度}},$$

否则,缺失概率是 1.编译阶段难以分析出不规则存取指令的地址相对于 cache 的变化规律,FLMS 算法认为它的缺失概率是 1.FLMS 算法根据存取指令的缺失概率来判断局部性,概率越低,局部性越好.对于缺失概率是 1 的存取指令,FLMS 算法认为它在每个循环体执行时都会发生 cache 缺失.

为了保证 FLMS 算法的基本前提,需要计算不使 II 变大时,load 指令延迟能达到的最大值(用 T_{SCC} 表示).FLMS 算法没有增加指令,不会增加 ResMII,因此 II 变大的唯一因素是 RecMII.FLMS 算法在传统模调度算法的基础上重新确定了 load 的延迟,表现在 DDG 上,需要改变相应依赖边的延迟,如果这些依赖边都不在 SCC 中,延迟的变大不会影响 RecMII,那么这条 load 指令的 T_{SCC} 是+∞.

在 FLMS 算法中,循环特点的分析由以下步骤组成:

1. 计算传统模调度算法的 II(下文中称为基准 II).
2. 计算每条 load 指令的 T_{SCC}.
3. 统计循环体中所有存取指令的缺失概率的最大值(用 P_{MISS} 表示).

下面列出了一个循环的除分支指令以外的所有指令.

op1: ld r1=[r2]

op2: r3=8+r1

op3: ld r5=[r3]

op4: r6=r5+1

op5: st [r3]=r6

op6: r2=r2+8

可以看出,op1 是规则 load 指令,其 UV 是 8,op3 是不规则 load 指令,op5 是不规则 store 指令,op1 和 op3 的 T_{SCC} 都是+∞,P_{MISS} 是 1.

2.1.2 计算 load 指令延迟的公式

完成对循环特点的分析后,就需要计算每条 load 指令的延迟(后文用 T_L 表示).T_L 与存储系统依赖,其计算过程需要存储系统的参数:cache 命中的访问延迟(用 T_{HIT} 表示)和 cache 缺失的访问延迟(用 T_{MISS} 表示).

如果 load 指令的 T_{SCC} 是+∞,其 T_L 的计算公式为

$$T_L=\text{Max}(\text{Min}(T_{MISS},II/P_{MISS}),T_{HIT}) \tag{3}$$

如果 load 指令的 T_{SCC} 不是+∞,T_L 的计算公式为

$$T_L=\text{Max}(\text{Min}(T_{MISS},T_{SCC}),T_{HIT}) \tag{4}$$

如果一个 SCC 中有多条 load 指令的 T_{SCC} 不是+∞,应该对它们采用优先级策略,因为改变一条 load 指令的延迟后,其他 load 指令的 T_{SCC} 会受到影响.一般来说,缺失概率越大的 load 指令的优先级越高.计算 SCC 的 RecMII 和准确计算 T_{SCC} 的复杂度比较高,为了减少编译的开销,可以采用近似公式

$$T_{SCC}=II-RecMII$$

和

$$RecMII=RecMII+T_L$$

来分别计算 T_{SCC} 和修改依赖边延迟后的 SCC 的 RecMII.

