基于剖析信息和关键路径长度的软件扇出树生成算法

曾 斌[1] 安 虹[1,2] 王 莉[1]

(中国科学技术大学 合肥 230027)[1] (中国科学院计算机体系结构重点实验室 北京 100080)[2]

摘 要 开发利用ILP(Instruction-level Parallelism)是现代高性能处理器取得高性能的关键要素之一。宽发射的超标量处理器、超长指令字处理器和数据流处理器只有在并行执行多条相邻的指令时才能获得较高的性能。数据流处理器的一个关键问题是如何把指令的计算结果高效地播送给目标指令而不用读写集中式寄存器文件。对于每条目标数大于指令所能编码的目标数的指令,编译程序都要插入一棵由MOV指令构成的软件扇出树来把计算结果播送给多条目标指令。为了暴露更多的ILP给硬件执行基底,提出了一种改进的软件扇出树生成算法,本算法根据目标指令的执行概率大小以及目标指令到该指令所在块的出口的关键路径长度来计算目标指令的权值,然后对各个叶子的优先权值进行排序,再根据优先权值的顺序来构造一棵软件扇出树,以便把指令的计算结果播送给多条目标指令。实验结果发现,本算法相对于传统的软件扇出树生成算法其性能有较大的提高。

关键词 软件扇出树,执行概率,关键路径,优先权

中图法分类号 TP302.1 **文献标识码** A

Profile Information and Critical Path Length Based Software Fanout Tree Generation Algorithm

ZENG Bin[1] AN Hong[1,2] WANG Li[1]

(School of Information Science and Technology, University of Science and Technology of China, Hefei 230027, China)[1]

(Key Laboratory of Computer System and Architecture, Chinese Acamemy of Sciences, Beijing 100080, China)[2]

Abstract Exposing and exploiting instruction-level parallelism (ILP) is one of the most critical components of high performance for modern processors. Wide-issue superscalar, very long instruction word (VLIW) and dataflow processors can only achieve high performance when they execute nearby instructions in parallel. In order to expose more instruction-level parallelism to hardware execution substrate, this paper proposed an improved fanout tree generation algorithm. This algorithm first calculates the priority value of every target instruction based on its execution probability and the critical path length from the target instruction to an exit of its block, and then sorts the priority values in ascending order and generates a software fanout tree. Experimental results show that the algorithm proposed in this paper improves the performance over the prior work modestly.

Keywords Software fanout free, Execution probability, Critical path, Priority

1 引言

处理器体系结构和编译技术是信息产业的基础研究,而信息产业是我国现代化建设的支柱产业之一。随着芯片制造工艺逐步接近硅原子的尺寸,传统的超标量超流水线处理器面临着诸多尚待解决的问题,比如处理器主频提升减速[1]、功耗快速增长[2]、芯片内部元部件之间的线延迟加长等一系列问题[3],种种迹象表明处理器时钟频率再也不能像过去十年那样以每年50%~60%的速度增长,而会降低到和制造工艺成线性关系地以每年12%~17%的速度提升[4]。然而,芯片内部集成的晶体管数目仍然按照摩尔定律以指数级的增长也带来了大量潜在计算资源。如何利用这些潜在计算资源来满足应用程序对计算资源的日益增长的需求成了当前处理器学术界和工业界亟需解决的问题。

为了解决所面临的问题,各大主流处理器厂商都相继推出了开发利用TLP(Thread-level Parallelism)的多核处理器,并且正致力于在单个芯片内部集成更多核。多核处理器能并行执行多条线程,提高IPC(Instructions per cycle)。然而,多核处理器需要对上层的编程模型做出改动才能提高单个应用程序的性能。虽然多核处理器能同时运行多条线程,但是它并不能提高单道线程的运行速度。为了降低功耗,多核处理器内部的单个核往往比较简单,时钟频率较低,因此单道线程的性能相对于在单核处理器上还会有所下降。为了加速应用程序的运行,程序员必须显式地线程化应用程序才能利用多核处理器的多核计算资源来加速单个应用程序,否则单个应用程序在多核处理器上的运行性能还会有所下降。而多线程

