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

To maintain a reasonable level of complexity, processor implementations contain Serializing Instructions (SIs) *— instructions, such as those that write control registers, that cannot be executed out-of-order (OoO). Maintaining sequential semantics may force SIs to serialize the pipeline and execute as the only instruction in the window.*

We examine the frequency of SIs in three ISAs, SPARC V9, X86-64, and PowerPC, for several system-intensive workloads. Across ISAs, we observe 2–8 SIs per thousand instructions for most workloads. As explained by Amdahl's Law, such frequent 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 observe a 4–17% performance difference between a modest out-of-order processor and a hypothetical processor which idealizes serializing instructions.

We examine the consumption of values produced by several SIs, and observe that most values are consumed, but that the values are Effectively Useless *(EU) — i.e. they do not actually change the execution of the consuming instructions. To improve the performance of such SIs, we propose EU prediction, which can allow younger instructions to proceed, possibly reading a stale value, and yet still correctly execute. This simple technique improves the performance of five of our seven workloads by 8–12%.*

1 Introduction

To maintain a reasonable level of complexity, processor implementations contain *Serializing Instructions (SIs)* — instructions, such as those that write control registers, that cannot be executed out-of-order (OoO). Maintaining sequential semantics may force SIs to serialize the pipeline and execute as the only instruction in the window. SIs predominately occur in Operating System (OS) or hypervisor code, and can occur frequently in system-intensive workloads — workloads which spend a significant fraction of their time executing OS or hypervisor code. For example, SPARC V9 instructions which write non-renamed control registers such as `%pstate` (processor state) are typically serialized in order to simplify the hardware implementation. But *Amdahl's Law* informs us that frequent SIs, which introduce a short sequential section into the instruction-level-parallel execution of a single thread, can greatly reduce the performance of an aggressive, OoO processor.

We observe a performance loss of 4–17% due to SIs for an moderately aggressive OoO processor, loosely representing current high-performance processors. But several trends are increasing the cost and frequency of SIs, including processors which can effectively maintain thousands of instructions in flight, processors which use speculative or redundant multithreading, and trap-and-emulate software virtual machines. Thus, we argue that SIs deserve a close examination in the context of system-intensive workloads.

Because SIs are serialized to simplify the microarchiture design, rather than because of true architectural limitations, specialized hardware *could* be implemented to allow OoO execution for each SI. In practice, however, implementing such hardware for all SIs is challenging and impractical. With that fact in mind, we seek to answer the following questions during our evaluation: 1) How frequent are SIs in current ISAs? 2) How much performance is lost to SIs, for reasonable processor implementations, and 3) Is it possible to provide one or more simple, *generic* mechanisms to recover most of this loss? In answering these questions, this paper makes the following contributions:

- We analyze the frequency of SIs from several workloads across three ISAs, SPARC V9, X86-64, and PowerPC. While significant differences among these platforms exist, we show that the major contributors to SIs are strikingly similar, and that SIs disproportionately affect OS code.
- We examine the *use* of register values produced by SIs, and discover that, while 75% of values are consumed within 128 instructions, 90% of values are *effectively useless* — i.e., they write a different value than the previous contents of the register, but the new value does not actually change the execution of the consuming instructions.
- We show that seven system-intensive workloads observe a 4–17% performance loss compared to a hypothetical non-serializing processor, and up to a 33% loss when considering a large-window processor. We also show that two simple techniques, *Late-Squash*, and *Scoreboarding* non-renamed registers, which are likely to have been implemented in real processors, can improve performance by 0–5%, but fall considerably short of the ideal, non-serial processor.
- Finally, we propose a novel technique which predicts when a value will be *effectively useless* during the time the producer is in-flight, and speculatively allows consumers to read a stale value. With this technique, we observe speedups of 8-12% for five of our workloads, and come within 5% of the ideal, non-serial processor for all workloads.

2 Serializing Instructions

Most instructions are defined by the ISA to have sequential (in-program-order) semantics: even if instructions are actually executed out of order (OoO), instructions older (in program order) than instruction A will not observe the effects of instruction A, and instructions younger than A will. *Serializing Instructions* (SIs) require, or arguably should require, sequential semantics, yet have many complex dependencies which may prevent OoO execution. As such, a processor implementation may be forced to *serialize* the pipeline in order to execute these instructions. We discuss the details of SIs throughout the rest of this section.

2.1 What are Serializing Instructions?

SIs fall into four broad categories: 1) instructions which write to non-renamed/control registers, 2) instruction with modify other low-level processor state, such as TLBs and segment registers, 3) instructions with cause exceptions, or other instructions which enter or return from privileged execution, and 4) explicit synchronization. We discuss each of these categories in turn.

Writes to Non-renamed/Control Registers The scope of a variable in a program refers to which locations in the program that variable has meaning and can be referenced. We use the term *scope of a register* to indicate which stages of the pipeline that register has meaning and is accessed. Most registers, including general purpose registers and condition code registers have very limited scope: they are read and written only at *execute* stage. Other 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. However, by not renaming these registers, writes to them cannot execute OoO, and must serialize. For example, the SPARC `wrpr` instruction is an example of an SI when it write 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.

