Extended version of ISPASS 2013 paper – updated 18 March 2021

Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations – Extended

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

Dan Terpstra
University of Tennessee
terpstra@icl.utk.edu

Shirley Moore
University of Texas at El Paso
svmoore@utep.edu

March 18, 2021

Extended version of ISPASS 2013 paper – updated 18 March 2021

Abstract

Ideal hardware performance counters provide exact deterministic results. Real-world performance monitoring unit (PMU) implementations do not always live up to this ideal. Events that should be exact and deterministic (such as retired instructions) show run-to-run variation and overcount on x86_64 machines, even when run in strictly controlled environments. These effects are non-intuitive to casual users and cause difficulties when strict determinism is desirable, such as when implementing deterministic replay or deterministic threading libraries.

We investigate eleven different x86_64 CPU implementations and discover the sources of divergence from expected count totals. Of all the counter events investigated, we find only a few that exhibit enough determinism to be used without adjustment in deterministic execution environments. On these machines it appears that the `iret` interrupt return instruction counts as a userspace instruction despite running in kernel space, which keeps the counts from being deterministic. The deterministic events we find are those that do not count for `iret`, such as conditional branches and retired stores.

We also briefly investigate ARM, IA64, POWER and SPARC systems and find that on these platforms the counter events have more determinism.

We explore various methods of working around the limitations of the x86_64 events, but in many cases this is not possible and would require architectural redesign of the underlying PMU.

1 Introduction

Most modern CPUs have hardware performance counters; these counters allow detailed, low-level measurement of processor behavior. The counters are most commonly used for performance analysis, especially in the High Performance Computing (HPC) field. Usage has spread to the desktop and embedded areas, with many new and novel utilization scenarios.

There are a wide variety of events that can be measured with performance counters, with event availability varying considerably among CPUs and vendors. Some processors provide hundreds of events; separating the useful and accurate ones from those that are broken and/or measure esoteric architectural minutia can be a harrowing process. Event details are buried in architectural manuals, often accompanied by disclaimers disavowing any guarantees of useful results.

Counter validation is a difficult process. Some events cannot be validated effectively, as they require exact knowledge of the underlying CPU architecture and can be influenced by outside timing effects not under user control [1]. This includes most cache events, cycle counts, and any event affected by speculative execution.

A subset of events exists that *is* architecturally specified; these measure various kinds of retired instructions. With a deterministic program (one that when provided with the same input traverses the exact same code path and generates the exact same output) the counter results should be the same for

every run. These counts should be consistent; otherwise the processor would not be ISA compatible with others in the same architecture.

1.1 The Need for Deterministic Events

There are many situations where deterministic software execution is necessary. Deterministic execution is useful when validating architectural simulators [2, 3], when analyzing program behavior using basic block vectors (BBVs) [4], when performing Feedback Directed Optimization (FDO) [5], when using hardware checkpointing and rollback recovery [6], when performing intrusion analysis [7], and when implementing parallel deterministic execution [31].

Parallel deterministic execution enables debugging and analysis of multi-threaded applications in a repeatable way. Deterministic lock interleaving makes it possible to track down locking problems in large parallel applications. There have been many proposals about how to best implement parallel deterministic execution; many require modified hardware or modified operating systems. A quick and easy way to build deterministic locks is to use hardware performance counters to ensure that previously non-deterministic lock behavior happens in a consistent, repeatable way [8, 9, 10, 11]. The need for parallel deterministic execution has been the primary impetus for the search for deterministic performance events.

1.2 Definitions

In this work we search for *useful deterministic events*. We define a useful deterministic event as one where the value does not change run-to-run due to the microarchitecture of the processor (it is not affected by speculative execution), the expected value can be determined via code inspection, and the event occurs with enough frequency and distribution to be useful in program analysis.

We find two primary causes for events to deviate from the expected result: *nondeterminism* (identical runs returning different values) and *overcount* (some instructions counting multiple times). We investigate both sources of deviation.

2 Experimental Setup

Processor vendors make no guarantees about determinism or counter accuracy; any limitations must be determined experimentally. We investigate multiple x86_64 implementations to see if any of the performance events can provide deterministic events with no overcount, suitable for applications such as parallel deterministic execution. We also investigate the availability of such events on other platforms.

2.1 External Sources of Non-Determinism

Measuring exact event counts can be difficult due to various external sources of variation found in a typical system, including operating system interac-

Table 1: Events used in this paper (part 1). Values in parenthesis are `perf` raw event numbers.

