基于 Trace-Cache 的多级动态优化框架设计

唐遇星, 邓 昆鸟, 周兴铭
(国防科学技术大学计算机学院, 湖南长沙 410073)

摘 要: 对指令集进行扩展和添加新功能部件是提高处理器性能的有效途径. 为了充分利用新的体系结构扩展, 已有应用必需经过全新的优化编译. 对于跨体系结构优化而言, 二进制翻译已经被证明是一种行之有效的技术. 本文结合 trace 技术和动态二进制翻译优化技术, 提出一种多级动态优化框架结构, 无需静态重新优化编译, 在程序动态运行期间, 引入多级动态优化方法和扩展指令调度. 模拟结果显示该结构具有能有效形成大尺寸的指令调度窗口, 准确选择热点代码及优化方法, 有效提升旧有应用性能的优点, 并有实现灵活, 可扩展好等特点.

关键词: Trace; 动态优化; 指令调度; 指令级并行
中图分类号: TP363 文献标识码: A 文章编号: 0372-2112 (2005) 11-1946-06

Trace-Cache Based Framework for Multi-Level Dynamic Optimization

TANG Yu-xing, DENG Kun, ZHOU Xing-ming
(*School of Computer, National University of Defense Technology, Changsha, Hunan 410073, China*)

Abstract: New extension of instruction set and new add-on function unit can improve the performance of microprocessor greatly. All the applications should be recompiled and rebuilt, otherwise they can't benefit from those new instructions. This paper proposes a framework of multi-level dynamic optimization, which introduces instruction scheduling and optimizing for the architecture extension based on trace-cache in runtime. Experimental results show that it can enlarge the instruction window to select hot codes and scheduling methods efficiently and effectively, and leverage the performance of original application without the need of recompiling. In addition, this framework is flexible and scalable to new optimizing chance and various platforms.

Key words: trace; dynamic optimization; instruction scheduling; ILP

1 引言

添加新的功能部件, 扩展原有指令集是提高处理器性能最直接的方法. 然而已有的应用如果不经过新优化编译器重新构建, 就无法从扩展指令和新功能部件中获得性能提高. 重新构建应用程序需要两个条件: 全部源程序和特定的优化编译器. 二进制共享库的广泛使用以及商业版权的限制, 使源代码的获取非常困难. 针对特定结构的静态优化编译器有使用复杂, 适应性差, 开发周期长等不足. 这些因素让静态重建整个应用往往很难实现[1]. 动态优化技术是解决这一问题的有效途径. 使用原有程序的二进制映像, 动态优化无需程序源代码和专用静态优化编译器. 通过在程序执行过程中对指令进行选择和调度, 利用准确的硬件分支预测机制, 发掘特有的动态优化机会, 动态优化技术可以显著的提升已有程序的性能[8]. 许多研究表明有效的动态优化可以平滑指令集迁移, 提高程序运行性能[1,2].

高效的动态优化策略应该解决以下问题:

(1) 选择有效的优化目标. 优化后的指令具有更好的性能, 但优化是有开销的, 不可能在动态运行时, 重新优化全部代码, 需要定位对性能影响最大的那部分指令.

(2) 合适的优化单位. 基本块内部的并行度和优化空间是有限的, 需要选择一个超越基本块调度窗口的优化单位.

(3) 性能平衡. 较大的指令窗口可以发掘更多的并行性, 获得更大的调度自由度. 但是经过跨越分支的指令调度后, 大尺寸的优化后代码在分支预测失败时的开销会更大. 并且由于包含的分支指令增多, 无效优化的概率也会增大. 动态优化框架需要在优化窗口大小和优化有效性之间取得平衡.

衡量动态优化的标准包括优化指令窗口的大小, 动态优化指令在程序执行过程中的命中率, 以及经过调度进入新功能部件使用扩展指令运行后程序性能的提升(本文使用加速比来表示)[3]. 本文也使用这三项指标对多级动态优化框架进行衡量.

2 相关研究

二进制翻译技术能够让应用程序无需重新编译就在扩展的指令集, 甚至是新的体系结构下运行. Transmeta 公司的 Crusoe 处理器已经证明了这一技术的可行性, 二进制动态翻译与动态优化技术成为体系结构研究的新热点[2,7].

