弹性数据相关与软件流水*

容红波， 汤志忠

(清华大学 计算机科学与技术系,北京 100084)

E-mail: ronghb98@mails.tsinghua.edu.cn

http://www.tsinghua.edu.cn

摘要: 最差路径是有分支循环软件流水的一大障碍.对于有分支循环,某些数据相关(称为弹性相关)在循环的动态执行中可能产生,也可能不产生实例.据此,可将严重限制并行性的弹性相关用限制较松的虚构相关代替,再进行软件流水.若调度没有遵守原来的弹性相关,则使用下推变换修正.从而缓解或者完全解除了最差路径的限制.该方法与经典的控制猜测互补,特点是允许调度含错,然后纠错.

关键词: 指令级并行;指令调度;软件流水;数据相关;循环

中图法分类号: TP338 **文献标识码**: A

软件流水[1~15]是一种循环调度方法,可以有效开发循环中的指令级并行性.在一个循环体(iteration)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).调度结果由装入、核心和排空3部分组成.在装入部分,前p个循环体以II为间隔依次启动,随之出现一个重复模式,即核心,其长度恰好等于II.它被反复执行直到所有循环体都被启动为止.离开核心后进入排空部分,完成最后p个已启动但未执行完的循环体.只要循环次数足够多,大部分执行时间将消耗在核心上.所以,软件流水的主要目标是寻找最短的核心,即最小的II.

限制II减小的因素有两种:(1) 资源限制.各种显式资源(如功能部件)和隐式资源(如结果总线)是有限的,当II小到一定程度时,无法解决资源竞争问题;(2) 相关限制.操作之间存在控制或数据相关,即某操作应该在另一操作执行完毕之后才可以启动.II小到一定程度时,操作之间无法保证先后顺序.这两种限制决定了最小启动间距(minimum initiation interval,简称MII).

对于无分支循环,资源和相关限制对不同的循环体都是一样的.用固定的II,就可能找到一个很好的调度[3].

对于有分支循环,问题就变得困难了.因为不同的循环体可以执行不同的路径,每条路径的资源和相关限制都与其他路径不同,如果调度采用固定II,则该II的值由最差路径决定,从而比较大.所谓最差路径是指资源要求最高或相关限制最紧的路径.显然,一个能随循环体的重叠情况而具有可变/自适应II的调度比一个固定II的调度性能会更好.

如何减轻最差路径所带来的限制是许多算法所面临的共同问题.

随着硬件的快速发展,资源会越来越丰富.相关限制将成为限制并行性的主要因素.在此假设下,本文解决下述问题:

---

* **收稿日期**: 1999-09-27; **修改日期**: 2000-03-23

基金项目: 国家自然科学基金资助项目(69773028)

作者简介: 容红波(1972-),男,陕西宝鸡人,博士生,主要研究领域为计算机并行编译,体系结构;汤志忠(1946-),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,并行体系结构.

© 中国科学院软件研究所 http://www.jos.org.cn

最差相关限制问题.给定一个具有分支语句的循环,如果MII由相关限制决定,并且一些路径严重受限于相关回路,而其他路径则不然,在不分离路径的情况下,如何找到最优或近优调度?这里,"最优"意味着任何循环体执行的任何路径都以最早可能时刻开始和结束.

由于MII决定于相关限制,所以松弛相关限制是通用答案.控制猜测(control speculation)就是这样一种技术.If下的操作O允许移动到If之前,抢先执行.这意味着从If到操作O的那条控制相关边从程序相关图(program dependence graph,简称PDG,或者control and data dependence graph)上被删除了.这个方法被一些算法,如EPS,SS,SP-EPS[1,9~11]大量使用.可是,只有当猜测移动的操作O不会覆盖If的另一个分支上的任何活变量,也不引起例外(exception)时,才可以进行控制猜测.如果忽略这些条件,就必须提供复杂的硬件以恢复被破坏的活变量和处理例外[16].另外,不是所有的相关回路都可以通过打断控制相关边而被打断,比如,一个只含有数据相关的回路.

如果没有硬件支持,不用控制猜测,不用路径分离,有可能找到一个最(近)优的调度吗?一个只含有数据相关的回路可能被打断吗?

作为回答,本文提出一种新方法:数据相关松弛(data dependence relaxation).这是解决本文问题的一个通用框架.

本文第1节通过一个观察和一个例子说明我们提出数据相关松弛的动机.第2节对本文方法做一概述.第3~5节分别详细介绍数据相关松弛、修正和核心识别.第6节给出另一个例子.第7节将该方法推广到松弛多个弹性相关的情况.第8节讨论相关工作.最后是结论.

1 动 机

因为发现小的II是软件流水的主要目标,首先我们来看如何计算II下界.

资源需求与相关限制分别决定一个II下界:ResMII(resource minimum initiation interval)和RecMII(recurrence minimum initiation interval).MII=Max(ResMII,RecMII),是II下界.在本文问题中已经假定,MII是由相关限制决定的,所以,MII=RecMII.一般地[3],

$$\mathrm{RecMII}=\max_{\forall\,\text{相关回路}C}\frac{\delta(C)}{d(C)}.$$

其中C是程序相关图上任一回路,$\delta(C)$是该回路上所有相关时延之和,$d(C)$是该回路上所有相关距离之和.具有最大的$\delta(C)/d(C)$的那些相关回路称为关键回路.注意:本文中,凡是回路都指的是相关图上的相关回路,而路径则指的是控制流图上的控制流路径.

