Architectural Primitives for a Scalable Shared Memory Multiprocessor *

Joonwon Lee *Umakishore Ramachandran*

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332 USA

Abstract

Since large memory latencies are not uncommon for a large scale multiprocessor, researchers have investigated memory models to relax memory access ordering. This paper presents a new memory model, and discusses implementation issues in a cache-based environment. These issues motivate a set of architectural primitives with which software can implement desired memory consistency. For efficient cache management, we propose a cache protocol that allows read-initiated actions for coherence maintenance. For efficient synchronization, we present a cache-based locking scheme that implements queued busy-waiting using cache lines. The scalability of the proposed schemes is explored through analytical modeling and simulation studies.

1 Introduction

Parallel programming based on the shared memory paradigm is a natural progression from sequential programming. Therefore, it is not surprising that shared memory multiprocessors (such as the Sequent and the BBN Butterfly) are popular for developing parallel applications. Message-passing multiprocessors (such as the Intel Hypercube) are intrinsically more scalable than shared memory multiprocessors primarily due to the potential contention for shared memory in the latter [22]. Due to the popularity of programming with the shared memory paradigm, this abstraction is being supported even on message-passing architectures [16]. However, simulating shared memory on message-passing architectures is inherently slower than true shared memory. The main motivation for the research presented in this paper is to understand the issues in realizing scalable shared memory multiprocessors and suggest architectural features that address these issues.

There are two main problems to be solved in realizing scalable shared memory multiprocessors: latency for memory accesses, and network contention generated by these memory accesses. Both these problems are related to the model of memory provided by the architecture. Traditionally, the model assumed by the programmer is that the contents of the shared memory is identical at all times from all the processors. Further, the model assumes that the completion order of the memory references from a single processor is strictly in program order. Both these assumptions restrict the scalability of shared memory multiprocessors. For example, the second assumption prohibits out-of-order completion of memory accesses which is important to enhance performance, especially when the memory latency is high. In parallel applications, it is not unusual to use synchronization operations to ensure the consistency of shared data. In such cases, a temporary inconsistency in the views of the shared data as seen from different processors may be tolerable in certain ranges of the program, e.g., inside a critical section and between barrier synchronization points. This observation leads to weaker consistency models [21, 1, 4] in which updates to shared memory may be delayed until a synchronization point. Using such weaker models of memory is one approach to realizing scalable shared memory multiprocessors.

Another approach to addressing the issues of latency and network contention is to associate private caches with each processor. The effectiveness of this approach depends on an efficient strategy for maintaining the coherence information as well as the choice of the coherence protocol. Snooping cache protocols [2] with distributed directories for maintaining the consistency information have been popular in bus-based shared memory multiprocessors. These protocols exploit the fast broadcast capability of the bus to efficiently implement

*This work is supported in part by the NSF PYI Award MIP-9058430

the coherence protocol. But it is well-known that a bus is not a scalable interconnection network. Unfortunately, more scalable interconnections such as a multistage interconnection network do not usually have such a fast broadcast capability. For this reason, coherence protocols based on a central directory for maintaining the consistency information are usually preferred for scalable shared memory multiprocessors [5, 3].

The coherence protocols define the actions to be taken when a cache line is modified. To maintain consistency on writes to a cache line, the protocol may choose to either invalidate or update copies of this line in other caches. In bus-based systems, it is possible to make a case for either choice depending on the memory reference pattern [2, 9]. In large-scale multiprocessors invalidation is preferred to updates because of the high network transit time [3, 4]. Both these strategies implicitly enforce consistency on writes to a shared location. A dual to this strategy is to explicitly request consistency on reads. In our view such a strategy would reduce the overhead of coherence maintenance especially in large-scale multiprocessors by allowing the consistency requirements to be customized by the software (compiler) instead of blind enforcement.

For performance reason, a cache line usually consists of several words and the cache protocol treats a line as the unit of consistency maintenance. However, program variables may vary in size from a bit to arbitrary length. Since a cache line may contain several words, unless explicit care is taken in writing the parallel program or by the compiler [11], it is quite likely that the same cache line may contain private variables of parallel threads executing on different processors. These cache lines appear shared from the point of view of the cache protocol leading to unnecessary invalidations or updates. This phenomenon is referred to as *false sharing* and limits the line size. From the performance point of view it is important to reduce the effects of false sharing especially for large-scale multiprocessors.

There are two types of memory accesses generated by a parallel program: accesses to normal data, and accesses to synchronization variables. An analysis of the memory reference pattern of parallel programs reveals that the synchronization accesses cause much greater network contention than accesses to normal shared data [18]. A serious limitation to scalability is the fact that the hardware does not distinguish between accesses to synchronization variables and accesses to normal data. Recognizing the importance of making this distinction researchers have proposed hardware primitives for synchronization [23, 8, 10, 13]. An important advantage of making this distinction is that it provides a setting for efficiently supporting the weak consistency model.

