Supercomputing '95

Lazy Release Consistency for Hardware-Coherent Multiprocessors [*]

Leonidas I. Kontothanassis, Michael L. Scott, and Ricardo Bianchini

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott,ricardo}@cs.rochester.edu

December 1994

Abstract

Release consistency is a widely accepted memory model for distributed shared memory systems. *Eager release consistency* represents the state of the art in release consistent protocols for hardware-coherent multiprocessors, while *lazy release consistency* has been shown to provide better performance for software distributed shared memory (DSM). Several of the optimizations performed by lazy protocols have the potential to improve the performance of hardware-coherent multiprocessors as well, but their complexity has precluded a hardware implementation. With the advent of programmable protocol processors it may become possible to use them after all. We present and evaluate a lazy release-consistent protocol suitable for machines with dedicated protocol processors. This protocol admits multiple concurrent writers, sends write notices concurrently with computation, and delays invalidations until *acquire* operations. We also consider a lazier protocol that delays sending write notices until *release* operations. Our results indicate that the first protocol outperforms eager release consistency by as much as 20% across a variety of applications. The lazier protocol, on the other hand, is unable to recoup its high synchronization overhead. This represents a qualitative shift from the DSM world, where lazier protocols always yield performance improvements. Based on our results, we conclude that machines with flexible hardware support for coherence should use protocols based on lazy release consistency, but in a less "aggressively lazy" form than is appropriate for DSM.

1 Introduction

Remote memory accesses experience long latencies in large shared-memory multiprocessors, and are one of the most serious impediments to good parallel program performance. Relaxed consistency models [6, 11] can help reduce the cost of memory accesses by masking the latency of write operations. Relaxed consistency requires that memory be consistent only at certain synchronization events, and thus allows a protocol to buffer, merge, and pipeline write requests as long as it respects the consistency constraints specified in the model.

[*]This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724, ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930), and Brazilian CAPES and NUTES/UFRJ fellowships.

Release consistency [10] is the most widely accepted relaxed consistency model. Under release consistency each memory access is classified as an ordinary access, an *acquire*, or a *release*. A release indicates that the processor is completing an operation on which other processors may depend; all of the releasing processor's previous writes must be made visible to any processor that performs a subsequent acquire. An acquire indicates that the processor is beginning an operation that may depend on some other processor; all other processors' writes must now be made locally visible.

This definition of release consistency provides considerable flexibility to a coherence protocol designer as to when to make writes by a processor visible to other processors. Hardware implementations of release consistency, as in the DASH multiprocessor [18], take an eager approach: write operations trigger coherence transactions (e.g., invalidations) immediately, though the transactions execute concurrently with continued execution of the application. The processor stalls only if its write buffer overflows, or if it reaches a release operation and some of its previous transactions have yet to be completed. This approach attempts to mask the latency of writes by allowing them to take place in the background of regular computation.

The better software coherence protocols adopt a *lazier* approach for distributed shared memory (DSM) emulation, delaying coherence transactions further, in an attempt to reduce the total number of messages exchanged. A processor in Munin [4], for example, buffers all of the "write notices" associated with a particular critical section and sends them when it reaches a release point. ParaNet (Treadmarks) [14] goes further: rather than send write notices to all potentially interested processors at the time of a release, it keeps records that allow it to inform an acquiring processor of all (and only) those write notices that are in the logical past of the releaser but not (yet) in the logical past of the acquirer.

Postponing coherence transactions allows a protocol to combine messages between a given pair of processors and to avoid many of the useless invalidations caused by false sharing [8]. Keleher et al. have shown these optimizations to be of significant benefit in their implementation of lazy release consistency [13] for DSM systems. Ideally, one might hope to achieve similar benefits for hardware-coherent systems. The sheer complexity of lazy protocols, however, has heretofore precluded their implementation in hardware. Several research groups, however, are now developing programmable protocol processors [16, 20] for which the complexity of lazy release consistency may be manageable. What remains is to determine whether laziness will be profitable in these sorts of systems, and if so to devise a protocol that provides the best possible performance.

In this paper we present a protocol that combines the most desirable aspects of lazy release consistency (reducing memory latency by avoiding unnecessary invalidations) with those of eager release consistency (reducing synchronization waits by executing coherence operations in the background). This protocol supports multiple concurrent writers, overlaps the transfer of write notices with computation, and delays invalidations until *acquire* operations. It outperforms eager release consistency by up to 20% on a variety of applications.

We also consider a lazier protocol that delays sending write notices until *release* operations. Our results indicate, however, that this lazier protocol actually hurts overall program performance, since its reduction of memory access latency does not compensate for an increased synchronization overhead. This result reveals a qualitative difference between software and hardware distributed shared-memory multiprocessors: delaying coherence operations as much as possible is appropriate for DSM systems, but not for hardware-assisted coherence.

The rest of the paper is organized as follows. Section 2 describes our lazy protocol, together with the lazier variant that delays the sending of write notices. Section 3 describes our experimental methodology and application suite. Section 4 presents results. It begins with a discussion of the sharing patterns exhibited by the applications, and proceeds to compare the performance of our lazy protocols to that of an eager release consistency protocol similar to the one implemented in the DASH multiprocessor. Finally, it describes the impact of architectural trends on the relative performance of the protocols. We present related work in section 5 and conclude in section 6.

2 A Lazy Protocol for Hardware-Supported Coherence

Our lazy protocol for hardware coherent multiprocessors resembles the software-based protocol described in [15], but has been modified significantly to exploit the ability to overlap coherence management and computation and to deal with the fact that coherence blocks can now be evicted from a processor's cache due to capacity or conflict misses. The basic concept behind the protocol is to allow processors to continue referencing cache blocks that have been written by other processors. Although write notices are sent as soon as a processor writes a shared block, invalidations occur only at acquire operations; this is sufficient to ensure that true sharing dependencies are observed.

