Perfmon2: a flexible performance monitoring interface for Linux

Stéphane Eranian
HP Labs
eranian@hpl.hp.com

Abstract

Monitoring program execution is becoming more than ever key to achieving world-class performance. A generic, flexible, and yet powerful monitoring interface to access the performance counters of modern processors has been designed. This interface allows performance tools to collect simple counts or profiles on a per kernel thread or system-wide basis. It introduces several innovations such as customizable sampling buffer formats, time or overflow-based multiplexing of event sets. The current implementation for the 2.6 kernel supports all the major processor architectures. Several open-source and commercial tools based on interface are available. We are currently working on getting the interface accepted into the mainline kernel. This paper presents an overview of the interface.

1 Introduction

Performance monitoring is the action of collecting information about the execution of a program. The type of information collected depends on the level at which it is collected. We distinguish two levels:

- the program level: the program is instrumented by adding explicit calls to routines that collect certain metrics. Instrumentation can be inserted by the programmer or the compiler, e.g., the `-pg` option of GNU cc. Tools such as HP Caliper [5] or Intel PIN [17] can also instrument at runtime. With those tools, it is possible to collect, for instance, the number of times a function is called, the number of time a basic block is entered, a call graph, or a memory access trace.

- the hardware level: the program is not modified. The information is collected by the CPU hardware and stored in performance counters. They can be exploited by tools such as OProfile and VTUNE on Linux. The counters measure the micro-architectural behavior of the program, i.e., the number of elapsed cycles, how many data cache stalls, how many TLB misses.

When analyzing the performance of a program, a user must answer two simple questions: where is time spent and why is spent time there? Program-level monitoring can, in many situations and with some high overhead, answer the first, but the second question is best solved with hardware-level monitoring. For instance, `gprof` can tell you that a program spends 20% of its time in one function. The difficulty is to know why. Is this because the function is called a lot? Is this due to algorithmic problems? Is it because the processor stalls? If so, what is causing the stalls? As this simple example shows, the two levels of monitoring can be complementary.

The current CPU hardware trends are increasing the need for powerful hardware monitoring. New hardware features present the opportunity to gain considerable performance improvements through software changes. To benefit from a multi-threaded CPU, for instance, a program must become multi-threaded itself. To run well on a NUMA machine, a program must be aware of the topology of the machine to adjust memory allocations and thread affinity to minimize the number of remote memory accesses. On the Itanium [3] processor architecture, the quality of the code produced by compilers is a big factor in the overall performance of a program, i.e, the compiler must extract the parallelism of the program to take advantage of the hardware.

Hardware-based performance monitoring can help pinpoint problems in how software uses those new hardware features. An operating system scheduler can benefit from cache profiles to optimize placement of threads to avoiding cache thrashing in multi-threaded CPUs. Static compilers can use performance profiles to improve code quality, a technique called Profile-Guided Optimization (PGO). Dynamic compilers, in Managed Runtime Environments (MRE) can also apply the same technique. Profile-Guided Optimizations can also be applied directly to a binary by tools such as iSpike [11]. In virtualized environments, such as Xen [14], system managers can also use monitoring information to guide load balancing. Developers can also use this information to optimize the layout of data structures, improve data prefetching, analyze code paths [13]. Performance profiles can also be used to drive future hardware requirements such as cache sizes, cache latencies, or bus bandwidth.

Hardware performance counters are logically implemented by the Performance Monitoring Unit (PMU) of the CPU. By nature, this is a fairly complex piece of hardware distributed all across the chip to collect information about key components such as the pipeline, the caches, the CPU buses. The PMU is, by nature, very specific to each processor implementation, e.g., the Pentium M and Pentium 4 PMUs [9] have not much in common. The Itanium processor architecture specifies the framework within which the PMU must be implemented which helps develop portable software.

One of the difficulties to standardize on a performance monitoring interface is to ensure that it supports all existing and future PMU models without preventing access to some of their model specific features. Indeed, some models, such as the Itanium 2 PMU [8], go beyond just counting events, they can also capture branch traces, where cache misses occur, or filter on opcodes.

In Linux and across all architectures, the wealth of information provided by the PMU is oftentimes under-exploited because a lack of a flexible and standardized interface on which tools can be developed.

In this paper, we give an overview of *perfmon2*, an interface designed to solve this problem for all major architectures. We begin by reviewing what Linux offers today. Then, we describe the various key features of this new interface. We conclude with the current status and a short description of the existing tools.

2 Existing interfaces

The problem with performance monitoring in Linux is not the lack of interface, but rather the multitude of interfaces. There are at least three interfaces:

- OProfile [16]: it is designed for DCPI-style [15] system-wide profiling. It is supported on all major architectures and is enabled by major Linux distributions. It can generate a flat profile and a call graph per program. It comes with its own tool set, such as `opcontrol`. Prospect [18] is another tool using this interface.
- perfctr [12]: it supports per-kernel-thread and system-wide monitoring for most major processor architectures, except for Itanium. It is distributed as a stand-alone kernel patch. The interface is mostly used by tools built on top of the PAPI [19] performance toolkit.
- VTUNE [10]: the Intel VTUNE performance analyzer comes with its own kernel interface, implemented by an open-source driver. The interface supports system-wide monitoring only and is very specific to the needs of the tool.

All these interfaces have been designed with a specific measurement or tool in mind. As such, their design is somewhat limited in scope, i.e., they typically do one thing very well. For instance, it is not possible to use OProfile to count the number of retired instructions in a thread. The perfctr interface is the closest match to what we would like to build, yet it has some shortcomings. It is very well designed and tuned for self-monitoring programs but sampling support is limited, especially for non self-monitoring configurations.

With the current situation, it is not necessarily easy for developers to figure out how to write or port their tools. There is a question of functionalities of each interfaces and then, a question of distributions, i.e., which interface ships with which distribution. We believe this situation does not make it attractive for developers to build modern tools on Linux. In fact, Linux is lagging in this area compared to commercial operating systems.

3 Design choices

First of all, it is important to understand why a kernel interface is needed. A PMU is accessible through a set of registers. Typically those registers are only accessible, at least for writing, at the highest privilege level of execution (pl0 or ring0) which is where only the kernel executes. Furthermore, a PMU can trigger interrupts which need kernel support before they can be converted into a notification to a user-level application such as a signal, for instance. For those reasons, the kernel needs to provide an interface to access the PMU.

