基于优化回溯模型的无重叠模调度算法

谭明星[1,2,3],刘先华[1,2],张吉豫[1,2],程 旭[1,2,3]
(1.北京大学信息科学技术学院,北京 100871;2. 微处理器及系统教育部工程研究中心,北京 100871;
3.北京大学深圳研究生院,广东深圳 518055)

摘 要: 软件流水技术通过重组循环体来挖掘指令级并行性,模调度是一类广泛使用的软件流水调度算法.传统模调度算法通常会产生变量活跃域重叠和寄存器压力增大问题,无法适用于嵌入式处理器.本文面向嵌入式处理器特性,建立了一种优化回溯模型,并基于该回溯模型提出了一种面向嵌入式处理器的无重叠模调度算法(NOn-Overlapped Iterative Modulo Scheduling,简称 NOOI).NOOI 算法使用循环相关反依赖消除变量活跃域重叠,并使用依赖约束和资源约束回溯模型消解节点冲突,从而提高了模调度的有效性.实验结果表明,NOOI 模调度算法能够有效改进模调度成功率和循环启动间距,并提高程序性能.

关键词: 嵌入式处理器; 软件流水; 模调度; 回溯模型
中图分类号: TP302.7 **文献标识码:** A **文章编号:** 0372-2112 (2012)08-1681-06
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2012.08.030

Non-overlapped Modulo Scheduling with Optimized Backtracking Model

TAN Ming-xing[1,2,3], LIU Xian-hua[1,2], ZHANG Ji-yu[1,2], CHENG Xu[1,2,3]
(1. *School of Electronics Engineering and Computer Science Peking University*, *Beijing* 100871, *China*;
2. *Engineering Research Center of Microprocessor & System*, *Ministry of Education*, *Beijing* 100871, *China*;
3. *Shenzhen Graduate School of Peking University*, *Shenzhen*, *Guougdong* 518055, *China*)

Abstract: Software pipelining exploits instruction level parallelism by reconstructing loops, while modulo scheduling is a kind of widely used scheduling algorithms for software pipelining. Traditional modulo scheduling algorithms usually cause overlapping register lifetimes and increase register pressure, and thus are not applicable to embedded processors. This paper presents the NOn-Overlapped Iterative (NOOI) modulo scheduling algorithm based on an optimized backtracking model for embedded processors. NOOI algorithm avoids the register lifetime overlap by adding loop-carried anti-dependence and resolves the scheduling conflicts using dependence-constrained and resource-constrained backtracking model. Our evaluations show that NOOI can significantly improve the success ratio and loop initial interval, which leads to better program performance.

Key words: embedded processor; software pipelining; modulo scheduling; backtracking model

1 引言

软件流水(Software Pipelining)[1]是一种针对循环的细粒度指令调度技术.模调度(Modulo Scheduling)[1~4]是一类常用的启发式软件流水调度方法,但是传统模调度算法[5,6]主要针对超标量和超长指令字处理器,在调度过程中会产生变量活跃域重叠(Lifetime overlap)[2]和寄存器压力增大问题[7].为了解决变量活跃域重叠问题,传统模调度算法主要使用旋转寄存器[7~9]、寄存器队列[10]、模变量展开[1]等技术,这些技术或者依赖于复杂硬件结构,或者会导致代码膨胀.为了获得更高的能耗有效性[11],嵌入式处理器一般不会提供旋转寄存器等特殊硬件,也不允许代码膨胀,因此传统模调度算法难以适用于嵌入式处理器.最近,Eric 等人[2]提出了一种可用于嵌入式处理器的模调度算法,通过添加循环相关反依赖以避免变量活跃域重叠问题.然而,额外的循环相关反依赖约束使得调度过程中违反模调度约束条件的节点冲突增多,导致现有的回溯模型失效,并导致调度算法的调度成功率降低和代码性能下降.

本文面向嵌入式处理器特点,提出了一种基于优化回溯模型的无重叠软件流水模调度算法(NOOI: NOn-Overlapped Iterative Modulo Scheduling).该算法针对冲突发生的不同原因分别建立依赖约束回溯模型和资源约束回溯模型,并从"回溯有效性判断"、"回溯节点选择"和"回溯节点重调度"三个方面对调度过程进行优化,重新调整节点状态和优先级.实验结果表明,NOOI 算法能够有效提高调度成功率,减小循环启动间距和寄存器压力,进而提高程序性能.