到稿日期:2009-04-20 返修日期:2009-07-01 本文受国家自然科学基金重点项目(60633040),国家973计划项目(2005CB321601),国家863重大项目(2006AA01A102-5-2),教育部-英特尔信息技术专项科研基金项目(MOE-INTEL-08-07)资助。

曾 斌(1983—),男,硕士生,研究方向为处理器体系结构和编译优化技术,E-mail:zengbin@ustc.edu.cn;安 虹(1963—),副教授,研究方向为并行处理器体系结构、微处理器体系结构;王 莉(1981—),女,博士生,研究方向为处理器体系结构和编译优化技术。

编程是学术界公认的难题。与传统的串行程序不同，多线程程序难以阅读，程序员难以理解；多线程程序还具有不可预测性和不确定性，多条线程可以以多种方式交织执行，程序行为不可预测，程序错误不能通过调试来纠正。此外，当多核处理器内部核的数目达到一定时，寻找到足够的线程来利用多核资源也是未来多核技术将面临的难题之一。

另一个可能的方向为以 EDGE (Explicit Data Graph Execution)处理器体系结构为代表的开发利用 ILP 的分片式处理器体系结构。它把计算和控制分布在多个高速片上通过互联网络相连的节点上，以降低设计和验证的复杂度，并且把一些高能耗的工作比如寄存器重命名和依赖探测等移交给软件来完成，因此可以有效降低处理器功耗。EDGE 处理器体系结构由 Texas 大学 Austin 分校提出[5]，它是一种新型可扩展、低功耗和高性能的类数据流处理器体系结构。TRIPS 是 EDGE 体系结构的第一个实例。通过开发 ILP，TRIPS 能够加快单道线程的运行速度。TRIPS 处理器体系结构有 3 大基本特征：第一是静态调度动态发射。程序在静态时通过编译器调度，并在取指时映射到执行基底上的各个执行单元，指令可以在动态时乱序发射，一旦一条指令所有的操作数到达并且谓词操作数的值和指令格式编码的谓词值相匹配时该指令可以立即发射，无需等到程序计数器指向该指令；第二是以块为基本执行单位。一个块最多可以有 128 条指令，块内的指令原子执行，或者都提交，或者都不提交；最后是块内指令之间的直接数据通信不用通过集中式寄存器文件，指令把计算结果直接播送给目标指令，而不是写入寄存器文件。此外，TRIPS 处理器体系结构采用了数据流谓词(Dataflow Predication)技术。在数据流谓词技术中，每条指令都有一个两位的域用来指定谓词操作数，以及两个目标域用来指定目标指令，在操作执行完后指令直接把计算结果播送给目标域指定的两条目标指令。数据流谓词技术结合了谓词执行技术和数据流技术的优点，但是也产生了新的问题，其中的一个关键问题就是如何把指令的计算结果高效地播送给目标指令。

在 TRIPS 处理器体系结构中，一条指令的目标指令可以有任意多条，然而由于 ISA(Instruction Set Architecture)的限制以及在硬件实现方面的权衡，指令格式中所能编码的目标指令数有限，普通指令只能编码两个目标，因此，对于每条目标指令数大于该指令所能编码目标指令数的指令，编译器都要插入一棵由 MOV 指令构成的软件扇出树来把指令的计算结果播送给多条目标指令。MOV 指令不对数据做计算而只是把接收到的值直接转发给更多的指令。通过实验发现，TRIPS 汇编文件中有 20%以上的指令为不直接参与计算的 MOV 指令，且其中大部分是由于指令的目标数过多而由编译器插入的用来构造软件扇出树的 MOV 指令。大量 MOV 指令的存在延迟了目标指令的发射，如果目标指令处于关键路径上，那么 MOV 指令将会延长块执行时间，降低程序的 ILP。为了把指令计算结果高效地播送给目标指令，优化软件扇出树的结构，提出了基于剖析信息和关键路径长度的软件扇出树生成算法。

