Pview:一种基于PMU的支持并行程序性能分析的新方法

闫 洁[1] 徐恒阳[1] 安 虹[1,2] 刘 玉[1] 王耀彬[1]
(中国科学技术大学计算机科学与技术系 合肥 230027)[1]
(中国科学院计算技术研究所系统结构重点实验室 北京 100080)[2]

摘 要 近年来,随着并行编程的普及,性能监测和剖析已经成为计算机系统领域最重要的研究课题之一。PMU(Performance Monitoring Unit),即现代处理器里集成的微体系事件性能计数器,为性能监测提供了底层支持,使得在以极小的额外开销和极少的对目标程序的干扰的情况下对程序进行性能监测成为可能。Pview(Performance View)是一种在系统级支持对并行程序尤其是多线程程序进行性能监测与分析的工具,它同时支持全系统和针对特定进程(线程组)的性能事件直接计数或者抽样的分析方法。Pview在Linux操作系统平台上通过扩展内核2.6.30,实现了一个新的系统调用Pview来提供性能监测服务;同时与以模块方式实现的数据收集引擎协作,可以实现抽样并将大规模样本数据传输到用户空间供进一步分析。

关键词 硬件性能计数器,性能监测,多线程程序分析
中图法分类号 TP311.56,TP302.7 **文献标识码** A

Pview:A Novel Implementation of Fundamental Supports for Parallel Programs Performance Monitoring Based on PMU

YAN Jie[1] XU Heng-yang[1] AN Hong[1,2] LIU Yu[1] WANG Yao-bin[1]
(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)[1]
(Key Laboratory of Computer System and Architecture, Chinese Academy of Sciences, Beijing 100080, China)[2]

Abstract In the past recent years, performance monitoring and profiling have been becoming one of the most important issues in the era of parallel programming. Performance Monitoring Unit(shortly PMU in the rest of this paper) embedded in the modern processor provides a hardware support that makes it possible to monitor a running program online with trivial system disturbance and overhead and further do more post-analysis in scrutiny based on the collected data. Pview, meaning performance view, is a novel scheme we designed to fundamentally support performance monitoring for multi-threaded applications in system level, and it is implemented in Linux 2.6.30 platform by extending the kernel. Pview uses an approach of system call to request performance monitoring services, and also co-works with a module called pview data collector engine to feed user analyzer tools the collected performance event data. This paper presented all of our minds in pview design.

Keywords PMU, Performance monitoring, Multi-threaded program analysis

1 概述

1.1 背景介绍

在过去几年里,单核处理器性能提升的速度逐渐慢下来,与此同时,片上多核处理器设计成为主流。为了将增加的处理器核转变为实际的性能提升,多线程编程方法变得前所未有的重要。在并行编程实践中,如何获得实际的高性能已经超出了传统的基于算法复杂度分析;通过实际运行程序来进行在线监测的性能剖析与调优的方法就变得尤其重要。

两类技术被用来对程序运行情况进行在线采集。第一类是instrumentation,通过在程序中静态或者动态地插入额外的代码来观察程序的实际行为;这种方法被广泛用于帮助开发者理解程序执行踪迹和与系统的交互行为。Instrumentation方法目前已经比较成熟,其中专注于用户程序分析的代表性工作包括Intel PIN[1],SUN Dtrace[2]和ATOM API[3]等。另一类是基于性能事件(如执行周期数、指令数、cache miss率和分支误预测率等)的剖析[9-11],为开发者提供目标程序运行引起的微体系性能事件特征。同时,借助于抽样方法

到稿日期:2010-03-20 返修日期:2010-06-12 本文受国家科技重大专项项目(2009ZX01036-001-002),国家自然科学基金重点项目(60633040),国家自然科学基金项目(60970023),国家973计划项目(2005CB321601),国家863计划重大项目(2006AA01A102),国家863计划项目(2009AA01Z106)资助。

闫 洁(1985-),男,硕士生,主要研究方向为计算机系统结构,E-mail:jieying@mail.ustc.edu.cn;徐恒阳(1985-),男,硕士生,主要研究方向为计算机系统结构;安 虹(1963-),副教授,主要研究方向为计算机系统结构;刘 玉(1987-),男,硕士生,主要研究方向为计算机系统结构;王耀彬(1982-),博士生,主要研究方向为计算机系统结构。

