基于蒙特卡罗方法的内核代码热点动态检测技术研究*)

杨 宁 卢显良

(电子科技大学计算机科学与工程学院 成都 610054)

摘 要 计算机操作系统是计算机系统中最为重要的系统软件,其性能的高低直接影响整个计算机系统的性能。因此,发现和优化操作系统内核中最经常运行的代码是一件重要的工作。本文分析了 Linux 2.4 内核中采用的基于蒙特卡罗方法的代码热点检测算法和实现机制,指出了其中的不足,并在此基础上提出了改进措施。

关键词 代码热点,蒙特卡罗方法,动态检测

Research of Operating System Kernel Code Hotspot's Dynamic Detection

YANG Ning LU Xian-Liang

(College of Computer Science & Engineering, UESTC of China, Chengdu 610054)

Abstract As the most important system software, the performance of operating system kernel is a key factor to the whole computer system's efficienty. So it is necessary to detect and optimize the codes that run mos frequently, which is called hotspot. In this paper, we analyse a mechanism based on Monte Carlo method, which is used by Linux 2.4 kernel to detect hotspot of the kernel dynamically. We also point out and amend a flaw of it.

Keywords Code hotspot, Monte carlo method, Dynamic detection

1 前言

操作系统内核是计算机系统中最为重要的系统软件,其各个部分的运行性能直接影响着整个系统的性能。因此,对操作系统内核的性能不断地进行优化,是一件非常必要的工作。

但是,现代操作系统内核代码非常庞大,所以优化要有针对性地进行。根据程序的时间局部性原理可知,优化的基本原则是对经常运行的代码或者函数进行优化。我们称经常被运行的那一部分代码为代码热点。如果能有效地找出内核代码实际运行时的代码热点,则优化的针对性就更强,优化后性能的提升就更高。

代码热点可以通过阅读代码进行分析和估计,也可以在操作系统实际运行时动态地检测和统计。我们称前者为静态分析的方法,称后者为动态检测的方法。显然,动态检测的方法更能得出真实可靠的结论。

Linux 2.4 内核实现了代码热点的动态检测。本文介绍了该技术所依据的蒙特卡罗方法的基本思想,然后对 Linux 操作系统内核运用该方法实现代码热点检测的算法及其实现进行了分析,并在此基础上提出了改进措施。

2 蒙特卡罗方法的基本思想

蒙特卡罗方法又叫随机抽样或者统计试验方法,是属于随机模拟的一种方法。Linux 2.4 内核使用了该方法实现了代码热点的检测和统计。本节介绍蒙特卡罗方法的基本思想。

当所求解的问题是某种事件出现的概率,或者是某个随机变量的期望值时,它们可以通过某种"试验"的方法,得到这种事件出现的频率或者这个随机变量的期望值,并用它们作为问题的解。这就是蒙特卡罗方法的基本思想。蒙特卡罗方法通过抓住事物运动的几何数量和几何特征,利用数学方法来加以模拟,即进行一种数字模拟实验。它是以一个概率模型为基础,按照这个模型所描绘的过程,通过模拟实验的结果,作为问题的近似解。可以把蒙特卡罗解题归结为 3 个主要步骤:构造或描述概率过程;实现从已知概率分布抽样;建立各种估计量。这 3 个步骤具体描述如下。

(1)构造或描述概率过程。

对于本身就具有随机性质的问题,主要是正确描述和模拟这个概率过程。对于本来不是随机性质的确定性问题,就必须事先构造一个人为的概率过程,它的某些参量正好是所要求问题的解,即要将不具有随机性质的问题转化为随机性质的问题。

(2)实现从已知概率分布抽样。

构造了概率模型以后,由于各种概率模型都可以看作是由各种各样的概率分布构成的,因此产生已知概率分布的随机变量(或随机向量),就成为实现蒙特卡罗方法模拟实验的基本手段。这也是蒙特卡罗方法被称为随机抽样的原因。

(3)建立各种统计量。

一般说来,构造了概率模型并能从中抽样后,即实现模拟实验后,我们就要确定一个随机变量的统计量,作为所要求的问题的解,我们称它为无偏估计。建立各种统计量,相当于对模拟实验的结果进行考察和登记,从中得到问题的解。

