计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
44(8):1429~1438, 2007

基于超块的统一分簇与模调度

胡定磊　陈书明　刘春林
(国防科学技术大学计算机学院　长沙　410073)
(dinglei.hu@gmail.com)

Hyperblock-Based Unified Cluster Assignment and Modulo Scheduling

Hu Dinglei, Chen Shuming, and Liu Chunlin
(*School of Computer Science, National University of Defense Technology, Changsha* 410073)

Abstract In order to exploit instruction level parallelism (ILP), multiple functional units with multi-ports register file are often used in very long instruction word (VLIW) processor. As the number of functional units rises, the number of register file ports will grow accordingly. At some point, the multiplexing logic on register ports can come to dominate the processor's cycle time. A reasonable solution is to partition the register file into independent clusters. Although clustered architectures reduce register file ports per cluster without performance degradation, they present new challenges to compiler which must assign every operation and operand to a specific cluster and coordinate data movement between clusters to achieve fine ILP. In this paper, a scheduling algorithm for clustered VLIW architectures—hyperblock-based unified cluster assignment and modulo scheduling (HBUCAMS) is proposed. Compared with basic block, hyperblock can provide more larger schedule region for exploiting ILP. Furthermore, because loop bodies with control flow can be converted into hyperblocks, there are more opportunities to apply modulo scheduling. Instead of performing clustered assignment and modulo scheduling sequentially, HBUCAMS put them into a single phase. This unified approach is more effective than phase-ordered approach, since it allows optimizing the global code generation problem instead of searching for optimal solutions to each individual step. Experiments in YHFT-DSP/700 compiler show that the proposed algorithm can obtain more optimized result than the ITSS algorithm.

Key words VLIW; compiler; hyperblock; cluster assignment; modulo scheduling; ILP

摘　要 超长指令字处理器为了提高指令集并行(ILP)往往采用多个功能单元,从而需要多端口的寄存器文件提供支持.但是寄存器文件会随着端口的增多变得更复杂,频率难以提升,成为系统的瓶颈.分簇是解决这一问题的有效手段.分簇在不影响处理器ILP的前提下减少了每簇寄存器文件的端口数目,但对编译器提出了挑战,编译器必须将指令和操作数在簇间进行合理分配才能得到较好的指令级并行.针对分簇超长指令字结构提出了一种基于超块的统一分簇与模调度编译方法.使用超块技术可以增大调度范围以获得更好的ILP,并且可以处理含有控制流的循环体,增加了模调度的适用范围.超块中指令的分簇与模调度则是统一进行的,这将比分阶段进行有更好的优化效果,因为统一进行是从全局的角度寻求优化而非寻求各个阶段局部优化.在YHFT-DSP/700编译器中的实验结果表明,与ITSS算法相比,该算法可以达到较好的优化效果.

关键词 超长指令字;编译器;超块;分簇;模调度;指令级并行

中图法分类号 TP314

收稿日期:2006-04-21;**修回日期**:2007-02-13
基金项目:国家自然科学基金项目(60473079);国家教育部高等学校博士点基金项目(20059998026)

超长指令字(very long instruction word,VLIW)处理器为了保证较高的指令级并行(instruction level parallelism,ILP)往往采用了较多的功能单元.通常在VLIW处理器中每个功能单元有几个端口与寄存器文件相连(典型的,如两读一写端口),这样若采用统一的寄存器文件,多个功能部件就需要多端口的寄存器文件提供支持.但是端口越多,寄存器文件越复杂,频率提升也越困难.为了减少寄存器文件的端口数,并且还能对多个功能单元提供支持,一种可行的方法就是把寄存器文件分为若干组,每组寄存器文件与若干功能单元相连,这样每组寄存器文件的端口数大为减少.这种结构被称为分簇结构,而每组寄存器文件和与其相连的若干功能单元称为一簇(cluster),通常簇间还有通信总线相连用于交换数据.目前来看,分簇结构在VLIW数字信号处理器(DSP)中应用比较广泛,如TI TMS320C6x[1],ADI TigerSharc[2],HP/ST 's Lx[3]以及"银河飞腾"高性能数字信号处理器(YHFT-DSP/700)[4]等.

本文所关注的就是这种分簇的VLIW体系结构.分簇结构降低了处理器的复杂度,对于处理器提升频率有很大的好处,但是也对编译器提出新的要求,编译器必须要为每条指令及其操作数选择合适簇.为了获得较高的ILP,编译器应当充分利用资源将指令分配到各个簇中去,但这样做的结果有可能使得簇间的通信开销变大,进而可能会影响到性能.因此编译器必须充分考虑这些情况,实现指令及其操作数有效的簇间分配,才能将VLIW体系结构的高性能特性发挥出来.

程序代码一般可以分为线性代码和循环代码两类,对这两类代码的调度是有区别的.通常使用列表调度(list scheduling)方法对线性代码进行调度;虽然循环代码也可以使用列表调度来处理,但一般使用更高效的软件流水方法进行处理.模调度(modulo scheduling)算法作为一种软件流水算法[5],通过使得循环迭代执行可以获得较高的ILP,比较适合于有多个功能单元的VLIW体系结构,因而成为了VLIW编译器中最主要的优化手段之一.

在用列表调度处理线性代码时,使用超块(hyperblock)是扩大调度氛围、提高ILP是一种重要手段[6].但对于模调度而言,传统的模调度算法一般只能处理循环体可以形成基本块的循环,倘若循环体中含有控制流(更确切的说是含有可能使得循环提前退出的分支),则该循环体无法形成基本块,也就不能进行模调度,这无疑限制了模调度的应用范围.针对这种情况,Lavery提出了一种基于超块的模调度方法,可以有效地解决含控制流的循环体的模调度问题[7].但Lavery提出的模调度算法是针对一般的通用RISC体系结构(非分簇)的,并不适合于分簇VLIW体系结构.

