Appears in *14th International Symposium on High-Performance Computer Architecture (HPCA '08)*

Serializing Instructions in System-Intensive Workloads: Amdahl's Law Strikes Again

Philip M. Wells and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
{pwells,sohi}@cs.wisc.edu

Abstract

Serializing instructions (SIs)*, such as writes to control registers, have many complex dependencies, and are difficult to execute out-of-order (OoO). To avoid unnecessary complexity, processors often serialize the pipeline to maintain sequential semantics for these instructions.*

We observe frequent SIs across several system-intensive workloads and three ISAs, SPARC V9, X86-64, and PowerPC. As explained by Amdahl's Law, these SIs, which create serial regions within the instruction-level parallel execution of a single thread, can have a significant impact on performance. For the SPARC ISA (after removing TLB and register window effects), we show that operating system (OS) code incurs a 8–45% performance drop from SIs.

We observe that the values produced by most control register writes are quickly consumed, but the writes are often effectively useless *(EU), i.e., they do not actually change the execution of the consuming instructions. We propose EU prediction, which allows younger instructions to proceed, possibly reading a stale value, and yet still execute correctly. This technique improves the performance of OS code by 6–35%, and overall performance by 2–12%.*

1. Introduction

For system-intensive workloads, those that spend a considerable fraction of their time executing OS or hypervisor code, the system code has a significant effect on overall performance. Yet we observe that system code has 50–85% higher *cycles per instruction* (CPI) than user code. Researchers have often noticed this performance discrepancy [1, 7, 9, 25], and shown it to be growing with new processor generations [3, 22]. This discrepancy is often blamed on worse cache locality, TLB, and branch behavior, which is exacerbated by user/system interference in these structures [6]. However, caches and branch predictors do not tell the whole story. We identify *serializing instructions* (SIs), such as those that write control registers and cannot be executed out-of-order (OoO), as an additional, major component of poor OS performance.

OoO processors achieve parallel execution of a sequential program, albeit at the level of instructions and memory operations. SIs introduce a short sequential section into this parallel execution. As Amdahl's Law explains, frequent serialization can greatly limit performance regardless of the amount of parallelism available the rest of the time.

We show that the OS performance impact of SIs rivals the impact of misses to main memory, accounting for 25–60% of the higher system CPI. Several additive trends are increasing the cost and frequency of SIs, including processors which can maintain thousands of instructions in flight, the use of speculative or redundant multithreading, and trap-and-emulate software virtual machines.

Despite being a major performance factor, SIs have received little public attention from academic or industry researchers. This lack of attention is likely because SIs are viewed as specific to a particular processor implementation. In reality, SIs share many common characteristics across of range of ISAs and processors, and thus, we believe deserve a close examination. We aim to explore the origin of SIs in more detail, investigate their impact on performance, and propose solutions to mitigate this impact.

2. Serializing Instructions

Most instructions are defined by the ISA to have sequential semantics, even if the instructions are actually executed out-of-order (OoO). We use the term *serializing instructions* (SIs) to identify instructions that may require sequential semantics, yet make OoO execution difficult due to complex dependencies. A processor implementation is free to implement SIs in a manner that allows OoO execution, but the complexities of doing so are very high and not typically undertaken, due to the belief that these instructions are infrequent. As a result, a processor is often forced to serialize the pipeline in order to execute these instructions.

With the exception of Section 3.5, we focus on a generic OoO processor and the commonalities of SIs across different ISAs, rather than a particular processor implementation.

2.1. What are Serializing Instructions?

SIs fall into three broad categories: 1) instructions that write to non-renamed state, 2) explicit synchronization, and 3) other complex instructions. This paper is primarily concerned with the first category, though we discuss the others as well.

Writes to Non-renamed State The *scope* of a variable in a program refers to the points in the program where that variable has meaning and can be referenced. We use the term *register scope* to indicate which stages of the pipeline a particular register has meaning and can be accessed. Most registers, including general purpose registers and condition codes have very limited scope: they are read and written only at *execute* stage. Some registers, such as control registers, have broad scope because they are visible to, and used by, control logic in many stages in the pipeline.

Registers with broad scope are generally not renamed because of the immense complexity required to deliver correct values to consumer instructions and control logic at a variety of pipeline stages. By not renaming these registers, writes to them cannot execute OoO, and must serialize. The SPARC `wrpr` instruction is an example of an SI when it writes to the `%pstate` register. In Section 3 we provide a list of registers for SPARC, X86-64, and PowerPC that we consider to be non-renamed.

Dynamic instructions that trigger an exception, and instructions such as SPARC's `retry` or X86's `iret` that return from an exception handler, are SIs because they implicitly write several non-renamed registers. Modifications to other non-renamable processor state, such as TLBs, are also SIs, because this state also has broad scope.

Explicit Synchronization ISAs may not require sequential semantics for all pairs of instructions, meaning that younger instructions may not observe the output of certain older instructions. Directly analogous to multiprocessor consistency and memory barriers, programmers must then introduce explicit synchronization, within a single thread, to ensure correct ordering of producers and consumers. Enforcing the programmer's desired ordering often requires these single-thread synchronizing instructions to serialize. Examples include `membar #sync` in SPARC, `cpuid` in X86, and `isync` in PowerPC.

Other Complex Instructions Several other instructions with complex semantics, such as atomic read-modify-write instructions, or instructions that synchronize multiple threads, are also potentially SIs [8]. We assume that they are implemented in a high-performance manner, i.e., are not serializing, and do not consider them further.

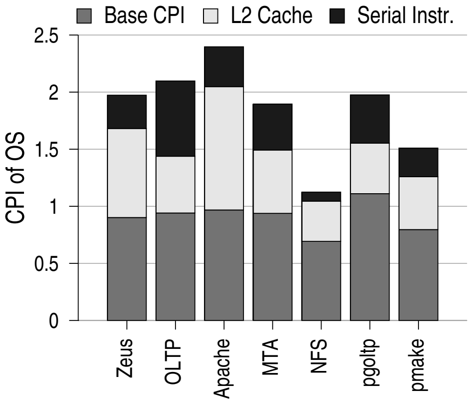


Figure 1. OS CPI (Ideal SPARC)

2.2. Why are SIs Important?

We motivate a more detailed analysis of SIs by briefly examining their impact on performance, especially for OS code. Figure 1 shows the contribution in *cycles per instruction* (CPI) for OS code from three sources. The lowest bar represents the base CPI of a 15-stage, 4-issue OoO processor with a 128-entry instruction window. The base CPI includes the contribution from branch prediction and the L1 caches, but uses a perfect L2 cache and ideal execution of SIs. This configuration uses the SPARC V9 ISA, but we have removed the effects of register window and software TLB traps to make the results relevant to other architectures (details are provided in Section 4).

