To appear in Proceedings of SPAA-9 (June, 1997)

Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models *

Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77005
{parthas|vijaypai|sarita}@rice.edu

Abstract

This paper studies techniques to improve the performance of memory consistency models for shared-memory multiprocessors with ILP processors. The first part of this paper extends earlier work by studying the impact of current hardware optimizations to memory consistency implementations, hardware-controlled non-binding prefetching and speculative load execution, on the performance of the processor consistency (PC) memory model. We find that the optimized implementation of PC performs significantly better than the best implementation of sequential consistency (SC) in some cases because PC relaxes the store-to-load ordering constraint of SC. Nevertheless, release consistency (RC) provides significant benefits over PC in some cases, because PC suffers from the negative effects of premature store prefetches and insufficient memory queue sizes.

The second part of the paper proposes and evaluates a new technique, speculative retirement, *to improve the performance of SC. Speculative retirement alleviates the impact of the store-to-load constraint of SC by allowing loads and subsequent instructions to speculatively commit or retire, even while a previous store is outstanding. Speculative retirement needs additional hardware support (in the form of a history buffer) to recover from possible consistency violations due to such speculative retires. With a 64 element history buffer, speculative retirement reduces the execution time gap between SC and PC to within 11% for all our applications on our base architecture; a significant, though reduced, gap still remains between SC and RC.*

The third part of our paper evaluates the interactions of the various techniques with larger instruction window sizes. When increasing instruction window size, initially, the previous best implementations of all models generally improve in performance due to increased load and store overlap. With further increases, the performance of PC and RC stabilizes while that of SC often degrades (due to negative effects of previous optimizations), widening the gap between the models. At low base instruction window sizes, speculative retirement is sometimes outperformed by an equivalent increase in instruction window size (because the latter also provides load overlap). However, beyond the point where RC stabilizes, speculative retirement gives comparable or better benefit than an equivalent instruction window increase, with possibly less complexity.

1 Introduction

Shared-memory multiprocessors are being increasingly used for a variety of applications. The performance of such systems, however, depends largely on their ability to tolerate memory latency. The memory consistency model of a shared-memory system is a key attribute that determines this ability by specifying the extent to which the system can appear to overlap or reorder memory operations. In previous-generation systems, memory consistency models posed an essential tradeoff between programmability and performance: the more intuitive models placed stronger constraints on system implementations. Of the models that are most widely implemented in commercial systems, sequential consistency (SC) [12] (adopted by HP and MIPS processors) is considered to provide the most intuitive programming interface while release consistency (RC) [4] (variants of which are adopted by DEC Alpha, IBM PowerPC, and SPARC V9 RMO-based processors) has been shown to provide the highest performance [5, 27]. Processor consistency (PC) [7, 4] (adopted by Intel processors) and total store ordering (TSO) [22] (adopted by SPARC processors) are two other commonly implemented models, and fall between SC and RC in performance [5].

The increased levels of instruction-level parallelism (ILP) employed by current processors enable two hardware optimizations to enhance the performance of consistency models – hardware-controlled prefetching and speculative load execution [6]. These optimizations exploit ILP features such as non-blocking loads, out-of-order scheduling, and speculative execution. The optimizations have already appeared in several commercial microprocessors (e.g. HP PA-8000 [8], Intel Pentium Pro [9], MIPS R10000 [13]) and were conjectured to equalize the performance of consistency models. However, recent work has shown that while these optimizations substantially narrow the performance gap between SC and RC, a significant gap remains for some of the applications considered [14]. This gap occurs because SC is less effective than RC in hiding store latency with current processors. Two key effects that result in this limitation are (i) the relatively

*This work is supported in part by the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, and CDA-9502791, and the Texas Advanced Technology Program under Grant No. 003604016. Vijay S. Pai is also supported by a Fannie and John Hertz Foundation Fellowship.

small instruction windows of current processors typically do not enable high-latency stores to be fully overlapped with other instructions in the window, and (ii) the store-to-load ordering requirement of SC is implemented by preventing a load from committing or retiring from the instruction window until previous stores are complete. These conditions imply that a high latency store can stop the flow of instructions through the processor's instruction window. RC, on the other hand, does not impose a constraint between stores and following loads to data locations, allowing most memory operations to retire from the instruction window even while previous stores are incomplete.

This paper makes three contributions. For the evaluations in all of the three parts, we run six applications on a detailed execution-driven simulator of an ILP-based multiprocessor.

As our first contribution, we evaluate the impact of the hardware prefetching and speculative load optimizations on PC and TSO. For our base system, PC and TSO have identical implementations; we therefore use PC to refer to both PC and TSO in the rest of the paper. PC differs from SC by relaxing the store-to-load constraint, which was a primary contributor to the performance difference between SC and RC in the previous study [14]. Our results show that the optimizations of hardware prefetching and speculative loads significantly improve the performance of PC. Comparing SC with PC, we find that PC gives 15% or more reduction in execution time in two of our six applications. However, RC still gives more than 15% reduction in execution time over PC in two of our six applications. The remaining difference between PC and RC occurs primarily due to the negative effects of premature store prefetches and insufficient memory queue sizes.

Our second contribution is a new optimization for SC, *speculative retirement*, that targets the limitation due to the store-to-load constraint of SC. Loads stalled only for previous incomplete stores are allowed to speculatively commit their values into the processor's architectural state and retire from the instruction window, freeing up space for later instructions. This technique can potentially eliminate the impact of the store-to-load constraint (potentially equalizing the performance of SC and PC), but at an additional cost in hardware. Hardware is needed to detect possible consistency violations due to speculatively retired loads and to recover from such violations. The detection hardware is the same as that used for the earlier optimization of speculative load execution. For recovery, we use a mechanism similar to a history buffer that was originally proposed for maintaining precise interrupts [21].

We study the performance benefits of speculative retirement on our base architecture with up to 64 element history buffers. We find that, for the two applications for which there originally was a significant execution time gap between SC and PC, speculative retirement reduces the gap from 15% to 5% in one application and from 24% to 11% in the other application. Thus, speculative retirement reduces the execution time gap between SC and PC to within 11% for all our applications with our base architecture. A significant, though much reduced, gap between SC and RC remains for two of our six applications. These results indicate that for our applications and architecture, with some hardware investment, the performance advantages offered by PC may not justify its additional programming complexity.

An alternative to speculative retirement to improve the performance of SC is to increase the instruction window size of the processor. We use the term instruction window to refer to the buffer that tracks all the in-flight instructions in the system (including instructions that have completed but have not yet committed). This may be different from the issue window, which only tracks instructions that are not yet issued to the functional units. Larger instruction windows can potentially increase the overlap available to stores and address the store latency responsible for the performance difference between SC and relaxed models. We compare the performance benefits of equal increases of instruction window and history buffer sizes over our base architecture, up to a doubling of the instruction window size. We find that for the applications that initially showed a significant performance gap between PC and SC, the two techniques of speculative retirement and larger instruction window size give comparable improvements. A larger instruction window sometimes gives better improvements because it increases overlap for both loads and stores while speculative retirement only targets limited store overlap. However, speculative retirement gives better improvements in some cases because it suffers less from the negative impacts of premature store prefetching and speculative loads. From an implementation point of view, the speculative retirement scheme appears less likely to impact cycle time or the number of pipeline stages compared to an increase in the instruction window size.

