第46卷第2期 中国科学技术大学学报 Vol. 46, No. 2
2016年2月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Feb. 2016

文章编号:0253-2778(2016)02-0104-09

分簇结构模调度框架研究

王向前[1,2],郑启龙[3],洪 一[1]

(1. 中国电子科技集团公司第三十八研究所,安徽合肥 230088; 2. 合肥工业大学计算机与信息学院,安徽合肥 230009;
3. 中国科学技术大学计算机科学与技术学院,安徽合肥 230027)

摘要:构建了面向分簇体系结构的模调度编译框架.介绍了分簇结构和支持向量化执行的体系结构的机器资源描述方法,研究了模调度和循环展开的关系,并给出循环展开的有效性条件判断,提出了向量化体系结构下模变量扩展算法框架,有效解决该体系结构下模调度的代码生成问题.实验结果表明,提出的面向分簇向量化体系结构的模调度编译框架,可以大幅提升程序循环部分的性能,加速比为170%~680%.

关键词:模调度;资源描述;循环展开;模变量扩展;代码生成模式

中图分类号:TP314 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2016.02.00

引用格式:Wang Xiangqian, Zheng Qilong, Hong Yi. Research on modulo scheduling framework for clustered architecture[J]. Journal of University of Science and Technology of China, 2016,46(2):104-112.
王向前,郑启龙,洪一. 分簇结构模调度框架研究[J]. 中国科学技术大学学报,2016,46(2):104-112.

Research on modulo scheduling framework for clustered architecture

WANG Xiangqian[1,2], ZHENG Qilong[3], HONG Yi[1]

(1. No. 38 *Research Institute, China Electronics Technology Group Corporation, Hefei* 230088, *China*;
2. *School of Computer and Information, Hefei University of Technology, Hefei* 230009, *China*;
3. *School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: An implementation method for modulo scheduling framework was established for clustered architecture. The description method for its machine resources was introduced. The relationship between module scheduling and loop unrolling was researched, and then the effective decision condition on loop unroll was offered. The modulo variable expansion algorithm framework for vectorization was proposed. The code generation schema based on speculation execution was described. The experiment result shows that the modulo algorithm framework could bring speed-up ratio to 170%~680%.

Key words: modulo scheduling; resources description; loop unroll; modulo variable expansion; code generation schema

收稿日期:2015-11-23;**修回日期**:2016-01-01

基金项目:国家核高基重大专项(2012ZX01034001-001)资助.

作者简介:王向前,男,1985年生,博士/工程师. 研究方向:编译优化与系统软件优化. E-mail: forward@mail.ustc.edu.cn

通讯作者:郑启龙,博士/副教授. E-mail:qlzheng@ustc.edu.cn

0 引言

软件流水是一种强大的循环调度技术，它允许指令跨迭代的移动实现循环迭代的重叠执行. 模调度是一种典型的软件流水实现方法. 它通过要求以确定的启动间隔启动和每个迭代都必须具有相同的调度来简化重复调度的生成.

“魂芯”数字信号处理器(以下简称魂芯 DSP 或 BWDSP)是具有分簇结构、向量化执行、支持超长指令字(VLIW)的信号处理器. 面向魂芯 DSP 实现优化编译系统时，模调度对于提升循环执行的性能具有重要地位.

在面向魂芯 DSP 构建模调度框架时，需要考虑分簇结构和向量体系结构对框架的影响. 本文以魂芯 DSP 平台为例研究分簇结构上模调度编译框架的高效实现，论述模调度在该体系结构下实现时遇到的问题以及解决方法.

1 相关工作

Ebcioğlu等[1]的增强流水调度算法把软件流水问题划分为非循环调度问题的连续序列. 在每一步通过放置跨越循环的一个或者多个流边的同步障碍来打破调度区域的循环属性. 它的主要优点是可以处理含有多个控制流路径的循环. 它的主要缺点是计算复杂度过高.

Aiken 等[2-4]提出的完美调度算法通过循环展开和压缩循环体的方式发现重复模式. 该算法分为两个阶段：第一个阶段是应用全局代码移动优化技术尽可能早地调度循环体的指令；第二个阶段包括对已经紧凑的循环体进行循环展开、以贪婪的方式调度指令、检验一个重复模式的执行历史. 该循环模式成为软件流水的核心代码. 该算法的优点在于对调度没有同步障碍等限制条件. 每一个迭代不限制具有相同的调度. 启动间隔 II 的值和组成重复模式(核心)的指令集合在迭代过程中确定. 它的缺点主要有两个：①缺少一个可以确定停止调度的明确目标；②算法本身具有过高的时间复杂度.

Petri net 软件流水算法[5]试图解决重复模式的识别. 调度过程的状态通过一个 Petri net 和一个调度行为资源表格组合表示. 这种方法的主要优势在于定义了一个核心识别的系统方法. 它的主要缺点在于 Petri net 结点的当前状态与前一个状态的对比不清晰.

