文章编号:1001 - 9081(2010)01 - 0274 - 03

一种基于寄存器压力的 VLIW DSP 分簇算法

雷一鸣[1,2],洪 一[3],徐 云[1,2],姜海涛[1,2]
(1. 国家高性能计算中心,合肥 230027; 2. 中国科学技术大学 计算机科学与技术学院,合肥 230027;
3. 中国电子科技公司 第 38 研究所,合肥 230027)
(lyming@mail.ustc.edu.cn)

摘 要:寄存器是程序运行时最宝贵的资源之一,软件流水在对 VLIW DSP 指令调度的同时,会显著增加寄存器的压力,从而导致寄存器溢出,软件流水中止。在以往的研究中,软件流水之前的指令分簇会更多地考虑指令并行性,往往会把寄存器的压力交给寄存器分配阶段,当物理寄存器不够分配时会造成寄存器溢出。通过考察指令运行时的寄存器压力情况对指令进行分簇,这样可根据各个簇的寄存器压力的动态信息减少寄存器的溢出,提高指令运行效率。

关键词:超长指令字;编译器;分簇;寄存器压力;软件流水;模变量扩展
中图分类号: TP314 **文献标志码**:A

Register based algorithm for VLIW DSP cluster assignment

LEI Yi-ming[1,2], HONG Yi[3], XU Yun[1,2], JIANG Hai-tao[1,2]
(1. *National High Performance Computing Center, Hefei Anhui* 230027, *China*;
2. *School of Computer Science and Technology, University of Science and Technology of China, Hefei Anhui* 230027, *China*;
3. *The* 38 *Research Institute, China Eletricity Company Technology, Hefei Anhui* 230027, *China*)

Abstract: Register is one of the most valuable resources. Software pipelining could bring about register pressure while scheduling instruction of VLIW DSP, which would cause register spill and software pipelining suspension. In the past research, instruction clustering pay more attension on instruction parallelism, and pass the register pressure to register allocator. In this paper, the authors clustered instruction through run-time inspection of the register command pressure. This is a dynamic method, according to all the pressure of each cluster's register, which can reduce register spill and improve the efficiency of the program.

Key words: Very Long Instruction Word (VLIW); compiler; clustering; register pressure; software pipelining; modulo variable extension

VLIW DSP 很多采用多簇的体系结构,每个簇上都有计算功能单元和寄存器文件堆。分簇就是把各条指令指定在某个簇上去执行。分簇能影响指令的并行性和各簇的资源使用,因此直接影响到指令的执行效率。软件流水一般发生在指令分簇之后,软件流水后的模变量扩展(Modulo Variable Extension,MVE)技术是用于消除软件流水模调度之后存在反依赖的寄存器,从而使 VLIW DSP 的指令并行度更高。如图 1 所示模变量扩展,由于第 1 次迭代的最后一条指令要使用第 1 次迭代开始时定义的值,因此第 2 次迭代不能早于第 1 次迭代的最后一条指令,从而使模调度的迭代间隔(Initiation Interval)II 增加到 2。若使用模变量扩展,把第 2 次迭代中的 r1 寄存器重命名为 r2,则可以使 II 降低到 1,提高了指令并行性,而此时也增加了一个虚拟寄存器,从而给寄存器分配带来压力。若此时物理寄存器不够分配,则需要插入溢出代码,从而破坏了软件流水。

最早提出分簇 VLIW 的编译分簇调度技术的是耶鲁大学的 J. ELLIS[1],他们提出的算法称为 BUG(Botton-Up Greedy)。BUG 算法依据指令之间的寄存器的依赖关系建立依赖图,再根据指令之间的延迟信息和关键路径信息,把处在同一关键路径上的指令努力放在同一簇内,从而减少了簇间传输指令。文献[2]中也提出一种分簇算法,该算法与 BUG 类似,也是首先建立各基本块的关键路径,但在决定指令簇时会更多考虑各簇的负载情况。上述算法都没有针对寄存器的压力采取特殊的措施。本文在分簇阶段采取一种启发式的方法来减少各簇的寄存器压力和通信开销。经过试验,该方法能针对机器模型结构,减少寄存器溢出,提高程序运行效率。

