Accuracy of Performance Counter Measurements

Dmitrijs Zaparanuks
Faculty of Informatics
University of Lugano
Email: zaparand@lu.unisi.ch

Milan Jovic
Faculty of Informatics
University of Lugano
Email: jovicm@lu.unisi.ch

Matthias Hauswirth
Faculty of Informatics
University of Lugano
Email: Matthias.Hauswirth@unisi.ch

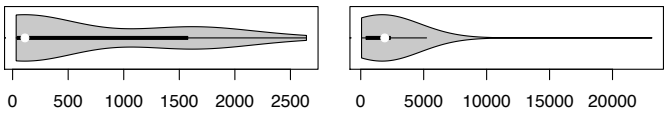
Abstract

Many experimental performance evaluations depend on accurate measurements of the cost of executing a piece of code. Often these measurements are conducted using infrastructures to access hardware performance counters. Most modern processors provide such counters to count micro-architectural events such as retired instructions or clock cycles. These counters can be difficult to configure, may not be programmable or readable from user-level code, and can not discriminate between events caused by different software threads. Various software infrastructures address this problem, providing access to per-thread counters from application code. This paper constitutes the first comparative study of the accuracy of three commonly used measurement infrastructures (perfctr, perfmon2, and PAPI) on three common processors (Pentium D, Core 2 Duo, and AMD ATHLON 64 X2).

1. Introduction

No serious time keeper would time a competitive 100m sprint using a stop watch: The short duration of the race (below 10s) and the miniscule differences between runners (1/100s) rule out stop watches as an accurate measurement tool. On the other hand, a stop watch might be an appropriate tool for timing a marathon race, with durations above 2 hours, and differences between runners above 1s. In our discipline, the quantitative evaluation of computer system performance, the athletic marathon corresponds to measuring the end-to-end behavior over an entire benchmark execution (billions of instructions), whereas the sprint corresponds to measuring the behavior of short segments of code (hundreds or thousands of instructions). Such short segments may correspond to a set of relevant functions, system calls or signal handlers, optimization phases in just-in-time method compilations, individual garbage collection phases, time spent in spin-locks, or application-level program phases.

Performance analysts often use the hardware performance counters available on many modern processors for measuring duration (cycle counts), instruction count, or other microarchitectural event counts. These counters can be difficult to configure, may not be programmable or readable from user-level code, and can not discriminate between events caused by different software threads. Various software infrastructures address these problems, providing access to per-thread counters from application code. However, such infrastructures incur a cost: the software instructions executed to access a counter, or to maintain a per-thread counter, may perturb that same counter. While this perturbation is presumably small, its significance depends on the specific measurement: it may be irrelevant for a marathon-length measurement, but it could significantly perturb the measurement of a sprint.



(a) User mode (b) User+kernel mode

Figure 1. Number of overcounted instructions

This paper provides a comparative study of the accuracy of three commonly used measurement infrastructures on three common processors. **Figure 1** highlights the magnitude of the possible measurement error. The figure consists of two violin plots[1] [1]. The x axes show the measurement error as the number of superfluous instructions executed and counted due to the measurement infrastructure. Following our athletic anology, this error corresponds to the time between a sprinter crossing the finish line and the time keeper pushing the button of his stop watch. Each violin summarizes the error of over 170000 measurements performed on a large number of different infrastructures and configurations. In the left violin plot, the error only includes instructions executed in user-mode, while the right plot shows user plus kernel mode instructions. While the minimum error is close to zero, a significant number of configurations can lead to errors of 2500 user-mode instructions or more. When counting user and kernel mode instructions, we observed configurations with errors of over 10000 instructions. These plots show that it is crucial to select the best measurement infrastructure and configuration to get accurate short-duration measurements.

1. A violin plot corresponds to a box plot that additionally shows (in gray) the density of the sample's distribution.

The remainder of this paper is structured as follows: Section 2 provides background information on hardware performance counters and on infrastructures that access them. Section 3 describes our evaluation methodology. The following three sections present our results. Section 4 evaluates the measurement error due to counter accesses. Section 5 studies how the error depends on measurement duration. Section 6 demonstrates the problem of evaluating the accuracy of cycle counts. Section 7 discusses the limitations and threats to the validity of our study. Section 8 compares this study to related work, and Section 9 concludes.

2. Background

This section introduces the relevant background on hardware performance counters and the software used to configure and use them.

Hardware Performance Counters. Most modern processors contain hardware performance counters [2], special-purpose registers that can count the occurrence of micro-architectural events. Often these counters are programmable: They can be enabled or disabled, they can be configured to cause an interrupt at overflow, and they can be configured to count different types of events. Commonly supported events include the number of committed instructions, clock cycles, cache misses, or branch mispredictions. As processors differ in their micro-architectures, they necessarily differ in the type of countable events. The number of counter registers also differs greatly between different micro-architectures. In addition to programmable counters, some processors also support fixed-function counters which provide limited programmability (i.e. they always count the same event, or they cannot be disabled).

Configuring & Accessing Counters. Processors provide special registers to configure the hardware performance counters (e.g. to enable or disable a counter, or to determine which event to count). They also provide special instructions to access the counter and the counter configuration registers. For example, on processors supporting the IA32 instruction set architecture, the `RDPMC` instruction reads the value of a performance counter into general purpose registers, `RDTSC` reads the value of the time stamp counter (a special kind of fixed-function performance counter), and `RDMSR`/`WRMSR` read/write the value of any model-specific register (such the time stamp counter, a performance counter, or the registers used to configure the counters).

All the above mentioned IA32 instructions can be executed when in kernel mode. `RDMSR` and `WRMSR` are unavailable in user mode. Whether `RDPMC` and `RDTSC` work in user mode is configurable by software and depends on the operating system.

Per-Thread Counters. Hardware performance counters count events happening on a given processor[2]. The counter register does not distinguish between different software threads that run on its processor[3]. Performance analysts often need to know the number of events incurred by specific threads. To support this per-thread counting, the operating system's context switch code has to be extended to save and restore the counter registers in addition to the general purpose registers.

