Accuracy of Performance Counter Measurements

Dmitrijs Zaparanuks
University of Lugano
zaparand@lu.unisi.ch

Milan Jovic
University of Lugano
jovicm@lu.unisi.ch

Matthias Hauswirth
University of Lugano
Matthias.Hauswirth@unisi.ch

ABSTRACT

Many workload characterization studies 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). We find significant differences in accuracy of various usage patterns for the different infrastructures and processors. Based on these results we provide guidelines for finding the best measurement approach.

1. INTRODUCTION

Many workload characterization studies 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 microarchitectural 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. However, such infrastructures incur a cost: the software instructions executed to access a counter, or to maintain a per-thread counter, may slightly perturb that same counter. While this perturbation is presumably small, its significance depends on the specific measurement. A study characterizing the end-to-end behavior of a workload may not be significantly affected by the cost of accessing a performance counter. However, that same cost may well affect a study that focuses on the behavior of shorter execution phases (e.g. optimization phases in just-in-time method compilations, time spent in signal handlers, individual garbage collection phases, time spent in spin-locks, or application-level program phases).

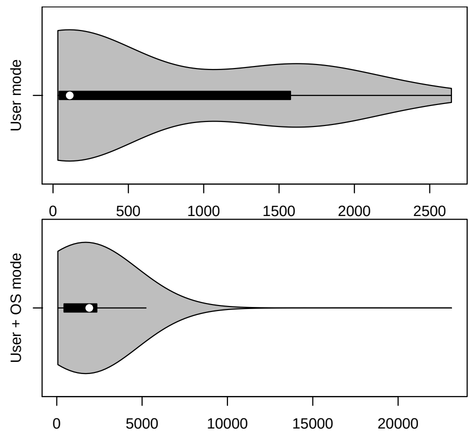

Figure 1: Measurement Error in Instructions

This paper provides the first comparative study of the accuracy of three commonly used measurement infrastructures on three common processors. **Figure 1** provides an overview of the accuracy of different measurement infrastructures. The figure consists of two violin plots [5]. The x axis shows the measurement error (the number of superfluous instructions executed and counted due to the measurement infrastructure). Each violin summarizes the error of over 170000 measurements performed on a large number of different infrastructures and configurations. In the upper violin plot, the error only includes instructions executed in user-mode, while the lower 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 measurements.

The remainder of this paper is structured as follows: Section 2 provides background information on hardware per-

University of Lugano
Faculty of Informatics
Technical Report 2008/05
September 2008

formance 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 presents guidelines for improving the accuracy of hardware performance counter measurements. Section 9 compares this study to related work, and Section 10 concludes.

2. BACKGROUND

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

2.1 Hardware Performance Counters

Most modern processors contain hardware performance counters [13], 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).

2.2 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.

2.3 Per-Thread Counters

Hardware performance counters count events happening on a given processor[1]. The counter register does not distinguish between different software threads that run on its processor[2]. 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.

[1]On multi-core processors each core usually contains its own set of counters.

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

2.4 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 [12] 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 [2], 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.

2.5 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.

3. EXPERIMENTAL METHODOLOGY

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

3.1 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

	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

Table 1: Processors used in this Study

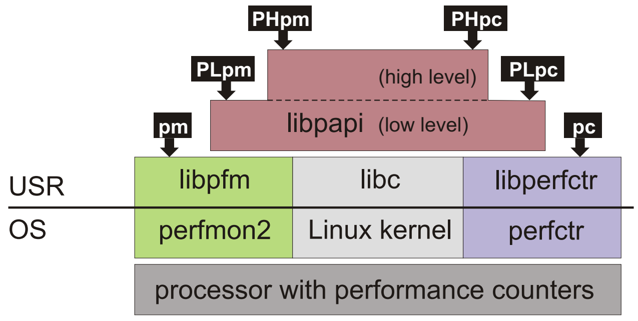

Figure 2: Counter Access Infrastructure

specified in the IA32 ISA and is available on any IA32 processor.

3.2 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).

