ISSN1000-9825 Journal of Software 软件学报 2000,11(3):401~409

一种支持多重循环软件流水的寄存器结构*

容红波 汤志忠

(清华大学计算机科学与技术系 北京 100084)

E-mail: ronghb@mail.cic.tsinghua.edu.cn

摘要 寄存器结构及其分配是软件流水算法的关键之一.为支持多重循环的软件流水,该文提出一种新颖的寄存器结构:半共享跳跃式流水寄存器堆.它可以有效地解决多重循环软件流水下的特殊问题,即:同层次和跨层次的寄存器重命名问题以及断流问题;有效地消除外层循环的体间读写相关,提高程序的指令级并行度.它有3种分配方式可供灵活使用:单个寄存器、流水寄存器和寄存器组方式.流水寄存器方式对生存期确定的、局限于一个循环层次的寄存器重命名问题提供简单而有效的支持.寄存器组分配方式解决了多重循环软件流水时变量生存期不确定的情况.跳跃操作为解决断流问题提供了快速数据传送.工程实践表明,这种寄存器堆结构及其分配方式是十分有效的.

关键词 指令级并行,寄存器堆,流水寄存器,数据相关,软件流水.

中图法分类号 TP302

指令级并行ILP(instruction level parallelism)是当前国际上的研究热点.VLIW体系结构下的软件流水问题吸收了学者们的许多精力.从资源管理的角度看,一个软件流水算法要解决两个问题:(1) 在某时刻将功能部件分配给操作;(2) 在某时刻将寄存器分配给变量.通常,算法对功能部件不作特殊要求,但对寄存器结构一般要专门设计,以便互相配合,从而最有效地实现算法.寄存器结构及其分配直接影响到整个机器的性能,因此,这是软件流水算法的关键之一.

目前,国际上有多种软件流水算法[1~3],主要是针对单重循环的.对多重循环,一般只对最内层循环进行流水,对外层循环串行执行.ILSP(interlaced inner and outer loop software pipelining)算法[4,5]采用内外层循环交替执行方式,比较成功地解决了多重循环的软件流水问题.本文探讨其寄存器结构.

ILSP算法提出了一种新颖的寄存器结构:半共享跳跃式流水寄存器堆.它可以高效地实现功能部件(function unit,简称FU)之间的通信.在算法消除循环程序的所有体内数据相关、所有内层循环体间数据相关的基础上,由寄存器分配消除外层循环的体间读写相关.在操作调度中,只要遵守外层的体间写读相关即可.从而大幅度提高程序的指令级并行度.

由于寄存器的分配与算法密切关系,下面首先回顾一下ILSP算法,然后讨论其中的寄存器分配问题.

1 ILSP算法的简单回顾

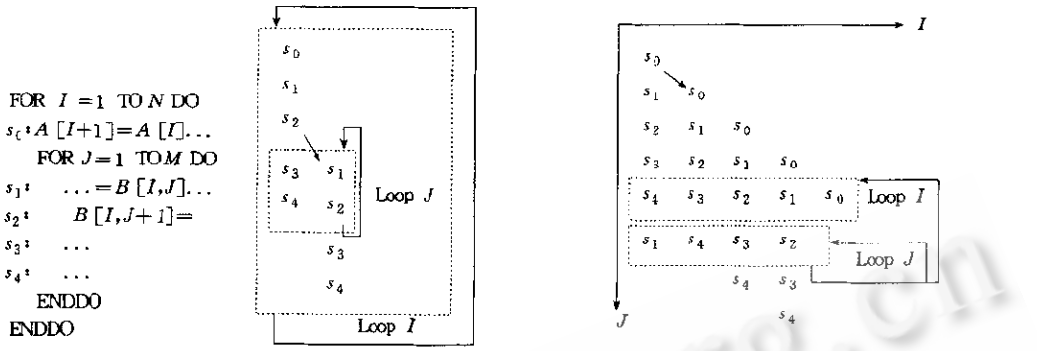
统计数字表明,对于典型的工程和科学计算,程序大部分时间都消耗在内层循环上[6].因此,当前的各种软件流水算法都致力于对最内层循环进行流水.但是,我们发现,这样做并非总是合理的.

图1中的有向线段表示写读相关.若从最内层的J循环进行流水,则循环启动间距II(initiation interval,简称II)为2,每两个周期执行一个J循环(如图1(b)所示).若从最外层的I循环进行流水,则II为1,每一个周期执行一个J循环.不仅II小,而且J循环执行周期也短(如图1(c)所示).