Software Support for Hardware Counters. The fact that some of the counter configuration or access instructions require kernel mode privileges, and the need to provide per-thread counts, have lead to the development of kernel extensions that provide user mode applications access to the counters. For Linux, the two frequently used kernel extensions are perfctr [3] and perfmon2 [4].

These kernel extensions are specific to an operating system. Thus, measurement code using these extensions becomes platform dependent. Moreover, even when using the same kernel extension, configuring counters for different processors requires processor-specific code. For this reason, many performance analysts use PAPI [5], a higher level API to access performance counters. PAPI provides a platform (OS and processor) independent programming interface. It achieves OS-independence by providing a layer of abstraction over the interface provided by kernel extensions that provide access to counters. It achieves processor-independence by providing a set of high level events that are mapped to the corresponding low-level events available on specific processors. PAPI also provides access to the machine-specific low-level events. To allow an even simpler programming model, PAPI provides a high level API that requires almost no configuration.

User and Kernel Mode Counting. Many processors support conditional event counting: they only increment a counter while the processor is running at a specific priviledge level (e.g. user mode, kernel mode, or either of the two). Whether a specific counter counts events that occur during user mode, kernel mode, or user+kernel mode, can be specified as part of that counter's configuration. Thus, if a counter is configured to count user mode events, as soon as the processor switches to kernel mode (e.g. due to a system call or an interrupt), it immediately stops counting.

In this paper we study the accuracy of event counts captured during user mode and of event counts captured during user+kernel mode. Depending on the type of performance analysis, analysts may be interested in only user-level counts, or they may want to include kernel-level counts. We do not study kernel-only event counts. Performance analysts who exclusively focus on kernel performance do not have to use the user level counter access infrastructures we evaluate in this paper.

2. On multi-core processors each core usually contains its own set of counters.

3. On cores supporting hyper-threading, some counters can be configured to count events of specific hardware threads.

Table 1. Processors used in this study

	Processor	GHz	μArch	Counters	
				fixed	prg.
PD	Pentium D 925	3.0	NetBurst	0+1	18
CD	Core2 Duo E6600	2.4	Core2	3+1	2
K8	Athlon 64 X2 4200+	2.2	K8	0+1	4

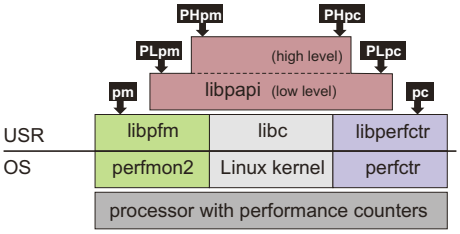


Figure 2. Counter access infrastructure

3. Experimental Methodology

In this section we present the methodology we use for our study.

Hardware. We use three state-of-the-art processors that all implement the IA32 instruction set architecture (ISA). They have significantly different micro-architectures and different performance counter support. **Table 1** shows the three processors, their micro-architectures, and the number of fixed and programmable counters they provide. The number of fixed counters includes the time stamp counter (TSC), which is specified in the IA32 ISA and is available on any IA32 processor.

Operating System. We run kubuntu Linux with kernel version 2.6.22. At the time of this writing, this kernel is the most recent kernel supported by both kernel extensions we study (perfmon2 and perfctr). To prevent the processor clock frequency from changing during our measurements, we disable frequency scaling by setting the Linux scaling governor to "performance". This causes the processors to continuously run at its highest frequency (the frequencies are shown in Table 1).

Counter Access Interfaces. Figure 2 presents the infrastructures we analyzed in this study. We created two patched versions of the Linux 2.6.22 kernel: one with the perfmon2 2.6.22-070725 kernel patch, and the other with the perfctr 2.6.29 patch. While these patched kernels allow user-level access to per-thread hardware counters, the protocol to interact with these kernel extensions is cumbersome. For this reason, both kernel extensions come with a matching user-space library that provides a clean API. The library supporting perfmon, libpfm, is available as a separate package. We use libpfm version 3.2-070725. The library supporting perfctr, libperfctr, is included in the perfctr package.

We use two builds of PAPI, one on top of perfctr, the other on top of perfmon. At the time of this writing, the released version 3.5.0 of PAPI does not build on all our configurations, and thus we use the most recent version of PAPI from CVS

```
__asm__ __volatile__("movl $0, %%eax\n"
                     ".loop:\n\t"
                     "addl $1, %%eax\n\t"
                     "cmpl $" MAX ", %%eax\n\t"
                     "jne .loop"
                     : : : "eax");
```

Figure 3. Loop micro-benchmark

(from "16 Oct 2007 0:00:00 UTC"). We build PAPI on top of libperfctr 2.6.29 and also on top of our version of libpfm 3.2-070725. PAPI provides two APIs for accessing counters. The low-level API is richer and more complex, while the high-level API is simpler.

Based on this infrastructure, we evaluate the accuracy of the six possible ways for accessing performance counters shown in Figure 2: directly through libpfm (pm), directly through libperfctr (pc), through the PAPI low-level API on top of libpfm (PLpm) or libperfctr (PLpc), or through the PAPI high-level API on top of libpfm (PHpm) or libperfctr (PHpc).

Benchmarks. In this paper we assess the *accuracy* of performance counter measurement approaches by comparing the measured counter values to the *true* event counts. How can we know the true counts? We could use accurate simulators of the respective processors, but no such simulators are generally available. For this reason we use micro-benchmarks, short pieces of code for which we can statically determine the exact event counts.

We want to measure two kinds of cost that affect measurement accuracy: the *fixed* cost of accessing the counters at the beginning and at the end of the measurement, and the *variable* cost of maintaining per-thread counts throughout the measurement. We expect the variable cost to change depending on the duration of the measurement.

To measure the fixed cost, we use a *null* benchmark, an empty block of code consisting of zero instructions. We know that this code should generate no events, thus any event count different from zero constitutes inaccuracy.

