Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors

Kourosh Gharachorloo, Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

The memory consistency model supported by a multiprocessor architecture determines the amount of buffering and pipelining that may be used to hide or reduce the latency of memory accesses. Several different consistency models have been proposed. These range from *sequential consistency* on one end, allowing very limited buffering, to *release consistency* on the other end, allowing extensive buffering and pipelining. The *processor consistency* and *weak consistency* models fall in between. The advantage of the less strict models is increased performance potential. The disadvantage is increased hardware complexity and a more complex programming model. To make an informed decision on the above tradeoff requires performance data for the various models.

This paper addresses the issue of performance benefits from the above four consistency models. Our results are based on simulation studies done for three applications. The results show that in an environment where processor reads are blocking and writes are buffered, a significant performance increase is achieved from allowing reads to bypass previous writes. Pipelining of writes, which determines the rate at which writes are retired from the write buffer, is of secondary importance. As a result, we show that the sequential consistency model performs poorly relative to all other models, while the processor consistency model provides most of the benefits of the weak and release consistency models.

1 Introduction

Memory accesses that are not satisfied within the processor environment experience long latencies in large scale shared-memory multiprocessors. Two powerful techniques that can help reduce or hide this latency are buffering and pipelining of memory accesses. Unfortunately, the distributed memory, caches, and general interconnection networks used by large scale multiprocessors [3, 11, 15] can cause multiple requests issued by a processor to execute out of order. Depending on the memory consistency model supported, various restrictions are placed on the amount of buffering and pipelining allowed. Thus, the choice of the memory consistency model directly affects the performance of the machine.

Several memory consistency models have been proposed in the literature. The strictest model is *sequential consistency* (SC) [10], which requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. While conceptually intuitive and elegant, the model imposes severe restrictions on the outstanding accesses that a process may have, thus restricting the buffering and pipelining allowed. One of the least strict models is *release consistency* (RC) [6], which allows significant overlap of memory accesses given synchronization accesses are identified and classified into acquires and releases. Two other models that have been discussed in the literature are *processor consistency* (PC) [6, 8] and *weak consistency* (WC) [4, 5]. These fall between sequential and release consistency models in terms of strictness.

While the less strict models provide a higher potential for performance, they also require more complicated hardware and result in a more complex programming model. Data indicating the performance achieved under each of the above models is essential in allowing a system designer to make an appropriate decision on the model to support. So far, detailed performance data has not been available. This paper presents simulation results comparing the performance of these consistency models based on three engineering applications. The applications are a particle-based simulator used in aeronautics (MP3D) [13], an LU-decomposition program (LU), and a digital logic simulation program (PTHOR) [18]. The simulated architecture is based on the Stanford DASH multiprocessor [11].

Our results show that in an architecture with blocking reads and buffered writes (typical for current commercial microprocessors), the sequential consistency model does significantly worse than all other models. While this is somewhat expected, the surprising result is that the processor consistency model does almost as well as the release consistency model, and for one of the benchmarks, better than the weak consistency model. These results indicate that the gain from allowing reads to bypass pending writes (allowed in PC, WC, and RC) is much more important than allowing writes to be pipelined (allowed in WC and RC).

The next two sections present a description of the class of multiprocessor architectures, the simulation environment, and the benchmark applications used in this study. Section 4 presents background information and an implementation-oriented view of the consistency models. Simulation results are presented in Section 5. Section 6 explores the effects of prefetching on the models. A general discussion about the results and the related work is given in Sections 7 and 8. Finally, we conclude in Section 9.

2 Multiprocessor Architecture and Simulator

To enable meaningful performance comparisons between the models it is necessary to focus on a specific class of multiprocessor architectures. The reason is that tradeoffs may vary depending on the architecture chosen. For example, the tradeoffs for a small bus-based multiprocessor where broadcast is possible and miss latencies are ten to twenty cycles are quite different than the tradeoffs for a large scale multiprocessor where broadcast is not possible and miss latencies may be a hundred or more cycles.

For our study, we have chosen an architecture that resembles the DASH shared-memory multiprocessor [11], a large scale cache-coherent machine currently being built at Stanford. We use the actual parameters from the DASH prototype wherever possible, but have removed some of the limitations that were imposed on the DASH prototype due to design-time constraints. Figure 1 shows the high-level organization of the simulated architecture, which consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes and cache coherence is maintained using a distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching the block, and point-to-point messages are sent to invalidate remote copies. Acknowledgement messages are used to inform the originating processing node when an invalidation has been completed. Although the DASH prototype has four processors within each node, we simulate an architecture with only one processor per node. This allows us to isolate the effect of the consistency models more clearly.

Due to the distribution of the memory system and the general interconnection network in this architecture, two accesses that are issued in order may complete in a different order. For example, if a producer process writes a variable and then sets a flag, the consumer process may see the flag set while still holding a stale value of the variable in its cache. To preserve the desired order, the setting of the flag can be delayed until the write to the variable is performed. In the simulated architecture, a read is considered *performed* when the return value is bound and can not be modified by other write operations. Similarly, a write is considered *performed* when exclusive ownership is acquired and all other copies have been invalidated. For simplicity, we assume writes are atomic [5]. The architecture provides a mechanism to keep track of multiple outstanding accesses for each processor and allows for the processor to be stalled until all pending accesses have performed (for details, see [6]). The notion of being performed and having completed will be used interchangeably in the rest of the paper.

Figure 1 also shows the organization of the processor environment. Each CPU in the system contains a 64 Kbyte write-through data cache. The write-through cache enables processors to do single cycle write operations. The first-level data cache interfaces to a 256-Kbyte second-level write-back cache. The interface consists of a read buffer and a write buffer. Both the first and second-level caches are direct-mapped and support 16-byte lines.

In the above architecture, the processor or the write buffer may be stalled due to constraints imposed by the consistency model or purely due to implementation constraints. For example, although a consistency model may allow multiple outstanding accesses, the implementation of the cache may not allow this. To separate out the consistency model issues from the implementation

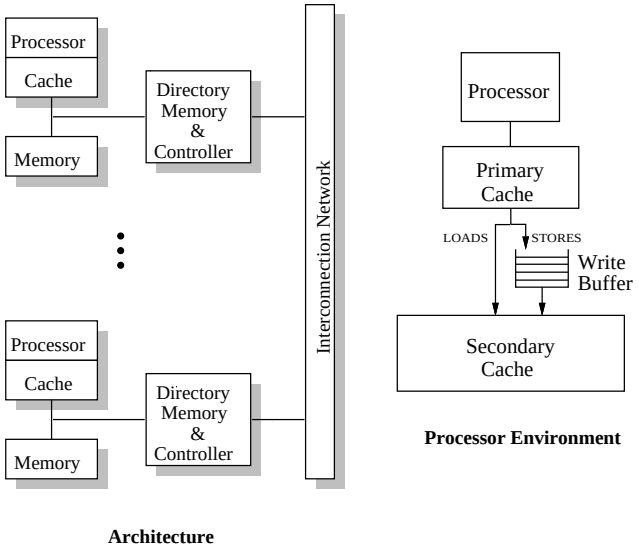

Figure 1: The simulated architecture and processor environment.

decisions, we will examine the performance of the models under implementations of varying aggressiveness. We will consider three versions of the implementation: (i) LFC, an aggressive implementation with lockup-free caches [9, 16], reads that bypass the write buffer, and a 16 word deep write buffer; (ii) RDBYP, which is the same as LFC, except caches are no longer lockup-free; and (iii) BASIC, which is the same as RDBYP, except that reads are not allowed to bypass the write buffer. Lockup-free caches allow multiple accesses to be in service simultaneously, as opposed to servicing one access at a time. Unless stated otherwise, all performance comparisons will use the LFC version of the CPU, since it minimizes shortcomings due to the implementation.