本文在Lavery提出的基于超块的模调度方法的基础上,针对分簇VLIW体系结构,提出了一种基于超块的统一分簇与模调度编译方法.使用超块技术可以增大调度范围以获得更好的ILP,并且可以处理含有控制流的循环体,增加了模调度的适用范围.超块中指令的分簇与调度则是统一进行的,这将比分阶段进行有更好的优化效果,因为统一进行是从全局的角度寻求优化而非寻求各个阶段局部优化.

1 相关工作

针对分簇结构的指令分配与调度算法的研究最早是由Ellis在编译器原型Bulldog中提出的[8].该算法分为两个阶段:先将每条指令分配到合适的簇中,然后依据簇分配的结果使用List调度算法对指令进行调度.在簇分配中使用了BUG(bottom-up greedy)算法,而在调度时根据需要插入簇间通信的指令.在Multiflow编译器中,Lowney等人研究了BUG算法在处理高度并行代码的不足,对其进行了改进[9].

Desoli提出了一个两阶段偏向部件分簇算法PCC(partial component clustering)[10].与PCC算法的结构相类似,Lapinskii等人也提出了两阶段簇间指令分派算法[11].第1阶段计算每条指令在每个簇上的分配代价,该代价考虑了指令分配到该簇上后,簇以及簇间通信总线的资源负载情况,然后依据每条指令分配代价的不同,建立初始的分配;第1阶段所花费的编译时间较少,如果需要进一步提高优化效果,则可以在第2阶段花费相对多的编译时间,通过反复的边界扰动分析进一步优化分配.

Jang等人提出了簇间寄存器分派的算法[12]:簇间寄存器分配时,首先建立寄存器构成图(register component graph),边和节点赋予一定的权重,实施改进的最小费用割集算法,将符号寄存器分配到不

同的簇仔,使得各簇的负载尽量平衡.

上述的这些研究有两个共同点:一是指令的簇间分配与调度是分阶段进行的,都会受到阶段转换问题(phase-ordering problem)的影响,由于只有当调度时才能准确知道资源的使用情况,调度之前对簇以及簇间通信总线的资源负载情况的估计只能是近似的,优化结果往往难以达到最优;二是这些算法的处理对象都是非循环代码,没有对于循环代码的特殊优化.

Leupers 提出了一个簇间指令分配的算法[13],为了解决分配与调度的阶段转换问题,该算法首选将指令随机的在各个簇间进行分配,然后使用模拟退火算法对分配进行改进,再进行调度,并将调度结果反馈给分配算法用以指导其进行更好的分配,如此反复直到达到最优. 但该算法因采用模拟退火算法并且分配与调度要进行反复迭代,会占用很大的编译时间.

与上述算法中先簇间分配而后调度不同,E Özer 等人提出了一种统一分配和调度的算法 UAS,即簇间分配和指令调度是同时进行的[14]. Nagpal 对这个算法进行了进一步的延伸,提出一种 ITSS 算法[15]. 这两种算法的思想与我们在本文中提出的基于超块的统一分簇与模调度的思想类似,但是主要有两点不同:UAS 和 ITSS 关注的是处理非循环代码的列表调度,而本文所关注的处理循环代码的模调度;UAS 和 ITSS 没有考虑寄存器簇间分配的问题,假设寄存器簇间分配在分簇调度之前已经完成,而我们提出的算法则是在进行统一的簇间分配与调度的同时确定寄存器的簇属性.

有关分簇结构下的模调度的研究也有一些,Fernandes 等人提出了一个在软件流水中同时进行分簇与调度的方法[16],但是针对的是一种特殊的寄存器文件组织方式,每个簇都有一组局部队列且每个通信通道都有一个队列文件.

Nystrom 等人提出了一个分簇与模调度的算法,它的簇间分配和模调度是分阶段进行的,如果任何一个阶段失败,则增加迭代间隔、重启算法[17]. 该算法主要考虑了两个因素:循环传递路径的影响和簇的过度使用的负面影响. 但它假设簇间有足够多的快速通道,认为通信问题造成的影响很小,当簇间的通信通道较少或有较大延迟时该算法的性能势必降低.

Akturan 等人根据嵌入式应用的特点,针对分簇 VLIW 体系结构提出了一种软件流水框架 CALiBeR,探讨了不同的优化方案:性能优化,代码大小优化,减小寄存器压力等. 但 CALiBeR 仍然是一个 phase-ordering 算法[18].

上述分簇结构下的模调度研究针对的都是循环体可以形成基本块的循环,适用性必然受到限制.

2 基于超块的统一分簇与模调度

2.1 分簇 VLIW 体系结构

本文给出的编译方法所面向的分簇 VLIW 体系结构如图 1 所示. 它由两个相似的簇组成,每个簇中都有几个不同的功能单元和本地的寄存器文件.在簇之间有交叉通路,使得一个簇中的功能单元可以读取另一个簇中的寄存器数据,但不能用于向另一个簇的寄存器文件写数据. TI C6x,YHFT-DSP/700 等就是采用的这种体系结构.

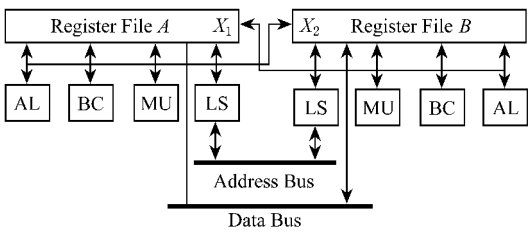

Fig. 1 Clustered VLIW architecture.

图 1 分簇 VLIW 体系结构

2.2 算法的总体框架

结合 IMPACT 编译平台的特点和 Lavery 提出超块模调度方法,我们设计了如图 2 所示的基于超块的统一分簇与模调度算法 HBUCAMS 的总体框架. 其中,HBUCAMS 的输入是与机器相关的中间代码 Mcode,而其输出则是经过软件流水的 Mcode,其中会用到目标体系结构的机器描述 MDES 和资源管理模块(相关内容见文献[19-20]). 下面我们详细阐述算法框架中每个阶段的具体内容.