Modifications to Other Processor State Modifications to other non-renamable processor state, such as TLBs, are also SIs, because this state also commonly has broad scope. Though possibly implemented as registers, this additional state is often accessed by special instructions.

Exceptions and Returns Dynamic instructions which trigger an exception, and instructions such as SPARC's `retry` or X86's `iret` which return from an exception handler, are also SIs because they implicitly write several non-renamed registers.

Explicit Synchronization ISAs may not require sequential semantics for all pairs or instructions, meaning that it is undefined whether younger instructions will actually observe the output of certain previous instructions. Directly analogous to multiprocessor consistency and memory barriers, programmers must then introduce explicit synchronization in order for software to ensure consistency among non-sequential instructions when necessary. These single-thread synchronizing instructions are also SIs. Examples include `membar #sync` in SPARC, `cpuid` in X86, and `isync` in PowerPC.

Other Instructions Several other instructions, such as atomic read-modify-write instructions, or instructions which synchronize multiple threads, are also potentially SIs [6]. However, in this paper, we assume they are implemented in a high-performance manner, i.e., are not serializing, and do not consider them.

2.2 How are Serializing Instructions Implemented?

Processor designers are forced to make a choice in order to ensure the sequential semantics of SIs: either utilize a simple mechanism to ensure correct execution, or design a set of complex mechanisms to execute them OoO while delivering correct values to wherever they are required. Because of the cost/complexity trade-offs, designers typically choose the former, and might implement SIs by flushing younger instructions from the pipeline, and waiting for all older instructions to retire, before allowing the SI to execute. With care, a processor implementation could instead block various consumer instructions at different pipeline stages depending on which registers are being written (see section 5.2).

As *Amdahl's Law* explains, frequent invocation of such mechanisms can drastically reduce ILP. ISA designers will thus often choose *not* to require sequential semantics for particular instructions, instead requiring explicit, programmer specified, synchronization. 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 the the segment lookaside buffer (SLB) in PowerPC.

Placing the burden of correctness on the programmer is not necessarily bad, as careful placement of synchronization can sometimes lead to less frequent serialization. However, a more important problem may be that explicit synchronization ties the hands of CPU designers more than does requiring sequential semantics. For example, if sequential semantics are required, an implementation could optimize the execution of SIs through novel microarchitectural innovation.

2.3 Where do Serializing Instructions 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 focusing on traditional benchmarks, such as SPEC CPU, or even by researchers focusing on short traces of commercial workloads.

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

2.4 Why are Serializing Instructions Important?

We motivate a more detailed analysis of SIs by briefly examining their impact on performance, especially their impact on the performance of OS code. Figure 1 shows the contribution in *Cycles per Instruction* (CPI) for OS code from various sources. The lowest bar represents the base CPI of a moderately aggressive, 15-stage, 4-issue OoO processor with a 128-entry instruction window. 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 as well. (The details of our configurations and methodology are presented in Section 4.)

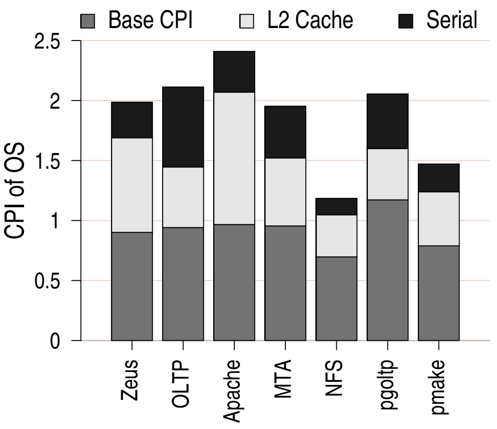


Figure 1: OS CPI (Ideal SPARC)

The base CPI shown in Figure 1 includes the CPI contribution from branch prediction and the L1 caches, but with perfect L2 cache and ideal execution of SIs. On top of that, we show additional CPI from a realistic 1MB 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.

High miss rates for system-intensive commercial workloads are often blamed for their high CPI [5], but Figure 1 shows that SIs can have a significant impact as well — certainly enough to justify taking a detailed look at SIs and exploring mechanisms to tolerate them.

2.4.1 Current Trends Cause Greater Impact

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

Large-Window Processors Several academic and industry proposals have appeared for creating processors that can *effectively* maintain thousands of instruction in-flight. These designs, whether using multiple cores to effectively build one large window [11, 23, 27], clusters of partitioned functional units [20, 21], or relatively simple extensions to current processors [24], share two common themes. First, creating a larger effective instruction window extracts more ILP, which increase 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 windows, we do use such a configuration as a proxy for these other designs in Section 5.

Redundant Multithreading When using redundant multithreading for reliability [18, 19, 22], all cores must verify older instructions before executing an SI since many SIs cannot be undone. 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 [22].

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) [13].

2.4.2 The Continued Need for Single Thread Performance

Effective thread-level parallelism can improve performance without requiring high ILP. However, as we usher in the era of multi- to many-core chips, Amdahl's Law forces us to remember that *single thread performance still matters*. Sequential programs, and serial sections in parallel programs must be executed quickly to provide good overall performance [9].

3 Characterization 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 present our simulation methodology.

3.1 Methodology

For this study, we use Simics [15], an execution driven, full-system simulator which 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, processors are run in functional mode. 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.

