顺序扫描实现程序并行化*

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: 提出扩展选择调度,统一处理循环和非循环代码,对它们不加区分但却分别产生软件流水和全局压缩的效果;程序并行化不需要分层简化,只要顺序扫描一遍即可.该方法打破了有环调度和无环调度的界限,是一种基于一般图而不是路径或有向无环图的全局调度算法.它从一个全新的角度来看待多重循环,通过恰当地计算可用集合和活变量集合,实现了多重循环的直接调度,对任意控制流程序都是适用的.

关键词: 指令级并行;全局压缩;软件流水;分支;多重循环

中图法分类号: TP338 **文献标识码**: A

指令级并行(instruction level parallelism,简称 ILP)从 20 世纪 80 年代开始成为推动计算机发展的显著力量.VLIW(very long instruction word)和超标量是 ILP 处理器的典型例子,而代码调度是 ILP 编译研究的热点,以便为这些处理器充分开发程序中的并行性.

代码调度分为局部和全局调度(local/global scheduling).局部调度仅处理无环基本块,在 70 年代已得到深入研究,一般可达到近优解.因此它已不是研究重点.

全局调度处理有分支的程序(多个基本块).它分为全局无环调度(global acyclic scheduling)和全局有环调度(global cyclic scheduling)两大类.无环调度常被称为全局压缩,它对相邻基本块进行重叠,又分为基于路径的技术(trace-based scheduling)和基于有向无环图的技术(DAG based scheduling)两类[1].基于路径的技术包括路径调度(trace scheduling)及其变体,如超块调度(superblock scheduling).基于有向无环图的技术包括选择调度(selective scheduling)[2]、渗透调度(percolation scheduling)[1,3]及其变体,如现实调度(realistic scheduling)[1].有环调度对相邻循环体进行重叠,包括循环展开和软件流水,其中软件流水是当前研究的焦点.

实际程序一般既有循环部分,又有非循环部分.非循环代码需要全局压缩,循环代码需要软件流水.目前的做法都是先对循环分层流水,再和非循环代码一起压缩.本文力图统一处理这两种代码,对它们不加区分但却分别产生全局压缩(基本块重叠)和软件流水(循环体重叠)的效果,且不降低性能.

难点在于含有任意分支的多重循环的软件流水.目前,大多数软件流水技术集中于单重(最内层)循环,对多重循环进行流水的工作很少.而在实际中,多重循环更为常见.因此,多重循环的高效并行化具有更普遍、更现实的意义.

但是,这一工作是富有挑战性的.其根本的制约在于分支和回路.多重循环从本质上讲是有回路的多层次分支结构,每一层次又可能含有分支.分支的存在,使得基本块的尺寸小,执行路径不唯一,严重影响了体内压缩和体间并行.即使对于单重循环,软件流水尤其是模调度类的软件流水技术,一直为分支的存在所困扰.在多重循环的情况下,多回路的存在使问题进一步复杂化了.

由于这个原因,目前的技术将多重循环转化为单重循环进行流水.其通用模式是:先对最内层循环进行流

* 收稿日期: 1999-05-17; 修改日期: 1999-09-23

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,主要研究领域为并行编译,并行体系结构;汤志忠(1946—),男,江苏海门人,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.

© 中国科学院软件研究所 http://www.jos.org.cn

水,完成后,装入和排空部分将自然地融入外层循环,而新循环体将被视为外层的一个不可分割的原子操作.这时,外层循环是一个单重循环,可以继续流水,融入更外层.如此反复进行,直至整个多重循环得以处理.我们称这种由内层向外层扩展的方法为外向流水(outward pipelining).几乎每一种已知方法都属于这一类,包括层次式削减(hierachical reduction)、GURPR(global unrolling pipelining rerolling)、增强型流水线调度(enhanced pipeline scheduling,简称 EPS)[1,3]、选择调度[2]、流水对接(pipelining-dovetailing)[4]、内外层循环交错流水(interlaced inner and outer loop software pipelining,简称 ILSP)[5]等.它们将单重循环流水技术简单地扩展到多重循环,思想显然是正确的,但是对性能几乎没有报道.