For our studies, an event-driven simulator is used to simulate the major components of the DASH architecture at a behavioral level. The simulations are based on a 16 processor configuration. The simulator is tightly coupled to the Tango [7] reference generator to assure a correct interleaving of accesses. For example, a process doing a read operation is blocked until that read completes, where the latency of the read is determined by the architecture simulator. Main memory is distributed across all nodes and allocated using a round-robin scheme for the applications.

The latency of a memory access in the simulated architecture depends on where in the memory hierarchy the access is serviced. Table 1 shows the latency for servicing an access at different levels of the hierarchy, in the absence of contention (the simulation results include the effect of contention, however). The latency shown for writes is the time for acquiring exclusive ownership of the line, which does not necessarily include the time for receiving acknowledgement messages from invalidations. The following naming convention is used for describing the memory hierarchy. The *local node* is the node that contains the processor originating a given request, while the *home node* is the node that contains the main memory and directory for the given physical memory address. A *remote node* is any other node.

Synchronization primitives are also modeled after DASH. The queue-based lock primitive [11] is used for supporting locks. In general, locks are not cached except when a processor is spinning on a locked value. When the lock is released, if there are any waiting processors, one is chosen at random and is granted

Table 1: Latency for various memory system operations in processor clocks. Numbers are based on 25MHz processors.

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	12 pclock
Fill from Local Node	22 pclock
Fill from Remote Node	61 pclock
Fill from Dirty Remote, Remote Home	80 pclock
Write Operations	
Owned by Secondary Cache	2 pclock
Owned by Local Node	17 pclock
Owned in Remote Node	57 pclock
Owned in Dirty Remote, Remote Home	76 pclock

the lock using an update message. Acquiring a free lock takes 18 and 59 processor cycles for local and remote locks, respectively. Barriers are implemented using fetch-and-increments in the gather stage and using update writes for releasing the waiters. The total latency to perform a barrier for 16 processors, given all reach the barrier at the same time, is about 330 processor cycles.

3 Benchmark Applications

The programs chosen for this evaluation represent applications used in an engineering computing environment. All of the applications are written in C and use the synchronization primitives provided by the Argonne National Laboratory macro package [12]. The three applications that we study are MP3D, LU, and PTHOR.

MP3D [13] is a 3-dimensional particle simulator. It is used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps. During each time step, the molecules are picked up one at a time and moved according to their velocity vectors. If two particles come close to each other, they may undergo a collision based on a probabilistic model. Collisions with the object and the boundaries are also modeled. The simulator is well suited to parallelization because each molecule can be treated independently at each time step. The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 10,000 particles, a 64x8x8 space array, and simulated 5 time steps.

LU performs LU-decomposition for dense matrices. The primary data structure in LU is the matrix being decomposed. Working from left to right, a column is used to modify all columns to its right. Once all columns to the left of a column have modified that column, it can be used to modify the remaining columns. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column has been produced, and then that column is used to modify all columns that the processor owns. Once a processor completes a column, it releases any processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

PTHOR [18] is a parallel distributed-time logic simulator based on the Chandy-Misra simulation algorithm. The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then looks up the net data structure to determine which elements are affected by the output change and schedules the newly activated elements on to task queues. For our experiments we simulated fifteen clock cycles of a multiplier circuit consisting of the equivalent of 5000 two-input gates.

Table 2: General statistics on the benchmarks.

Program	*Busy Cycles (×1,000)*	*Shared References (×1,000)*	*Processor Utilization on IDEAL*
MP3D	6,400	2,087	98.8
LU	27,728	8,276	94.1
PTHOR	89,984	22,556	96.2

Table 2 presents some general information about the three applications. To minimize the architectural dependence for such measurements, a latency of one cycle was assumed for all accesses (we will refer to this as the IDEAL architecture). Busy cycles specify the amount of useful cycles in the program, while shared references indicates the number of read, write, and synchronization accesses issued by the program. The high processor utilization on the IDEAL architecture implies there is sufficient parallelism in the algorithms for 16 processors. Table 3 presents more detailed statistics on the access behavior of the applications. The rate of read and write misses and the rate of synchronization are important factors in determining the relative performance of the models. Table 3 shows that MP3D and PTHOR have relatively high miss rates, while the miss rates are substantially lower in LU. In the case of LU, the caches are large enough to hold the matrix completely and the application does not have a lot of communication. For all three application, the number of read misses is higher than the number of write misses. Finally, PTHOR is shown to have the highest rate of synchronization compared to the other two applications.

4 Consistency Models and Their Implementation

This section provides a general overview of the consistency models. We also describe the implementation restrictions that are imposed by each model. The main goal of this section is to develop intuition about situations in which the less strict models are expected to perform better. The contents of this section will also form the basis of arguments used to explain simulation data in the following sections.

4.1 General Overview

A consistency model imposes restrictions on the order of shared memory accesses initiated by each process. The strictest model, originally proposed by Lamport [10], is sequential consistency (SC). Sequential consistency requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Processor consistency (PC) was proposed by Goodman [8] to relax some of the restrictions imposed by sequential consistency. Processor consistency requires that writes issued from a processor may not

Table 3: Number of shared read, write, and synchronization accesses for a 16 processor configuration. Numbers in parentheses are rates given as references per thousand cycles on the IDEAL architecture.

Program	*reads (×1,000)*	*writes (×1,000)*	*r/w ratio*	*read misses (×1,000)*	*write misses (×1,000)*	*rmiss/wmiss ratio*	*locks*	*unlocks*	*barriers*
MP3D	1,454	633	2.3	210	151	1.4	0	0	560
	(227)	(99)		(33)	(24)		(0.0)	(0.0)	(0.09)
LU	5,543	2,727	2.0	254	75	3.4	3,184	3,184	32
	(200)	(98)		(9.2)	(2.7)		(0.1)	(0.1)	(0.00)
PTHOR	19,576	2,240	8.7	2,267	935	2.4	360,269	360,269	19,536
	(218)	(25)		(25)	(10)		(4.0)	(4.0)	(0.22)

be observed in any order other than that in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. The constraints to satisfy processor consistency are specified formally in [6].

A more relaxed consistency model can be derived by relating memory request ordering to synchronization points in the program. The weak consistency model (WC) proposed by Dubois et al. [4, 5] is based on the above idea and ensures that memory is consistent only at synchronization points. As an example, consider a process updating a data structure within a critical section. Under SC, every access within the critical section is delayed until the previous access completes. But such delays are unnecessary if the programmer has already made sure that no other process can rely on the data structure to be consistent until the critical section is exited. Weak consistency exploits this by allowing accesses within the critical section to be pipelined. Correctness is achieved by guaranteeing that all previous accesses are performed at the beginning and end of each critical section.

Release consistency (RC) [6] is an extension of weak consistency that exploits further information about synchronization by classifying them into acquire and release accesses. An *acquire* synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A *release* synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access (see [6] for discussion of read-modify-write accesses). Similarly, a release is always associated with a write synchronization access. In contrast to WC, RC does not require accesses following a release to be delayed for the release to complete; the purpose of the release is to signal that previous accesses are complete, and it does not have anything to say about the ordering of the accesses following it. Similarly, RC does not require an acquire to be delayed for its previous accesses.