On top of the base CPI, Figure 1 shows the additional CPI from a realistic L2 cache, and from realistic execution of SIs. We see that the CPI contribution from SIs rivals the performance impact of misses to main memory for many benchmarks. Including user code, we observe that SIs cause a 3–17% overall performance loss for this configuration.

High cache miss rates for system-intensive commercial workloads are often blamed for their high CPI [7], but Figure 1 shows that SIs have a significant performance impact as well — certainly enough to justify a detailed analysis of SIs and an exploration of mechanisms to tolerate them.

2.3. How are SIs Implemented?

The broad scope of writes to non-renamed state forces ISA and processor designers to make one of three choices when implementing these instructions: 1) Provide sequential semantics for these SIs, and implement a complex set of mechanisms to execute them OoO, 2) provide sequential semantics, but implement a simple mechanism and serialize the pipeline to execute them, or 3) execute them OoO with a simple mechanism, but require explicit programmer synchronization to ensure correct ordering.

Writes to most control registers are guaranteed to have sequential semantics in SPARC, X86, and PowerPC. Because of cost and complexity, however, the first choice for implementing them is impractical for most SIs. Real processors instead appear to choose the second option (see Section 3.5). An implementation might serialize by flushing younger instructions from the pipeline, and waiting to execute an SI until all older instructions retire. A processor could also block different consumer instructions at various pipeline stages to ensure their dependencies are honored, avoiding the flush, but still serializing (see section 5.2).

Due to the high performance cost of serialization, ISA designers will often choose option three for certain instructions, placing the burden of correct ordering on the system programmer. For example, none of the following are guaranteed to have sequential semantics: loads and stores to many Address Space Identifiers (ASIs) in SPARC V9, reads and writes to `cr8` in X86, and reads and writes to the segment lookaside buffer (SLB) in PowerPC.

The best choice between options two and three remains an open question. We argue, however, that requiring sequential semantics releases the programmer from the burden and performance cost of explicit synchronization, while allowing the processor to optimize SI execution through novel microarchitectural innovation (see Section 5.4).

2.4. Where do SIs Arise?

SIs arise predominantly when software is exercising low-level control over the processor — typically when executing privileged instructions in the Operating System (OS) or hypervisor (though some SIs are occasionally executed by user code). Their impact will thus go unnoticed by researchers and industry designers focusing on traditional benchmarks such as SPEC CPU, or even short traces of commercial workloads.

In this paper, we focus on system-intensive workloads — those that spend a considerable fraction of their time executing OS or hypervisor code. We primarily study commercial workloads in this paper, which spend 15–99% of cycles in the OS, though we expect our results will translate to any system-intensive application, including many desktop applications and virtual machine environments.

2.5. Trends Causing Increased Impact

Several trends are conspiring to make serializing instructions more frequent and more costly, providing additional motivation for a rigorous study of SIs.

Large-Window Processors Several processors that can maintain thousands of instructions in-flight have been proposed by academic and industry researchers. These designs, whether using multiple cores to effectively build one large window [14, 29, 33], clusters of partitioned functional units [26, 27], or relatively simple extensions to current processors [30], share two common themes. First, a larger instruction window extracts more ILP. This reduces the time between serializing events, and increases the fraction of time spent serializing (Amdahl's Law). Second, both the size of the window and the latency to communicate to all components increase the time required to drain and refill the window. While we do not expect to see processors with large, monolithic instruction windows, we do use such a configuration as a proxy for these other designs in Section 5.

Redundant Multithreading The effects of most SIs cannot be "undone," i.e., SIs are non-idempotent. When using redundant multithreading for reliability (e.g., [28]), all cores must thus verify the correctness of older instructions before executing an SI. In addition, no core can start executing younger instructions until the SI is committed, to ensure that dependencies are honored. Smolens, et al., report that the verification latency between cores has a dramatic impact on performance, largely due to SIs [28].

Trap and Emulate VMMs Software virtual machines such as VMWare, which perform trap-and-emulate and/or binary rewriting, can increase the frequency of serializing instructions, since they can turn one SI (for example, a write to privileged state) into several SIs (for example, trapping to emulation, performing the requested operations, and returning from emulation) [16].

2.5.1 Still Need Single Thread Performance

Effective thread-level parallelism can improve throughput without requiring high ILP. However, as we usher in the era of multi- to many-core chips, we must remember that *single thread performance still matters*. Sequential programs, and serial sections in parallel programs, must be executed quickly to provide good overall performance [12].

3. Characterizing of Serializing Instructions

In this section we examine the nature and frequency of serializing instruction across several system-intensive commercial workloads and three platforms. Before we present this data, however, we describe the three platforms and our simulation methodology.

3.1. Methodology

For this study, we use Simics [20], an execution driven, full-system simulator that functionally models various machines in sufficient detail to boot unmodified OSs and run unmodified commercial workloads. We use Simics to functionally model three CPUs that implement three ISAs: UltraSPARC IIICu, which implements the SPARC V9 ISA; AMD Hammer, which implements X86-64; and PowerPC 750 (G4), which implements the 32-bit PowerPC ISA. For the characterization study presented in this section, we use an idealized, one IPC processor model. Workloads are run for several simulated seconds, and sometimes minutes, to warm up the application and OS disk cache. Experiments are run for one billion instructions.

Apache	We use the Surge client [4] to drive the open-source Apache web server, version 2.0.48. We do not use any think time in the Surge client to reduce OS idle time. Due to a bug in our version of the PowerPC Linux loader, we we unable to run Surge client on this machine. The X86 Surge client is compiled for 32-bit mode.
MTA	MTA runs a Mail Transport Agent similar to *sendmail*. SPARC and X86 workloads use the *postfix* MTA, the PowerPC workload uses *Exim*. All MTAs are driven by the *postal-0.62* benchmark to randomly deliver mail among 1000 users.
NFS	NFS runs the *iozone3* benchmark to perform random read/writes from NFS mounted files. The tests do not include mounting and unmounting the filesystems.
OLTP	OLTP uses the IBM DB2 database to run queries from TPC-C. The database is scaled down to about 800MB and runs 192 user threads with no think time. DB2 is not supported on any of the X86 or PowerPC platforms that Simics models.
pgoltp	pgoltp also runs queries from TPC-C, but uses the PostgreSQL 8.1.3 database [24] driven by OSDL's DBT-2 [23]. Unlike IBM's DB2, PostgreSQL performs I/O through the OS's standard interfaces and utilizes the OS's disk cache.
pmake	Parallel compile of PostgreSQL using GNU make with the `-j 8` flag. Compilation is performed without optimizations. The SPARC workload uses the Sun Forte Developer 7 C compiler, X86-86 and PowerPC use GCC 4.1.0 and 2.95.3, respectively.
Zeus	We use the Surge client again to drive the commercial Zeus web server, configured similarly to Apache.