和编译技术，可以将这些提取出的数据特征与其在应用程序代码中的位置相对应，帮助开发者反思程序设计本身的问题，如性能瓶颈和并行编程中的任务划分失衡等。与 instrumentation 方法不同，第二类方法并不涉及任何关于算法本身的直接信息；同时大部分这类工具都基于硬件 PMU[4]，因而对程序本身的执行几乎不会造成明显干扰。这类方法中，Intel VTune[5]，Oprofile[6]，perfmon2[7] 和 Perf[8] 等都是其代表性工作。

Pview 也属于上述第二类方法的工作，但是同时以系统调用的形式提供了一组 API 可供 instrumentation 方法调用来对一段代码进行性能事件统计。

1.2 相关工作

基于 PMU 的性能监测工具，有两种数据收集方法，即直接计数和抽样，监测范围为全系统或者针对特定进程。

Intel VTune 支持全系统抽样的数据采集方法，并且可以将抽样信息对应到目标程序的汇编代码和源代码中。在其 Linux 版本中，底层的数据收集模块是以 module 的形式实现的。Oprofile 是 Linux 下的一个工具，其内核层部分使用了同样的实现方式来支持基于全系统抽样的数据收集方法，提供基本的和 VTune 类似的功能。目前，Oprofile 驱动模块作为 Linux Kernel 中 mainline 的一部分发布。

Perfmon2 实现了一组系统调用，以提供全系统和针对特定进程的直接计数或者抽样的数据收集方法，同时实现了一个 module 来支持各种其他 Linux 下的同类工具，如 Oprofile 的抽样数据格式。另外，Perfmon2 支持 PMU 的虚拟化和 multiplexing 使用。在功能和硬件平台支持上，Perfmon2 非常完备。但是对 Linux 内核改动过多，例如其系统调用部分使用了多达 12 个系统调用号。

吸取了 Perfmon2 的一些经验，Ingo 等人在内核核心实现了另一种 PMU 支持方案，即 Perf。Perf 只需要一个系统调用，并且对内核核心的其他代码几乎不造成影响。

这些工作的侧重点在于多平台和多功能支持，除 VTune 外，对并行程序本身以及系统实现多线程的方式并没有作为主要考虑因素。Pview 除了吸收这些工作的优点外，另外着重考虑了对多线程程序进行监测的内核部分支持。

1.3 Pview 的特点和优势

与上述同类工具相比，Pview 的特点包括以下几点。

首先，Pview 同时支持全系统和针对特定进程的直接计数和抽样的数据收集方法。目前，在应用程序剖析和调优中，全系统抽样和针对特定进程的性能事件计数是两种最有用的数据收集方法；但是其他两种组合也是有用的，比如将性能事件计数的总体(population)选为目标分析程序的进程而不是全系统，可以避免随机性假设失败时的抽样偏差。

第二，Pview 可以通过创建一个新进程并加载目标程序代码的方式来对还没有运行的应用程序进行分析；此外，Pview 实现了 attach 和 detach 功能，可以对系统中已经运行着的程序在指定时间段进行性能分析。

第三，Pview 提供了一个统一的接口，使得性能分析成为一种基本的系统服务。所有的控制操作都是由 Pview 系统调用完成的，该系统调用可以在线导出指定进程的性能事件统计数据；还可将抽样过程以及样本数据的存储和传输分离出来由独立的数据收集引擎负责。

第四，Pview 通过内核扩展对多线程程序分析提供了完全支持；Pview 可以记录所有被监测进程的性能统计数据，有效管理大量线程的动态创建和撤销时的性能数据，当线程数目和活动处理器核数大量增加时，不会引起性能事件记录需要的存储空间以及额外时间开销的明显上升。

2 Pview 软件架构

Pview 工具集由用户层的前端与内核层的后端两部分组成。前端包括分析器和库，后端包括数据收集引擎模块和内核核心扩展，前后端通过系统调用 Pview 和设备伪文件系统进行通信。图 1 给出了基于 Pview 的性能监测软件架构。

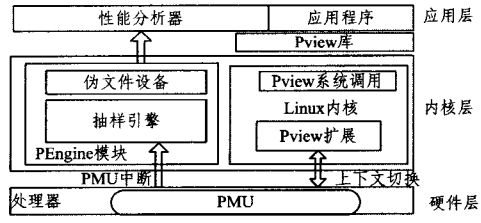

图 1 基于 Pview 的性能监测软件架构

不论是分析器还是库，都通过系统调用 Pview 来完成与内核和硬件 PMU 的交互；抽样过程会产生大量的原始数据，我们实现一个伪文件系统设备，通过这个设备，用户层可以获得这些数据供以后分析。

