第35卷 第10期 计 算 机 工 程 2009年5月
Vol.35 No.10 Computer Engineering May 2009
·软件技术与数据库· 文章编号：1000—3428(2009)10—0044—05 文献标识码：A 中图分类号：TP311

动态跟踪系统的性能模型

陶 捷，杨 珉
(复旦大学并行处理研究所，上海 201203)

摘 要：能用于生产环境中进行实时监控和实时调优的动态跟踪系统在跟踪过程中会给被跟踪的程序和系统引入未知的性能影响。为估算和量化这一影响值，通过对动态跟踪系统的软件架构和运行流程等方面的分析，提出计算该影响值的方法，并实测获取了计算过程中所需的各种参数。实验结果表明，该性能模型能够准确地对影响值进行预判。

关键词：动态代码插桩；动态跟踪；性能分析

Performance Model of Dynamic Tracing System

TAO Jie, YANG Min
(Parallel Processing Institute, Fudan University, Shanghai 201203)

【Abstract】 Dynamic Tracing(DTrace) can be used on live production systems to tune and monitor both user programs and operating system itself. When DTrace is enabled on live systems, it introduces some performance penalty. This paper presents the performance model of DTrace by analyzing its software architecture and its internal work flow, and obtains this model's parameters in real world. Experiments show this performance model can predicate the performance penalty correctly.

【Key words】 Dynamic Binary Instrumentation(DBI); Dynamic Tracing(DTrace); performance analysis

1 概述

动态代码插桩(Dynamic Binary Instrumentation, DBI)技术分为3类。第1类是基于即时编译(JIT)方式的动态插桩技术，典型的实现为Valgrind和Pin等。第2类是基于探测器方式的动态插桩技术，典型的实现有DynInst和Kprobes。第3类是结合虚拟化技术和即时编译方式的动态插桩技术，可以弥补第1类插桩方式下无法覆盖到内核态代码的缺陷，典型的实现有PinOS等。

动态跟踪(Dynamic Tracing, DTrace)系统[1]是第2类动态代码插桩技术的一种应用实现。通过动态跟踪可以对运行着的生产系统和运行着的程序进行跟踪、收集并分析其行为。在平台方面，动态跟踪系统在Solaris, FreeBSD和Mac OS平台上均已实现。在程序方面，动态跟踪系统支持C, C++, Java和多种脚本语言，可以在开发过程中帮助调试和进行性能评估。在维护方面，通过动态跟踪系统能够在不修改代码和不使用其他的分析工具的情况下，进行长期的性能数据采样并分析性能瓶颈。在系统安全方面，可以帮助进行各种安全行为的审计和分析，抑或通过对程序行为的跟踪来检测恶意软件和入侵事件。

针对跟踪程序可自定义化的特点和需要在生产系统上进行长期运行的可能性，本文通过对动态跟踪系统的分析，提出了一个完整的性能分析模型。通过该模型可以对影响值进行预测，也可以帮助对跟踪程序进行优化。