2.3 选择合适的循环并形成超块

选择可以进行模调度的循环其标准有两个:一是循环的执行次数必须大于一个阈值. 由于在软件流水时有填充和排空阶段,只有循环执行次数较大,足以抵消填充/排空的开销,这样进行模调度才有意义.

二是循环体可以形成超块. 超块就是一种可包

含控制路径的单入口、多出口的编译路径,它是通过对程序行为的考察,根据一定的启发式规则,选择部分基本块来构造的[21]. 然后通过 if-conversion 过程将条件分支转换为谓词定义指令,将依赖于分支结果的指令转换为谓词执行指令. 对于循环而言,通常占用了程序的大部分执行时间,因而也是重点优化的对象. 但有时循环的内核未必是一个基本块,可能含有控制流,若形成超块可以取得比对各个基本块分别优化更好的优化的效果. 而且,由于超块将不经常执醒的基本块剔除,减少了这些块的资源占用和控制流相关限制,使得超块内有可能取得更高的 ILP. 由于在形成超块的过程中引入了谓词,可以继续应用文献[22]中方法对谓词进行分析与优化.

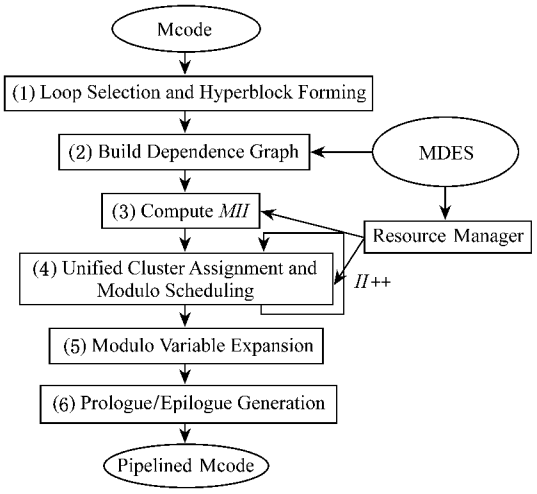

Fig. 2 Framework of HBUCAMS algorithm.

图 2 HBUCAMS 编译方法的总体框架

2.4 构造依赖图

与处理非循环代码不同,循环体超块的依赖分析除了要分析超块内的依赖关系,更主要的是分析循环体迭代间的依赖关系:数据依赖关系和控制依赖关系,确定其种类和依赖长度. 图 3 给出一个循环体的依赖图构造示例,其中图 3(a)是源代码,图 3(a)中的 for 循环经转换变成图 3(b)中的机器相关中间代码 Mcode,图 3(c)中则是该循环体的依赖图(图中的节点代表在图 3(b)中的相应指令). 在依赖图中,每条边上均有两个数字,第 1 个数字代表相关节点的延迟;第 2 个数字代表相关节点的距离,即两个节点发生相关所需的循环迭代次数. 若相关节点间的距离为 0,则是循环内相关;若距离大于 0,则是跨越循环的相关依赖.

```
int dotp (short a[],short b[])
{
  int sum=0,i;
  for (i=0;i<100;i++)
    sum+=a[i]*b[i];
  return(sum);
}
```

(a)

```
op1: ra=load(*pa++); load ai from mem
op2: rb=load(*pb++); load bi from mem
op3: rc=mul(ra,rb); ai*bi
op4: rm=add(rm,rc); sum+=(ai*bi)
op5: pi=sub(pi,1); dec loop counter
op6: b L1 if pi; branch to loop
```

(b)

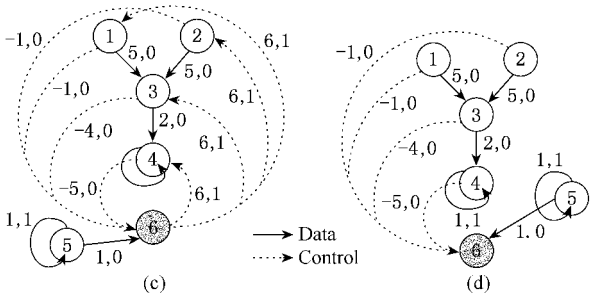

Fig. 3 Dependence graph for example loop. (a) Source code; (b) Mcode; (c) Dependence graph; and (d) After speculation.

图 3 循环体的依赖图.(a) 源代码;(b) 机器代码;(c) 依赖图;(d) 允许前瞻执行后依赖图

2.5 计算最小迭代间隔 *MII*

迭代间隔 II 是衡量模调度的重要指标,它的下限是最小迭代间隔(minimum initiation interval, MII),由两个因素决定:由于资源限制所造成的迭代间隔下限 *ResMII* 以及由于迭代间依赖关系限制所造成的迭代间隔下限 *RecMII*,*MII* 就取两者的最大值. 计算公式如下,其中依赖回路主要表现为依赖图(dependence graph, DG)上的回路.

$$MII = \mathrm{Max}(resMII, recMII). \tag{1}$$

$$recMII = \max_{\forall cycle \in DG} \left\lceil \frac{\text{sum of delays in cycle}}{\text{sum of distances in cycle}} \right\rceil. \tag{2}$$

$$resMII = \max_{\forall FU} \left\lceil \frac{\#\text{ of required FUs}}{\#\text{ of hardware FUs}} \right\rceil. \tag{3}$$

2.5.1 通过前瞻执行降低 *RecMII*

在模调度算法中,*RecMII* 是其包含的所有依赖回路的 *RecMII* 的最大值. 以图 3(c)为例,其中共有 10 条回路:4→4,5→5,6→4→6,6→3→6,6→2→6,6→1→6,6→2→3→6,6→2→3→4→6,6→1→3→6,

6→1→3→4→6,其回路“延迟之和/距离之和”中最大值为 8,即有 $RecMII=8$.