The third contribution of this paper is an evaluation of the interaction of the various optimizations with much larger instruction window sizes as may be found in future processors. We first evaluate the effect of large instruction window sizes on the best implementations of the various consistency models, not including speculative retirement. We find that as instruction window sizes increase, initially, these implementations generally improve in performance due to increased load and store overlap. However, with further increases, the negative effects of premature store prefetching and speculative loads in SC start to dominate for some applications, and the performance gap between the consistency models for these applications increases. The performance of PC and RC, however, stabilizes at some instruction window size for each application because of diminished potential for load overlap beyond this point. SC, however, may still be able to benefit from increased store overlap beyond this point. We find that, beyond this point, speculative retirement gives comparable or better benefits than an equivalent increase in the instruction window size, with possibly less complexity.

The remainder of this paper is organized as follows. Section 2 gives background on current implementations of consistency models. Section 3 describes the speculative retirement technique. Section 4 describes our simulation environment. Section 5 discusses the performance and limitations of previous implementations of SC, PC, and RC. Section 6 evaluates the performance of speculative retirement. Section 7 evaluates the interaction of the various optimizations with large instruction window sizes. Section 8 discusses possible future directions motivated by our results. Section 9 concludes the paper.

2 Background

This section gives background information about memory consistency models, particularly focusing on implementations for multiprocessors built from ILP processors. We assume state-of-the-art ILP processors that examine several instructions at a time, issuing independent instructions in parallel and potentially out of program order. Although instructions may complete out of program order, they modify the architectural state of the processor in program order to maintain precise interrupts [21]. For this purpose, all in-flight instructions are stored in an instruction window (also referred to as the active list or reorder buffer) in program order. Instructions leave the window (or retire) in program order, and modify the architectural state of the processor only when they retire.

The memory consistency model of a shared-memory system is an architectural specification of the order in which memory operations must *appear* to execute to the programmer. The most intuitive model, *sequential consistency* (SC) [12], guarantees that all memory operations will appear to execute one at a time and in program order. *Processor consistency* (PC) [7, 4] relaxes some of the ordering requirements of sequential consistency. Specifically, with PC, a load following a store (to a different location) in program order may appear to be executed out of program order. *Release consistency* (RC) [4] further relaxes ordering constraints, allowing arbitrary reordering of memory operations (to different locations) except at synchronization points.

Straightforward implementations of the consistency models can enforce their ordering constraints by prohibiting a memory operation from entering the memory system until all previous operations of its processor for which it must appear to wait (as defined by the consistency model) have completed[1]. However, ILP processors allow more aggressive implementations by exploiting the observation that the memory consistency model of a system only requires that the system *appear* to execute memory operations according to the specified constraints.

The technique of *hardware prefetching* from the instruction window [6] hides some memory latency by issuing non-binding prefetches for instructions in the instruction window for which addresses are known, but which are blocked for consistency constraints. *Speculative load execution* [6] increases the benefits of prefetching by actually consuming the values of locations prefetched into the cache, regardless of consistency constraints. The technique requires on-chip hardware support to detect any violations of ordering requirements due to early consumption of values, and hardware support to recover from such violations. A violation is detected by monitoring for coherence requests and secondary cache replacements for cache lines accessed by outstanding speculative loads. The mechanism for recovery is similar to that used to recover from branch mispredictions or exceptions. Both of the above techniques are implemented in current systems with SC [8, 13] and PC [9].

Store buffering [5] is a technique that allows stores to retire from the head of the instruction window even before they complete. Store buffering is already implemented in current implementations of PC and RC (the buffering logic is responsible for maintaining any ordering between multiple stores imposed by the consistency model). However, on SC, the benefits of store buffering are unclear since the system now blocks on the first load to reach the head of the instruction window while stores are pending. Current commercial systems support both implementations of SC – with store buffering [8] and without store buffering [13]. In this study, we analyze the benefits due to store buffering separately only for SC, and integrate store buffering with the straightforward implementations of PC and RC.

3 A New Technique to Improve the Performance of SC

3.1 The Speculative Retirement Technique

Speculative retirement alleviates the impact of the store-to-load ordering constraint in SC. With current SC implementations, when a completed load reaches the head of the instruction window, it must wait for all previous stores to complete before it can retire or commit its value. Thus, a high-latency store can block the flow of instructions through the processor's instruction window. Speculative retirement allows completed loads at the head of the instruction window (and subsequent completed instructions) to speculatively retire and commit their values even while previous stores are outstanding. Thus, instructions are never prohibited from retiring from the instruction window solely due to consistency constraints. If the processor later detects that a speculatively retired load may have caused a consistency violation, it activates a recovery mechanism to rollback to a valid state before the offending load. Analogous to speculative load execution, we expect speculative retirement to be successful in most cases and rollbacks to be infrequent.

To be effective, a system with speculative retirement should also use the earlier optimizations of hardware store prefetching, speculative load execution, and store buffering; this paper assumes that the earlier optimizations are used whenever speculative retirement is used. An SC system with speculative retirement appears similar to a PC system that employs the earlier optimizations. There is still a difference between the optimized SC and RC systems, however, since SC with speculative retirement continues to impose ordering between stores (similar to PC).

3.2 Implementation of Speculative Retirement

Analogous to speculative load execution [6], the implementation of speculative retirement consists of three parts – a speculation mechanism that allows speculative retires past an incomplete store, a detection mechanism that identifies possible violations of memory ordering constraints, and a recovery mechanism that allows the system to recover to a valid state after a violation. We discuss one possible implementation of these mechanisms below.

Speculation mechanism. Speculative retirement is initiated when the following two conditions are satisfied: i) a load at the head of the instruction window has completed, but cannot retire solely due to consistency constraints (a previous store is incomplete) and ii) there are no free entries in the instruction window (i.e., the flow of instructions through the instruction window of the processor is stalled due to the instruction at the head of the window). Once speculative retirement is initiated, a subsequent instruction following the load is speculatively retired when the instruction is complete and at the head of the instruction window and the instruction window is full; stores are an exception since they are retired as soon as they reach the top of the instruction window as before. We use a structure similar to a history buffer [21] to store information about the instructions that are speculatively retired[2]. For recovery from a possible violation later, the history buffer also stores the previous value and the old logical-to-physical register mapping

[1] A load operation is complete when the processor receives the value for the load. A store operation is complete when all the processors in the system have seen the value of the store, usually indicated by an acknowledgment from the directory or memory to the store's processor.

[2] An alternative, but potentially more complex, implementation could checkpoint the state of the processor when speculative retirement is initiated.

(assuming register renaming similar to the MIPS R10000) of the destination register of the speculatively retired instruction. An entry is removed from the history buffer when all previous stores have completed.

In contrast to a traditional history buffer where the history buffer entries are used to retire all instructions [21], the speculative retirement implementation uses the history buffer only to store state needed to recover from possible consistency violations. The history buffer entries are used only when instructions are stalled at the head of the instruction window solely due to consistency requirements (i.e., whenever the flow of instructions is stopped due to a high latency store operation). Additionally, stores and branches are not added to the history buffer, as these instructions have no destination register and do not change the state of the processor.

This implementation adds a small amount of area for the history buffer. For a 64-bit processor, each entry in the history buffer needs about 18 bytes of information (8 bytes for the program counter, 8 bytes for the old value of the destination register, and 2 bytes for the register mapping information), requiring about 1 KB of storage for a 64-entry history buffer. The history buffer may also require additional register ports to read the previous values of destination registers. Alternatively, an implementation can exploit the observation that the history buffer is activated only when there is a long latency store and the instruction window is full. Thus the history buffer can share ports already used by functional units, just as some processor designs already have functional units sharing ports with each other [2].