在此公式中,如果(1)打断某些关键回路C;或者(2)降低最大的$\delta(C)/d(C)$,都将使RecMII减小.从而在RecMII起决定作用时,使MII减小.这正是相关松弛的最初想法,也是它的作用和优点.

一个关键回路可能既有控制相关,也有数据相关.如果控制猜测(打断控制相关)不可以,那么从传统的角度来看,是不可能打断这个回路或者说降低MII的.

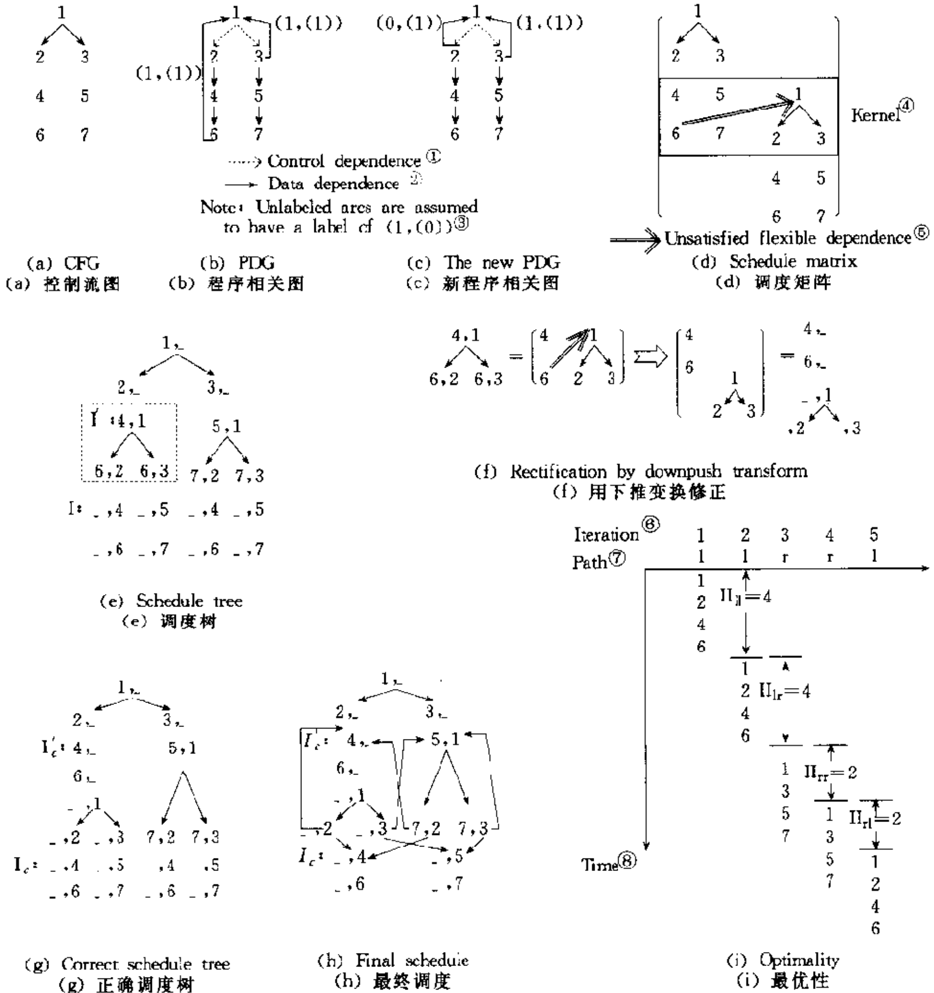
但是,如果控制相关是可以打断的,数据相关为什么就不可以呢?

通常,数据相关被认为是操作之间必须遵守的顺序关系.一个重要的事实是:在循环存在分支的情况下,数据相关虽然必须遵守,但并不意味着它在任何情况下都会出现.

图1(a)是一个简单循环的控制流图(control flow graph,简称CFG),为简明起见,那些显而易见的控制流关系(箭头)未标出.图1(b)是对应的程序相关图.注意:数据相关6→1在循环执行过程中不一定出现,即:它不一定有动态实例.因为操作6位于一条路径上,该路径不一定被执行.仅

© 中国科学院软件研究所 http://www.jos.org.cn

当一个循环体执行该路径时,这个相关才有实例,才应该迫使下一个循环体在几个周期后启动,反之则不必.那么,对这种在动态执行中或有或无的相关,在静态调度中为什么总要严格地遵守呢?理想的做法是,只有在它有实例的情况下,才应该遵守.



①控制相关,②数据相关,③注:未标注的边默认其标注为(1,(0)),④核心,⑤未遵守的弹性相关,⑥循环体,⑦路径,⑧时间.

Fig. 1 An example
图1 例子

下面通过调度图1的例子来说明我们的想法.

参看图1(b)的程序相关图.每条边上都有一对标注(δ,(d)),δ是相关时延,d是相关距离;下文用($O_1 \rightarrow O_2$,δ,(d))代表一条相关边,表示第$i+d$个循环体的操作O_2应比第i个循环体的操作O_1至少晚启动δ个周期(一个正确的调度必然保证这一时间关系.保证这一时间关系就是使该相关得到遵守,反之则未遵守).该相关可简写为$O_1 \rightarrow O_2$.这些记法与文献[18]一致.

图中共有两条相关回路,左边回路决定的RecMII是4,右边是2.因此,MII至少是4.但是,该循环执行一次本来就只占4个周期,II≥4意味着它无法并行化.

注意,在左边回路中,相关6→1不一定有动态实例.我们的想法是,用另一个对调度限制较松

的相关去代替这个相关,得到一个 II 较小的调度. 该调度可能并未遵守 6→1 这个相关,因此可能含有错误. 如果确有错误,那么,设法加以修正,最终得到一个正确的调度.