2.2 FLMS算法的优化原理

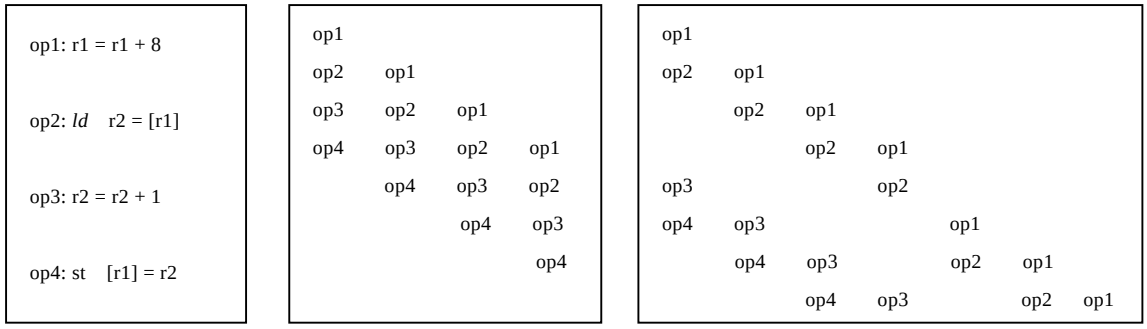
2.2.1 FLMS 算法必须保障不增大 II 的前提

假设 FLMS 算法允许 II 变大,用 ΔT 表示 II 变大后的计算时间的增量.存储延迟的隐藏是通过数据读取过程

与指令执行过程的重叠来实现的,因此存储阻塞时间至多能减少ΔT.由此可得,阻塞时间至多能减少ΔT,再根据公式(1),II 的增加不会进一步提高性能.在编译阶段,很难保证数据读取过程与指令执行过程的完全重叠,为了避免性能降低,FLMS 算法必须保证 II 不被增大.在有的体系结构中,II 的增大可以避免 load 指令与 store 指令间的冲突[7],从而带来性能提高,这不属于 FLMS 算法的研究范围,本文就不作分析了.

2.2.2 FLMS 算法能够隐藏 load 指令的访问延迟

与传统模调度相比,FLMS 算法使 load 指令的延迟变大,但是 II 不变大,因此 load 指令能与更多的其他指令同时执行,从而更好地隐藏存储延迟.对于图 2(a)中的循环,经过传统模调度后的调度结果如图 2(b)所示,经过 FLMS 算法增大 op2 的延迟后的调度结果如图 2(c)所示.可以看出,使用 FLMS 算法后,在同一循环体的 op2 与 op3 的调度间隔内,调度了多条其他循环体的指令,这样就有更多的指令与 op2 重叠执行.



(a) Instructions of loop (b) Result of cache-hit modulo scheduling (c) Result of FLMS

(a) 循环体 (b) cache 命中延迟模调度结果 (c) FLMS 的调度结果

Fig.2 FLMS can hide load latency

图 2 FLMS 隐藏 load 的延迟

2.2.3 计算 T_L 时,需要考虑循环体中所有存取指令的缺失概率的最大值

在有的情况下,保证不增大 II 的前提,即使让所有 load 指令以 cache 缺失的延迟进行调度,也做不到完全隐藏它们的存储延迟,因为循环执行时所必需的存取时间比计算时间多.例如,设 II 是 2,T_{MISS} 是 16,cache 块的长度是 128,则 UV 是 32 的规则 load 指令的缺失概率是 1/4,即每经过 8 个周期的计算时间就需要 16 个周期的存取时间,因此不能完全隐藏这条 load 指令的延迟,只要 T_L=8,就能实现计算时间与存取时间的尽量重叠.

缺失概率最大的存取指令的局部性最差,所需的存取时间最多.在计算每条 load 指令的 T_L 时,应该把 P_{MISS} 考虑进来,这样能够避免增加不必要的计算时间,也能实现尽量隐藏存储延迟.这一点体现在公式(3)中.

2.2.4 FLMS 算法的其他问题

FLMS 算法的第 1 步就是确定基准 II.一种方法就是直接把传统模调度算法的 MII 作为基准 II,另一种方法是把传统模调度结束后的 II 作为基准 II,这意味着 FLMS 算法需要进行两次模调度.在这两种方法中,后者得到的基准 II 更加准确,但是也会增加编译开销.在我们的实现中,采用的是第 2 种方法.

FLMS 算法通过隐藏存储延迟来提高程序性能.与传统模调度算法相比,FLMS 算法不会使 II 变大,但是增加了 load 指令的延迟,导致 SL 变大.由公式(2)可得,FLMS 算法会增大计算时间.当循环的执行次数较多时,SL 在计算时间中所占的比重很小,FLMS 算法不会造成程序性能的大幅降低,因此可以选择放弃对执行次数较少的循环的优化.

FLMS 算法增加了 load 指令目标寄存器的生存期,因此 FLMS 算法会增加软件流水的寄存器需求量[8].一般说来,FLMS 算法不会导致寄存器不足,但是如果发生了寄存器不足,可以有两种方法解决:把传统模调度的结果作为最终的调度结果,或者将每个 load 指令的 T_L 变小些,然后再进行模调度.

