In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 194-203, February 1995.

Reducing the Communication Overhead of Dynamic Applications on Shared Memory Multiprocessors

Anand Sivasubramaniam

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
anand@cse.psu.edu

Abstract

Shared memory machines offer the convenience of a shared address space. This makes them particularly appealing for applications with dynamic communication behavior since the mechanisms for data transfer between processors is hidden from the programmer. But the scalability of these machines is often limited by the latencies incurred in accessing locations in remote memories. Caches alleviate this problem by exploiting the temporal and spatial locality in an application. However, the induced traffic for maintaining coherence can have a large impact on limiting performance. Invalidation-based protocols for coherence maintenance are conservative and always resort to receiver-initiated communication. Thus the receiver may have to experience the entire latency of the data transfer even though the data item may have been available much earlier. Update-based schemes, though sender-initiated, can incur high write overheads by sending redundant updates to processors that may not need them.

The goal of this research is to reduce the read and write latencies of applications with dynamic communication behavior by employing intelligent sender-initiated data transfer mechanisms. In the process, we would like to keep our demands from the programmer, the compiler, and the hardware as low as possible. Towards this goal, we present a set of write primitives that lower the communication overhead for shared memory accesses governed by locks. We demonstrate the performance benefits of these primitives using a database application drawn from the Geographical Information Systems (GIS) domain. We explore the competitive update mechanism for the remaining shared memory accesses. Using a set of applications, we examine the amount of history that we need to maintain for an effective competitive update scheme. We also show how this effective scheme can be implemented in software on emerging shared memory architectures with little hardware support.

1 Introduction

Shared memory multiprocessors offer the convenience of a shared address space making it easier to write parallel programs. But the scalability of such machines from the performance viewpoint is often limited by the communication overhead incurred in accessing locations in remote memories. Application knowledge and compile-time information can be used in some cases for intelligent data placement to statically place the data closer to the processors that need them most. Prefetch/poststore [17] and bulk data transfer mechanisms [27] can also help reduce/tolerate communication overheads. However, these techniques can work well for applications with known static communication behavior where the programmer or the compiler can glean sufficient information from the program to exploit these mechanisms/techniques. However, several applications fall in the dynamic category where the communication behavior varies with the program execution and cannot be predetermined. As observed by Kranz el al. [14], shared memory multiprocessors are particularly more appealing (than their message-passing counterparts) for dynamic applications.

Cache-Coherent Non-Uniform Memory Access (CC-NUMA) machines have become increasingly popular because of their ability to lower the probability of remote memory accesses by exploiting the spatial and temporal locality characteristics of the application. But caches introduce the consistency problem, and the choice of the coherence protocol can have a significant impact on application performance. Cache coherence protocols dictate the actions to be performed on a write (the data item is produced) and they broadly fall in two categories: *write-invalidate* and *write-update*. Invalidation-based schemes are more suited to migratory data and can become inefficient when the producer-consumer relationship for shared data remains relatively unchanged during the course of execution. On the other hand, update-based schemes can result in repeated and redundant updates when the set of consumers for a data item changes during the execution. Adaptive prefetching and migratory sharing [7] are some hardware techniques proposed to augment the coherence mechanism towards lowering the communication overhead. Another interesting scheme that has been proposed is competitive update [8] which dynamically switches between updating and invalidating. The underlying protocol is update-based but a history of redundant updates to each individual cache block is maintained in hardware and is used by a processor to self-invalidate and thus avoid future updates to the block. There have been two main criticisms with this scheme. First, it is not clear how long a history is needed. Second is the hardware cost in maintaining the history (counter) for each cache block. Recent research [11, 12] has shown that there is not a significant degradation of performance by implementing coherence protocols in software while providing the added flexibility of tailoring the coherence mechanism for an application. Falsafi el al. [9] show that we can gain substantially by matching the coherence protocol to an application's communication pattern and memory semantics. However, with this approach there is a concern of increasing the programming complexity by requiring the application developer to choose the right coherence protocol unlike schemes like competitive update which rely on the underlying system to adapt to application behavior. Software-directed cache coherence schemes [6, 5, 16, 13] have also been proposed for specific application behaviors but compilation techniques for efficient communication in dynamic applications are still in their infancy.

We can view the communication supported by an underlying

shared memory system as either *sender-initiated*, where the producer sends the data item to one or more consumers as soon as the data item is produced [13], or *receiver-initiated*, where a consumer requests and receives the data item from the producer. As far as possible, it would be beneficial to employ sender-initiated communication since the data transfer can be initiated as soon as the data item is produced instead of waiting until the time the data item is actually needed (as is the case with receiver-initiated schemes) thus increasing the overlap of useful computation with communication. Invalidation-based coherence schemes rely on receiver-initiated communication while update-based schemes use sender-initiated data transfer.

A related study [22] shows that nearly all the communication in some applications can be overlapped with computation provided we are able to exactly predict the next consumer(s) for a data item and effect sender-initiated communication. Towards this goal, this research sets out to explore how we can intelligently use a mix of invalidates and updates so that we can get close to the read overheads of update-based schemes and the write overheads of invalidation-based schemes. In the process, we would like to keep our demands from the programmer, the compiler and the hardware as low as possible. We address this problem by classifying shared memory accesses into two types. In the first part of this paper, we attempt to reduce communication overhead for accesses to shared variables within critical sections which constitute an important phase of the execution in several parallel applications. Earlier research [19] has identified a primitive called SYNC_WRITE for write operations within critical sections but its performance benefits have not been clearly illustrated. Apart from conducting a detailed performance evaluation of SYNC_WRITE in this paper, we introduce variations of this primitive that can improve performance even further. In the latter part of this paper, we look at lowering the communication overhead for normal data accesses. In particular, we examine the competitive update mechanism in greater depth to address its limitations. We address the questions of how long a history is needed and how this scheme can be implemented efficiently in software without significant hardware costs. This work uses an execution-driven simulator called SPASM [25, 26] to simulate the execution of applications on different shared memory architectural platforms. Dahlgren and Stenstrom [8] point out the importance of network bandwidth on the performance results of a cache study. Consequently, we have simulated the interconnection network, a 2-D mesh, in its entirety to accurately model network latency and contention, instead of assuming a constant delay model.

