第23卷 第6期 计 算 机 学 报 Vol. 23 No. 6
2000年6月 CHINESE J. COMPUTERS June 2000

576-584

指令级并行编译器的数据预取及优化方法

连瑞琦 张兆庆 乔如良
(中国科学院计算技术研究所 北京 100080)

TP311

摘 要 微处理器芯片的处理能力越来越强,但是,存储器的速度却远远不能与其匹配,造成了整个系统的性能不理想.为解决这个问题,编译器发展了局部性优化、数据预取等多种技术.文中将介绍一种用于ILP(Instruction level Parallelism)优化编译器的数据预取技术以及一种利用寄存器堆减少主存访问次数、对程序进行优化的方法.利用它们可以提高平均存储性能,对科学和工程计算的应用是相当有效的.

关键词 数据预取,时间局部性,寄存器堆,预取优化
中图法分类号: TP311

指令级并行编译器

A Data Prefetching Method Used in ILP Compilers and Its Optimization

LIAN Rui-Qi ZHANG Zhao-Qing QIAO Ru-Liang
(*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100080)

Abstract With the development of instruction level parallelism (ILP) technology, the processing capability of microprocessor has been increasing dramatically. Unfortunately, the speed of the whole system has not kept pace because of the imperfect speed of memory. To improve the performance of memory, locality optimization and data prefetching are developed. This paper introduces an approach of data prefetching used in ILP compiler and a method to optimize the memory system by decreasing the frequency of accessing memory. The scheme can improve the average performance of memory, especially for the science and engineering application.

Keywords data prefetching, temporal locality, register file, prefetching optimization

1 引 言

随着芯片制造工艺的提高和指令级并行技术的广泛应用,基于微处理芯片的计算机系统的速度不断提高,今天它们在多项性能上都能和十年前的大型机相匹敌,所以,这种计算机系统越来越多地被用于科学和工程计算领域.然而,在这样一个系统中,如果没有足够强大的存储子系统,那么开发并行性、进行指令级并行的能力就得不到足够的发挥.不幸的是,近年来,存储系统的性能提高速度远远落后于处理器性能的提高.所以,绝大多数的计算机系统采用了一级或多级 cache 以弥补不足.但是,cache 是为通用代码所设计的,对于科学和工程计算应用来说,它的效果并不十分理想.

在有 cache 的基础上,解决存储器速度慢这个问题的方法有两种:一是加快访存速度,二是隐藏访存延迟.数据局部性优化,如循环分块、交换、倾斜、反转[4,5]等可以改变数据的局部性,它是加快平均访存速度的优化方式.而隐藏访存延迟可以通过设立缓冲区和数据预取来实现.存储器写操作的延迟可以尽量通过写缓冲区来实现.设立写缓冲区可以使

收稿日期:1999-06-22;修改稿收到日期,2000-03-19.**连瑞琦**,女,1973年生,博士研究生,研究方向为程序分析与优化及并行编译技术.**张兆庆**,女,1938年生,研究员,博士生导师,研究领域包括并行编译技术、并行程序设计环境、自动并行化与并行可视化工具以及并行程序的正确性验证等.**乔如良**,男,1937年生,研究员,研究领域包括并行编译技术、并行程序设计环境与工具、存储复杂性.

处理器不等一个写操作执行完就可以继续后继操作,这样不仅可使流水线不停,而且可使流水线中多个写操作同时执行.然而,存储器的读操作却不能简单地通过设立缓冲区来解决.因为,一般应用中处理器要读取一个数据是因为它立即要用这个数据.所以,通过缓冲区方式来解决这个问题是很难的.一般的商用微处理器芯片都不支持这种方案,而是通过数据预取的方法来解决这个问题的.数据预取是将取数据和用数据人为地分开,使它们能够充分并行地执行,而不会因为等待数据去停止流水线.数据预取又可以分为两种方式:硬件控制的数据预取和软件控制的数据预取.硬件控制的数据预取是硬件动态地发射预取指令.它不需要增加预取指令,且能够运用只有在运行时才能得到的信息[7,8].但是,硬件数据预取必须建立在能动态预测分支的基础上,所以预取的范围很小,且会增加系统整体开销.软件控制的数据预取是靠编译器插入预取指令,提前把数据取入 cache,这种方法不增加系统开销,不会减慢访问 cache 的速度,在有预取指令的指令系统的支持下,是一种很灵活有效的数据预取方式.

