基于路径分组与数据相关松弛的软件流水*

容红波, 汤志忠

(清华大学 计算机科学与技术系,北京 100084)

E-mail: ronghb@mail.cic.tsinghua.edu.cn

http://www.tsinghua.edu.cn

摘要: 软件流水是循环调度的重要方法.有分支循环的流水依然是个难题.现有算法可以分为4类:循环线性化、路径分离、整体调度和路径选择.它们都未能和谐地解决两个对立问题:转移时间最小化和最差约束问题.提出了基于路径分组和数据相关松弛的软件流水框架,试图无矛盾地解决上述问题.其主要思想是:(1) 路径分组,即按照路径的执行概率和转移概率将路径分组,力求最小化转移时间;(2) 数据相关松弛,力求避免最差约束,即当循环有多条路径时,有些相关在循环执行中并不一定有实例,理想的策略是仅当它有实例时才遵守.初步实验和定性分析表明,此方法的时间效益优于路径分离和整体调度类算法.

关键词: 指令级并行;软件流水;路径;概率;数据相关

中图法分类号: TP311 **文献标识码**: A

通常,程序的大部分运行时间花费在一小部分代码上,循环就属于这一部分代码.有效调度循环将使整个程序的性能得以提高.

作为一种循环调度方法,软件流水[1-15]可以有效地开发程序中隐藏的指令级并行性(instruction level parallelism,简称ILP).它通过重叠不同循环体的执行来提高速度.在一个循环体尚未执行完毕之前,可以启动下一个循环体,二者之间的时间差称为启动间距(initiation interval,简称II).软件流水产生的调度结果由3部分组成:装入、核心和排空.在装入部分,循环体依次启动,直到出现一个重复模式(即核心)为止.核心以II为周期反复执行,直到所有循环体都被启动为止.排空部分完成剩余的已启动但未执行完毕的循环体.

一般而言,绝大部分执行时间花在核心上.由于核心重复一次需要II个周期,因此II表明了并行化的程度.寻找最小可能的II是软件流水的主要目标.

II受两个因素的影响.(1) 资源限制.显式资源(如功能部件)和隐式资源(如结果总线)有限,当II太小时,并发操作将引起资源竞争.(2) 相关限制.操作之间存在着控制和数据相关,即某个操作只有等到另一个操作结束后才能开始执行.当II太小时,操作之间的优先次序不能保证.

对于不含分支操作的循环(直线代码),考虑这两种因素已足以决定一个好的调度.因为循环每次执行的操作集合都一样.按照一个固定的II,就可以形成核心.这类算法已趋于成熟[1-3].

对于含有分支操作的循环,只考虑这两种因素是不够的.因为每个循环体都存在多条路径,每条路径的资源与相关约束都不一样.与直线代码不同,不同循环体的重叠不仅可能是同一路径的重叠(intra-path overlapping),也可能是不同路径的重叠(inter-path overlapping).一个循环体的最

* 收稿日期: 1999-11-15; 修改日期: 2000-01-25

基金项目: 国家自然科学基金资助项目(69773028)

作者简介: 容红波(1970-),男,陕西宝鸡人,博士生,主要研究领域为并行编译,体系结构;汤志忠(1941-),男,浙江宁波人,教授,博士生导师,主要研究领域为计算机并行算法,并行编译技术.

早启动时间取决于前面的循环体执行了哪些路径以及自己将执行哪条路径.即II受到另一种因素——不同循环体的重叠状态(执行过程中相邻循环体执行的是哪些路径)的影响.随着重叠状态的不同,II可以有所变化,这是一个值得开发利用的机会.当某些重叠状态出现时,可以使下一个循环体以较小的II启动,而不必是另一些状态下较大的II.难点在于如何使调度具有这种可变的自适应的II.

由于执行过程中出现哪种重叠状态是带有随机性的,静态调度难以预知,因此,我们不得不考虑概率.

本文认为,有两种概率,即路径的执行概率和路径间的转移概率,对于产生一个高性能调度是非常重要的.

应用概率,本文解决下述问题.

问题1(转移时间最小化).对于一个有分支循环,如果软件流水产生了多个核心,如何使核心间的转移时间最小化.

当循环中的各个路径被分离开,分别进行流水时会出现这个问题.在各个路径产生的核心之间,需要插入转移代码,因为相邻循环体可能执行不同路径.传统方法[3,16]由于核心间不规则的频繁转移,而使调度性能受损.虽然每条路径都被最紧凑地并行化了,但如果相邻循环体倾向于执行不同路径时,大量时间就消耗在并行度低的转移代码上.这样,一个在静态看来十分紧凑的调度在运行过程中却是低效的.

许多软件流水算法,尤其是一些模调度算法[4~8],还会遇到另一个更难、更常见的最差约束问题.

问题2(最差约束).对于一个有分支循环,如果有些路径严重受限于某些相关回路,而其他路径的限制较松,如何不分离路径而找到一个最优或近优调度.

当II是由受限最严重的路径决定时,就会出现这个问题.如果采用固定II,将使其他路径的并行化程度降低.