以页面为单位的动态优化方法(Page-fold)是最直接的优化策略, 即在每个代码页载入的时候, 扫描并优化整个页面[6]. 这类方法的优点是硬件实现起来非常简单, 只需要修改

收稿日期: 2004-11-16; 修回日期: 2005-08-04
基金项目: 国家自然科学基金(No. 90307001)

处理器 page_ fault 陷入的处理过程. 但是在一个代码页面中可能只有很少指令会被执行, 一概而全的优化会耗费大量的运行时间, 并难以针对关键代码进行深度优化.

当前的许多研究都使用 trace 作为优化单位[2,4,5,8]. Trace 是程序动态运行时一组在逻辑上连续执行的指令. 阈值控制方法(Begin End Threshold) 用简单的 trace 起始和终止条件来控制 trace 的选取[8]. 在形成 trace 之后, 在 trace 之上应用所有的优化调度方法[5]. 这一策略的优点是 trace 形成的控制逻辑简单, 优化控制逻辑对每条 trace 而言都是相同的. 但每条 trace 的执行情况不同, 实际有效的优化也不尽相同, 盲目优化可能反而降低执行效率.

循环展开和软流水是发掘并行性的最佳方法. 许多优化调度框架和策略专门针对循环而设立[4]. 但是在动态运行时选择循环展开的次数是一个尚未解决的问题. 保守的策略将尽可能少的展开循环, 以避免过于复杂的失效恢复. 深度优化策略则需要候选 trace 多次重复执行, 多用在面向科学计算的静态编译优化中[5].

现有的研究表明动态翻译与优化技术在平滑指令集变化和体系结构迁移上有明显的作用和优势. 尚待解决的问题是: 如何有效选择指令调度的窗口, 并且根据被调度指令流的执行行为选择合适的调度方法. 本文提出一种基于 trace_ cache 的动态多级优化框架, 依据指令流的动态执行特征调整优化方法和指令调度窗口, 在扩大指令调度窗口的同时能够保证调度和优化方面的有效性. 并且这一框架结构具有良好的可扩展性, 实现灵活.

3　动态多级优化框架

3.1　基于 trace cache 的指令调度

程序的执行被控制流指令分割成多个基本块, 每个基本块只有一个入口和一个出口. 基本块内部的指令连续存储在正文段, 并且总是被顺序执行. 由于以分支指令结尾, 顺序执行的几个基本块很可能没有连续分布在存储空间上. E Roterr ber[10]提出了 Trace cache 结构将逻辑上顺序执行指令连续存放在一个额外的 cache 空间中, 以此获得更好的取值带宽.

图 1 中的有背景横线部分是通常的 Trace Cache 结构, 该结构已经在商业处理器如 Intel Pentium4 中实现, 阴影部分是多级优化框架添加的部分. 动态 profiling 技术弥补传统分支预测中的不足, 为获取程序行为提供准确, 实时的统计信息. Trace 处理单元根据每条 trace 内部控制流, 和指令数目的不同选择合适的优化级别和优化方法. 当 Trace 形成之后, 下次执行取值部件将从 trace cache 而不是传统的指令 cache 中获取指令, 获得更高的取指速度.

