

## 一种基于模拟器的动态性能数据的统计方法

朱 浩<sup>1,2</sup>, 薛志远<sup>1,2</sup>, 王东辉<sup>2</sup>, 侯朝焕<sup>2</sup>

(1 中国科学院大学, 北京 100049; 2 中国科学院 声学研究所, 北京 100190)

**摘 要:** 给出了一种基于模拟器的动态性能数据的统计方法. 通过该方法, 可以高效地实现对应用程序的分支执行概率、关键路径的统计. 这种方法相比于依靠编译器来实现的性能分析工具而言, 它具备实现简单, 复杂度低等优势.

**关键词:** 模拟器; 动态性能统计; 分支概率; 关键路径

**中图分类号:** TN7

**文献标识码:** A

**文章编号:** 1000-7180(2014)01-0124-04

## A Method of Dynamic Performance Statistics Based on Simulator

ZHU Hao<sup>1,2</sup>, XUE Zhi-yuan<sup>1,2</sup>, WANG Dong-hui<sup>2</sup>, HOU Chao-huan<sup>2</sup>

(1 University of Chinese Academy of Sciences, Beijing 100049, China;

2 Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China)

**Abstract:** This paper presents a method of dynamic performance statistical based on simulator. Through this method, the statistics of branch execution probability and critical path can be achieved. Compared with the analysis tools that implemented depends on the compiler, this method has the advantage of simplicity and low complexity.

**Key words:** simulator; dynamic performance data; branch execution probability; critical path

### 1 引言

动态性能统计<sup>[1]</sup>是评估应用程序执行性能是否满足需求, 编译优化手段是否有效以及程序中是否存在可优化部分的方法, 它的重要性不言而喻. 各大主流处理器厂商都提供适用于自身处理器的性能动态性能分析工具, 方便程序员编写出更高质量的代码, 如 Intel 推出 Intel Parallel Studio XE 2011 工具. AMD 公司同样推出了类似工具. 这些分析工具基本都是依赖编译器进行代码插桩实现, 因此增加了编译器的负担.

针对上述问题, 本文结合 SuperV DSP 处理器平台, 提出一种基于模拟器的动态性能统计方法, 它通过向汇编代码中插入伪指令, 并在模拟器端定义伪指令, 实现对分支概率、关键路径的快速统计.

### 2 伪指令定义

目前已有的集成开发工具如 Microsoft 的 Visual Studio 等, 它们通常是依托编译器对目标源程序进行代码插桩, 配合本地硬件的反馈信息实现对函数访问次数、运行时间等评估. 本文在动态性能分析中进行的相关研究均是在模拟执行平台上而非与处理器直接交互实现. 尽管这种模式下无法获取硬件的运行数据, 但仍可以利用伪指令插桩辅助动态性能的统计. 伪指令是按照目标处理器指令集编码格式独立设计的, 不可与目标指令集系统存在冲突, 只运行在模拟平台上的指令. 它们实现简单, 并提供额外功能来辅助数据计算与分析. 在关键路径分析过程中, 本文共定义了 Leaf、Interior、RootOver、FuncName 这四条伪指令, 具体使用规则在后文中详细描述.

收稿日期: 2013-04-29; 修回日期: 2013-05-22

基金项目: 国家“九七三”计划资助项目(2009CB320202)

### 3 分支概率统计

对分支概率的统计是从统计学的角度,利用现有数据评估应用程序在各种情况下运行时控制流的走向,它在传统全局指令调度算法,硬件转移预测算法中获得了广泛应用.一般可见的分支概率统计是在编译器端实现,编译器首先将高级语言切割为多个基本块并构成控制流图,再通过 Profiling 对控制流图的边进行概率加权,这些步骤在编译优化前进行的.然而,目标源程序很难与前端划分的基本块之间建立映射关系,这用统计的分支概率对后续的优化不一定有效,因此,从汇编级构建分支概率统计模型变得十分必要. SuperV DSP 处理器平台中有表 1 中所示的 6 条分支类指令,分支概率的统计等价于应用程序执行过程中,每条分支指令推断寄存器的值为 1 的概率及汇编级控制流图的概率加权.