总的说来,蒙特卡罗方法就是用试验的方法确定事件的概率和随机变量的数学期望,以此作为问题的解。

3 Linux 内核代码热点检测机制

3.1 采样的时机

Linux 内核在每次对时钟中断进行处理时,判断中断前

*)本文受信息产业部生产发展基金项目网络安全集成防护系统支持,项目代号信运部[2002]546。杨 宁 硕士研究生,研究方向:操作系统;卢显良 教授、博士研究生导师,研究方向:操作系统。

系统是否运行于内核状态。如果是，则对中断前 EIP 寄存器的值进行采样，这个值实际上就是每次时钟中断处理结束后将要执行的第一条指令。基于 x86 平台的时钟中断频率为 100Hz，即每 10ms 采样一次。

3.2 代码执行的随机性

在一次时钟中断前，如果系统运行于内核状态，则中断前保存的 EIP 值是不确定的，即中断恢复后执行的第一条指令具有随机性。换句话说，代码热点问题本身就是一个具有随机性的问题，适合用蒙特卡罗方法求解。

3.3 算法及其实现

首先明确，我们要检测的是某个代码段（函数）是否为代码热点，而不是针对某条指令。为此，首先将整个内核代码段分为 $m=2^N$ 块（称 N 为伸缩因子）。设离散随机变量 H，其可能取值为 $i(i=0,1,\cdots,m-1)$。事件 $\{H=i\}$ 的意义为：在一次采样中，EIP 值落在了序号为的代码块内。存在 $k(0\leqslant k\leqslant m-1)$ 使得 $P\{H=k\}=\max(P\{H=i\})$，此时序号为 $k(0\leqslant k\leqslant m-1)$ 的代码块为代码热点。由此可见，这里采用的是离散随机变量概率分布模型，统计量就是随机变量本身。

问题的关键是求出 $\max(P\{H=i\})$，为此就要通过采样求出随机变量 H 的概率分布。在算法中，我们使用事件 $\{H=i\}$ 出现的频率数来模拟 $P\{H=1\}$。

以上为算法的机理，现在我们对 Linux 中具体实现进行描述。

(1)初始化过程

首先需要求得整个内核代码段的长度。我们知道，在整个内核映象中，内核代码段从_stext 开始，到_etext 结束。在内核初始化时，直接求出内核代码段的长度 TLen=_etext-_stext，代码段的块数为 $m=2^N$，则每块代码的长度为 $Len=\frac{TLen}{m}=\frac{TLen}{2^N}$，可见每块代码的长度由伸缩因子 N 决定。伸缩因子 N 可在系统启动时由启动参数 profile 指定。

为每一块代码建立一个整型计数器，并组织成一个数组 $counter[m]$，其中 $counter[i](0\leqslant i\leqslant m-1)$ 对应序号为 i 的代码块的计数器。图 1 为其示意图。

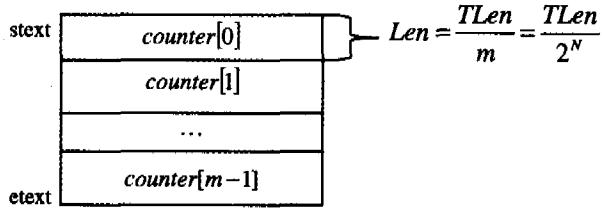

图 1 代码分块示意图

此外，内核在初始化时，在/proc 文件系统中建立一个 profile 节点，允许用户实用程序读取计数数据。

(2)采样计数过程

如前面所述，采样任务在时钟中断处理过程内完成。第一步是判断在中断前系统的运行状态。这是通过检查进入中断处理程序前被压入堆栈的寄存器 CS 的值中的特权位来实现的。第二步，如果是运行于内核状态，则从堆栈中取出 EIP 寄存器的值，这是中断恢复后将要执行的第一条指令。对 EIP 做如下运算，以计算该指令属于哪个代码块。