3.3 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 (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).

```
__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

3.4 Benchmarks

In this paper we assess the *accuracy* of performance counter measurement approaches. To do this, we compare 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.

3.5 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.

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

Table 2: Counter Access Patterns

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.

3.6 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 Fewer Counters = Smaller Error?

Modern processors provide multiple performance counter registers (see Table 1 for the number of counters available on the three processors we use). This section explores whether the measurement error depends on the number of counters measured.

Use of Time Stamp Counter. Besides the micro-architecture-specific number of hardware performance counters, the IA32 architecture also prescribes the existence of a time stamp counter register (TSC). Perfctr provides access to that register, and allows us to enable or disable its use. We wanted to determine by how much we could reduce the error by disabling these additional TSC readouts.

Against our expectations, we found that disabling the TSC actually increases the error. **Figure 4** shows the error of perfctr (pc) on the Core 2 Duo (CD). The figure consists of two matrices of box plots. The left matrix represents user+kernel mode while the right one shows only user mode. Both matrices contain four box plots, one box plot per counter access pattern. Each box plot contains two boxes, the left box represents the error when TSC is off, the right one when TSC is on. Each box summarizes 960 runs using different compiler optimization levels and different selections of performance counter registers.

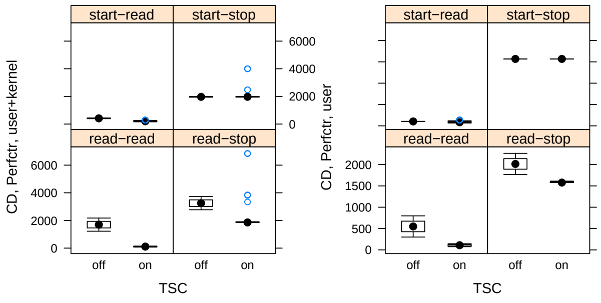

Figure 4: Using TSC Reduces Error on Perfctr

We see that in particular for the read-read and read-stop patterns, measuring the TSC in addition to the other counters *reduces* the error. A closer look at the perfctr library provides the explanation of this unintuitive result. Perfctr implements a fast user-mode approach to reading its virtualized performance counters. However, when TSC is not used, perfctr cannot use that approach, and needs to use a slower system-call-based approach. As Figure 4 shows, all patterns that include a read call are affected by TSC. The read-read and read-stop patterns, which both begin with a read, are equally affected. The start-read pattern, where the read happens at the end of the measurement interval, is less affected. The start-stop pattern, which does not include a read, is not affected. Overall, the figure shows that enabling the TSC when using perfctr can drastically reduce the measurement error (e.g. the median error for read-read drops from 1698 instructions down to 109.5).

Number of Performance Counters. In the following section we study how the number of measured performance counters affects the measurement error. Given the benefit of using the TSC on perfctr, we enabled the TSC in all perfctr experiments. Our experiments include measurements for all possible combinations of enabled counters. **Figure 5** shows how the measurement error depends on the number of counters used in the Athlon processor (K8). The figures for the other processors exhibit similar trends and are omitted for brevity. The left side of the figure shows the data for user+kernel mode, the right focuses on user mode. Perfmon (pm) is at the top and perfctr (pc) is at the bottom.