由于控制依赖的存在,$RecMII$ 难以降低,会使得模调度难以取得较高的 ILP. 所以打破控制依赖是降低 $RecMII$ 的主要途径. 消除控制依赖关系主要是通过前瞻执行实现的. 一条指令提前到分支指令 B 之前执行,必须满足 3 个条件:

1) 该指令不能重写一个在分支 B 的出口处活跃的寄存器,即该寄存器在分支目标中会被用到;

2) 该指令不会产生导致程序中止的异常;

3) 该指令不能往存储器写入数据.

从硬件对前瞻执行机制的支持来说,有 3 种体系结构模型:Restricted,General,Boosting[23],其中 Restricted 模型采用的方法是禁止异常的发生,即对于可能产生异常的指令不让编译器进行前瞻调度,遵循 ABC 三种限制;General 模型通过将可能产生异常的指令在前瞻时替换为同样功能的非陷入(non-trapping)指令,将限制条件减少为 AC;而 Boosting 模型则在硬件上设置了足够的缓冲区来保存前瞻指令执行的结果,直到分支条件确定,无 ABC 三种限制.

对于 VLIW DSP 而言,因为所有计算指令都是定点的,并且在硬件上没有对访存地址的越界检查,对越界地址的访问只会返回随机数,不会中断程序执行. 因此可以认为 VLIW DSP 的所有指令都是非陷入的,在体系结构上符合 General 模型,遵循 A 和 C 的限制. 我们用寄存器重命名和禁止 store 指令的前瞻来满足 A 和 C 的约束.

以图 3 为例,图 3(c)中的依赖图在允许前瞻执行的情况下,去掉与分支指令相关的控制依赖,得到图 3(d). 在图 3(d)中,依赖回路只剩下了两条:4→4,1→1,而 $RecMII$ 也降到了 1.

2.5.2 通过谓词分析降低 $ResMII$

通过资源管理器(resource manager)为每条指令的资源使用进行建模,可以分析得出 $ResMII$. 在分簇体系结构下,通常指令可以在多个簇的功能单元执行,因此计算 $ResMII$ 时需要考虑到指令的多个备选. 在计算 $ResMII$ 时还应当考虑谓词的影响. 在循环体含有控制流的情况下,经过 if-conversion 过程,超块中会含有谓词. 如文献[22]中所述,占用同一资源的多条指令,若它们的谓词不相交(即不同时为真),则在计算 $ResMII$ 时,只算做占用该资源的一条指令. 通过谓词分析降低了 $ResMII$.

2.6 统一的分簇与模调度

统一的分簇与模调度的处理对象是前面形成的循环体超块,目的是以尽可能小的迭代间隔实现循环体的软件流水. 因为在第 2.5 节中已经计算出最小迭代间隔 MII,故在此基础上由小到大依次尝试对循环体超块进行统一分簇与模调度,直到成功. 这里为算法设置了两个限制条件:一是用超块的列表调度长度作为尝试迭代间隔的上限 $MaxII$,如果迭代间隔大于或等于 $MaxII$,模调度的性能已经低于列表调度,没有必要进行下去了;二是设定的一个阈值 $Budget=BudgetRatio\times NumberOfOperations$,作为尝试以迭代间隔 II 对循环体超块进行统一分簇与模调度时的最大指令调度数,超过此数则表示本次尝试调度失败,然后放宽迭代间隔,进入下一次尝试.

在给定迭代间隔和调度阈值 Budget 的情况下,统一的分簇与模调度将采取如图 4 所示的步骤实现调度工作. 下面对该算法的一些步骤进行解释说明:

Step1: Compute scheduling priorities of all instructions in hyperblock, and put all instructions into set $S:=\{OP_1,\cdots,OP_m\}$;
Step2: Determine if $S\neq\varnothing$ (emptyset) and Budget>0; If not, goto Step8;
Step3: Select instruction $OP_i\in S$ with highest priority; According to scheduled predecessors of OP_i; Compute its earliest start time Estart; Set initial scheduling time of OP_i, $cycle:=Estart$;
Step4: Assign instructions and operands to clusters according to function cluster_assign(); If OP_i isassigned to cluster C, find all function units supporting OP_i incluster C and put them into set $\Omega:=\{f_1,\cdots,f_n\}$, let $\Omega':=\Omega$;
Step5: Randomly Select $f_j\in\Omega'$, determine if OP_i can be scheduled in f_i at current $cycle$ time; If yes, goto Step7; if not, $\Omega':=\Omega'\setminus\{f_j\}$;
Step6: If $\Omega'==\varnothing$ and $cycle<Estart+II$, let $\Omega':=\Omega$, cycle++; Goto Step5;
Step7: Assign OP_i to f_j; Insert necessary inter-cluster copy instructions, and replace operands not belonging to cluster C in OP_i by their copies in cluster C; Schedule OP_i at current $cycle$ time; Let $S:=S\setminus\{OP_i\}$, $Budget--$, goto Step2;
Step8: if S==$\varnothing$, Unified cluster assignment and modulo scheduling for the hyperblock is successful; If $Budget==0$, Initiation interval should be relaxed to try to schedule;

Fig. 4 Main steps of unified cluster assignmentand modulo scheduling.

图 4 统一的分簇与模调度的主要步骤

2.6.1 调度优先级的计算

Lavery 所提出的模调度算法[7]在为指令计算调度优先级时是高度(height)优先的,这对于通用微处理器而言是合适的:通用处理器的计算单元相对较少而寄存器相对较多;但对于 VLIW DSP 而言

是不合适的:它的寄存器文件相对较少. 模调度是将多个循环重叠执行,目的是使得模调度核(kernel)的调度长度(即迭代间隔 II)尽可能小,寄存器压力较大. 因此这里模调度指令优先级的计算,我们从 3 个方面考虑:

1) 垂直松弛度($Slack$)限制. 优先调度松弛度小的指令,其中 $Slack = ALAP - ASAP$,这里 $ALAP$(ASAP)是指令的最晚(最早)调度时间. 松弛度反映了变量在循环中的活跃周期,优先调度松弛度小的指令,可以减轻模调度中的寄存器压力[24],同时也避免了以高度或深度优先的情况下,由于调度的过度'贪婪'造成的处理器资源分配不均匀,进而使得迭代间隔 II 增大.