对于此例,虚构一个相关边(2→1,0,(1)),用它代替(6→1,1,(1)). 由此产生了一个新的程序相关图(如图 1(c)所示). 此时,左边回路的 RecMII 降至 1,比右边回路的还小,已经不再影响 MII 了.

不过,根据这个新相关图进行调度,所得结果(如图 1(d)和 1(e)所示)含有错误. 如虚框和箭头所示,第 1 个循环体的操作 6 位于第 2 个循环体的操作 1 之后,而按照原来的相关图,6 应该在 1 之前.

幸运的是,除了虚框部分有错以外,其余部分都是正确的. 将虚框中的这部分重新调度(如图 1(f)所示),得到正确的调度(如图 1(g)所示). 然后构造循环回边,得到最终结果(如图 1(h)所示).

可以验证,这是一个最优调度. 定义 II_{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]. 当 $i=j$ 时,II_{ij} 代表同一路径的重叠程度;当 $i\neq j$ 时,II_{ij} 代表不同路径的重叠程度. 给定一串相邻循环体,设它们分别执行某些路径. 用 l 表示左路径,r 表示右路径. 按此调度,可分析出各循环体的执行情况(如图 1(i)所示). 容易看出,$II_{l*}=4$,$II_{r*}=2$. 即只有当一个循环体走左路径时,它才要求下一个循环体滞后 4 个周期启动,因为只有在此情况下,弹性相关 6→1 才有实例;反之,下一个循环体可以在 2 个周期后启动. 这是此循环可能达到的最好的调度结果.

因此,我们提出如下命题:

命题:最差相关限制问题可以这样解决:

(1) 数据相关松弛. 选择关键回路中一个不一定存在实例的数据相关 A,构造另一个对调度限制较松的相关 A' 来代替,从而得到一个新的相关图.

(2) 软件流水. 根据资源限制和新相关图,用软件流水算法对循环进行调度.

(3) 错误修正. 在所得调度中,如果相关 A 未被遵守,则设法定位,并重新调度含错部分,使相关 A 得以遵守.

(4) 核心识别. 在修正后的调度中,识别重复模式,构造循环回边.

2 概 述

这一节概述解决最差相关限制的框架性方案,包括 4 步:数据相关松弛、软件流水、错误修正和核心识别. 下面对每一步做一简单解释.

2.1 数据相关松弛

在关键回路中,找到一个相关 A,A 具有这样的性质:它在执行过程中可能有也可能没有实例. 如果能虚构一个限制更松的相关 A'(下节说明如何构造),就将原来的相关 A 从程序相关图中删除,将 A' 加上去. 从而得到一个新的程序相关图.

2.2 软件流水

在资源限制下,根据新的程序相关图,对此循环应用软件流水.

任何软件流水算法都可以用,只要在每个循环体中操作的顺序一致即可. 也就是说,如果在第 1 个循环体的一条路径上,操作 O_1 在操作 O_2 之前,那么在其他所有循环体的同一条路径上,它也必须在 O_2 之前,虽然 O_1 与 O_2 的距离可以随循环体的不同而不同. 这个性质一般的软件流水算法都遵守.

© 中国科学院软件研究所 http://www.jos.org.cn

在这一步,软件流水算法肯定已经发现了核心(这是软件流水的任务),并且连上了循环回边.但是,这个调度可能有错误.即原来的相关 A 没有遵守,虽然其他所有相关,包括那个虚构的相关 A' 都被遵守了(正确的软件流水必然保证相关图上的所有相关都被遵守).因此,我们要修正错误.修正后,将产生一个新的核心,老的循环回边没有用.因此,在这一步可以不要循环回边.这样,调度可以用一个调度矩阵和调度树等价表示.对于第 1 个例子,调度矩阵和调度树如图 1(d)和 1(e)所示.

调度矩阵反映了这样一个概念:软件流水是不同循环体的重叠.假定软件流水共重叠了 u 个循环体,则矩阵有 u 列.每列代表一个循环体,每行代表一个时钟周期.矩阵中的空白表示"没有操作".

调度树是由指令与指令的顺序关系构成的.指令是由一个或多个操作组成的,每个操作将分配给不同的功能单元同时执行.在调度树中,各操作用逗号隔开,连在一起成为指令.其等价的矩阵表示是单行矩阵,如:1,_,_等价于[1_ _],其中"_"表示空白,即没有操作.

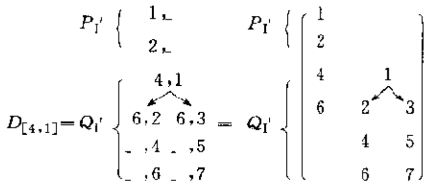


Fig.2 A maximum subschedule tree
图2 最大调度子树

调度树中的一条指令 I,从树根到达它具有惟一的一条路径 P_I,以它为树根的最大子树 Q_I 也惟一.由 P_I 和 Q_I 所组成的子树称为指令 I 决定的最大调度子树,简记为 D_I.

例如,在图 1(e)中,指令 $I'=[4\ 1]$决定的最大调度子树如图 2 所示.显然,$P_{I'}$只有线性列(只有一条路径的列),而 $Q_{I'}$可以有非线性列.

2.3 修 正

如果调度有错,则需要修正.方法是找出调度树中所有含错的最小部分,将它们分别重新调度.重新调度的目的是要遵守那个因为相关松弛而被删除的弹性相关 A,而仍然遵守所有其他已经得到遵守的相关.

我们关心的是重新调度是否简单,以及在重新调度后是否存在一个新的重复模式.