* 本文研究得到国家自然科学基金(No.69773028)资助.作者容红波,1972年生,博士生,主要研究领域为计算机并行编译,体系结构.汤志忠,1946年生,教授,博士生导师,主要研究领域为计算机并行算法,并行编译技术,并行体系结构.

本文通讯联系人:容红波,北京 100084,清华大学计算机科学与技术系

本文1998-12-31收到原稿,1999-04-02收到修改稿



(a) Source program
(a) 源程序

(b) Common software pipelining algorithms
(b) 通常的软件流水算法

(c) ILSP algorithm. The outer loop is unrolled along the abscissa, and the inner loop is unrolled along the ordinate
(c) ILSP算法. 外层循环沿横坐标展开，内层循环沿纵坐标展开

Fig. 1 Comparison between algorithms
图1 算法比较

注意到ILSP是从最外层的I循环开始进行流水的，但依然找到了J循环的重复模式. 因此，我们是先从外层循环进行流水，在发现内层循环的重复模式后，转入内层循环的流水；执行完内层循环后，再转回外层继续流水. 反复这个外→内→外的过程，直到整个程序执行完.

因此，对一个多重循环，不一定非要从最内层循环开始流水不可，而是可以选择一个最佳层次开始流水，产生一个内外层交错流水的方案. 选择标准将另文讨论. 本文假定从最外层开始流水. 凡是"外层"都指最外层.

2 数据寄存器结构

目前，主要的数据寄存器结构有以下几种.

(1) 全局共享寄存器堆[7~9]. 所有寄存器被所有功能部件及其内存共享. 其硬件结构复杂，每个寄存器的输入、输出端口很多，可扩展性有限. 但它对编译的支持比较好.

(2) 局部寄存器堆[8]. 各FU只能访问属于本FU的局部寄存器，通过互连网或共享存储器来通信. 硬件结构最简单.

(3) 寄存器通道[10]. 在局部寄存器堆的基础上，在各FU的局部寄存器之间提供少量几条数据通道.

(4) 流水寄存器堆[3,11]. 每个FU的所有局部寄存器按顺序排成一个队列，称为垂直流水. 各FU的垂直流水线之间建立若干通道，称为水平流水. 整个寄存器堆像一个矩阵网格. 水平流水主要用于FU之间的通信，垂直流水用于寄存器重命名.

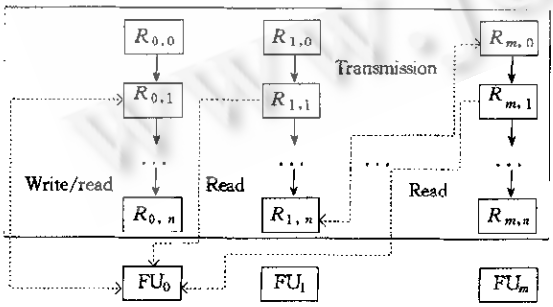


Fig. 2 Partly-Shared leaping pipeline register file
图2 半共享跳跃式流水寄存器堆

(5) 半共享寄存器堆[8]. 又分为写共享(写全局读局部)和读共享(读全局写局部)两种. 这是全局共享寄存器堆和局部寄存器堆的一个折衷. 读共享对编译的支持比较好，但硬件相对复杂.

我们综合了各种结构的优点，提出半共享跳跃式流水寄存器堆. 它是读全局写局部的，并且各FU的局部寄存器组织成流水，但是增加了任意两个寄存器之间数据传送的功能.

如图2所示，每个功能部件FU都有一条属于自己的局部寄存器流水线，所有FU的寄存器流水线形成整个寄存器堆. 流水方向为$R_{i,0}\rightarrow R_{i,1}\rightarrow\ldots\rightarrow R_{i,n}$，$i=0,1,\ldots,m$. 采用流水方式组织寄存器，可以简单、自然地解决不同循环体之间的寄存器重命名问题，并有助于生成高效代码.

半共享的含义是各个FU可以从整个寄存器堆中的任何一个寄存器读,但只能写入属于本功能部件的局部寄存器.这样,可以在硬件复杂度较小的情况下有效地支持优化编译以及各FU间的通信.

跳跃的含义是寄存器堆中的一个寄存器可以向另外任何一个寄存器直接传送数据.这就打破了流水寄存器只能在所属流水线中传送数据的惯例.这样做是为了解决内外层循环交错执行时的断流问题,并对FU间的数据传送提供快速支持.

