Paper Appeared in the 2013 FastPath Workshop

Linux perf_event Features and Overhead

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

***Abstract*—Most modern CPUs include hardware performance counters: architectural registers that allow programmers to gain low-level insight into system performance. In 2009 the perf_event subsystem was added to the Linux kernel, allowing users to access the performance counters easily, without kernel patches or recompiles. Performance counter access is now readily available for all Linux users, from Top500 supercomputers down to small embedded systems. Tools such as PAPI can use the perf_event interface to provide CPU event support while also providing information on other system components such as GPUs, networking, the operating system, energy, power, and more. We investigate the features and limitations provided by perf_event and compare its overhead against previous Linux performance counter implementations.**

I. Introduction

With processor speeds holding constant, we cannot depend on Moore's Law to deliver increased performance. Workload optimized systems have the advantage of having only a single workload of interest, allowing cross-stack hardware, kernel, and application optimization. Code analysis and optimization can provide targeted speedups on such code stacks when the details of the underlying architecture are exposed; hardware performance counters are the perfect tool for conducting this type of optimization.

Most moderns CPUs contain hardware performance counters: architectural registers that allow low-level analysis of running programs. Performance counter usage has traditionally been most pervasive in the High Performance Computing (HPC) field. The Linux operating system has come to dominate this arena; in the November 2012 TOP500 Supercomputer list 93% of the machines were running some form of Linux [1]. Despite Linux's importance to HPC, performance counter support has been lacking. There have been various counter implementations over the years: all were developed outside of the main Linux codebase and required custom-patching the kernel source code.

The relatively recent (in 2009) inclusion of the perf_event subsystem has introduced performance counter access for a wide range of computers (from embedded through supercomputers) that previously had no default operating system support. This eases development of tools that allow advanced workload optimization.

II. Background

There are a wide variety of events that can be measured with performance counters; event availability varies considerably among CPUs and vendors. Some processors provide hundreds of events; sorting which ones are useful and accurate from those that are broken and/or measure esoteric architectural minutia can be difficult. Event details are buried in architectural manuals, often accompanied by disclaimers disavowing any guarantees for useful results.

Access to hardware performance counters requires directly accessing special hardware registers (in the x86 world these are known as Model Specific Registers, or MSRs). There are usually two types of registers: *configuration registers* (which allow starting and stopping the counters, choosing the events to monitor, and setting up overflow interrupts) and *counting registers* (which hold the current event counts). In general between 2 - 8 counting registers are available, although machines with more are possible. Reads and writes to the configuration registers usually require special privileged (ring 0 or supervisor) instructions; the operating system can allow access by providing an interface that translates the users' desires into the proper low-level CPU calls. Access to counting registers can also require special permissions; some processors provide special instructions that allow users to access the values directly (`rdpmc` on x86).

A typical operating system performance counter interface allows selecting which events are being monitored, starting and stopping counters, reading counter values, and, if the CPU supports notification on counter overflow, some mechanism for passing overflow information to the user. Some operating systems provide additional features, such as *event scheduling* (addressing limitations on which events can go into which counters), *multiplexing* (swapping events in and out while extrapolating counts using time accounting to give the appearance of more physical counters), *per-thread counting* (by loading and saving counter values at context switch time), *process attaching* (obtaining counts from an already running process), and *per-cpu counting*.

High-level libraries and tools provide common cross-architecture and cross-platform interfaces to performance counters; one example is PAPI [2] which is widely used in the HPC community. PAPI in turn can be used by other tools that provide graphical frontends to the underlying performance counter infrastructure. Analysis works best if backed by an operating system that can provide efficient low-overhead access to the counters.

A. Performance Counters and Linux

Patches providing Linux support for performance counters appeared soon after the release of the original Pentium processor. While there were many attempts at Linux performance counter interfaces, only the following saw widespread use.

1) Oprofile: Oprofile [3] is a system-wide sampling profiler by Levon which was included into Linux 2.5.43 in 2002. It allows sampling with arbitrary performance events (or a timer if you lack performance counters) and provides frequency

graphs, profiles, and stack traces. Oprofile has limitations that make it unsuitable for general analysis: it is system-wide only, it requires starting a daemon as root, and it is a sampled interface so cannot easily provide aggregate event counts.