分解式软件流水算法[5]提出了一个新观点，把软件流水看作将一维指令向量变换成二维指令矩阵的过程. 指令矩阵的行代表指令的执行时刻，指令矩阵的列代表操作所属的循环体. 它把软件流水问题自然分解为两个子问题：确定每条指令在指令矩阵的行号和确定其在指令矩阵的列号.

线性规划软件流水方法[5]是针对软件流水的调度和限制进行数学描述，然后根据形式化的描述写出线程规划方程组，再根据用户与定义的开销评价函数求最优解.

模调度[6-7]最初由 Rau 等提出. 它克服了前几种算法的复杂性和实际困难，通过要求以恒定的速率启动每次迭代以及循环的所有迭代具有相同的调度，形成核心. 在调度之前根据循环依赖关系和机器资源情况确定一个固定的启动间隔 II，然后依据该固定间隔构建一个有效的调度. 模调度的优点是启动间隔 II 给调度器一个明确的目标，并且由于以固定的间隔启动每个迭代的调度，因此有助于减小寄存器压力.

模调度是最优秀的循环调度方法，它已经在众多产品编译系统(如 Cydrome，HP，SGI)里实现. 从某种意义上讲，模调度算法已经成为软件流水算法的代表. 国内外研究机构围绕模调度算法开展了大量软硬件层面的研究.

由 HP 和 intel 设计的安腾体系结构[8-9]提供支持软件流水的硬件机制，包括特殊的循环指令、旋转寄存器、循环阶段谓词(loop stage predication). 这些机制使得软件流水能够高效执行，而不需要增加代码体积(code size). 旋转寄存器是一种硬件支持的动态寄存器重命名机制，用于支持软件流水，而不需要在软件层面通过循环展开进行寄存器重命名的方式支持软件流水. 其通过特殊的软件流水循环跳转指令支持寄存器的旋转，并且和谓词机制一起，消除了入口代码(prologue)和出口代码(epilogue)的代码生成，形成只包括流水核心(kernel)的极为紧凑的软件流水代码形式，既提高了循环执行效率，又不会带来代码的膨胀. 这对于对代码体积比较敏感的嵌入式领域非常重要. 它是软件流水生成代码形式的极致，但采用硬件支持旋转寄存器、流水阶段谓词等会导致硬件开销过大，不符合现代处理器设计的理念.

Rau[6]提出了迭代模调度算法，对模调度的核心调度算法进行了详细的阐述，并对模调度的代码

生成形态做了全面系统的研究.

Llosa 等[10-11]针对模调度的代码膨胀问题进行了研究,通过合适的调度启发因子和合适的代码生成模式,可以大幅缓解模变量扩展导致的代码体积膨胀问题.

中科院计算技术研究所[12-14]针对 IA-64 架构进行软件流水的框架设计,并研究考虑寄存器压力因子、与循环展开的结合等场景下的软件流水算法的应对策略.

清华大学汤志忠教授[15]研究针对弹性数据相关依赖条件下的软件流水算法实现,取得了部分研究成果.

2 分簇架构支持 SIMD 的模调度框架

根据魂芯 DSP 体系结构特征,构建魂芯 DSP 的模调度框架如图 1 所示. 在此架构上支持模调度框架遇到的主要困难在于支持分簇结构与向量化. 传统的模调度框架并不支持分簇结构,并不能支持分簇的向量化体系架构,而针对分簇以及向量化体系结构构建软件流水是魂芯 DSP 发挥性能的关键. 针对分簇的向量化体系结构构建模调度编译框架的主要问题可以分为分簇结构机器资源的描述、模调度与循环展开的关系、模变量扩展、代码生成模式的选择. 本节先介绍模调度的基本架构.

(Ⅰ) 循环检测及选择

检查循环是不是适合进行模调度. 例如当循环体含有函数调用、循环体的指令条数过多或者过少时,是不适宜进行模调度的.

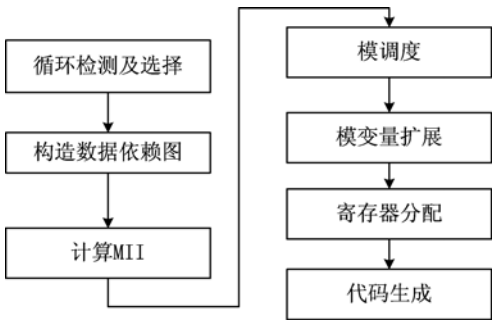

图 1 魂芯 DSP 模调度框架

Fig. 1 Modulo scheduling framework for BWDSP

(Ⅱ) 构建数据依赖图

进行非循环代码的调度(例如全局调度、局部调度)时构建的依赖图仅包括迭代内的依赖关系,而模调度启动的依赖图构建还包括迭代间的依赖关系. 用于循环调度的依赖图包括循环携带依赖、迭代内依赖关系,并不包括反依赖和输出依赖(但是包括由于专用寄存器 dedicated register 引起的反依赖,因为专用寄存器无法通过模变量扩展进行重命名). 存储依赖通过比较访存指令的存储流关系获得. 在进行模调度时,控制依赖(包括跨迭代的和迭代内的)可以忽略而不添加到依赖图.