国际上,数据预取已得到了广泛的重视和研究.Rice 大学的 Porterfield 等人第一个提出用于单处理器的软件预取方案[12],具体地说,它是在预处理时将所有的内层循环的数组元素访问提前一个循环迭代.在此基础上,他又提出一种基于依赖向量的较为复杂的方案.UIUC 的 Chen,Mahlke 等人指出了他们用于 IMPACT 的数据预取方案[6],他们的方案可以用于非数值计算的应用.在此方案中,尽可能地将取数据及其相关的地址计算提前,而无须考虑 LD 和 ST 之间的数据依赖.系统靠设置预取更新寄存器(update register)、寄存器冲突缓冲区(register conflict buffer)和预取缓冲区(preload buffer)等专门的硬件来预取数据.它们能保证使那些可能和其后执行的 ST 指令有依赖关系的取数结果无效并更新之.Stanford 大学的 Mowry 等人提出他们用于 SUIF 的预取方案[1]:在局部性分析的基础上插入预取指令代码.在此之上,他们还提出了当且仅当需要预取时才插入预取指令的观点.总之,局部性分析结果、迭代次数、cache 大小以及其它条件应综合考虑,才能保证程序中的预取指令是必要且有效的.此外,Power PC[9]和 PA-8000[10]等机器中也采用了各自的行之有效的预取方案.本文提出了一种适用于软流水系统的预取方法,它用在一个支持指令级并行的微处理器芯片的编译器中,通过分析 II(Initial Interval)和访存操作之间的关系,使数据预取技术有效地用于软流水系统中.

数据预取的目的是使所有的访存操作全部命中 cache.然而,对于数值计算类型的应用来说,即使进行了上述预取处理,仍然有潜力可挖.它们的计算形式的规整性为我们进一步利用它的数据局部性提供了方便.本文将着重介绍进一步利用数据局部性的一种数据预取优化方法.此优化方法充分利用局部性分析结果和硬件体系中提供的两个大寄存器堆.通过将一部分具有时间局部性的数据预取到 RF 中,进一步减少访存所用的时间.

本文的剩余部分是这样安排的:第 2 节介绍以上所提到的用于 ILP 编译器的预取方法,第 3 节详细介绍怎样利用寄存器堆进行预取优化的具体算法和一个实例.第 4 节给出一些实验结果以及对这些结果的分析,最后是结论和今后的工作.

2　数据预取

2.1　系统简介

本编译系统的目标机是一种支持指令级并行的微处理芯片——M-Machine.它有 16 个独立的算术逻辑部件,在同一个周期内可以同时发射 4—6 条指令,这些指令组成一条宏指令,被并行执行.存储系统由 DRAM,cache 和两个寄存器堆组成.Cache 的大小是 1M,寄存器堆是 256 个 64 位的寄存器.这两个寄存器堆对编译器是完全可见的,即编译器可以决定哪些数据将放入寄存器堆,哪些数据将被替换出寄存器堆,以及怎样分布它们.在这个存储子系统中,在 cache 命中的情况下,取一个数需要 2 个周期,每两个周期只能发一条访存指令.这样的供数能力显然不能满足每周期 4—6 条指令的用数需要.所以,即使在 cache 命中的情况下,访存速度仍然是制约处理器利用率的关键.但是,从寄存器堆中取一个数只需要一周期,且每个周期可以发两条从寄存器堆中取数的指令,无须间隔.显然寄存器堆的供数速度是主存的四倍.所以,怎样充分利用好寄存器堆从而加快供数速度就成为极迫切的要求了.本文提出的预取优化方法就是为了充分地利用这两个寄存器堆.现在计算机体系中,寄存器堆或寄存器文件是十分常见的,故本文提出的优化方法也是相当具有推广价值的.

针对这样的体系结构，编译器的主要任务是充分调度指令，尽可能地开发程序的并行性. 它在开发指令级并行性的关键阶段，采用了先进的选择调度(selective schedule)算法，以确保软流水的技术可以用于有分支的循环上. 本编译器在优化方面，除了常规优化编译器的功能之外，还增加了许多有利于开发指令级并行性的优化，如缩短循环中依赖环的长度，删除跨迭代的公共子表达式等. 本文介绍的预取优化是其中十分重要的一种优化.

2.2 数据预取

本系统采用软件控制预取的方式. 在软件预取的方法中，通过由编译器插入预取指令来实现尽量使访存命中 cache 的效果. 任何一个软件预取方法中都包含有以下两个必不可少的部分：分析和调度. 编译器通过重用和局部性分析来决定哪些数据被预取. 首先，简要介绍一下有关的重用和局部性的概念.

定义 1. 重用(reuse)是指在一个程序中同一数据被使用两次以上.

定义 2. 假定数组元素存取式 $A[Hi+c]$的引用矩阵是 H，A 具有时间重用性(temporal reuse)的意思是指存在迭代 i_1 和 i_2 引用相同的数组元素，也就是 i_1 和 i_2 使方程 $H(i_1-i_2)=0$ 成立. 换句话说，该数组引用的重用存在于方向 r 上，r 满足 $Hr=0$. 该数组引用的重用空间是以上方程的解空间 kerH. 该解空间不空，我们就认为该数组引用具有时间重用性.

