Self-monitoring Overhead of the Linux perf_event Performance Counter Interface

Vincent M. Weaver
Electrical and Computer Engineering
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. Low-overhead access to these counters is necessary for accurate performance analysis, making the operating system interface critical to providing low-latency performance data. We investigate the overhead of self-monitoring performance counter measurements on the Linux perf_event interface.**

We find that default code (such as that used by PAPI) implementing the perf_event self-monitoring interface can have large overhead: up to an order of magnitude larger than the previously used perfctr and perfmon2 performance counter implementations. We investigate the causes of this overhead and find that with proper coding this overhead can be greatly reduced on recent Linux kernels.

I. Introduction

Most moderns CPUs contain hardware performance counters: architectural registers that allow low-level analysis of running programs. Low-overhead access to these counters (both in time and in CPU resources) is necessary for useful and accurate performance analysis. Raw access to the underlying counters typically requires supervisor-level permissions; this access is usually managed by the operating system. The operating system thus enters the critical path of low-overhead counter access.

The Linux operating system kernel is widely used in situations where performance analysis is critical. This includes High Performance Computing (HPC) where as of June 2014 97% of the Top500 supercomputers in the world run some form of Linux [1]. At the other extreme are embedded systems where performance optimization enables the maximum use of limited hardware resources. Linux is key here too, as Android phones (which run Linux kernels) make up 81% of all smartphones shipped in 2013 [2].

Until 2009 the default Linux kernel did not include support for performance counters (there were out-of-tree implementations available but these were not included in most distributions and required custom kernel patching). The 2.6.31 Linux release introduced first-class performance counter support with the perf_event subsystem. The perf_event interface differs from previous Linux performance counter implementations, primarily in the amount of functionality handled by the operating system kernel. Many of the tasks perf_event does in-kernel were previously done in userspace libraries, impacting measurement overhead.

There are three common ways of using performance counters: *aggregate measurement*, *statistical sampling* and *self-monitoring*.

Aggregate measurement is the easiest and lowest-overhead method of gathering performance results. The counters are enabled just before a program starts running and total results are gathered when it finishes. The operating system aids measurement by saving and restoring values on context switch and also handling overflows where a value exceeds the counter width. This methodology provides exact total measurements, showing high-level overviews of program behavior. For detailed per-function or per-instruction results other methods must be used.

With *statistical sampling* the counters are programmed to periodically gather measurements at regular intervals via the generation of timer or overflow interrupts. By recording the instruction pointer at time of sampling, function-granularity performance results can be extrapolated statistically. This has the advantage that the program does not have to be modified with calls to measurement routines, but does have the disadvantage that the values are not exact and resolution can be coarse. Setting the sampling rate higher can mitigate this, but if set too high leads to increased interrupt and operating system overhead which can overwhelm the actual measurements. Not all hardware supports sampling; some systems lack a working overflow interrupt (for example various ARM systems[1]), but this can be worked around in software.

This paper concentrates instead on *self-monitoring* usage of the counters, with an emphasis on the time overhead imposed by the operating system interface. With self-monitoring a program is instrumented to gather exact performance measurements for blocks of code. Calls to measurement routines are inserted into the code either by modifying the source code or by using some form of binary instrumentation. While statistical sampling can tell you, on-average, which function in your program typically has the most of a metric (say L2-cache misses), with self-monitoring you can insert calipers around that function and find exact measurements.

The widely used cross-platform PAPI [3] performance measurement library uses self-monitoring to provide detailed measurements. When PAPI transitioned to perf_event from the previous perfctr and perfmon2 interfaces, a sharp increase in overhead was found. At the time it was hoped that this was

[1]The overflow interrupt is not available on Raspberry Pi hardware, and there are various errata that can cause overflow interrupts to be lost on Cortex A8 and Cortex A9 systems

978-1-4799-1957-4/15/$31.00 ©2015 IEEE

due to the newness of perf_event and that the overhead would decrease over time as the code was improved. As seen in Figure 1 this was found not to be the case, which led to this investigation of why perf_event overhead is high and what can be done to get results more inline with the previous perfctr and perfmon2 interfaces.

II. Background

The hardware interface for performance counters involves accessing special CPU 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 many more are possible [4]. Reads and writes to the configuration registers usually require special privileged (ring 0 or supervisor) instructions; the operating system can allow access by translating user requests into the proper low-level CPU calls. Access to counting registers may require extra 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 events to monitor, 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 (handling limitations on which events can go into which counters), multiplexing (swapping events in and out and using time accounting to approximate the availability 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 tools provide common cross-architecture and cross-platform interfaces to performance counters. PAPI [3] is a widely-used performance counter abstraction interface that is in turn used by other tools (such as TAU [5], Vampir [6], or HPCToolkit [7]) that provide graphical frontends to the underlying performance counter infrastructure. Although users will typically work with these higher-level interfaces directly, performance analysis works best if backed by libraries and operating systems that can provide efficient low-overhead access to the counters.

A. Counter Implementations

Most modern processors have support for performance counters and thus many operating systems support interfaces to access them. Commercial UNIX systems [8], [9], [10] provided some of the earliest hardware performance counter implementations. Microsoft Windows does not include native support for hardware performance counters; third-party tools (such as Intel VTune [11], Intel VTune Amplifier, Intel PTU [12], and AMD's CodeAnalyst [13]) provide access by programming the CPU registers directly. Since counter state is not saved on context switch only system wide sampling is available. Apple OSX is similar to Windows in that only system wide sampling is available by directly programming the hardware. A tool that does this is Shark [14]. FreeBSD[15] provides a more feature-rich interface that is similar to that available for Linux.

Patches providing Linux support appeared soon after the release of the original Pentium processor, the first x86 processor with counter hardware. These early implementations [16], [17], [18], [19], [20], [21] exported raw access to the underlying Model Specific Registers (MSRs) but also gradually added more features such as saving values on context-switch. Despite these many attempts at Linux performance counter interfaces, only three saw widespread use before the adoption of perf_event: *oprofile*, *perfctr*, and *perfmon2*.