The goal of our work is to solve the hardware-based monitoring interface problem by designing a single, generic, and flexible interface that supports all major processor architectures. The new interface is built from scratch and introduces several innovations. At the same time, we recognize the value of certain features of the other interfaces and we try to integrate them wherever possible.

The interface is designed to be built into the kernel. This is the key for developers, as it ensures that the interface will be available and supported in all distributions.

To the extent possible, the interface must allow existing monitoring tools to be ported without many difficulties. This is useful to ensure undisrupted availability of popular tools such as VTUNE or OProfile, for instance.

The interface is designed from the bottom up, first looking at what the various processors provide and building up an operating system interface to access the performance counters in a uniform fashion. Thus, the interface is not designed for a specific measurement or tool.

There is efficient support for *per-thread* monitoring where performance information is collected on a kernel thread basis, the PMU state is saved and restored on context switch. There is also support for system-wide monitoring where all threads running on a CPU are monitored and the PMU state persists across context switches.

In either mode, it is possible to collect simple counts or profiles. Neither applications nor the Linux kernel need special compilation to enable monitoring. In per-thread mode, it is possible to monitor unmodified programs or multithreaded programs. A monitoring session can be dynamically attached and detached from a running thread. Self-monitoring is supported for both counting and profiling.

The interface is available to regular users and not just system administrators. This is especially important for per-thread measurements. As a consequence, it is not possible to assume that tools are necessarily well-behaved and the interface must prevent malicious usage.

The interface provides a uniform set of features across platforms to maximize code re-use in performance tools. Measurement limitations are mandated by the PMU hardware not the software interface. For instance, if a PMU does not capture where cache misses occur, there is nothing the interface nor its implementation can do about it.

The interface must be extensible because we want to support a variety of tools on very different hardware platforms.

4 Core Interface

The interface leverages a common property of all PMU models which is that the hardware interface always consists of a set of configuration registers, that we call PMC (Performance Monitor Configuration), and a set of data registers, that we call PMD (Performance Monitor Data). Thus, the interface provides basic read/write access to the PMC/PMD registers.

Across all architectures, the interface exposes a uniform register-naming scheme using the PMC and PMD terminology inherited from the Itanium processor architecture. As such, applications actually operate on a logical PMU. The mapping from the logical to the actual PMU is described in Section 4.3.

The whole PMU machine state is represented by a software abstraction called a *perfmon context*. Each context is identified and manipulated using a file descriptor.

4.1 System calls

The interface is implemented with multiple system calls rather than a device driver. Per-thread monitoring requires that the PMU machine state be saved and restored on context switch. Access to such routine is usually prohibited for drivers. A system call provides more flexibility than `ioctl` for the number, type, and type checking of arguments. Furthermore, system calls reinforce our goal of having the interface be an integral part of the kernel, and not just an optional device driver.

The list of system calls is shown in Table 1. A context is created by the `pfm_create_context` call. There are two types of contexts: per-thread or system-wide. The type is determined when the context is created. The same set of functionalities is available to both types

int pfm_create_context(pfarg_ctx_t *c, void *s, size_t s)
int pfm_write_pmcs(int f, pfarg_pmc_t *p, int c)
int pfm_write_pmds(int f, pfarg_pmd_t *p, int c)
int pfm_read_pmds(int f, pfarg_pmd_t *p, int c)
int pfm_load_context(int f, pfarg_load_t *l)
int pfm_start(int fd, pfarg_start_t *s)
int pfm_stop(int f)
int pfm_restart(int f)
int pfm_create_evtsets(int f, pfarg_setdesc_t *s, int c)
int pfm_getinfo_evtsets(int f, pfarg_setinfo_t *i, int c)
int pfm_delete_evtsets(int f, pfarg_setdesc_t *s, int c)
int pfm_unload_context(int f)

Table 1: perfmon2 system calls

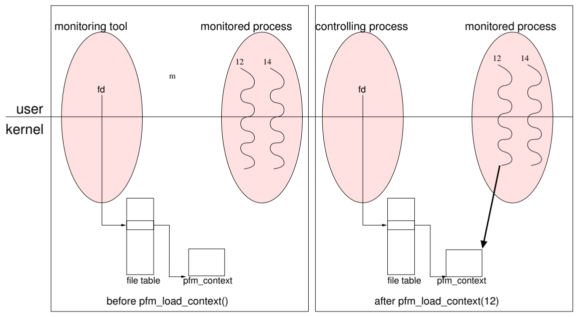

Figure 1: attaching to a thread

of context. Upon return from the call, the context is identified by a file descriptor which can then be used with the other system calls.

The write operations on the PMU registers are provided by the `pfm_write_pmcs` and `pfm_write_pmds` calls. It is possible to access more than one register per call by passing a variable-size array of structures. Each structure consists, at a minimum, of a register index and value plus some additional flags and bitmasks.

An array of structures is a good compromise between having a call per register, i.e., one register per structure per call, and passing the entire PMU state each time, i.e., one large structure per call for all registers. The cost of a system call is amortized, if necessary, by the fact that multiple registers are accessed, yet flexibility is not affected because the size of the array is variable. Furthermore, the register structure definition is generic and is used across all architectures.

The PMU can be entirely programmed before the context is attached to a thread or CPU. Tools can prepare a pool of contexts and later attach them on-the-fly to threads or CPUs.

To actually load the PMU state onto the actual hardware, the context must be bound to either a kernel thread or a CPU with the `pfm_load_context` call. Figure 1 shows the effect of the call when attaching to a thread of a dual-threaded process. A context can only be bound to one thread or CPU at a time. It is not possible to bind more than one context to a thread or CPU. Per-thread monitoring and system-wide monitoring are currently mutually exclusive. By construction, multiple concurrent per-thread contexts can co-exist. Potential conflicts are detected when the context is attached and not when it is created.

An attached context persists across a call to `exec`. On `fork` or `pthread_create`, the context is not automatically cloned in the new thread because it does not always make sense to aggregate results or profiles from *child* processes or threads. Monitoring tools can leverage the 2.6 kernel `ptrace` interface to receive notifications on the `clone` system call to decide whether or not to monitor a new thread or process. Because the context creation and attachment are two separate operations, it is possible to *batch* creations and simply attach and start on notification.

Once the context is attached, monitoring can be started and stopped using the `pfm_start` and `pfm_stop` calls. The values of the PMD registers can be extracted with the `pfm_read_pmds` call. A context can be detached with `pfm_unload_context`. Once detached the context can later be re-attached to any thread or CPU if necessary.