收稿日期:2011-03-18;修回日期:2012-01-10
基金项目:国家核高基重大专项(No.2009ZX01029-001-002,No.2009ZX01036-001-003);北京市自然科学基金(No.4123098)

2 模调度技术

模调度是一类广泛使用的启发式软件流水调度算法[8].在模调度中,相邻两个循环体启动时刻的差值称为启动间距(II:Initiation Interval),其大小决定了循环的性能.模调度使用"先假设启动间距后调度"启发式策略寻找尽量小的启动间距以完成循环重组.重组后的循环由装入(prolog)、核心(kernel)和排空(epilog)3部分组成,如图1所示.在prolog部分,前若干个循环体以II为间隔依次启动,然后进入一个长度为II、反复执行的kernel部分,直到进入epilog部分并结束循环.

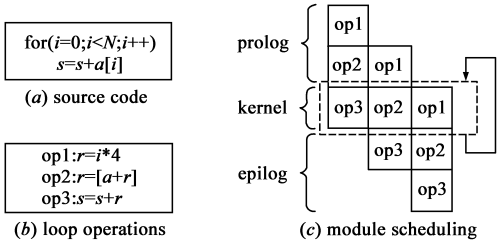

图1 模调度示意图

传统模调度算法主要面向超标量和超长指令字处理器,在调度过程中会产生"变量活跃域重叠"问题,即:变量活跃域的长度大于循环启动间距,循环体将新值写入变量之后,上一循环体却仍然需要使用该变量的旧值,此时变量已经被改写,从而导致程序出错.例如,在图1中op1和op2在同一个周期先后写入同一寄存器r,那么op1写入r的结果在被使用之前,就已经被op2的写操作所覆盖.针对这一问题,传统模调度算法主要采用模变量展开[1]、旋转寄存器[9]、寄存器队列[10]技术,它们或者会导致代码尺寸膨胀[1],或者需要特殊硬件支持[9],难以适用于嵌入式处理器.

最近,Eric等人[2]提出了一种无需复杂硬件支持即可消除变量活跃域重叠问题的机制.该机制在传统数据依赖图中加入了循环相关反依赖(Loop-carried anti-dependence)以避免变量活跃域重叠问题.然而,该机制所引入的循环相关反依赖增加了数据依赖图的复杂性,引起较多的节点冲突,即:因数据依赖约束和资源约束条件的限制而无法完成当前节点调度的情况.大量的节点冲突会严重降低调度成功率和程序性能.高效的回溯模型可以有效消除节点冲突,然而已有的软件流水回溯模型都较为简单.Huff等人[7]最早提出了一种受限的软件流水回溯模型,当遇到节点冲突时,从最早开始时间到最晚开始时间范围内选择一个最小未调度位置,作为当前节点的调度位置,并取消所有与之发生冲突的节点调度.Rau等人[8]进一步完善了该回溯模型,它首先取消当前节点所有直接后继节点的调度结果,然后在新的约束条件下重新调度当前节点.Eric等人[2]在节点取消方面对回溯模型进行了改进,在发生数据依赖冲突时保留部分不影响当前调度的节点,从而尽量减少节点取消数量.然而,该回溯模型仅在"取消调度结果"方面进行改进,其误预测率仍然较高.

3 依赖约束和资源约束回溯模型

3.1 回溯模型优化设计

本文回溯模型对节点冲突时的调度过程进行扩展.当节点冲突发生时,根据冲突的不同原因,分别从回溯可行性判断、回溯节点选择和回溯节点重调度三个方面进行优化.相对于已有回溯模型,本文回溯模型主要有两个特点:

(1)根据调度原因的不同分别使用不同回溯模型.引起节点冲突的原因主要有数据依赖约束和资源约束,数据依赖约束由程序语义所决定,而资源约束主要影响执行性能,它们对前驱后继的影响不同.因此,本文分别建立了依赖约束回溯模型和资源约束回溯模型.

(2)从回溯可行性判断、回溯节点选择和回溯节点重调度三个方面进行优化设计.首先根据当前调度状态和依赖约束条件,判断是否值得进行回溯,然后根据优化目标和约束条件,仅取消少量回溯节点的调度结果;对于被取消的回溯节点,重新设置它们的调度优先级,以提高回溯节点重调度的有效性.