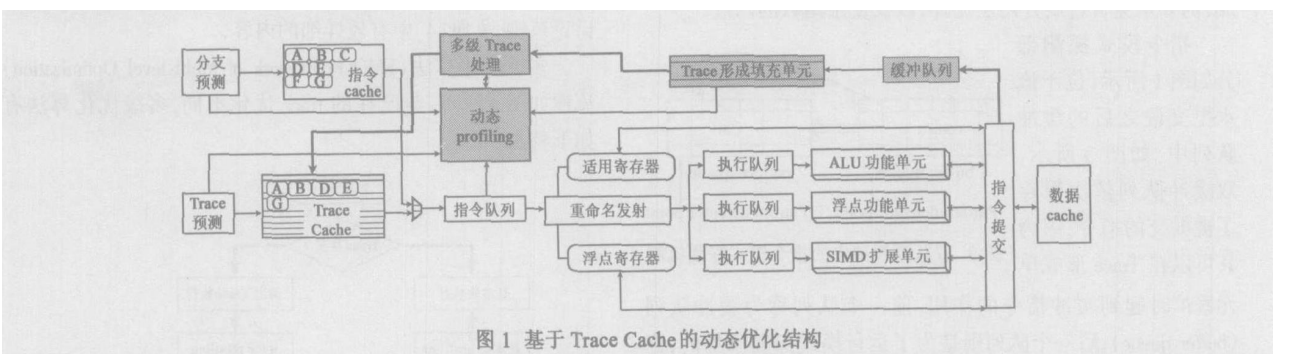

图 1　基于 Trace Cache 的动态优化结构

Trace cache 中保留的 Trace 是一组经常被连续执行的指令, 其中包含多个基本块和多条控制流指令. 一条 Trace 可以表示成为多个连续执行基本块的集合, 如图 2 所示的 T= 〈A, B, D, E, G〉. 从图 1 中可以看到在传统指令 cache 中基本块 A/ B 和基本块 D/ E 以及基本块 G 之间是不连续存储的. 而在 Trace Cache 中它们被连续存放. 并且在 Trace 形成以及以后的优化过程中 Trace〈A, B, D, E, G〉作为一个大的指令窗口, 为跨基本块边界的全局优化和调度提供条件.

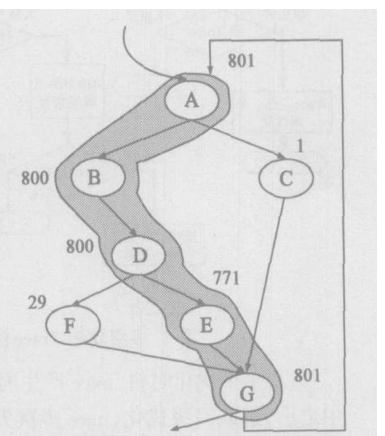

图 2　程序控制流图中的 Trace 选取

当程序从基本块 A 入口点开始执行时, Trace〈A, B, D, E, G〉是最可能的执行路径, 如果实际执行最终沿 Trace 最终从 G 退出, 称之为 Trace 命中. 如果在执行中分支跳转到基本块 C 或 F, Trace 没有执行完成, 称之为 Trace 失效. 从图 2 中不难看出, 包含基本块越多, 越长的 trace 越容易失效. 如果在 trace 中进行了全局优化, 失效 trace 必需以 check point 或者补偿代码等形式清除失效时错误执行路径上的指令.

Trace Cache 中基本块连接的结构与静态编译中为 VLIW 等宽发射处理器提出的 trace 调度机制非常类似[5]. Trace 的形成, 使得跨基本块边界的全局优化算法成为可能, 从而获得更多的指令级并行性. 许多静态编译优化和指令调度方法被修改应用于 Trace cache[11,12]. 已往的 trace 优化都是作为 trace 形成部件, 或者预处理部件的一部分来实现的. 一旦 trace 形成之后, 无法根据其后的执行行为对 trace 再进行修正和再优化. 本文提出的多级优化框架可以根据 profiling 采集的执行情况, 动态调整 trace 的优化级别, 根据 trace 中分支指令的类型和执行模式调整 trace 的长度和适用的优化方法.

3.2　指令行为模式与优化方法的选择

传统的 trace 优化只对部分代码的特定行为模式有

效[11, 12]. 静态编译器的 trace 调度和优化机制有足够的时间和空间资源在所有 trace 上尝试全部优化方法[5]. 而现有的动态 trace 调度多选择几种简单优化或者一种专用优化. 本文提出多级优化框架的一个设计依据就是利用 profiling 采集的信息, 猜测指令的行为模式, 依据指令行为模式选择优化方法和优化级别. 表 1 给出了现有框架中所关注的指令行为.

表 1 指令行为与优化可能

指令模式	描述
有偏向的直接条件分支指令	相邻基本块可连接
直接条件分支指令+ backward 分支指令	内层循环, 分支连续发生次数提示可以展开的循环体个数
有偏向的间接分支指令	有限的间接分支目标, 并且分支目标有严重的倾向性, 可类似直接条件分支优化
小函数调用指令	Call stack 可以记录被调用函数的大小, profiling 记录调用频度, 用于内嵌

直接无条件分支指令不会被作为基本块边界来收集 Profiling 信息. 每个基本块的 profiling 会依据结尾分支指令的类型分别记录基本块大小, 分支偏向度, 分支连续发生次数, 调用函数尺寸等信息. 在 trace 形成过程中, 累计的基本块大小用来预测可能获得的并行度, 抑制过大或常失效 trace 的形成, 例如避免盲目展开内层循环, 以及复杂函数的内嵌.

指令模式探测部分如图 1 所示, 位于流水提交段之后的缓冲队列中. 如图 3 所示, 双缓冲队列依次保存了被提交的指令. 因为其可以在 Trace 形成单元繁忙时起到缓冲指令的作用, 前一个队列称为缓冲队列(buffer queue); 后一个队列则是为了运行模式探测添加的, 称为模式队列(pattern queue).