求得相对于内核代码段起点的位置：$RPos=EIP-\_stext$；

计算出所属的代码块的序号：$i=\frac{RPos}{m}(m=2^N)$；

相应的计数器加 1：$counter[i]=counter[i]+1$。

(3)统计过程

统计工作由用户实用程序实现。用户实用程序通过读取/proc/profile 文件得到采样数据。原则上说，直接找出计数器中的最大计数值，即可确定哪一块内核代码是代码热点。但是，这样的信息表达对用户没有意义，用户关心的是哪一个函数或者过程是代码热点，因此必须将求得的代码块序号转换为具体的函数名。

我们知道，Linux 内核编译后，生成了一个名为 System.map 的符号文件，其中包含了内核中所有函数名（或者过程名）和对应的起始地址，也包括_stext 和_etext 的值。因此，这个文件可以看作是一个函数，实现了函数地址到函数名的映射。设这个函数为：fname=AddrToName(faddr)，其中 faddr 为函数地址。使用下列 C 伪代码描述的算法，可以输出每个函数的执行频率：

```
1  fn_addr = _stext;
2  while(fn_addr ! = _etext) {
3        ticks=0;
4        index=0;
5        next_fn_addr = get_next_function_addr();
6        while(index < (next_fn_addr -fn_addr)/Len)
7              ticks = ticks + counter[index++];
8        printf ("% s % i \ n", AddrToName (fn_addr),
         ticks);
9        fn_addr = next_fn_addr;
10 }
```

注意，一个函数和下一个函数之间的代码宽度 next_fn_addr - fn_addr 可能跨越了几个代码块（跨越的块数为(next_fn_addr - fn_addr)/Len），所以在算法的 6、7 行把包含在这个宽度内的各个代码块的计数累加起来作为该函数的执行频率。

从以上的讨论可以看出，伸缩因子 N 越大，代码被划分成的块数就越大，则每块代码的长度就越小，进行热点检测的精度就越高。以下是在伸缩因子 N=4 时测试结果：

readprofile:step=4

4	_stext	0.0625
1456	default_idle	22.7500
51	system_call	0.8500
131	so_page_fault	0.1026
60	schedule	0.0872
1205	vm_set_pte	10.7589
228	vm_pte_clear	3.5625
118	zap_page_range	0.3512
655	_rdtsc_delay	20.4688

step=4 表明伸缩因子为 4；最左列的数字表示该函数所在代码执行的次数，最右列的数字表示该函数所在代码的执行次数占该函数所在代码块的总执行次数的比率。从该测试结果可以看见，中断处理函数没有被统计到。

4 问题与改进措施

我们知道，Linux 操作系统对中断处理是严格串行化的。因此，上述机制由于采样时机是在时钟中断内，所以实际上无法采样到各个中断处理代码的数据。然而，我们在做统计计算时，由于 System.map 文件中包含了中断处理函数的地址

和函数名的记录，所以我们有可能会错误地计算出中断处理函数的执行频率。事实上，由于没有中断处理代码的执行计数，所有中断处理函数的执行频率都应该为0。

为了能够取得各个中断处理函数的计数，我们修改采样时机，将采样点改在中断处理程序的总入口函数中，不仅对堆栈中的EIP值进行采样，也对即将执行的中断处理程序的首地址进行采样。中断处理程序的总入口是do_IRQ函数，do_IRQ函数调用handle_IQ_event函数遍历执行共享同一IRQ信号的中断处理程序。故我们修改handle_IRQ_event函数如下：

```
1  int handle_IRQ_event(unsigned int irq, struct pt_regs * regs,
struct irqaction * action)
2  {
3          int status;
4          int cpu = smp_processor_id();
5
6          irq_enter(cpu, irq);
7
8          status = 1;
9
10         if (! (action->flags & SA_INTERRUPT))
11         __sti();
12
13         do {
14             status |= action->flags;
14-1           do_hot_spot((unsigned int) action->handler, regs->
               eip);
15             action->handler(irq, action->dev_id, regs);
16             action = action->next;
17         } while (action);
18         if (status & SA_SAMPLE_RANDOM)
19         add_interrupt_randomness(irq);
20         __cli();
21
22         irq_exit(cpu, irq);
23
24         return status;
25 }
```