This paper presents a new memory consistency model, and discusses implementation issues in a cache-based environment with respect to this model. These issues motivate a machine architecture that provides primitives for addressing the scalability issues raised in this section. The paper concludes with preliminary performance implications of our machine architecture, and directions for future research.

2 A New Consistency Model

In shared memory multiprocessors there is a need for defining a consistency model that specifies the order of execution of memory accesses from multiple processors. Such a consistency model would facilitate reasoning about the correctness of programs written for multiple processors. For simplicity, it is usually assumed that the result of a write operation be immediately observable by other processors. With this assumption, Lamport [12] has proposed *sequential consistency* as the ordering constraint for the correct execution of a multiprocess program: The multiprocessor execution of the program should have the same effect as a sequential execution of any arbitrary interleaving of the operations of all the processes (that comprise the program). The allowed interleavings are those that preserve the program order of operations of each individual process. With the sequential consistency model, read and write operations are sufficient to implement synchronization operations correctly. However, this model is inherently inefficient since it imposes a strong ordering constraint for all memory accesses regardless of the usage of shared data. Further, each memory access has to wait until the previous memory access is completed. Thus large scale shared memory multiprocessors are expected to incur long latencies for memory accesses if this ordering constraint is imposed, leading to poor performance.

In parallel program design, it is not unusual to use synchronization operations to enforce a specific ordering of shared memory accesses. Based on this observation Dubois et al. [7] have proposed *weak ordering* that relaxes the ordering constraint of sequential consistency by distinguishing between accesses to synchronization variables and ordinary data. Their model requires (a) that synchronization variables be strongly consistent, (b) that all global data accesses be globally notified before synchronization operations, and (c) that all global data accesses subsequent to the synchronization operation be delayed until the operation is globally performed. Thus this model requires strong consistency of global data accesses with respect to synchronization variables.

There are several types of synchronization operations: barrier, lock and unlock, and semaphore P and V. While synchronization variables need to be strongly consistent, the consistency requirements for shared data preceding or succeeding accesses to synchronization variables may be different.For example, consider a critical section.

Write operations on shared data performed by a processor prior to entering the critical section need not be globally performed. It is enough if the write operations inside the critical section are globally performed just before exiting it. Therefore, inside the critical section all reads and writes may be treated as entirely operations local to the processor. Similarly, write operations performed after exiting the critical section need not wait for the completion of the synchronization operation that signals the exit.

This observation regarding the consistency requirements and the relative ordering requirements lead to several extensions to the weak ordering model proposed by Dubois et al [7]. *Buffered consistency* is the memory model used in this paper and is defined as follows. There are two types of accesses: synchronization and normal read/write. Synchronization accesses are further subdivided into two classes: consistency preserving (*CP-Synch*), and non-consistency preserving (*NP-Synch*). An NP-Synch operation does not wait for the completion of writes to shared data preceding it. A CP-Synch operation is allowed to be performed only after all writes to shared data preceding it have been globally performed. Writes to shared data issued after a synchronization operation need not be delayed until the synchronization operation is globally performed. Other researchers [4, 1] have proposed weaker models that allow relaxing the consistency requirements within critical sections. Our model allows further weakening of the consistency and ordering requirements even in other synchronization scenarios.

Examples of synchronization operations that belong to the NP-Synch class are: lock, and semaphore P; examples of CP-Synch class of operations include: unlock, semaphore V, and barrier synchronization. Buffered consistency differs from other consistency models including weak ordering [21] and *release consistency* [4] as follows: Consider an NP-Synch operation such as a lock. An implementation of this operation in a cache-based environment may require sending a request to a global directory. An acknowledgment from this directory may signal the acquisition of the lock. However, the global completion of this operation may entail updating cache directories distributed in all the processors. Our model allows the requesting processor to continue with its local computation as soon as the acknowledgment is received without waiting for the operation to be globally performed. Similar arguments apply for CP-Synch operations. In a large-scale shared memory multiprocessor the weakening of the ordering constraint proposed in our model may be crucial to reducing the waiting time for synchronization operations to be globally performed.

3 Issues in Implementing Buffered Consistency

Implementation of weak consistency models requires that either the hardware or the software keep track of updates to shared data so that these updates may be propagated at appropriate points during the execution of the program. To ensure that all the updates inside a critical section be globally performed before the critical section is released, the hardware or the software should be able to detect these pending operations. In this paper we propose efficient hardware primitives for implementing scalable shared memory multiprocessors. The proposed hardware primitives are in the context of a cache-based system implementing the buffered consistency model. In order to implement this model the processor-cache interface should provide the following minimal capabilities:

- local reads and writes
- global writes
- wait until all global writes have been completed

With this interface the model is implemented as follows: Updates to shared data use global writes. Since synchronization variables need to be strongly consistent, the processor uses global writes for updating such variables, and in addition may choose to wait for the completion of such writes depending on the semantics of the synchronization operation. Before performing a CP-Synch operation, the processor waits until all global writes to shared data have been completed.