表 1 SuperV DSP 支持的分支指令

| 分支指令  | 功能描述    |
|-------|---------|
| Call  | 调用      |
| Calli | 间接调用    |
| J     | 相对跳转    |
| Ji    | 间接跳转    |
| Ret   | Call 返回 |
| Rfi   | 中断或异常返回 |

计算分支指令执行概率时,处理器内核模型需要为每条分支指令提供独立的计数器,并在运行时记录它们是否被执行.而处理器内核模型判断分支指令与计数器之间一对一的映射关系时,只能依赖分支指令对应程序计数器的值.因此,处理器内核模型中流水线开始执行前,必须确定各分支指令在存储系统中被分配的地址,这就需要获得汇编器和链接器的支持.然而,应用程序中的分支指令通常占据较大比重,如果在运行时进行顺序判断,算法复杂度将为  $O(n)$  ( $n$  为分支指令总条数),如此庞大的时间开销势必影响动态分析工具执行性能,这是难以接受的,如何降低判断过程中的算法复杂度成为了关键因素.哈希算法通常是解决这类问题的首选.但是,分支指令在地址空间上一般是离散分布的,直接对它们的内存地址进行哈希算法所带来的存储空间浪费是十分庞大.诸如 Visual Studio 等在编译器架构中以及 Intel 等公司 CPU 处理器体系结构中,在处理分支指令时采用了跳转表的概念.尽管 SuperV

DSP 平台中并未采用该结构,但却可以通过在汇编期对比汇编代码进行修改来实现,并将汇编代码中存在的与跳转表中的分支指令建立一一映射关系,这样进行分支概率统计时等价于对跳转表内分支指令执行概率的统计,而跳转表内的分支指令在地址空间上是连续分布的,全散列哈希算法将不会导致存储空间的浪费.图 1 展示了添加跳转表前后程序结构发生的变化,可以看到整个过程中仅需修改已有分支指令的目标地址,而 SuperV DSP 处理器相对跳转指令  $Ji$  的寻址空间为 21 位,链接期只需为跳转表分配合适的地址空间即可满足使用,整个过程中对应用程序的额外开销为  $O(n)$  ( $n$  为应用程序执行完时分支指令总数),这样的开销是可以接受的.

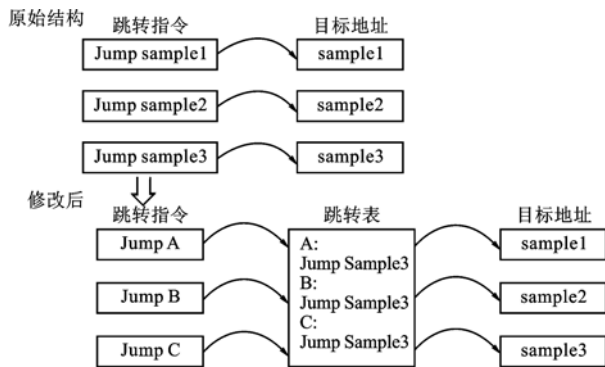


图 1 分支跳转表

跳转表建立完成后,它的每一条表项将对应一个独立的计数器,32 位指令系统的 SuperV DSP 平台下即可建立如下全散列哈希函数.其中, address 代表跳转表地址空间中的首地址, ireg 是跳转表内每条分支指令的存储地址, value 为对应计数器的指针,这种算法的时间复杂度为  $O(1)$ ,效率最高.

$$\text{value} = \text{hash}[(\text{ireg} - \text{address}) / 4];$$

本文中所生成的跳转表中分支指令的序关系是利用深度优先算法遍历控制流图中各节点顺序生成,动态分析工具则按照跳转表中分支指令的逻辑序关系生成概率数据报表,当概率数据生成后再次遍历控制流图即可完成加权.

图 2 中显示了整个编译后优化平台统计分支概率的流程,步骤如下:

① 汇编期平台中分析汇编文件,生成汇编级控制流图;

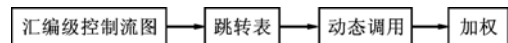


图 2 分支概率统计流程

② 遍历汇编级控制流图,获取分支指令的序关

系,修改缓存中的汇编信息以及跳转表,继续完成汇编、链接并生成可执行文件;

③动态分析工具计算得出分支概率报表,汇编期平台再次遍历汇编级控制流图,顺序添加概率数据,且不包含分支指令的节点所引出边的执行概率默认为 1.