2) Perfctr: *Perfctr* [4] is a widely-used performance counter interface introduced by Pettersson in 1999. The kernel interface involves opening a `/dev/perfctr` device and accessing it with various `ioctl()` calls. Fast reads of counter values are supported on x86 without requiring a system call using `rdpmc` in conjunction with `mmap()`. A libperfctr is provided which abstracts the kernel interface.

3) Perfmon2: Eranian developed *Perfmon2* [5], an extension of the original itanium-specific *Perfmon*. The perfmon2 interface adds a variety of system calls with some additional system-wide configuration done via the `/sys` pseudo-filesystem. Abstract PMC (config) and PMD (data) structures provide a thin layer over the raw hardware counters. The libpfm3 library provides a high-level interface, providing event name tables and code that schedules events (to avoid counter conflicts). These tasks are done in userspace (in contrast to perf_event which does this in the kernel).

4) Post perf_event Implementations: perf_event's greatest advantage is that it is included in the Linux kernel; this is a huge barrier to entry for all competing implementations. Competitors must show compelling advantages before a user will bother taking the trouble to install something else. Despite this, various new implementations have been proposed.

LIKWID [6] is a method of accessing performance counters on Linux that completely bypasses the Linux kernel by accessing MSRs directly. This can have low overhead but can conflict with concurrent use of perf_event. It is limited to x86 processors, and only enables system-wide measurement (since MSR values cannot be saved at context-switch time).

LiMiT [7] is an interface similar to the existing perfctr infrastructure. They find up to 23 times speedup versus perf_event when instrumenting locks. Their methodology requires modifying the kernel and is x86 only.

III. The perf_event Interface

The *perf_event* subsystem was created in 2009 by Molnar and Gleixner in response to a proposed merge of perfmon2. perf_event entered Linux 2.6.31 as "Performance Counters for Linux" and was subsequently renamed perf_event in the 2.6.32 release. The interface is built around file descriptors allocated with the new `sys_perf_event_open()` system call. Events are specified at open time in an elaborate `perf_event_attr` structure; this structure has over 40 different fields that interact in complex ways. Counters are enabled and disabled via calls to `ioctl()` or `prctl()` and values are read via the standard `read()` system call. Sampling can be enabled to periodically read counters and write the results to a buffer which can be accessed via `mmap()`; signals are sent when new data are available.

The primary design philosophy is to provide as much functionality and abstraction as possible in the kernel, making the interface straightforward for ordinary users. The everything-in-the-kernel theme includes the flagship `perf` analysis tool: its source is bundled with the main Linux kernel source tree. What follows is a list of perf_event features.

A. Generalized Events

perf_event provides in-kernel *generalized events*: a common subset of useful events that are available on most modern CPUs (such as cycles, retired instructions, cache misses, etc.) Creating general events is fraught with peril: events are not documented well by vendors and usually require validation [8]. It is relatively easy to update tables in user code (a user can even replace a library in their own home directory without system administrator intervention) but a buggy kernel event table requires a new kernel and a reboot, which might take months or years in a production environment.

On AMD processors the "branches" generalized event mistakenly mapped to "taken branches" rather than "total branches" for over a year without anyone noticing. (On ARM Cortex A9 "taken branches" is used in place of "branches" *intentionally*, as no suitable all-encompassing branch event exists). We generate phase plots for SPEC CPU 2000 [9] to see if the wrong event definitions could lead to wrong conclusions. Figure 1 shows total versus taken branches for selected benchmarks. We compile using 64-bit gcc-4.3 using the `-O3 -msse3` options. The phase plot data was gathered using the `task_smpl` example from libpfm4 that allows recording event totals at fixed (in this case 100 million instruction) intervals. For `gcc` and `gzip` the behavior is complicated; sometimes peaks in total are not matched by a corresponding peak in taken, and in `gcc` there are times where total and taken move in opposite directions. Figure 2 shows the same benchmarks, but with branch miss percentage rather than aggregate totals. This is a common optimization metric; if the generalized "branches" and "branch-misses" events are wrong then time will be wasted optimizing in the wrong place. We find in `gcc` and `gzip` behavior that is divergent enough that using taken branches as a synonym for total could misdirect optimizers.