A context is destroyed using a simple `close` call. The other system calls listed in Table 1 relate to sampling or event sets and are discussed in later sections.

Many 64-bit processor architectures provide the ability to run with a *narrow* 32-bit instruction set. For instance, on Linux for x86_64, it is possible to run unmodified 32-bit i386 binaries. Even though, the PMU is very implementation specific, it may be interesting to develop/port tools in 32-bit mode. To avoid data conversions in the kernel, the perfmon2 ABI is designed to be portable between 32-bit (ILP32) and 64-bit (LP64) modes. In other words, all the data structures shared with the kernel use fixed-size data types.

4.2 System-wide monitoring

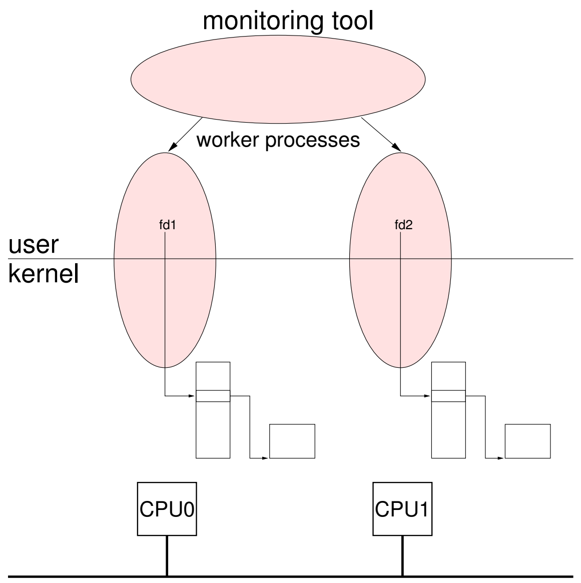

Figure 2: monitoring two CPUs

A perfmon context can be bound to only one CPU at a time. The CPU on which the call to `pfm_load_context` is executed determines the monitored CPU. It is necessary to set the affinity of the calling thread to ensure that it runs on the CPU to monitor. The affinity can later be modified, but all operations requiring access to the actual PMU must be executed on the monitored CPU, otherwise they will fail. In this setup, coverage of a multi-processor system (SMP), requires that multiple contexts be created and bound to each CPU to monitor. Figure 2 shows a possible setup for a monitoring tool on a 2-way system. Multiple non-overlapping system-wide attached context can co-exist.

The alternative design is to have the kernel propagate the PMU access to all CPUs of interest using Inter-Processor-Interrupt (IPI). Such approach does make sense if all CPUs are always monitored. This is the approach chosen by OProfile, for instance.

With the perfmon2 approach, it is possible to measure subsets of CPUs. This is very interesting for large NUMA-style or multi-core machines where all CPUs do not necessarily run the same workload. And even then, with a uniform workload, it possible to divide the CPUs into groups and capture different events in each group, thereby overlapping distinct measurements in one run. Aggregation of results can be done by monitoring tools, if necessary.

It is relatively straightforward to construct a user-level helper library that can simplify monitoring multiple CPUs from a single thread of control. Internally, the library can pin threads on the CPUs of interest. Synchronization between threads can easily be achieved using a barrier built with the POSIX-threads primitives. We have developed and released such a library as part of the `libpfm` [6] package.

Because PMU access requires the controlling thread to run on the monitored CPU, processor and memory affinity are inherently enforced thereby minimizing overhead which is important when sampling in NUMA machines. Furthermore, this design meshes well with certain PMU features such as the Precise-Event-Based

Sampling (PEBS) support of the Pentium 4 processor (see Section 5.4 for details).

4.3 Logical PMU

PMU register names and implementations are very diverse. On the Itanium processor architecture, they are implemented by actual PMC and PMD indirect registers. On the AMD Opteron [1] processors, they are called PERFSEL and PERFCTR indirect registers but are actually implemented by MSR registers. A portable tool would have to know about those names and the interface would have to change from one architecture to another to accommodate the names and types of the registers for the read and write operations. This would defeat our goal of having a uniform interface on all platforms.

To mask the diversity without compromising access to all PMU features, the interface exposes a *logical* PMU. This PMU is tailored to the underlying hardware PMU for properties such as the number of registers it implements. But it also guarantees the following properties across all architectures:

- the configuration registers are called PMC registers and are managed as 64-bit wide indirect registers
- the data registers are called PMD registers and are managed as 64-bit wide indirect registers
- counters are 64-bit wide unsigned integers

The mapping of PMC/PMD to actual PMU registers is defined by a PMU description table where each entry provides the default value, a bitmask of reserved fields, and the actual name of the register. The mapping is defined by the implementation and is accessible via a `sysfs` interface.

The routine to access the actual register is part of the architecture specific part of a perfmon2 implementation. For instance, on Itanium 2 processor, the mapping is defined such that the index in the table corresponds to the index of the actual PMU register, e.g., logical PMD0 corresponds to actual PMD0. The read function consists of a single `mov rXX=pmd[0]` instruction. On the Pentium M processor however, the mapping is defined as follows:

```
% cat /sys/kernel/perfmon/pmu_desc/mappings
PMC0:0x100000:0xffcfffff:PERFEVTSEL0
PMC1:0x100000:0xffcfffff:PERFEVTSEL1
PMD0:0x0:0xffffffffffffffff:PERFCTR0
PMD1:0x0:0xffffffffffffffff:PERFCTR1
```

When a tool writes to register `PMD0`, it writes to register `PERFEVTSEL0`. The actual register is implemented by MSR `0x186`. There is an architecture specific section of the PMU description table that provides the mapping to the MSR. The read function consist of a single `rdmsr` instruction.

On the Itanium 2 processors, we use this mapping mechanism to export the code (IBR) and data (DBR) debug registers as PMC registers because they can be used to restrict monitoring to a specific range of code or data respectively. There was no need to create an Itanium 2 processor specific system call in the interface to support this useful feature.

To make applications more portable, counters are always exposed as 64-bit wide unsigned integers. This is particularly interesting when sampling, see Section 5 for more details. Usually, PMUs implement narrower counters, e.g., 47 bits on Itanium 2 PMU, 40 bits on AMD Opteron PMU. If necessary, each implementation must emulate 64-bit counters. This can be accomplished fairly easily by leveraging the counter overflow interrupt capability present on all modern PMUs. Emulation can be turned off by applications on a per-counter basis, if necessary.

