This was presented at the 2017 Extreme-Scale Programming Tools Workshop

Enhancing PAPI with Low-Overhead `rdpmc` Reads

Yan Liu[1] and Vincent M. Weaver[1]

University of Maine, Orono ME 04469, USA
{yan.liu,vincent.weaver}@maine.edu

Abstract. The PAPI performance library is a widely used tool for gathering self-monitored performance data from running applications. A key aspect of self-monitoring is the ability to read hardware performance counters with minimum possible overhead. If read overhead becomes too large then the act of measurement will start to interfere with the gathered results, adversely affecting the performance analysis.
On Linux systems PAPI uses the perf_event subsystem to access the counter values via the `read()` system call. On x86 systems the special `rdpmc` instruction allows userspace measurement of counters without the overhead of entering the operating system kernel. We modify PAPI to use `rdpmc` rather than `read()` and find it typically improves the latency by at least a factor of three (and often a factor of six or more) on most modern systems. The improvement is even better on machines using a KPTI enabled kernel to avoid the Meltdown vulnerability. We analyze the effectiveness and limitations of the `rdpmc` interface and have gotten the rdpmc interface enabled by default in PAPI.

1 Introduction

PAPI [16] is a portable, cross-platform library for accessing hardware performance counters. These counters are found on most modern CPUs and are widely used when evaluating system and program performance. Various tools are available that can read the values of these performance counters (such as perf [7], LIKWID [23] and VTUNE [27]). While all of these tools can measure overall aggregate counts and perform statistical sampling, PAPI is one of the few that allows easy *self-monitoring*.

Self-monitoring is the ability to read the values of the counters from within the running program, allowing fine-grain "caliper" measurements solely around the code of interest. Other tools can provide overall counts for an entire program run, or gather samples periodically that can be used to extrapolate statistically where a program spends most of its time. However a self-monitoring tool like PAPI is required to get exact fine-grained measurements for a single function, or to measure the impact of just a few lines of program code.

Self-monitoring is a powerful methodology, but care must be taken to keep overhead low. To use PAPI the code of interest must be instrumented, which involves adding extra code to the program. If the extra code needed to read the counter values becomes too long or intrusive then the resulting measurements will start to be affected. Mytkowicz et al. [17] found that instrumentation which increased instruction count by just 2.5% interfered with properly correlating performance results. Mytkowicz et al. [18] also showed that simply adding an additional PAPI counter could be enough to cause noticeable perturbations. Low overhead is critical for accurate performance measurements.

Instrumenting a program with PAPI is a multi-step process. First, setup code is added to the beginning of the program that initializes PAPI and sets up an "event set" with the chosen performance events of interest. These setup routines can end up calling a large amount of library code, but since this is run only once during program initialization it has minimal impact on a long-running process. Next, caliper code is added around the region of interest. It is critical that that code has minimal overhead. The routines involved are `PAPI_start()` which starts the measurements, `PAPI_read()` which reads the counters, and `PAPI_stop()` which stops the measurements. The `PAPI_start()` and `PAPI_stop()` calls can be put away from the critical code section to avoid overhead by using two reads (before and after) and calculating the difference. This leaves `PAPI_read()` as the most important routine requiring low-overhead.

In an ideal system a hardware counter read would simply be an assembly language instruction loading from the special CPU counter register, followed by a store of the value to memory for later analysis. On actual systems there is additional overhead caused by the operating system, as well as indirection and housekeeping overhead inside the measurement library. The PAPI library is a cross-platform abstraction layer and so the read call involves additional instructions, memory accesses, and branches. In addition, reading counters on Linux traditionally involves using the `read()` system call which involves a relatively slow entry to the Linux kernel. This is essentially a software interrupt which brings the CPU to a halt, changes to privileged mode, branches to internal kernel code that does some housekeeping, reads the value from the CPU, ensures all buffers are valid, writes the results out to userspace, and then finally switches back to the original running program. All of this overhead can take hundreds to thousands of cycles, much higher than the tens of cycles needed for a raw counter read [24].

Much of this overhead can be avoided if we bypass the `read()` system call and read the counters directly from userspace, without involving the operating system at all. On x86 systems there is a special `rdpmc` instruction which allows exactly this. Setting up and using this instruction can be complex and it was not available in the initial perf_event release. Once the Linux kernel added support, PAPI's perf_event still lacked `rdpmc` support and used the `read()` interface. We extend PAPI to use the lower-overhead `rdpmc` interface and run a number of tests to evaluate the change in performance. We run on a wide variety of x86 machines and find a typical speedup of around six times when using the new interface. The work revealed four bugs in the low-level Linux interface, but we have gotten these fixed upstream. Due to our work, PAPI uses `rdpmc` by default as of the 5.6 release of the library.

2 Background

The concept of performance counters is straightforward: they are hardware counters that increment when certain architectural events happen on a processor. Gathering these results in a fast, efficient fashion involves complex interactions between the hardware, operating system, libraries, and applications.

2.1 Performance Counter Hardware

Hardware performance counters are configured by setting values in a series of special low-level CPU registers. On x86 machines these are called Model Specific Registers (MSRs) which are described in the vendor documentation [2, 9].