This research makes the following contributions:

- We present and evaluate a set of primitives to lower the communication overhead for variables accessed within critical sections. We show the performance benefits of these primitives over the traditional read/write mechanisms for an application drawn from the Geographical Information Systems (GIS) domain. We also use a synthetic benchmark to exercise different application behaviors and study the effectiveness of our primitives.

- We show how we can implement an effective competitive update scheme in software with very little hardware cost on emerging shared memory architectures such as the Wisconsin Typhoon [20]. Such an implementation also provides us the flexibility of tailoring the history information to an application's demands giving us a menu of coherence protocols ranging from invalidation-based schemes to completely update-based schemes with an interesting spectrum of competitive update schemes in between. We study the communication performance of four dynamic applications under different coherence schemes and memory consistency models. The results for these four applications show that it would be better for a producer to send updates only when the consumer has read the data item since the last update (only the most recent history is needed for an efficient competitive update scheme). These results confirm the observations made in [8] where the authors point out that the network bandwidth limits the performance of competitive update schemes with longer histories. A competitive update scheme which uses only the most recent history can be efficiently implemented on our proposed hardware-software combination system without requiring any hardware counter.

Section 2 gives details on the simulator and the shared memory platform that is used for the evaluations. Section 3 identifies the primitives for optimizing shared memory accesses within critical sections and evaluates their performance. Section 4 addresses the lowering of communication overheads for the other shared memory accesses. Section 5 summarizes the contributions of this paper and Section 6 identifies directions for future research.

2 Simulated Architectural Platform

The base shared memory platform that we use for the simulation studies in this paper is a CC-NUMA machine similar to the Stanford DASH [15] and MIT Alewife [1]. Each node has a piece of the shared memory with its associated full-mapped directory information, a private cache, a write buffer and a write cache [8]. Figure 1 shows the relevant details of the node architecture. A cache block can exist in one of three states, INVALID, READ-ONLY and READ-WRITE. READ-ONLY state is a potentially shared clean state, while READ-WRITE state is an exclusive state requiring a write-back on cache line replacement. The write buffer is used in the implementation of memory systems with weaker memory models. For such memory systems, the write buffer keeps track of pending writes, and stalls the processor on a write when the buffer is full. Read operations are assumed to be blocking, and the processor is stalled until the operation is complete. The write cache [8, 12] is a small associate cache that collates updates to a cache block. An entry in the write cache is moved to the write buffer when it has to be replaced. This cache has been found effective in reducing the repeated update problem of update-based protocols [8]. Note that the write cache is used only in a release consistent (RC) memory model. The write buffer and write cache are both flushed when a processor reaches a synchronization fence.

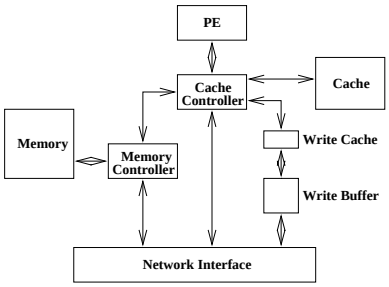

Figure 1: The Node Architecture

The performance benefits from the enhancements discussed in this paper are to be compared with traditional invalidation and update based cache coherence protocols across different memory

consistency models. Specifically, the three base systems that we use for comparisons are SCinv, RCinv and RCupd. SCinv and RCinv use a Berkeley-style write-invalidate protocol. SCinv works under the sequential consistency model with the processor stalling on each read and write that involves the network. The write buffer does not play any role for SCinv. RCinv works under a release consistent memory model, and a write that misses in the cache is recorded in the write buffer without stalling the processor. The pending write is retired when the ownership of the block is obtained from the directory controller. The processor stalls when the write-buffer is full or when the buffer is non-empty at a release point. RCupd uses a release consistent memory model with a write-update protocol. We have discounted an update-based protocol working under sequential consistency since its performance is expected to be very poor.

Cache Access	1 cycle
Memory Access	10 cycles
Link Bandwidth	1.65 cycles/byte
Network Messages	8-40 bytes
Cache Block Size	32 bytes
Write Buffer	4 entries
Write Cache	1 block
Network Interface Buffer	16 entries

Table 1: Simulation Parameters

This paper uses an application-driven evaluation approach and we simulate the execution of applications on the above-mentioned shared memory platforms implemented on SPASM [25], an execution-driven parallel architecture simulator. Simulating the interconnection network is very important in the performance evaluation of shared memory systems since the network contention can constitute a significant component of the overall execution time [25, 26] and can have a bearing on the conclusions drawn from these evaluation studies. Consequently, we simulate the interconnection network in its entirety for the studies in this paper. The simulated interconnection network is a 2-D mesh which uses wormhole switching with deterministic XY-routing. Since the focus of this paper is on reducing the communication overhead resulting from true sharing in applications, we assume an infinite cache at each node in our simulation and eliminate capacity misses. However, extending this study for finite caches is part of our future work. Table 1 gives the specifics of the base hardware used in our simulations.

SPASM gives detailed statistics on the performance of the parallel system [25]. Of these statistics, the profile of the execution time of a processor will be used for the studies in this paper. Apart from the Computation Time (CT) spent in executing the useful work in the application by the CPU, the execution time of a processor can be broken up into the Read Overhead (RO), Write Overhead (WO), Buffer Flush Overhead (BFO), and Synchronization Overhead (SO) components. Read Overhead (RO) is the wait time incurred by a processor for read misses. All the shared memory systems that we mentioned earlier would incur this overhead since we assume blocking reads. Write Overhead (WO) is the wait time incurred by a processor for write misses. SCinv will definitely incur WO on misses, but RCinv and RCupd can hide this overhead when the write buffer is not full. Buffer Flush Overhead (BFO) is the wait time incurred by a processor for outstanding writes to complete at the synchronization point for RCinv and RCupd. SCinv will not incur any BFO. The Synchronization Overhead (SO) is the gap between the time the application makes a call to the synchronization library (mutual exclusion locks, barriers) and the time it returns from this call.