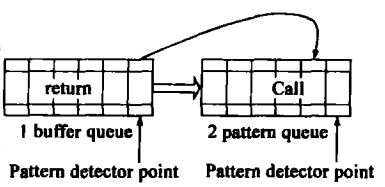

图 3 双队列缓冲中的指令模式探测机制

两个队列尾部作为模式探测的触发指针, 当有合适的控制类指令到达尾部触发位置时将引起 trace 形成之前的指令模式匹配, 按照可能的优化方式来指导 Trace 形成. 当图 3 中的一条回跳(backward) 分支指令运行到缓冲队列尾部时, 模式探测器将在模式队列中寻找分支目标, 以确定潜在的循环是否适合被展开; 而当 Call 指令运行到模式队列尾部时, 反过来将同时在缓冲和模式队列中寻找对应的 return 指令, 以确定函数调用是否能被嵌入.

值得注意的是, 指令模式在运行过程中是动态变化的. 内层循环连续执行次数的改变会影响到循环展开的次数, 而间接分支目标地址和偏向的改变将决定 trace 的存亡. 多级优化框架在 trace 形成之后依然监视基本块和 trace profiling 的变化, 调整优化方法和优化级别. 精确的 profile 是 trace 管理的关键.

3.3 多级优化方法

依据优化可能带来的性能提高以及在存储和运行时间上的消耗, 我们将优化分成不同级别. 低级优化适用范围广, 但获得的性能有限; 高级优化可以获得性能大幅提高的指令流, 但是只适用于部分指令流, 并且高级优化需要更长运行时间和存储空间. 现有的优化分为 3 级:

(1) 拷贝传递, 常量传递等. 由于指令集编码及数据依赖的影响, 动态指令流中仍然有这类优化机会.

(2) 低级循环展开. 展开的循环可以结合寄存器重命名, 进行指令调度或者指令替换以加快并行执行.

(3) 高级循环展开, 函数内嵌.

在实际设计中, 不同系统尤其是嵌入式或专用系统, 可以针对应用特点选择不同的优化算法和优化分级. 本文选用了目前在动态优化中使用较多的循环展开和函数内嵌作为深入研究对象, 目的是为了与传统单层次优化研究进行比对.

我们以 block_P 和 trace_P 来分别预测基本块和 trace 优化后可能获得的性能提高. Profiling 获得的基本块尺寸, 结尾分支指令信息和可用优化方法被用于计算这两个值. 例如以带有严重偏向性分支指令结尾的基本块, 经过较少执行次数之后其 block_P 值就增加到足以引发 trace 产生条件检测, 而总是以 50% 概率进行跳转的分支 block_P 值会维持在较低水平. Trace 的形成和再优化都依据这两个预测值来进行. Block_P 和 Trace_P 的计算是启发式的, 许多阈值和参数的选取参考了已有的研究, 详细的选取过程与方法超出了本文的讨论范围, 文献[4] 中有较详细的内容.

多级优化算法(FMO Framework of Multi-level Optimization) 流程如图 4 所示. 与已往的 trace 优化不同, 多级优化算法有如下特点:

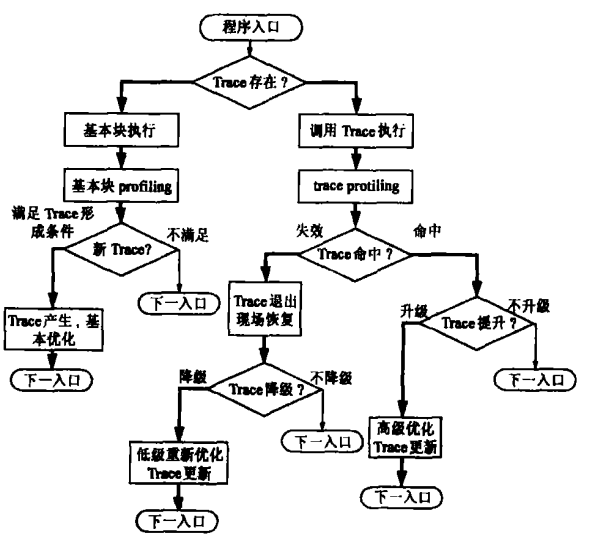

图 4 多级动态 Trace 优化算法流程

(1) 多个优化时机. trace 产生时的基本优化; trace 多次命中之后, 进入高级优化; trace 多次失效后恢复简单优化, 直到 trace 被清除. 而传统的动态优化只在 trace 形成时进行优化.