Detection mechanism. The detection mechanism to identify possible violations of ordering constraints with speculative retirement is identical to that used with speculative load execution, involving monitoring of coherence requests and secondary cache replacements for cache lines accessed by outstanding speculative loads [6, 14].

Recovery mechanism. If the violating load is in the instruction window (i.e., it has not been speculatively retired yet), recovery proceeds as for speculative load execution, using the processor's exception recovery or branch misprediction logic. If the violating load has already retired, then the recovery mechanism uses the history buffer to restore the processor's architectural state to a valid state. Specifically, the history buffer entries must be processed in reverse order, using the history information to restore the previous values of the registers used and the previous mappings for registers (assuming register renaming similar to the MIPS R10000). This is similar to the processing needed when history buffers are used to support precise exceptions [21].

4 Simulation Methodology

The following subsections respectively describe the simulated architectures, the simulation methodology and environment, the performance metrics, and the applications used in this work. These are similar to those used in several previous studies [14, 16, 18].

4.1 Simulated Architectures

Memory System and Network. We simulate a hardware cache-coherent shared-memory multiprocessor with a full-mapped, invalidation-based, three-state directory coherence protocol. The processing nodes are connected by a two-dimensional mesh network. Each processing node consists of a processor, two levels of cache, and a part of the main memory and directory. In our cache hierarchy, the first level cache is dual-ported, write-back, and write-allocate. The second-level cache is pipelined, write-back, and write-allocate. Both levels are non-blocking with eight Miss Status Holding Registers (MSHRs) [11].

ILP Processor	
Processor speed	300MHz
Maximum fetch/retire rate (instructions per cycle)	4
Instruction issue window	64 entries
Functional units	2 integer arithmetic 2 floating point 2 address generation
Branch speculation depth	8
Memory queue size	64 entries
Network parameters	
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles
Cache parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped, 16 K
L1 request ports	2
L1 hit time	1 cycle
Number of L1 MSHRs	8
L2 cache (off-chip)	4-way associative, 64 K
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of L2 MSHRs	8
Memory parameters	
Memory access time	18 cycles (60 ns)
Memory transfer bandwidth	16 bytes/cycle
Memory interleaving	4-way

Figure 1: System parameters.

When there is a store or exclusive prefetch request to a line that has a load pending, the MSHR logic stalls the later request and issues an ownership request only when the load reply returns, as in many current system implementations. We refer to such stalls as *write-after-read stalls*. The alternative, allowing this ownership request to overlap with the previous read request, increases the complexity at the directory controller and at the MSHRs, since both components would need to handle potential reordering of requests in the network. In addition to preventing overlap of the store ownership request with load latency, our implementation of write-after-read stalls also blocks a cache port, possibly preventing later requests from issuing to the cache.

Figure 1 gives our default memory system parameters. We have chosen smaller cache sizes than commercial systems, commensurate with our application input sizes (Section 4.4) and following the working-set evaluations of Woo et al. [25]. Our secondary caches are chosen such that secondary working sets of our applications do not fit in cache; we choose primary cache sizes such that fixed-size primary working sets in our applications fit in cache.

Base Processor Model. To exploit instruction-level parallelism, our default processor model employs widely used techniques like multiple instruction issue, dynamic (out-of-order) scheduling, register renaming, speculative execution, and non-blocking loads. The processor microarchitecture is most closely based on the MIPS R10000 design [13]. The processor includes a memory queue, which is used for issuing memory operations and for maintaining consistency constraints.

Figure 1 gives the processor parameters used in our simulations. These parameters were chosen to model next-generation aggressive processors. The default latencies for the various execution units approximate those for the UltraSPARC [23].

Application	Input Size	Processors
Erlebacher	64 by 64 by 64 cube, block 8	16
FFT	65536 points	16
LU	256 by 256 matrix, block 8	8
MP3D	50000 particles	8
Radix	1024 radix, 512K keys, max 512K	8
Water	343 molecules	16

Figure 2: Application parameters.

Processor Variations. We model variations on the base processor model that implement different consistency models (SC, PC, and RC). For each of these consistency models, the processor can support a straightforward implementation, an optimized implementation that supports hardware prefetching and speculative load execution, or an optimized implementation that also supports speculative retirement (on SC systems only). As discussed in Section 2, the processor supports a separate implementation with store buffering only for SC; store buffering is already integrated in the straightforward implementations of PC and RC.

We implement hardware prefetching and speculative load execution as discussed in Section 2. We force a rollback when the system observes a coherence request or a secondary cache replacement for a cache line accessed by an outstanding speculative load. We do not rollback on primary cache replacements since those lines will still remain visible to external coherence. The mechanism used to restart execution on a rollback is similar to, but more optimistic than, that used in the MIPS R10000 [13]. As in the MIPS R10000, we require a violating speculative load to reach the head of the instruction window before activating a recovery; however, we optimistically assume a zero cycle recovery penalty (the number of cycles that are taken to flush subsequent instructions from the instruction window). Our results show that, on our base system (with 64-element instruction window sizes), the number of rollbacks is very small, and thus the rollback penalty does not significantly impact the performance of the optimized models. However, in our studies with larger instruction windows (Section 7), this assumption gives an upper bound on the performance of speculative loads. Speculative retirement is implemented as discussed in Section 3 sharing features of speculative load execution.

In our implementations, speculative loads and retired but incomplete stores, are maintained in the memory queue until their consistency constraints are met. We use a memory queue size larger than those found on current processors to account for the increased stress on memory queue resources. An alternative implementation could keep speculative loads and outstanding stores in a separate buffer to reduce the pressure on memory queue resources.

4.2 Simulation Methodology and Environment

We use RSIM (the Rice Simulator for ILP Multiprocessors), a detailed execution-driven simulator that models both the processor pipelines and the memory subsystem, including contention at various resources [15]. RSIM uses application executables rather than traces so that interactions between the processors during the simulation can affect the course of the simulation.

The applications are compiled with a version of SPARC V9 gcc modified to eliminate branch delay slots and restricted to 32 bit code, optimized with `-O2 -funrollloop`.

To speed up the simulation, we assume all instructions hit in the instruction cache (with 1 cycle hit time) and private (i.e., non-shared) variables also hit in the data cache. These assumptions are widely used by previous work.

4.3 Performance Metrics

We divide execution time into its various components, namely CPU time and stall time due to data loads, data stores (henceforth referred to as just loads and stores respectively), and synchronization. We follow the convention of several previous studies with ILP processors [19, 14, 16, 18] to assign busy and stall time. We count a cycle as part of busy time if we can retire the maximum number of instructions possible in that cycle (four in our system). Otherwise, we charge that cycle to the stall time component corresponding to the first instruction that could not retire that cycle.

4.4 Applications

We use six applications in this study – Radix, FFT and LU from the SPLASH-2 suite [25]; Water and MP3D from the SPLASH suite [20]; and Erlebacher, obtained from the Rice parallel Fortran compiler group [1]. These applications were slightly modified from their original distributions as discussed in our previous work [14, 16, 18]. Both LU and FFT include loop transformations and procedure inlining optimizations in order to better exploit non-blocking loads in ILP processors (similar to those discussed in [16]).

Figure 2 gives the input sizes and system sizes used for the various applications. Because our simulation times are much higher than those seen in studies using direct-execution or trace-driven simulation, we were restricted to using problem sizes one size smaller than generally recommended for LU and Water. The system sizes are chosen for various applications depending on the scalability of the application. Since the recommended problem size for LU and Water are for configurations with up to 64 processors, and we run these applications on much smaller configurations, we do not expect the decreased input sizes with these applications to qualitatively affect our results.