2) 平行自由度限制. 能执行节点代表的指令的功能单元数($Func\_num$)越多,则指令的调度优先权越低.

3) 关键节点优先. 节点在依赖图中的后继节点($Succ$)越多,则它的调度时间和资源占用对后继节点的影响越大.

综合考虑上述因素得到下面式(4)所示的指令调度优先权. 其中 $n \in DAG$,A,B,C 分别为微调系数. 这里对于这 3 个因素是依次考虑的,即首先考虑垂直松弛度的限制,在松弛度相等的情况下,再考虑自由的限制,最后考虑关键节点的限制,因此这里设 $A = 1.0$,$B = 0.1$,$C = 0.01$,每个系数相差一个量级.

$$Sched\_priority(n) = -Slack(n) \times A - Func\_num(n) \times B + Succ(n) \times C. \quad (4)$$

2.6.2 簇及功能单元分配优先级的计算

统一的分簇与模调度是在对指令进行调度时,同时为其分配合适的簇、功能单元,并确定指令的操作数属于哪一簇(簇属性).

Nagpal 等人提出一种计算簇的优先级的方法 ITSS[15],公式如下,即通过计算将指令 i 分配到簇 c 中所引起的通信开销来确定簇分配的优先级:

$$comm\_cost(i,c) = A' \times current\_comm + B' \times future\_comm + C' \times explicit\_mv, \quad (5)$$

这里,$current\_comm$ 是当前的通信开销(即指令 i 中需要从簇间交叉通路读取的非 c 簇操作数的数量);$future\_comm$ 是将指令 i 放到 c 簇后在指令 i 的后继节点中引起的通信开销;$explicit\_mv$ 是由于簇间交叉通路饱和而必须借助显式的 MV 指令将另一个簇中的操作数复制到 c 簇中所引起的开销(通常簇间拷贝指令插入到当前拍之前的合适位置).

在 ITSS 算法中,认为在式(5)中取 $A' = 0.5$,$B' = 0.75$,$C' = 1.0$ 时效果较好.

ITSS 方法是对 UAS 方法[14]的一个扩展. 与 UAS 方法相比,ITSS 方法不仅考虑当前的通信开销情况,而且考虑了对于后继节点的影响,因而全局性更好. 但二者都是在 List 调度的同时进行簇分配,面向的是非循环代码;并且二者都是部分算法,只考虑了指令的簇分配问题,而没有考虑操作数的簇分配问题,它们都假设在指令调度之前,寄存器分配已经完成(即操作数的簇属性已经确定),然后根据操作数的簇属性计算通信开销. 因此寄存器分配的情况对于 ITSS 的性能的有很大影响.

对非循环代码,在列表调度中处理分簇时,相较直接从交叉通路读取其他簇的操作数而言,在当前调度的前面插入拷贝指令既不会推迟当前指令的调度,还可以为在该簇中调度的其他后继指令提供操作数替换的可能. 但在模调度中,显式的簇间拷贝指令的插入会占用核的资源,并有可能引起 $RecII$ 或 $ResII$ 的增大,因此需要将式(5)中的惩罚值 C' 增大(经实验,$C' = 1.25$ 较为合适).

统一分簇与调度的基本思想是根据资源的实时负载来分配资源,而操作数与指令也是紧密相连的,因此对指令及其操作数进行统一的分配才能达到最佳的分簇效果. VLIW DSP 中的指令一般规定目的操作数与指令的功能单元一致,对指令的功能单元进行簇分配等同于对该指令的目的操作数进行簇分配;而源操作数中的一个可以通过簇间的交叉通路从另外一个簇读取. 我们将出现在循环体超块中的所有操作数集合记为 ALL,而由循环体超块中的指令进行定义的操作数集合记为 DEF,则 $ALL \setminus DEF$ 则是在循环体超块外部定义的操作数. 于是在按照调度优先级选取一条指令后,可以按照如下算法(图 5)进行分簇:

```
function cluster_assign (instruction inst)
{
  if (inst.cluster == null && inst.dest.cluster != null)
    inst.cluster = inst.dest.cluster;
  else if (inst.cluster == null && inst.dest.cluster == null) {
    inst.dest.cluster = min_comm_cost_dest (inst.dest);
    inst.cluster = inst.dest.cluster;
  }
  for (every source operand src∈ ALL \ DEF of inst)
    inst.src.cluster = min_comm_cost_src (inst.src);
}
```

Fig. 5 Cluster assignment algorithm.

图 5 分簇算法

此处的两个函数 $min\_comm\_cost\_dest()$ 和 $min\_comm\_cost\_src()$ 分别返回使得该操作数的通信开销最小的簇. 操作数的通信开销计算公式如下:

$$comm\_cost(dest, c) = B' \times future\_comm + D' \times spill\_cost. \quad (6)$$

$$comm\_cost(src, c) = A' \times current\_comm + B' \times future\_comm + C' \times explicit\_mv + D' \times spill\_cost. \quad (7)$$

$$\text{where}, spill\_cost = \begin{cases} 0, \text{ if } \#\text{of } \{oprd \mid oprd \in live\_in(inst) \,\&\&\, oprd.cluster == c\} < \#\text{of local regsters cluster } c, \\ 2, \text{ therwise.} \end{cases}$$

为操作数分配簇时,要考虑到若在此时该簇中的活跃变量已经达到局部寄存器的容量,再往该簇中分配操作数就会引起溢出代码,因此需要引入惩罚值 $spill\_cost$ 加以标示(经实验,$D' = 1.0$ 较为合适).