#### 4 关键路径统计

函数是一个定义了特定功能的语句块或作用域,高级程序设计语言对函数调用无一例外的提供了支持.C 语言编写的应用程序通常是由多个函数组成,包括唯一的一个入口函数,用于异常处理的出口函数,多个节点函数.其中不再调用其它函数的节点函数又称作叶子函数.叶子函数分为两种,一种是静态代码中不存在函数调用指令,另一种则是动态过程中数据激励致使分支判断失败而不发生函数调用.应用程序执行轨迹或控制流就是从入口函数经过节点函数到达出口函数或叶子函数时,反复进行的压栈和入栈操作形成的访问链.分析关键路径就是通过评估访问链中各子路径的处理器执行时间,以试图找出应用程序中的瓶颈,最终交由程序员或优化工具挖掘出可优化的部分.现有的性能分析工具在查找关键路径时,一般是以函数为单位配合编译器进行代码插桩完成的,这主要因为函数内基本块内指令一般较少,分析它们之间的关键路径对代码优化而言意义并不大.本文进行的关键路径分析是路径剖析(Programs Profiling)[Ball]技术的一种,路径剖析在动态分析领域占据十分重要的地位,它用于分析、计算应用程序中全部执行路径以及执行次数,这些性能数据被广泛的应用在编译优化等多方面研究工作中.现有技术在实现方案上较为一致,一般是向应用程序的控制流图 CFG<sup>[2]</sup>(Control Flow Graph)中插入探针(probe)或代码插桩,最后在动态执行过程中收集、计算探针或桩代码统计的性能信息,来计算出各条路径及执行次数.如处理不回边(backedge)的路径时,Ball<sup>[3-4]</sup>等人提出的过程内单变量剖析方法 EPP(efficient path profiling);处理包含循环路径时,Tallam<sup>[5]</sup>提出的近似剖析方法,以及 Roy<sup>[6]</sup>等提出的改进方法,都在路径剖析上取得较佳效果.然而,在处理结构相复杂、循环体较多的应用程序时,这些方法仍有不足.归其原因,不难发现这些方法均是对高层语言的行为级处理,它们在诸多方面受到限制.针对这些情况,本文进行的关键路径分析仍然是通过模拟执行实现的,这样的

好处在于可以在不影响程序逻辑的基础上强化部分指令,而且在代码插桩时处理起来更加灵活.与路径剖析中统计次数的方法不同,本文评估一条执行路径是否有效,是以运行的平均时间作为依据.

SuperV DSP 处理器指令集中,直接调用指令 Call 和间接调用指令 Calli 用于函数调用,Ret 指令用于函数返回.在此基础上,本文在动态性能分析工具中提供下列功能来辅助关键路径的分析:

①建立全局计数器 total\_cycles. 执行流水线时,根据每个指令并行包中指令的最大执行 cycles 数模拟计算当前时间点:

$$\text{time} = \frac{\text{total\_cycles}}{\text{frequency}} \times 10^3 \text{ms} (\text{frequency 为目标}$$

机器主频)

②建立函数调用、返回函数的时间戳.集成开发环境 Visual Studio 统计函数执行时间时,是通过编译器向目标函数的入口基本块和出口基本块中添加首尾时间戳,它最终获取的是 PC 机中 CPU 的硬件时间.本文的性能分析是依托于模拟平台实现,指令功能描述是在高级语言层完成,因此,可以不利用编译器进行代码插桩,而可以通过修改 Call、Calli 以及 Ret 指令的功能描述完成时间戳的添加.图 3(a)展示一条完成的访问链,实线部分代表函数间的逐层调用,虚线部分代表访问链到达叶子函数后的逐层返回.当发生函数调用时,将当前时间点 time(i),即 total\_cycle 的值存储入栈结构 TimeStack 中,当目标函数执行完成,执行 Ret 指令发生返回,TimeStack 执行一次出栈操作并将结果赋值给 total\_cycle,整个过程如图 3(b)所示.图 3(c)中为修改函数调用、函数返回指令功能的伪代码.

程序执行完叶子函数或出口函数时代表一条路径的结束.因此,模拟执行部分需提供机制判断当前函数是否为叶子函数或出口函数.前文中对叶子函数已做出定义,因此静态分析并不能明确一个函数是否为叶子函数,需获得动态分析的辅助,因此,添加了三条伪指令,Leaf,Interior,RootOver.处理器内核模型中添加一个全局 bool 型变量 WhetherLeaf,为 true 时代表进入叶子函数,为 false 时代表为非叶子函数.Leaf 和 Interior 指令就是用于修改变量 WhetherLeaf 的值,Leaf 将 WhetherLeaf 置 false,Interior 则相反,RootOver 伪指令用于关键路径输出.汇编期处理时,在每个函数的入口基本块内添加 Leaf 指令,推断寄存器为 Pr0 代表确定执行,在函数内每条函数调用指令之后添加 Interior 指

令,且推断寄存器与函数调用指令一致,在函数内的函数返回指令之前添加 RootOver 指令,推断寄存器同样为 Pr0. 因此,一旦进入目标函数,WhetherLeaf 将被置为 true,当函数内的函数调用指令被执

行后,WhetherLeaf 的值则被置为 false,这样在动态执行过程中即可明确路径是否执行到叶子函数,图 4 显示了这三条伪指令的插桩方式.