2.2.5 FLMS 算法流程

FLMS 算法.

```
void FLMS(loop body) {
```

```
1  according to cache-hit-latency of loads, construct DDG;
2  compute basic II;
3  For every load in loop body, compute its T_SCC ;
4  compute P_MISS ;
5  set priorities for the loads whose T_SCC is not +∞;
6  for every load in loop body, compute its T_L ;
7  for every load in loop body, modify its latency by its T_L on the DDG;
8  modulo scheduling;
}
```

首先以 load 指令的 cache 命中延迟构造 DDG,并求出基准 II,以及每条 load 指令 T_{SCC} 的和循环体的 P_{MISS},然后为 T_{SCC} 是+∞的 load 指令设定优先级,再求出每条 load 指令的 T_L,并以此修改 DDG,最后再根据 DDG 进行模调度.

3 实验数据分析

本节以传统模调度算法为基准(base),首先以 NAS kernel benchmarks 为例,对数据预取、数据猜测、CSMS 算法和 FLMS 算法 4 种存储优化技术进行分析,然后列出 FLMS 算法为 SPECfp2000 带来的性能加速比.全部程序采用 ORC[9]2.1 进行编译,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[8].以前的研究人员已经在 ORC 中实现了数据预取,我们在 ORC 中实现了 FLMS 算法,并在软件流水中实现了数据猜测.我们选择了 IPF(Itanium processor family)中的 Itanium2[10]作为测试用的处理器.IPF 为数据预取和数据猜测提供了硬件支持.Itanium2 是第 2 代 IA-64(EPIC)体系结构的处理器.

在第 2 节中提到过,FLMS 算法需要从体系结构中提取出参数 T_{MISS}.现在很多处理器的 cache 分成多级,对于 T_{MISS},应该灵活选择.Itanium2 的 cache 分为 3 级:L1,L2 和 L3,L3 缺失的延迟比较大,如果它作为 T_{MISS},会造成很多软件流水因寄存器不够而失败,而且 L3 容量较大,cache 替换没有 L2 和 L1 频繁,针对 L3 的局部性分析的准确率较低,在我们的实验中,T_{MISS} 是以 L2 缺失而 L3 命中的延迟.

3.1 编译结果

表 1 列出了 II 和 SL,由于每个程序中有若干循环做了软件流水,因此表中的数据是平均值.从表 1 中可以看出,FLMS 算法严格遵守了不增加 II 的前提,但是明显地增加了 SL,这是 FLMS 增加了 load 指令延迟的结果.数据预取和数据猜测在一定程度上增加了 II 和 SL,因为这两种优化方法都会在循环体中加入新的指令.CSMS 算法比 FLMS 算法更显著地增加了 SL,同时还增加了 II,因为 FLMS 算法根据循环的特点,进一步限制了 load 指令延迟的大小,而在 CSMS 算法中,没有考虑算法对 II 的影响.

3.2 性能分析

图 3~图 6 是 NAS kernel benchmarks 的性能数据.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,图 3~图 6 中没有 IS.

图 3 是分别使用 4 种存储优化技术后计算时间的变化情况,其中纵轴是使用优化技术后的计算时间与基准的计算时间的比值.从平均数据来看,4 种技术都会使计算时间变大.相比之下,数据预取和数据猜测使计算时间增加得不太明显,因为它们使 II 增加得不多,这与第 3.1 节的分析结果是一致的.与 CSMS 算法相比,FLMS 算法对计算时间的影响小些,因为它根据循环特点和 load 指令的局部性灵活地确定了 T_L,最终使得 SL 比 CSMS 算法的 SL 要小.

图 4 是分别使用 4 种存储优化技术后阻塞时间的改善情况.平均看来,数据预取最能减小阻塞时间,但它使几个程序阻塞时间变大,这是因为预取指令属于存取指令,数据预取方法会增加存储系统的压力.与 FLMS 算法相比,尽管 CSMS 算法的 T_L 比较大,但是对阻塞时间的改善程度要小些,这是因为过大的 T_L 会要求更多的存取指令被缓存起来,增加存储系统的压力,这也是 FLMS 算法需要结合循环特点的原因之一.数据猜测对阻塞时间

的作用不是很明显,因为数据猜测的 load 指令在整个执行的 load 指令中占的比例不大.

