Linux 软件交互性能异常根源探测系统

何嘉权[1]　陈　渝[2]　茅俊杰[2]　肖奇学[2]　史元春[2]　邢春晓[2]

[1]清华大学软件学院，北京　100084

[2]清华大学计算机科学与技术系，北京　100084

摘　要

近年来，计算机软件的发展和智能设备的普及使得用户对软件用户体验的要求越来越高。为了更充分地利用底层硬件资源，软件开发者在开发过程中采用了诸如多线程编程技术的软件手段，以提高应用软件的用户体验。然而在很多应用软件的使用过程中，依然存在着达不到预期性能的现象，极大影响了用户正常使用软件。而且由于软件开发过程中使用了复杂的开发技术，这些能够感知的交互性能异常现象显得更难分析。为了顺应基于 Linux 的桌面系统和移动操作系统日益普及的潮流，填补 Linux 平台上软件交互异常性能分析的空缺，本文从 Linux 桌面系统上几款流行的应用软件的交互性能异常现象为切入点，设计 Linux 软件交互异常根源探测系统。

关键词：软件交互；性能异常；异常根源；软件动态分析

1　引言

交互性能是应用软件一个很重要的属性，只有保证软件交互效率足够高，才能满足用户对体验的高追求。根据有关研究，软件交互延迟一旦超过 500 毫秒，用户使用软件就会遭遇比较严重的影响[1]。用户能够感知到的应用软件交互性能异常现象的根源有两种情况，一种是软件内部设计逻辑导致的，另一种是软件的运行时的环境导致的。然而广大应用软件的用户不一定能够辨别两种性能异常的情况，而都会直接认为软件的响应速度变得缓慢，软件界面没有在预期的时间内呈现预期的结果。这两种情况都属于软件的交互性能发生了异常。

为了在软件使用过程中出现交互性能异常现象后，能够自动尽快诊断出来异常的根源，并且及时报告给开发者，需要设计相应系统，在软件运行时候识别出软件的交互性能异常，并且推导计算异常现象的根源。类似的动态分析在学术界已经有了不少研究，但他们的工作有的只适用于 Java 面向对象编程技术开发的软件，不具有普遍使用性，有的是专门针对 Microsoft Windows 操作系统展开工作，而专门针对 Linux 的研究工作或分析系统比较少。为了顺应基于 Linux 内核的操作系统日益普遍的潮流，本文将研究总结已有的分析方法，在 Linux 内核的操作系统上实现软件交互性能异常根源探测系统。

2　相关工作

学术界在软件性能异常动态领域已经有了一些的工作。2009 年，Milan Jovic 等人认为传统的分析工具不能找到重要的交互性能异常，于是他们开发相应的工具在实际工作中

资助项目：本课题承蒙国家科技重大专项（2012ZX01039-004，2013ZX01039001-002）

捕获 Eclipse IDE 的 881 个性能问题。[2]他们一方面提出了测量函数调用的延迟是非常有用的；另一方面提出了在软件发布之后的使用过程中发现性能异常的方法。他们以 Eclipse IDE 作为研究对象，选定与 Eclipse 和面向对象编程密切相关 Landmark 函数，对应用软件进行动态插桩来跟踪这些 Landmark 函数。将跟踪结果数据收集起来后，处理分析诊断异常的根源。

除此之外，Shi Han，Tao Xie 等人在 Windows 上通过对和性能异常有关的事件跟踪的信息流进行挖掘和聚类，帮助分析性能异常[3]。他们的工作对象是海量的动态事件流，主要包括以下三步：

(1) AOI 提取 (Area of Interest)：从实际场景中捕捉到的海量事件流中提取出感兴趣部分。

(2) 调用栈模式挖掘：(1) 得到事件及事件的函数调用栈后，挖掘出其中开销较大的部分。

(3) 调用栈模式聚类：(2) 得到开销大的调用栈集合中，去掉重复部分。

最后他们将开发出来的系统 StackMine 应用在 Windows 7 和第三方应用上进行分析。

他们随后还基于事件跟踪的信息流，发现了 38% 的系统性能问题是由设备驱动引起的，26% 是和模块之间交互相关的[4]。对此，他们提出了对软件模块之间的影响分析和根源分析的方法：

