计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
50(5): 1100-1109, 2013

# 一种监测函数语义信息访存地址序列的方法

陈荔城[1,2] 崔泽汉[1,2] 包云岗[1] 陈明宇[1] 沈林峰[3] 梁 祺[3]
[1](中国科学院计算技术研究所计算机体系结构国家重点实验室(筹) 北京 100190)
[2](中国科学院大学 北京 100049)
[3](IBM 中国系统与科技开发中心 北京 100193)
(chenlicheng@ict.ac.cn)

# An Approach for Monitoring Memory Address Traces with Functional Semantic Information

Chen Licheng[1,2], Cui Zehan[1,2], Bao Yungang[1], Chen Mingyu[1], Shen Linfeng[3], and Liang Qi[3]
[1](*State Key Laboratory of Computer Architecture (preparatory), Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)
[2](*University of Chinese Academy of Sciences, Beijing* 100049)
[3](*IBM China Systems and Technology Laboratory, Beijing* 100193)

**Abstract** Accurate monitoring memory traces of applications running on real systems is the basis of memory system scheduling and architecture optimization. HMTT is a hybrid hardware/software memory traces tracker system, which is able to track full-system memory traces in real time. But there exists a semantic gap between memory traces and high level application events, such as synchronization problem between upper functional execution flow and memory traces. In this paper, we propose a hybrid hardware/software approach for monitoring memory traces with functional level semantic information. It directly modifies the process image in memory with binary instrumentation at the beginning of a process, by respectively inserting an extra tag memory access instruction at the entry and exit of each target function, which will be tracked and identified by HMTT. With binary instrumentation, there is no need to require source code of applications, no need to re-compile or re-link applications, and the run time overhead is quite low. The experimental results show that our hybrid hardware/software approach can effectively track memory traces with functional semantic information. For memory intensive applications in SPECCPU2006, the average run time overhead is 62%, and the pure software approach run time overhead with Pin is at least 10.4 times, compared with the original run time.

**Key words** hybrid memory trace toolkit (HMTT); memory traces; functional level semantic gap; binary instrumentation; executable and linkable format (ELF); tag memory access

**摘 要** 准确地获取应用程序在真实系统上运行的访存地址序列(traces)是进行内存系统调度及结构优化的基础.HMTT 是自主研发的软硬件结合的内存监测分析系统,能够实时获取完整的全系统访存 traces.但是得到的 traces 与应用程序上层事件之间存在语义鸿沟问题,比如上层函数执行流与访存 traces 的同步问题.针对该问题提出了一种软硬件结合获取包含函数级别语义信息访存 traces 的方法,软件方面通过二进制插桩的方式,直接修改内存中的进程映像,在目标函数的入口及出口各插入标记

**收稿日期**:2011-08-30;**修回日期**:2012-02-21
**基金项目**:国家自然科学基金项目(60925009,60921002,60903046,61272132);国家"九七三"重点基础研究发展计划基金项目(2011CB302502);中国科学院战略性先导专项课题(XDA06010401);IBM 共享大学研究(SUR)项目

tag 访存指令,进而能够被 HMTT 卡监测并识别.采用二进制插桩不需要程序的源代码,不需要对程序重新编译链接,而且引入的运行开销很小.实验表明采用软硬件结合的方式能够有效地获取包含函数级别语义信息的访存 traces,对于 SPECCPU2006 中的访存密集型程序引入的性能开销只是原程序的 62%,而使用 Pin 工具的纯软件方式获取访存 traces 将导致至少 10.4 倍的性能开销.

**关键词** HMTT;访存 traces;函数级别语义鸿沟;二进制插桩;ELF;tag 访存

**中图法分类号** TP333

随着处理器上集成的核个数不断增加,内存作为多核系统的共享资源,将受到越来越大的访问压力,内存系统已成为多核系统性能的主要瓶颈[1].因此准确地获取程序在真实系统上运行的访存地址序列(traces),并基于此对程序执行过程中的访存行为进行分析变得至关重要,这是进行内存系统调度及结构优化的基础.

混合内存监测系统(hybrid memory trace toolkit,HMTT)[2-3]是我们自主研发的一个独立于平台的软硬件结合的访存行为监测与分析系统,通过硬件侦听内存总线上的访存信号,实时获取全系统访存物理地址 traces,通过反查页表获取进程标识和虚拟地址.采用硬件侦听方案,对程序透明,不会引入额外的访存干扰,能够得到包括物理地址、读/写、时间戳、进程号、虚拟地址等丰富信息.目前 HMTT 已经实现到第 3 个版本,支持 DDR3-800(工作频率为 400 MHz),HMTT 的主页为 http://asg.ict.ac.cn/hmtt/.图 1 为 HMTT 的整体框架,详细介绍请参考文献[2].

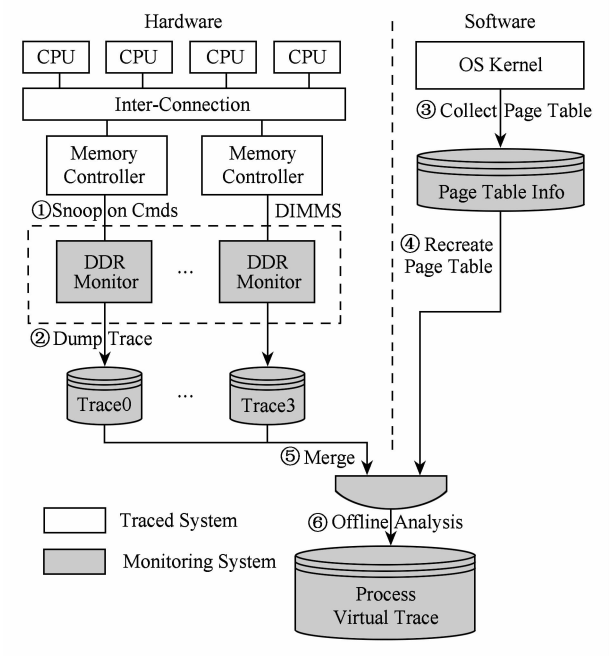



Fig. 1 HMTT framework.