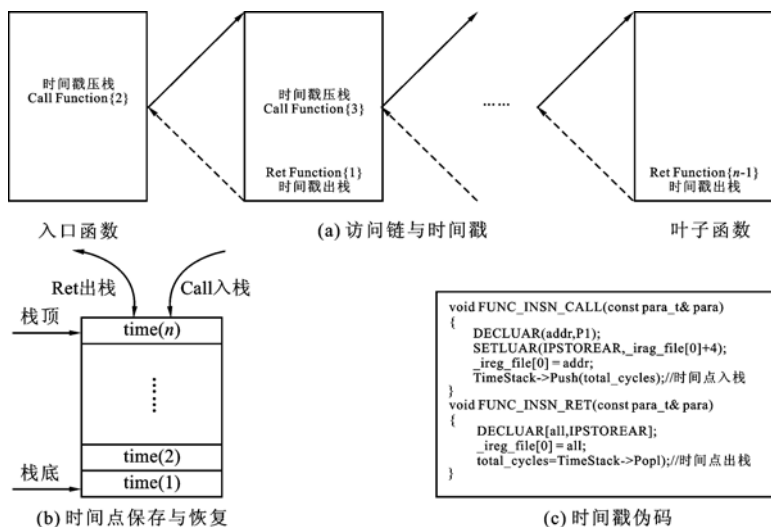


图 3 时间戳实现

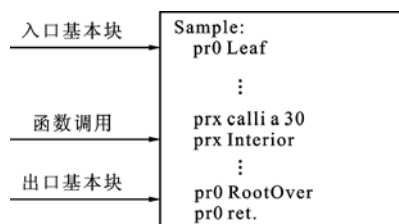


图 4 伪指令插桩

建立函数名与程序指针的映射. 尽管 ELF 格式文件中提供了函数名称信息,但处理器内核模型运

行过程中进行反复判断、读取势必影响性能. 因此,添加伪指令 FuncName 来辅助函数名称的读取. 下图 5 展示了全部的流程,首先,汇编文件处理时将伪指令 FuncName 插入各函数入口基本块,指令编码中存储的值与预处理表中的顺序一致;其次,内核模型部分建立数据结构 MapTable<sub>t</sub>,加载预处理表后建立映射表 MapTable;最后,内核模型提供全局 unsigned int 型变量 MapPoint,执行 FuncName 指令时将存储的值的值赋给 MapPoint.

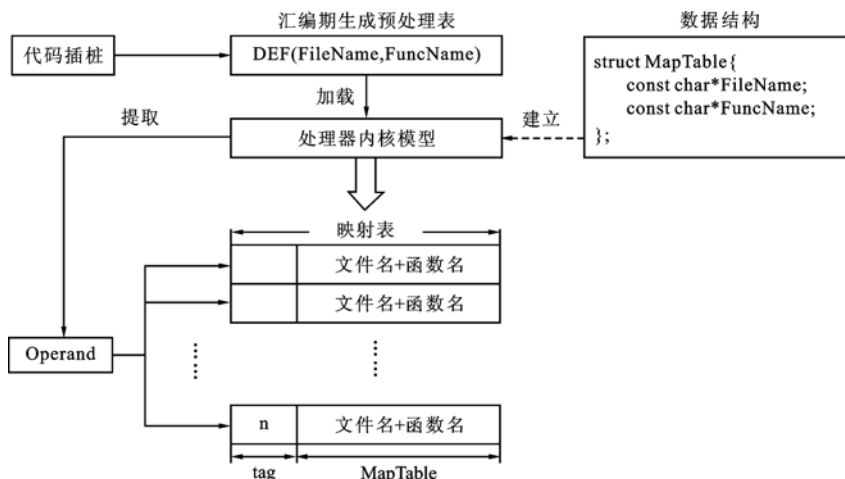


图 5 MapTable 建立

当程序运行到叶子函数时,需沿路输出路径中的各个节点. 通常,操作系统中建立用于栈区来存储应用程序的活动记录,软件只需定义简单的数据结构即可获取函数调用关系,前文提到的伪指令 Ro-

otOver 的功能正是用于输出路径上的节点. 如图 6 所示,处理器内核中建立如下数据结构,其中 MAX\_STACK\_LEVEL 代表目标平台上支持的最大栈深度,有符号整形变量 call\_stack\_idx 初始值为-1,