(Ⅲ) 计算 MII

最小启动间隔 MII 是启动间隔的下界. 两个因子决定了 MII:关键资源使用情况和数据依赖图的关键依赖环,分别称为 ResMII 和 RecMII. ResMII 计算很简单:先把一个循环体对应需要的资源需求列表计算出来,可以通过简单的累加得到;然后把所需的每种资源数量除以体系结构提供的资源数量,该值最大者为 ResMII. RecMII 则是依赖图形成的各个环的延迟除以环跨越的迭代距离的最大值. 它可以通过 Monica Lam 的方法获得:首先使用 Tarjan 算法找出数据依赖图的所有强联通图(SCC),强联通图也是基本的环,然后使用 Floyd 算法为每一个强联通图分别计算所包含点彼此的最长路径.

(Ⅳ) 模调度

该步骤是模调度算法的核心步骤. 它的主要任务是在考虑机器资源和数据依赖的条件下,试图把循环的一个迭代的指令依次调度到空闲发射槽. 目前主流实现的是 Huff 的松弛调度[16-17]. 松弛调度的基本思想是循环体的每条指令有一定的调度自由,指令的松弛度定义为最晚启动时间和最早启动时间的差. 然后根据启发因子给每条指令分配一个优先级进行启发调度.

(Ⅴ) 模变量扩展

模变量扩展可以消除寄存器的反依赖和输出依赖. 首先计算出每个虚拟寄存器的生命周期,然后根据最长的生命周期和调度启动间隔 II 确定核心的循环展开次数. 进行循环展开后,就可以对有关寄存器进行重命名.

(Ⅵ) 寄存器分配

在进行模变量扩展之后,重新计算每个虚拟寄存器的生命周期,然后构建冲突图,进行寄存器分配. 此处的寄存器分配和常规的寄存器分配(包括基于图着色的全局寄存器分配算法和基于线性扫描的局部寄存器分配)有所不同.

(Ⅶ) 代码生成

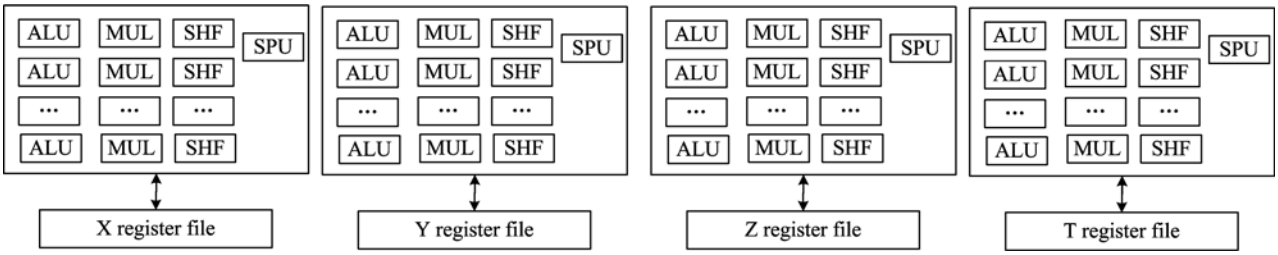

图 2　魂芯 DSP 分簇结构

Fig. 2　Clustered architecture in BWDSP

代码生成是模调度框架的最后一步. 它是把模调度的结果形成软件流水的代码形式.

3　分簇结构机器资源的描述

分簇结构可以有效增加芯片体系结构的指令级并行度,而又不会产生过多硬件代价. 它和集中式的体系结构相对应,是无线通信、视频、图像等数字信号处理领域高数据并行处理应用发展的需要. 魂芯 DSP 的分簇结构如图 2 所示. 分为 X 簇、Y 簇、Z 簇、T 簇,每簇由对应的寄存器文件和运算单元组成.

为了提高数据访存带宽,魂芯 DSP 的片上内存设计为分块结构,如图 3 所示.

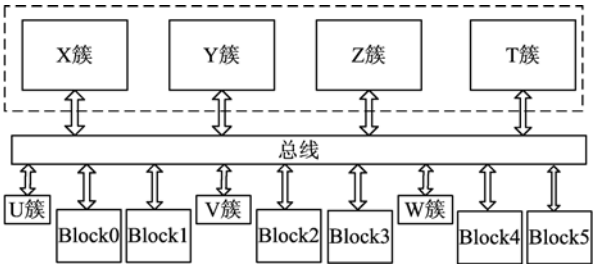

图 3　魂芯 DSP 分块内存结构

Fig. 3　Multi-block memory in BWDSP

魂芯 DSP 的主要机器资源如表 1 所示. 图 2 和图 3 所示的体系结构特征已经涵盖于该表. 魂芯 DSP 的指令类型主要分为 ALU 运算指令、MUL 指令、SHF 指令、双字指令、跳转指令、访存指令、计算簇间传输指令、计算簇和地址簇传输指令、地址单元计算指令、立即数赋值指令等几大类别.