2 动态跟踪系统

2.1 动态跟踪系统的软件构架

动态跟踪系统[1]的设计目标是为了能在系统中提供一个既能用于用户态程序又能用于内核态程序的完整的跟踪工具，其整体结构如图1所示。

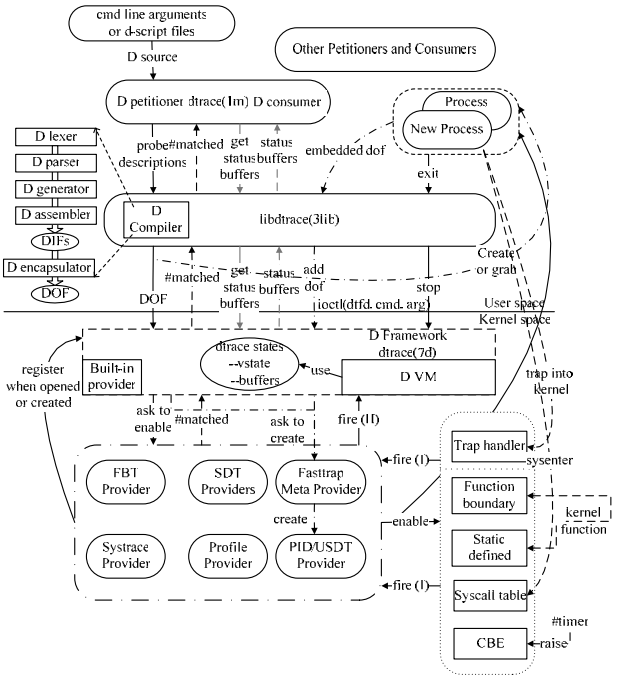

图1 DTrace体系结构

整个DTrace系统分为5个部分，自上而下分别是DTrace用户程序、DTrace库、DTrace内核模块、DTrace提供器和分散在内核代码中的各种插桩点及中断/异常处理函数。

位于内核中的DTrace模块(dtrace(7d))是整个结构的核心

作者简介：陶 捷(1982－)，男，硕士研究生，主研方向：系统软件，机群监控和管理，高性能计算；杨 珉，博士

收稿日期：2008-09-08 **E-mail：**052053007@fudan.edu.cn

部分。DTrace 内核模块独立构成了整个动态跟踪系统的框架(图 1 中由点划框、虚线框和点线框表示的部分)。独立的 DTrace 框架使得不同的插桩方式(称为提供器)可在该框架下独立实现。提供器(图 1 中虚线框中的各个部分)是一个个独立的内核模块，实现了一种同步或者异步的插桩方式。提供器作为可访问的伪设备，在被加载的时候向 DTrace 框架进行注册；框架在需要提供器协同工作之时，通过回调函数来使用这些提供器。用户态中 DTrace 分为 2 部分：(1)负责接收跟踪请求和显示收集到的数据的 DTrace 用户程序(dtrace(1m))。(2)DTrace 库(libdtrace(3lib))。

DTrace 库提供了统一的用于控制 DTrace 内核模块的接口，主要负责：

(1)对跟踪脚本进行编译并传递给 DTrace 内核模块(图 1 中 D Compiler 部分)。

(2)创建或抓取需要受控的用户进程，并请求 Fasttrap 元提供器动态生成与用户进程相关的提供器(图 1 中点划线箭头所表示的流程)。

(3)接收并执行 DTrace 用户关于读取 DTrace 内核状态和收集到的数据的请求。DTrace 用户程序和 DTrace 库加上用户自定义的跟踪脚本，便构成一个完整的 DTrace 跟踪程序(图 1 中浅色箭头所表示的流程)。

2.2 跟踪程序的运行流程

典型的跟踪程序执行过程如图 2 所示。跟踪程序首先打开各提供器(伪设备)和 DTrace 模块(伪设备)，如果这些设备是第一次被访问，则系统会要求先加载这些设备。然后各提供器收集系统内所有由其负责的插桩点。接着对跟踪脚本内的探测器说明进行检查、编译并封装成 DOF[1]。如需跟踪用户进程，则创建或者抓取那些需要被受控的进程映像。随后 DTrace 模块根据收到的 DOF 的内容，启用相应的探测器。在所有被请求了的探测器都启用之后，将先前创建和被抓取的进程恢复为运行状态，然后系统正式进入跟踪状态。此后跟踪程序进入一个循环过程(图 2 中的深色部分)。循环体中定期地通过 ioctl()取得 DTrace 内核中的状态信息、主缓冲区和聚合缓冲区(如果有)的内容，分析并显示收集到的信息(该过程称为刷新)。

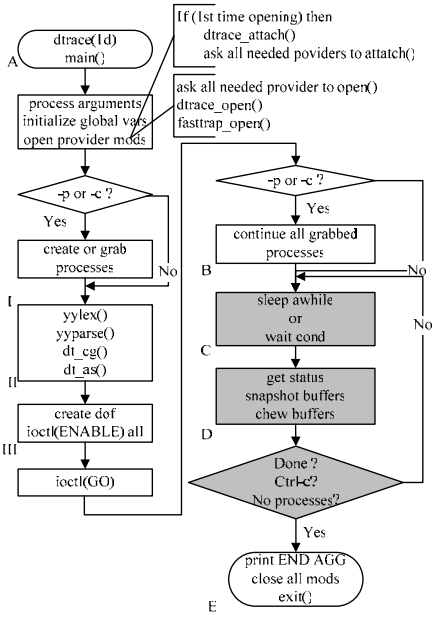

图 2　DTrace(1cmd)用户程序流程

2.3 提供器和探测器触发

DTrace 在统一的框架内通过不同的提供器实现了多种可用于内核和用户进程的插桩方式。常用的提供器为函数边界跟踪(FBT)提供器、静态定义跟踪(SDT)提供器、系统调用跟踪(Syscall)提供器、快速陷阱(Fasttrap)元提供器和异步时钟采样(Profile)提供器。通过 Fasttrap 元提供器可以动态地生成与指定进程相关的用户进程跟踪(PID)提供器和用户程序静态定义跟踪(USDT)提供器。