Sometimes different tools will disagree about which underlying event to use. For the "Level 1 Data Cache Reads" event on the Nehalem processor PAPI has a `PAPI_L1_DCR` predefined event which maps to `L1D_CACHE_LD:MESI` (described by the Intel Volume 3B documentation [10] as "Counts L1 data cache read request"). The equivalent perf_event generalized event is described at the syscall interface as the similar-sounding `L1D.OP_READ.RESULT_ACCESS` (although the `perf` tool calls it `L1-dcache-loads`); this maps to `MEM_INST_RETIRED:LOADS` ("Counts the number of instructions with an architecturally-visible load retired on the architected path"). Figure 3 shows phaseplots for both. For some benchmarks the events match relatively closely, but in the two shown the results are different, in some case by a factor of three!

B. Event Scheduling

Some events have elaborate hardware constraints and can only run in a certain subset of available counters. Perfmon2 and perfctr rely on libraries to provide event scheduling in userspace; perf_event does this in the kernel. Scheduling is performance critical; a full scheduling algorithm can require O(N!) time (where N is the number of events). Various heuristics are used to limit this, though this can lead to inefficiencies where valid event combinations are rejected as unschedulable.

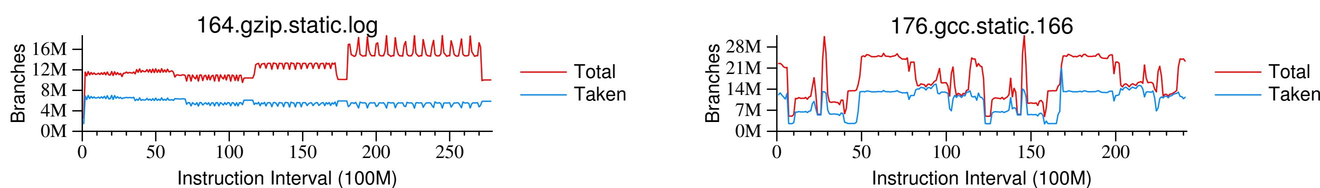

Fig. 1. AMD taken branches versus total branches phaseplot. The former was mistakenly used for the latter by perf_event through Linux 2.6.35. The examples show how the two events can have different (though related) characteristics that could confuse a user that did not know about the mixup.

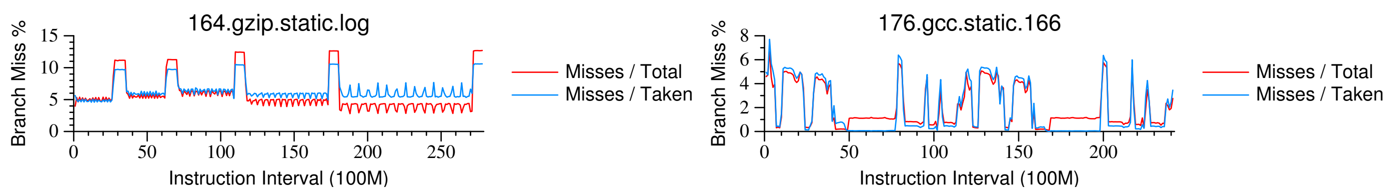

Fig. 2. These phaseplots show that calculating branch miss rates using the confused taken and total branches can have different results, including behavior that trends in opposite directions. This is why generalized events need to be chosen carefully and should be exposed to the user for easy validation.

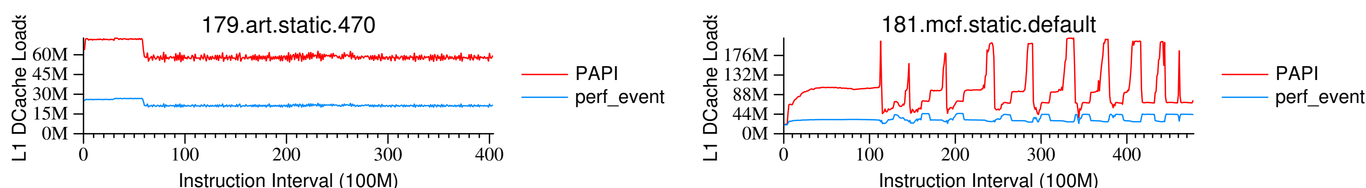

Fig. 3. A hazard of generalized event names: on Nehalem `PAPI_L1_DCR` (L1 Data Cache reads) and perf_event `L1D.OP_READ.RESULT_ACCESS` may sound similar but these phase plots show they have different results.