(2) 快速 trace 识别. 由于 trace 产生条件容易满足, 而初级优化相当简单易行, 多级优化算法能够快速发现程序执行热点, 尽可能早的利用 trace cache 获得性能提升.

(3) 准确的指令窗口扩大. 循环展开的遍数在 profiling 的指导下得到控制, 避免过大或者过小, 内嵌函数的尺寸和复

杂度也受限在合理范围内.

(4) 实时Trace 管理. 和已往优化中 trace 只优化一次,定期清空整个 trace cache 的做法不同,多级 trace 优化更精确的进行trace 管理. 代价是额外 profiling 空间和多级优化管理部件.

多级优化框架将分支指令的执行行为和适用优化方法之间建立的直接联系,并用额外的 profile 以及 trace 优化级别的调整来适应程序动态行为的变化,在 trace 大小和trace 命中率之间进行折中.

3.4　多级动态优化的实现与模拟

基于 simplescalar v30d,我们实现了如图 5 所示的多级动态优化框架.图中左侧是常见的超标量 RISC 处理器六段流水线结构.右侧是本文讨论的多级动态优化框架 FMO,以及 Trace Cache 机制.

处理器中添加实现了一个 SIMD 功能部件以及相关的寄存器,指令队列和保留站资源.这一部件和同时添加的指令用于模拟前言部分提到的处理器体系结构迁移.这也是目前很多处理器升级过程中常见的形式.

SIMD 功能部件具有类似 Intel MMX/SSE 部件的功能,能流水一拍完成乘加指令,128 位操作带宽可分为独立并行的两个 64 位或者 4 个 32 位或者 8 个 16 位操作. FMO 可以替换Trace 中的指令为 SIMD 指令,尤其是在内层循环的软流水(softwarepipelining)处理中可以将多个循环体(Iteraion)并行在 SIMD 部件上运行.

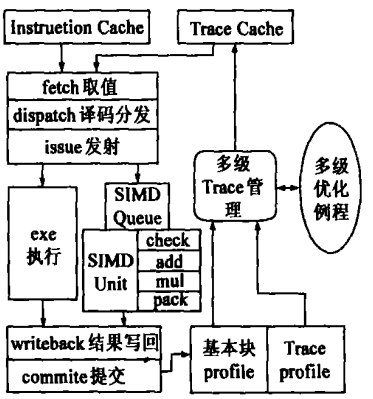

图 5　多级动态优化框架

一个可以保存 1024* 16(256* 64)条指令操作的 Trace Cache 用于保存优化调度后的 trace.Trace CacheLine 可以在 16-64 条指令之间动态改变.硬件实现的基本块 profile 机制用于检测程序执行热点,触发低级别 trace 的形成和简单优化.如果提交的指令取自 Trace Cache,Trace Profile 将把收集到的 trace 完成情况通知多级 trace 管理逻辑.由后者根据每条 trace 的不同情况,使用前面提到的多级优化算法调用存储在 Flash 或者 ROM 中的优化例程,将热点 trace 升级,或者降低失败 trace 级别.

传统的 Trace Cache 以提高处理器取值带宽为主要目标. FMO 通过对Trace 指令流的优化和调度来进一步发掘 trace 中的指令级并行(ILP).对于添加新功能部件,SIMD/多媒体扩展,甚至添加协处理器这样的体系结构扩展,FMO 还能以Trace 为单位进行指令选择和替换.进一步 FMO 可以应用到动态二进制翻译系统中,为在两种完全不同的硬件体系结构中运行相同指令集提供支持.本文的模拟选择 SIMD 扩展可以同时体现 FMO 在动态二进制指令调度和动态优化方面的优势.