指令和机器资源是一对多的关系. 每条指令都会占用一个 Slot. 除此之外,ALU/MUL 类指令会占用一个 ALU 或者 MUL;跳转指令会占用 Branch 单元;访存指令还会占用内存访问读写总线;双字指令还会占用双字发射指令槽;等等.

魂芯 DSP 的向量指令分为两种,访存向量化指令和计算向量化指令. 计算向量化是 R5 = R1 * R3,该指令的含义是 X,Y,Z,T 这 4 个簇同时执行该乘法指令,所以它占用分布在 4 个簇上的 xMUL,yMUL,zMUL,tMUL 4 个乘法器.

表 1　机器资源简表

Tab. 1　Machine resources

名　称	数量	备　注
Slot	16	一个指令行最多可以有 16 条指令
Branch	1	一个指令行最多只能执行一个跳转指令
xALU/yALU zALU/tALU	8	X/Y/Z/T 每个簇分别包括 8 个算术逻辑运算单元
xMUL/yMUL/ zMUL/tMUL	8	X/Y/Z/T 每个簇分别包括 8 个乘法器
MemReadBus	2	1 个指令行可以有最多 2 条读访存指令
MemWriteBus	2	1 个指令行可以有最多 2 条写访存指令

下面以“点积”为例,说明该架构下的 ResMII 的计算方法. 图 4 是“点积”向量化优化后的汇编表示形式. 注意,该代码假定考虑了分块内存,把“点积”的两个数组分别分布于两块内存上,并且对应的地址寄存器分别分配在 U 地址生成单元和 V 地址生成单元上. 这是为了利用魂芯 DSP 的高数据访存带宽.

```
__LOOP:
R1:0 = [U1+= 8,1]
R3:2 = [V2 += 8,1]
R5 = R1 * R3
R4 = R0 * R2
R7 += R5
R6 += R4
XR9 +=8
If XR9 < R10 b __LOOP
```

图 4　“点积”汇编表示形式

Fig. 4　Assembly representation for dot product

该循环体所需的 Slot 资源量为 8(8/16),Branch 为 1(1/1),xALU 为 3(3/8),yALU/zALU/tALU 分别为 2(2/8),xMUL/yMUL/zMUL/tMUL 分别为 2(2/8),MemReadBus 为 2(2/2),

MemU/MemV 分别为 1(1/1),所以 ResMII 为 1.

可见,在向量化指令条件下,把 SIMD 所需要的资源进行类似的累加操作,即可得到 ResMII. 该数据依赖图如图 5 所示,可以很容易计算出 RecMII=1. 由此可得 MII 为 1.

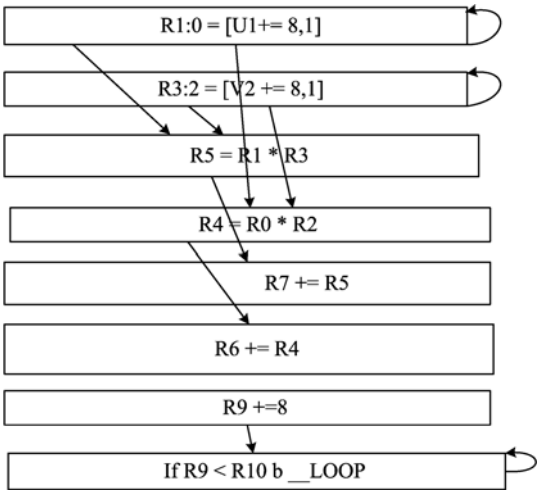

图 5 "点积"数据依赖图

Fig. 5 Data dependence graph for dot product

假定在 MII=1 时模调度成功,可以得出访存带宽的读带宽利用率为 100%(该循环没有写访存操作). 但是主要的计算资源 MUL 和 ALU 并没有得到充分利用:每个簇上的 MUL 的利用率为 25%,X 簇上的 ALU 利用率为 37.5%,Y/Z/T 簇上 ALU 的利用率仅为 25%.

4 循环展开与模调度

为了提高计算资源的利用率,可以进行循环展开. 针对上节"点积"的例子,循环展开因子确定为 2,称为"循环展开的点积",如图 6 所示. 注意,程序已经进行了累加变量扩展变换. 很容易知道 ResMII=2,RecMII =1,故而 MII=2. 假定在启动间隔 MII=2 时模调度成功,则计算资源利用率得到比较完美的提升.

```
__LOOP:
R1:0 = [U1+= 8,1]
R3:2 = [V2 += 8,1]
R11:10 = [U3+= 8,1]
R13:12 = [V4 += 8,1]
R5 = R1 * R3
R4 = R0 * R2
R15 = R11 * R13
R14 = R10 * R12
R7 += R5
R6 += R4
R17 += R15
R16 += R14
XR9 +=16
If XR9 < R10 b __LOOP
```

图 6 "循环展开的点积"的汇编表示形式

Fig. 6 Assembly representation for unrolled dot product

"点积"和"循环展开的点积"经过模调度过程得到的模调度和核心(kernel)如图 7、图 8 所示.