1) Oprofile: Oprofile [22] is a system-wide sampling profiler included in the 2002 Linux 2.5.43 release. It allows sampling with arbitrary performance events (or a timer if you lack performance counters) and provides frequency graphs, profiles, and stack traces. The kernel interface is through a pseudo-filesystem under `/dev/oprofile`; profiling is started and stopped by writing there. 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 does not support self-monitoring. Currently Oprofile is still supported, although it is considered deprecated (with perf_event being the replacement).

2) Perfctr: Perfctr [23] is a widely-used performance counter interface introduced in 1999. The kernel interface involves opening a `/dev/perfctr` device and accessing it with various `ioctl()` calls. Fast counter reads (that do not require a slow system call into the kernel) are supported using the x86 `rdpmc` instruction in conjunction with `mmap()`. A libperfctr is provided which abstracts the kernel interface.

3) Perfmon2: Perfmon2 [24] is an extension of the 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 original v2 interface involved 12 new system calls. In response to concerns raised during code review this was reduced to just 4, but in the end this approach was abandoned by the Linux developers in preference for perf_event. We use the last publicly released perfmon2 git tree in this work, a patched Linux 2.6.30 kernel which uses version 2.9 of the interface with all 12 system calls. 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).

B. perf_event

The *perf_event* subsystem [25] was created in 2009 as a response to the 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 underlying perf_event design philosophy is to provide as much functionality and abstraction as possible in the kernel, making the interface straightforward for ordinary users.

The interface is built around file descriptors allocated with the new `perf_event_open()` system call [26]. 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 ring buffer which can be accessed via `mmap()`; new data availability can be determined via a signal or by `poll()`.

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). Heuristics are used to limit this, though this can lead to inefficiencies where valid event combinations are rejected as unschedulable. 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.

Some users wish to measure more simultaneous events than the hardware can physically support. This requires multiplexing: events are repeatedly run for a short amount of time, then switched out with other events and an estimated total count can be statistically extrapolated [27], [28]. perf_event multiplexes in the kernel (using a round-robin fixed interval), which can provide better performance [29] but less flexibility. More advanced sampling patterns and randomized intervals could provide lower error [30], [31] but perf_event does not support such modes.

C. 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 [32] 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, as well as causing security issues (a regular user can obtain root privileges if given write access to arbitrary MSRs; see CVE-2013-0268). Only x86 processors are supported, and per-process measurements are not possible (due to lack of MSR save on context-switch). True self-monitoring is not possible, although "markers" can be added to user code that signal an external monitoring process to read the counters.

LiMiT [33] 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. Experimental Setup

I compare the self-monitoring overhead of perf_event against the perfctr and perfmon2 interfaces on various x86_64 machines as listed in Table I.

TABLE I: Machines used in this study.

Processor	Counters Available	
Intel Atom Cedarview D2550	2 general	3 fixed
Intel Core2 P8700	2 general	3 fixed
Intel IvyBridge i5-3210M	4 general	3 fixed
AMD Bobcat G-T56N	4 general	

Performance measurements have the potential to disrupt the execution of programs; therefore it is necessary to keep instrumentation overhead as small as possible. Self-monitoring tools such as PAPI add two different chunks of code to an instrumented program. The first initializes the library and sets up the events to be measured. This initialization step can involve a considerable amount of code, but if placed early in the program execution it hopefully does not affect measurements that happen later. The second piece of code added is the actual instrumentation, the calipers conducting measurements around the area of interest. This code starts the counters and then stops and reads values when finished (often the raw values are stored for later analysis, as any additional processing would add extraneous overhead). These operations are the ones that need to have the lowest possible overhead as not to interfere with measurement.

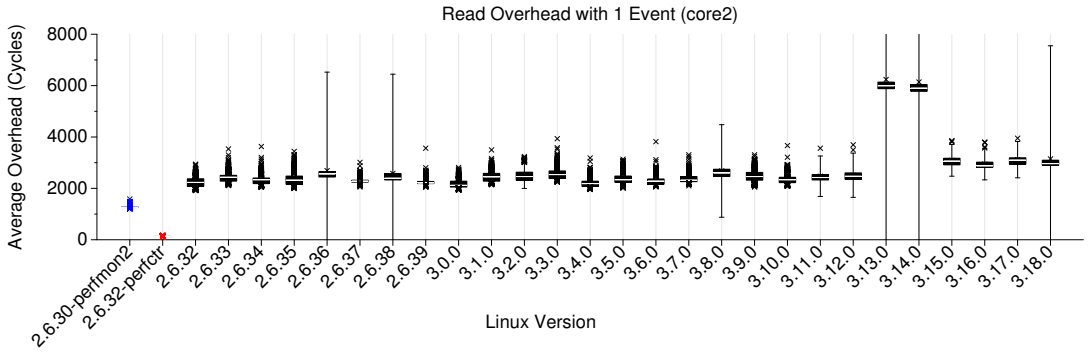
We use the x86 `rdtsc` timestamp counter to obtain fine-grained timing of overhead. We measure the overhead of start, stop, and read on four x86_64 machines using perf_event, perfctr, and perfmon2. An empty code block (a start followed by an immediate stop) is used to avoid any impact that arbitrary test code might have on measurement overhead. All three measurements are made in a single run by placing the timestamp instruction in between function calls, as shown in the following perf_event example (the perfmon2 and perfctr code is similar):

```
/* Events opened/initialized previously */

start_before=rdtsc();
ret1=ioctl(fd[0], PERF_EVENT_IOC_ENABLE,0);
start_after=rdtsc();

ret2=ioctl(fd[0], PERF_EVENT_IOC_DISABLE,0);
stop_after=rdtsc();

ret3=read(fd[0],buffer,BUFFER_SIZE*
     sizeof(long long));
read_after=rdtsc();
```

