ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.7, July 2008, pp.1603−1612
DOI: 10.3724/SP.J.1001.2008.01603

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于子结构分析的基本块重排算法*

刘先华+, 杨 阳, 张吉豫, 程 旭

(北京大学 计算机科学技术系,北京 100871)

A Basic-Block Reordering Algorithm Based on Structural Analysis

LIU Xian-Hua+, YANG Yang, ZHANG Ji-Yu, CHENG Xu

(Department of Computer Science and Technology, Peking University, Beijing 100871, China)

+ Corresponding author: E-mail: lxh@mprc.pku.edu.cn, http://mprc.pku.edu.cn/

Liu XH, Yang Y, Zhang JY, Cheng X. A basic-block reordering algorithm based on structural analysis. *Journal of Software*, 2008,19(7):1603−1612. http://www.jos.org.cn/1000-9825/19/1603.htm

Abstract: Basic-Block reordering is a kind of compiler optimization technique which has the effect of reducing branch penalty and I-cache miss cost by reordering basic blocks in memory. A new basic-block reordering algorithm based on structural analysis is presented. The algorithm takes the architectural branch cost model and basic-block layout cost model into consideration, uses the execution frequencies of control-flow edges from profile information, builds a local structural optimization policy and utilizes it in reordering program's basic blocks. The algorithm is implemented based on UniCore architecture, experimental results show that it better improved programs' performance with a complexity of only $O(n\times\log n)$.

Key words: basic-block reordering; structural analysis; compiler optimization

摘 要: 基本块重排是一类通过重新排布基本块在存储中的位置,以减少转移开销和指令 cache 失效率的编译优化技术.介绍了一种基于子结构分析的基本块重排算法.该算法通过统计剖视信息中控制流图的边执行频率,基于处理器转移预测策略构建转移开销模型和基本块排布收益模型.算法采用局部子结构优化的策略,改善基本块在存储中的排列顺序,从而减少转移开销,并提高指令 cache 的使用率,改善程序的总体性能.在 UniCore 处理器平台上进行了实验.实验结果表明,与其他基本块重排算法相比,该基本块重排算法在更大程度上减少转移开销和指令 cache 失效率的同时,其时间复杂度保持为 $O(n\times\log n)$.

关键词: 基本块重排;子结构分析;编译优化
中图法分类号: TP314 **文献标识码**: A

程序的控制流信息在编译优化过程中通常表示为控制流图,而可执行程序在存储空间中以一维线性序列形式排布,因此,编译器需要在代码生成时确定其排布顺序.此排布顺序会对程序运行的性能产生影响:一方面,转移指令的目标与其自身在存储中往往不连续分布,转移是否发生以及其目标地址通常在流水中后期阶段才能确定,这会使流水线出现空闲从而带来性能损失.另一方面,代码的排布还会影响指令 cache 的利用率.转移发

* Supported by the National High-Tech Research and Development Plan of China under Grant Nos.2002AA1Z2203, 2003AA1Z1010, 2005AA111010 (国家高技术研究发展计划(863))

Received 2006-11-21; Accepted 2007-03-26

生时,若转移指令与跳转目标地址存在于 cache 的不同行内,则相应的 cache 行中均可能有一些指令被装入但未被执行,这会带来更多的冷启失效和容量失效,降低指令 cache 的利用率.

随着嵌入式系统对功耗和成本的要求越来越高,很多流行的面向嵌入式应用的处理器都采用了简洁的流水线结构和转移预测策略以及相对较小的cache,这也对编译优化改善性能提出了更高的要求.通过基本块重排将经常执行及交互性强的代码紧密排列在临近地址空间,能够减少存储层次间的代码传输,改进cache使用率.现代处理器采用了各类转移预测策略来改善转移预测正确率和降低转移开销,这些策略更多地集中在对处理器体系结构的改进上.如果代码在存储中的排布与处理器预测策略能够较好地契合,就可能进一步减小转移带来的性能损失.这都使得代码重排在编译优化中占据了重要的地位.另外,由于链接时刻优化技术的发展,也促使传统的代码重排算法在代码优化工具方面得到广泛应用[1−3].

本文介绍了一种新的基于代码子结构分析的基本块重排算法.由于程序的控制流图大部分由若干种固定的子结构组成[4],如果针对每种子结构计算出其中各控制流边在一定执行频度下的转移开销,即可计算出该子结构在确定排布下的总体开销,从而选择针对该子结构的最优排布.本文所提方法基于程序控制流分析方法中的结构分析方法的基础而提出,它利用了高级语言转换为机器语言后仍持有的结构特点,在取得更好的性能提升的同时,算法时间复杂度保持为$O(n\times\log n)$,这里n为控制流图中的边数.

1 代码基本块重排

代码重排技术通过重新排布基本块在存储映像中的位置来减少转移开销和指令cache失效,作为一种关键的编译优化技术,在 20 世纪 80 年代被提出之后得到了广泛应用.早期的代码排列工作主要针对提高虚拟存储系统利用率、减少虚拟存储系统的活动页面数来进行,研究人员提出了一些代码重排的算法,其中多数以过程为基本单位.后来,人们将该工作扩展到存储层次的其他级别,特别是针对指令cache性能提高来开展优化.McFarling[5]利用剖视(profile)信息基于基本块执行频率信息对存储访问进行重排序,减少了冲突失效.Pattis和Hansen[6]利用控制流边的执行频率剖视信息进行代码重排,他们提出了自底向上和自顶向下两种方法,前者由执行频度最高的边开始处理,判断该边连接的两个基本块所在的链是否可以首尾相连合并;后者由程序入口基本块开始,挑选执行频度最高的边连接的基本块,排在前一基本块后,从而连接成若干基本块链.他们的算法在链接时刻实现,自底向上的算法(以下简称PH算法)取得了更好的效果.作为最著名的代码基本块重排算法, PH算法在现代许多代码重排工具和技术领域得到了应用[1−3,7].