5 Performance of Current Consistency Implementations

This section discusses the impact of the hardware prefetching, speculative load execution, and store buffering optimizations on the performance differences between SC, PC, and RC, with the base architecture described in Section 4. The results for PC and for the impact of store buffering with SC are presented for the first time. The other results appeared in our previous study [14] and are presented here for comparison with PC and the store buffering optimization, and also to provide necessary background for the following sections.

Figure 3 summarizes our results. `Plain` represents a straightforward implementation of each consistency model, `+SL` adds hardware prefetching and speculative loads to the `Plain` system[3], and `+SB` adds store buffering to the `+SL` system. For each application, we show the execution times of the `SC+SL`, `SC+SL+SB`, `PCplain`, `PC+SL`, `RCplain`, and `RC+SL` systems, normalized to the time for the `SC+SL` system[4]. Each execution time bar is further subdivided into four components – CPU, load stall time, store stall time, and synchronization stall time. We do not show the times for `SCplain`

[3] We do not show benefits of hardware prefetching and speculative loads separately for brevity. The benefits of only hardware prefetching for SC and RC appear in [14]. For PC, hardware prefetching helps most applications; speculative loads finds further benefits for most applications.

[4] Some numbers in Figure 3 differ from those in [14] because of the loop transformations in FFT and LU discussed in Section 4, and a difference in cache sizes.

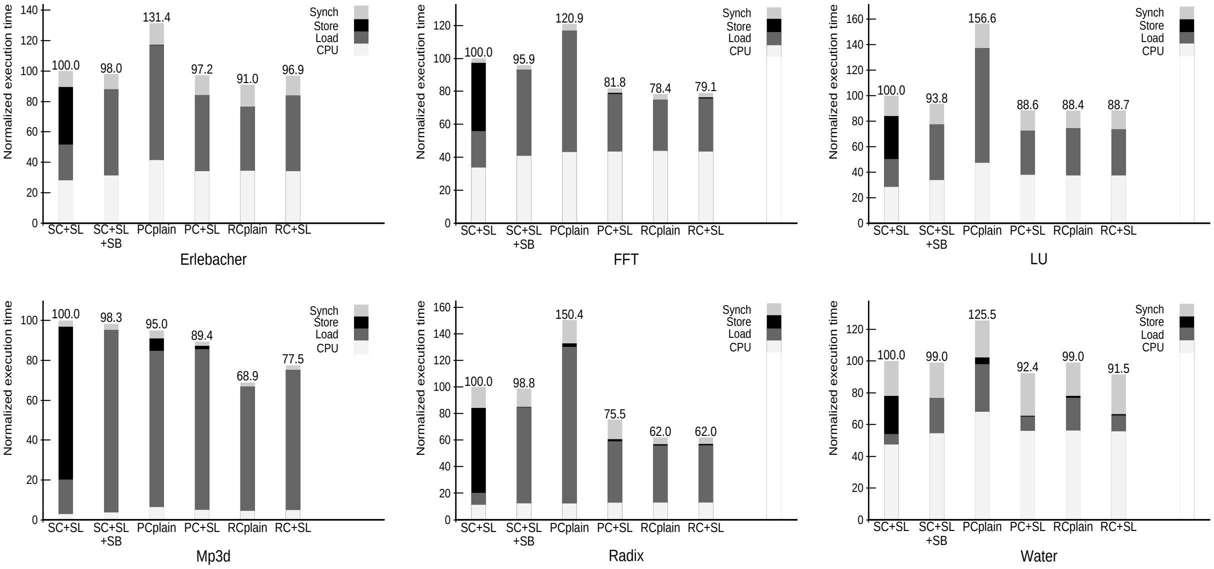

Figure 3: Evaluation of current optimized consistency implementations.

because our previous study showed that SC+SL performs significantly better than SCplain for our applications.

Section 5.1 summarizes our overall results. Section 5.2 describes the factors that contribute to the performance difference between SC, PC, and RC, and Section 5.3 identifies the factors that determine the performance of individual applications.

5.1 Overall Results

The best SC and PC implementations for all applications are SC+SL+SB and PC+SL respectively. Although hardware prefetching, speculative loads, and store buffering optimizations result in significant benefits for SC [14] and PC, the more relaxed models continue to achieve significantly higher performance compared to the less relaxed models. In the case of PC, the optimizations are necessary for PC to outperform the best SC. The optimizations do not have much impact on the performance of RC. For each pair of models, the following lists the applications for which the relaxed model offers 15% or more reduction in execution time compared to the best implementation of the stronger model (the parenthetical number is the actual reduction in execution time offered by the relaxed model):

SC vs. PC: FFT (15%), Radix (24%).

PC vs. RC: Radix (18%), Mp3d (23%).

SC vs. RC: FFT (18%), Mp3d (30%), Radix (37%).

5.2 Factors Contributing to Performance Differences

The difference in the performance of the various consistency implementations arises primarily due to a difference in data memory stall time. The difference between PCplain and the other systems arises because PCplain prevents the overlap and reordering of two loads, two stores, and a load followed by a store in program order. In contrast, the other implementations allow such overlap and reordering with either demand or prefetch accesses. This gives other systems a significant advantage over PCplain except in the case of Mp3d. In Mp3d, PCplain slightly outperforms the best SC system due to the negative effects of premature prefetches associated with the optimized implementations of Mp3d (discussed below). We next discuss six factors that affect the performance differences between implementations other than PCplain.

Limited instruction window size. With current instruction window sizes (64 instructions for our processor), it may not be possible to overlap the entire latency of a memory instruction with other instructions in the window, unless there are other equally high latency memory instructions in the instruction window. Loads that are not completely overlapped are handled similarly in all implementations; however, the manner in which stores that reach the head of the instruction window are handled can lead to differences between the consistency implementations. In SC+SL, stores are retired from the instruction window only when they complete. Therefore, in SC+SL, incomplete stores can block the flow of instructions through the instruction window. Additionally, stores retire only one at a time in SC+SL. SC+SL+SB, in contrast, can retire (possibly multiple) stores from the instruction window even while they are incomplete, buffering them in the memory queue. A subsequent completed load, however, cannot retire until previous stores complete. PC systems are more aggressive since they allow stores, as well as subsequent completed loads, to retire while the stores are incomplete. Stores in PC lead to stall time only when the memory queue is full due to previous incomplete (but retired) stores. Finally, RC systems are the most aggressive since they not only retire incomplete stores, but also issue multiple demand stores in parallel, and thus do not need to buffer incomplete stores in the memory queue after issue.

Thus, the limited instruction window size can have a negative impact on all our implementations, but the impact increases from the RC systems to PC+SL to SC+SL+SB to SC+SL.

Limited memory queue size. If the memory queue is full, then the issue of subsequent demand or prefetch requests is delayed. This can potentially cause the instruction window to fill. The memory queue full time depends on (i) the rate at which memory instructions arrive in the queue (arrival rate), and (ii) the time a given memory instruction occupies the queue (occupancy). The arrival rate is typically higher for the more relaxed models because these models have fewer constraints on retirement and so instructions enter and leave the instruction window (and hence the memory queue) at a higher rate. On the other hand, the occupancy decreases with the more relaxed models. Specifically, a load occupies a memory queue entry only until the load and all previous acquires complete for RC, until all previous loads complete for PC, and until all previous loads and stores complete for SC. A store occupies a memory queue entry only until it is issued to the cache in RC, and until all previous stores complete in PC and SC. The net effect of the differing arrival rates and occupancies for our applications is that the memory queue fill time is high for some applications with `PC+SL` and negligible for all applications with SC and RC.