The protocol employs a distributed directory to maintain caching information about cache blocks. The directory entry for a block resides at the block's *home node*—the node whose main memory contains the block's page. The directory entry contains a set of status bits that describe the state of the block. This state can be one of the following.

Uncached – No processor has a copy of this block. This is the initial state of all cache blocks.

Shared – One or more processors are caching this block but none has attempted to write it.

Dirty – A single processor is caching this block and is also writing it.

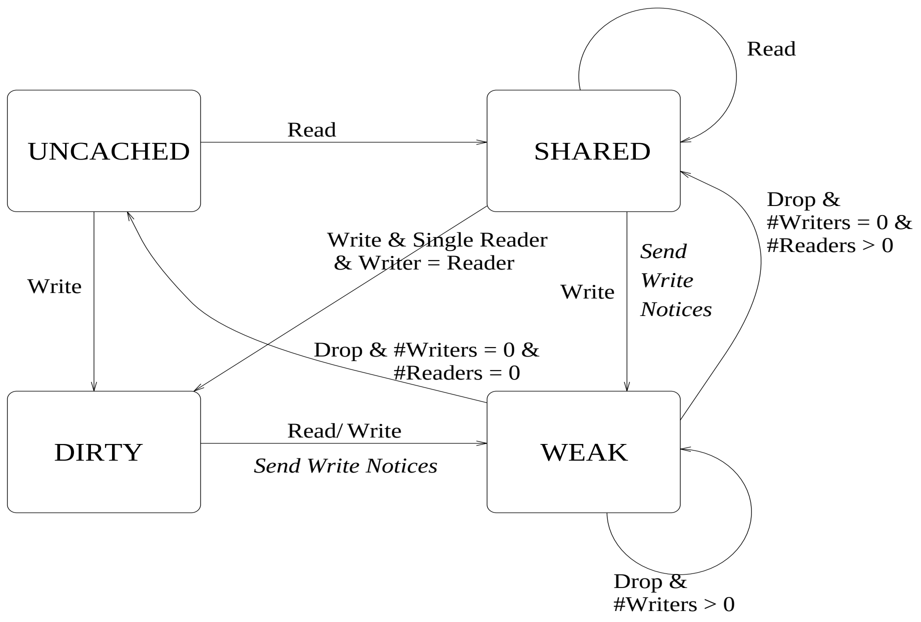
Weak – Two or more processors are caching this block and at least one of them is writing it.

In addition to the block's status bits, the directory entry contains a list of pointers to the processors that are sharing the block. Each pointer is augmented with two additional bits, one to indicate whether the processor is also writing the block, and the other to indicate whether the processor has been notified that the block has entered the weak state. To simplify directory operations two additional counters are maintained in a directory entry: the number of processors sharing the block, and the number of processors writing it. Figure 1 shows the directory state transition diagram for the original version of the protocol. Text in italics indicates additional operations that accompany the transition.

The state described above is a global property associated with a block, *not* a local property of the copy of the block in some particular processor's cache. There is also a notion of state associated with each line in a local cache, but it plays a relatively minor role in the protocol. Specifically, this latter, local state indicates whether a line is invalid, read-only, or read-write; it allows us to detect the initial access by a processor that triggers a coherence transaction (i.e. read or write on an invalid line, or a write on a read-only line). An additional local data structure is maintained by the protocol processor; it describes the lines that should be invalidated at the next acquire operation. The size of this data structure is upper bounded by the number of lines in the cache. There is no need to maintain such information for lines that have been dropped from the cache.

On a read miss by a processor the node's protocol processor allocates an "outstanding transaction" data structure that contains the line (block) number causing the miss. The outstanding transaction data structure is the equivalent of a RAC entry in the DASH distributed directory protocol [17]. It then sends a message to this block's home node asking for the data. When the request reaches the home node, the protocol processor issues a memory read for the block, and then starts a directory operation—reading the current state of the block and computing a new state. As soon as the memory returns the requested block, the protocol processor sends a message to the requesting node containing the data and the new state of the block. If the block has made the transition to the weak state an additional message is sent to the current writer.[1] It is worth noting that the protocol never requires the home node to forward a read request. If the block is not currently being written,

[1]The only situation in which a block can move to the weak state as a result of a read request is if it is currently in the dirty state (i.e. it has a single writer).



UNCACHED
SHARED
DIRTY
WEAK
Read
Read
Write
Write & Single Reader & Writer = Reader
Write
Send Write Notices
Drop & #Writers = 0 & #Readers > 0
Drop & #Writers = 0 & #Readers = 0
Read/ Write
Send Write Notices
Drop & #Writers > 0

Figure 1: Directory state diagram for a variant of lazy release consistency

then the memory module contains the most up-to-date version. If it is being written, then the fact that the read occurred indicates that no synchronization operation separates the write from the read. This in turn implies (in a correctly synchronized program) that true sharing is not occurring, so the most recent version of the block is not required.

Writes are placed in the write buffer and the main processor continues execution, assuming the write buffer is not full. If the write buffer accesses a missing cache line, the protocol processor allocates an outstanding transaction data structure and sends a write request message to the home node. If the block was not present in the processor's cache (the local line state was invalid), then the entry in the write buffer cannot be retired until the block's data is returned by the home node. If the block was read-only in the processor's cache, however, we still need to contact the home node and inform it of the write operation, but we do not need to wait for the home node's response before retiring the write buffer entry. This stems from the fact that we allow a block to have multiple concurrent writers; we do not need to use the home node as a serializing point to choose a unique processor as writer.