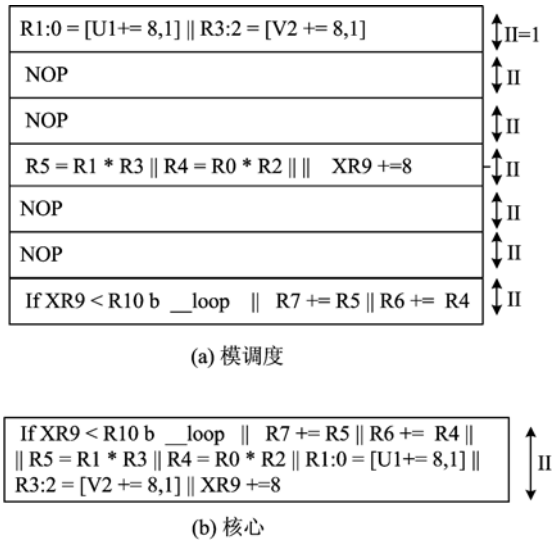

图 7 "点积"的模调度

Fig. 7 Modulo scheduling for dot product

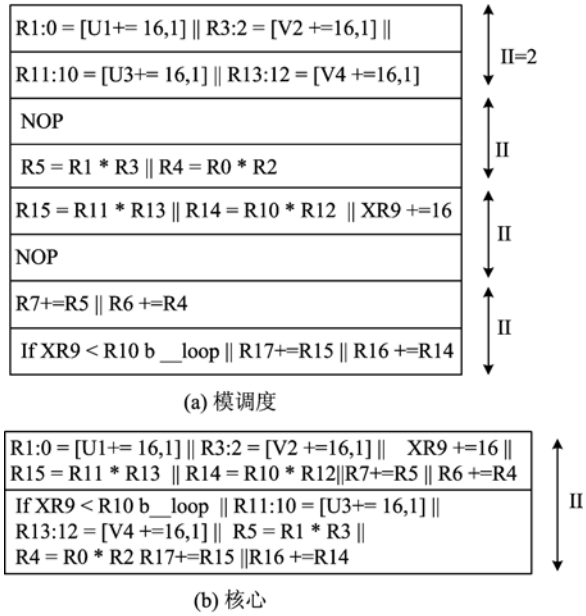

图 8 "循环展开的点积"模调度

Fig. 8 Modulo scheduling for unrolled dot product

假定点积循环次数为 N,则"循环展开的点积"的模调度后循环次数为 N/8,"循环展开的点积"模调度后的时间复杂度为(N/16)×2 = N/8. 可见,"循环展开的点积"计算资源利用率得到完美的提

升是以 MII=2 为前提的.“循环展开的点积”与“点积”相比,其资源利用率并没有提升.因为启动间隔 MII 从 1 变为了 2.

假如魂芯 DSP 的 MemReadBus 是 4,则“点积”的 MemReadBus 资源需求量为 2(2/4),进行模调度后的 MemReadBus 资源利用率为 50%.

此时循环展开因子仍确定为 2,则进行模调度后的 MemReadBus 资源利用率为 100%,MUL 的资源利用率为 50%,ALU 的利用率约为 50%,模调度优化的效果会得到改善.

假定某资源量为 Ri,循环体所需要的资源量为 ri,成功模调度的启动间隔为 II,则该资源 Ri 的利用率= ri/(Ri×II)×100%.

假定循环展开不影响循环体的 RecMII,假定按照循环因子 UF 循环展开后的启动间隔为 uMII,成功模调度后的启动间隔为 uII,则每个资源的利用率为(UF×ri)/(Ri×uMII) ×100%.

只要存在一个 i 有

(UF×ri)/(Ri×uII) > ri/(Ri×II),

并且对于其他 i,有

(UF×ri)/(Ri×uII) >= ri/(Ri×II),

则按照循环因子 UF 进行循环展开后,可以获得较高的资源利用率,提高循环执行时间.

推论 1　当循环体的资源需求量 ri/Ri 都小于 II 时,适度循环展开可能提高资源利用率,提高性能,循环展开因子 UF 根据最大的资源需要量 ri/Ri 确定,此时的 Ri 和 ri 分别记为 maxRi 和 maxri,则 UF= (maxRi×II)/maxri.

推论 2　当循环体的某个资源需求量的利用率 ri/(Ri×II)×100%=100%时,该资源利用率已达到最大,此时进行循环展开不会提高模调度的优化效果.

5　模变量扩展

首先需要确定模调度核心的循环展开次数.通过每个虚拟寄存器的生命周期确定循环展开次数.虚拟寄存器的生命周期是从它的第一个定值(D)到最后一个使用(U).循环变量 V 的生命周期的计算公式为

Lifetime(V)=Issue(U) - Issue(D)+II×Dist(V)

式中,Issue(U) 和 Issue(D)是发射时间,该发射时间是模调度调度成功后的调度时间;Dist(V)为 D 和 U 跨越的迭代距离,一般为 0 或 1.