3 Accesses Governed by Locks

3.1 The Mechanisms

Shared memory machines provide a shared address space that can be potentially accessed (read/written) by any processor. Synchronization support is provided to the programmer to enforce ordering between these accesses. The most prevalent of these synchronization constructs is the mutual exclusion lock which is used to implement critical sections in a large number of applications. In this section, we attempt to lower the communication overhead for accesses to variables accessed within critical sections governed by mutual exclusion locks. Prior work [19] has identified a primitive called SYNC_WRITE for these scenarios but the performance benefits have not been clearly illustrated. Apart from conducting a detailed performance evaluation of SYNC_WRITE, we introduce extensions of this primitive that can improve performance even further. Note that these primitives work within the confines of the coherence protocol (unlike other proposed mechanisms such as [27]) and keep the caches consistent as specified by the memory consistency model. Thus, there is no explicit address space management or correctness to be addressed by the programmer in using these primitives.

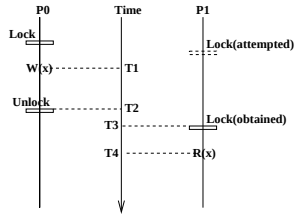

Figure 2: Accesses for variables governed by locks

Let us revisit the SYNC_WRITE primitive identified in [19]. Consider the data items written in a critical section governed by a mutual exclusion lock. Several producer-consumer applications such as those which manage task/data queues dynamically use locks to control accesses to the shared queue. Also, many scientific applications (CHOLESKY, BARNES-HUT and RADIOSITY to name a few from the SPLASH suite [23]) and database applications (such as the one evaluated in the next subsection drawn from the Geographical Information Systems domain) employ locks to govern critical sections. Figure 2 shows an example scenario where P0 writes a variable in a critical section at time T1 which is subsequently read by P1 at time T4 after obtaining the lock. On an invalidation-based system, P0's write would invalidate the block in P1's cache, and P1 would miss in the cache when it reads at time T4. Apart from losing the valuable time between P0's write and P1's read (ie T4 - T1) that can potentially be overlapped with communication, an invalidation-based scheme would incur two roundtrip messages at the very least (ignoring directory lookups). Piggybacking the written value along with the lock to combine synchronization with data transfer [4, 10] would still not effectively utilize the gap between the time when the data item has been produced and the unlock point (ie. T2 - T1) though we can lower the number of messages. A release consistent update based protocol holds the most promise for this scenario if the updates can be completed in the gap T2 - T1. But redundant updates to processors that may not need a copy any longer may cause the write overhead to be high (and may not be hidden within T2 - T1). The SYNC_WRITE mechanism proposed in [19] uses the

synchronization information (ie. a processor waiting for a lock is likely to access the data item associated with the lock) to effect intelligent updates as early as the time when the data item is produced (T1). To implement this, we assume that the programmer can explicitly provide the association between a data item and the corresponding lock structure governing the data item. This would be fairly simple for the programmer since an implicit association between the two is made in any case when the program is being written. We also assume that some help is available in implementing the lock library calls so that any processor waiting for a lock is identifiable through the lock structure. The compiler then performs a straightforward translation from the normal *WRITE(address,value)* to a *SYNC_WRITE(address,value,pmask)* for the writes governed by a lock. *pmask* is the processor mask for a singleton processor that is next in line to acquire the lock and is obtained by de-referencing the lock structure governing the critical section. Similar to a normal update write message, the underlying system sends the SYNC_WRITE message to the directory along with the pmask value. The directory invalidates those processors (except the one initiating the SYNC_WRITE) which currently have a copy and which are not present in pmask, and sends updates to the processor(s) in pmask (even if they may not have a previous copy). If pmask is empty (ie. there are no processors waiting for the lock), SYNC_WRITE degenerates to a normal write. This primitive will thus be particularly useful when there is high lock contention since there is a higher likelihood of a processor waiting for the lock.

The SYNC_WRITE primitive presented in [19] has been optimized for the scenario where the processor that is next in line for the lock (the consumer) will only be reading the data item. But in several applications such as the ones maintaining queues dynamically, the processor is not restricted to reads within the critical section. For instance, a typical operation found in applications is an atomic increment/decrement of a global counter that can be implemented as a read followed by a write within a critical section. In such cases, while the read would result in a hit because of the update sent by the previous lock owner's SYNC_WRITE, a write on the other hand would not be a local operation since an invalidation or an update would have to be sent to the previous lock owner to keep the two copies consistent. Unless the producer reads the data item subsequently after the unlock point without guarding this access with locks (which is very unlikely in well-written parallel programs), it would be better for the producer to relinquish its copy on the last write to a data item before the unlock point. The next processor to acquire the lock can thus get an exclusive copy and may not incur network messages for the reads or writes within the critical sections. Hence, we propose a variation of SYNC_WRITE called *SYNC_WRITE_REL* with the same set of parameters, which along with the update message relinquishes ownership of the block to the processor next in line for the lock and self-invalidates the producer's copy.

The typical mutual exclusion lock-unlock mechanism is restrictive in concurrency since it does not distinguish between the types of accesses that it governs. It serializes all the accesses to the data regardless of whether they are reads or writes. There are two reasons for its popularity. First, the cost for implementing fancier synchronization primitives such as read-write locks may not make them worthwhile. Second, read-write locks are only useful when the type of access that will be made in the critical section is known at the lock point. In several scientific codes, since the type of access that will be made is not known beforehand, the normal lock-unlock mechanism is conservatively used. On the other hand, several database applications can benefit from the added concurrency provided by read-write locks especially when the type of access a transaction would make can be predicted beforehand. While we are not advocating any particular synchronization construct in this paper, we are merely pointing out that in cases where *read-write locks* are employed, we can use SYNC_WRITE and SYNC_WRITE_REL more intelligently to lower communication overhead. For a write governed by a write lock, the compiler can generate code to look up the lock structure to find out whether reader(s) or a writer will gain access next. If the processors next in line are readers, the normal write would be substituted with a SYNC_WRITE with *pmask* being the reader set. If the processor next in line is a writer, the normal write would be substituted with a SYNC_WRITE_REL with *pmask* set to the next writer.