When the write request arrives at the home node, the home node's protocol processor consults the directory entry to decide what the new state of the block should be. If the new state does not require additional coherence messages (i.e. the block was uncached, or cached only by the requesting processor) then an acknowledgment can be sent to the requesting processor. However if the block is going to make a transition to the weak state then notification messages must be sent to the other sharing processors. A response is sent to the requesting processor, instructing it to wait for the collection of acknowledgements. Acknowledgements could be directed to, and collected by, either the requesting processor or the home node (which would then forward a single acknowledgement to the requesting node). We opted for the second approach. It has lower complexity and it allows us to collect acknowledgments only once when write requests for the same block arrive from multiple processors. The home node keeps track of the write requests and acknowledges all of them when it has received the individual acknowledgments from all of the sharing processors.

Lock releases need to make sure that all writes by the releasing processor have globally performed, i.e. that all processors with copies of written blocks have been informed of the writes, and that written data has made its way back to main memory. We ensure this by stalling the processor until (1) its write buffer has been flushed, (2) its outstanding requests have been serviced (i.e. all outstanding request data structures have been deallocated), and (3) memory has acknowledged any outstanding write-backs or write-throughs (see below).

Lock acquires need to invalidate all lines in the acquiring processor's cache for which write notices have been received. Much of the latency of this operation can be hidden behind the latency of the lock acquisition itself. When a processor attempts to acquire a lock its protocol processor performs invalidations for any write notices that have already been received. When it receives a message granting ownership of the lock, the protocol processor performs invalidations for any additional notices received in the intervening time. Invalidating a line involves notifying the home node that the local processor is no longer caching the block. This way the home node can update the state of the block in the directory entry appropriately. If a block no longer has any processors writing it, it reverts to the shared state; if it has no processors sharing it at all, it reverts to the uncached state. If a block is evicted from a cache due to a conflict or capacity miss, the home node must also be informed.

One last issue that needs to be addressed is the mechanism whereby data makes its way back into main memory. With a multiple-writer protocol, a write-back cache requires the ability to merge writes to the same cache block by multiple processors. Assuming that there is no false sharing within individual words, this could be achieved by including per-word dirty bits in every cache, and by sending these bits in every write-back message. This approach complicates the design of the cache, however, and introduces potentially large delays at release operations due to the cache flush operations. A write-through cache can solve both these problem by providing word granularity for the memory updates and by overlapping memory updates with computation. For most programs, however, write-through leads to unacceptably large amounts of traffic, delaying critical operations like cache fills. A coalescing fully associative buffer [12] placed after the write-through cache can effectively combine the best attributes of both write strategies. It provides the simple design and low release synchronization costs of the write-through cache, while maintaining data traffic levels comparable to those of a write-back cache [15].

We also consider a lazier version of the protocol that attempts to delay the point at which write notices are sent to other processors. Under this protocol, the node's protocol processor will refrain from sending a write request to a block's home node as long as possible. Notification is sent either when a written block is replaced in a processor's cache, or when the processor performs a release operation. Writes are buffered in a local data structure maintained by the protocol processor. Processing writes for replaced blocks allows us to place an upper bound on the size of this data structure (proportional to the size of the processor's cache) and to avoid complications in directory processing that arise from having to process writes from processor's that may no longer be caching a block. Delaying notices has been shown to improve the performance of software coherent systems [4, 15]. In a hardware implementation, however, delayed notices do not take full advantage of the asynchrony in computation and coherence management and can cause significant delays at synchronization operations.

3 Experimental Methodology

We use execution-driven simulation to simulate a mesh connected multiprocessor with up to 64 nodes. Our simulator consists of two parts: a front end, Mint [23], that simulates the execution of the processors, and a back end that simulates the memory system. The front end is the same in all our experiments. It implements the MIPS II instruction set. Our back end is quite detailed, with finite-size caches, full protocol emulation, distance-dependent network delays, and memory access

System Constant Name	Default Value
Cache line size	128 bytes
Cache size	128 Kbytes direct-mapped
Memory setup time	20 cycles
Memory bandwidth	2 bytes/cycle
Bus bandwidth	2 bytes/cycle
Network bandwidth	2bytes/cycle (bidirectional)
Switch node latency	2 cycles
Wire latency	1 cycle
Write Notice Processing	4 cycles
LRC Directory access cost	25 cycles
ERC Directory access cost	15 cycles

Table 1: Default values for system parameters

costs (including memory contention). Our simulator is capable of capturing contention within the network, but only at a substantial cost in execution time; the results reported here model network contention at the sending and receiving nodes of a message, but not at the nodes in-between. We have also simplified our simulation of the programmable protocol processor, abstracting away such details as the instruction and data cache misses that it may suffer when processing protocol requests. We believe that this inaccuracy does not detract from our conclusions. Current designs for protocol processors incorporate very large caches with a negligible miss rate for all but a few pathological cases [16]. In our simulations we simply charge fixed costs for all operations. The one exception is a write request to a shared line, where the cost is the sum of the directory access, and the dispatch of messages to the sharing processors. Since in most cases directory processing can be hidden behind the memory access cost, the increased directory processing cost of the lazy protocol does not affect performance. Table 1 summarizes the default parameters used in our simulations.

Using these parameters and ignoring any contention effects that may be seen at the network or memory modules, a cache fill would incur the cost of a) sending the request message to the home node through the network, b) waiting for memory to respond with the data, c) sending the data back to the requesting node through the network, and d) satisfying the fill through the node's local bus. Assuming a distance of 10 hops in the network the cost of sending the request is $(2 + 1) * 10 = 30$ cycles, the cost of memory is $20 + 128/2 = 84$ cycles, the cost of sending the data back is $(2 + 1) * 10 + 128/2 = 94$ cycles, and the cost of the local cache fill via the node's bus is $128/2 = 64$. The aggregate cost for the cache fill is then $(a + b + c + d) = 30 + 84 + 94 + 64 = 272$ processor cycles.

We report results for 7 parallel programs. We have run each program on the largest input size that could be simulated in a reasonable amount of time and that provided good load-balancing for a 64-processor configuration. Three of the programs are best described as computational kernels: `Gauss`, `fft`, and `blu`. The rest are complete applications: `barnes-hut`, `cholesky`, `locusroute`, and `mp3d`.