定义 3. 局部迭代空间(localized iteration space)是可以携带局部性的一些循环迭代的集合. 它是根据循环的次数和 cache 的大小、替换规则等性质求得的.

定义 4. 如果该数据被再次使用时，它仍在 cache 中，那么它具有局部性(locality). 局部性分析是重用分析的结果和给定 cache 的局部迭代空间综合考虑而得到的[5]. 时间局部性就是时间重用性和局部迭代空间的交集. 其中，如果用向量空间(vector space)来表示重用和局部性存在的空间，Reuse Vector Space 表示一个数据实例存在的重用空间，Locality Vector Space 表示它的局部性存在的空间，它们之间的关系如下：

$$Reuse\ Vector\ Space \cap Localized\ Iteration\ Space \Rightarrow Locality\ Vector\ Space \quad (1)$$

这些分析，我们采用了前人成熟的算法[5]. 余下的工作就是决定何处的数据将遇到 cache 不命中的情况以及怎样插入预取指令，将它们提前取到 cache 中.

2.2.1 cache 失效的判定和分析

在这里，我们用预取谓词来表示数据的 cache 命中与否[1]. 每一次迭代中，当该数据实例的预取谓词为真时，它将会遇到 cache 不命中的情况. 在一个循环起始值为 0 的循环中，具有时间局部性的数据的预取谓词仅在第一个迭代中为真(例如，$i=0$)；具有空间局部性的数据的预取谓词是当新的 cache 行开始时为真；具有成组局部性的数据组中不是第一个元素的数据的预取谓词永远为假；其余情况的数据的预取谓词永远为真，如图 1 所示. 同一数组元素存取式在多层循环嵌套中，数据的预取谓词是它各维预取谓词相与的结果.

局部性类型	将发生不命中的实例	预取谓词
无局部性	每个迭代中	True
时间局部性	第一个迭代中	$i=0$
空间局部性	每 l 个迭代中的第一个(l 是 cache 行的大小)	$i \bmod l=0$
成组局部性(是该组中第一个引用)	每个迭代中	True
成组局部性(不是该组中第一个引用)	不会出现	False

图 1

2.2.2 预取调度

在分析的基础上，预取调度将根据数组元素存取式的局部性特征的不同，采用不同的变换方案. 如对具有时间局部性的数据(图 1 中第二行的情况)，采用循环剥离(loop peeling)的方式变换，把 $i=0$ 的情况分离出来，从而使循环内的访存不会遇到不命中的情况. 对于具有空间局部性的数据，一般采用循环展开(loop unrolling)和流水预取结合的方式. 循环展开可以消除循环中有关预取指令的分支条件，而软流水则是通过在将遇到 cache 不命中的数据前若干迭代插入预取指令而避免 cache 不命中的情况. 因为要进行软流水调度，我们既要消除分支语句对调度带来的影响，又不希望由于循环展开增加代码长度对调度增加负担，所以采用了条件语句的方式，控制预取语句的执行与否. 究竟提前多少迭代插入预取指令，通常由以下公式决定

$$c=\left\lceil \frac{s}{l} \right\rceil,$$

其中，c 是预取指令应提前的迭代数，s 为 cache 不命中情况下访存需要的时间，l 是一个迭代需要的最短执行周期数. 在软流水系统中这个公式就不确切

了.预取要在调度之前进行,所以我们无法预知一个循环的迭代时间.我们把 l 替换为 II,因为 II 才是调度后每个迭代可能的最短执行时间.所以,以上公式变为

$$c=\left\lceil \frac{s}{II} \right\rceil .$$

2.2.3　实例和结论

我们用下面的例子,来说明预取的效果.

例 1.

```
for (i=0; i<100; i--){
    A[i+1]=A[i]+1;
}
```

假设 cache 行的长度为 2,cache 不命中需 24 周期.操作 load,store,add 的执行周期数各为 2 周期,依赖环长 6 周期.通过循环展开和软流水变换后,上例变为如下形式:(p 为语句的执行谓词)

```
prefetch(A[0]);
for (i=0; i<4; i+=2)          /*前缀*/
    prefetch (A[i+1]);
for (i=0; i<100; i++){        /*稳定状态*/
    p=((i mod 2)==0) && (i<94);
    prefetch(A[i+5])  p;
    A[i+1]=A[i]+1;
}
```

从此例可以看出,用流水的方式预取后产生的循环后继部分,通过条件语句并入其上的循环,减少了循环的开销.产生的先导循环的循环次数如果不多,还可以展开和其前面的语句混合,它们可以并行执行,进一步减少 *prefetch* 造成的开销.这样,程序的代码长度就会基本保持不变,既进行了数据预取,又不至于增加调度的负担.