外向流水的合理性在于:(1) 先得到内层的装入排空部分,就能将外层代码和它们压缩在一起,从而提高并行度;(2) 迄今为止还没有直接对多重循环流水的成熟技术.本文提出的扩展选择调度(generalized selective scheduling,简称 GSS)可解决任意多重循环的直接调度问题.它适用于任意程序,具有足够的理论基础.它不是由内而外分层简化,而是顺序扫描一遍.

扩展选择调度对选择调度的扩展在于:放松了对输入的限制.输入不限于有向无环图(directed acyclic graph,简称 DAG),而可以是任意有向图,可以无环,也可以有环,只要图的根结点不在环路中即可.其意义表现在如下几个方面:

(1) 它是首次基于一般图的全局调度算法.GSS 的输入特点决定了它不属于纯粹的无环或有环调度.

(2) 由于输入允许任意环路,将它用于软件流水时,可以自然地处理多重循环,成为一种通用的流水方案.

(3) 将非循环代码与循环统一看待,程序的并行化只需从头到尾扫描一遍.通过仔细计算可用集合和活变量集合,GSS 在内层循环未进行流水之前,就将其可用操作提到了外层用于压缩,这相当于提前计算内层流水的装入部分.这种提前的贪心计算与压缩是扩展选择调度保证顺序扫描和性能的关键.

1 选择调度的简单回顾

选择调度[2]是基于有向无环图的全局调度算法,其作用是在控制流图上将无关操作聚集在一起.其特点是:

(1) 计算可用右手(right-hand sides,简称 RHS)的集合,而不是整个操作的集合;

(2) 穿过所有执行路径计算可用集,允许穿过分支的两路目标进行代码移动;

(3) 用于软件流水,跨循环体计算可用集;

(4) 代码移动后,增量式重新计算可用集.

选择调度的输入是一个有根 DAG.在根结点之前放置一个空并行组,该组与一定的资源约束相关联,选择调度的目的是用该 DAG 上无关的、可执行的操作,将该并行组填满.为此,反复进行以下两步,直至并行组由于资源限制或数据相关限制而不能再填充.

① 计算.计算所有可移入该并行组的可用右手的集合,从中选择最佳右手.

② 代码移动.将最佳右手移动到并行组中,并赋值给一个合适的目标寄存器.

在上面两步中,并行组 group 的可用右手集 av(group)通过增量式计算 DAG 中每个结点 n 的可用右手集 $av(n)$而得到.选择最佳右手需要对 av(group)中的每个右手 Γ 计算猜测度 spec(Γ),即 Γ 从原来位置移动到并行组中所跨越的分支的最大个数.选择合适的目标寄存器则需要对每个结点 n 计算活变量集 $lv(n)$.这 3 个集合是选择调度的依据.

在处理多重循环时,这种基于无环图的调度有其固有的弱点.为了使外层循环满足无环的要求,内层循环的核心不得不被看成是一个原子操作,其内部代码对外不可见.当外层代码较多时,内外层代码不能充分融合,失去了更进一步的并行性.

2 扩展选择调度

当资源较为丰富,而外层代码自身的并行性不足时,基于有向无环图的调度难以充分开发外层并行性的弱点更加明显.为了克服这一弱点,扩展选择调度将输入扩大到一般的有向图,它可以无环,也可以有环.

© 中国科学院软件研究所 http://www.jos.org.cn

扩展选择调度具有上述选择调度的4个特点.此外,还具有其他一些特点:

(1) 输入是一般的有向图,只要根结点不在环路中;

(2) 在对外层进行全局调度的过程中,自动对内层进行流水或部分流水.

(3) 用于软件流水,内层的代码对外是可见的,内层不是不可改变的超指令,而是仍可向外移动的代码.

在算法上,扩展选择调度与选择调度在计算方面有所不同(计算是算法的关键),而在代码移动方面基本相同(各种全局调度在这方面都比较相似).

