一种运行时消除指针别名歧义的新方法*

汤志忠[1] 乔 林[1] 张赤红[1] 苏伯珙[2]

[1](清华大学计算机科学与技术系 北京 100084)

[2](William Paterson大学计算机科学系 美国)

摘要 提出一种采用软硬件结合的运行时消除指针别名歧义的新方法SHRTD(software/hardware run-time disambiguation).为延迟运行时不正确的内存访问及其后继操作,SHRTD的功能单元执行NOP操作.为保证所有延迟操作执行顺序的一致性,编译时就确定执行NOP操作的所有功能单元的顺序和NOP操作的数目.SHRTD方法适用于不可逆代码,同时它的代码空间受限,也不存在严重的代码可重入性问题.新方法有效地解决了指针别名问题,为获得潜在的指令级并行加速提供了可能.

关键词 指令级并行性,超长指令字,指针别名,运行时检查,运行时补偿.

中图法分类号 TP338

当前的超长指令字(very-long instruction word,简称VLIW)编译器都采用静态代码调度和软件流水的方法开发程序的指令级并行性(instruction-level parallelism,简称ILP)[1].这两种方法最大的局限是存在内存访问的歧义相关性(ambiguous dependence),因而即使编译器能够处理数组静态别名分析,也不能够很好地处理指针别名(pointer aliasing)分析.为了解决指针别名问题,以获得更高的潜在指令级并行处理加速,文献[2]提出了两种运行时消除歧义性(run-time disambiguation,简称RTD)的方法:运行时检查(run-time check)方法和运行时补偿(run-time compensation)方法.将这两种方法应用于软件流水时,运行时补偿方法虽然允许不确定的内存访问,但它只适合那些可逆代码[2].运行时检查方法虽然适用于任何代码,但存在代码可重入性(rerollability)问题.这两种方法共同的缺陷是存在严重的代码空间问题,尤其是在全局软件流水中可能导致巨大的补偿代码空间开销.

本文提出一种新的基于软硬件结合的运行时检查方法SHRTD.SHRTD的基本思想是:(1) 为延迟运行时不正确的内存访问及其后继操作,功能单元执行NOP操作而不是执行补偿代码;(2) 为保证所有延迟操作执行顺序的一致性,编译时就确定执行NOP操作的所有功能单元的顺序和NOP操作的数目.

1 SHRTD硬件基本结构

一个完整的指令级并行计算机加速系统主要由三大部分组成:主机、采用超标量体系结构的单处理机和采用VLIW体系结构的8个处理单元(PE)串联的多处理机.图1是一个简化的PE体系结构和SHRTD硬件支持环境.该体系结构包含7个功能单元:2个ALU、2个乘法器、2个内存访问端口和1个分支和循环控制单元(BRLC).该VLIW处理器能够在1个时钟周期中处理4个整数操作、2个内存访问操作和4个分支操作.SHRTD的硬件支持环境在指令存储器上添加了一个存储延迟操作的指令缓冲区、一个从指令缓冲区或正常的指令存储器选择操作的多路选择器集合和一个带有SHRTD WORD只读存储器的控制指令缓冲区.

* 本文研究得到国家自然科学基金资助.作者**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体系结构,并行算法,并行编译技术.**乔林**,1972年生,博士生,主要研究领域为计算机并行编译技术,Petri网,并行程序的形式语义.**张赤红**,1964年生,副教授,主要研究领域为计算机并行算法,并行编译技术.**苏伯珙**,1938年生,教授,主要研究领域为软件流水算法,并行编译技术.