通过插入预取指令,当真正需要循环中的数据时,便不会发生 cache 不命中的情况,不会使流水线中断,好处是十分明显的.但是,在 cache 命中的情况下,存取数耗费的周期数仍然很多,以致在许多应用中,即使全部的访存都命中 cache,数据存取仍然是加快其运行速度的瓶颈.这就是我们研究预取优化的动机.

3　预取优化

3.1　预取优化分析

首先,看下面一个例子.

例 2.

```
for (i=0; i<16; i++)
for (j=0; j<16; j++)
{
    …= A[i+j][i+j];
    …= A[16i+j][16i+j];
}
```

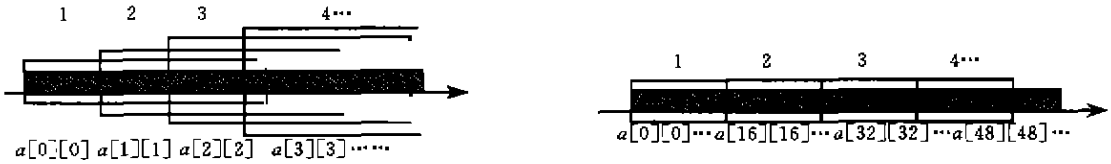
假定通过以上所讲的预取变换,使访存全部命中 cache.对 $a[i+j][i+j]$ 做进一步分析可以看出,实际访存的次数是 16×16=256 次,整个循环中实际用到的数只有 16+16=32 个.这是具有时间局部性的数组元素存取数的共性,即在整个过程中有许多冗余的取数操作.减少冗余取数,在以前因为寄存器的短缺是可望而不可及的.现在,寄存器堆和寄存器文件的出现,为解决这个问题提供了可能性.我们这里的预取优化是指利用寄存器堆存放具有时间局部性的数据,从而减少从主存中取数的冗余操作.与通常意义的预取相似,预取优化也分为两阶段:分析阶段和程序变换阶段.

3.1.1　时间局部性分析

分析阶段首先要完成的一个任务是分析得到具有时间局部性的数组元素存取式.上文讲过,时间局部性空间是局部迭代空间和时间重用性空间的交,如式(1)所示.其中,局部迭代空间是一组可以携带局部性的循环.但是,精确地求局部迭代空间就相当于预测在某时刻 cache 中有哪些数据,是十分复杂的,故通常使用的求近似局部迭代空间的方法是把每一维当作一个整体,看它是否可携带局部性.这种方法将求局部性空间简化为看某一层循环内用到的数据量是否大于 cache 的大小,如果大于,则该局部性空间中不包括该层循环.由于我们这里是要看数据预取入寄存器堆是否可以缩短访存时间,而不是判断该数再次使用时还在不在 cache 中,所以,我们不采用这种求法.我们用以下两点判据来判断一个具有时间重用性的数据是否具有时间局部性.

第一,我们要考虑循环界的大小.如例 2 循环中,$A[i+j][i+j]$ 和 $A[16i+j][16i+j]$ 都具有时间重用性.通过和该循环嵌套的局部迭代空间相交,具有时间局部性的只有 $A[i+j][i+j]$,$A[16i+j][16i+j]$ 就不具有时间局部性.因为 $A[16i+j][16i+j]$ 的重用空间是 $span\{(1,-16)\}$,而在循环的迭代空间和这个重用空间在整数域上交集为空,所以在整个循环中通过该数组引用访问的数据都是不同的,不存在重用.通过下面的图2可以看出 $A[i+j][i+j]$,$A[16i+j][16i+j]$ 访问数据的结果(横轴代表数据,方框代

表迭代,方框内是该迭代将要用到的数据),前者有重用而后者无重用.



(a) $A[i+j][i+j]$使用数据情况 (b) $A[16i+j][16i+j]$使用数据情况

图 2 例2中两个数组引用使用数据的情况

其次,我们要比较由于某数组引用式而产生的访存的次数和通过它实际访问到的数组元素的个数.因为要保证优化后的取数时间小于优化前的取数时间,所以只有前者大于后者一定程度时,对该数组引用式执行预取优化才是有意义的.当然,做以上判断,各层循环的上下界和步长信息是需要知道的.在我们的目标机中,从内存中取一个数需要 2 周期,而从寄存器堆中取一个数需要 1/2 周期,所以如果用 *size_data* 和 *size_iteration* 来分别表示要访问到的数组元素的个数和访存的次数,而且这些数据没被写过,那么要保证优化的效果,它们应满足以下不等式:

$$size\_iteration \times 2 > size\_data \times 2 + size\_iteration \times 1/2 \quad (2)$$