重新调度的方法可以找到许多种,但并不是所有方法都能保证简单性和重复模式的存在性.我们找到了一种方法,称为下推变换,可以保证这两个要求.

2.4 核心识别

可以证明,在用下推变换修正之后,在正确的调度树上,一定存在一个新核心,而且它和老核心密切相关.根据老核心的信息,可以直接发现新核心,然后就可以在调度树上连上循环回边.

注意:调度矩阵和调度树是指修正前的、没有循环回边的调度,其中可能含有错误.修正后称为正确的调度矩阵/调度树,此时仍然没有循环回边.加上循环回边后,称为最终调度.

2.5 假 设

本文方法的正确性依赖于如下 4 个假设:

(1) 循环模型:单重循环,可以含有分支操作.

(2) 资源模型:全流水线功能部件(每个周期可以接受一个新操作).

(3) 相关模型:任何相关的相关时延为非负整数.

(4) 软件流水算法保证:每个循环体中,操作的顺序一致(见第 2.2 节的解释).

© 中国科学院软件研究所 http://www.jos.org.cn

3 数据相关松弛

3.1 弹性相关

数据相关共有流相关、反相关和输出相关3种.反相关和输出相关是由于寄存器的复用引起的.通过软件或硬件的寄存器重命名可以消除.流相关一般是不可消除的.

本文对这3种相关不作区分,也不探讨反相关和输出相关的消除问题.而只认为,文中出现的相关都是必须遵守的.但是如前所述,这并不意味着它们在执行中总有实例.

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(注意:对一个循环的CFG,不画循环回边,所以文中所说的CFG都是有向无环图,参见图1(a)的例子).

在CFG中,任何一条从entry到exit的有向(无环)路径都叫做控制流路径(control flow path),简称路径(path),无论控制该路径的分支条件成立与否.

两个不同操作O_1,O_2,如果在任何一条路径上,它们要么都出现,要么都不出现,则称它们是控制等价的(control equivalent).

如果操作O与entry控制等价,那么它是每个循环体必然要执行的操作.一个数据相关$(O_1\rightarrow O_2,\delta,(d))$,如果$O_1$与$O_2$中有一个不是必然要执行的,则此相关是否存在动态实例是不一定的.即它带来的限制随动态执行情况的不同而或有或无,因此可称之为弹性数据相关(flexible data dependence).假定用$O(i)$表示第i个循环体的操作O.一般的调度是笼而统之地置$O_2(i+d)$于$O_1(i)$之后δ个周期,而理想的调度应该是:在此相关有实例时,$O_2(i+d)$一定在$O_1(i)$之后δ个周期,反之则不必.

如果弹性相关出现在某关键回路中,放松它的限制将使整个循环的调度结果得到改善.例如,在图1(b)中,弹性相关6→1在关键回路中.通过用虚构的相关2→1代替它,放松了限制,从而得到了一个II较小的调度(如图1(d)和1(e)所示).

3.2 数据相关松弛

设有弹性数据相关$(O_1\rightarrow O_2,\delta,(d))$,$d>0$,数据相关松弛是:将它从程序相关图中删掉,构造一个相关$(O'_1\rightarrow O_2,0,(d))$加上去,$O'_1$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.这个人为构造的相关称为虚构相关.删除了原来的弹性相关,并加入了虚构相关后的相关图称为新相关图.

例如,从图1(a)中我们知道,操作2与操作6控制等价.从图1(b)中我们知道,在同一循环体中,操作2总是在操作6之前调度,因为存在一个相关链2→4→6,其中每个相关边的相关距离都是0.因此,选定操作2来构造相关(2→1,0,(1)),以代替原来的弹性相关(6→1,1,(1)).

下面,我们解释为什么相关松弛应该具备上述条件.

(1) 按新相关图所得调度,不一定遵守那个被删除的弹性相关$(O_1\rightarrow O_2,\delta,(d))$.如果未遵守,就要对调度进行修正.如果$d=0$,那么,就要改变$O_2(i)$和$O_1(i)$的先后次序.就是说,同一循环体内的操作顺序发生了改变.这虽然可以,却不是我们所希望的.因此规定$d>0$,即只松弛体间数据相关.

(2) 当$O'_1(i)$在$O_1(i)$之前调度,并与之控制等价时,按新相关图进行调度,如果有错,错误将局限于若干子树上,并且容易修正.这是一个良好的性质.

© 中国科学院软件研究所 http://www.jos.org.cn

如果调度有错,错误只可能出现在这样一颗子树上:它的根指令含有 $O'_1(i)$,某一层(设为第 a 层)所有指令均含有 $O_1(i)$,某一层(设为第 b 层)的某些指令含有 $O_2(i+d)$,二者的关系是:$a+\delta>b$. 原因是:

(1) $O'_1(i)$ 在 $O_1(i)$ 之前调度,并与之控制等价,这决定了在调度树中,在且仅在出现了 $O'_1(i)$ 的所有路径上,出现 $O_1(i)$. 所以,在以含有 $O'_1(i)$ 的指令为根的子树中,必有某一层,其中所有结点均含有 $O_1(i)$;

(2) 一个层次对应一个周期,若有错误,弹性相关 $(O_1\rightarrow O_2,\delta,(d))$ 必然未被满足,所以 $O_1(i)$ 所在层次号 a+相关时延 $\delta>O_2(i+d)$ 所在层次号 b.

需要注意的是,"错误局限于上述子树"仍然是一个笼统的说法. 事实上,并非整个子树都错了,通常只是它的一部分错了.