We measure the variable cost using the *loop* benchmark shown in **Figure 3**. We wrote this loop in gcc inline assembly language, so it is not affected by the C compiler used to generate the benchmark harness (the measurement code containing the loop). This code clobbers the `EAX` register, which it uses to maintain the loop count. The number of iterations is defined by the `MAX` macro at compile time and gets embedded in the code as an immediate operand. The loop takes 1 + 3 `MAX` instructions to execute.

Counter Access Patterns. Each interface in Figure 2 provides a subset of the following functions: *read* a counter, *start* a counter, *stop* a counter, and *reset* a counter. To read a counter, the infrastructure ultimately needs to use the `RDPMC` instruction. To start or stop a counter, the infrastructure enables or disables counting using `WRMSR`. Finally, the infrastructure can reset a counter by setting its value to 0

with WRMSR. Note that some of these instructions can only be used in kernel mode, and thus some functions incur the cost of a system call. Moreover, since the infrastructures we study support "virtualized" (per-thread) counters, the above functions require more work than just accessing or configuring a hardware register. We expect these differences to affect measurement accuracy.

The above functions allow us to use four different measurement patterns. We define these patterns in **Table 2**. All patterns capture the counter value in a variable (c_0) before starting the benchmark, then run the benchmark and capture the counter's value after the benchmark finishes in variable c_1. Thus, $c_\Delta = c_1 - c_0$ provides the number of events that occurred during the benchmark run.

Table 2. Counter access patterns

Pattern		Definition		
ar	start-read	c_0=0, reset, **start**	...	c_1=**read**
ao	start-stop	c_0=0, reset, **start**	...	**stop**, c_1=read
rr	read-read	start, c_0=**read**	...	c_1=**read**
ro	read-stop	start, c_0=**read**	...	**stop**, c_1=read

Not every interface supports all four patterns. In particular, the PAPI high-level API does not support the read-read and read-stop patterns, since its read function implicitly resets the counters after reading.

Compiler Optimization Level. To run our measurements, we embed the given benchmark code in a measurement harness (that is, we surround it with the library calls required by the given counter access pattern). We use gcc version 4.1.2 with the default options (except for the optimization level, which we explicitly specify) to compile the resulting C file. Because the benchmark code is written in gcc's inline assembly language, gcc does not optimize that code. However, gcc can optimize the surrounding measurement code. To determine the impact of the different compiler optimization levels on the measurement error, we compile the C file using each of the four optimization levels provided by gcc (O0 to O3).

4. Measurement Error

Figure 1 gave a high-level picture of the error caused by hardware counter accesses. Most notable is the significant variability of that error – the inter-quartile range amounts to about 1500 user-level instructions. In this section we study the factors that affect this error by measuring event counts for the *null* benchmark. We expect these counts to be zero (as there are no instructions in this benchmark), and we assume that every deviation from zero constitutes a measurement error.

4.1. Best Infrastructure and Pattern

We have described six different counter access infrastructures and four different counter access patterns. Which configuration should a performance analyst use to get the most accurate measurements? **Figure 4** breaks down the over 340000 measurements underlying the two violins of Figure 1 according to the most important factors. The left half of the figure shows the error when counting only user-mode events, and the right shows the error in user+kernel mode counts. Each half is broken down into four columns, one column for each counter access pattern. The figure is vertically partitioned into groups by the tool used to access the counters (high level PAPI, low level PAPI, or no PAPI) and the kernel extension used for counter virtualization (perfctr and perfmon). One group, perfctr without PAPI, is further subdivided: perfctr provides the option to also capture the TSC (time stamp counter) register along with the other performance counters, and given that this option significantly affects measurement error we include this additional factor. Moreover, we split each group into multiple rows, one for each machine (CD, K8, and PD). Because the read call of high-level PAPI resets the counters, the read-read and read-stop patterns do not apply to high-level PAPI, and the corresponding cells in the figure are empty. Each cell in the figure contains a chart showing the measurement error along the x-axis. The x-axis only extends to 4000 instructions to enable the visual comparison of the more desirable configurations (the configurations with smaller errors). This crops a few bars that range higher (the maximum error is 23124 instructions). Each individual bar summarizes at least 40 measurements[4]. Unlike a traditional bar chart, where bars range from 0 to some value, our bars do not start at 0, but they range from the minimal to the maximal error observed for that specific configuration. The label next to each bar shows the minimum and maximum value (the smallest and largest error observed for that configuration).

The figure shows that the measurement error differs drastically between configurations. Performance analysts interested in user mode counts should use perfmon with the read-stop or start-stop pattern (error of 32 to 41 instructions). If an analyst has to use perfctr, she should make sure the TSC is enabled and use the start-read pattern (55 to 136 instructions). Using PAPI instead of the perfctr or perfmon APIs incurs an extra cost: The best low-level PAPI configuration is to run on top of perfmon and to use start-read (134 to 135 instructions). High-level PAPI costs around 236 instructions.

When counting user+kernel mode events, the optimal configuration differs: perfctr with enabled TSC and the read-read pattern is the overall winner (64 to 226 instructions). Switching to perfmon incurs a significant cost: The best perfmon configuration is read-stop (492 to 4098 instructions). Low-level PAPI on top of perfctr makes use of perfctr's TSC optimization and thus gets a relatively low error with the read-read pattern (185 to 280 instructions). Finally, high-level

4. Each bar represents an aggregation of between 40 and 12800 measurements. Our experiments include additional factors (such as the number of enabled registers, or which of the enabled registers we retreived the measurement from). The number of levels of some of these factors varies between configurations. We conducted 5 measurements for each possible combination of levels.

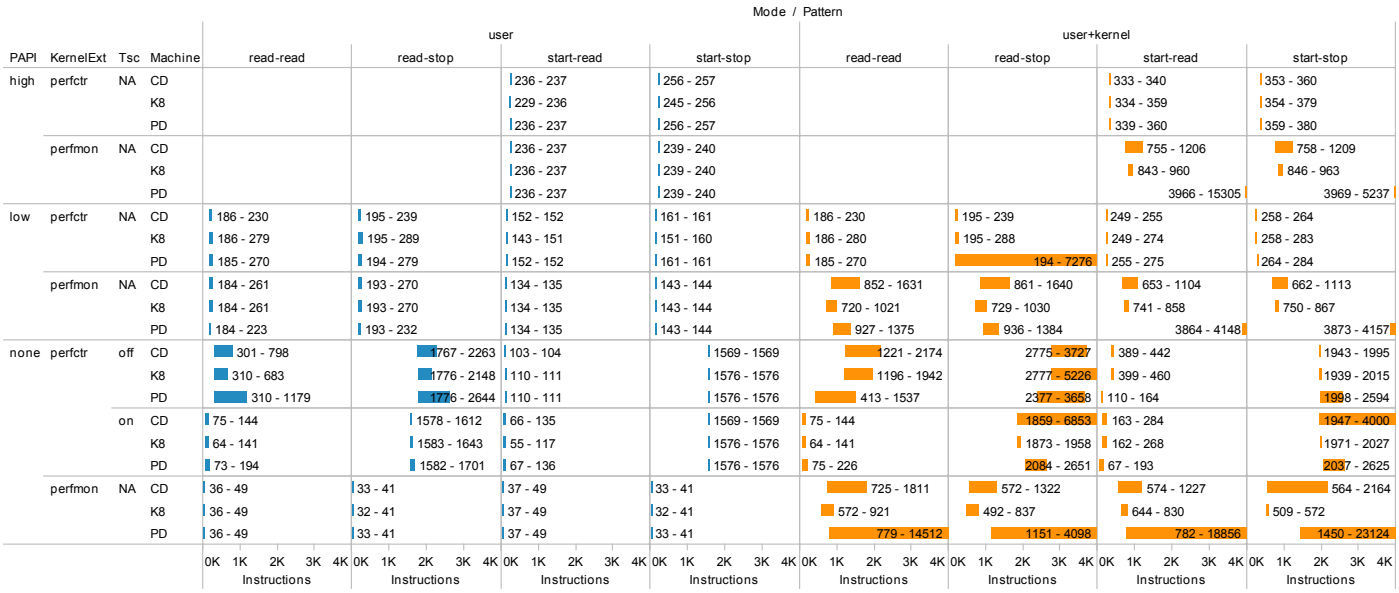


Figure 4. Error depends on infrastructure and pattern

PAPI is best used with start-read on top of perfctr (333 to 360 instructions).

Besides the above recommendations for which infrastructure to use in which situation, we would like to point out three interesting aspects.

(1) Difference between machines. A quick glance at the figure shows that in most cases the error for a given configuration is similar across all three machines. For configurations with differences between machines, the Pentium D (PD) is usually the outlier. To some degree this is due to its large number of performance counters on the Pentium D. Each additional enabled counter adds to the error (Section 4.2), and the bars for PD summarize measurements with up to eight enabled counters.

(2) Time stamp counter with perfctr. Perfctr enables the TSC by default. A performance analyst who wants to improve measurement accuracy may decide to disable the TSC when there is no need to measure the time stamp count. Our data shows that this "optimization" significantly degrades accuracy. This is surprising, because enabling the TSC implies maintaining an additional counter. The reason for the benefit of enabling the TSC is that perfctr, when the TSC is enabled, can read counters from user mode without calling into the kernel. It needs the TSC in order to ensure that no context switch happened between the start and the end of the user-mode readout.

(3) Difference between user+kernel and user mode. The error for user+kernel mode is significantly higher than for user mode-only measurements. When counting only user mode events, the CPU immediately stops counting when transitioning to kernel mode for starting, reading, or stopping the counters. Thus the cost of the calls to start, read, or stop does not include the instructions spent in the corresponding system calls. Because the read-read pattern on perfctr with TSC does not transition to kernel mode, the error for user mode is the same as the error for user+kernel mode in that case.

4.2. Number of Enabled Counters

Figure 4 left out one significant factor: As **Figure 5** shows, the measurement error also depends on the number of enabled counter registers. The figure focuses on the Pentium D, because this processor has the largest number of registers. The top of the figure shows results for user mode instruction counts, the bottom shows user+kernel mode. The left half of the figure shows the perfmon tool, and the right shows perfctr (with enabled TSC for reduced error). In each graph, the x-axis shows the number of enabled counter registers[5] (1-8), and the y-axis shows the measurement error in number of instructions. The y-axes vary between graphs to allow quickly spotting configurations that depend on the number of enabled registers (graphs with slopes). Each graph consists of multiple (possibly overlapping) lines: each line represents the register (one of the enabled registers) from which we took the instruction count (all enabled registers counted instructions).

The error for perfmon in user mode (top left), does not depend on the number of registers, unlike for user+kernel mode (bottom left). This is because perfmon uses system calls to read, start, or stop counters, and the user mode error basically corresponds to the cost of entering and leaving the kernel (the processor stops counting immediately when it enters kernel mode). When counting user+kernel instructions, the count varies with the number of enabled registers, because

5. The Pentium D has 18 counter registers (plus the TSC), but not every register can count every kind of event. We limit ourselves to the 8 registers that can count the events we were interested in for these experiments.

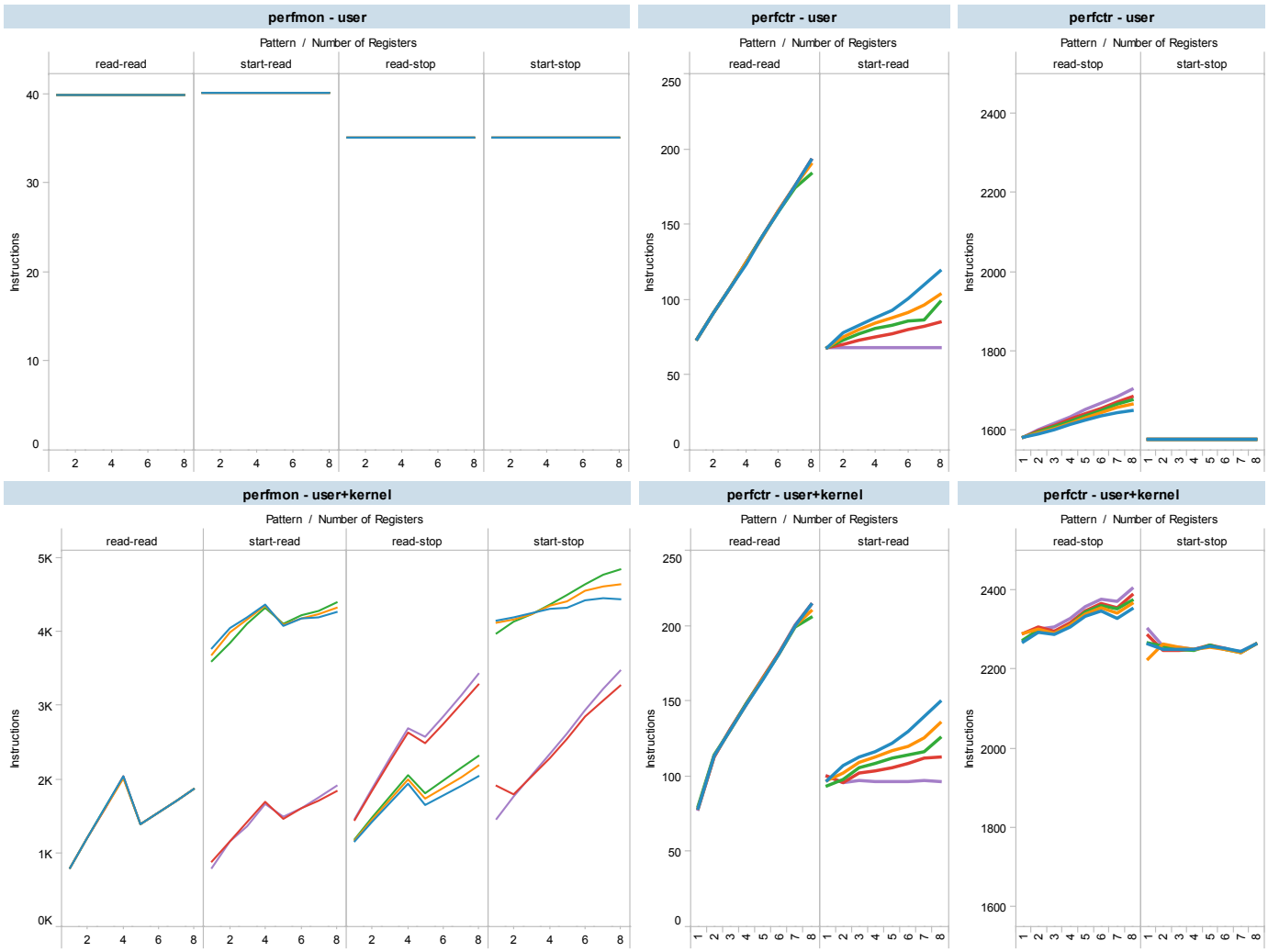


Figure 5. Error depends on number of enabled counter registers

each additional register that needs to be maintained adds an extra cost.

Perfmon in user+kernel mode (bottom left) shows two surprising phenomena. First, the graphs for all patterns but read-read show a reduction in error when moving from 4 to 5 enabled registers. We do not have an explanation for this dip: the phenomenon is repeatable, and none of our controlled factors correlate with the dip. It could be caused by a difference in behavior in perfmon's read system call when more than 4 registers are enabled. The second surprise is that all graphs except for read-read show multiple lines, and thus the error depends on which of the enabled registers we take the instruction count from. E.g. for start-read, two of the registers lead to an error of 1000 to 2000 instructions, and the other registers lead to an error around 4000 instructions! Given that the only pattern unaffected by this phenomenon is read-read, we believe that the cause lies in the behavior of perfmon's start and stop system calls.

Perfctr behaves differently: its error depends on the number of enabled registers even in user mode (not for start-stop, but in every pattern involving a read). This is due to perfctr's user-mode read function, which reads out the counters without system call, and thus loops through each counter while user-mode counting is still enabled.

Overall, this data shows that the measurement error significantly depends on the number of enabled counter registers, and, more surprisingly, even on the specific register used for counting the given event.

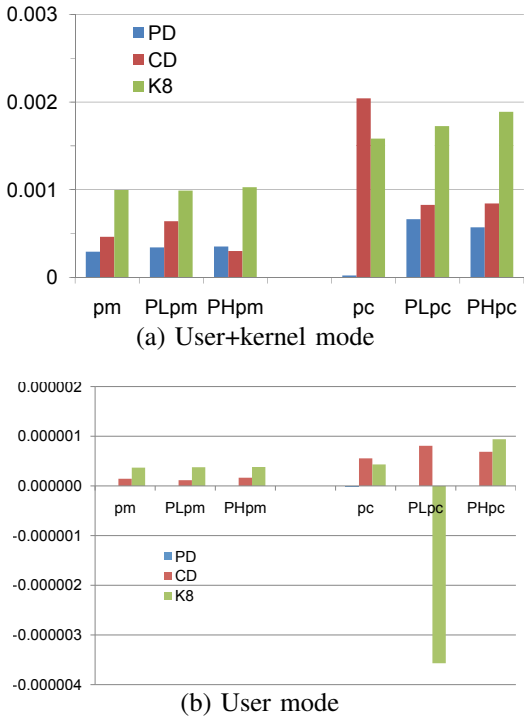
5. Error Depends on Duration

In the prior section we studied the error due to the overhead of accessing the counters at the beginning and the end of the measurement. That information was particularly relevant for measuring short sections of code. In this section we evaluate whether the measurement error depends on the duration of the benchmark. For this purpose we use our *loop* microbenchmark with a varying number of iterations.

We expect the measured data to fit the model $i_e = 1 + 3l$ (where i_e is the number of instructions, and l is the number of loop iterations). We consider any deviation from that model a measurement error.

The figures in this section show data for up to 1 million loop iterations. We verified that longer loops (we performed up to 1 billion iterations) do not affect our conclusions.

We expect the error for user+kernel measurements to depend on the benchmark duration because of interrupts (such as the timer interrupt or i/o interrupts). The interrupt handlers are executing in kernel mode, and the events that occur during their execution may be attributed to the kernel event counts of the currently running thread. Thus, the longer the duration of a measurement, the more interrupt-related instructions it will include. We expect that the error in user mode instructions does not depend on the benchmark duration.



(a) User+kernel mode

(b) User mode

Figure 6. Extra instructions per loop iteration

In our first experiment we studied the user+kernel mode instruction error. We computed the error i_Δ by subtracting the expected instruction count i_e (based on our model) from the measured count i_m. To determine how the error i_Δ changes with increasing loop iterations l, we determined the regression line through all points (l, i_Δ), and computed its slope (i_Δ/l). **Figure 6 (a)** shows the results. The x-axis shows six groups of bars, one group for each measurement infrastructure. Each group consists of three bars, one bar for each micro-architecture. The y-axis shows the number of extra instructions per loop iteration (i_Δ/l), which corresponds to the slope of the regression line. The figure shows that the slopes of all the regression lines are positive. This demonstrates that the error depends on the duration of the benchmark; the more loop iterations, the bigger the error. For example, for perfmon on the Athlon processor (K8) we measure 0.001 additional instructions for every loop iteration executed by the benchmark. As Figure 6 (a) shows, the error does not depend on whether we use the high level or low level infrastructure. This makes sense, as the fact that we use PAPI to read, start, or stop counting does not affect what happens in the kernel during the bulk of the measurement.

In our second experiment we computed the same regression lines for the user instruction counts. **Figure 6 (b)** shows that the error is several orders of magnitude smaller. For example, for perfmon on K8 we measure only 0.0000004 additional instructions per loop iteration. In general, the slopes of the regression lines are close to zero; some are negative while others are positive.

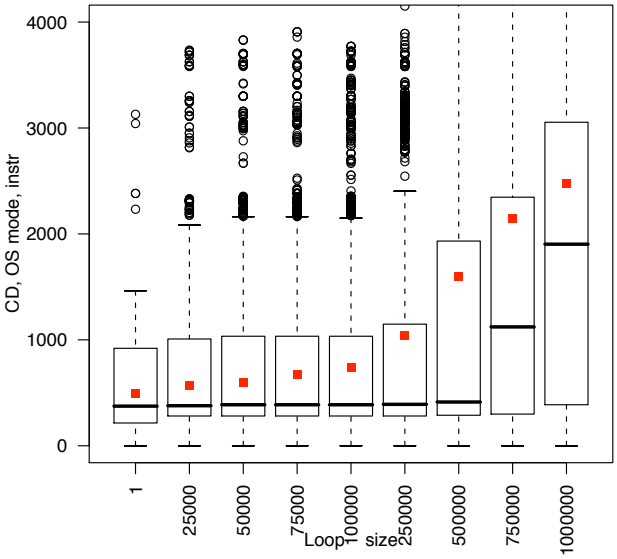


Figure 7. Kernel mode instructions by loop size (pc on CD)

To crosscheck the above results, we conducted an experiment in which we only measured the kernel-mode instruction count. Because the benchmark code does not directly cause any kernel activity, the entire number of measured kernel instructions can be attributed to measurement error. **Figure 7** presents the resulting measurements for a Core 2 Duo processor using perfctr. Each box represents the distribution of instruction count errors i_Δ (y-axis) for a given loop size l (x-axis). Note that the boxes are equally spaced along the x-axis, but the loop sizes are not (a linear growth in error will not lead to a straight line in the figure). A small square within each box indicates the average error for that loop size. Because interrupts (the prospective cause of this measurement error) are relatively infrequent events, and thus the probability of a short loop being perturbed by an interrupt is small, we use a large number of runs (several thousand) for each loop size.

Figure 7 shows that we measure an average of approximately 1500 kernel instructions for a loop with 500000

iterations, and approximately 2500 kernel instructions for a loop with 1 million iterations. This leads to a slope of 0.002 kernel instructions per iteration. The regression line through all the data underlying Figure 7 confirms this. It has a slope of 0.00204 kernel instructions per loop iteration, the exact number we report in Figure 6 (a) for Core 2 using perfctr.

We conclude that there is a small but significant inaccuracy in long-running measurements of user+kernel mode instruction counts using current performance counter infrastructures.

6. Accuracy of Cycle Counts

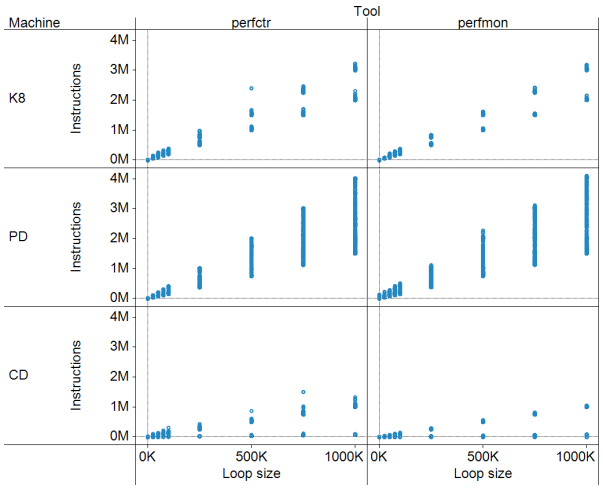
In the prior sections we have studied the accuracy of instruction count measurements. In particular, we measured the number of non-speculative instructions. This count is easy to model analytically: it is independent of the microarchitecture and only depends on the instruction set architecture. Every execution of a deterministic benchmark should lead to the exact same count. Moreover, our *loop* benchmark provides us with the "ground truth": it allows us to create a straightforward analytical model of instruction count based on the number of loop iterations.

Besides instructions, hardware performance counters can count many other types of events. The one event that most succinctly summarizes system performance is the number of cycles needed to execute a benchmark. All other events directly or indirectly affect this cycle count. We thus focus on investigating the accuracy of the measured cycle counts.

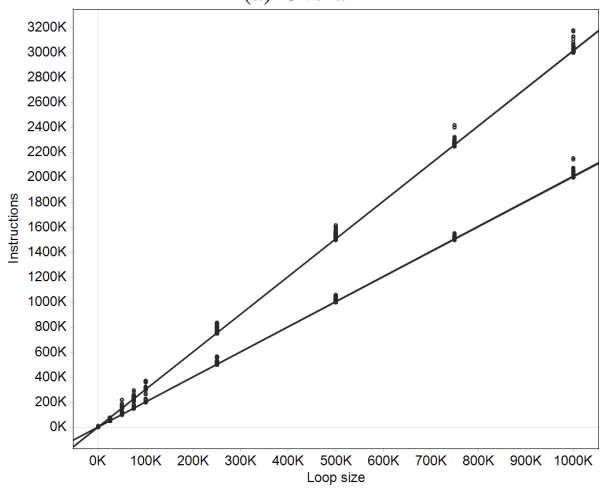
Figure 8 (a) shows the cycle measurements for loops of up to 1 million iterations. It consists of a matrix of six scatter plots. It includes plots for all three processors (rows "K8", "PD", and "CD") for perfctr and perfmon. The x-axis of each plot represents the loop size, and the y-axis represents the measured user+kernel cycle count. We can see that for a given loop size the measurements vary greatly. For example, on Pentium D, we measure anywhere between 1.5 and 4 million cycles for a loop with 1 million iterations. Is this variation all attributable to measurement error? Is the measurement instrumentation indeed affecting the cycle count by that much for such a long loop?

To answer this question, **Figure 8 (b)** focuses on the data for perfmon on the Athlon processor (the "K8" "pm" plot at the top right of Figure 8 (a)). It shows that the measurements are split into two groups. We include two lines in the figure to show two possible models for this cycle count: $c = 2i$ and $c = 3i$ (where c is the cycle count and i is the number of loop iterations). The figure shows that these lines bound the two groups from below (i.e. in each group, a measurement is as big as the line or bigger).

Figure 9 shows a breakdown of the same data based on two factors, measurement pattern (rows "start-stop", "start-read", "read-stop", and "read-read") and optimization level (columns "-O0", "-O1", "-O2", and "-O3"). This figure explains the reason for this bimodality. We can see that the points in each



(a) Overall



(b) With pm on K8

Figure 8. Cycles by loop size

of the 16 plots form a line with a different slope. The figure shows that neither the optimization level nor the measurement pattern determines the slope, only the combination of both patterns allows us to separate the data into groups with specific slopes. The explanation for this strange interaction is simple: Since a change in any of these two factors leads to a different executable, the loop code (which is identical across executables and does not contain any loads or stores) is placed at a different location in memory. We have verfied this and found that all the instructions making up the 3-cycles per iteration loops straddle a 8-byte boundary, but the 2-cycles per iteration loops fit within an aligned block of 8 bytes. This straddling of an 8-byte boundary could affect the performance of a branch predictor, i-cache, i-TLB, or a more sensitive microarchitectural optimization like a trace cache or a loop stream detector [6], and thus can lead to a different number of cycles per loop iteration.

The fact that a cycle measurement can be off by such a large factor is worrysome. This problem cannot solely be attributed to the performance counter measurement in-

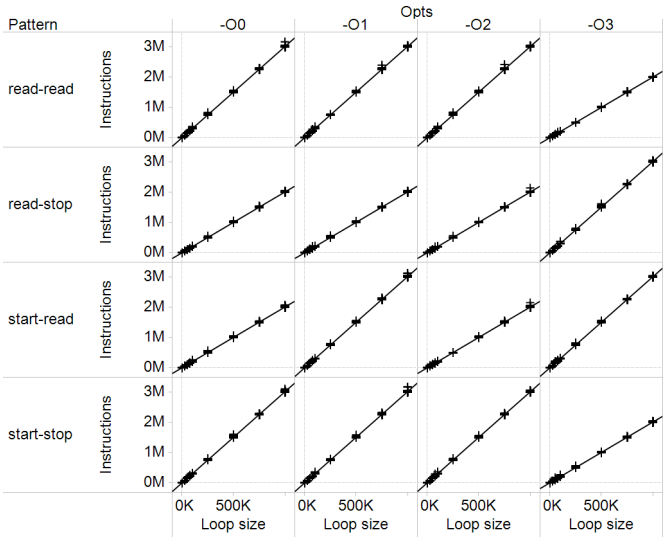


Figure 9. Cycles by loop size with pm on K8 for different patterns and optimization levels

frastructure: any change in environment (e.g. the use of a different compiler, or a different optimization level) could affect the cycle count in the same way. We strongly believe that such drastic external influences on micro-architectural performance metrics dwarf the direct influence any reasonable performance measurement infrastructure could possibly have on measurement accuracy.

7. Limitations

This section identifies the limitations and threats to the validity of the results presented in the three previous sections.

Platforms. Our experiments focused exclusively on Linux on the IA32 architecture. This enabled a direct comparison of all the six counter access interfaces we studied. Because some of these interfaces do not exist on other operating systems or on other hardware architectures, the question how they would compare on other platforms is a hypothetical one. With this study we raise the general issue of accuracy of hardware performance counter measurements, we show that the measurement error on the prevalent platforms can range over two orders of magnitude (from tens to thousands of instructions) depending on the measurement approach, and we encourage performance analyst to take the accuracy of their specific infrastructure into consideration.

Benchmarks. While our microbenchmarks (*null* and *loop*) are definitely not realistic applications, we have a good reason to use them: In order to determine the error of using performance counters, we need benchmarks for which we know the ground truth, that is, where we can analytically determine the exact number of events that have to occur. Moreover, our *loop* benchmark, while admittedly simple, is not entirely unrepresentative: Even some real-world benchmarks (such as SPEC CPU's bzip2) are dominated by a single tight loop.

Micro-architectural counts. In this paper we studied the error for instruction and for cycle count measurements. We discovered that the cycle count measurements are drastically perturbed by aspects (i.e. code placement) that are not directly attributable to the measurement infrastructure. Because cycle counts represent the result of all micro-architectural activity (e.g. they are affected by cache miss counts, branch mispredicts, stalls, ...), we expect similar perturbation of various individual micro-architecural event counts. Because we do not have a precise model of the three micro-architectures we used, we do not have a ground truth to compare our measurements to. The order of magnitude of the cycle count perturbation completely overshadows the errors we would expect due to the measurement infrastructure.

8. Related Work

Researchers in the PAPI group have previously studied the accuracy of their infrastructure. Moore [7] reports the cost for start/stop and for read as the number of cycles on five different platforms. On Linux/x86, she reports 3524 cycles for start/stop and 1299 for read, numbers which lie in the range of our results. However, as we show in this paper, cycle counts can drastically vary, even when measuring the exact same block of code (e.g. just due to the fact that the code is located at a different memory address). Moore's work does not mention the specific processor or operating system used, does not report the exact measurement configuration (PAPI high level or low level API, compiler optimization level, how many registers used), and it does not mention the number of runs executed to determine that value. It just reports one single number. Also in the context of PAPI, Dongarra et al. [8] mention potential sources of inaccuracy in counter measurements. They point out issues such as the extra instructions and system calls required to access counters, and indirect effects like the pollution of caches due to instrumentation code, but they do not present any experimental data.

Korn et al. [9] study the accuracy of the MIPS R12000 performance counters using micro-benchmarks. They compare the measured event counts to analytical results (the expected event counts of their micro-benchmarks) and to results from simulating their micro-benchmarks on SimpleScalar's sim-outorder micro-architectural simulator. They use three micro-benchmarks: a linear sequence of instructions (to estimate L1 cache misses), a loop (they vary the number of iterations, like we do), and a loop accessing a large array in memory (to estimate d-cache and TLB misses). They use a single infrastructure (libperfex on SGI's IRIX operating system) to access the counters. Maxwell et al. [10] broaden Kron et al.'s work by including three more platforms. Neither paper explores the space of possible measurement configurations. Our results show that many of the factors not controlled in these two studies can drastically affect measurement accuracy.

Based on Maxwell et al.'s work, Araiza et al. [11] propose to develop a suite of platform-independent micro-benchmarks

to allow the comparison of measured performance counts and analytically determined counts. In this paper we use two trivial micro-benchmarks, the empty *null* benchmark for which we expect an event count of 0, and the *loop* benchmark for which we can analytically determine the expected number of instructions. We believe that the refinement of such benchmarks to allow the analytical determination of counts for any micro-architectural event requires special consideration of the sensitivity of modern processors to initial conditions such as the memory layout (Mytkowicz et al. [12]). Najafzadeh et al. [13] discuss a methodology to validate fine-grained performance counter measurements and to compensate the measurement error. They propose to measure the cost of reading out counters by injecting null-probes at the beginning of the code section to be measured. They indicate that this placement would lead to a more realistic context for measuring the cost of reading. They do not include a quantitative evaluation of their idea. Weaver and McKee [14] compare measured instruction counts with counts gathered using binary instrumentation. They investigate the overall variability of marathon timings, while we study the factors impacting the measurement of sprints.

Mytkowicz et al. [15] study measurement accuracy when the number of events to measure is greater than the number of the available performance counter registers. They compare two "time interpolation" approaches, multiplexing and trace alignment, and evaluate their accuracy. Their work does not address the measurement error caused by any software infrastructure that reads out and virtualizes counter values.

9. Conclusions

Accurately measuring sprints is difficult. We evaluate the accuracy of a range of measurement infrastructures on different processors, and we identify the factors of the measurement setup that can lead to inaccuracies of several thousand instructions. For future studies involving sprint measurements, we recommend to control these factors, to evaluate their impact in the specific setups used (e.g. using our simple microbenchmarks), and to report their levels in the experimental methodology.

Acknowledgments

This work has been conducted in the context of the Binary Translation and Virtualization cluster of the EU HiPEAC Network of Excellence. It has been funded by the Swiss National Science Foundation under grant number 200021-116333/1.

References

[1] J. Hintze and R. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, May 1998.

[2] B. Sprunt, "The basics of performance monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002.

[3] M. Pettersson, "perfctr," http://user.it.uu.se/~mikpe/linux/perfctr/.

[4] S. Eranian, "perfmon2: A flexible performance monitoring interface for linux," in *Proceedings of the Linux Symposium*, July 2006, pp. 269–288.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC'00)*, Dallas, Texas, November 2000.

[6] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual, Section 3.4.2.4: Optimizing the Loop Stream Detector," http://www.intel.com/products/processor/manuals/.

[7] S. V. Moore, "A comparison of counting and sampling modes of using performance monitoring hardware," in *Proceedings of the International Conference on Computational Science-Part II (ICCS'02)*. London, UK: Springer-Verlag, 2002, pp. 904–912.

[8] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 289.2.

[9] W. Korn, P. J. Teller, and G. Castillo, "Just how accurate are performance counters?" in *Proceedings of the IEEE International Conference on Performance, Computing, and Communications (IPCCC'01)*, 2001, pp. 303–310.

[10] M. Maxwell, P. Teller, L. Salayandia, and S. Moore, "Accuracy of performance monitoring hardware," in *Proceedings of the Los Alamos Computer Science Institute Symposium (LACSI'02)*, October 2002.

[11] R. Araiza, M. G. Aguilera, T. Pham, and P. J. Teller, "Towards a cross-platform microbenchmark suite for evaluating hardware performance counter data," in *Proceedings of the 2005 conference on Diversity in computing (TAPIA'05)*. New York, NY, USA: ACM, 2005, pp. 36–39.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[13] H. Najafzadeh and S. Chaiken, "Validated observation and reporting of microscopic performance using pentium ii counter facilities," in *Proceedings of the 4th international workshop on Software and performance (WOSP'04)*. New York, NY, USA: ACM, 2004, pp. 161–165.

[14] V. Weaver and S. McKee, "Can hardware performance counters be trusted?" in *Proceedings of the International Symposium on Workload Characterization (IISWC'08)*, Sept. 2008, pp. 141–150.

[15] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *Proceedings of the International Symposium on Microarchitecture (MICRO'07)*, 2007.