如图 7 所示,寄存器 R0,R1,R2,R3,R4,R5,R6,R7,R9 的生命周期为 3.而 II=1,所以循环展开因子 UF 为 3.按照如图 9 所示代码生成算法,生成软件流水的入口代码、流水核心、出口代码,如图 10 所示.

```
Procedure GenCodeLayout()
begin
for(index = 0; index < SL; index++)
    unroll[0][index] =: schedule[index]
od
for(uf = 1 ; uf < UF ; uf ++)
  for(index = 0; index < SL; index++)
    unroll[uf][index] =: CopyandRenameReg(schedule[index])
  od
od
for(sc := 0; sc < sc_count -1 ;sc ++ )
  for(k := 0;k < II; k++)
     nil = new_instruction_line(pro_BB);
     for(uf := sc,index = 0; uf > 0;uf --,index++)
      if(unroll[uf%UF][k + index*sc*II] != NULL)
        append(nil,unroll[uf%UF][k + sc*II])
      fi
     od
  od
od
for(sc < sc_count + UF ;sc ++ )
  for(k := 0;k < II; k++)
     nil = new_instruction_line(kernal_BB);
     for(uf := sc,index = 0; uf > 0;uf --,index++)
      if(unroll[uf%UF][k + index*sc*II] != NULL)
        append(nil,unroll[uf%UF][k + sc*II])
      fi
    od
  od
od
end
```

图 9　模变量扩展及代码生成算法

Fig. 9　Algorithm for MVE and code generation

根据确定的循环展开因子 UF 对已形成的调度进行循环展开,形成二维数组 unroll,存储 UF 次循环展开的循环体(包括一个原始的循环体和新形成的 UF-1 个循环体).在生成过程中要对循环变量依照循环展开因子进行重命名.其中软件流水阶段数 SC= SL/II,式中 SL 为调度长度.

在生成出口代码时需要确定和对应跳转指令同步的指令.考虑到魂芯 DSP 支持部分指令的推测执行[19],在流水核心里认可执行的指令为原循环体结束时活跃的指令,包括写访存指令和原循环体结束后活跃变量发生定值的指令,将在第 6 节里详细论述.

6　代码生成与寄存器分配方法

代码生成需要选择代码生成模式[18-19],代码生成模式是模调度生成流水代码时遵循的规则,它直

```
Header:
If xr10 < 24 b __exit_loop
Prologue: (xr9 = 0 ; xr19 = 8; xr29 = 16)
R1:0 = [U1+= 8,1] || R3:2 = [V2 += 8,1]
R11:10 = [U1+= 8,1] || R13:12 = [V2 += 8,1]
R21:20 = [U1+= 8,1] || R23:22 = [V2 += 8,1]
R1:0 = [U1+= 8,1] || R3:2 = [V2 += 8,1] || R5 = R1 * R3 || R4 = R0 * R2 || XR9 +=24
R11:10 = [U1+= 8,1] || R13:12 = [V2 += 8,1] || R15 = R11 * R13 || R14 = R10 * R12 || XR19 +=24
R21:20 = [U1+= 8,1] || R23:22 = [V2 += 8,1] || R25 = R21 * R23 || R24 = R20 * R22 || XR29 +=24
__Kernel:
If XR9 >= R10 b EPI_1 || R7 += R5 || R6 += R4 || R1:0 = [U1+= 8,1] || R3:2 = [V2 += 8,1] || R5 = R1 * R3 || R4 = R0 * R2 || XR9 +=24
If XR19 >= R10 b EPI_2 || R17 += R15 || R16 += R14 || R11:10 = [U1+= 8,1] || R13:12 = [V2 += 8,1] || R15 = R11 * R13 || R14 = R10 * R12 || XR19 +=24
If XR29 < R10 b __Kemal || R27 += R25 || R26 += R24 || R21:20 = [U1+= 8,1] || R23:22 = [V2 += 8,1] || R25 = R21 * R23 || R24 = R20 * R22 || XR29 +=24
EPI_0:
 R7 += R5 || R6 += R4
 R17 += R15 || R16 += R14
 B __exit_loop
EPI_1:
 R17 += R15 || R16 += R14
 R27 += R25 || R26 += R24
 B __exit_loop
EPI_2:
 R27 += R25 || R26 += R24
 R7 += R5 || R6 += R4
__exit_loop:
```

图 10 “点积”模变量扩展

Fig. 10 MVE for dot product

接确定模调度的最终代码形态.需要结合硬件体系结构考虑模调度的代码生成模式.魂芯 DSP 允许 load 执行推测执行,所以代码生成的主要规则及代码生成方法如下:

(Ⅰ)允许原循环体结束时不活跃变量的推测执行,包括 load 执行和普通的运算指令;

(Ⅱ)流水结束时,流水核心中认可执行的指令为在原循环体结束时活跃的指令,包括写访存指令和原循环体结束后发生活跃变量定值的指令;

(Ⅲ)如果循环控制为零开销循环时,流水代码只包括入口代码和流水核心两部分,不会生成出口代码.