Table 1 Results of modulo scheduling

表 1 模调度的结果

Benchmark	Kind	II	SL	Benchmark	Kind	II	*SL*
BT	Base	11	38.84	CG	Base	6.81	16.22
	FLMS	11	48.32		FLMS	6.81	26.72
	Prefetch	11.14	39.16		Prefetch	6.88	16.28
	CSMS	10.98	48.56		CSMS	7	28.78
	Speculation	11	38.84		Speculation	6.81	16.22
EP	Base	11	57	FT	Base	8.88	22.53
	FLMS	11	69.67		FLMS	8.88	35.18
	Prefetch	11.33	57.33		Prefetch	9.06	21.88
	CSMS	14	70.5		CSMS	9.94	36.53
	Speculation	11	57		Speculation	8.88	22.53
IS	Base	7.4	11.6	LU	Base	11.29	44.85
	FLMS	7.4	16.2		FLMS	11.29	54.54
	Prefetch	7.4	11.6		Prefetch	11.35	45.04
	CSMS	7.4	18.2		CSMS	11.55	56.45
	Speculation	7.4	11.6		Speculation	11.29	44.85
MG	Base	9.65	24.77	SP	Base	12.30	35.29
	FLMS	9.65	35.86		FLMS	12.30	47.03
	Prefetch	9.86	24.77		Prefetch	12.40	35.33
	CSMS	10.53	36.25		CSMS	13.86	49.49
	Speculation	9.65	24.77		Speculation	13.12	37.54

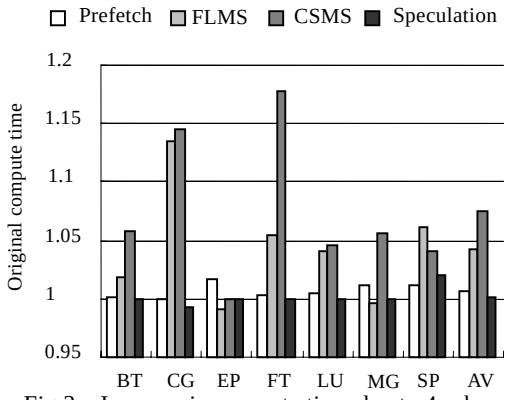


Fig.3 Increase in compute time due to 4 schemes

图 3 4 种存储优化技术的计算时间

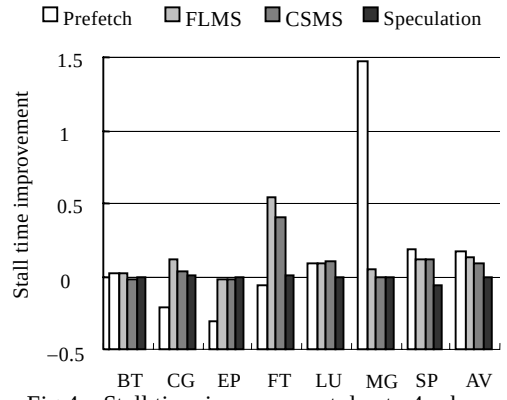


Fig.4 Stall time improvement due to 4 schemes

图 4 4 种存储优化技术的阻塞时间的改善

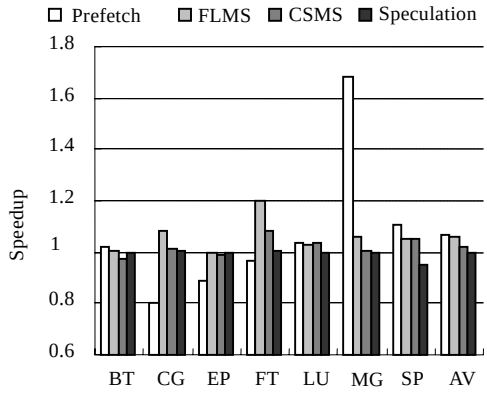


Fig.5 Speedup due to 4 schemes

图 5 4 种存储优化技术的性能加速比

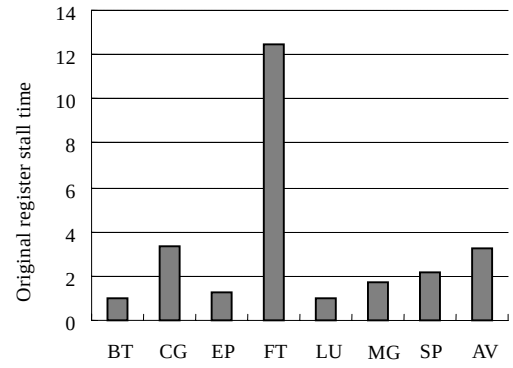


Fig.6 Increase in register tall time due to FLMS

图 6 使用 FLMS 后,寄存器阻塞时间的变化

图 5 是 4 种存储优化技术的性能加速比.可以看出,FLMS 算法不仅能够提高程序的性能,而且还比较稳定,没有造成一个程序的性能降低.