在不分离路径的前提下,解决这一问题的通用答案是松弛某些相关.控制相关通常是松弛的目标,即打断控制相关边,这称为猜测执行.有些相关回路可能因此而被打断,不再是回路,因而不继续对II产生影响.控制相关的打断意味着允许分支下的操作先于分支(猜测性地)执行.然而,猜测执行仅当该操作的结果变量不是另一条路径上的活变量,并且该操作不引起例外(exception)时进行.如果忽略这两个条件,就必须提供复杂的硬件支持,以便恢复某些被错误覆盖的变量值和处理例外[21,22].此外,并非所有相关回路都能通过去除控制相关而被打断.例如,一个只含有数据相关的回路就不能于借助猜测执行来打断.

为了系统地解决以上两个问题,本文提出一种新的软件流水框架——路径分组与数据相关松弛(path grouping and data dependence relaxation,简称PGDDR).其核心思想是:

(1) 路径分组.循环中的路径按照各自的执行概率和彼此间的转移概率分组,保证各组间的转移时间最小.这是为了解决问题1.

(2) 数据相关松弛.对于一个路径组,通过打断某些关键回路中的数据相关,最小化II,于是组内执行时间得以最小化.这是为了解决问题2.

由于循环的运行时间是由组间转移时间和组内执行时间决定的,最小化二者意味着一个近优/最优调度.这种方法的特色是:

(1) 基于概率对路径分组.既不是武断地将所有路径全部分开(像GURPR和APP那样),也

不是笼统地将它们全部合一(像 GURPR * 和 PNP 那样),而是按照具体循环的实际情况分组并行化.

(2) 是松弛数据相关而不是控制相关.相关一直被认为是必须遵守的关系.然而,对于有分支循环,有些相关并非在任何时候都有动态实例.理想调度应该是:仅当相关有实例时才遵守.本文做到了这一点.这是打断数据相关的首次尝试.

读者也许会认为概率难以统计,或者对用户来说太麻烦.我们部分地同意这个观点.但是,本文的方法仍是有价值的,因为:

(1) 正如某些循环没有确定的概率一样,另一些循环则有.我们相信,总是有许多影响程序整体性能的关键循环,需要最短执行时间.因此,用户值得花时间去寻找概率.

(2) 路径分组和数据相关松弛是本文所提出方法的两个独立部分.任何一个都可以独立应用到其他许多软件流水算法中去.即使不做路径分组,数据相关松弛依然可用.

本文第 1 节介绍相关工作,将现有算法分为 4 类.第 2 节以两个例子说明提出 PGDDR 的动机.第 3 节对 PGDDR 作了详细描述.第 4 节通过实验比较 PGDDR 与其他算法.第 5 节是一些有关的讨论.最后是结论.

1 相关工作

有许多软件流水算法用于并行化有分支循环.按照对路径如何处理,本文将这些算法分为 4 类:循环线性化、路径分离、整体调度和路径选择.

第 1 类,循环线性化(loop linearization),消除分支,将循环由多个基本块变为单个基本块.其代表是 Lam 算法[4]和带条件执行的模调度[5].优点是有分支循环被变成了直线代码,调度大为简化;缺点是可能造成时间和资源的浪费,概率也未被考虑,最差约束依然是个问题.

第 2 类,路径分离(path splitting),将所有路径分开,分别流水,然后在各个路径的并行化代码之间插入转移代码,将它们结合起来[9,10].在所有循环体都执行同一路径的情况下,效率将达到最大.然而,没有理由说明为什么相邻循环体应该执行同一路径.当各循环体执行不同路径时,大量时间将消耗在核心之间的转移代码上.实际上,它的隐含假设是路径间的转移概率低.因为只有在这种情况下,转移代码才不会占去过多的时间.最差约束被局部化,只对某些路径起作用.

第 3 类,整体调度(as-a-whole scheduling)却走了另一个极端,将所有路径放在一起进行调度[11~19].基于有向无环图的算法[11~13]将有环调度这个复杂问题简化成了无环调度,适于复杂控制流.但是,贪婪的猜测调度容易使 VLIW 指令被无用的猜测代码填满,代码复制较多.其他算法[16~19]受到最差约束的影响.这类算法实际上隐含了假设各路径间的转移概率比较高的事实.因为只有此时,相对于路径分离而言,它才节省了不必要的路径之间的转移时间.

第 4 类,路径选择(path selecting)以优化执行概率最高的路径为己任.将该路径与其他路径分开,对该路径施以模调度,对其他路径不流水[20].由于它基于路径执行概率,当各路径执行概率大致相同时,将难以决定优化哪条路径.此外,它只关注路径执行概率而不考虑路径间的转移概率,只优化一条主路径而不优化其他路径,这是不尽合理的.

表 1 对这些技术做了全面的描述.该表对于寻找改进机会或发明新算法都是有用的.

一个算法产生的调度可以从 8 个方面进行刻画:(1) II 是可变(自适应)的吗?(2) 需要显式的转移代码和时间吗?(3) 整个调度受最差约束影响吗?(4) (因分支重叠而)代码膨胀吗?(5) 基于概率吗?(6) 需要特殊硬件吗?(7) 是由核心识别类算法产生的吗?(8) 是由模调度算法产生的吗?