图 6 是取自一维快速傅立叶变换 FFT 中的一段代码,在变量 i 取值小的时候,内层循环中的 if 语句总是执行 *a_rl* 数值的加法.图 6 右侧显示了内层循环的汇编代码,黑体标识了循环开始时的高频执行指令.

```
for (i = 0; i<ex; i++){
  lenb /= 2, timb = 0;
  for (j = 0; j<numb; j++){
        w = 0;
        for(k = 0; k < lenb; k++){

              j1 = timb + k;
              j2 = j1 + lenb;
              if (i < ex/2)
                    a_rl[j1] = a_rl[j1] + a_rl[j2];
              else
                    b_im[j1] = b_im[j1] + b_im[j2];

              w += numb;
        }timb += (2*lenb);
  }numb *= 2;
}
```

```
LoopK        slt   R2, R3, R4
             bne   R2, R0, LoopJ_END
             add   R11,R5,R3
             add   R12,R4,R11
             sra   R10, R9, 1
             slt   R10, R1, R10
             bne   R10, R0 ELSE1
             lw    R16, R11(R7)
             lw    R17, R12(R7)
             add   R16, R16, R17
             sw    R11(R7), R16
             j     Loop_Kend
ELSE1:       lw    R18, R11(R8)
             lw    R19, R12(R8)
             add R18, R18, R19
             sw    R11(R8), R11
LoopK_END:   add   R6, R6, R20
             add   R3, R3, 1
             j     LoopK
LoopJ_END:
```

图 6　FFT 代码中被优化指令示例

图 7 显示了在一般阈值控制算法,FMO 以及经过 FMO 对SIMD 指令调度之后的指令 trace.

阈值控制算法选择高频执行的指令形成 trace(图 7(*a*)).由于内层循环控制变量 K 的上界 lenb 是一个变量,不进行trace profile 的阈值控制算法很难确定内层循环的展开次数. FMO 形成的低级 trace 与阈值控制算法相同,但是经过 trace profile 之后,FMO 将内层循环的 trace 展开(图 7(*b*)),同时使用寄存器重命名和指令调度.进一步 FMO 将部分可并发的指令调度到 SIMD 部件(图 7(c)).两个循环体需要的 *a_rl* 数组元素分别被载入到 SIMD 寄存器中,一条 addvE 完成了两个数组元素运算.同时两次对变量 w 的递增也通过一条乘加指令 mulad 完成.图 6 中的 trace 从左至右在FMO 中级别依次增高.当然在经过进一步 trace profile 之后,循环可能被进一步展开,从而形成更高级别的 trace.

```
LoopK    slt  R2, R3, R4          LoopK
         bne  R2, R0, LoopJ_END
         add  R11,R5,R3
         add  R12,R4,R11
         lw   R16, R11(R7)
         lw   R17, R12(R7)
         add  R16, R16, R17
         sw   R11(R7), R16
         j    LoopK_end
         add  R6, R6, R30
LoopK_END: add R3, R3, 1
         j    LoopK
LoopJ_END:
```

(a) 阈值控制选择高频执行指令

```
slt  R2, R3, R4                    LoopK
bne  R2, R0, LoopJ_END
add  R11,R5,R3    add  R21,R5, R20
add  R12,R4,R11   add R22,R4, R21
lw   R16, R11(R7) lw R26, R21(R7)
lw   R17, R12(R7) lw R27, R22(R7)
add  R3, R3, 2    add R20, R3, 1
add  R6, R6, R30  add R6, R6, R30
add  R16, R16, R17 add R26, R26, R27
sw   R11(R7), R16 sw R21(R7), R26
j LoopK                            LoopJ_END
```

(b) FMO 两层循环展开

```
slt  R2, R3, R4
bne  R2, R0, LoopJ_END
lwvh v1, R11(R7)   lwvl v1, R21(R7)
lwvh v2, R12(R7)   lwvl v2, R22(R7)
add  R3, R3, 2     add R20, R3, 1
add  R11,R5,R3     add R21,R5, R20
add  R12,R4,R11    add R22,R4, R21
addvK v1, V1, V2   mpind M6,R6,R30,2
swvh R11(R7), v1   swvh R21(R7), v2
j LoopK
```

(c) FMO 两层循环并进行 SIMO 调度

图 7 优化后的 Trace

4 试验方法,结果与分析

本节将使用第一节中提到的 trace 优化标准来分析多级优化框架的性能,4 类优化框架被用于性能测试比较:

(1) Baseline. 模拟测试中目前最常用的阈值控制算法作为基准. 用简单的 trace 起始和终止阈值来控制 trace 形成,在其上运行简单的优化和调度[8]. 从图 8 可以看到,当调整 Baseline 阈值以获得大指令调度窗口时,越长的 trace 越容易引起 trace 失效.

(2) Loop. 只针对最重要的内存循环进行优化[4],依据本文动态框架 FMO 选取展开次数.