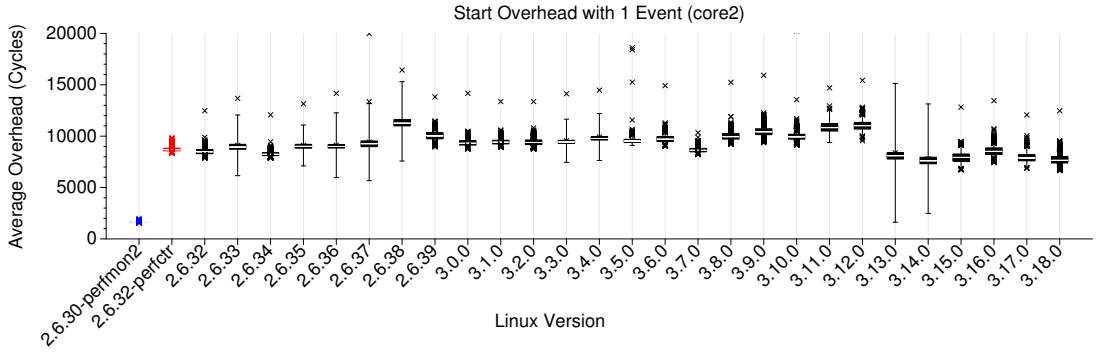
Direct comparisons of implementations are difficult; perf_event support was not added until Linux 2.6.31 but perfmon2 development was halted in 2.6.30 and the most recent perfctr patch is against 2.6.32.

I test a full range of perf_event kernels starting with 2.6.32 and running through 3.18, and show cross-machine results using a recent 3.16 kernel. For comparisons between the old and new interfaces older hardware (such as core2 systems) are needed as the old interfaces do not support newer CPUs.

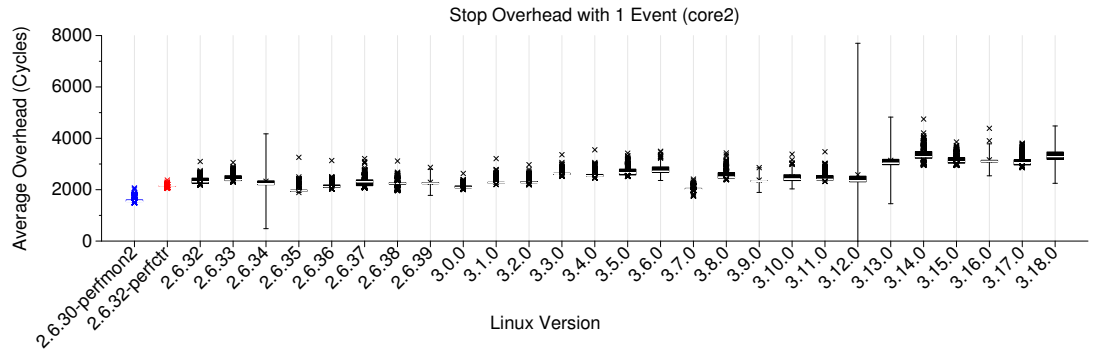
Unless otherwise specified the kernels were compiled with gcc 4.4 with configurations chosen to be as identical as possible (the configuration files used are available from our website).



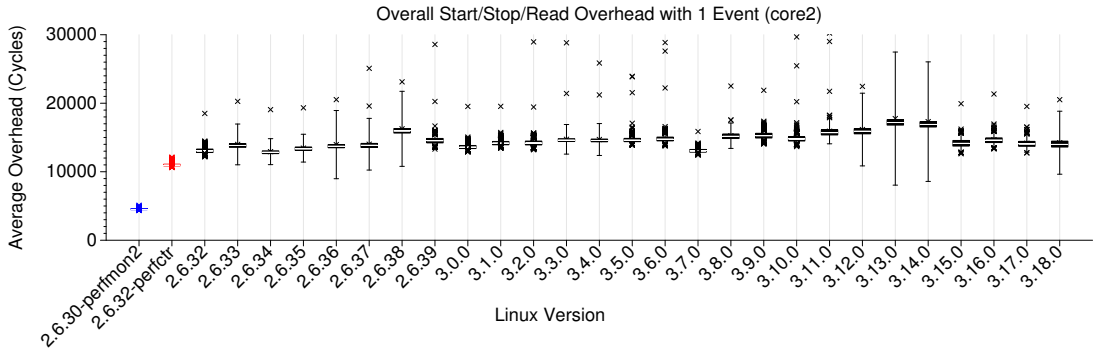
(a)



(b)



(c)



(d)

Fig. 1: Initial per-kernel self-monitoring measurement overhead comparison on core2. A core2 is used as perfctr and perfmon2 are not supported on more modern CPUs. The plots are boxplots: the solid box shows 25th to 75th percentile with a white line at the median; the error bars show one standard deviation, and Xs mark outliers. Outliers are typically due to cache-misses.

The `/proc/sys/kernel/nmi_watchdog` value is set to 0 on perf_event to keep the kernel watchdog from "stealing" an available counter. We also disable dynamic frequency scaling during the experiments, as we found this affects the timing measurements on some systems.

IV. Overhead Comparison

What follows is an investigation of the causes of self-monitoring overhead. I investigate total overhead, and then break out the the read, start, and stop results separately.

A. Total Overhead

The overall plot in Figure 1d shows the initial total overhead 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. The outliers in the graph (which affect standard deviation) we find to be due to TLB and cache misses. The results shown were measured on an older core2 machine (as perfctr and perfmon2 do not support newer machines), but we find similar perf_event results on more recent platforms such as Intel Atom, Ivybridge, and AMD Bobcat processors.

B. Read Overhead

When conducting measurements the most critical source of overhead comes from read operations. While start and stop are necessary, they can be pushed earlier or later (letting the counter run in a free-running mode). Reads however must happen in the area of interest, and often multiple reads are done and stored for later analysis.

A performance counter read involves reading a low-level processor register; these reads can be slow, often taking hundreds of cycles. Some processors provide a method of allowing users direct access to these registers (the `rdpmc` instruction on x86); if that is not available then a register read involves a privileged access and must go through the operating system's (slower) system call interface.