在PH算法之后,Calder等人[8]在研究中利用基本块重排序来减少转移指令引起的流水线暂停,他将这种技术称为转移对齐(branch alignment),并提出在基本块重排时应该考虑不同的处理器转移预测策略对转移指令在不同情况下开销的影响.Calder对PH算法提出改进,将边按照执行频度排列,从频度最高的边开始考虑,对于边e:$A\rightarrow B$,使用开销模型检测将基本块B排在A后是否比排在B的其他前驱的后面更好.

随着处理器的转移预测策略逐渐变得更加复杂,而且Young等人[9]将一些情况下的代码重排归约成为TSP问题,该问题的研讨热点逐渐转向通过调整跳转指令地址以配合复杂的硬件转移预测策略,从而改善跳转预测正确率[7].近年来出现了不少针对代码重排算法的改进,基于两个基本块交替出现次数的时序信息[10]、静态调用图[11]、cache行染色[12]和软件踪迹cache[13]等各类模型都被用来指导过代码重排列.代码基本块重排的另一个发展方向,则是基于各类剖视信息利用代码重排从编译优化角度对系统功耗、代码大小等进行优化[14,15].这些研究进展主要侧重对现有的代码重排算法的有效利用,对代码重排算法本身并未提供改进.

2 基于子结构分析的基本块重排算法

2.1 转移开销及排列收益的分析和计算

对于特定的程序流图和编译技术,当机器的转移策略不同时,代码排布带来的开销将有所不同,这种模型称

为面向特定机器转移策略的转移开销模型.本文选择了以下的体系结构和编译技术来建立转移开销模型:

- 我们在这里选择了 UniCore-I 处理器作为实例.在 UniCore 处理器中,流水线分为取指、译码、执行、访存和写回 5 个阶段.处理器转移模型采用预测转移不发生的策略,跳转指令自身在流水线中开销为 1 个周期,如果跳转发生,将产生 2 个周期的开销.因此,条件转移指令发生跳转时共需要 3 个周期完成(2 个周期的开销加上条件转移指令本身的 1 个周期),不发生跳转时需要 1 个周期.另外,无条件转移指令的执行总是需要 3 个周期.
- 基于 PH 代码重排算法改进的 Calder 算法,基于子结构分析的代码重排算法.

这里,我们首先引入一些相关概念:设 $G=\langle N,E\rangle$为一个有向图,N 为基本块节点集合,E 为有向控制流边集合.对于 $B\in N$,其后继集合 $Succ(B)$和前驱集合 $Pred(B)$定义如下:$Succ(B)=\{X\in N|\langle B,X\rangle\in E\}$,$Pred(B)=\{B\in N|\langle B,X\rangle\in E\}$,边的权重 $Freq(e)$表示在剖视信息中该边的执行次数.程序的转移开销与控制流图中的边直接相关,控制流在经过一条边时产生相应的转移开销.对于控制流图 $G=\langle N,E\rangle$,一条从基本块 B 到基本块 X 的边 $e=B\to X(B,X\in N)$的转移开销定义为执行这条控制流边所需要的总周期数(这里只考虑该指令的执行周期、因转移预测错误引起的延迟两个因素,不考虑 cache 失效带来的损失),用 $Cost(e)$表示,则 $Cost(e)$与首尾基本块的特点及位置存在如表 1 所示的 4 种相关情况.

Table 1 Layout profit of execution edges in UniCore (period)

表 1 UniCore 中边 e: $A\to B$ 的潜在排列收益(周期)

	$Cost(e)$ when B is not followed A	$Cost(e)$ when B is followed A	Potential layout profit
A has only one successor B	$3\times C_{NT}$	0	$3\times C_{NT}$
A has more than one successors	$(3\text{-}4)\times C_{NT}$	C_{NT}	$(2\text{-}3)\times C_{NT}$

Young[9]提出的开销公式概括了各种体系结构下的预测模型及相应的基本块重排开销.将基本块X排列在基本块B之后的开销计算公式为

$$Cost(B,X)=C_{BX}P_{NN}+I_{BX}P_{TN}+\sum_{B'\neq X}(C_{BB'}P_{TT}+I_{BB'}P_{NT}) \tag{1}$$

其中,P_{TN}表示预测转移发生而实际不发生时的执行开销,P_{TT}表示预测转移发生并且实际发生的执行开销,P_{NT}和P_{NN}同理定义,P值仅与当前基本块结尾处指令类型有关.C_{BX}表示正确预测控制流从基本块B转移到X的次数,I_{BX}表示预测控制流从基本块B转移到除X以外的其他基本块的次数,但实际上,即为控制流从B转移到X的次数.在UniCore体系结构中,由于总是预测转移不发生,所以公式(1)可以被简化为

$$Cost(B,X)=C_{BX}P_{NN}+\sum_{B'\neq X}I_{BB'}P_{NT} \tag{2}$$

定义(程序控制流图中边 e 的潜在排列收益). 基于给定的转移开销模型,对于程序控制流图的边 e:$A\to B$,其首节点 A 和尾节点 B 所代表的基本块在各种排列方式下可能带来的最大开销和最小开销之差,称为边 e 的潜在排列收益.