`Gauss` performs Gaussian elimination without pivoting on a 448×448 matrix. `Fft` computes a one-dimensional FFT on a 65536-element array of complex numbers, using the algorithm described. `Blu` is an implementation of the blocked right-looking LU decomposition algorithm presented in [5] on a 448×448 matrix. `Barnes-Hut` is an N-body application that simulates the evolution of 4K bodies under the influence of gravitational forces for 4 time steps. `Cholesky` performs Cholesky factorization on a sparse matrix using the `bcsstk15` matrix as input. `Locusroute` is a VLSI standard cell router using the circuit Primary2.grin containing 3029 wires. `Mp3d` is a wind-tunnel airflow simulation of 40000 particles for 10 steps. All of these applications are part of the Splash suite [21]. Due to simulation constraints our input data sizes for all programs are smaller than what would be run on a real machine. As a consequence we have also chosen smaller caches than are common on real machines, in order to capture the effect of capacity and conflict misses. Experiments with larger cache sizes overestimate the advantages of lazy release consistency, by eliminating a significant fraction of the misses common to both eager and lazy protocols.

4 Results

Our principal goal is to determine the performance advantage that can be derived on hardware systems with a lazy release consistent protocol. To that end we begin in section 4.1 by categorizing the misses suffered by the various applications under eager release consistency. If false sharing is an important part of an application's miss rate then we can expect a lazy protocol to realize substantial performance gains. We continue in section 4.2 by comparing the performance of our lazy protocol to that of eager release consistency. Section 4.3 then evaluates the performance implications of the lazier variant of our protocol. This section provides intuition on the qualitative differences between software and hardware implementations of lazy release consistency.

4.1 Application Characteristics

As mentioned in section 2, the main benefits of the lazy protocols stem from reductions in false sharing, and elimination of write buffer stall time when data is already cached read-only. In an attempt to identify the extent to which these benefits might be realized in our application programs, we have run a set of simulations to classify their misses under eager release consistency. Applications that have a high percentage of false sharing, or that frequently write miss on read-only blocks will provide the best candidates for performance improvements under lazy consistency.

Table 2 presents the classification of the eager protocol's miss rate into the following components: Cold misses, True-sharing misses, False-sharing misses, Eviction misses, and Write misses. The individual categories are presented as percentages of the total number of misses. The classification scheme used is described in detail in [3]. Write misses are of a slightly different flavor from the other categories: they do not result in data transfers, since they occur when a block is already present in the cache but the processor does not have permission to write it. As can be seen from the table, the applications with a significant false miss rate component and the ones that we would expect to see performance improvements for the lazy protocol are `barnes-hut`, `blocked-lu`, `locus-route`, and `mp3d`. The remaining applications (`cholesky`, `fft`, and `gauss`) should realize no gains in the lazy protocol since they have almost no false sharing. We have decided to evaluate them to examine whether the lazy protocol is detrimental to performance for applications without false sharing.

4.2 Lazy v. Eager Release Consistency

This section compares our lazy release consistency protocol (presented in section 2) to an eager release consistency protocol like the one implemented in DASH [17]. The performance of a sequentially consistent directory-based protocol is also presented for comparison purposes. The relaxed consistency protocols use a 4-entry write buffer which allows reads to bypass writes and coalesces writes to the same cache line. The eager protocol uses a write-back policy while the lazy protocol uses write-through with a 16-entry coalescing buffer placed between the cache and the memory system.

Table 3 presents the miss rates of our applications under the different protocols. In all cases the lazy variants exhibit the same or lower miss rate than the eager implementation of release

Application	Cold	True	False	Eviction	Write
Barnes-Hut	6.9%	9.0%	11.4%	62.9%	9.7%
Blocked-LU	8.6%	24.7%	24.1%	12.7%	29.8%
Cholesky	26.1%	5.9%	1.6%	28.0%	38.2%
Fft	13.3%	1.0%	0.0%	54.0%	31.7%
Gauss	7.5%	0.2%	0.1%	75%	17.1%
Locusroute	6.1%	13.0%	33.0%	15.6%	32.3%
Mp3d	3.1%	31.1%	5.7%	13.5%	46.5%

Figure 2: Classification of misses under eager release consistency

Application	Miss Rate		
	Eager	Lazy	Lazy-ext
Barnes-Hut	0.43%	0.41%	0.40%
Blocked-LU	2.08%	1.94%	1.45%
Cholesky	1.24%	1.24%	1.24%
Fft	0.47%	0.47%	0.47%
Gauss	2.72%	2.72%	2.33%
Locusroute	1.86%	1.24%	1.02%
Mp3d	4.81%	3.78%	2.57%

Figure 3: Miss rates for the different implementations of release consistency

consistency. For the applications with an important false sharing miss rate component, miss rates are reduced, while for the remaining applications miss rate remains the same.

Figure 4 presents the normalized execution time of the different protocols on our application suite. Execution time is normalized with respect to the execution time of the sequentially consistent protocol (the unit line in the graph). The lazy protocol provides a performance advantage on the expected applications, with the advantage ranging from 5% to 17%. The application with the largest performance improvement is `mp3d`. `Mp3d` has the highest overall miss rate, with false sharing and write misses being important components of it. `Barnes-hut`'s performance also improves by 9% when using a lazy protocol, but unlike all the remaining programs the performance benefits are derived from a decrease in synchronization wait time. Closer study reveals that this decrease stems from better handling of migratory data in the lazy protocol.

`Blocked LU` and `Locusroute` suffer from false sharing and the lazy nature of the protocol allows them to tolerate it much better than eager release consistency, resulting in performance benefits of 5% and 13% respectively. `Gauss` on the other hand has no false sharing, no migratory data, and still realizes performance improvements of 9% under lazy consistency. We have studied the program and have found that the performance advantage of lazy consistency stems from the elimination of 3-hop transactions in the coherence protocol. Sharing in `gauss` occurs when processors attempt to access a newly produced pivot row which is in the dirty state. Furthermore this access is tightly synchronized and has the potential to generate large amounts of contention. The lazy protocol eliminates the need for the extra hop and reduces the observed contention, thus improving performance. One could argue that the eager protocol could also use the write-through policy and realize the same benefits. However this would be detrimental to the performance of other applications. For the lazy protocol, write-through is necessary for correctness purposes.

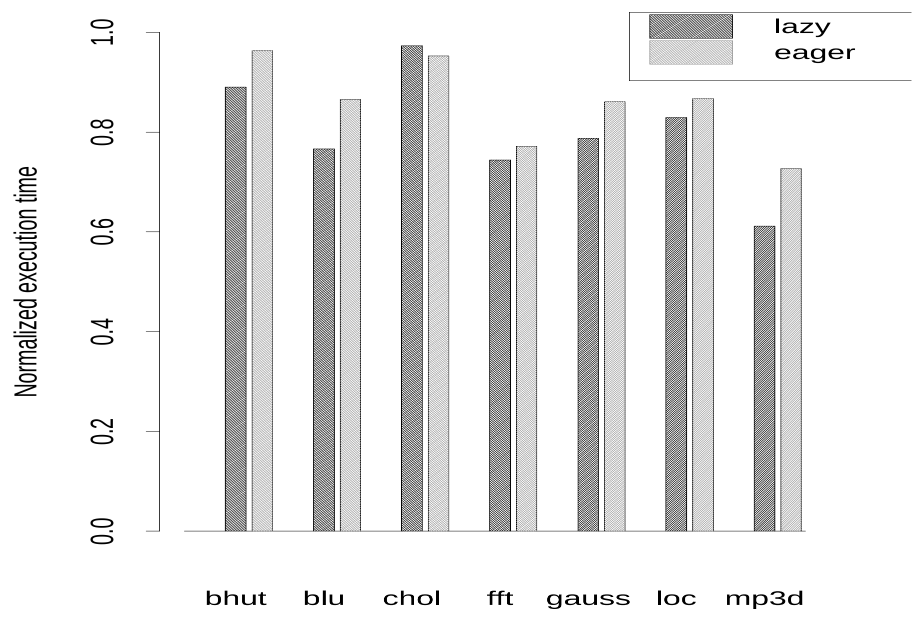


Figure 4: Normalized execution time for lazy-release and eager-release consistency on 64 processors

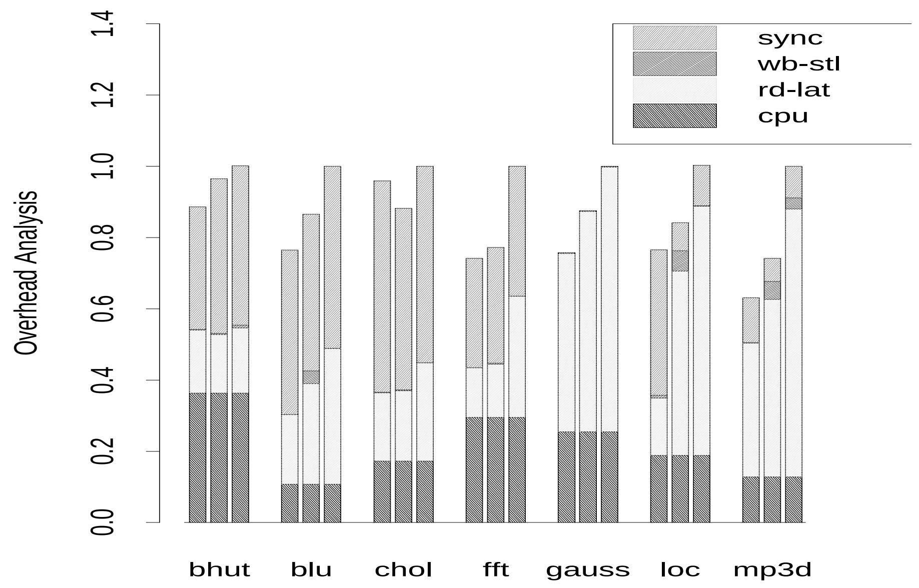


Figure 5: Overhead analysis for lazy-release, eager-release, and sequential consistency (left to right) on 64 processors

`Cholesky` and `fft` have a very small amount of false sharing. Their performance changes little under the lazy protocol: `fft` runs a little faster; `cholesky` runs a little slower.

Figure 5 presents a breakdown of aggregate cycles (over all processors) into four categories: cycles spent in cpu processing, cycles spent waiting for read requests to return from main memory, cycles lost due to write-buffer stalls, and cycles lost to synchronization delays. Costs for each category in each protocol are presented as a percentage of the total costs experienced by the sequentially consistent protocol. Results indicate that the lazy consistency protocol reduces read latency and write buffer stalls, but has increased synchronization overhead. For all but one of the programs the decrease in read latency is sufficient to offset the increase in synchronization time, resulting in net performance gains.

Two of our application `mp3d` and `locus-route` do not obey the release consistency model (they have unsynchronized references). It is possible that the additional time before a line is invalidated may hurt the quality of solution in the lazy protocol. To qualify this effect we have experimented with two versions of mp3d running natively on our SGI. One version uses software caching to capture the behavior of the lazy protocol in data propagation while the other version captures the behavior of a *sequentially* consistent protocol (that of our SGI). We have compared the cumulative (over all particles) velocity vector after 10 time steps for the two programs. We found that the Y and Z coordinates of the velocity vector were less than one tenth of a percent apart while the X coordinate was 6.7% apart between the two versions.

We believe that for properly synchronized programs with false sharing the lazy protocol will provide an important performance advantage. The same is true for programs with data races whose quality of solution is not affected by the additional delay in invalidations. For the remaining programs (which may not be suitable for relaxed consistency models in the first place) the lazy protocol can match the performance of the eager protocol simply by adding fence operations in the code that would force the protocol processor to process invalidations at regular intervals.

4.3 How Much Laziness is Required?

Unlike the basic lazy protocol we have evaluated so far, software-coherent systems implementing lazy release consistency attempt to further postpone the processing of writes to shared data, by combining writes and processing them at synchronization release points. This "aggressive laziness" allows unrelated acquire synchronization operations to proceed without having to invalidate cache lines that have been modified by the releasing processor. As a result programs experience reduced miss rates and reduced miss latencies. However, moving the processing of write operations to release points has the side effect of increasing the amount of time a processor spends waiting for the release to complete. For software systems this is not usually a problem, since write notices cannot be processed in parallel with computation, and the same penalty has to be paid regardless of when they are processed. On systems with hardware support for coherence, however, the coherence overhead associated with writes can be overlapped with the program's computation, so long as the program has something productive to do. The aggressively lazy protocol effectively eliminates this overlap, and can end up hurting performance due to increased synchronization costs.

Figure 6 shows the normalized execution times of our original lazy protocol and its lazier variant. We will refer to the lazier protocol from now on as `lazy-ext`. The analysis of overheads for the two versions of the protocols is shown in figure 7. As in the previous section normalization is done with respect to the run time and the overheads experienced by a sequentially consistent protocol.

For all but one of the applications the lazier version of the protocol has poorer overall performance. This finding stands in contrast to previously-reported results for DSM systems [13], and is explained by the ability to overlap the processing of non-delayed write notices. As can be seen from figure 7, the `lazy-ext` protocol improves the miss latency experienced by the programs, but

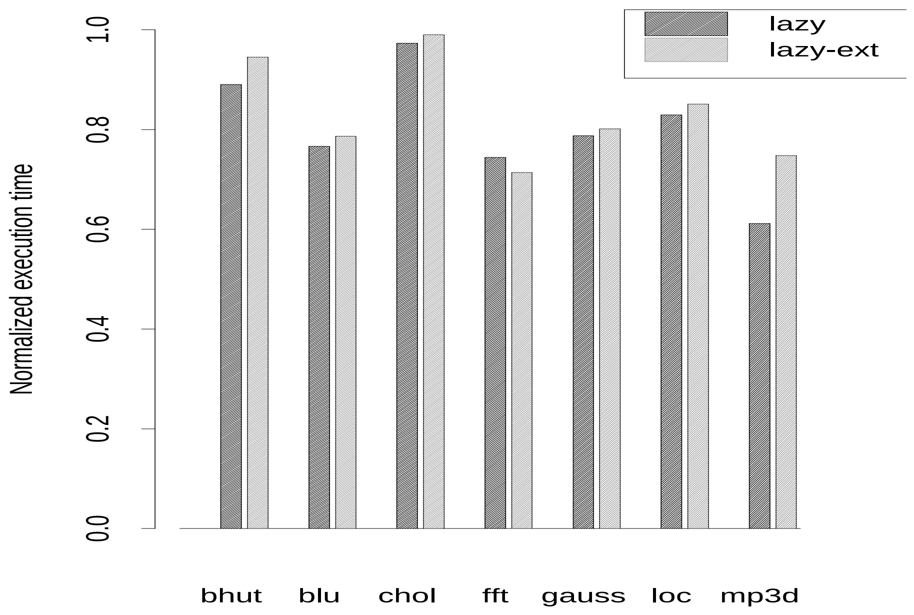


Figure 6: Normalized execution time for lazy and lazy-extended consistency on 64 processors

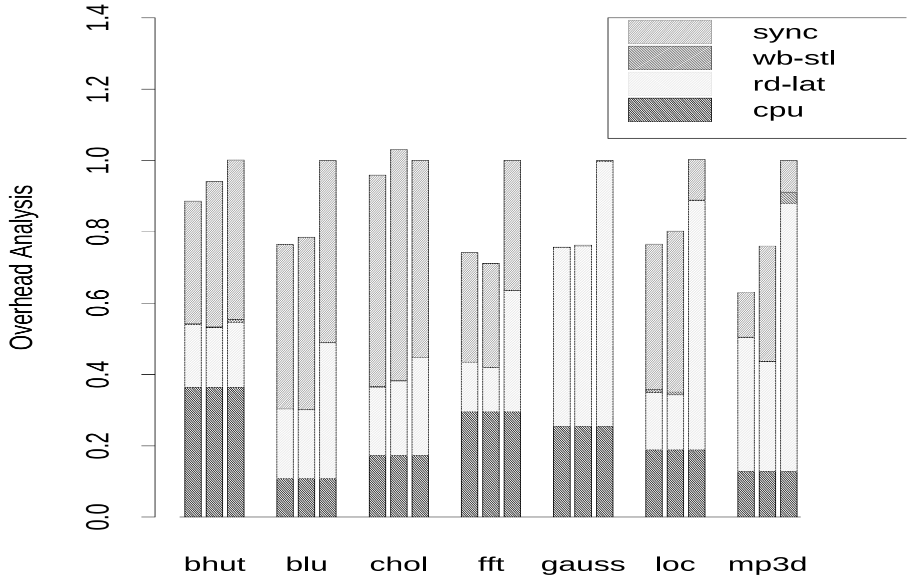


Figure 7: Overhead analysis for lazy, lazy-extended, and sequential consistency (left to right) on 64 processors

increases the amount of time spent waiting for synchronization. The former is insufficient to offset the latter, resulting in program performance degradation. The exception to this observation is `fft`. `Fft` computes a 1-D FFT in phases, separated by a barrier. Delaying the processing of writes allows home nodes to combine the processing of write requests to the same block, since these requests arrive more-or-less simultaneously at the time of the barrier. There is no increase in synchronization time, and processors experience shorter delays between barriers since all events between barriers are local. The observed miss rate for the applications also agrees with the observed miss latency, and is lowest under the `lazy-ext` protocol.