The read plot in Figure 1a shows the overhead of just reading a performance counter value on a core2 machine. 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. I ran many experiments to determine the causes of the high perf_event overhead.

1) Compiler: I first look at the compiler as a possible source of overhead differences. In this paper I always use gcc 4.4 compiled kernels for consistency unless otherwise specified. This older version of the compiler is made necessary because older kernel will not compile with newer gcc versions.

Figure 2 shows results found when varying the gcc version (version 4.4, 4.6, 4.7 and 4.8). The ensuing variation is small and thus not likely a factor with overhead.

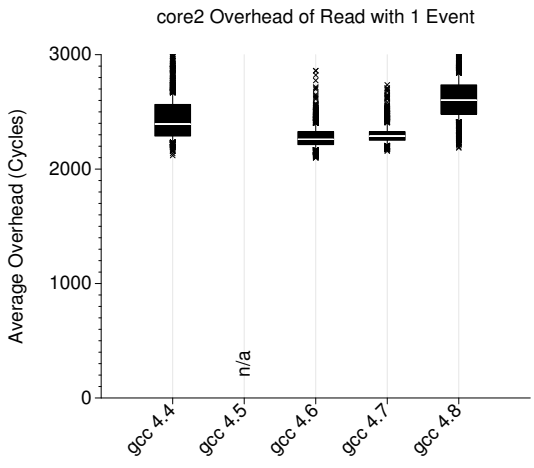


Fig. 2: Overhead boxplot of perf_event read on core2 while varying gcc version used to compile the Linux kernel. Compiler version does not majorly contribute to overhead.

2) Dynamic Frequency Scaling: At least on core2 systems having Dynamic Frequency Scaling would alter the results of my measurements. This is most likely due to the scaling affecting the `rdtsc` instruction used for timing measurements. To avoid this I disable frequency scaling on the test machines.

3) Dynamic vs Static Linking: I profiled the kernel at a low-level in an attempt to find sources of overhead, only to find that much of the overhead was coming from userspace.

Unlike previous implementations, perf_event does not use a custom system call to read the counters, but rather uses the stock `read()` call. This is implemented by the C library, which on Linux is typically dynamically linked. With such a binary the `read()` symbol is not resolved until run time at first use. This introduces the overhead of the dynamic link (which can be considerable) into our measurements. This is partly a limitation in the benchmark, but is a potential real-world problem with actual performance measurements if the executable's first call to `read()` happens in the self-monitoring instrumentation code.

To avoid this issue, one can statically link the executable (using the `-static` compiler option) or else hard-code the system call directly. The perfmon2 test does not see this overhead as it uses a custom system call from a staticly linked copy of the libpfm3 library. The perfctr test uses the `rdpmc` instruction directly and no function calls at all are used during a read.

Figure 3 shows that methods using static linking (static and syscall_static) improve the overhead results versus dynamic linking, with results approaching the perfmon2 measurement values.

4) `rdpmc` *Instruction:* The x86 architecture supports a processor state flag that allows user programs direct, low-overhead, access to the (usually restricted) performance counter registers. Programs can read the counters without having to go through the operating system.