本文通讯联系人:汤志忠,北京 100084,清华大学计算机科学与技术系

本文1998-05-11收到原稿,1998-09-01收到修改稿

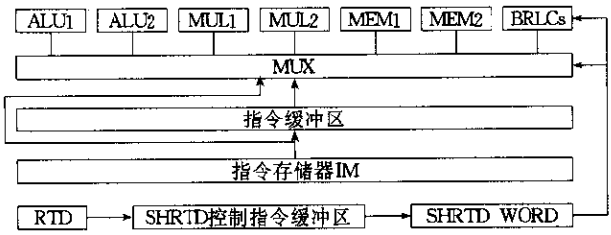

图1 单个PE的体系结构

2 相关定义与定理

本文假设:所有的操作都只占用 1 个时钟周期;所有的 PE 共享一个单一的内存片;且每个 PE 只有一个内存读、内存写和 BRLC 单元,每个 BRLC 单元可以同时处理 4 个分支操作.

定义 1(操作距离). 设 op_1 和 op_2 是程序中的两个操作,则它们之间间隔的操作数目加 1 称为这两个操作的操作距离,记为 $dis(op_1, op_2)$.

定义 2(安放距离). 设 op_1 和 op_2 是程序中已安放的两个操作,且在原始串行代码中操作 op_1 在操作 op_2 之前. 若安放后它们之间间隔的 VLIW 操作数目为 N,则这两个操作的安放距离

$$d(op_1,op_2)=\begin{cases}N+1, & \text{如果 } op_1 \text{ 在 } op_2 \text{ 之前},\\ 0, & \text{如果 } op_1 \text{ 和 } op_2 \text{ 在同一个 VLIW 中},\\ -N-1, & \text{如果 } op_1 \text{ 在 } op_2 \text{ 之后}.\end{cases}$$

定义 3(代码补偿量). 设 op_1 和 op_2 分别是程序中两个已安放的歧义 STORE 和 LOAD 操作,且它们的安放距离 $d(op_1,op_2)<0$. 当检测到地址冲突时,必须补偿一些空操作以延迟不正确的 LOAD 及其后继操作,我们称这些补偿的空操作数目为代码补偿量(code compensation measure).

显然,若 op_1 和 op_2 安放在不同的内存端口,则相应的代码补偿量 $\Omega=|d(op_1,op_2)|-1$.

定义 4(体内安放距离和体间安放距离). 对任意一个迭代次数为 n 的循环中的操作 op_1 和 op_2,设 $op_1^{(k)}$ 和 $op_2^{(j)}$ 分别表示 op_1 和 op_2 的第 k 次和第 j 次迭代,$1\leqslant j\leqslant n$,$1\leqslant k\leqslant n$. 如果 $j\neq k$,称安放距离 $d(op_1^{(k)},op_2^{(j)})$ 为体间安放距离;如果 $j=k$,称安放距离 $d(op_1^{(k)},op_2^{(j)})$ 为体内安放距离. 考虑到操作 op_1 和 op_2 在循环体不同迭代的体内安放距离是相同的,故可将体内安放距离简记为 $d_{inn}(op_1,op_2)$.

循环程序的软件流水算法必须在循环调度前确定循环的体间启动间距 II,即相邻两次循环迭代的第 1 个操作之间的体间安放距离. 一旦确定了循环体间启动间距 II,则有如下定理.

定理 1. 给定循环的体间启动间距 II. 设 $op_1^{(k)}$ 和 $op_2^{(j)}$ 分别是循环程序中两个已安放的歧义 LOAD 和 STORE 操作,且它们的体内安放距离为 $d_{inn}(op_1,op_2)$,体间安放距离 $d(op_1^{(k)},op_2^{(j)})<0$,$j<k$. 若 $op_1^{(k)}$ 和 $op_2^{(j)}$ 安放在不同的内存端口,则一次迭代需要插入的 SHRTD 操作个数 $p=\left\lfloor\frac{d_{inn}(op_1,op_2)}{II}\right\rfloor$.

