In Proceedings of ASPLOS-VII (October, 1996)

An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors

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

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892
{vijaypai|parthas|sarita|harton}@rice.edu

Abstract

Relaxed consistency models have been shown to significantly outperform sequential consistency for single-issue, statically scheduled processors with blocking reads. However, current microprocessors aggressively exploit instruction-level parallelism (ILP) using methods such as multiple issue, dynamic scheduling, and non-blocking reads. Researchers have conjectured that two techniques, hardware-controlled non-binding prefetching and speculative loads, have the potential to equalize the hardware performance of memory consistency models on such processors.

This paper performs the first detailed quantitative comparison of several implementations of sequential consistency and release consistency optimized for aggressive ILP processors. Our results indicate that hardware prefetching and speculative loads dramatically improve the performance of sequential consistency. However, the gap between sequential consistency and release consistency depends on the cache write policy and the complexity of the cache-coherence protocol implementation. In most cases, release consistency significantly outperforms sequential consistency, but for two applications, the use of a write-back primary cache and a more complex cache-coherence protocol nearly equalizes the performance of the two models.

We also observe that the existing techniques, which require on-chip hardware modifications, enhance the performance of release consistency only to a small extent. We propose two new software techniques – fuzzy acquires *and* selective acquires *– to achieve more overlap than allowed by the previous implementations of release consistency. To enhance methods for overlapping acquires, we also propose a technique to eliminate control dependences caused by an acquire loop, using a small amount of off-chip hardware called the* synchronization buffer.

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

1 Introduction

Long memory latencies remain a significant impediment to achieving the full performance potential of shared-memory systems. The memory consistency model of a shared-memory system determines the extent to which memory operations may be overlapped or reordered for better performance. Previous studies have shown that the release consistency model significantly outperforms the conceptually simpler model of sequential consistency [9, 13, 11, 27], albeit with increased programming complexity [2, 8]. However, the first two of these studies [9, 13] assumed single issue statically scheduled processors with blocking reads. The third study [11] assumed an aggressive processor, but examined only straightforward implementations of the consistency models, and used trace-driven simulations requiring significant approximations. The fourth study [27] examined one optimization with non-blocking reads, but assumed single issue statically scheduled processors.

Current and next generation high-performance microprocessors exploit increased levels of instruction-level parallelism (ILP), using aggressive techniques such as multiple issue, dynamic scheduling, speculative execution, and non-blocking reads. For such processors, Gharachorloo et al. proposed two techniques – hardware prefetching and speculative loads – to enhance the performance of both sequential consistency and release consistency [10]. They conjectured that these techniques can equalize the hardware performance of the two models. These techniques have recently begun to appear in commercial microprocessors (e.g. HP PA-8000 [15], Intel Pentium Pro [16], and MIPS R10000 [19]), and re-open the issue of whether the hardware performance advantages of relaxed consistency models justify the tradeoff in programming complexity. Furthermore, for earlier processors with blocking reads, the decision to support a relaxed consistency model did not necessarily have to be made at processor design time, since writes can be made non-blocking by simply providing an early acknowledgment from an external memory controller. Non-blocking reads, however, bring in a value needed by other instructions and must be integrated into the processor design, Thus, the consistency model now has a larger impact on processor design, further increasing the importance of understanding the benefits of relaxed consistency on current processors.

This paper performs the first detailed quantitative comparison of several implementations of sequential consistency (**SC**) and release consistency (**RC**) with processors supporting dynamic scheduling and non-blocking reads, and proposes new techniques to overlap acquire latencies using such processors.

We use instruction-level simulation on six applications to compare the hardware performance of SC and RC with ILP processors, in both simple implementations as well as with the enhancements provided in current ILP processors: hardware prefetching and speculative loads. Our simulator accurately models the internals of an aggressive ILP processor similar to the MIPS R10000 along with an aggressive memory system. The key results of this study are as follows.

- For SC, the two techniques dramatically improve performance, providing a speedup over 2 in several cases.
- For RC, overall, the two techniques are not very effective, because RC already manages to hide all store latency and a large part of load latency.
- The difference between RC and SC performance depends primarily on whether the first level cache is write-through or write-back and on the complexity of the cache-coherence protocol. With our base protocol, which is fairly aggressive and represents many current implementations, we find that RC consistently outperforms SC. With write-through primary caches, RC achieves a factor of 2 speedup over the best SC for two of the six applications, and over 1.5 speedup for two others. With write-back primary caches, the speedups are less dramatic, but still fairly large (1.5 or more for three applications). With a more aggressive, but more complex, cache-coherence protocol, optimized SC achieves performance comparable to RC for two applications, but a significant gap remains for others. Our results show that the performance of SC is highly sensitive to cache write policy and the aggressiveness of the cache-coherence protocol, while the performance of RC is generally stable across all implementations.

RC sees little benefit from the two techniques because these optimizations conservatively assume that all operations in program order after an acquire depend on that acquire. In many cases, however, an acquire is followed by operations that are independent of it, but may be interspersed with other dependent operations. We propose two software techniques, *fuzzy acquires* and *selective acquires*, which seek to non-speculatively overlap independent operations with the latency of an acquire. To enhance methods for overlapping acquires, we also propose a method to eliminate control dependences caused by an acquire loop using a small amount of off-chip hardware. We evaluate the new techniques for the two applications for which the previous optimizations provided a performance benefit. We find that the new techniques provide slightly better performance improvements than the previous techniques, without the on-chip hardware support required for speculative loads. However, our techniques require additional analysis of the program to identify operations which actually depend upon the acquires we seek to overlap.

The remainder of the paper is organized as follows. Section 2 discusses current implementations of SC and RC. Section 3 presents our simulated architectures, methodology, and applications. Section 4 describes the results of our comparison of current implementations. Section 5 motivates, presents, and evaluates our new techniques for overlapping acquires. Section 6 discusses related work. Section 7 concludes the paper.

2 Current Implementations of Consistency Models

The most intuitive memory consistency model, sequential consistency (**SC**) [18], guarantees that memory operations appear to execute in program order. Release consistency (**RC**) [8] distinguishes between data operations and acquire and release synchronization operations. The primary relaxation that RC provides over SC is that data operations of a processor can be reordered with respect to each other. The primary constraint imposed by RC is that data operations must appear to await the completion of previous (by program order) acquire operations. Simple implementations of the two models achieve the above constraints by prohibiting a memory operation from entering the memory system until all previous operations for which it must appear to wait have completed.

The two optimizations for consistency models we evaluate are hardware prefetching and speculative loads, as proposed by Gharachorloo et al. [10]. These techniques take effect whenever the constraints of a consistency model could restrict the issue of a memory operation. Both techniques exploit the instruction lookahead window in an aggressive ILP processor. Similar techniques are used in the HP PA-8000 [15], the Intel Pentium Pro [16], and the MIPS R10000 [19].

The prefetch technique issues a hardware-controlled non-binding prefetch [13] for a decoded memory operation in the instruction window as soon as its address is available, and if the operation cannot be issued otherwise. Prefetch allows an SC system to obtain remote data for reads while a regular memory operation is pending; prefetch allows an RC system to prefetch reads past acquire operations. Since processors typically implement precise exceptions, stores cannot issue to the memory system until reaching the head of the instruction window. The prefetching technique allows both consistency models to issue exclusive prefetches for such stores.