2.1 预处理

在并行化之前,需要先建立控制流图(control flow graph,简称CFG).在CFG上依据两个条件识别循环:(1) 具有唯一的入口结点(header),所有从循环外到达循环内任一结点的路径都要经过它;(2) 是强连通的.如果循环L的一个子集是循环,则称L为嵌套循环(nested loop),即多重循环.

对于循环中的一个结点,如果它的一个后继结点不在循环中,则称它为退出结点(exit),称这个后继结点为循环退出后的首结点(postexit),从exit到postexit的边称为退出边(exit edge).为了调度的需要,对每个循环移入一个辅助结点,称为preheader.preheader的后继是header,所有原来从循环外进入header的边(不包括循环回边)改为进入preheader.为了指出循环结束后控制的转移方向,从preheader向此循环的所有postexit连一条有向边,为区别起见,这种边用虚有向线表示,而其他边用实有向线表示,在后面的公式中分别写作$n\cdots\gg x$和$n\rightarrow x$.

preheader唯一地代表了一个循环.preheader与它射出的所有有向线构成了循环的缩影.preheader射出的实有向线的目标就是循环的入口结点header,它指出进入循环后第1个要执行的结点;虚有向线的目标则是退出结点exit的后继,它指出退出循环后第1个要执行的结点.退出结点可能不止一个,因此一个preheader可能射出多条虚有向线,但实有向线只有一条.当循环是死循环时,preheader不射出虚有向线.

显然,任何经过预处理的CFG一定是有根的有向图,且根结点不在环路中,因而满足扩展选择调度的输入要求.

如果$n\cdots\gg x$或$n\rightarrow x$,则称n是x的前驱.如果结点的前驱多于1个,或者结点是一个条件分支的目标结点,则称该结点是基本块的头bb_header(basic block header).按此定义,一个preheader的所有后继x都是bb_header,因为无论是$preheader\cdots\gg x$还是$preheader\rightarrow x$,x的前驱除了preheader,还有其他的.为统一处理起见,将preheader也作为一种特殊的bb_header.

为所有bb_header计算正确的活变量集lv,其中$lv(\text{preheader})=lv(\text{header})$,作为初始值.

2.2 扩展选择调度的问题描述

给定一个经过上述预处理的CFG,$G=\langle V,E\rangle$,G有且仅有一个根结点,从它可以到达其他所有结点,而从其他结点不能到达该结点.在根结点前插入一个空并行组,则扩展选择调度的问题是,在满足资源和数据相关限制的条件下,将该图中无关的可执行操作尽可能多地聚集在该并行组中.

扩展选择调度也分为计算与代码移动.下面主要描述计算部分.

2.3 计算$av(n)$

给定结点n,定义n的可用右手集合$av(n)$为:与n无关的、可执行的、处于同一个循环或更里面的嵌套循环的操作右手的集合.$av(n)$在所有点处计算,但仅在bb_header处保存.

最初,忽略图G中的所有循环回边,假想此时图中的所有叶子结点有一个后继,但其av为空,则对图中的一个结点n,增量式计算$av(n)$如下:

(1) 若$\text{opcode}(n)=\text{if}$,则

$$av(n)=((av'(T)\cup av'(F))-\text{AllCondBranches})\cup\{\text{if } cc_i\},$$

其中if cc_i是n的代码,T和F为n的两路目标结点.如果$n\rightarrow T$是循环退出边,则$av'(T)=\varnothing$(空集),否则$av'(T)=av(T)$.如果$n\rightarrow F$是循环退出边,则$av'(F)=\varnothing$,否则$av'(F)=av(F)$.这样规定的含义是不允许外层循环的代码进入内层循环,否则会引起语义错误.

© 中国科学院软件研究所 http://www.jos.org.cn

此外,规定一个分支不能越过另一个分支,因此,$av^t(T)$和$av^t(F)$中的分支操作也被去掉.