3.2 依赖约束回溯模型

依赖约束回溯模型(DBM:Dependence-constrained Backtracking Model)用于消解由数据依赖约束所引起的数据依赖冲突.数据依赖冲突是指找不到满足依赖约束条件的调度位置,当前节点调度窗口为空.依赖约束回溯模型通过取消部分前驱后继节点,以获得一个非空调度窗口.下面从回溯可行性判断、回溯节点选择和回溯节点重调度三部分给出具体设计.

3.2.1 依赖约束回溯模型的可行性判断

为确保回溯过程是有益的,节点依赖关系不应太复杂.若节点有较多前驱后继节点未被调度,则进行回溯可能会引起颠簸,因此当前节点 v 应满足以下条件:

$$|PRED(v)-SCHED|+|SUCC(v)-SCHED| < k*|PRED(v)+SUCC(v)| \quad (1)$$

其中 $SCHED$ 为已完成调度的节点集合,$PRED$ 和 $SUCC$ 分别表示当前节点的前驱和后继节点集合,参数 $k(0<k<1)$ 用于控制尚未调度的前驱和后继节点的比重.

3.2.2 依赖约束回溯模型的回溯节点选择

当前节点发生数据依赖冲突时,需要取消部分已调度节点以减弱约束条件,这些被取消的节点就是回溯节点.选择最优的回溯节点需要考虑许多因素,本文

从回溯过程的有效性出发,主要考虑以下因素:

(1)目标一:$\min(USLD(PS,v))$:PS 为当前调度状态,v 为当前发生冲突的节点,$USLD$ 为因依赖约束冲突而被取消的节点集合.回溯模型的目标是在不增加启动间距的条件下消解节点冲突并继续调度,因此在回溯过程中应尽可能少地取消已经完成调度的节点.

(2)目标二:$\max(SW(PS,v))$:SW 为当前节点 v 的调度窗口,即:回溯过程应该使得当前节点调度窗口尽量大,从而给后续调度提供较大选择空间.

(3)目标三:$\max \sum_{\forall n\in(V\text{-}SCHED)} SW(PS,n)$:$V$ 为所有节点集合,$SCHED$ 为已完成调度的节点集合.回溯过程还应使得未调度节点的调度窗口尽可能大.

除了以上三个节点选择目标,回溯节点选择过程还应该满足一些基本约束:

约束一:节点取消操作应该是有效的,即被取消节点的调度位置应在最早开始时间和最晚开始时间之间:$\forall_{n\in USLD(PS,v)} ASAP(v)\leqslant SC(n)\leqslant ALAP(v)$,其中 $USLD$ 为被取消节点集合,SC 为节点调度位置,$ASAP$ 和 $ALAP$ 分别为当前节点的最早开始时间和最晚开始时间.

约束二:回溯节点被取消后仍然是可调度的,即回溯节点被取消后其对应的调度窗口不能为空:$\forall_{n\in USLD(PS,v)}(END(PS,n)-START(PS,n))>0$,其中 $START$ 和 END 分别为调度窗口的开始和结束位置.

约束三:当前回溯不会引起颠簸.为避免节点因反复调度到同一位置而发生颠簸,要求调度窗口中存在尚未被调度过的位置:$\exists_{START(PS,v)\leqslant t\leqslant END(PS,v)} t\notin HISTORY(v)$,其中 $HISTORY$ 为当前节点曾经调度过的历史位置.

将回溯节点选择过程看成一个给定约束条件的目标优化问题,得到回溯节点选择问题为:

$$\begin{aligned}&\min(USLD(PS,v)),\max(SW(PS,v))\\&\max\sum_{\forall n\in(V-SCHED)} SW(PS,n)\end{aligned}\tag{2}$$

$$\text{s.t.}\begin{cases}\forall_{n\in USL-D(PS,v)} ASAP(v)\leqslant SC(n)\leqslant ALAP(v)\\ \forall_{n\in USLD(PS,v)} END(PS,n)-START(PS,n)>0\\ \exists_{START(PS,v)\leqslant t\leqslant END(PS,v)} t\notin HISTORY(v)\end{cases}$$

3.2.3　依赖约束回溯模型的回溯节点重调度

回溯节点被取消后,需要重新设置其优先级.本文为保证调度有效性,按以下原则更新回溯节点的优先级.原则一:回溯节点优先级应小于从未被调度的节点;原则二:回溯节点被取消次数越多,其优先级就应越小;原则三:被取消相同次数的回溯节点,其优先级顺序应和初始优先级的大小顺序保持一致.