Experimentation with new schedulers is difficult with an in-kernel implementation as this requires kernel modification and a reboot between tests rather than simply recompiling a userspace library.

C. Multiplexing

Some users wish to measure more simultaneous events than the hardware can physically support. This requires multiplexing: events are run for a short amount of time, then switched out with other events and an estimated total count is statistically extrapolated. perf_event multiplexes in the kernel (using a round-robin fixed interval), which can provide better performance [11] but less flexibility. Userspace multiplexing is possible; PAPI can use May's [12] method of switching events based on a software timer.

D. Software Events

perf_event provides support for kernel software events not provided by hardware. Events such as context switches and page faults are exposed to the user using the same interface as hardware events, allowing easier access by tools.

E. New Hardware Features

As newer chips are released, they have new functionality that requires operating system support. perf_event has added support for many of these important new features.

1) Offcore Response: Nehalem and newer Intel chips provide Offcore Response events that allow filtered measuring of memory accesses (and other activity) that leave the core. Configuration of an offcore event requires programming *two* MSRs rather than one; this requires extra operating system support and also arbitration to keep multiple users from programming the limited set of MSR registers simultaneously.

2) Uncore and Northbridge Events: Modern processors include many cores in one CPU package. This leads to shared infrastructure (L2 and L3 caches, interconnects, and memory controllers). There are various events that measure these shared resources: Intel refers to them as *uncore events*, AMD calls them *northbridge events*; IBM Power has similar events.

Nehalem and later Intel chips have a diverse set of uncore PMUs that vary from generation to generation. The Nehalem-EX supports multiple uncores (C-Box, S-Box, B-Box, etc.); support for these was added in the recent Linux 3.6 release. Since uncore events access shared resources, the potential exists that information can be leaked to other users on a multi-user system. To avoid this security risk access to the shared events requires special operating system permission.

3) Sampled Interfaces: Recent processors provide sampling interfaces: certain instructions can be tagged and extra detailed information to be returned to the user.

AMD *Instruction Based Sampling (IBS)* [13] provides advanced sampling, where address, latency, cache miss information, and TLB miss information can be obtained with minimal, known, skid (that is, the statistics are provided for the actual instruction causing them and not some later instruction due to problems of attribution due to out-of-order execution). This is a huge improvement over normal sampling, where the skid can be a large unknown quantity.

Intel *Precise Event Based Sampling (PEBS)* allows specifying information be periodically be gathered for the instruction

TABLE I. MACHINES USED IN THIS STUDY.

Processor	Counters Available
Intel Atom 230	2 general 3 fixed
Intel Core2 T9900	2 general 3 fixed
AMD Athlon 64 X2	4 general

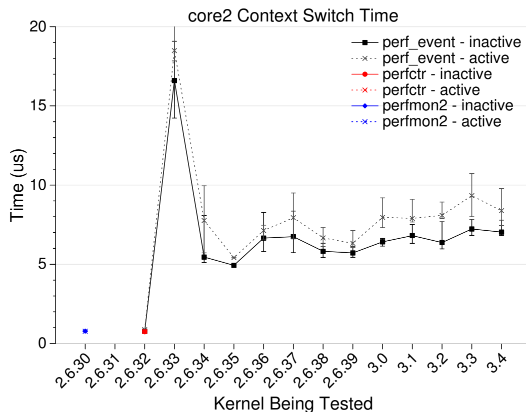

Fig. 4. Context Switch overhead as measured by lmbench when performance measurement is active and inactive. It is unclear why context switch time changed so drastically between 2.6.32 and 2.6.34.

immediately following a triggered event; this can include full register state as well as latency. A latency threshold can be configured for memory access events.

perf_event support for IBS and PEBS currently is used only for low-skid instruction profiling, but work is underway to export the extended information available with these interfaces.