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

Apache	We use the Surge client [3] 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 from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time. Due to the proprietary nature of DB2, we are only able to run this workload on the SPARC platform.
pgoltp	pgoltp also runs queries from TPC-C, but uses the PostgreSQL 8.1.3 database [1] driven by OSDL's DBT-2 [17]. 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 versions 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

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 OS code, and compiler optimizations for a particular target, conspire to 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 which 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 Serializing Instructions

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-serializing 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 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	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 registers
X86-64	cr0, cr2, cr3, cs, eflags, msr, msw, tr, ldtr, idtr, gdtr
PowerPC	msr, ssr0, ssr1, ctrl, dar, dsisr, sdr1, accr, dabr, iabr

Table 2: Registers Considered Non-renamed due to Broad Scope

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 non-renamed registers

The SPARC ISA uses *Address Space Identifiers* (ASIs) to perform a variety of operations using loads and stores. 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.

Instructions `invd`, invalidate caches, and `invlpg`, invalidate TLB entry are also serializing. 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.

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.

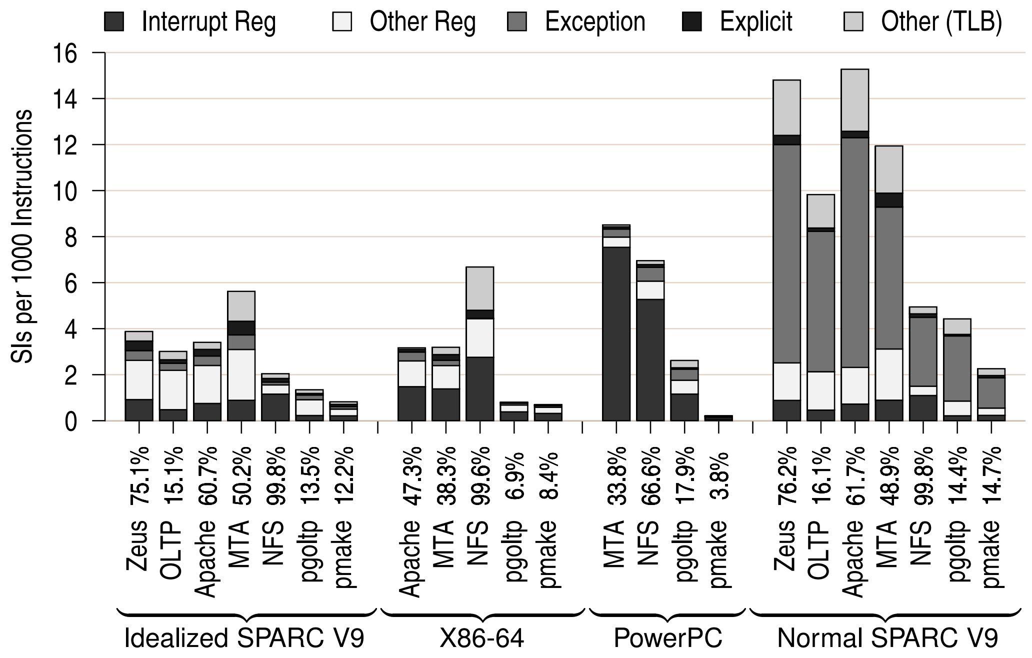


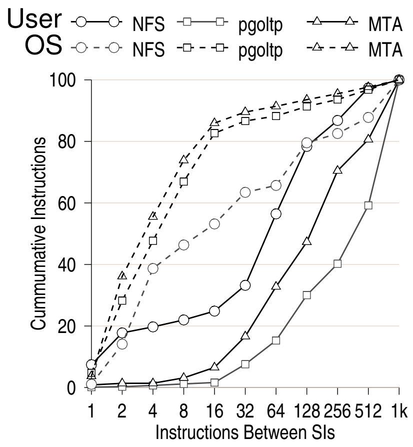
Figure 2: Fraction of dynamic instructions that are serializing

3.3 SI Frequency

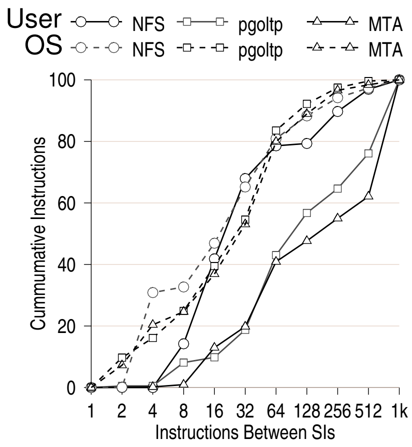
Figure 2 shows the number of dynamic instructions that are serializing for each of the three platforms, including the idealized SPARC and normal SPARC execution. Bars are broken down into writes to non-renamed registers, exceptions, explicit synchronization (e.g., PowerPC `isync`), and other instructions (mostly TLB manipulations in all three platforms). The labels below the bars represent the percent of instructions from the OS.

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 in the OS).