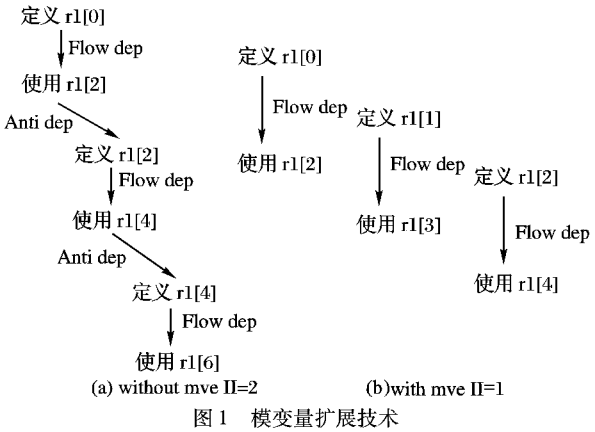

图 1 模变量扩展技术

1 编译器整体框架

本文的机器模型是多簇体系结构,各个簇上的资源数量相等。每个簇上有多个计算功能部件和寄存器文件堆,各个簇通过总线相连。在该结构的基础上开发了一款 BWDSP100

收稿日期:2009 - 07 - 16;**修回日期**:2009 - 08 - 27。

作者简介:雷一鸣(1984 -),男,湖南常德人,硕士研究生,主要研究方向:编译器技术、高性能计算; 洪一(1943 -),男,安徽铜陵人,教授,主要研究方向:雷达信号处理; 徐云(1960 -),男,安徽合肥人,副教授,博士,主要研究方向:网络计算、并行算法、生物信息学; 姜海涛(1988 -),男,安徽阜阳人,硕士研究生,主要研究方向:高性能计算、生物信息学。

编译器,该编译器是以可重定向编译基础设施 IMPACT[3] 为基础,自主开发的一款高性能 C 语言编译器。BWDSP100 编译器后端分为 6 大模块,分别为指令注释、指令分簇、软件流水[4]、寄存器分配[5]、指令调度[6]和汇编代码生成,如图 2 所示,其中实线部分为生成正确的目标代码所必须经历的阶段,虚线框是为了提高目的代码的运行效率而采取的优化手段。后端的输入是前端生成的与机器无关的 Lcode;指令注释把 Lcode 指令注释成与 BWDSP100 处理器相关的指令;指令分簇是把指令指定在特定的簇上去执行;软件流水对循环进行调度,提高循环部分执行效率;寄存器分配是把虚拟寄存器转换为物理寄存器;指令调度对非循环指令进行调度,提高非循环代码的执行效率;汇编代码生成是生成 BWDSP100 的目标汇编代码。

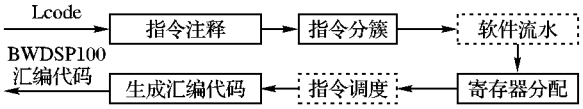

图 2　编译器框架

2　分簇算法

通常情况下,对指令的分簇是对函数的每个控制块的指令进行分簇。对函数体中的每个控制块建立数据依赖图,再对每条指令进行分簇。这样可以提高指令的并行度。而我们这样会存在如下问题。1)当前数据依赖图只有局部而不是全局的信息。比如,对于一条来说,可能有多条指令在其他的控制块中对它有流依赖。2)无法获得全部关键路径信息。某些靠后的指令块中的指令有可能比前面块的指令更应该先分簇。比如:图 3 中 CB1 中定义 r1 而后只有 CB2 中使用 r1 一次。CB3 中定义 r2,而后有两次使用 r2 的值,因此 r2 应该比 r1 先分簇,只有建立全局的依赖图才能有此信息。