4) Virtualized Counters: Recent investments in cloud computing and virtualization have raised interest in using performance counters inside of virtual machines. Recent work has enabled trapping access to the relevant MSRs to provide counter support for guest operating systems. perf_event provides additional support for gathering host KVM statistics.

5) AMD Lightweight Profiling: AMD Lightweight Profiling (LWP) [14] is a feature introduced with Bulldozer processor that allows using performance counters entirely from userspace, with minimal kernel intervention. Counters can be configured to measure events and place results directly into a buffer. This buffer can be polled occasionally by the user or, optionally, the operating system can send a signal when the buffer is full. Support for this requires extra saved state on context switch; this is done by extensions to the XSAVE/XRSTOR instructions. Support for LWP is not yet available in the Linux kernel.

IV. OVERHEAD COMPARISON

We compare the overhead of perf_event against the perfctr and perfmon2 interfaces on various x86_64 machines as listed in Table I. The older interfaces are obsolete with the introduction of perf_event, so to be a suitable replacement perf_event should have equal or better performance characteristics.

A. Context Switch Overhead

To provide per-process performance data, the operating system saves and restores the counter MSRs at context switch. We use the lmbench [15] `lat_ctx` context switch overhead benchmark (with size 0 / procs 8) to measure the overhead on various Linux kernels.

Figure 4 shows the average time and error from 10 runs of each kernel on a Core2 machine, with performance measurement active and inactive (active means the benchmark is run by the perf tool, inactive it is run standalone). We find the average overhead can increase up to 20% when the perf_event `perf` tool is monitoring the benchmark. lmbench is documented as having a 10-15% error range so this might not be significant. Kernels before 2.6.34 have wildly different behavior due to unknown causes; this makes makes comparison with older perfctr and perfmon2 kernels difficult. On these older kernels perfctr and perfmon2 only show a 2% overhead penalty.

B. Measurement Overhead

Users often conduct analysis via self-sampling; this involves instrumenting their code (either directly or via a library such as PAPI) to explicitly take performance measurements. The counter events are setup at program initialization and code is inserted at points of interest to start, stop, and read from the counters. The user wants to gather the counter data with the minimum possible overhead, as the measurements themselves will disrupt the flow of the program. We use the x86 `rdtsc` timestamp counter instruction to measure overhead on three x86_64 machines using perf_event, perfctr, and perfmon2. All three measurements are made in a single run by placing the timestamp instruction in between function calls. We compare the overheads for the most recent perfctr and perfmon2 releases. Direct comparisons of implementations is difficult; perf_event support was not added until version 2.6.31 but perfmon2 development was halted in version 2.6.30 and the most recent perfctr patch is against Linux 2.6.32. We also test a full range of perf_event kernels starting with 2.6.32 and running through the 3.5 release. The kernels were all compiled with gcc 4.4 with configurations chosen to be as identical as possible. Recent kernels need `/proc/sys/kernel/nmi_watchdog` set to 0 to keep the kernel watchdog from "stealing" an available counter. We also disable frequency scaling during the experiments.

Figure 5 shows the timing results obtained when starting a pre-existing set of events, immediately stopping them (with no work done in between) and then reading the counter results. This gives a lower bound for how much overhead is involved when self-monitoring. In these tests only one counter event is being read. We run 1024 tests and create boxplots that show the 25th and 75th percentiles, median, errorbars indicating one standard deviation, and any additional outliers. The perfmon2 kernel has the best results, followed by perfctr. In general all of the various perf_event kernels perform worse, with a surprising amount of inter-kernel variation.

Figure 6 shows the overhead of just performance counter reads. perfctr has the lowest overhead by far, due to its use of the `rdpmc` instruction. perfmon2 is not far behind, but all of the perf_event kernels lag, including kernels which have newly-added support for using `rdpmc`. While `rdpmc` helps for perf_event, it does not get near the low overhead from the instruction that `perfctr` does. This is partly because the perf_event interface requires reading the counter twice and subtracting to get final results (perfctr only needs one read).