Early store prefetches. Store prefetching does not result in reduced latency if the prefetched line is either invalidated, replaced, or downgraded to read-only state before the corresponding demand access. Such prefetches are called *early prefetches.* Such prefetches may actually degrade performance if they invalidate a line that is yet to be used by the remote processor, as may be possible in the presence of false sharing, races, or if the prefetch occurred before the synchronization corresponding to the data access. Early prefetches may also degrade performance if they replace a line that will be used before the demand access. Finally, early prefetches may increase system contention, hindering other more useful accesses. These negative effects of early prefetches are present in all implementations with store prefetching, but have the largest impact with the SC implementations since SC stalls retirement at an incomplete store (with `SC+SL`) or at the first load following an incomplete store (with `SC+SL+SB`).

Early prefetches can occur when the prefetch is issued sufficiently in advance of the demand access. Thus, a larger instruction window size (for SC and PC) and a larger memory queue size (for PC) can further increase the number of early prefetches. Additionally, the exposed latency due to an early prefetch can increase the time between subsequent prefetches and their corresponding demand accesses, thus increasing the probability that other prefetches are also invalidated, replaced, or downgraded before their use.

Early speculative loads. Speculative loads in the `+SL` implementations may also degrade performance if they result in rollbacks, in cases analogous to early store prefetching. With our implementation, a rollback is triggered when the offending load reaches the head of the instruction window; therefore, the work lost due to rollback can increase with a larger instruction window size and may be significant. Further, the number of rollbacks can also increase with larger instruction window sizes, as such window sizes allow speculative loads to issue earlier.

Stronger memory models can see more rollbacks than more relaxed models, as loads must wait longer before being considered non-speculative. Specifically, loads in SC must wait for all previous loads and stores to complete and loads in PC must wait for all previous loads, whereas loads in RC must only wait for previous acquires to complete.

Write-after-read stalls. As described in Section 4, if a cache sees a store request to a line for which it has a load request outstanding, the store request is stalled leading to potential cache stalls. Such write-after-read stalls are not usually seen with `Plain` implementations, as the stores following loads to the same line generally depend on the values of those loads and are thus not issued in `Plain` until the loads complete. In the `+SL` implementations, however, store prefetches are issued even while the values of the stores are unknown, leading to possible performance degradations due to write-after-read stalls. Again, a delay in stores hurts the SC implementations the most.

Application characteristics. The extent to which the above factors influence performance is greatly determined by application characteristics. In our experiments, a key aspect is whether misses occur together with other independent memory misses of similar or higher latency within the space of an instruction window. We call misses that occur together in an instruction window as *clustered misses.* Latencies of clustered misses are overlapped with each other, mitigating part of the negative influence of the limited instruction window and memory queue sizes. Clustered loads impact all our implementations in a similar manner since loads are similarly handled by all implementations. Clustered stores, however, have a greater impact on stronger models than on more relaxed models, and so can narrow the performance gap between the models. A store clustered with a load is more effective at closing this gap; two stores clustered with each other only partly address the gap since the latency of one of the stores is still seen.

5.3 Application-Specific Analysis

This section identifies the dominant factors from Section 5.2 responsible for the performance differences among the various implementations for a specific application.

FFT, Radix, and Mp3d are the applications that see the most disparity in performance among the different consistency models. In FFT and Mp3d, stores are not clustered with other independent misses of similar latency. Therefore, these applications exhibit the negative effects of limited instruction window size (primarily for SC) and limited memory queue size (for PC). Further, for Mp3d, the versions that employ store prefetches see a performance degradation due to write-after-read stalls and due to early prefetches occurring from high false sharing and races. Finally, certain phases of both Radix and Mp3d have stores with addresses dependent on the values of previous loads. This serializes store prefetches behind loads, aggravating the impact of limited instruction window and memory queue sizes.

Erlebacher, LU, and Water see less than 10% performance difference between the different consistency models. In Erlebacher, stores are clustered with other stores, resulting in a lower performance gap among the different models. Write-after-read stalls incurred by SC and PC, however, result in a slight performance degradation in these models compared to `RCplain`.

In LU, most L1 store misses are L2 hits (caused by L1 conflicts). These L2 hits are often clustered with remote load misses, narrowing the gap between the consistency models. These stores show write-after-read stalls, but usually the offending loads are also L2 hits and the stores have enough time to overlap behind other remote load misses. Nevertheless, PC performs a little better than SC because there are a few remote store misses that PC is able to overlap better than SC. Note that LU is the only application for which the addition of store buffering to SC (`SC+SL+SB` system) gives more than 5% improvement in execution time. This improvement occurs for two reasons. First, store prefetching is often successful, so a large part of the store latency in the `SC+SL` case is due to the limited rate of retirement of stores; the `SC+SL+SB` system can retire stores at the peak rate. Second, even in cases where store prefetching is not successful, a large part of the store latency is from L2 hits; a large fraction of this L2 hit latency can be overlapped before the next load comes to the head of the instruction window and stalls for the store.

Water has small critical sections, where data loads and the lock acquire are overlapped in the systems with speculative loads. This is the only application where RCplain performs slightly worse than the optimized implementations of the other models, as RCplain stalls data loads until the acquire completes. Store prefetches in Water experience write-after-read stalls and little clustering; the remaining store latency hinders the performance of the SC implementations in the +SL versions. However, the differences between the consistency models are small, as Water spends the least time in data memory stalls among our applications.

In all of the above applications, the effect of early speculative loads was negligible, as there were very few rollbacks.

6 Evaluation of Speculative Retirement

The previous section showed that with current instruction window sizes, the store-to-load ordering constraint of SC can significantly affect performance. This section evaluates the benefits achieved by the speculative retirement technique of Section 3, which potentially alleviates the impact of this store-to-load ordering constraint.

6.1 Performance Benefits of Speculative Retirement

Figure 4 summarizes the benefits of speculative retirement. The graph for an application shows the performance of the best implementations of SC (SC+SL+SB), PC (PC+SL), and RC for that application (from Section 5). In RC, the best implementation is RCplain for all applications except Water, where it is RC+SL. Henceforth, the best implementation of RC for an application is referred to as RCbest. Figure 4 additionally includes three implementations of SC with speculative retirement (labeled +SR), and, for comparison, three versions of SC+SL+SB with larger instruction windows (labeled +IW). In the implementations of speculative retirement, the number below each column represents the size of the history buffer used in that implementation. Similarly, each of the three +IW implementations is labeled with a number indicating the increase in the instruction window size for that version. We study the performance benefits of increases in the history buffer size and equivalent increases in the instruction window size up to a doubling of the instruction window size of the base system (64 elements). The memory queue in each of the +SR and +IW implementations is increased by an amount equal to the history buffer size or the increase in instruction window size over the base case, respectively.

Focusing on the two applications where the difference between SC and PC was significant in Section 5 (FFT and Radix), we find that speculative retirement is able to yield significant benefits in both cases. Specifically, speculative retirement reduces execution time up to 11% for FFT and up to 14% for Radix. With the addition of speculative retirement to SC, the execution time reduction provided by PC relative to SC drops from 15% to 5% in FFT and from 24% to 11% in Radix.