图 3(a)是当 $\delta\leqslant1$ 时含错子树的一般形状:此子树的根部指令含有 $O'_1(i)$,所有第 a 层指令均含有 $O_1(i)$,中间第 b 层的某些指令含有 $O_2(i+d)$,$a\geqslant b$. 这个子树的某些子树有错(如虚框所示),而其他部分仍是对的. 例如,在图 1(e)中,错误局限于虚框圈定的子树中,其中 $O_1=6,O'_1=2,O_2=1$.

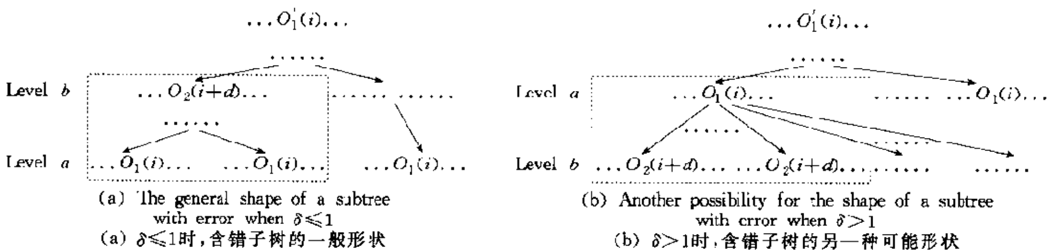
当 $\delta>1$ 时,含错子树的形状可能与 $\delta\leqslant1$ 时相同,也可能不同:虽然 $O'_1(i)$ 仍然含于根部,但是不一定有 $a\geqslant b$. 如图 3(b)所示,$a<b$,但是仍然有 $a+\delta>b$.

总之,无论形状如何,$a+\delta>b$,而且,只是该子树的某些子树有错(如图 3 中虚框所示),这些子树称为最小含错子树(minimum schedule tree with error,简称 MSE),下面给出精确定义:

假定 $(O_1\rightarrow O_2,\delta,(d))$ 是未遵守的弹性相关. 给定调度树中的一棵子树,如果满足下面两个条件,就称为最小含错子树:

(1) 它的根指令中含有 $O_2(i+d)$,所有叶子中含有 $O_1(i)$;或者根指令中含有 $O_1(i)$,而所有叶子中含有 $O_2(i+d)$;

(2) 无论上述哪种情况,设含有 $O_1(i)$ 的那一层是第 a 层,含有 $O_2(i+d)$ 的那一层是第 b 层,则 $a+\delta>b$.



(a) The general shape of a subtree with error when $\delta\leqslant1$
(a) $\delta\leqslant1$时,含错子树的一般形状

(b) Another possibility for the shape of a subtree with error when $\delta>1$
(b) $\delta>1$时,含错子树的另一种可能形状

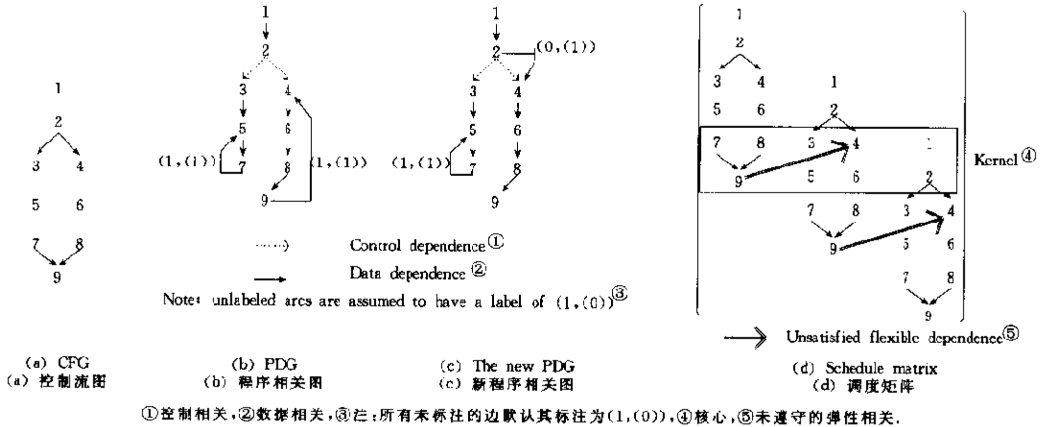
Fig. 3 The general shape of a subtree with error
图3 含错子树的形状

注意:虚构相关定义为 $(O'_1\rightarrow O_2,0,(d))$,其中时延为 0. 换言之,不允许 $O_2(i+d)$ 先于 $O'_1(i)$ 调度,但可以同时. 这样是对弹性相关尽可能地进行了松弛.

相关松弛的好处是:(1) 可以减小 RecMII. 例如在图 1 中,RecMII 从 4 减小到了 2. (2) 可以打断某些关键回路. 例如,在图 4 中,当弹性相关(9→4,1,(1))被虚构相关(2→4,0,(1))代替时,右边的回路,原来是关键回路,现在被打断了.

我们必须将相关松弛和冗余相关消除这两个概念区分开. 后者出现在某些算法如 DESP[2]中. DESP 分两步:第 1 步按资源无限情况进行软件流水,第 2 步删除某些数据相关,得到无环程序相

© 中国科学院软件研究所 http://www.jos.org.cn



①控制相关,②数据相关,③注:所有未标注的边默认其标注为(1,(0)),④核心,⑤未遵守的弹性相关.