DTrace 框架为探测器的触发定义了一个统一的流程，如图 3 所示。

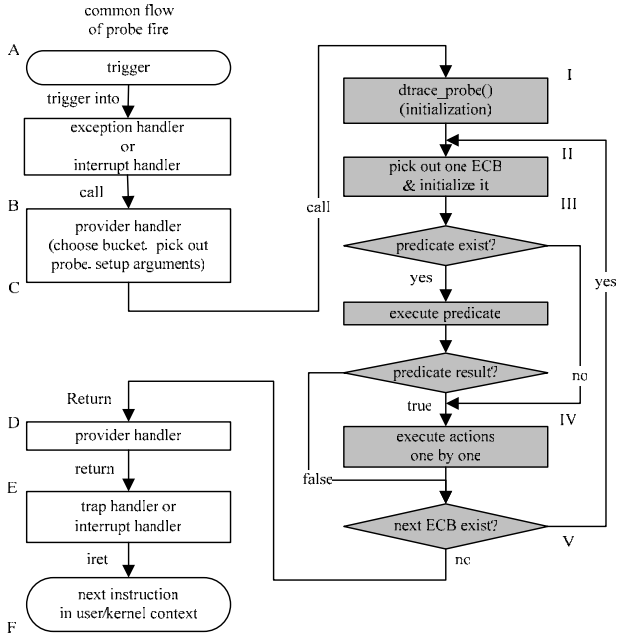

图 3　探测器触发流程

整个流程可以划分为 5 个阶段：

阶段 1(图 3 中 A—>B)：探测器在正常的上下文环境中触发后，首先转入相应的异常/中断处理函数中，进行现场保护的工作，然后回调提供器处理函数。

阶段 2(图 3 中 B—>C)：转入到提供器内，找出触发点所对应的探测器。获取探测器下挂载的 dtrace_probe，准备交由 D 虚拟机进行下一步的工作。

阶段 3(图 3 中 C—>D)：D 虚拟机中验证探测器的权限和触发状态，然后执行 dtrace_probe 内的各控制块中所定义的探测器操作。此阶段中需要关闭中断。

阶段 4(图 3 中 D—>E)：当 D 虚拟机执行完毕后，返回提供器处理函数进行余下的工作。

阶段 5(图 3 中 E—>F)：回到异常/中断处理函数中，模拟执行插桩点上被替换掉的原指令。完成所有的工作之后重建被中断的现场，并将程序控制转回到被中断的上下文中，整个跟踪过程完成。

系统调用提供器(Syscall)的插桩方式是修改系统调用表，其触发方式上不依赖于异常或者中断处理函数，所以触发流程中没有阶段 1 和阶段 5。

PID 提供器可以针对用户进程实行细粒度的代码插桩，所以需要能支持对任意指令的模拟。简单的指令(跳转、RET、PUSH EBP、CALL 和 NOP 指令)会在提供器内部的阶段 4 中进行模拟执行。对于其他指令，将该指令和一条跳转回正常执行序列的跳转指令(JMP rel32)一起复制到用户进程中的可

执行段中(记为 addr)。返回被中断的用户进程时，将先返回到 addr 处执行原有指令，然后跳转回正常的指令处继续执行。

2.4 D 虚拟机与探测器复用

探测器在内核和用户进程中是唯一存在的，为了支持系统中一个或多个不同的脚本对同一探测器的多个跟踪请求，DTrace 通过对探测器的抽象化实现了探测器的复用。DTrace 内核中以 dtrace_probe 描述每个探测器，每个对该探测器的使用请求均以一个独立启用控制块(ECB)[1]来表示。控制块由断言和操作列表 2 部分组成。同一个 dtrace_probe 下的各个控制块以单链表的方式互相链接。