	Intel Atom	Intel Core2	Intel Pentium D
Retired Instructions	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTR_RETIRED:NBOGUSNTAG *(instructions:u)*
Retired Branches	BRANCH_INSTRUCTIONS _RETIRED *(branches:u)*	BRANCH_INSTRUCTIONS _RETIRED *(branches:u)*	BRANCH_RETIRED :MMNP:MMNM:MMTP:MMTM *(branches:u)*
Retired Conditional Branches	n/a	BR_CND_EXEC *(r53008b:u)*	RETIRED_BRANCH_TYPE :CONDITIONAL
Retired Loads	n/a	INST_RETIRED:LOADS *(r5001c0:u)*	FRONT_END_EVENT:NBOGUS, UOPS_TYPE:TAGLOADS
Retired Stores	n/a	INST_RETIRED:STORES *(r5002c0:u)*	INSTR_RETIRED:NBOGUSTAG, UOPS_TYPE:TAGSTORES
Multiplies	MUL:AR *(r508112:u)*	MUL *(r510012:u)*	n/a
Divides	DIV:AR *(r508113:u)*	DIV *(r510013:u)*	n/a
FP1	X87_COMP_OPS_EXE:ANY_AR *(r508110:u)*	FP_COMP_OPS_EXE *(r500010:u)*	EXECUTION_EVENT:NBOGUS1, X87_FP_UOP:ALL:TAG1
FP2	X87_COMP_OPS_EXE:ANY_S *(r530110:u)*	X87_OPS_RETIRED:ANY *(r50fec1:u)*	n/a
SSE	SIMD_INST_RETIRED *(r501fc7:u)*	SIMD_INSTR_RETIRED *(r5000ce:u)*	EXECUTION_EVENT:NBOGUS2, PACKED_SP_UOP:ALL:TAG2, PACKED_DP_UOP:ALL:TAG2
Retired Uops	UOPS_RETIRED *(r5010c2:u)*	UOPS_RETIRED *(r500fc2:u)*	UOPS_RETIRED:NBOGUS
Hardware Interrupts	HW_INT_RCV† *(r5100c8:u)*	HW_INT_RCV *(r5000c8:u)*	n/a

†This counter does not work on Atom N270 or 230.

Table 2: Events used in this paper (part 2). Values in parenthesis are `perf` raw event numbers.

	Intel Nehalem Intel Nehalem-EX Intel Westmere	Intel Sandybridge	Intel IvyBridge
Retired Instructions	INSTRUCTIONS_RETIRED *(instructions:u)*	INSTRUCTION_RETIRED *(instructions:u)*	INSTRUCTION_RETIRED *(instructions:u)*
Retired Branches	BRANCH_INSTRUCTIONS _RETIRED *(branches:u)*	BRANCH_INSTRUCTIONS _RETIRED *(branches:u)*	BR_INST_RETIRED *(branches:u)*
Retired Conditional Branches	BR_INST_RETIRED :CONDITIONAL *(r5301c4:u)*	BR_INST_RETIRED :CONDITIONAL *(r5301c4:u)*	BR_INST_RETIRED:COND *(r5301c4:u)*
Retired Loads	MEM_INST_RETIRED :LOADS *(r50010b:u)*	MEM_UOP_RETIRED :ANY_LOADS *(r5381d0:u)*	MEM_UOPS_RETIRED :ALL_LOADS *(r5381d0:u)*
Retired Stores	MEM_INST_RETIRED :STORES *(r50020b:u)*	MEM_UOP_RETIRED :ANY_STORES *(r5382d0:u)*	MEM_UOPS_RETIRED: :ALL_STORES *(r5382d0:u)*
Multiplies	ARITH:MUL *(r500214:u)*	n/a	UOPS_ISSUED :SINGLE_MUL *(r53400e:u)*
Divides	ARITH:DIV *(r1d40114:u)*	ARITH:FPU_DIV *(r1570114:u)*	ARITH:FPU_DIV *(r1570114:u)*
FP1	FP_COMP_OPS_EXE:X87 *(r500110:u)*	FP_COMP_OPS_EXE:X87 *(r5302c0:u)*	FP_COMP_OPS_EXE:X87[†] *(r5302c0:u)*
FP2	INST_RETIRED:X87 *(r5002c0:u)*	INST_RETIRED:X87 *(r5302c0:u)*	Undocumented Used SandyBridge event *(r5302c0:u)*
SSE	FP_COMP_OPS_EXE :SSE_FP *(r500410:u)*	FP_COMP_OPS_EXE :SSE_DOUBLE_PRECISION *(r538010:u)*	FP_COMP_OPS_EXE :SSE_DOUBLE_PRECISION[†] *(r538010:u)*
Retired Uops	UOPS_RETIRED:ANY *(r5001c2:u)*	UOPS_RETIRED:ANY *(r5301c2:u)*	UOPS_RETIRED:ALL *(r5301c2:u)*
Hardware Interrupts	HW_INT:RCV[‡] *(r50011d:u)*	HW_INTERRUPTS :RECEIVED *(r5301cb:u)*	HW_INTERRUPTS *(r5301cb:u)*

[‡] Event support dropped in 6/2010 Intel Vol3B, interacts poorly with HyperThreading

[†] Documentation for this event not added by Intel until September 2013.

Table 3: Events used in this paper (part 3). Values in parenthesis are `perf` raw event numbers.