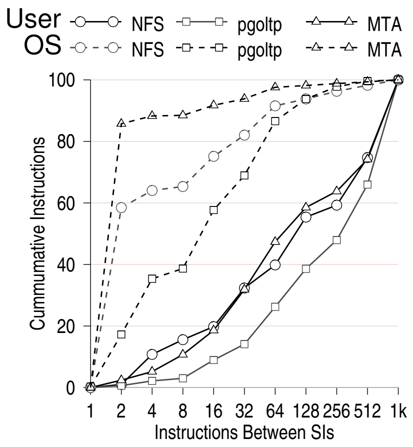
We separate writes to enable/disable interrupts from other register writes, since these are the most common non-renamed register writes for all three platforms. Frequent updates to the *Interrupt Privilege Level* register have also been observed on the VAX architecture [13]. While it is debatable whether or not such writes would need to be serialized, 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 `%pstate` or `msr` register, respectively, and writes to it cannot be distriguished from other serializing



(a) SPARC (ideal)



(b) X86-64



(c) PowerPC

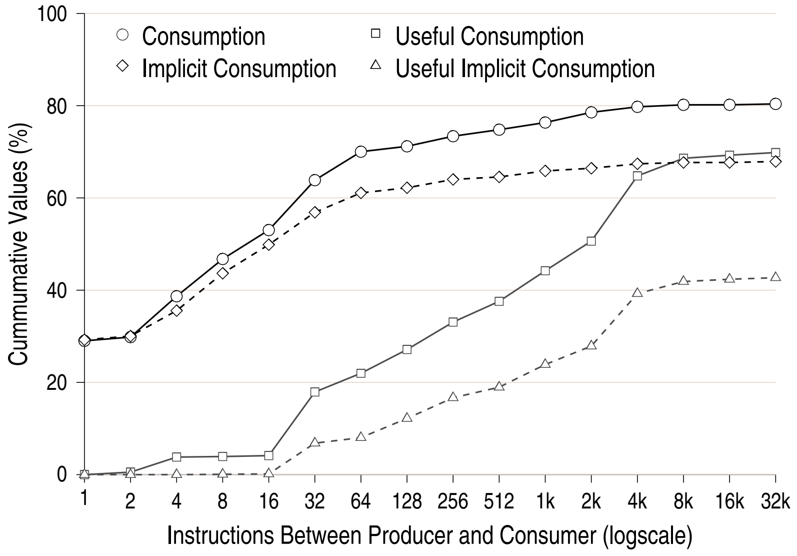
Figure 3: Cumulative Distribution of Instructions Between SIs

writes to this register until execute. PowerPC register writes are dominated by these interrupt writes, though they are a much smaller fraction of X86 and SPARC writes. *Interrupt reg* 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 (primarily TLB writes) occurs infrequently for all platforms except normal SPARC. However, it is interesting that 1) most of the *Other* instructions do not require sequential semantics, and 2) the frequency of *Explicit* ordering SIs and non-ordered *Other* SIs is similar for many workloads, especially on idealized SPARC and X86. These facts imply that the overhead of explicitly ordering *Other* SIs may be similar to the overhead of instead requiring sequential semantics and not using synchronization.

Finally, it is also interesting to note that the frequency of SIs for both X86 and PowerPC lie mostly within the frequencies for idealized and normal SPARC. While we are only able to provide a detailed performance evaluation for the SPARC platform in Section 5, this latter observation indicates that our results are likely to translate to these other platforms without major distortion.

Distribution If SIs appear close together, the length of the serial execution increases, but the cost of draining and refilling the pipeline is better amortized. Figure 3 shows a cumulative distribution of the number of instructions between SIs for three workloads for all three platforms. In most cases, SIs are spread out for user code, but more clustered for OS code. This is expected, since SIs are much more frequent in the OS. At one extreme, the OS for MTA on PowerPC observes that 85% of SIs occur within 2–3 instructions, primarily when returning from frequent exceptions. Most user code has considerably more instructions



(a) Use Distance of Non-renamed Register Writes for Apache

	Useful	# Bef. Use
Zeus	65%	39.4
OLTP	34%	62.8
Apache	79%	35.2
MTA	29%	43.4
NFS	6%	747.1
pgoltp	76%	34.3
pmake	52%	59.2

(b) Use Fraction and Distance of TLB Writes

Figure 4: Useful Consumption of Non-renamed Register and TLB Writes

between SIs: MTA on SPARC observes SIs within 63 or fewer instructions only 18% of the time.

In general, SI distribution in user code is similar across all three platforms. For OS code, two of the three workloads observe much more clustered SIs on both SPARC and PowerPC than on X86-64. There is considerable variation within both SPARC and PowerPC (less for X86), but overall, the distribution of SIs in both OS and user code is strikingly similar across all three platforms.

3.4 Examining Useful Consumption

The top contributors to SIs for idealized SPARC, as well as X86 and PowerPC, are writes to non-renamed registers. Further analysis, however reveals that not all data written to these registers impacts the execution of nearby, younger instructions. For example, many instructions are dependent on the SPARC `%gsr`, though several fields in this register (such as *mask*) are only used by one instruction type. But since writes overwrite the entire register, dependencies are created for all consumers. This example helps highlight the difference between the *consumption* of a value (all VIS instructions are implicit consumers of `%gsr`), and a *useful* consumption of the value — i.e. the new value actually changes the execution of consumer instructions compared to the previous contents of the register. We also examine the difference between *explicit* and *implicit* consumers. Explicit consumers name the register they consume, and use it only at execute stage. Implicit consumers do not explicitly identify the consumed register, and often use its value at various pipeline stages (i.e., the broad scope of control registers arises because of implicit consumers).