(1) 所谓影响分析，是指定了需要分析的软件模块，通过对不同场景的事件流进行分析，根据软件模块的等待时间和运行时间，判断软件模块对性能的影响。

(2) 一旦确定了对性能有影响的软件模块，便对其进行根源分析。

李子拓通过分析线程之间共享资源建立线程之间的依赖关系模型，开发工具分析寻找桌面应用软件的性能异常根源[5]。他的工作认为，线程之间经常有资源竞争的现象，如果竞争资源的线程是 UI (User Interface，用户交互) 线程，则会导致交互性能异常的现象。从这一问题出发，他为线程建立了依赖关系。随后他还提出分析性能异常根源的算法。基于线程依赖关系的模型，他在 Windows 平台上设计了动态分析系统，对 Windows 资源管理器、PowerPoint 等软件访问网络地址或非法地址的性能异常现象做了分析。

3 本文主要成果

本文成果不同于相关工作中的研究。Milan Jovic 的工作[2]只适用于基于 Eclipse IDE 开发的软件，而且绝大多数为 Java 语言编写。Shi Han，Tao Xie 等工作[3][4]和李子拓的工作[5]都是针对 Windows 操作系统展开，最后的系统没法应用在 Linux 平台上。本文的贡献包括：

(1) 总结介绍了 Linux 内核中有助于分析交互性能异常现象的功能特性。

(2) 在 Linux 平台上设计了性能异常现象根源探测系统。

4 系统设计的底层支持

4.1 Linux 内核事件跟踪

Linux 内核，从 2.6.27 版本开始引入了 ftrace[6]。

4.1.1 跟踪点

Linux内核结构庞杂，包含了内存管理、进程调度、网络等多个子系统。其内核系统之复杂，对调试和故障来说无疑带来了不少的障碍。为了解决上述问题，Linux内核开发者在各大子系统中都加入了大量的跟踪点[9]（Tracepoint）。跟踪点实际上是Linux源代码中的特定的标记，通过跟踪点，可以在Linux内核运行过程中对内核进行探测或者插桩。

4.1.2 DebugFS

DebugFS[10]是一个基于内存的特殊文件系统，由Greg Kroah-Hartman于Linux内核版本2.6.10－rc3引入，旨在为内核调试提供一个方便的用户接口支持。通过DebugFS，开发者可以在用户态通过文件读写的方式获得调试信息，并对内核进行分析。Linux的内核事件跟踪，就是以DebugFS作为用户态的接口。

4.1.3 ftrace

通过ftrace可以跟踪几乎所有的内核公共函数的调用和返回，同时也可以跟踪分散在内存管理、进程调用、系统调用等各个子系统中的上千种内核事件。它的实现架构依赖了跟踪点、DebugFS等多个部分的功能，对于不同子系统的事件跟踪，还需要子系统内部的支持。

4.2 Linux性能计数器

Linux性能计数器[11]（Performance Counters for Linux，PCL），是Linux内核中的一个子系统，其用户态前端称为perf，是一个常用的Linux内核性能分析工具。其相应的内核支持，以及唯一的系统调用sys_perf_event_open，从2.6.31版本起引入到Linux内核中。

4.2.1 事件与性能计数器

操作系统在计算机上的工作过程中实际上伴随着大量的事件，通过perf的支持可以对这些事件进行测量统计。这些事件，一部分是来自内核的事件，例如上下文切换、页缺失等。基于perf可以开展开销非常低的性能分析。首先通过软件和硬件多个层面的计数器，可以对软件运行过程中关心的事件进行探测和统计，为特定现象的诊断提供信息，也可以基于这些事件进行采样统计。

4.2.2 基于事件的采样

在使用基于硬件事件的采样时，会触发不可屏蔽中断（Non maskable Interrupt，NMI），强制系统完成采样工作。在采样阶段，可以捕获到当前软件的执行状态，例如处于用户态还是处于内核态，指令地址，当前进程信息等。通过控制采样的频率，就可以按特定速度在软件运行过程中采样。通过采样得到的动态信息能还原目标软件的函数调用栈信息。

4.3 控制组群

控制组群[12]（control group，简称cgroups），是Linux内核对一组进程的资源（CPU、内存、I/O等）进行控制和隔离的功能。该项目最早由Google的工程师Paul Menage和Rohit Seth于2006年发起，在2007年命名为cgroups，并合并到2.6.24版本的Linux内核中。

5 系统设计

基于第 4 章的 Linux 内核相关技术，本文最后设计实现了 Linux 软件交互性能异常根源探测系统。系统包括事件跟踪管理、线程关系解析、动态数据采集三个模块，最后通过 Linux 控制组群部署。

5.1 模块设计

本文将已有的动态分析方法做了适当调整，使其能够用在 Linux 环境中。最终的系统包含了事件跟踪管理模块，线程关系解析模块动态数据采集模块三个主要模块，如图 1 所示。