(2) 若 opcode(n)≠preheader(为统一起见,将 preheader 也看成操作),则

$$av(n)=(\text{moveup\_set\_rhs\_through\_loop}(\bigcup_{n\to x}av(x),n)-\text{AllCondBranches})\bigcup_{n\to x}av(x).$$

其中 moveup_set_rhs_through_loop(Δ,n)=$\{\Gamma\mid(\Gamma\in\Delta)\wedge(\text{sources}(\Gamma)\bigcap\text{dests\_in\_loop}(n)=\varnothing)\}$,sources($\Gamma$)是 RHS Γ 的源寄存器的集合,dests_in_loop(n)是以 n 为 preheader 的循环中所有目标寄存器的集合.

上述公式的含义是,仅当 RHS 与循环中任一操作无流相关时,才可以穿过循环.因此有一个推论,即如果穿过循环,则无替换发生.此外,分支操作不允许跨越循环,因为循环的结束判断语句本身就是分支语句,而分支是不允许越过另一个分支的.

(3) 否则,n 是一般操作:

$$av(n)=\text{moveup\_set\_rhs}(\bigcup_{n\to x}av(x),n)\bigcup\{\text{rhs}(n)\},$$

其中 moveup_set_rhs(Δ,n)是右手集 Δ 中能通过操作 n 的那些元素的集合,其具体定义见文献[2];rhs(n)是操作 n 的右手.

2.4 spec 的计算

对于结点 n,在 $av(n)$算出后,需要对 $av(n)$中的所有 RHS 计算猜测度.

(1) 若 opcode(n)=if,则

spec(rhs(n))=spec(if cc_j)=0;对于 $av(n)$中的其他 RHS Γ,若 Γ 同时在 $av(T)$和 $av(F)$中,则 spec(Γ)等于它在二者中的最大值;若 Γ 只在 $av(T)$或 $av(F)$中,则其 spec 加 1.

(2) 若 opcode(n)=preheader,则对 $av(n)$中的所有 RHS,其 spec 保持原值.

(3) 否则,n 是一般操作:

spec(rhs(n))=0;对 $av(n)$中的其他 RHS Γ:若 opcode(n)=copy,且两个不同的 RHS,Γ_1 与 Γ_2,通过替换成为同一 RHS Γ,则取 spec(Γ)=max(spec(Γ_1),spec(Γ_2));否则 spec 不变.

2.5 $lv(n)$的计算

lv 用于为最佳的 RHS 决定一个适当的目标寄存器.$lv(n)$表示结点执行之前的活变量集合.与 av 相似,在当前移动路径上,结点的 lv 可能变成非法,要随代码移动而动态更新.每一步代码移动有 3 种可能,与此对应,lv 的计算也有 3 种可能:

(1) RHS 在同层中移动,没有跨越循环.

如图 1(a)所示,RHS 从结点 n_1 移到了前驱 n_2 之前,则在当前移动路径上,沿此移动方向,按下式依次更新 $n=n_1,n_i$ 时的 $lv(n)$:

① 当 opcode(n)=if 时,$lv(n)=lv(T)\bigcup lv(F)\bigcup\{cc_j\}$;

② 否则,opcode(n)≠if,opcode(n)≠preheader(没有跨越循环,不会遇到 preheader),$lv(n)=(_{n\to x}lv(x)-\{\text{dest}(n)\})\bigcup\{\text{sources}(n)\}$,其中 dest($n$)是 n 的目标寄存器.

(2) RHS 在同层中移动,但跨越了循环(如图 1(b)所示).

由于是一步跨越循环,RHS 原来一定位于循环的出口边上.如图 1(b)所示,RHS 原来所在结点 n_1 是循环退出首结点,RHS 从 n_1 移到了循环的 preheader 之前.

① $lv(n_1)$按情况(1)重新计算.