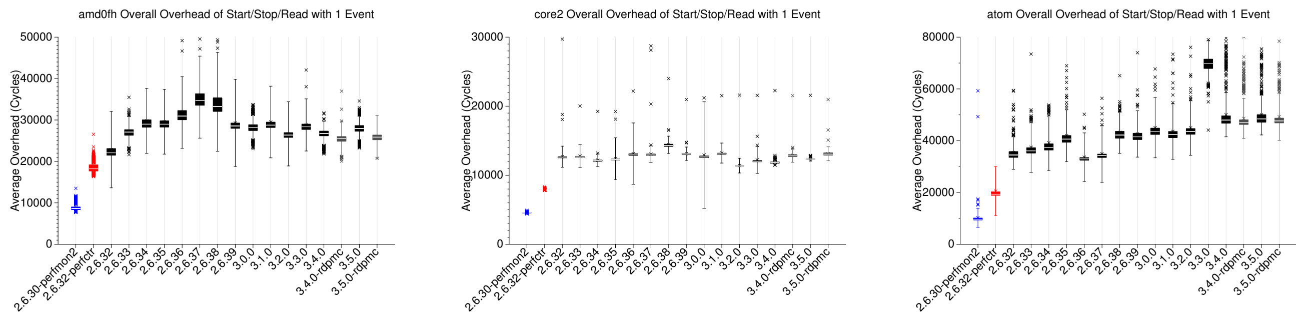

Fig. 5. Total time overhead boxplots when doing a Start/Stop/Read of one performance counter. With a boxplot, the box shows 25th to 75th percentile (with median); the error bars show one standard deviation, and any additional outliers are shown.

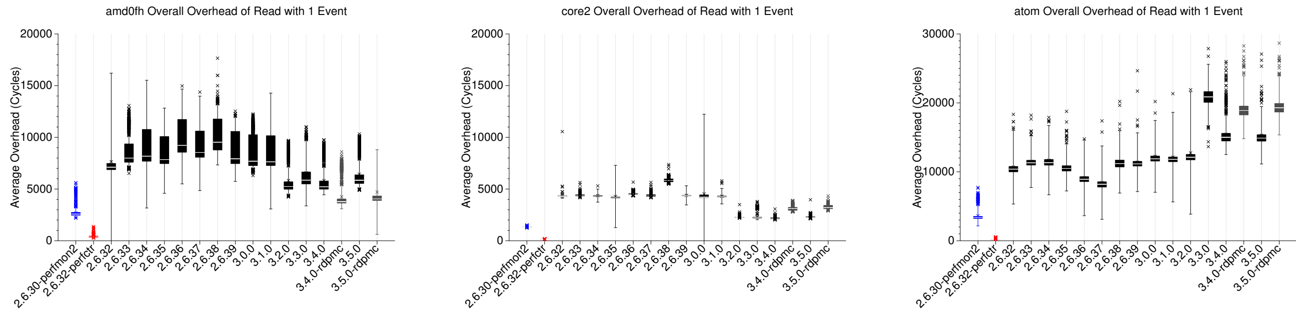

Fig. 6. Time overhead boxplot when reading one performance counter.

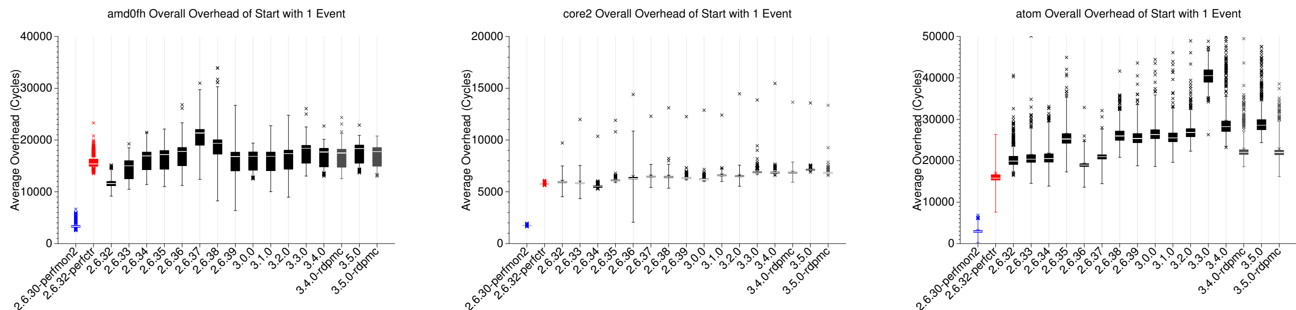

Fig. 7. Time overhead boxplot when starting one performance counter.