如图 1 所示,在 UniCore 体系结构中,图 1(a)中基本块 A 只有一条出边,基本块 A 的结尾无须任何转移指令,边 $A\to B$ 执行开销为 0.图 1(b)中基本块 A 只有一条出边,但其后继 B 不在 A 之后相邻的位置,A 的结尾需要有一条无条件转移指令,边 $A\to B$ 的执行开销为 3 个周期.图 1(c)中基本块 A 有两条出边,如将 B 排在 A 之后相邻的位置,A 的结尾处需有一条条件转移指令,边 $A\to B$ 的执行开销为 1 个周期,边 $A\to C$ 的开销为 3 个周期.图 1(d)中基本块 A 仍然有两个后继 B 和 C,但是 B,C 均不排在 A 之后相邻的位置,此时,在 A 的结尾处需要有两条转移指令,故边 $A\to B$ 的执行开销为 3 个周期,边 $A\to C$ 为 4 个周期.根据上述计算,对于控制流图的一条边 e,其潜在排列收益见表 1.

执行开销值取 4 的情况仅在基本块A的两个后继都不在A后相邻排列时发生.由于在进行代码排列前无法确定此类情况,且这种情况比较少见,故为了简便起见,当e的首节点A有多于 1 个后继时,我们统一将潜在收益 $Profit(e)$设为 $2\times C_{NT}$.因此在我们的模型中,根据A的后继数目确定e:$A\to B$的潜在排列收益如下:$Profit(e)=3\times C_{NT}$(A仅有 1 个后继)或$Profit(e)=2\times C_{NT}$(A有多个后继).实际上,这里,C_{NT}边等于e的执行次数,由此可记为

Profit(*e*)=[2|3]×*Freq*(*e*),*Freq*(*e*)为边*e*的执行次数.

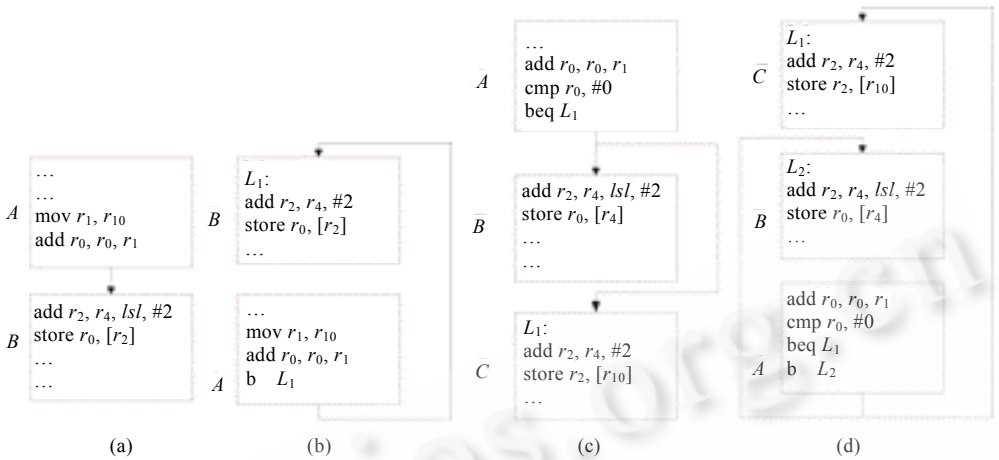


Fig.1 Different branch cost of edges in control flow graph

图 1 控制流图中不同边的不同转移开销

2.2 程序控制流图局部结构分析与优化

Pattis 和 Hansen 在其代码重排算法中采用边短接的方式连接基本块.由于该问题为 NP 难问题,作为妥协,我们可以使用贪心法得到时间复杂度为 $O(n\times\log\ n)$的解(n 为边数).Calder 对跳转预测的影响进行了考虑,不过其算法仍是贪心算法,在一些情况下可能得到非最优解.以图 2 为例,基本块 *A*,*B*,*C* 组成一个 if-then 结构,除图中显示的边以外,3 个基本块没有其他边相连,执行次数如边上标注所示.由于 *Freq*(*A*→*C*)>*Freq*(*A*→*B*),故 *Profit*(*A*→*C*)>*Profit*(*A*→*B*),因此,算法会优先选择边 *A*→*C*,基本块排列顺序将是 *A*→*C*,*B* 排在链外.但是,在 UniCore 处理器下,此排列的总转移开销是 6500+3500×3+3500×3=27500 个周期;如果按照 *A*-*B*-*C* 的序列排列,得到的开销是 3500+6500×3=23000 个周期.可以看到,后一种排列更优,这说明,PH 和 Calder 的算法在某些常见的情况下并不能得到最优结果.