2 相关工作

据调研，数据流处理器硬件和编译技术学术界提到了它们需要编译器插入扇出指令，但是，并没有描述或比较关于软件扇出树问题的算法。本节介绍与软件扇出树相关的 Huffman 算法和 Hartley 与 Casavant 提出的算法以及它们之间的区别。

Huffman 提出了一个基于消息字符出现频率的编码算法[6]。该算法根据信息出现的频率不同来构造一棵最优二叉树，使得该二叉树的所有叶子节点的加权之和最小，以达到编码的平均长度最短的目的。该算法首先把叶子节点按照权值递增的顺序排序，然后从中选取具有最小权值的两个节点并把它们作为新生成的内部节点的孩子节点，新生成的内部节点的权值为两个孩子节点的权值之和，然后把两个孩子节点从队列中删除，把新生成的内部节点按序插入到队列中。重复上述过程直到队列中的元素个数为 1 为止。Huffman 算法优化了所有叶子节点的权值之和。我们把 Huffman 算法应用到数据流处理器 ISA 的软件扇出树的构造问题上。

Hartley 和 Casavant 提出了一个 Huffman 算法的变体，用来重构算术表达式和加法器电路[7]。他们的优化目标主要是通过最小化树高度来最小化电路和 shim(用来保存已到达的操作数的存储电路)的延迟。该算法和 Huffman 算法类似，只是每个叶子节点的权重用叶子节点的延迟即叶子节点的操作数到达的周期数来表示而非频率，每个新生成的内部节点的权重是它的孩子节点的权重的最大值，而不是 Huffman 算法中的孩子节点权重之和。由于延迟已知并且固定，因此这个算法可以最小化电路中的关键路径长度。我们也把该算法应用到软件扇出树生成问题上，并与 Huffman 编码树算法以及我们提出的基于剖析信息和关键路径长度的软件扇出树生成算法相比较。

3 基于剖析信息和关键路径长度的软件扇出树生成算法

本节介绍所提出的基于剖析信息和关键路径长度的软件扇出树的启发式算法。首先介绍该算法的基本思想，然后再详述软件扇出树的生成过程。

3.1 算法思想

我们提出了基于剖析信息和关键路径长度的软件扇出树生成算法。在 TRIPS 体系结构中，块内的分支指令都被 If 转换技术线性化，块内不再有分支指令。每条指令都有一个两位的谓词操作数用来确定该指令都是否执行。如果指令的数据操作数都已经达到，并且指令中编码的谓词与谓词操作数的值相匹配，则该指令可以在硬件资源允许的条件下立即发射，否则指令不发射。程序中大部分的分支指令是具有倾向性的，分支指令的一个分支的执行概率大大超过另一个分支，因此如果优先给执行概率较高的分支播送数据，则可以提高数据播送的准确率、硬件资源的利用率和程序的 ILP。目标指令的执行概率可以通过剖析技术来获取。首先在控制流图中插入记录剖析信息的代码，记录剖析信息的数据结构是一个具有 15 个成员变量的结构体。结构体的成员变量声明代码片段如下所示：

```
FieldDecl fname = new FieldDecl("fname",pfpct); //指向保存函数名的串的指针
FieldDecl numb = new FieldDecl("numblks",pfit);//存储函数中基本块的数目的整形变量
FieldDecl lblk = new FieldDecl("lblk",pfpit);//指向存储基本块的行号数组的指针
```