For the other applications, the performance potential of speculative retirement is limited, as PC provides little reduction in execution time relative to SC even without speculative retirement. In these applications, speculative retirement provides some gains; however, these gains depend on the history buffer size and application characteristics. All of these applications see nearly as much benefit from a 16-element history buffer as from a 64-element history buffer. In Mp3d, the implementation with a 16-element history buffer actually outperforms the more aggressive speculative retirement implementations by stopping instruction processing before too many early prefetches or early speculative loads are issued.

6.2 Comparison with Increase of Instruction Window Size

A potential alternative to speculative retirement to improve the effectiveness of SC is to simply increase the size of the processor instruction window. In this way, store prefetches can issue earlier and are also more likely to have other operations available to them for clustering. The results of Figure 4 show that, for the history buffer and instruction window sizes we consider, similar performance can be achieved through either of these types of hardware support for most of our applications; however, some cases do see differences between these two techniques.

Increasing the instruction window size outperforms a similar sized history buffer implementation of speculative retirement in some cases (especially LU and Water) because the former provides increased overlap for both loads and stores, while the latter only targets limitations due to store latency.

On the other hand, a history buffer implementation of speculative retirement can outperform an equivalent increase in the instruction window in three ways. First, the history buffer used in speculative retirement does not need to include stores and branches; this allows more instructions to be brought into the instruction window and may allow better overlap in certain types of code, such as the permutation phase of Radix. Second, in cases where early prefetches hurt performance, speculative retirement is more likely to restrain prefetches from issuing too early, as it will not bring in new memory instructions when a load miss is pending at the head of the instruction window. This benefit is seen in LU when comparing 16 and 32-element history buffer sizes to similar instruction window increases. Finally, recoveries in speculative retirement can be enacted as soon as they are detected. In contrast, the common implementation of recoveries in speculative load execution marks an exception on the violating load and waits for the marked load to reach the head of the instruction window before enacting the rollback [26]. Although the penalties associated with such waiting are not inherently high in our base processor, the impact of such waiting will increase with larger instruction windows.

As speculative retirement and increased instruction windows achieve similar performance, depending largely on application characteristics, the choice of one technique over the other depends largely on implementation issues. We discuss two important implementation issues: storage and complexity. Both topics are discussed in the context of a system similar to the MIPS R10000 or DEC Alpha 21264, which use a unified physical register file for architectural state and renaming registers [10, 26].

The storage requirements corresponding to each element of the instruction window and a history buffer are nearly identical. In both cases, each element (whether of the instruction window or the history buffer) must include a logical-to-physical register mapping for the destination register. In the case of the instruction window, new registers must be added to the physical register file in order to enable the use of the new instruction window entries. Similarly,

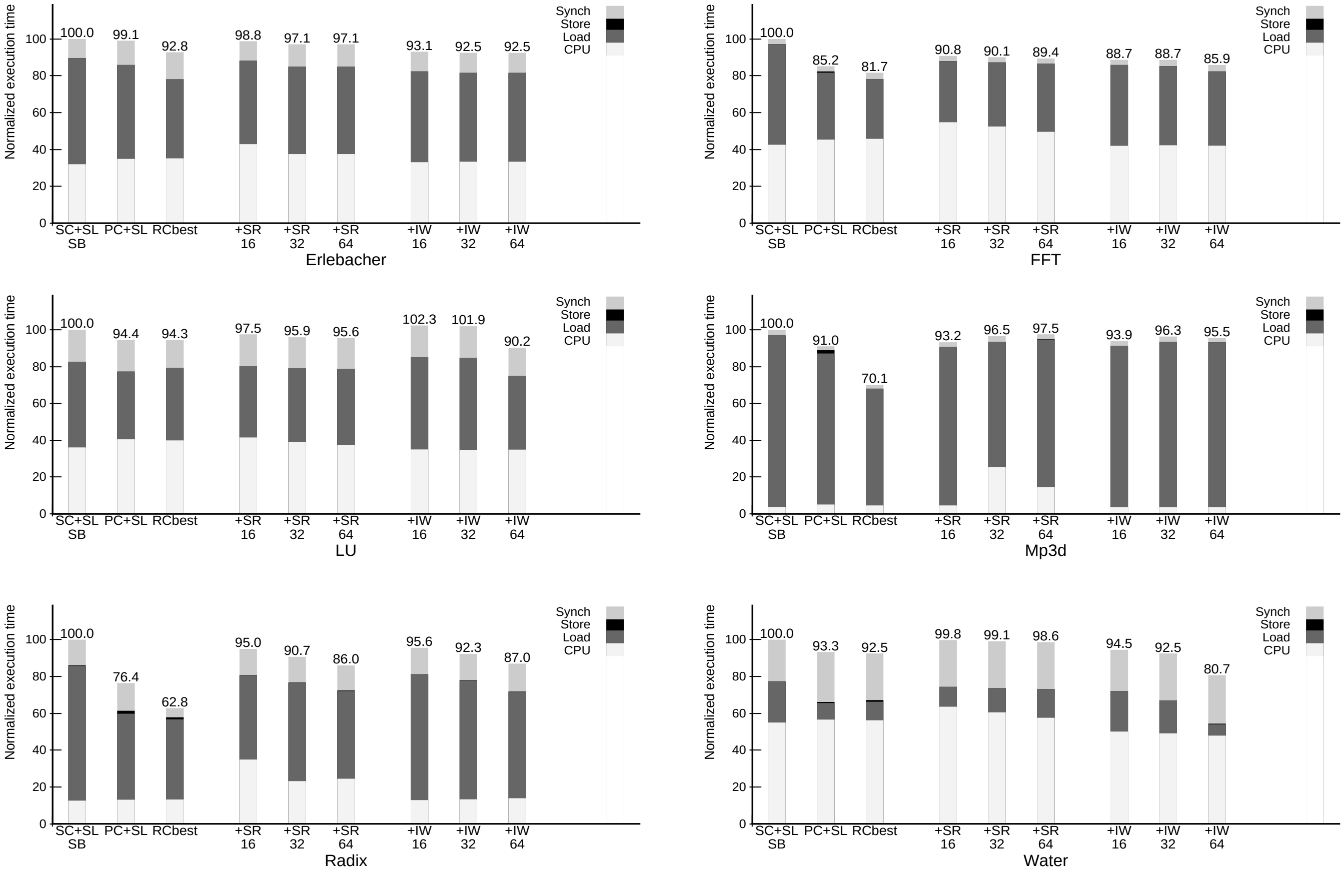

Figure 4: Effectiveness of speculative retirement.

each element of the history buffer must have a data field equivalent to the register size in order to hold the previous register value. Additionally, instruction window entries would need flags to mark whether or not the instruction has completed; however, this is a minor investment.

The complexity of the two systems, however, is different. Specifically, the history buffer implementation needs ports into the register file in order to retrieve the previous values of registers, while a larger instruction window needs more registers in the physical register file. The history buffer can share already-existing ports with other functional units, as the history buffer is only used during periods of speculative retirement and the processor does not always retire instructions at peak rate. Such port-sharing should be possible without affecting the critical path; some processor designs already have register ports shared among functional units [2]. On the other hand, with an increased instruction window, new registers must be added to the physical register file, potentially affecting register-file access time [3]. However, it may be possible to move this increase in register file access time off the critical path by pipelining register file access [24] or clustering the register file [17]. Nevertheless, such changes in register file access time will affect performance at all times, including the uniprocessor case, whereas any degradations caused by port-sharing in speculative retirement will only affect performance during periods when the history buffer is active.