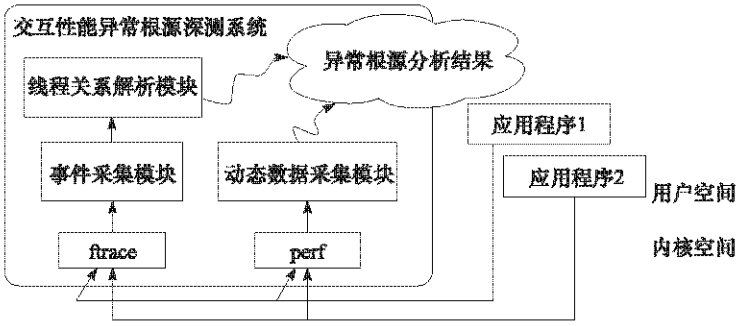

图 1 软件交互性能异常根源探测系统架构图

5.1.1 事件跟踪管理模块

事件跟踪管理模块是建立在 ftrace 之上的。首先通过 ftrace 不断地监视听 Linux 内核事件，将捕捉到事件加入到事件队列，供其他模块分析处理。

5.1.2 线程关系解析模块

由于软件交互性能异常会在线程之间传播，为了探测出交互性能异常的根源，需要根据线程依赖关系模型[5]的依赖关系发现算法，分析出线程的依赖关系。

根据本文对实际场景的研究，软件交互性能异常发生时，往往存在睡眠的线程，而且每次睡眠的线程被另一个线程唤醒，可以认为前者依赖于后者。线程关系解析模块正是通过这种方式发现线程依赖关系的。

由于线程依赖关系并不总是能够发现软件性能异常的根源，根据开发者的经验，这种场景需要软件运行的动态状态辅助。动态数据采集模块基于 perf 采样记录功能来开发，可以与应用程序的符号表和调试信息配合工作。将 perf 采样得到的软件调用栈信息，以一定时间段进行统计，便能从某种程度上反应软件的运行状态。

5.2 系统部署

在内存资源严重不足等极端场景，为了让性能异常根源探测系统能够继续正常运行，避免系统本身遭遇性能异常或者被 OOM Killer 强行终止，本文的实验从系统启动开始，

限制除了探测系统以外的所有进程能访问的资源。限制资源的方案是基于4.3章节中提到的cgroups实现的。实现方式如图2所示。

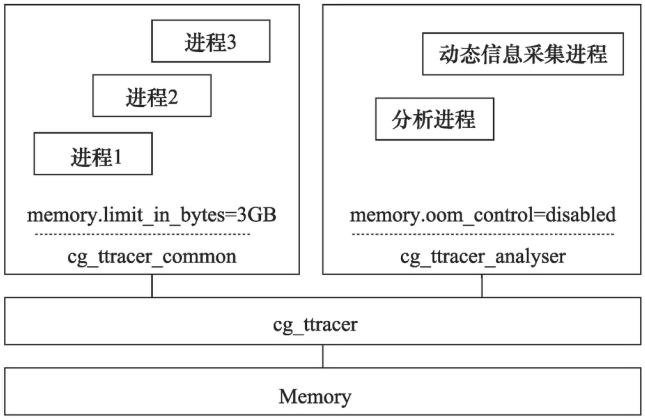

图2 cgroups管理下探测系统工作环境

6 系统应用

6.1 LibreOffice Writer打开特定文件时用户交互体验下降

实验采用4.3.72 430（Build：2）。这个性能缺陷源于LibreOffice Writer在线Bugzilla的Bug 76260[13]。当LibreOffice Writer打开附件中的docx文件时，需要消耗非常长的时间来读取文件，如图3所示。

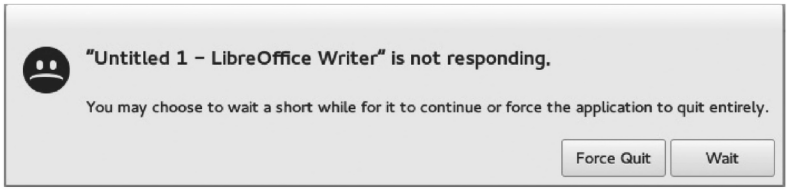

图3 LibreOffice Writer打开特定文件时用户交互出现卡顿，并弹出强行关闭询问窗口

此时我们感兴趣的是软件在执行的操作。经过软件交互异常根源探测系统检测得到这个过程中软件的动态运行状态，也就是每5秒为单位，软件执行频率最高的栈顶函数，按照频率从高到低排列为：