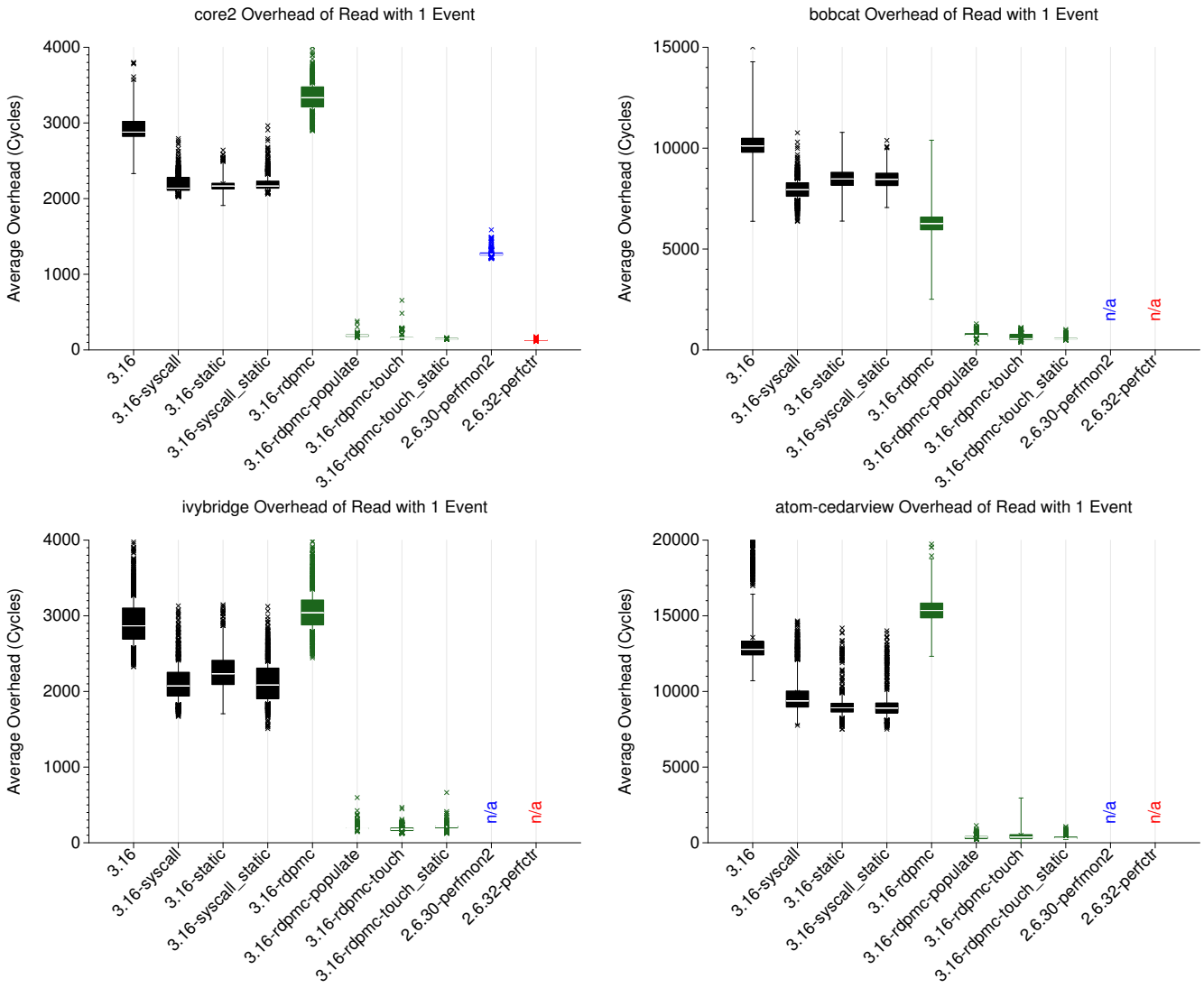


Fig. 3: Read results on various machines showing the overhead improvements gained using methods described in this paper. Perfmon2 and perfctr support is not available for newer architectures.

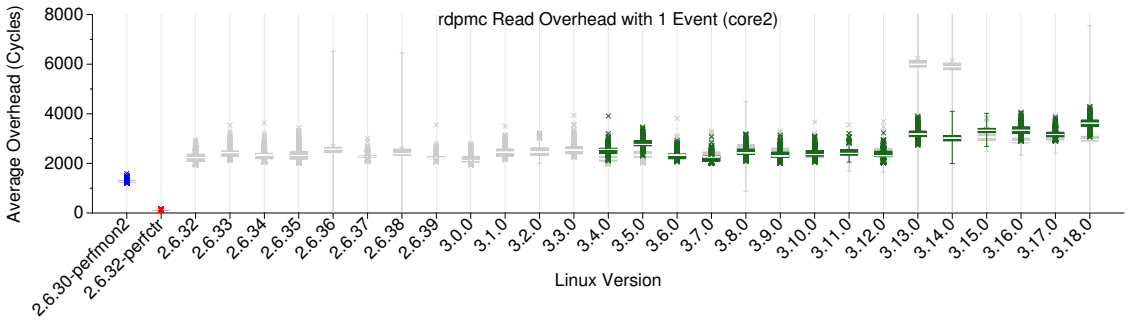


Fig. 4: Initial self-monitoring read overhead on perf_event while using `rdpmc`. The `rdpmc` interface was introduced in Linux 3.4 and is shown in dark green, overlaying regular read results shown in light grey. The `rdpmc` results were expected to be much lower, but were not due to page faults in the mmap() interface.

The perfctr interface explicitly uses `rdpmc` for counter reads. Support for this was added to perf_event with version 3.4 of the Linux kernel. To use `rdpmc` on perf_event a helper kernel page must first be mapped into the process address space via `mmap()`. This mmap page contains information such as which raw counter to read (counters can be moved around due to multiplexing support) as well as how long the event has been scheduled (also for multiplexing support). After the mmap page is accessed for info, then the proper counter is read via `rdpmc`. A unique number is provided in the mmap page to allow verification that no overflows or other changes have happened that will invalidate the result (if so the read must be retried).

Our initial experiments found (to our great surprise) that the perf_event `rdpmc` code was no better, and in in some cases worse, than the overhead of using the read system call. This can be seen in Figure 4. After some investigation it was found that the overhead happens because the first access to the mmap'd kernel page causes a page-fault. Handling the page-fault and mapping the data into the program's address space can take thousands of cycles. perfctr avoids this overhead (despite also using a mmap page) by specifically inserting the page directly into the process' address space at initialization time so no page fault is necessary.

The page fault behavior can be mitigated by either using the `MAP_POPULATE` option to `mmap()` when creating the page, or else by explicitly touching the page to bring it in at creation time (rather than at first read time). Figure 3 shows the improved performance when using these methods (rdpmc-populate and rdpmc-touch). These methods entail much lower overhead, on par with perfctr, despite needing two `rdpmc` calls for perf_event[2].

5) Reading Multiple Times: The results presented so far have shown read overhead when doing a single counter read. Often a user will read multiple times, such as reading upon each iteration of a loop. This spreads around the startup cost and in general each subsequent read should mitigate the initial high overhead. Figure 5 shows this effect happens when using `rdpmc` but the behavior when using default `read()` based measurements is more complex. Upon further investigation it turns out that the subsequent read operations are causing L1 Data Cache misses which add extra overhead.

C. Start Overhead

As can be seen in the start graph in Figure 1b, the overhead of start on perf_event could be better. Events are started with an `ioctl()` call which can trigger a lot of kernel work.

perfmon2 has impressively fast start code; costly setup (such as event scheduling) is done in userspace and can be done in advance. In contrast perf_event's expensive in-kernel event scheduling happens at start time (instead of at event creation). It is surprising that the perfctr results are not better, but this could be due to its unusually complex interface involving complicated `ioctl` arguments for start and stop.

[2]The perf_event interface does not zero the counter at start time, so to calculate the count `rdpmc` must be run before and after the instrumented code and the difference calculated

Figure 6 shows the start overhead behavior on various x86 systems when optimized in the same way that reads were. Using a statically linked binary helps with overhead (due to dynamic linking overhead, as seen with read) but perfmon2 start overhead is still much lower.

D. Stop Overhead

The stop graph in Figure 1c shows the overhead of stopping a counter on core2. The differences between implementations are not as pronounced; the kernel does little besides telling the CPU to stop counting. Still, perfctr and perfmon2 have less overhead. This is not affected much by overhead reduction methodology, as shown in Figure 7. Static linkage does not make as much difference because in our test the start `ioctl()` call pays the price for dynamic linking and the code is already set up when the stop `ioctl()` happens.