Recent x86 processors tend to have between four to seven counters per CPU, as can be seen in Table 1. This number can be affected by the existence of hardware multithreading. These counters are used to measure per-core architectural events such as cache behavior, branch predictor behavior, cycle and instruction counts, etc. Recent CPUs often have additional events, such as "uncore" and RAPL power measurement; these are measured by a different interface and cannot be accessed via the `rdpmc` interface we describe here.

To start measurement the desired events (from a list of potentially hundreds) are programmed into the event configuration registers. A bit is set in another configuration register to start the counting. The current values can be read out of the counter registers, typically from 40 to 48 bits in size. An interrupt can be configured for when the counter overflows; this allows both statistical sampling as well as keeping track of total event counts when they overflow.

2.2 Linux perf_event Interface

Access to performance counter registers requires supervisor level permissions; because of this the operating system is usually responsible for the interface. The operating system might further restrict access for security reasons, as a clever user can monitor in detail what a system is doing based on the fine grained performance information (one prime worry is being able to reverse engineer encryption happening on other cores by monitoring cycle or cache miss counts). The standard counter interface on Linux is known as `perf_event` and the primary way of accessing it is the `perf_event_open()` system call [25]. This system call is used to configure and open a performance counter event; it is a complex call with over forty interacting parameters. The system call returns a file descriptor which can be used to control and access the event. Values can be read with the `read()` system call, and memory can be set up with `mmap()` that allows both sampling to a circular buffer as well as gathering additional information about the event. Various `ioctl()` calls are used to start and stop the events. Advanced features, such as event scheduling, event multiplexing, and save/restore on context switch, are all provided by the interface.

2.3 PAPI Library

The PAPI performance library [16] is a cross-platform library designed to allow access to performance counters on a wide variety of machines. On current Linux machines PAPI uses the perf_event interface. Before perf_event became standard (in 2009 with the Linux 2.6.31 release) PAPI used the perfmon2 [6] and perfctr [21] interfaces (which required custom patching of your Linux kernel). perfctr in particular has extremely fast counter reads due to using the `rdpmc` call, something perf_event initially lacked.

2.4 Linux `rdpmc` support

The merging of perf_event into Linux was not without controversy. Due to the complaints from the PAPI developers about the high overhead of the `read()` system call, a userspace interface to allow fast `rdpmc` reads was eventually added with the Linux 3.4 release in 2012. An interface-breaking bug was found and fixed in the 3.11 release in 2013 [4] involving overlapping fields in a union which had unintentionally disabled some of the functionality. This was fixed, but this makes fully supporting both old and new kernels in a backwards compatible way tricky.

2.5 PAPI `rdpmc` Code

The `rdpmc` instruction itself only takes a short amount of time to run, on the order of a few tens of cycles [24]. Enabling userspace `rdpmc` support on x86 is simply a matter of the kernel setting a bit in the special `CR4` system register. After that, one might think access would be as simple as inserting `rdpmc` instructions into your code. However the complications of modern multi-tasking operating systems lead to a more complicated interface. Because there might be multiple users of perf_event, we cannot simply set counters to be free-running and use an assembly-language call to `rdpmc` to access them (this was a typical way to use `rdpmc` before perf_event was merged into Linux).

The recommended code for using `rdpmc` with perf_event is complicated, as seen in the example code found in Figure 1. This boilerplate code more than doubles the overhead of a read, on the order of a few hundred cycles. Despite this overhead, this code all runs in userspace, so it is still much faster than using the default `read()` interface which must go through the kernel.

The reason for the extra code is that PAPI needs to be sure that the event configuration has not been changed by the kernel since the last time the event was read. The kernel is free to rearrange event counter mappings at any time. This might happen on a context switch, or due to multiplexing.

Multiplexing is when the kernel allows adding more events than the physical number available, providing estimated total event counts as if the hardware had that many counters. This is done by periodically stopping the counters and swapping in ones currently not running, so all events have a turn to run. The

```c
do {
   /* The kernel increments pc->lock any time */
   /* perf_event_update_userpage() is called  */
   /* So by checking now, and the end, we     */
   /* can see if an update happened while we  */
   /* were trying to read things, and re-try  */
   /* if something changed                    */
   /* The barrier ensures we get the most     */
   /* up-to date version of pc->lock          */

   seq=pc->lock;
   barrier();

   /* For multiplexing */
   /* time_enabled: time the event was enabled */
   enabled = pc->time_enabled;
   /* time_running: time the event was */
   /* actually running */
   running = pc->time_running;

   /* if cap_user_time is set we can use rdtsc */
   /* to calculate more exact enabled/running  */
   /* for more accurate multiplex calculations */
   if ( (pc->cap_user_time) &&
         (enabled != running)) {
      cyc = rdtsc();
      time_offset = pc->time_offset;
      time_mult = pc->time_mult;
      time_shift = pc->time_shift;

      quot = (cyc>>time_shift);
      rem = cyc & (((uint64_t)1<<time_shift)-1);
      delta = time_offset + (quot * time_mult) +
         ((rem * time_mult) >> time_shift);
   }
   enabled+=delta;

   /* Index of register to read               */
   /* 0 means stopped/not-active              */
   /* Need to subtract 1 to get rdpmc() index */
   index = pc->index;

   /* count is the value of the counter the    */
   /* last time the kernel read it.            */
   /* If we don't sign extend, we get negative */
   /* numbers which break if IOC_RESET is done */
   width = pc->pmc_width;
   count = pc->offset;
   count<<=(64-width);
   count>>=(64-width);

   /* Only read if rdpmc enabled and index     */
   /* valid, otherwise return the older count  */
   if (pc->cap_usr_rdpmc && index) {

      /* Read counter value */
      pmc = rdpmc(index-1);

      /* sign extend result */
      pmc<<=(64-width);
      pmc>>=(64-width);

      /* add value into existing kernel count */
      count+=pmc;
      running+=delta;
   }

   barrier();

} while (pc->lock != seq);

if (en) *en=enabled;
if (ru) *ru=running;

return count;
```