Figure 4(a) shows the cumulative distribution of the distance (in committed instructions) between the producer (e.g. `wrpr`) and the consumer (e.g. `rdpr`) of non-renamed register values for Apache on idealized SPARC. First, the figure demonstrates the difference between a consumption of the value (the top two lines), and a useful consumption of the value (the bottom two lines). The dichotomy is striking: 50% of register writes are consumed within 16 instructions, but only 5% of the writes change the execution of those first 16 instructions. For 17.5% of values, no intervening consumers appear before the register is written again, i.e., the producers are *dynamically-dead* [4] (this van be observed by looking at the rightmost point of the top line). For 14% of writes, no useful consumption occurs because the producers wrote the same value as existed previously, i.e., the producers are *silent* [14] — 4% are both silent and dynamically dead.

The top dashed line, *Implicit Consumption*, shows that only 71% of values are consumed by implicit consumers. But most notably, the bottom dashed line, *Implicit Useful Consumption*, shows that less than 25% of writes are useful to implicit consumers within 1024 instrutions. Processors serialize these writes in order for younger instructions to observe the writes' effects, but this is highly inefficient, since a majority of consumers do not find the value to actually be useful for several hundred or more instructions. We take advantage of this observations in Section 5.4 to improve the performance of writes to non-renamed registers.

A second class of frequent SIs in all platforms are writes to TLB state. TLB demap operations are common for all architectures, and are likely candidates for serialization (if the ISA requires sequential semantics for them), since all future memory operations are implicitly dependent on the demap. But until the program attempts to access the newly demapped page, the implicit consumption will not be *useful*. Similarly, other TLB control operations are not immediately useful. The first column of the table in Figure 4(b) shows the fraction of TLB writes which are useful to an instruction that speculatively enters the window before the TLB write commits (we are using a 128-entry window; details in Section 4). The second column of the table shows the average number of instructions following the write which do *not* find it to be useful. Although the number of writes which observe a useful consumption is fairly high, on average, we could continue to process 35–62 instructions after a TLB write without observing an incorrect translation. The NFS workload has many more demaps than the rest, and thus observes fewer useful consumptions. Again in Section 5.4, we use this observation to improve the performance of this class of SIs as well.

3.5 Serializing Instructions in Real Implementations

When studying serializing instructions 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 the three processor manuals for further insights.

UltraSPARC III Cu Using a table of instruction latencies and dispatch blocking properties, 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 [25]. The V9 architecture does not require sequential semantics for stores to most ASI-mapped registers and their dependent instructions, instead requiring software synchronization. It should be noted that this processor is not OoO, and has substantially different cost of serialization than the microarchitecture we model.

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 CS), 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 simultaneously will force subsequent instructions to stall [10].

Alpha 21264 The Alpha 21264 (EV6) uses Privileged Architecture Library (PAL) code to explicitly access internal (i.e. privileged) registers [7]. In the case of a write to a privileged register, the EV6 uses a scoreboard mechanism to only serialize instructions which may be dependent (through a read or a write, either explicitly or implicitly) on that register. We investigate a similar mechanism in Section 5.2. Though, for some registers, many, if not all, subsequent instructions are dependent, and the write is completely serialized.[1]

3.5.1 Real Processors 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 uses a mechanism similar 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.

4 Performance Evaluation Methodology

For the performance study, we have developed a detailed, out-of-order processor and memory model using Simics Micro-Architectural Interface (MAI). This model consists of a functional simulator (Simics) and a

[1]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.

Fetch, issue, commit	4 instructions / cycle
Integer pipeline	15 stages
Instruction window	128 entries
YAGS branch predictor	Choice table: 16k entries, exception tables: 4k entries each
Load and store queues	32 entries each, bypassing enabled
Load/store disambiguation	1k entry, direct mapped "safe distance" predictor
L1 instr. cache	32kB, 4-way, 2-banks, 2-cycle latency, coherent
L1 data cache	32kB, 4-way, 2-banks, 2-cycle latency, write-back protocol
L2 unified cache	1MB, 8-way, 14 cycle load-to-use, 4-stage pipelined, inclusive
Main Memory	265 cycles load-to-use

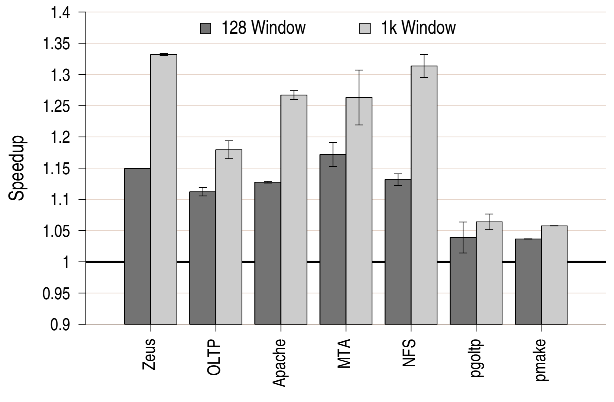
Table 3: Baseline processor parameters

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, where 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. Additionally, 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.

