Can Hardware Performance Counters be Trusted?

Vincent M. Weaver and Sally A. McKee
Computer Systems Laboratory
Cornell University
{vince,sam}@csl.cornell.edu

Abstract

When creating architectural tools, it is essential to know whether the generated results make sense. Comparing a tool's outputs against hardware performance counters on an actual machine is a common means of executing a quick sanity check. If the results do not match, this can indicate problems with the tool, unknown interactions with the benchmarks being investigated, or even unexpected behavior of the real hardware. To make future analyses of this type easier, we explore the behavior of the SPEC benchmarks with both dynamic binary instrumentation (DBI) tools and hardware counters.

We collect retired instruction performance counter data from the full SPEC CPU 2000 and 2006 benchmark suites on nine different implementations of the x86 architecture. When run with no special preparation, hardware counters have a coefficient of variation of up to 1.07%. After analyzing results in depth, we find that minor changes to the experimental setup reduce observed errors to less than 0.002% for all benchmarks. The fact that subtle changes in how experiments are conducted can largely impact observed results is unexpected, and it is important that researchers using these counters be aware of the issues involved.

1 Introduction

Hardware performance counters are often used to characterize workloads, yet counter accuracy studies have seldom been publicly reported, bringing such counter-generated characterizations into question. Results from counters are treated as accurate representations of events occurring in hardware, when, in reality, there are many caveats to the use of such counters.

When used in aggregate counting mode (as opposed to sampling mode), performance counters provide architectural statistics at full hardware speed with minimal overhead. All modern processors support some form of counters. Although originally implemented for debugging hardware designs during development, they have come to be used extensively for performance analysis and for validating tools and simulators. The types and numbers of events tracked and the methodologies for using these performance counters vary widely, not only across architectures, but also across systems sharing an ISA. For example, the Pentium III tracks 80 different events, measuring only two at a time, but the Pentium 4 tracks 48 different events, measuring up to 18 at a time. Chips manufactured by different companies have even more divergent counter architectures: for instance, AMD and Intel implementations have little in common, despite their supporting the same ISA. Verifying that measurements generate meaningful results across arrays of implementations is essential to using counters for research.

Comparison across diverse machines requires a common subset of equivalent counters. Many counters are unsuitable due to microarchitectural or timing differences. Furthermore, counters used for architectural comparisons must be available on all machines of interest. We choose a counter that meets these requirements: number of retired instructions. For a given statically linked binary, the retired instruction count *should* be the same on all machines implementing the same ISA, since the number of retired instructions excludes speculation and cache effects that complicate cross-machine correlation. This count is especially relevant, since it is a component of both the Cycles per Instruction (CPI) and (conversely) Instructions per Cycle (IPC) metrics commonly used to describe machine performance.

The CPI and IPC metrics are important in computer architecture research; in the rare occasion that a simulator is actually validated [19, 5, 7, 24] these metrics are usually the ones used for comparison. Retired instruction count and IPC are also used for vertical profiling [10] and trace alignment [16], which are methods of synchronizing data from various trace streams for analysis.

Retired instruction counts are also important when generating basic block vectors (BBVs) for use with the popular SimPoint [9] tool, which tries to guide statistically valid partial simulation of workloads that, if used properly, can greatly reduce experiment time without sacrificing accuracy

in simulation results. When investigating the use of DBI tools to generate BBVs [26], we find that even a single extra instruction counted in a basic block (which represents the code executed in a SimPoint) can change which simulation points the SimPoint tool chooses to be most representative of whole program execution.

All these uses of retired instruction counters assume that generated results are repeatable, relatively deterministic, and have minimal variation across machines with the same ISA. Here we explore whether these assumptions hold by comparing the hardware-based counts from a variety of machines, as well as comparing to counts generated by Dynamic Binary Instrumentation (DBI) tools.

2   Related Work

Black et al. [4] use performance counters to investigate the total number of retired instructions and cycles on the PowerPC 604 platform. Unlike our work, they compare their results against a cycle-accurate simulator. The study uses a small number of benchmarks (including some from SPEC92), and the total number of instructions executed is many orders of magnitude fewer than in our work.

Patil et al. [18] validate SimPoint generation using CPI from Itanium performance counters. They compare different machines, but only the SimPoint-generated CPI values, not the raw performance counter results.

Sherwood et al. [20] compare results from performance counters on the Alpha architecture with SimpleScalar [2] and the Atom [21] DBI tool. They do not investigate changes in counts across more than one machine.

Korn, Teller, and Castillo [11] validate performance counters of the MIPS R12000 processor via microbenchmarks. They compare counter results to estimated (simulator-generated) results, but do not investigate the `instructions_graduated` metric (the MIPS equivalent of retired instructions). They report up to 25% error with the `instructions_decoded` counter on long-running benchmarks. This work is often cited as motivation for *why* performance counters should be used with caution.

Maxwell et al. [14] look at accuracy of performance counters on a variety of architectures, including a Pentium III system. They report less than 1% error on the retired instruction metric, but only for microbenchmarks and only on one system. Mathur and Cook [13] look at hand-instrumented versions of nine of the SPEC 2000 benchmarks on a Pentium III. They only report relative error of using sampled versus aggregate counts, and do not investigate overall error. DeRose et al. [6] look at variation and error with performance counters on a Power3 system, but only for startup and shutdown costs. They do not report total benchmark behavior.

3   Experimental Setup

We run experiments on multiple generations of x86 machines, listed in Table 1. All machines run the Linux 2.6.25.4 kernel patched to enable performance counter collection with the perfmon2 [8] infrastructure. We use the entire SPEC CPU 2000 [22] and 2006 [23] benchmark suites with the reference input sets. We compile the SPEC benchmarks on a SuSE Linux 10.1 system with version 4.1 of the gcc compiler and `-O2` optimization (except for `vortex`, which crashes when compiled with optimization). All benchmarks are statically linked to avoid variations due to the C library. We use the same 32-bit, statically linked binaries for all experiments on all machines.

We gather Pin [12] results using a simple instruction count utility via Pin version pin-2.0-10520-gcc.4.0.0-ia32-linux. We patch Valgrind [17] 3.3.0 and Qemu [3] 0.9.1 to generate retired instruction counts. We gather the DBI results on a cluster of Pentium D machines identical to that described in Figure 1. We configure `pfmon` [8] to gather complete aggregate retired instruction counts, without any sampling. The tool runs as a separate process, enabling counting in the OS; it requires no changes to the application of interest and induces minimal overhead during execution. We count user-level instructions specific to the benchmark.

We collect at least seven data points for every benchmark/input combination on each machine and with each DBI method (the one exception is the Core2 machine, which has hardware problems that limit us to three data points for some configurations). The SPEC 2006 benchmarks require at least 1GB of RAM to finish in a reasonable amount of time. Given this, we do not run them on the Pentium Pro or Pentium II, and we do not run `bwaves`, `GemsFDTD`, `mcf`, or `zeusmp` on machines with small memories. Furthermore, we omit results for `zeusmp` with DBI tools, since they cannot handle the large 1GB data segment the application requires.

4   Sources of Variation

We focus on two types of variation when gathering performance counter results. One is inter-machine variations, the differences between counts on two different systems. The other is intra-machine variations, those found when running the same benchmark multiple times on the same system. We investigate methods for reducing both types.

4.1   The `fldcw` instruction

For instruction counts to match on two machines, the instructions involved must be counted the same way. If not, this can cause large divergences in total counts. On Pentium 4 systems, the `instr_retired:nbogusntag` per-

Processor	Speed	Bits	Memory	L1 I/D Cache	L2 Cache	Retired Instruction Counter / Cycles Counter
Pentium Pro	200MHz	32	256MB	8KB/8KB	512KB	inst_retired cpu_clk_unhalted
Pentium II	400MHz	32	256MB	16KB/16KB	512KB	inst_retired cpu_clk_unhalted
Pentium III	550MHz	32	512MB	16KB/16KB	512KB	inst_retired cpu_clk_unhalted
Pentium 4	2.8GHz	32	2GB	12Kμ/16KB	512KB	instr_retired:nbogusntag global_power_events:running
Pentium D	3.46GHz	64	4GB	12Kμ/16KB	2MB	instr_completed:nbogus global_power_events:running
Athlon XP	1.733GHz	32	768MB	64KB/64KB	256KB	retired_instructions cpu_clk_unhalted
AMD Phenom	2.2GHz	64	2GB	64KB/64KB	512KB	retired_instructions cpu_clk_unhalted
Core Duo	1.66GHz	32	1GB	32KB/32KB	1MB	instructions_retired unhalted_core_cycles
Core2 Q6600	2.4GHz	64	2GB	32KB/32KB	4MB	instructions_retired unhalted_core_cycles

Table 1. Machines used for this study.

benchmark	fldcw instructions	% overcount
482.sphinx3	23,816,121,371	0.84%
177.mesa	6,894,849,997	2.44%
481.wrf	1,504,371,988	0.04%
453.povray	1,396,659,575	0.12%
456.hmmer retro	561,271,823	0.03%
175.vpr place	405,499,739	0.37%
300.twolf	379,247,681	0.12%
483.xalancbmk	358,907,611	0.03%
416.gamess cytosine	255,142,184	0.02%
435.gromacs	230,286,959	0.01%
252.eon kajiya	159,579,683	0.15%
252.eon cook	107,592,203	0.13%

Table 2. Dynamic count of `fldcw` instructions, showing all benchmarks with over 100 million. This instruction is counted as two instructions on Pentium 4 machines but only as one instruction on all other implementations.

formance counter counts `fldcw` as two retired instructions; on all other x86 implementations `fldcw` counts as one. This instruction is common in floating point code: it is used in converting between floating point and integer values. It alone accounts for a significant divergence in the `mesa` and `sphinx3` benchmarks. Table 2 demonstrates occurrences in the SPEC benchmarks where the count is over 100 million. We modify Valgrind to count the `fldcw` instructions, and use these counts to adjust results when presenting Pentium 4 data. It should be possible to use statistical methods to automatically determine which type of opcode causes divergence in cases like this; this is part of ongoing work. We isolated the `fldcw` problem by using a tedious binary search of the `mesa` source code.

4.2 Using the Proper Counter

Pentium 4 systems after the model 6 support a `instr_completed:nbogus` counter, which is more accurate than the `instr_retired:nbogusntag` counter found on previous models. This newer counter does not suffer the `fldcw` problem described in Section 4.1. Unfortunately, all systems do not include this counter; our Pentium D can use it, but our older Pentium 4 systems cannot. This counter is not well documented, and thus it was not originally available within the `perfmon` infrastructure. We contributed counter support that has been merged into the main `perfmon` source tree.

4.2.1 Virtual Memory Layout

It may seem counterintuitive, but some benchmarks behave differently depending on where in memory their data structures reside. This causes much of the intra-machine variation we see across the benchmark suites. In theory, memory layout should not affect instruction count. In practice, both `parser` and `perlbench` exhibit this problem. To understand how this can happen, it is important to understand the layout of virtual memory on x86 Linux. In general, program code resides near the bottom of memory, with initialized and uninitialized data immediately above. Above these is the heap, which grows upward. Near the top of virtual memory is the stack, which grows downward. Above that are command line arguments and environment variables.

Typical programs are insensitive to virtual address assignments for data structures. Languages that allow pointers to data structures make the virtual address space "visible". Different pointer values only affect instruction counts if programs act on those values. Both `parser` and `perlbench` use pointers as hash table keys. Differing table layouts can cause hash lookups to use different num-

bers of instructions, causing noticeable changes in retired instruction counts.

There are multiple reasons why memory layout can vary from machine to machine. On Linux the environment variables are placed above the stack; a differing number of environment variables can change the addresses of local variables on the stack. If the addresses of these local variables are used as hash keys then the size and number of environment variables can affect the total instruction count. This happens with `perlbench`; Mytkowicz et al. [15] document the effect, finding that it causes execution time differences of up to 5%.

A machine's word size can have unexpected effects on virtual memory layout. Systems running in 64-bit mode can run 32-bit executables in a compatibility mode. By default, however, the stack is placed at a higher address to free extra virtual memory space. This can cause inter-machine variations, as local variables have different addresses on a 64-bit machine (even when running a 32-bit binary) than on a true 32-bit machine. Running the Linux command `linux32 -3` before executing a 32-bit program forces the stack to be in the same place it would be on a 32-bit machine.

Another cause of varied layout is due to virtual memory randomization. For security reasons, recent Linux kernels randomize the start of the text, data, bss, stack, heap, and mmap() regions. This feature makes buffer-overrun attacks more difficult, but the result is that programs have different memory address layouts each time they are run. This causes programs (like `parser`) that use heap-allocated addresses as hash keys to have different instruction counts every time. This behavior is disabled system wide by the command:

```
echo 0 >
/proc/sys/kernel/randomize_va_space
```

It is disabled at a per-process level with the `-R` option to the `linux32` command. For our final runs, we use the `linux32 -3 -R` command to ensure consistent virtual memory layout, and we use a shell script to force environment variables to be exactly 422 bytes on all systems.

4.3 Processor Errata

There are built-in limitations to performance counter accuracy. Some are intended, and some are unintentional byproducts of the processor design. Our results for our 32-bit Athlon exhibit some unexplained divergences, leading us to investigate existing errata for this processor [1]. The errata mention various counter limitations that can result in incorrect total instruction counts. Researchers must use caution when gathering counts on such machines.

4.3.1 System Effects

Any Operating System or C library call that returns non-deterministic values can potentially lead to divergences. This includes calls to random number generators; anything involving the time, process ID, or thread synchronizations; and any I/O that might involve errors or partial returns. In general, the SPEC benchmarks carefully avoid most such causes of non-determinism; this would not be the case for many real world applications.

OS activity can further perturb counts. For example, we find that performance counters for all but the Pentium 4 increase once for every page fault caused by a process. This can cause instruction counts to be several thousands higher, depending on the application's memory footprint. Another source of higher instruction counts is related to the number of timer interrupts incurred when a program executes; this is possibly proportional to the number of context switches. The timer based perturbation is most noticeable on slower machines, where longer benchmark run times allow more interrupts to occur. Again, the Pentium 4 counter is not affected by this, but all of the other processors are. In our final results, we account for perturbations due to timer interrupt but not for those related to page faults. There are potentially other OS-related effects which have not yet been discovered.

4.4 Variation from DBI Tools

In addition to actual performance counter results, computer architects use various tools to generate retired instruction counts. Dynamic Binary Instrumentation (DBI) is a fast way to analyze benchmarks, and it is important to know how closely tool results match actual hardware counts.

4.4.1 The `rep` Prefix

An issue with the Qemu and Valgrind tools involves the x86 `rep` prefix. The `rep` prefix can come before string instructions, causing the the string instruction to repeat while decrementing the `ecx` register until it reaches zero. A naive implementation of this prefix counts each repetition as a committed instruction, and Valgrind and Qemu do this by default. This can cause many excess retired instructions to be counted, as shown in Table 3. The count can be up to 443 billion too high for the SPEC benchmarks. We modify the DBI tools to count only the `rep` prefixed instruction as a single instruction, as per the relevant hardware manuals.

4.4.2 Floating Point Rounding

Dynamic Binary Instrumentation tools can make floating point problematic, especially for x86 architectures. Default x86 floating point mode is 80-bit FP math, not commonly

found in other architectures. When translating x86 instructions, Valgrind uses 64-bit FP instructions for portability. In theory, this should cause no problems with well written programs, but, in practice, it occasionally does. The move to SSE-type FP implementations on newer machines decreases the problem's impact, although new instructions may also be sources of variation.

The `art` benchmark. The `art` benchmark uses many fewer instructions on Valgrind than on real hardware. This is due to the use of the "==" C operator to compare floating point numbers. Rounding errors between 80-bit and 64-bit versions of the code cause the 64-bit versions to finish with significantly different instruction counts (while still generating the proper reference output). This is because a loop waiting for a value being divided to fall below a certain limit can happen faster when the lowest bits are being truncated. The proper fix is to update the DBI tools to handle 80-bit floating point properly. A few temporary workarounds can be used: passing a compiler option to use only 64-bit floating point, having the compiler generate SSE rather than x87 floating point instructions, or adding an instruction to the offending source code to force the FPU into 64-bit mode.

The `dealII` benchmark. The `dealII` SPEC CPU 2006 benchmark is problematic for Valgrind, much like `art`. In this case, the issue is more critical: the program enters an infinite loop. It waits for a floating point value to reach an epsilon value smaller than can be represented with 64-bit floating point. The authors of `dealII` are aware of this possibility, since source code already has a `#define` to handle this issue on non-x86 architectures.

benchmark	rep counts	% overcount
464.h264ref sss_main	443,109,753,850	15.7%
464.h264ref fore_main	45,947,752,893	14.2%
482.sphinx3	33,734,602,541	1.2%
403.gcc s04	33,691,268,130	18.8%
403.gcc c-typeck	30,532,770,775	21.7%
403.gcc expr2	26,145,709,200	16.3%
403.gcc g23	23,490,076,359	12.1%
403.gcc expr	18,526,142,466	15.7%
483.xalancbmk	15,102,464,207	1.2%
403.gcc cp-decl	14,936,880,311	13.6%
450.soplex pds-50	11,760,258,188	2.5%
453.povray	10,303,766,848	0.9%
403.gcc 200	10,260,100,762	6.1%

Table 3. Potential excesses in dynamic counted instructions due to the `rep` prefix (only benchmarks with more than 10 billion are shown).

4.4.3 Virtual Memory Layout

When instrumenting a binary, DBI tools need room for their own code. The tools try to keep layout as close as possible to what a normal process would see, but this is not always possible, and some data structures are moved to avoid conflicts with memory needed by the tool. This leads to perturbations in the instruction counts similar to those exhibited in Section 4.2.1.

5 Summary of Findings

Figure 1 shows the coefficient of variation for SPEC CPU 2000 benchmarks before and after our adjustments. Large variations in `mesa`, `perlbmk`, `vpr`, `twolf`, and `eon` are due to the Pentium 4 `fldcw` problem described in Section 4.1. Once adjustments are applied, variation drops below 0.0006% in all cases. Figure 2 shows similar results for SPEC CPU 2006 benchmarks. Larger variations for `sphinx3` and `povray` are again due to the `fldcw` instruction. Once adjustments are made, variations drop below 0.002%. Overall, the CPU 2006 variations are much lower than for CPU 2000; the higher absolute differences are counterbalanced by the much larger numbers of total retired instructions. These results can be misleading: a billion-instruction difference appears small in percentage terms when part of a three trillion instruction program, but in absolute terms it is large. When attempting to capture phase behavior accurately using SimPoint with an interval size of 100 million instructions, a phase's being offset by one billion instructions can alter final results.

5.1 Intra-machine results

Figure 3 shows the standard deviations of results across the CPU 2000 and CPU 2006 benchmarks for each machine and DBI method. DBI results are shown, but not incorporated into standard deviations. In all but one case the standard deviation improves, often by at least an order of magnitude. For CPU 2000 benchmarks, `perlbmk` has large variation for every generation method. We are still investigating the cause. In addition, the Pin DBI tool has a large outlier with the `parser` benchmark, most likely due to issues with consistent heap locations. Improvements for CPU 2006 benchmarks are less dramatic, with large standard deviations due to high outlying results. On AMD machines, `perlbench` has larger variation than on other machines, for unknown reasons. The `povray` benchmark is an outlier on all machines (and on the DBI tools); this requires further investigation. The Valgrind DBI tool actually has worse standard deviations after our methods are applied due to a large increase in variation with the `perlbench` benchmarks. For the CPU 2006 benchmarks, similar platforms
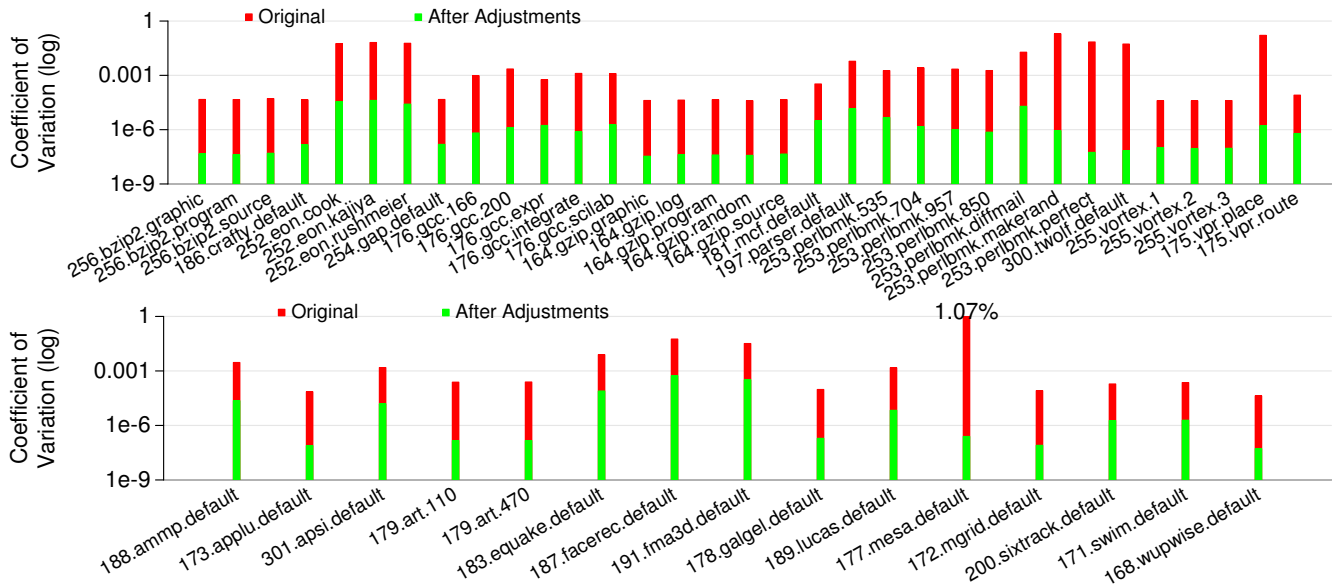
Figure 1. SPEC 2000 Coefficient of variation. The top graph shows integer benchmarks, the bottom, floating point. The error variation from `mesa`, `perlbmk`, `vpr`, `twolf` and `eon` are primarily due to the `fldcw` miscount on the Pentium 4 systems. Variation after our adjustments becomes negligible.
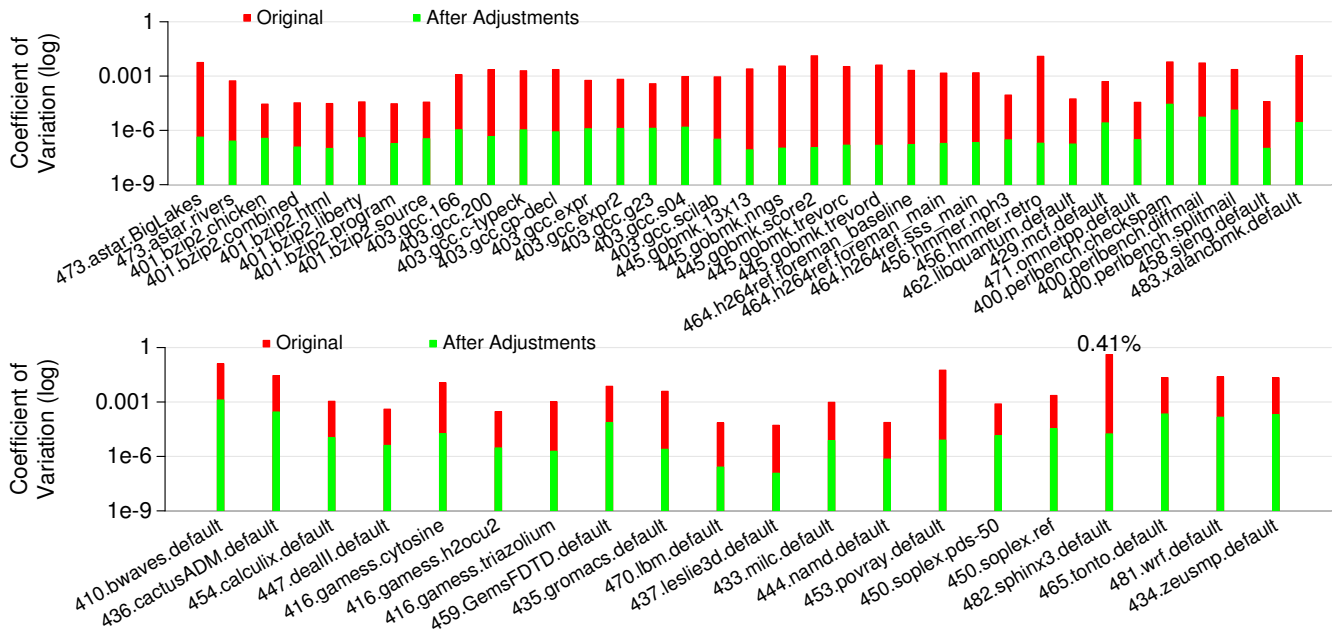

Figure 2. SPEC 2006 Coefficient of variation. The top graph shows integer benchmarks, bottom, floating point. The original variation is small compared to the large numbers of instructions in these benchmarks. The largest variation is in `sphinx3`, due to `fldcw` instruction issues. Variation after our adjustments becomes orders of magnitude smaller.
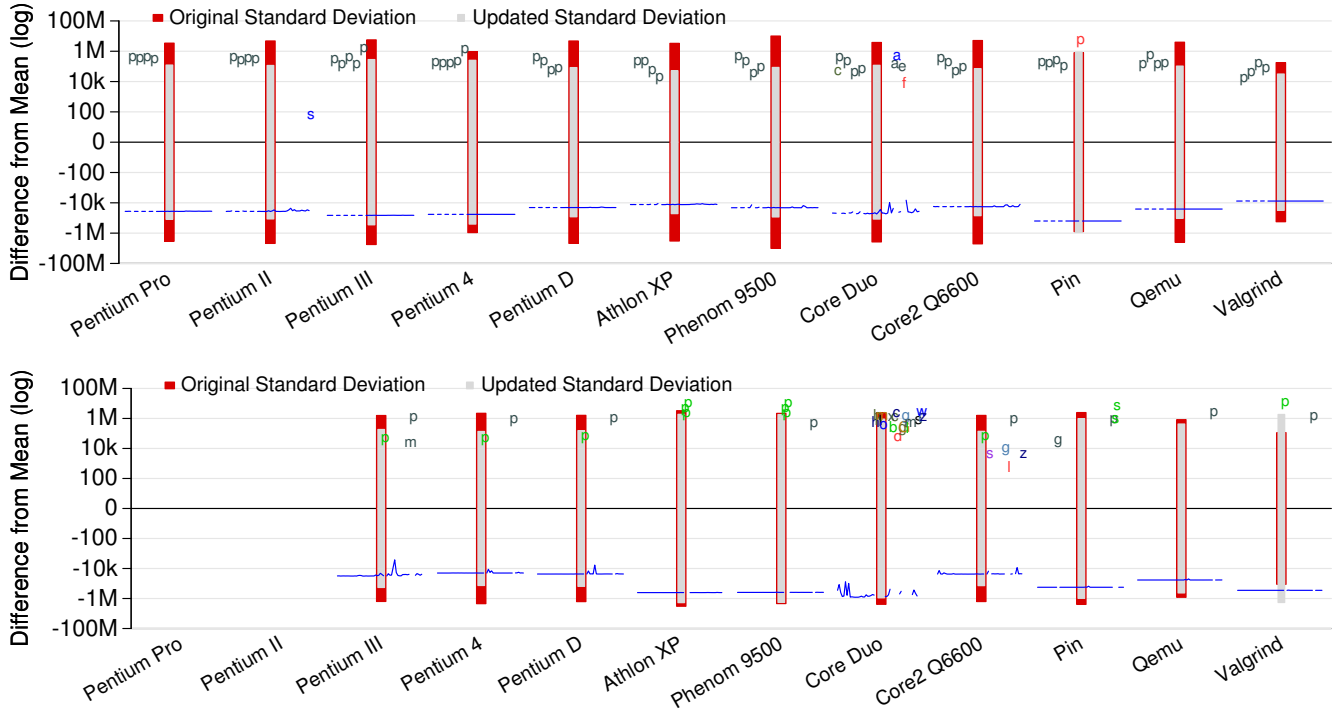
Figure 3. Intra-machine results for SPEC CPU 2000 (above) and CPU 2006 (below). Outliers are indicated by the first letter of the benchmark name and a distinctive color. For CPU 2000, the `perlbmk` benchmarks (represented by grey 'p's) are a large source of variation. For CPU 2006, the `perlbench` (green 'p') and `povray` (grey 'p') are the common outliers. Order of plotted letters for outliers has no intrinsic meaning, but tries to make the graphs as readable as possible. Horizontal lines summarize results for remaining benchmarks (they're all similar). The message here is that most platforms have few outliers, and there's much consistency with respect to measurements across benchmarks; Core Duo and Core2 Q6600 have many more outliers, especially for SPEC 2006. Our technical report provides detailed performance information — these plots are merely intended to indicate trends. Standard deviations decrease drastically with our updated methods, but there is still room for improvement.

have similar outliers: the two AMD machines share outliers, as do the two Pentium 4 machines.

5.2 Inter-machine Results

Figure 4 shows results for each SPEC 2000 benchmark (DBI values are shown but not incorporated into standard deviation results). We include detailed plots for five representative benchmarks to show individual machine contributions to deviations. (Detailed plots for all benchmarks are available in our technical report [25].) Our variation-reduction methods help integer benchmarks more than floating point. The Pentium III, Core Duo and Core 2 machines often over-count instructions. Since they share the same base design, this is probably due to architectural reasons. The Athlon frequently is an outlier, often under-counting.

DBI results closely match the Pentium 4's, likely because the Pentium 4 counter apparently ignores many OS effects that other machines cannot.

Figure 5 shows inter-machine results for each SPEC 2006 benchmark. These results have much higher variation than the SPEC 2000 results. Machines with the smallest memories (Pentium 3, Athlon, and Core Duo) behave similarly, possibly due to excessive OS paging activity. The Valgrind DBI tool behaves poorly compared to the others, often overcounting by at least a million instructions.

6 Conclusions and Future Work

Even though originally included in processor architectures for hardware debugging purposes, when used correctly, performance counters can be used productively for
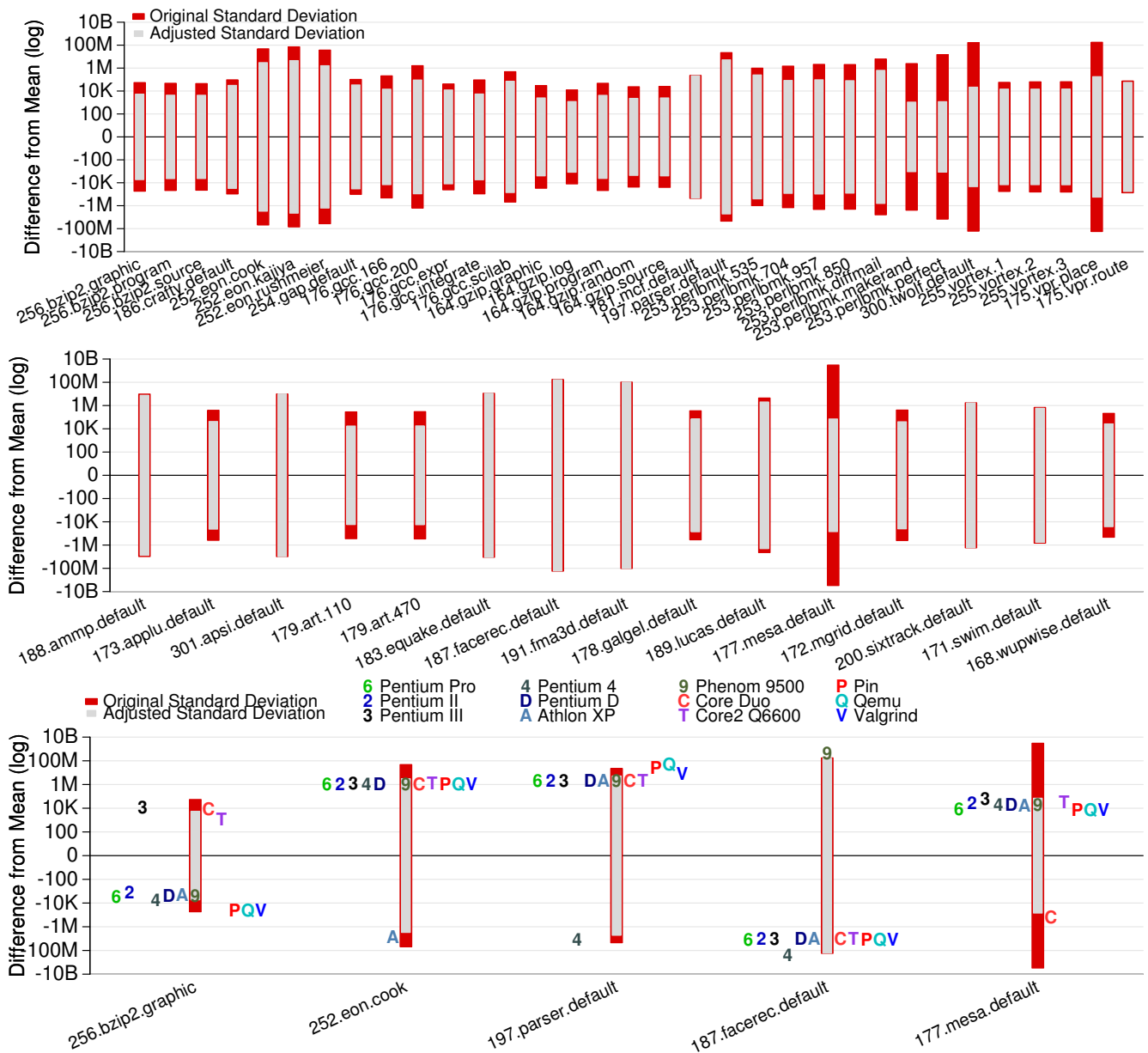
Figure 4. Inter-machine results for SPEC CPU 2000. We choose five representative benchmarks and show the individual machine differences contributing to the standard deviations. Often there is a single outlier affecting results; the outlying machine is often different. DBI results are shown, but not incorporated into standard deviations.

Figure 5. Inter-machine results for SPEC CPU 2006. We choose five representative benchmarks and show the individual machine differences contributing to the standard deviations. Often there is a single outlier affecting results; the outlying machine is often different. DBI results are shown, but not incorporated into the standard deviations.

many types of research (as well as application performance debugging). We have shown that with some simple methodology changes, the x86 retired instruction performance counters can be made to have a coefficient of variation of less than 0.002%. This means that architecture research using this particular counter can reasonably be expected to reflect actual hardware behavior. We also show that our results are consistent across multiple generations of processors. This indicates that older publications using these counts can be compared to more recent work.

Due to time constraints, several unexplained variations in the data still need to be explored in more detail. We have studied many of the larger outliers, but several smaller, yet significant, variations await explanation. Here we examine only SPEC; other workloads, especially those with significant I/O, will potentially have different behaviors. We also only look at the retired instruction counter; processors have many other useful counters, all with their own sets of variations. Our work is a starting point for single-core performance counter analysis. Much future work remains involving modern multi-core workloads.

Acknowledgments

References

[1] Advanced Micro Devices. *AMD Athlon Processor Model 6 Revision Guide*, 2003.

[2] T. Austin. Simplescalar 4.0 release note. http://www.simplescalar.com/.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Apr. 2005.

[4] B. Black, A. Huang, M. Lipasti, and J. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proc. IEEE International Conference on Computer Design*, pages 478–485, Oct. 1996.

[5] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Lueh. XTREM: A power simulator for the intel XScale core. In *Proc. ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 115–125, 2004.

[6] L. DeRose. The hardware performance monitor toolkit. In *Proc. 7th International Euro-Par Conference*, pages 122–132, Aug. 2001.

[7] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in multiprocessor simulation. In *Proc. 28th IEEE/ACM International Symposium on Computer Architecture*, pages 266–277, June 2001.

[8] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[10] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 281–296, 2005.

[11] W. Korn, P. J. Teller, and G. Castillo. Just how accurate are performance counters? In *20th IEEE International Performance, Computing, and Communication Conference*, pages 303–310, Apr. 2001.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, June 2005.

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

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

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We have it easy, but do we have it right? In *NSF Next Generation Systems Workshop*, pages 1–5, Apr. 2008.

[16] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Time interpolation: So many metrics, so few registers. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, 2007.

[17] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[18] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proc. IEEE/ACM 37th Annual International Symposium on Microarchitecture*, pages 81–93, Dec. 2004.

[19] D. Penry, D. August, and M. Vachharajani. Rapid development of a flexible validated processor model. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, pages 21–30, June 2005.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[21] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

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

[23] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2006/, 2006.

[24] V. Weaver and S. McKee. Are cycle accurate simulations a waste of time? In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, June 2008.

[25] V. Weaver and S. McKee. Can hardware performance counters be trusted? Technical Report CSL-TR-2008-1051, Cornell University, Aug. 2008.

[26] V. Weaver and S. McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. In *Proc. 3rd International Conference on High Performance Embedded Architectures and Compilers*, pages 305–319, Jan. 2008.