也就是:

$$size\_iteration > size\_data \times 4/3 \quad (3)$$

如果这些数据被写过,那么我们需要把数据再从寄存器堆写回内存,所以式(2)中的 *size_data* 的系数应为 4,*size_data* 和 *size_iteration* 应满足的关系是:

$$size\_iteration > size\_data \times 8/3 \quad (4)$$

所以,我们为了判别时的形式统一,也考虑其它作预取优化的开销,把是否作预取优化的标准定为

$$size\_iteration > size\_data \times 5 \quad (5)$$

由于满足后一条判据一定满足前一条判据,所以,我们仅使用第二条判据式(5)就够了.

3.1.2 时间局部性分析算法

判断一个数组引用式是否具有预取优化要求的时间局部性的算法如下.

算法 1.

输入:数组引用式 A 和它外层的循环集合 *loop_set*

返回值:表示该数组引用式是否具有时间局部性的布尔值

副作用:计算出 *loop_set* 中每个循环的执行次数记在数组 *loop_range* 中

计算出 A 中和 $H1$ 非全 0 行相对应的维的上下界记在数组 *upper_bound* 和 *lower_bound* 中

```
is_temporal_reuse(A,loop_set) {
    H=get_reference_matrix(A, loop_set);
    H1=Guass_elimination(H);
    none_all_zero_line=H1 中的非全 0 行的行数;
    num_of_column=H1 的列数;
    if (none_all_zero_line>=num_of_column)
                                    return False;
    size_iteration=1;
    对于每个 loop_set 中的 loop {
        loop_range[loop]=该循环将要执行的次数;
        size_iteration *=loop_range[loop];
    }
    size_data=1;
    对 A 中和 H1 非全 0 行相对应的维 dim{
        upper_bound[dim]=该维的下标的上界;
        lower_bound[dim]=该维的下标的下界;
        size_data *=upper_bound[dim]-
                    lower_bound[dim];
    }
    if (size_iteration>5×size_data) return Ture;
    else return False;
}
```

以上算法中,对 A 的引用矩阵 H 进行高斯消去,如果高斯消去后的矩阵 $H1$ 中非全零行的数目小于矩阵列的数目,那么它具有时间重用性.事实上,这一点恰恰保证了 H 的零空间不为空.接下来,计算 *size_iteration* 和 *size_data*. *size_iteration* 可由各循环的上下界和步长信息得到;$H1$ 中非全零行对应的维的长度相乘可得 A 的 *size_data*,各维长度是该维的下标的范围中包含的整数的个数,它可根据相关循环的循环次数得到.例如,以 i, j, k 为循环变量的三层循环嵌套中,i, j, k 都是从 0 到 9,各层循环的步长都为 1 的话,$a[i][i+j][0]$ 的 *size_data* 是 200,*size_iteration* 是 1000.例 2 循环中 $A[i+j][i+j]$和 $A[16i+j][16i+j]$的分析结果如图 3 所示.

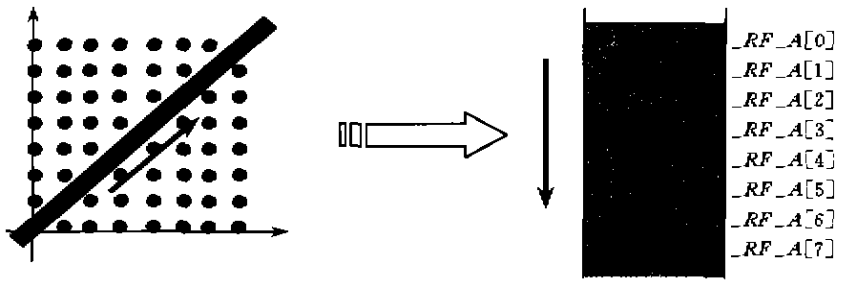
数组元素存取式	H	$H1$	*size-iteration*	*size-data*	是否具有时间局部性
$A[i+j][i+j]$	$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 1\\0 & 0\end{bmatrix}$	$16\times 16=256$	$16+16=32$	是
$A[16i+j][16i+j]$	$\begin{bmatrix}16 & 1\\16 & 1\end{bmatrix}$	$\begin{bmatrix}16 & 1\\0 & 0\end{bmatrix}$	$16\times 16=256$	$16\times 16+16=272$	否

图　3

经过这样的分析，我们确定了具有时间局部性的数组引用式，即预取优化的候选引用式. 如果一个循环中有若干个具有时间局部性的数组元素存取式，那么，我们将根据由 *size_iteration*/*size_data* 作为每个候选式的权值，由大到小排列. 权值大的优先分到寄存器堆中，直到寄存器堆分完为止.