Pview 库是对 Pview 系统调用的封装，在底层功能上与 Pview 系统调用是等价的；用户代码可以直接调用库函数，对一段代码或一次程序的运行进行性能事件统计或者抽样分析。在库的基础上可以实现各种性能分析工具，分析器就是一个这样的实现。

Pview 后端运行在内核态，主要有两部分。一部分是对内核核心的扩展，新添加了一个系统调用(即 Pview)来对用户提供统一的性能监测服务接口；其内部实现对进程上下文进行了扩展(添加了 pview context)，复用了部分 Unix Ptrace 的进程控制机制，为多线程程序的分析提供了强有力的支持。另一部分是数据收集引擎模块，完成抽样及其数据处理，并由其中的伪文件设备传输到用户空间；这部分以驱动模块的形式实现。

本文主要讨论 Pview 的后端实现部分，第 3 节介绍 Pview 系统调用的功能；第 4 节介绍数据收集引擎模块的功能；第 5 节讨论 Pview 的使用模式；第 6 节介绍内核扩展中的关键技术，包括对多线程程序分析的支持；最后是总结。

3 Pview 系统调用

系统调用 Pview 实现了一个系统对用户提供性能监测服务的统一接口，其函数声明如下：

asmlinkage long sys_pview(long request, long pid, long addr, long data)

Request 参数决定系统调用的功能，pid 为目标进程的进程号，addr 和 data 两个参数在传递设置参数或者传出统计数据时用来指定地址或其他附加信息。它主要完成以下 4 类功能。

第一，设置和维护内核 pview context 结构(pview context 包含了所有关于 PMU 的配置信息和相关的数据处理选项，细节将在后文中说明)。用户并不直接读写 PMU 硬件寄存

器，而是将性能事件、抽样间隔、计数范围(用户空间或内核空间)等配置写入 pview context，然后由内核在合适的时间读写 PMU 寄存器。

第二，性能监测动作控制，如开始、结束和暂停性能监测服务。

第三，进程控制。Pview 通过在内核层复用 ptrace 的进程机制，实现了一种父进程对子进程(或者一组线程)进行性能监测和分析的方法。但是，与 ptrace 不同，Pview 并不干涉进程的执行行为，而仅仅是一个观测者的角色。我们定义处于观测者地位的父进程为 viewer，被监测的进程为 viewee；其中 viewer 负责性能监测的控制和管理收集来的 viewee 进程的性能数据；另外，在全系统分析时，没有 viewee 进程，这时 viewer 仅仅是一个守护进程。如果 viewee 进程尚未启动，viewer 可以创建子进程并对其进行监测，然后子进程运行 viewee 代码。对一个已经运行的进程进行性能监测，则需要将 pview context 结构 attach 到该目标进程并为其分配记录性能事件的数据结构；当结束对目标进程的监测时，则将收集到的性能数据进行综合并在 detach 其 pview context 的同时销毁之前分配的用于记录的数据结构。

第四，在线获取性能事件的统计数据。Pview 系统调用可以在线获取指定进程的当前性能统计数据，只需要利用 addr 参数指定一个用户空间地址即可；如果要导出 viewee 进程的整个线程组的性能统计数据视图，则需要使用内存文件映射的方式。

Pview 系统调用负责 PMU 抽样参数的设置，但是其本身包括的内核核心部分的实现里没有抽样的函数。抽样过程以及之后将抽样数据传输到用户空间，是由独立的数据收集引擎模块完成的；Pview 进行设置时会检查该模块是否已经加载到系统。

Pview 库对 Pview 系统调用进行了封装，提供了友好的用户层 API；以下在用户层提到 Pview 系统调用时均指 Pivew 库函数。

4 数据收集引擎模块

数据收集引擎模块(以下简称 PEngine)包括 PMU 中断抽样例程和伪文件设备两部分，由 PMU 溢出中断驱动。在加载该模块时，即向操作系统注册其 NMI 中断处理例程；当 PMU 的计数寄存器达到设定的间隔值溢出时，进行抽样，并重置引起该中断的计数器的值；抽样的数据暂存于 PEngine 分配的内核存储结构并在合适的时候导出到用户空间。