图 1 HMTT 框架

然而基于硬件侦听的 HMTT 卡无法直接获取程序执行过程中的高层事件流信息,这被称为硬件侦听与高层事件的语义鸿沟问题(semantic gap),如图 2 所示.图 2(a)为 HMTT 获取的一个访存地址 traces,但不包含任何程序执行的高层语义信息.从程序员的角度来看,一个程序(或进程)的执行过程可以看作是高层事件的执行序列,比如函数调用序列,基本块执行序列、甚至精细到指令执行序列.按照高层事件流可以将一个进程的执行过程划分成不同的阶段,比如进入一个函数,从函数返回等,如图 2(b)所示.由于在不同的阶段,程序往往具有不同的访存行为特征,如果能够将底层硬件获取的访存地址流与高层事件序列进行同步并关联起来,那么我们就可以在更细粒度上分析程序在不同阶段的访存行为,比如在函数级别,如图 2(c)所示,从而对那些导致大量不规则访存的函数进行重点优化.

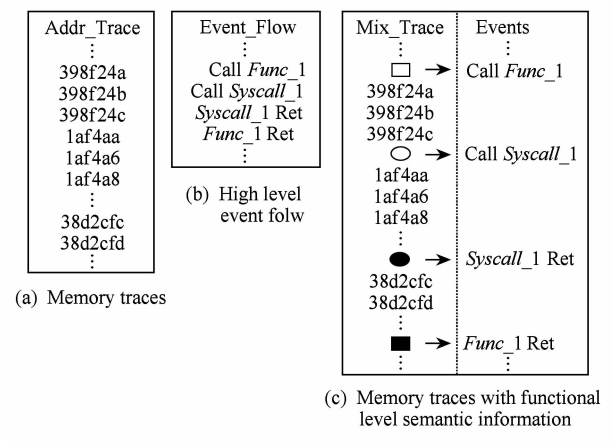



Fig. 2 Semantic gap between memory traces and high level event flow.

图 2 访存 traces 与高层事件的语义鸿沟问题

为了解决上述函数级别语义鸿沟问题,本文实现对 HMTT 系统功能的扩展,提出了一种基于二进制插桩技术实现获取函数级别语义信息的访存地址序列的方法(functional level binary instrumenta-

tion toolkit for HMTT, HMTT_FBI). 通过直接修改内存中的进程映像(process image),在目标函数的入口及出口位置分别插入一条额外的 tag 访存指令,这些 tag 访存与程序正常的访存都将被 HMTT 硬件侦听到. 通过解析不同 tag 访存与函数入口、出口之间的映射关系,就能够将底层的访存 traces 与高层的函数执行序列关联起来,从而实现对访存 traces 的函数级别分割.

我们通过 Linux 系统提供的 ptrace API 来获取对进程的控制权. 插桩实现是在每个插桩点把原有指令替换为一条跳转指令. 我们把对目标函数的调用指令选为插桩点,这样能够避免由于函数存在多个可能出口点带来的复杂处理. 插桩代码执行完后,通过一条跳转指令将执行流跳转回到原来的调用位置. 进程除了在执行到每个插桩点时额外增加的两次跳转以及执行插桩代码本身的开销外,没有引入其他的开销. 而传统的动态插桩工具,如 Pin[4], DynamoRIO[5] 等,则需要在程序的整个执行过程中完全控制进程,即时编译(just in time compilation)生成包含插桩代码的代码块. 这种实现不仅导致比较严重的性能开销,还会引入比较严重的访存干扰.

实验表明,基于二进制插桩的实现不仅开销低,而且只引入少量的干扰访存. 对于访存密集型的程序,HMTT_FBI 增加的平均运行时间开销为 62%,相对而言,Pin 实现的开销至少为原程序正常运行时间的 10.4 倍. 另外 HMTT_FBI 引入的平均干扰访存为原程序访存量的 52%.

## 1 实 现

### 1.1 整体框架

图 3 示出 HMTT_FBI 的整体框架,这里涉及 2

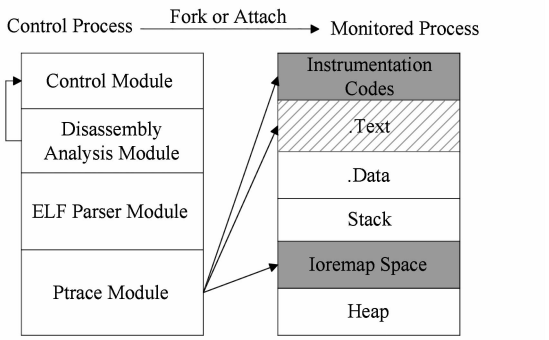



Fig. 3 The overall framework of HMTT_FBI.

图 3 HMTT_FBI 的整体框架

个进程,一个运行特定的应用程序,称为被监控进程;另一个进程负责分析被监控进程,并向被监控进程插桩代码,称为控制进程. 控制进程通过 ptrace API 获取对被监控进程的控制,支持 2 种模式:1)控制进程创建被监控进程,这样允许控制进程从被监控进程执行的最开始位置就对其进行插桩,从而能够收集到被监控进程的完整信息;2)被监控进程正在运行中,控制进程附着(attach)到被监控进程上,然后对其进行插桩,这样允许我们灵活设置只对感兴趣的运行区间进行插桩并收集信息,而不需要重新启动进程. 这对于监测那些长期运行的进程是很有好处的.