The figure shows that the infrastructure (perfmon and perfctr) and the pattern interact with the number of counters. This is particularly striking for perfmon in user+kernel mode (top left): when using read-read, measuring an additional counter increases the error by approximately 100 instructions; when using start-stop, adding a counter can slightly reduce the error. For perfmon in user mode (top right), the error is independent of the number of registers.

For perfctr (bottom), both in user+kernel and in user mode, the error marginally increases with increasing number of registers. The pattern with the most pronounced increase (difficult to see in the figure due to the scale of the y axis) for perfctr is read-read. There the error changes from a median of 84 instructions for one counter to 125 instructions for four counters. Perfctr's read-read pattern causes the same errors

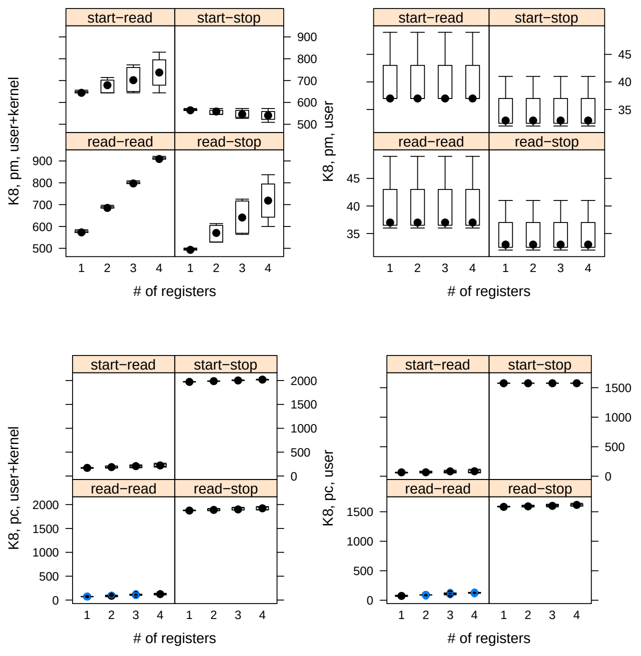

Figure 5: Error Depends on Number of Counters

in user+kernel mode as it does in user mode, because with the enabled TSC, read-read never enters kernel mode.

Overall, the number of measured performance counter registers can significantly affect the measurement error (e.g. the median error for read-read for user+kernel mode on perfmon increases from 573 instructions for one register to 909 instructions for four registers). Thus, depending on the measurement infrastructure and the pattern used, reducing the number of concurrently measured hardware events can be a good way to improve measurement accuracy.

4.2 Dependency on Infrastructure

We expect different measurement infrastructures to cause different measurement errors. In this section we quantify these differences and answer two questions: (1) What impact does the use of a high level API have on the error? (2) How does the error differ between perfmon and perfctr?

Mode	Tool	Best Pattern	Median	Min
user+kernel	pm	read-read	726	572
user+kernel	PLpm	start-read	742	653
user+kernel	PHpm	start-read	844	755
user+kernel	pc	start-read	163	74
user+kernel	PLpc	start-read	251	249
user+kernel	PHpc	start-read	339	333
user	pm	read-read	37	36
user	PLpm	start-read	134	134
user	PHpm	start-read	236	236
user	pc	start-read	67	56
user	PLpc	start-read	152	144
user	PHpc	start-read	236	230

Table 3: Error Depends on Infrastructure

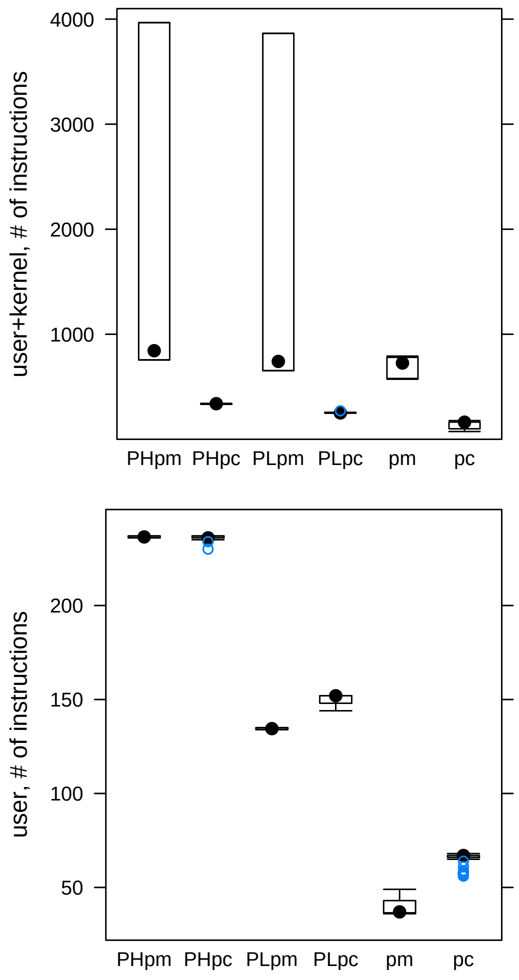

Figure 6: Error Depends on Infrastructure

Figure 6 and **Table 3** provide the basis for answering these questions. Figure 6 consists of two box plots. The top box plot shows the error for user+kernel mode, the bottom one focuses on user mode. Each box plot consists of six boxes, three for perfmon (*pm) and three for perfctr (*pc). The left-most two boxes represent the PAPI high level API, the next two boxes the PAPI low level API, and the two boxes on the right represent the direct use of the perfmon and perfctr libraries. We use the best readout pattern for each specific infrastructure. Moreover, for perfctr, we enable the TSC to enable the benefit of user-mode readout. We only use one counter register to avoid bias due to the different numbers of registers in the different platforms. Each box represents multiple measurements for all hardware platforms and compiler optimization levels. Table 3 shows the patterns used for the boxes in Figure 6 as well as the median and minimum of each box.