PEngine 自己定义抽样属性(如指令地址 IP、进程/线程号和系统时间等)及其数据格式以及中断处理时向系统登记哪些统计信息；但是 PEngine 自己不提供 PMU 相关的用户控制接口。

PEngine 的抽样过程和抽样数据存储结构与 Oprofile 完全相同，其实现是在 Oprofile 基础上改写的。利用 sysfs 设备机制，为用户提供调整 PEngine 中抽样数据的内核存储空间大小分配等参数的项；但是在 PEngine 中取消了像 Oprofile 中那样向用户暴露的可以通过写入值而启动或者修改处理器 PMU 配置的那些项。

PEngine 同时实现了一个用于向用户空间传输大量抽样数据的伪设备文件系统，可以由用户空间守护进程读出并进行分析。

5 使用 Pview 进行性能监测与分析

Pview 中，由 Pview 系统调用提供统一的性能监测的设置和控制接口，由 PEngine 模块的设备文件提供抽样数据。

使用 Pview 进行性能监测和分析的基本过程如图 2 所示。viewer 进程需要首先创建并设置 Pview context；之后设置 Pview 选项，主要包括修改 pview context 进程继承开关和对 viewee 返回的数据进行处理的 aggregation 函数指针选项。viewer 完成以上设置工作后，可以用图 2 中所示的任何一种使用方式对 viewee 进行性能监测。对于监测数据的用户层处理，包括作为守护进程从 PEngine 的设备文件系统读取并处理大规模抽样数据，都由 viewer 完成。

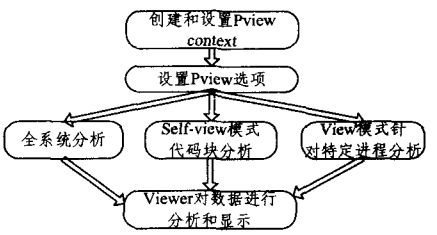

图 2 利用 Pview 进行性能监测的流程图

5.1 全系统范围的性能监测

全系统范围内的性能监测，无论直接计数还是抽样，都不需要干涉具体进程的执行及其与系统的交互行为。viewer 进程仅仅作为一个守护进程，在使用期间维护 pview context，收集与处理性能事件数据，并对用户提供一个控制性能监测操作和显示的界面。将 viewer 进行扩展，就是用户层的性能分析工具界面，例如图 1 软件架构图中的分析器。

Pview 在全系统分析方面可以提供与 Intel VTune 和 Oprofile 等价的底层支持。

5.2 针对特定进程的性能监测

Pview 作为一种基本的系统服务，针对特定进程可以提供两种模式的性能监测，即 view 模式和 self-view 模式。

5.2.1 view 模式

在 view 模式下，viewer 对 viewee 进程进行性能监测。

viewer 进程可以通过利用 execv 系统调用创建一个新进程并以加载 viewee 代码的方式启动对目标程序的监测；简化后的代码如图 3 所示。

```
pid_t viewee;
viewee = fork ();
if (viewee == 0){
  pview (PVIEW_VIEWME, 0, 0, 0);
  execv (program, allargs);
}
wait(viewee);
pview (PVIEW_CONT, viewee, 0, 0);
wait(viewee);
```

图 3 viewer 创建一个进程并对它进行性能监测的伪代码

如果要结束对该 viewee 的监测，viewer 作为父进程可以直接将 viewee 进程杀死，也可以等 viewee 进程运行完毕。

如果是对已经运行着的进程进行监测，则需要 Pview 的 attach 和 detach 机制；viewer 可以通过调用 Pview(PVIEW_ATTACH，pid)和 Pview(PVIEW_DETACH，pid)来实现对进程 id 为 pid 的 viewee 进程进行监测。

viewee 进程可以是多线程程序，viewer 通过设置开关 PVIEW_O_INHERITABLE 可以决定 viewee 里创建的线程是否被监测。

5.2.2 self-view 模式

在 self-view 模式中，不存在独立的 viewer，viewer 与 viewee 是一体的；这种模式一般以用户自己在源代码里调用 Pview 库函数的方式使用；其与前面的 view 模式相似，但是有两点重要的区别。