3.2　程序变换

对数据的预取优化，是通过程序变换来实现的. 预取优化的变换步骤如下：首先，要做一个预变换，它分析一下要预取的数据是否能装入寄存器堆，即它的 *size_data* 是否比寄存器堆的大小 512 大. 如果放不下，就要在可能的情况下进行循环分块变换. 如不能分块，则放弃该候选式. 接下来，构造预取指令和包含该指令的循环嵌套. 要达到这个目的，就要先弄清原来的数据是怎样映射到寄存器堆中的. 例 2 中的数组引用式 $A[i+j][i+j]$和在寄存器堆中分给数组 A 的那部分的映射关系如图 4所示.



(a) 数组 A　　(b) RF 中分配给 A 的相应部分

图　4　数组引用与寄存器堆分配的对应关系

从图 4 可看出，预取的任务是将数组 A 对角线上的数据一次取出装入寄存器堆中相应部分. 例 2 中的循环可变换为如下形式(前缀_RF_表示是在寄存器堆中的数组)：

```
for (k=0; k<32; k++)
    _RF_A [k]=A [k][k];
for (i=0; i<16; i++)
    for (j=0; j<16; j++) {
        …=_RF_A [i+j];
        …=A[16i+j][16i+j];
    }
```

可以看出，在例 2 原来的循环之前，加了一个将数据预取入寄存器堆中的循环. 构造这个循环的算法如下.

算法 2.

输入：数组引用式 A 和 A 外层的循环集合 *loop_set*，A 的引用矩阵 H 和高斯消去后的形式 $H1$

副作用：构造循环和预取语句，加在原循环嵌套的前面

data_prefetching_opt(*A*,*loop_set*) {

```
对A 中和 H1 非全 0 行相对应的维 dim{
  upper_bound=upper_bound[dim];
  lower_bound=lower_bound [dim];
  stride=1;
  loop=gen_new_loop(upper_bound,lower_bound,
                    stride);
  将 loop 加入新循环嵌套 new_loop_nest;
}
left_hand=gen_RF_array_reference(new_
          loop_nest);
right_hand=gen_A_reference(new_loop_nest,H);
assign_stmt=gen_new_stmt(left_hand,
                         right_hand);
将 assign_stmt 插入新循环嵌套 new_loop_nest;
将新循环嵌套 new_loop_nest 插入原循环前;
}
```

算法 2 分为两步：第一步是构造预取语句所用的循环. 由于这里无法求到取数的精确步长，所以把步长规定为 1. 这样，预取到的数会多于或等于要用

到的数,即预取的数据是要用到的数据的超集.这样做造成了一定的浪费,但是前面判定它是否具有时间局部性的时候就保证了预取后关于该引用式的访存时间小于预取优化前的,这样仍然是值得的.例 2 中,$A[i+j][i+j]$的存在子空间是一维的,所以只要构造一个一维的循环,它的界由 $i+j$ 的上下界决定,也就是下界是 0,上界是 32.第二步是构造预取到寄存器堆的指令.这个指令的左端是一个_RF_数组的引用式,_RF_数组就是寄存器堆中分配给 A 的那一块,我们用该数组名加前缀_RF_来表示.该数组的维数是第一步所构造的循环的数目,各维的下标是相应循环的循环变量.右端是原数组的引用式,是要预取入寄存器堆中的数据,如变换后的例 2 中的 $A[k][k]$. $A[k][k]$是由 $A[i+j][i+j]$转变而成的.因为 $H1$ 是由 H 高斯消去得到的,并且在 $H1$ 中有全 0 行出现.所以 H 中的各行向量是线性相关的,换句话说,就是在消去之前,H 变为全 0 行的行向量是可以用其它行向量来线性表示的.所以,我们用代数的方法能很容易地得到这种表示.举一个例子,在三层循环 i, j, k 中,数组元素存取式 $a[i+2\times j+k][2\times i+3\times j+3k][i+j+2\times k]$的引用矩阵是 $\begin{pmatrix}1&2&1\\2&3&3\\1&1&2\end{pmatrix}$,经过变换为 $\begin{pmatrix}1&2&1\\0&-1&1\\0&0&0\end{pmatrix}$;并且,在变换过程中还可以得到 3 个行向量的关系是 $e_2-e_1=e_3$.如果我们对 $a[i+2\times j+k][2\times i+3\times j+3k][i+j+2\times k]$进行预取优化,且新构造的循环变量为 $k1$, $k2$,那么,预取入寄存器堆的指令的右段的数组引用式即为 $a[k1][k2][k2-k1]$.最后,将真正用数时的数组引用式的数组替换为带_RF_的数组.当然,如果数据被改动过,我们还会以同样的方式构造写回语句写回数据.