7 Impact of Larger Instruction Window Sizes on Consistency Implementations

This section examines the interaction between various optimizations for consistency models with larger instruction window sizes that may be supported by future processors. Section 7.1 examines the previous best implementations of consistency models with larger window sizes. Section 7.2 examines the use of speculative retirement with larger instruction window sizes.

7.1 Earlier Optimizations with Large Instruction Windows

Figure 5 summarizes the results for this section. The graph for each application shows three curves representing the best previous implementation of SC, PC, and RC (not including speculative retirement) for each application from Section 5. Each curve plots the execution time as a function of the instruction window size. All execution times are shown normalized to the performance of SC with a 64-element instruction window for the application. As in Section 6, all the systems in this section have as many entries in the memory queue as the number of elements in the instruction window.

Our results show that the performance benefits of increasing the instruction window size vary across the different consistency models. Overall, comparing the different consistency implementations, we find that the performance gap between SC and the more relaxed models drops at first, but then changes only marginally or increases for most applications. Thus, increasing instruction-window size alone is not sufficient to equalize the performance of consistency

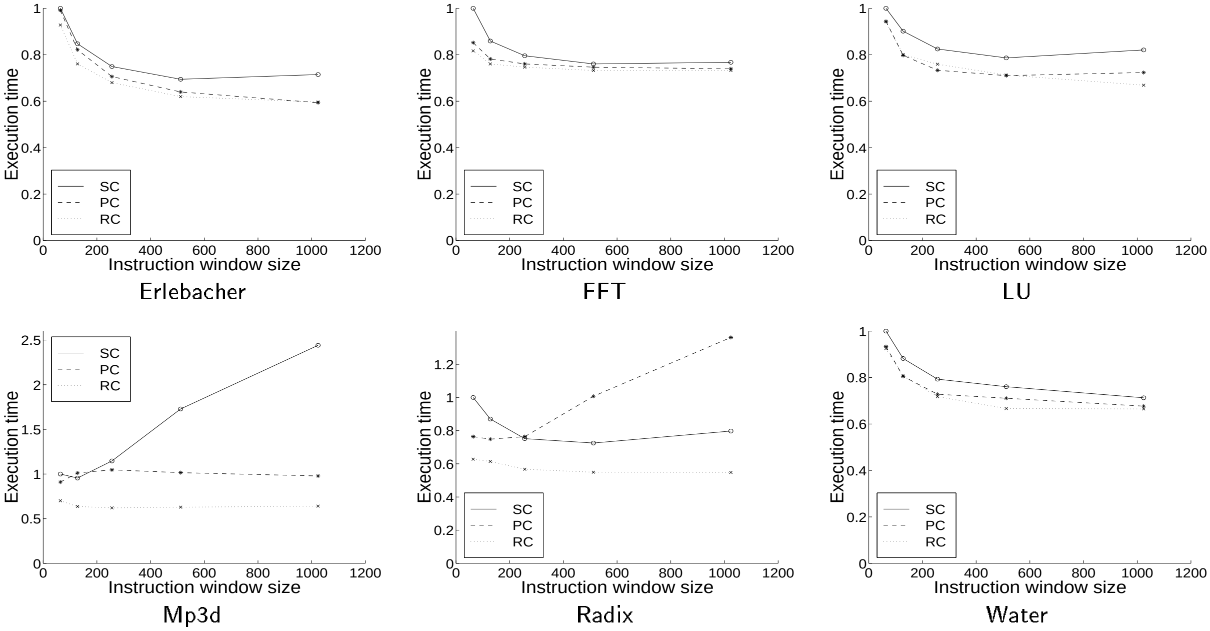

Figure 5: Effect of varying the instruction window size.

models. To determine the reason for this effect, we discuss the behavior of each model with larger instruction window sizes below.

With SC, increasing the size of the instruction window initially leads to significant reductions in execution time in all applications except Mp3d. However, in most of our applications, after a particular value of the instruction window size, the negative impact of early store prefetches and speculative loads (see Section 5) becomes more pronounced, actually leading to a degradation in execution time relative to smaller window sizes.

In the PC and RC systems, increasing the instruction window size initially leads to a reduction in execution time for most applications. However, unlike the SC case, the negative effects of larger instruction windows are not seen in most applications; instead, execution time mostly stabilizes after a particular instruction window size. PC and RC systems are not negatively impacted to the same extent as SC because these implementations either do not have the negative effects, or are less likely to experience negative effects, or are more resilient to these negative effects when they do occur. For example, `RCplain` does not issue speculative loads or store prefetches, therefore avoiding early prefetching and rollbacks; `PC+SL` and `RC+SL` are less likely to rollback on speculative loads, as the ordering constraints in these systems are less strict; and `RC+SL` can pipeline demand store misses for accesses that experience early prefetches.

Mp3d and Radix do not follow the general trend for PC. In Mp3d, the initial doubling of instruction window size leads to a performance loss, as the larger instruction window increases true and false sharing, consequently leading to more rollbacks and early prefetches; these early prefetches can further lead to more early prefetches, as discussed in Section 5. As the instruction window size increases, early prefetches increase further; however, the number of rollbacks actually decreases as the memory queue fills up with stores that were prefetched early, preventing loads from issuing too early. Consequently, performance then improves slightly with larger instruction window sizes. In Radix, the greater potential for prefetching provided by the larger instruction window and the PC model, combined with false-sharing in the application, leads to long sequences of early prefetches as discussed in Section 5.

7.2 Speculative Retirement with Large Instruction Windows

Figure 5 shows that for each application, after a certain instruction window size, the RC implementation sees either no further improvements or only marginal further reduction in execution time. We refer to this instruction window size as the *RC-knee*. This point is the maximum desirable instruction window size for RC designs, as further additions to the instruction window increase complexity without significantly reducing execution time. The RC-knee also represents the instruction window size that gives the maximum load overlap (either as a result of application limitations or resource contention); this observation holds regardless of consistency model, as successful speculative loads in SC and PC are handled like loads in RC.

Although load overlap is exhausted, some applications continue to see a significant performance difference between SC and RC at the RC-knee because of limitations of stores for SC. Hardware techniques to better address store latency could provide SC with additional improvements. Again, we have the alternative of using speculative retirement or increasing instruction window size. At the window sizes of Section 6, increasing instruction window size gave better performance in some cases because of increased load overlap. However, the processor has already exhausted load overlap at the RC knee; therefore, speculative retirement should be a better choice at this point, since it gives the additional benefits of not holding elements for stores and branches, potentially fewer early prefetches, and faster rollbacks in case of violations (as described in Section 6.2). The impact of faster rollbacks is particularly expected to become more important as instruction window size increases. Additionally, the increases in register-file size mandated by further increases in the instruction window are likely to elongate

	(Approximate RC-knee)			(knee+32)		(2 x knee)	
Application	RCbest	PC+SL	SC+SL+SB	+SR	+IW	+SR	+IW
Erlebacher	2.36	2.45	2.6	2.56	2.61	2.45	2.41
FFT	1.08	1.11	1.22	1.19	1.17	1.15	1.13
LU	5.85	5.83	6.59	6.19	6.26	6.03	6.03
Mp3d	3.26	5.18	4.89	4.91	5.05	5.4	5.87
Radix	3.84	5.17	5.09	5.04	5.05	4.87	4.92
Water	4.2	4.26	4.64	4.62	4.68	4.43	4.45

Figure 6: Performance impact of speculative retirement.

already complex register-file-access paths, as discussed in Section 6.2.