Speculative load execution goes one step beyond prefetching by actually using the value of a load as soon as that value becomes available (typically through the prefetches discussed above). The technique preserves correctness by requiring that any data that is speculatively loaded remain visible to the coherence mechanism. This is achieved by using additional on-chip hardware in the form of a *speculative load buffer*. The speculative load buffer must communicate with the cache, tracking any invalidation, update, or cache replacement operations on cache lines that have had loads issued speculatively to them. If such a message reaches the speculative load buffer, the unit must then interface with the processor's window of active instructions and not only reissue the speculated load, but also roll back all subsequent processor operations. The MIPS R10000 supports this rollback mechanism by stopping an incorrectly speculated load when it seeks to retire from the processor's instruction window; at that time, the hardware re-issues the load and also flushes the rest of the instruction window [19].

3 Evaluation Methodology

The following sub-sections respectively describe the simulated architectures, the simulation methodology and environment, the performance metrics, and the applications used in this work.

3.1 Simulated Architectures

Memory System and Network. We simulate a hardware cache-coherent multiprocessor with a full-mapped, invalidation-based, three-state directory coherence protocol, where processing nodes are connected with a two-dimensional mesh network. The number of processing nodes we simulate is dependent on the application, as explained in Section 3.4. 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 is always dual-ported, but can be either write-through with no-write-allocate or write-back with write-allocate. We evaluate both first-level cache configurations since write-hits in SC expose the second-level access latency in a write-through configuration, but only the first-level access in a write-back configuration. In the absence of resource constraints, RC will hide the latency of writes in either configuration. Thus, we expect the comparative results of SC and RC to differ with these different first-level cache configurations. If we have a write-through first-level cache, we also include a coalescing line write-buffer between the two levels of cache. Regardless of first-level cache configuration, the second-level cache is always a pipelined write-back, write-allocate cache. Both levels are non-blocking with 8 Miss Status Holding Registers (MSHRs) [17]. The MSHRs store information about misses and coalesce multiple requests to the same cache line.

When there is a write request to a line which has a load pending, the MSHR buffers the write and issues an ownership request only when the read reply returns, as in many current processor implementations. Allowing this ownership request to overlap with a previous read request increases the complexity at the directory controller and at the MSHRs, since they would need to handle potential reordering of requests in the network. Although our system is representative of current systems, this decision can potentially affect the performance of store prefetching. Section 4.4 also assesses the impact of the more aggressive but complex protocol where the read and ownership requests are overlapped.

Figure 1 gives our default primary memory system parameters. We have chosen smaller cache sizes than commercial systems, commensurate with our application input sizes (Section 3.4) and following the working-set evaluations of Woo et al. [26]. Our secondary caches are chosen such that secondary working sets of most of our applications do not fit in cache; we choose primary cache sizes such that any applications with fixed-size primary working sets fit in cache. For representative applications, we also investigate performance with cache sizes similar to those found in commercial systems, and we find little change in overall results, as discussed in Section 4.5.

The processor, network, and base memory system parameters are fairly aggressive, and meant to represent future implementations. The parameters were chosen by extrapolating from numbers given by various system vendors.

Base Processor. To exploit instruction-level parallelism, our base processor model employs widely used techniques like multiple instruction issue, dynamic (out-of-order) scheduling, register renaming, speculative execution, and non-blocking reads. The processor exploits ILP by examining a large window of instructions at a time, and executes the instructions that are not dependent on the completion of any previous incomplete instructions. This allows instructions to issue and complete out of program order. Except for stores in the RC models, an instruction retires (graduates [19]) when it is complete and when all preceding instructions (by program order) have retired. A store in RC retires when its address and value are resolved, and when all previous instructions have retired. To guarantee precise interrupts, stores are not issued into the memory system until they reach the head of the instruction window. We use the SPARC V9 `MEMBAR` [25] instructions (memory fences) to enforce ordering of memory operations as required by the consistency model.

The processor micro-architecture is most closely based on the MIPS R10000 design [19]. 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.

Variations on the Base Processor. The base processor model directly supports the simple implementation of release consistency. Variations on our processor and memory system include a sequentially consistent processor model, support for hardware-controlled non-binding prefetching, and support for speculative load execution.

For a simple implementation of SC, we modify the aggressive base memory system to issue a memory operation only when the previous memory operations of that processor have completed. This method maintains ordering of all memory operations as required by SC. Furthermore, a store in SC does not retire from the instruction window till it is globally performed.

To implement hardware prefetching, we issue prefetch requests to the cache as described in Section 2. We prefetch requests to the level of cache appropriate for the corresponding demand fetch; thus write prefetches with the write-back write-allocate primary cache and all read prefetches go to the primary cache. Write prefetches with the write-through non-write-allocate primary cache only fetch into the secondary cache; bringing these to the primary cache would defeat the purpose of a no-write-allocate cache.

We implement the speculative load buffer near the processor and use the mechanism employed by the MIPS R10000 to rollback execution as described in Section 2. In SC systems, we use the speculative load buffer whenever we want to issue a load out of order; in RC systems, this buffer is only used past the memory fences corresponding to acquires. We do not impose a constraint on the size of the speculative load buffer, limiting it only by the number of loads in the memory unit. We force a rollback when the primary cache gets a coherence request from an external source or an invalidation request from the secondary cache for inclusion; there is no need to rollback on primary cache replacements since those lines will still remain visible to external coherence.

3.2 Simulation Methodology and Environment

We have developed the Rice Simulator for ILP-based Multiprocessors (RSIM) to model the architecture described in Section 3.1. In contrast to many current execution-driven simulators, RSIM is an instruction-driven simulator that models both the processor pipelines and the memory subsystem in great detail, including contention at various resources. The code for the memory system and network is heavily drawn from RPPT (the Rice Parallel Processing Testbed) [6, 22]. RSIM is driven by application executables rather than traces so that interactions between the processors during the simulation can affect the course of the simulation. The detail in our simulator thus leads to increased simulation times compared to those seen in either

Memory Hierarchy Parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped,4K
L1 cache ports	2
L1 MSHRs	8
L2 cache (off-chip)	4-way associative 64K
L2 MSHRs	8
Write buffer (coalescing)	8 line entries
Memory interleaving	4-way
Memory Latency Components	
L1 cache access	1 cycle
L2 cache access	8 cycles
Memory bus arbitration delay	3 cycles
Directory and memory access	18 cycles
Memory transfer bandwidth	16 bytes/cycle
Network Parameters	
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles
Processor Parameters	
Processor Speed	300MHz
Peak issue, retire rate	4 instructions/cycle
Instruction window size	64
Memory queue size	32
Functional units	2 integer arithmetic
	2 floating point
	2 address generation
Renaming registers	128
Branch speculation depth	8
Maximum rollback penalty	8 cycles

Figure 1: Default simulation parameters.

execution-driven or trace-driven simulations; however, the detail is necessary for the problems addressed by this paper.

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. Both of these approximations have been widely used in shared-memory multiprocessor performance studies.

3.3 Performance Metrics

We divide execution times into its various components, namely CPU time and stall time due to Reads, Writes, Locks, Flags, and Barriers. However, with ILP processors, each instruction can potentially overlap its execution with both previous and following instructions. Hence, it is difficult to assign stall time to specific instructions. We count a cycle as part of busy time if we 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 in that cycle. This convention is also followed by Rosenblum et al. [23] Thus, effectively, our statistics for individual stall components represent the cumulative time instructions in each class stall at the top of the instruction window before retiring. If an instruction retires without having spent any time at the top of the instruction window, it is considered to have fully overlapped with previous instructions. We use these detailed statistics only to gain insight into the nature of the various applications and to identify the portions of the computation overlapped by various optimizations. For purposes of comparing various implementations, however, we use the total execution time as the primary performance metric.