E. Varying number of events

The previous graphs look at overhead when measuring one event at a time; Figure 8 show variation on Core2 as we measure more than one event. On most implementations the overhead increases linearly with more events, as the number of MSR reads increases with the event count. perfctr does not grow as fast because it has the ability to read the counters for multiple events using only one memory page, while perf_event requires accessing a different mmap page for each event.

V. Related Work

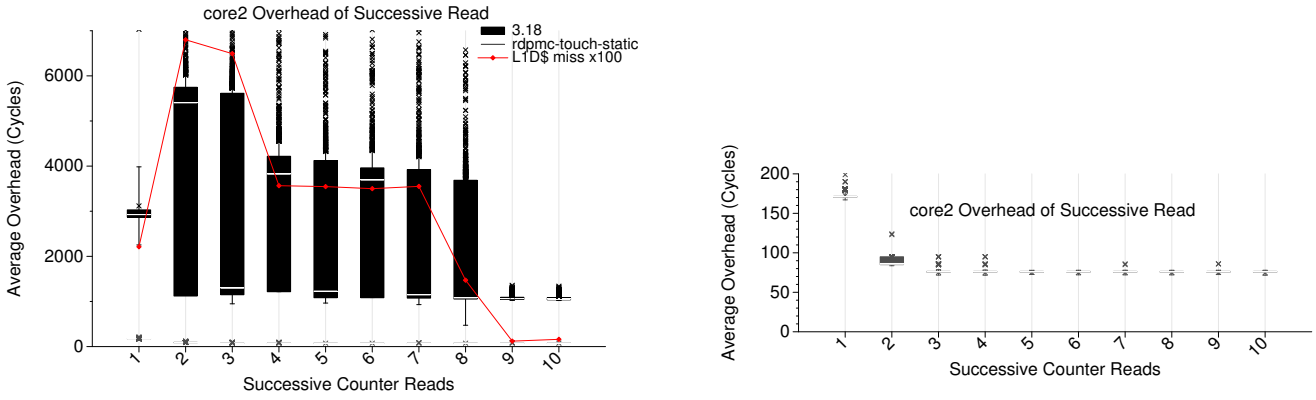
Previous performance counter investigations concentrate either on the underlying hardware designs or on high-level userspace tools; my work focuses on the often overlooked intermediate operating system interface.

Mytkowicz et al. [34] 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. [35] 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.

DeRose et al. [36] investigate performance counter variation and error on a Power3 system with regard to startup and shutdown costs. They do not discuss the underlying operating system interface.

Maxwell et al. [37], Moore et al. [38] and Salayandia [39] investigate PAPI overhead, but they do not explore operating system differences. Their work predates PAPI support for perfmon2 or perf_event.



(a) Multiple read overhead

(b) Bottom of graph zoomed to show rdpmc detail

Fig. 5: core2 perf_event read overhead showing the cost of repeated calls. One would expect the overhead to drop off; we find that due to L1 Cache misses the behavior is more complex.

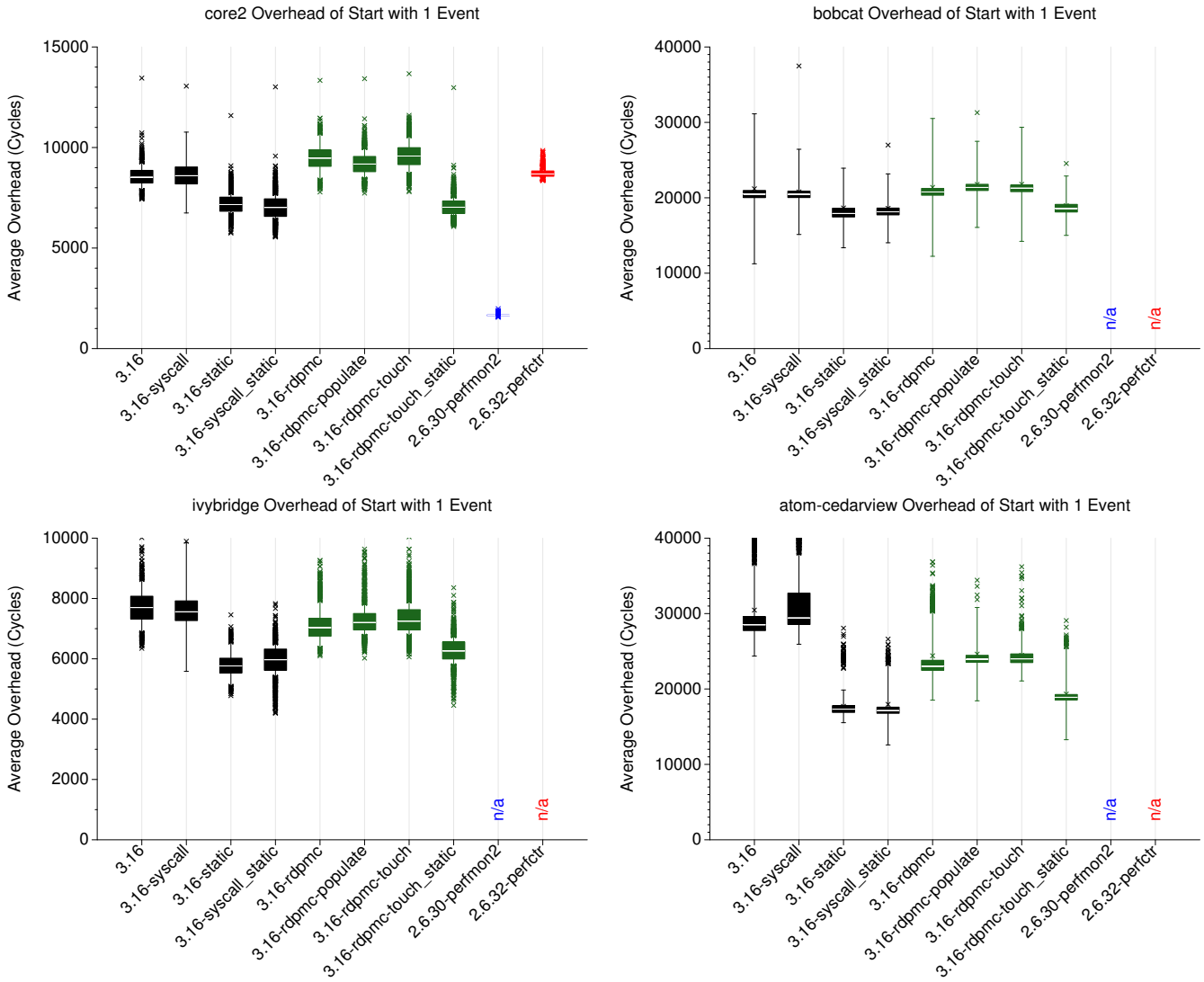


Fig. 6: Time overhead of starting counters on various machines when using different overhead reduction methods.