FieldDecl blkcnt = new FieldDecl("blkcnt",pfpit);//指向保存基本块执行次数数组的指针

FieldDecl nume = new FieldDecl("numedges",pfit);//保存边数目的整形变量

FieldDecl lsrc = new FieldDecl("lsrc",pfpit); //指向边起点行号数组的指针

FieldDecl ldst = new FieldDecl("ldst",pfpit); //指向边终点行号数组的指针

FieldDecl edgecnt = new FieldDecl ("edgecnt",pfpit);/*指向边执行次数数组的指针*/

FieldDecl ptblsize = new FieldDecl("ptblsize",pfpit);/*保存函数中路径数目的整形变量*/

FieldDecl pathnums = new FieldDecl("pathnums",pfpit64);/*指向路径编号数组指针*/

FieldDecl pathcnt = new FieldDecl("pathcnt",pfpit64);/*指向路径执行次数数组指针*/

FieldDecl loopcnt = new FieldDecl("loopcnt",pfit);/*保存循环个数的整形变量*/

FieldDecl lloop = new FieldDecl("lloop",pfpit); //指向循环行号数组的指针

FieldDecl ltchtable = new FieldDecl("ltchtable",pfpit64);/*指向循环行程计数数组的指针*/

FieldDecl licnt = new FieldDecl("licnt",pfpit);//指向循环内指令条数数组的指针

然后在预编译的程序中插入记录信息的代码,一个块、一条控制流边或者一条执行路径每执行一次则在记录剖析信息的结构体中的相应项加1,接着把剖析信息写入到与源代码文件同名且扩展名为 pft 的文件中,下一次编译时再读入剖析信息指导重编译过程。

另一个影响叶子节点权值的关键因素是目标指令到该指令所在块的出口的最大执行延迟,称之为该目标节点到程序块出口的关键路径长度。关键路径长的指令应当具有更高的优先级。关键路径决定程序的执行时间,如果赋给关键路径长的目标指令更高的优先级则可以提前块的执行完成时间,从而提高程序的 ILP。由于在静态时无法判断动态执行的是哪一条路径,选取静态延迟最长的路径作为关键路径。关键路径长度包括处于关键路径上的指令的执行延迟以及指令之间的通信延迟。每条指令的执行延迟是固定的,可以通过指令的操作码获得,通信延迟可以通过已经确定调度位置的锚点之间的距离来计算。在 TRIPS 处理器体系结构上,指令在操作数到来以后就可以立即执行,因此,调度算法对关键路径计算的影响可以忽略。硬件资源冲突是另一个影响关键路径长度计算的因素,然而在编译时我们无法预知执行时硬件资源冲突的情况,因此不考虑硬件资源冲突这个因素。一个程序块可以有多个出口,一条目标指令到它所在的程序块的出口的路径可以有多条,通过后序遍历数据流图来计算目标指令到出口的路径长度的最大值,后序遍历保证了所有的孩子节点在它们的父亲节点被访问之前都已经被访问,因此在计算需要插入软件扇出树的指令的关键路径长度时,它们的目标指令到树的出口的关键路径长度都已知。

根据计算得到的目标指令的执行概率和关键路径长度,设计了一个软件扇出树生成启发式算法。一方面,如果叶子节点执行概率较大,则需要优先给它播送操作数;另一方面,如果叶子节点的到块出口的关键路径的长度较长,则说明赋给它更高的优先级可以提前关键路径执行上指令的起始执行时间。因此,用叶子节点的执行概率和它到它所在块的出口的关键路径长度的积作为叶子节点的权值。然后对由叶子节点组成的队列按照权值递增的顺序排序,从该排好序的列表的头部选取权值最小的两个节点作为新生成的内部节点的孩子节点。新生成的内部节点的权值为它的孩子节点的最大值加上孩子节点的执行概率之和,这是因为新插入的 MOV 指令的执行延迟为1,执行概率为它的两个孩子节点的执行概率之和。然后把两个叶子节点从队列中删除,并按顺序把新生成的节点插入到该队列的适当位置。重复上述过程直到队列中的元素个数为1为止。在 TRIPS 中,MOV 指令目标数可以有2,3或者4个,为了讨论简单,在这里假设所有 MOV 指令的所能编码的目标数为2,本算法可以推广到 MOV 指令目标数大于2的情况。算法形式描述如下:

Input //算法输入

Set $C=\{c_1,c_2,c_3,\cdots,c_n\}$//$n$ 个叶子节点的集合

Set $L=\{l_1,l_2,l_3,\cdots,l_n\}$//$n$ 个叶子节点的延迟的集合

Set $P=\{p_1,p_2,p_3,\cdots,p_n\}$ //n 个叶子节点的执行概率的集合

l_i=latency(c_i),1<=i<=n,p_i= probability(c_i)

Output //算法输出

A binary tree $T(C,L,P)$ whose leaves are from set T and internal nodes are inserted MOV instructions.

Goal //算法目标

let $WLS(T)=\sum_{i=1}^{n}(lat\_leaf_i+depth\_leaf_i)\times freq\_leaf_i$ be the weighted latency sum of all the leaves of the binary tree T. *Condition*: $WLS(T)<=WLS(U)$ for any binary tree $U(C,L,P)$.

function WeightedLatencySum(Set C,Set L,Set P,int n) { /*C 为叶子节点集合,L 为它们的延迟,P 为叶子节点的执行概率,n 为叶子节点的数目*/

PriorityQueue pQueue;//声明一个 priorityQueue 用来保存节点的加权延迟

for(i=1;i<=n;i++){

c_i.wls=c_i.$prob$ * c_i.$latency$;//计算叶子节点的加权延迟

add c_i into $pQueue$;//把叶子节点存入到 $pQueue$ 中

}

while($pQueue$.$size$() > 1){ //循环直到 $pQueue$ 中的元素个数为1为止

node a=$pQueue$.$poll$();

node b=$pQueue$.$poll$();//从 $pQueue$ 中取出具有最小加权延迟的两个节点 a 和 b

generate a new node t;// 生成新的内部节点

t.$lchild$=a;

t.$rchild$=b;//a 和 b 作为新节点的孩子节点

t.wls=(a.wls+b.wls)+(a.$prob$+b.$prob$);/*新生成的内部节点的加权延迟为孩子节点的加权延迟之和加上孩子节点的执行概率之和*/

}

}

PriorityQueue 的插入、删除和查找操作的时间复杂度均为 O(log$n$),因此算法的时间复杂度为 O($n$log$n$),PriorityQueue 的大小为 n,生成树的节点个数为 $2n-1$,所以算法的空间复杂度为 O(n),其中 n 为叶子节点数目。

3.2 软件扇出树的生成过程

初始的数据流图并没有扇出指令。对于每条目标数大于

指令所能编码的目标数的指令，编译器都需要插入一棵由 MOV 指令构成的软件扇出树。MOV 指令把从其它指令传送过来的值直接转发给更多目标指令。图 1 为插入扇出树之前的数据流图。这个数据流图是从向量相加的核心程序中提取出来的，根据条件分支指令 tlt 的计算结果来判断是否把基地址值播送给多条 load 和 store 指令以及加指令。为了简单起见，目标指令以下的部分被省略掉。在这个图中，每个椭圆代表一条指令，椭圆内的括号里的数据表示该节点的权值，边旁边的数据表示该边的执行概率。

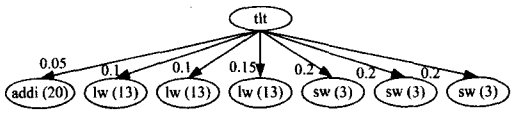

图 1　原始数据流图

如果忽略叶子节点的关键路径长度以及执行概率信息并赋给每个叶子节点权值 1，则 Huffman 算法和 Hartley 与 Casavant 算法构造的软件扇出树都一样，称之为平衡扇出树。如图 2 所示，所有叶子节点的加权延迟为 11.5。平衡扇出树的性能不佳是因为不同的目标指令有不同的权值，因此应该赋给它们不同的优先级。