证明:因为并行程序每隔 II 启动一次循环迭代,则在操作 $op_1^{(j)}$ 和 $op_2^{(j)}$ 之间共启动了 $\left\lfloor\frac{d_{inn}(op_1,op_2)}{II}\right\rfloor$ 次循环迭代. 注意到不在这段时间内启动的循环迭代并不存在歧义相关性,从而只需要在这些循环迭代所属的操作 $op_1^{(k)}$ 之前插入相应的 SHRTD 操作,以判断操作 $op_1^{(k)}$ 和 $op_2^{(j)}\left(j+1\leqslant k\leqslant j+\left\lfloor\frac{d_{inn}(op_1,op_2)}{II}\right\rfloor\right)$ 是否存在歧义相关性即可,$p=\left\lfloor\frac{d_{inn}(op_1,op_2)}{II}\right\rfloor$ 即为一次迭代需要插入的 SHRTD 数目. □

定理 2. 给定循环的体间启动间距 II. 设 $op_1^{(k)}$ 和 $op_2^{(j)}$ 分别是循环程序中两个已安放的歧义 LOAD 和 STORE 操作,当 SHRTD 检测到地址冲突时,相应的代码补偿量

$$\Omega=|d(op_1^{(k)},op_2^{(j)})|+1=d_{inn}(op_1,op_2)-(k-j)\times II+1.$$

证明:因为对存在歧义相关性的任意操作 $op_1^{(k)}$ 和 $op_2^{(j)}$,$\Omega=|d(op_1^{(k)},op_2^{(j)})|+1$ 为其代码补偿量. 设操作 $op_1^{(j)}$ 的启动周期为 t,则操作 $op_2^{(j)}$ 的启动周期为 $t+d_{inn}(op_1,op_2)$,操作 $op_1^{(k)}$ 的启动周期为 $t+(k-j)\times II$,故操

作 $op_1^{(k)}$ 和 $op_2^{(j)}$ 的体间安放距离

$$d(op_1^{(k)},op_2^{(j)}) = (t+(k-j)\times II)-(t+d_{inn}(op_1,op_2)) = (k-j)\times II - d_{inn}(op_1,op_2).$$

既然 $op_1^{(k)}$ 在 $op_2^{(j)}$ 之前启动，上述结果显然是负值. 当 SHRTD 检测到歧义相关性时，$op_1^{(k)}$ 已经超前执行了 $|(k-j)\times II-d_{inn}(op_1,op_2)|$ 个时钟周期，并读取了 $op_2^{(j)}$ 执行前的数据. 为保证程序的正确执行，必须在 $op_1^{(k)}$ 之前插入 $|(k-j)\times II-d_{inn}(op_1,op_2)|+1$ 个补偿空操作，从而有相应的代码补偿量 $\Omega=|d(op_1^{(k)},op_2^{(j)})|+1=d_{inn}(op_1,op_2)-(k-j)\times II+1$. □

3　SHRTD 基本原理

表 1 和表 2 说明了如何在软件流水过程中使用 SHRTD，原始的程序代码如图 2 所示，插入 RTD 代码之后的程序如图 3 所示. 表 1 是无地址冲突的软件流水结果，操作号和指针别名后加括号的上标编号表示该操作属于哪次迭代.