（1）218 430秒到2 184 535秒之间：函数SubstitutePathVariables::GetWorkVariableValue命中16次；函数GtkData::Yield命中8次；函数PaperInfo::getSystemDefaultPaper命中8次。

（2）2 184 535秒到2 184 540秒之间：函数XML_ParseBuffer命中52次；函数operator new命中21次；函数dlopen_doit命中16次；

（3）2 184 540秒到2 184 545秒之间：函数XML_ParseBuffer命中97次；函数writerfilter::ooxml::OOXMLFactory::attributes命中30次；函数operator new命中46次。

（4）2184545秒到2184550秒之间：函数XML_ParseBuffer命中101次；函数

writerfilter::ooxml::OOXMLFactory::attributes 命中 58 次；函数 operator new 命中 26 次。

根据实验的结果可以看到，在用户选择打开附件中脚注较多的 docx 文件时，LibreOffice Writer 遭遇交互性能异常的根源是它在频繁执行名字为 XML _ ParseBuffer 的函数。根据对 docx 文件的既有认识，docx 文件中实际上用 XML 格式存储着较多信息。知道了 LibreOffice Writer 性能异常发生时频繁执行的函数，可以为开发者的诊断提供帮助。

6.2 OpenSSH 服务器内存耗尽时客户端交互受阻

实验采用 OpenSSH_6.6.1p1。OpenSSH 服务器程序是广泛应用于各种服务器的远程连接服务端软件，只要在服务器上架设了 sshd 服务，则可以在其他机器上通过 SSH 客户端远程连接操作服务器。在实际生产或实验环境中，服务器软件存在内存泄露现象，或消耗内存过大没及时释放，导致内存耗尽的现象时有发生。此时通过 SSH 客户端远程连接服务器，用户也会感觉交互受阻，例如在命令行中进行操作之后，没有看到期待的服务器的反馈。在这个场景下，交互响应体现在客户端，然而我们尝试从服务器端角度分析这一交互性能异常现象。实验采用 cgroups 对内存使用的控制，来模拟服务器内存紧张的场景的步骤如下：

最后得到的线程状态如图 4 所示。其中进程 514 是 sshd 为了处理客户端连接而创建的进程，而后根据 sshd 的设计逻辑，进程 514 又创建了进程 2319。随后进程 514 陷入睡眠，并且两次被进程 2319 唤醒。

图 4 OpenSSH 内存耗尽场景中的线程关系

通过观察这个过程中的关键的事件片段，我们可以推断进程 514 的行为。如表 1 所示，我们可以看到在这个过程中存在两个时间较长的睡眠，从 182 466 秒到 182 487 秒的睡眠，以及从 182 487 秒到 182 499 秒的睡眠。这两段时间的睡眠正好和内存耗尽的时间相吻合。第一次内存资源可用时，进程 2319 于 182 487.855 秒时刻唤醒了进程 514，第二次内存资源可用的时候，进程 2319 于 182 499.492 秒时刻唤醒了进程 514，最后 514 于 182 499.599 秒时刻完成并退出。

表 1 OpenSSH 内存耗尽场景实例的事件流

时间（秒）	事件	参数	注释
…	…	…	…
182466.813	WAKEUP	514—>2319	进程 514 唤醒进程 2319
182466.813	ENTER_POLL	514	进程 514 调用 sys_poll 睡眠
182487.855	WAKEUP	0—>2319	进程 0 唤醒进程 2319
182487.855	WAKEUP	2319—>514	进程 2139 唤醒进程 514
182487.855	EXIT_POLL	514	进程 514 sys _ poll 返回
…	…	…	…
182487.870	WAKEUP	9—>514	进程 9 唤醒进程 514
182499.492	WAKEUP	0—>2319	进程 0 唤醒进程 2319
182499.492	WAKEUP	2319—>514	进程 2319 唤醒进程 514
…	…	…	…
182499.599	EXIT	514	进程 514 结束

一方面，根据线程依赖关系模型，最后得到的结果是进程514依赖于进程2319，但没有得到更详细的信息。另一方面，由于进程514和进程2319在这个过程中存在两次不可打断的睡眠时间，如果尝试通过perf对进程动态执行状态进行采样，也不会得到任何信息，原因是在那两段睡眠时间内，sshd进程一直没有占据任何CPU资源，所以在事件触发的采样时刻，当前正在执行的进程并非sshd进程，采样时刻没有命中sshd进程，也就没法得到sshd进程的函数调用栈信息，这是合理的现象。