设 $P(n)$为回溯节点 n 调度前固有优先级,$HISTORY(n)$为历史位置集合,mp 为调度前所有节点的最大优先级,$SCHED$ 为已经完成调度的节点集合.根据以上三个原则,设置回溯节点 n 新的优先级 $P'(n)$:

$$\begin{aligned}mp&=\max\nolimits_{\forall n\in V}P(n),P(n)>0\\P'(n)&=-|HISTORY(n)|*mp+P(n)\end{aligned}\tag{3}$$

可以证明,该优先级更新方式满足以上三个原则:

(1)原则一是被满足的,即:任何回溯节点 n_1 的优先级 $P'(n_1)$总是小于从未被调度节点 n_2 的优先级 $P'(n_2)$.

$\forall n_1\in SCHED, n_2\notin SCHED,$

有 $HISTORY(n_1)\geqslant 1, HISTORY(n_2)=0$

$$\begin{aligned}\Rightarrow P'(n_1)&=-|HISTORY(n_1)|*mp+P(n_1)\\&\leqslant -mp+P(n_1)\leqslant 0\end{aligned}$$

$$P'(n_2)=-|HISTORY(n_2)|*mp+P(n_2)=P(n_2)>0$$

$$\Rightarrow P'(n_1)<P'(n_2)$$

(2)原则二是被满足的,即:若节点 n_1 被取消的次数大于节点 n_2,则 n_1 的优先级 $P'(n_1)$小于 $P'(n_2)$.

$\forall n_1,n_2\in SCHED,$

$$|HISTORY(n_1)|\geqslant|HISTORY(n_2)|+1$$

$$\begin{aligned}\Rightarrow P'(n_1)&=-|HISTORY(n_1)|*mp+P(n_1)\\&\leqslant -|HISTORY(n_2)|*mp-mp+P(n_1)\\&<-|HISTORY(n_2)|*mp\\&<-|HISTORY(n_2)|*mp+P(n_2)=P'(n_2)\end{aligned}$$

(3)原则三是被满足的,即:若节点 n_1、n_2 取消次数相同,则更新后优先级 $P'(n_1)$、$P'(n_2)$大小关系保持不变,即若 $P(n_1)\leqslant P(n_2)$则必有 $P'(n_1)\leqslant P'(n_2)$.

$\forall n_1,n_2\in SCHED,$

$$|HISTORY(n_1)|=|HISTORY(n_2)|,\ P(n_1)\leqslant P(n_2)$$

$$\begin{aligned}\Rightarrow P'(n_1)&=-|HISTORY(n_1)|*mp+P(n_1)\\&=-|HISTORY(n_2)|*mp+P(n_1)\\&\leqslant -|HISTORY(n_2)|*mp+P(n_2)=P'(n_2)\end{aligned}$$

3.3　资源约束回溯模型

资源约束回溯模型(RBM:Resource-constrained Backtracking Model)用于消解由资源抢占所引起的资源约束冲突.资源约束冲突是指在当前调度状态条件下,无论将当前节点安排在调度窗口中的任何位置,都会导致多个指令同时占用某个独占资源.资源约束回溯模型需要同时遵循依赖约束和资源约束,其目标是从调度窗口中选择合适的位置作为当前节点的调度位置.

回溯节点重调度过程与节点冲突原因无关,因此本文仅对资源约束回溯模型中可行性判断和回溯节点选择进行优化设计,而重调度过程则与 DBM 一致.

3.3.1　依赖约束回溯模型的可行性判断

现代处理器一般使用互锁等硬件机制解决资源抢占,因此资源抢占只影响程序性能而不影响正确性,这给回溯过程带来更大灵活性.由于嵌入式处理器中资

源冲突较简单,可认为所有资源回溯都是有益的.