在确定了指令的分簇之后,从簇中支持该指令的空闲功能单元中随机选取一个作为该指令的功能单元即可.

当需要在当前指令前面插入拷贝指令时,先要查看当前 C 簇中是否已有非 C 簇操作数的拷贝(这通常是由其他指令引入的拷贝指令造成的),若有则无需再插入拷贝指令.

在当前拍调度指令 OP 主要是改写模调度的资源保留表,加入该指令的资源占用情况. 在当前指令前面插入拷贝指令时,也要采取同样动作. 由于谓词的存在,可以将处理器资源进行分类:必须使用的(must-use)和可能使用的(may-use). 在考虑谓词的影响下,调度器在为指令分配资源时,对于 may-use 资源可以将其同时分配给多条指令,只要这些指令的谓词是不相交的(即不同时为真). 这样做可以更充分地利用资源,降低了对调度的资源限制[22].

2.7 模变量扩展 MVE

模调度使得循环可以重叠执行,但也会引起循环变量生命周期重叠的问题,典型情况是当循环变量 v 的生命周期 $L(v) > \mathrm{II}$(迭代间隔)时,则对循环变量 v 的定义在被引用之前就有可能被后续的循环重新定义,造成语义错误. 这实际上是由于在进行模调度时没有考虑循环变量间的反相关(anti-dependence)关系. 解决这个问题用到了 MVE(modulo variables expansion)技术,即使用循环展开和寄存器重命名的方法来消除循环变量间的反相关关系. 图 6 给出了 MVE 运用的一个例子. 在 MVE 中由于使用了寄存器重命名,因而引入的新的寄存器需要为新引入的寄存器确定其簇属性. 既然新寄存器是由原寄存器重命名而来的,其簇属性与原寄存器保持一致.

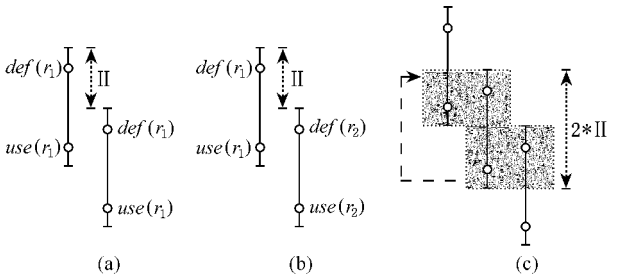

Fig. 6 Modulo variables expansion. (a) Without MVE; (b) With MVE; and (c) Modulo schedule after MVE.

图 6 模变量扩展. (a) 未使用 MVE;(b) 使用 MVE;(c) 使用 MVE 后的模调度

2.8 模调度的填充与排空的生成

模调度经过上述步骤的处理后,为其生成软件流水所需的填充和排空(prologue/epilogue)阶段,需要处理 3 个方面的问题:

1) 为超块循环体中的多个退出循环的分支指令分别生成排空,这里需要考虑两种调度情况,前瞻执行(提前到分支指令之前)和延迟执行(推迟到分支指令之后). 延迟执行的指令在语义上是应当在退出循环的分支指令前执行的,因此要将其拷贝到该退出分支指令的排空里. 在经过 MVE 之后,前瞻执行的指令不会影响从该循环退出指令的 live-out 活跃变量,可以不予考虑;

2) 在 MVE 中进行了寄存器重命名,在退出指令的排空中,需要将 live-out 活跃变量变回原来的名字,供后续指令使用,即插入一些拷贝指令;

3) 从模调度的核(kernel)中拷贝相应的段(stage)组成软件流水的填充,并将填充中的循环退出指令的排空映射到已生成的排空上.

填充与排空生成阶段虽然会引入补偿代码,但是没有引入新的寄存器,不会对既有的分簇造成影响.

3 实验结果

我们在 YHFT-DSP/700 编译器中对本文中提

出的基于超块的统一分簇与模调度(HBUCAMS)方法进行了验证.YHFT-DSP/700 编译器是我们在可重定向编译基础设施 IMPACT[25] 的基础上自主开发的一款高性能编译器.YHFT-DSP/700 编译器使用 HMDES 机器描述语言定义目标体系结构和指令集[20],编译器的各模块通过查询接口从机器描述模块获取目标机的相关信息,为验证体系结构的改动提供了方便.实验所用的程序来自于 DSPstone[26],主要是一些典型的 DSP 核心程序,以及 MediaBench 中的一些实际媒体处理应用程序[27].

在面向图 1 中的分簇 VLIW 结构的 DSP 编译器中应用本文提出的基于超块的分簇与模调度(HBUCAMS)方法,我们得到图 7 中的编译结果.这里 HBUCALS 指的是应用本文的统一分簇与调度的方法于列表调度、对所有循环或非循环代码进行调度所得到的结果,而 ITSS 则是在文献[15]中提出的方法.HBUCALS 是对 ITSS 算法的改进,而 HBUCAMS 则对 HBUCALS 算法进行了改进.

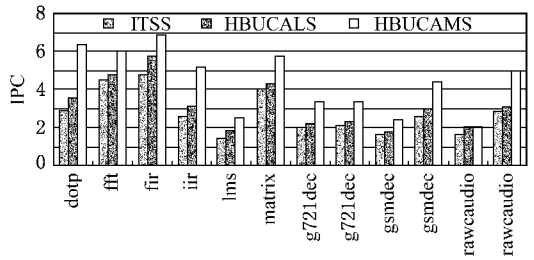

Fig. 7 Experiment results of HBUCAMS algorithm.