Lower Level API = Smaller Error? One would expect to reduce the measurement error by directly using the low-level infrastructures (i.e. perfctr or perfmon) instead of going through PAPI. Figure 6 shows that this indeed is the case. The reduction in measurement error when using lower level infrastructures is significant. The use of low level PAPI, instead of high level PAPI, reduces the error between 12% (from 844 down to 742 instructions for perfmon in user+kernel mode) and 43% (from 236 down to 134 instructions for perfmon in user mode). When using perfmon or perfctr directly, instead of the PAPI low level API, the error is reduced between 2% (from 752 down to 744 instructions for perfmon in user+kernel mode) and 72% (from 134 down to 37 instructions for perfmon in user mode).

Perfctr or Perfmon? A performance analyst using hardware performance counters has to decide whether to use perfctr or perfmon. Here we provide guidance for this decision. As Figure 6 and Table 3 show, neither perfmon nor perfctr is the clear winner. However, if the performance analyst knows whether she needs to measure user+kernel mode or just user mode instructions, the decision is clear. This decision does not depend on whether the analyst intends to use PAPI or not. For user mode measurements, using perfmon instead of perfctr reduces the median error by 45% (pm), 22% (PLpm), or by 0% (PHpm). For user+kernel mode measurements, using perfctr instead of perfmon reduces the median error by 77% (pc), 66% (PLpc), or by 59% (PHpc).

4.3 Factors Affecting Accuracy

The prior subsections show how specific factors, such as the number of measured registers, affect accuracy. To verify whether this effect is statistically significant, we have performed an n-way analysis of variance (ANOVA [6]). We used the processor, measurement infrastructure, access pattern, compiler optimization level, and the number of used counter registers as factors and the instruction count as the response variable. We have found that all factors but the optimization level are statistically significant ($Pr(> F) < 2 \cdot 10^{-16}$). The fact that the compiler optimization level is not affecting the measurement error is not surprising, as the code that is actually optimizable is restricted to the small number of instructions constituting the calls to the measurement methods.

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.

In our first experiment we studied the user+kernel mode

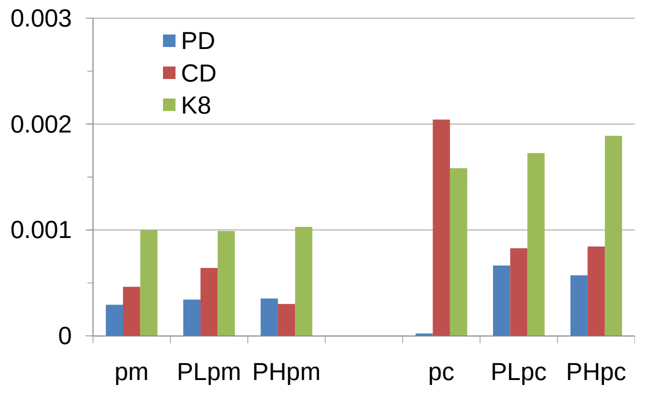

Figure 7: User+Kernel Mode Errors

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 7** 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 7 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.

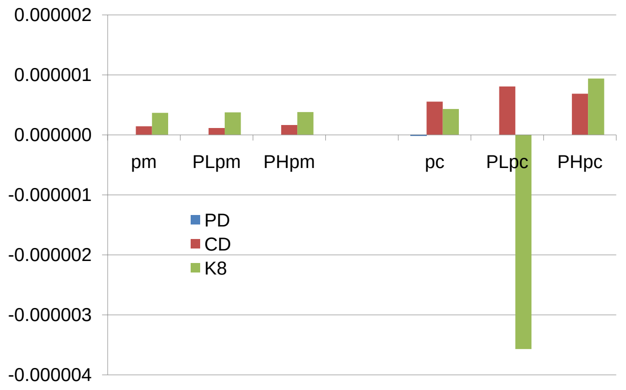

Figure 8: User Mode Errors

In our second experiment we computed the same regression lines for the user instruction counts. **Figure 8** shows that the error is several orders of magnitude smaller. For example, for perfmon on the Athlon processor (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.

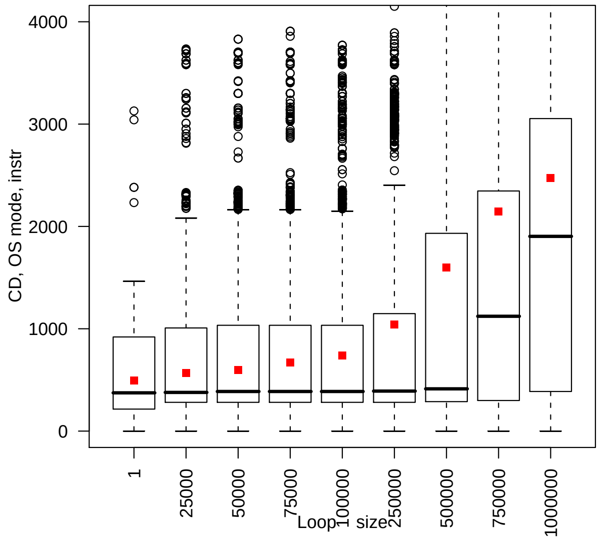

Figure 9: Kernel Mode Instructions by Loop Size (pc on CD)