Fig.4 Another example

图4 另一个例子

关图,然后按实际资源情况对第1步得到的调度进行修改.之所以删除一些数据相关,是因为第1步已经保证,在第2步中,即使它们不出现在程序相关图中,也一定能得到遵守.但是,在我们的方法中,相关被删除后,不能保证它得以遵守.如果恰好得以遵守,也只是出于偶然.相关被删除不是因为它冗余,而是因为这样可以产生一个更小的II.正因为如此,才需要错误修正.

4 错误修正

改正最小含错子树用下推变换.如图5所示.其中O_{ij}表示第i行第j列的操作,这与$O_i(j)$不同,后者是指第j个循环体的操作O_i.在图中,为方便起见,各列都画成了线性列.实际上并不限于此.未遵守相关用一条穿越第a行到第b行的箭头表示.为了修正,将第y列到第u列下推若干周期,使得该相关刚好被遵守即可(参见图1(f)的例子).

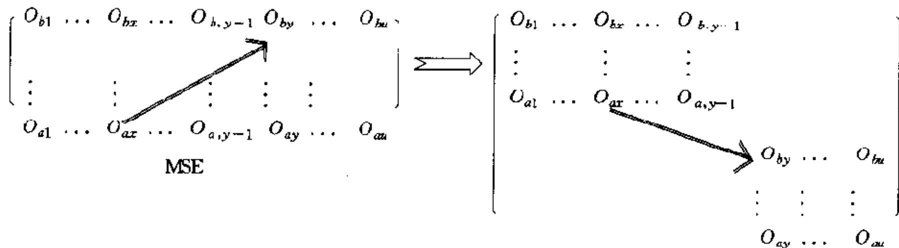


Fig.5 Downpush transform

图5 下推变换

下推变换具有如下性质:

性质1. 原来已经遵守的相关依然遵守.在修改之前,所有相关,包括虚构相关,均已被遵守,仅仅是那个被松弛的弹性相关未被遵守.对于任何一个已被遵守的相关($O_1 \to O_2 \delta,(d)$),按相关模型,在修正之前,必然满足$0 \leqslant \delta \leqslant \text{start}(O_2, i+d) - \text{start}(O_1, i)$,其中$\text{start}(O_2, i+d)$和$\text{start}(O_1, i)$分别是$O_2(i+d)$和$O_1(i)$的启动时刻.所以$\text{start}(O_1, i) \leqslant \text{start}(O_2, i+d)$,即$O_2(i+d)$被调度在$O_1(i)$之后.在此情况下,下推变换若影响此相关,只是将$O_2(i+d)$下推得更靠后,即$\text{start}(O_2, i+d)$增大.这样,仍然保证$\delta \leqslant \text{start}(O_2, i+d) - \text{start}(O_1, i)$,所以,相关仍然是遵守的.

性质2. 变换不需要考虑资源冲突.下推变换使每一周期同时执行的操作个数减少了,相应地,

© 中国科学院软件研究所 http://www.jos.org.cn

资源使用压力减少.因为资源模型是全流水线式的,可以认为每个周期所有资源都可用.因此,既然在变换前,每个周期操作个数较多的情况下都没有资源冲突,那么,在变换后,操作个数减少后就更不会有冲突了.

性质3. 一次变换将MSE的数目减少一个.对于整个调度树,可以用下推变换逐个修正其中的MSE.

5 核心识别

在修正之前,软件流水生成的调度树虽然有错,却一定有重复模式(核心)存在.

在修正过程中,下推变换并不考虑一个MSE是在核心内部、外部,还是与核心相交.那么,在变化之后的正确调度树上还能找到新的重复模式吗?如果不能,那么变换是不能用的.

幸运的是,可以证明,在上述变换下,一定能找到新的重复模式.

5.1 核 心

给定一个调度.进入核心后第1个周期执行的指令称为核心入口指令,退出核心前最后一个周期执行的指令称为核心出口指令;退出核心后,进入排空部分第1个周期执行的指令称为排空入口指令.

核心出口指令总是有两种后继,一种是核心入口指令,另一种是排空入口指令.这两种后继一一对应并且特征相似.给定一个排空入口指令$[\times O_{x2}O_{x3}...O_{xu}]$,一定有一个对应的核心入口指令$[O_{x2}O_{x3}...O_{xu}@]$,其中×表示该列所代表的循环体已经完全执行完毕,从此没有操作;@表示该列所代表的循环体从这个或更后面的周期启动,而在此之前没有操作.反之,给定一个核心入口指令,也一定有一个对应的排空入口指令.

给定两条指令$I=[\times O_{x2}O_{x3}...O_{xu}]$和$I'=[O_{x2}O_{x3}...O_{xu}@]$,其中×与@表示的含义同上,则称$I$与$I'$是相似指令(similar instructions).例如,在图1(e)中,I和I'是相似的.

回顾我们的假设:在每个循环体中,操作顺序都一致.因此,对于任何循环体,给定它的当前操作,就足以知道以后它要执行哪些操作.就是说,该操作代表了该循环体的状态.在上面的定义中,指令I的第1个循环体已经结束,后面的$u-1$个循环体和指令I'的最初$u-1$个循环体的操作都相同,因此,I和I'具有相同的后继状态,只不过从指令I'之后,又启动了一个新的循环体.所以,在I的直接前驱看来,如果在以后的周期里不启动新循环体的话,控制就应转移到I,否则应转移到I'.后一种情况意味着进入了重复模式.

5.2 相似定理与核心识别

设调度树中有一条指令$I=[O_{x1}O_{x2}...O_{xu}]$,在修正后的正确调度树中,$I$的第1个操作$O_{x1}$所在的指令$I_c=[O_{x1}...]$,称为指令$I$的修正版本(rectified version of instruction I).

相似定理. 调度树中的两条相似指令,它们的修正版本也相似.