	AMD Phenom / Istanbul	AMD fam14h
Retired Instructions	RETIRED_INSTRUCTIONS *(instructions:u)*	RETIRED_INSTRUCTIONS *(instructions:u)*
Retired Branches	RETIRED_BRANCH _INSTRUCTIONS *(r5000c2:u)*	RETIRED_BRANCH _INSTRUCTIONS *(branches:u)*
Retired Conditional Branches	n/a	n/a
Retired Loads	n/a	n/a
Retired Stores	n/a	n/a
Multiplies	DISPATCHED_FPU :OPS_MULTIPLY *(r500200:u)*	RETIRED_SSE_OPERATIONS: SINGLE_MUL_OPS: DOUBLE_MUL_OPS *(r531203:u)*
Divides	n/a	RETIRED_SSE_OPERATIONS: SINGLE_DIV_OPS: DOUBLE_DIV_OPS *(r524003:u)*
FP1	RETIRED_MMX_AND_ FP_INSTRUCTIONS:X87 *(r5001cb:u)*	RETIRED_FLOATING_ POINT_INSTRUCTIONS *(r5303cb:u)*
FP2	RETIRED_MMX_AND_ FP_INSTRUCTIONS:ALL *(r5007cb:u)*	DISPATCHED_FPU: ANY *(r530300:u)*
SSE	RETIRED_SSE_ OPERATIONS:ALL *(r507f03:u)*	RETIRED_SSE_ OPERATIONS:ALL *(r537f03:u)*
Retired Uops	RETIRED_UOPS *(r5000c1:u)*	RETIRED_UOPS *(r5000c1:u)*
Hardware Interrupts	INTERRUPTS_TAKEN *(r5000cf:u)*	INTERRUPTS_TAKEN *(r5300cf:u)*

Table 4: Events used in this paper (part 4). Values in parenthesis are `perf` raw event numbers.

	Intel Haswell
Retired Instructions	INSTRUCTION_RETIRED *(instructions:u)*
Retired Branches	BR_INST_RETIRED *(branches:u)*
Retired Conditional Branches	BR_INST_RETIRED :CONDITIONAL *(r5301c4:u)*
Retired Loads	MEM_UOPS_RETIRED :ALL_LOADS *(r5381d0:u)*
Retired Stores	MEM_UOPS_RETIRED: :ALL_STORES *(r5382d0:u)*
Multiplies	UOPS_ISSUED :SINGLE_MUL *(r53400e:u)*
Divides	n/a
FP1	n/a
FP2	n/a
SSE	n/a
Retired Uops	UOPS_RETIRED:ALL *(r5301c2:u)*
Hardware Interrupts	undocumented *(r5301cb:u)*

tion [12], program layout [13, 1], measurement overhead [14], multi-processor variation [15], and hardware implementation details [13, 16]. In our experiments we attempt to avoid these sources of variability by carefully controlling our test environment.

Benchmarks often have internal sources of non-determinism that are inherent in their design, usually unintentionally. If a program depends on the time, pointer values, or I/O input, then the application can take unpredictable paths through its codebase. Even benchmarks designed to give repeatable results, such as SPEC CPU, can vary in subtle ways due to a changing operating system environment [13]. We carefully construct our test-cases to avoid these sources of variation as much as possible.

2.2 Our Custom Assembly Benchmark

Analysis of performance counter accuracy is difficult; it requires exact knowledge of all executing instructions and their effects on a system. This precludes using existing benchmarks written in high level languages as the resulting binaries are compiler dependent and no "known" overall instruction count is available. Compilers rarely use the full complement of available processor opcodes, leaving many unexplored corner cases. Total aggregate event measurements over large benchmarks can make major divergences from estimated totals visible, but the root causes can be nearly impossible to discover. Counts can vary due to complex interactions deep within a program and can be perturbed by debugging.

We avoid the variation inherent in high-level benchmarks by writing a large assembly language benchmark. This microbenchmark has over 200 million dynamic instructions, which is larger than the interval size used in many computer architecture investigations. The benchmark attempts to exercise most x86_64 instructions while having no outside dependencies (by calling operating system syscalls directly, as in our previous code density investigation [17]).

Due to the CISC nature of the x86 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 (single byte accesses up through 128-bit SSE), 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. The complete annotated source for the microbenchmark is available from our website:
`http://www.eece.maine.edu/~vweaver/projects/deterministic/`

We measure userspace events generated by our benchmark alone; the operating system provides process-specific counts by saving and restoring the counter values at context switch time and the CPU performance monitoring unit (PMU) differentiates between events happening in user and kernel domains. There are many other conceivable sources of variation, such as crossing cache-line boundaries, crossing page boundaries, unaligned instruction fetches, unaligned memory accesses, etc. We have not found these to affect event counts.

2.3 Events

Modern processors have hundreds of available performance events (a full list can be found in the various vendor's architectural manuals [18, 19]). We limit our search to those described as counting retired or committed instructions.