Table 1 Overview of current algorithms

表 1 当前算法一览

		Self-Adaptable II	Transfer code and time	Worst-Case effect	Code explosion	Prifile Based	Special hardware	Kernel recognition	Modulo
Linearization	Lam			√					√
	MS with if-conversion			√			Rotating reg file		√
	EMS			√	√				√
	Slack			√			Rotating reg file		√
	HRMS			√			Rotating reg file		√
Splitting	GURPR	√	√					√	
	APP	√	√					√	
As a Whole	DAG-Based: EPS, SS, SP-EPS	√			√				
	Unrolling-Based: RCSP	√			√			√	
	Unrolling-Based: RDLP	√			√			√	
	Petri net Based: Gao, PNP			√	√			√	
	GURPR*			√	√			√	
	GPMB			√	√		√	√	
Selecting	Superblock					√			√

表中的空白栏表示"No". 从表中可以看出:

(1) 所有循环线性化一类的算法,包括 Lam 算法[4]、带 If-转换的模调度算法[5]、EMS[6]、Slack Scheduling[7]及 HRMS[8]都面临最差约束问题. 除了 Lam 算法以外,都依赖 If 变换以去掉分支. 虽然控制相关被变成了数据相关,相关回路并不能因此而被打断,但所产生的调度具有固定 II,受最差约束影响.

(2) 路径分离类的算法,包括 GURPR[9]和 APP[10],使最差约束得以局部化,但面临转移时间最小化问题. 自适应 II 是通过简单地插入转移代码,将各路径的核心连接起来得到的. 虽然同一路径内重叠紧凑,但当路径间转移概率较高时,松散的路径间重叠损害了性能.

(3) 在整体调度算法中,基于 DAG 的算法,包括 EPS[11],SS[12]和 SP-EPS[13],不借助转移代码而得到了可变 II. 最差约束得以局部化,因为不同的路径可以按不同的 II 启动.

基于循环展开的算法,包括 RCSP[14]和 RDLP[15],产生可变 II,而面临代码膨胀和核心识别的问题.

基于 Petri 网的算法,包括 Gao's model[16]和 PNP[17],借助图论实现核心识别. 因为 Petri 网中的所有结点不得不形成一个强连通块,以便实现一个固定节拍,所以,不可避免地存在最差约束.

其他算法,包括 GURPR*[18]和 GPMB[19],试图展开循环,寻找重复模式. 通过从一个包含所有操作的代码框中删除冗余操作,强迫核心出现. 没有转移代码和时间问题,但是存在最差约束.

(4) 与其他算法不同,路径选择类算法、超块调度[20]是基于概率的. 由于主路径与其他路径分开,最差约束得以局部化. 但是,它对路径间的转移概率未加注意.

通过表 1 可以看出,当路径不被分离的时候,即当循环线性化或整体调度时,调度几乎注定受害于最差约束和固定 II. 反之,当路径被分开时,最差约束和固定 II 被避免,但需要转移代码和时间.

因此,最差约束问题(问题1)与转移时间最小化问题(问题2)是互相敌对的.当其中一个被解决时,另一个就变得显著起来.

为此,本文试图将这两个问题无冲突地系统解决.

2 动 机

下面以两个例子来说明提出PGDDR的动机.

图1中的循环有3个分支操作,分别判断输入字符是EOF、字母、数字还是其他字符.假如输入是随意的(经验说明,字母、数字是最容易出现的,其他字符次之),EOF只出现一次.此外,字母、数字交错出现的可能性也较大,其他字符交错的可能性较小.因此,既不应该将路径全部分离,也不应该整体调度而应并行化这个程序.一个比较合理的安排是:将处理字母、数字的两条路径作为一组,处理其他字符的作为第2组,处理EOF的作为第3组.每组分别流水.显然,从一组转移到另外一组的频率是很低的.虽然仍需要转移代码,但因为转移概率低,不会消耗过多执行时间.这个例子说明,按照路径的执行概率和路径间的转移概率进行分组并行化,将使静态编译的依据和动态执行的情况比较吻合,执行效率较高.本文由此提出"路径分组"的概念.

```
while ((c=getchar())! =EOF)
  if ('A'≤c&&c≤'Z')
        X1
    else if ('0'≤c&&c≤'9')
        X2;
      else
        X3;
```

Fig.1 An example

图1 例子

既然组间转移时间可以由分组保证最小化,那么组内执行时间呢?

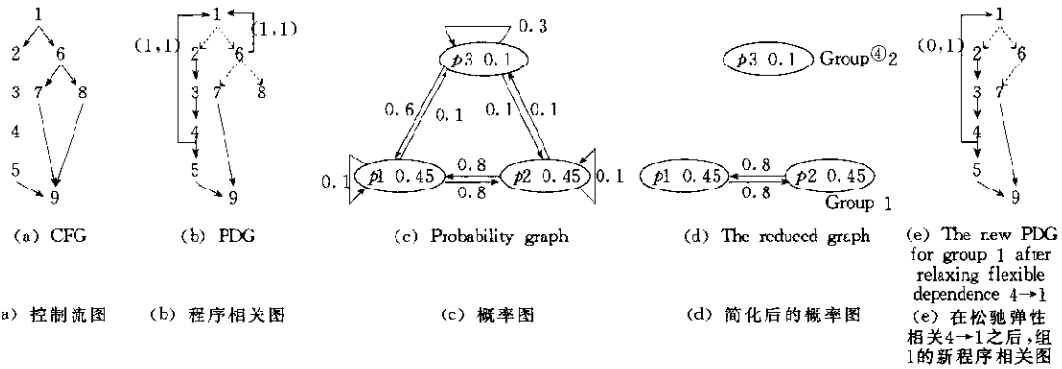
如果把一个组作为一个单独的循环,那么任何不分离路径的软件流水算法都可以将它并行化.其中的主要问题是如何处理最差路径.将该路径简单地从组中分离出去是不合理的,因为它是为了最小化转移时间才被放在组里的.