Figure 2 shows the restrictions imposed by each of the four consistency models studied in the paper.[1] The ordering restrictions are presented in terms of when an access is allowed to perform. As shown, sequential consistency can be guaranteed by requiring shared accesses to perform in program order. Processor consistency allows more flexibility over SC by allowing read operations to bypass previous write operations. Weak consistency and release consistency differ from SC and PC in that they exploit information about synchronization accesses. Both WC and RC allow accesses between two synchronization operations to be pipelined, as shown in Figure 2. The numbers on the blocks denote the order in which the accesses occur in program order. The figure shows that RC provides further flexibility by exploiting information about the type of synchronization.

[1]The weak consistency and release consistency models shown are the WCsc and RCpc models, respectively, in the terminology presented in [6].

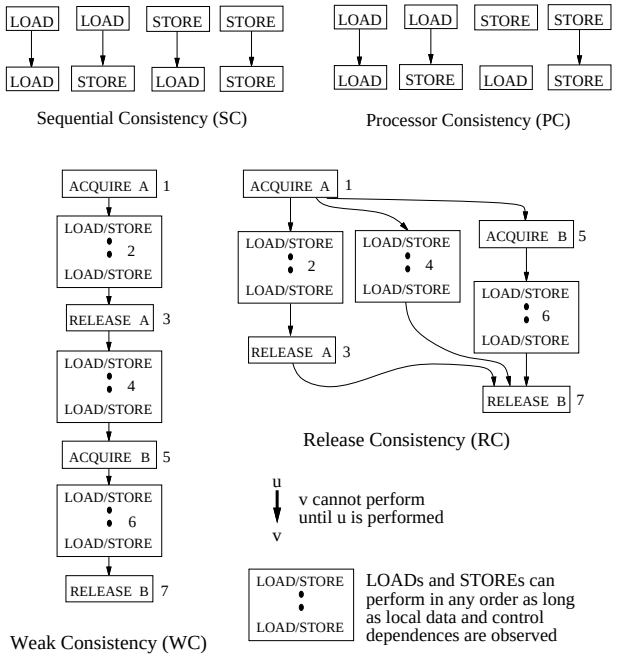

Figure 2: Ordering restrictions on memory accesses.

4.2 Implementation Oriented View

Table 4 summarizes the implementation restrictions imposed by each of the models given processors with blocking reads (these are sufficient, but not necessary, restrictions for satisfying each model). A *BASE* model has been added to the four consistency models discussed earlier. This is the most constrained model and is used as baseline for all performance comparisons. It incorporates no buffering or pipelining and waits for each read and write to complete before proceeding. The implementation restrictions are considerably simplified for processors that block on loads (typical of most commercial microprocessors). All consistency model constraints, where an access must block for a preceding load or acquire, are automatically satisfied.

To qualitatively compare the performance of the consistency models, consider the sources of idle time for a processor. At the architecture level, the idle time consists of (i) read accesses

Table 4: Implementation of consistency models.

Operation	**BASE**	**SC**	**PC**	**WC**	**RC**
1. Read	(a) Processor issues read and stalls for read to perform. Note: (i) No pending writes are possible. See point 2.	(a) Processor stalls for pending writes to perform (or, in very aggressive implementations, to gain ownership [1]). (b) Processor issues read and stalls for read to perform.	(a) Processor issues read and stalls for read to perform. Note: (i) Reads are allowed to bypass pending writes.	(a) Processor issues read and stalls for read to perform. Notes: (i) Reads are allowed to bypass pending writes. (ii) For interaction with pending releases, see point 4.	(a) Processor issues read and stalls for read to perform. Note: (i) Reads are allowed to bypass pending writes and releases.
2. Write	(a) Processor issues write and stalls for write to perform. Note: (i) No need for write buffer.	(a) Processor sends write to write buffer (stalls if write buffer is full). Note: (i) Write buffer retires a write only after the write is performed, or in very aggressive implementations, when ownership is gained [1].		(a) Processor sends write to write buffer (stalls if write buffer is full). Notes: (i) Write buffer does not require ownership to be gained before retiring a write. (ii) For interaction with acquires/releases, see points 3,4.	
3. Acquire	*Treated as Read*	*Treated as Read*	*Treated as Read*	(a) Processor stalls for pending writes and releases to perform. (b) Processor issues acquire and stalls for acquire to perform.	(a) Processor issues acquire and stalls for acquire to perform. Note: (i) Processor does not need to stall for pending writes and releases.
4. Release	*Treated as Write*	*Treated as Write*	*Treated as Write*	(a) Processor sends release to write buffer (stalls if write buffer is full). Notes: (i) Write buffer can not retire the release until all previous writes are performed. (ii) Write buffer stalls for release to perform. (iii) Processor stalls at next read after release for release to perform.	(a) Processor sends release to write buffer (stalls if write buffer is full). Note: (i) Write buffer can not retire the release until all previous writes and releases are performed.

stalling for data, (ii) processor stalling for previous writes (or releases) to complete, (iii) write (or release) accesses stalling when write buffer is full, and (iv) acquire accesses spinning until the corresponding release is observed. The performance with a particular consistency model depends on how effective the model is in reducing this idle time.

Let us first compare the BASE and SC models as shown in Table 4. While both models guarantee sequential consistency, SC is a more aggressive formulation. The main difference is in how writes are treated. For the BASE model, the processor stalls immediately after each write until the write completes. With SC, the writes are put in the write buffer and the processor is stalled at the following read operation instead. As a result, part of the write latency can be overlapped with computation up to the next read. In most cases, however, a read access occurs soon after a write miss, and most of the latency for completing the write miss is seen by the processor.

Between the SC and PC models, the major difference is that PC allows reads to be performed without waiting for pending writes in the write buffer. This significantly decreases the total delay experienced by the processor, thus increasing the performance. Of course, one side effect is that there is a higher probability that the write buffer may get full and stall the processor in PC. As will be shown in the simulation results section, this rarely happens in practice. The reason for this is intuitive: write misses are well interleaved with read misses rather than being clustered, and the number of read misses usually dominates. Since the processor has blocking reads, the stall duration for read misses provides time for write misses to be retired from the write buffer, thus preventing it from getting full.

Comparing the PC and WC models, the WC model exploits knowledge about synchronization accesses. In PC, the order among write accesses has to be preserved. In WC, however, reads and writes between two synchronization accesses can be performed in any order provided uniprocessor control and data dependences are maintained. As a result, reads can bypass writes, as in the case of PC. However, in contrast to PC, writes can be pipelined. The pipelining allows writes to be retired from the write buffer without requiring ownership to be obtained. Thus the writes can be retired at a much faster rate than is possible in PC. This can help performance in two ways. First, the chances of the write buffer filling up and stalling the processor are smaller compared to PC. Second, if there is a release operation (e.g., unlock operation) behind several writes in the write buffer, then a remote processor trying to do an acquire (e.g., lock on the same variable) can observe the release sooner, thus spinning for a shorter amount of time. Although the release operation in WC must wait until all invalidations have completed

for the previous writes, it does allow the invalidations for multiple writes to be overlapped. These reasons indicate that WC will provide performance advantages over PC if there is a significant clustering of writes, especially if the clustering is before release accesses.

The disadvantage of WC as compared to PC is the following. In PC, the processor is never stalled at a read (or acquire) for pending writes (or release) to be performed. However, in WC, the processor stalls at an acquire for previous writes or releases to complete, and at the first read access after a release for the release to complete. This can seriously degrade performance if the application has a high rate of synchronization. We will show in the results section that for one of the programs that uses fine-grain synchronization (PTHOR), WC does worse than PC.

Finally, comparing WC and RC, the RC model removes the shortcoming of WC described above. Similar to PC, RC never stalls the processor at a read or an acquire for previous writes or releases to complete. Consequently, RC can offer the best performance of all models. However, as in the case of WC, the performance gains over PC occur only when there is a significant clustering of writes.

The performance gains from relaxing the consistency model are accompanied by a more complex implementation. The implementation difference among the models arises from the different number of outstanding requests allowed by each model. For the implementation to allow multiple outstanding requests, the cache needs to be lockup-free. In addition, there needs to be a mechanism for keeping track of each outstanding request to know when the request is complete. BASE and SC do not require a lockup-free cache since at most one outstanding access is allowed. PC can at most have one read and one write access outstanding. WC and RC allow multiple outstanding accesses (blocking reads limit the number of outstanding read accesses to one, however). The lockup-free cache for PC is simpler than for WC and RC since there are at most two accesses, a read and a write, outstanding at any time. Furthermore, WC and RC require additional counters to keep track of when the outstanding accesses are complete (see [6] for more details).

There are several secondary effects that are not discussed above. For example, less strict models allow accesses to be issued at a faster rate that in turn may result in more contention at the network or in the memory system, thus increasing the latency of accesses. This may offset the gains from the less strict models. We will further discuss the impact of such side effects in the next section.

5 Simulation Results

In this section, we present a comparative analysis of the performance achieved by the various consistency models. Section 5.1 presents simulations for the LFC architecture described in Section 2. Since LFC is the most aggressive architecture, the results are primarily affected by the constraints imposed by the consistency model and not by limitations of the implementation. Section 5.2 presents results on the relative performance of the models for less aggressive implementations.

5.1 Results on the LFC Architecture

Figure 3 shows the relative performance of the models for each application. For the purposes of this paper, we define performance as the processor utilization achieved in an execution. The reason for using processor utilization as the figure of merit is that it provides reasonable results even when the program's control path is not deterministic and depends on relative timing of synchronization accesses. The processor utilization for each model is normalized to the performance of the BASE model for that program. The results show a wide range of performance gains due to the less strict models. Moving from BASE to SC, the gains are minimal. The largest gains in performance arise when moving from SC to PC. Surprisingly, WC does worse than PC for PTHOR. RC performs better than all the other models, but the gains over PC are small. The maximum gain from relaxing the consistency model is about 41% for MP3D, 29% for PTHOR, and 11% for LU.

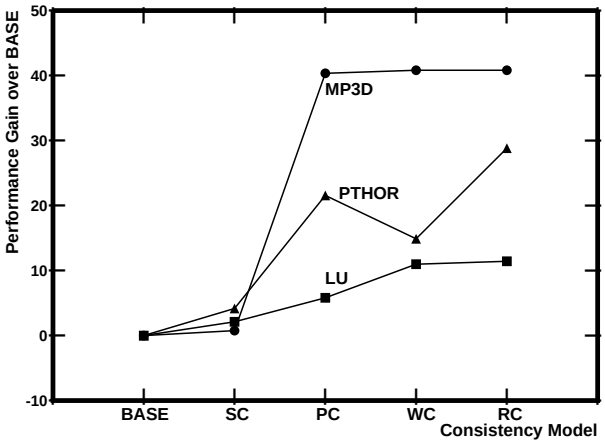

Figure 3: Relative performance of models on LFC.

To better understand the above results, in Figure 4 we present a breakdown of the execution time for the applications under each of the models. The execution time of the models are normalized to the execution time of BASE for each application. The bottom section of each column represents the busy time or useful cycles executed by the processor. The black section above it represents the time that the processor is stalled waiting for reads. This time does not include the time that a read/acquire access may be stalled waiting for previous writes to perform. This time is represented by the section above it. The three sections on top of that represent the stalls due to write buffer being full, time spent spinning while waiting for acquires to complete, and time spent waiting at a barrier.[2]

Some general observations that can be made from the breakdown are: (i) the latency of read misses forms a large portion of the idle time, especially once we move to PC, WC, and RC; (ii) the major reason for BASE and SC to be worse than the other models is the stalling of the processor before reads (and acquires) for pending writes to complete; (iii) the write buffer being full does not seem to be a factor in hindering the performance of PC; and finally, (iv) the reason for WC performing worse than PC and RC is the extra processor stalls at acquires and the first read after release accesses (as described in Section 4). The small variation in busy times for PTHOR is due to the non-deterministic behavior of the application for the same input. We now look at the comparative performance of the models in greater detail.

[2]It is difficult to distinguish the shade of some sections with a very small height. Usually, the small height implies that those sections are not very important, so they can safely be ignored.

Table 5: Statistics on write misses (including releases).

Program	*Write Miss Rate for BASE (per 1000 cycles)*	*Fraction followed by a Write Miss*	*Average Distance (cycles)*	*Fraction followed by a Read*	*Average Distance (cycles)*
MP3D	4.2	0.01	3.6	0.99	2.0
LU	1.3	0.01	11.9	0.99	5.9
PTHOR	2.4	0.18	12.6	0.82	5.1

5.1.1 Performance of SC versus BASE

The performance differences between SC and BASE arise from the fact that SC is able to overlap the latency of write misses up to the next read access. Thus, the expected gains depend on the frequency with which write misses occur, and the average distance between a write miss and the following read. Clustering of write misses, with no reads in between, also helps SC as the processor is not stalled.

The data in Figures 3 and 4 show that the SC model does not perform significantly better than BASE. Table 5 provides relevant data to explain these results. For executions under the BASE model it presents: (i) the frequency of write misses per 1000 cycles; (ii) the fraction of write misses that were followed by another write miss with no intervening reads (indicates write clustering) and the average distance between them; and (iii) the fraction of write misses that were followed by reads and the average distance between the two.

Given the contents of Table 5, the reasons for the relative performance gains of the applications are apparent. In MP3D we see that although the frequency of write misses is relatively high, the writes are almost immediately followed by a read access—the read is on average 2 cycles away. Thus the complete write miss penalty of about 75 cycles is observed by the read. In the case of LU the gains are small because the write miss rate is low, and again, the read follows the write miss within 6 cycles. In PTHOR, there is some write clustering observed, and the distance between write misses is larger. Consequently, the gain is higher than in MP3D and LU.[3]

In general, we expect performance gains from BASE to SC to be small for most applications. This is because reads are expected to be closely interleaved with writes. Significant write clustering may occur sometimes, for example, when initializing data structures, but such occurrences are expected to be infrequent.

5.1.2 Performance of PC versus SC and BASE

PC is the first model where sequential consistency is abandoned. For this extra complexity in the programming model, what are the benefits? The main benefit comes from the fact that reads do not have to stall for pending writes to perform. However, some of the benefits may be lost if the write buffer gets full and stalls the processor. As we will show later, this second factor turns out not to be an issue.

Focusing on the magnitude of gains from PC, the gains will be large if the frequency of write misses is high or the cost of write misses is high. Since the cost of write misses is about the same for all applications, the relative gains depend primarily on the frequency of misses. From Table 5 we see that for the BASE architecture, the frequency of write misses (including release

[3]The results for BASE are slightly pessimistic due to the extra write hit latency of the two-level cache structure assumed in our simulations. This is because BASE stalls the processor on every write, including write hits. SC is less sensitive to this effect since writes are buffered and there is usually enough computation to overlap the small latency of write hits.

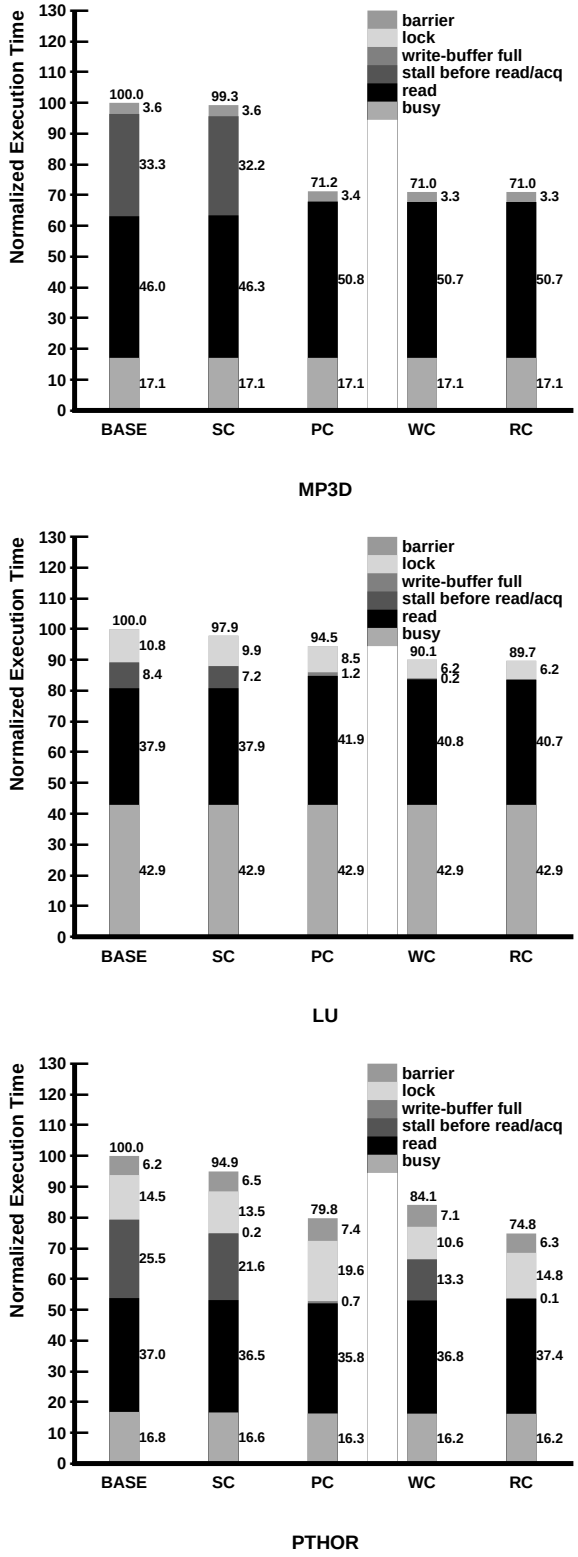

Figure 4: Breakdown of execution time on LFC.

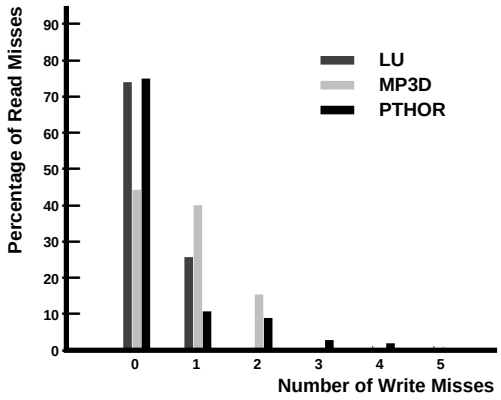

Figure 5: Distribution of write misses (including releases) between read misses (including acquires).

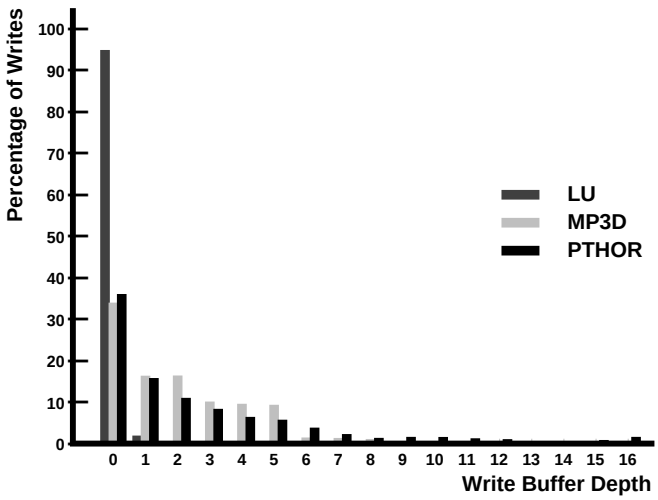

Figure 6: Distribution of writes in the write buffer for PC.

accesses) per 1000 cycles is 4.2 for MP3D, 1.3 for LU, and 2.4 for PTHOR. Consequently, we should expect large gains for MP3D, medium gains for PTHOR, and small gains for LU. This is indeed the case, as can be seen from Figures 3 and 4.

Another secondary effect also becomes apparent from Figure 4. Since time is compressed in PC, that is, the same number of useful instructions is executed in a shorter time compared to BASE, congestion in the memory system increases and read latencies go up. For example, for MP3D, the fraction of (normalized) cycles spent on read misses goes up from 46% to 51% as we move from BASE to PC. Consequently, some of the savings due to hiding write miss latencies are lost.

We now return to the issue of the write buffer being full. Although difficult to make out from Figure 4, the write buffer stall times are negligible for MP3D, 1.2% for LU, and 0.7% for PTHOR. These numbers are small primarily because: (i) the multiprocessor has blocking reads, and (ii) the number of read misses is larger than the number of write misses (see Table 3). As a result, on average, there is enough time for the write buffer to retire writes while the processor is stalled for read misses. Of course, the write buffer can build up for short periods of time due to clustered writes, but in our benchmark applications we did not see significant clustering. Some indication of the interleaving of read and write misses is given by the data in Figure 5 which is a histogram of the number of write misses between read miss pairs. The data shows that read misses and write misses are well interleaved. In Figure 6, we show the fill depth of the write buffer as encountered by write accesses for each of the applications. The write buffer depth of 16 used in our simulated architecture seems more than sufficient for the clustering present in the applications.

In summary, we see that the PC model is relatively successful in hiding almost all of the latency of writes given a reasonably deep write buffer. Since the comparison of PC and WC is more involved than the comparison with RC, we next examine PC versus RC. Subsequently, we compare PC and WC.

5.1.3 Performance of PC versus RC

The RC model provides all the performance benefits of the PC model. In addition, by exploiting information about the synchronization accesses, it allows pipelining of writes. That is, writes can be retired from the write buffer before ownership has been obtained. The fact that writes are retired at a faster rate has two implications: (i) the write buffer becoming full is less of a problem, and (ii) in cases where a release operation is behind several writes in the write buffer, that release can be observed sooner by a processor waiting to do an acquire. As was discussed in the previous subsection, the write buffer getting full is really not a problem for PC. Therefore, most gains, if observed, will be due to faster completion of synchronization.

Figure 3 shows that the performance of MP3D is comparable under PC and RC. As can be seen from Figure 4, PC is not stalled due to a full write buffer in MP3D, and the idle time corresponding to synchronization is approximately the same for the two models. In MP3D, the rate of synchronization is too small to have an effect, and thus the benefits of RC are negligible.

In contrast to MP3D, LU and PTHOR do exhibit a small performance gain from PC to RC. For LU, the idle time breakdown shows that the difference is mainly due to faster synchronization and because RC does not have write buffer stalls (a gain of about 2.3% and 1.2%, respectively, as shown in Figure 4). In LU, processors synchronize waiting for the pivot column to become ready. The processor updating the pivot column does a series of writes before doing the release operation, indicating that the column is ready. By allowing pipelining of writes, RC allows the release to be issued earlier and thus reduces the synchronization time. The detailed simulation statistics show that the average waiting time for gaining access to a column goes down from 1730 cycles under PC to 1250 cycles for RC. The large waiting times in LU are due to the slight load imbalance in the application (see Table 2).

The idle time breakdown for PTHOR also shows that most of the performance gain is due to faster synchronization (about 4.8% in normalized cycles). Indeed, simulations show that the average time to obtain a lock goes down from 285 cycles for PC to 218 cycles for RC. The faster time for synchronization occurs because PTHOR exploits fine-grain synchronization with tight producer-consumer relationships. Although PTHOR has a very high rate of synchronization, several of the releases (unlocks) are not in the critical path, that is, there is usually no process waiting for the lock to be released. Therefore, the gains from making releases visible slightly earlier is not as large as would otherwise be expected. Barriers also take slightly less time under RC. The

total time for read misses has gone up, however. A closer look at the simulation statistics shows that the increase in the read miss idle time was due to a larger number of read misses. This is most probably due to the fact that the program took a slightly different execution path under RC.

5.1.4 Performance of WC versus PC and RC

We first compare WC to RC and explain why WC performs worse than RC. The differences between WC and RC arise because WC does not differentiate between acquire and release synchronization operations. Consequently, any synchronization operation must conservatively satisfy the constraints of both release and acquire. Thus, compared to RC, WC stalls the processor at an acquire until pending writes and releases complete. In addition, the processor is stalled for pending releases if it attempts to do a read operation.

The results in Figure 3 show that MP3D and LU perform comparably under WC and RC. The reason is that MP3D and LU have very low synchronization rates (about once every 10,000 cycles on the IDEAL architecture). Therefore, the few extra stalls in WC do not substantially increase the idle time. However, for PTHOR, RC performs noticeably better than WC since PTHOR has a much higher synchronization rate.

To understand the performance of PTHOR better, we gathered data regarding the extra stalls introduced by WC over RC. We found that there is frequently a write miss or release about 20 cycles before an acquire operation. Similarly, there is a read access within a small number of cycles after a release. Therefore, at most 20 cycles of the latency of the write miss or release could be hidden and the rest of the latency is visible to the processor. More detailed data show that virtually all releases caused a delay for the next read and 30% of the acquires were delayed due to previous write misses.

We now compare WC and PC and explain the surprising result that PC sometimes performs better than WC. WC has the advantage that writes can be retired at a faster rate from the write buffer. The disadvantage of WC to PC is the same as the disadvantage of WC to RC, in that WC stalls the processor at some points for pending writes and releases to perform.

Figure 3 shows that WC performs slightly better than PC for LU, the same for MP3D, and worse for PTHOR. For LU, the performance is better because pipelining of writes was shown to be effective in reducing the idle time due to synchronization delays. In addition, the extra stalls introduced by WC were not an issue in LU. For MP3D, the discussion on PC and RC explained why pipelining does not gain anything over PC. In addition, WC is not hindered due to the extra stalling as described above. This explains why the two models perform comparably on MP3D. PTHOR shows an interesting result. In PTHOR, pipelining was important to reduce the time for synchronization. However, we also saw that WC causes stalls quite often. The disadvantage of the stalls turns out to be greater than the advantage of pipelining writes in this case. Therefore, we see WC performing worse than PC.

The relative performance of WC as compared to PC and RC may be affected by the fact that our simulated architecture does not cache locks. Caching locks is beneficial when a processor acquires and releases a lock several times with no other processor accessing the lock in between. All models gain from a faster acquire if the lock is found in the cache. As for the reduced latency of the release, PC and RC do not benefit since they already hide write and release latencies. WC, however, can potentially benefit. This may reduce the difference in performance observed for WC versus PC and RC in PTHOR.

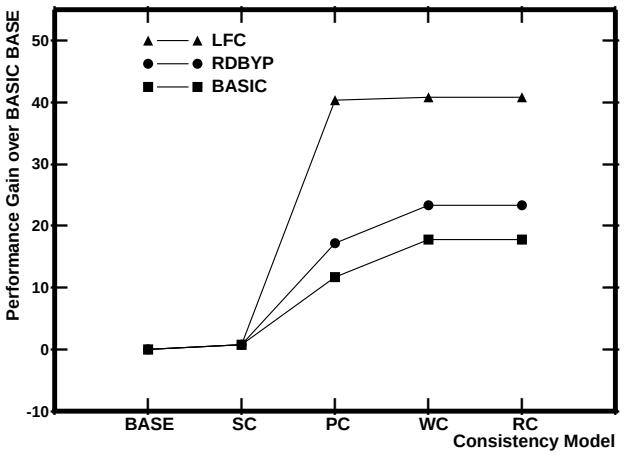

Figure 7: Performance of MP3D under LFC, RDBYP, and BASIC implementations.

5.2 Effect of Implementation Variations

In the previous section, we evaluated the performance of the consistency models using an aggressive implementation with lockup-free caches. The goal was to minimize the influence of the implementation in the comparison of the models. In this section, we explore less aggressive implementations and study their impact on the relative performance of the models.

The three specific implementations we compare are: (i) LFC, the aggressive implementation studied so far with lockup-free caches, reads that bypass the write buffer, and a 16 word deep write buffer; (ii) RDBYP, which is the same as LFC, except caches are no longer lockup-free; and (iii) BASIC, which is the same as RDBYP, except that reads are not allowed to bypass the write buffer.

Figure 7 shows the relative performance of the models for each implementation for the MP3D application. Results for the other two applications indicated a similar trend. As before, we use processor utilization as the figure of merit for performance. The performance curves are normalized to that for the BASE model on the BASIC implementation. As shown, BASE and SC are not affected by these implementation changes since neither model can exploit lockup-free caches or the bypassing of reads. However, the performance of PC, WC, and RC experience a large decline as we move from LFC to RDBYP and a smaller decline as we go to BASIC. This result shows that lockup-free caches are essential for realizing the full potential of the less strict models. Below we analyze these results in more detail.

5.2.1 RDBYP Implementation

Comparing the LFC and RDBYP implementations, we see a large performance difference (about 20%) for MP3D under the PC, WC, and RC models. We examine the performance under PC model first. The only difference between LFC and RDBYP for PC is that LFC can have both a read miss and a write miss outstanding, while the RDBYP can have either a read miss or a write miss outstanding, but not both. Thus, in RDBYP, a read

Table 6: Average latency for a read miss (in cycles).

Architecture	*MP3D*	*LU*	*PTHOR*
RDBYP	106	92	91
LFC	89	86	74

miss that follows closely behind a write miss will have to wait until this previously initiated write access completes, which can be tens of cycles. The statistics for MP3D show that for over 40% of the read misses in the application, there is a write miss within 30 cycles before that read miss. In addition, under LFC, about 55% of the read misses were serviced while there was a write miss outstanding, hence the large performance difference between LFC and RDBYP.

The results for WC and RC also show a large decrease in performance if lockup-free caches are not used. The lockup-free cache provides two benefits for WC and RC: (i) a read miss is serviced right away regardless of previous write misses (same as the benefit for PC), and (ii) write misses are retired at a faster rate by the write buffer since the cache allows multiple outstanding accesses. As shown in the previous subsection, the pipelining of write misses was not effective in increasing the performance of MP3D. Detailed simulation results show that, under LFC, there was rarely more than three outstanding requests at any one time for either WC or RC. Thus, similar to PC, the gain for LFC comes primarily from the read access not having to stall while the cache is servicing the previous write miss. Table 6 shows the average latency of read misses for the two implementations under RC across all applications. The results show that, indeed, the latency of read misses is substantially reduced due to the lockup-free cache.