证明:设调度树中,指令I和I'相似,它们的修正版本分别为I_c和I'_c.因为在调度树中,I只出现在子树D_I中,因此它的修正版本I_c完全由发生在D_I中的下推变换所决定.同理,I'_c由$D_{I'}$决定.现在我们来分析矩阵D_I和$D_{I'}$.

如图6所示.图中用波浪线标出的是指令I和I',它们分别是Q_I和$Q_{I'}$的根.Q_I和$Q_{I'}$可以有非线性列,但为了简单起见,都画成了线性列.这并不影响结论.

显然,D_I与$D_{I'}$基本相同.如果D_I没有第1列,$D_{I'}$没有最后一列,它们就是完全一样的.

© 中国科学院软件研究所 http://www.jos.org.cn

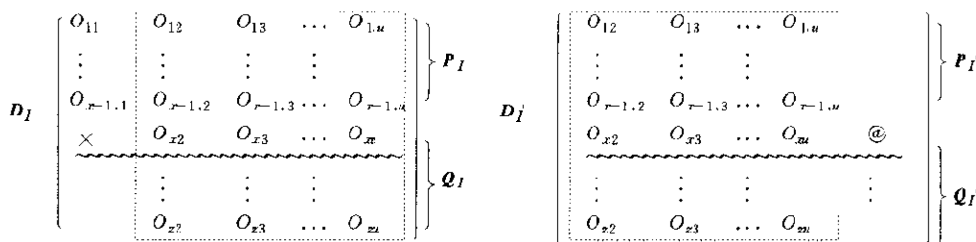


Fig. 6 The maximum subschedule tree determined by similar instructions
图6 相似指令决定的最大调度子树

要使 I 变化,它必须位于一棵最小含错子树中,或者说,有一个未遵守相关边穿越了 I. 这样,才会使它由于下推变换而分成两部分. 按照是否有这样一条边,分为两种情况:

(1) 如果 D_I 中有一条未遵守相关边穿越了 I,则它一定出现在 D_I 的第 2 列到最后一列(虚框部分). 因为从 I 所在的那一行开始,D_I 的第 1 列已经没有操作了,所以,第 1 列不可能射出一条未遵守相关边. 因为 D_I 和 $D_{I'}$的虚框部分完全相同,D_I 的虚框部分怎样变化,$D_{I'}$的虚框部分一定有相同的变化. 故在修正后,若 $I_c=[\times O_{x2}...O_{xy-}..._]$,其中 O_{xy}后有 $u-y$ 个空位置,一定有 $I'_c=[O_{x2}...O_{xy}\ \ ..._]$,其中 O_{xy}后有 $u-y+1$ 个空位置(注意:I'中的@已被下推到更晚的周期去了).

指令 I_c 的第 1 个循环体已经结束,后面 $u-1$ 个循环体与指令 I'_c 的前 $u-1$ 个循环体的操作相同. 这些循环体来自两个完全相同的虚框部分,经历了相同的变化. I_c 和 I'_c 的惟一区别是在 I'_c 之后,一个新循环体又启动了. 根据定义,I_c 与 I'_c 相似.

(2) 如果 D_I 没有射出穿越了 I 的未遵守相关边,那么,I 没有变化,即 $I_c=I=[\times O_{x2}O_{x3}...O_{xu}]$,而 I'的变化有两种情况:

① 没有从 $D_{I'}$ 的虚框部分的某操作射向@的相关. 此时,I' 也没有变化,即 $I'_c=I'$. 按已知条件,I 与 I' 相似,所以 I_c 与 I'_c 相似.

② 有从 $D_{I'}$ 的虚框部分的某操作射向@的相关. 此时,$I'_c=[O_{x2}O_{x3}...O_{xu}_]$(注:@已被下推到更晚的周期去了),显然,$I_c$ 与 I'_c 仍相似.

例如,在如图 1(e)所示的调度树中,I 和 I' 相似. 在正确调度树中(如图 1(g)所示),它们对应的修正版本 I_c 与 I'_c 也相似.

按前面所讨论的,在正确调度树中,如果令控制从 I_c 的直接前驱转向 I'_c,就进入了重复模式(参看图 1(h)的例子). 加入一条新边来代表这种转移,这条边就是循环回边. 所有循环回边都可以这样构造. 因此有以下推论:

推论. 修正后,正确调度树一定有重复模式.

这是非常重要的结论. 这表明,相关松弛和下推变换能够保证产生一个新的重复模式,而且根据相似定理,新的重复模式与含错调度中的老重复模式是有联系的.

6 例 子

现在给出另一个例子.

循环的 CFG 如图 4(a)所示. 在程序相关图(如图 4(b)所示)中,右边的回路是一个关键回路. 它决定了 MII=4. 注意,控制猜测不能打断此回路,因为该回路中没有控制相关.

现在松弛该回路中的弹性相关(9→4,1,(1)). 注意,操作 2 与操作 9 控制等价,并且从相关链

© 中国科学院软件研究所 http://www.jos.org.cn

2→4→6→8→9 可知,在同一循环体中,操作 2 在操作 9 之前出现.因此选择操作 2,虚构一个相关(2→4,0,(1))以代替原来的相关(9→4,1,(1))(如图 4(c)所示).可以看出,右边的回路被打断了,图中只有一个回路.

给定充足的资源,从这个松弛后的相关图可以得到一个调度(如图 4(d)所示).然后,在调度树上,用下推变换逐一修正每个最小含错子树;寻找相似指令,构造循环回边,得到最终调度.由于调度树太大,限于篇幅,不再画出.