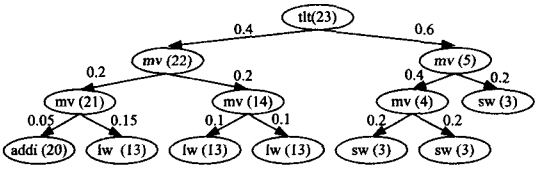

图 2　平衡扇出树

在图 1 中，store 指令离图的底端最近，应当赋给它较低的优先级，而 load 指令处于依赖链的顶部并且依赖链的执行延迟较大，因此给 load 指令播送数据更加关键，应该给它们优先播送计算结果。

图 3 为用 Huffman 算法构造的扇出树，由于没有考虑到执行概率，Huffman 编码树的加权平均延迟为 11.05。图 4 为应用 Hartley 和 Casavant 算法构造的扇出树，它的加权延迟为 11.25。图 5 为利用基于剖析信息和关键路径长度的软件扇出树生成算法构造的软件扇出树，它的加权平均延迟为 10.2，相对于 Huffman 算法和 Hartley 与 Casavant 提出的算法有较大的性能提升。

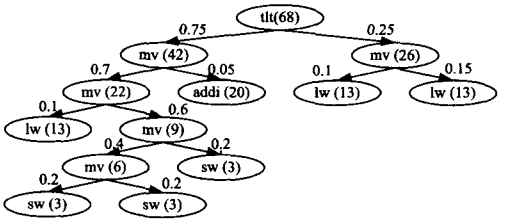

图 3　Huffman 算法构造的扇出树

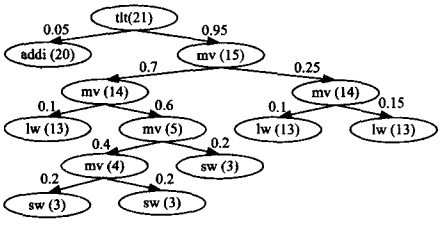

图 4　Hartley 和 Casavant 算法构造的扇出树

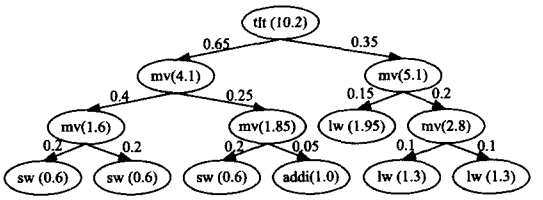

图 5　基于剖析信息和关键路径长度的软件扇出树生成算法构造的扇出树

4　实验结果

我们的实验平台为 TRIPS 的时钟精确模拟器以及 TRIPS 编译器 Scale 和 TRIPS 调度器。TRIPS 编译器把基本块合并成超块并生成一个指令格式类似于 RISC 的 TIL (TRIPS Intermediate Language)文件。然后调度器读入 TIL 文件，把操作数格式指令转化为目标格式指令，并对每个超块都构造一个数据流图。初始的数据流图并不包含任何的扇出指令。在把数据流图映射到执行基底上之前，对每条目标数都大于该指令格式所能编码的目标数的指令调度器都插入一棵由 MOV 指令构成的软件扇出树。