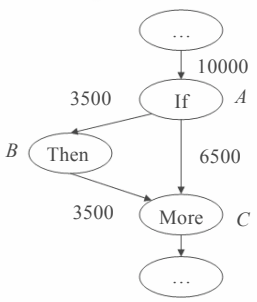


Fig.2 Example of if-then

图 2 If-then 结构示例

当尝试在代码重排中避免上述问题时我们注意到,现代的程序设计往往使得代码具备很好的结构特征,程序的控制流图在多数情况下可以被划分为若干典型的控制结构.我们可以利用程序语言的自身特点,分析控制流图中的控制结构.程序的大部分的控制结构都在文献[4]中定义的典型子结构之列.其中,不可归约区域在良好的结构化的程序中出现较少且没有固定结构,自循环(self-loop)结构只有一个基本块,Case/Switch 结构在二进制代码中反映为跳转表,均与基本块重排没有关系.因此,本文不考虑上述情况,仅讨论下面 7 种典型子结构并分析其最优排列方式.

7 种典型子结构图示如图 3 所示,其边上的权重 *x*,*y*,*x*1,*x*2,*y*1,*y*2 为根据程序执行的剖视信息统计得到的该控制流边的执行次数,针对各子结构的具体说明如下:

a) **Sequential**: $B_1;B_2;\ldots;B_n$. $Succ(B_i)=\{B_{i+1}\}$, $Succ(B_i)=\{B_{i+1}\}$.

b) **If-then**: if B_1 then B_2; end if; B_3.
$Succ(B_1)=\{B_2,B_3\}$, $Pred(B_2)=\{B_1\}$, $Succ(B_2)=\{B_3\}$, $Pred(B_3)=\{B_1,B_2\}$.

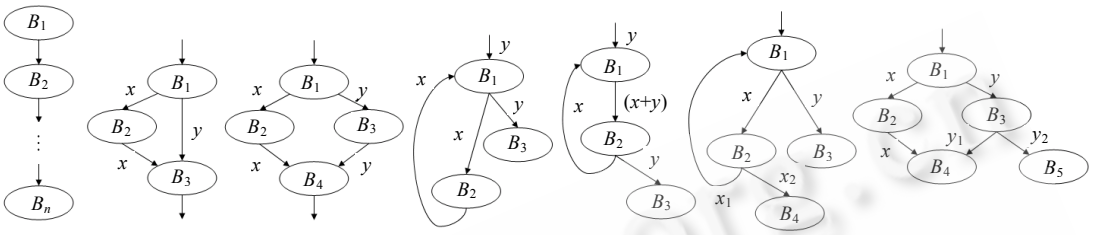
c) **If-then-else**: if B_1 then B_2; else B_3; end if; B_4.
$Succ(B_1)=\{B_2,B_3\}$, $Pred(B_2)=Pred(B_3)=\{B_1\}$, $Succ(B_2)=Succ(B_3)=\{B_4\}$.

d) **While-loop**: while B_1 do B_2; B_3. $Succ(B_1)=\{B_2,B_3\}$, $Pred(B_2)=Pred(B_3)=\{B_1\}$, $Succ(B_2)=\{B_1\}$.

e) **Repeat-loop**: repeat B_2 until B_1; B_3.其中,$Succ(B_1)=\{B_2\}$,$Pred(B_2)=\{B_1\}$,$Succ(B_2)=\{B_1,B_3\}$.

f) **Natural-loop**:除While-loop,Repeat-loop外的由单一入口节点和一条回边构成的循环,可有多个出口.这里,考虑有两个出口的情况,$Succ(B_1)=\{B_2,B_3\}$,$Pred(B_2)=Pred(B_3)=\{B_1\}$,$Succ(B_2)=\{B_1,B_4\}$.

g) **Proper-break**:有单一入口、多个出口的无环区域[4].本文中考虑有两个出口的情况,由B_1,B_2,B_3,B_4,B_5构成.$Succ(B_1)=\{B_2,B_3\}$,$Pred(B_2)=Pred(B_3)=\{B_1\}$,$Succ(B_2)=\{B_1,B_4\}$,$Succ(B_3)=\{B_5\}$.



(a) Sequential b) If-then (c) If-then-else (d) While-loop (e) Repeat-loop (f) Natural-loop (g) Proper-break

Fig.3 Seven typical structures in program CFG

图 3 程序控制流图的 7 种典型结构

以if-then结构为例,如图 3(b)所示,B_1有两个后继可供基本块链连接算法选择,可能的排列有B_1-B_2-B_3或B_1-B_3.根据表 1 计算,在B_1-B_2-B_3排列下,由于B_1的后继有B_2和B_3,而B_2被直接排列在B_1后面,因此,边B_1-B_2的转移开销为x;B_2的后继仅为B_3,因此,边B_2-B_3的转移开销为 0;B_3没有相邻排列在B_1后面,因此,边B_1-B_3的转移开销为$3y$,因此,B_1-B_2-B_3排列的转移开销总计为$x+3y$.同理可以得到在B_1-B_3,B_2的排列下,边B_1-B_2的转移开销为 $3x$.由于B_3不再排在B_2后面,故边B_2-B_3的转移开销也为 $3x$,而边B_1-B_3的转移开销减少为y,总计为 $6x+y$.如果选择排列B_1-B_2-B_3,要求 $Cost_{B_1\text{-}B_2\text{-}B_3} \le Cost_{B_1\text{-}B_3}$,即$x+3y\le 6x+y$,$y\le 2.5x$.同理,若选择排列$B_1$-$B_3$,则要求$y\ge 2.5x$.如果$y=2.5x$,为了更好地利用指令 cache 局部性,则应选择第 1 种排列,由此可以得到针对 if-then 结构的基本块顺序选择标准.与此类似,我们可以计算得到其他典型子结构的最优局部排列及其判断条件列于表 2.