In general the following types of retired instruction counts are available:

- **total retired instructions**
- **retired branches** (total or conditional),
- **retired loads and stores**, and
- **retired floating point and SSE**.

In addition, many processors provide retired counts of unusual instructions, such as `fxch`, `cpuid`, move operations, serializing instructions, memory barriers, multiplies and divides, and not-taken branches. While these are useful when analyzing specific program bottlenecks, they are less useful for large-scale validation work. Other retired events, such as retired μops, are unsuitable because they are speculative and implementation dependent.

Tables 1, 2, 3 and 4 list the names of the events for which we provide detailed results.

2.4 The Experiments

We ran our assembly benchmark ten times each on eleven different x86_64 machines as shown in Table 5. We compare the results of our benchmarks against an expected value determined via code inspection. Due to circumstances beyond our control the test machines are running different Linux kernel revisions; we ran tests of various kernels and performance counter implementations on the same machine and found that the different kernel infrastructures have no impact on userspace-only aggregate counter results. We use the `perf` tool on systems that support the `perf_events` interface, and the `pfmon` tool systems using `perfmon2` [20].

The `perf` tool only supports a small number of common "generic" events; many events have to be specified using a raw event code. We use the libpfm4 library to determine these codes. We run `perf` as follows:

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

In this example `r5001c0` corresponds to the Core2 `RETIRED_LOADS` event 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
```

Table 5: Machines used in this study.

Processor	Linux Kernel	
Intel Atom 230	3.2	perf events
Intel Core2 X5355	2.6.36.2	perf events
Intel Nehalem X5570	2.6.38.6	perf events
Intel Nehalem-EX X7550	2.6.32-RHEL6	perf events
Intel Westmere-EX 8870	3.2	perf events
Intel SandyBridge-EP	2.6.32-RHEL6	perf events
Intel IvyBridge i5-3427U	3.2	perf events
Intel Haswell i7-4770	3.13	perf events
Intel Pentium D	2.6.28	perfmon2
AMD Phenom 9500	2.6.29	perfmon2
AMD Istanbul 8439	2.6.35	perf events
AMD fam14h E-350	3.2	perf events
AMD fam15h A10-6800B	3.13	perf events

3 Evaluation

We first look at results found using our assembly micro-benchmark 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 a full benchmark suite.

3.1 Sources of Overcount and Non-Determinism on x86_64

We use our hand-crafted assembly language benchmark to find deviation from the known expected count. We are interested in *nondeterminism* (run-to-run variations) and *overcount* (always-the-same predictable offsets against known event count due to errata in the chip design).

We calculate known total event counts for the various metrics via code inspection, and then validate the expected counts with the Pin [21] dynamic binary instrumentation (DBI) tool. We use a script to gather performance counter totals for each platform; in the common case where counter results do not match expectations we manually comment out parts of the assembly benchmark and re-run until we localize the source of variation.

Table 6 shows a summary of the overcount and nondeterminism found on each system. The actual event totals gathered are not important; they are arbitrary values related to the instruction mix of the benchmark. They key below the table describes the sources of variation, as described below.

Table 6: Summary of sources of nondeterminism and overcount for retired instructions.

	Total Instructions	Total Branches	Conditional Branches	Loads	Stores
Atom	hpEF	hp	–	–	–
Core2	hpEF	hpD	p	hpD	DETERMINISTIC
Nehalem	hpEF	hp	D	hpM	hpD
Westmere	hpEF	hp	DETERMINISTIC	hp	hpD
Sandybridge IvyBridge Haswell	hpEF	hp	DETERMINISTIC	U	U
Pentium D	hpEFD	hp	!	hpU	hpU
Phenom Istanbul fam14h	hpEFD	hp	–	–	–

Sources of nondeterminism:	**h**	Hardware Interrupts
	p	Page Faults
Sources of overcount:	**E**	x87/SSE exceptions
	F	OS Lazy FP handling
	D	Instructions Overcounted
	M	Instructions Undercounted
	U	Counts micro-ops
Missing Results:	–	Event not available
	!	Test not run

Table 7: Retired μops, multiplies, and divides in the microbenchmark; these values vary from machine to machine.

Machine	μops	Multiplies	Divides
Atom	12,650,929,921± 10,048	13,700,000± 0	7,000,000± 0
Core2	14,250,314,285± 38,796	16,300,012± 13	5,800,058± 16
Nehalem	11,746,800,094± 38,192	17,719,572±1,992,446	3,180,368± 7,409
Nehalem-EX	11,746,938,597± 27,708	19,835,890± 215,301	3,265,181±21,966
Westmere-EX	11,740,683,274± 218,900	19,866,413± 196,031	5,800,072± 64
SandyBridge-EP	12,292,221,237± 7,258	n/a	5,800,304± 56
IvyBridge	12,315,297,486±4,669,700	620,550± 17,451	3,244,139±17,414
Haswell	12,128,839,684± 4684	600,024± 6	n/a
Pentium D	12,555,222,761±6,650,825	n/a	n/a
Phenom	10,550,974,722± 36,819	69,242,930± 62,492	n/a
Istanbul	10,557,954,252± 168,608	69,988,147± 317,885	n/a
Fam14h	11,366,903,273± 153,234	1,800,000± 0	2,400,000± 0

Table 8: Retired FP, MMX and SSE instructions in the microbenchmark. These values vary from machine to machine. Some may be deterministic, but cannot be used with integer-only workloads.

Machine	FP1	FP2	SSE
Atom	38,800,000± 0	44,000,221± 341	88,293,855± 70,345
Core2	72,601,258± 215	39,099,997± 0	23,200,000± 0
Nehalem	50,234,437± 6,800	17,199,998± 2	24,203,034± 563
Nehalem-EX	50,230,521± 5,827	17,199,998± 4	24,028,996±222,406
Westmere-EX	50,015,343±43,898	17,199,998± 2	24,921,548± 38,051