(3) Inline. FMO 仅针对可内嵌函数进行优化.

(4) FMO(Framework of Multi-level Optimization). 本文提出的多级优化框架,综合使用多种调度优化.

所用的测试程序选自 SPEC CPU2000 测试集,用 GCC v2.7 带"-O2,-finline-functions,-funroll-loops"优化开关编译. 修改后的 simplescalar3.0 sim-outorder 用于精确到 cycle 的 alpha 平台模拟.

图 8 表明虽然传统 baseline 简单易行,但是随着优化窗口增大,trace 失效概率显著增大. 从图 9 中可以看到,虽然 GCC 已经在静态编译期间完成了循环展开,动态优化仍然有很多运行机会. 但也可以看到,对于 *vpr* 和 *gcc* 这样频繁使用多目标间接分支的程序,动态循环展开可用的范围有限,仅仅针对循环级并行进行动态优化调度是不充分的. 从后续模拟结果中(图 11)可以看到,使用多级多种动态优化方法后,FMO 动态框架仍然可以发掘 *vpr* 和 *gcc* 的性能潜力.

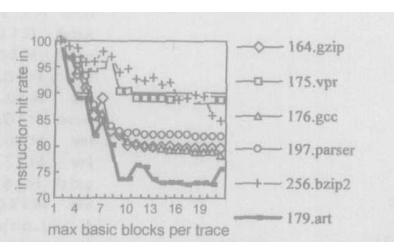

图 8 baseline trace 长度与命中率的关系

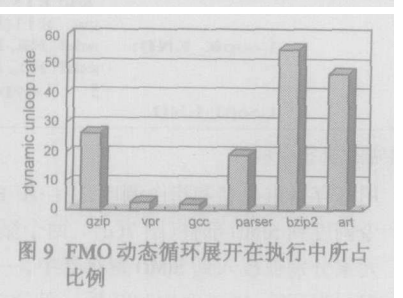

图 9 FMO 动态循环展开在执行中所占比例

在图 10 的模拟中,强制设定一个较大 trace 尺寸(23 条指令),对所有优化框架在的 trace 命中率比较. 所有结果以 baseline(阈值控制算法)的测试结果做了归一化处理. 可以看到使用动态框架,在大指令窗口下,trace 命中率获得了提高(Bzip2 最高 137%, *Vpr* 最低 2.8%,平均 50%).

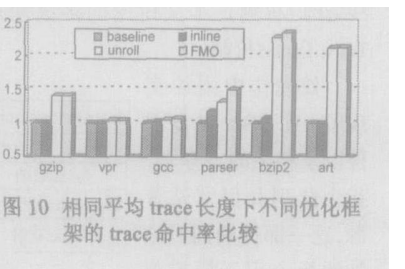

图 10 相同平均 trace 长度下不同优化框架的 trace 命中率比较

最终我们将优化的 trace 尽可能的调度到 SIMD 功能部件中,以利用 SIMD 的并行执行带宽来加速运行. 在 baseline 优化框架下,由于 trace 窗口小以及循环展开次数的不确定性,原有指令流很少能被替换成 SIMD 形式,所做的优化很少能利用到 SIMD 功能部件,大多数性能提升来自 trace cache 结构对取值带宽的优化. 而 FMO 则能够动态运行期间挖掘 SIMD 性能潜力. 同样我们 baseline 的 trace cache 执行作为基准来考察不同优化框架获得的加速比.

如图 11 所示,Bzip2 由于良好的程序结构,适当的函数尺寸获得了最大的加速比 28%. 而 gzip 和 gcc 由于大量多目标间接分支指令的存在而很难进行高层优化,加速比分别只有 5% 和 6%. Benchmark 的平均加速比仍然达到了 17%. 这一性能提高来自与多级动态优化框架以及 SIMD 功能部件,而正是动态优化框架选择合适的指令调度发掘并利用的 SIMD 功能部件.