Table 2 Condition table of layout adjustment

表 2 结构排列调整及判断条件

Structure type	Order of basic blocks		Transfer costs	Selection conditions
Sequential	Sequential order		0	Always
If-Then	B_1-B_2-B_3		x+0+3y	$y\le 2.5x$
	B_1-B_3,B_2		3x+3x+y	$y>2.5x$
If-Then-Else	B_1-B_2-B_4,B_3		x+0+3y+3y	$y\le x$
	B_1-B_3-B_4,B_3		3x+3x+y+0	$y\ge x$
While-Loop (Note 1)	B_1 has precursor layout before B_1	B_1-B_3,B_2	y+3x+3x	$y>x$
		B_2-B_1-B_3	3y+y+0+3x	$y\le x$
	Other situations	B_1-B_3,B_2	y+3x+3x	Not considered
		B_2-B_1-B_3	y+0+3x	Always
Repeat-Loop	B_1 has precursor layout before B_1	B_1-B_2-B_3	0+3x+y	Always
		B_2-B_1,B_3	3y+x+3(x+y)+3y	Not considered
	Other situations	B_1-B_2-B_3	0+3x+y	Always
		B_2-B_1,B_3	x+3(x+y)+3y	Not considered
Natural-Loop (Note 2)	B_1 has precursor layout before B_1	B_1-B_2-B_4,B_3	x+3x_1+x_2+3y	$y\le x$
		B_1-B_3,B_2-B_4	y+3x+3x_1+x_2	$y>x$
		B_2-B_1-B_3,B_4	3(x+y−x_1)+x_1+y+3x+3x_2	Not considered
	Other situations	B_1-B_2-B_4,B_3	x+3x_1+x_2+3y	$y\le x$,$y\le 2x_2$
		B_1-B_3,B_2-B_4	y+3x+3x_1+x_2	$y\ge x$,$x_1\le x_2$
		B_2-B_1-B_3,B_4	x_1+y+3x_1+3x_2	$x_1\ge x_2$,$y\ge 2x_2$
Proper-Break	B_1-B_2-B_4,B_3-B_5		x+0+3y+3y_1+y_2	$y\le x$,4$y_1\le 5x$
	B_1-B_3-B_4		y+y_1+3x+3x+3y_2	$4y_1\ge 5x$,$3x+2y_2\le 2y_1$
	B_1-B_3-B_5,B_2-B_4		y+y_2+3x+0+3y_1	$y\ge x$,$3x+2y_2\ge 2y_1$

注 1:当子结构的第 1 个基本块已存在某前驱相邻排列在其之前时,如果在其前再插入一个新的前驱基本

块,则会带来 3x(进入该基本块的边执行次数)的转移开销.

注 2:这里,根据图 3(f),$x=x_1+x_2$,同样的根据图 3(g),$y=y_1+y_2$.

算法即可依照该表对程序进行局部结构分析:遍历所有基本块,若某基本块及其后继与典型子结构匹配,则按最优排列方式重组此段代码.以图 2 为例,$Freq(A\to B)=6500<2.5\times3500=2.5\times Freq(A\to C)$,应调整顺序为 A-B-C(这里,$Freq(A\to B)$指的是边 $A\to B$ 的执行次数).

2.3 基于子结构分析的基本块重排算法

我们的基本块重排算法流程分为以下几个阶段:

1. 建立链接时刻的加权控制流图.此阶段在整个程序范围内构建程序控制流图.算法首先识别出所有基本块之间的连边,并标识连边属性.同时利用剖视信息,对控制流图边加入该边被执行的次数 $Freq(e)$作为权重信息.在链接时刻实现此项工作,构成一个程序的所有代码将被整体考虑形成一个完整的控制流图,扩大了基本块重排算法的作用范围,能够更好地对整个程序进行优化.
2. 形成高频执行路径基本块链.本阶段首先通过构造转移开销模型和收益权重,基于收益权重将所有的边排序,每次选择未处理的收益权重最大的边.在可能的情况下,将该边所连接的两个基本块排在一起,最终将基本块连成若干条基本块链.当控制流图的边的数目为 n 时,这一部分算法时间复杂度与排序复杂度均为 $O(n\times\log n)$,空间复杂度为 $O(n)$.
3. 通过局部结构分析优化基本块排列.由于基本块重排本身是 NP 完全问题,时间复杂度太高.所以,上一阶段选择通过贪心算法获得近似解,不能保证取得最优解.本阶段基于结合处理器的转移开销模型求出的每种子结构的最优排列方式,对程序进行扫描,如果某段代码片段和典型子结构相匹配,则根据该子结构中各边执行次数信息按最优方式对其重新排列.这部分算法时间复杂度为 $O(n)$,n 为控制流图边数目.
4. 基本块链间排列.上一阶段结束后,代码被划分为若干条基本块链.本阶段将这些基本块链进行线性排列,得到最终的代码排列.在这一阶段主要完成冷热代码分离,将很少执行的基本块与其他代码分离,放置在程序的尾部;同时,会将控制流交互度高的链排列在临近的位置,以减小 cache 和 TLB 冲突,这一部分内容采用的启发式算法简述如下:

   (1) 计算并记录链与链之间的相关情况,我们采用了一个 $n\times n$ 的矩阵 $rel[]$对其进行记录,n 为上一过程结束时的基本块链数目,该矩阵描述了链和链之间交互次数的情况,$rel[i,j]$表示为第 i 条基本块链中的所有基本块和第 j 条链的所有基本块之间连边的执行次数的总和,它在一定程度上反映了各个基本块链之间相互引用的相关情况.令 $Freq(A\to B)$表示边 $A\to B$ 的执行次数,它可以通过统计剖视信息得到,用 $Chains[i]$表示基本块链 i 中的所有基本块的集合,该集合表现为一个链表,在上一阶段结束之后形成.该矩阵的计算公式如下:

$$rel[i,j]=\sum_{A\in Chain[i],B\in Chain[j]\ or\ A\in Chain[j],B\in Chain[i]} Freq(A\to B)\ i\neq j\ .$$

   (2) 查找矩阵,找到 $rel[i,j]$的最大值,它表示基本块链 i 和基本块链 j 之间具有最多的交互,为了改善 cache 性能,应该优先将这两条基本块链连在一起,将 j 合并到 i 中.然后更新相关度矩阵,即

$$rel[i,k]=rel[i,k]+rel[j,k]\ (k\neq i,j),$$
$$rel[j,k]=0\ (k\neq i,j).$$

   (3) 重复(2),直到所有基本块链合并为一条链.

整体算法伪码描述如下:

```
struct block
{
    bool visit;        //记录该基本块是否被访问过,初始值为 false,表示该基本块未被访问
    bool isTail;       //为真时,表示该基本块为某基本块链的尾基本块,初始值为 true
    bool isHead;       //为真时,表示该基本块为某基本块链的首基本块,初始值为 true
```

```
        struct block *next;     //此基本块的相邻下一个基本块,初始值为 null
    } A, B;
struct edge
    {
        struct block *pre;      //控制流边的首节点
        struct block *suc;      //控制流边的尾节点
        unsigned int freq;      //控制流边的执行次数
        unsigned int profit;    //控制流边的收益
    };
procedure reorder(E)            //E 为控制流图的边集合
begin
    sort_edge(E); //将所有的边按 profit 排序,排序结果仍存放在 E 中
    for e in E do //按边的 profit 值从大到小访问所有的边,并尽可能地将其串成一条链
    begin
        A=e→pre; B=e→suc;
        if A.visit==false && B.visit==false;   //A,B 均未被访问过
            B.visit=true; A.visit=true;
            A→next=B;
            B.isTail=true; A.isHead=true;   //A→B 形成一个独立的链
        else if B.visit==false && A.isTail==true   //B 未被访问过,A 是某链尾基本块
            B.visit=true; A→next=B;
            A.isTail=false; B.isTail=true;   //B 成为该链的尾基本块
        else if A.visit==false && B.isHead==true   //A 未被访问过,B 是某链首基本块
            A.visit=true; A→next=B;
            A.isHead=true; B.isHead=false;   //A 成为该链的首基本块
        else if A.isTail==true && B.isHead==true;   //A,B 均被访问过,A 是某链尾基本块,B 是某链首基本块
            A→next=B;
            A.isTail=false; B.isHead=false;   //将两条链连接起来形成新的链
    end
    structural_analysis(N); //对已有的链进行扫描,检测存在的子结构并根据表 2 进行优化,输入 N 为顶点集
    connect_chains(); //将所有的链链接起来,形成基本块重排的序列,根据第 2.3 节描述的步骤 4 进行处理
end
procedure structural_analysis(N)          //N 为控制流图的顶点集合
begin
    for B in N(深度优先序) do
    begin
        //判断基本块 B 及其后继是否形成 if-then,根据各边权值并按表 2 计算选择最优排列
        V1=GetSucc(B); V2=GetSucc(B);   //得到B的前两个后继分别赋给V1和V2
        if Succ(B)=={V1,V2}                    //判断B的后继是否仅为V1和V2
if Pred(V1)=={B} && Pred(V2)=={B} && Succ(V1)=={V2} && GetFreq(B,V2)≤2.5*GetFreq(B,V1)
            B→next=V1; V1→next=V2; V1.visit=true;
            //调整基本块排列顺序为B-V1-V2,反之不作调整,函数GetFreq(B,V)返回边B→V的执行次数
if Pred(V1)=={B} && Pred(V2)=={B} && Succ(V2)=={V1} && GetFreq(B,V1)≤2.5*GetFreq(B,V2)
            B→next=V2; V2→next=V1; V2.visit=true;   //调整基本块排列顺序为B-V2-V1,反之不作调整
        //判断基本块 B 及其后继是否形成其他子结构,若是,则根据各边权值并按表 2 计算选择最优排列,伪码
            从略
```

```
        ……
    end
end
```

3 实验结果及数据分析

本文从面向嵌入式系统的基准程序集Mediabench中选取了 11 个基准程序作为本方案的评测用例(其余程序由于配套newlib库支持不够,没有选用),该评测程序覆盖了数据压缩和解码、图形编解码、音频编解码、3D处理等多个嵌入式领域,是目前嵌入式系统评测中最常用的评测程序之一,在该领域具备相当的代表性[16].评测选用的编译器基于gcc-3.2.1 移植到UniCore处理器上,编译时刻优化选项为"O2".本文采用基于SimpleScalar[17]代码移植改写的周期级流水线模拟器sim-pipeline进行评测,该模拟器对流水线互锁和前递进行了处理,考虑了多种存储访问情况下的cache开销,真实地反映了硬件执行情况,具体参数见表 3.