控制进程主要包括 4 个模块:控制模块、反汇编分析模块、可执行连接格式(executable and linkable format, ELF)解析模块和 ptrace 模块,如图 3 所示. 其中,控制模块向用户提供交互式操作接口,用户可以在这里指定附着(attach)到被监控进程上,获取被监控进程的状态信息(如寄存器的值、某段地址空间内的值),对被监控进程进行代码插桩,恢复被监控进程的进程映像,从被监控进程断开(detach)控制. 这些都是通过 ptrace 模块提供的接口实现的. ptrace 模块对 Linux 提供的 ptrace API 进行面向对象的封装,使用户能够更方便地对被监控进程进行操作. ELF 解析模块则负责分析被监控进程的 ELF[6-9] 格式的可执行文件,获取被监控进程代码段的位置和大小,通过分析 ELF 文件内的符号表 (.sym)能够进一步得到每个函数的名称、入口地址及代码大小,进而获取每个函数的二进制代码. 反汇编分析模块[10] 则负责对得到的目标函数的二进制代码进行反汇编,主要分析函数体内每条指令的类型、位置(PC 值)、大小、操作数等;判断一条指令是否为函数调用指令、分支指令;如果是则进一步分析得到函数调用指令的目标函数地址或分支指令的目标地址. 这些信息将被用于后面对被监控进程的代码插桩. 反汇编分析模块基于一个开源的反汇编工具 udis86[11],支持 x86 和 x86_64 指令集.

为了存放插桩代码,需要在被监控进程的地址空间内额外开辟一块空间,这通过往被监控进程的代码段内临时插桩一段 *mmap*() 调用代码来实现. 在插桩这段代码之前需要先将代码段内原来的内容取回保存,还需要保护进程的寄存器状态;当这段代码执行完毕,控制权将重新回到控制进程,之后恢复被监控进程的代码段内容和寄存器状态. 以同样的

方式,将系统的一个虚拟设备 ioremap 映射到被监控进程的地址空间内,这段空间的特点是对其访问不通过处理器高速缓存(cache),将被用于作为对应高层函数事件的 tag 访存,这将在 1.3 节介绍.图 3 中被监控进程的两个阴影较深部分是我们额外再分配的两段地址空间,而斜线部分(代码段)部分表示需要对其进行修改(通过二进制插桩实现),白色部分则表示不需要作任何改动.

### 1.2 二进制插桩实现

传统的函数级别插桩都只提供对函数的入口进行插桩的接口[4-5,12].由于函数的入口是唯一的,通过将函数体内的前几条指令替换成跳转指令就可以将控制转到插桩代码区域,从而实现收集所有与函数入口相关的信息(比如统计函数被调用次数).而函数的出口往往不是唯一的,而是可能有多个,比如下面一个简单的计算斐波那契(Fibonacci)数的递归实现函数就包含 3 个可能的函数出口.

```
① int fib(int n)
② {
③    if(n==0)
④       return 0;
⑤    else if(n==1)
⑥       return1;
⑦    else
⑧       return fib(n-1)+fib(n-2);
⑨ }
```

多个可能的函数出口使得直接插桩非常困难,通过复杂的指令流分析也许可以得到所有的出口位置,然后分别在这些点上进行代码插桩.这样不仅实现复杂,而且插桩的开销会变得很大.因此我们不直接在目标函数体内进行插桩,相反,我们在源头上解决问题,选择目标函数的调用指令作为插桩点,通过将其替换为无条件跳转(jmp)指令,把控制转移到插桩代码位置,然后将目标函数调用指令重新安排到插桩代码区域,并在该调用指令的上方和下方分别安排 tag 访存指令.在插桩代码区域内安排 tag 访存指令不会影响原程序的其他代码,保证程序的正确执行.

注意对函数的出口插桩 tag 访存指令是必须的.这是因为一个程序中函数之间的相互调用关系往往比较复杂,如果只标记出函数的入口,将无法判断下一个函数是在上一个函数体之内被调用,还是在上一个函数结束之后被调用,这样就无法实现对访存 traces 以函数为单位进行分割.

图 4 示出一个函数调用的正常执行流程.当执行到函数调用(call)指令时,函数的返回地址(即 call 指令的下一条指令的 PC 值)将被压入栈顶,然后转到函数入口地址往下执行,待函数执行到返回(ret)指令,根据之前放入栈的返回地址,转回到函数的调用点位置之后继续执行.

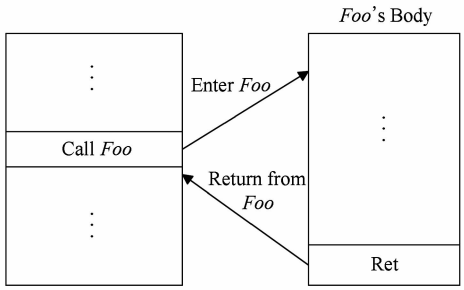



Fig. 4 Normal execution flow of a function.

图 4 函数正常执行流程

图 5 示出插桩后的目标函数执行流程.首先目标函数的调用指令已被替换成一条无条件跳转指令,将控制转移到对应的插桩代码区域.在这里,先执行函数的入口 tag 访存,然后重新调用目标函数,等到目标函数执行完毕,将返回到这个新的调用点位置.紧接着执行函数的出口 tag 访存,最后通过一条无条件跳转指令将控制重新转回到原来函数的调用点位置.在对函数执行额外的 tag 访存之前需要保护进程的状态,执行完之后恢复进程的状态,保证不影响进程的正确执行.有一点需要注意,由于此时新的函数调用指令的位置(PC 值)已经变化,而使用相对偏移的函数调用指令是根据下一条指令的 PC 值及指令内提供的相对偏移量来计算目标函数地址,所以在这里我们需要重新计算相对偏移量,以指向正确的目标函数地址.