Oftentimes, it is interesting to associate PMU-based information with non-PMU based information such as an operating system resource or other hardware resource. For instance, one may want to include the time since monitoring has been started, the number of active networks connections, or the identification of the current process in a sample. The perfctr interface provides this kind of information, e.g., the virtual cycle counter, through a kernel data structure that is re-mapped to user level.

With perfmon2, it is possible to leverage the mapping table to define *Virtual PMD* registers, i.e., registers that do not map to actual PMU or PMU-related registers. This mechanism provides a uniform and extensible naming and access interface for those resources. Access to new resources can be added without breaking the ABI. When a tool invokes `pfm_read_pmds` on a virtual PMD register, a read call-back function, provided by the PMU description table, is invoked and returns a 64-bit value for the resource.

4.4 PMU description module

Hardware and software release cycles do not always align correctly. Although Linux kernel patches are produced daily on the `kernel.org` web site, most end-users really run packaged distributions which have a very different development cycle. Thus, new hardware may become available before there is an actual Linux distribution ready. Similarly, processors may be revised and new steppings may fix bugs in the PMU. Although providing updates is fairly easy nowadays, end-users tend to be reluctant to patch and recompile their own kernels.

It is important to understand that monitoring tool developers are not necessarily kernel developers. As such, it is important to provide simple mechanisms whereby they can enable early access to new hardware, add virtual PMD registers and run experimentations without full kernel patching and recompiling.

There are no technical reasons for having the PMU description tables built into the kernel. With a minimal framework, they can as well be implemented by kernel modules where they become easier to maintain. The perfmon2 interface provides a framework where a PMU description module can be dynamically inserted into the kernel at runtime. Only one module can be inserted at a time. When new hardware becomes available, assuming there is no changes needed in the architecture specific implementation, a new description module can be provided quickly. Similarly, it becomes easy to experiment with virtual PMD registers by modifying the description table and not the interface nor the core implementation.

5 Sampling Support

Statistical Sampling or *profiling* is the act of recording information about the execution of a program at some interval. The interval is commonly expressed in units of time, e.g., every 20ms. This is called Time-Based sampling (TBS). But the interval can also be expressed in terms of a number of occurrences of a PMU event, e.g., every 2000 L2 cache misses. This is called Event-Based sampling (EBS). TBS can easily be emulated with EBS by using an event with a fixed correlation to time, e.g., the number of elapsed cycles. Such emulation typically provides a much finer granularity than the operating system timer which is usually limited to millisecond at best. The interval, regardless of its unit, does not have to be constant.

At the end of an interval, the information is stored into a *sample* which may contain information as simple as where the thread was, i.e., the instruction pointer. It may also include values of some PMU registers or other hardware or software resources.

The quality of a profile depends mostly on the duration of the run and the number of samples collected. A good profile can provide a lot of useful information about the behavior of a program, in particular it can help identify bottlenecks. The difficulty is to manage to overhead involved with sampling. It is important to make sure that sampling does not perturb the execution of the monitored program such that it does not exhibit its normal behavior. As the sampling interval decreases, overhead increases.

The perfmon2 interface has an extensive set of features to support sampling. It is possible to manage sampling completely at the user level. But there is also kernel-level support to minimize the overhead. The interface provides support for EBS.

5.1 Sampling periods

All modern PMUs implement a counter overflow interrupt mechanism where the processor generates an interrupt whenever a counter wraps around to zero. Using this mechanism and supposing a 64-bit wide counter, it is possible to implement EBS by expressing a sampling period p as $2^{64} - p$ or in two's complement arithmetics as $-p$. After p occurrences, the counter overflows, an interrupt is generated indicating that a sample must be recorded.

Because all counters are 64-bit unsigned integers, tools do not have to worry about the actual width of counters when setting the period. When 64-bit emulation is needed, the implementation maintains a 64-bit software

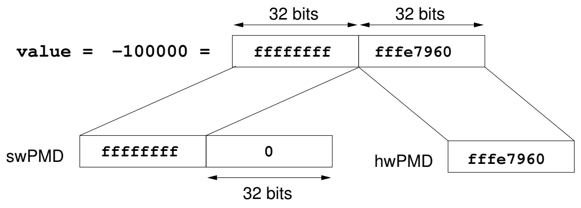

Figure 3: 64-bit counter emulation

value and loads only the low-order bits onto the actual register as shown in Figure 3. An EBS overflow is declared only when the 64-bit software-maintained value overflows.

The interface does not have the notion of a sampling period, all it knows about is PMD values. Thus a sampling period p, is programmed into a PMD by setting its value to $-p$. The number of sampling periods is only limited by the number of counters. Thus, it is possible to overlap sampling measurements to collect multiple profiles in one run.

For each counter, the interface provides three values which are used as follows:

- *value*: the value loaded into the PMD register when the context is attached. This is the initial value.
- *long_reset*: the value to reload into the PMD register after an overflow with user-level notification.
- *short_reset*: the value to reload into the PMD register after an overflow with no user-level notification.

The three values can be used to try and mask some of the overhead involved with sampling. The initial period would typically be large because it is not always interesting to capture samples in initialization code. The long and short reset values can be used to mask the *noise* generated by the PMU interrupt handler. We explain how they are used in Section 5.3.

5.2 Overflow notifications

To support sampling at the user level, it is necessary to inform the tool when a 64-bit overflow occurs. The notification can be requested per counter and is sent as a message. There is only one notification per interrupt even when multiple counters overflow at the same time.

Each perfmon context has a fixed-depth message queue. The fixed-size message contains information about the overflow such as which counter(s) overflowed, the instruction pointer, the current CPU at the time of the overflow. Each new message is appended to the queue which is managed as a FIFO.

Instead of re-inventing yet another notification mechanism, existing kernel interfaces are leveraged and messages are extracted using a simple `read` call on the file descriptor of the context. The benefit is that common interfaces such as `select` or `poll` can be used to wait on multiple contexts at the same time. Similarly, asynchronous notifications via `SIGIO` are also supported.

Regular file descriptor sharing semantic applies, thus it is possible to delegate notification processing to a specific thread or child process.

During a notification, monitoring is stopped. When monitoring another thread, it is possible to request that this thread be blocked while the notification is being processed. A tool may choose the block on notification option when the context is created. Depending on the type of sampling, it may be interesting to have the thread run just to keep the caches and TLB warm, for instance.

Once a notification is processed, the `pfm_restart` function is invoked. It is used to reset the overflowed counters using their *long* reset value, to resume monitoring, and potentially to unblock the monitored thread.