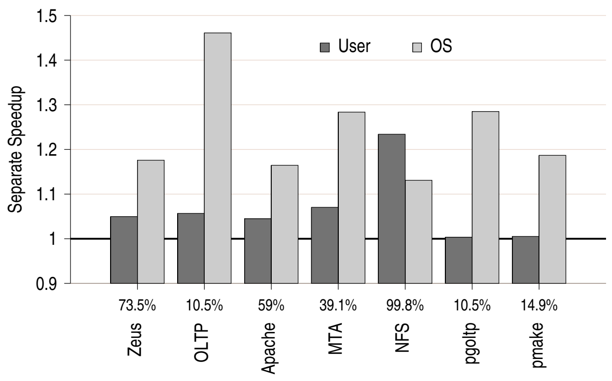
Methodology Due to inherent variability in the commercial workloads, we add small, random variations in main memory latency and run multiple trials per benchmark [2]. For statistics with especially high variability between trials, we show the 95% confidence interval using error bars in addition to the sample mean. All commercial workloads are warmed up and running in a steady state. We run these detailed timing simulations for 300M 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 Our baseline processor serializes 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. It pessimistically treats all SIs as having potential dependencies with any other instructions. Thus, when a serializing instruction is detected (typically at decode though it can be later for certain instructions such as exceptions), all younger instructions in the window are squashed and fetch is stalled. The SI is executed after all older instructions retire and it becomes the only instruction in the window.

Register window management is handled entirely within the rename logic; saves and restores do not



(a) Speedup of Non-serializing Configurations



(b) OS vs. User Non-serial Speedup (128 Window)

Figure 5: SI Performance Impact (Idealized SPARC)

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 Performance Impact and Microarchitectural Improvements

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 technique, *Effectively Useless Write Prediction* optimizes writes that are not actually useful for nearby 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 which does not serialize any instructions. This hypothetical implementation is unrealistic because assumes hardware exists to handle all explicit dependencies, and it ignores any timing constrains imposed by implicit dependencies.

We investigate two processor configurations. The first is a moderately aggressive, 15-stage, out-of-order processor with a 128-entry instruction window intended to loosely represent a modern high-performance core. The second is an optimistic, monolithic 1k-instruction window processor (load and store queus are scaled with the window, other parameters are like Table 3), which is intended solely to illustrate the potential impact of some of the future trends discussed in Section 2.4.1. In this section we focus only on ideal SPARC. We see from Figure 5(a) that for the modest OoO processor, not serializing SIs increases performance by 4–17%. For the 1k-window processor, the improvement is 5–33%. The 1k processor uses the same branch predictor as the baseline, which makes is hard to maintain a window full of useful instructions. Better branch prediction would increase the impact of serializing instructions for this configuration.

For the baseline (128-entry) system, we break down the non-serial speedup into that observed by user code and by OS code. The labels below the bars indicate the fraction of cycles the baseline spends in the OS. For all workloads (except MTA), user code observes only minor improvement, but the OS observes a 10–45% increase in performance. MTA spends $<1\%$ of the time in user mode.

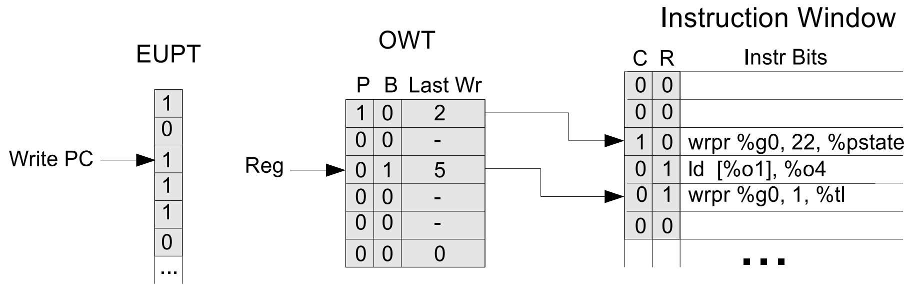
5.2 Scoreboarding Non-renamed Registers

Inspired by a vague reference to *scoreboarding* PAL code registers in the Alpha EV6 processor manual [7], we investigate a mechanism to reduce the frequency and cost of serializing write to non-renamed registers and other 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 [26] to selectively block instructions which have a dependence with an outstanding write. This mechanism may or may not be similar to that implemented by the Alpha, but is relatively straightforward, and seems to attain a performance/complexity trade-off that makes it plausible for recent processors to have implemented it.

In our implementation, the scoreboard is a table which tracks up to one outstanding write to each register. Explicit readers of non-renamed registers block at issue stage if they are involved in a RAW hazard with an outstanding SI. Younger independent instructions proceed, possibly out-of-order with respect to the SI write. Implicit readers in a RAW hazard block at issue, decode, or fetch depending on the 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 can ensure all implicit dependencies are met while waiting for an SI. But we observe that instructions can speculatively enter the window in the shadow of 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*.



(a) EU Pred Table **(b)** 1k-Instruction Window Processor (Idealized SPARC)

Figure 6: EU Prediction Mechanisms

At the risk of using more power, Late-Squash allows instructions as 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 Effectively Useless Prediction for Non-renamed/Control Registers

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 (EU) prediction to increase the performance of these writes by predicting values that will not be immediately useful to consumers, allowing them to execute OoO with respect to the SI.