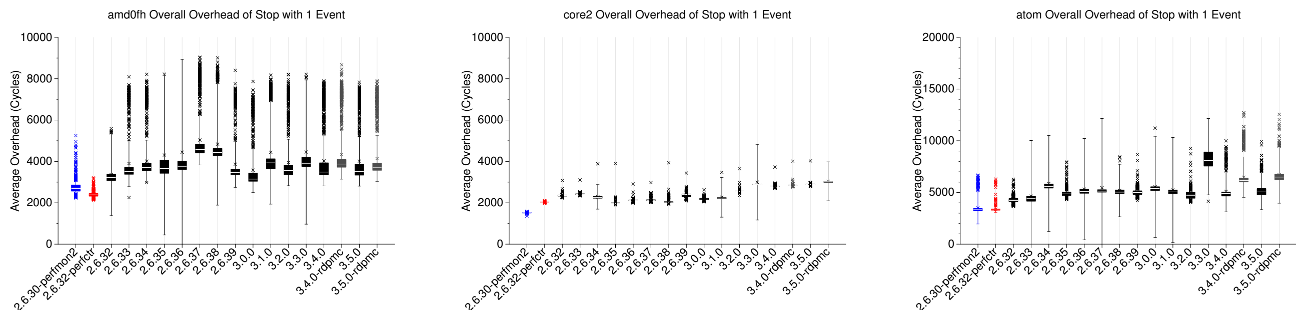

Fig. 8. Time overhead boxplot when stopping one performance counter.

Figure 7 shows the overhead of starting a counter. While this appears as a simple `ioctl()` call to the user, it can trigger a lot of kernel work. perfmon2 is fast, as costly setup such as event scheduling is done in advance. In contrast the perf_event kernel has to do scheduling at start time. It is surprising that the perfctr results are not better, but this could be due to its strangely complicated interface, with complicated `ioctl` arguments for start and stop.

Figure 8 shows the overhead of stopping a counter. The differences between implementations are not as pronounced; the kernel does little besides telling the CPU to stop counting.

The perf_event self-monitoring interface has higher overhead than previous implementations. This is mostly due to the amount of work the kernel does at event start time (which other interfaces can do in advance in user-space). Another factor is the slowness of counter reads, which are not improved even when using `rdpmc` to avoid kernel entry. Overhead for perf_event has a lot of inter- and intra-kernel variation; it has proven difficult to isolate the reason for this. It is possible that other unrelated changes to the kernel perturb caches and other CPU structures enough to cause the variations. Attempts to narrow down the sources of variation through "git-bisect" binary searches did not show any obvious candidates for the variation.

The previous graphs look at overhead when measuring one event at a time; Figure 9 show variation as we measure more than one event. As expected the overhead increases linearly as more events are added since the number of MSRs read increases with the event count. Surprisingly the `rdpmc` perf_event code shows the worst overall performance.

V. Related Work

Previous performance counter investigations concentrate either on the underlying hardware designs or on high-level userspace tools; our work focuses on the often overlooked intermediate operating system interface.

Mytkowicz et al. [16] explore measurement bias and sources of inaccuracy in architectural performance measurement. They use PAPI on top of perfmon and perfctr to investigate performance measurement differences while varying the compiler options and link order of benchmarks. They measure at the high level using PAPI and do not investigate sources of operating system variation. Their work predates the introduction of the perf_event interface.

Zaparanuks et al. [17] study the accuracy of perfctr, perfmon2, and PAPI on Pentium D, Core 2 Duo, and AMD Athlon 64 X2 processors. They measure overhead using libpfm and libperfctr directly, as well as the the low and high level PAPI interfaces. They find measurement error is similar across machines. Their work primarily focuses on counter accuracy and variation rather than overhead (though these effects can be related). They did not investigate the perf_event subsystem as it was not available at the time.