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

Figure 2: Application parameters

3.4 Applications

We use six applications in this study – Radix, FFT and LU from the SPLASH-2 suite [26]; Water and MP3D from the SPLASH suite [24]; and Erlebacher, obtained from the Rice parallel Fortran compiler group. Erlebacher solves partial differential equations by performing 3-D vectorized tridiagonal solves using Alternating-Direction-Implicit (ADI) integration. The key data structures are 3-dimensional arrays which are distributed by assigning a consecutive block of X-Y planes to each processor. One phase dominates the execution time, and contains all the communication and synchronization of this application. The computation in this phase consists of a forward-substitution pipeline and a backward-substitution pipeline, with flags to synchronize processors sharing a boundary plane. The block size determines the size of each pipeline stage.

The SPLASH and SPLASH-2 application suites have been widely used in architecture research. LU was slightly modified to use flags instead of barriers for synchronization, since flag synchronization improved performance. We also inlined a daxpy function and interchanged a key loop nest so that read misses occurred closer together for better overlap of memory operations. Water was modified to move certain private calculations outside of key critical sections in the function `UPDATE_FORCES`. Radix employs a tree-based algorithm for its prefix sum calculation.

Figure 2 gives the input sizes and number of processors used for the various applications. Because our simulation times are necessarily much higher than seen in studies using execution-driven or trace-driven simulation, we were restricted to using problem sizes one size smaller than generally recommended for two applications (LU and Water). However, we decrease the number of processors appropriately to ensure reasonable speedups (the recommended problem sizes are for configurations of up to 64 processors). We also use a smaller configuration for MP3D and Radix because of low speedup; these applications are included to represent applications that stress the memory system.

4 Evaluation of Current Consistency Implementations

Sections 4.1 and 4.2 investigate the effect of the prefetching and speculative load optimizations on SC and RC respectively, and Section 4.3 compares SC with RC. Section 4.4 examines the impact of a more aggressive coherence protocol, while Section 4.5 evaluates the impact of larger cache sizes on our results.

Figure 3 summarizes our results. As motivated by Section 3.1, we investigate systems with write-through and write-back first-level caches (shown on the left and right side respectively of the figure for each application). For each of SC and RC, and for each first-level cache configuration, we examine three systems: `Simp` refers to the simple implementation, `+PF` adds load and store prefetching as discussed in Section 2, and `+SL` further adds speculative loads to the `+PF` configuration. For each implementation, the figure shows the total execution time, normalized to the time for a simple implementation of SC using the write-through L1 cache. These times are divided into stall time due to reads, writes, various synchronization constructs, and the remaining CPU time, as discussed in Section 3.3. Recall that the value of each component represents time stalled at the top of the instruction window. Thus, the low CPU times in our results do not generally imply poor speedup; rather, these values indicate that a large part of the busy time is completely overlapped with previous longer latency operations.

4.1 Sequential Consistency Implementations

Sections 4.1.1 and 4.1.2 respectively analyze the performance impact of hardware-controlled prefetching and speculative loads in SC systems with write-through L1 caches. Section 4.1.3 summarizes the results with write-through L1 caches. Section 4.1.4 discusses the impact of a write-back L1 cache.

4.1.1 Prefetching with Write-Through L1 Caches

With a write-through cache, hardware-controlled prefetching helps SC performance for all applications, but to a variable extent. Three applications see an improvement in execution time ranging from 14% to 25% (LU, FFT, and Erlebacher), while three applications see less than 10% improvement (Radix, MP3D, and Water). Overall, most of the benefits of prefetching appear from reducing read stall time; prefetching is generally unsuccessful in reducing write stall time.

Several factors limit the benefits of hardware-controlled prefetching in our applications. First, the finite size of the instruction window limits how early a prefetch can be issued. In particular, if there are no other long latency operations before the prefetched instruction in the instruction window, the potential for overlap is limited. Thus, prefetching is most effective when several memory misses occur close together within the instruction window. As a measure of the amount of overlap obtained, our simulator records the fraction of time that a given number of L1 or L2 MSHR entries were occupied by memory accesses. We use this information (not shown here for lack of space) in our analysis below. Second, the address of the instruction to be prefetched may depend on the value of a load instruction that is also blocked from issuing. Since the value cannot be used until the load completes, the later memory operation may not be prefetched early enough. Third, for SC with write-through caches, all writes must propagate to at least the L2 cache before they are considered complete, and before they retire from the instruction window. This minimum write latency cannot be overlapped by prefetching. Finally, as explained in Section 3.1, in our default system, if a store prefetch is issued while a demand or prefetch load to the same cache line is outstanding, the ownership request for the prefetch is blocked until the outstanding read returns. (Sections 4.1.4 and 4.4 respectively describe the impact of eliminating the last two limitations.) We next discuss how the above effects impact the individual applications.

Radix and MP3D show less than 10% improvements with prefetching. The key reason is that the most memory-intensive portions of these codes contain memory operations whose addresses depend on values returned by previous loads. This kind of dependence is exhibited by Radix in its permutation phase and by MP3D in its cell array accesses. Additionally, MP3D has limited write overlap when a write prefetch to a cell following a read to the same cell is blocked at the MSHRs.

Erlebacher shows the most benefits from prefetching. It shows a high occupancy of both L1 and L2 MSHRs, indicating significant overlap. Erlebacher sees benefits from store prefetching because in the main computation (a finely synchronized pipeline), writes occur in a strided manner and miss on the first access to each line in the pipeline stage. As a result, for each pipeline stage, the first few writes miss in the L2 cache and occur close together in the instruction window. This clustering yields effective write prefetching. A large part of the write latency still remains since only writes that miss are overlapped with each other in small clusters, and all writes (including hits and misses) still see the L2 hit latency. Further, writes often follow reads to the same word, resulting in the ownership request of a write prefetch to be serialized behind the corresponding read. Finally, Erlebacher follows the owner-computes rule, and so all write misses are serviced by the memory on the writer's node exhibiting a relatively low latency. In this application, reads also have a strided access pattern, and so exhibit large benefits from prefetching.

FFT behaves like Erlebacher in that it has clustered, strided writes in the important transpose phase, making store prefetching effective. As in Erlebacher, FFT still has a significant write latency because it pays the penalty of an L2 hit for every write.

LU has a high degree of read overlap and benefits primarily from read prefetching. However, the benefits are limited because most of the overlapped misses in LU are L2 hits, and the longer remote misses are not overlapped with one another. A high (98%) secondary cache write hit rate (stemming from the blocked nature of the computation) effectively prevents store prefetching from getting any benefits, since even prefetched writes in this configuration must access the secondary cache.

Water is successful in eliminating most of its read miss latency stalls with prefetching; however, this does not translate to a large improvement in execution time since read misses form a relatively small fraction of the total execution time. The primary computation in Water consists of an update of the force vector of a molecule in a critical section. The lock acquire and a data read from the critical section can be overlapped with each other. Water continues to show a large write stall component because the prefetch ownership request for the force update is blocked behind the outstanding read to the same line. Further, several consecutive loads and stores occur to different elements of the same molecule structure. These accesses are likely to access the same cache line, limiting the potential for overlap within the window. Water has two or more L2 cache MSHRs (which include both read and write misses) occupied for less than 10% of the time.