We have also run experiment varying the latency, bandwidth, and cache line size parameters for our systems. We have found that as latency and bandwidth increase the performance gap between the lazy and eager protocols decreases, with the lazy protocol maintaining a modest performance advantage over all latency/bandwidth combinations. Varying the cache line size results in predictable behavior. Longer cache lines increase the performance gap between the lazy and eager protocols since the induce higher degrees of false sharing. While the trend toward increasing block sizes is unlikely to go on forever [2], 256-byte cache blocks seem plausible for the near-term future, suggesting that lazy protocols will become increasingly important. A performance comparison among the lazy and eager protocols for a future hypothetical machine with high latency (40 cycles memory startup), high bandwidth (4bytes/cycle), and long cache lines (256 bytes) can be seen in figure 8. Lazy release consistency can be seen to outperform the eager alternative for all applications. In the applications where lazy release consistency was important in our earlier experiments, the performance gap has increased even further. In `mp3d` the performance gap has widened by an additional 6% over the original experiment; lazy consistency outperforms the eager variant by 23%. Similar gains were achieved by the other applications as well. The performance gap between lazy and eager release consistency has increased by 2 to 4 percentage points when compared with the performance difference seen in the original experiments. The observations made for the earlier overhead breakdown graphs continue to apply. As can be seen in figure 9 the lazy protocols trade increased synchronization time for decreased read latency and write buffer stall time. The additional advantage of laziness for this future machine stems from increases in cache line size and memory startup latency (as measured in processor cycles): longer lines increase the potential for false sharing, and increased memory startup costs increase the cost of servicing read misses.

5 Related Work

Our work builds on the research in programmable protocol processors being pioneered by the Stanford FLASH [16] and Wisconsin Typhoon [20] projects. In comparison to silicon implementations, dedicated but programmable protocol processors offer the opportunity to obtain significant performance improvements with no appreciable increase in hardware cost.

On the algorithmic side, our work bears resemblance to a number of systems that provide shared memory and coherence in software using a variant of lazy release consistency. Munin [4] collects all write notices from a processor and posts them when the processor reaches a synchronization release point. ParaNet (Treadmarks) [14] relaxes the Munin protocol further by postponing the posting of write notices until the subsequent acquire. Both Munin and ParaNet are designed to run on networks of workstations, with no hardware support for coherence.

Petersen and Li [19] have presented a lazy release consistent protocol for small scale multiprocessors with caches but without cache coherence. Their approach posts notices eagerly, using a centralized list of weak pages, but only processes notices at synchronization acquire points. The protocol presented in this paper is most closely related to a protocol developed for software coherence on large-scale NUMA multiprocessors [15]. Both protocols use the concept of write notices and of a distributed directory. Unlike the software protocol, however, the one presented here works

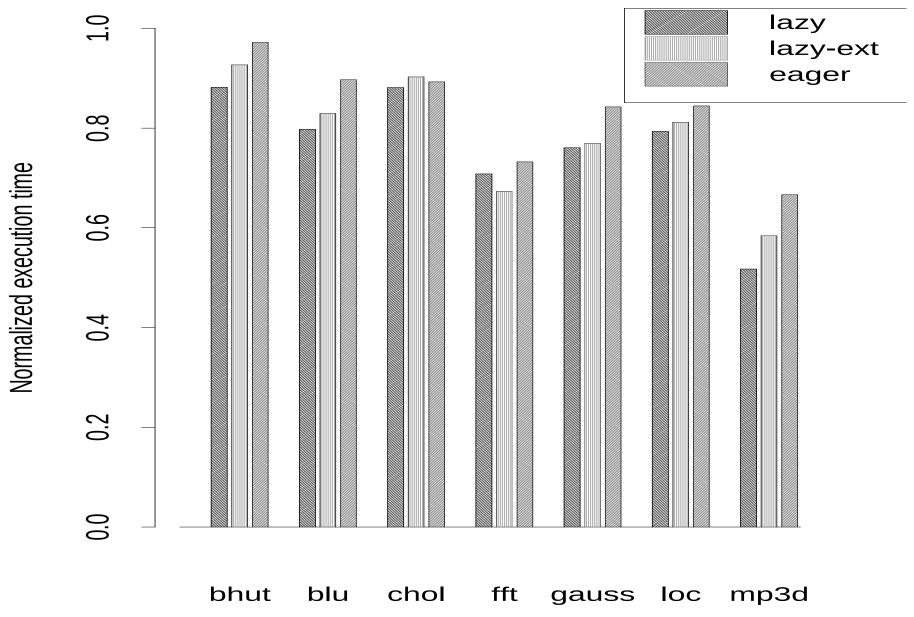


Figure 8: Performance trends for lazy, lazier, and eager release consistency (execution time)