3 寄存器分配

寄存器分配之后,根据变量(或常数)的生存期和位置的不同,整个堆的寄存器被分为3类(或者说该寄存器堆有3种分配方式):

(1) 单个寄存器.用于存放常数和生存期小于等于循环启动间距II的变量.常数在循环执行过程中不变;生存期小于等于II的变量,其相应寄存器的内容不会被下一个循环体的内容冲掉.因此,给它们分配1个寄存器即可.其分配方法有很多[1],本文不作讨论.

(2) 流水寄存器.分配给生存期大于II,并且局限于1个循环层次的变量.

在属于某个FU的一条流水线中,为该变量分配两个或更多个前后相连的寄存器,称为流水寄存器.流水寄存器属于同一个逻辑变量,却包含有多个物理寄存器.物理寄存器的个数就是流水寄存器的长度.

设流水寄存器为$R_{i,j}\rightarrow R_{i,j+1}\rightarrow\cdots\rightarrow R_{i,j+l-1}$,$l$是其长度.下面定义该流水寄存器的3种可用操作.① 写操作.只能向第1个寄存器$R_{i,j}$写,称$R_{i,j}$为尾部寄存器.② 读操作.可以从$R_{i,j},R_{i,j+1},\ldots,R_{i,j+l-1}$中的任何一个读.③ 流水操作.令$R_{i,j+k}=R_{i,j+k-1}$,$k=l-1,l-2,\ldots,1$,也就是这个队列向前流动一个节拍,队列头$R_{i,j+l-1}$的内容将被冲掉.

(3) 寄存器组.分配给生存期超过了1个循环层次的变量.

即那些在外层循环定义,在内层循环引用的变量;或者在内层循环定义,在外层循环引用的变量.这是多重循环流水引出的特殊问题.

4 流水寄存器的分配

在通常的软件流水算法下,最内层循环是并行的,因此要消除最内层循环的体间读写相关.但在ILSP下,外层循环是并行的,导致不同外层循环体的内层循环也并行,因此,需要消除外层循环的读写相关(包括不同外层循环体的内层循环之间的相关).流水寄存器的分配方式,实际上是通过硬件进行寄存器重命名来消除这些相关.流水寄存器只分配给生存期局限于一个循环层次的变量,因此,下面只以单重循环为例,所得结果对从外到内的各重循环都适用.

4.1 相关的消除

对于软件流水,II必须大于所有变量的生存期,才能保证本循环体定义的变量值不会被下一循环体的同一变量的值所覆盖.

变量的生存期是指变量的最后一次引用时刻减去它的定义时刻.在上一条指令中定义,紧接着在下一条指令中引用的变量,其生存期不足一个周期,认为是0.

设图3(a)中所有操作的执行时间为1个周期.由于变量T_1,T_2,T_3的生存期分别为1,0,0,软件流水最好的结果是以$II=2$启动.变量T_1的生存期制约了II.

为了减小II,如图3(d)所示,在源程序中插入操作$s_{1'}$:$T_{1'}=T_1$,则数据相关图变成图3(e).此时,所有变量$T_1,T_{1'},T_2,T_3$的生存期为0(定义后,立刻作最后一次引用),所以软件流水可以$II=1$启动,效率提高 倍.

如果为T_1和$T_{1'}$分配的是某个FU局部流水线上的两个相邻寄存器,那么,操作$s_{1'}$:$T_{1'}=T_1$就是流水操作.对比图3(b)和图3(e),可以看到,从s_2到s_0的体间读写相关被消除了.

4.2 流水寄存器的长度

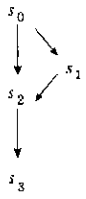
令流水寄存器的长度为l,则

$$l=\lfloor \text{变量的生存期}/II \rfloor+1, \tag{1}$$

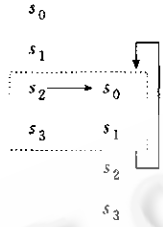
其中 II 是指在只考虑体间写读相关的情况下,软件流水的启动间距.

FOR $I=1$ TO N
DO
s_0: $T_1=A[I]$
s_1: $T_2=T_1+1$
s_2: $T_3=T_1+T_2$
s_3: $A[I]=T_3$
ENDDO

(a) Source program
(a) 源程序



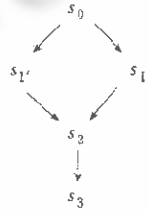
(b) Data dependency graph. The directed edges represent flow dependency
(b) 数据相关图.箭头代表写读相关



(c) Software pipelining results. The directed edge represent the loop-carried anti-dependency turned from the loop-independent flow dependency
(c) 软件流水结果.箭头代表由体内写读相关转化而来的体间读写相关