5.3 Kernel sampling buffer

It is quite expensive to send a notification to user level for each sample. This is particularly bad when monitoring another thread because there could be, at least, two context switches per overflow and a couple of system calls.

One way to minimize this cost, it is to *amortize* it over a large set of samples. The idea is to have the kernel directly record samples into a buffer. It is not possible to take page faults from the PMU interrupt handler, usually a high priority handler. As such the memory would have to be locked, an operation that is typically restricted to privileged users. As indicated earlier, sampling must be available to regular users, thus, the buffer is allocated by the kernel and marked as reserved to avoid being paged out.

When the buffer becomes full, the monitoring tool is notified. A similar approach is used by the `OProfile` and `VTUNE` interfaces. Several issues must be solved for the buffer to become useable:

- how to make the kernel buffer accessible to the user?
- how to reset the PMD values after an overflow when the monitoring tool is not involved?
- what format for the buffer?

The buffer can be made available via a `read` call. This is how `OProfile` and `VTUNE` work. Perfmon2 uses a different approach to try and minimize overhead. The buffer is re-mapped read-only into the user address space of the monitoring tool with a call to `mmap`, as shown in Figure 4. The content of the buffer is guaranteed consistent when a notification is received.

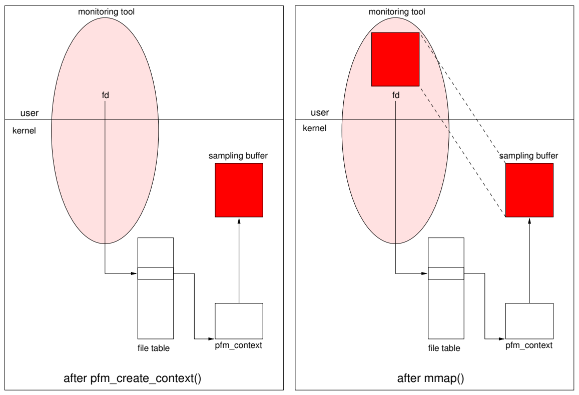

Figure 4: re-mapping the sampling buffer

On counter overflow, the kernel needs to know what value to reload into an overflowed PMD register. This information is passed, per register, during the `pfm_write_pmd` call. If the buffer does not become full, the kernel uses the *short* reset value to reload the counter.

When the buffer becomes full, monitoring is stopped and a notification is sent. Reset is deferred until the monitoring tool invokes `pfm_restart`, at which point, the buffer is marked as empty, the overflowed counter is reset with the *long* reset value and monitoring resumes.

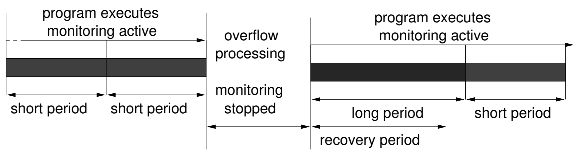

Figure 5: *short* vs. *long* reset values.

The distinction between *long* and *short* reset values allows tools to specify a different, potentially larger value, for the first period after an overflow notification. It is very likely that the user-level notification and subsequent processing will modify the CPU state, e.g., caches and TLB, such that when monitoring resumes, the execution will enter a *recovery* phase where its behavior may be different from what it would have been without monitoring. Depending on the type of sampling, the *long* vs. *short* reset values can be leveraged to hide that recovery period. This is demonstrated in Figure 5 which shows where the *long* reset value is used after overflow processing is completed. Of course, the impact and duration of the recovery period is very specific to each workload and CPU.

It is possible to request, per counter, that both reset values be randomized. This is very useful to avoid biased samples for certain measurements. The pseudo-random number generator does not need to be very fancy, simple variation are good enough. The randomization is specified by a seed value and a bitmask to limit the range of variation. For instance, a mask of `0xff` allows a variation in the interval `[0-255]` from the base value. The existing implementation uses the Carta [2] pseudo-random number generator because it is simple and very efficient.

A monitoring tool may want to record the values of certain PMD registers in each sample. Similarly, after each sample, a tool may want to reset certain PMD registers. This could be used to compute event deltas, for instance. Each PMD register has two bitmasks to convey this information to the kernel. Each bit in the bitmask represents a PMD register, e.g., bit 1 represents PMD1. Let us suppose that on overflow of PMD4, a tool needs to record PMD6 and PMD7 and then reset PMD7. In that case, the tool would initialize the sampling bitmask of PMD4 to `0xc0` and the reset bitmask to `0x80`.

With a kernel-level sampling buffer, the format in which samples are stored and what gets recorded becomes somehow fixed and it is more difficult to evolve. Monitoring tools can have very diverse needs. Some tools may want to store samples sequentially into the buffer, some may want to aggregate them immediately, others may want to record non PMU-based information, e.g., the kernel call stack.

As indicated earlier, it is important to ensure that existing interfaces such as `OProfile` or `VTUNE`, both using their own buffer formats, can be ported without having to modify a lot of their code. Similarly, It is important to ensure the interface can take advantage of advanced PMU support sampling such as the PEBS feature of the Pentium 4 processor.

Preserving a high level of flexibility for the buffer, while having it fully specified into the interface did not look very realistic. We realized that it would be very difficult to come up with a *universal* format that would satisfy all needs. Instead, the interface uses a radically different approach which is described in the next section.

5.4 Custom Sampling Buffer Formats

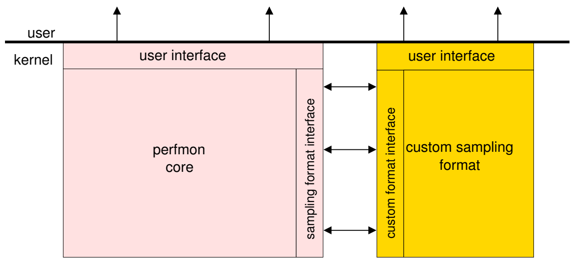

Figure 6: Custom sampling format architecture

The interface introduces a new flexible mechanism called *Custom Sampling Buffer Formats*, or formats for short. The idea is to remove the buffer format from the interface and instead provide a framework for extending the interface via specific sampling formats implemented by kernel modules. The architecture is shown in Figure 6.

Each format is uniquely identified by a 128-bit Universal Unique IDentifier (UUID) which can be generated by commands such as `uuidgen`. In order to use a format, a tool must pass this UUID when the context is created. It is possible to pass arguments, such as the buffer size, to a format when a context is created.

When the format module is inserted into the kernel, it registers with the perfmon core via a dedicated interface. Multiple formats can be registered. The list of available formats is accessible via a `sysfs` interface. Formats can also be dynamically removed like any other kernel module.