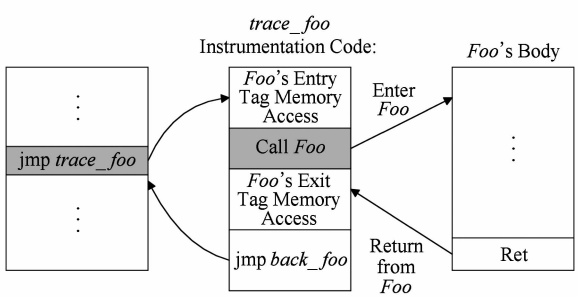



Fig. 5 Execution flow of the target function after instrumentation.

图 5 对目标函数进行插桩后的执行流程

上面插桩实现存在一个问题,在 x86_64 平台上,指令集是变长的.对于函数调用(call)指令其长度可能为 3 B, 5 B 或 9 B[13],对应不同寻址范围的函数调用.通过分析 SPECCPU2006 中的 28 个应用程序的可执行文件(ELF 格式)的代码段,我们发现平均 97.25%的函数调用使用 32 b 的相对偏移 call 指令.比如对于程序 444.namd,其代码段(.text)内一共包含 1 164 个函数调用指令,其中 1 147 个为 32 b 的相对偏移调用指令,占总的函数调用指令数的 98.54%.32 b 的相对偏移调用指令的操作码为"e8 cd",指令格式为"call rel32",指令长度为 5 B(即 40 b),这刚好就是一个 32 b 相对偏移无条件跳转(jmp)指令的长度.所以可以将这类的函数调用指令直接使用无条件跳转指令替换,而不会影响其他的指令.基于前面分析,可以预计使用这种替换方法能够捕获程序中绝大部分的函数调用.

### 1.3 tag 访存

由于现代的处理器上都含有 2 级或 3 级的高速缓存(cache),用于缓存最近访问过的数据.一旦将被访问的数据在 cache 内命中,该访存请求将不会被发送到内存系统,也就无法被 HMTT 卡采集到.由于 tag 访存直接对应到高层函数入口或出口事件,如果由于 cache 命中导致 tag 访存无法被 HMTT 采集到,那么本次函数高层事件就会丢失,这样对访存 traces 的函数级别分割将不完整甚至出错.

解决这个问题的方法就是保证每次 tag 访存都不通过处理器 cache,总能被 HMTT 监测到.通过修改系统的 grub.conf 文件配置保留一段物理内存地址空间(对操作系统不可见),使用 ioremap 调用将保留的这段空间映射成一个虚拟设备,并将其属性设置为不通过处理器 cache.然后通过 ptrace 接口,将这个虚拟设备(命名为 ioremap)映射到被监控进程的地址空间内.最后建立这段 ioremap 空间到不同函数入口及出口 tag 访存的映射,就是说访问这段空间内的某个位置将代表一个高层函数事件的发生.

使用 ioremap 实现的好处主要有两个:1)对这段空间的访问不通过处理器 cache,总能被 HMTT 采集到,不会"丢失".2)保留的物理内存空间对操作系统是不可见的,所以除了插桩的 tag 访存外,不会再受到其他的访问(如操作系统),避免受到干扰,保证采集到的 tag 访存的正确性和唯一性.

## 2 实 验

### 2.1 实验方法

本文所用的实验平台是 Intel 至强系列 E5504 服务器,包含两个处理器(socket),每个 socket 包含 4 个处理器核(core),工作频率为 2 GHz.每个 socket 上的高速缓存(cache)为 3 层结构,其中 L1,L2 为每个核私有, L1 的指令 cache 和数据 cache 都是 32 KB, L2 cache 大小为 256 KB,L3 cache 为同一个 socket 内的 4 个核共享,大小是 8 MB,每个缓存块(cache line)的大小均为 64 B.物理内存使用 DD3-800,总的容量为 4 GB,其中 2 GB 为操作系统可用空间,另外 2 GB 被保留作为 HMTT 配置空间及 ioremap 空间.IO 系统采用 10 个 1T B 的磁盘组成一个 raid 0 阵列.由于访存 traces 都是大文件,文件系统选用对大文件读写性能最优的 xfs 文件系统,使用 iozone 测试程序实测的 IO 写带宽可达700 MBps 以上.实验平台的操作系统使用 CentOS 5.3,内核为 Linux 2.6.32.

我们使用的测试程序是 SPEC CPU2006 中的访存密集型程序,表 1 列出了这些测试程序的基本

**Table 1 Individual Memory-Intensive Benchmark Characteristics**

**表 1 访存密集型程序的特征**

| Benchmark | Function Number | Average Size/B | Average Instruction Number |
|---|---|---|---|
| 401.bzip2 | 78 | 550.51 | 135.6 |
| 410.bwaves | 7 | 5 332 | 868.29 |
| 416.gamess | 2 840 | 2 621.47 | 476.18 |
| 429.mcf | 24 | 325.04 | 80.75 |
| 433.milc | 207 | 413.18 | 96.54 |
| 434.zeusmp | 63 | 4 463.49 | 765.24 |
| 436.cactus | 1 246 | 430.52 | 93.14 |
| 437.leslie3d | 21 | 6 201.71 | 1 045.1 |
| 450.soplex | 995 | 303.44 | 71.57 |
| 456.hmmer | 488 | 429.73 | 106.11 |
| 459.Gems | 98 | 3 636.34 | 645.14 |
| 462.libquan | 94 | 252.99 | 69.55 |
| 470.lbm | 19 | 524.26 | 102.89 |
| 471.omne | 2 377 | 168.64 | 41.68 |
| 473.astar | 97 | 299.9 | 76.05 |
| 482.sphinx3 | 327 | 366.05 | 92.38 |
| 483.xalan | 15 559 | 155.54 | 39.89 |
| Mean | 199 | 698.27 | 152.69 |

Note: We just analyze functions in the .text section of each application, without those dynamic functions.