SandyBridge-EP	48,784,041± 1,325	17,200,028± 8	23,136,313± 18,585
IvyBridge	49,025,110±37,400	17,200,040± 27	5,434,935± 26,195
Haswell	n/a	n/a	n/a
Pentium D	100,400,310± 413	n/a	28,795,097± 5,662
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
Fam14h	115,199,563± 21	276,217,480±541,728	15,800,000± 0

3.1.1 Nondeterministic Hardware Interrupts

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 variation). This interrupt behavior was originally undocumented when we first described it, but now appears in some vendor documentation. The number of extra events is inherently unpredictable, but often can be measured with an additional "hardware interrupts" event that can be used to adjust the total aggregate results. If an event is affected by hardware interrupts, then it cannot be a deterministic event, as it is impossible to predict in advance when these events will happen.

Another source of interrupts is generated when a page fault occurs; in general the first time a page of memory is accessed it causes a page fault that counts as an extra instruction. This variation is more predictable than other interrupts, but can still be affected by the behavior of the operating system and other programs running on the system.

The source of the overcount remained mysterious for a while, but some discussion on Twitter around 2020 gave a plausible hypothesis. The `iret` interrupt return instruction is documented by Agner Fog as being broken into 4 uops. What likely happens is that some of these uops count as being retired in kernel space, but the tail end gets retired after the kernel/user transition happens, leading to an extra instruction being registered by the performance counters. This also explains why the various deterministic events are retired stores or conditional branches, as neither of those events would count for an `iret` instruction.

3.1.2 Sources of Instruction Overcount

There are various sources that can cause overcount on x86 processors.

On all the systems we tested an extra instruction event happens if the x87 top-of-stack pointer overflows; care is taken in our benchmark to avoid this condition.

An additional count may happen when the floating point unit is used for the first time; this is due to the lazy floating-point save mechanism used by Linux to avoid context-switch overhead for non-floating point applications.

A major source of overcount is when an instruction event is incremented multiple times for a single instruction, or when an instruction is not counted at all. This is likely due to missing terms in the instruction classifying hardware on the PMU.

One last source of overcount is when an event measures microcoded events rather than retired events. Sometimes these events are deterministic, but it is hard to verify because microcode is system dependent and undocumented. Recent counter documentation has gotten much better at indicating which events are architectural instructions and which are microcoded.

Total Retired Instruction Overcount The total retired instructions event is high-profile and often used, but still may be affected by overcount.

While not strictly a source of overcount, some instructions are actually pseudo-instructions and can confuse a user determining expected instruction counts via code inspection. Various x87 floating point instructions have "wait" and "no wait" versions that optionally force execution to wait to see if an exception has occurred. The wait versions are pseudo-ops for instructions with a wait prefix and count twice.

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. Despite being interrupt related, this variation is an overcount because it can be predicted and happens deterministically.

The Pentium D processor has two different retired instruction events. 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 restart). This has the potential to be a deterministic event; however it suffers from overcount with the following instructions: `fldcw`, `fldenv`, `frstor`, `maskmovq`, `emms`, `cvtpd2pi` (mem), `cvttpd2pi` (mem), `sfence`, and `mfence`. The `fldcw` instruction is particularly troublesome as it is a common instruction used when converting floating point values to integers (and it has been shown to cause up to 2% overcount on some SPEC CPU benchmarks [13]).

Retired Branches Overcount The retired branches event counts control flow changes, including system call entry.

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

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

Retired Conditional Branches Overcount Not all processors support counting conditional branches (and we were unable to test on Pentium D as the machine we used for the other results has been decommissioned).

Noll [22] reports that this event is deterministic on SandyBridge; we have verified this result and found that the equivalent event is likewise deterministic on IvyBridge, Haswell, Westmere and Nehalem. The Nehalem event suffers from overcount: in addition to conditional branches (which start with opcode `0F`) many instructions are counted that also start with opcode `0F`, including various non-branch MMX and SSE instructions.

3.1.3 Retired Load Overcount

Retired loads are 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.