Each format provides a set of call-backs functions invoked by the perfmon core during certain operations. To make developing a format fairly easy, the perfmon core provides certain basic services such as memory allocation and the ability to re-map the buffer, if needed. Formats are not required to use those services. They may, instead, allocate their own buffer and expose it using a different interface, such as a driver interface.

At a minimum, a format must provide a call-back function invoked on 64-bit counter overflow, i.e., an interrupt handler. That handler does not bypass the core PMU interrupt handler which controls 64-bit counter emulation, overflow detection, notification, and monitoring masking. This layering make it very simple to write a handler. Each format controls:

- how samples are stored
- what gets recorded on overflow
- how the samples are exported to user-level
- when an overflow notification must be sent
- whether or not to reset counters after an overflow
- whether or not to mask monitoring after an overflow

The interface specifies a simple and relatively generic default sampling format that is built-in on all architectures. It stores samples sequentially in the buffer. Each sample has a fixed-size header containing information such as the instruction pointer at the time of the overflow, the process identification. It is followed by a variable-size body containing 64-bit PMD values stored in increasing index order. Those PMD values correspond to the information provided in the sampling bitmask of the overflowed PMD register. Buffer space is managed such that there can never be a partial sample. If multiple counters overflow at the same time, multiple contiguous samples are written.

Using the flexibility of formats, it was fairly easy to port the `OProfile` kernel code over to perfmon2. An new format was created to connect the perfmon2 PMU and `OProfile` interrupt handlers. The user-level `OProfile`, `opcontrol` tool was migrated over to use the perfmon2 interface to program the PMU. The resulting format is about 30 lines of C code. The OProfile buffer format and management kernel code was totally preserved.

Other formats have been developed since then. In particular we have released a format that implements *n*-way buffering. In this format, the buffer space is split into equal-size regions. Samples are stored in one region, when it fills up, the tool is notified but monitoring remains active and samples are stored in the next region. This idea is to limit the number of blind spots by never stopping monitoring on counter overflow.

The format mechanism proved particularly useful to implement support for the Pentium 4 processor Precise Event-Based Sampling (PEBS) feature where the CPU is directly writing samples to a designated region of memory. By having the CPU write the samples, the skew observed on the instruction pointer with typical interrupt-based sampling can be avoided, thus a much improved *precision* of the samples. That skew comes from the fact that the PMU interrupt is not generated exactly on the instruction where the counter overflowed. The phenomenon is especially important on deeply-pipelined processor implementations, such as Pentium 4 processor. With PEBS, there is a PMU interrupt when the memory region given to the CPU fills up.

The problem with PEBS is that the sample format is now fixed by the CPU and it cannot be changed. Furthermore, the format is different between the 32-bit and 64-bit implementations of the CPU. By leveraging the format infrastructure, we created two new formats, one for 32-bit and one for 64-bit PEBS with less than one hundred lines of C code each. Perfmon2 is the first to provide support for PEBS and it required no changes to the interface.

6 Event sets and multiplexing

On many PMU models, the number of counters is fairly limited yet certain measurements require lots of events. For instance, on the Itanium 2 processor, it takes about a dozen events to gather a cycle breakdown, showing how each CPU cycle is spent, yet there are only 4 counters. Thus, it is necessary to run the workload under test multiple times. This is not always very convenient as workloads sometimes cannot be stopped or are long to restart. Furthermore, this inevitably introduces fluctuations in the collected counts which may affect the accuracy of the results.

Even with a large number of counters, e.g., 18 for the Pentium 4 processor, there are still hardware constraints which make it difficult to collect some measurements in one run. For instance, it is fairly common to have constraints such as:

- event A and B cannot be measured together
- event A can only be measured on counter C.

Those constraints are unlikely to go away in the future because that could impact the performance of CPUs. An elegant solution to these problems is to introduce the notion of *events sets* where each set encapsulates the full PMU machine state. Multiple sets can be defined and they are multiplexed on the actual PMU hardware such that only one set if active at a time. At the end of the multiplexed run, the counts are scaled to compute an *estimate* of what they would have been, had they been collected for the entire duration of the measurement.

The accuracy of the scaled counts depends a lot of the switch frequency and the workload, the goal being to avoid blind spots where certain events are not visible because the set that measures them did not activate at the right time. The key point is to balance to the need for high switch frequency with higher overhead.

Sets and multiplexing can be implemented totally at the user level and this is done by the PAPI toolkit, for instance. However, it is critical to minimize the overhead especially for non self-monitoring measurements where it is extremely expensive to switch because it could incur, at least, two context switches and a bunch of system calls to save the current PMD values, reprogram the new PMC and PMD registers. During that window of time the monitored thread usually keeps on running opening up a large blind spot.

The perfmon2 interface supports events sets and multiplexing at the kernel level. Switching overhead is significantly minimized, blind spots are eliminated by the fact that switching systematically occurs in the context of the monitored thread.

Sets and multiplexing is supported for per-thread and system-wide monitoring and for both counting and sampling measurements.

6.1 Defining sets

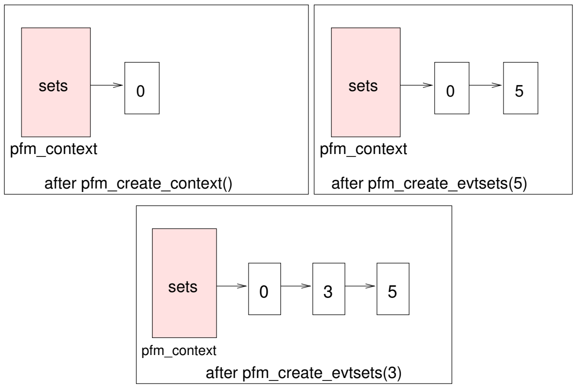
sets
0
pfm_context
after pfm_create_context()
sets
0
5
pfm_context
after pfm_create_evtsets(5)
sets
0
3
5
pfm_context
after pfm_create_evtsets(3)

Figure 7: creating sets.

Each context is created with a default event set, called *set0*. Sets can be dynamically created, modified, or deleted when the context is detached using the `pfm_create_evtsets` and `pfm_delete_evtsets` calls. Information, such as the number of activations of a set, can be retrieved with the `pfm_getinfo_evtsets` call. All these functions take array arguments and can, therefore, manipulate multiple sets per call.