用于指向软件调用栈 call\_stack. 程序运行时出现函数调用, 当运行到 FuncName 伪指令时将函数调用指令地址、映射表的索引值构建成 StackTable 变量后, 一起存储入 call\_stack 中, call\_stack\_idx 加 1. 当出现函数返回时, call\_stack\_idx 减 1.

```

Struct StackTable {
    unsigned int MapIdx; //映射表索引
    unsigned int addr; //程序地址
};

StackTable call_stack[MAX_STACK_LEVEL];
int call_stack_idx=-1;

```

图 6 StackTable 数据结构

执行 RootOver 指令时, 首先判断 WhetherLeaf 变量是否为 true, 否则直接退出. 尽管函数之间不可避免的存在递归、相互间的嵌套调用以及同一函数在不同代码块可能调用相同的目标函数, 但 call\_stack\_idx 的伴随着出栈、压栈操作反复累加、递减已删除冗余, 因此, 输出路径时只需根据 call\_stack\_idx 的值顺序输出 call\_stack 内的 MapIdx 的值并完成函数名的映射, 而 total\_cycle 计算出的时间点是运行时的时间开销. 下面是一条路径的示例:

Time: 1.312 s

Loader → main decodeAACfile → NeAACDecDecode → aac\_frame\_decode → output\_to\_PCM → to\_PCM\_16bit

函数调用路径输出. 每个函数在执行时可被多种途径调用, 如函数指针等, 这些在静态分析中是无法获取的. 对各个函数的调用是分布在各个关键路径之中, 因此, 只需在 Leaf 指令中添加额外功能即可. Leaf 指令读取当前 StackTable 中已有函数的 MapIdx 值形成单向链表, 判断是否已经存在重复路径, 如果存在, 考虑到每条路径下执行时间可能不一致, 只能通过计算出平均时间来反应路径开销, 计算公式为  $(\sum_{i=0}^{n-1} \text{Time}(i))/n$ ,  $n$  为路径总数,  $\text{Time}(i)$  为路径  $i$  的执行时间.

程序员可评估出应用程序执行时的关键路径以及函数的各调用路径的时间代价, 从算法的角度进一步优化来提高程序最终的执行效率.

## 5 结束语

本文介绍了一种基于模拟器的动态性能数据统计方法. 通过建立跳转表来跟踪分支指令, 可较便捷的获取应用程序的分支概率. 通过定义伪指令, 添加时间戳以及跟踪程序进行函数调用时的入栈、出栈操作, 可获取应用程序中执行代价较大的访问路径, 并最终交由程序员辅助优化. 从实现方法上来看, 在模拟器端实现动态性能分析较利用编译器进行代码插桩更灵活, 且工作量小.

## 参考文献:

- [1] Brdeniag D, Garnett T, Amarasing S. An infrastructure for adaptive dynamic optimization[C]// International Symposium on Code Generation and Optimization, San Francisco, California: IEEE, 2003:265-275.
- [2] Ferrante Jeanne, Karl J Ottenstein, Joe D Warren. The program dependence graph and its use in optimization[J]. ACM TOPLAS, 1987, 9(3):319-349.
- [3] Ball T, Larus JR. Optimally profiling and tracing programs[J]. ACM Trans on Programming Languages and Systems, 1994, 16(4):1319-1360.
- [4] Ball T, Larus JR. Efficient path profiling[C]//Proc of the 29 th Annual ACM/IEEE Int' Syrup on Microarchitecture. Washington: IEEE Computer Society, 1996:46-57.
- [5] Tallam S, Zhang X, Gupta R. Extending path profiling across loop backedges and procedure boundaries[C]// Proc of the Int' Symptom Code Generation and Optimization: Feedback Directed and Runtime Optimization (CGO). Washington: IEEE Computer Society, 2004:251-264.

## 作者简介:

朱 浩 男, (1987-), 博士研究生. 研究方向为代码优化.