首先，需要在源代码里显式地创建和初始化 pview_context，如果在主进程里创建了 pview context，那么主进程就是其 owner，并且可以被其子进程继承；如果在子进程里初始化或者重新初始化了 pview_context，则仅对该子进程有效。

第二，由于没有独立的 viewer 进程，因此如果设置了针对特定进程的抽样功能，那么只能通过 pview 请求得到抽样的统计数据；如果需要具体的抽样数据，则必须在该进程外启动一个守护进程来通过数据收集引擎的伪文件设备接受抽样数据。

图 4 为 self-view 模式下的 pview API 调用示例。viewer_A 为对多线程进行性能事件直接计数，建立 n 个线程并继承 pview context，当 n 个线程运行结束后，将进程组性能事件统计数据以文件形式导出；viewer_B 为对指定的代码段进行性能事件直接计数，利用 PVIEW_GET 功能获得其在线值。

```
Viewer_A() {
  ...
  pview(PVIEW_CTX_CREATE, 0);
  pview(PVIEW_COUNTING_SET, 0, addr, nr);
  pview(PVIEW_OPTIONS_SET, 0, 0, PVIEW_O_INHERIT);
  pview(PVIEW_START);
  {
    fork_nthreads(n);
    do_thread(i);
  }
  wait();
  pview(PVIEW_DUMP,0,stat_fd,0);
  pview(PVIEW_STOP);
}
Viewer_B() {
  ...
  pview(PVIEW_CTX_CREATE, 0);
  pview(PVIEW_COUNTING_SET, 0, addr, nr);
  pview(PVIEW_OPTIONS_SET, 0, 0, PVIEW_O_INHERIT);
  pview(PVIEW_START);
  {
  /*待分析代码段*/
    ... ...
  }
  pview(PVIEW_PAUSE);
  pview(PVIEW_COUNTING_GET,0,stat,0);
  pview(PVIEW_STOP);
}
```

图 4 self-view 模式下的 pview 伪代码

6 Pview 内核扩展的关键技术

基于全系统抽样的性能数据分析，其内核空间部分用驱动模块的方式实现是最好的方法。这种实现几乎完全不用修改内核核心代码，因为 PMU 设置和抽样过程只涉及到内核级运行权限，并不对进程执行上下文或系统核心部分的其他数据结构进行任何修改操作。

但是，无论直接计数还是抽样的方法，针对特定进程（线程组）的分析都需要在上下文切换时对被监测进程（线程组）分别保存和恢复其性能计数器值。另外，还需要在进程创建和退出时进行特殊处理。这些操作必然要求对操作系统内核核心的进程管理部分代码进行修改。本节讨论 Pview 针对特定进程（线程组）性能监测实现的一些关键技术。

6.1 多线程程序性能分析支持机制

Linux 系统里的用户线程，不论何种实现，在内核看来实际上是共享同一进程空间的特殊进程；而性能事件的数据收集也是在内核空间进行的。为了在内核层支持对多线程程序进行性能监测，我们主要做了两个专门的工作，即引入进程控制机制和将记录进程性能事件计数的数据结构从 pview context 中分离。

首先，Pview 实现了一个简单的进程控制机制，由于重用了 ptrace 机制的一些标志位和调试状态下的子进程链表，Pview 不可以和 ptrace 的其他功能一起使用；另外，由于基于 PMU 的性能监测并不试图影响或者观察目标程序进程的执行行为，因此不能将 Pview 合并到 ptrace 下实现。在 Pview 服务使用期间，viewer 成为 viewee 的父进程，拥有中断和恢复 viewee 进程执行的特权；viewee 在运行完毕退出时，需要通知 viewer 并将其统计性能事件的记录结构交给 viewer 处理，同时解除 pview context 引用。

另一方面，我们将配置信息结构与记录结构分离成两个独立的数据结构；在内核核心部分，我们分别定义这两个数据结构为 pview context 和 pview stat。Pview context 数据结构包含了基于 PMU 进行性能监测的所有相关寄存器配置信息和数据处理函数指针，pview stat 结构存储 PMU 寄存器值和统计信息。图 5 为运行中的 viewer 与 viewee 内核数据结构的关系示例。