Table 3 Configuration parameters of simulator

表 3 模拟器配置参数

Branch prediction strategy	Predict "not taken"
Instruction cache	8KB, 2-way set associate, 32 byte block size, FIFO
Data cache	8KB, 4-way set associate, 32 byte block size, FIFO, write back, write allocate

我们将从算法对转移开销、指令cache和总体性能 3 个方面的影响来进行对比评测和分析.在参照方法的选择上,近年来出现了一些针对特定体系结构进行优化的基本块重排方法,例如Ramirez提出的软件踪迹Cache(software trace cache,简称STC),其侧重点在于创建了一个可以改善代码局部性的空间(context free area,简称CFA),但在重要的基本块串链过程中仍然沿用了PH算法,而且STC算法相对来说更适合循环比较少、执行路径单一的大型程序,如数据库和整点SPEC程序等[13].而Shu Xiao[14]以及Akihiro[15]提出的算法则是尝试基于各类剖视信息,利用代码重排以获得系统功耗方面的优化,对代码重排算法本身并没有进行改进.考虑到目前大多数实际应用的编译优化工具都采用了PH算法,最终我们选择了Calder[8]的算法进行对比,因为该算法利用了体系结构信息,相比PH的经典算法存在一定程度的改进.下面我们将从 3 个角度对未经基本块重排优化的程序(即采用gcc-3.2.1 原始未修改original版本编译,下称ORIG)、经过Calder使用转移预测模型改进的PH算法优化的程序(下称CALDER)和经本文中基于子结构分析优化算法(structural analysis based optimization,简称SABO)的程序进行对比评测和分析.

(1) 对转移开销的影响

如图 5 所示,SABO 算法可使程序的平均转移开销降低到未经优化时原程序的 27%左右,与 CALDER 算法 22%的平均改进相比有一定优势.对于 epic 程序,SABO 算法可以减少 42%的转移开销,而对于 texgen 和 mipmap 程序,也可以分别减少 37%和 34%的转移开销;对于所有的评测程序,SABO 算法都可以取得 12%以上的转移开销的降低,且都比 CALDER 有不同程度的提高.这表明 SABO 算法在降低转移预测开销方面具有普遍的显著效果.对于采用深度流水的处理器,转移目标地址预测错误的开销通常更大,可以预测,在深度流水线处理器中采用 SABO 算法对转移开销的降低将更显著.

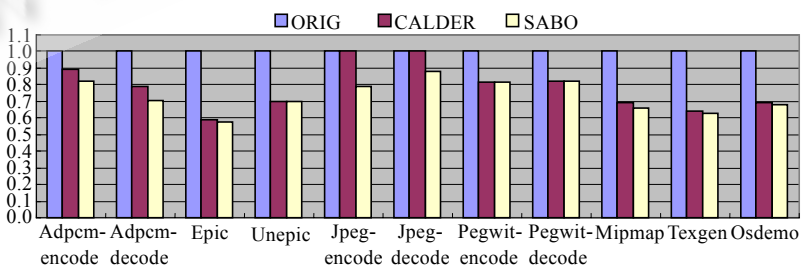


Fig.5 Comparison of branch cost by different layout algorithms

图 5 不同基本块排布下转移开销的对比

(2) 对指令 cache 的影响

由于本文算法可以将频繁执行的基本块放至相邻位置,因此可以改善代码的局部性,提高指令 cache 的性能.如图 6 所示,本算法可使指令 cache 的失效平均减少约 38%,其中,对于 pegwit 的编、解码可分别减少 80%和 62%的指令 cache 失效.对于 epic 程序,CALDER 和 SABO 都引起了更多的指令 cache 失效,分别为 32%和 39%,原因还有待深入分析.对除 epic 以外的其他程序,SABO 算法都至少减少 10%以上的指令 cache 失效.

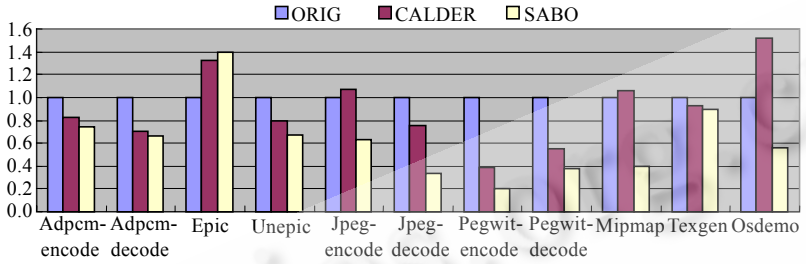


Fig.6 Comparison of I-cache misses by different layout algorithms

图 6 不同基本块排布下指令 cache 失效的对比

(3) 对总体性能的影响

图 7 显示了评测基准程序在不同优化算法下的性能差异.经本算法优化后的程序总执行周期均有所改善,平均减少了 7%的执行时间,相比 CALDER 算法减少了 3%.其中,mipmap 程序减少了 16%的执行时间,对于所有评测程序,最少的情况也有 2%的性能提高,且均比 CALDER 算法有所提高.