FOR $I=1$ TO N DO
s_0: $T_1=A[I]$
s_1: $T_2=T_1+1$
$s_{1'}$: $T_{1'}=T_1$
s_2: $T_3=T_{1'}+T_2$
s_3: $A[I]=T_3$
ENDDO

(d) The equivalent program after change
(d) 修改后的等价程序



(e) The new data dependency graph
(e) 新的数据相关图

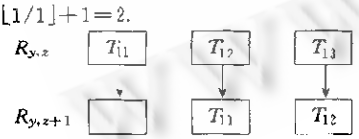
Fig. 3　Eliminating anti-dependency
图3　消除读写相关

变量的生存期通过分析源程序中的写读相关而得到.假设有写读相关$(OP_i \rightarrow OP_j, d)$,$d$ 为体差(iteration difference or iteration distance),表示 X 号体的操作 OP_i 对某个变量 A 赋值,$X+d$ 号体的操作 OP_j 对 A 引用,则这个写读相关所确定的 A 的生存期为

$$t_j+d^*II-t_i-delay(OP_i), \tag{2}$$

其中 t_i 和 t_j 分别表示 OP_i 和 OP_j 在同一个循环体中的启动时刻,$delay(OP_i)$是指 OP_i 的执行时间.列出所有与变量 A 有关的写读相关,分别算出生存期,取最大者,即为 A 的生存期.

以图3中的变量 T_1 为例.在只考虑体间写读相关的情况下,软件流水的启动间距 II 可以为1.由源程序(如图3(a)所示)中的写读相关$(s_0 \rightarrow s_1, 0)$,$(s_0 \rightarrow s_2, 0)$,根据公式(2),得 T_1 的生存期分别为0,1.取最大者,则 T_1 的生存期为1,即 T_1 被定义后,最多再隔1个周期,就会被消耗掉.由公式(1),应给 T_1 分配的流水寄存器长度为 $\lfloor 1/1 \rfloor+1=2$.



(a) $I=1$　(b) $I=2$　(c) $I=3$

Fig. 4　The effect of a pipeline register
图4　流水寄存器的作用

假设 T_{1i} 表示 $I=i$ 时语句 s_0 对 T_1 的赋值,且流水寄存器为 $R_{y,z} \rightarrow R_{y,z+1}$,则当 $I=1$ 时,将 T_{11} 写入 $R_{y,z}$(如图4(a)所示).$I=2$ 时,先执行流水操作,将 T_{11} 送往下一个寄存器 $R_{y,z+1}$,再将 T_{12} 写入 $R_{y,z}$(如图4(b)所示).同样地,当 $I=3$ 时,先执行流水操作,将 T_{1i} 送往下一个寄存器 $R_{y,z+1}$,再将 T_{13} 写入 $R_{y,z}$,注意,此时 T_{11} 的生存期已经结束,所以被冲掉了(如图4(c)所示).

这样,在变量的生存期未结束以前,流水寄存器一直保存它,直至其生存期结束.

4.3　流水寄存器的寻址与代码生成

一般地,在源程序中,假设关于变量 A 有写读相关$(OP_i \rightarrow OP_j, d)$,且流水寄存器的尾部寄存器为 $R_{y,z}$,则 OP_i 访问的寄存器为 $R_{y,z}$,OP_j 访问的寄存器为 $R_{y,z+k}$,其中

$$k=\lfloor (t_j+d^*II-t_i-delay(OP_i))/II \rfloor, \tag{3}$$

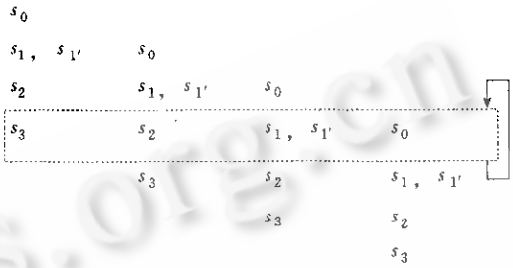
t_i, t_j 是 OP_i, OP_j 在同一个循环体中的启动时刻;$delay(OP_i)$ 是 OP_i 的执行时间.

在生成代码时,每个对变量进行赋值的操作之后都要附加流水操作.