Note that the hardware costs for implementing SYNC_WRITE and SYNC_WRITE_REL are not very different. As mentioned in [19], these mechanisms only require that the directory be told (by giving it a mask) which processors to invalidate and which processors to update. Also there is no requirement that they be implemented in hardware. They can be easily implemented in software on recently proposed flexible architectures [11, 20]. Unlike the explicit communication mechanisms proposed in [27], these primitives do not require address space management nor explicit coherence management by the programmer since they are integrated with the underlying cache coherence protocol. Further, as we have pointed out in this section, the insertion of these primitives in the application code can be easily automated without requiring extensive programmer or compiler support.

3.2 Performance Evaluation

To evaluate the performance benefits of the SYNC_WRITE and SYNC_WRITE_REL primitives, we have used a database application [18] drawn from the Geographical Information Systems (GIS) domain. In this application, processors periodically read and update data structures called quad-trees [21] which are used to maintain spatial geographical information such as landscapes and rainfall. Even though read-write locks can improve the concurrency in this application, the implementation for this study uses only mutual exclusion locks. The three main operations that are performed on the quad trees are *build*, *retrieve* and *join*. The build routine constructs (writes) quad trees from data files and the retrieve routine reads information from the quad trees. The join routine performs both reads and writes (typically ANDs and ORs of quad-trees) to answer user queries.

We study the execution of this application on a 16 processor shared memory machine under two scenarios. In the first scenario, the number of retrieve and join operations are much higher than the build operation (ie. number of reads in critical sections are higher than the writes). In the second scenario, the number of builds and joins are higher than the retrieve operation (ie. number of writes in critical sections are higher than the reads). Consequently, the second scenario exhibits a more migratory sharing pattern for the data items in the quad-tree compared to the first. Thus, an update-based protocol would be more suited for the former while an invalidation-based protocol would suit the latter. The Computation Time and the Synchronization Overhead (SO) for this application are nearly the same across the different combinations of coherence protocols. Hence, to focus on the communication performance of the different shared memory systems, we will concentrate on the Read Overhead (RO), Write Overhead (WO) and the Buffer Flush Overhead (BFO)[1].

[1]To put these overheads in perspective, the sum of RO, WO and BFO shown in Figure 3 constitutes nearly 40% of the total execution time for the SCinv case.

Figures 3 and 4 show the communication overheads for the non-migratory and migratory sharing scenarios of the application respectively on the different shared memory platforms being considered. RCsyn and RCsynrel are release consistent shared memory models using an invalidation-based protocol for normal writes, and the SYNC_WRITE and SYNC_WRITE_REL primitives respectively for writes governed by locks. From these figures, we make the following observations:

- Figure 3 clearly shows that the update protocol performs much better than the invalidation protocol. The read overheads on invalidation-based schemes are higher while the write overheads are lower than on their update-based counterparts. For the non-migratory sharing pattern exhibited by the application in the first scenario, the large reduction in RO far outweighs the higher WO on update protocols. In fact, since we have discounted capacity misses in our simulation, the RO for update protocols is only due to cold misses and is thus very low.

- In the migratory sharing behavior of the application shown in Figure 4, the relative performances of the invalidation and update based schemes are reversed compared to Figure 3. Because of the lower relative probabilities for the read (*retrieve*) accesses, the number of processors sharing a data item at any particular time decreases. Also, the set of processors sharing a data item changes during the execution. In the earlier case since there were several readers for a data item, the probability of redundant updates was much lower than in this case. Consequently, the write overheads on update-based schemes dominate over any benefits gained by lower read misses.

- RCsyn and RCsynrel uniformly perform better across migratory and non-migratory patterns. The read overheads with these primitives are even better than what we set out to achieve (ie. read overheads of the update-based scheme). This is possible because we can avoid some of the cold misses since the producer processor sends the data item to a waiting consumer even though the consumer may not have read the data item earlier (which does not occur with normal update based schemes). Also, the write overheads with these primitives are much lower than those on update based scheme because of avoiding redundant updates. This difference is accentuated for migratory data (Figure 4). The performance improvement of SYNC_WRITE_REL proposed in this paper over SYNC_WRITE is not significant in Figure 3 since the probability of writes within critical sections is lower than the reads. However, when the probability of writes increases, the performance improvement becomes more significant (Figure 4). Using a synthetic benchmark we have also studied the impact of the access type (read/write) in critical sections on the relative performance of SYNC_WRITE and SYNC_WRITE_REL [24].

- Release consistency lowers the communication overhead of the invalidation-based scheme across both sharing patterns by lowering WO. The graphs also show that for the invalidation protocol, WO is almost completely hidden when we move from SC to RC, but this is not the case for the update protocol. Further, the buffer flush overhead (BFO) is marginal for all the cases suggesting that the write buffer is nearly always empty when the processor reaches an unlock point. These observations suggest that (a) the writes in the application are fairly close to each other since the update protocol has a significant WO even with RC (ie. the write buffer is full most of the time and the processor has to stall until at least one pending write is retired); (b) invalidations are not too expensive for either sharing pattern since even the small gap between successive writes suffices to hide the WO for RCinv; and (c) a deeper write buffer (greater than a size of 4 used in this study) would help RCupd, RCsyn and RCsynrel even further since WO is much higher than BFO.

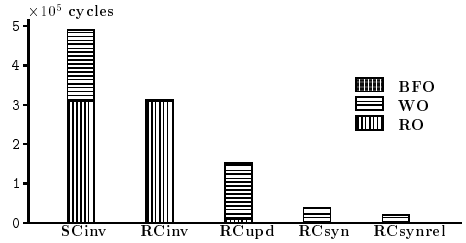

Figure 3: Communication Overhead for Quad-tree Accesses with Non-Migratory Shari ng

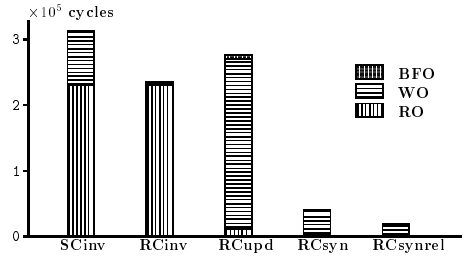

Figure 4: Communication Overhead for Quad-tree Accesses with Migratory Sharing

While application-driven studies lend realism and credibility to the evaluation results, a common problem with such studies is that the results may be restricted to the applications and system parameters under consideration. However, synthetic benchmarks which mimic typical application characteristics and which allow us to study the impact of a range of parameters, can sometimes be used to augment an application-driven study. Hence, apart from using the quad-tree GIS application discussed here, we have also used a synthetic benchmark for detailed analysis of the relative performance benefits of SYNC_WRITE and SYNC_WRITE_REL under varying application demands. However, due to space constraints we are not presenting the performance results in this paper and the reader is referred to [24] for further details.