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 9** presents the resulting measurements for a Core 2 Duo processor using perfctr. Each box represents the distribution of instruction count errors $i_\triangle$ (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 9 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 9 confirms this. It has a slope of 0.00204 kernel instructions per loop iteration, the exact number we report in Figure 7 for Core 2 Duo 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 micro-architecture 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.

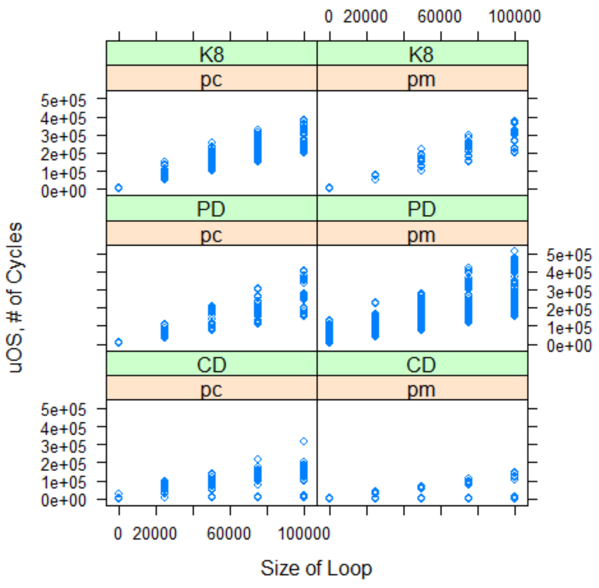

Figure 10: Cycles by Loop Size

Figure 10 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 (column "pc") and perfmon (column "pm"). 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 11** focuses on the data for perfmon on the Athlon processor (the "K8" "pm" plot at the top right of Figure 10). 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

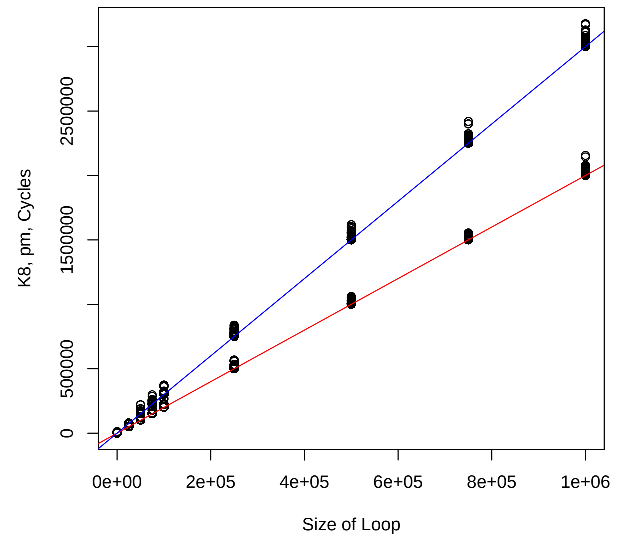

Figure 11: Cycles by Loop Size with pm on K8

$c = 3i$ (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).

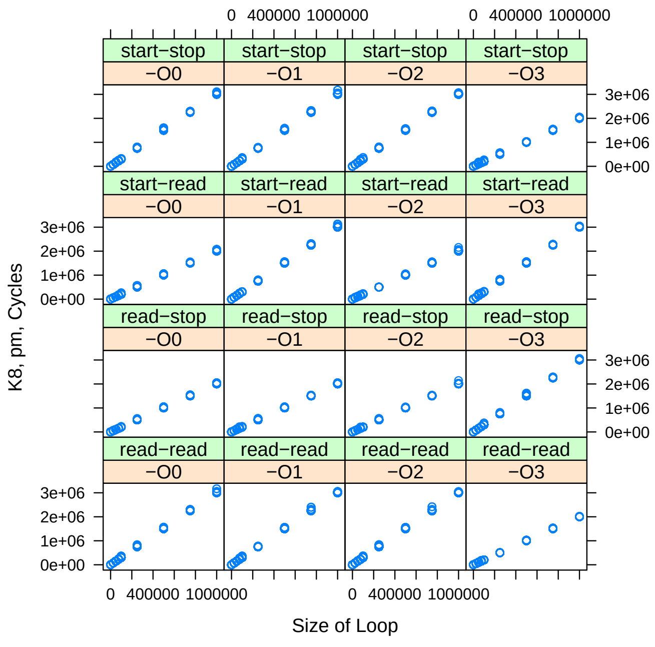

Figure 12: Cycles by Loop Size with pm on K8 for Different Patterns and Optimization Levels

Figure 12 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 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. This shift in code placement affects branch predictor, i-cache, and i-TLB performance, 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 infrastructure: 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 (PHpm, PHpc, PLpm, PLpc, pm, and pc) 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 raised the issue of accuracy of hardware performance counter measurements, and we encourage performance analyst to always take the accuracy of their specific infrastructure into consideration.

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. The perturbation of specific event counts due to code placement is an interesting topic for future research.

8. GUIDELINES

In this section we present guidelines for conducting more accurate hardware performance measurements. These guidelines are based on our experience and the results we presented in the previous three sections.

Frequency scaling. Rarely do studies that include a performance evaluation report whether frequency scaling was enabled or disabled on the processor. In gathering the data for this paper, we originally did not control this factor. This resulted in significant variability in our results, caused by the power daemon dynamically changing the processor's clock frequency. Depending on the power management configuration and the workload characteristics of the benchmarks, the clock frequency may rarely change, it may change only after running a number of experiments, or it may change several times during a single measurement. Each of these cases will introduce different types of error into performance measurements. Since the frequency setting of the processor does not affect the bus frequency, changing the CPU frequency can affect the memory access latency (as measured in CPU cycles), and thus ultimately the measured CPU cycle count. We recommend setting the processor frequency (of all cores) to a fixed value. In Linux, this is done by chosing the "performance" governor (for the highest possible frequency), or the "powersave" governor (for the lowest possible frequency).