4.1.2 Speculative Loads with Write-Through L1 Caches

The addition of speculative load execution helps every application significantly, with improvements in execution time (relative to simple SC) ranging from 29% in MP3D to 51% in LU. These improvements are seen in CPU time, read stall time, and store stall time.

Compared to SC with prefetching alone, CPU time decreases significantly for FFT, LU, Erlebacher, and Water. Each of these applications benefit from the ability to consume the values of loads speculatively, as this ability allows computation dependent on those loads to be largely over-

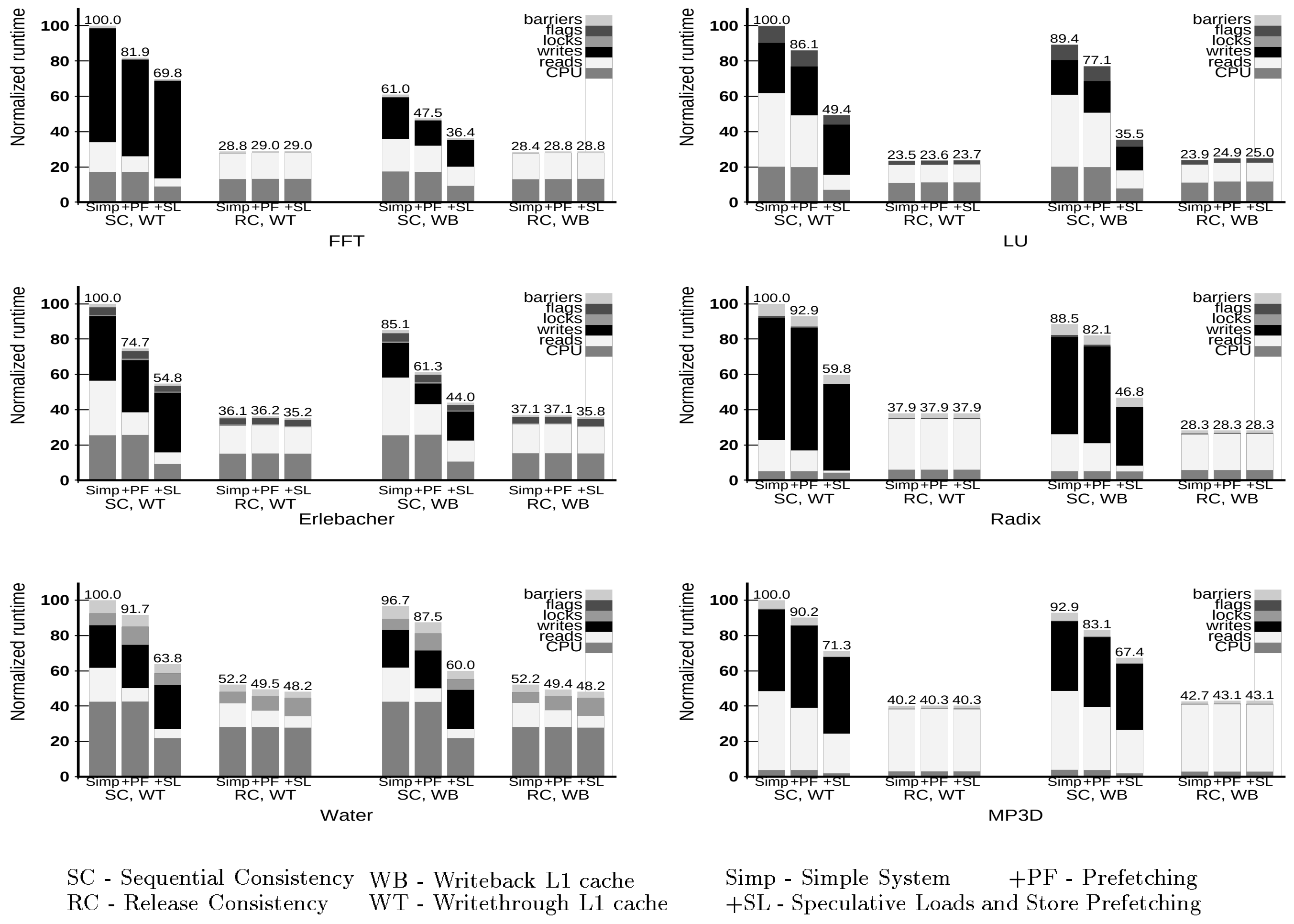

Figure 3: Evaluation of Consistency models

lapped with a long latency memory access stalled at the head of the instruction window.

FFT, LU, Radix, and MP3D see a significant decrease in their memory component. In the case of FFT and LU, this decrease arises from a decrease in conflict misses caused by reordering of accesses; in either simple SC or SC with prefetching, these applications see repeated conflict misses among subsequent demand read accesses, since these must occur in order. With speculative load execution, loads can issue in parallel and out of order; as a result, several loads to the conflicting lines can occur concurrently. Load misses to the same line can coalesce into the same MSHR, while other loads can hit a line despite the fact that its set has a pending MSHR. This eliminates some of the conflict misses seen using a system (such as SC or SC with prefetching alone) in which demand reads can only occur one-at-a-time and in order.

Radix and MP3D decrease in read time since each of these applications has reads with addresses dependent on the values of previous reads; such applications benefit from the ability to consume the value of reads as soon as possible, since this gives them more potential for further overlap.

Among all the applications, only Radix sees benefits in store latency. Radix gets these benefits that it did not get with prefetching alone because it also has stores whose addresses depend on values returned by previous long latency loads. The ability to consume values for loads and use those values in sending out prefetches for the above stores improves the write overlap for Radix.

One potential limitation of speculative execution is that overly optimistic speculation could lead to excessive rollbacks, which may hurt performance. However, we found the number of rollbacks to be small for each of our applications. In the SC case, only LU sees a significant number of rollbacks (6094), and even there, fewer than 0.2% of all loads cause rollbacks; rollback penalties make up less than 0.05% of total execution time. As expected, RC sees fewer rollbacks (almost zero on all applications except LU) than in the SC case, largely because RC issues far fewer loads speculatively than SC does.

4.1.3 Summary for Write-Through L1 Caches

For sequentially-consistent systems with a first-level write-through cache, we find that hardware-controlled prefetching alone improves performance but the improvements are small for some applications; the addition of speculative load execution consistently and significantly increases system performance showing up to a factor of two speedup. Nevertheless, we find that for an architecture with write-through L1 caches, neither technique is sufficient to handle the large store latency component associated with SC. We next determine the possible benefits of using a write-back cache instead.

4.1.4 Impact of Write-Back L1 Caches

Figure 3 shows that the primary change from a write-through to a write-back L1 cache is in the decreased contribution of write latency to execution time (Figure 3). In FFT, LU, and Erlebacher, the relative contribution of write latency decreases significantly, since many writes that hit in the write-through L1 cache had to experience L2 cache latency; with a write-back L1 cache, these writes must only take L1 access time. In contrast, the write latency component does not drop much in Radix, Water, and MP3D. In each of these applications, the write stall component is dominated by remote write misses, on which write-back caches have little effect.

The overall benefits of the two techniques of hardware prefetching and speculative loads on SC systems with write-back L1 caches are qualitatively similar to those with write-through L1 caches. The improvements in execution time range from 7% (Radix) to 28% (Erlebacher) for prefetching, and 28% (MP3D) to 60% (LU) for speculative load execution (relative to simple SC). As for the write-through case, the two techniques are more successful at reducing read latency rather than write latency; the difference from the write-through case is that the write latency forms a smaller part of the execution time.