特征.这 4 列分别代表程序名字、包含的函数个数、每个函数的平均字节数以及每个函数的平均指令条数.这里只统计程序代码段(.text)内的静态函数,不包括动态函数.编译器使用 gcc 4.1.2 版本,使用默认的-O2 优化选项.每个测试程序都使用 ref 输入集,保证实验结果的可靠性.

## 2.2 正确性实验

为了验证 HMTT_FBI 对进程注入插桩代码后,能够在每次函数调用之前及函数返回之后正确地发出 tag 访存,并被 HMTT 采集,进一步分析并得到包含函数执行事件的访存 traces,我们设计了如下实验:写一些简单的测试程序,这些程序包括:1)调用固定次数的函数;2)多个不同函数之间相互调用;3)递归函数.通过分析 HMTT 采集到的包含特殊 tag 访存的 traces,我们能够得到不同函数之间的相互调用关系图以及每个函数被调用的次数.实验结果表明,与直接对程序源代码的分析结果是完全一致的,这样就验证了 HMTT_FBI 工作的正确性.

进一步,我们分析更加复杂的 SPECCPU2006 内的几个程序来验证.由于这些程序包含的函数个数较多,函数调用关系复杂,我们只验证占执行时间百分比最多的前 10 个函数的被调用次数.通过 GNU 的 gprof 工具能够得到程序中每个函数的执行时间百分比和被调用次数.然后与通过 HMTT 访存 traces 分析得到的函数调用次数进行对比.

表 2 列出了程序 433.milc 的对比结果,其中第 2 列为 gprof 得到的前 10 个函数的被调用次数;第 3 列为 HMTT 采集到的对应函数调用的 tag 访存个数;最后 1 列为它们之间的偏差百分比.从表 2 可以看出 HMTT 能够准确获取程序 433.milc 的函数执行事件,误差都在 2%以下,其中偏差最大的是函数 mult_su3_mat_vec,HMTT 比 gprof 少了 1.90%.对其他几个程序的误差也都在 2%以下,说明本方法的正确性.

**Table 2 Number of Functions Comparison between Gprof and HMTT_FBI for 433.milc Program**

**表 2 HMTT 和 gprof 得到的程序 433.milc 的函数调用次数对比**

| Function Name | Gprof | HMTT | Difference/% |
|---|---|---|---|
| mult_su3_na | 4.61E+08 | 4.63E+08 | 0.52 |
| mult_su3_nn | 4.62E+08 | 4.68E+08 | 1.30 |
| mult_su3_mat_vec | 6.50E+08 | 6.38E+08 | −1.90 |
| mult_adj_su3_mat_vec | 6.32E+08 | 6.38E+08 | 0.84 |
| add_force_to_mom | 4.10E+03 | 4.16E+03 | 1.60 |
| scalar_mult_add_su3_matrix | 1.00E+09 | 9.96E+08 | −0.73 |
| mult_su3_an | 1.19E+08 | 1.18E+08 | −1.27 |
| su3mat_copy | 2.70E+08 | 2.72E+08 | 0.61 |
| su3_projector | 6.55E+08 | 6.61E+08 | 0.93 |
| u_shift_fermion | 8.02E+03 | 8.12E+03 | 1.35 |

## 2.3 开销分析实验

图 6 说明了 HMTT_FBI 各个功能模块的运行开销,我们使用 SPECCPU2006 中的访存密集型程序,这里列出的是各模块开销占每个程序原始运行时间的百分比.可以看出,HMTT_FBI 的运行开销是很低的,其中解析 ELF 的平均开销占原程序运行时间的 0.008%,反汇编分析占 0.028%,向进程映像内插桩代码占 0.015%,整个 HMTT_FBI 的运行开销只占 0.051%,所以对原程序几乎不引入开销.其中开销最大的是程序 416.gamess 达到 0.417%,这是因为程序 416.gamess 包含的函数多达 2 840 个,

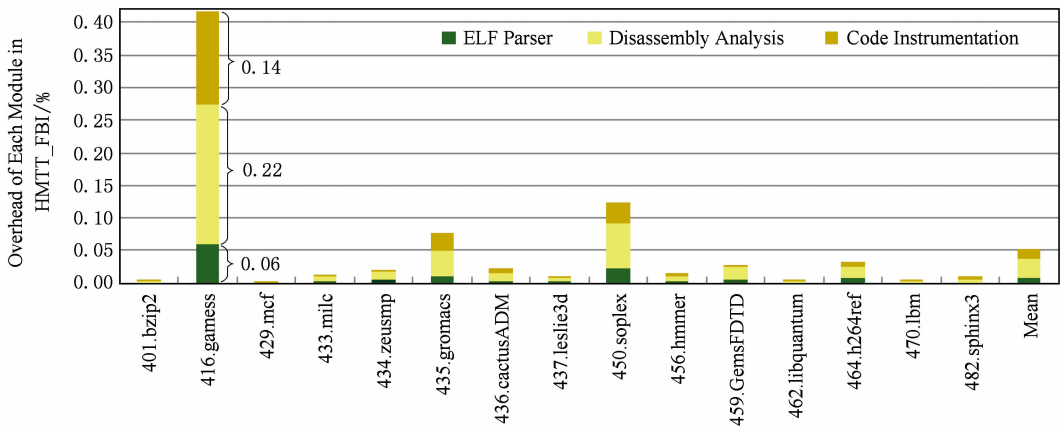



Fig. 6 Run time overhead of each module in HMTT_FBI.

图 6 HMTT_FBI 各模块运行开销

而且每个函数所含的平均字节数达到 2 621.47(如表 1 中所列),这样需要解析的工作量就比较大,开销所占百分比也就比较大.