例如,对于图 3 中的变量 T_1,由源程序(如图 3(a)所示)中的写读相关$(s_0 \rightarrow s_1, 0)$确定 s_0 中的 T_1 应访问的寄存器为 $R_{y,z}$,而 s_1 中的 T_1 应访问的寄存器也是 $R_{y,z}(k=\lfloor(1+0*1-0-1)/1\rfloor=0)$;由$(s_0 \rightarrow s_2, 0)$确定 s_2 中的 T_1 应访问的寄存器为 $R_{y,z+1}(k=\lfloor(2+0*1-0-1)/1\rfloor=1)$.附加流水操作后的等价程序如图 5(a)所示,生成代码如图 5(b)所示.对照图 3、图 4 和图 5,其正确性不难分析.

```
FOR I=1 TO N DO
s0: R_{y,z} = A[I]
s1: T2 = R_{y,z} + 1
s1': R_{y,z+1} = R_{y,z}
s2: T3 = R_{y,z+1} + T2
s3: A[I] = T3
ENDDO
```

(a) The equivalent program after register allocation. The registers allocated for variable T_2 and T_3 are not shown for clearness
(a) 分配流水寄存器后的等价程序.为清晰起见,为 T_2 和 T_3 分配的寄存器没有写出



(b) Software pipelining results after register allocation
(b) 分配流水寄存器后的软件流水结果

Fig. 5 Pipelined register addressing and code generation
图5 流水寄存器的寻址与代码生成

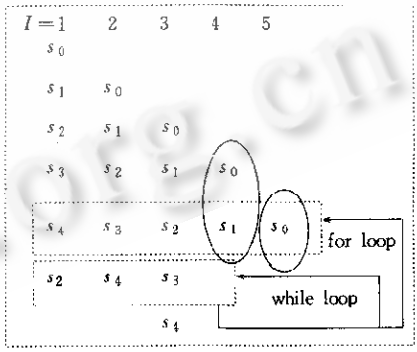
5 寄存器组的分配

ILSP 是一个贪心算法.只要内层循环的重复模式还没有出现,它就会再启动一个外层循环.这样,对于跨循环变量,造成了两个特殊的现象,即变量的生存期与该变量属于第几个外层循环体有关;变量在赋值后可能长期睡眠.

如图 6 所示,A 是一个跨循环变量.当 $I=1,2,$ 或 3 时,A 的生存期为 2.也就是说,A 在赋值后,至多再过 2 个周期,就会被最后一次引用.但在 $I=4,5$ 时,A 的生存期变得不确定了.图中用圆圈标出了 $I=4,5$ 时所进行的操作.以 $I=4$ 为例,外层循环只执行了操作 s_0, s_1 之后,就被迫停止,因为此时发现了内层循环的重复模式,下面将转入内层循环的流水,专门执行 $I=1,2,3$ 时的内层循环.在此过程中,$I=4$ 时所定义的那个 A 不被引用,处于睡眠状态.直到 $I=1,2,3$ 的内层循环中任意一个不满足 $X>Y$ 而结束时,才能继续 $I=4$ 时的其他操作.这时,$I=4$ 时所定义的那个 A 才苏醒过来,得到引用.

```
FOR I=1 TO N DO
s0: A=...
s1:  =A
s2: WHILE X>Y
s3:     A=A+1
s4:     .....
    ENDWHILE
ENDDO
```

(a) Source program
(a) 源程序



(b) Software pipelining results
(b) 软件流水结果

Fig. 6 Nondeterministic sleeping period of a variable
图6 变量的不定期睡眠

由于内层是一个 WHILE 循环,次数不定.因此,难以确定 A 究竟何时苏醒.这样,$I=4$ 时,A 的生存期是不确定的.

如果为 A 分配流水寄存器的话,按照式(1),要先得到 A 的生存期.但这是可变的,不能确定.此时,无法计算流水寄存器的长度.为解决这一问题,需要为 A 分配寄存器组.

5.1 寄存器组的分配、寻址与代码生成

在实践中,跨循环变量的数目很少.因此,可保守地假定其生存期是全局的,并为其分配寄存器组,以解决上

述问题.

设有一个跨循环变量 A,为 A 分配一组寄存器 $R_{f(i),z},R_{f(i+1),z},\ldots,R_{f(i-e-1),z}$. 其中 e 是寄存器组的深度,它也是同时打开的外层循环的个数;单射函数 $f(k)$指出第 k 号循环中的 A 所对应的寄存器的第 1 个下标;z 是第 2 个下标. 函数 f 保存在一张硬件表中.

e 由下式确定:

$$e=\lceil(\text{外层循环长度}-1)/II\rceil+1, \tag{4}$$

其中外层循环长度是指组成外层循环的所有操作的个数,包括内部嵌套循环的操作个数(这是因为,ILSP 算法将某一层循环内部的所有操作都看作是这一层的操作,无论是否嵌套有其他循环). 如图 6 所示,外层的 FOR 循环共有 5 个操作:s_0,s_1,s_2,s_3,s_4(包括内部 WHILE 循环的 3 个操作 s_2,s_3,s_4),因此,长度为 5. 则

$$e=\lceil(\text{外层循环长度}-1)/II\rceil+1=\lceil(5-1)/1\rceil-1=5,$$

即寄存器组的深度为 5,或者说,同时打开了 5 个外层循环.