Load events are subject to various forms of under and overcount. 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 complicated overcount, likely because it is recording microcoded loads and not architectural loads. Unlike other x86 processors, software prefetches *are not* counted as loads and page faults count as five loads total. Pop of a segment (fs/gs), `movdqu` (load), `lddqu`, `movupd` (load), and `fldt` all count as two loads instead of one. `fldenv` counts as seven 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 the other x86 load events that treat a rep-prefixed string instruction as a single atomic instruction, on Pentium D the loads are 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 moves in blocks of 64-bytes, then goes one-by-one for the remainder (see Figure 1). The `rep cmps` instruction counts each compare instruction as two loads.

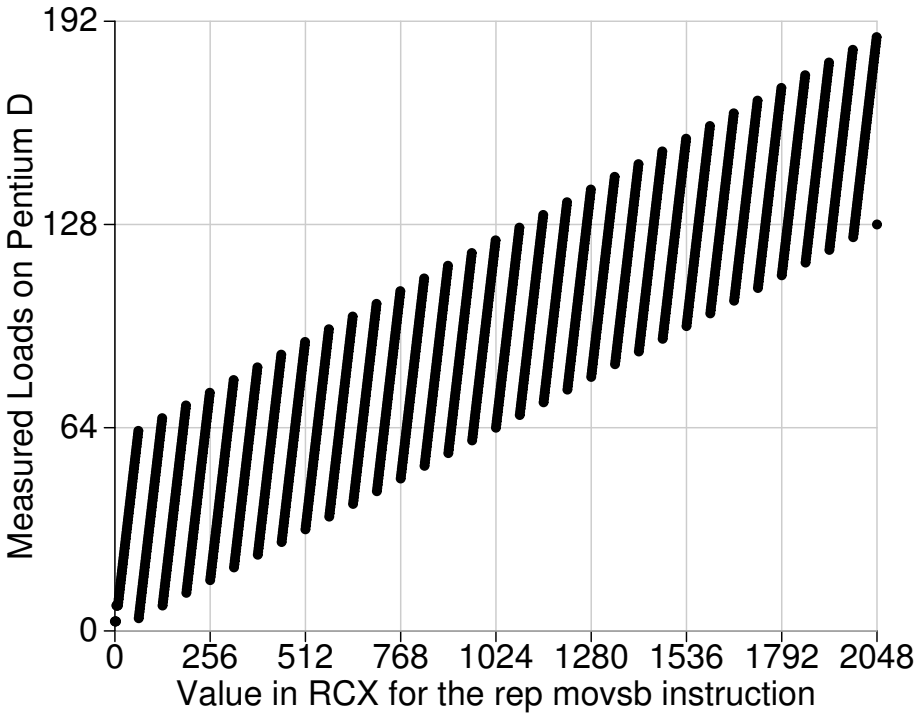

Figure 1: On Pentium D, the retired loads event shows unusual behavior with `rep movs` string instructions. The observed count is related to 64-byte chunks being moved with individual moves for the remainder: $(floor(reps/64) * 4) + (reps\%64)$.

The SandyBridge load event measures load μops so it has limitations similar to Pentium D. On IvyBridge and Haswell the event name for this event was changed to make its μop nature more obvious.

Retired Store Overcount On Core2 processors the retired store event was found by Olszewski et al. [8] to be deterministic with no overcount, and we have reproduced this result. All other processors count hardware interrupts and page faults with store events.

On Nehalem and Westmere 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 storing backwards in memory). The `rep movs` instruction counts stores in 16B blocks.

The SandyBridge, IvyBridge, and Haswell store events measure μops and have similar limitations to Pentium D.

3.1.4 Other Events

Tables 7 and 8 show results from other events that we investigated as possibly being useful deterministic events, but found to have too much microarchitectural variation. Total events are show to give a feel for the variation, for comparison the dynamic total retired event count is 226,990,030. The values shown are the average of ten runs, with the plus/minus value indicating the maximum distance from the average.

Retired μops Despite the "retired" modifier in the event name, μop behavior is nondeterministic as well as implementation specific and cannot be relied on when comparing different machines. The values are almost two orders of magnitude higher than the total instruction count; this is skewed by the fact that repeated string instructions are only counted once by the retired instruction event but counted individually by the μop event.

Multiplies and Divides Table 7 also shows the numbers of multiplies and divides for each processor. Some of these counts are speculative or else count μops; the documentation for the counters is not always clear. The implementation of these events varies from model to model; some count integer only, some also count floating point and SSE, and some count multiple times for one instruction. On Core2 `divq` (64-bit divide) instructions also count as a multiply, and `mulq` (64-bit multiplies) count twice.

On Atom and Fam14h processors the events are deterministic, but these instructions are rare enough in most code that they would likely not be useful in practice.

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 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 would not be useful for obtaining deterministic counts on integer-only benchmarks.

3.2 Other Architectures

In addition to the x86_64 architecture, we investigate other architectures to see if they have similar limitations with regard to determinism.

Creating a detailed microbenchmark like the one used on x86_64 is a long and tedious process, and we do not currently have sufficient resources to do this for every architecture. Instead we use the `ll` assembly benchmark [17] modified to repeat 10,000 times. This is not as comprehensive as the x86_64 test (it does not test every possible opcode so may miss issues with overcount), but should catch any obvious determinism issues (such as hardware interrupts being counted).

ARM We count retired instructions on ARM Cortex-A8 and Cortex-A9 processors. 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 Merced `STORES_RETIRED`, `LOADS_RETIRED`, and `IA64_INST_RETIRED` appear to be deterministic.

POWER On a POWER6 system we find `instructions:u` to be deterministic, but `branches:u` suffers from overcount.

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

4 Compensating for Overcounts

Now that we have determined the factors causing non-determinism and overcount, we investigate if it may be possible to compensate for the limitations and derive deterministic events where there are none.

Overcount on its own does not provide a problem for applications such as deterministic locking. The run-to-run counts will be the same, just different from the expected value. This is only a problem if applying results gathered on one machine to runs on a different one with different level of overcounts. In this case adjusted results can be generated if the exact opcode mix of a program is known; this is usually not possible without extra analysis by an external tool and in general not possible to determine in real time.

Compensating for non-determinism is a more difficult problem. When measuring aggregate totals, a compensation factor can be subtracted at the end of a run. For events that include hardware interrupts, corrected counts can be generated by measuring a hardware interrupt event (if available) and adjusting the total by this count. Many implementations include an event which can be used for this purpose; some CPUs do not (such as Atom or Pentium D) and on some the event is unreliable when HyperThreading is enabled (Nehalem) [23]. When no interrupt event is available it is possible (at least on Linux) to use values from `/proc/interrupts` instead (although this adds additional error and may count interrupts that happen outside of process context).