D 虚拟机的执行过程如图 3 中深色部分所示。在阶段 B 中，提供器通过 dtrace_probe()函数启动 D 虚拟机。D 虚拟机中首先进行针对本次触发的初始化操作。然后依次调入 dtrace_probe 下挂载的控制块。对于每个控制块，先进行控制块相关的初始化工作。如果控制块中存在断言，则取出断言中的 DIFO 部分进行执行，根据执行结果判断该断言是否被满足。DIFO 是 D 编译器根据用户对探测器的描述而生成的一系列虚拟指令，类似于 Java 的字节码。其指令集类似精简指令集(RISC)，并包括一组有限的模拟寄存器。通过 dtrace–S 命令可以获取编译后 DIFO 的内容。断言满足时，D 虚拟机读取控制块内的操作列表，依次执行列表上的各个操作。操作分为 3 类：(1)DIFO，由 D 虚拟机解释执行；(2)DTrace 框架中已经预定义的动作，例如 stack()获取内核栈等；(3)数据的聚合操作。当控制块中的操作都执行完之后，D 虚拟机调入下一个控制块继续执行。

当虚拟机执行完探测器下挂载的所有控制块后打开中断，然后转回到提供器内开始阶段 C 的工作。

3 DTrace 性能影响模型

研究动态跟踪系统的性能模型在于分析：探测器数量和触发次数与时间消耗的关系；探测器复用对时耗的影响以及内核态和用户态中维持一个跟踪程序运行所需的基本时间消耗。

3.1 探测器带来的影响

系统内某一探测器 Probe，隶属于提供器类型 K。Probe 中挂载的控制块的数量为 M。其触发流程中各个阶段的时间消耗为 $Stage(K, x)$, $x\in\{A, B, C, D\}$。在虚拟机内执行时，虚拟机准备阶段所需的时间消耗为 $Cost_{DVM\_prepare}$，每个控制块的准备阶段所需的时间消耗为 $Cost_{ECB\_prepare}$，则整个触发流程(不包括虚拟机内执行断言和操作)的时间消耗为

$$flow = \sum_{x\in\{A,B,C,D\}} Stage(K,x) + Cost_{DVM\_prepare} + \sum_{i=1}^{M}(Cost_{ECB\_prepare}) \tag{1}$$

控制块 ECB_i, $i\in[1,M]$中断言部分所需的时间消耗为 $pred_i$，控制块的操作列表部分所需的时间消耗为 $action_i$。断言的 DIFO 中只包括虚拟指令的执行、对内建变量的使用和通过 CALL 指令调用子例程 3 个部分。对于断言 $pred_i$，DIFO 中除了 CALL 指令以外的其他指令(包括同一指令的重复执行)的数量为 I_i-#$call$，所有需要访问的内建变量(包括对同一变量的多次访问)数量为 V_i，通过 CALL 指令调用的子例程(包括对同一子例程的多次访问)数量为 S_i，则 $pred_i$ 执行一次的时间消耗为

$$pred_i = \sum_{j=1}^{I_i-\#call} instr_j + \sum_{v=1}^{V_i} bivar_v + \sum_{s=1}^{S_i} subr_s \tag{2}$$

根据 2.4 节所述，控制块内的动作可以由操作列表上的多个不同类型的操作组成。一个包含 O_i 个 DIFO、A_i 个聚合操作和 X_i 个动作的操作列表，其执行一次所需的时耗为

$$action_i = \sum_{o=1}^{O}\left(\sum_{j=1}^{I_o-\#call} instr_{o,j} + \sum_{v=1}^{V_o} bivar_{o,v} + \sum_{s=1}^{S_o} subr_{o,s}\right) + \sum_{a=1}^{A_i} aggr_a + \sum_{x=1}^{X_i} act_x \tag{3}$$

当系统中稳定地运行着一些进程时，可以认为在该段时期内系统中各探测器的触发频率是一定的，触发时各控制块断言为真的概率是一定的。对于断言 $pred_i$，某段时间内该断言为真的概率恒为 $Prob_i$，则探测器在该时间段内触发一次所需的时间消耗为

$$Cost(Probe) = flow + \sum_{i=1}^{M}\left(pred_i + Prob_i \times action_i\right) \tag{4}$$