4 Other Shared Memory Accesses

4.1 The Mechanisms

In the previous section we showed how the additional information provided by the lock construct may be used to effect sender-initiated communication for variables accessed within critical sections. In this section, we explore how we can employ sender-initiated communication for the accesses which use the other common synchronization construct, namely the barrier. The following discussion would also be applicable for critical section accesses that have little lock contention (the information about the processor next in line for the lock is not available).

When a processor arrives at a barrier synchronization point, it is stalled until all other processors have arrived at the barrier. The barrier construct is usually employed to order accesses between

different phases of program execution or to order accesses across iterations in a program. The barrier ensures that a write by a processor in the current phase/iteration will complete before a read by any processor in the next phase/iteration. The read accesses in the next iteration would incur misses on an invalidation-based protocol while the update-based scheme would send updates to all processors who have read the data item in any of the preceding iterations regardless of whether they will be reading in the succeeding iteration. The ideal communication mechanism for these scenarios would effect sender-initiated data transfer at the write point in the current phase/iteration to the processors who will be reading them next. But this would need an oracle to predict the future access pattern of the application. Since we are focusing on the domain of dynamic applications whose communication behavior is not known beforehand, we need good heuristics to predict the future shared memory accesses of the processors. The problem has close similarities to predicting the memory reference behavior of uniprocessor programs for effective cache/page replacement policies. The locality principle, which advocates the use of the recent past history to predict the near future and which has been proven to work well for cache/page replacement policies, may also be applicable for this problem. The competitive update scheme [8] makes use of the past history of the application to predict future behavior. In this scheme, the underlying cache coherence protocol is update-based. A hardware counter is maintained for each cache block and is initially set to zero. Whenever the local processor accesses a cache block, the corresponding counter is reset to zero. Each time the cache controller receives an update message from the directory, it increments the associated counter. When the counter exceeds a preset threshold value, the cache controller self-invalidates the cache block (ie. sends a self-invalidate message to the directory and changes the cache state to INVALID). The counter thus keeps track of the number of external updates received since the last access by the local processor and the threshold is used to limit the number of redundant updates in the system. But the two main criticisms of the competitive update scheme are: (a) it is not clear how long a history to maintain; and (b) the high cost and inflexibility in maintaining these counters in hardware. In the following discussion, we address both these problems. Our results confirm observations from an earlier study [8] which show that very small counters (a threshold of 1) suffices for the four applications considered in this study. With marginal hardware cost, we show how the competitive update scheme can be implemented efficiently in software on emerging shared memory architectures. In fact, a competitive update scheme with a threshold of 1 does not require a counter in our hardware-software combination system.

4.2 Performance Evaluation

There have been few studies [8] which have evaluated the trade-off between the lowering of read misses with higher threshold values of competitive update versus the increase in the update traffic. By not simulating the contention in the network, the performance benefits of a higher threshold value may be exaggerated. Hence, we have simulated the interconnection network, a 2-D mesh, in its entirety in this study. We have also looked at applications that are different from the ones considered in the earlier study [8]. As we will shortly observe, these applications exhibit varying communication behavior. The four dynamic applications that we consider for this study are Barnes-Hut [23], Maxflow [2], CHOLESKY [23] and CG [3]. We simulate their execution on the SCinv, RCinv and RCupd platforms discussed earlier and compare their performance with competitive update implementations (denoted as RCcomp) using different threshold (t) values for a 16 processor system. For two of the applications (Barnes-Hut and Maxflow), we find the competitive update scheme adapting better to the sharing pattern of the application than either an invalidation or an update based protocol. For CHOLESKY which has a very migratory sharing pattern, the invalidation scheme performs the best but competitive update comes fairly close. For CG which has a non-migratory sharing pattern, the competitive update and normal update schemes perform equally well. In all four applications we find that the detrimental effect of the higher update traffic outweigh the advantages of lower read misses for competitive update schemes with threshold greater than 1.

Barnes-Hut

Barnes-Hut [23] is an N-body application which simulates the movement of a set of bodies in space due to the gravitational forces exerted on each other over a sequence of time steps. The parallel implementation statically allocates a set of bodies to each processor and iterates over time steps separated by barriers. To calculate the new position of a body at each time step, a processor needs to know (read) the center of mass of the bodies within a certain sphere of influence obtained from the calculations (writes) of the previous iteration. Over successive time steps, the bodies are not expected to move significantly in space. Consequently, the set of neighbors from which a processor will be reading data items will not change significantly over successive time steps. But gradually over a period of time, the body could move to a different sphere of influence (different neighbors). The problem size for this study uses 128 bodies over 50 time steps.

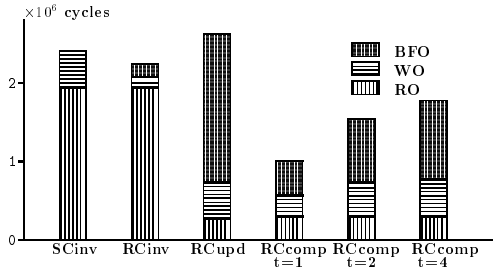

Figure 5: Communication Overhead for Barnes-Hut

Figure 5 shows the communication overhead of this application on the chosen shared memory platforms. Invalidation-based protocols incur higher read misses since they are not optimized for the communication in successive time steps when the sharing pattern of a body does not change significantly. On the other hand, update-based protocols incur a high write traffic (which manifests itself as WO and BFO) since the writes propagate to the old set of neighbors even if a body has moved significantly in space. Even under a release consistent model, the higher write overheads cause update-based schemes to perform worse than their invalidation-based counterparts. However, we find that the competitive update schemes are able to send updates to near neighbors on successive time steps (resulting in lower RO compared to invalidation based schemes) as well as drop processors when the body moves significantly away (resulting in lower WO/BFO than update based schemes). Further, we can also observe that RCcomp with t=1 performs better than for higher threshold values. When a body moves in a particular direction in a time step, the probability of it reversing the direction in the next few iterations is fairly low. Consequently, even with t=1, we are able to get close to the low read misses of fully update-based schemes. Hence, even on faster networks we do not expect higher threshold values to perform much better than with t=1 for this application.