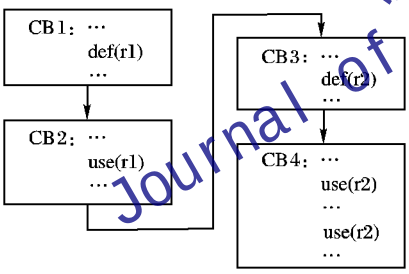

图 3　一个依赖的例子

本文提出一种全局的分簇算法,如图 4 所示。算法的输入是经过注释后与机器相关的 Mcode,输出是带分簇信息的 Mcode。

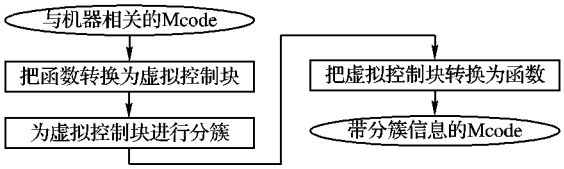

图 4　分簇算法整体图

2.1　把函数转换为虚拟控制块

转换函数为虚拟控制块主要是对函数中每条指令进行复制。首先建立一个虚拟控制块 GCB,再把函数中的所有指令依次拷贝到 GCB 中。BWDSP100 中大部分指令是需要使用寄存器的,但也有些指令是不需要使用寄存器的,比如:无条件跳转指令,函数调用指令等,这些指令不用指定簇的信息,因此就不用把它们复制到 GCB 中。在转换过程中还必须忽略控制流信息,把这种指令转换为一般指令即可。转换算法描述见算法 1。

算法 1　转换函数为虚拟块算法。

```
//oper represent each instruction
convert_func_to_virtual_cb(func)
 {build_virtual_cb(GCB);                          //建立 GCB
 for( each oper in func)
  {if(oper 是 br 指令)              //条件跳转指令作一个转换
     convert oper to a ordinary oper and copy to VCB;
  else if(oper 不是寄存器指令)
    continue;
  else
    copy_oper_to_CB(oper, VCB);                   //复制每一条指令
  }}
```

2.2　为虚拟控制块进行分簇

分簇算法是一种类似表调度的算法,与表调度不一样的是。调度过程中不是真正在某个时刻发射指令,而是在该时刻给指令分簇,因此分簇算法不必要检测资源冲突。算法首先是为指令块的指令建立数据依赖图,这里的指令块是已经转换为虚拟控制块的函数块;再计算每条指令的优先级;然后从 0 时刻开始对优先级从大到小的指令拟调度,拟调度并不是真正的在该当前时刻调度该指令,而是对该指令进行分簇。分簇算法描述见算法 2。

算法 2　分簇算法。

```
Cluster (control_block cb)
 {build_dependence_graph (cb);  //为指令块建立数据依赖图
  SM_calculate_priorities(cb);  //计算指令块中各拽令的优化级
  init_sm_queue(cb, sm_queue);                    //初使化调度队列
  for( issue_time = 0; issue_time + + )
  {                                  //从时间 0 开始拟调度每条指令
    while( 1 )
     {sched_op = NULL;
     for(scan_op = sm_queue -> first_op; scan_op;
     scan_op = scan_op -> next_queue_op)
     if (has_tested(scan_op))
       Continue;
     schd_op = scan_op;        //选取没有调度过的指令拟调度
     }
    if (sched_op == NULL)
      break;
    SM_cluster_oper(sched_op, issue_time);
    //拟调度指令即为每条分簇
    Mark_op_tested(sched_op);                     //标记指令已调度过
  }}
```

建立依赖图　函数 build_dependece_graph(cb)就是为当前的指令块建立依赖图。数据依赖图能反应指令的指令间的数据依赖关系,父节点必须先于子节点分簇。

计算节点优先级　指令优先级反应指令调度的先后顺序。根据以下规则来确定指令优先级:

规则 1　所有指令的最晚的最早调度时间与指令的最晚调度时间的差值越大,则该指令的优先级越高。

规则 2　若规则 1 中的值相等则后继节点越多的节点优先级越大。

规则 3　若规则 2 中的值也相等,则指令出现得越早优先级越大。

确定指定所在簇信息 BWDSP100 中除了 MOV 指令外,其他指令必须严格的保证其操作数在同一簇上,使用同一簇上资源。因此为指令分簇即为操作数分簇。确定指令的算法见算法 3。

算法 3　指令分簇算法。

```
SM _cluster_oper(sched_op, issue_time){
```

```
    determine_dest_core(sched_op);      //确定源操作数所在的簇
    determine_src_core(sched_op);     //确定目的操作数所在的簇
    update_reg_core(sched_op);          //更新寄存器的使用情况
    adjust_ sm_queue(sm_queue);                //调整调度队列
}
```

其中 determine_dest_core(sched_op)是为指令确定目的操作数所在簇,determine_src_core(sched_op)是为源操作数确定所在簇,update_reg_core(sched_op)是更新实时的寄存器使用情况。SM_cluster_oper()开始之前,启发式的定义 X,Y,Z,T 簇寄存器溢出阈值。在确定目的操作数所在簇之前,首先扫描目的操作数的所在簇是否已经确定,若已经确定,则把目的操作数指定在该簇上。若目的操作数没有确定,则分别计算每个簇的得分,见式(1)。

$$\text{Score\_cost}(\text{dest},c)=\begin{cases}0-\text{reg\_core\_count}(c), & \text{reg\_core\_count}(c)<\text{SHRD\_CORE}\\ \text{reg\_core\_count}(c), & \text{reg\_core\_count}(c)\geqslant\text{SHRD\_CORE}\end{cases}\tag{1}$$

其中 score_cost(dest,c)为目的操作数 dest 放在 c 簇上的得分,reg_core_count(c)为当前时刻 c 簇上的寄存器压力,SHRD_CORE 为定义的寄存器溢出的阈值。由于最终的寄存器使用情况要到模变量扩展以后才能得到,而分簇要先于模变量扩展,因此 SHRD_CORE 的值只能以一种启发式的方式给出。若当前 c 簇上可用寄存器的数量小于 SHRD_CORE 的值,则说明之前已经有指令使用 c 簇上的寄存器了,此时为了减少簇间传输指令,应该把当前指令放到寄存器使用越多的簇上,因此 reg_core_count(c)越大,score_cost(dest,c)的得分越小。若当前 c 簇上可用寄存器的数量不小于 SHRD_CORE,则说明模变量扩展带来的寄存器极有可能导致寄存器溢出,此时为了避免溢出,应该使用其他簇上的寄存器,因此 reg_core_count(c)越多,score_cost(dest,c)越大。最后根据每个簇的得分情况选择得分最小的簇给目的操作数。确定源操作数与确定目的操作数类似。但若发现源操作数与目的操作数不在同一簇上,则需要插入显式的 MOV 指令把源操作数的值从该簇转移到目的操作数所在的簇上。当源操作数和目的操作数的簇都确定以后,实时的更新每个簇上的寄存器使用情况,即更新 reg_core_count(c)的值,最后调整分簇队列,准备后续的指令分簇。

3 仿真结果

本文在 BWDSP100 编译器中对上述算法进行了验证。选取 DSP 典型应用 dspstone[7],用来测试编译器的效率。图 5 为在有分簇(SHRD_CORE 为 25)的情况下和没有分簇的情况下各典型算法主要部分所用的时间(单位 cycle)。