Compensation becomes more difficult when using hardware counters in over-

flow or sampling mode (as is often used in performance analysis or deterministic threading). Users may want hardware to signal an interrupt after *exactly* one million retired instructions; aggregate compensation methods will not work in this case. One workaround for this is described in the ReVirt project [7]; they set the counter to overflow at a value before the value wanted, adjust the count to be accurate, and then slowly single step the program until the desired count occurs.

4.1 Dynamic Binary Instrumentation Results

To aid in determining expected instruction counts, as well as determining per-opcode instruction frequency, we used various dynamic binary instrumentation (DBI) tools. These tools are used in program analysis and are capable of measuring program execution at a per-instruction level; ideally the counts generated will match actual hardware.

We evaluate Pin [21] version 2.8-33586, the `exp-bbv` and `cachegrind` tools that come with Valgrind [24] version 3.8, and a current git checkout of Qemu [25] that is patched to generate instruction statistics.

Initial results did not match expected values; this is because all of the DBI tools report string instructions with a `rep` repeat prefix as having a count equivalent to the times repeated; this contrasts with real hardware which reports `rep`-prefixed string instruction as only one instruction. We have modified the tools to take this into account, and for Pin the results for the assembly benchmark match the expected values exactly.

We were unable to fully evaluate Valgrind as it currently does not handle numerous infrequent instructions that are not generated by gcc but are generated by our test. Qemu works well, but the patches needed for it to generate counts are intrusive and make it a poor candidate for this type of analysis.

4.2 Full-sized benchmarks

We apply our methods to the SPEC CPU 2000 [26] benchmarks and investigate how much variation is found in "real-world" applications. We compile these programs statically using gcc 4.3 and the `-O3 -sse3` compiler options. We run on a Core2 machine with a perf_event enabled kernel. SPEC CPU 2000 is outdated compared to more recent benchmarks, but it provides enough runtime to show any variations without completely overwhelming analysis with orders of magnitude larger instruction counts.

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 variations [13]. Despite these precautions, 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, which is deterministic on Core2. The results show large divergences that are still under inves-

Table 9: Measured Core2 retired instructions for 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: Measured Core2 retired stores for 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

tigation, although some seem to be related to `malloc()` and `strlen()` being non-deterministic at runtime.

It is extremely difficult to track down the causes of divergences in benchmarks this large, so new methodologies need to be designed to analyze these kinds of problems. This will be even more difficult when analyzing parallel applications.

5 Related Work

The primary use of deterministic events is for parallel deterministic execution and deterministic replay. In these cases any deterministic event will do, and once one is found it tends to be mentioned in passing without discussing the methodology used to analyze the determinism.

Olszewski et al. [8], 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. [11] use retired instructions while doing deterministic multi-threading; they use the methodology of Dunlap et al. [7] which used retired branches on AMD machines but stopped early and single-stepped to avoid hardware interrupt issues.

Many other studies use hardware performance counters in various ways, but there has been little research into deterministic variation or overcount. 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.

Stodden et al. [6] use assembly-language programs to validate use of hardware counters for log-based rollback recovery, but they do not analyze the determinism of the events, only the amount of interrupt lag when trying to stop at a precise instruction address.

Zaparanuks et al. [14] investigate the performance counter accuracy as provided by various high-level counter APIs on three different x86 architectures. They measure overhead of the cycle and total retired instruction events, but use a very small (4 instruction long) assembly benchmark and do not fully explore the underlying causes of the variation.

Mytkowicz et al. [1] investigate sources of measurement bias and non-determinism in program execution. The cycles event was used in this work, and the problems found focused on high-level executable layout and operating system issues and not limitations of the underlying PMU.

Korn, Teller, and Castillo [27] validate MIPS R12000 performance counters with microbenchmarks, reporting up to 25% error with `INSTRUCTIONS_DECODED` when comparing against a hardware simulator. Black et al. [28] 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.