② 对于被跨越的循环,

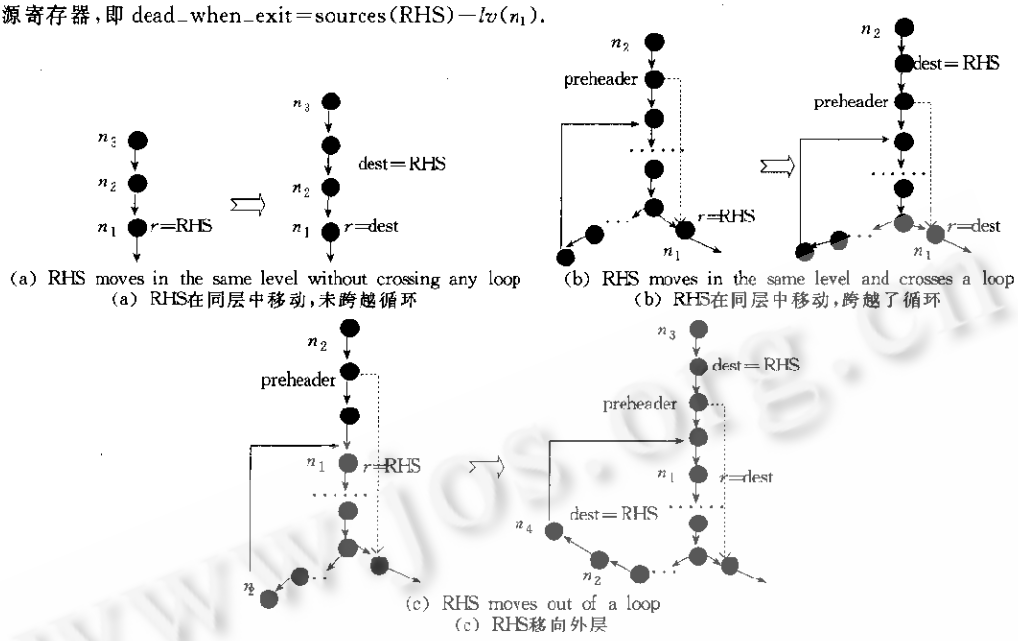
$$lv(n)=(lv(n)\bigcup\{\text{dest}\}-\text{dead\_when\_exit})\bigcup\text{still\_live\_in\_loop}.$$

其中 dest 是为 RHS 选择的目标寄存器;dead_when_exit=sources(RHS)$-lv(n_1)$;still_live_in_loop=sources(RHS)$\bigcap$sources_in_loop(preheader),sources_in_loop(preheader)是 preheader 所代表的循环中所有操作的源寄存器集合;n 为 preheader 或其所代表的循环中的一个 bb_header.

对该式的解释是,按照目标寄存器的选择规则,dest 在 preheader 代表的循环中未出现.因此,从图中可以看出,dest 在循环之前定义,在循环之后引用,在循环中应该是活的,故将其加入.在 RHS 移到循环上面之前,RHS 中的源寄存器在循环中肯定是活的,而移上去后就不一定了.因此应该从中去掉那些在循环出口已死去的 RHS

© 中国科学院软件研究所 http://www.jos.org.cn

的源寄存器,即 dead_when_exit=sources(RHS)−$lv(n_1)$.



(a) RHS moves in the same level without crossing any loop
(a) RHS在同层中移动,未跨越循环

(b) RHS moves in the same level and crosses a loop
(b) RHS在同层中移动,跨越了循环

(c) RHS moves out of a loop
(c) RHS移向外层

Fig. 1　Code motion
图1　代码移动情况

有些 RHS 中的源寄存器虽然在循环出口死去,但在循环中却是活的. dead_when_exit 包括了它们,减去 dead_when_exit 时减去了它们,这是不应该的. 所以,还应将它们补回来,即并上 still_live_in_loop=sources(RHS)∩sources_in_loop(preheader).

(3) RHS 移向外层

此时,RHS 一定位于循环 header 的后继结点上. 如图 1(c)所示,RHS 从 n_1 移到了 preheader 之前,同时在循环回边上产生了一个 bookkeeping copy n_4.