在 Scale 编译器中获取剖析信息，把剖析信息写入 TIL 文件，在每条测试指令之后都附加有 prob = doubleValue，它表示该测试指令执行 true 路径时的概率，false 路径的概率为 1-prob，然后在 TRIPS 调度器中读入 TIL 文件以及剖析信息。在 TRIPS 调度器中实现了平衡扇出树生成算法、Huffman 算法、Hartley 和 Casavant 提出的算法以及基于剖析信息和关键路径长度的软件扇出树生成算法。图 6 显示了各个算法在 microbench 上的性能。由于 SPEC_CPU2000 在模拟器上的运行速度太慢，运行时间太长，针对 microbench 测试了算法的性能，microbench 是从 SPEC_CPU2000 中抽取出来的内核程序，它具有 SPEC_CPU2000 的基本特征。我们测试了 microbench 的前 8 个程序，ammp_1，ammp_2，art_1，art_3，bzip2_1，bzip2_2，bzip2_3 和 doppler_GMTI。图 6 中的横坐标表示不同的基准测试程序，纵坐标表示相对于平衡扇出树算法的性能提升，Huffman 表示 Huffman 提出的算法，H/C 表示 Hartley 和 Casavant 提出的算法，Profile 表示基于剖析信息和关键路径长度的软件扇出树生成算法。从图中可以看出，基于剖析信息和关键路径长度的软件扇出树生成算法相对于平衡扇出树算法性能平均提升了 7.94%，而 Huffman 算法和 Hartley 与 Casavant 提出的算法分别提升了 1.03%和－0.83%。对于大多数测试程序，基于剖析信息和关键路径长度的软件扇出树生成算法的性能都有所提高，在 bzip2_1 上的性能提升了 44.59%之多，bzip2_1 的性能提升是由于算法大大提高了程序块的分支预测准确率，软件扇出树优先把计算结果播送给将要执行路径上的指令。另外有一些程序的性能有所降低，这是剖析信息与关键路径长度计算的不准确和随机因素所造成的。

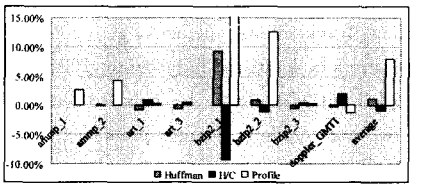

图 6　各软件扇出树生成算法的性能提升

为了更好地衡量不同的扇出树生成算法的性能，我们还

统计了各个算法生成的软件扇出树的一些静态信息。表1显示了软件扇出树的平均高度，软件扇出树的高度用从根节点到任意一个叶子节点的路径上所插入的最大的MOV指令条数来表示。为了最小化关键路径长度，Hartley和Casavant算法产生了最不平衡的软件扇出树。从表中可以看出，各个算法所生成的软件扇出树的平均高度比较接近，基于剖析信息和关键路径长度的软件扇出树生成算法生成的软件扇出树的平均高度略小于其它几种算法生成的软件扇出树的平均高度。各个算法生成的软件扇出树的最大高度也比较接近。但是各个算法所生成的软件扇出树的内部结构不同，因而导致了性能的差异。在基于剖析信息和关键路径长度的软件扇出树生成算法生成的软件扇出树中，执行概率大的和关键路径长的目标指令距离根节点较近，可以优先获得指令的计算结果，而在其它算法所生成的软件扇出树中却没有区分执行概率大的和关键路径长的目标指令，因而导致了性能的差异。以后的优化工作可以着重去优化软件扇出树的内部结构。

表1 各扇出树生成算法生成的扇出树的平均高度

	Average fanout tree height			
	Balanced	Huffman	H/C	Profile
ammp_1	3.04	3.04	3.04	3.04
ammp_2	3.03	3.03	3.03	3.03
art_1	3.25	3.24	3.25	3.22
art_2	2.81	2.81	2.83	2.76
art_3	3.23	3.25	3.24	3.22
bzip2_1	3.18	3.18	3.18	3.18
bzip2_2	3.07	3.07	3.07	3.07
bzip2_3	3.14	3.14	3.14	3.14
doppler	3.10	3.08	3.08	3.08