如果控制块 M_i 中没有断言部分，则 $pred_i$=0, $Prob_i$=1。

3.2 DTrace 跟踪程序运行所带来的影响

跟踪程序 D 每秒的刷新频率为 R。每次进行刷新时，会造成系统中额外的 2 次进程切换，进程切换所需的时间消耗为 csw。

刷新时需要从 DTrace 内核框架中获取数据。DTrace 内核中首先对当前状态及各缓冲区进行统计工作，然后将统计结果和收集到的数据通过函数 *copyout*()分 2 次从内核缓冲区复制到用户态中指定的缓冲区内。如果存在聚合缓冲区且缓冲区策略为 SWITCH[1]，则每次刷新将导致缓冲区的更换，新的缓冲区在被使用前需要重建一份 Hash 表。每次刷新时所需的内核时间消耗总和记为 co，则 co 可以表述为

$$co = Cost_{status\_snapshot} + Cost_{buf\_snapshot} + Cost_{agg\_snapshot} + \frac{\sum_{b=\{buf,agg\}} datasize_b}{bandwidth} + Cost_{aggbuild} \tag{5}$$

对于从内核中获取到的数据，跟踪程序先对数据进行分析和统计，然后打印在屏幕或者输出到文件中，这部分时间消耗记为 *chew*&*print*。

从 3.1 节中对于探测器触发频率的说明可以推断出：在某段时间内，因探测器触发而产生的数据量也会是一定的。因此，在该段时期内，单位时间内维持跟踪进程 D 运行需要消耗的用户态和内核态时间之和为

$$Cost(D) = \sum_{j=1}^{R} chew\&\ print + \sum_{j=1}^{R} co + 2\times\sum_{j=1}^{R} csw \tag{6}$$

3.3 动态跟踪给系统带来的影响

系统中存在 S 个跟踪程序 D_i, $i\in[1,S]$，S 个跟踪程序一共启用了 N 个探测器。在某段时期内，对每个探测器 $Probe_i$, $i\in[1, N]$均有比较稳定的触发频率 $Freq_i$, $i\in[1, N]$。综合式(4)和式(6)，每秒内动态跟踪系统给系统带来的负载为

$$Cost_{DTrace} = \sum_{i=1}^{S}\left(\sum_{j=1}^{R_i} chew\&\ print_i\right) + \sum_{i=1}^{N}\left(Freq_i \times \left(flow_i + \left(\sum_{j=1}^{M_i}\left(pred_j + Prob_j \times action_j\right)\right)\right)\right) + \sum_{i=1}^{S}\left(\sum_{j=1}^{R_i} co_i\right) + 2\times\sum_{i=1}^{S}\left(\sum_{j=1}^{R_i} csw\right) \tag{7}$$

简化起见，定义 *chew*&*print* 部分的时耗之和为 $Cost_A$，探测器触发部分的时耗之和为 $Cost_B$，co 和 csw 部分的时耗之和为 $Cost_C$，所有跟踪程序执行时的用户态时耗为 $Cost_{usr}$，内核态时耗为 $Cost_{sys}$，则每秒内给系统带来的负载能表述为

$$Cost_{DTrace} = Cost_A + Cost_B + Cost_C = Cost_A + \left(Cost_B + Cost_C\right) = Cost_{usr} + Cost_{sys} \tag{8}$$

3.4 参数数据

式(1)～式(8)说明了如何计算单个跟踪程序和整个动态跟踪系统所带来的影响。对于各表达式中需要用到的参数，可以通过在源代码中各相关位置上加入额外的 rdtsc[2]指令获取高分辨率时间戳后统计相邻时间点间的时耗来获得。通过 rdtsc 指令收集到的时间数据以时钟周期数为单位，除以 CPU 主频可以换算为秒，再根据每秒内占用多少时间可以换算成每秒内占用 CPU 的百分比。测试时 rdtsc 指令和 mov 指令(用于将获得的数据保存至内存中)会引入额外的时耗，结果中需要扣除这部分的时耗。

