软件流水的开销模型和决策框架*

李文龙+, 林海波, 汤志忠

(清华大学 计算机科学与技术系,北京 100084)

Cost Model and Decision Framework for Software Pipelining

LI Wen-Long+, LIN Hai-Bo, TANG Zhi-Zhong

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

+ Corresponding author: Phn: +86-10-51532108, E-mail: liwenlong00@mails.tsinghua.edu.cn, http://www.cs.tsinghua.edu.cn

Received 2003-02-25; Accepted 2003-10-16

Li WL, Lin HB, Tang ZZ. Cost model and decision framework for software pipelining. *Journal of Software*, 2004,15(7):1005~1011.

http://www.jos.org.cn/1000-9825/15/1005.htm

Abstract: Software pipelining tries to improve the performance of a loop by overlapping the execution of several successive iterations. Modulo scheduling is a kind of widely used scheduling technique. The drawbacks of software pipelining, such as increased register pressure, would sometimes degrade the performance improvement that software pipelining gains. This kind of cost varies with the processor architecture, compiler optimization, and characteristics of programs. In this paper, a program characteristics oriented cost model for software pipelining is proposed, and the cost is evaluated in some aspects. A dependency based cost testing (DBCT) algorithm is developed to provide information for the compiler to decide whether to apply software pipelining or not. Experimental results show that DBCT algorithm boosts performance greatly.

Key words: software pipelining; cost model; dependency analysis

摘 要: 软件流水是一种重要的指令调度技术,它通过重叠地执行不同的循环体来提高指令级并行性(instruction level parallelism,简称 ILP).模调度是一类被广泛采用的软件流水调度算法.软件流水并非一种无损的优化方法,它具有一定的开销,比如延长了编译时间、增加了寄存器压力等.而且,受到体系结构、调度算法以及程序特性的限制,进行软件流水并不一定能达到理想的加速比,有时反而会引起性能下降.提出了一种面向程序特性的软件流水开销模型,对此模型下的软件流水开销进行了量化分析,并提出了一种基于相关性分析的软件流水开销测试算法(dependency based cost testing,简称 DBCT),为软件流水决策提供了判断依据.实验结果表明,该算法收到了较好的效果.

关键词: 软件流水;开销模型;相关性分析

中图法分类号: TP338 文献标识码: A

* Supported by the National Natural Science Foundation of China under Grant No.60173010 (国家自然科学基金)

作者简介: 李文龙(1977－),男,辽宁鞍山人,博士生,主要研究领域为指令级并行算法;林海波(1978－),男,博士,研究员,主要研究领域为计算机系统结构,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.

软件流水[1]是一种重要的指令调度技术,它通过重叠地执行不同的循环体来提高指令级并行性(instruction level parallelism,简称 ILP).模调度是一类软件流水调度算法,它对一个循环体中的指令进行调度,使得相继的循环体在以固定的启动间距(initiation interval,简称 II)开始执行时,不发生资源冲突和相关冲突.模调度在 20 世纪 80 年代初被首次提出,此后一直是软件流水调度算法的研究热点,并已经在某些产品编译器中得以实现[2].要使经过软件流水调度后的循环能在目标机器上正确执行,编译器必须准确地为循环中的各个变量分配寄存器.已有较为成熟的软件流水寄存器分配算法[3,4],其中某些算法需要处理器体系结构的支持.

本文第 1 节描述软件流水的开销模型,并从寄存器压力、代码和调度等方面分析软件流水的开销.第 2 节提出基于相关性分析的软件流水开销测试算法.第 3 节是实验结果和分析.最后总结全文.

1 软件流水的开销模型

软件流水的调度算法有很多种,本文的讨论只限于模调度算法.模调度规定相继的循环体必须以固定的启动间距(II)启动,而且以相同的模式运行,从而简化了调度的复杂性.模调度使得同一个变量(包括源程序中的变量和编译器生成的临时变量)对应于多个运行实例,因此要求对寄存器进行换名.寄存器换名可以通过模变量扩展(modulo variable expansion,简称 MVE)的软件方法和旋转寄存器的硬件方法[5]来实现.对于有分支循环的软件流水,也有不同的实现方法,比如选择某条主路径进行软件流水,或通过条件执行消除分支,将多个基本块转化成单基本块进行软件流水.后者需要条件执行指令的硬件支持.