3.3.2 依赖约束回溯模型的回溯节点选择

在资源回溯模型中,当前节点的调度窗口已经获得,窗口内所有位置都满足依赖约束条件但会发生资源抢占,此时需取消部分与当前节点发生资源抢占的节点.本文从资源有效性出发建立回溯节点选择目标:

目标一:$\min(USLR(PS,v))$:回溯导致的节点取消应该尽可能少.其中 PS 为当前部分调度状态,$USLR$ 是指因资源冲突而被取消的节点集合.

目标二:$\min\sum_{\forall c} KR(c)$:回溯节点所占用的关键资源尽可能少.关键资源 $KR(c)$ 是指在周期 c 中仅剩下一个可用的资源,若当前节点使用了该资源,则这个周期将不允许再安排任何其它指令.

从调度有效性和资源使用有效性出发,资源约束模型的回溯节点选择过程还应该满足以下约束:

约束一:回溯节点被取消后仍然是可调度的,即 $\forall_{n\in USLR(PS,v)} SW(PS',n)>0$,其中 PS' 为节点被取消之后的调度状态,SW 为节点 v 的调度窗口.

约束二:当前回溯不会引起颠簸.通过记录节点的调度历史 $HISTORY$,保证节点不会被重复调度到相同位置,即 $SC\notin HISTORY$,其中 SC 为当前调度位置.

约束三:回溯过程不增加寄存器压力,以避免引起寄存器溢出,即:$RP(PS')<\max(RP(PS),LIMIT)$,其中 PS 和 PS' 分别为回溯前后的调度状态,RP 为循环所用寄存器数量,$LIMIT$ 为最大可用寄存器数量.

将回溯节点选择过程看成一个给定约束的目标优化问题,得到回溯节点选择问题为:

$$\min(USLR(PS,v)),\min\sum_{\forall c} KR(c) \qquad (4)$$

$$\text{s.t.}\begin{cases}\forall_{n\in DC(PS,v)} SW(PS',v)>0\\ SC\notin HISTORY\\ RP(PS')<\max(RP(PS),LIMIT)\end{cases}$$

4 NOOI 模调度算法

4.1 NOOI 模调度算法总体框架

本文基于优化回溯模型 DBM 和 RBM,实现了一种面向嵌入式处理器的无重叠模调度算法(NOn-Overlapped Iterative Modulo Scheduling,NOOI).NOOI 算法通过添加循环相关反依赖避免变量活跃域重叠问题,并使用优化回溯模型消除节点冲突.NOOI 模调度算法流程如图 2 所示,主要分为六个部分:

(1)循环选择:选择最内层单基本块循环作为调度对象;

(2)依赖和资源约束描述:建立数据依赖图(DDG)和资源表(MRT),通过添加循环相关反依赖以避免活跃域重叠;

(3)计算初始启动间距:根据数据依赖约束和资源约束计算启动间距相应的下限值 recMII 和 resMII[1],并取二者的最大值作为初始值 MII,即:MII = max{recMII, resMII};

(4)节点调度:给定初始启动间距 MII,逐个节点尝试进行调度,并使用 DBM 和 RBM 回溯模型消解冲突,直到完成所有节点调度.若在当前启动间距条件下无法完成调度,则增加启动间距进行下一轮调度;

(5)寄存器压力检测:若寄存器压力过大时放弃当前调度;

(6)装入和排空代码生成:根据调度结果生成循环的装入和排空代码,确保程序正确性.