测试机为 Dell 340 工作站。其基本配置为 Intel Pentium4 1.7 GHz CPU，512 MB 内存，WD 80BB 硬盘，3Com 3C905C 百兆网卡，操作系统是 OpenSolaris Build74。测试使用的脚本选自 DTraceToolKit[3]。最终收集并整理得到的各参数数据如表 1～表 6 所示。在表 1 中，所需周期数的数值形如 $m\pm n$ 的形式，表示某类型指令的平均时耗为 m，各指令间最大偏差为 n。在表 3 中，带宽为 684.53 MB/s。表 6 的单位是时钟周期数，*表示数据已经计入在阶段 1 中，$Cost_{DVM\_prepare}$=432，$Cost_{ECB\_prepare}$=298。

表 1　式(2)、式(3)所需的参数 1

指令类型	所需周期数	指令名
逻辑	44±6	OR, XOR, AND, NOT
加减	41±2	ADD, SUB
乘法	62±0	MUL
除法，模	127±4	SDIV, UDIV, SREM, UREM
加载初始化表项	31±1	SETX, SETS
压栈	49±0	PUSHTV
压栈(ref)	318±0	PUSHTR
清空栈	31±0	FLUSHTS
子例程调用	由子例程决定	CALL
两数比较	60±0	CMP
与 0 比较	38±0	TST
字符串比较	与字符串长度有关	SCMP
条件跳转	37±13	BA, BE, BG, BL 等 10 条
返回	28±0	RET
M->R 数据传送	90±13	LDSB/SH/SW 等 7 条
R->M 数据传送	985±0	STB
R->R 数据传送	29±0	MOV
载入内部变量	由内建变量决定	LDGS
载入全局变量	252±0	LDGS
保存全局变量	33±0|15 000	STGS|STGS(by ref)
载入线程变量	982±0	LDTS
保存线程变量	1 562±0|16 000	STTS|STTS(by ref)
载入局部变量	40±0	LDLS
保存局部变量	57±0|15 000	STLS|STLS(by ref)
载入全局关联数组	999±0	LDGAA
保存全局关联数组	1 566±0	STGAA
载入线程关联数组	2 227±0	LDTAA
保存线程关联数组	2 692±0	STTAA

表 2　式(2)、式(3)所需的参数 2

变量名	所需周期数
curthread	69±0
timestamp	239±0
vtimestamp	77±0
walltimestamp	1 987±0
*arg**	46±3
*probe**	81±6
**id*	165±32
uregs[*x*]	232±64

表 3　式(5)所需的参数

ioctl 类型	数据量	所需周期数
$Cost_{status\_snapshot}$	88 B	7 941
$Cost_{buf\_snapshot}$	40 B＋0 B	7 037
$Cost_{agg\_snapshot}$	40 B＋0 B	5 973
$Cost_{aggbuild}$	4 MB	786 208

表 4　式(2)、式(3)所需的参数 3

类型	名称	所需周期数
子例程	copyin/copyinstr	1 362±130
子例程	strlen	406±0
子例程	alloca	972±0
子例程	bcopy	1 140±0
子例程	strjoin	774±0
操作	printf/printa	8±2
操作	stack	1 454±0
操作	ustack	2 352±0
聚合	count/max/min/avg/sum	1 402±353
聚合	quantize/lquantize	2 418±126

表 5　式(6)所需的参数

系统状态	进程/线程数	空闲 CPU/(%)	每秒切换次数	1 min 内平均负载	*csw*/μs
空闲	58/185	99	75~85	≈0	8

表 6　式(1)所需的参数

提供器类型	阶段 1	阶段 2	阶段 4	阶段 5	合计
FBT(entry)	2 313	97	17	651	3 078
SDT	1 789	61	0	544	2 394
Syscall(entry)	…	64	213	…	277
Fasttrap(pid)	1 207	968	354	*	2 529
Fasttrap(usdt)	1 059	968	354	*	2 381
Profile	482+x	248	0	356+x'	1 086+x+x'

参数可以分为 3 类：(1)固定型：如表 1 中绝大多数指令的时耗、表 2 中内建变量的时耗和表 6 中各阶段的时耗。由于它们的功能单一、代码逻辑单一，因此其时耗较为固定。(2)配置相关型：如表 3 中各 *Cost* 数据是在默认 4 MB 缓冲区大小的情况下测得的。改变配置后，所需处理的数据量会有所改变，因此，所需时耗也会根据配置相应地发生变化；但在同一配置的情况下，其时耗较为固定。(3)实际相关型：如表 1 中的字符串比较指令和表 4 中各子例程和各操作的时耗，其时耗长度与它们所需处理的实际数据量有关。所以表中所列的数据是运行了 88 个选自 DTraceToolKit 的脚本后所测得的平均时耗。