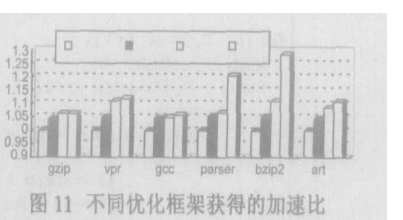

图 11 不同优化框架获得的加速比

图 12 显示了在图 11 的模拟过程中,不同优化级别 trace 在运行中所占的比例. Normal 表示指令未取自 trace cache. 从图中可以看到,大部分的指令执行为 trace cache 所捕获. 注意到所有高级别 trace 实际上都需要先成为低级别 trace,但是优化级别的提升有严格的条件限制以避免频繁 trace 失效,因此在 gzip 和 art 中 L3 级 trace 执行比

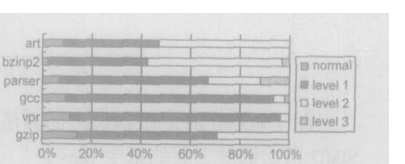

图 12 不同级别优化 trace 在动态执行中所占的比例

例小于 1%.回顾图 8,可以看到正是这两个测试程序,如果在传统 baseline 框架盲目引入高级优化,扩大优化窗口会严重降低 trace 命中率.多级优化算法以不同的优化级别很好的控制了优化的选取.

5 结束语

基于 trace cache 结构的多级动态优化框架能够根据程序动态执行行为选择合理的优化方法,并对 trace 进行更加细致的管理.在指令集扩展的过程中,程序无需重新编译也可以获得理想的性能提升.在实现中可以根据应用领域的不同,灵活选择优化方法和优化级别,具有实现了灵活性和可扩展性.未来工作将集中于更多的动态优化方法,并将优化目标由执行性能扩展到低功耗,低设计复杂度等领域.

参考文献:

[1] E R Altman, K Ebcioglu, M Gschwind, S Sathaye. Advances and future challenges in binary translation and optimization [J]. Proceedings of IEEE, 2001, 89(11): 1710- 1722.

[2] James C Dehnert, Brian K Grant, John P Banning, et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges [A]. Proceedings of the 2003 International Symposium on Code Generation and Optimization[C]. San Francisco: IEEE Computer Society Press, 2003. 15- 24.

[3] M M Merten, A Trick, et al. An architectural framework for runtime optimization[J]. IEEE Transactions on Computers, 2001. 50(6): 567- 589.

[4] Lian Li, Jingling Xue. A trace-based binary compilation framework for energy-aware computing[A]. Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools[C]. New York: ACM Press, 2004. 95- 106.

[5] J A Fisher. Trace scheduling: a technique for global microcode compaction[J]. IEEE Transaction on Computer. 1981, 30(7): 478- 490.

[6] Ebcioglu K, Altman E R. DAISY: dynamic compilation for 100% architectural compatibility[R]. IBM T J Watson Research Center, IBM Research Report: RC20538, 1996.

[7] T R Halfhill. Transmeta breaks x86 low-power barrier [R]. MicroDesign Resources, Microprocessor Report. (2000) NO: 2/14/00-01.

[8] Vasanth Balsa, E Duesterwald, S Banerjia. Dynamo: a transparent dynamic optimization system[A]. Proceedings of the ACM SIGPLAN' 00 Conference on Programming Language Design and Implementation[C]. New York: ACM Press, 2000. 1- 12.

[9] Thomas M Conte, Kishore N Menezes, Mary Ann Hirsch. Accurate and practical profile-driven compilation using the profile buffer[A]. Proceedings of the 29th Annual International Symposium on Microarchitecture[C]. Paris: IEEE Computer Society Press, 1996. 36- 45.

[10] Eric Rotenberg, Steve Bennett, James E Smith. Trace cache: a low latency approach to high bandwidth instruction fetching [A]. Proceedings of the 29th Annual International Symposium on Microarchitecure [C]. Paris: IEEE Computer Society Press, 1996. 24- 35.

[11] Quinn Jacobson, James E Smith. Trace preconstruction[A]. Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-00) [C]. Vancouver: IEEE Computer Society Press, 2000. 37- 46.

[12] Daniel Holmes Friendly, Sanjay Jeram Patel, Yale N Patt. Putting the full unit to work: dynamic optimizations for trace cache microprocessors[A]. In the proceedings of 31st Annual International Symposium on Microarchitecure[C]. Dallas: IEEE Computer Society Press, 1998. 173- 181.

作者简介:

唐遇星　男,1977 年生,博士生,主要研究方向:高性能处理器体系结构,二进制翻译,动态编译优化.E-mail:tyx@mudt.edu.cn.

邓　昆鸟　男,1976 年生,博士,副教授,主要研究方向:高性能处理器体系结构,二进制翻译与动态优化技术.

周兴铭　男,1938 年生,中国科学院院士,教授,博士生导师,主要研究方向:高性能计算机体系结构,并行与分布式数据库,CSCW.