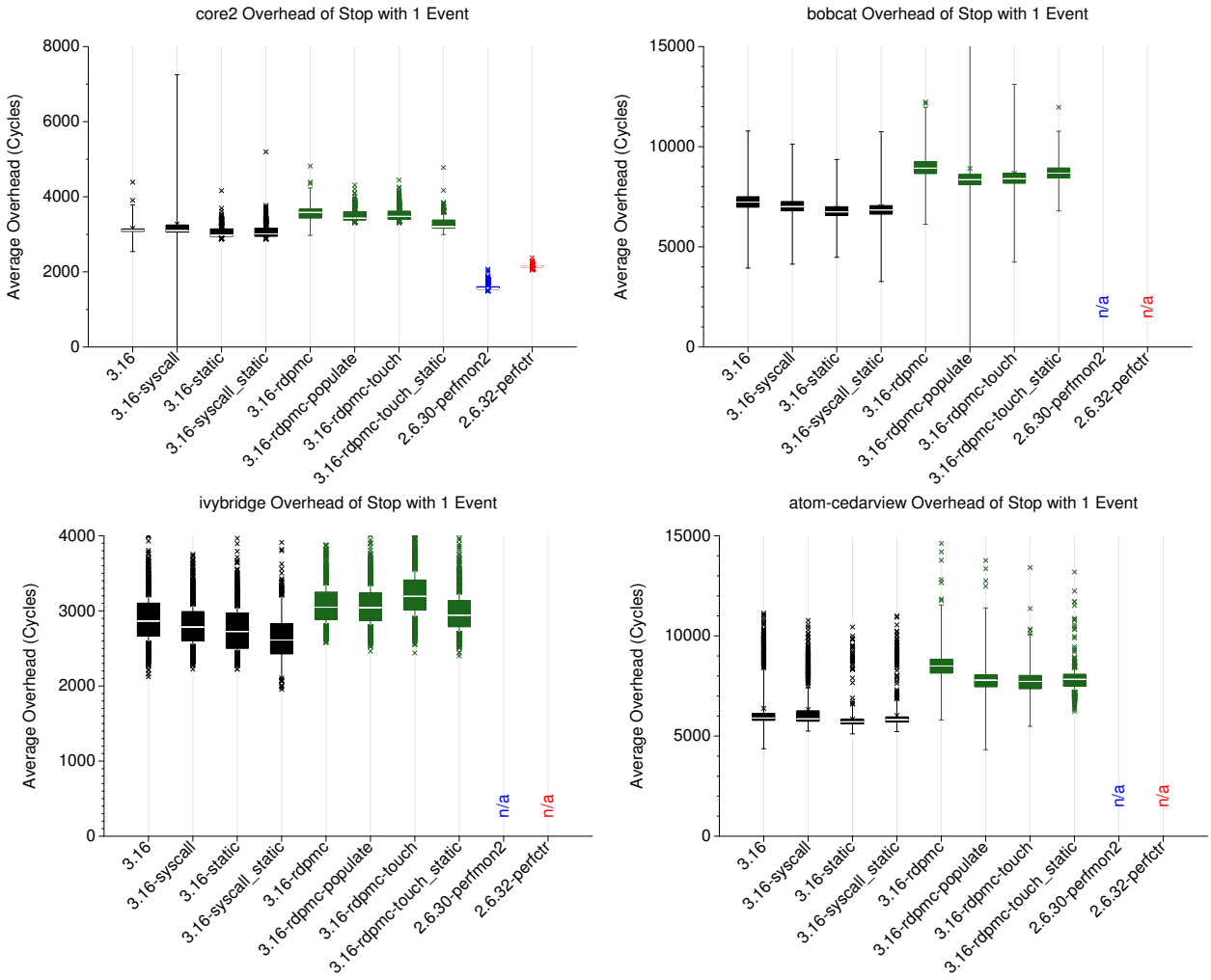


Fig. 7: Time overhead of stopping counters on various machines when using different overhead reduction methods.

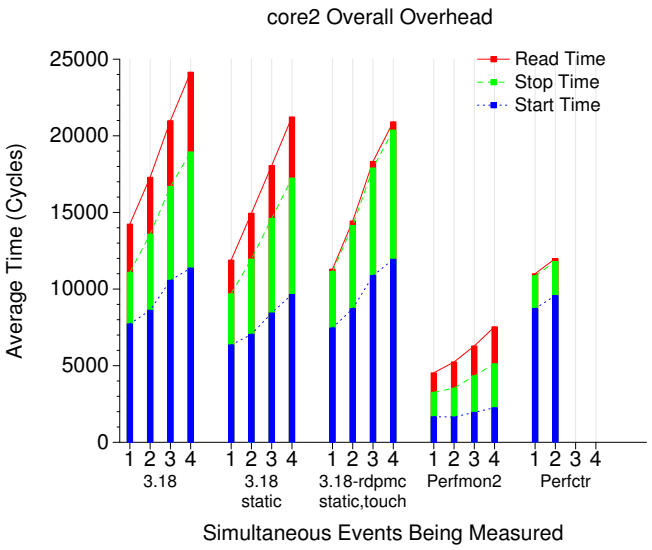


Fig. 8: Overall overhead on core2 while varying the number of events being measured.