4　模型验证

为了对式(1)～式(8)进行验证，设计如下的测试方法：测试程序在 30 s 内重复执行系统调用 *read*(或函数调用 *func*)，并使调用频率维持在 50 000 Hz 左右。在测试程序的开始和结束处通过 kstat[4]获取关于 CPU 使用的信息，比较 2 次 kstat 的结果，然后程序退出。

事件频率定在高频(50 000 Hz)是为了能够积累足够多的因探测器触发而消耗的时间，这样可以使其他因素产生的干扰变得能被忽略。系统调用 *read* 每次只读取 1 个 Byte，按照 Solaris 的 seg_map 文件缓冲机制[5]，每 8 192 次读操作之后才会引起一次实际的 I/O 请求，这样可以避免过多 I/O 请求而造成的不确定性。

对于 *read*，有 Syscall 和 FBT 类型的各 2 个脚本。sys/fbt 脚本中探测器控制块长度为 1，没有断言。sys10/fbt10 脚本

中探测器控制块长度为 10，其中前 9 个控制块的断言恒不成立。对于 *func*，pid 脚本中控制块长度为 1，没有断言。Profile 是异步触发的探测器，通过探测器名直接申明为 50 000 Hz 的触发频率(即 profile-50 000 Hz)，因此，无需在测试中引入系统调用或者函数调用。

在表 7 中第 2 列数据是根据式(1)～式(4)计算得出的每个脚本中探测器触发一次所需的时耗，单位为时钟周期数。所有测试脚本中均使用了聚合，根据式(5)～式(8)，由 $Cost_C$ 而引入的内核态时间约为

$$Cost_C \approx 2\times 9\ \mu s+1\times \big((7\ 941+7\ 037+7\ 973)\ \text{clocks}+1\ \text{KB}\div 684.53\ \text{MB/s}+786\ 208\ \text{clocks}\big)\approx 18\ \mu s+13.5\ \mu s+1.43\ \mu s+462.48\ \mu s=495.41\ \mu s\approx 0.049\ 5\%$$

表 7　各不同事件下不同跟踪脚本的实测数据与理论数据

测试量	时耗	频率/Hz	sys/(%)	理论 sys/(%)	实测 sys/(%)	usr/(%)	理论 usr/(%)	实测 usr/(%)
read	-	49 562	17.223 8	-	-	-	-	-
sys	2 281	46 631	23.030 2	6.306 3	6.825 0	-	-	-
fbt	5 082	43 980	27.093 3	13.196 9	11.809 4	-	-	-
*sys*10	8 864	39 389	35.101 3	20.587 4	21.412 8	-	-	-
*fbt*10	11 665	36 940	38.965 4	25.396 9	26.128 0	-	-	-
func	-	50 665	0.263 0	-	-	99.737 0	-	-
pid	4 533	44 747	0.316 7	0.049 5	0.053 7	99.683 3	11.923 7	11.655 3
空闲	-	0	0.264 7	-	-	-	-	-
profile	3 090	50 000	9.458 8	9.088 2	9.194 1	-	-	-

根据探测器时耗、事件触发频率和 $Cost_C$ 部分的时耗，可以计算得出理论上由于动态跟踪而给测试机造成的内核时耗(表 7 中的理论 sys/(%)部分)。pid 探测器触发时由于仍旧处于用户态上下文中，因此探测器触发的时间消耗须记在用户态时间内，计算所得的探测器触发时耗记为理论 usr/(%)部分。

实测中未启用动态跟踪时，事件频率为 f，每次事件的内核态或用户态时耗为 T。启用动态跟踪之后，事件频率降为 f'，探测器时耗为 C，由此可以得出：

$$read\begin{cases} sys\% = f\times T \\ sys'\% = f'\times(T+C) = f'\times T + f'\times C \end{cases}$$

$$func\begin{cases} usr\% = f\times T \\ usr'\% = f'\times(T+C) = f'\times T + f'\times C \end{cases}$$