① 原来 lv(preheader)中的变量依然活着,同时增加了一个寄存器 dest,即 lv(preheader)=lv(preheader)∪{dest}. 同理,lv(header)=lv(header)∪{dest},$lv(n_1)=lv(n_1)\cup\{dest\}$.

② 对于 bookkeeping copy n_4,$lv(n_4)=lv$(header)−{dest}.

③ 循环内其他结点的 lv 不变.

RHS 每步移动的情况只属于这 3 种情况之一. 更为复杂的情况,如 RHS 移向外层后又跨越了循环,是以上 3 种情况的综合,可随 RHS 的逐步移动而逐步计算.

在上述情况中,lv 的计算与目标寄存器的选择有关. 假如 dest=RHS 最终被移到了组边界 x_i 上,则与选择调度一样,dest 必须满足下列条件[2]:

① 在从 x_i 到原操作的任何路径上未被读或写;

② 不在原操作之下的任何路径的活变量集中,除非就是原操作的目标寄存器;

③ 如果该操作不是两个分支都有,则寄存器不在该变量经过的分支的另一条路径的活变量集上.

2.6　计算 av(group)

并行组 group 是一棵树. 设组边界为 $x_1,x_2,\ldots,x_m$,每个组边界分别指向结点 $n_1,n_2,\ldots,n_m$. 通过上述有向图上增量式计算,每个 $av(n_i)$都已算出,而且并行组(树)是一个有向无环图,因此,av(group)的计算与选择调度完全相同,不再赘述.

最佳 RHS 的选择与代码移动也与选择调度基本相同,但应注意:

(1) 选择最佳 RHS 依然以循环体号、猜测度、原循环体中深度优先序号为标准,但此处循环体号可能是一

© 中国科学院软件研究所 http://www.jos.org.cn

个向量,$(i_1,i_2,\ldots,i_n)$,表示该 RHS 原来位于 n 重循环之中,第 1,2,...,n 重循环体号分别为 $i_1,i_2,\ldots,i_n$.

(2) 在计算 av 集合和寻找原来操作时,循环回边被忽略(认为不存在),因为我们只从各层循环的当前循环中提取可用操作.但是在代码移动时,各循环回边被认为是存在的.因为每个循环回边指向循环的 header,这是一个结合点,操作通过此点移出循环需要复制,即在循环回边上产生 bookkeeping copy,这样才能保证正确性.Lv 集合的计算与代码移动密切相关,因此,计算 lv 集合时也同样考虑到循环回边.应当指出,在有回边时能增量式地计算各个集合,这是扩展选择调度的关键.

(3) 代码移动时,preheader 是不复制的,每个循环有且仅有一个 preheader.

(4) 从 preheader 到各个 postexit 的虚有向线是为了计算方便,代码移动时无用(认为不存在).

扩展选择调度的主过程 GSS(oper)与选择调度的主过程 fill_ins(oper)的伪码描述相同,但是,其 av 和 lv 的计算子过程完全不同,需要按上面所叙述的改过来.

3 通用内向软件流水与全局压缩

3.1 算 法

算法 Pipeline_compaction( )反复应用以下(1)、(2)两步,直至程序中的所有操作被调度.

(1) **截断**某些边,以产生一个根部无环的有向图.

(2) 对此图应用扩展选择调度.后面产生的并行组能吸收前面产生的并行组,由此产生紧密的流水线核心.

算法通过维护栅栏和操作序号来实现.每个操作最初的序号由深度优先拓扑排序[2]产生.一个从大序号操作到小序号操作的边称为后向边(backward edge),其他边称为前向边(forward edge).在流水之前,循环回边就是后向边;随着过程的进行,会出现新的后向边.后向边的存在构成了环路.栅栏的作用就是打断后向边,以产生根部无环的有向图.

算法是对选择调度 Pipeline( )过程的改写.其主要变化是:

(1) 任一操作 n 可以在也可以不在循环中.也就是说,算法是对循环和非循环统一进行处理的.