表 1　循环程序实例：无地址冲突时的软件流水结果

CLK	ALU1	ALU2	MUL1	MUL2	MEM1	MEM2	BRLC0	BRLC1	BRLC2
1			$op_1^{(1)}$						
2			$op_1^{(2)}$		$op_5^{(1)}$				SHRTD($op_5^{(2)}$,$op_8^{(1)}$)
3	$op_6^{(1)}$		$op_1^{(3)}$		$op_5^{(2)}$			SHRTD($op_5^{(3)}$,$op_8^{(1)}$)	SHRTD($op_5^{(3)}$,$op_8^{(2)}$)
4	$op_6^{(2)}$	$op_7^{(1)}$	$op_1^{(4)}$		$op_5^{(3)}$		SHRTD($op_5^{(4)}$,$op_8^{(1)}$)	SHRTD($op_5^{(4)}$,$op_8^{(2)}$)	SHRTD($op_5^{(4)}$,$op_8^{(3)}$)
5	$op_6^{(3)}$	$op_7^{(2)}$	$op_1^{(5)}$		$op_5^{(4)}$	$op_8^{(1)}$	SHRTD($op_5^{(5)}$,$op_8^{(2)}$)	SHRTD($op_5^{(5)}$,$op_8^{(3)}$)	SHRTD($op_5^{(5)}$,$op_8^{(4)}$)
6	$op_6^{(4)}$	$op_7^{(3)}$	$op_1^{(6)}$	$op_9^{(1)}$	$op_5^{(5)}$	$op_8^{(2)}$	SHRTD($op_5^{(6)}$,$op_8^{(3)}$)	SHRTD($op_5^{(6)}$,$op_8^{(4)}$)	SHRTD($op_5^{(6)}$,$op_8^{(5)}$)
7	$op_6^{(5)}$	$op_7^{(4)}$	$op_1^{(7)}$	$op_9^{(2)}$	$op_5^{(6)}$	$op_8^{(3)}$	SHRTD($op_5^{(7)}$,$op_8^{(4)}$)	SHRTD($op_5^{(7)}$,$op_8^{(5)}$)	SHRTD($op_5^{(7)}$,$op_8^{(6)}$)
8	$op_6^{(6)}$	$op_7^{(5)}$	$op_1^{(8)}$	$op_9^{(3)}$	$op_5^{(7)}$	$op_8^{(4)}$	SHRTD($op_5^{(8)}$,$op_8^{(5)}$)	SHRTD($op_5^{(8)}$,$op_8^{(6)}$)	SHRTD($op_5^{(8)}$,$op_8^{(7)}$)
9	$op_6^{(7)}$	$op_7^{(6)}$	$op_1^{(9)}$	$op_9^{(4)}$	$op_5^{(8)}$	$op_8^{(5)}$	SHRTD($op_5^{(9)}$,$op_8^{(6)}$)	SHRTD($op_5^{(9)}$,$op_8^{(7)}$)	SHRTD($op_5^{(9)}$,$op_8^{(8)}$)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

表 2　循环程序实例：SHRTD($op_5^{(6)}$,$op_8^{(4)}$)检测到 $op_5^{(6)}$ 和 $op_8^{(4)}$ 地址冲突

CLK	ALU1	ALU2	MUL1	MUL2	MEM1	MEM2	BRLC0	BRLC1	BRLC2
6	$op_6^{(4)}$	$op_7^{(3)}$	$op_1^{(6)}$	$op_9^{(1)}$	$op_5^{(5)}$	$op_8^{(2)}$	SHRTD($op_5^{(6)}$,$op_8^{(3)}$)	SHRTD($op_5^{(6)}$,$op_8^{(4)}$)	SHRTD($op_5^{(6)}$,$op_8^{(5)}$)
7	$op_6^{(5)}$	$op_7^{(4)}$	NOP	$op_9^{(2)}$	NOP	$op_8^{(3)}$	SHRTD($op_5^{(7)}$,$op_8^{(4)}$)	SHRTD($op_5^{(7)}$,$op_8^{(5)}$)	SHRTD($op_5^{(7)}$,$op_8^{(6)}$)
8	NOP	$op_7^{(5)}$	NOP	$op_9^{(3)}$	NOP	$op_8^{(4)}$	SHRTD($op_5^{(8)}$,$op_8^{(5)}$)	SHRTD($op_5^{(8)}$,$op_8^{(6)}$)	SHRTD($op_5^{(8)}$,$op_8^{(7)}$)
9	NOP	NOP	$op_1^{(7)}$	NOP	$op_5^{(6)}$	NOP	NOP	NOP	NOP
10	$op_6^{(6)}$	NOP	$op_1^{(8)}$	NOP	$op_5^{(7)}$	NOP	NOP	NOP	NOP
11	$op_6^{(7)}$	$op_7^{(6)}$	$op_1^{(9)}$	$op_9^{(4)}$	$op_5^{(8)}$	$op_8^{(5)}$	SHRTD($op_5^{(9)}$,$op_8^{(6)}$)	SHRTD($op_5^{(9)}$,$op_8^{(7)}$)	SHRTD($op_5^{(9)}$,$op_8^{(8)}$)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
for (i=0;i<n;i++)
{
 op1:R2=2*R1
 op2:R1=M(P)
 op3:R4=R2-R1
 op4:R4=R4+R3
 op5:M(Q)=R6
 op6:R7=R4*R5
}
```

图 2　原始循环体代码