图 3~图 5 的平均数据的具体数值见表 2.

图 6 是使用 FLMS 算法优化后,寄存器阻塞时间的变化情况.可以看出,FLMS 算法较多地增加了寄存器开销.但是,表 3 所示的寄存器阻塞时间在程序运行时间中占的比重很小,因此 FLMS 算法对寄存器开销的影响不足以导致性能的降低.

表 4 是数据预取和 FLMS 算法结合后的性能加速比.两者结合后,发挥了各自的优点,使程序性能平均提高了 15.6%.由此可见,FLMS 算法并非独立,它能与其他存储优化方法结合,共同促进性能提高.

表 5 列出了 SPECfp2000 使用 FLMS 算法优化后的性能加速比.由于 ORC 在编译 wupwise 和 apsi 的过程中存在问题,因此表 5 中没有列出这两个程序.FLMS 算法优化 SPECfp2000 后,性能平均提高了 4.3%.

Table 2 Average various performance data of NAS kernel benchmarks due to 4 schemes

表 2 4 种存储优化技术下,NAS 基准测试程序的 3 种性能的平均数据

	Original compute time	Stall time improvement	Speedup
Prefetch	1.007	0.169	1.071
FLMS	1.042	0.128	1.061
CSMS	1.075	0.088	1.022
Speculation	1.002	−0.007	0.994

Table 3 The ratio of register stall time in execution time

表 3 寄存器阻塞时间在程序运行时间中的比例

BT	CG	EP	FT	LU	MG	SP	AV
1.95×10^{-3}	1.79×10^{-7}	7.25×10^{-8}	6.62×10^{-5}	3.30×10^{-5}	4.63×10^{-7}	2.71×10^{-6}	2.94×10^{-4}

Table 4 Speedup due to FLMS combined with prefetch

表 4 FLMS 与数据预取结合后的性能加速比

Benchmark	BT	LU	CG	MG	EP	SP	FT	AV
Speedup	1.012	1.047	1.036	1.734	0.889	1.157	1.215	1.156

Table 5 Speedup of SPECfp2000 due to FLMS

表 5 FLMS 优化 SPECfp2000 后的性能加速比

Benchmark	Speedup	Benchmark	Speedup	Benchmark	Speedup	Benchmark	Speedup
swim	1.069	mgrid	1.096	applu	1.053	mesa	1.010
galgel	1.104	art	1.104	equake	1.046	facerec	1.000
ammp	1.005	lucas	1.022	fma3d	1.016	sixtrack	1.000
AV	1.043						

4 结 论

存储优化对提高程序性能有着重要的意义.数据预取、数据猜测等存储优化技术被应用到模调度中.它们在实现程序性能提高的同时,也暴露出一些不足.本文提出了一种延迟可计算的模调度算法(FLMS).它根据局部性、存储系统和循环特点,为 load 指令确定了合理的延迟,在实现隐藏存储延迟的同时,避免了计算时间过分增大.FLMS 算法还能与数据预取技术较好地结合起来.实验结果表明,FLMS 算法有效地提高了编译器的性能.

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] 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.

[4] Ju RDC, Nomura K, Mahadevan U, Wu LC. A unified compiler framework for control and data speculation. In: Hurson AR, ed. Proc. of the 2000 Int'l Conf. on Parallel Architecture and Compilation Techniques. IEEE Press, 2000. 157−168.

[5] Sanchez FJ, Gonzalez A. Cache sensitive modulo scheduling. In: Proc. of the 30th Annual IEEE/ACM Int'l Symp. on Microarchitecture. IEEE Press, 1997. 338−348.

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

[7] Collard JF, 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] 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.

[9] 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.

[10] Intel Corp. Intel@ Itanium@ 2 Processor Reference Manual For Software Development and Optimization. Intel Corporation, 2004.