(2) 增加了对 preheader 的处理.当栅栏中出现 preheader 时,意味着下面是一个(内层)循环,该循环在外层调度时,可能已经自动得到了全部或部分流水(不同内层循环体重叠),但是,它内部的操作尚未被收集到各个并行组中,因此还需要处理.首先,将 preheader 的序号变为当前序号最大值,然后将其后继 header 加入栅栏,从而在下面的处理中,将对该循环进行流水(或者说将其操作收集到各个并行组中).这个小变化意味着思维方式的转折:从外而内顺序流水.

(3) 对栅栏中的操作,使用 GSS 进行全局调度.它的集合计算方法确保外层能在内层流水之前就提前获得内层的装入部分,使外层代码能与之压缩,提高并行度.

算法.

```
PROCEDURE pipeline_compaction(first_oper)
{
  FOR each operation n in the program DO //Any operation of the program, including that in the nested
                                         //loops, should be depth_first ordered.
    seqno(n)=depth_first_ordering_number(n);
  ENDFOR
  orig_max_seqno=max({seqno(n)|n is an operation in the program})
  fence={first_oper}; global_level=0;
  WHILE (fence is not empty) DO
    FOR each operation n in the fence DO
      IF opcode(n)=preheader THEN //there is a loop that has not been pipelined
        let min_f be the minimum seqno used in the current fence;
        seqno(n)=seqno(n)+highest_seqno_in_use+1-min_f; //mark the loop as pipelined
        n=header;
      ENDIF
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
    ENDFOR
    let max_f and min_f be the maximum and miniumum seqno used in the current fence;
    FOR each operation n in the fence DO
        GSS(n); //insert a new empty parallel group R before n, let seqno(R)=seqno(n)-max_f  1,
                //any operation x moved into this group is given seqno(x)=seqno(R)
    ENDFOR
    new_fence={s|∃ p (p is a predecessor of s, and seqno(p)=seqno(R), where R is a group
                                just filled, and 0<seqno(s)<=orig_max_seqno)};
    FOR each filled fence group R DO
      increase the seqno of the opeations in R by highest_seqno_in_use + 1 + (max_f-min_f+1);
    ENDFOR
    fence=new_fence; //start a new stage
    global_level++;
  ENDWHILE
ENDPROCEDURE
```

算法 pipeline_compaction( )的重要性质是:对于一个完整的程序,不区分非循环压缩和循环流水,从第1个操作到最后一个操作扫描一遍,就对非循环做了全局压缩,而对循环做了软件流水.这是由于扩展选择调度对有环和无环图采用同样的贪婪计算和代码移动的结果.这表明,循环的软件流水和非循环的全局压缩没有本质区别,仅仅是输入不同而已.

算法的另一个性质是,它同时具有一趟计算(one-pass computation)和增量式计算(incremental computation)的特点.程序的每个操作被且仅被扫描一次,但操作的有关集合却是增量式计算的.这样大大减少了计算量.

3.2 抑制代码膨胀

GSS用于软件流水,内层代码对外可见,得到的是更紧密的并行组,付出的代价是更多的代码复制.膨胀的原因主要是内层循环的分支操作移入了外层,这样就造成了同一分支的多个拷贝.在它们共同的目标点处,将引起代码复制.

为抑制这种不必要的膨胀,我们提出同步结点(synchronization node)的概念.如果内层循环的分支操作移入了外层,则定义该分支的两个拷贝的共同目标点为同步结点.假设同步结点为 n,则强令 $av(n)=\varnothing$,并禁止 n 的进入边上的栅栏组移动,直到 n 的所有进入边都出现了栅栏组为止.此时,才按正常情况计算 $av(n)$,并允许 n 的进入边上的栅栏组移动.在所有进入边都出现了栅栏组的情况下,在 n 处是不会产生代码复制的.实践中,由于资源有限和数据相关限制,膨胀会进一步得到抑制.