Zhang et. al [40] propose providing high-performance counters directly to the operating system for use in scheduling and other tasks. In this case the counters are not exposed to the user, limiting usefulness in analysis or optimization.

VI. Conclusion and Future Work

Performance counters are a key resource for finding and avoiding system bottlenecks. Modern high performance architectures are complex enough that it is hard to create theoretical models of performance; analysis is even more difficult when using parallel code bases. The easiest way to gather detailed actual performance data is via hardware performance counters. Low-overhead access to these counters is critical when providing detailed performance measurements.

I investigate in detail the self-monitoring overhead of the Linux perf_event interface. I find that straightforward implementations of the interface (as used by PAPI) can have much larger overhead than the previous perfmon2 and perfctr interfaces. I show that with some minimal changes the perf_event read overhead can be reduced to values that are competitive with the previous interfaces; this involves using `rdpmc` for

counter reads, pre-faulting in the mmap kernel page, and using statically linked libraries.

I plan to use this knowledge to suggest improvements to both user tools such as PAPI as well as the Linux kernel so that the benefits of low-overhead measurements can be obtained without extra user intervention. In addition I would like to investigate overhead on other architectures such as Power and ARM. Modern processors have underutilized advanced counter implementations; the overhead of other interfaces (such as AMD Lightweight Profiling) and other PMUs (such as Uncore and Northbridge events) remain to be investigated.

All code and data used in this paper can be found at the following website: http://web.eece.maine.edu/~vweaver/projects/perf_events/overhead/

REFERENCES

[1] Top500, "Top 500 supercomputing sites, operating system family," http://www.top500.org/statistics/list/, 2014.

[2] IDC, "Android pushes past 80% market share," http://www.idc.com/getdoc.jsp?containerId=prUS24442013, 2013.

[3] 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.

[4] V. Salapura, K. Ganesan, A. Gara, M. Gscwind, J. Sexton, and R. Walkup, "Next-generation performance counters: Towards monitoring over thousand concurrent events," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2008, pp. 139–146.

[5] S. Shende and A. Malony, "The Tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[6] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[8] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, *Performance Analysis Using the MIPS R10000 Performance Counters*, Silicon Graphics Inc., 1996.

[9] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proc. IEEE/ACM 30th International Symposium on Microarchitecture*, Dec. 1997, pp. 292–302.

[10] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, L. Sites, M. Vandevoorde, A. Waldspurger, and W. Weihl, "Continuous profiling: Where have all the cycles gone?" in *Proc. 16th ACM Symposium on Operating Systems Principles*, Oct. 1997, pp. 357–390.

[11] J. Wolf, *Programming Methods for the Pentium™ III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment*, Intel Corporation, 1999.

[12] Intel, "Intel™ Performance Tuning Utility," http://software.intel.com/en-us/articles/intel-performance-tuning-utility/, 2014.

[13] P. Drongowski, *An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer*, Advanced Micro Devices, Inc., 2008.

[14] Apple Developer Connection, "Optimizing with shark: Big payoff, small effort," http://developer.apple.com/tools/shark_optimize.html, 2004.

[15] J. Koshy, "PmcTools," http://wiki.freebsd.org/PmcTools, 2009.

[16] D. Mentrè, "Hardware counter per process support," https://lkml.org/lkml/1997/11/26/53, 1997.

[17] M. Goda and M. Warren, "Linux performance counters pperf and libpperf," 1997.

[18] R. Berrendorf and H. Ziegler, "PCL – the performance counter library: A common interface to access hardware performance counters on microprocessors," Forschungzentrum Jülich GmbH, Tech. Rep. FZJ-ZAM-IB-9816, 1998.

[19] H. Hoyer, "p5 performance counter MSR driver," 1998.

[20] E. Hendriks, "PERF 0.7," 1999.

[21] D. Heller, "Rabbit: A performance counters library for Intel/AMD processors and Linux," http://www.scl.ameslab.gov/Projects/Rabbit/index.html, 2001.

[22] J. Levon, "Oprofile," http://oprofile.sourceforge.net, 2002.

[23] M. Pettersson, "The perfctr interface," http://user.it.uu.se/~mikpe/linux/perfctr/2.6/, 1999.

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

[25] T. Gleixner and I. Molnar, "Performance counters for Linux," 2009.

[26] V. Weaver, "perf_event_open manual page," in *Linux Programmer's Manual*, M. Kerrisk, Ed., Dec. 2013.

[27] W. Mathur and J. Cook, "Improved estimation for software multiplexing of performance counting," in *Proc. 13th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Sep. 2005, pp. 23–34.

[28] W. Mathur, "Improving accuracy for software multiplexing of on-chip performance counters," Master's thesis, New Mexico State University, May 2004.

[29] 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.

[30] T. Mytkowicz, P. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, 2007, pp. 286–300.

[31] M. Casas, R. Badia, and J. Labarta, "Multiplexing hardware counters by spectral analysis," in *Para 2010 - State of the Art in Scientific and Parallel Computing*, Jun. 2010.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] L. DeRose, "The hardware performance monitor toolkit," in *Proc. 7th International Euro-Par Conference*, Aug. 2001, pp. 122–132.

[37] M. Maxwell, P. Teller, L. Salayandia, and S. Moore, "Accuracy of performance monitoring hardware," in *Proc. Los Alamos Computer Science Institute Symposium*, Oct. 2002.

[38] S. Moore, D. Terpstra, K. London, P. Mucci, P. Teller, L. Salayandia, A. Bayona, and M. Nieto, "PAPI deployment, evaluation, and extensions," in *Proc. of User Group Conference*, Jun. 2003.

[39] L. Salayandia, "A study of the validity and utility of PAPI performance counter data," Master's thesis, The University of Texas at El Paso, Dec. 2002.

[40] X. Zhang, S. D. amd G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Proc. of the 11th USENIX workshop on Hot topics in operating systems*, 2007, pp. 226–231.