Maxflow

This application finds the maximum flow from a source to a sink in a directed graph with edge capabilities. In the implementation [2], each processor accesses a local work queue for tasks to perform. These may in turn generate new tasks which are added to this local work queue. Each task involves reads and writes to shared data. The local queues of all processors interact via a global queue for load balancing. The problem size used in this study uses a graph with 200 nodes and 800 edges.

Figure 6 shows the communication overhead of this application on the different shared memory systems. The sharing pattern in this application is migratory. Consequently, the invalidation based schemes perform much better than their update based counterparts. However, we find that competitive schemes track the sharing pattern of the application fairly well giving better performance than ordinary invalidation or update based schemes. We also find that the reduction in read misses for larger threshold values is overshadowed by the update traffic suggesting that we stick to a threshold of 1 for this application.

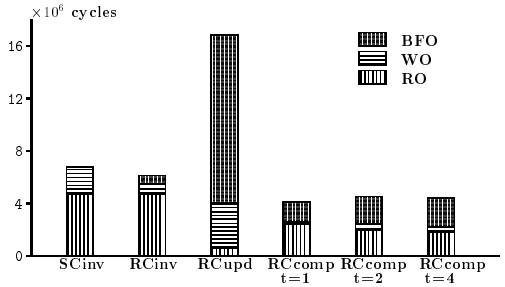

Figure 6: Communication Overhead for Maxflow

CHOLESKY

CHOLESKY is from the SPLASH benchmark suite [23] and performs a cholesky factorization of a sparse positive definite matrix. The algorithm requires an initial symbolic factorization of the input matrix which is done sequentially because it takes relatively less time. Only the numerical factorization is parallelized and analyzed. Sets of columns having similar non-zero structure are combined into supernodes at the end of the symbolic factorization. Processors get tasks from a central task queue. Each task involves fetching the associated supernode, modifying it and using it to modify other supernodes. At the end of it, one or more supernodes may be added to the task queue. The sparse nature of the input matrix and the runtime assignment of tasks to processors makes this application behavior very dynamic. The problem size used is a 1086*1086 matrix with 30,824 floating point non-zeroes.

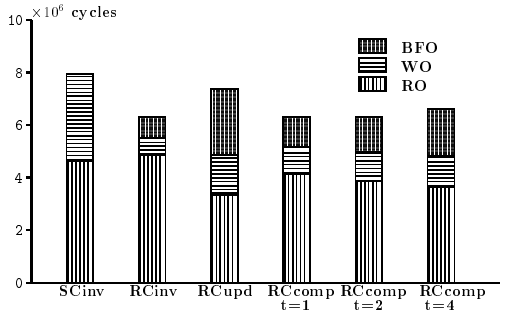

Figure 7: Communication Overhead for CHOLESKY

As with Maxflow, the sharing pattern in this application is very migratory. Figure 7 thus shows the invalidation protocol outperforming the update protocol for this application. While the recent history is a good indication of the immediate future for Maxflow, it is not the case for CHOLESKY. Thus the competitive update scheme does not fair as well. Still, this scheme performs much better than the normal update protocol. As with the earlier two applications, we do not find any significant change in performance when we move from t=1 to t=4 to warrant higher threshold values for competitive update.

CG

CG is a "Conjugate Gradient" application drawn from the NAS [3] benchmark suite which uses the Conjugate Gradient method to estimate the smallest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeroes that is typical of unstructured grid computations. The sparse input matrix and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor. We present the results for five iterations of the Conjugate Gradient Method in trying to approximate the solution of a system of linear equations. There is a barrier at the end of each iteration. The main operation in each iteration is a matrix-vector product. For the computation of the matrix-vector product, each processor performs the necessary calculations for the rows assigned to it in the resulting matrix (which are also the same rows in the sparse matrix that are local to the processor). But the calculation may need elements of the vector that are not local to a processor. Since the elements of the vector that are needed for the computation are dependent on the randomly generated sparse matrix, the communication pattern for this phase is random. However, since the sparse matrix does not change across iterations, a processor reads the same elements of the vector that it read in the previous iteration. A sparse matrix of size 1400*1400 with 100,300 non-zeroes has been used for this study.

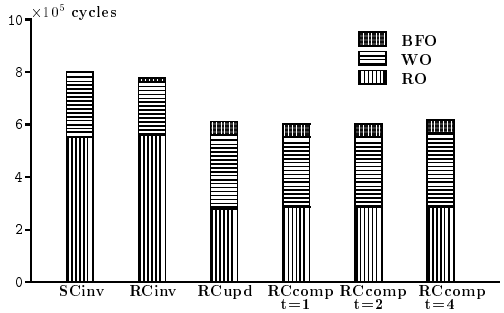

Figure 8: Communication Overhead for CG

The set of consumers for an element of the vector will not be known at the time of the write for only the first iteration (leading to cold misses for the reads). For succeeding iterations, the processors that will be reading a data item does not change (since the non-zero pattern of the sparse matrix does not change). Thus there are no redundant updates and we would expect the normal update protocol to perform the best for this application. Figure 8 confirms this observation. Also note that there is very little difference between the update protocol and the competitive update schemes. Consumers do not drop out and rejoin from the sharing pattern for this application. Consequently, the counter value for competitive update gets reset by the local processor access most of the time before exceeding 1. Thus the performance across various t values remains similar.

4.3 Implementation of the Mechanisms

The performance results in the previous subsection confirm earlier observations [8] that a threshold of 1 would suffice for

competitive update when network contention is taken into account. However, the scheme proposed in [8] uses a hardware implementation of competitive update, and even a threshold of 1 would require a 1-bit hardware counter to be associated with each block. Apart from the hardware cost, such a scheme would not provide the flexibility of changing threshold values if a new application were to demand higher threshold values or even if we would like to associate different thresholds for different variables in a program. In the following discussion, we present an implementation of competitive update that performs coherence actions in software with with very little hardware demands on emerging computer architectures such as the Wisconsin Typhoon [20] and Stanford FLASH [11].