Table 1. Workloads used for this study

The SPARC machine runs Solaris 9, the X86-64 runs Linux 2.6.15, and the PowerPC runs Linux 2.4.17. We examine several system-intensive workloads across these three platforms, which are described in more detail in Table 1. Significant effort was undertaken to keep the workload configurations as similar as possible across platforms. Nonetheless, differences in the OS, the structure of platform-specific code, and compiler optimizations for a particular target make direct comparisons difficult.

SPARC Register Window and TLB Traps To isolate the effects of SPARC's register windows and software managed TLB, our characterization includes data for both normal SPARC execution, and also for an idealized configuration that ignores register window traps and uses a hardware filled TLB. The hardware TLB causes us not to see otherwise frequent TLB fill handlers, while still observing page fault behavior.

3.2. Description of SIs

Below we discuss the specific instructions in each of the three platforms that we consider to be SIs, and the registers we consider to be non-renamed due to broad scope. But we wish to reiterate that the registers and SIs described below *could* be renamed or executed OoO by a particular implementation, and some possibly are. However, we aim to study SIs in general, not any one particular implementation. Other instructions and registers that we do not observe in our warmed-up workloads might also be non-renamed and serializing, such as debug registers and performance counters.

SPARC V9 Non-renamed SPARC registers are listed in Table 2, and include most privileged registers. We assume that all general purpose registers, including all windowed registers, alternate globals, and floating point registers *are* renamed. We also assume that the integer and floating-point condition codes, and all register window management registers except `%wstate` are renamed. Excepting instructions and exception return (`done` and `retry`) are SIs because they implicitly write several non-renamed registers.

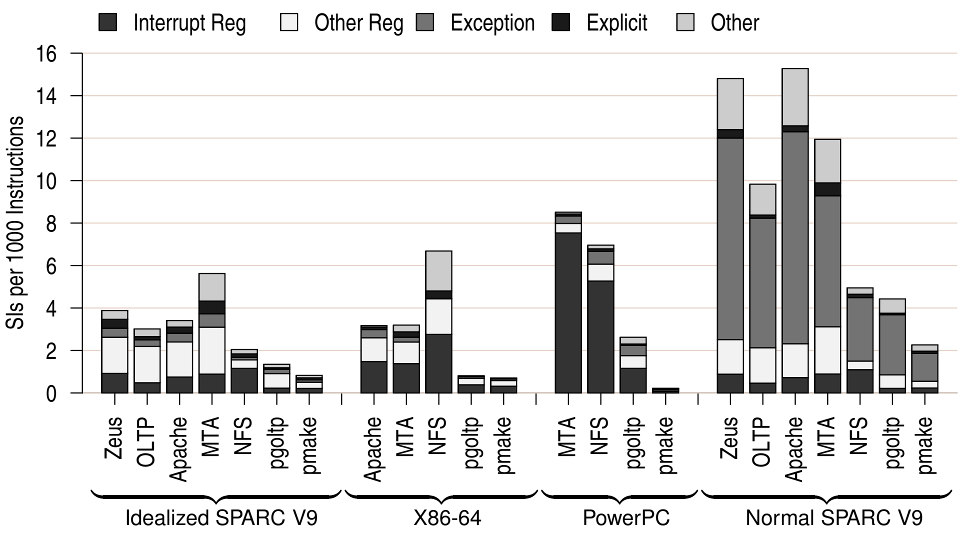
SPARC	pstate, fprs, pcr, pic, dcr, gsr, pil, wstate, tba, tpc, tnpc, tstate, tt, tl, tick, stick, softint_set, softint_clr, softint, tick_cmpr, stick_cmpr, fsr, ASI-mapped
X86-64	cr0, cr2, cr3, cs, eflags, msr, msw, tr, ldtr, idtr, gdtr
PPC	msr, ssr0, ssr1, ctrl, dar, dsisr, sdr1, accr, dabr, iabr

Table 2. Registers Considered Non-renamed

The SPARC ISA uses *Address Space Identifiers* (ASIs) to perform a variety of operations. Though not required by the ISA to have sequential semantics, we also examine writes to ASI-mapped registers, which include TLB registers and hardware functions such as interrupting another CPU. We do not consider to be serializing atomic read-modify-write instructions (e.g. `casa`), writes to block ASIs (e.g. ASI_BLK_PRIMARY), write to "AS_USER" ASIs, or memory barriers other than `membar #sync`.

X86-64 For our X86 target, the non-renamed registers are also listed in Table 2. Some MMX control registers would also likely be non-renamed, but we do not observe accesses to them in our workloads. Several SIs implicitly write `CS`, the code segment register, including `sysenter`, `sysret`, `iret`, and the 'far' versions of `call`, `ret`, and `jmp`. We assume segment registers contain not only the descriptor, but also the offset and flags loaded from the descriptor table. We assume other segment registers are renamed.

We consider instructions `invd`, invalidate caches, and `invlpg`, invalidate TLB, to be SIs . Other instructions are defined by the ISA to be serializing, such as `cpuid` and `rsm`, and various instructions that load the descriptor table registers. Like SPARC, exceptions and return instructions (e.g., `iret`) are SIs. We do not consider the various `fence` instructions to be SIs.



(a) Fraction of dynamic instructions that are SIs

Bench	OS Inst	OS SIs
Ideal SPARC		
Zeus	75.1%	94.0%
OLTP	15.1%	64.9%
Apache	60.7%	89.0%
MTA	50.2%	93.7%
NFS	99.8%	99.9%
pgoltp	13.5%	88.5%
pmake	12.2%	93.3%
X86-64		
Apache	47.3%	91.9%
MTA	38.3%	96.5%
NFS	99.6%	100%
pgoltp	6.9%	94.4%
pmake	8.4%	94.8%
PowerPC		
MTA	33.8%	98.2%
NFS	66.6%	96.8%
pgoltp	17.9%	92.1%
pmake	3.8%	91.4%

(b) Instr. and SIs from OS

Figure 2. Fraction of dynamic instructions that are serializing

PowerPC Non-renamed PowerPC registers are listed in Table 2, however, the only non-renamed registers for which we actually see accesses are `msr`, `ssr0`, and `ssr1`. Similar to the other target architectures, `isync` is serializing, but the other memory barrier variants are not. Instructions that read and write the segment registers `sr0-15` are not required to have sequential semantics, though we examine them as well. Dynamic exceptions and exception returns (`rfi`) are SIs.

3.3. SI Frequency

Figure 2(a) shows the number of dynamic instructions that are serializing for each of the three platforms, including idealized and normal SPARC. Bars are broken down into writes to non-renamed registers, exceptions, explicit synchronization, and other instructions (mostly TLB manipulations in SPARC and PowerPC, and cache invalidations in X86).

We observe that writes to non-renamed registers (the bottom two bars) comprise a significant fraction of SIs for all platforms except normal SPARC. Nearly two, and often more, non-renamed register writes occur every thousand instructions for all platforms and workloads except *pmake* and *pgoltp* (the two that spend the smallest fraction of time in the OS).