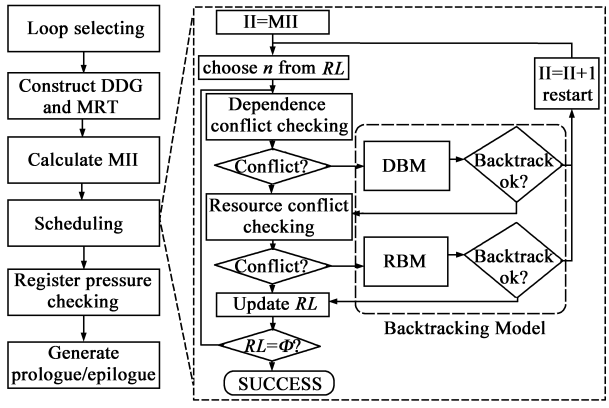

图2 NOOI算法流程图

4.2 变量活跃域重叠的消除

本文借鉴文献[2]提出的方法,通过在数据依赖图中添加循环相关反依赖以消除变量活跃域重叠问题.图 3(a)给出了一个包含四条指令的循环代码示例,假定真依赖(True Dependence)的延时为 3 周期,循环相关依赖(Loop-carried dependence)延时为 1 周期,处理器每周期执行一条指令.图 3(b)和(c)分别为添加循环相关反依赖前后的数据依赖图.由于该嵌入式处理器没有旋转寄存器等特殊硬件支持,因此需要添加循环反依赖以避免变量活跃域重叠问题.图 3(d)为添加循环反依赖后的最长依赖环,其延时之和为 10,循环体之差(循环相关依赖边的数量)为 1,最小启动间距为[1]:

$$recMII=\max_{\forall 依赖回路}\left\{\frac{依赖回路延时之和}{依赖回路的循环体之差}\right\}=\frac{10}{1}=10$$

为了减小启动间距,NOOI 算法通过插入寄存器拷

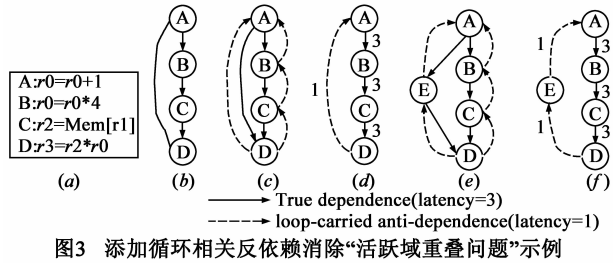

图3 添加循环相关反依赖消除"活跃域重叠问题"示例

贝指令 E 实现显式寄存器换名,将节点 A 到 D 的依赖关系进行分解,如图 3(e)所示.分解之后的最长依赖环如图 3(f)所示,其延时之和为 11,循环体之差为 2,因此最小启动间距减小为 $recMII = \lceil 11/2 \rceil = 6$.

4.3 可回溯的节点调度过程

图 2 给出了 NOOI 算法中节点调度(Scheduling)的具体过程.给定启动间距Ⅱ,首先选取就绪节点 n,然后分别检测依赖约束条件和资源约束条件,判断是否发生节点冲突.若发生冲突,则使用依赖约束回溯模型(DBM:Dependence-constrained Backtracking Model)消解依赖约束冲突,使用资源约束回溯模型(RBM:Resource-constrained Backtracking Model)消解冲突资源约束冲突.重复以上过程直到所有节点完成调度.若调度次数达到最大值或节点冲突无法消解,则终止本轮的节点调度过程,增加Ⅱ进行下一轮调度.

图 4 中 *DBM_Backtracking* 函数给出了依赖约束回溯模型的具体实现.首先使用 *DBM_Profitable* 根据公式(1)进行"回溯可行性判断",若回溯收益为 0,则放弃当前节点的回溯过程;否则使用 *DBM_ChooseNode* 根据公式(2)进行"回溯节点选择",从已经完成调度队列中选择部分节点,取消它们的调度结果,并使用 *DBM_SetPriority* 根据公式(3)重新设置被取消节点的调度优先级;最后调用 *DBM_Reschedule* 再次对当前节点进行调度.由于回溯节点选择过程能保证消除当前节点的依赖冲突,因此 *DBM_Reschedule* 不会产生数据依赖冲突.资源约束回溯模型的具体过程也是相似的.

```
Bool DBM_Backtracking(Node n)
{
  if (! DBM_Profitable(n))
    return FALSE;
  NodeList L = DBM_ChooseNode(n);
  foreach (v in L)
    DBM_SetPriority(v);
  DBM_Reschedule(n);
  return TRUE;
}
```

图 4　依赖约束回溯模型 DBM 的具体实现

5 实验评估

实验环境基于 GCC-4.4.2 编译器[12]和北大众志 UniCore-2 微处理器平台[13].北大众志 UniCore-2 微处理器采用单发射、按序执行的八级流水线结构,并采用旁路(bypass)和互锁(interlock)等硬件方式解决数据相关问题,当相邻指令之间存在数据冒险或资源冒险时,流水线需要暂停若干周期.UniCore-2 微处理器具有 32 个通用整点寄存器,其中包含了 PC、LR 等特殊寄存器,本文限定模调度最大可用寄存器数量为 27.本文使用 Mediabench 作为评测程序,这是一个面向多媒体应用的基准测试程序,覆盖了数据压缩、图形编解码、音频编解码、加密解密等多个嵌入式领域,是目前常用的嵌入式系统评测程序之一.