A set is identified with a 16-bit number. As such, there is a theoretical limit of 65k sets. Sets are managed through an ordered list based on their identification numbers. Figure 7 shows the effect of adding *set5* and *set3* on the list.

Tools can program registers in each set by passing the set identification for each element of the array passed to the read or write calls. In one `pfm_write_pmcs` call it is possible to program registers for multiple sets.

6.2 Set switching

Set switching can be triggered by two different events: a timeout or a counter overflow. This is another innovation of the perfmon2 interface, again giving tools maximum flexibility. The type of *trigger* is determined, per set, when it is created.

The timeout is specified in micro-seconds when the set is created. The granularity of the timeout depends on the granularity of kernel internal timer tick, usually 1ms or 10ms. If the granularity is 10ms, then it is not possible to switch more than 100 times per second, i.e., the timeout cannot be smaller than 100μs. Because the granularity can greatly affect the accuracy of a measurement, the actual timeout, rounded up the the closest multiple of the timer tick, is returned by the `pfm_create_evtsets` call.

It is also possible to trigger a switch on counter overflow. To avoid dedicating a counter as a trigger, there is a trigger threshold value associated with each counter. At each overflow, the threshold value is decremented, when it reaches zero, switching occurs. It is possible to have multiple trigger counters per set, i.e., switch on multiple conditions.

The next set is determined by the position in the ordered list of sets. Switching is managed in a round-robin fashion. In the example from Figure 7, this means that the set following *set5* is *set0*.

Using overflow switching, it is possible to implement *counter cascading* where a counter starts counting only when a certain number of occurrences, n, of an event E is reached. In a first set, a PMC register is programmed to measure event E, the corresponding PMD register is initialized to $-n$, and its switch trigger is set to 1. The next set is setup to count the event of interest and it will activated only when there is an overflow in the first set.

6.3 Sampling

Sets are fully integrated with sampling. Set information is propagated wherever is necessary. The counter overflow notification carries the identification of the active set. The default sampling format fully supports sets. Samples from all sets are stored them into the same buffer. The active set at the time of the overflow is identified in the header of each sample.

7 Security

The interface is designed to be built into the base kernel, as such, it must follow the same security guidelines.

It is not possible to assume that tools will always be well-behaved. Each implementation must check arguments to calls. It must not be possible to use the interface for malicious attacks. A user cannot run a monitoring tool to extract information about a process or the system without proper permission.

All vector arguments have a maximum size to limit the amount of kernel memory necessary to perform the copy into kernel space. By nature, those calls are non-blocking and non-preemptible, ensuring that memory is eventually freed. The default limit is set to a page.

The sampling buffer size is also limited because it consumes kernel memory that cannot be paged out. There is a system-wide limit and a per-process limit. The latter is using the resource limit on locked memory (`RLIMIT_MEMLOCK`). The two-level protection is required to prevent users from launching lots of processes each allocating a small buffer.

In per-thread mode, the user credentials are checked against the permission of the thread to monitor when the context is attached. Typically, if a user cannot send a signal to the process, it is not possible to attach. By default, per-thread monitoring is available to all users, but a system administrator can limit to a user group. An identical, but separate, restriction is available for system-wide contexts.

On several architectures, such as Itanium, it is possible to read the PMD registers directly from user-level, i.e., with a simple instruction. There is always a provision to turn this feature off. The interface supports this mode of access by default for all self-monitoring per-thread context. It is turned off by default for all other configurations, thereby preventing spy applications from peeking at values left in PMD registers by others.

All size and user group limitations can be configured by a system administrator via a simple `sysfs` interface.

As for sampling, we are planning on adding a PMU interrupt throttling mechanism to prevent Denial-of-Service (DoS) attacks when applications set very high sampling rates.

8 Fast user-level PMD read

Invoking a system call to read a PMD register can be quite expensive compared to the cost of the actual instruction. On an Itanium 2 1.5GHz processor, for instance, it costs about 36 cycles to read a PMD with a single instruction and about 750 cycles via `pfm_read_pmds` which is not really optimized at this point. As a reference, the simplest system call, i.e., `getpid`, costs about 210 cycles.

On many PMU models, it is possible to directly read a PMD register from user level with a single instruction. This very lightweight mode of access is allowed by the interface for all self-monitoring threads. Yet, if actual counters width is less than 64-bit, only the partial value is returned. The software-maintained value requires a kernel call.

To enable fast 64-bit PMD read accesses from user level, the interface supports re-mapping of the software-maintained PMD values to user level for self-monitoring threads. This mechanism was introduced by the perfctr interface. This enables fast access on architectures without hardware support for direct access. For the others, this enables a full 64-bit value to be reconstructed by merging the high-order bits from the re-mapped PMD with the low-order bit obtained from hardware.

Re-mapping has to be requested when the context is created. For each event set, the PMD register values have to be explicitly re-mapped via a call to `mmap` on the file descriptor identifying the context. When a set is created a special *cookie* value is passed back by `pfm_create_evtset`. It is used as an offset for `mmap` and is required to identify the set to map. The mapping is limited to one page per set. For each set, the re-mapped region contains the 64-bit software value of each PMD register along with a status bit indicating whether the set is the active set or not. For non-active sets, the re-mapped value is the up-to-date full 64-bit value.

Given that the merge of the software and hardware values is not atomic, there can be a race condition if, for instance, the thread is preempted in the middle of building the 64-bit value. There is no way to avoid the race, instead the interface provides an atomic sequence number for each set. The number is updated each time the state of the set is modified. The number must be read by user-level code before and after reading the re-mapped PMD value. If the number is the same before and after, it means that the PMD value is current, otherwise the operation must be restarted. On the same Itanium 2 processor and without conflict, the cost is about 55 cycles to read the 64-bit value of a PMD register.

9 Status

A first generation of this interface has been implemented for the 2.6 kernel series for the Itanium Processor Family (IPF). It uses a single multiplexing system call, `perfmonctl`, and is missing events sets, PMU description tables, and fast user-level PMD reads. It is currently shipping with all major Linux distributions for this architecture.

The second generation interface, which we describe in this paper, currently exists as a kernel patch against the latest official 2.6 kernel from `kernel.org`. It supports the following processor architectures and/or models:

- all the Itanium processors
- the AMD Opteron processors in 64-bit mode
- the Intel Pentium M and P6 processors
- the Intel Pentium 4 and Xeon processors. That includes 32-bit and 64-bit (EM64T) processors. Hyper-Threading and PEBS are supported.
- the MIPS 5k and MIPS 20k processors
- preliminary support for IBM Power5 processor

Certain ports were contributed by other companies or developers. As our user community grows, we expect other contributions to both kernel and user-level code. The kernel patch has been widely distributed and has generated a lot of discussions on various Linux mailing lists.