We separate writes to enable/disable interrupts from other register writes, since these are the most common non-renamed register writes for all three platforms. Frequent updates to the *Interrupt Privilege Level* register have also been observed on the VAX architecture [16]. It is debatable whether or not such writes would need to be serialized, but X86 is the only platform for which these particular writes can be identified at decode time, since it uses a special opcode. In SPARC and PowerPC, the interrupt enable field is part of the non-renamed `%pstate` or `msr` register, respectively. The following SPARC code sequence, from Solaris 9, illustrates the problem:

```
1: rdpr   %pstate, %o5
2: andn   %o5, 2, %o4
3: wrpr   %o4, 0, %pstate
```

The first instruction reads the `%pstate` register, the second clears a bit, and the third writes the entire contents of `%o4` back to `%pstate`. Without looking at the whole sequence, the decode logic cannot distinguish this write (which simply turns off interrupts) from other writes to `%pstate` which *do* need to serialize. PowerPC register writes are dominated by these interrupt writes, though they are a much smaller fraction of X86 and SPARC writes. The *Interrupt Reg* category also includes SPARC's `%pil`.

For normal SPARC execution, excepting instructions are very frequent. The other three platforms observe infrequent exceptions (primarily when making system calls or observing hard page faults). Both explicit synchronization and *Other* SIs occurs infrequently for all platforms except normal SPARC. But it is interesting to note that 1) most of the *Other* instructions do not require sequential semantics, and thus often need explicit synchronization to ensure correct semantics, and 2) the frequency of *Explicit* ordering SIs and these unordered *Other* SIs is similar for many workloads. These facts imply that there is minimal benefit from the choice of avoiding sequential semantics for some instructions.

The table in Figure 2(b) shows the fraction of dynamic instructions and SIs that are from the OS. Many of these applications spend a considerable fraction of their instruction in the OS. All workloads incur a large majority of their

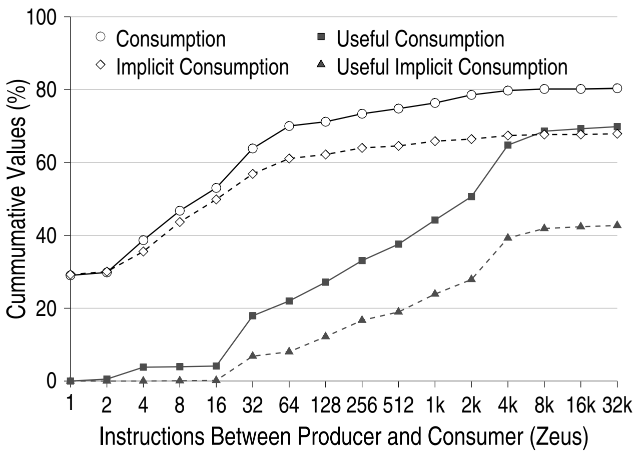


Figure 3. Use Distance of Non-renamed Regs

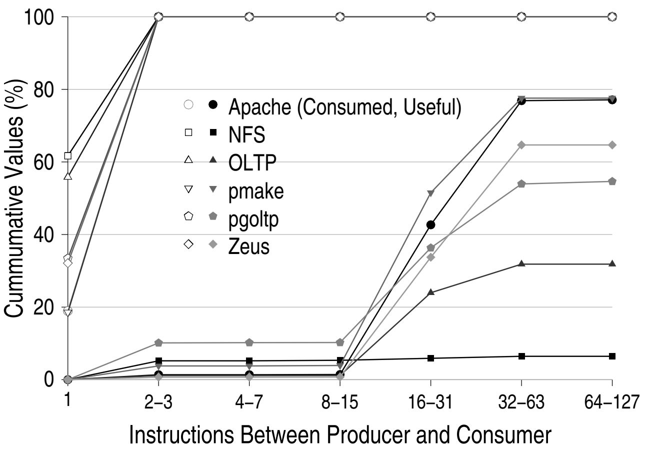


Figure 4. Use Distance of MMU Writes

SIs in the OS. OLTP, which frequently writes the SPARC `%fprs` register, is the only application with more than 12% of SIs coming from user code. "Normal" SPARC is not shown in the table, but despite the difference in SI frequency, all values are within 1% of the Ideal SPARC.

3.4. Examining Useful Consumption

The most frequent SIs for idealized SPARC, as well as X86 and PowerPC, are writes to non-renamed registers. Further analysis, however, reveals that many of these writes are *effectively useless* (EU). EU writes are those that do not observe consumers whose execution is affected by the write. EU writes occur, among other reasons, when only one field of a control register is updated by the write. For example, the *mask* field in the SPARC `%gsr` register is only used by one instruction, `bmask`. Like the interupt enable field discussed previously, updates to *mask* cannot be distinguished at decode from writes to other fields. All SPARC VIS instructions are thus consumers of the `%gsr` register even though that consumption is often useless.

We also examine the difference between *explicit* consumers, those that name the register they consume as an operand, and *implicit* consumers, those that do not. Implicit consumers often read their registers at various pipeline stages (i.e., they create the broad scope of control registers), and are the primary reason these registers are not renamed.

Figure 3 shows the cumulative distribution of the number of committed instructions between the producer (e.g. `wrpr`) and the consumer (e.g. `rdpr`) of values written to non-renamed registers. We only show data for Zeus on idealized SPARC, though all benchmarks on this platform are similar. The solid lines include both implicit and explicit consumers, and demonstrate the difference between a consumption (the top line), and a useful consumption of the value (the lower solid line). The dichotomy is striking: 50% of writes are consumed within 16 instructions, but only 5% of those writes are useful to the first 16 instructions.

The dashed lines only consider implicit consumers. The top dashed line, *Implicit Consumption*, shows that 71% of values are read by implicit consumers. But more notably, the bottom dashed line, *Implicit Useful Consumption*, shows that only 43% of writes are ever useful to implicit consumers, and less than 25% of writes are useful within 1024 instrutions. Processors serialize these writes to ensure implicit consumers observe the new value, but this serialization is rarely necessary for a particular dynamic instance of the write. We take advantage of this observation in Section 5.4 to improve the performance of these writes.

Dynamically-dead writes, meaning that no intervening consumers appear before another write occurs to the same register [5], are a subset of EU writes. *Silent* writes, meaning that it wrote the same value currently in the register [18], are also a subset. Dynamically-dead writes occur 17.5% of the time, corresponding to the rightmost point of the top line. Fourteen percent of writes are *silent*, and 4% are both. Looking at the *Useful Consumption* line in Figure 3, we see that virtually all non-silent, non-dead writes (73%) are eventually useful. However, the novelty of the EU characterization lies in the fact that it takes several thousand instructions for most values to *become* useful.