Recent studies [11] have shown that there is not a significant degradation of performance by implementing coherence protocols in software. Recognizing the importance of providing the flexibility to tailor the protocol to an application's demands [9], emerging shared memory architectures provide programmable network interfaces such as the MAGIC chip provided on the Stanford FLASH [11] and the programmable processor provided on the Wisconsin Typhoon [20]. On such architectures, the network interface would be programmed to take appropriate actions for local processor events (read miss, write miss) as well as external events (read request, write request, invalidate/update request). Only the local processor's read and write hits would not (and should not due to performance reasons) involve the network interface. Hardware tags [20], such as INVALID, READ-ONLY and READ-WRITE, are provided with each block for access control so that the runtime system can detect any access violations (read/write misses, write for a read-only block) and take appropriate actions. In earlier CC-NUMA machines such as the DASH, the current state of the cache block as well as state transitions were maintained in hardware. On the other hand, in emerging flexible architectures only the state of the cached block is maintained in hardware while the state transitions are effected by the software. Let us look into the issue of implementing a competitive update scheme by maintaining the counters in software (instead of the hardware) on top of such emerging architectures.

An efficient and flexible implementation of this scheme requires (a) the specification of the counter threshold value in software; (b) incrementing the counter for every external update event; (c) self-invalidation when the counter reaches a certain threshold and (d) reset of the counter each time the processor accesses the corresponding block. Except for (d), all the other operations can be easily implemented in software on the programmable network interface. The problem is in implementing (d) since it requires certain actions to be performed even on read/write hits which would not be possible on architectures like the Typhoon [20] since no access violations are incurred in these cases to invoke software handlers. To fix this problem, we suggest a marginal hardware modification in requiring an additional tag (state) to be associated with the block and introducing one hardware effected state transition (apart from the conventional transitions effected in software). For instance, consider the four tags/states (INVALID, READ-ONLY, READ-WRITE, UNACCESSED) shown in Figure 9. Of these, the first three are well understood states/tags and which are available on the Typhoon [20]. To these three, we add a fourth state/tag called UNACCESSED which in addition to being a VALID state also says that the local processor has not accessed the block since the last update to that block. Note that from the hardware point of view, this will not need any additional bits since the two bits that are used to implement the three earlier tags can still accommodate the fourth tag. Figure 9 also shows the state transitions. We use solid lines to indicate the transitions effected by the software and the dotted line to indicate the additional transition effected by our suggested hardware. When the network interface receives an update message for a block which is in the READ-ONLY state, it changes the tag to the UNACCESSED state and the corresponding counter is set to 1 by the software. On the other hand, if the block is already in the UNACCESSED state, the software increments the counter associated with the block. Next, the software checks to see if the counter for the block has exceeded the threshold. If it has, the software self-invalidates the block and changes its state to INVALID. The only transition that needs to be effected by the hardware (shown in dotted line in Figure 9) is on a read hit when the block is in UNACCESSED state. In such cases, the hardware should change the state to READ-ONLY while returning the requested data item to the CPU. Writes in the UNACCESSED state are treated like writes for the VALID state and should be handled by the software since an access violation can be detected.

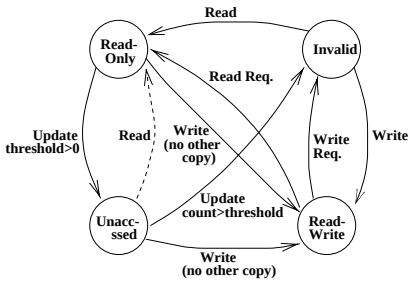

Figure 9: State Transitions for Competitive Update

The above scheme provides us the flexibility and lowered cost of implementing competitive updates in software. The counter thresholds may be tailored to an application's demands, and potentially we can set different thresholds for different data items in the program. Incidentally, by setting the threshold to 0, the above scheme would degenerate to an invalidation-based protocol while a very high threshold would make it perform as an update-based protocol. Also, in the case where we would like to allow just one update between between successive local processor accesses (threshold of 1 for which we showed the best performance in the previous subsection), *we don't need a counter at all* and the UNACCESSED state which keeps track of whether the local processor accessed the block since the last update would suffice.

5 Summary

Shared memory multiprocessors offer the convenience of a shared address space. This makes them particularly appealing for applications with dynamic communication behavior since the mechanisms for data transfer between processors is hidden from the programmer. But the scalability of these machines is often limited by the overhead incurred in accessing locations in remote memories. Caches alleviate this problem by bringing recently/frequently used data items closer to the processor to minimize network traffic. Still, the induced traffic for maintaining coherence between the caches can have a large impact on limiting performance. Invalidation based protocols have the disadvantage of always relying on receiver-initiated communication and are inefficient when the sharing pattern for data items does not change. Update-based protocols, though sender-initiated, could send redundant updates to processors that may not need them thus increasing write overheads. In this paper, we have tried to exploit

the benefits of sender-initiated communication wherever possible without significantly increasing the number of redundant updates. In the process, we have tried to keep our demands from the programmer, the compiler, and the hardware as low as possible.

First, we have focussed on writes to variables accessed within critical sections governed by locks. Earlier research [19] has identified a primitive called SYNC_WRITE which uses the information associated with the lock to find the processor next in line for the lock, and effects data transfer directly to this processor (and invalidates all other processors with a copy). However, detailed performance benefits of this primitive have not been clearly demonstrated in earlier research. Using a database application drawn from the Geographical Information Systems (GIS) domain, we have illustrated the performance benefits of this primitive over traditional invalidation and update based protocols across different memory models for both migratory and non-migratory behavior of the application. We have also introduced a variation of this primitive called SYNC_WRITE_REL which transfers ownership to the processor next in line for the lock, and we have shown the additional performance improvement with this enhancement for the GIS application. The write overheads for these primitives are significantly lower than those for update based schemes. Surprisingly, we achieve even lower read overheads than the normal update scheme using these primitives for the GIS application because we are able to reduce some of the cold misses. From the programmer viewpoint, the only requirement is the association of a data item with the corresponding lock structure governing it. From the compiler, we only require that it recognize a write within a critical section and transform the write to the new primitive using the association with the lock provided by the programmer. From the hardware, we require that the memory directory controller accepts masks of processors to update and invalidate instead of merely picking this information up from the directory as in conventional shared memory multiprocessors. Such actions would be even easier to implement on emerging shared memory architectures [11, 20] where the coherence mechanisms are implemented in software. We have also pointed out the use of SYNC_WRITE and SYNC_WRITE_REL in conjunction with read-write locks though we have not evaluated this issue in this paper.