```
for (i=0;i<n;i++)
{
 op1:R2=2*R1
 op2:SHRTD
 op3:SHRTD
 op4:SHRTD
 op5:R1=M(P)
 op6:R4=R2-R1
 op7:R4=R4+R3
 op8:M(Q)=R6
 op9:R7=R4*R5
}
```

图 3　插入 SHRTD 后的代码

从时钟周期1到时钟周期5是循环的装入阶段;时钟周期6之后是循环的流水阶段.在循环的流水阶段,每条VLIW指令其实执行的是相邻6次循环迭代的语句,其中各语句分属于不同的循环迭代,即一次循环迭代只需要一个时钟周期.当循环次数远远大于循环体内的操作时,循环装入和排空过程可以忽略不计,从而在无地址冲突时,程序的并行加速比约为6.

因为循环程序的体间启动间距 $II=1$,$d_{inn}(op_5,op_8)=3$,从而需要插入的SHRTD操作个数 $p=3$.这些SHRTD操作将分别判断紧接着的LOAD操作是否与前3次迭代的STORE操作存在循环体间内存地址冲突的问题,即是否存在体差为1,2或3的体间相关.当SHRTD$(op_5^{(6)},op_8^{(4)})$检测到 $op_5^{(6)}$ 和 $op_8^{(4)}$ 地址冲突时(时钟周期为6),必须添加一些补偿空操作,相应的代码补偿量 $\Omega=d_{inn}(op_1,op_2)-(k-j)\times II+1=2$.

此时,操作 $op_5^{(6)}$ 和 $op_8^{(4)}$ 的循环体差为2,$op_5^{(6)}$ 必须在 $op_8^{(4)}$ 之后完成,如表2所示.因为这两个操作的体间安放距离 $d(op_5^{(6)},op_8^{(4)})=-1$,所以功能单元必须插入两个NOP操作,这些NOP操作延迟了第6次迭代中有歧义的LOAD操作及其后继操作的执行.这里存在两个时钟周期的延迟,执行顺序在时钟周期11返回到正常状态,整个过程不存在任何代码可重入性的问题.

4 SHRTD的并行加速比分析

定理3. 设循环程序的体间启动间距为 $II=1$,循环的串行代码总长度为 $l=6$,循环次数为 n,设 op_1 和 op_2 分别是循环程序中两个已安放的歧义LOAD和STORE操作,且体间安放距离 $d_{inn}(op_1,op_2)=d=3$,则某次发生 j_3 次体差为3,j_2 次体差为2,j_1 次体差为1的地址冲突后的并行程序加速比 $S=\dfrac{6n}{n+3j_1+2j_2+j_3+8}$.发生 m 次地址冲突后的算术平均并行加速比 $\overline{S(m)}=\dfrac{6n}{n+2m+8}$.

证明:由定理2知,发生体差为 x 的地址冲突时的代码补偿量 $\Omega_x=d-x+1$,$1\leqslant x\leqslant d$,则在某次发生 j_3 次体差为3,j_2 次体差为2,j_1 次体差为1的地址冲突后,总的代码补偿量 $\Omega=3j_1+2j_2+j_3$.

串行执行该程序时,总的时钟周期为 $6n$,并行执行时装入和排空阶段分别需要5个时钟周期,在没有检测到地址冲突时,流水阶段需要 $n-2$ 个时钟周期.由于在运行时检测到地址冲突,则总的并行执行周期为 $(n-2)+2\times5+(3j_1+2j_2+j_3)=n+3j_1+2j_2+j_3+8$,从而此时程序的并行程序加速比 $S=\dfrac{6n}{n+3j_1+2j_2+j_3+8}$.在一次迭代的过程中,发生一次地址冲突后的算术平均代码补偿量 $\overline{\Omega(1)}=\dfrac{1}{3}(3+2+1)=2$,从而发生 m 次地址冲突后的算术平均并行加速比 $\overline{S(m)}=\dfrac{6n}{n+2m+8}$. □

当 $j_1=0$,$j_2=0$,$j_3=0$ 时,不存在任何地址冲突,加速比当 $n\to\infty$ 时的极限 $\lim\limits_{n\to\infty}S=6$;当 $j_1=n$,$j_2=0$,$j_3=0$ 时,全部地址冲突体差都为1,$\lim\limits_{n\to\infty}S=1.5$;当 $j_1=0$,$j_2=n$,$j_3=0$ 时,全部地址冲突体差都为2,$\lim\limits_{n\to\infty}S=2$;当 $j_1=0$,$j_2=0$,$j_3=n$ 时,全部地址冲突的体差都为3,$\lim\limits_{n\to\infty}S=3$.考虑到程序本身的特殊性——循环体内的所有操作都是不可并行的,获得这样的加速比还是令人满意的.

使用同样的方法可以分析检测到其他地址冲突时的指令级并行流水结果.

5 结 论

上面的例子表明,SHRTD方法有效地解决了指针别名问题,并获得了与使用软件实现的运行时补偿方法同样的效果.SHRTD方法可以与诸如内存缓冲区等硬件支持联合工作,以加快地址比较的速度.

SHRTD方法具有下述3个优势:(1) 因为运行时检查方法没有代码重做问题,所以它特别适合任何不可逆代码;(2) 因为任何SHRTD只需要一个SHRTD控制指令,补偿代码的代码空间并不大;(3) 不存在代码可重入性问题.

SHRTD方法需要下述的特别硬件支持:(1) 一个大小为 $D\times W$ 的指令缓冲区,W 是VLIW指令的宽度,D 等于 $d_{max}+1$,这里,d_{max} 是大多数流行程序中的最大值;(2) 一个多路选择器MUX,MUX的数目等于VLIW指

令字的操作域数目;(3) SHRTD 控制指令缓冲区和 SHRTD WORD 寄存器.

将来的研究将着重考虑如何处理嵌套循环和在流水安全法[3]中使用 SHRTD 方法.

参考文献

1 Rau B R, Fisher A. Instruction-level parallel processing: history, overview, and perspective. The Journal of Supercomputing, 1993,7(1):9~50

2 Nicolau A. Run-Time disambiguation: coping with statically unpredictable dependencies. IEEE Transactions on Computers, 1989,38(5):663~678

3 汤志忠,张赤红,乔林. 流水安全法——一个面向软件流水技术的新的数据相关性分析方法. 计算机学报,1998,21(增刊):201~206

(Tang Zhi-zhong, Zhang Chi-hong, Qiao Lin. Pipelining safe method——a new way to support data dependence analysis for software pipelining. Chinese Journal of Computers, 1998,21(supplement):201~206)

A New Run-Time Pointer Aliasing Disambiguation Method

TANG Zhi-zhong[1] QIAO Lin[1] ZHANG Chi-hong[1] SU Bo-gong[2]

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

[2](*Department of Computer Science William Paterson University USA*)

Abstract In this paper, a new run-time pointer aliasing disambiguation method, called SHRTD (software/hardware run-time disambiguation), which combines hardware and software techniques is presented. During run time, the SHRTD method lets function units execute NOPs to implement the postponement of the incorrect memory load operation and its successive operations. To guarantee the consistency of the execution sequence of all postponed operations, the order of function units which executes NOPs and the number of NOPs must be determined during compiler time. The SHRTD can be used for irreversible code, and it has very limited compensation code space and no serious rerollability problem. The SHRTD method solves pointer aliasing problem efficiently and makes it possible to obtain potential instruction-level parallel speedup.

Key words Instruction-level parallelism, very-long instruction word, pointer aliasing, run-time checking, run-time compensation.