本文将 NOOI 算法分别与传统的 SMS 算法(Swing Modulo Scheduling,SMS[3])和 Eric 等人近期提出的模调度算法(Eric's Modulo Scheduling,EMS[2])进行对比.SMS 算法是广泛应用到 GCC[12]等现代编译器中的模调度算法,该算法是一种寄存器压力敏感的模调度算法,其核心思想是通过指令排序和摆动调度实现减小寄存器压力和提高调度性能的目标.EMS 算法是针对嵌入式处理器的最新模调度算法,在遇到节点冲突时该算法仅取消一部分与当前节点发生冲突的后继节点,并重新进行当前调度过程.此外,为了控制调度时间,回溯因子设置为 3,即在单个启动间距条件下最多允许发生 $3N$ 次调度,其中 N 为循环体指令数.

5.1 调度成功率

图 5 给出了 SMS 算法、EMS 算法和 NOOI 算法的调度成功率,即不考虑寄存器压力情况下调度成功循环数占总循环数的比例.Mediabench 中共有 749 个最内层单基本块循环可用于模调度,SMS 算法因根本不进行回溯,调度成功率较低,而 NOOI 则能够成功完成约 98% 的循环调度.平均情况下,NOOI 算法调度成功率比 SMS 算法提高了 27%,比 EMS 算法提高了 8%.

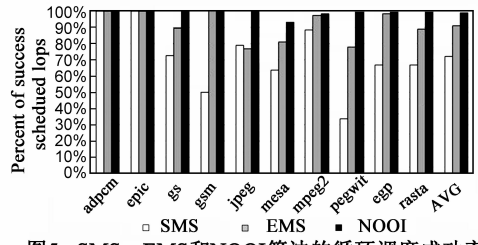

图5　SMS、EMS和NOOI算法的循环调度成功率

5.2 启动间距

启动间距也就是循环核心的长度,它决定了循环的性能.图 6 给出了 NOOI 算法相对于 SMS 算法和 EMS 算法对启动间距的改善情况.不同程序差别较大,其中 epic 和 mesa 等程序中有较多循环的循环启动间距获得改善,而 adpcm 和 rasta 等程序中启动间距获得改善的循环数目非常少,这主要是由于这些程序中单基本块最内层循环数量较少且依赖关系简单,EMS 算法和 NOOI 算法都能够达到最优值.相对于 SMS 算法,NOOI 算法能够使得 45% 的循环启动间距得到减小;相对于 EMS 算法,NOOI 算法能够改进 28% 的循环启动间距.

5.3 性能

图 7 给出了 SMS、EMS 和 NOOI 模调度算法相对于 GCC -O3 无环全局调度方法(baseline)的加速比.本文 NOOI 算法比 baseline 平均提高了 4.7% 的性能.相对于 SMS 算法和 EMS 算法,NOOI 算法分别提高性能 2.9%

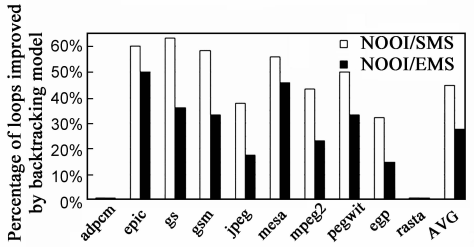

图6 NOOI相对于SMS和EMS改进循环启动间距的比例

和 1.4%.NOOI 算法能够获得更好性能,主要得益于其较高的调度成功率和较小的循环启动间距.从图中可以看到,gsdecode 和 mpeg 等部分程序的性能提升超过 10%,而 adpcm 和 jpeg 等部分程序性能变化很小,这一方面是由于启动间距和寄存器压力得到改善的循环数目较少,另一方面也是由于这些程序最内层单基本块循环执行时间占总体执行时间的比例较小.