Next, we have looked at more general data accesses for which information about the next consumer(s) for the data items is not readily available. These could be accesses governed by barriers or even accesses governed by locks in situations where there are no processors waiting for the lock (low lock contention). For these accesses, we have employed the competitive update scheme [8] which uses sender-initiated data transfer but tries to limit the number of redundant updates to a processor by keeping track of the past history of reference by that processor. Earlier criticisms with competitive update schemes was the inflexibility and higher cost of implementing it in hardware. Further, it was not clear how long a history needs to be maintained since there is a trade-off between lower read overheads and higher update traffic. In this paper, we have shown how the competitive update scheme can be efficiently implemented in software with very little hardware support. Such an implementation also provides us the flexibility of tailoring the history information to an application's demands giving us a menu of coherence protocols ranging from invalidation based schemes to completely update based schemes with an interesting spectrum of competitive update schemes in between. Using four dynamic applications and a detailed simulation of a mesh network, we have shown that a competitive update scheme with a threshold of 1 performs as well or even better than for higher threshold values. For the application with very migratory sharing pattern, this scheme performs close to the invalidation-based protocol, and for the application where the sharing pattern does not change, this scheme performs close to the update-based protocol. For two other applications, this scheme performs better than either the normal invalidation or update based schemes. A competitive update scheme with a threshold of 1 can be efficiently implemented on our proposed hardware-software combination system. In fact, we do not need to maintain a counter at all and the extra state that we use in implementing the coherence protocol would suffice. Over the evolving hardware of recent architectures [20], we only require an additional tag/state with each block. These architectures already use some tags for access control, and we may be able to encode this additional tag/state using the existing bits. From the programmer and compiler viewpoint, the underlying optimizations performed on a read/write are hidden by the hardware/runtime system.

6 Future Work

This study has suggested several interesting directions for future research:

- We would like to augment this study with more applications spanning diverse domains. While many of the architectural studies have targeted specific application domains, we need to widen our horizons. Particularly, we would like to explore the geographical information systems and computational molecular biology domains. The GIS application considered in this study is a step in that direction. By studying more applications, we may be able to identify other scenarios for intelligent sender-initiated data transfer mechanisms, apart from being able to generalize the results presented in this paper.

- Even though we have identified mechanisms for sender-initiated data transfer for variables governed by read-write locks we have not studied their performance benefits in this paper. This is part of our ongoing work. These mechanisms can particularly benefit database applications, such as the GIS application examined in this paper.

- Our experiences with the GIS application also suggests that it may be useful to have bulk-transfer mechanisms in conjunction with the SYNC_WRITE primitives. For instance, we could have a primitive called *SYNC_XFER(address,nbytes,lock)* which could transfer the specified number of bytes to processor(s) waiting for the lock. It is not clear at this point how we can efficiently integrate such a mechanism with the underlying coherence protocol or whether we should worry about coherence at all for such data transfers.

- As we mentioned, an advantage of implementing competitive updates in software is that we have a menu of coherence protocols which can be tailored to an application's demands (by setting the threshold value appropriately). In fact, each data item in the program could potentially have different threshold values. We would like to explore compiler and runtime issues for exploiting this feature.

- The evaluations in this paper use an infinite cache since we are more interested in coherence misses than in capacity misses. We feel that the results presented in this paper will only be accentuated when we move from infinite to finite caches. However, finite cache studies are also part of our future plans.

Acknowledgments

The author would like to thank the members of the architecture group at Georgia Tech for the many helpful discussions leading to the ideas in this paper. The SYNC_WRITE primitive was co-designed with U. Ramachandran, G. Shah and A. Singla and the SPASM simulator was developed when the author was a graduate student at Georgia Tech.

References

[1] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] R. J. Anderson and J. C. Setubal. On the Parallel Implementation of Goldberg's Maximum Flow Algorithm. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 168–177, June 1992.

[3] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[4] P. Bitar and A. M. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[5] H. Cheong and A. V. Veidenbaum. Stale data detection and coherence enforcement using flow analysis. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 138–145, August 1988.

[6] R. Cytron, S. Marlovsky, and K. P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 229–238, 1988.

[7] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined performance gains of simple cache protocol extensions. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 187–197, April 1994.

[8] F. Dahlgren and P. Stenstrom. Reducing the Write Traffic for a Hybrid Cache Protocol. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages I 166–173, 1994.

[9] B. Falsafi et al. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, November 1994.

[10] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, June 1988.

[11] M. Heinrich et al. The performance impact of flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[12] 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, 1995.

[13] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared memory multiprocessors. In *Proceedings of the ACM 1995 International Conference on Supercomputing*, pages 255–264, July 1995.

[14] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H Lim. Integrating message-passing and shared memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[15] D. Lenoski et al. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[16] S. L. Min and J-L. Baer. A Timestamp-based Cache Coherence Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I: 23–32, August 1989.

[17] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[18] D. J. Peuquet and L. Qian. An Integrated Database Design for Temporal GIS. In *Seventh Inernational Symposium on Spatial Data Handling*, 1996.

[19] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of Supercomputing '95*, December 1995.

[20] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[21] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley, Reading, MA, 1990.

[22] G. Shah, A. Singla, and U. Ramachandran. The quest for a zero overhead shared memory parallel machine. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995.

[23] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[24] A. Sivasubramaniam. Reducing the communication overhead of dynamic applications on shared memory multiprocessors. Technical Report CSE-96-047, Dept. of Computer Science and Engineering, The Pennsylvania State University, July 1996.

[25] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[26] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On characterizing bandwidth requirements of parallel applications. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 198–207, May 1995.

[27] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.