然而，通过记录进程在运行的过程中使用内存的情况可以发现，如图5所示，内存的使用情况和进程睡眠状态相吻合，开发者根据这一信息可以对该现象做出诊断。但是完成这一信息的采集，前提是在这个过程中交互性能异常分析系统一直处于正常工作状态，这也就依赖于3.4章节中提到的相关背景技术。

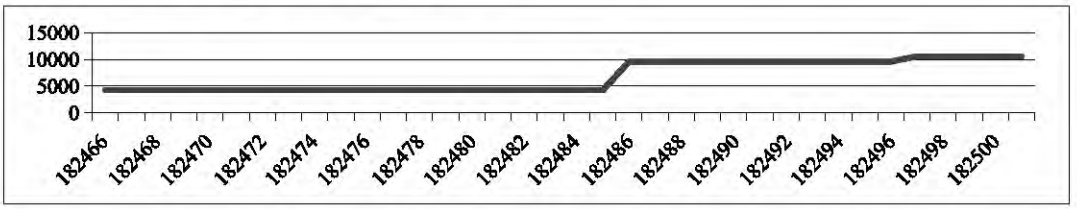

图5 OpenSSH内存使用趋势，其中横坐标为时间（秒），纵坐标为OpenSSH的内存使用情况（字节）

7 总结

软件的用户交互体验是软件很重要的属性。发生交互性能异常后，及时探测异常根源对开发者来说很有帮助。本文借鉴了前人在软件性能异常分析领域的动态分析方法，在Linux平台上研究总结了有助于分析软件交互性能异常的技术，并设计实现了能够捕获软件交互性能异常并能够自动探测异常根源的系统，填补了专门针对Linux平台的软件交互性能异常研究工作的空白。根据实验结果，该系统能够分析出大型软件实际情况的交互性能异常根源，理论上来说，该系统能够分析所有运行在Linux平台上的应用软件。

本文设计的系统能分析软件交互性能异常，但系统本身存在较多缺陷，例如借鉴了前人的线程依赖关系模型，但尚未根据更多场景对模型进行深入的研究优化，现实中存在更多情况并不适用已有模型。而且虽然系统已经能够多数软件并探测出交互性能异常现象根源，但还需要更多优化，以便应用于在线分析。

参考文献

[1] Liu Zhicheng, Jeffrey Heer. The Effects of Interactive Latency on Exploratory Visual Analysis[J]. TVCG, 2014, 20: 2122-2131.

[2] Jovic Milan, Andrea Adamoli, Matthias Hauswirth. Catch me if you can: performance bug detection in the wild[J]. ACM SIGPLAN Notices, 2011, 46(10).

[3] Han Shi, et al. Performance debugging in the large via mining millions of stack traces[C]. Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012.

[4] Yu Xiao, et al. Comprehending performance from real-world execution traces: A device-driver case [C]. Proceedings of the 19th international conference on Architectural support for programming

languages and operating systems, ACM, 2014.
[5] 李子拓. 桌面操作系统中交互性能异常的诊断[D]. 北京：清华大学计算机系，2008.
[6] Ftrace-Function Tracer[EB/OL]. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.
[7] Using the Linux KernelTracepoints[EB/OL]. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.
[8] DebugFS[EB/OL]. https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt.
[9] Perf(Linux)[EB/OL]. http://en.wikipedia.org/wiki/Perf_(Linux).
[10] Cgroups[EB/OL]. http://en.wikipedia.org/wiki/Cgroups.
[11] Bug 76260-FILEOPEN: extremely slow loading. docx with lots of footnotes[EB/OL]. https://www.libreoffice.org/bugzilla/show_bug.cgi?id 76260.

Software Interactive Anomaly Root Cause Discovering System on Linux

He Jiaquan[1] Chen Yu[2] Mao Junjie[2] Xiao Qijie[2] Shi Yuanchun[2] Xing Chunxiao[2]

[1]School of Software, Tsinghua University, Beijing 100084
[2]Department of Computer Science and Technology, Tsinghua University, Beijing 100084

Abstract

The development and popularity of computer software as well as smart devices leads to higher demand for user experience. Software developers employ advanced programming techniques such as multi-threading, to take full advantages of hardware technology and improve software user experience. However, in many situations, users are still suffering from software interactive performance anomalies, which are difficult to analyze because of complicated programming techniques. Though Linux-based desktop and mobile systems are becoming more and more popular, few interactive anomaly-analyzing systems designed for Linux exist. This paper will introduce our designing of a software interactive anomaly root cause discovering system on Linux.

Key words: Software interaction; Performance anomaly; Anomaly root cause; Software dynamic analysis