寄存器寻址与代码生成十分简单:将流水结果(如图 6(b)所示)中的所有变量 A 用一个统一的"假"寄存器名,如 R_1,代替即可. 这个寄存器名只有一个下标,所以称为"假"寄存器名. 实际上,它代表 $R_{f(k),1}$,其中 k 是循环体号,由一组控制寄存器(称为循环控制寄存器组)跟踪保持. 也就是说,每个循环体的操作都隐含知道自己究竟属于哪个体. 这样,虽然所有循环体的代码是相同的,但是,经过硬件映射后,当 $I=1,2,3,4,5$ 时,A 分别映射到了不同的寄存器 $R_{f(1),1},R_{f(2),1},R_{f(3),1},R_{f(4),1},R_{f(5),1}$. 如果定义函数 f 为:$f(1)=0,f(2)=1,f(3)=2,f(4)=4,f(5)=3$,则 A 分别映射到了 $R_{0,1},R_{1,1},R_{2,1},R_{4,1},R_{3,1}$.

这样,对于图 6(b),不同 I 循环中的 A 映射到不同的寄存器. 在一次 I 循环结束前,它将保持这一映射. 它不需要考虑变量的生存期会发生不确定的情况,而完全按确定的情况去处理,因为不同循环体的 A 不会互相覆盖. 实际上,寄存器组方式是用分配一组单个寄存器的方法来实现寄存器重命名的,但它为了生成统一代码,使用了假寄存器名和映射表.

5.2 寄存器组与流水寄存器的比较

寄存器组和流水寄存器在逻辑上都是一个寄存器,但在物理上却包含多个寄存器. 寄存器组是通用方法,它可以代替流水寄存器的功能. 但是,它需要硬件表保持单射函数 $f(k)$,以及循环控制寄存器组来保持循环体号. 这张硬件表和循环控制寄存器组都需要动态更新.

流水寄存器可以简单、自然地消除体间读写相关,不需要其他硬件的支持. 但它仅限于变量的生存期确定的情况.

5.3 关于硬件映射表和 ILSP 的控制

ILSP 的控制机制主要包括一个循环控制寄存器组、一个系统返回堆栈、流水控制器、外层循环的装入排空控制器和内层循环的断流控制器等. 在循环运行之前,由编译向硬件提供这些控制信息,一旦循环进入运行,则硬件会利用控制信息自动控制循环的执行,并同步更新自身内容. 因此,这样不会增加循环的运行时间,或者说开销.

映射表如图 7 所示. 如果 VLIW 处理机共有 n 个功能部件,那么,它能同时执行的外层循环的个数最多是 n. 所以,映射表的长度定为 n. 一般地,n 取 32 已经足够了. 从假寄存器名到真寄存器地址的映射是一个简单的查表过程,如图 7 所示.

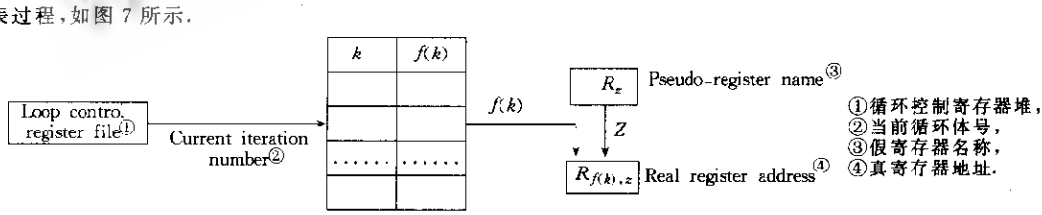


Fig. 7 The hardware mapping table and the loop control register file
图7 硬件映射表和控制寄存器组

6 跳跃操作

在ILSP算法下,特殊的"断流"问题要求寄存器堆中的任意两个寄存器能进行数据传送,而不能像传统的流水寄存器堆那样,仅在局部流水线中传递数据.