Figure 6 lists the execution times (in millions of cycles) seen by different consistency implementations. The first three columns list the performance of the best implementations of RC, PC, and SC at the approximate value of the RC-knee[5]. The next two columns show the execution time achieved by SC implementations with a 32-element history buffer and a 32-element increase in the instruction window, respectively (for a base instruction window of RC-knee elements). The last two columns show an implementation with a history-buffer and instruction window size equal to the RC-knee, and an instruction window size equal to twice the RC-knee, respectively. As in Section 6, memory queue sizes are also increased according to the history buffer size or instruction window increase. We note that, for all cases, the performance impact of speculative retirement is very close to or better than the performance of an equivalently increased instruction window size implementation for the reasons described above. Thus, at larger instruction window sizes, speculative retirement appears preferable for providing similar performance with potentially less impact on complexity.

8 Discussion

Section 6 discussed the effectiveness of a hardware implementation of speculative retirement, showing that this technique can improve the performance of SC and reduce the performance gap between SC and relaxed consistency models. Sections 6 and 7 compare the benefits of speculative retirement with another hardware technique, larger instruction windows. These sections have focussed on hardware techniques in order to investigate the performance benefits achievable with a previously-existing SC code base. This study has found that all of these hardware optimizations are limited to varying extents by early prefetches and rollbacks, particularly on applications with significant amounts of true and false sharing. Using speculative retirement instead of larger instruction windows can reduce the amount of early prefetches in some cases and can reduce the impact of some rollbacks; however, these problems remain important for performance. One possible optimization to reduce the number of early prefetches and rollbacks is to dynamically adjust the size of the history buffer according to the current program behavior. Specifically, the processor could limit the number of instructions allowed into the history buffer if too many prefetches were invalidated or replaced before use. A similar optimization may not be practical in the case of larger instruction windows, as the physical register file must be designed according to the largest possible instruction window size; consequently, the processor would still have to suffer from the elongated register file access time even during periods when it wanted to limit the number of instructions in the system.

Additionally, attempts to narrow the performance gap between consistency models can focus on software techniques. Recent work has considered the interaction of optimized consistency models with software-controlled nonbinding prefetching on ILP-based multiprocessors and has found that such software-controlled prefetching can reduce the gap between SC and RC in some cases, but that a significant performance gap remains in other cases (again due to write latency) [18]. Previous work has shown that compiler optimizations that schedule load misses closer together (within the same instruction window) can improve performance in RC systems by better exploiting clustering [16]. Similarly, compiler techniques to schedule store misses closer to load misses may better hide the impact of store miss time on SC. However, any technique that requires modification or recompilation of source code will not provide benefits to the previously-existing code base.

9 Conclusion

This paper makes three contributions. First, we analyze the performance of processor consistency (PC) (and, equivalently, total-store ordering [TSO]) with implementations suitable for shared-memory multiprocessors with state-of-the-art ILP processors. Second, we propose and evaluate *speculative retirement*, a hardware optimization for sequential consistency (SC) aimed at reducing the impact of the store-to-load ordering constraint imposed by SC. Speculative retirement builds upon the currently implemented optimization of speculative loads, and can be implemented with a structure similar to a history buffer. Third, we investigate the interaction of consistency optimizations with larger instruction windows that may represent future processor trends.

In the first part of our study, we find that the optimized implementation of PC yields more than a 15% reduction in execution time relative to the best implementation of SC for two of our six applications (FFT and Radix). PC realizes these benefits by relaxing the store-to-load ordering constraint of SC and thus seeing fewer performance limitations from store latency. However, release consistency (RC) provides more than 15% additional reduction in execution time relative to the optimized implementation of PC for two applications (Radix and Mp3d). These differences stem primarily from insufficient memory queue sizes and the negative effects of early store prefetches on the performance of PC.

Our results evaluating the performance of speculative retirement show that speculative retirement using a 64-element history buffer reduces SC execution time notably for the two applications which saw a significant difference between SC and PC. Specifically, the execution time reduc-

[5] We only have approximate values of the RC-knee because our experiments only include instruction window sizes in successive powers of 2, rather than every window size in the range considered.

tion provided by PC relative to SC drops from 15% to 5% in FFT and from 24% to 11% in Radix. Comparing the history buffer implementation of speculative retirement to equivalent increases in instruction window size, we find that both forms of hardware support can improve performance and outperform the other in certain cases. However, speculative retirement appears likely to have less impact on cycle time or the number of pipeline stages, as increasing the instruction window size requires increases in the register file, consequently slowing register file access.

When evaluating the interaction of previous consistency implementations with the larger instruction window sizes that may appear in future processors, we find that increases in instruction window size initially lead to significant reductions in execution time for all models with most applications. However, after certain instruction window sizes, SC performance tends to degrade as a result of the negative effects of early prefetching and speculative loads, while PC and RC performance tends to stabilize. At the instruction window size where RC performance stabilizes, the potential for load overlap has been virtually exhausted. Beyond this instruction window size, additional benefits in SC performance can come primarily from store overlap. Thus, beyond this point, speculative retirement appears to be a more attractive choice than further increases in instruction window size for more aggressive SC designs, as speculative retirement can give comparable or better benefits than an equivalent increase in the instruction window size, with possibly less complexity.

10 Acknowledgments

We would like to thank Norm Jouppi, Jim Smith, David Wood, and the anonymous reviewers for valuable feedback on earlier drafts of this paper.

References

[1] V. S. Adve et al. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proc. of Supercomputing '95*, 1995.

[2] T. A. Diep et al. Performance Evaluation of the PowerPC 620 Microarchitecture. In *Proc. of the 22nd ISCA*, 1995.

[3] K. I. Farkas et al. Register File Design Considerations in Dynamically Scheduled Processors. In *Proc. of the 2nd HPCA*, 1996.

[4] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th ISCA*, 1990.

[5] K. Gharachorloo et al. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. of the 4th ASPLOS*, 1991.

[6] K. Gharachorloo et al. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of ICPP*, 1991.

[7] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.

[8] D. Hunt. Advanced Features of the 64-bit PA-8000. CompCon 1995, Hewlett Packard Company.

[9] Intel Corporation. *Pentium (r) Pro Family Developer's Manual.*

[10] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. 9th Annual Microprocessor Forum, 1996.

[11] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. of the 8th ISCA*, 1981.

[12] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[13] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual, Version 1.1*, 1996.

[14] V. S. Pai et al. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. of the 7th ASPLOS*, 1996.

[15] V. S. Pai et al. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proc. of the 3rd Workshop on Computer Architecture Education*, 1997.

[16] V. S. Pai et al. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. of the 3rd HPCA*, 1997.

[17] S. Palacharla et al. Complexity-Effective Superscalar Processors. In *Proc. of the 24th ISCA*, 1997.

[18] P. Ranganathan et al. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proc. of the 24th ISCA*, 1997.

[19] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[20] J. P. Singh et al. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[21] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proc. of the 12th ISCA*, 1985.

[22] Sparc International. *The SPARC Architecture Manual, Version 9*, 1993.

[23] Sun Microsystems. *The UltraSPARC Processor – Technology White Paper*, 1995.

[24] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd ISCA*, 1996.

[25] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, 1995.

[26] K. C. Yeager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, 1996.

[27] R. N. Zucker and J.-L. Baer. A Performance Study of Memory Consistency Models. In *Proc. of the 19th ISCA*, pages 2–12, 1992.