One difference from the write-through case occurs with the speculative load configuration for LU. LU sees previously unseen benefits in store stall time. These benefits arise for reasons similar to the reduction in read stall time for LU with a write-through L1: reordering of accesses and prefetches cause many L1 conflict misses to instead coalesce in the MSHRs. This benefit did not arise in the write-through cache since all store accesses saw at least the L2 cache access time, regardless of whether or not they hit in the L1 cache. The majority of the benefits with speculative load execution for LU, however, come from faster loads and computation, as in the write-through case.

4.2 RC Implementations

Performance with Write-Through L1 Caches. While SC showed significant improvements from the optimizations of hardware-controlled prefetching and speculative loads, Figure 3 shows that RC does not in general experience much benefit from these optimizations.

Qualitatively, there are two key differences in the way the optimizations affect performance with RC and SC. As explained in Section 2, RC already allows increased overlap compared to SC; the optimizations help RC only when there is an outstanding acquire or when a store with a known address is waiting to reach the head of the instruction window. Furthermore, once a store reaches the head of the window, it retires immediately. This implies that most write latency is already hidden. Therefore, the net effect on performance of store prefetch is expected to be limited, and the effects caused by the write-through L1 cache with SC are not observed with RC.

Water is the only application helped unequivocally and significantly from the current optimizations to RC. It shows improvement in execution time from both prefetching (5.2% improvement over simple RC) and from speculative load execution (7.7% improvement over simple RC). The benefits are achieved by overlapping the prefetch of the critical section lock for the force updates, and the data within the critical section. Since the critical sections have low contention, the prefetch brings in valid data and is useful in hiding a large part of the latency. However, benefits from these techniques are far less significant than the corresponding benefits in SC.

Erlebacher experiences marginal benefits from speculative loads, with about 2.5% improvement in execution time. The main communication in Erlebacher proceeds in a pipeline, where the stages of the pipeline are tightly synchronized with flags. If the pipeline is slightly unbalanced, the flag and data values can be prefetched prematurely, eliminating some of the potential performance benefit from these optimizations.

The remaining applications either remain unchanged or degrade very slightly with the optimizations, possibly due to premature prefetches.

Impact of Write-Back L1 Caches. Overall, unlike the SC case, the choice of the first-level cache does not have significant impact on RC. The impact of the various optimizations with write-back caches is virtually identical to that with write-through caches.

For a given RC configuration, only one application (Radix) sees a significant reduction in execution time by replacing write-through caches with write-back caches. The difference arises with Radix only because it has a bursty irregular write pattern which overwhelms the secondary cache. Eventually, writes to remote data fill up the MSHRS, causing backup of subsequent requests, including write-throughs from L1. This resource backup eventually reaches into the L1 and the processor memory unit, adding contention to both reads and memory writes. With an L1 write-back cache, many writes hit in the L1; since these writes do not propagate to the L2, they relieve some of the contention and saturation present in the write-through configuration.

MP3D and LU show a marginal degradation in performance when replacing a write-through cache with a write-back cache. The reason is that our write-through cache is a no-write-allocate cache while the write-back cache is a write-allocate cache. Thus, the writes into the write-back cache exacerbate conflicts within the cache.

Summary for RC. In summary, for our applications, the optimizations used in RC do not provide much benefit; the best improvement in execution time was 7.7% for Water. For four applications, the optimizations did not make a difference or resulted in a very slight degradation. Thus, our experiments indicate that for RC, the cost of the on-chip hardware for the optimizations may not be justified. Regarding L1 cache write policy, our results show that except for one application, write-through L1 caches performed comparable to write-back L1 caches for RC.

4.3 Comparing SC and RC

Our results so far show that for our application suite, the simplest RC implementation outperforms the most optimized SC. This is especially pronounced in the case with write-through L1 caches, where simple RC provides over a factor of two speedup for FFT and LU, a speedup of 1.5 or more for Erlebacher, Radix and MP3D, and a speedup over 1.2 for Water. In the case of write-back L1 caches, the performance improvement is less dramatic, but still significant. The speedups for simple RC relative to the most optimized SC range from 1.15 for Water to 1.65 for Radix; three applications (LU, Radix, and MP3D) see a speedup of 1.5 or more. The difference in the results for write-through and write-back L1 caches arises because SC variants improve in absolute cycle count from write-through to write-back, while the absolute cycle counts seen by RC systems stay the same in both configurations, except for Radix. Thus, while SC needs a write-back cache for best performance, the performance of RC is largely independent of cache write policy. Accounting for possible additional latencies of having a write-back cache, the gap between RC (which can run just as well on a write-through L1 cache) and a high-performance SC (which needs a write-back L1 cache for best results) may increase further. Finally, either write-through or write-back primary caches with multiple cycle latencies are also likely to increase the gap between SC and RC.

4.4 Impact of a More Aggressive Protocol

As discussed in Section 3.1, to model a protocol with reasonable complexity, our caches delay ownership requests for writes to lines with pending shared reads. Thus, a write-prefetch seeking to obtain ownership of a cache line would be delayed if the cache had an earlier load miss to the same line. A more aggressive system could allow such ownership requests to be sent on to the directory, but would need to handle possible races from network reordering at the directory and cache; this aggressive system may improve the overlap of ownership requests, but at the cost of added design complexity. Such an enhancement may have an impact on applications with migratory read-write sharing such as Water and MP3D, or on applications with producer-consumer sharing such as Erlebacher where the producer reads the old value of the data before producing a new value.[1] However, this protocol enhancement will not have much impact on applications such as Radix or FFT, since we observe that these two applications do not see delayed ownership requests in the key sections of their code. On our architecture, the enhancement is also not likely to have a significant effect on LU since the lines that are read are likely to be replaced in the L1 cache by conflicts before they are written, and LU already exhibits an L2 write hit rate of 98%.

We did not simulate the above aggressive protocol because of its significantly higher complexity. Instead, to approximate the impact of such a system, we inserted (by hand) explicit software exclusive prefetch instructions immediately before all read operations that are soon followed by a write to the same word, for Water, MP3D, and Erlebacher. Below, we report results comparing the performance of the three applications with and without software prefetch instructions, for the versions of SC and RC that support speculative load execution and hardware store prefetching.

For SC, all three applications see further benefits in terms of reduced store latency with one exception. The exception is Erlebacher with a write-through primary cache; this configuration is not amenable to further benefits in the store component since Erlebacher achieves a very high secondary-cache hit rate. The benefits for other experiments range from a 12% improvement in execution time for Erlebacher with a write-back L1 cache to 22% improvement for Water with a write-back L1 cache (relative to the best SC without the software prefetch).

For RC, MP3D and Erlebacher do not achieve any improvements, since RC already hides store latencies effectively. Water sees a further 12% improvement in execution time because of more effective store-prefetching, which leads to faster releases and consequently faster acquires.

[1]In Erlebacher, some writes following reads to the same word access non-boundary planes in the 3-D array; such planes are accessed by only one processor. For such accesses, a four state protocol with a valid-exclusive state can get benefits similar to a protocol that overlaps read and ownership requests to the same line.