7 松弛多个弹性相关

回顾我们的结论来自 4 个假设:单重循环;全流水线式功能部件;相关时延非负;每个循环体中的操作顺序都相同.一次下推变换后,调度仍然满足这些假设.因此可以直接将我们的方法推广到松弛多个弹性相关的情况.

根据相关松弛的条件,找到一些弹性相关,将它们删除,加入相应的虚构相关.然后根据新相关图进行软件流水.选择任何一个未遵守的弹性相关,修正调度树以遵守之.过程如前所述.然后选择另一个未遵守的弹性相关,再次修正调度树.如此反复,直至所有未遵守的弹性相关都得到选择和修正.

因为所有的假设在每次下推变换后仍然成立,新核心一定是存在的,所以,最后通过寻找相似指令,构造出循环回边.

8 相关工作

有分支循环的软件流水算法可大致分为如下 5 种:

第 1 种方法.消除分支.将循环体由多个基本块变为单个基本块.其代表是带条件执行的模调度[5].这种算法的优点是将循环体由分支结构变成了无分支结构,调度大为简化.缺点是造成了时间和资源的浪费.

第 2 种方法.分离循环体的所有路径,对每条路径分别进行软件流水,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[7,8].在所有循环体都执行同一路径的情况下,效率将达到最高.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.

第 3 种方法.将所有路径放在一起调度[6,9~13].基于有向无环图的算法[9~11]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,不利于产生高效和紧凑的核心.其他算法[6,12,13],其 II 将取决于资源和数据相关限制最严重的那条路径,惩罚其他所有的路径.

第 4 种方法.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,对其他路径不进行流水[14].这种方法是路径调度[17]的变体.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.

第 5 种算法.按路径的执行概率和转移概率将路径分组[15].彼此转移比较频繁的路径将被归类到同一组,组内使用以上任何一种方法进行软件流水.各组间插入转移代码.这种方法使路径间的转移时间达到了最小化.不过,为了使组内代码的执行时间也最小化,需要处理那些严重限制 II 的相关回路,这是一般算法都要面临的问题.事实上,本文的方法就是为了解决这一问题而提出

© 中国科学院软件研究所 http://www.jos.org.cn

来的.

9 结 论

分支给循环的调度带来了许多困难,弹性数据相关即是其中之一.它不一定有动态实例,静态调度时却不得不遵守.对于现有的软件流水技术,这通常是一个障碍.

然而,任何问题都有其两面性.弹性数据相关不仅妨碍了并行性,它也蕴涵着并行性,具有可挖掘的新特点.数据相关松弛正是看到了问题的另一面,从而提出了一个崭新的思路:先放松限制,再修正错误.通过(1)将控制等价和数据相关结合起来考虑,(2)抛弃一步到位的调度方式,采取 try-and-catch 的方法,(3)抛弃传统的去除分支的思想,利用分支,使错误局部化,成功地将时间最优化问题转化成了空间最优化的问题,即将寻求最优(时间)调度的问题转化成了如何在实际中抑制代码膨胀的问题.

在某种意义上,相关松弛是控制猜测的对应物:控制猜测是操作不服从控制相关,而相关松弛是操作不服从数据相关.文献[18]提示:将控制猜测和相关松弛结合起来,可以同时优化时间和空间(既解除最差相关限制,又无代码膨胀),使相关松弛可应用到实际的多分支循环中去.

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] Rau, B.R., Fisher, J.A. Instruction-Level parallel processing: history, overview and perspctive. The Journal of Supercomputing, 1993,7(1/2):9~50.

[5] Dehnert, J.C., Towle, R.A. Compiling for the Cydra-5. The Journal of Supercomputing, 1993,7(1/2):181~227.

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

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

[8] 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.

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

[10] 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.

[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] 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. Los Alamitos, CA: IEEE Computer Society Press, 1993. 21~29.

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

[14] 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.

© 中国科学院软件研究所 http://www.jos.org.cn

[15] Rong, Hong-bo, Tang, Zhi-zhong. Path grouping and data dependence relaxation for software pipelining. In: Feng, Yu-lin, eds. Proceedings of the World Computer Congress (WCC). Software: Theory and Practice. Beijing: Electronics Industry Press, 2000. 779~788.

[16] 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.

[17] Fisher, J. A. Trace scheduling: a technique for global microcode compaction. IEEE Transactions on Computers, 1981,C-30(7):478~492.

[18] Rong, Hong-bo. Software pipeling of nested loops [Ph.D. Thesis]. Tsinghua University, 2001 (in Chinese).

附中文参考文献:

[18] 容红波.多重循环软件流水算法研究[博士学位论文].清华大学,2001.

Flexible Data Dependence and Software Pipelining*

RONG Hong-bo, TANG Zhi-zhong

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

E-mail: ronghb98@mails.tsinghua.edu.cn

http://www.tsinghua.edu.cn

Abstract: For software pipeling of loops with conditional statements, the worst-case path is a great abstacle. In such a loop, some data dependencies called flexible dependencies, may or may not have instances during execution of the loop. From this fact, flexible dependencies that severely limit parallelization of the loop are identified and replaced with less tight virtual dependencies. Then software pipelining is applied. If the schedule does not satisfy the original flexible dependencies, downpush transform is used to rectify. The resulting schedule is partly or completely free of the worst-case effects. This approach is a complement to the classical control speculation. It is characterized by a try-and-catch way: errors are first allowed to be present in the schedule, and then rectified.

Key words: instruction-level parallelism; instruction scheduling; software pipelining; data dependence; loop

---

* Received September 27, 1999; accepted March 23, 2000

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