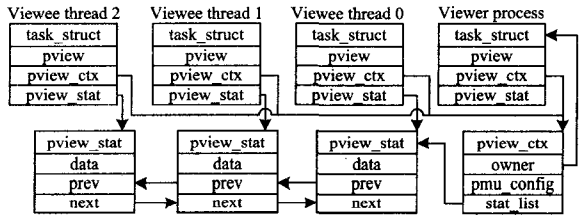

图 5 pview 运行中的 viewer-viewee 关系示意图

Pview context 中的 owner 域指向 viewer 进程，由该 viewer 进程创建、拥有和维护；viewer 和 viewee 进程 PCB 结构 task_struct 中都有引用该 pview context 的指针，但是对于 viewee 只是在需要的时候读取其中的信息，没有权利进行修改；pview context 采用读写锁进行保护。当 viewee 退出或停止被监测时，将把其 pview stat 结构置成非活动，并通知 viewer，viewer 根据 pview context 中的数据处理函数（即 aggregation 函数）对该 viewee 进程交回的统计数据进行处理，然后释放该 pview stat。

每个被监测的线程都有自己独立的 pview stat 结构，每次上下文切换，都通过其 pview stat 中的数据域保存和恢复（实际采用的是 lazy update 的方法，后文将有叙述）PMU 寄存器值，不需要读取 pview context 结构。当 viewee 进程标志被设置了 pview 位进行监测时，创建 pview stat 并将其进程对象结构 task_struct 中的相关指针指向它；然后，将该 pview stat 加入其 pview context 中的 pview stat 链表。当 viewee 线程退出或者 pview detach 时，只解除该进程对 pview context 和 pview stat 的引用，然后通知其父进程（viewer）来处理。

当 viewee 是多线程程序时，可以通过设置父进程 task_struct 中的 pview 变量中的 PV_O_INHERITABLE 位，来决定新建立的线程是否继承其父进程的 pview context 和 pview 操作；也就是说，我们可以选择新创建的线程是否被监测。

这种统计数据统一管理和上下文切换时分布式控制相结合的机制既可以避免多处理器平台多线程程序访问共享变量时使用锁引起的性能损失，也对多线程实现了完整跟踪。我们可以导出整个 viewee 线程组的整个性能数据统计视图；另外，我们在 pview context 中定义了 aggregation 函数指针，可

以对 viewee 线程的统计数据进行归并，然后及时释放非活动的 pview stat 结构。

6.2 进程上下文切换时 PMU 操作的优化

在大多数集成了 PMU 的处理器（例如 IA、MIPS 和 POEWER 系列）中，PMU 作为特殊的寄存器不属于执行上下文寄存器组的范围；无论从概念上还是从其访问特点来看，把 PMU 寄存器划分为硬件执行上下文都是不合适的，因此不能通过修改内核里执行上下文的数据结构来扩展 PMU 上下文。我们把 PMU 上下文视为进程的一个特殊属性。

Pview 对上下文切换时 PMU 的 save/restore 操作做了两点重要优化。

第一，在进程的硬件上下文描述数据结构 thread_struct 中添加一个长整型变量 pmu_bitmap，以位图形式指示当前 PMU 寄存器对的使用情况；pview_stat 结构中保存 PMU 控制寄存器值和计数寄存器值。在上下文切换时这两个数据结构就可以保证 PMU 的保存与恢复；由于这两个结构都是线程私有的，因此监测在多处理器平台上运行多线程 viewee 程序时，可以不需要任何锁相关的操作。Pview context 结构只在 viewee 中线程对监测环境初始化、统计数据处理等操作时才去读。

第二，对 PMU 寄存器的恢复采取 lazy update 的策略。PMU 寄存器的操作一般都非常耗时；以 Intel 的 CPU 为例，对 PMU MSR(Model Specific Register)的访问指令 throughput 高达 100 个时钟周期以上[12]，这对于上下文切换这种频繁的调用是难以接受的性能损失。但是，基于两点事实，我们有理由对其 lazy update。首先，程序性能监测过程中，绝大多数情况下，同一个处理器核在一段时间内只会使用一个 pview context。多处理器系统中，同一进程的线程调度倾向于分配到一组核上，由于这些线程共享 pview context，它们的 PMU 配置数据结构也是相同的，因此不需要经常更新 PMU 的控制寄存器。其次，由于进程调度的 CPU affinity 考虑，一个进程总是会优先被调度到上次执行所在的处理器核上执行，这样如果在两次执行期间，该处理器核上执行的其他程序没有使用 PMU，那么上次执行的 PMU 现场就没有被破坏，进而 PMU 的控制和计数寄存器都不需要进行更新，只需开始继续计数。