3.3 性 能

一般来讲,扩展选择调度的时间效益要比选择调度好,至少不会比它差.因为前者生成的 av 集合包含了后者生成的 av 集合,由此生成的并行组更为紧密.另一方面,它的空间效益一般会比选择调度差,因为它的代码移动更多,由此引起的代码复制也更多.所以,性能评价取决于空间膨胀和时间减少的相对速度.

表1列出了一些含有多分支、多重循环的典型程序的手工实验结果.其中,空间膨胀比反映出了空间效益,它是调度前后所有操作总数的比值;而每个并行组的平均操作个数反映时间效益,它是调度后循环部分的操作总数除以循环部分并行组的总数.(不计非循环代码,因为它们只占空间,而执行时间相对于循环而言可以认为是0.)

表1指出,*GSS* 以平均3.14的空间膨胀比获得了12.1的平均每组操作个数,比选择调度分别高出0.34和4.2.然而,加速比还不能从平均每组操作个数中简单测算,因此,更为准确的数据还有待于大型实验来测定.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 1 Experimental result
表1 实验结果

Program①	Nesting level②	Number of loops③	Total operations④	Total branches⑤	Selective scheduling⑥		Generalized selective scheduling⑦	
					Code expansion ratio⑧	Average operations per group⑨	Code expansion ratio	Average operations per group
Trans-Sparmat	2	2	18	3	2.0	10.7	2.6	13.3
qkpass	2	3	21	3	2.3	6.2	2.3	6.2
Replace-Selection	3	3	35	6	3.5	4.96	3.5	4.96
Bubblesort	2	2	18	3	2.1	7.5	2.3	10.0
FFT	3	4	41	4	4.1	10.2	5.2	13.6

Resource constraints⑩			
Machine	ALU-OPs	Mem-OPs	ALU-Ops+Mem-OPs	n-way branching
16-ALU	16	8	16	16

①程序名,②嵌套层次,③循环个数,④操作个数,⑤分支操作个数,⑥选择调度,
⑦扩展选择调度,⑧空间膨胀比,⑨每组平均操作个数,⑩资源约束.

4 结 论

扩展选择调度的意义是多方面的.(1) 它从一个全新的角度来看待多重循环,通过恰当地计算 av 和 lv 集合,实现了多重循环的直接调度;(2) 对于一般的任意控制流程序,它都是适用的;(3) 它同时打破了非循环代码全局压缩和循环代码软件流水这两种技术的界限,指出它们没有本质的不同,只是同一过程对不同输入产生的不同结果而已;(4) 程序的并行化不需要分层简化,只要顺序扫描一遍即可.

References:

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

[2] 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.

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

[4] Wang, J., Gao, G.R. Pipelining-Dovetailing: a transformation to enhance software pipelining for nested loops. In: Gyimhy, T. ed. Compiler Construction: the 6th International Conference, CC'96. New York: Springer-Verlag, 1996. 1~17.

[5] Yu, T., Tang, Z.Z., Zhang, C.H., *et al*. Control mechanism for software pipelining on nested loop. In: Regina, S.S. ed. Proceedings of the Conference on Advances in Parallel and Distributed Computing. Los Alamitos: IEEE Computer Society Press, 1997. 345~350.

Parallelizing Programs with Sequential Scanning

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

Received May 17, 1999; accepted September 23, 1999

Abstract: Generalized selective scheduling (GSS) is presented to uniformly process loops and acyclic code. GSS does not differentiate acyclic code from cyclic code, but generates the result of global compaction and software pipelining for them respectively. The program is parallelized not by hierarchical simplification, but by only one-pass sequential scanning. As the first global scheduling based on general graphs instead of traces or directed acyclic graphs, GSS breaks the boundary between acyclic and cyclic code scheduling. It views nested loops from a fresh angle, realizing the direct scheduling of nests by properly calculating availability sets and live variable sets. It is applicable to programs with arbitrary control flow.

Key words: instruction-level parallelism; global compaction; software pipelining; branch; nested loop

© 中国科学院软件研究所 http://www.jos.org.cn