如图6所示,3个外层循环 $I=1,2,3$ 的内层 while 循环在同时执行.设 $I=1$ 时的那个 while 循环因为不满足 $X>Y$ 而停止("断流"),那么,$I=1$ 时的循环已经完成.此时,注意到 $I=4$ 的那个外层循环只执行了 s_0 和 s_1,其内层的 while 循环尚未执行.因此,应将它的内层循环补到断流的功能部件上,继续执行.但是,由 s_0 所定义的变量 A 位于另一个功能部件的局部寄存器中,这给将来重新定义 A 带来了困难,因为我们的寄存器堆是写局部的.所以,必须将 A 从那个功能部件的局部寄存器传到断流的功能部件的一个局部寄存器中,这就是跳跃操作.该操作发生在断流时.

跳跃操作支持寄存器间的快速数据传送.同时,它还对循环控制寄存器组进行调整,使补到断流处的新循环体能正常运行.

跳跃操作由断流控制器控制完成,过程如下:(1) 在断流发生的那个时钟周期的末尾,断流控制器发出断流信号,迫使所有FU暂时中断运行,进入等待周期;(2) 假设数据传送是从 $R_{u,v}$ 到 $R_{x,y}$,那么,断流控制器通知 FU_x 首先将 $R_{u,v}$ 的值读出,然后再写入 $R_{x,y}$(注意到寄存器堆是读全局,写局部的);(3) 调整循环控制寄存器组的参数,令所有FU转入正常运行状态.以上过程共占用两个周期.

7 实验结果

我们设计完成了一个指令级并行体系结构,优化编译器和体系结构的模拟系统.定义寄存器堆的大小为32×9(9个FU,每个FU有32个局部寄存器).分别按(1) 全局共享寄存器堆(单个寄存器)分配方式运行ILSP算法;(2)同时以单个寄存器、流水寄存器和寄存器组分配方式运行ILSP算法.通过实际运行一些典型的多重循环,得到了一些重要数据,节选见表1.

Table 1 Partial experimental results
表1 部分实验结果

Program①	Toral loop levels②	Number of the operations in the outer loop③	Number of the operations in the inner loop④	Cycles′⑤	Cycles″	Cycles‴	Speedup⑥
Inner product⑦ (100×100)	2	23	14	33 730	8 031	200	4.20
Fibonacci sequence⑧ (100×100)	3	28	10	1 152 685	338 031	200	3.4
Matrix sum⑨ (100×100)	2	23	17	34 440	8 299	200	4.15
Matrix mulplification⑩ (100×100)	3	32	14	2 339 478	649 855	200	3.60
Fandemon matrix⑪ (100×100)	2	32	16	16 936	7 247	200	4.38
Flower⑫ (1~1000)	3	36	17	2 196	835	46	2.63

说明:(1)′ 按全局共享寄存器方式运行、无跳跃操作时的时钟周期数
(Cycles using a global shared regiser file and without the leaping operation);
(2)″ 按3种寄存器分配方式运行、有跳跃操作时的时钟周期数
(Cycles using the three register allocation styles and with the leaping operation);
(3)‴ 跳跃操作和动态更新所占的时钟周期数
(Cycles used by leaping operations and dynamic refreshing).
①程序名称,②循环层数,③外层操作个数,④内层操作个数,⑤时钟周期数,⑥加速比,
⑦向量内积,⑧裴波那契数列,⑨矩阵加法,⑩矩阵乘法,⑪范德蒙矩阵,⑫找水仙花数.

从表中可以看出,采用半共享跳跃式流水寄存器堆,ILSP算法的加速比约为3.29.主要原因是:(1) 用硬件方法(流水寄存器和寄存器组)消除体间读写相关,避免了软件对寄存器重命名时的显式数据传送以及代码不统一的麻烦,从而缩小了循环启动间距和代码长度;(2) 跳跃操作用硬件提供断流时的快速数据传送以及一系列循环控制寄存器的更新,使代码无需考虑断流问题.不仅代码紧凑,而且断流与补流之间的衔接速度很快.

注:每执行完一次外层循环就要进行一次跳跃操作和动态更新,每次占2个周期,因此总周期数是外层循环次数的2倍.

8 半共享跳跃式流水寄存器堆和全局共享寄存器堆的比较

半共享跳跃式流水寄存器堆是对全局共享寄存器堆的简化.

多个功能部件的并行运行,使寄存器堆的组织成为一个重要问题.如果采用全局共享寄存器堆,堆的可扩展性将受到极大限制.因为要满足每个周期每个功能部件对寄存器堆执行的两个读操作和一个写操作,读端口和写端口的数目很多,译码和读写速度将会受到影响.