根据上述方程可以计算得出 $f'\times C$ 所占用的 CPU 时间，并记入相应的实测 sys/(%)和实测 usr/(%)栏中。

对比表 7 中的理论数据和实测数据可以发现，两者结果接近。CPU 误差在±2%之内，相应的探测器时耗误差在±800 个时钟周期之内，说明表达式所构建的性能模型符合实际情况。造成误差的主要原因是：(1)参数的测定受测试手段和测试精度限制而存在着一定的误差；(2)实测中事件频率本身存在 1%左右的不稳定性。

表 7 中的理论数据是根据实测条件下相应的事件频率计算所得，且因为实验设计的缘故，探测器触发时各控制块的断言概率恒定，所以理论数据与实测数据相当吻合。在实际运用时，估算过程中的事件频率使用的是未启用跟踪之前的数据，且探测器触发频率和断言概率属于估测数据，因此，进行评估时误差会有所增大。

5　结束语

动态跟踪系统带来了一种强大而便捷的调试和分析方法。动态跟踪程序可以帮助在生产系统上进行长期的监测和分析，甚至自主地进行反馈控制。作为可以长期运行的工具，动态跟踪的性能如何是一个关键的问题。其性能影响主要来自 2 方面：(1)跟踪系统自身实现的因素；(2)跟踪脚本编写的因素。本文在分析了动态跟踪系统的实现和运行流程后，给出了动态跟踪系统和动态跟踪程序的性能模型，并通过实测获取了模型中与系统相关的各种参数。与实例的比对证明，表达式能很好地表述动态跟踪工具给系统引入的额外负载。在分析和测试的过程中，也发现了 DTrace 框架、提供器和 D 编译器中存在的不足之处。改进这些问题，对于动态跟踪工具在生产系统上的长期化运行有着很大的帮助和性能提高。

参考文献

[1] Cantrill B, Shapiro M, Leventhal A. Dynamic Instrumentation of Production Systems[C]//Proc. of USENIX Annual Technical Conference. [S. l.]: USENIX Association, 2004: 2-17.

[2] Intel Corporation. Using the RDTSC Instruction for Performance Monitoring[EB/OL]. (1997-04-19). http://pasta.east.isi.edu/algorithms/IntegerMath/Timers/rdtscpm1.pdf.

[3] DTrace Community. DTraceToolKit[EB/OL]. (2007-10-23). http://www.opensolaris.org/os/community/dtrace/dtracetoolkit.

[4] Boothby P. Solaris Kernel Statistics[EB/OL]. (2001-07-03). http://developers.sun.com/solaris/articles/kstatc.html.

[5] McDougall R, Mauro J. Solaris Internals[M]. 2nd ed. 北京: 机械工业出版社, 2007: 707-722.

编辑　顾逸斐

(上接第 43 页)

在实际应用中，解集中包含点的个数都不是很多，因此不影响其应用。

6　结束语

本文提出一种新的动态优先搜索树，只采用叶节点来存储数据，用户接口简单友好。由于减小了旋转操作的时间，因此加快了节点插入与删除的速度，可以在 $O(\log n)$时间完成插入、删除操作，在 $O(\log n+k)$时间内实现搜索查询。其中，n 为数据点的个数；k 为满足搜索条件的解的个数。该 DPST 特别适合于数据量庞大的应用，如超大规模集成电路的物理设计等。

参考文献

[1] Berg M, Van Kreveld M, Overmars M, et al. Computational Geometry Algorithms and Applications[M]. 2nd ed. Berlin, Germany: Springer, 2000.

[2] McCreight E M. Priority Search Trees[J]. SIAM Journal of Computing, 1985, 14(2): 257-276.

[3] Berman P, Kahng A B, Vidhani D, et al. Optimal Phase Conflict Removal for Layout of Dark Field Alternating Phase Shifting Masks[J]. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, 2000, 19(2): 175-187.

[4] Tamassia R. Priority Search Tree[EB/OL]. (2008-03-01). http://www.cs.brown.edu/courses/cs252/misc/proj/src/Spr96-97/mjr/doc/09.ps.

[5] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms[M]. 2nd ed. Cambridge, UK: MIT Press, 2002.

编辑　顾逸斐