Writes to TLB State A second class of frequent SIs in both SPARC and PowerPC are manipulations of TLB state. TLB demap operations are particularly common for PowerPC. Writes to the idealized SPARC TLB include demaps, insertions after a page fault, and changes to the context register. We observe that nearly all memory operations are implicitly dependent on changes to the TLB state, but again expect that not all writes to TLB state will actually be useful to future memory operations. For example, unless the program attempts to access a virtual address previously translated by the demapped entry, the TLB values changed by the demap will not actually be used.

Similar to Figure 3, the top lines in Figure 4 show, for ideal SPARC, the number of instructions between a TLB write and an implicit consumption of that write. We are only able to observe TLB consumers within 127 instructions of the write, so we limit the range of the graph. We almost never observe explicit consumption within the window, so all lines represent implicit consumers only. Figure 3 shows that all TLB writes are consumed almost immediately. They become useful, however, only after the new value causes a different translation for younger instructions. The lower set of lines show the distance before the first use happens. Compared to non-renamed register writes, TLB writes are generally useful much more quickly. Nonetheless, there is a large discrepancy between the pessimistic expectation that all consumers need the value of the write versus only those that actually find it useful. Again in Section 5.4, we use this observation to improve the performance of this class of SIs as well. The NFS workload has many more demaps than the rest, and thus observes fewer useful consumptions.

3.5. SIs in Real Implementations

When studying SIs using simulation we are forced to make assumptions about which instructions are likely to be SIs in a realistic implementation. Though we have carefully chosen which instructions to consider, we also examine three processor manuals for further insights.

UltraSPARC III Cu Using a table of instruction latencies and conditions that block dispatch, it appears that the UltraSPARC III Cu serializes atomic memory instructions (e.g. `casa`), reads and writes to many privileged registers, `done` and `retry`, and certain memory barriers [31]. The V9 architecture does not require sequential semantics for stores to most ASI-mapped registers and their dependent instructions, instead requiring software synchronization.

Pentium M The X86-64 ISA implemented by Simics' AMD Hammer functional model is virtually identical to that implemented by Intel processors. All of the SIs described in Section 3.2 are defined by the ISA to be serializing. The segment registers are not defined to be serializing (except the `CS` register), and the Pentium M family does provide special-purpose hardware to handle OoO execution with pending reads and writes to these registers. However, it has only two copies of these registers, thus multiple writes in the window will force subsequent instructions to stall [13].

Alpha 21264 The Alpha 21264 (EV6) uses Privileged Architecture Library (PAL) code to explicitly access privileged registers [10]. In the case of a write to a privileged register, the EV6 uses a scoreboard mechanism to only serialize instructions that may be dependent on that register. We investigate a similar mechanism in Section 5.2. Interestingly, earlier Alpha processors did not require sequential semantics for reads and writes to any of these registers, forcing the OS developer to avoid dependencies by scheduling instructions with full knowledge of pipeline latencies or using explicit serialization.

Width	4 instructions / cycle
Integer pipeline	15 stages
Instr. window	128 entries (Large Window: 1k)
Branch pred.	12kB YAGS
Ld/St queues	32 entries each (Large Window: 256)
Ld/St pred.	1k entry "safe distance"
L1 instr. cache	32kB, 4-way, 2-banks, 2-cycle latency
L1 data cache	32kB, 4-way, 2-banks, 2-cycle, write-back
L2 unified cache	1MB, 8-way, 14 cycle load-to-use, inclusive
Main Memory	265 cycles load-to-use

Table 3. Baseline processor parameters

Summary Though the implementation details for these processors are not available, the processor manuals provide a good indication of the performance implications of particular SIs, and we believe these implementations align well with our assumptions. In particular, the UltraSPARC appears to implement SIs much like our baseline processor, while the EV6 appears to use a similar mechanism to one that we examine in Section 5.2. It is unclear how SIs are implemented in the Pentium M, but many instructions are said to be serializing. The PowerPC 750 manual provides no new insights beyond those in Section 3.2.

4. Performance Evaluation Methodology

For the performance study, we have developed a detailed, out-of-order processor and memory model, for the SPARC platform only, using Simics Micro-Architectural Interface (MAI). This model consists of a functional simulator (Simics) and a timing simulator (our OoO processor and memory). Simics MAI imposes its own serializing instructions, and other limitations on the timing of certain instructions, which prevents us from modeling a microarchitecture more aggressive than Simics. To alleviate these problems, we run Simics MAI as a dynamic trace generator. Our timing simulator steps the MAI functional model through *all* stages of an instruction's execution when the timing model first attempts to fetch an instruction. Unlike static traces, these dynamic traces adapt to changes in timing (e.g., due to OS scheduling decisions) that arises due to microarchitectural effects. Since the timing model requests particular instructions from the functional model, as opposed to the functional model feeding instructions to the timing model, the simulator faithfully models wrong-path events including speculative exceptions.

Methodology Due to inherent variability in the commercial workloads, we add small, random variations in main memory latency and run multiple trials per benchmark [2]. We show the 95% confidence interval using error bars in addition to the sample mean. All commercial workloads are warmed up and running in a steady state. We run these detailed timing simulations for one billion instructions. While IPC is a poor metric for multiprocessor simulations with these workloads, it is adequate for a uniprocessor since we observe no spinning or idle time.

Microarchitecture We conduct our experiments using two processor configurations. Our baseline processor is intended to loosely represent a modern, standard high-performance core. The second is an optimistic large window processor intended solely to illustrate the impact of some of the future trends discussed in Section 2.5. The parameters for both processors are shown in Table 3.

Both processors serialize all of the SPARC V9 SIs discussed in Section 3.2, except for writes to ASI-mapped registers for which the ISA does not require sequential semantics. They pessimistically treat all younger instructions as consumers of an SI. Thus, when a SI is detected, all younger instructions in the window are squashed and fetch is stalled. Typically SIs are detected at decode, though it can be later in certain cases such as exceptions. The SI is executed after all older instructions retire.

Register window management is handled entirely within the rename logic; saves and restores do not introduce any synchronization. Block (64-byte) loads and stores are "cracked" at decode so that they can be handled within the existing load-store-queue mechanisms. We assume special hardware to detect virtual address aliases that occur when using AS_USER ASIs (ASIs which the OS uses to copy in or out of user data structures). While not serializing, reads to certain non-renamed registers, particularly those written by hardware such as interrupt status registers, execute non-speculatively.

5. Microarchitectural Improvements for SIs

In this section we evaluate the performance impact of SIs on our baseline microarchitecture using the SPARC V9 platform, and then explore three mechanisms to reduce the number and cost of serializing instructions. The first, *scoreboarding* reduces serializing instructions by handling dependencies for non-renamed registers. The second, *late-squash* is a simple prefetching technique to reduce the cost of serializing instructions. The third, *effectively useless prediction* optimizes writes that are not actually useful for consumers. It is quite possible that these first two techniques have been implemented in real processors, though the third, to our knowledge, is a novel technique taking advantage of new observations.