下面定量评估向进程内插桩额外代码导致的开销,这里插桩代码的运行开销包括 2 部分:第 1 个开销是插入的跳转指令,因为我们这里使用两次无条件跳转(一次从函数调用点跳转到插桩代码区,另外一次从插桩代码区跳转回到原执行流),每次跳转都会刷处理器流水线,导致一定的性能下降;第 2 个开销是插入的额外不过 cache 的 2 次 tag 访存(目标函数的入口和出口各 1 次),因为每次访存都需要上百个处理器周期(cycles),所以 tag 访存的开销是比较大的,如果程序本身的访存比较少,额外的 tag 访存将导致明显的性能下降.

为了分别评估这 2 个开销带来的影响,我们设计了 2 种实验:第 1 种是我们在插桩代码区内插入空指令(nop),由于空指令基本没有开销,这样我们就能评估由于跳转引入的开销;第 2 种就是插入完整的 tag 访存指令,这样得到跳转加 tag 访存的开销.

图 7 说明了这两种开销,这里将运行时间归一化到原程序的运行时间. 可以看出插入跳转指令引起的开销比较小,平均为 1.042,也就是导致 4.2% 的开销,说明使用跳转指令实现控制流转移是合理的实现. 而跳转加上插入访存的平均开销为 1.62,也就是导致 62% 的开销,其中开销最大的是程序 450.soplex 达到 2.91. 由此可见主要的开销还是由不通过处理器 cache 的 tag 访存引起的. 一个可能的改进是通过准确记录每个函数的入口和出口的执行时间并与访存 traces 包含的时间信息进行同步,从而消除 tag 访存,降低开销.

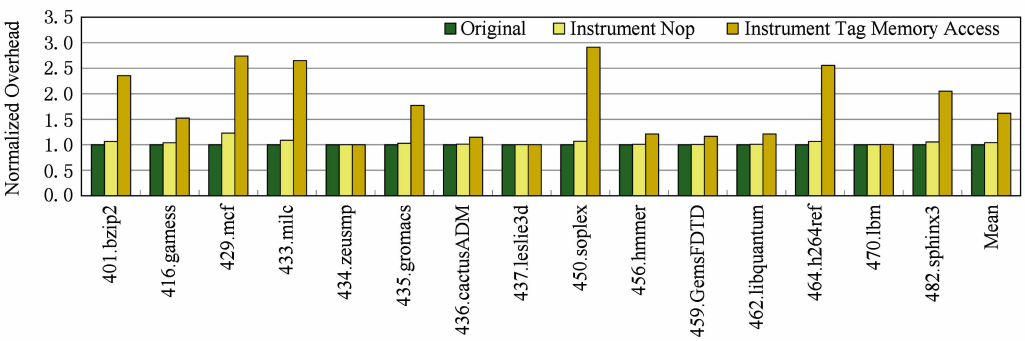



Fig. 7 Overhead of the instrumentation code.

图 7 HMTT_FBI 插桩的代码运行导致的开销

表 3 示出二进制插桩引入的干扰访存的开销,这里统计的干扰访存个数包括额外插入的 tag 访存以及为了保护进程状态而引起的访存(压栈、出栈). 其中第 2 列为原程序的访存个数;第 3 列为插桩后的访存个数;最后 1 列为归一化的访存干扰. 从表 3 可以看出,平均引入的干扰访存量为原程序访存量的 52%,其中访存干扰最低的是 436.cactusADM, 437.leslie3d 和 470.lbm 这样的访存密集型程序几乎为 0. 干扰访存最大的是程序 401.bzip2,达到 729%,这是因为程序 401.bzip2 为计算密集型程序,自身的访存量不多,所以对每个函数都进行插桩导致比较严重的访存干扰. 一个优化就是不对那些计算密集型的函数进行插桩,这是因为它们产生的访存量很少,对于内存优化意义不大. 这样一方面能够降低引入的干扰访存,另一方面还能降低运行开销.

**Table 3 Interference Memory Accesses Introduced by HMTT_FBI**

**表 3 HMTT_FBI 引入的干扰访存**

| Application | Original Memory | Instrumented Memory | Interference Rate |
|---|---|---|---|
| 401.bzip2 | 3.73E+08 | 3.09E+09 | 7.29 |
| 429.mcf | 2.21E+10 | 3.62E+10 | 0.64 |
| 433.milc | 3.57E+10 | 4.92E+10 | 0.38 |
| 436.cact | 1.67E+10 | 1.68E+10 | 0.00 |
| 437.lesli | 3.99E+10 | 3.99E+10 | 0.00 |
| 462.libq | 7.66E+10 | 7.84E+10 | 0.02 |
| 470.lbm | 6.60E+10 | 6.60E+10 | 0.00 |
| Mean | | | 0.52 |

同时对比图 7 和表 3 可以看出,引入干扰访存小的其造成的性能开销也较小. 对函数进行有选择性的插桩是很有必要的,如只对那些访存密集型的函数进行插桩.

### 2.4 与其他插桩工具开销对比实验

使用 Pin 纯软件方式也能获取进程执行过程中的访存虚拟地址序列 *traces*. Pin 的实现需要在指令

级进行插桩,分析每条指令是否需要访存,如果有访存,就将访存的地址记录下来,并保存到文件中.这里我们采用2种方案:第1种是先将访存地址*traces*写到内存缓冲区中,同时再开一个进程,专门负责从内存缓冲区中将访存地址*traces*写到文件中,我们采用多个缓冲区策略,保证写文件进程对Pin进程不会引入额外的IO开销,这样Pin进程只引入将*traces*写入内存的开销,称这种方法为Pin写到内存(Pin+Mem);第2种也是首先将访存地址*traces*写到一个内存缓冲区内,等到缓冲区满后,再写入到文件中,这样将引入IO开销,称为Pin写到文件(Pin+File).注意我们这里使用的IO系统为10个磁盘组成的raid 0阵列,写带宽可达700 MBps以上,如果IO系统配置比较差,Pin写到文件的性能将更差.