编译器后端会在生成代码时将带_RF_的数组引用自动翻译成从寄存器堆中的取数指令.每个数组元素的位置是寄存器堆中分给该数组的相应块的起址和通过下标计算得出的偏移量相加得到的.经过这样的程序变换,程序就可以先将具有时间局部性的数取入寄存器堆,真正用数时从寄存器堆中直接取用,从而达到减少访存时间的效果.

3.3 实 例

下面,我们给出一个较复杂的实例——矩阵乘的例子.

例 3.

```
for (i=0; i<64; i++)
    for (j=0; j<64; j++) {
        for (k=0; k<64; k++)
            C[i][j]+=A[i][k]*B[k][j];
```

局部性分析的结果是:

数组存取式	局部性分析结果	$H1$ 全 0 行所对应的循环
$A[i][k]$	$\begin{bmatrix}i\\j\\k\end{bmatrix}=\begin{bmatrix}none\\temperal\\spatial\end{bmatrix}$	循环 j
$B[k][j]$	$\begin{bmatrix}i\\j\\k\end{bmatrix}=\begin{bmatrix}temperal\\none\\none\end{bmatrix}$	循环 i
$C[i][j]$	$\begin{bmatrix}i\\j\\k\end{bmatrix}=\begin{bmatrix}none\\none\\temperal\end{bmatrix}$	循环 k

图 5

图 5 中列出数组存取式的引用矩阵全 0 行所对应的循环是为了找到插入预取语句的合适位置用的,因为预取语句应插在携带其时间局部性的循环之外才会有效.由于要预取的数据寄存器堆中装不下,所以在进行预取变换之前还要进行循环分块变换.并且发现数组 C 中要预取的数据每次只有一个,所以对它做循环不变量外提的优化,不作预取优化.经过相应的变换,以上循环变为如下形式.

```
for (i1=0; i1<64; i1+=8)
  for (j1=0; j1<64; j1+=8)
    for (k1=0;
      for (j=j1; j<j1+8; j++)
                          /*预取 B 中元素*/
        for (k=k1; k<k1+8; k++) {
          _RF_B[k-k1][j-j1]=B[k][j];
        }
      for (i=i1; i<i1+8; i++) {
        for (k=k1; k<k1+8; k++) {
                          /*预取 A 中元素*/
          _RF_A[k-k1]=A[i][k];
        }
        for (j=j1; j<j1+8; j++) {
          aa=C[i][j];
          for (k=k1; k<k1+8; k++) {
            aa+=_RF_A[k-k1]*
                  _RF_B[k-k1][j-j1];
          }
          C[i][j]=aa;
        }
      }
  }
```

上例中,我们可以看出原有访存的操作有 $4\times 64\times 64\times 64=1048576$ 次,而变换之后,预取 B 访

存 8×8×8×8×8=32768 次,预取 A 中数据也访存 32768 次,访问 C 中数据访存 2×8×8×8×8×8=65536 次,共计 131072 次,约为原来的 1/8. 专门为寄存器堆设计的预取指令预取成块数据的能力很强,它能按数据块的起址、大小、步长把数据从内存取到寄存器堆中,所以上例中有关预取的若干循环都没有什么循环开销. 优化后的程序中,访存操作加上新加的访问寄存器堆的操作所用的时间,也就是用于取数存数的周期数,大约是原来的 1/4. 另外,原来这个程序的最内层循环的 II(Initial Interval)是 4,是由访存操作决定的,因为最内层循环每个迭代至少要取两个数,需要 4 周期. 通过预取优化,最内层循环每个迭代取两个数只需 1 周期,II 不再由访存操作决定,所以预取优化完全隐藏了本例中访存延迟对执行效率的影响.

4 实验结果

预取优化在本编译器中已实现,我们选择了几个科学计算型的例子对它们进行模拟测试,下面将给出一部分测试的结果. 表 1 是我们给出的例子的描述,为了给出数据方便,我们缩小了它的规模,但足以说明问题.

表 1　测试用例描述

基准程序	问题规模	问题描述
Convolution	100×100	计算卷积
Mm	64×64	计算两矩阵的乘积
Kernel2	$N=100, loop=10$	Livermore kernel 2
Kernel6	$N=100, loop=10$	Livermore kernel 6
Kernel7	$N=100, loop=10$	Livermore kernel 7
Kernel19	$N=100, loop=10$	Livermore kernel 19

表 2 中列出了用或不用预取优化的情况下,从访存的次数,从寄存器堆取数的次数和总访存时间. 其中所有例子都是在做了数据预取的基础上的,也就是说,假设从主存取数时,都是 cache 命中的.