我们为每个处理器核定义了两个 per-cpu 变量，即 last_pv_ctx 和 last_pv_stat，分别指示该处理器（核）上次使用的 pview context 和 pview stat 的 ID；pview context 的 ID 为一个整数，每次创建 pview context 或修改其中的 PMU 配置相关域，该 pview context 的 ID 值都要自增；pview stat 的 ID 使用拥有它的进程 pid；每次上下文切换时，通过对比所在处理器与被调度进程的这两个变量是否一致来决定是否更新 PMU 寄存器值。

结束语 本文介绍了一种 Linux 下基于硬件性能计数单元的程序性能分析工具——Pview。它支持全系统范围和针对特定进程的性能事件直接计数和抽样的数据收集方法，以系统调用的方式为用户提供了一种利用 PMU 进行性能监测的统一接口，同时在内核层次对多线程程序性能事件数据统计给予了特别的支持；抽样过程是由独立的数据收集引擎以内核模块的形式实现的。

对 Pview 系统调用进行封装后的 Pview 库，可以为用户提供使用 PMU 来获取应用程序性能事件数据的 API；在此基础上，可以进一步构造用户层的工具链。

参 考 文 献

[1] Luk C K, Cohn R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]// Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). Chicago, ILN, US, June 2005: 190-200

[2] Cantrill B M, Shapiro M W, Leventhal A H. Dynamic Instrumentation of Production System[C]// Proc. of USENIX ATEC. Boston, MA, US, June 2004: 2-2

[3] Srivastava A, Eustace A. Atom: A system for building customized program analysis tools[C]// Proc. of PLDI. Orlando, Florida, US, 1994: 196-205

[4] Sprunt B. The basics of performance monitoring hardware[J]. IEEE Micro, July 2002: 64-71

[5] Intel Corp. VTune [EB/OL]. http://www.intel.com/software/products/vtune/

[6] Levon J. Oprofile[EB/OL]. http://oprofile.sourceforge.net/about/

[7] Eranian S. Perfmon2[EB/OL]. http://perfmon2.sf.net/

[8] Molnar I. Perf[EB/DK]. Linux Kernel Document, 2.6.33

[9] Ahn D H, Vetter J S. Scalable analysis techniques for microprocessor performance counter metrics[C]// Proc. of Conference on Supercomputing. Baltimore, Maryland, US, Nov. 2002

[10] Sweeney P F, Hauswirth M, Cahoon B, et al. Using hardware performance monitors to understand the behavior of Java applications[C]// Proc. of 3rd Virtual Machine Research and Technology Symposium. San Jose, CA, US, May 2004: 5-5

[11] Duesterwald E, Cascaval C, Dwarkadas S. Characterizing and predicting program behavior and its variability[C]// Proc. of 12th Intl. Conference on Parallel Architecture and Compiliation Techniques(PACT). Dec. 2003

[12] Intel Corp. Intel 64 and IA-32 Architectures Optimization Reference Manual [M]. Nov. 2009: C. 3

（上接第 263 页）

[3] 倪国强. 基于视觉神经动力学的图像融合与处理技术若干新进展[J]. 激光与红外, 2005, 35(11): 817-821

[4] Luo Yan, Liu Rong, Zhu Yu-feng. Fusion of Remote Sensing Image Base on the PCA + ATROUS Wavelet Transform[J]. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 2008, 37(B7): 1155-1158

[5] Burt P J, Adelson E H. A multiresolution spline with application to image mosaics[J]. ACM Trans. Graphics, 1983, 2(4): 217-236

[6] Garzelli A. Wavelet-based Fusion of Optical and SAR Image Data OverUrban /area[C]// Photogrammetric Computer Vision. ISPRS Commision III, symposium2002. Graz, Austria. 2002: 59-62

[7] 李晖晖, 郭雷, 李国新. 基于脊波变换的 SAR 与可见光图像融合研究[J]. 西北工业大学学报, 2006, 24(4): 418-422

[8] Choi M, Kim R Y, Kim M G. The curvelet transform for image fusion[C]// International Society for Photogrammetry and Remote Sensing, ISPRS. 2004, 35: 59-64

[9] Zhang Qiang, Guo Bao-long. Multifocus image fusion using the nonsubsampled contourlet transform [J]. Signal Processing, 2009, 89: 1334-1346