图 10 所示的代码在流水循环结束时,认可执行的指令为

R7+=R5||R6+=R4

R17+=R15||R16+=R14

R27+=R25||R26+=R24

而其他指令,如 load 指令和 mul 指令计算的结果则不予理睬.需要注意这种生成模式会影响循环执行次数的计算.

而软件流水的寄存器分配拟借用已经实现的寄存器分配框架.直接生成的软件流水代码的寄存器生命周期不符合常规寄存器分配的要求.所以软件流水代码并不直接生成于程序之中,而是建构与软件流水代码活跃变量等价的基本块代码,并放置在程序对应位置;随后参与寄存器分配;寄存器分配之后,把等价的基本块再替换为软件流水基本块,借用建构的与软件流水代码寄存器生命周期等价的基本块完成软件流水的寄存器分配.

7 性能测试

由于魂芯 DSP 支持向量化指令执行,因此进行模调度评价时,以向量化优化的结果为基准,进行模调度的优化效果评估.

本节基于开源编译基础设施 Open64 实现本文提出的模调度框架,以数字信号处理领域经典的内核算法为测试集合(包括 FFT_Radix2,FFT_Radix 4, FIR_cplx, FIR_real, FIR_singlesample, IIR, lmsFIR, VECTOR_dotprod, VECTOR_max, VECTOR_sum),在魂芯 DSP 的软件模拟器 BWSIM 上评估模调度的优化效果.

为了从实验中验证节 4 的模调度框架中关于循环展开必要性的推论 1 和推论 2,我们用分析出的对循环因子敏感的典型用例 VECTOR_max 为例进行实验评估.在对 VECTOR_max 进行模调度时,依次以循环因子 UF=1,UF=2,UF=4,UF=8,UF=16 进行循环展开,对应的模调度优化测试结果如图 11 所示.可见在 UF=2 时,VECTOR_max

模调度优化的性能最高，加速比超过 600%. 但随着 UF 的逐渐增大，VECTOR_max 模调度优化的加速比急剧减小至 100%以下. 这是由于当 UF＝2 时，访存利用率为 100%，而 ALU 的利用率为 50%，和 UF＝1 相比，大幅提高了资源的利用效率；而且此时的寄存器需求也未超过魂芯 DSP 提供的 caller 寄存器，进行寄存器分配时不需要寄存器的溢出操作. 当 UF＝4 时，虽然有可能提高 ALU 的利用率（实际上并未提高），但是此时的寄存器需求已经大大超过 caller 寄存器数量，寄存器分配时溢出的代价降低了模调度优化的收益. 当 UF＝8 和 UF＝16 时，寄存器溢出的代价已经超过了模调度优化的收益而使得模调度优化的加速比小于 100%. 可见，节 4 中推论 1 和推论 2 的结论是有效的.

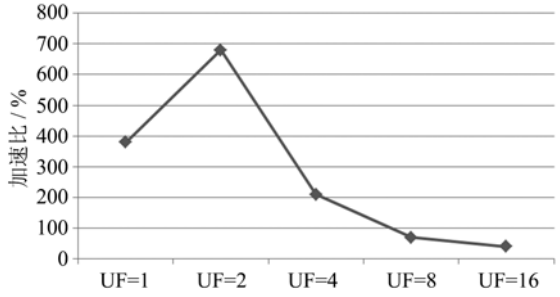

图 11　VECTOR_max 的模调度优化效果

Fig. 11　Performance of modulo scheduling for VECTOR_max

在向量化优化的基础上，进行模调度优化的性能评测，测试结果如图 12 所示，其中 VECTOR_max 用例的模调度过程中循环展开因子 UF 为 2，其他用例的 UF 均为 1. 测试集合各个测试用例分别获得了加速比为 170%～680%的收益. 注意这里的性能加速比是相对于向量化优化的效果. 其中

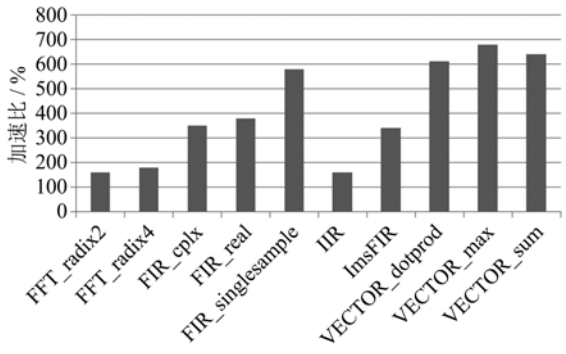

图 12　模调度优化效果

Fig. 12　Performance of modulo scheduling

IIR 也获得了较为明显的改进，而该用例未成功进行向量化转换；FFT_radix2 和 FFT_radix4 在模调度优化的表现明显不如其他测试用例的表现，主要是 FFT 算法最内层循环控制较为复杂，随着外层循环的迭代，内层循环的循环次数是变化的，进行模调度的代价较高；VECTOR_max 用例获得的性能加速比最高，超过 600%，可见模调度时进行必要的循环展开的重要性. 由此可见，模调度优化比向量化优化具有更广泛的适用性，其不用考虑循环携带依赖而直接进行模调度优化，可以改进更多循环类型的性能.