由于使用Pin获取访存*traces*的运行时间较长,我们只选取其中8个访存密集型测试程序作对比实验.图8列出了Pin获取访存*traces*相对于原程序运行时间的归一化开销.可以看出使用Pin的开销明显比软硬件结合实现的HMTT_FBI开销大,对于Pin写到内存,其平均开销为原程序的10.4倍;而Pin写到文件则为原程序的22.53倍.

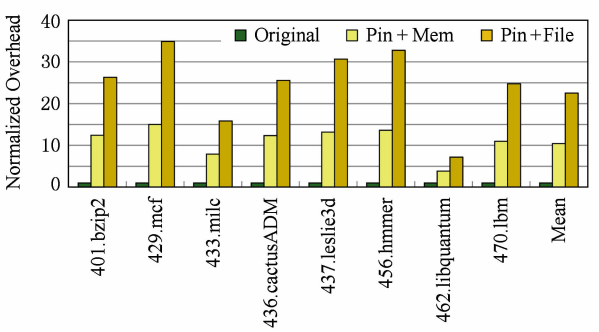



Fig. 8 Overhead of tracking memory traces with Pin.

图8 使用Pin获取访存*traces*的开销

## 3 相关工作

目前国内外获取或分析访存*traces*的方式主要有以下几种:模拟器、代码插桩(instrumentation)、性能计数器、硬件仿真、硬件侦听.模拟器是进行访存行为研究的重要工具,常用的模拟器有DramSim2[14],Simics[15],Gem5[16]等,但是模拟器的速度很慢,只能用于分析很小时间段内的访存序列,无法运行完整的应用程序.代码插桩通过解析可执行程序的指令流,能够得到完整的访存地址序列,但是代码插桩的开销比较大,而且引入大量的干扰访存.性能计数器只能得到关于访存行为的整体统计信息,比如cache miss, TLB miss等,不能获取应用的完整访存*traces*,无法对应用访存行为作更深入的分析.硬件仿真只能获取简化系统的完整访存*traces*,与真实系统有一定的偏差,而且硬件仿真实现成本高,不易推广.IBM, Intel等企业都研制出硬件侦听工具用于分析访存行为,如PHAMYME[17], MemorIES[18]等.这些工具一般都是插在特定总线上侦听访存命令,所以移植性较差,而且无法有效输出高达几十GB的访存*traces*.

二进制插桩技术在不影响进程正确执行的前提下,通过向应用程序注入额外的代码以实现收集程序运行过程中的信息.这些运行信息能被有效用于指导硬件及系统设计、程序调试及优化、安全验证及性能建模与预测[19].

Pin[4]是一个应用很广的动态插桩工具,其工作方式是利用即时编译(just in time compilization)器动态产生最终被运行的代码,在产生代码的同时注入插桩代码.Pin为用户提供了一套非常丰富的API,隐藏底层指令集细节,使用户能够很方便地获取进程执行过程中的上下文信息(如寄存器的值).用户只需根据API编写实现特定功能的插桩代码,Pin将自动在执行到插桩代码时保护进程寄存器状态,并在执行完毕后恢复寄存器状态,保证进程能够正确往下执行.正是由于Pin的易用性、灵活性及丰富功能,使得Pin成为学术界影响力最大的插桩工具之一.Pin也是基于ptrace API来获取对进程的控制,它不但能够在进程入口点就获取对进程的控制,还允许对一个正在运行中的进程进行插桩.但是使用JIT导致Pin的运行开销比较大,常常导致数十倍的性能下降,而且Pin的内部实现是不开源的,用户很难对插桩进行优化.

DynamoRIO[5]也是一个基于即时编译器的功能丰富的动态插桩工具.由于在产生代码时加入了很多优化,其开销甚至比Pin要低一些.但是DynamoRIO是基于Linux系统下的LD_PRELOAD环境变量将动态共享库加载到进程空间,因此无法对正在运行中的进程进行插桩.同样使用JIT尽管能够提供强大的插桩功能,其开销还是很大.

Dyninst[12]和PEBIL[19]都是通过将插桩点的指令替换成跳转指令来实现将控制转移到插桩代码位置,在插桩代码位置安排额外的指令实现功能,重新执行原来被替换指令,最后通过跳转指令将控制从

插桩代码区域转回到原来位置.其中 Dyninst 既支持在进程运行时对内存中的进程映象动态插桩,也支持向固态的可执行文件内静态插桩,生成一个新的可执行文件,但是 Dyninst 的插桩代码功能都是通过函数调用实现,对于那些实现简单功能的插桩开销比较大.而 PEBIL 则只能对可执行文件(为 ELF 格式)进行处理,生成一个新的包含插桩代码的可执行文件,而无法对正在运行中的进程进行插桩,缺乏灵活性.

上面介绍的几种插桩工具都有一个缺点,就是无法对函数的出口进行有效地插桩.使用 Pin 和 DynamoRIO 可以在函数级别进行插桩,但是它们提供的 API 只支持对函数入口的插桩.尽管能够以指令级别进行插桩,找到目标函数的调用指令,将插桩代码安排在调用指令之后,从而实现对函数出口的插桩.以指令级别进行插桩意味着即时编译(jit)需要对每条指令都需要进行解析判断,重新生成代码,开销将非常严重,由此引入的干扰访存也将大大增加.

## 4 总结与展望

准确地获取程序在真实系统上运行的访存 *traces* 是进行内存系统研究的基础.本文提出了一种基于二进制插桩的软硬件结合获取包含函数级别语义信息的访存 *traces* 的方法,通过直接修改内存中的进程映像,在函数的入口及出口插入 tag 访存指令,建立 tag 访存与上层函数执行流的对应关系,实现对访存 *traces* 的函数级语义分割,为后续的重点优化提供基础.实验结果显示了该方法的准确性、低开销和低访存干扰.