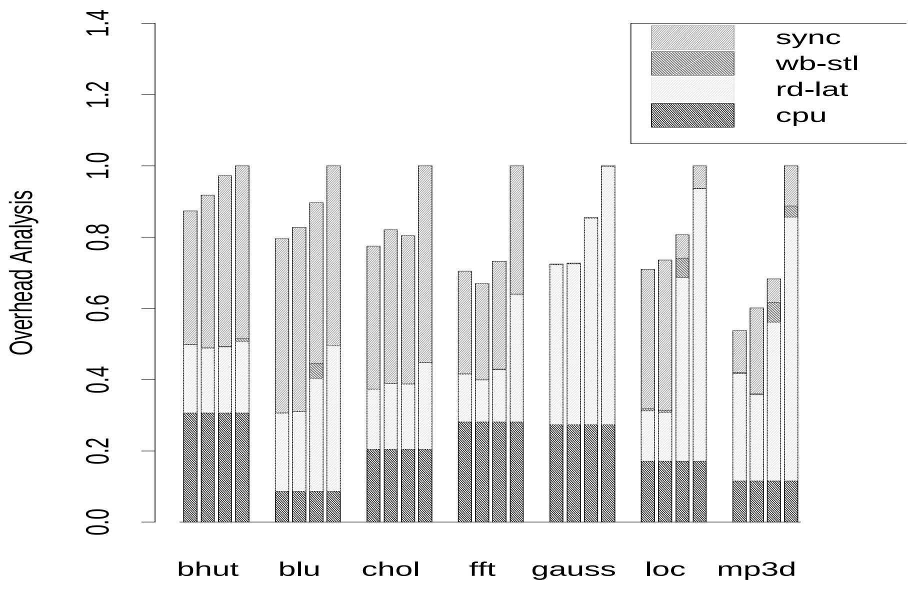


Figure 9: Performance trends for lazy, lazier, eager, and sequential consistency (left to right), (overhead analysis)

better when it does not postpone posting notices. It has also been modified to execute coherence operations and application code in parallel, and to deal with the fact that a processor can lose a block due to capacity and conflict evictions.

Our work is also similar to the delayed consistency protocol and invalidation scheduling work of Dubois et al. [7, 8]. Both protocols (ours and theirs) attempt to reduce the impact of false sharing in applications. Their work however assumes a single owner and requires a processor to obtain a new copy on write hits to a stale block.[2] Their protocol incurs longer delays for write accesses to falsely-shared blocks and increases the application's miss rate. Their experiments also assume infinite caches, which exaggerate the importance of coherence misses, and they use miss rate as the measure of performance. As we have shown in section 4.3, miss rate is only indicative of program performance and can sometimes be misleading. Excessively lazy protocols can actually hurt performance, even though they improve the application's miss rate.

The formalization of data-race-free-1 by Adve and Hill [1] allows the same optimizations as used in our protocol. We focus however on the hardware design and performance evaluation aspects of the protocol, while they concentrate in formally defining the behavior of the consistency model for different access patterns.

False sharing can be dealt with in software using compiler techniques [9]. Similarly, the latency of write misses on blocks that are already cached read-only can be reduced by a compiler that requests writable copies when appropriate [22]. These techniques are not always successful, however. The former must also be tuned to the architecture's block size, and the latter requires a load-exclusive instruction. We view our protocol as complementary to the work a compiler would do. If the compiler is successful then our protocol will incur little or no additional overhead over eager release consistency. If, however, the application still suffers from false sharing, or from a significant number of write-after-read delays, then lazy release consistency will yield significant performance improvements.

6 Conclusions

We have shown that adopting a lazy consistency protocol on hardware-coherent multiprocessors can provide substantial performance gains over the eager alternative on a variety of applications. For systems with programmable protocol processors, the lazy protocol requires only minimal additional hardware cost (basically storage space) with respect to eager release consistency. We have introduced two variants of lazy release consistency and have shown that on hardware-based systems, delaying coherence transactions helps only up to a point. Delaying invalidations until a synchronization acquire point is almost always beneficial, but delaying the posting of write notices until a synchronization release point tends to move background coherence operations into the critical path of the application, resulting in unacceptable synchronization overhead.

We have also conducted experiments in an attempt to evaluate the importance of lazy release consistency on future architectures. We find that as miss latencies and cache line sizes increase, the performance gap between lazy and eager release consistency increases as well. We are currently investigating the interaction of lazy hardware consistency with software techniques that reduce the amount of false sharing in applications. As program locality increases the performance advantage of lazy protocols will decrease, as a direct result of the decrease in coherence transactions required. Our results indicate, however, that lazy protocols can improve application performance even in the absence of false sharing, e.g. by replacing 3-hop transactions with 2-hop transactions, as in `Gauss`, or by eliminating write-buffer stalls due to write-after-read operations, as in `Barnes-Hut`. Moreover, since most parallel applications favor small cache lines while the current architectural

[2]Stale blocks are similar to weak blocks in our protocol.

trend is towards longer lines, we believe that lazy consistency will provide significant performance gains over eager release consistency for the foreseeable future.

Acknowledgements

We would like to thank Jack Veenstra for his help and support with this work.

References

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

[2] R. Bianchini and T. J. LeBlanc. Can High Bandwidth and Latency Justify Large Cache Blocks in Scalable Multiprocessors? In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, August 1994. Expanded version available as TR 486, Computer Science Department, University of Rochester, January 1994.

[3] R. Bianchini and L. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the Twenty-Eighth Annual Simulation Symposium*, Phoenix, AZ, April 1995. Earlier version available as TR 533, Computer Science Department, University of Rochester, September 1994.

[4] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, October 1991.

[5] K. Dackland, E. Elmroth, B. Kagstrom, and C. V. Loan. Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 VF/600J. *The International Journal of Supercomputer Applications*, 6(1):69–97, Spring 1992.

[6] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, 21(2):9–21, February 1988.

[7] M. Dubois, J. C. Wang, L. A. Barroso, K. L. Lee, and Y. Chen. Delayed Consistency and its Effect on the Miss Rate of Parallel Programs. In *Supercomputing'91 Proceedings*, pages 197–7206, Albuquerque, NM, November 1991.

[8] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[9] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:377–381, St. Charles, IL, August 1991.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[11] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[12] N. Jouppi. Cache Write Policies and Performance. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[14] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. ParaNet: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, San Francisco, CA, January 1994.

[15] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 286–295, Raleigh, NC, January 1995.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[19] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[20] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

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

[22] J. Skeppstedt and P. Stenstrom. Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 286–296, San Jose, CA, October 1994.

[23] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January – February 1994.