看另外一个例子(如图2所示).图2(a)是控制流图(control flow graph,简称CFG);图2(b)是相应的程序相关图(PDG:program dependence graph,or C&DDG:control and data dependence graph).其中,一些控制相关没有画出,隐含在传递性的相关链中.每条相关边都有一对标注(d,m),d是最小时延,m是相关距离.一条相关边记作$(O_1\rightarrow O_2,d,m)$,表示第$i+m$个循环体的操作O_2应在第i个循环体的操作O_1启动至少d个周期以后才可以启动;在无歧义时,简记作$O_1\rightarrow O_2$.

在图2(b)中,有两个相关回路.左边回路决定II至少为4,右边回路决定II为2.由于数据相关一直被认为是必须遵守的关系,对于现有的技术如Lam[4],EMS[6],HRMS[8],GURPR*[18]和GPMB[19]等等,如果控制相关1->2不能被打断的话,要想找到一个II小于4的调度是不可能的.

因此,II至少为4.但是,原来的串行循环执行一个循环体本来就只用4个周期,II≥4意味着这个循环不能并行化.

注意,数据相关4→1在循环执行过程中不一定出现,也就是说它不一定有实例.这是因为,操作4位于一个可能不被执行的路径上.只有当一个循环体执行了这条路径时,这个相关才可能有实例,才应该迫使下一个循环体在几个周期后启动.否则,就不存在这种限制.那么,我们为什么总要严格遵守这个相关呢?理想的策略是仅当相关有实例时才遵守.由此,本文提出了数据相关松弛的概念.像4→1这样可能有、也可能没有实例的数据相关称为弹性相关.它们在调度中被故意忽略,调度后通过修改调度而强迫遵守.结果是产生了一个具有自适应II的调度,而且最差约束被克服.



①控制相关,②数据相关,③注:未标注的边假定其标注为(1, 0),④组.

Fig. 2
图2

3 软件流水框架

软件流水框架共有 4 步:概率测试、路径分组、组内并行化和组间转移.

3.1 概率测试

图 1 的例子说明,对于有分支循环,各路径的执行概率和路径间的转移概率是影响调度性能的重要因素.概率测试的目的就是求出这两种概率,为路径分组提供可靠依据.

设经过测试,n 个相邻循环体分别执行下列路径:$t_{x1}, t_{x2}, \ldots, t_{xn}$,定义路径 t_i 的执行概率=t_i 在上述序列中的出现次数/n,记作 EP_i.定义路径 t_i 到路径 t_j 的转移概率=模式"t_it_j"在上述序列中的出现次数/t_i 在上述序列中的出现次数,记作 TP_{ij}.注意,TP_{ij}和 TP_{ji}的含义不同.

3.2 路径分组

分组的原则是使执行概率较高并且彼此间转移概率也较高的路径能处在同一组.

给定一个循环的概率图 $G=\langle V,E\rangle$,其中一个结点代表一条路径,每个结点向其他所有结点都射出一条有向边.每个结点用它所代表的路径及其执行概率标记,每条边用源结点到目的结点的转移概率标记.例如,图 2(a)所对应的概率图为图 2(c).

图 3 是路径分组算法.通过删除一些边,原来的概率图 G 就被精简成了另一个图 G'.新图中的每个强连通块就是一个路径组.

1. Delete from G all the arcs like $p_i \rightarrow p_j$, where p_i and p_j are two nodes (paths), $EP_i < \alpha$ or $TP_{ij} < \beta$, α and $\beta \in [0,1]$ are two thresholds of EP and TP respectively.
2. In the resulted graph G', find all the strongly connected components (SCC). The nodes (paths) in SCC compose of a group.

Fig. 3 Path grouping algorithm
图 3 路径分组算法

图 G'具有以下特性:

(1) 执行概率低的结点单独成一组.根据算法,由于它的 EP 小于阈值 α,因此它的所有射出边都被删掉.虽然它可能仍有进入边,但它已不可能与其他任何结点处在同一个强连通块了.这是合理的,因为它与其他路径放在一起会影响那些路径的调度,增大资源需求和相关限制,从而可能导

致 II 增大.现在将它单独作为一组,固然从其他组转移到它需要执行转移代码,但因为它很少执行,这种转移当然也不多,所以也不会占用太多时间.

(2) 彼此之间转移比较频繁的高执行概率结点,将位于同一个强连通块中.既然它们都有较高的执行概率,所以彼此间转移概率也较高,如果将它们放在不同组中,组间势必会出现频繁转移,占用许多时间.将它们归为一组,将消除不必要的转移代码,可能会显著缩短循环的总执行时间.

(3) 位于不同组中的两条路径 p_i 和 p_j,从 p_i 到 p_j 或者从 p_j 到 p_i 的转移概率一定比较低.这意味着组间转移时间被最小化了.

α 和 β 是算法分组的依据,它们的值可以在具体实现中估计.

算法虽然简单,却能有效地缩短总执行时间.这一点可以从下面的实验中看出来.

设 α 和 β 分别为 0.3,0.6,则图 2(c)的概率图 G 经过概率分组算法的处理,成为图 G'(如图 2(d)所示).

3.3 组内并行化

逐一处理所有的组.如果某个组只含有一条路径 p_i,则

(1) 如果路径 p_i 转移到自身的概率高于某一阈值 γ(如 0.7),那么,以路径 p_i 为循环体,对其实施任何一种软件流水算法.