We previously investigate the determinism of the RETIRED_INSTRUCTION counter on a wide range of 32-bit x86 processors using the SPEC CPU benchmarks [13], finding upward of 2% error on Pentium D. This work found many sources of variation but was limited to one event and did not fully explore the causes of non-determinism.

Maxwell et al. [29] 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. [30] 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.

6 Conclusions and Future Work

In our experiments we have found only a small minority of x86_64 events to be deterministic and without overcount: RETIRED_STORES on Core2 and BR_INST_RETIRED_CONDITIONAL on SandyBridge and Westmere. This lack of useful events limits the use of performance counters for advanced applications such as deterministic replay and threading libraries on the popular x86_64 architecture.

Many potentially deterministic events are rendered unusable by including the unpredictable hardware interrupt count. This can be mitigated by subtracting off a separate interrupt counter event (if available), but this will not help in the deterministic use case where exact overflow is desired in order to stop at precise locations.

Our investigation of other architectures shows that deterministic events are more common on non-x86 hardware. This shows that deterministic events can be accomplished and are not an unsolvable problem. Unfortunately these platforms are typically not available to most users.

New users of performance counters are often frustrated that the results they measure are not the ones they "know" to be correct. Eventually the users learn the sources of the error, and undertake analysis that allows for run-to-run variation in the results. It becomes almost a rite of passage, learning why the counters work the way they do, and working around them. This fatalistic view of the quality of counters explains the lack of impetus for fixing the underlying problem.

We propose that there are definite benefits to providing deterministic counters with little overcount or variation. Existing methodologies that can stand some variation will not be harmed, and new and better uses for the counters will be found. Use of counters by non-experts can then be encouraged, as there will be so many fewer caveats to their use.

The various x86_64 vendors need to be strongly encouraged to fix the performance monitoring units on their respective CPUs. There are many inherent hardware problems with providing deterministic counters, but other non-x86 architectures seem to have solved them. This may mean simplifying the available counters or limiting the number of available events, but in practice few people use the counters at all, let alone the full feature set.

A change like this will not happen overnight; In the meantime more work on analyzing the causes and amounts of variations can be done. Manually generating and validating test suites is a slow, tedious process. We are investigating a method of automated testcase generation and validation that can vastly improve the process.

When deterministic counters do become available, they will be welcomed not only by those working on deterministic replay and simulator validators, but also by all users of performance counters.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0910899 and by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research under award number DE-SC0006733.

References

[1] 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.

[2] V. Weaver and S. McKee, "Are cycle accurate simulations a waste of time?" in *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, Jun. 2008, pp. 40–53.

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

[4] 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*, Jan. 2008, pp. 305–319.

[5] 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*, Apr. 2010, pp. 42–53.

[6] D. Stodden, H. Eichner, M. Walter, and C. Trinitis, "Hardware instruction counting for log-based rollback recovery on x86-family processors," in *Proc. 3rd International Service Availability Symposium*, May 2006, pp. 106–119.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] V. Weaver and S. McKee, "Can hardware performance counters be trusted?" in *Proc. IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 141–150.

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

[15] 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.

[16] V. Weaver, "Using dynamic binary instrumentation to create faster, validated, multi-core simulations," Ph.D. dissertation, Cornell University, May 2010.

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

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

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

[20] S. Eranian, "Perfmon2: a flexible performance monitoring interface for Linux," in *Proc. 2006 Ottawa Linux Symposium*, Jul. 2006, pp. 269–288.

[21] 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*, Jun. 2005, pp. 190–200.

[22] A. Noll, Personal Communication, 2011.

[23] C. Segulja, Personal Communication, 2012.

[24] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2007, pp. 89–100.

[25] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annual Technical Conference, FREENIX Track*, Apr. 2005, pp. 41–46.

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

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

[28] 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*, Oct. 1996, pp. 478–485.

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

[30] L. DeRose, "The hardware performance monitor toolkit," in *Proc. 7th International Euro-Par Conference*, Aug. 2001, pp. 122–132.

[31] RR Project, "RR Project Website," https://rr-project.org/, 2021.