8　结论

本文针对魂芯 DSP 分簇、向量化执行的体系结构特点，研究了其模调度实现框架的几个主要问题，包括该体系结构下的机器资源描述方法、模调度与循环展开的关系、模变量扩展方法、代码生成模式的选择以及寄存器分配方法等. 选用经典的算法内核进行性能评测，实验结果表明，该模调度框架能够较好地利用魂芯 DSP 的体系结构特点，提升循环执行的性能.

参考文献（References）

[1] Ebcioğlu K, Nakatani T. Anew compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture[C] // Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing. London, UK : Pitman Publishing, 1990.

[2] Aiken A, Nicolau A, Novack S. Resource-Constrained Software Pipelining [J]. IEEE Transactions on Parallel & Distributed Systems, 1995, 6(12):1 248-1 270.

[3] Ayala M, Benabid A, Artigues C, et al. The resource-constrained modulo scheduling problem: an experimental study[J]. Computational Optimization & Applications, 2011, 54(3):645-673.

[4] Wei H, Yu J, Yu H, et al. Software Pipelining for Stream Programs on Resource Constrained Multicore Architectures[J]. IEEE Transactions on Parallel & Distributed Systems, 2012, 23(12):2 338-2 350.

[5] Kejariwal A, Nicolau A. Modulo Scheduling and Loop Pipelining[M]// Encyclopedia of Parallel Computing. SpringerUS, 2011:1 158-1 173.

[6] Rau B R. Iterative modulo scheduling: An algorithm for software pipelining loops[C]// Proceedings of the 27th Annual International Symposium on

Microarchitecture. New York, NY, USA : ACM, 1994:63-74.

[7] Kim W, Yoo D, Park H, et al. SCC based modulo scheduling for coarse-grained reconfigurable processors [C]// IEEE International Conference on Field-Programmable Technology (FPT). Piscataway: IEEE Press, 2012:321-328.

[8] Huck J, Morris D, Ross J, et al. Introducing the IA-64 Architecture[J]. Micro IEEE, 2000, 20(5):12-23.

[9] 李兆晖. 基于 IA64 平台 OSPFv2/v3 协议的设计与移植[D]. 北京:北京交通大学, 2015.

[10] Llosa J, Freudenberger S M. Reduced code size modulo scheduling in the absence of hardware support [C]// Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. Los Alamitos, CA, USA : IEEE Computer Society Press, 2002:99-110.

[11] Hwu W M W, Merten M C. Method and apparatus for modulo scheduled loop execution in a processor architecture: USA, 7725696[P]. 2010-05-25.

[12] 林海波. 基于 EPIC 体系结构的软件流水技术研究[D]. 北京:清华大学, 2003.

[13] Lin Haibo, Li Wenlong, Tang Zhizhong. Reducing software pipelining failure on IA-64[J]. Journal of Tsinghua University (Science and Technology), 2003, 43(7):997-1 000.
林海波, 李文龙, 汤志忠. IA-64 中软件流水失败的解决方法[J]. 清华大学学报(自然科学版), 2003, 43(7):997-1 000.

[14] Lin Haibo, Li Wenlong, Tang Zhizhong. Research on register requirements of software pipelined loops in the IA-64 architecture[J]. Journal of Computer Research and Development, 2004, 41(1):22-27.
林海波, 李文龙, 汤志忠. IA-64 中软件流水的寄存器需求研究[J]. 计算机研究与发展, 2004, 41(1):22-27.

[15] Rong Hongbo, Tang Zhizhong. Flexible data dependence and software pipelining[J]. Journal of Software, 2001, 12(6):894-906.
容红波, 汤志忠. 弹性数据相关与软件流水[J]. 软件学报, 2001, 12(6):894-906.

[16] Huff R A. Lifetime-sensitive modulo scheduling[C]// Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation. New York, NY, USA:ACM,1993:258-267.

[17] Eichenberger A E, Davidson E S, Abraham S G. Author retrospective for optimum modulo schedules for minimum register requirements[C]//Proceedings of the 25th ACM International Conference on Supercomputing. New York, NY, USA: ACM, 2014:35-36.

[18] Schlansker M S, Tirumalai P P. Code generation schema for modulo scheduled loops[J]. Acm Sigmicro Newsletter, 1992, 23:158-169.

[19] Schlansker M. Cydra 5[M]// Encyclopedia of Parallel Computing. Springer US, 2011:488-497.