5.1. Performance Impact

We examine the difference between a baseline implementation, which serializes all SIs except ASI-mapped register writes, and a hypothetical implementation that does not serialize any instructions. This hypothetical implementation is unrealistic because it assumes hardware exists to handle all explicit dependencies, and it ignores any timing constrains imposed by implicit dependencies.

For the modest, medium-window processor, we break down the non-serial speedup into that observed by user code and by OS code. For all workloads, user code observes only minor improvement, but the OS observes a 8–45% increase in performance. The labels below the bars indicate the overall non-serial speedup including user and OS.

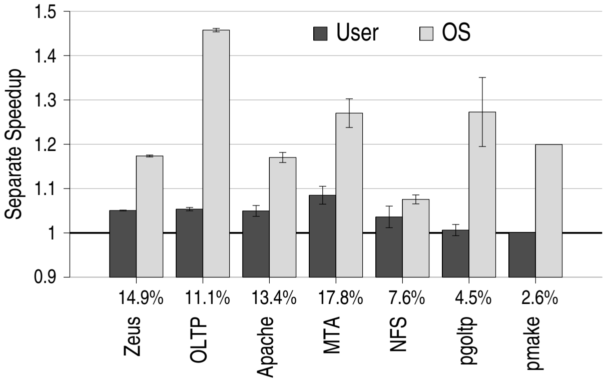


Figure 5. OS vs. User Non-serial Speedup

5.2. Scoreboarding Non-renamed Regs

Inspired by a reference to *scoreboarding* PAL code registers in the Alpha EV6 processor manual [10], we investigate a mechanism to reduce the frequency and cost of serializing writes to non-renamed state. Instead of completely serializing, or somehow providing multiple copies of broadly scoped registers, we apply a concept similar to scoreboarding from the CDC 6600 [32] to selectively block instructions that have a dependence with an outstanding write. Scoreboarding takes advantage of the fact that many writes will not observe consumers while they are in the window (∼40% of the time for Zeus, from Figure 3).

The scoreboard performs two primary functions for improving performance: 1) for each decoded instruction, the scoreboard determines whether the instruction is independent of all older SI writes, and can thus proceed, possibly OoO with respect to those writes, and 2) for each instruction that *is* dependent on an older write, the scoreboard determines which pipeline stage the consumer reads the register, allowing the instruction to proceed until that stage. In our implementation, the scoreboard is a table that tracks up to one outstanding write to each register. To ensure writers are non-speculative, we force them to be the head of the instruction window when they execute, which handles WAR hazards without any need to track consumers. WAW hazards cause the second write to block at decode.

5.3. Late Squash

Blocking the front-end while waiting for an SI can ensure all implicit dependencies are met. But we observe that instructions can speculatively enter the window behind a SI if they are later squashed and rerun through the pipeline to ensure they observe any updates from the SI. We refer to this scheme as *late-squash*. Late-squash allows instructions as

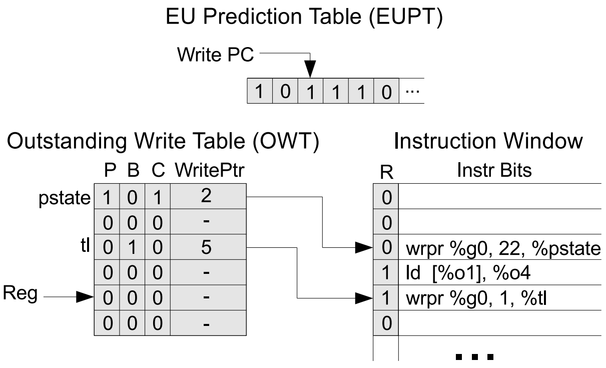


Figure 6. EU Prediction Mechanisms

well as loads and stores to prefetch their cache lines. Since few instructions actually require the value from the SI for correct execution, an accurate prefetch trace is generated.

5.4. EU Prediction for Non-renamed Regs

In Section 3.4, we observed that most control register writes are consumed within a few instructions, but those values are *effectively useless* to the first several hundred instructions after the write. We propose effectively useless prediction to allow a majority of EU writes and their (useless) consumers to execute OoO. Using simple hardware extensions, we can then guarantee that non-faulting consumers executed correctly even after receiving a stale value.

5.4.1 Prediction Mechanisms

Our implementation of EU prediction for control registers requires two simple structures. First, the EU Prediction Table (EUPT) consists of 512, non-tagged, 1-bit entries indexed by the SI's PC. Each entry indicates whether this SI is likely to be EU during the time it is in-flight. This table is shown in Figure 6.

Second, we utilize the Outstanding Write Table (OWT), which contains several entries for each control register, corresponding to the number of allowable outstanding writes to that register (we observe that three entries per register is sufficient). Entries consist of a pointer to the instruction window entry for the write (WritePtr); a *Pred* bit (P), indicating whether this write was predicted to be EU; a *Consumed* bit (C), indicating whether any implicit consumer has potentially accessed the register; and a *Blocked Front-end* bit (B), indicating whether this write has caused the front end to block. An OWT allowing only one write per register is depicted in Figure 6. The figure shows writes to two registers: the first is predicted EU, and the second is not.

We also add one bit to each instruction window entry. For all instructions, the R bit indicates that the instruction has potentially *Read* any of the control registers with older outstanding writes. Note that an SI can also be a consumer. Below, we enumerate the steps for performing EU prediction at decode and commit for both SIs and consumers.

SI Decode At decode, for each SI write to a control register, we use the EUPT to predict whether the write will be effectively useless during the time it is in-flight. We update the *Pred* bit with this prediction, update the *WritePtr*, and clear the *B* and *C* bits.

Consumer Decode When an implicit consumer is decoded, it checks each input control register for any outstanding writes in the OWT that are *not* predicted to be EU. If one is found, the consumer is squashed, the *Blocked Front-end* bit is set, and fetch is stalled until the write commits. This was the case for the second write in Figure 6 (the consumer is no longer in the window). If outstanding writes exist, but they are all predicted EU, then the *Consumed* bit is set for each write, the *Read* bit is set for the consumer, and the instruction can proceed. This case happened for both the load and the second `wrpr` with respect to the `%pstate` write.

SI Commit When a write commits, it locates its OWT entry and checks its *Consumed* bit. If not set, it updates the EUPT to indicate that it was EU. If the write's *Consumed* bit is set (and the *Pred* bit is also set), then consumers potentially accessed a stale value. We then compare the new value to the previous contents of the register, and determine whether younger instructions that potentially read the stale value were correctly executed (see below).

If a predicted EU write is found to have made useful changes to the register *and* has observed potential consumers (C bit), all younger instructions are squashed, and the write commits before re-fetching the consumers, who will now observe the updated value. In the case of multiple outstanding writes to the same register, any of them can trigger a squash.