Fig. 1. Sample code for a perf_event `rdpmc` read.

time an event has actually spent running is tracked, so by scaling this based on the total time you can estimate how many counts would have happened if the event had been running the full time. Multiplex handling is a big part of the extra `rdpmc` measurement code, as due to multiplexing the events currently scheduled might be changed by the operating system at any time. Also, before reporting the final event counts, you need to scale any events that did not run for the full time during measurement.

The perf_event interface provides helper information that can be mapped into the program's address space with a call to `mmap()`. Each event you want to read via `rdpmc` must have an associated `mmap()` page. This potentially adds overhead issues: the `read()` interface allows grouping multiple events so they can be read with one single call. However with `rdpmc` each event needs to be read individually and with large numbers of events this could potentially hurt performance. In addition each `mmap()` page takes up a valuable TLB slot and could hurt performance if a large number of events are mapped. On architectures with large page sizes events can take up large amounts of RAM, which can be troublesome since by default the amount of mmap area that perf_event can pin into memory is limited to 516kB.

A `rdpmc` read involves the following series of events. First, the `seq` sequence field is read, followed by a memory barrier to make sure it is synchronized with the kernel. Next, check `time_running` and `time_enabled`. If they are equal then multiplexing is not happening, otherwise the result needs to be scaled appropriately. The `count` value (which needs to be sign extended) holds the value from the last time the kernel has read the counter. This needs to be accounted for, as the value in the actual counter might have been reset on context switch, CPU migration, or if an overflow happened. Finally use `rdpmc` to obtain the current counter value which is added to `count`. While all of this is happening various things could happen that would make the values inconsistent (such as a context switch). To verify this has not happened, the `seq` value should be read again to verify it matches the earlier value. If this has changed then the whole process needs to be repeated until we complete the process without a change. From our experiments we find it is rare for `seq` to change unless the system is under heavy load. A livelock could potentially happen where the sequence checking could never make progress if the kernel is busy updating the page. Code could be added to break out and fall back to a `read()` in this situation.

This code path may seem like it has a lot of overhead, but it it still much faster than performing a `read()` system call (which is slow, disruptive to the CPU, and involves running an unpredictable amount of kernel code).

This code has been added to PAPI and is enabled by default in the 5.6 release of the library. Use the `--enable-perfevent-rdpmc=yes/no` configure option to explicitly enable or disable the feature when building and installing.

2.6 Linux rdpmc Bugs Found

Once we started testing the rdpmc code in PAPI, the PAPI regression tests turned up a number of bugs. After some analysis, most of these bugs were found to be in the Linux kernel implementation.

The first bug found was that various pthread tests would randomly cause general protection faults (GPF) and crash. This is due to a change made in the Linux 4.0 kernel that disabled rdpmc support when a process had no events running. Prior to this, when perf_event was started the CR4 bit that enables rdpmc support was globally enabled, so even processes without active events could still read the counter values. This is a possible information leakage security issue, so the kernel was modified to only allow using rdpmc if a process was actively using an event. There was a bug in the implementation of this fix: a wrong field was checked and sometimes when multiple threads were active the reference count would get out of sync and rdpmc support would be disabled while events were still running, leading to a GPF. This bug was reported by us and fixed in the Linux 4.12 release.

Another related bug happened when a process created a perf_event mmap mapping, but then called the exec() system call without closing the mapping first. This would cause the mmap reference count to go negative and again GPFs would happen on rdpmc access. This bug was reported by us and fixed in the Linux 4.13 release.

Another test that failed was one that created a large number of events in a large number of threads. This was a kernel limitation: the number of mmap() pages is limited by the value in sysctl kernel.perf_event_mlock_kb to a default of 516kB. We were hitting this limit and PAPI was crashing. We modified PAPI to only use 1 mmap page per process when using rdpmc (except when sampling), and if mmap space runs out it will now fall back to using read() which is slower but should always work.

The final bug involves time accounting when attaching to another process. With perf_event it is possible for one process to monitor another by specifying a process id at event creation time (this is how tools like perf can monitor a separate process). The enabled_time accounting code did not handle the case where an event was disabled while the attached processor was asleep, leading to the value being reported as negative. PAPI saw the non-matching enabled and running times and assumed this was a multiplexed event and scaled the results accordingly leading to impossibly large values. This bug was reported by us and fixed in the Linux 4.13 release.

3 Related Work

Low-overhead counter access is an important area with a lot of previous research. PAPI is widely used and is often the comparison point for such studies.

3.1 Lower-Level Interface Overhead

Prior to the introduction of perf_event with the 2.6.31 Linux kernel, there were external patches to provide performance counter support to Linux. PAPI used two of these: perfctr [21] (which had `rdpmc` support) and perfmon2 [6] (which did not). Most previous PAPI comparisons predate the introduction of perf_event and use one of these interfaces. These results are out of date now, as work on the alternate interfaces stopped once perf_event was merged into the mainline Linux kernel.

We [26] previously investigated the overhead of perf_event in terms of start / stop / read overhead on various x86_64 machines. The measurements are at the raw system call level, one level lower than the PAPI interface we investigate. We found that perf_event `read()` has relatively high overhead, but that the perf_event `rdpmc` interface could be competitive with the previous perfctr and perfmon2 interfaces.

3.2 PAPI Overhead

Our work, as well as much of the previous work, primarily looks at the effect in cycle time when adding instrumentation. Instrumentation can affect other metrics, and the reduced overhead from `rdpmc` should help in these cases too.

Maxwell et al. [12] and Moore et al. [15] compare the overhead of PAPI, including read calls, on various architectures available in 2002. This predates perf_event so making direct comparisons to our work is difficult.

Lehr [10] finds that even though PAPI instrumentation causes less than a 10% slowdown in SPEC CPU 2006, the actual counter measurements (including stores and cache events) can be perturbed enough to give misleading results.

Huang et al. [8] investigate the power overhead of using PAPI. This is not directly related to our work, but any/time instruction overhead is also going to lead to a certain amount of power and energy overhead.

Babka and Tůma [3] investigate the overhead of PAPI in both cycle count and other metrics on AMD and Intel machines. Their primary concern is overhead of memory metrics. Their measured overhead is high, as it appears they were using perfmon2. Using a `rdpmc` capable interface would reduce the overhead.

Zaparanuks, Jovic and Hauswirth [28] investigate measurement overhead of both user and user+kernel counters using PAPI on top of perfmon2 and perfctr, as well as using perfmon2 and perfctr directly. It is a detailed investigation into obtaining minimum overhead on these interfaces, but predates the introduction of perf_event.

3.3 Other Performance Counter Tools

Röhl et al. [22] investigate the performance of likwid-perfctr and the LIKWID Marker API under the Linux OS on Intel IvyBridge-EP, Intel Haswell and AMD Interlagos. At the time LIKWID did not support the perf_event interface, and instead directly accesses the relevant MSRs using the Linux `/dev/msr` interface.

Table 1. Machines used in this study. Note that on Intel machines more counters may be available if hyperthreading is disabled.

Processor	Counters Available
Intel Pentium II	2 general
Intel Pentium 4	18 general
Intel Core 2 P8700	2 general 3 fixed
Intel Atom Cedarview D2550	2 general 3 fixed
Intel IvyBridge i5-3210M	4 general 3 fixed
Intel Haswell i7-4770	4 general 3 fixed
Intel Haswell-EP E5-2640	4 general 3 fixed
Intel Broadwell i7-5557U	4 general 3 fixed
Intel Broadwell-EP E5-2620	4 general 3 fixed
Intel Skylake i7-6700	4 general 3 fixed
AMD fam10h Phenom II	4 general
AMD fam15h A10-6800B	6 general
AMD fam15h Opteron 6376	6 general
AMD fam16h A8-6410	4 general

Using `/dev/msr` still requires entry/exit from the kernel so can still have high overhead. The Marker API allows calipered measurement of code, although it is not full self-monitoring as the values measured are written straight to disk without the running application having access. They find that moving to `rdpmc` would greatly reduce overhead, but since the kernel disables `rdpmc` by default if not using perf_event, they cannot use it without patching the kernel. They compare their results to PAPI, but do not break out the read overhead separately. LIKWID does show an advantage over PAPI in their results, but this was before our addition of `rdpmc` support.

Demme and Sethumadhaven propose LiMiT [5], a Linux interface to provide fast, userspace access to performance counters reminiscent of the much older perfctr project. It requires patching the Linux kernel, and a note on the project's website notes that the patch is unstable and can cause system crashes. They claim LiMiT is 90x faster than PAPI and 23x faster than perf_event, although the test is not described in detail nor what kernel versions used for the test so it is a bit unclear what is being compared. The addition of `rdpmc` support to PAPI should make it compare more favorably since pure userspace accesses are being used.

AMD proposed an advanced Lightweight Profiling [1] interface providing userspace-only access to all aspects of controlling performance counters, not just reads. This could potentially speed up much more than reads, however the Linux kernel developers have refused to add support for the interface unless it was moderated by the kernel, which would defeat the entire purpose [14].

4 Experimental Setup

We test on fourteen different machines as shown in Table 1. This covers multiple generations of Intel and AMD processors from a 20 year old Pentium II machine up to and including more modern machines. Most machines are running the Linux 4.9 kernel provided with the Sid release of Debian Linux. A few of the machines are running the 3.16 kernel provided with Jessie Debian Linux. A full list of operating system, compiler, and cpu information is available for download along with our raw measurement information.

Most of our experiments are against a PAPI development git snapshot from March 2017, as at that time no full PAPI release contained `rdpmc` support. For comparison we also look at the 5.4.0, 5.4.1, 5.4.3, 5.5.0, and 5.5.1 official PAPI releases.

We measure the overhead of the core PAPI calls using the `papi_cost` utility that comes with PAPI. This runs each PAPI library call of interest one million times, measuring the latency using `PAPI_get_real_cyc()`. On x86 systems this maps to a `rdtsc` read timestamp instruction. We extend `papi_cost` to also return the median and 25th and 75th percentile values so that we could use those to make boxplots. For the more complicated results, such as the outlier analysis, we modify `papi_cost` further to log performance counter data for each iteration. In addition, we instrument the STREAM [13] and Linpack [20] benchmarks to investigate how the `PAPI_read()` overhead changes when a system is under load.

5 Results

We compare the overhead for traditional PAPI using `read()` to our modified PAPI using the `rdpmc` instruction.

Table 2 summarizes the `read()` vs `rdpmc` speedup found on the fourteen x86 machines. The results are given based on the median out of 1 million consecutive calls to read. We use the median, and not the average, as the measurement code occasionally has extremely large outliers which skew the average and standard deviation. See Section 5.1 for more discussion of these outliers. The speedup found is at least 2.6x in all cases, and is typically around 6x on recent Intel machines. This speedup is still large, but not quite as high on AMD machines and low end machines such as the Atom processors.

Figure 2 shows the `PAPI_read()` overhead gathered for the past few PAPI releases, as well as the current git snapshot we use for testing. This was mostly a sanity check to make sure the values have not changed greatly over time. The plots are boxplots: the black box shows the range between the 25th and 75th percentiles, the white line is the median, and the lines are showing the maximum outliers. Since the outliers are large, we zoom in on the plot and label at the top of the graph their numerical value. It can be seen that the overhead has not changed much in the recent past on the Haswell machine that we measure on.

By default the `papi_cost` benchmark measures two events. That is a typical number to measure, especially if you are interested in metrics such as Instruction

Table 2. Median rdpmc speedup in `papi_cost` running the read test 1 million times.

Vendor	Machine	read() cycles	rdpmc cycles	Speedup
Intel	Pentium II	2533	384	6.6x
Intel	Pentium 4	3728	704	5.3x
Intel	Core 2	1634	199	8.2x
Intel	Atom	3906	392	10.0x
Intel	Ivybridge	885	149	5.9x
Intel	Haswell	913	142	6.4x
Intel	Haswell-EP	820	125	6.6x
Intel	Broadwell	1030	145	7.1x
Intel	Broadwell-EP	750	118	6.4x
Intel	Skylake	942	144	6.5x
AMD	fam10h Phenom II	1252	205	6.1x
AMD	fam15h A10	2457	951	2.6x
AMD	fam15h Opteron	2186	644	3.4x
AMD	fam16h A8	1632	205	8.0x

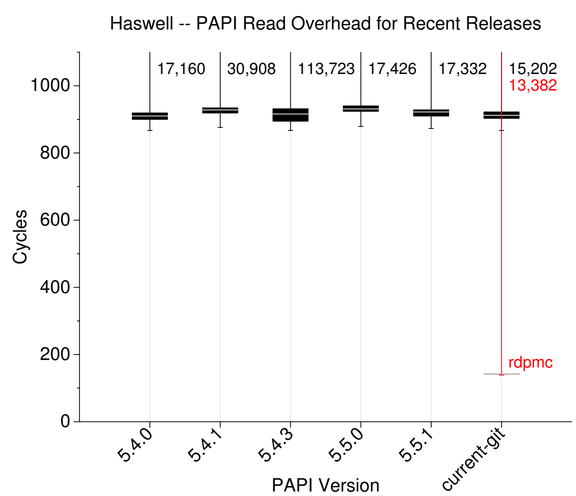

Fig. 2. Boxplot comparison of read overheads for the past few releases of PAPI.

per Cycle (IPC). To get a wider range of results we modify `papi_cost` to measure from one to four events. Figure 3 shows how the overhead increases on a Haswell machine. Both the `read()` and `rdpmc` results increase, but the increase is linear as expected.

The `read()` code uses the perf_event format group feature to read multiple events with a single system call. Despite grouping multiple events into on system call, the time still grows linearly as the internal kernel code still has to read the counters out one by one. The `rdpmc` code must read out the results one by one,

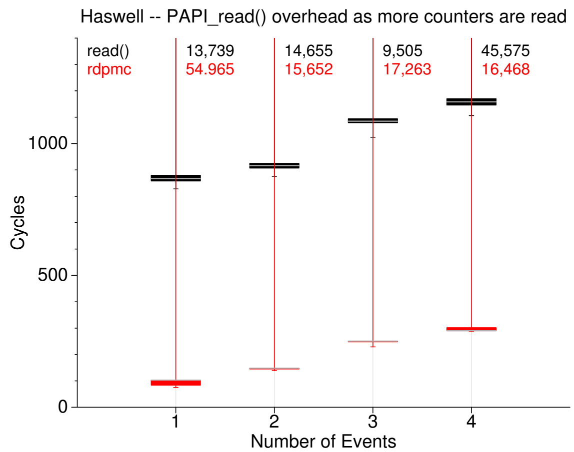

Fig. 3. Boxplot comparison of read overheads as more simultaneous events are measured.

Table 3. Results under load. Note: the cycle counter cycles aren't necessarily the same as rdtsc cycles.

Routine	Type	Cycles		L1 DMiss		DTLB Miss	
		User	Kernel	User	Kernel	User	Kernel
HPL_pdpanel_init (low memory pressure)	rdpmc	512	0	5	0	0	0
	read()	461	1,755	7	20	0	0
HPL_pdfact (high memory pressure)	rdpmc	4,019	0	39	0	11	0
	read()	4,551	13,545	43	123	16	16

Table 4. TLB misses for various number of simultaneous events. When using `rdpmc` more mmap pages are used, which could potentially increase the TLB pressure on a memory-intense workload.

Routine	Type	2 Events		3 Events		4 Events	
		User	Kernel	User	Kernel	User	Kernel
HPL_pdpanel_init (low memory pressure)	rdpmc	0	0	0	0	0	0
	read()	0	0	0	0	0	0
HPL_pdfact (high memory pressure)	rdpmc	11	0	14	0	16	0
	read()	16	16	15	17	16	18

with the additional overhead from the fixup code for each read. There has been an interface suggested [29] that would allow grouping multiple events into one mmap() page but this interface has not been implemented yet.

In addition to the `papi_cost` results, which only look at overhead when doing `PAPI_read()` calls and nothing else, we also investigate overhead found in more real-world situations. We look at the architectural overhead of the `PAPI_read()`

call. This is difficult, as the traditional way of gathering such measurements would be to use PAPI, but using PAPI to measure PAPI does not work well. Instead we put raw calls to `rdpmc` around the `PAPI_read()` calls under the assumption that for such short time intervals it is unlikely that the kernel will move events around.

Table 3 shows results for the overhead of `PAPI_read()` while instrumenting two different Linpack functions: `HPL_pdpanel_init()` and `HPL_pdfact()`. The former does not access memory much, and so the cycle count, L1 misses, and TLB misses are low. (Note that the cycle counts reported here are CPU cycles, which are not the same as the rdtsc bus cycles reported for other results in this paper). The `rdpmc` results show that the kernel is not entered at all, and that some of the `read()` overhead is caused by cache misses when running kernel code. The `HPL_pdfact()` routine is memory intensive, so the addition of `PAPI_read()` to the code causes cache and TLB misses which generate a lot more overhead than when the same routine is added to `HPL_pdpanel_init()`. In both cases the `rdpmc` version of `PAPI_read()` has much lower overhead overall.

Table 4 investigates the same routines as more events are being measured by `PAPI_read()`. This is to see if the additional mmap pages required by the rdpmc interface cause enough TLB pressure to adversely affect the measured overhead. While the TLB misses do grow, overall they are still less than for the `read()` version of the code.

5.1 Outliers

Our overhead results mostly cluster around the median, but there are occasional outliers of over an order of magnitude. We initially suspected the `rdtsc` cycle measurements, but on newer x86 processors the cycle counter has had many improvements to make it invariant in the face of frequency scaling. PAPI follows most of the suggestions by Intel for how to obtain accurate cycle readings [19].

An example of the magnitude of the outliers can be seen in Figure 4 which shows the overhead of the first 3000 `rdpmc` reads in a `papi_cost` run. We use the performance counter results to determine the source of the outliers. For these results we are using an AMD A10 machine as it has a richer set of events to choose from (including a hardware interrupt event and a SMI system monitoring interrupt event). We find that many of the extreme outliers (but not all of them) are caused by a hardware interrupt happening in the middle of a read.

There are also some interesting recurring patterns every 500 reads or so. Figure 5 plots a different run, this time showing L2 cache misses. We observe L2 cache misses are happening approximately every 500 iterations. The benchmark, outside of the critical measurement loop, stores the gathered values (which are 64-bit integers) to a large array for later analysis. If you write 512 8-byte values to memory, that works out to be 4096 bytes, which is the size of a page. So our measurement code is potentially causing a TLB or cache miss when crossing a page boundary which is likely the cause of that regular pattern.

The outlier immediately at the beginning on both plots is caused by a page-fault and TLB miss the first time the mmap page is accessed. We noted this

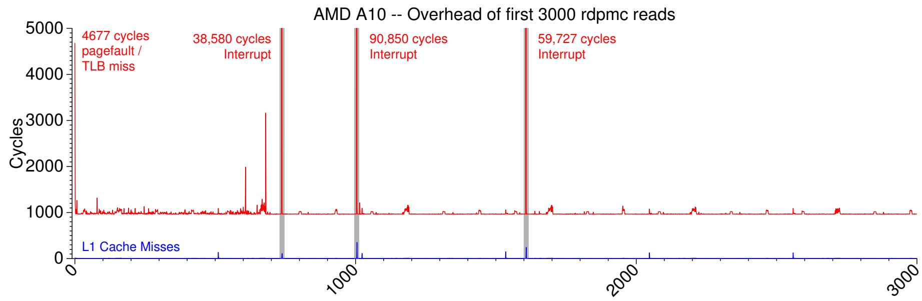

Fig. 4. Overhead seen in the first 3000 iterations of a rdpmc `papi_cost` run. The larger outliers are caused by hardware interrupts, while the initial is caused by a pagefault from the first access to the mmap() page.

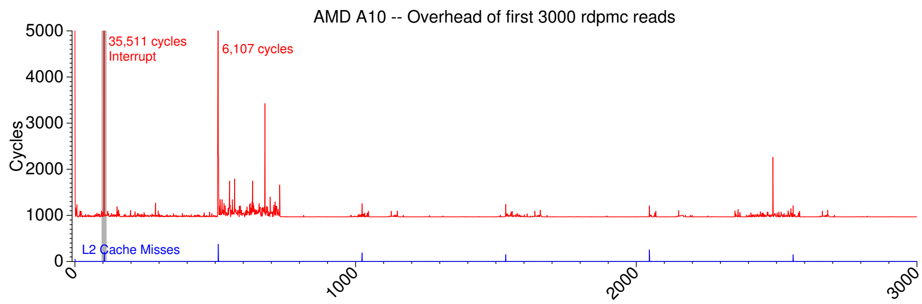

Fig. 5. Overhead seen in the first 3000 iterations of a different rdpmc `papi_cost` run. This plots L2 instead of L1 cache misses. There is a repeating pattern approximately every 500 iterations, likely caused by accesses to our results array (512 64-bit writes will fill one 4096 byte page).

previously [26], and suggested using `MAP_POPULATE` or touching the mmap page to avoid this issue. However, in Figure 5 we tried enabling `MAP_POPULATE` and it did not help. The initialization of the event happens so far in advance of the first read that by the time it gets to our read code the page is no longer in the TLB so preloading does not help. This behavior is probably typical of what would be found in most PAPI instrumented code. This page-fault issue means that if you are using PAPI to do a single read, the first `rdpmc` overhead is large. However when using `read()` the first-access overhead is high for other reasons (including shared-library setup if you are the first user of the system call) so `rdpmc` is still better. In both cases, if more than one read is done, the initial first read overhead is mitigated.

5.2 Historical Comparison

Table 5 and Figure 6 show a comparison of the performance interfaces historically supported by PAPI on Linux. The results are on a Core 2 machine, as the older

Table 5. Comparison of various historical perf counter interfaces on a Core 2 machine. Core 2 is used as the older interfaces do not support more modern CPUs.

Interface	Kernel	Read results	slowdown vs perf_event `rdpmc`
perf_event `rdpmc`	3.16	199	—
perfctr `rdpmc`	2.6.32	200	1.0x
perfmon2 `read()`	2.6.30	1216	6.1x
perf_event `read()`	3.16	1587	8.0x
perf_event KPTI `read()`	4.15-rc7	3173	15.9x

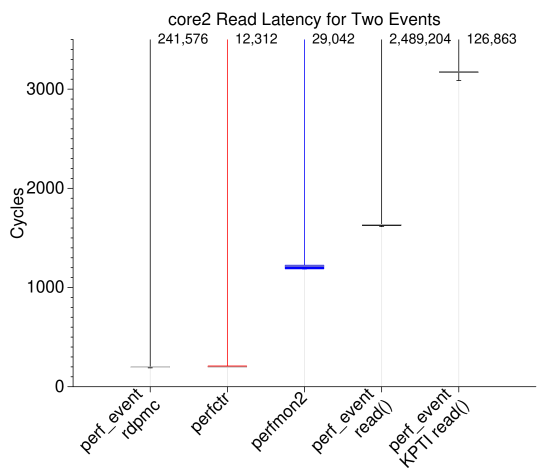

Fig. 6. Boxplot comparison of historical PAPI methods of doing reads. The few outliers are large, off the graph.

Table 6. Overhead caused by the KPTI workaround for the Meltdown security vulnerability found on Intel processors.

Processor	rdpmc	KPTI=off read	KPTI=on read
Core2	199	1634 (8.2x)	3173 (15.9x)
Haswell	139	958 (6.9x)	1411 (10.2x)
Skylake	142	978 (6.9x)	1522 (10.7x)

interfaces do not support more modern CPUs as they are no longer maintained now that perf_event became standard with Linux 2.6.31. The perfctr interface has a custom `rdpmc` interface that is similar to the one used by perf_event, whereas perfmon2 does not have a `rdpmc` interface. We find that the perf_event `rdpmc` interface is more or less the same speed as perfctr and much faster than perfmon2 and perf_event `read()`. It appears that after a many year absence, PAPI read overhead can finally return to the levels that were seen back when perfctr was the primary method of accessing performance counters.

One additional change to recent Linux has affected these results. The release of the Meltdown security vulnerability [11] on Intel processors has led to the Kernel Page Table Isolation (KPTI) patchset being enabled by default. This moves the kernel and user address spaces to be completely different, causing a costly TLB flush on every system call. We measure the overhead caused by this and indeed the read() overhead is much larger, as seen in Table 6.

6 Conclusion and Future Work

We have added userspace (`rdpmc`) performance counter read support to the PAPI library and found that we can reduce overhead by at least three times (and more typically around six times) on a wide variety of x86 hardware. We have validated the results, which resulted in finding and getting fixed a number of bugs in the Linux kernel. We also investigated and found the source of the large outliers in the results (found on all interfaces and machines) that make analysis of timing results difficult.

Our results provide sufficient evidence that the perf_event rdpmc interface consistently has less overhead than the `read()` interface, and we have enabled the new interface in PAPI by default as of the 5.6 release. This allows PAPI to once again obtain low-overhead performance counter data via `rdpmc`, a feature that had been lost when the perfctr interface was abandoned with the introduction of the Linux perf_event component. We plan to investigate adding userspace read support on other architectures that support it, most notably the ARM and ARM64 architectures.ARM64 has a `rdpmc` alike interface, but currently the Linux kernel does not support it. If support is added in a perf_event compatible way then PAPI should be able to use the interface with minimal changes.

Full data for the work presented in our paper can be downloaded from our website: `http://web.eece.maine.edu/~vweaver/projects/papi-rdpmc/`

The reduced overhead provided by `rdpmc` should greatly help users of PAPI, especially those in the high performance computing community. Performance analysis will be greatly aided by the detailed performance results obtained with less overhead than was recently possible.

Acknowledgment

This work was supported by the National Science Foundation under Grant No. SSI-1450122.

References

1. Advanced Micro Devices: Lightweight Profiling Specification (2010)
2. AMD: AMD Family 15h Processor BIOS and Kernel Developer Guide (2011)
3. Babka, V., Tůma, P.: Effects of memory sharing on contemporary processor architectures. In: Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. pp. 15–22 (2007)

4. Corbet, J.: A perf ABI fix. Linux Weekly News (Sep 2013), https://lwn.net/Articles/567894/
5. Demme, J., Sethumadhavan, S.: Rapid identification of architectural bottlenecks via precise event counting. In: Proc. 38th IEEE/ACM International Symposium on Computer Architecture (Jun 2011)
6. Eranian, S.: Perfmon2: a flexible performance monitoring interface for Linux. In: Proc. 2006 Ottawa Linux Symposium. pp. 269–288 (Jul 2006)
7. Gleixner, T., Molnar, I.: Performance counters for Linux (2009)
8. Huang, S., Lang, M., Pakin, S., Gu, S.: Measurement and characterization of Haswell power and energy consumption. In: Proc. of the 3rd International Workshop on Energy Efficient Supercomputing (Nov 2015)
9. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide (Jun 2015)
10. Lehr, J.: Counting performance: hardware performance counter and compiler instrumentation. In: Jahrestagung der Gesellschaft für Informatik. LNI, vol. P-259, pp. 2187–2198. GI (Sep 2016)
11. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. ArXiv e-prints (Jan 2018)
12. Maxwell, M., Teller, P., Salayandia, L., Moore, S.: Accuracy of performance monitoring hardware. In: Proc. Los Alamos Computer Science Institute Symposium (Oct 2002)
13. McCalpin, J.: STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/ (1999)
14. Molnar, I.: Re: [RFC 0/3] basic support for LWP. http://marc.info/?l=linux-kernel&m=128630554614635 (2010)
15. Moore, S., Teller, P., Maxwell, M.: Efficiency and accuracy issues for sampling vs counting modes of performance monitoring hardware. In: Proc. of the DoD High Performance Computing Modernization Program's User Group Conference (Jun 2002)
16. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: A portable interface to hardware performance counters. In: Proc. Department of Defense HPCMP User Group Conference (Jun 1999)
17. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.: Understanding measurement perturbation in trace-based data. In: Proc. 21st IEEE/ACM International Parallel and Distributed Processing Symposium (Mar 2007)
18. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.: We have it easy, but do we have it right? In: Proc. 22nd IEEE/ACM International Parallel and Distributed Processing Symposium (Apr 2008)
19. Paoloni, G.: How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Intel Corporation (Sep 2010)
20. Petitet, A., Whaley, R., Dongarra, J., Cleary, A., Luszczek, P.: HPL — a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Innovative Computing Laboratory, Computer Science Department, University of Tennessee, v2.2, http://www.netlib.org/benchmark/hpl/ (Dec 2017)
21. Pettersson, M.: The perfctr interface. http://user.it.uu.se/~mikpe/linux/perfctr/2.6/ (1999)
22. Röhl, T., Treibig, J., Hager, G., Wellein, G.: Overhead analysis of performance counter measurements. In: International Conference on Parallel Programming Workshops. pp. 176–185 (Sep 2014)

23. Treibig, J., Hager, G., Wellein, G.: 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)
24. Weaver, V.: [patch] perf_event use rdpmc rather than rdmsr when possible in kernel. https://lkml.org/lkml/2012/2/20/418 (2012)
25. Weaver, V.: perf_event_open manual page. In: Kerrisk, M. (ed.) Linux Programmer's Manual (Dec 2013)
26. Weaver, V.: Self-monitoring overhead of the linux perf_event performance counter interface. In: Proc. IEEE International Symposium on Performance Analysis of Systems and Software (Mar 2015)
27. Wolf, J.: Programming Methods for the Pentium™ III Processor's Streaming SIMD Extensions Using the VTune™Performance Enhancement Environment. Intel Corporation (1999)
28. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: Proc. IEEE International Symposium on Performance Analysis of Systems and Software. pp. 23–32 (Apr 2009)
29. Zijlstra, P.: Re: [patch 1/2] perf/x86/intel: enable cpu ref_cycles for gp counter. https://marc.info/?l=linux-kernel&m=149616517431438&w=2 (May 2017)