5.2.2 BASIC Implementation

Comparing the RDBYP and BASIC implementations, we see a relatively smaller decrease in performance. The obvious advantage of allowing bypassing is that the read miss does not have to wait for the write buffer to empty before being serviced. The performance gains from such bypassing depend on the clustering of write misses before the read miss—the greater the clustering the bigger the gains.

The detailed simulation statistics for MP3D show that although many read misses have a write miss that occurs shortly before them, the clustering of such write misses is small. Thus, the gains of moving from BASIC to RDBYP are small. Finally, the reason the performance of PC is better than that of SC for BASIC is the following. In SC, both first-level cache read hits and read misses are delayed for the write buffer to empty out. In PC, read hits in the first-level cache are not delayed for pending writes and only first-level read misses suffer that penalty.

6 Effect of Prefetching on Consistency Models

The results presented so far in this paper are based on an architecture with blocking reads. While the less strict consistency models are successful in hiding the latency of writes in such an architecture, nothing is done to alleviate the latency of reads. Consequently, we see in Figure 4 that a large percentage of the idle cycles are due to read misses, especially when PC or RC models are used. In this section, we explore the effects of reduced read latency on the relative performance of the models. In particular, we model a prefetch mechanism similar to that provided in the DASH architecture [11].

In general, there are two types of prefetching: binding and non-binding. In binding prefetch, the value of the access is bound at the time the prefetch is completed. In contrast, a non-binding prefetch simply brings a copy of the location to a higher level in the memory hierarchy (closer to the processor). With a non-binding prefetch, the location is still accessible to the coherency mechanism and is kept coherent until the processor actually binds the value through a regular (binding) access. While binding prefetching interacts with the consistency models, non-binding prefetch does not affect the consistency models at all. Therefore, non-binding prefetch is more flexible, easier to use (does not affect correctness), and can provide benefits for all consistency models. Indeed, the prefetch mechanism provided in DASH is non-binding, and we will assume such a prefetch mechanism in this section. For a detailed evaluation of non-binding prefetching, see [14].

The results in this section show that the pipelining of writes allowed by RC (and WC) becomes significantly more important in achieving high performance once the latency of reads is reduced through prefetching. Section 6.1 presents simulation results for ideal prefetching of reads. Section 6.2 provides a more realistic assessment through a case study of one of the applications in which prefetching was added to the program.

6.1 Ideal Prefetching Results

We modeled the effects of ideal non-binding prefetch by artificially making read misses take only one cycle in the simulations (latency of writes and synchronization was not changed, however). Figure 8 shows the results of the simulation for the three applications. The performance of each model is normalized to that of BASE for each application.

Comparing the prefetching results to those presented in Figure 3, we see that the pipelining of writes in RC becomes much more important. This is shown dramatically in the case of MP3D. While PC performed comparably to RC with no prefetching, RC is shown to outperform PC significantly once prefetching is used. Detailed simulation results show that most of the idle cycles in PC are caused by stalls due to the write buffer being full. Simulation with a much deeper write buffer did not show much gain for PC since the rate of write misses was very large. PTHOR also shows RC performing better than PC once prefetching is used. Comparing the performance of PC and WC, PTHOR shows that pipelining of writes became more important and PC no longer outperforms WC (as in Figure 3). The results for LU show the same trend as the results in Figure 3, with the gains due to the less strict models being accentuated.

The results in this section pertain to prefetching for reads. In an invalidation-based coherence scheme, one can also do *read-exclusive* prefetches. The read-exclusive prefetch invalidates other copies of the location and brings an exclusively owned copy closer to the processor that issued the prefetch; this reduces the latency for both reads and writes. Given non-binding prefetching, this technique is applicable to all the models. Since the major performance difference from the models comes from their effectiveness in reducing and hiding write latency, the reduction in the latency of writes through prefetching will in general make the difference among the models less pronounced. Results on the effects of read prefetching and read-exclusive prefetching for MP3D are presented in the next subsection.

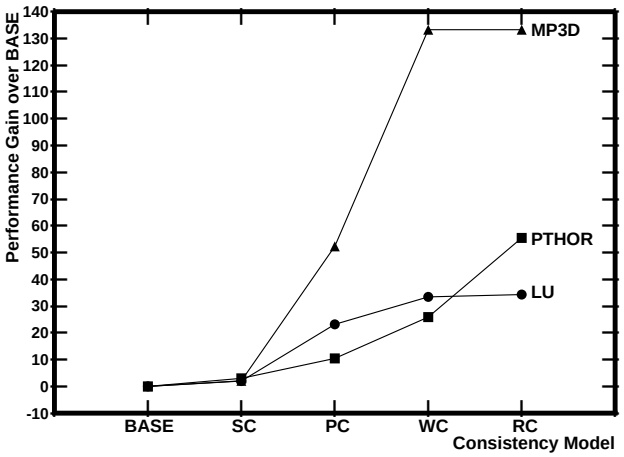

Figure 8: Results for ideal read prefetching on LFC.

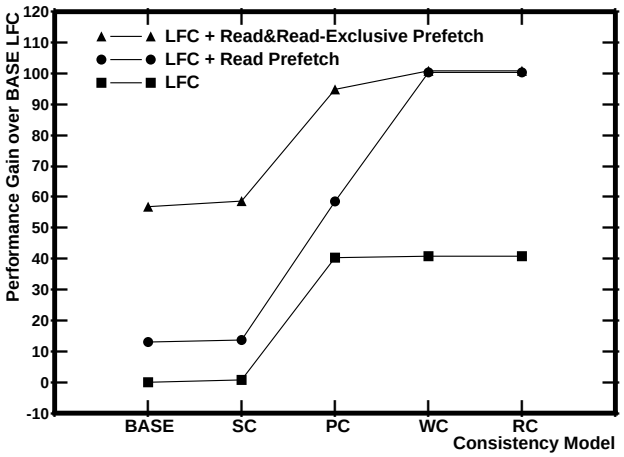

Figure 9: Read and read-exclusive prefetching in MP3D

6.2 Case Study for MP3D

In this subsection, we present results for two versions of MP3D, one with explicit read prefetching, and another with explicit read and read-exclusive prefetching. Prefetch instructions were inserted in appropriate places in the program. Prefetching within the simulator is modeled after the way prefetches work in DASH [11]; the prefetch brings a copy of the location into a special cache associated with each node. Figure 9 shows the relative performance of the models. The performance in each case is normalized to the performance of LFC with the BASE model. The results show that prefetching benefits the performance of all the models. The relative performance of the models in the case of read prefetching matches well with our results from ideal prefetching of reads. Clearly, the reduction in the latency of reads makes the latency due to write misses more critical and allows WC and RC to perform substantially better than PC. Simulation results show that there were occasionally five requests outstanding at the same time. However, once read-exclusive prefetching is done, the difference between PC and WC/RC diminishes considerably. Indeed, reducing the latency of write misses increases the rate at which the write buffer can retire writes, and thus reduces the chances for a full write buffer. An interesting observation is that read-exclusive prefetching does not help WC and RC over read prefetching (see top right of Figure 9). The reason is that WC and RC are hiding the latency of writes completely anyway, and therefore reducing the latency of writes offers no extra performance benefits.

In summary, read and read-exclusive prefetching substantially improve the performance of all models, including BASE and SC. We should note, however, that prefetching is not always successful in reducing the latency of accesses [14]. In cases where read prefetching is successful, we show that the pipelining of writes becomes more important. We expect read-exclusive prefetching to also be successful if read prefetching works. Read-exclusive prefetching is shown to diminish the importance of write pipelining, thus allowing PC to perform comparably to RC once again.