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 256, 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(a).

Second, we utilize the Outstanding Write Table (OWT), which contains an entry for each control register. Entries consist of a pointer to the instruction window entry for the youngest outstanding write to the register (Last Wr); a *Pred* bit (P), indicating whether this write was predicted to be EU; and a *Blocked Front-end* bit (B), indicating whether this write has caused the front end to block. The OWT is depicted in Figure 6(b), and shows outstanding writes to two registers: the first is predicted EU, and the second is not.

We also add two bits to each instruction window entry. For control register writes, the *Consumed* (C) bit indicates if any implicit consumer has potentially accessed the register. Explicit writes only write one register. For all instructions, the R bit indicates that the instruction has potentially *Read* any of the control registers with older outstanding writes.

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, updating the *Pred* bit and instruction window pointer.

When an implicit consumer is decoded, is checks for any outstanding writes in the OWT which are predicted to be non-EU. If one is found, the consumer is squashed, and fetch is blocked until the write commits, similar to the scoreboarding technique. This was the case for the second write in Figure 6(b) (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 (using the Last Wr pointer), 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.

Multiple write to a given register can be outstanding, but only the youngest is tracked by the CWT. If writes are squashed due to a branch misprediction, for example, the CWT state needs to be restored. This requirement is similar to that of the register rename map, and can use the same mechanisms.

Commit When a write commits, its *Consumed* bit is checked and, if not set, it updates the EUPT to indicate that it was EU. If its *Consumed* bit is set (and the *Pred* bit is set in the CWT), then consumers potentially accessed a stale value. We then compare the new value to the previous value of the register, and determine whether younger instructions that potentially read the stale value were correctly executed.

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), or 2) if the write changes fields in ways that only affects excepting instructions. This latter case actually happens fairly frequently. For example, if the `FEF` (floating-point enable) field of the `%fprs` register is zero, then any floating-point instructions would generate an exception. If a write to this register sets the bit to one, younger instructions can observe the stale value of zero. But these consumers will either not be affected by the register, or will cause an exception. The same thing is true for several other, frequently written registers.

If a predicted EU write is useful *and* has observed potential consumers, all younger instructions are squashed, and the write commits before re-fetching the consumers, who will now observe the updated value.

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, and fetch is blocked until the window is empty.

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

	Apache	NFS	OLTP	pgoltp	pmake	MTA	Zeus
Extra Fetch	1.21	1.16	1.19	1.07	1.06	1.28	1.23
Extra Execute	1.13	1.09	1.12	1.04	1.03	1.14	1.15

Table 4: Fraction of Fetched and Executed Instructions using Late-Squash Compared to Baseline

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 Effectively Useless Prediction for TLB Writes

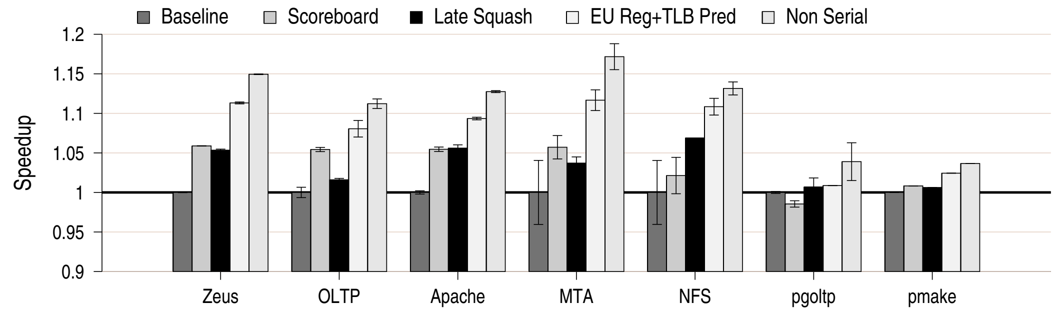
As shown in Figure 4(b), 34–63+ instructions on average can execute OoO with respect to a TLB write without affecting their translations. We simply predict that all TLB writes are effectively useless, allow the write to execute only when it becomes non-speculative, and allow younger instructions to execute OoO.

To verify this prediction, instructions present in the window during 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. This scheme behaves similarly to Late-Squash when writes are not EU, except that several additional instructions after the write are committed on average, and are not reexecuted. This scheme behaves similarly to the ideal non-serial configuration when writes are EU.