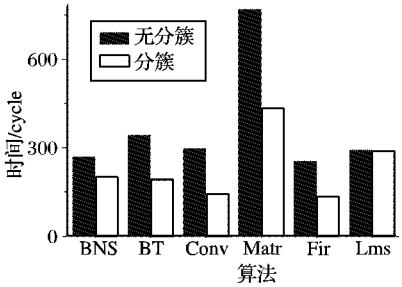

图 5 分簇与无分簇对比图

由于在没有分簇的情况下,所有指令都使用同一簇上的寄存器资源,使得有些需要很多寄存器的程序产生寄存器溢出,从而导致软件流水失败,影响程序的执行效率。Lms 中有分簇和无分簇的执行时间一样,是因为 Lms 算法的目标代码所用的寄存器数量较少,分簇以后的指令也是在同一个簇上使用的资源。

本文所提出的算法中,影响效率最关键的因子是 SHRD_CORE,因为如果 SHRD_CORE 设置过大,则分簇算法就等同于所以指令都使用同一簇上寄存器;如果设置过小,会使簇间传输指令增多,代码长度增大,影响效率。因此,为了进一步验证本文提出的算法,选取不同的 SHRD_CORE 因子进行测试。图 6 为 SHRD_CORE 不同值时,上述 6 个 DSP 典型应用程序执行的平均时间。

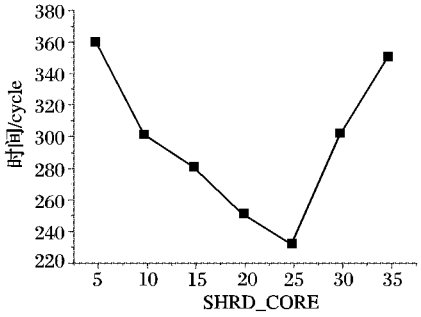

图 6 SHRD_CORE 曲线图

若 SHRD_CORE 的值很小,则在同一簇上使用了很少的寄存器后就开始使用其他簇上的寄存器了,这样会使簇间传输指令增多,增加了代码的长度,降低了指令的并行性。当 SHRD_CORE 进一步扩大时,簇间指令会减少,指令并行性会增大,但到一定程度时,会导致同一簇上的寄存器的使用增多,从而导致寄存器分配溢出。从图 6 中,当 SHRD_CORE 在 25 以内时,寄存器在同一簇上的使用需求量增加,使用同一簇上的寄存器能提高指令的运行效率。但当 SHRD_CORE 大于 25 以后,寄存器压力开始增大,开始变得容易溢出,当 SHRD_CORE 超过 35 以后相当于只使用了 1 个簇上的资源。

4 结语

本文针对 VILW DSP 分簇体系结构,提出了一种基于寄存器压力的分簇算法,给出了整个方法的原理和实现,并在 BWDSP100 编译器平台上对该方法进行了测试和评估。讨论了影响该算法的主要参数 SHRD_CORE,实验结果表明了当 SHRD_CORE 选取在一定范围内,能显著减少寄存器的溢出,提高指令的执行效率。

参考文献:

[1] DESOLI G. Instruction assignment for clustered VLIW DSP compilers: A new approach[EB/OL]. [2009 - 06 - 20]. http://www.hpl.hp.com/techreports/98/HPL-98-13.pdf.

[2] LAPINSKII V, JACOME M F, VECIANA G A. Cluster assignment for high performance embedded VLIW processors[J]. ACM Transactions on Design Automation of Electronic Systems, 2002, 7(3): 430 - 454.

[3] HWU W W. The IMPACT Research Group[EB/OL]. [2009 - 03 - 15]. http://impact.crhc.illinois.edu/.

[4] RAU B R. Iterative modulo scheduling: An algorithm for software pipelining loops[C]// Proceedings of the 27th International Symposium on Microarchitecture. New York: ACM, 1994: 63 - 74.

[5] CHOW F. Register allocation by priority-based coloring[J]. ACM SIGPLAN Notices, 1984, 19(6): 222 - 232.

[6] PHILIP B. Gibbons Efficient instruction scheduling for a pipelined architecture[J]. ACM SIGPLAN Notices, 1986, 21(7): 11 - 16.

[7] The Institute for Integrated Signal Processing Systems . DSPstone [EB/OL]. [2009 - 03 - 20]. http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone htmt.