Our goal is to establish perfmon2 as the standard Linux interface for hardware-based performance monitoring. We are in the process of getting it reviewed by the Community in preparation for a merge with the mainline kernel.

10 Existing tools

Several tools already exists for the interface. Most of them are only available for Itanium processors at this point, because an implementation exists since several years.

The first open-source tool to use the interface is *pfmon* [6] from HP Labs. This is a command-line oriented tool initially built to test the interface. It can collect counts and profiles on a per-thread or system-wide basis. It supports the Itanium, AMD Opteron, and Intel Pentium M processors. It is built on top of a helper library, called `libpfm`, which handles all the event encodings and assignment logic.

HP Labs also developed `q-tools` [7], a replacement program for `gprof`. Q-tools uses the interface to collect a flat profile and a statistical call graph of all processes running in a system. Unlike `gprof`, there is no need to recompile applications or the kernel. The profile and call graph include both user- and kernel-level execution. The tool only works on Itanium 2 processors because it leverages certain PMU features, in particular the Branch Trace Buffer. This tool takes advantage of the interface by overlapping two sampling measurements to collect the flat profile and call graph in one run.

The HP Caliper [5] is an official HP product which is free for non-commercial use. This is a professional tool which works with all major Linux distributions for Itanium processors. It collects counts or profiles on a per-thread or per-CPU basis. It is very simple to use and comes with a large choice of preset metrics such as flat profile (`fprof`), data cache misses (`dcache_miss`). It exploits all the advanced PMU features, such as the Branch Trace Buffer (BTB) and the Data Event Address Registers (D-EAR). The profiles are correlated to source and assembly code.

The PAPI toolkit has long been available on top of the perfmon2 interface for Itanium processors. We expect that PAPI will migrate over to perfmon2 on other architectures as well. This migration will likely simplify the code and allow better support for sampling and set multiplexing.

The BEA JRockit JVM on Linux/ia64, starting with version 1.4.2 is also exploiting the interface. The JIT compiler is using a, dynamically collected, per-thread profile to improve code generation. This technique [4], called Dynamic Profile Guided Optimization (DPGO), takes advantage of the efficient per-thread sampling support of the interface and of the ability of the Itanium 2 PMU to sample branches and locations of cache misses (Data Event Address Registers). What is particularly interesting about this example is that it introduces a new usage model. Monitoring is used each time a program runs and not just during the development phase. Optimizations are applied in the end-user environment and for the real workload.

11 Conclusion

We have designed the most advanced performance monitoring interface for Linux. It provides a uniform set of functionalities across all architectures making it easier to write portable performance tools. The feature set was carefully designed to allow efficient monitoring and a very high degree of flexibility to support a diversity of usage models and hardware architectures. The interface provides several key features such as custom sampling buffer formats, kernel support event sets multiplexing, and PMU description modules.

We have developed a multi-architecture implementation of this interface that support all major processors. On the Intel Pentium 4 processor, this implementation is the first to offer support for PEBS.

We are in the process of getting it merged into the mainline kernel. Several open-source and commercial tools are available on Itanium 2 processors at this point and we expect that others will be released for the other architectures as well.

Hardware-based performance monitoring is the key tool to understand how applications and operating systems behave. The monitoring information is used to drive performance improvements in applications, operating system kernels, compilers, and hardware. As processor implementation enhancements shift from pure clock speed to multi-core, multi-thread, the need for powerful monitoring will increase significantly. The perfmon2 interface is well suited to address those needs.

References

[1] AMD. *AMD64 Architecture Programmer's Manual: System Programming*, 2005. `http://www.amd.com/us-en/Processors/DevelopWithAMD`.

[2] David F. Carta. Two fast implementations of the minimal standard random number

generator. *Com. of the ACM*, 33(1):87–88, 1990. http://doi.acm.org/10.1145/76372.76379.

[3] Intel Coporation. The Itanium processor family architecture. http://developer.intel.com/design/itanium2/documentation.htm.

[4] Greg Eastman, Shirish Aundhe, Robert Knight, and Robert Kasten. Intel dynamic profile-guided optimization in the BEA JRockit^{TM} JVM. In *3rd Workshop on Managed Runtime Environments, MRE'05*, 2005. http://www.research.ibm.com/mre05/program.html.

[5] Hewlett-Packard Company. The Caliper performance analyzer. http://www.hp.com/go/caliper.

[6] Hewlett-Packard Laboratories. The pfmon tool and the libpfm library. http://perfmon2.sf.net/.

[7] Hewlett-Packard Laboratories. q-tools, and q-prof tools. http://www.hpl.hp.com/research/linux.

[8] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, April 2003. http://www.intel.com/design/itanium/documentation.htm.

[9] Intel. *IA-32 Intel Architecture Software Developers' Manual: System Programming Guide*, 2004. http://developer.intel.com/design/pentium4/manuals/index_new.htm.

[10] Intel Corp. The VTune^{TM} performance analyzer. http://www.intel.com/software/products/vtune/.

[11] Chi-Keung Luk and Robert Muth *et al.* Ispike: A post-link optimizer for the Intel Itanium architecture. In *Code Generation and Optimization Conference 2004 (CGO 2004)*, March 2004. http://www.cgo.org/cgo2004/papers/01_82_luk_ck.pdf.

[12] Mikael Pettersson. the Perfctr interface. http://user.it.uu.se/~mikpe/linux/perfctr/.

[13] Alex Shye *et al.* Analysis of path profiling information generated with performance monitoring hardware. In *INTERACT HPCA'04 workshop*, 2004. http://rogue.colorado.edu/draco/papers/interact05-pmu_pathprof.pdf.

[14] B.D̃ragovic *et al.* Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/architecture.html.

[15] J.Ãnderson *et al.* Continuous profiling: Where have all the cycles gone?, 1997. http://citeseer.ist.psu.edu/article/anderson97continuous.html.

[16] John Levon *et al.* Oprofile. http://oprofile.sf.net/.

[17] Robert Cohn *et al.* The PIN tool. http://rogue.colorado.edu/Pin/.

[18] Alex Tsariounov. The Prospect monitoring tool. http://prospect.sf.net/.

[19] University of Tenessee, Knoxville. Performance Application Programming Interface (PAPI) project. http://icl.cs.utk.edu/papi.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada