Can Hardware Performance Counters Produce Expected, Deterministic Results?

Vincent M. Weaver and Jack Dongarra
Innovative Computer Laboratory
University of Tennessee
{vweaver1,dongarra}@eecs.utk.edu

Abstract

Experiments involving hardware performance counters would ideally have deterministic results when run in strictly controlled environments. In practice counters that should be deterministic (such as retired instructions) show variation from run to run on the x86_64 architecture. This causes difficulties when undertaking certain performance-counter related tasks, such as simulator validation and performance analysis. These variations also impede software-based deterministic thread-interleaving, useful for debugging and tuning multi-threaded workloads on modern CMP systems.

We investigate a variety of x86_64 implementations (including DBI tools) and discover the sources of variations from expected count totals. The largest impact on retired instruction totals is due to the inclusion of hardware interrupt counts. This is difficult to compensate for, limiting the utility of the counters. In addition, counts generated by specific instructions can be counted differently across implementations, leading to cross-machine variations in aggregate counts.

We briefly investigate ARM, IA64, POWER and SPARC systems and find that on these platforms the counts do not include hardware interrupts. Non-deterministic limitations to counter use may be a particular feature of the x86_64 architecture.

We also apply our methodology to larger programs and find that run-to-run variation can be minimized, but it is difficult to determine known "good" reference counts for comparison.

1 Introduction

Before using performance counters, it is important to understand the limitations of the underlying hardware. A naïve user might expect deterministic results: repeated identical program runs should give repeatable results. In practice this is nearly impossible to achieve.

It is well known that most programs run on modern computing systems will behave in a non-deterministic manner. This is usually due to hardware effects passed on to applications by the operating system. Even benchmarks designed to give repeatable results, such as SPEC CPU, can accidentally expose operating system behaviors [23].

Many important hardware events, such as cache and cycle counts, are by definition not deterministic on modern out-of-order machines [17]. There are three qualities we look for when evaluating whether an architecture has an available useful deterministic event:

- It does not change run-to-run due to the microarchitecture of the processor,
- It is obvious from code investigation what the expected count should be, and
- The count happens with enough frequency to be useful in program analysis.

In general the only events capable of deterministic behavior are the various retired instruction counters, as these (in theory) depend solely on the underlying ISA and should be stable run-to-run and across machines. Even these retired instruction events exhibit variation on typical applications, due to:

- Operating system interaction [16],
- Program layout [23, 17],
- Measurement overhead [27],
- Multi-processor variation [1], and
- Hardware implementation details [23, 21].

In our experiments we carefully control our environment to avoid these sources of variability.

Many users of performance counters can tolerate variation in the results. Often, due to the small number of counters available, sampling or multiplexing is used during performance measurements. This adds significant run-to-run variation; in this situation the determinism of the underlying counter hardware does not matter. In other cases the inherent non-determinism of the application is so large that it overshadows any limitation of the counting infrastructure.

Despite the common case of not caring about determinism, there are situations where obtaining exact counts is important. Benchmarks used when validating architectural simulators against real hardware should in theory give exact, identical results [22, 9]. Repeatable analysis involving basic block vectors (BBVs) requires identical results, as divergences of just one instruction are enough to change the basic blocks chosen [24]. Feedback Directed Optimization (FDO) works best when performance samples match exactly to the code being executed [7]. Performance comparison between systems with the same instruction set architecture (ISA) using the instructions per cycle (IPC) metric requires that the instruction count match as closely as possible between machines. Debugging and analyzing multithreaded applications can be difficult without some manner of deterministic threading to ensure repeatable program behavior; various implementations of this require a deterministic performance event [19, 26, 3, 5]. Certain methods of intrusion analysis that involve logging and replaying of system behavior also work best if a deterministic performance event is available [10].

This paper is not concerned about the determinism of applications or the operating system, but whether the hardware performance counter subsystem itself can be used under ideal conditions to obtain deterministic results. Processor vendors will make no guarantees about determinism or counter accuracy; therefore we must study the limitations of the counters so that we can use them with confidence in future research.

2 Experimental Setup

Analysis of hardware performance counter accuracy is difficult: it requires exact knowledge of all executing instructions. This precludes using existing benchmarks written in high level languages as the resulting binaries are compiler dependent and there is not a "known" correct instruction count. Compilers also rarely use the full complement of available opcodes, leaving many unexplored corner cases. Even though some divergences in counts might be visible in the total aggregate values, the root causes of the divergence can be nearly impossible to discover, depending on complex interactions deep within a program.

We avoid the variation problems inherent in high-level benchmarks by writing a large custom benchmark in assembly language. This initial test case is an elaborate microbenchmark with over 200 million dynamic instructions. This is larger than the interval size used in many computer architecture investigations. The benchmark attempts to exercise the full x86_64 ISA while having no outside dependencies (by calling OS syscalls directly, much like the techniques used by Weaver and McKee [25]).

Due to the CISC nature of the architecture it is difficult to make a completely comprehensive test. We exercise most integer, x87 floating point, MMX, and SSE instructions (up to and including SSE3). We attempt to use various combinations of register accesses, operand sizes (8 through 128 bit), memory accesses, and the wide variety of x86 addressing modes. Sections of the code are looped many thousands of times to make anomalies stand out in the overall instruction count and to allow binary searches for extra counts. We do not test the effects of more subtle potential causes of variation, such as crossing cache-line boundaries, crossing page boundaries, causing unaligned instruction fetches, unaligned memory accesses, etc., as our experiments show these have minimal effect on event counts. The source to the benchmark is available from our website.

Table 1. Machines used in this study.

Processor	Kernel
Intel Atom 230	2.6.32 perf events
Intel Core2 T9900	2.6.32 perf events
Intel Nehalem X5570	2.6.31 perf events
Intel Nehalem-EX X7560	2.6.34 perf events
Intel Pentium D	2.6.28 perfmon2
AMD Phenom 9500	2.6.29 perfmon2
AMD Istanbul 8439	2.6.32 perf events

We ran our assembly benchmark 10 times each on 7 different x86_64 machines as shown in Table 1. Due to circumstances beyond our control the the test machines are running different Linux kernel revisions. We find that the different kernel and performance measurement infrastructures have no impact on the results. We use the `perf` tool on systems that support the `perf_events` interface, and the `pfmon` tool systems using perfmon2 [11] enabled kernels.

The `perf` tool only has built-in named support for a small subset of events; others have to be specified using a raw event code. We use the libpfm4 library to determine these codes. We run `perf` as such:

```
perf stat -e r5001c0:u ./retired_instructions
```

The `0x5001c0` code corresponds to `RETIRED_LOADS` on Core2 processors and the `:u` mask specifies we only care about user-space (not kernel) counts.

The `pfmon` utility included with perfmon2 has a much more user-friendly interface that uses proper event names. It is run like this:

```
pfmon -e RETIRED_LOADS ./retired\_instructions
```

A full list of available counters can be found in the architectural manuals available from the various vendors [12, 2].

We compare the results of our benchmarks against an "expected" value determined via code inspection. We also run more limited tests on additional architectures.

Modern processors have hundreds of available performance events; we limit our search to those described as

Table 2. Events used in this paper. Values in parenthesis are raw events used by the `perf` utility

Event	Intel Atom	Intel Core2	Intel Nehalem / Nehalem-EX	Intel Pentium D	AMD Phenom / Istanbul
Retired Instructions	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTR_RETIRED:NBOGUSNTAG *(instructions:u)*	RETIRED_INSTRUCTIONS *(instructions:u)*
Retired Branches	BRANCH_INSTRUCTIONS_RETIRED *(branches:u)*	BRANCH_INSTRUCTIONS_RETIRED *(branches:u)*	BRANCH_INSTRUCTIONS_RETIRED *(branches:u)*	BRANCH_RETIRED :MMNP:MMNM:MMTP:MMTM *(branches:u)*	RETIRED_BRANCH_INSTRUCTIONS *(r5000c2:u)*
Retired Loads	n/a	INST_RETIRED:LOADS *(r5001c0:u)*	MEM_INST_RETIRED :LOADS *(r50010b:u)*	FRONT_END_EVENT:NBOGUS, UOPS_TYPE:TAGLOADS	n/a
Retired Stores	n/a	INST_RETIRED:STORES *(r5002c0:u)*	MEM_INST_RETIRED :STORES *(r50020b:u)*	INSTR_RETIRED:NBOGUSTAG, UOPS_TYPE:TAGSTORES	n/a
Multiplies	MUL:AR *(r508112:u)*	MUL *(r510012:u)*	ARITH:MUL *(r500214:u)*	n/a	DISPATCHED_FPU :OPS_MULTIPLY *(r500200:u)*
Divides	DIV:AR *(r508113:u)*	DIV *(r510013:u)*	ARITH:DIV *(r1d40114:u)*	n/a	n/a
FP1	X87_COMP_OPS_EXE :ANY_AR *(r508110:u)*	FP_COMP_OPS_EXE *(r500010:u)*	FP_COMP_OPS_EXE:X87 *(r500110:u)*	EXECUTION_EVENT:NBOGUS1, X87_FP_UOP:ALL:TAG1	RETIRED_MMX_AND_FP_INSTRUCTIONS:X87 *(r5001cb:u)*
FP2	n/a	X87_OPS_RETIRED:ANY *(r50fec1:u)*	INST_RETIRED:x87 *(r5002c0:u)*	n/a	RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL *(r5007cb:u)*
SSE	SIMD_INST_RETIRED *(r501fc7:u)*	SIMD_INSTR_RETIRED *(r5000ce:u)*	FP_COMP_OPS_EXE :SSE_FP *(r500410:u)*	EXECUTION_EVENT:NBOGUS2, PACKED_SP_UOP:ALL:TAG2, PACKED_DP_UOP:ALL:TAG2	RETIRED_SSE_OPERATIONS:ALL *(r507f03:u)*
Retired Uops	UOPS_RETIRED *(r5010c2:u)*	UOPS_RETIRED *(r500fc2:u)*	UOPS_RETIRED:ANY *(r5001c2:u)*	UOPS_RETIRED:NBOGUS	RETIRED_UOPS *(r5000c1:u)*
Hardware Interrupts	HW_INT_RCV* *(r5100c8:u)*	HW_INT_RCV *(r5000c8:u)*	HW_INT:RCV *(r50011d:u)*	n/a	INTERRUPTS_TAKEN *(r5000cf:u)*

*This counter does not appear to work, tested on an Atom N270 and an Atom 230.

counting retired or committed instructions. In general the following types of retired instruction counts are available on most processors:

- **Overall retired instructions**. This event is available and well tested on most architectures. Weaver and McKee [23] find upward of 2% error on 32-bit x86 with SPEC CPU.
- **Retired branches**
- **Retired loads and stores**. Olszewski et al. [19] found retired stores to be the sole deterministic counter on Core 2 systems.
- **Retired multiplies and divides**.
- **Retired μops** Unfortunately μops are implementation dependent.
- **Retired floating point and SSE** There is no standard way of counting FP or SSE. The instructions counted vary, as does whether full ops or μops are counted.
- **Miscellaneous** Many processors provide retired counts of unusual instructions, such as `fxch`, `cpuid`, move operations, serializing instructions, memory barriers, and not-taken branches. While these are useful when analyzing specific program bottlenecks, they are less useful for large-scale validation work.

Table 2 lists the names of the events for which we present detailed results. We do not present results for various uncommon events, events that always return zero (or fail in other ways), or events that turn out to be speculative.

3 Evaluation

We first look at results found using our assembly microbenchmark on x86_64. We then look at other architectures to see if the same limitations apply. We analyze methods for mitigating the variations in counts. Finally we attempt to apply our methodology to larger more realistic benchmarks.

3.1 Assembly Benchmark Results

3.1.1 Retired Instructions

Even a common and well understood event like Retired Instructions can have significant variation, as shown in Table 3. The sources of overcount we found are described below.

Hardware Interrupt Extra Counts Most x86_64 events are incremented an extra time for every hardware interrupt that occurs (the most common hardware interrupt is the periodic timer, causing a noticeable runtime-related overcount). This interrupt behavior was originally undocumented when we first described it, but now appears in some vendor documentation. This overcount is inherently unpredictable, but often can be measured with an additional performance event which allows adjustment of total aggregate results.

Instruction Double-counts All x86 processors count FP instructions containing the `fwait` prefix as two instructions (this makes sense, as "single" instructions containing an implicit `fwait` are actually pseudo-ops for the individual instructions. The Pentium D `INSTRUCTIONS_RETIRED:NBOGUSNTAG` event double counts the following instructions: `fldcw`, `fldenv`, `frstor`, `maskmovq`, `emms`, `cvtpd2pi` (mem), `cvttpd2pi` (mem), `sfence`, and `mfence`.

FP Exception Counts On all x86 processors an extra instruction happens if the x87 top-of-stack pointer overflows; care is taken in our benchmark to avoid this condition. The AMD machines overcount by one when `fninit`, `fnsave`, and `fnclex` instructions execute and one of the FP exception status word flags (such as PE or ZE) is set.

Operating System Counts We find an additional count happens when the floating point unit is used for the first time; this is likely due to the lazy FP saving mechanism used by the OS to avoid context-switch overhead for non-floating point applications. Additional counts are generated when a page fault occurs; in general the first time a fresh page of memory is accessed it causes a page fault which counts as an extra instruction.

Pentium D Counts The Pentium D processor has two different events counting retired instructions. The newer (not available on earlier Pentium 4 models) event is `INSTRUCTIONS_COMPLETED:NBOGUS` which behaves like the corresponding event on other processors. The other event, `INSTRUCTIONS_RETIRED:NBOGUSNTAG` is very different. It is not affected by hardware interrupts (unless those interrupts cause a string instruction to re-start). This has the potential to be a deterministic event, however various instructions are counted twice (as mentioned previously).

3.2 Other "Deterministic" Events

Retired Branches Table 4 shows results for retired branches. As with retired instructions, the hardware interrupt and page table counts affect the results. Branch counts include all control flow changes, including syscalls.

On AMD processors, the perf_event preset `branches:u` event counts the wrong value. We supplied a fix which was incorporated into the 2.6.35 kernel; care must be taken to use the proper raw event on kernels before then.

On Core2 processors the `cpuid` instruction also counts as a branch.

Retired Loads Table 5 shows results for retired loads.

Table 3. Retired instructions compared to expected 226,990,030. AMD due to FP instr when PE set. Pentium D due to instr counted twice.

Machine	Before Adjustment	Adjusted
Core2	10,879±319	13±1
Atom	11,601±495	-41±12
Nehalem	11,409±3	8±2
Nehalem-EX	11,915±9	8±2
Pentium D *(inst retired)*	2,610,571±8	561±3
Pentium D *(inst completed)*	10,794±28	-50±5
Phenom	310,601±11	12±0
Istanbul	311,830±78	11±1

Table 4. Retired branches compared to expected 9,240,001. Core2 due to `cpuid` overcount. Roughly 10k overcount due to HW interrupt/page-faults.

Machine	Before Adjustment	Adjusted
Core2	111,002±332	13±1
Atom	11,542±11	-43±4
Nehalem	11,409±4	8±1
Nehalem-EX	11,914±7	8±1
Pentium D	10,773±2	-56±5
Phenom	10,598±5	9±0
Istanbul	11,819±10	8±2

Table 5. Retired load differences compared to expected value of 79,590,000. Over/under due to instructions being counted twice or not at all.

Machine	Before Adjustment	Adjusted
Core2	1,710,807±376	14±1
Atom	—	—
Nehalem	-288,590±3	9±1
Nehalem-EX	-288,086±7	8±3
Pentium D	2,402,843,955±12	3096±17
Phenom	—	—
Istanbul	—	—

Table 6. Retired store differences compared to expected value of 24,060,000. Overcounts due to instruction double-counts.

Machine	Before Adjustment	Adjusted
Core2	0±0	0±0
Atom	—	—
Nehalem	411,408±4	9±1
Nehalem-EX	411,914±6	9±1
Pentium D	163,402,604±185	11,776±175
Phenom	—	—
Istanbul	—	—

Table 7. Retired μops, multiplies, and divides. These values vary greatly from machine to machine.

Machine	μops	Multiplies	Divides
Core2	14,234,856,824± 8,926	15,800,049± 68	5,800,016± 33
Atom	12,651,163,475± 63,870	13,700,000± 0	7,000,000± 0
Nehalem	11,746,070,128±258,282	19,975,243± 1202	3,125,067± 48
Nehalem-EX	11,746,732,506± 47,996	8,514,161±758,870	3,246,165±9,162
Pentium D	12,551,781,963± 4,601	n/a	n/a
Phenom	10,550,974,722± 36,819	69,242,930± 62,492	n/a
Istanbul	10,551,189,637±139,283	69,796,975±219,398	n/a

Table 8. Retired FP, MMX and SSE instructions.

Machine	FP1	FP2	SSE
Core2	72,600,239±187	39,099,997±0	23,200,000± 0
Atom	38,800,000± 0	n/a	88,299,597± 792
Nehalem	50,150,590±131	17,199,998±2	24,200,849± 154
Nehalem-EX	50,155,704±562	17,199,998±2	24,007,005± 197,401
Pentium D	100,400,310±413	n/a	54,639,963±4,943,158
Phenom	26,600,001± 0	112,700,001±0	15,800,000± 0
Istanbul	26,600,001± 0	112,700,001±0	15,800,000± 0

This event is not supported on all of the processors we investigate.

Extra loads are counted on exceptions: first floating point usage, page faults, x87 FPU exceptions and SSE exceptions. Conditional move instructions will *always* register a load from memory, even if the condition is not met. The `fbstp` "store 80-bit BCD" instruction counts as a load. The `cmps` string compare instruction (where two values from distinct memory are loaded and then compared) counts as only being a single load.

On Core2 machines the `leave` instruction counts as two loads. The `fstenv`, `fxsave`, and `fsave` floating point state-save instructions also count as loads. The `maskmovq` and `maskmovdqu` count loads even though they only write to memory. The `movups`, `movupd` and `movdqu` instructions count as loads even if their operands indicate a store-to-memory operation.

On Nehalem processors the `paddb`, `paddw`, and `paddd` do *not* count as load operations even if the their operands indicate a load from memory.

The Pentium D event has very complicated limitations. The counter exposes internal state, apparently counting the microcoded loads separately. Unlike other x86 processors, software prefetches *are not* counted as loads and page faults count as 5 loads total. Push of a segment (fs/gs), `movdqu` (load), `lddqu`, `movupd` (load), and `fldt` all count as two loads instead of one. `fldenv` counts as 7 loads, `frstor` counts as 23 loads, and `fxrstor` counts as 26. The `movups` (store) instruction counts as a load. The `fstps` instruction counts as two (not zero) loads.

Unlike other x86 counters that treat a rep-prefixed string instruction as a single atomic instruction, on Pentium D the loads are broken out and counted separately, sometimes at a cache-line granularity. The `rep lods` and `rep scas` instructions count each repeated load individually. The `rep movs` instructions performs the moves in blocks of 16-bytes, then goes one-by-one for the remainder (see Figure 1). The `rep cmps` instruction counts each compare instruction as two individual loads.

Retired Stores Table 6 shows results for retired stores. This is an important event: the corresponding Core2 event was found by Olszewski et al. [19] to be deterministic and we have reproduced this result.

All processors (with the exception of Core2) also count hardware interrupts and page faults as a store.

On Nehalem processors the `cpuid`, `sfence`, and `mfence` instructions all count as stores (these are all serializing instructions). `clflush` also counts as a store.

As with retired loads, the Pentium D processor has elaborate retired store behavior that likely exposes internal microcode behavior. As with Nehalem, the `cpuid`, `sfence`, `mfence` and `clflush` instructions count as a stores. The `enter` instruction counts an extra store for each nested stack frame. The `fbstp`, `fstps`, `fstpt`, `movups` (store), `movupd` (store), `movdqu` (store), and `maskmovdqu` instructions counts as two stores. The `fstenv` instruction counts as seven stores, `fsave` as 23 and `fxsave` as 25. The `rep stos` string instruction counts stores in 16B blocks (unless going backwards where it's individual). The `rep movs` instruction counts stores in 16B blocks.

Multiplies and Divides Table 7 shows the numbers of multiplies and divides for each processor. Some of these counts are obviously speculative; the documentation for the counters is not always clear. The implementation of this event varies from model to model; some count integer only, some count floating point and SSE too, and some count multiple times for one instruction. On Core2 `divq` (64-bit divide) instructions count also as a multiply and `mulq` (64-bit multiplies) count as two. These instructions are possibly rare enough in some codes that they may not be useful.

Floating Point and SSE Table 8 shows results for various floating point, MMX and SSE events. Some of these events appear to be deterministic, most notably the events on the AMD machines. Unfortunately these events are hard to predict via code inspection. Some events are retired, some speculative; some count retired instructions, some count retired μops. Some instruction count only math instructions, some count any sort of instruction where floating point is involved. Comparisons between machines will not work due to these variations, and these events are not useful for obtaining deterministic counts on integer-only benchmarks.

μops Table 7 shows the number of retired μops for each processor. Unfortunately μop behavior is implementation specific and cannot be relied on when comparing different machines.

3.3 Other Architectures

In addition to the x86_64 architecture, we investigate other architectures to see if they have similar limitations with regard to determinism. We use the `ll` assembly benchmark [25] modified to repeat 10,000 times. This is not as comprehensive as the test used for x86, but should catch any obvious issues (such as hardware interrupts being counted).

ARM We count retired instructions on an ARM Cortex-A8 core. Unfortunately the performance counters on this architecture cannot select only user-space events; kernel events are always counted too, which makes all of the available events non-deterministic.

IA64 On a Merced system the `STORES_RETIRED`, `LOADS_RETIRED`, and `IA64_INST_RETIRED` counters appear to be deterministic.

POWER On a POWER6 system we find the `instructions:u` count to be deterministic, but

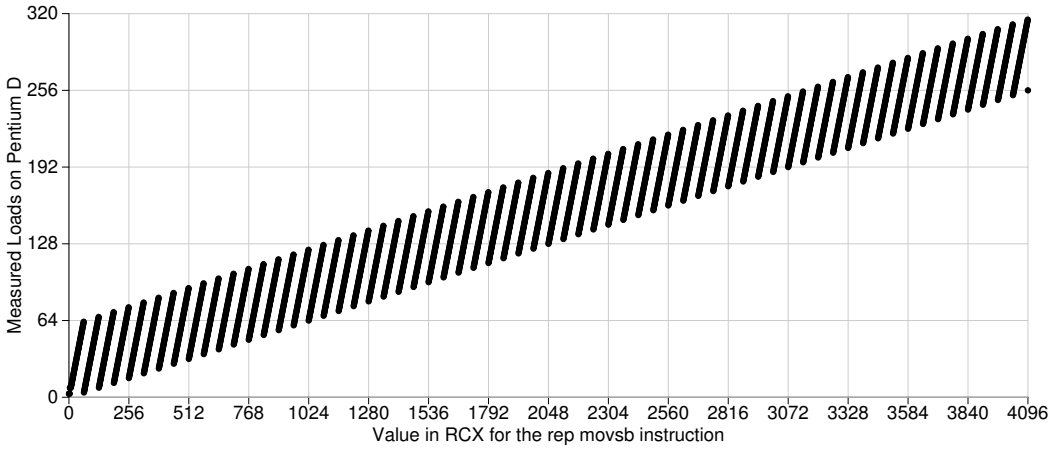

Figure 1. On Pentium D, the `rep movs` string instruction counts retired loads based on 16-byte chunks moved, plus individual loads for any remainder.

the `branches:u` count is not.

SPARC Finally, on a SPARC Niagara T-1 system we find that the `INSTR_CNT` event is deterministic.

3.4 Compensating for Overcounts

Once the limitations to deterministic events are known, attempts can be made to compensate for them.

For aggregate counts, the compensation factors can be applied at the end. Thus for retired instructions if counters are available that measure hardware interrupts and page faults, then an adjusted count can be created that matches expected values. Some implementations (such as Atom and Pentium D) do not have a built-in hardware interrupt event. It is possible to work around this under Linux by reading the `/proc/interrupts` file before and after a benchmark run, but this adds additional error and counts interrupts that happen outside of process context. Events that double-count instructions are not possible to adjust using solely hardware methods, in our experiments we use DBI instrumentation to determine the instruction mix.

Compensation becomes even more difficult when using hardware counters in overflow mode, such as wanting the hardware to signal an interrupt after exactly 1million retired instructions (this is what is wanted for deterministic threading implementations). One workaround for this is described in the ReVirt project [10]; they set the counter to underflow at a value before the value wanted, then adjust the count to be accurate and then slowly single step the program until the desired count occurs.

3.5 DBI Tools

In addition to counts generated by the hardware counters, we also investigate results attainable with dynamic binary instrumentation (DBI) tools. These tools are often used in program analysis, and it is beneficial if their results match actual hardware.

We use Pin [14] version 2.8-33586 to generate the reference results in this study. We also evaluate the exp-bbv and cachegrind tools that come with Valgrind [18] version 3.6 and a current git checkout of Qemu [4] that is patched to generate instruction statistics.

The current versions of all the tools by default report repeated string instructions as having a count equivalent to the times repeated. Actual hardware reports a repeated string instruction as only one instruction. Without modifying the code base, neither Pin, Valgrind, nor Qemu will match the hardware counters.

We have modified the tools to take this into account, and for Pin the results for the assembly benchmark match the expected values and real hardware exactly. We were unable to fully evaluate Valgrind as it currently does not handle numerous infrequent instructions that are not generated by gcc but generated by our test. Qemu works well, but the patches needed for it to generate counts are intrusive and complicated.

3.6 Full-sized benchmarks

We attempt to use our methods to get deterministic runs of the SPEC CPU 2000 [20] benchmarks. We compile these programs statically using gcc 4.3 and the `-O3 -sse3`

Table 9. Results for retired instructions on SPEC CPU 2000.

Benchmark	Pin Results	Counter Results	Difference
164.gzip.graphic	65,982,806,258+/-0	65,985,332,330+/-9	2,526,072
164.gzip.log	27,630,471,231+/-0	27,630,661,869+/-297	190,638
164.gzip.program	134,182,216,830+/-0	134,184,158,711+/-25	1,941,881
164.gzip.random	50,551,063,959+/-0	50,553,651,410+/-241	2,587,451
164.gzip.source	63,534,557,188+/-0	63,534,886,361+/-711	329,173
168.wupwise	360,553,377,202+/-0	360,553,378,908+/-175	1,706
171.swim	211,144,484,205+/-0	211,145,870,699+/-235	1,386,494
172.mgrid	317,894,840,723+/-0	317,902,191,070+/-37	7,350,347
173.applu	329,639,819,901+/-0	329,639,964,577+/-135	144,676
175.vpr.place	91,801,778,868+/-0	91,801,906,033+/-48	127,165
175.vpr.route	65,840,452,950+/-0	65,842,333,845+/-65	1,880,895
176.gcc.166	26,039,501,852+/-0	26,053,619,535+/-69	14,117,683
176.gcc.200	69,280,861,993+/-0	69,333,288,826+/-106	52,426,833
176.gcc.expr	7,253,042,753+/-71	7,257,808,289+/-43	4,765,536
176.gcc.integrate	7,594,306,527+/-0	7,598,639,195+/-69	4,332,668
176.gcc.scilab	38,687,677,208+/-12	38,718,412,887+/-127	30,735,679
177.mesa	224,909,291,041+/-0	225,141,328,681+/-36	232,037,640
178.galgel	265,298,711,252+/-0	265,315,417,293+/-91	16,706,041
179.art.110	37,455,717,089+/-0	37,684,112,743+/-46	228,395,654
179.art.470	41,559,174,782+/-0	41,815,556,622+/-70	256,381,840
181.mcf	47,176,435,708+/-0	47,178,182,387+/-41	1,746,679
183.equake	91,830,166,829+/-0	91,831,754,253+/-486	1,587,424
186.crafty	140,410,682,095+/-0	140,491,624,577+/-46	80,942,482
187.facerec	249,446,706,530+/-0	249,466,271,565+/-20	19,565,035
188.ammp	282,267,674,633+/-0	282,273,791,341+/-85	6,116,708
189.lucas	205,650,970,148+/-0	205,650,971,675+/-54	1,527
191.fma3d	252,617,528,064+/-0	252,621,707,010+/-130	4,178,946
197.parser	263,198,435,420+/-0	263,268,978,039+/-227	70,542,619
200.sixtrack	542,747,136,304+/-0	542,751,505,285+/-13	4,368,981
252.eon.cook	59,410,255,668+/-144	59,432,884,285+/-211	22,628,617
252.eon.kajiya	79,522,489,405+/-92	79,548,194,010+/-119	25,704,605
252.eon.rushmeier	46,636,612,121+/-577	46,652,449,863+/-73	15,837,742
253.perlbmk.535	2,696,610,456+/-2	2,698,843,490+/-199	2,233,034
253.perlbmk.704	2,764,426,301+/-4	2,766,432,903+/-243	2,006,602
253.perlbmk.850	5,655,963,871+/-22	5,661,167,625+/-253	5,203,754
253.perlbmk.957	4,508,337,217+/-2	4,512,393,547+/-203	4,056,330
253.perlbmk.diffmail	30,233,369,642+/-22	30,339,690,700+/-164	106,321,058
253.perlbmk.makerand	1,090,891,857+/-22	1,090,909,156+/-150	17,299
253.perlbmk.perfect	19,657,248,256+/-22	19,666,664,723+/-198	9,416,467
254.gap	183,293,201,373+/-0	183,443,753,693+/-20	150,552,320
255.vortex.1	162,104+/-0	162,215+/-10	111
255.vortex.2	161,905+/-0	162,016+/-10	111
255.vortex.3	162,024+/-0	162,135+/-10	111
256.bzip2.graphic	104,650,996,309+/-0	104,716,216,837+/-399	65,220,528
256.bzip2.program	92,138,659,767+/-0	92,195,366,446+/-283	56,706,679
256.bzip2.source	75,683,045,767+/-0	75,737,142,438+/-309	54,096,671
300.twolf	294,394,181,323+/-0	294,395,384,751+/-203	1,203,428
301.apsi	335,965,776,144+/-0	335,998,221,972+/-190	32,445,828

Table 10. Results for retired stores on SPEC CPU 2000.

Benchmark	Pin Results	Counter Results	Difference
164.gzip.graphic	9,220,255,442+/-0	9,220,318,816+/-1	63,374
164.gzip.log	2,869,442,570+/-0	2,869,475,599+/-2	33,029
164.gzip.program	15,043,298,768+/-0	15,043,347,481+/-0	48,713
164.gzip.random	7,333,288,257+/-0	7,333,345,900+/-1	57,643
164.gzip.source	7,099,846,266+/-0	7,099,884,570+/-1	38,304
168.wupwise	33,509,937,868+/-0	33,509,937,948+/-0	80
171.swim	18,657,590,092+/-0	18,657,604,499+/-0	14,407
172.mgrid	19,780,977,379+/-0	19,780,992,153+/-0	14,774
173.applu	36,944,783,307+/-0	36,944,806,144+/-0	22,837
175.vpr.place	10,506,996,023+/-0	10,507,367,334+/-1	371,311
175.vpr.route	8,498,211,242+/-0	8,498,625,210+/-1	413,968
176.gcc.166	6,126,548,968+/-0	6,126,646,078+/-2	97,110
176.gcc.200	10,809,876,957+/-0	10,810,247,099+/-14	370,142
176.gcc.expr	1,262,579,952+/-14	1,262,641,060+/-4	61,108
176.gcc.integrate	1,472,392,036+/-0	1,472,436,588+/-3	44,552
176.gcc.scilab	6,544,043,598+/-1	6,544,314,779+/-10	271,181
177.mesa	35,256,814,647+/-0	35,256,814,675+/-0	28
178.galgel	25,736,467,292+/-0	25,736,468,525+/-0	1,233
179.art.110	3,467,916,650+/-0	3,467,916,650+/-0	0
179.art.470	3,792,351,365+/-0	3,792,351,365+/-0	0
181.mcf	3,101,673,836+/-0	3,101,673,836+/-0	0
183.equake	6,401,707,007+/-0	6,401,707,013+/-0	6
186.crafty	14,715,329,050+/-0	14,715,329,550+/-0	500
187.facerec	17,108,726,507+/-0	17,175,891,130+/-6	67,164,623
188.ammp	31,435,756,072+/-0	31,435,756,072+/-0	0
189.lucas	18,135,992,918+/-0	18,135,993,050+/-0	132
191.fma3d	42,289,894,809+/-0	42,326,598,083+/-13	36,703,274
197.parser	32,254,247,249+/-0	32,254,090,688+/-0	-156,561
200.sixtrack	24,831,293,048+/-0	24,831,447,915+/-1	154,867
252.eon.cook	9,168,538,965+/-10	9,168,538,925+/-21	-40
252.eon.kajiya	12,616,424,674+/-5	12,616,424,618+/-39	-56
252.eon.rushmeier	7,321,524,013+/-47	7,321,523,805+/-0	-208
253.perlbmk.535	502,744,026+/-0	502,853,217+/-1	109,191
253.perlbmk.704	515,446,194+/-1	515,464,538+/-0	18,344
253.perlbmk.850	1,077,046,593+/-2	1,077,124,158+/-1	77,565
253.perlbmk.957	853,729,475+/-0	853,824,516+/-0	95,041
253.perlbmk.diffmail	5,192,919,547+/-2	5,192,873,218+/-0	-46,329
253.perlbmk.makerand	188,774,998+/-2	188,774,884+/-1	-114
253.perlbmk.perfect	3,498,063,997+/-2	3,498,435,094+/-0	371,097
254.gap	25,380,689,015+/-0	25,380,688,751+/-0	-264
255.vortex.1	22,413+/-0	22,405+/-0	-8
255.vortex.2	22,403+/-0	22,395+/-0	-8
255.vortex.3	22,410+/-0	22,402+/-0	-8
256.bzip2.graphic	14,992,496,929+/-0	14,992,496,932+/-0	3
256.bzip2.program	12,378,627,404+/-0	12,378,627,408+/-0	4
256.bzip2.source	8,647,185,380+/-0	8,647,185,382+/-0	2
300.twolf	30,735,278,724+/-0	30,735,278,725+/-0	1
301.apsi	39,722,966,049+/-0	39,722,972,988+/-0	6,939

compiler options. We run on a Core2 machine with a perf_events enabled kernel.

Care is made to turn off address layout randomization and attempt to set the environment up in an exacting way previously shown to minimize run-to-run variation [23]. Despite this care, some variation is caused by the Pin DBI tool, as it adds various environment variables.

Table 9 shows results for retired instructions on each benchmark, with the reference Pin result, the adjusted measured value, and the difference between the two. Likewise, Table 10 shows results for retired stores.

The results show large divergences which are still under investigation. A first glance might indicate that there are roughly 5 extra instructions for every `malloc()`. In addition slight difference in the alignment of the stack cause the glibc `strlen()` and other string instructions to follow different paths and accumulate extra instruction counts.

4 Related Work

While many studies use hardware performance counters, there has been little research into the accuracy of the counts. Our work is unique in looking at a wide range of architectures and a wide variety of modern 64-bit machines, as well as determining correctness based on code inspection rather than using a simulator.

Zaparanuks et al. [27] investigate the accuracy of the cycle count on various x86 processors and Mytkowicz et al. [17] investigate sources of non-determinism in run-time. Neither of these metrics are defined by the ISA to be deterministic so it is expected that they vary.

Korn, Teller, and Castillo [13] validate performance counters of the MIPS R12000 processor via microbenchmarks, reporting up to 25% error with `instructions_decoded` when comparing against a hardware simulator. Black et al. [6] investigate the number of retired instructions and cycles on the PowerPC 604 platform, comparing their results against a cycle-accurate simulator. Cycle-accurate simulators have their own inherent error, so unless that is known exactly it limits what can be learned about the accuracy of the hardware counters being compared.

Our previous work [23] investigates the determinism of the RETIRED_INSTRUCTION counter on a wide range of 32-bit x86 processors using the SPEC CPU benchmarks; while this found many sources of variation it was limited to one event and did not fully determine the causes of non-determinism.

Maxwell et al. [15] look at accuracy of performance counters on a variety of architectures, reporting less than 1% error with retired instructions when using a microbenchmark. DeRose et al. [8] 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.

Olszewski et al. [19], while attempting to create a user-space deterministic multi-threading library, find that RETIRED_STORES is deterministic on Core2 processors. They do not describe their methodology for how this was determined, nor do they look at any other architectures. Bergan et al. [5] use retired instructions while doing deterministic multi-threading; they use the methodology of Dunlap et al. [10] which used retired branches on AMD machines but stopped early and single-stepped to avoid hardware interrupt issues. All of these works need deterministic counters to work, but they do not comment in detail on the relative strengths of the counters they end up using.

5 Conclusions and Future Work

Most potentially deterministic events on x86_64 are affected by the hardware interrupt count. This severely limits the usefulness of events in situations where exact deterministic behavior is necessary. We were able to compensate for the counts after the fact, but this is complicated and not possible in certain situations. The only widely-available non-floating point deterministic event found on any of the x86_64 systems investigated was `RETIRED_STORES` on the Core2.

Our investigation of non-x86 architectures shows that deterministic events are possible and that the hardware interrupt count issue is a limitation particular to x86 systems. Ideally the various chip vendors can improve the handling of deterministic events in future processor revisions.

We have started to apply our methodology to larger benchmarks; so far the results are inconclusive. Initial results seem to show that the differences in counts are due to non-determinisms in the applications rather than further issues with the hardware.

We hope to extend our work to the study of non-deterministic events, such as cycle counts and cache events. This type of event is more difficult to analyze; this initial study of deterministic events provides confidence in the underlying hardware.

Despite the fact that hardware counters are used extensively for performance analysis, chip designers remain reluctant to make guarantees about the accuracy of the counts. It would be advantageous to researchers and other users of the counts if deterministic events could be made available on x86 hardware.

Acknowledgements

We would like to thank Heechul Yun, Dan Terpstra, and the PAPI team their valuable feedback on this paper.

This work was supported in part by the U.S. Department of Energy Office of Science under contract DE-FC02-06ER25761, by the National Science Foundation under Grant No. 0910899, as well as Software Development for Cyberinfrastructure (SDCI) Grant No. NSF OCI-0722072 Subcontract No. 207401.

References

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded commercial workloads. In *Proc. 9th IEEE Symposium on High Performance Computer Architecture*, 2003.

[2] AMD. *AMD Family 10h Processor BIOS and Kernel Developer Guide*, 2009.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proc. 9th USENIX Symposium on Operating System Design and Implementation*, Oct. 2010.

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

[5] T. Bergan, N. Hunt, L. Ceze, and S. Gribble. Deterministic process groups in dOS. In *Proc. 9th USENIX Symposium on Operating System Design and Implementation*, Oct. 2010.

[6] 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.

[7] D. Chen, N. Vachharajani, R. Hundt, S.-W. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for FDO compilation. In *Proc. 8th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 42–53, Apr. 2010.

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

[9] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2001.

[10] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. 5th USENIX Symposium on Operating System Design and Implementation*, Dec. 2002.

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

[12] Intel. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2009.

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

[14] 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.

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

[16] N. McGuire, P. Okech, and G. Schiesser. Analysis of inherent randomness of the Linux kernel. In *Proc. 11th Real-Time Linux Workshop*, 2009.

[17] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

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

[19] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

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

[21] V. Weaver. *Using Dynamic Binary Instrumentation to Create Faster, Validated, Multi-core Simulations*. PhD thesis, Cornell University, May 2010.

[22] V. Weaver and S. McKee. Are cycle accurate simulations a waste of time? In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pages 40–53, June 2008.

[23] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Proc. IEEE International Symposium on Workload Characterization*, pages 141–150, Sept. 2008.

[24] 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.

[25] V. Weaver and S. McKee. Code density concerns for new architectures. In *Proc. IEEE International Conference on Computer Design*, pages 459–464, Oct. 2009.

[26] H. Yun. DPTHREAD: Deterministic multithreading library, 2010.

[27] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32, Apr. 2009.