VI. Future Work and Conclusion

We investigate the features of the Linux perf_event and compare its overhead against the existing perfmon2 and perfctr interfaces. We find that it has higher overhead than previous implementations at least in part due to its "everything goes into the kernel" philosophy. Despite the higher overhead, perf_event is becoming feature complete and widely used, and as such provides great benefit to the Linux community.

CPU vendors continually work on providing faster access to the counters, including AMD Lightweight Profiling and the `spflt` instruction (which can start counters without entering the kernel) found in the new Intel MIC chip [18]. More work needs to be done to encourage interaction between the operating system developers and the hardware designers to make sure new interfaces are suitable for implementation with perf_event.

With modern high performance architectures only becoming more complicated, performance counters are a key resource for finding and avoiding system bottlenecks. The merge of perf_event has provided easy access to hardware counters to Linux users for the first time; the pressure is on to keep extending the infrastructure until it can meet the needs of all performance analysts and programmers.

All code and data used in this paper can be found here: git://github.com/deater/perfeventoverhead.git.

References

[1] "Top 500 supercomputing sites, operating system family," http://www.top500.org/statistics/list/.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[3] J. Levon, "Oprofile," http://oprofile.sourceforge.net.

[4] M. Pettersson, "The perfctr interface," http://user.it.uu.se/˜mikpe/linux/perfctr/2.6/.

[5] S. Eranian, "Perfmon2: a flexible performance monitoring interface for Linux," in *Proc. 2006 Ottawa Linux Symposium*, Jul. 2006, pp. 269–288.

[6] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, Sep. 2010.

[7] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proc. 38th IEEE/ACM International Symposium on Computer Architecture*, Jun. 2011.

[8] V. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[9] Standard Performance Evaluation Corporation, "SPEC CPU benchmark suite," http://www.specbench.org/osg/cpu2000/, 2000.

[10] Intel, *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2009.

[11] R. Azimi, M. Stumm, and R. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *Proc. 19th ACM International Conference on Supercomputing*, 2005.

[12] J. May, "MPX: Software for multiplexing hardware performance counters in multithreaded programs," in *Proc. 15th IEEE/ACM International Parallel and Distributed Processing Symposium*, Apr. 2001, p. 8.

[13] P. Drongowski, *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*, Advanced Micro Devices, Inc., 2007.

[14] *Lightweight Profiling Specification*, Advanced Micro Devices, 2010.

[15] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proc. of the 1996 USENIX Annual Technical Conference*, 1996, pp. 279–294.

[16] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[17] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 23–32.

[18] Intel, *Knights Corner Performance Monitoring Units*, May 2012.

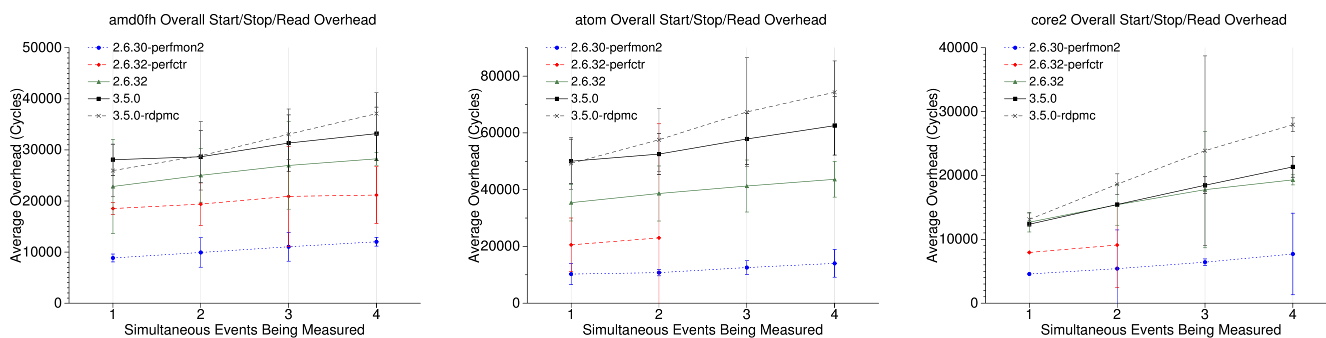

Fig. 9. Overall overhead while varying the number of events being measured.