Use of low level APIs. Performance analysts may assume that using a lower level measurement infrastructure automatically leads to more accurate measurements. This is only the case if the low level infrastructure is used in exactly the right way. For example, turning off the time stamp counter when measuring with perfctr, while seemingly reducing the amount of work required by the infrastructure (one less counter to read), will lead to a degradation of accuracy (since it prevents the use of fast user-mode reads). Moreover, in our results we show which measurement patterns lead to the smallest error. We encourage performance analysts to ensure that they are using the best performing pattern, no matter whether they use low-level or high-level APIs.

Micro-architectural event counts. We have shown that cycle counts, unlike the counts of retired instructions, are easily affected by conditions such as the placement of the measured code in memory. This effect dwarfs any expected error due to measurement overhead, and it lead to significant errors even in long-running benchmarks. We caution performance analysts to be suspicious of cycle counts, and any micro-architectural event counts (cycles are but a result of all other micro-architectural events), gathered with performance counters.

Error depends on duration. We have found a small but significant inaccuracy in long-running measurements using current performance counter infrastructures. However, this variable error, which occurs in addition to the fixed error due to the calls to the measurement infrastructure at the start and end of the measurement, only manifests itself when including kernel mode instructions in the measurements.

9. RELATED WORK

Researchers in the PAPI group have previously published work including discussions of the accuracy of their infrastructure. Moore [9] distinguishes between the accuracy of two distinct performance counter usage models: counting and sampling. The section on counting accuracy presents the cost for start/stop and for read as the number of cycles on five different platforms: Linux/x86, Linux/IA-64, Cray T3E, IBM POWER3, and MIPS R12K. 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. [3] 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. [7] 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, and they do not explore the space of possible measurement configurations. We show the accuracy of instruction count measurements of the three prevalent measurement infrastructures on Linux on three state-of-the-art processors. We study a large parameter space, varying factors from the compiler optimization level, over the measurement pattern, to the exact set of performance counters used for a measurement. Our results show that varying these factors can drastically affect measurement accuracy.

Besides using the libperfex library, Korn et al. [7] also evaluate the accuracy of the perfex command line tool based on that library. To no surprise, this leads to a huge inaccuracy (over 60000% error in some cases), since the perfex program starts the micro-benchmark as a separate process, and thus includes process startup (e.g. loading and dynamic linking) and shutdown cost in its measurement. We have also conducted measurements using the standalone measurement tools available for our infrastructures (`perfex` included with perfctr, `pfmon` of perfmon2, `papiex` available for PAPI), and found errors of similar magnitude. Since our study focuses on fine-grained measurements we do not include these numbers.

Maxwell et al. [8] broaden Kron et al.'s work by including three more platforms: IBM POWER3, Linux/IA-64, and Linux/Pentium[3]. They do not study how the factors a performance analyst could control (such as the measurement pattern) would affect accuracy. They report on performance metrics such as cycles and cache misses. We show in Section 6 that such information is very sensitive to initial conditions, such as the placement of the loop code, and thus may drastically change in a slightly different context.

Based on Maxwell et al.'s work, Araiza et al. [1] propose to

[3]They do not specify which version of the Pentium processor they studied (different Pentium processors are based on entirely different micro-architectures).

develop a suite of platform-independent micro-benchmarks to allow the comparison of measured performance counts and analytically determined counts. In a related effort with a different goal, Yotov et al. [14] have developed micro-benchmarks to explore the size of micro-architectural structures. 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.

Najafzadeh et al. [11] 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.

Mytkowicz et al. [10] also study the accuracy of performance counter-based measurements. However, their focus is on the accuracy of measurements 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.

10. CONCLUSIONS

This paper constitutes the first comparative study of the accuracy of performance counter measurement infrastructures. We evaluate three commonly used infrastructures, PAPI, perfctr, and perfmon2, on three different modern processors, Athlon 64 X2, Pentium D, and Core 2 Duo. Our results show significant inaccuracies even in instruction counts (up to thousands of instructions). We study the factors with the biggest impact on measurement accuracy, and we provide guidelines that help performance analysts improve the accuracy of their performance counter measurements.

11. REFERENCES

[1] Roberto Araiza, Maria Gabriela Aguilera, Thientam Pham, and Patricia 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)*, pages 36–39, New York, NY, USA, 2005. ACM.

[2] 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.

[3] Jack Dongarra, Kevin London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You, and Min 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)*, page 289.2, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Stephane Eranian. perfmon2: A flexible performance monitoring interface for linux. In *Proceedings of the Linux Symposium*, pages 269–288, July 2006.

[5] J.L. Hintze and R.D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, May 1998.

[6] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Pearson Education International, fifth edition, 2002.

[7] 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)*, pages 303–310, 2001.

[8] 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.

[9] Shirley 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)*, pages 904–912, London, UK, 2002. Springer-Verlag.

[10] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Time interpolation: So many metrics, so few registers. In *Proceedings of the International Symposium on Microarchitecture (MICRO'07)*, 2007.

[11] Haleh Najafzadeh and Seth 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)*, pages 161–165, New York, NY, USA, 2004. ACM.

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

[13] Brinkley Sprunt. The basics of performance monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[14] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS'05)*, pages 181–192, New York, NY, USA, 2005. ACM.