在基于模调度的软件流水算法中,II 是衡量软件流水性能的重要指标.II 常受到以下 3 个方面的限制:

限制 1(体系结构限制).例如,功能部件的个数、指令延迟的长短、寄存器的个数以及体系结构对软件流水的硬件技术,如条件执行、旋转寄存器、特殊的循环分支指令等.

限制 2(编译技术限制).例如,采用何种调度和寄存器分配算法以及该算法的性能等.

限制 3(程序特性限制).例如,循环的结构(DO-LOOP/WHILE-LOOP)、操作的类型、相关性约束、寄存器需求等.

定义 **1**(面向程序特性的软件流水开销模型). 在给定上述限制 1 和限制 2 的条件下,即对于特定的体系结构和编译技术,对不同程序特性的循环进行软件流水所带来的开销将有所不同,这种模型称为面向程序特性的软件流水开销模型.

同样地,可以定义面向体系结构的软件流水开销模型和面向编译技术的软件流水开销模型,以研究不同的体系结构(如流水线深度、指令发射宽度、寄存器结构等)和编译优化技术(如基于循环展开的编译优化等)对软件流水性能以及开销的影响,但此内容不在本文的研究范围之内.

在定义 1 所定义的模型中,本文假定以下的体系结构和编译技术:

- EPIC(explicitly parallel instruction computing)体系结构(IA-64),具有 9 个全流水的功能部件(2 整数、2 浮点、2 内存、3 分支),128 个寄存器(128 整数、128 浮点),支持寄存器旋转,支持条件执行(64 个条件位寄存器),支持特殊的软件流水循环分支指令(br.ctop,br.wtop 等).
- 基于 Huff 的模调度算法[3],基于 Rau 的寄存器分配算法[4].

基于以上模型,软件流水主要存在以下几方面的开销:

- 寄存器压力开销
- 装入和排空代码开销
- 初始化和清理代码开销
- 调度开销

1.1 寄存器压力

软件流水并行执行来自于多个循环体的指令,因此会产生较多的活变量,所需的寄存器也相应增加.即使对于同一个变量,不同的循环体也会对应于多个运行实例.在支持寄存器旋转的体系结构中,虽然可以通过硬件来进行寄存器换名工作,使得在软件流水代码中每个变量只对应于一个寄存器,但在运行时还需要占用多个物理

寄存器.在模调度过程中,为了安放指令,有时会拉长变量的生存期,这也是寄存器压力的一个来源.在软件流水之前,往往还需要进行循环展开、公共表达式删除、代码外提、冗余 LOAD/STORE 删除等编译优化,以提高功能部件的利用率、减小循环体的控制高度、松弛相关性限制等.而这些优化技术往往会增加指令条数或拉长变量的生存期,从而增加对寄存器的需求.

当软件流水所需的寄存器超过了物理寄存器个数时,就需要引入寄存器的 spill/refill 操作,即把寄存器中的值写入内存或从内存取出.这就引入了额外的 LOAD/STORE 操作,有时会抵消一些软件流水所带来的加速比.寄存器压力过高还会对程序的其他部分产生间接的影响,即虽然软件流水循环的执行速度提高了,但可能会导致程序其他部分(如循环前/后的代码)没有足够的寄存器可分配,从而引入多余的 LOAD/STORE 操作.当这些部分的执行频率较高时,这种开销就更大了.

1.2 装入和排空代码

软件流水线在充满之前要进行装入,在循环执行完毕后要进行排空.在装入和排空阶段,功能部件都没有被充分地利用.在 EPIC 体系结构中,虽然可以利用条件执行机制省去显式的装入和排空代码,但是在运行时仍然会通过设置条件位来形成装入和排空部分,如图 1 所示.

(p16)OP1
(p17)OP2
(p18)OP3
(p19)OP4

(a) Compile-Time code

	OP1			
Prologue	OP1	OP2		
	OP1	OP2	OP3	
Kernel	OP1	OP2	OP3	OP4
		OP2	OP3	OP4
Epilogue			OP3	OP4
				OP4

(b) Run-Time code

(a) 编译后的软件流水代码 (b) 运行时的软件流水代码

Fig.1 Compile-Time and run-time codes of software pipelined loop

图 1 编译后和运行时的软件流水代码

装入和排空代码的开销与启动间距 II 和软件流水的段数 SC(stage count)有关.一般来说,装入或排空代码所需的时钟周期数 T 可用下面公式表示:

$$T=(SC-1)\times II.$$

当循环次数较少时,装入和排空代码的开销就相对比较显著.另外,如果软件流水的循环不是最外层循环,则当外层循环的执行次数较多时,也会加大装入和排空代码带来的开销.

1.3 初始化和清理代码

与软件流水的装入和排空代码不同,初始化和清理代码是指在软件流水开始装入前和排空完毕后所需的代码.例如在本文所涉及的模型中,在装入之前,需要进行寄存器的初始化、设置段数、旋转寄存器基址指针(rotating register base,简称 RRB)清零等操作;在排空之后,也需要重新设置旋转寄存器基址指针等.

初始化和清理代码的执行次数与装入和排空代码相同,因此,当循环次数较少,或循环所在的外层循环执行次数较多时,开销会比较明显.

1.4 调 度

为了开发循环体间的并行性,在进行软件流水的调度时,有时会拉长循环体内的调度,以安放来自其他循环体的指令.这在整体上提高了功能部件的吞吐量,但当循环次数较少时,会使得在到达峰值吞吐量之前就退出了软件流水,这时反而不如用普通的列表调度代替软件流水.

模调度有时还会产生单段流水循环.

定义 2(单段流水循环). 如果模调度后循环的段数为 1,即 $SC=1$,则此循环称为单段流水循环.

单段流水循环意味着循环体之间是串行执行的,即只开发了循环体内的并行度.从理论上讲,在这种情况下使用循环体内的无环调度算法也可以达到同样的效果,而且可以省去软件流水的装入、排空等开销.另外,模调度比一般的无环调度要复杂,需要占用较长的编译时间.

多个循环体的并行执行,也要求访问多个循环体所对应的数据,这在一定程序上会破坏数据访问的局部性,从而导致 cache 缺失率的提高.

2 基于相关性分析的软件流水开销测试算法

由于软件流水在编译和运行时都具有一定的开销,因此一个智能的编译器在进行软件流水时应该保证有足够的性能提高,以抵消其相应的开销.为此,本文提出一种基于相关性分析的软件流水开销测试算法 DBCT(dependency based cost testing),为编译器提供有效的软件流水决策支持.

2.1 算法描述

软件流水的动机在于开发循环体间的并行性,它特别适用于循环体内的相关限制严重、而循环体间相关限制宽松的情形,即当循环体内可并行的指令不足时,软件流水可以从其他循环体中抽取可以并行的指令,以提高并行度.而当循环体内的并行度较高时,进行软件流水就没有太多的优势.

给定一个循环的数据相关图(控制相关已通过 IF-conversion 转化为数据相关,因此只需考虑数据相关关系)(data dependency graph,简称 DDG)$G(V,E)$,其中 N 是结点的集合,每个结点表示一条操作;E 是边的集合,每条边 $e(vi,vj,\min,\mathrm{diff})$表示从结点 vi 到结点 vj 的相关关系(假定输出相关和反相关已经通过寄存器换名消除),min 表示 vi 的操作延迟,diff 表示相关关系所跨越的循环体,称为相关距离.对于图 G 中的任意环路 θ,可以计算由数据相关导致的软件流水启动间距的下限 RecMII:

$$RecMII=\max_{\forall\theta}(\min_{\theta}/diff_{\theta}).$$

删掉 G 中 $diff>0$ 的边,即不考虑体间相关关系,对于 G 中的任意一条路径,可以计算此无环调度的关键路径长度 CPL(critical path length):

$$CPL=\max_{\forall path}(\min_{path}).$$

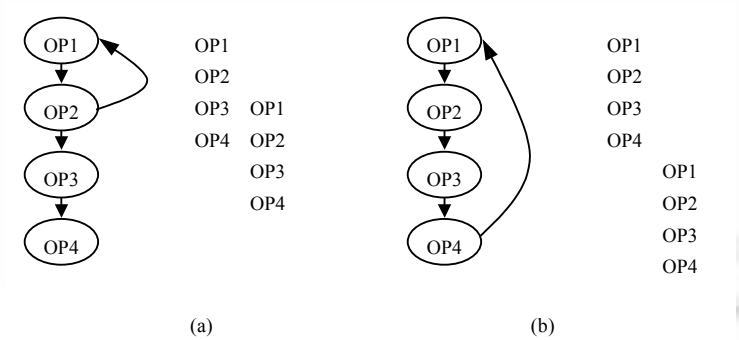

Fig.2 DDG and software pipelined code

图 2 数据相关图与软件流水代码

当 $CPL<=RecMII$ 时,意味着循环体间相关关系严重,此时进行软件流水没有很大优势,即可放弃软件流水,进行无环调度.如图 2 所示为两个数据相关图的示例,假定它们表示一个最内层循环体的相关关系.图 2(a)和图 2(b)的体内相关关系都很严重(OP1->OP2->OP3-> OP4).在图 2(a)中,只有一个由 OP1 和 OP2 构成的环(强连通块),即不同循环体的 OP1 和 OP2 都必须串行执行,而不同循环体的 OP3 和 OP4 却可以并行执行,此时利用软件流水恰好可以开发这种并行性.在图 2(b)中,所有的操作构成一个环,即这 4 条操作必须严格串行执行,此时就无须进行软件流水了.

2.2 软件流水决策框架

在进行软件流水之前,需要对循环的特点进行分析,以判断是否需要进行软件流水.除 DBCT 算法之外,还可以根据第 1 节的分析得到以下一些放弃软件流水的启发式:

软件流水循环的执行时间:

$$Time_{SWP}=Time_{K}+Time_{P\&E}+Time_{I\&C}+Time_{S}.$$

$Time_K$ 代表软件流水核心的执行时间,它等于启动间距与核心执行次数的乘积,$Time_{P\&E}$ 代表软件流水的装入和排空部分执行时间,$Time_{I\&C}$ 代表初始化和清理代码部分的执行时间,$Time_S$ 代表处理机花费在 Cache Miss 上的等待时间.

当循环次数较小时.循环次数小意味着软件流水线处于充满状态的时间短,$Time_K$的值很小,而其他3项因为与循环的执行次数没有太大关系,因此这些开销会因为软件流水的核心执行时间较小而变得更加显著,在这种情况下,放弃软件流水,借助于循环展开和无环调度,可以避免软件流水的装入和排空开销,减小初始化和排空代码的执行时间,加快循环的执行速度.

当循环体的操作较多时.操作较多通常意味着可以并行的操作较多,循环体内的并行度已经可以充分利用功能部件,没有必要通过折叠循环体来提高资源的利用率;另外一个考虑是,过多的操作会加重模调度的负担,导致编译时间过长.

整个软件流水决策框架如图 3 所示.其中的参数 $C1$ 表示循环次数的下限,$C2$ 表示循环体内操作个数的上限,$C1$ 和 $C2$ 的值可以通过实验测得.

```
Bool Decision_For_SWP {
    /* 1. loops with low trip count */
    If (trip_count<C1) return false;
    /* 2. loops with too many operations */
    If (num_of_ops>C2) return false;
    /* 3. DBCT algorithm */
    Build_DDG(g);
    RecMII=Compute_RecMII(g,cyclic);
    CPL=Compute_CPL(g,acyclic);
    If (CPL<=RecMII) return false;
}
```

Fig.3 Software pipelining decision framework
图 3 软件流水决策框架

放弃软件流水改用无环调度的循环执行时间:

$Time_{No\ SWP}=CPL\times\#iterations+Time_{I'\&C'}+Time_{S'}$,

其中$\#iterations$ 代表循环的执行次数,$Time_{I'\&C'}$代表初始化和排空代码的执行时间,$Time_{S'}$表示发生 Cache Miss 时处理机的等待时间.

相比软件流水,无环调度能够更好地利用数据访问的局部性特点.实验结果表明,软件流水导致 Cache 的效率变差,缺失率增大,对于同一个循环,软件流水花费在 Cache Miss 上的时间不会小于无环调度中的这部分时间,即 $Time_S>=Time_{S'}$.

在初始化和排空代码开销方面,无环调度因为不需要寄存器的旋转支持,不必进行流水的装入和排空,所以初始化和排空部分的代码相比软件流水要少得多,这方面的开销也要小于或者等于软件流水相对应的部分,即 $Time_{I\&C}>=Time_{I'\&C'}$.

软件流水的核心执行时间 $Time_K=II\times\#kernal\ iterations$,其中$\#kernal\ iterations$ 代表核心的执行次数.当循环的执行次数很大时,$\#kernal\ iteraions$ 近似等于循环的执行次数,即$\#kernal\ iterations\approx\ \#iterations$.另外,因为 II=Max($RecMII$,$ResMII$),当 $CPL<=RecMII$ 时,可以推出 $CPL<=II$.比较软件流水和无环调度的各部分执行时间可以发现,当 $CPL<=RecMII$ 时,软件流水的执行时间大于或等于无环调度循环的执行时间,即 $Time_{SWP}>=Time_{No\ SWP}$,因此在这种情况下放弃软件流水改用无环的调度是比较明智的选择.

3 实验结果

本节将以 NAS kernel benchmarks 和 SPECfp2000 中的部分程序为例,对软件流水的开销和 DBCT 算法的有效性进行测试.全部程序采用 ORC[6]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 Huff 的模调度算法和 Rau 的寄存器分配算法,对于不做软件流水的循环,进行列表调度.编译的结果在 Itanium 处理器[7]上执行,Itanium 是第一代 IA-64(EPIC)体系结构的处理器,对软件流水提供条件执行、旋转寄存器、循环分支指令以及循环次数计数器等硬件支持.

3.1 软件流水的开销

Itanium 处理器提供了一种称为寄存器堆栈引擎(register stack engine,简称 RSE)的硬件机制[8],专门用来处理寄存器的 spill/refill,RSE 访问的频率可以在一定程度上反映动态的寄存器压力.表 1 列出了通过 pfmon 测得的软件流水前后的 L2 cache miss 和 RSE 访问情况,数据表明,软件流水极大地提高了 cache 缺失率和 RSE 访问频率,特别是 FT,软件流水使得其 RSE 访问频率提高了 245 倍,这将极大地降低软件流水所带来的加速比.

表 1 的最后一列显示了单段流水循环占所有软件流水循环的比例.它说明 20%的软件流水循环存在改进的潜力.

Table 1 Cost of software pipelining

表 1 软件流水的开销

Benchmarks	L2 cache miss Ratio (SWP/No-SWP)	RSE reference Ratio (SWP/No-SWP)	Percentage of 1-stage SWP-ed loops (%)
CG	2.10	1.95	26
EP	1.58	1.44	25
FT	2.07	245.45	6
IS	1.08	1.00	50
MG	1.01	1.01	12
Average	1.31	41.81	19.83

图 4 和图 5 统计这 5 个 kernel benchmarks 中所有 102 个软件流水循环的段数(SC)和启动间距(II)的分布情况.图 4 表示只有 35%的软件流水循环是单重循环,即 65%的循环是多重循环.图 5 则表明 70%以上的循环的启动间距在 5 以上.虽然软件流水的装入/排空和初始化/清理代码开销与循环的执行次数有很大的关系,但多重循环以及启动间距这些静态信息也可以供我们做一些大致的估计.

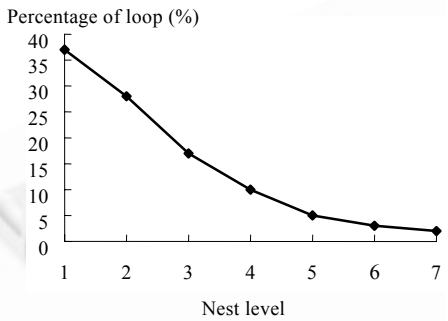

Fig.4 Distribution of nest level

图 4 循环层数分布

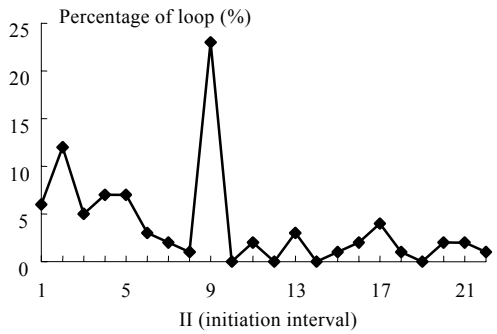

Fig.5 Distribution of II

图 5 启动间距分布

3.2 DBCT算法

我们在 ORC 中实现了第 2 节提到的软件流水决策框架,在此我们只测试了决策 3,即 DBCT 算法的性能.表 2 中的第 1 列是测试用例,其中前 5 个来自 NAS kernel benchmarks,后 4 个来自 SPECfp2000.第 2 列表示应用 DBCT 算法前后的性能加速比,即 TDBCT/T×100%.第 3 列表示应用 DBCT 算法后放弃软件流水的循环个数.

Table 2 Performance of the DBCT algorithm

表 2 DBCT 算法的性能

Benchmarks	Speedup (%)	# of SWP-skipped loops
CG	9.49	9
EP	0.07	1
FT	–0.75	2
IS	16.46	5
MG	–2.85	6
171.swim	1.22	1
177.mesa	5.45	426
183.equake	20.02	9
200.siztrack	6.84	169

表中的数据表明,DBCT 算法的效果是相当明显的.首先,它没有导致明显的性能下降,最坏情况是只有 2.85%(MG)的性能损失;其次,它对某些程序的改善非常显著,IS 和 183.equake 的性能提高分别达到了 16%和 20%.

性能提高的原因一方面是排除了单段的软件流水,造成单段流水的主要原因是循环的体间相关严重,即 RecMII 较大.在单段流水中,*RecMII*>=*CPL*,DBCT 算法恰好能够排除这样的情况,表 1 的最后一列统计了单段流水所占的比例;而对于操作较多或者循环执行次数很少的循环,DBCT 算法都能提供正确的决策,对于那些非单

段的软件流水循环,如果体间相关限制严重,DBCT 算法能够作出很好的预测,选择正确的调度算法,避免软件流水对性能造成的负面影响.

4 结 论

关于软件流水的研究工作很多,但大都偏重于提高软件流水的有效性,比如研究高效的调度算法和寄存器分配算法等,而关于软件流水的开销,或者说是其副作用,则研究得很少.本文从编译器系统的观点出发,指出一个好的编译器应该合理地采用适当的优化技术,而不是专注于某一特定的优化方法.因此,本文提出了一种面向程序特性的软件流水开销模型,并在此模型下分析了软件流水在寄存器压力、装入/排空和初始化/清理代码以及调度等方面的开销.最后,本文还提出了一种基于相关性分析的软件流水开销测试算法(DBCT),用来为软件流水决策提供判断依据.实验结果表明,DBCT 算法有效地提高了编译器的性能.

References:

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

[2] Dehnert JC, Towle RA. Compiling for the Cydra 5. Journal of Supercomputing, 1993,7(1-2):181~228.

[3] 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.

[4] Rau BR, Lee M, Tirumalai P, Schlansker MS. Register allocation for software pipelined loops. In: Allen R, ed. Proc. of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1992. 283~299.

[5] Dehnert JC, Hsu PY, Bratt JP. Overlapped loop support in the Cydra 5. In: Hennessy J, ed. Proc. of the 3rd Int'l Conf. on Architectural Support for Programming Languages and System. New York: ACM Press, 1989. 26~38.

[6] 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.

[7] Intel Corporation. Intel Itanium™ Architecture Software Developer's Manual. Volume 1: Application Architecture. Intel Corporation, 2001.

[8] Intel Corporation. Intel Itanium™ Architecture Software Developer's Manual. Volume 2: System Architecture. Intel Corporation, 2001.