Comparing SC and RC, we find that this optimization has narrowed the performance gap by reducing SC's store limitations. Nevertheless, in MP3D, simple RC still sees a speedup of 1.44 relative to the most optimized SC with a write-through L1 cache and 1.3 with a write-back L1 cache. Erlebacher sees a simple RC speedup of 1.43 over the most optimized SC with a write-through L1, but finds nearly equal performance for the most optimized SC and simple RC with write-back caches. Water stands apart from the other two applications; the best SC now actually performs *better* than simple RC. However, since RC also sees benefits from the more aggressive protocol for Water, the best RC achieves a speedup of 1.18 over the best SC with a write-through L1 cache and a speedup of 1.1 with a write-back L1 cache.

Overall, these results suggest that the gap between SC and RC performance can be decreased by adding complexity to the cache-coherence protocol for some applications; however, a significant gap still remains for many applications. Furthermore, RC does not need the additional support to achieve its level of performance.

The above results should not be interpreted as indicative of the effects of software-controlled prefetching on the performance difference between SC and RC, since we have inserted software prefetches in a simple way only to approximate the more aggressive protocol. Previous work has shown that aggressive use of software prefetching can enhance the performance of both SC and RC on a processor with blocking reads [9, 13]. Processors with non-blocking reads can also be expected to see benefits for both models; however, the interaction between software prefetching and non-blocking reads is more complex [21] and an evaluation is outside the scope of this paper.

4.5 Impact of Larger Cache Sizes

We have also performed an analysis similar to that in Section 4.3 using much larger caches (first level 32 KB, 2-way associative, second level 2 MB, 8-way associative) for three representative applications: Water, in which the best SC and RC are close in both write-through and write-back; Radix, in which there is a large difference between SC and RC in both versions; and FFT, where the difference drops significantly from write-through to write-back. The results for Water and FFT are almost identical to those of our primary runs. With Radix, write-back caches no longer perform any better than write-through primary caches for RC; the difference is caused by a higher secondary-cache hit rate, leading to lower MSHR occupancy and subsequently preventing request backup even in the write-through case.

5 New Techniques to Tolerate Acquire Latency

Although RC provides significant benefits over SC, only two applications (Water and Erlebacher) saw any improvement over simple RC by using the two techniques of hardware-controlled prefetching and speculative loads. In Water, the two techniques obtain their performance benefits by overlapping data accesses within a critical section with the acquire of the lock. In Erlebacher, speculative load execution only shows benefit in the backward pipeline stage of the computation. Both of these applications spend a significant portion of their execution time in synchronization. Although speculative load execution can improve performance, RC must assume that all memory operations following an acquire in

```
Code for Processor P_j

for(i=1; i < N+1; i++){
    WaitFlag(Flag[j][i])
    AcquireMemBar
    DoWork(X[i])
    ReleaseMemBar
    SetFlag(Flag[j+1][i])
}
```

(a)

```
/* Prologue code */
WaitFlag(Flag[j][1])

for(i=1; i < N; i++) {
    AcquireMemBar
    WaitFlag(Flag[j][i+1])
    DoWork(X[i])
    ReleaseMemBar
    SetFlag(Flag[j+1][i])
}

/* Epilogue code */
AcquireMemBar
DoWork(X[N])
ReleaseMemBar
SetFlag(Flag[j+1][N])
```

(b)

Figure 4: Application of fuzzy acquires.

program order depend on that acquire. Therefore, operations after an acquire can only speculatively overlap with the acquire latency; if there is a mis-speculation, the system must rollback all later instructions. We propose two techniques to reduce the effect of acquire dependences by recognizing that some operations after an acquire can be independent of that acquire and can thus non-speculatively overlap with the acquire latency. We also show a way to further exploit this independence by decoupling acquire dependences from control dependences.

5.1 Fuzzy Acquires

Our first technique, *fuzzy acquires*, is similar to the notion of fuzzy barriers[14]. Fuzzy acquires can be used on systems which provide memory barrier or fence instructions (as supported by most current architectures, including SPARC V9, MIPS IV, and Alpha). On such systems, an acquire can be implemented as an ordinary non-blocking load immediately followed by a memory barrier. Fuzzy acquires explicitly separate the acquire load and the memory barrier following the load. Operations dependent on the acquire must follow the memory barrier; however, operations independent of the acquire can now be inserted between the read and the memory barrier. This allows the acquire latency to be overlapped with the independent instructions.

Fuzzy acquires are especially applicable when acquires occur in computation loops. For example, consider the code in Figure 4(a), performing a pipelined computation. In iteration `i`, processor P_j waits on flag `Flag[j][i]` (set by the previous processor in the pipeline), performs some computation on data `X[i]`, and sets the flag `Flag[j+1][i]` for the next processor. The `AcquireMemBar` and the `ReleaseMemBar` are memory fence instructions that enforce ordering between memory operations. The `AcquireMemBar` ensures that all previous loads (including acquires) complete before any subsequent operations are issued into the memory system, and the `ReleaseMemBar` ensures that all previous operations complete before subsequent stores (including releases) are issued. These partial fences can be implemented using different classes of `MEMBAR` instructions available in the SPARC V9 architecture[2].

The key observation used to exploit fuzzy acquires for the code in Figure 4(a) is that the flag acquire access in iteration `i+1` (i.e., acquire of `Flag[j][i+1]`) is independent of the computation of iteration `i`. We use software pipelining with fuzzy acquires to rewrite the given code segment as in Figure 4(b) so that the acquire of `Flag[j][i+1]` is accessed in the same iteration as the independent computation on data `X[i]`. The `AcquireMemBar` at the beginning of the iteration insures that the acquire of `Flag[j][i]` has completed before the access to `X[i]`. Since the computation on `X[i]` does not depend on the acquire of `Flag[j][i+1]`, it can now be overlapped with that acquire. Using fuzzy acquires has effectively increased the space between the acquire operation and its subsequent memory barrier operation and inserted useful computation from the previous iteration to overlap the latency of the acquire.

[2]Fuzzy acquires can also be used with full memory fences, as supported by the MIPS IV and Alpha architectures, with slight modifications to the code of Figure 4.

The following factors may limit the performance gain due to fuzzy acquires. The first limitation occurs because typical implementations of acquires involve spin-waiting in a loop. In such a case, if the flag is not available, the processor branch prediction algorithm is likely to fill up the instruction window with useless instructions from various iterations of the acquire loop. Section 5.3 describes a technique to eliminate this problem. Second, the size of the instruction window limits the overlap we can get, since the acquire does not retire from the instruction window until it is complete. Section 5.4 describes how this effect can be alleviated. Third, in a case with critical sections, if the independent computation added between the acquire read and the memory barrier is not completely overlapped, then the optimization may lengthen the critical section and increase subsequent acquire times.

The software pipelining for fuzzy acquires can be implemented transparent to the application programmer, by a suitably aggressive optimizing compiler in places where it can improve performance. In particular, moving the acquire read one iteration ahead still retains the ordering of memory operations imposed by the RCpc model [8].

5.2 Selective Acquires

Our second technique, *selective acquires*, eliminates memory fences which force all subsequent operations to await the completion of an acquire. Instead, this technique uses arithmetic instructions to explicitly and selectively establish only the needed acquire dependences. We achieve this by inducing a data dependence from the value returned by the acquire to the addresses of those memory operations that depend on the acquire. These dependences prevent the issue of dependent operations, while allowing other operations to issue non-speculatively.