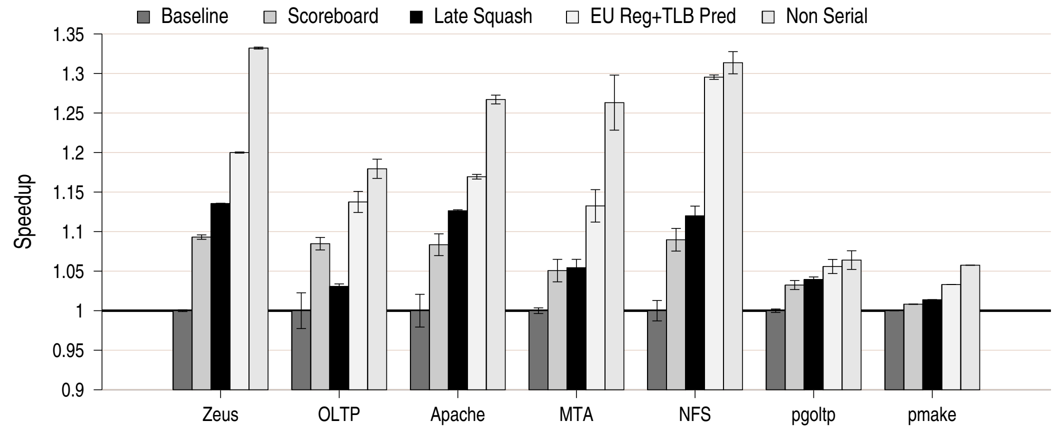
Our baseline microarchitecture does not serialize writes to ASI-mapped registers, since SPARC does not require sequential semantics for them. However, as noted in Section 2.2, an implementation could provide sequential semantics, eliminating the need for explicit synchronization. Though not guaranteed 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 these 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 Performance of SI Mitigation Techniques

Figures 7(a) and 7(b) show the speedup when using these techniques. The second bar for each benchmark shows the speedup over the baseline configuration when scoreboarding both accesses to non-renamed registers, and accesses to non-renamed ASI-mapped registers. Scoreboarding results in a 0–5% performance improvement for the 128-entry window, and 0–10% for the 1k-entry window.



(a) Baseline OoO Processor (Idealized SPARC)



(b) 1k-Instruction Window Processor (Idealized SPARC)

Figure 7: Speedup of SI Mitigating Techniques

The third bar shows the performance of Late-Squash. Some benchmarks benefit from late-squash as much as they do from scoreboarding. Benchmarks with many off-chip misses, notably Zeus and Apache, benefit the most. To give a rough idea of extra power and front-end bandwidth that may be required, Table 4 shows the increase in fetched and executed instructions for the same number of committing instructions. For Apache, 13% more instructions are executed, and 21% more are fetched. While late-squash is relatively simple to implement, it is unlikely that the additional power would justify 0–5% performance improvements.

The performance of EU Prediction for control registers and TLB writes is shown as the fourth bar in Figure 7. For the baseline, 128-entry window processor, EU Register and TLB prediction provide 8–12% speedups for five workloads, coming within 5% of the ideal non-serializing configuration for all workloads. For the monolithic 1k-entry window processor, EU prediction provides 3–30% speedups, but only comes within 15% of the ideal non-serial configuration. EU prediction for TLB writes incurs an additional 0.2–1.0% of TLB lookups for instructions executed in the shadow of a TLB write.

6 Related Work

Chou, et al., briefly mention SIs in the context of atomic instructions and memory barriers for synchronizing multiple threads [6]. While such instructions are potential SIs, we focus instead 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 [22].

Similar to Late-Squash, Runahead [8,16] is a prefetching technique that aims to continue execution during a long-latency event such as a cache miss. Unlike the Late-Squash mechanism, runahead mode is not entered until the long-latency instruction reaches the head of the window, and thus, is unlikely to provide benefit.

Zilles, et al., [28] and Jaleel, et al., [12] propose mechanisms for handling exceptions without serializing. Such mechanisms are especially important for handling software TLBs, and are orthogonal to our proposals.

Real processor such as the Alpha EV6 [7] likely implement mechanisms similar to the scoreboard we examine. To our knowledge, effectively useless prediction is a novel application of a new observation.

7 Conclusions

In this paper, we identify *serializing instructions* (SIs) as a limiting performance factor for system-intensive workloads. We analyze the frequency of SIs from several workloads across three ISAs, SPARC V9, X86-64, and PowerPC, and show that seven system-intensive workloads observe a 5–17% performance loss compared to a hypothetical non-serializing processor. We also show that two simple techniques, *Late-Squash*, and *Scoreboarding* non-renamed registers, which are likely to have been implemented in real processors, can improve performance by 0–5%, but fall considerably short of the ideal, non-serial processor.

We examine the *use* of register values produced by SIs and discover that, while 75% of values are consumed within 128 instructions, 90% of values are *effectively useless* — i.e., they write a different value than the previous contents of the register, but the new value does not actually change the execution of the consuming instructions. Using this information, we propose a novel technique which predicts when a value will be *effectively useless* during the time the producer is in-flight, and speculatively allows consumers to read a stale value. With this technique, we observe speedups of 8–12% for five of our workloads on a 128-entry instruction window processor, and come within 5% of the ideal, non-serial processor for all workloads. Speedups are 4–30% for all workloads on a hypothetical 1k-window processor.

References

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

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

[3] 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.

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

[5] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proc. of 14th SOSP*, pages 120–133, New York, NY, USA, 1993. ACM Press.

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

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

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

[9] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. Technical Report CS-TR-2007-1593, University of Wisconsin-Madison, April 2007.

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

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

[12] A. Jaleel and B. Jacob. In-line interrupt handling for software-managed tlbs. In *Proc. of ICCD*, 2001.

[13] 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.

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

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

[16] 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.

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

[18] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of 27th ISCA*, 2000.

[19] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. of 29th Fault-Tolerant Computing*, 1999.

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

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

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

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

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

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

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

[27] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. 2005.

[28] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proc. of 32th MICRO*, pages 219–229, Washington, DC, USA, 1999. IEEE Computer Society.