图 7 HBUCAMS 的实验结果

由于 ITSS 方法没有考虑寄存器的分簇问题,即它是在寄存器已经分配的前提下寻找一种优化的分簇方案,其性能受寄存器分配的影响很大.而着色图法寄存器分配时考虑的是干涉图,而不考虑实时的资源负载情况,在这种情况下应用 ITSS 方法,则必须加入许多簇间拷贝指令进行簇间的通信,降低了性能.HBUCALS 方法在分簇的同时考虑寄存器的分簇问题,指导寄存器分配,比之 ITSS 方法性能更高.HBUCAMS 结合了超块、模调度、指令与数据的统一分簇等特点,较之 ITSS 和 HBUCALS 更有大幅的性能提升.

为了进一步验证本文提出的统一分簇与模调度算法,我们针对不同的寄存器文件配置,对测试程序进行了测试.机器的配置如表 1 所示,这里用于测试的 A,B,C 三种机器配置都是分为对称的两簇(结构示意图见图 1),所不同的是每个簇对应的局部寄存器的数目.

Table 1 Different Configuration of Target Machines

表 1 目标机器配置

Instruction Latency of				Register Number per Cluster		
AL	MU	LS	BC	Processor A	Processor B	Processor C
1	2	5	6	16	32	64

图 8 中对于 ITSS 和 HBUCAMS 两种算法在不同机器配置的情况下的性能进行了测试比较,度量标准是 HBUCAMS 相对于 ITSS 的加速比($=IPC\_of\_HBUCAMS/IPC\_of\_ITSS$).我们将 ITSS 算法也在模调度中进行了实现,在模调度中对这两种算法进行比较.如前所述,HBUCAMS 算法将资源分簇与模调度的工作统一起来,可以实时地根据指令调度和资源使用的情况进行资源分簇和调度,相较不考虑操作数分簇的 ITSS 算法,应当可以取得更好的调度效果.实验结果也证明了这一点.从图 8(a)中可以看出,在 3 种不同的机器配置情况下,绝大部分测试程序由 HBUCAMS 获得的 IPC 均明显优于 ITSS 算法($Speedup>1$),近似理想情况下(64regs/cluster),由于寄存器相对较多,ITSS 算法的性能接近于 HBUCAMS 算法.而当寄存器文件相对较少时,HBUCAMS 由于统一考虑了操作数的分簇问题,故相比较 ITSS 算法,有效地减少了簇间通

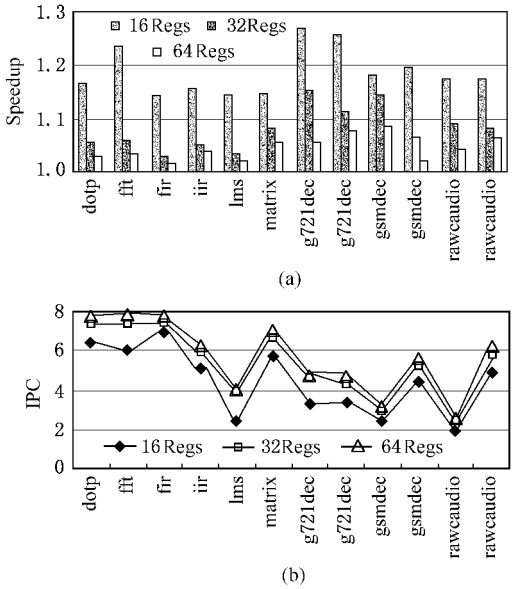

Fig. 8 Comparisons of experiment results. (a) Speedup of HBUCAMS to ITSS and (b) IPC of HBUCAMS under different configuration.

图 8 测试结果及比较.(a) HBUCAMS 相对于 ITSS 的加速比;(b) HBUCAMS 在不同配置下的 IPC

信和代码溢出,从而性能明显占优,这在 A 机器配置情况中尤其显著.

另外如图 8(b)所示,当簇中寄存器较少时(16regs/cluster),受溢出代码的影响,HBUCAMS 算法性能未能完全发挥;当簇中寄存器增多时,算法性能随之提高;但是如果寄存器相对较多,算法性能已接近理想性能,对寄存器的多少不再敏感,这一点可以从 64regs/cluster 与 32regs/cluster 相比性能只有少许提升看得出来.

4 小 结

本文针对分簇超长指令字体系结构,提出了一种基于超块的统一分簇与模调度方法,给出了整个方法的编译框架,并对其中的算法进行了详细讨论.该调度方法使用超块作为模调度的处理对象,扩大了优化范围;而统一的分簇与调度策略使得对机器资源的利用更加充分.在我们开发的分簇结构 VLIWDSP 编译器平台上对该调度方法进行了测试,实验结果表明了该方法的有效性.

参 考 文 献

[1] TMS320C6000 CPU and Instruction Set Reference Guide (Rev.F)[G]. Dallas, TX: texas Instruments Inc, 2000

[2] J Fridman, Z Greenfield. The tiger SHARC DSP architecture [J]. IEEE Micro, 2000, 20(1): 66-76

[3] P Faraboschi, G Brown, *et al*. Lx: A technology platform for customizable VLIW embedded processing [C]. In: Proc of the 27th Annual Int'l Symp on Computer Architecture. New York: ACM Press, 2000. 203-213

[4] Chen Shuming, Li Zhentao, *et al*. Research and development of high performance YHFT digital signal processor [J]. Journal of Computer Research and Development, 2006, 43(6): 993-1000 (in Chinese)
(陈书明, 李振涛, 等. 银河飞腾高性能数字信号处理器研究进展[J]. 计算机研究与发展, 2006, 43(6): 993-1000)

[5] B R Rau. Iterative modulo scheduling: An algorithm for software pipelining loops [C]. In: Proc of the 27th Annual Int'l Symp on Microarchitecture. New York: ACM Press, 1994. 63-74

[6] P Faraboschi, J A Fisher, *et al*. Instruction scheduling for instruction level parallel processors [J]. Proceedings of the IEEE, 2001, 89(11): 1638-1659