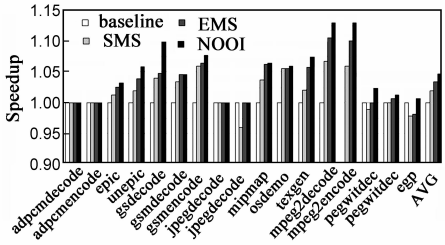

图7 EMS、SMS和NOOI算法的性能

本文也评测了各模调度算法的编译时间开销.相比于 GCC 全局无环调度算法,SMS、EMS 和 NOOI 算法的编译时间分别增加 1.4%、2.9% 和 2.3%,对总体编译时间影响很小.SMS 算法完全没有回溯,其调度时间开销最小.本文 NOOI 模调度算法每次取消的节点数更少,因此比 EMS 收敛的更快,时间开销也更小.

6 结论

本文针对嵌入式处理器特点,提出了一种面向嵌入式处理器的无重叠模调度算法 NOOI.该算法首次根据冲突发生的不同原因分别设计了优化的依赖约束回溯模型和资源约束回溯模型,并从回溯可行性判断、回溯节点选择和回溯节点重调度三个方面对回溯模型进行优化,以有效消解回溯型模调度算法中较多的节点冲突.实验结果表明,本文 NOOI 算法能够有效改进调度成功率并提高程序性能.

参考文献

[1] M Lam. Software pipelining: An effective scheduling technique for VLIW machines[A]. Int'l Conference on Programming Language Design and Implementation[C]. New York: ACM, 1988.318 – 328.

[2] J S Eric and L L Ernst. Modulo scheduling without overlapped lifetimes[A]. Int'l Conference on Languages, Compilers, and Tools for Embedded Systems[C]. New York: ACM, 2009.1 – 10.

[3] J Llosa. Swing modulo scheduling: a lifetime-sensitive approach[A]. Int'l Conference on Parallel Architectures and Compilation Techniques[C]. IEEE Computer Society, 1996.

[4] R Hongbo, et al. Single-dimension software pipelining for multidimensional loops[J]. ACM Transaction on Architecture and Code Optimization, 2007.4(1): 1 – 44.

[5] M C Josep, et al. A comparative study of modulo scheduling techniques[A]. Int'l Conference on Supercomputing[C]. New York: ACM, 2002.97 – 106

[6] 刘利,等.软件流水中隐藏存储延迟的方法[J].软件学报,2005.16(10):1833-1841.
LIU L, et al. Hiding memory access latency in software pipelining[J]. Journal of Software, 2005, 16(10): 1833 – 1841. (in Chinese)

[7] R A Huff. Lifetime-sensitive modulo scheduling[A]. Int'l Conference on Programming Language Design and Implementation[C]. New York: ACM, 1993.258 – 267.

[8] B R Rau. Iterative modulo scheduling: an algorithm for software pipelining loops[A]. Int' l Symposium on Microarchitecture[C]. New York: ACM, 1994.63 – 74.

[9] C Dulong, et al. An overview of the Intel® IA-64 compiler[J]. Intel Technology Journal, 1999.

[10] G S Tyson, et al. Evaluating the use of register queues in software pipelined loops[J]. IEEE Transaction on Computer, 2001.50(8): 769 – 783.

[11] 孙含欣,等.基于簇的寄存器堆功耗管理方法[J].电子学报,2008.36(2):278 – 284.
SUN H X, et al. Cluster-based power management mechanism for register files[J]. Acta Electronica Sinica, 2008, 36(2): 768-783. (in Chinese)

[12] M Hagog and A Zaks. Swing modulo scheduling for gcc[A]. Proceeding of the 2004 GCC Developers' Summit[C]. 2004. 55 – 64.

[13] X Cheng, et al. Research progress of unicore CPUs and PKUnity socs[J]. Journal of Computer Science and Technology, 2010, 25(2): 200 – 213.

作者简介

谭明星 男,1985 年生于湖南茶陵,北京大学信息科学技术学院博士研究生.主要研究方向为微处理器设计、软硬件协同设计和编译优化.
E-mail: tanmingxing@mprc.pku.edu.cn

刘先华(通讯作者) 男,1978 年生于湖北孝感,北京大学信息科学技术学院讲师,主要研究方向为计算机系统结构、编译优化.
E-mail: liuxianhua@mprc.pku.edu.cn