If the *Blocked Front-end* bit is set, fetch is restarted. The OWT entry is cleared after commit. The entry must also be cleared if writes are squashed due to a branch misprediction, for example.

No additional actions are necessary at consumer commit.

Determining Correctness of Stale Values We can easily guarantee that implicit consumers receiving a stale value were correctly executed under two circumstances: 1) if the write was silent ($\sim$14% of the time for Zeus), or 2) if the write changes fields in ways that only affect excepting instructions ($\sim$50% of the time). For example, if the `FEF` (floating-point enable) field of the `%fprs` register is zero, then any floating-point instruction would generate an exception. If a write to this register sets the bit to one, younger instructions can observe the stale value of zero. However, these consumers will either not be affected by the register, or will cause an exception. The same thing is true for several other, though not all, frequently written registers. Thus, this simple mechanisms conservatively assumes a minority of EU writes are useful.

Exceptions When an instruction with its *Read* bit set incurs an exception, it may have been affected by an outstanding control register write. It is squashed, fetch is blocked until the window is empty, and the instruction is (re)fetched and executed with correct inputs.

Explicit Dependencies Once we have a mechanism to properly synchronize implicit consumers, handling explicit consumers is easy, since they simply need their input values delivered to the functional units the same as most instructions. Since control registers have very limited scope with respect to explicit consumers, they can actually now be renamed to satisfy explicit dependencies. Explicit consumers may thus receive their values OoO, but the architected register contents are updated (and become visible to implicit consumers) only at commit — just like normal registers.

5.5. EU Prediction for TLB Writes

As shown in Figure 4, several instructions can execute after a TLB write without affecting their translations. In addition, using a realistic processor model, we often observe events, such as returns from system code, that already prevent younger instructions from entering the window and executing early. Thus, in practice, more than 95% of TLB writes end up being useless for instructions in the window (though this also makes the benefits of optimizing TLB writes much less).

We again use the EUPT to predict useless TLB writes, allow predicted EU write to execute only when they become non-speculative, and allow younger instructions to execute OoO. To verify this prediction, instructions present in the window with a TLB write retranslate their virtual addresses at commit. If a translation is different, or if a TLB fault occurs, the consumer (which turned out to be a useful consumer) and all younger instructions are squashed.

Our baseline microarchitecture does not serialize writes to ASI-mapped registers, since SPARC does not require sequential semantics for them. However, as noted in Section 2.3, an implementation could provide sequential semantics, eliminating the need for explicit synchronization. Though not guaranteed to be the case, in our evaluation we assume that all `membar #sync` instructions are used to order ASI writes (as well as I/O, etc). We provide sequential semantics for all instructions by serializing, while optimizing TLB writes using EU prediction, and elide `membar #sync` instructions. (While not shown for brevity, serializing these ASI-mapped registers in the baseline instead of serializing explicit synchronization has very similar performance.)

5.6. Why not Value Prediction?

We have investigated value prediction [19] for SIs that write non-renamed registers, and have observed that *last value* prediction can be modified to accurately predict the values of many SIs. Value prediction can potentially avoid serializing all non-renamed register writes, not just those that are EU. Unfortunately, SI value prediction has one major problem: using a predicted value requires that prediction to be delivered to every stage of the pipeline where it could be used. Yet the complexities of doing so are exactly the reason SIs are serialized in the first place. EU prediction, in contrast, requires only one predicted bit to be delivered to one stage (decode).

5.7. Performance of SI Mitigation

The rightmost bar of Figures 7 and 8 shows the speedup of the hypothetical non-serializing configuration for the modest and large window processors, respectively. For the modest processor, avoiding serialization increases performance by 3–17%. For the large window processor, the improvement is 5–33%.

The second bar for each benchmark shows the speedup over the baseline when scoreboarding accesses to non-renamed (including ASI-mapped) registers. Scoreboarding results in a 0–5% performance improvement for the standard window, and 0–10% for the large window. This mechanism is relatively straightforward, and seems to attain a performance/complexity trade-off that makes it plausible for recent processors to have implemented it.

The third bar shows the performance of late-squash. Apache and Zeus, which incur many off-chip misses, each observe 5% and 13% speedups for the modest and large window processors, respectively. Other benchmarks see little benefit. To give a rough idea of extra power and front-end bandwidth required, we observe that benchmarks fetch 6–28% more instructions, and execute 3–14% more instruction on the modest proessor. Thus, for Apache, 28% more fetches and 14% more executions translate into 5% performance. While late-squash is relatively simple to implement, it is unlikely that the additional power is justified.

The performance of EU prediction for control registers and TLB writes is shown as the fourth bar in Figures 7 and 8. EU register and TLB prediction on the modest processor perform within 5% of the ideal non-serializing configuration for all workloads. For the large window processor, however, EU prediction only comes within 10% of the ideal non-serial configuration. Closing this remaining gap would require mechanisms to handle both non-EU writes, and implicit writes to non-renamed registers (such as exceptions and returns). Though not shown, EU prediction improves OS performance on the modest processor by 15–18% for Zeus, Apache, MTA, pgoltp and pmake, by 35% for OLTP, and 6% for NFS. EU prediction for TLB writes incurs an additional 0.2–1.0% of TLB lookups for instructions executed behind a TLB write.

Using a 512-entry non-tagged EUPT, we observe conflicts on 0.5–4% of updates to the table with the standard processor. The EUPT predicts that 60–95% of non-renamed

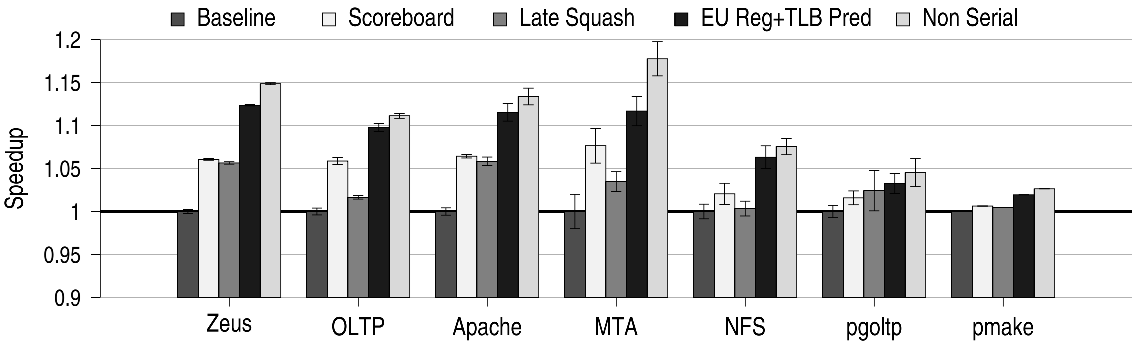


Figure 7. Baseline OoO Processor (Idealized SPARC)