For example, consider Figure 5(a), which shows two processors accessing a shared task queue. There is no semantic reason for the unrelated operations following P1's task enqueue to await the completion of the enqueue. A system could allow these later operations to issue non-speculatively while P1's lock access for the queue is pending. Selective acquires provide exactly this facility: exactly those operations dependent upon the acquire have their addresses set by (and thus dependent on) lock values. Every dependent operation does not need an explicitly added dependence. For example, in the case of multiple dereferencing of a pointer, only the first dereference must be made dependent on the synchronization; others become dependent by transitivity.

The pseudocode in Figure 5(b) represents an enqueue operation with selective acquires. Operations within the critical section are set to have address dependences on the lock. In this case the lock is expected to return 0 when it is free, so we add the value returned by the lock to the address of the dependent operation (i.e., the access to the tail pointer).

```
P1                               P2
...
generate new task;               ...
/* enqueue new task */           /* dequeue new task */

lock(L)                          lock(L)
insert task in queue             remove task from queue
unlock(L)                        unlock(L)
...                              ...
```

(a) Enqueue-dequeue

```
                                       ! new task pointer is in r5

EnQ:    acq    [lock],X                ! Acquire lock
        add    X, qtailptr, ptr        ! Add lock value to tail
        load   [ptr], tail             ! Get the queue tail
        add    tail, nextoffset, tail  ! Reserve queue space
        st     task, [tail]            ! Put task into list
        rel    0, [lock]               ! Release lock
Work:   ld     [var1], val
        ld     [var2], val2
        cmp    val1, val2
        ...
```

(b) Enqueue with added dependences

Figure 5: Code for a task queue.

When the lock is obtained, the address of the tail pointer will be unchanged, but until then, the address will appear unknown and the load of the pointer will not be issued. In cases where the synchronization returns a value other than zero, other operations, such as a subtraction followed by an addition, can be used for inducing explicit dependences. The pseudocode in Figure 5(b) explicitly generates a dependence only on the first load within the critical section, so only that load and operations dependent upon it must wait on the lock. The unrelated operations in the `Work` section may continue independently.

Selective acquires are not always applicable. For example, Processor P2 in Figure 5(a) performs a dequeue operation, where all memory operations subsequent to the critical section intrinsically depend on the values obtained within the critical section. The compiler, library writers, or high-level language programmers should use selective acquires only where appropriate; a simplification of this process is part of our ongoing work [1].

5.3 Decoupling Acquire Dependences from Control Dependences

Fuzzy acquires and selective acquires improve performance by eliminating artificial synchronization dependences in traditional memory consistency models. This section describes a technique to remove artificially introduced control dependences caused by implementations of synchronization.

Usually, synchronization operations are implemented using looping constructs, imposing a control dependence on subsequent instructions. It is possible that processor branch prediction may fill up the processor's instruction window with useless instructions from various speculative iterations of the synchronization loop. These useless instructions can prevent the processor from finding useful subsequent accesses to overlap with acquire latency. To address this issue, Gharachorloo et al. explicitly assume that the branch predictor will always predict that an acquire will succeed [10]. Forcing static prediction for a branch, however, may lead to repeated mispredictions and rollbacks in cases where the synchronization variables are not yet available.

We propose a mechanism to replace the acquire loop construct with a single equivalent instruction, avoiding the use of branches. This mechanism, called the *synchronization buffer*, can be implemented entirely off-chip and does not require changes to current commodity microprocessors.

Specifically, we tag the acquire memory operation as uncacheable and make it identifiable by the memory system in some way (e.g., through a certain range of addresses as possible with the MIPS R10000 [19], or through alternate space identifier bits as provided by the SPARC V8 and V9 architectures [25]). The off-chip synchronization buffer captures the tagged access and assumes responsibility for its completion. If the value returned by the memory system for the access does not indicate a successful acquire, the logic associated with the synchronization buffer transparently reissues the access, effectively implementing the synchronization loop in hardware. The synchronization buffer need only re-issue requests when an external coherence action (such as an invalidate) takes place. The buffer also acts as a small cache for synchronization variables. The small amount of hardware required for the buffer can sit at any level of the off-chip cache hierarchy, and does not require any modification to current processors. This synchronization buffer scheme can be used with both the previous and the new optimizations discussed in this paper.

5.4 Preliminary Evaluation of the New Techniques

This section evaluates our techniques, and also investigates a combination of our techniques with hardware-controlled prefetching. We do not combine our techniques with speculative loads because the latter seeks to speculatively perform all accesses subsequent to an acquire, while our techniques seek to non-speculatively perform some accesses after an acquire. We focus on Water and Erlebacher, which are the only applications that give benefits with the optimizations for RC seen in Section 4. Both of these applications spend a significant amount of execution time in synchronization.

Water. The simple RC version of Water has a synchronization time of 20.1% of which 12.2% is due to lock synchronization and 7.9% is due to barrier synchronization. The lock time is a rough estimate of the maximum improvement we can expect from our optimizations. For Water, we evaluate selective acquires along with the synchronization buffer; the fuzzy acquire optimization is not applicable. We apply the optimization primarily to the critical sections where a processor atomically updates several force fields of a water molecule structure. Only the operations within the critical section are dependent on the acquire; operations after the critical section can proceed in parallel with the acquire.

To use selective acquires, we remove any memory fence instructions for the critical section lock acquires, and make the loads in the critical section dependent on the acquire through an arithmetic dependence. Since the loads within the critical sections in Water act on different variables in the same molecule structure, it suffices to add only one explicit dependence to the molecule pointer.

The use of selective acquires, along with the synchronization buffer and store prefetching, achieves an 11.0% reduction in total execution time compared to simple RC. In contrast, prefetching alone fetched an improvement of 5.2% and the addition of speculative loads to prefetching fetched an improvement of 7.7% compared to simple RC. Recall that the maximum achievable reduction when all lock accesses are completely overlapped is 12.2%. Without the synchronization buffer, the benefits of selective acquires slightly exceed those with prefetching alone, and are comparable to gains with speculative loads. Thus, overall, our techniques provide small improvements over previous optimizations, but do not require the on-chip support for the speculative load buffer and its associated data and control paths.

Erlebacher. We evaluate the use of fuzzy acquires on Erlebacher. Only one phase in Erlebacher contains any communication or synchronization. The key computation in this phase consists of a forward substitution step and a backward substitution step. This computation involves a recurrence, where each value in the X-Y plane to be computed depends on its neighbor in the Z plane below it (for the forward substitution) and above it (for the backward substitution). The computation is distributed among processors along the Z-dimension, and is parallelized by implementing a pipeline. In each pipeline stage, a processor computes a few values in the X-Y planes allocated to it. At the end of a pipeline stage, processors that share a boundary plane synchronize with each other using flags, similar to the code in Figure 4(a). Here we focus only on the backward substitution pipeline; only this phase sees benefits from previous techniques.[3]

We use software pipelining to incorporate fuzzy acquires; loop iteration i issues an acquire of the flag for iteration $i+1$ and accesses data for iteration i. A memory fence at the beginning of each iteration ensures that data of iteration i is not accessed until the relevant acquire (issued in iteration $i-1$) completes. The code transformation roughly follows the example shown in Figure 4.