但是,如果将全局共享的寄存器堆划分为若干相对独立的部分,则可扩展性要好得多.可以有许多部分,但每部分规模都较小.进一步将每部分的寄存器串成队列,可扩展性就进一步提高了.因为队列无论有多长,相邻两个寄存器间的传送速度是不受影响的,而这是半共享跳跃式流水寄存器堆主要的数据传送方式.

简言之,半共享跳跃式流水寄存器堆是对全局共享寄存器堆进行划分,并重新组织为队列.在编译的帮助下,实现数据存取局部化,尽量减少各队列之间的数据传送.

跳跃操作和硬件映射表确实增加了硬件复杂度.但是,这是两种简单的机制,复杂度很低.所以,比较而言,半共享跳跃式流水寄存器堆的复杂度要比全局共享寄存器堆低得多,而且,对多重循环调度的支持也要好(见表1).

9 结束语

内外层交错流水给寄存器分配带来了特殊的问题,主要是同层次和跨层次的寄存器重命名问题,以及断流问题.本文对这些问题进行了深入分析,提出了具有普遍意义的寄存器结构和分配方法.从工程的实际结果来看,这些方法是十分有效的.

参考文献

1 Yu T, Tang Z Z, Zhang C H *et al*. Control mechanism for software pipelining on nested loop. In: Regina S S ed. Proceedings of the Conference on Advances in Parallel and Distributed Computing. Los Alamitos: IEEE Computer Society Press, 1997. 345～350

2 Luo Jun, Tang Zhi-zhong, Zhang Chi-hong *et al*. Algorithm to data allocation in software pipelining. Journal of Software, 1998,9(1):74～79)
(罗军,汤志忠,张赤红等.软件流水中的一种数据分配算法.软件学报,1998,9(1):74～79)

3 Rau B R, Fisher J A. Instruction-level parallel processing: History, overview and perspctive. Journal of Supercomputing, 1993,7(1/2):9～50

4 Hwang K. Advanced Computer Architecture: Parallelism, Salability, Programmability. New York: McGraw-Hill, 1993. 457～496

5 Lam M. Software pipelining: an efficient scheduling technique for VLIW architectures. SIGPLAN Notices, 1988,23(7): 318～328

6 Duesterwald E, Gupta R, Soffa M L. Register pipelining: an integrated approach to register allocation for scalar and subscripted variables. In: Kastens U, Pfahler P eds. Proceedings of the Compiler Construction: 4th International Conference CC'92. New York: Springer-Verlag, 1992. 192～206

7 Hoogerbrugge J, Corporaal H. Register file port requirements of transport triggered architectures. Professional Engineering, 1994,7(21):191～195

8 Tang Zhi-zhong, Wang Lei, Qian Jiang. Software pipelining on program with complicated loops. Journal of Software, 1996,7(7):422~427
(汤志忠,王雷,钱江. 多重循环的软件流水技术. 软件学报,1996,7(7):422~427)

9 NewBurn C J, Huang A S, Shen J P *et al*. Balancing fine-and medium-grained parallelism in scheduling loops for the XIMD architecture. Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (A-23). North Holland: Elsevier Science Publishers, 1993. 39~52

10 Su B G, Wang J, Tang Z Z *et al*. URPR-1: a single-chip vliw architecture. Microprocessing and Microprogramming, 1993,39(1):25~41

11 Wen Yu-hong. The research and design of VLIW optimizing compiler based on GURPR* global software pipelining algorithm [M S. Thesis]. Tsinghua University, 1991
(温钰洪. 基于GURPR*全局软件流水算法的VLIW优化编译器的研究与设计[硕士学位论文]. 清华大学,1991)

A Novel Register File Structure Supporting for Software Pipelining of Nested Loops

RONG Hong-bo TANG Zhi-zhong

(*Department of Computer Science and Technology Tsinghua University Beijing* 100084)

Abstract The structure and allocation of the register file is a key factor affecting the performance of software pipelining. To support software pipelining of nested loops, a novel register file, partly-shared leaping pipeline register file, is presented and its allocation is discussed. The register file effectively addresses the special problems in software pipelining nested loops, i.e., intra- and inter-level register renaming, and iteration discontinuity. Three allocation styles are designed for flexible application: single, pipeline and groupe register styles. Pipeline registers effectively supports intra-level register renaming with deterministic lifetime. A variable, which generally has deterministic lifetime, may have non-deterministic lifetime in software pipelining nested loops. Group registers address such a problem. To make a paused iteration continue execution again, a leaping operation provides quick data transmission between registers. The engineering practice has proved the efficiency of the structure and its allocation method.

Key words Instruction-Level parallelism, register file, pipeline register, data dependence, software pipelining.