结束语 本文提出了一种基于剖析信息和关键路径长度的软件扇出树生成算法。本算法利用了目标指令的执行概率和关键路径长度等信息来构造一棵软件扇出树，以便把指令的计算结果播送给目标指令。根据目标指令的执行概率和关键路径长度信息，本算法优先把计算结果播送给执行频率较高的和关键路径较长的目标指令。实验结果证实了本算法较传统的平衡扇出算法、Huffman算法和Hartley与Casavant算法在性能上有所提高。但是该算法还有待进一步的研究，即着重提高剖析信息和关键路径长度的计算精度，赋给剖析信息和关键路径长度不同的权重等。

参考文献

[1] Ronen R, Mendelson A, Lai K, et al. Coming Challenges in Microarchitecture and Architecture [C] // Proceedings of The IEEE. 2001:325-340

[2] Agarwal V, Hrishikesh M S, Keckler S W, et al. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures [C]// Proceedings of the 27th Annual International Symposium on Computer Architecture. July 2000:248-259

[3] Albonesi D H. Dynamic opc/block rate optimization [C]// Proceedings of the 20th Annual International Symposium on Computer Architecture. June 1998:282-292

[4] Amruthur B S, Horowitz M A. Speed and power scaling of SRAMs [J]. IEEE Journal of Solid State Circuits, 2000, 35(2): 175-185

[5] Burger D, Keckler S W, Dahlin M, et al. Scaling to the End of Silicon with EDGE Architecture. Computer [J]. IEEE Computer Society, 2004, 37(7):44-55

[6] Huffman D A. A method for the contruction of minimum-redundancy codes[C]// Proceedings of the IRE. September 1952, 40: 1098-1101

[7] Hartley R, Casavant A. Tree-height minimization in pipelined architectures[J]. IEEE Transactions on Computer Aided Design, 1989, 8:112-115

(上接第211页)

[7] Duarte A, Martí R. Tabu search and GRASP for the maximum diversity problem [J]. European Journal of Operational Research, 2007, 178:71-84

[8] Ghosh J B. Computational aspects of the maximum diversity problem [J]. Operations Research Letters, 1996, 19:175-181

[9] Palubeckis G. Iterated tabu search for the maximum diversity problem [J]. Applied Mathematics and Computation, 2007, 189: 371-383

[10] Hopfield J J, Tank D W. Neural computation of decisions in optimization problems [J]. Biological Cybernetics, 1985, 52: 141-152

[11] 旷章辉，王甲海，周雅兰. 用改进的竞争Hopfield神经网络求解多边形近似问题[J]. 计算机科学，2009，36(3)：179-182

[12] Smith K A, Abramson D, Duke D. Hopfield neural networks for timetabling: formulations, methods, and comparative results [J]. Computers & Industrial Engineering, 2003, 44: 283-305

[13] He H, Sykora O, Makinen E. An improved neural network model for the two-page crossing number problem [J]. IEEE Trans. Neural Network, 2006, 17(6): 1642-1646

[14] Hansen P, Mladenovic N, Moreno-perez J A. Variable neighborhood search [J]. European Journal of Operational Research, 2008, 191: 593-595

[15] 潘全科，王文宏，朱剑英，等. 基于粒子群优化和变邻域搜索的混合调度算法[J]. 计算机集成制造系统，2007，13(2)：323-328

[16] Lan G, DePuy G W. On the effectiveness of incorporating randomness and memory into a multi-start metaheuristic with application to the Set Covering Problem [J]. Computers & Industrial Engineering, 2006, 51: 362-374

[17] Créput J C, Koukam A. A memetic neural network for the Euclidean traveling salesman problem [J]. Neurocomputing, 2009, 72(4-6): 1250-1264

[18] Dongarra J J. Performance of various computers using standard linear equations software [R]. CS-89-85. Department of Computer Science, University of Tennessee, USA. http://www.netlib.org/benchmark/performance.ps, 2007

[19] Saif S M, Abbas H M, Nassar S M. An FPGA implementation of a competitive Hopfield neural network for use in histogram equalization [C]// Proceeding of 2006 International Joint Conference on Neural Networks. 2006: 2815-2822