(2) 否则,TP_{π}还不够高,即相邻循环体同时执行路径 p_i 的可能性不大.相应地,从这一组到其他组的转移将相对较多.如果对它实施软件流水,该组到其他组的转移代码将消耗许多时间,因此,对路径 p_i 只压缩,不流水.

若组中含有多条路径,由于我们不允许进一步分离路径,若要得到理想的并行化结果,最差约束问题必须解决.

3.3.1 数据相关松弛

正如第 2 节的例子所示,对于有分支循环,数据相关虽然必须遵守,却并非总会出现.这是一个重要的事实,提示我们数据相关可以在它没有实例时松弛,而在它有实例时才遵守.

然而,这看起来似乎很难,甚至不可能.因为相关何时有或没有实例是动态决定,而非静态所能预测的.然而问题并非这样悲观.例如,在图 2(a)和图 2(b)中,虽然我们不知道操作 4 将在哪个循环体内被执行,但我们确实知道,只在操作 4 出现的地方,弹性相关 4→1 一定有一个实例(注意,操作 1 是每个循环体必然执行的).

我们的想法是,首先将一个弹性相关从循环的相关图中删除(忽略),然后根据新的相关图对循环进行软件流水,最后,如果调度结果没有遵守被删掉的弹性相关,即含有一个错误,则修正调度结果,使之遵守该相关.在修正过程中可以发现该相关何时有、何时没有实例,从而仅在有实例时遵守它.后面的例子将说明这个过程.

真正的问题是怎样将相关松弛引起的错误局部化,并保证它易于改正.

通过仔细定义数据相关松弛的条件,并提出一个简单的修正错误的方法(称为下推变换),可以打断某些关键的相关回路,避免最差约束.最终的调度结果将具有自适应 II.

这个方法在文献[23]中做了详细描述.下面对其主要内容做简要回顾.

为方便定义起见,在循环体的 CFG 中,为第 1 个结点加入一个前驱,称作 entry;为所有无后继的结点加入一个后继,称作 exit.

在 CFG 中,任何从 entry 到 exit 的有向路径称为控制流路径,简称为路径,无论控制该路径的分支条件成立与否.

在 CFG 上,如果从 entry 到结点 i 的每一条路径都含有结点 d,则称结点 d 控制(dominates)结点 i,记作 d dom i. 对称地,可定义后控制(postdominate)的概念. 如果从 i 到 exit 的每条路径都含有结点 p,则称 p 后控制 i,记作 p pdom i[24].

两个结点 i,j,如果 i dom j,j pdom i,且 $i\neq j$,则称 i,j 是控制等价的(control equivalent).

如果操作 O 与 entry 控制等价,那么它是循环体必然要执行的操作. 一个相关$(O_1\rightarrow O_2,1,m)$,如果 O_1 与 O_2 中有一个不是必然要执行的,则此相关在循环的动态运行中可能存在、也可能不存在实例. 即它带来的限制随动态情况的不可而或有或无,因此可称之为有弹性的相关(flexible dependence).

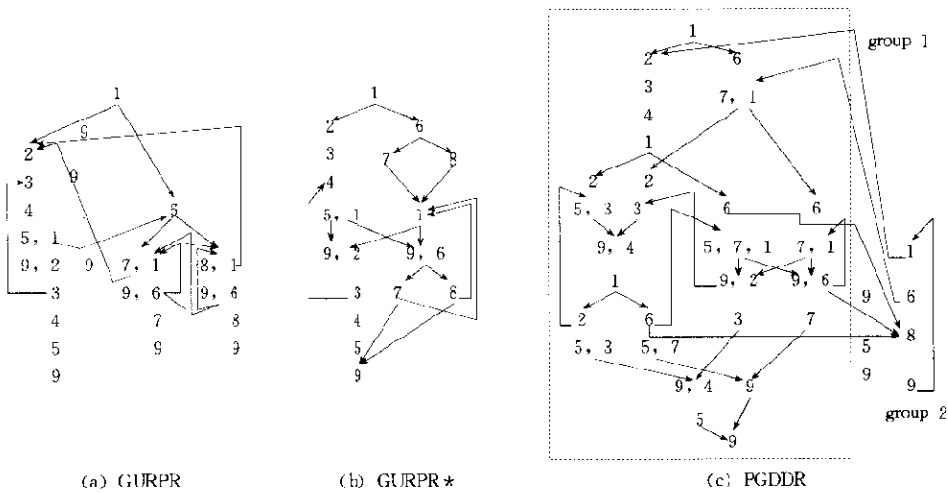
设有弹性数据相关$(O_1\rightarrow O_2,1,m)$,$m>0$,相关松弛是:将它暂时删掉,虚构一个相关$(O'_1\rightarrow O_2,0,m)$来代替. O'_1 具有如下性质:(1) 与 O_1 控制等价;(2) 在同一循环体中,一定在 O_1 之前被执行.

按松弛后的相关图进行调度,调度结果可能有错误. 设有弹性相关$(O_1\rightarrow O_2,1,m)$被松弛,调度后,如果 $O_2(i+m)$ 先于 $O_1(i)$ 启动,则调度有错,需要修正.

已经证明,如果调度含有错误,错误必然局限于该调度的某些子树上,并且易于修正. 修正后的调度保证具有新的重复模式.