[7] D M Lavery. Modulo scheduling for control-intensive general-purpose programs: [Ph D dissertation] [D]. Urbana, IL: University of Illinois, 1997

[8] J R Ellis. Bulldog: A Compiler for VLSI Architectures [M]. Cambridge, MA: MIT Press, 1986

[9] P G Lowney, S M Freudenberger, *et al*. The multiflow trace scheduling compiler [J]. Journal of Supercomputer, 1993, 7(1-2): 51-142

[10] G Desoli. Instruction assigment for clustered VLIW DSP compilers: a new approach [R]. Hewlett-Packard Laboratories, Tech Rep: HPL-98-13, 1998

[11] V S Lapinskii, M F Jacome, *et al*. Cluster assignment for high-performance embedded VLIW processors [J]. ACM Trans on Design Automation of Electronic Systems, 2002, 7(3): 430-454

[12] S Jang, S Carr, *et al*. A code generation framework for VLIW architectures with partitioned register banks [C]. The 3rd Int'l Conf on Massively Parallel Computing Systems, Colorado Springs, CO, 1998

[13] R Leupers. Instruction scheduling for clustered VLIW DSPs [C]. In: Proc of the 2000 Int'l Conf on Parallel Architectures and Compilation Techniques. Los Alamitos, CA: IEEE Computer Society Press, 2000. 291-230

[14] E Özer, S Banerjia, *et al*. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In: Proc of the 31st Annual ACM/IEEE Int'l Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society Press, 1998. 308-315

[15] R Nagpal, Y N Srikant. Integrated temporal and spatial scheduling for extended operand clustered VLIW processors [C]. In: Proc of the 1st Conf on Computing Frontiers. New York: ACM Press, 2004. 457-470

[16] M M Fernandes, J Llosa, *et al*. Distributed modulo scheduling [C]. In: Proc of the the 5th Int'l Symp on High Performance Computer Architecture. Los Alamitos, CA: IEEE Computer Society Press, 1999. 130-134

[17] E Nystrom, A E Eichenberger. Effective cluster assignment for modulo scheduling [C]. In: Proc of the 31st Annual ACM/IEEE Int'l Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society Press, 1998. 103-114

[18] C Akturan, M F Jacome. CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors [C]. In: Proc of the 2001 IEEE/ACM Int'l Conf on Computer-Aided Design. Piscataway, NJ: IEEE Press, 2001. 112-118

[19] Hu Dinglei, Chen Shuming, *et al*. Design and implementation of clustered VLIW DSP compiler [J]. Mini-Micro Systems, 2006, 27(2): 348-353 (in Chinese)
(胡定磊, 陈书明, 等. 分簇结构超长指令字 DSP 编译器的设计与实现[J]. 小型微型计算机系统, 2006, 27(2): 348-353)

[20] J C Gyllenhaal, W W Hwu, *et al*. HMDES version 2.0 specification [R]. The IMPACT Research Group, Tech Rep: IMPACT-96-03, 1996

[21] S A Mahlke, D C Lin, *et al*. Effective compiler support for predicated execution using the hyperblock [C]. In: Proc of the 25th Annual Int'l Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society Press, 1992. 45-54

[22] Hu Dinglei, Chen Shuming, *et al*. Optimizing compiler based on complementary predicate [J]. Acta Electronica Sinica, 2006, 34(7): 1280-1286 (in Chinese)
(胡定磊, 陈书明, 等. 基于互补谓词的编译优化[J]. 电子学报, 2006, 34(7): 1280-1286)

[23] P P Chang, N Warter, *et al*. Three architectural models for compiler-controlled speculative execution [J]. IEEE Trans on Computers, 1995, 44(4): 481-494

[24] R A Huff. Lifetime-sensitive modulo scheduling [C]. In: Proc of the ACM SIGPLAN '93 Conf on Programming Language Design and Implementation. New York: ACM Press, 1993

[25] W W Hwu. The IMPACT Research Group [OL]. http://www.crhc.uiuc.edu/Impact/, 2006

[26] The Institute for Integrated Signal Processing Systems. DSPstone [OL]. http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html, 2006

[27] C Lee, M Potkonjak, *et al*. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems [C]. In: Proc of the 30th Annual ACM/IEEE Int'l Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society Press, 1997. 330-335

Hu Dinglei, born in 1976. Ph. D. in computer science and technology at the National University of Defense Technology (NUDT). His main research interests are VLIW and low-power compilation technologies.
胡定磊, 1976 年生, 博士, 主要研究方向为超长指令字编译、低功耗编译技术.

Chen Shuming, born in 1961. Currently professor and Ph. D. supervisor at NUDT. His main research interests include computer architecture, microprocessor design, and DSP techniques.
陈书明, 1961 年生, 教授, 博士生导师, 主要研究方向为计算机体系结构、微处理器设计、DSP 技术.

Liu Chunlin, born in 1963. Currently professor of NUDT. His main research interests include programming language design and compiler.
刘春林, 1963 年生, 教授, 主要研究方向为程序语言设计和编译技术.

Research Background

At present, very long instruction word (VLIW) architecture has been adopted popularly by most high-end digital signal processors (DSPs). VLIW machines tend to use static scheduling which allows the compiler to directly schedule machine resource usage, so that the performance is heavily dependent on the compiler arrangements to code. NUDT's DSP Design Group led by Prof. Chen Shuming has developed a high performance DSP (YHFT-DSP/700) with clustered VLIW architecture. A VLIW compiler based on retargetable compiler infrastructure IMPACT has been also developed. In order to resolve the compilation problem of operations and operands assignment in clusters, we propose a scheduling algorithm—Hyperblock-based unified cluster assignment and modulo scheduling (HBUCAMS). This unified approach is more effective than phase-ordered approach. This work is supported by the National High Technology Research and Development Program of China (2004AA1Z1040) and the National Research Foundation for the Doctoral Program of Higher Education of China (20059998026).