下一步工作包括:1)对函数进行选择性地插桩,只对那些访存密集型的函数进行插桩;2)对得到的访存 *traces* 进行更细粒度的以基本块为单位的划分;3)寻找一种开销更小的全局同步方法,插入不过 cache 的 tag 访存开销比较大;4)实现对多线程程序的支持,如 PARSEC 测试程序.

## 参 考 文 献


[1] Onur M, Thomas M. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems [C] //Proc of the 35th Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2008: 63-74

[2] Bao Yungang, Chen Mingyu, Ruan Yuan, et al. HMTT: A platform independent full-system memory trace monitoring system [C] //Proc of the 2008 ACM SIGMETRICS Int Conf on Measurement and Modeling of Computer Systems. New York: ACM, 2008: 229-240

[3] Ruan Yuan, Bao Yungang, Chen Mingyu, et al. The design and implementation of MIT—A hardware-based memory trace tool [J]. Acta Electronica Sinica, 2008, 36(8): 1519-1525 (in Chinese)
(阮元, 包云岗, 陈明宇, 等. 基于硬件的内存 trace 工具——MTT 的设计与实现[J]. 电子学报, 2008, 36(8): 1519-1525)

[4] Chi K L, Robert C, Robert M, et al. Pin: Building customized program analysis tools with dynamic instrumentation [C] //Proc of the 2005 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2005: 190-200

[5] Derek L B. Efficient, transparent, and comprehensive runtime code manipulation [D]. Cambridge, MA: Massachusetts Institute of Technology, 2004

[6] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2 [OL]. [2012-02-07]. http://refspecs.linuxbase.org/elf/elf.pdf

[7] Asim S. A system for process checkpointing and restarting (using a core dump) [OL]. [2012-02-07]. http://geocitiessites.com/asimshankar/checkpointing/report.pdf

[8] Silvio C, Matrix Z. Shared library call redirection using ELF PLT infection [OL]. [2012-02-07]. http://vxheavens.com/lib/vsc06.html

[9] Yang Hao, Tang Feng, Xie Haibin, et al. Library function disposing approach in binary translation [J]. Journal of Computer Research and Development, 2006, 43(12): 2174-2179 (in Chinese)
(杨浩, 唐锋, 谢海斌, 等. 二进制翻译中的库函数处理[J]. 计算机研究与发展, 2006, 43(12): 2174-2179)

[10] Susanta N, Wei L, Lap C L, et al. BIRD: Binary interpretation using runtime disassembly [C] //Proc of the Int Symp on Code Generation and Optimization. Los Alamitos, CA: IEEE Computer Society, 2006: 358-370

[11] Vivek T. Udis86 disassembler library for x86 and x86-64 [CP/OL]. [2012-02-07]. http://udis86.sourceforge.net/

[12] Bryan B, Jeffrey K H. An API for runtime code patching [J]. International Journal of High Performance Computing Applications, 2000, 14(4): 317-329

[13] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer's Manual [OL]. [2012-02-07]. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[14] Paul R, Elliott C B, Bruce J. DRAMSim2: A cycle accurate memory system simulator [J]. IEEE Computer Architecture Letters, 2011, 10(1): 16-19

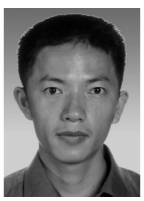
[15] Peter S M, Magnus C, Jesper E, et al. Simics: A full system simulation platform [J]. Computer, 2002, 35(2): 50-58

[16] Nathan B, Bradford B, Gabriel B, et al. The Gem5 simulator [J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1-7

[17] Chalainanont N, Nurvitadhi E, Morrison R, et al. Real-time L3 cache simulations using the programmable hardware-assisted cache emulator (PHA$E)[C] //Proc of 2003 IEEE International Workshop on Workload Characterization. Los Alamitos, CA: IEEE Computer Society, 2003: 86-95

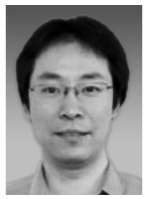
[18] Ashwini N, Kwok K M, Krishnan S, et al. MemorIES3: A programmable, real-time hardware emulation tool for multiprocessor server design [C] //Proc of the 9th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2000: 37-48

[19] Michael L, Mustafa M T, Laura C, et al. PEBIL: Efficient static binary instrumentation for Linux [C] //Proc of Int Symp for Performance Analysis of Systems and Software. Los Alamitos, CA: IEEE Computer Society, 2010: 175-183

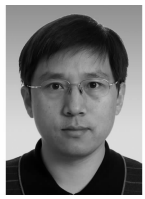


**Chen Licheng**, born in 1986. PhD candidate. Student member of China Computer Federation. His main research interests include computer architecture, memory architecture and optimization.

**Cui Zehan**, born in 1989. PhD candidate. His main research interests include computer architecture, memory architecture and power(cuizehan@ict.ac.cn).



**Bao Yungang**, born in 1980. Associate processor. Member of China Computer Federation. His main research interests include computer architecture, operating system and system performance modeling and evaluation(baoyg@ict.ac.cn).



**Chen Mingyu**, born in 1972. Professor and PhD supervisor. Member of China Computer Federation. His main research interests include computer architecture, operating system and algorithm optimization for high performance computers(cmy@ict.ac.cn).

**Shen Linfeng**, born in 1979. Master. IBM advisory software engineer. His main research interests include parallel file system, storage and big data analytics (shenlinf@cn.ibm.com).

**Liang Qi**, born in 1976. Bachelor. IBM advisory software engineer. His main research interests include performance tool and performance analysis (liangqi@cn.ibm.com).