例如,图 2(e)是数据相关松弛后第 1 组(路径 1 和路径 2)的新 PDG. 其中,原来的弹性相关(4→1,1,1)被虚拟相关(2→1,0,1)所代替. 因为从图 2(a)可以知道,操作 2 与操作 4 控制等价;从图 2(b)可以知道,有一条相关链 2→3→4,从而保证了在同一个循环体中,操作 2 总是在操作 4 之前调度. 这样就满足了相关松弛的条件.

根据新的 PDG,用软件流水对该组并行化. 所得调度被修正,以便遵守被忽略的弹性相关 4→1. 最终结果如图 4(c)虚框所示.



(a) GURPR (b) GURPR* (c) PGDDR

Fig. 4 Comparison of algorithms
图4 算法比较

可以说明,对该组而言,这个调度是可能达到的最好结果,并且其 Π 是自适应的. 当定义 Π_{ij} 为相邻两个循环体分别走 i 和 j 路径时,它们的平均启动间距[9]. 容易看出,$\Pi_{11}=\Pi_{12}-4$,而 $\Pi_{21}=\Pi_{22}=2$. 也就是说,只有当一个循环体执行左路径时,才要求下一个循环体等待 4 个周期后启动. 因为只有此时,弹性相关 4→1 有实例. 否则,下一个循环体可以在两个周期后启动. 显然,由弹性相关(4

→1,1,1)带来的最差约束问题已不存在.

3.4 组间转移

从一组转移到另一组需要插入转移代码,包括:(1) 本组的排空代码;(2) 转移目标组的装入代码;(3) 寄存器传递代码,将本组活出的寄存器的值传到恰当的寄存器中,以便转移目标组的代码能够正确访问.

由于这一步与每组所用并行化算法有关,在此不可能深入讨论.但一般来说,这一步是容易的.可参看文献[9,10]的例子.

4 例子与实验

这一节首先以图2程序为例,说明为什么PGDDR比路径分离和整体调度所得结果更好.然后给出初步的实验结果.

给定循环次数n,原串行循环的执行时间为

$T=3$条路径的执行时间总和$=EP_1*n*L_1+EP_2*n*L_2+EP_3*n*L_3=4.9n$,

其中L_i是路径i的长度,因此EP_i*n*L_i是路径i的总执行时间.

现在我们分别用GURPR(路径分离类算法),GURPR*(整体调度类算法)和PGDDR分别并行化该循环.在PGDDR中,为易于比较,选择GURPR作为组内并行算法.

定义TT_{ij}为从路径i所在组的核心转移到路径j所在组的核心所需要的转移时间.如果路径i和j在同一个组,则TT_{ij}为0.

对该循环使用GURPR,得到调度如图4(a)所示.此时,总执行时间为

$$\begin{aligned}T&=n*EP_1*(TP_{11}*II_{11}+TP_{12}*II_{12}+TP_{13}*II_{13})+n*EP_2*(TP_{21}*II_{21}+\\&\quad TP_{22}*II_{22}+TP_{23}*II_{23})+n*EP_3*(TP_{31}*II_{31}+TP_{32}*II_{32}+TP_{33}*II_{33})\\&=n*0.45*(0.1*4+0.8*5+0.1*5)+n*0.45*(0.8*3+0.1*2+0.1*2)+\\&\quad n*0.1*(0.6*3+0.1*2+0.3*2)=3.725n,\end{aligned}$$

其中$n*EP_i*TP_{ij}*II_{ij}$是当相邻两个循环体分别执行路径i和j时,路径i所用的总执行时间.在总执行时间中,路径间的转移时间为

$$\begin{aligned}TT&=n*EP_1*(TP_{11}*TT_{11}+TP_{12}*TT_{12}+TP_{13}*TT_{13})+\\&\quad n*EP_2*(TP_{21}*TT_{21}+TP_{22}*TT_{22}+TP_{23}*TT_{23})+\\&\quad n*EP_3*(TP_{31}*TT_{31}+TP_{32}*TT_{32}+TP_{33}*TT_{33})\\&=n*0.45*(0.1*0+0.8*2+0.1*2)+n*0.45*(0.8*3+0.1*0+0.1*0)+\\&\quad n*0.1*(0.6*3+0.1*0+0.3*0)=2.07n,\end{aligned}$$

其中$n*EP_i*TP_{ij}*TT_{ij}$是从路径i到路径j的转移时间.

虽然路径内的重叠程度达到了最大,但是转移代码占用了许多时间(2.07n),占总执行时间的55%.尤其是路径1、2之间的转移时间占到了总执行时间的48%,如下所示:

$n*EP_1*(TP_{11}*TT_{11}+TP_{12}*TT_{12})+n*EP_2*(TP_{21}*TT_{21}+TP_{22}*TT_{22})=1.8n$.

如果应用GURPR*算法,调度如图4(b)所示.此时,总执行时间为

$$\begin{aligned}T&=n*EP_1*(TP_{11}*II_{11}+TP_{12}*II_{12}+TP_{13}*II_{13})+n*EP_2*(TP_{21}*II_{21}+\\&\quad TP_{22}*II_{22}+TP_{23}*II_{23})+n*EP_3*(TP_{31}*II_{31}+TP_{32}*II_{32}+TP_{33}*II_{33})\\&=n*0.45*(0.1*4+0.8*4-0.1*4)+n*0.45*(0.8*3+0.1*3+\end{aligned}$$

$0.1*3)+n*0.1*(0.6*3+0.1*3+0.3*3)=3.45n$.