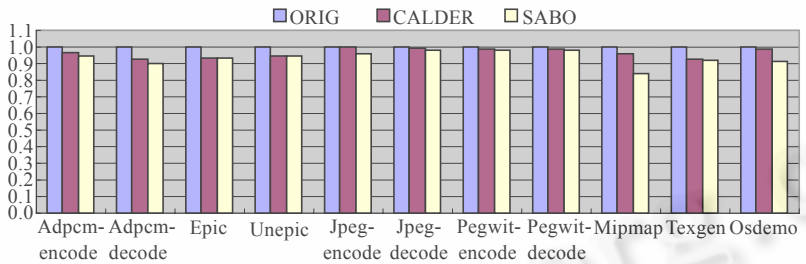


Fig.7 Comparison of performance by different layout algorithms

图 7 不同基本块排布下总体性能的对比

4 总 结

基本块重排是一种重要的编译优化技术,它通过重新组织基本块在存储中的排列顺序来减少程序中的转移开销和指令 cache 的失效损失.本文的工作主要针对减少转移开销来进行,同时考虑了 cache 利用率的提高.该算法在链接时刻实现.优化工具在链接时刻可以看到程序完整的控制流图,能够在整个程序范围内开展基本块调度,与传统的在编译时刻开展工作相比有更广的优化范围.同时,在链接时刻开展优化工作也便于将程序的冷、热代码进行分离,从而进一步提高性能.实验选用了 Mediabench 作为基准程序,在 UniCore 处理器平台上进行了评测.实验结果显示,基于体系结构开销模型和子结构分析的代码重排方法可以普遍且显著地减少程序的转移开销,降低指令 cache 失效率,从而有效提高程序运行的整体性能.

References:

[1] Cohn R, Goodwin P, Lowney PG, Rubin N. Spike: An optimizer for alpha/NT executables. In: Proc. of the USENIX Windows NT Workshop. Seattle, USENIX Association, 1997. 17–24.

[2] Scharz B, Debray S, Andrews G, Legendre M. PLTO: A link-time optimizer for the Intel IA-32 architecture. In: Proc. of the 3rd Workshop on Binary Translation (WBT 2001). Barcelona, 2001. http://research.ac.upc.edu/pact01/pwbt.htm

[3] Bus BD, Sutter BD, Put LV, Chanet D, Bosschere KD. Link-Time optimization of ARM binaries. In: Proc. of the ACM SIGPLAN 2004 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004). New York: ACM, 2004. 211−220.

[4] Muchnick SS. Advanced Compiler Design and Implementation. San Francisco: Morgan Kaufmann Publishers, 1997.

[5] McFarling S. Program optimization for instruction caches. In: Proc. of the 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III). New York: ACM, 1989. 183−191.

[6] Pettis K, Hansen RC. Profile guided code positioning. In: Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI'90). New York: ACM, 1990. 16−27.

[7] Jimenez DA. Code placement for improving dynamic branch prediction accuracy. In: Proc. of the ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation (PLDI 2005). New York: ACM, 2005. 107−116.

[8] Calder B, Grunwald D. Reducing branch costs via branch alignment. In: Proc. of the 6th Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS-VI). New York: ACM, 1994. 242−251.

[9] Young C, Johnson DS, Karger DR, Smith MD. Near-Optimal inter-procedural branch alignment. In: Proc. of the ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation (PLDI'97). New York: ACM, 1997. 183−193.

[10] Gloy N, Blackwell T, Smith MD, Calder B. Procedure placement using temporal ordering information. In: Proc. of the 30th Annual Int'l Symp. of Microarchitecture (MICRO-30). Washington: IEEE Computer Society, 1997. 303−313.

[11] Hashemi AH, Kaeli D, Calder B. Procedure mapping using static call graph estimation. In: Proc. of the 1st Workshop on the Interaction Between Compilers and Computer Architectures (INTERACT-1). Washington: IEEE Computer Society, 1997.

[12] Hashemi AH, Kaeli D, Calder B. Efficient procedure mapping using cache line coloring. In: Proc. of the ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation (PLDI'97). New York: ACM, 1997. 171−182.

[13] Ramirez A, Larriba-Pey JL, Valero M. Software trace cache. IEEE Trans. on Computers, 2005,54(1):22−35.

[14] Xiao S, Lai EMK. Instruction scheduling of VLIW architectures for balanced power consumption. In: Proc. of the 2005 Conf. on Asia South Pacific Design Automation (ASP-DAC 2005). Shanghai: IEEE Computer Society, 2005. 824−829.

[15] Chiyonobu A, Sato T. Energy-Efficient instruction scheduling utilizing cache miss information. ACM SIGARCH Computer Architecture News, 2006,34(1):65−70.

[16] Lee C, Potkonjak M, Mangione-Smith WH. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In: Proc. of the 30th Annual Int'l Symp. on Microarchitecture (MICRO-30). Washington: IEEE Computer Society, 1997. 330−335.

[17] Burger D, Austin TM. The Simplescalar tool set, version 2.0. Technical Report, CS-TR-97-1342, University of Wisconsin-Madison, 1997. http://www.simplescalar.com/docs/users_guide_v2.pdf

刘先华(1978－),男,湖北孝感人,博士生,主要研究领域为计算机系统结构,编译优化.

张吉豫(1982－),女,博士生,CCF 学生会员,主要研究领域为计算机系统结构,编译优化.

杨阳(1981－),女,硕士生,主要研究领域为计算机系统结构,编译优化.

程旭(1967－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为计算机系统结构,微处理器设计,编译优化.