表 2　部分测试结果

基准程序	有无预取优化	访存次数	寄存器堆取数次数	总访存周期
Convolution	有	20100	0	40200
	无	400	20100	10850
Mm	有	1048576	0	2097152
	无	131072	524288	524288
Kernel 2	有	13790	0	27580
	无	300	13790	7495

续　表

基准程序	有无预取优化	访存次数	寄存器堆取数次数	总访存周期
Kernel 6	有	200000	0	400000
	无	50200	200000	200400
Kernel 7	有	10000	0	20000
	无	400	10000	5800
Kernel1 9	有	4000	0	8000
	无	400	4000	2800

从上表中的数字看来,大部分预取优化减少了 1/2 到 2/3 的访存时间,访存时间最少有可能减少到原来的 1/4. 数据预取优化明显地起到了优化的作用,使访存时间大大减少,而且携带数据的时间局部性的循环迭代次数越多,效果就越明显. 这些实例效果不等的原因有:有些例子对主存的数据有修改,所以必须在计算结束后再写回主存,增加了访问主存的次数. 有些例子数据规模太大,我们先做了循环分块,这样就不能把冗余取数全部消除. 虽然,由于其它诸多因素的影响,访存周期的减少幅度并不等于整个程序执行时间的减少幅度. 但是,对可以进行预取优化的实例来说,预取优化往往能明显减少执行时间,因为我们这里访存是阻碍提高执行速度的重要因素.

5 结论和今后的工作

随着处理器与存储器速度差异的增大,减少存储访问和隐藏访存延迟显得越来越重要. 在这方面,软件控制的数据预取由于简单易行,效果显著而得到广泛的应用. 在数据预取的基础上,本文介绍的利用寄存器堆做预取优化的方法加快了平均的访存速度,充分发挥编译器在这方面的作用. 但是,若想进一步改进以上方案,还需要硬件的配合. 如果,硬件在寄存器堆中加一位或若干个一致位,就可以减轻编译器的分析负担,增加被预取对象的范围,减少不必要的写回. 再如,硬件若加上了后台往寄存器堆中写数的功能,使我们可以以流水的方式取数,那么可以进一步节省取数时间.

本方法在一定程度上起到了隐藏访存延迟的作用,但也还存在一些需要改进的地方. 首先,我们选择要进行预取优化的数据时,是以单个循环嵌套为单位考虑的,而没考虑各循环嵌套之间的相互作用,从而丧失了一些重用的机会,导致了一些冗余. 在今后的工作中,我们将尽可能地通盘考虑数据之间的关系,更合理地安排数据预取. 其次,因为预取优化

对循环界、步长等信息要求比较高，所以它的适用范围还很有限，进一步扩大处理范围也是今后需做的工作.

参 考 文 献

1 Todd C M, Monica S L, Anoop G. Design and evaluation of a compiler algorithm for prefetching. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operation System, MA, 1992. 62—73

2 Colwell R P, Nix R P, O'Donnell J J *et al*. A VLIW architecture for a trace scheduling compiler. In: Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, 1987. 180—192

3 Dehnert J C, Hsu P Y-T, Bratt J P. Overlapped loop support in the cydra 5. In: Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS III), 1989. 26—38

4 Mowry T C. Tolerating latency through software-controlled data prefetch[Ph D dissertation]. Stanford, CA: Stanford University, 1994

5 Wolf M E, Lam M S. A data locality optimizing algorithm. In: Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario, 1991. 30—44

6 Chen W Y-W. Data preload for superscalar and VLIW processors[Ph D dissertation]. University of Illinois, Urbana-Champaign, 1993

7 Lee R L, Yew P C, Lawrie D H. Data prefetching in shared memory multiprocessors. In: Proceedings of the 16th International Conference on Parallel Processing, 1987. 28—31

8 Chen Tien-Fu, Baer Jean-Loup. Reducing memory latency via non-blocking and prefetching caches. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System, MA, 1992. 51—61

9 Bernstein D, Cohen D, Freund Avi, Maydan D E. Compiler techniques for data prefetching on the power PC. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Liamssol, Cyprus, 1995. 123—128

10 Vatsa S, Edward H G, Hsu Wei-Chung. Data prefetching on the HP PA-8000. In: Proceedings of the 24th International Symposium on Computer Architecture, Denver Co, 1997. 264—273

11 Nathaniel M. Compiler support for software prefetching[Ph D dissertation]. Rice University, TX, 1998

12 Callahan D, Kennedy K, Porterfield A. Software prefetching. In: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operation Systems, Santa Clara, CA, 1991. 40—52