7 Discussion

In this section, we discuss the validity of the results presented in the paper in a broader context. In addition, we briefly point out other issues and tradeoffs concerning consistency models.

The results presented in this paper were based on a specific set of architectural parameters and a limited set of benchmarks. However, we expect many of the conclusions to hold in a broader context. One of the major conclusions is that, given reads are blocking, PC performs almost as well as RC. To achieve this, PC relies on the write buffer not getting full. Intuitively, if write misses are distributed evenly (as is the case for the applications studied), there is enough time for the write buffer to retire the writes given the processor stalls for read misses. More formally, the write buffer will not fill up if $rl_r > wl_w$, where r and w are the number of read and write misses with corresponding latencies of l_r and l_w, respectively.[4] Most applications, including the ones studied in this paper, have more read misses than write misses, especially because data is usually read before it is written. In addition, most architectures have comparable latencies for reads and writes. Therefore, PC is expected to perform comparably to RC over a wide range of architectures and applications.

This study has been primarily concerned about the gains that arise from relaxing consistency constraints at the hardware level. Consistency constraints can also affect performance at the software level, mainly through enabling compiler optimization for shared variables. For example, common compiler optimizations such as register allocation and common subexpression elimination involve changing the order of accesses in a program. More aggressive compiler optimizations like blocking of loops and register allocation of arrays also result in reordering of accesses. Under SC and PC, these optimizations can not be legally performed by the compiler. This can significantly degrade the performance of the system. In contrast, RC and WC provide considerable flexibility to the compiler for optimizing shared accesses and are thus desirable models. In fact, the gains achieved by doing such software optimizations can exceed the gains achievable in the hardware.

[4] This is a conservative condition since the time the processor is busy executing instruction or waiting for acquire synchronizations also contributes to the amount of time the write buffer has to retire writes.

Another issue concerning consistency models is the complexity of the model as presented to the programmer. There have been several efforts in specifying programming restrictions that result in the relaxed models being equivalent to sequential consistency as far as correctness is concerned [2, 6]. In practice, we have found that satisfying such restrictions is not a problem. Most user-level applications developed in our group already satisfy the restrictions with no change to the programs. Porting the Silicon Graphics IRIX operating system to DASH did not require more than a handful of simple changes either. Still more research is needed, however, to help the programmer in identifying and avoiding unwanted race conditions.

The simulation results presented in this paper are based on two major assumptions. Both assumptions limit the gain achievable from relaxing the consistency model. The first assumption is that the processors block on a read access. While the conditions for WC and RC allow multiple read accesses to be overlapped and pipelined, an implementation with blocking reads does not allow the latency of reads to be hidden in this manner. The design of processors that allow multiple outstanding reads and out-of-order execution of instructions and their effectiveness in hiding the latency of reads is a current topic of research.

The results in this paper are also dependent on the assumption that an invalidation-based coherence scheme is used. The tradeoffs for an update-based coherence scheme can be quite different. Although most programs exhibit a larger number of read misses than write misses in an invalidation-based scheme, it is likely that the number of read misses will decrease and write updates will substantially increase in an update-based scheme. This may in turn make the pipelining of writes more important to prevent the write buffer from filling up. To determine the tradeoffs described above will require a more in-depth study of such update-based schemes.

8 Related Work

In this section, we discuss some alternative implementations that have been proposed for sequential consistency and weak consistency and briefly describe previous evaluation efforts.

Adve and Hill [1] have proposed a very aggressive implementation for sequential consistency. Their scheme requires an invalidation-based cache coherence protocol. At points where our SC implementation stalls for the full latency of pending writes, their implementation stalls only until ownership is gained. To make the implementation satisfy sequential consistency, the new value written is not made visible to other processors until all previous writes by this processor have completed. Simulations show, however, that the latency of obtaining ownership is only slightly smaller than the latency for the write to complete. This is because the number of invalidations caused by a write is usually small [20]. Since the visibility-control mechanism reduces the stall time for SC only slightly, we still expect PC to perform significantly better than SC.

Adve and Hill [2] have also proposed an implementation for weak ordering that is less strict than WC. The constraints in their implementation are quite similar to the constraints imposed by RC. Therefore, the performance of their model is expected to be comparable to RC in practice, and thus not significantly better than PC.

Scheurich and Dubois [5, 17] provide simple analytical models to estimate the benefits arising from weak consistency and lockup-free caches. The gains predicted by these models are large, sometimes close to an order of magnitude gain in performance. We can not compare our results to this work, however, since their models are based on non-blocking reads, while our simulation results assume the processor stalls on reads.

Torrellas and Hennessy [19] present a more detailed analytical model of a multiprocessor architecture and estimate the effects of relaxing the consistency model on performance. A maximum gain of 20% in performance is predicted for using weak consistency over sequential consistency. This prediction is lower than the results in our study due to mainly two reasons: (i) the latencies assumed in [19] are lower than the ones in our simulated architecture; and (ii) the bus bandwidth assumed in their architecture became a limiting factor for cases with high sharing, where the weaker models could gain more otherwise.

9 Concluding Remarks

To enable multiprocessors to hide and reduce memory latency, a number of memory consistency models have been proposed in the literature. However, so far, no detailed performance results had been reported. In this paper, we have presented a simulation-based study that characterizes the performance of these consistency models.

Our results showed that, for architectures with blocking reads, the sequentially consistent models (BASE and SC) have significantly worse performance than the less strict models (PC, WC, and RC). For the three benchmark applications studied, the less strict models were shown to improve the processor utilization by as much as 10-40% over the BASE model. The gains are expected to increase with larger memory latencies that will be seen in future machines.

The paper further showed that most of the benefits achieved by the less strict models were due to buffering of writes and allowing reads to bypass pending writes. Lockup-free caches were shown to be essential for achieving the full potential of these models. The ability to pipeline writes was not as critical to performance, especially when reasonably deep write buffers were used. The surprising consequence was that processor consistency performed almost as well as the release consistency model. This was shown to be true under several architectural variations. Since processor consistency is simpler to implement in hardware, the results suggest that choosing processor consistency as the hardware model may be the most reasonable approach given current processors.

10 Acknowledgments

We thank Hank Levy and the reviewers for helpful comments. The following people also provided useful feedback on an earlier version of the paper: Sarita Adve, Phillip Gibbons, James Laudon, Christoph Scheurich, Per Stenstrom, Josep Torrellas, and Shigeru Urushibara. We wish to thank Todd Mowry for providing the Dixie simulator and for helping with the changes we needed to make to Dixie. The simulation results would not have been possible without his generous help. We thank the application writers for their applications: Larry Soule for PTHOR, Jeff McDonald for MP3D, Todd Mowry for MP3D with explicit prefetching, and Ed Rothberg for LU. We also thank Rohit Chandra for helpful discussions. This research was supported by DARPA contract N00014-87-K-0828. Kourosh Gharachorloo is partly supported by Texas Instruments and Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award.

References

[1] Sarita Adve and Mark Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 47–50, August 1990.

[2] Sarita Adve and Mark Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[4] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[5] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[6] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[7] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[8] James R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[9] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[11] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[12] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[13] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[14] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in scalable shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, to appear in June 1991.

[15] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[16] Christoph Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, May 1989.

[17] Christoph Scheurich and Michel Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 118–125, August 1988.

[18] Larry Soule and Anoop Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[19] Josep Torrellas and John Hennessy. Estimating the performance advantages of relaxing consistency in a shared-memory multiprocessor. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 26–33, August 1990.

[20] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.