14-1行是我们加入的一个采样点。do_hot_spot(ip, eip)函数是我们新写的一个采样函数，其流程如下：

求得中断处理程序相对于内核代码段起点的位置：$RPos = ip - stext$；

计算出所属的代码块的序号：$i = \frac{RPos}{m}(m = 2^N)$；

相应的计数器加1：$cornter[i] = counter[i] + 1$；

此外对EIP进行常规采样：

求得相对于内核代码段起点的位置：$RPos = EIP - stext$；

计算出所属的代码块的序号：$i = \frac{RPos}{m}(m = 2^N)$；

相应的计数器加1：$counter[i] = counter[i] + 1$。

以下是改进后在伸缩因子$N=4$时的测试结果：

readprofile: step=4

4	_stext	00.0625
14	handle_IRQ_event	0.1250
11568	set_rtc_mmss	30.1250
183	bust_spinlocks	1.9062
224	schedule	0.3256
1140	vm_ptep_set_wrprotect	35.6250
2117	unexpected_intr	12.0284

从该测试结果可以看见，已经能够统计中断处理函数了。

小结 本文首先介绍了操作系统内核代码热点的概念，然后具体分析了基于蒙特卡罗方法的代码热点动态检测和统计算法。该算法已经在Linux操作系统内核中实现。本文对其实现也做了分析，并指出了其中的不足，然后在此基础上进行了改进，从而修正了原有实现的不足。

参 考 文 献

1 Daniel P R, Marco C. Understanding The Linux Kernel. O'Reilly Media, 2003

2 徐全智，杨晋浩. 数学建模. 北京：高等教育出版社，2003

（上接第237页）

要注意的是那些可能会被多次调用而其中却没有涉及多少工作量的函数。

结束语 面向高性能科学计算的构件技术已成为了并行计算领域的研究热点。由美国能源部、犹他州大学、印第安那大学等联合提出的CCA正是为高性能科学计算设计的一套构件环境规范。本文对该规范及其主要工具软件—Babel进行了详细介绍，并利用实例详细介绍了在Linux平台下Babel的使用方法。

通过实验，我们发现Babel对于算法复杂度不高的程序来说，性能开销是明显的。但是对于Babel真正的应用对象—大规模高性能科学计算程序来说，由于Babel所带来的开销与程序的内部实现及算法复杂度无关，因此这些开销完全是可以接受的。这也是Babel适合高性能科学计算程序的重要原因之一。

目前，不论是CCA规范，还是Babel工具，都处在不断发展、完善的过程当中。虽然Babel已经在一些实际项目当中应用，但是关于它的示例程序却几乎没有。探索各种语言在Babel当中的具体实现方法，以及如何实现远程调用Babel构件等问题都还是需要进一步研究的内容。CCA规范和Babel工具是高性能科学计算领域采用软件构件技术具有代表性的尝试，是此领域中一个比较活跃的研究方向，值得我们关注和借鉴。

参 考 文 献

1 Balay S, Gropp W D, McInnes L C, et al. PETSc homepage. http://www.mcs.anl.gov/petsc, July 1997

2 Balay S, Gropp W D, McInnes L C, et al. PETSc 2.0 Users Manual, Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, Apr. 1998

3 CORBA homepage. http://www.omg.org/corba

4 J2EE homepage. http://java.sun.com/javaee/index.jsp

5 NET homepage. http://www.microsoft.com/net/default.mspx

6 CCA Forum homepage. http://www.cca-forum.org

7 Babel homepage. http://www.llnl.gov/CASC/components/babel.html

8 CCA Forum Tutorial Working Group. Common Component Architecture Tutorial. April 2005

9 CCAFFEINE homepage. http://www.cca-forum.org/~baallan/ccafe

10 XCAT homepage. http://www.extreme.indiana.edu/xcat

11 SCIRun homepage. http://www.sci.utah.edu

12 Dahlgren T, Epperly T, Kumfert G, et al. Babel Users' Guide. version 0.10.8 edition. July, 2005

13 NASA Advanced Supercomputing (NAS) Division homepage. http://www.nas.nasa.gov/NAS/NPB

14 Bernholdt D E, Elwasif W R, Kohl J A, et al. Epperly. A Component Architecture for High-Performance Computing. 2002

15 Kohn S, Kumfert G, Painter J, et al. Divorcing language dependencies from a scientific software library. In: Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing. Society for Industrial and Applied Mathematics, 2001