The use of fuzzy acquires gives a 4.3% reduction in execution time (relative to simple RC) without the synchronization buffer, and 7% with the synchronization buffer. This is comparable to the 6.3% reduction seen with the best RC using the previous techniques in the backward substitution phase. The synchronization buffer does not give significant further improvements because we do not allow the acquire to issue speculatively; the latter restriction arises because our synchronization buffer is assumed to be off-chip and because we map the acquire operation into an uncached region. As a result, the acquire stays at the head of the instruction window longer and the instruction window fills up. An alternative scheme would have the processor treat this acquire as a store, thus allowing it to retire earlier, and allowing the synchronization buffer to internally convert the operation to a flag acquire. For such a case, the `AcquireMemBar` would be substituted with a memory fence that stopped later loads and stores until all previous *stores* had completed. We do not evaluate such a scheme here, but expect that it will allow greater potential for overlap than the current system.

6 Related Work

Several studies have evaluated the performance of memory consistency models [9, 11, 13, 27]. This paper presents the first instruction-driven (or program-driven) simulation study for consistency models for aggressive ILP processors, evaluating two performance-enhancing techniques for consistency models used in such processors. We also study new techniques for overlapping acquire latency.

Two previous quantitative evaluations of memory consistency models have used processors with non-blocking reads. Gharachorloo et al. studied simple implementations of SC and RC; further, their study was trace-driven (as opposed to instruction or program-driven) and did not accurately model the effects of synchronization and network contention [11]. Zucker and Baer studied SC and RC, implementing SC both in a straightforward fashion and also with the prefetching optimization; however, the processors they inspected were single-issue and statically scheduled [27].

Previous studies have also considered techniques to overlap acquire latency. Several consistency models allow an acquire to be overlapped with previous operations of its processor, but not with later operations [2, 8]. Entry consistency allows acquires to be overlapped with certain subsequent operations [5]. However, entry consistency is motivated by software distributed shared-memory systems; therefore, quantitative studies of entry consistency focus on the reduction in the number of messages and amount of data communicated due to synchronization rather than on overlapping acquires with later operations. Our fuzzy acquire technique is a generalization of the fuzzy barrier technique [14] but requires only a memory fence instruction and applies to general acquires in RC. Rapid context switching on synchronization [3, 4] can effectively overlap synchronization latency, but requires special processor support. The QOLB primitive prefetches a lock variable and gets data along with the lock, resulting in decreased lock latency and overall communication [12]. Our techniques of overlapping acquires can be used in combination with QOLB since our technique allows the prefetching to be overlapped with more instructions, making the prefetching more effective.

There has been substantial work in prefetching for multiprocessors (e.g., [7, 20]). This paper addresses the use of non-binding hardware-controlled prefetching that exploits an aggressive processor's existing instruction window, to enhance the performance of consistency models. The interaction between non-binding software-controlled prefetching and the consistency model has been studied for simpler processors with blocking reads [9, 13]; software prefetching was found to significantly enhance the performance of both SC and RC. The interaction between software prefetching and non-blocking loads is more complex [21], and its impact on consistency models deserves further study.

7 Conclusions

Current ILP processors use the optimizations of non-binding hardware-controlled prefetching and speculative loads to enhance the performance of consistency models. Qualitatively, these optimizations seem to bring the performance of sequential consistency (SC) closer to release consistency (RC), potentially making SC more attractive to build in hardware for its easier programmability. This paper provides the first quantitative evaluation of various implementations of SC and RC for aggressive ILP processors. We evaluated the effects of hardware-controlled load and store prefetching, and the effects of speculative loads with hardware-controlled store prefetching. We found that for SC, these two techniques enhanced performance considerably (giving a speedup of over a factor of 2 in some cases). For RC, however, the optimizations showed an execution time improvement greater than 5% for only one case. The difference in performance between the two models, however, depends on the write policy of the primary cache and on the complexity of the cache-coherence protocol. For our fairly aggressive base cache-coherence protocol, the simplest RC implementation significantly outperforms the most optimized SC. The difference is higher with write-through primary caches than with write-back primary caches, but remains significant in both cases (three applications show a speedup of 1.5 or more with RC for both cache configurations). With a more complex cache-coherence protocol, SC achieves performance comparable to RC for two applications, but a significant per-

[3]In the forward substitution pipeline, remote memory latency is hidden because of a fortuitous branch misprediction, which allows even simple RC to overlap synchronization latency with later work.

formance gap remains for others. The performance of SC is highly sensitive to cache write policy and the aggressiveness of the cache-coherence protocol, while the performance of RC is generally stable across all implementations. Overall, our results show that RC hardware has significant performance benefits over SC hardware, and at the same time requires less system complexity with ILP processors.

Although RC performs well, the best RC implementation had significant acquire overhead in some applications. We proposed the alternative software techniques of fuzzy and selective acquires to overlap acquire latency. To alleviate the effect of control dependences, we proposed an off-chip hardware synchronization buffer to replace an acquire loop with a single equivalent instruction. Our experiments with the new optimizations show that they can provide comparable or better improvements than previous techniques, without the complexity of an on-chip speculative load buffer.

In choosing a consistency model, the hardware designer must consider both system performance and programmability. The techniques of this paper address hardware performance. However, SC at the application programming level also restricts compiler optimizations. To avoid these restrictions, it is likely that high-performance compilers will expose a release-consistent model to the applications programmer. If compilers mandate RC, then the improved performance and lower complexity of RC further favor supporting RC in hardware for systems with ILP processors.

This study has focused on increasingly used hardware techniques for enhancing the performance of consistency models. In the future, we plan to study the interaction of software-controlled prefetching with consistency models on aggressive ILP processors. We also plan to study the impact of increased overlap of memory operations through better compiler instruction scheduling for such processors.

8 Acknowledgments

We would like to thank Preston Briggs, Kourosh Gharachorloo, Norm Jouppi, Dan Lenoski, Satoshi Matsushita, Steve Scott, Marc Tremblay, and Rick Zucker for a variety of information on commercial systems. We also thank Vikram Adve, Jon Hall, Vivek Pai, Ram Rajamony, Willy Zwaenepoel, and the anonymous reviewers for their valuable comments on previous drafts of this paper.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Replacing Locks by Higher-Level Primitives. Technical Report TR94-237, Computer Science, Rice University, 1994.

[2] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd ISCA*, pages 2–13, 1995.

[4] R. Alverson et al. The Tera Computer System. In *Proc. of the Intl. Conf. on Supercomputing*, pages 1–6, 1990.

[5] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. *Compcon*, 1992.

[6] R. G. Covington et al. The Efficient Simulation of Parallel Computer Systems. *Intl. Journal of Computer Simulation*, 1:31–58, January 1991.

[7] F. Dahlgren and P. Stenstrom. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of the 1st Intl. Symp. on High Performance Computer Architecture*, 1995.

[8] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th ISCA*, pages 15–26, May 1990.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. of ASPLOS IV*, pages 245–257, 1991.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proc. of the 19th ISCA*, pages 22–33, 1992.

[12] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. of ASPLOS III*, pages 64–75, 1989.

[13] A. Gupta et al. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th ISCA*, pages 254–263, May 1991.

[14] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. of ASPLOS III*, pages 54–63, April 1989.

[15] D. Hunt. Advanced Features of the 64-bit PA-8000. Hewlett Packard, 1996.

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

[17] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. of the 8th ISCA*, pages 81–87, 1981.

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

[19] MIPS Technologies, Inc. *R1000 Microprocessor User's Manual, Version 1.1*, January 1996.

[20] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching. *JPDC*, pages 87–106, June 1991.

[21] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. Technical report, Rice University, July 1996.

[22] U. Rajagopalan. The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors. Master's thesis, Rice University, January 1995.

[23] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 285–298, 1995.

[24] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

[26] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, pages 24–36, 1995.

[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.