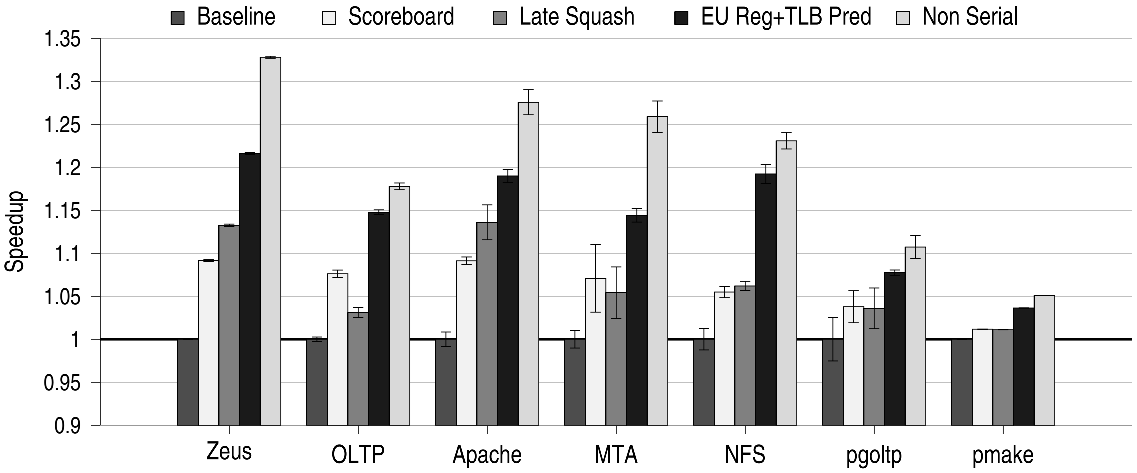


Figure 8. 1k-Entry Instruction Window Processor (Idealized SPARC)

register writes are EU based on the previous execution of that PC, and 99.1–99.8% of those prediction are correct. A 256-entry table observes conflicts on 15–18% of updates, an accuracy of 92–97%, and marginally worse performance.

6. Related Work

Chou, et al., briefly mention SIs in the context of atomic instructions and memory barriers for synchronizing multiple threads [8]. While such instructions are potential SIs, we focus on instructions within a single thread. Smolens, et al., also briefly mention SIs, and report that the verification latency between dual-redundant cores has a dramatic impact on performance, largely due to SIs [28].

Similar to late-squash, *runahead* [11, 21] is a prefetching technique to continue execution during a cache miss. Unlike the late-squash mechanism, runahead mode is not entered until the missing instruction reaches the head of the window, and thus, is unlikely to provide benefit for SIs.

Zilles, et al., [34], Keckler, et al., [17], and Jaleel, et al., [15] each propose mechanisms for handling exceptions without serializing. Such mechanisms are especially important for software TLBs, and are orthogonal to our proposals.

7. Conclusions

We present, to the best of our knowledge, the first analysis of *serializing instructions* (SIs) in system-intensive workloads. SIs, such as writes to control registers, have many complex dependencies, making OoO execution of these instructions difficult. SIs thus introduce a short sequential section into the instruction-level parallel execution of a single thread. As Amdahl's Law explains, frequent serialization limits performance despite a processor's ability to extract parallelism the rest of the time. We analyze the frequency of SIs across three ISAs, SPARC V9, X86-64, and PowerPC, and conclude that frequent SIs are a major contributor to the high CPI of operating system code, rivaling the performance impact of misses to main memory. Several additional factors are making the future outlook for SIs even worse, including proposals for large instruction window processors, speculative and redundant multithreading, and trap-and-emulate software virtual machines.

We examine the use of register values produced by SIs and discover that 90% of control register writes are *effectively useless* (EU) within the first 128 instructions — i.e., any consumers are unaffected by the new value of the write. We propose EU prediction, a novel technique that speculatively allows consumers to execute OoO, and read a stale control register value, but guarantees correct execution nonetheless. With this technique, we improve OS performance by 6–35%, and overall performance by 2–12%.

We further make two observations about ISA design. First, avoiding sequential semantics, but instead requiring programmer synchronization, provides little benefit. Explicit ordering can even hurt if execution of those SIs can be optimized in some way. Second, providing a way to identify, through the instruction opcode, which fields of a control register are written by a particular static instruction would eliminate a majority of EU writes.

8. Acknowledgments

This work is supported in part by National Science Foundation grants CCF-0702313 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin. We also wish to thank Matthew Allen, Koushik Chakraborty, and Jichaun Chang for their helpful comments and discussions.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of 9th HPCA*, 2003.

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proc. of 4th ASPLOS*, 1991.

[4] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *1998 Conf. on Meas. & Model. of Comp. Sys.*, 1998.

[5] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proc. of 10th ASPLOS*, 2002.

[6] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proc. of 12th ASPLOS*, 2006.

[7] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proc. of 14th SOSP*, 1993.

[8] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of 31st ISCA*, 2004.

[9] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 translation buffer: simulation and measurement. *ACM Trans. Comput. Syst.*, 3(1):31–62, 1985.

[10] Compaq Computer Corp. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, 2000.

[11] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of 11th ICS*, 1997.

[12] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *To appear in IEEE Computer*, 2008.

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2007.

[14] E. Ípek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proc. of 34th ISCA*, 2007.

[15] A. Jaleel and B. Jacob. In-line interrupt handling and lockup free translation lookaside buffers (TLBs). *Trans. on Comp.*, 55(5):559–574, 2006.

[16] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[17] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee, and W. J. Dally. Concurrent event handling through multithreading. *Trans. on Comp.*, 48(9):903–916, 1999.

[18] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proc. of 27th ISCA*, 2000.

[19] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of 29th MICRO*, 1996.

[20] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Comp.*, 35(2):50–58, Feb 2002.

[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. of 9th HPCA*, 2003.

[22] D. Nellans, R. Balasubramonian, and E. Brunvand. A case for increased operating system support in chip multiprocessors. In *Proc. of 2nd IBM Watson P=ac^2*, 2005.

[23] Open Source Development Labs. Database test suite. http://osdldbt.sourceforge.net/.

[24] PostgreSQL. http://www.postgresql.org/.

[25] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proc. of 9th ASPLOS*, 2000.

[26] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *Proc. of 38th MICRO*, 2005.

[27] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proc. of 30th ISCA*, 2003.

[28] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of 39th MICRO*, 2006.

[29] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of 22nd ISCA*, 1995.

[30] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proc. of 11th ASPLOS*, 2004.

[31] Sun Microsystems, Inc. *UltraSPARC III Cu User's Manual*, 2003.

[32] J. E. Thorton. Parallel operation in the control data 6600. In *Proc. of Fall Joint Comp. Conf.*, 1964.

[33] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. of 14th PACT*, 2005.

[34] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proc. of 32th MICRO*, 1999.