这里,路径1带来的最差约束使得II很大(注意,由于该最差约束调度中的II被固定为4,即对任意 i 和 j,$II_{ij}=4$.在消除一些空操作之后,II有点可变了,即 $II_{1*}=4$,而 $II_{2*}=II_{3*}=3$).但是,另一方面,在一半执行时间里,路径1并不出现(它的执行概率 EP_1 是45%).

对该循环应用PGDDR,得到如图4(c)所示的调度.执行时间大约为

$$T=n*EP_1*(TP_{11}*II_{11}+TP_{12}*II_{12}+TP_{13}*II_{13})+n*EP_2*(TP_{21}*II_{21}+TP_{22}*II_{22}+TP_{23}*II_{23})+n*EP_3*(TP_{31}*II_{31}+TP_{32}*II_{32}+TP_{33}*II_{33})$$
$$=n*0.45*(0.1*4+0.8*4+0.1*6)+n*0.45*(0.8*2+0.1*2+0.1*3)+n*0.1*(0.6*4+0.1*4+0.3*4)=2.835n,$$

其中,组间转移时间为

$$TT=n*EP_1*(TP_{11}*TT_{11}+TP_{12}*TT_{12}+TP_{13}*TT_{13})+n*EP_2*(TP_{21}*TT_{21}+TP_{22}*TT_{22}+TP_{23}*TT_{23})+n*EP_3*(TP_{31}*TT_{31}+TP_{32}*TT_{32}+TP_{33}*TT_{33})$$
$$=n*0.45*(0.1*0+0.8*0+0.1*2)+n*0.45*(0.8*0+0.1*0+0.1*1)+n*0.1*(0.6*5+0.1*2+0.3*0)=0.455n.$$

注意,组1具有一个可变II($II_{11}=II_{12}=4$,而 $II_{21}=II_{22}=2$),而这不是通过路径分离(像GURPR那样),或猜测执行(像EPS那样),而是通过数据相关松弛而达到.

现在我们来比较结果.对于上例,PGDDR产生的调度所用执行时间分别是原串行循环、GURPR和GURPR*的58%,76%和82%.时间减少的主要原因是:(1)彼此转移频繁的高执行概率路径(1和2)被放在了同一组中,而GURPR却将它们分开,致使一半时间花费在了低并行度的转移代码上;(2)通过数据相关松弛,产生了一个具有自适应II的调度,避免了路径1给路径2带来的不必要的最差约束,于是只要资源可用并且相关允许,循环体就可以启动,而不需要按照一个最差约束确定的固定II启动.而GURPR*却恰好陷入了这一问题.

假定资源不受限制,表1给出了一些初步实验结果.可以看出,PGDDR的时间效益一般比GURPR和GURPR*都好,转移时间总是比GURPR少,分别是GURPR转移时间的2%~71%不等.这正是由于概率分组将转移主要局限于本组的缘故.只有当各组所含路径数目很少(1~2条)时,PGDDR的转移时间才和GURPR接近.

另一方面,PGDDR需要更多的空间.因为在数据相关松弛产生一个更紧的调度的同时,它也产生了更多的分支组合.

Table 2 Preliminary experimental results

表2 初步实验结果

Examples①	Loops②	GURPR	GURPR*	PGDDR
1	4.8n/7	3.52n/35/0.32	3.8n/27	2.44n/42/0.62
2	6n/9	4.06n/42/0.43	4n/12	3.3n/63/0.01
3	5n/10	3.88n/58/0.59	2n/24	1.8n/61/0.22
4	7n/9	2.6n/52/0.61	2n/27	1.7n/57/0.18
5	5.6n/13	1.48n/131/0.32	n/284	1.3n/192/0.23

Each item: execution time/space requirement/percentage of transfer time③

①例子,②循环,③每一行:执行时间/空间需求/转移时间占总执行时间的百分比.

5 讨 论

PGDDR的主要目标是最小化转移时间和避免最差约束.通过概率分组和数据相关松弛达到

了这两个目的.

当每条路径的执行概率都比较低,或者彼此间的转移概率比较低时,每组只含一条路径,此时,PGDDR 算法变为路径分离算法.

另一个极端情况是,当每条路径的执行概率都比较高且彼此间的转移概率比较高时,所有路径都含在一个组里,此时,若没有数据相关松弛,PGDDR 算法就会变为整体调度.

由于基于概率测试,用户使用会比较麻烦,所以,对静态测试方法的研究还应该继续进行.对于那些难以确定概率的程序,其他方法如 EPS[10],可能更为适用.

数据相关松弛与控制相关松弛(猜测执行)并不矛盾.二者可以结合使用,以提高打断关键回路的可能性.

6 结 论

分支使循环的执行出现了随机性,因此,基于概率论的软件流水方法具有其本质上的合理性.影响软件流水效果的 3 种重要参数是路径执行概率、路径间的转移概率以及路径的资源需求和相关限制.若不充分考虑这 3 种因素,是不能作出合理的并行化决策的.与路径分离和整体调度方法不同,PGDDR 不是对所有程序作相同的假设,而是全面考虑这 3 种因素在特定程序中的实际情况,因此具有更强的适应性,更有利于产生高效代码.此外,最差约束问题得以避免.其中的两个主要技术,路径分组和数据相关松弛都可以独立应用到许多其他算法中.

References:

[1] Allan, V.H., Jones, R.B., Lee, R.M., *et al*. Software pipelining. ACM Computing Surveys, 1995,27(3):367~432.

[2] Calland, P.Y., Darte, A., Robert, Y.R. Circuit retiming applied to decomposed software pipelining. IEEE Transactions on Parallel and Distributed Systems, 1998,9(1):24~35.

[3] Govindarajan, R., Altman, E.R., Gao, G.R. A framework for resource-constrained rate-optimal software pipelining. IEEE Transactions on Parallel and Distributed Systems, 1996,7(11):1133~1149.

[4] Lam, M. Software pipelining: an effective scheduling technique for VLIW architectures. SIGPLAN Notices, 1988,23(7):318~328.

[5] Rau, B.R., Yen, D.W.L., Yen, W., *et al*. The cydra-5 departmental supercomputer: design philosophies, decisions, and trade-offs. Computer, 1989,22(1):12~34.

[6] Warter, N.J., Mahlke, S.A., Hwu, W.W., *et al*. Reverse if-conversion. SIGPLAN Notices, 1993,28(6):290~299.

[7] Huff, R.A. Life-Time sensitive modulo scheduling. SIGPLAN Notices, 1993,28(6):258~267.

[8] Llosa, J., Valero, M., Ayguade, E., *et al*. Modulo scheduling with reduced register pressure. IEEE Transactions on Computers, 1998,47(6):625~638.

[9] Su, B., Ding, S., Wang, J., *et al*. GURPR: A method for global software pipelining. In: Proceedings of the 20th Annual Microprogramming Workshop. 1987. 88~96.

[10] Stoodley, M.G., Lee, C.G. Software pipelining loops with conditional branches. In: Proceedings of the 29th Symposium on Microarchitecture. 1996. 262~273.

[11] Ebcioglu, K. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In: Gelernter, D., Nicolar, A., Padua, D., eds. Languages and Compilers for Parallel Computing. London: Pitman/MIT Press, 1989. 213~229.

[12] Moon, S.M., Ebcioglu, K. Parallelizing nonnumerical code with selective scheduling and software pipelining. ACM Transactions on Programming Languages and Systems, 1997,19(6):853~898.

[13] Shim, S.M., Moon, S.M. Split-Path enhanced pipeline scheduling for loops with control flows. In: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture. 1998. 93~102.

[14] Aiken, A., Nicolau, A., Novack, S. Resource-Constrained software pipelining. IEEE Transactions on Parallel and Distributed Systems, 1995,6(12):1248~1270.

[15] Novack, S., Nicolau, A. Resource-Directed loop pipelining: exposing just enough parallelism. The Computer Journal,

1997,40(6):311~321.

[16] Gao, G. R., Wong, W. B., Ning, Q. A timed petri-net model for fine grain loop scheduling. SIGPLAN Notices, 1991,26(6):204~218.

[17] Rajagopalan, M., Allan, V. H. Specification of software pipelining using petri-nets. International Journal of Parallel Programming, 1994,22(3):273~301.

[18] Su, B., Wang, J. GURPR*: a new global software pipelining algorithm. In: Proceedings of the 24th Symposium on Microarchitecture. 1991. 212~216.

[19] Tang, Z., Chen, G., Zhang, C., *et al*. GPMB — software pipelining branch-intensive loops. In: Kavanaugh, M. E., ed. Proceedings of the 26th Symposium on Microarchitecture. 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993. 21~29.

[20] Lavery, D. M., Hwu, W. W. Modulo scheduling of loops in control-intensive nonnumeric programs. In: Proceedings of the 29th Symposium on Microarchitecture. 1996. 126~137.

[21] Chang, M., Lai, F. Efficient exploitation of instruction level parallelism for superscalar processors by the conjugate register file scheme. IEEE Transactions on Computers, 1996,45(3):278~292.

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

[23] Rong, H. B., Tang, Z. Z. Flexible dependence and software pipelining. In: Werner, B., ed. Proceedings of the 4th International Conference/Exhibition on High Performance Computing in Asia Pacific Region. Los Alamitos, CA: IEEE Computer Society, 2000,1:250~252.

[24] Muchnick, S. S. Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997. 200~250.

Software Pipelining Based on Path Grouping and Data Dependence Relaxation*

RONG Hong-bo, TANG Zhi-zhong

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

E-mail: ronghb@mail.cic.tsinghua.edu.cn

http://www.tsinghua.edu.cn

Abstract: Software pipelining is an effective approach of loop scheduling. Pipelining of loops with conditional branches remains a challenge. Current algorithms are divided into four classes: loop linearizaton, path splitting, as-a-whole scheduling and path selecting. All of them fail to solve concurrently two conflicting problems: transfer time minimization and worst-case constraints. In this paper, a novel software pipelining framework is presented to do so. The key ideas are: (1) Path grouping, which splits paths into distinct groups based on their execution and transfer probability, so as to minimize transfer time; (2) Data dependence relaxation. Some data dependences may not have instances during execution when the loop has multiple paths. The ideal policy is to obey them only when they have instances. Thus the worst case constraints are avoided. Preliminary experiments and qualitative analysis all suggest that the temporal benefit of the proposed approach is better than that of path splitting and as-a whole scheduling.

Key words: instruction level parallelism; software pipelining; path; probability; data dependence

* Received November 15, 1999; accepted January 25, 2000

Supported by the National Natural Science Foundation of China under Grant No. 69773028