The above simple interface places the entire burden of implementing our model on the software. For example, the software has to distinguish between shared and private data, has to distinguish between synchronization variables and normal read/write variables, and when to wait for certain writes to be globally performed. From the performance standpoint this burden on the software may prove to be very inefficient.

The rest of this section identifies the issues that need to be addressed by the machine architecture for efficiently supporting this model.

1. There is a delay between the initiation of a global write and its completion. This delay depends on several factors including the number of nodes in the multiprocessor, the size and type of the interconnection network, and the amount of memory contention. Therefore, if there are successive global writes generated by a processor, then it would have to stall unnecessarily unless there is some kind of buffering of the global writes. Such a buffer would help smooth the traffic on the interconnection network as well as allow the local computation to pro-

ceed independent of network latencies (see Section 4.2).

2. In the model there is a necessity to keep track of pending global writes so that a processor may choose to wait on the completion of these writes. Adve and Hill [1] suggest a counter to denote the number of pending global operations. A processor may be stalled until the counter becomes zero when it waits for the completion of global writes. In a software approach [6], the definition and use of shared data are tracked by the compiler, and modified words in the cache are selectively written back and purged depending on the usage pattern of the program. However, this software approach is not practical since shared data may be accessed through pointers, and thus it is impossible to detect all the updates statically. The number of pending operations in the write buffer which we propose in our machine architecture (see Section 4.2) implicitly implements the counter of Adve and Hill [1].

3. The simple interface requires the software to specify each write as local or global. This requirement can be eliminated if the cache distinguishes between shared and private data. The cache would perform the write locally if the data is private and globally if the data is shared. However, this scheme forces all writes to shared objects to be global. In reality, the software is the best judge as to when a shared object has to be globally updated. Thus in our proposed machine architecture (see Section 4.2) the cache does not distinguish between shared and private data. The software is responsible for instructing the cache when writes to shared objects have to be performed globally.

4. As we mentioned in Section 1, there are two approaches to propagating the effect of a global write in a cache-based system: invalidate, and update. Both these schemes are coherence initiated by the writer. Using the buffered consistency model, when a writer updates some shared variables globally, these values may be needed by some readers in the future. If an invalidation approach is used, then such readers would have to request these values again. On the other hand, if an update approach is used, the updates may be sent to readers who may no longer be interested in these values.

   In general, it has been observed that in strongly consistent cache-based systems invalidations are less frequent than updates for the same memory reference pattern [2, 9]. In spite of this observation, the performance of write-update schemes has been comparable to that of the invalidation schemes in bus-based systems since invalidations generate more overhead than updates[1]. However, in large-scale multiprocessors employing more scalable interconnects, the network transit time is the dominant cost. In such networks the increased number of network transactions due to updates would make these schemes much less attractive than invalidation-based schemes [4, 3]. However, invalidation cache schemes have their sources of inefficiency as illustrated in Section 4.1. This observation motivates our new reader-initiated coherence protocol.

5. The buffered consistency model assumes that there are two types of accesses: synchronization and normal read/write. The simple interface places the burden of making this distinction on the software. Parallel programs invariably use some form of synchronization for coordinating access to shared data. Efficient synchronization is important to assure good performance. In our machine architecture (see Section 4.3), we propose synchronization primitives that are merged with the coherence protocol.

6. A cache line usually consists of several words and the cache protocol treats a line as the unit of consistency maintenance. Treating a cache line as the unit of consistency maintenance introduces a problem for the buffered consistency model: If two processors update different words in the same cache line and if the writes are delayed the word that is written back to memory first is lost. In the absence of any hardware support, the software is burdened with having to allocate shared variables carefully to avoid such situations. In the extreme, the software may be forced to allocate one shared variable per cache line, which is inefficient both from the point of view of spatial locality as well as usage of cache memory. In our machine architecture, we provide a mechanism to relieve the software of this burden. This mechanism also solves the problem of false sharing encountered in multiprocessor cache protocols.

The software is still responsible for the following:

- determining when to instruct the cache to perform writes globally,
- determining when to stall the processor waiting for global writes to be completed,
- distinguishing whether a synchronization operation belongs to the CP-Synch or the NP-Synch class and taking the appropriate actions, when software-based synchronization is used,

[1] An invalidation is usually accompanied by a line transfer.

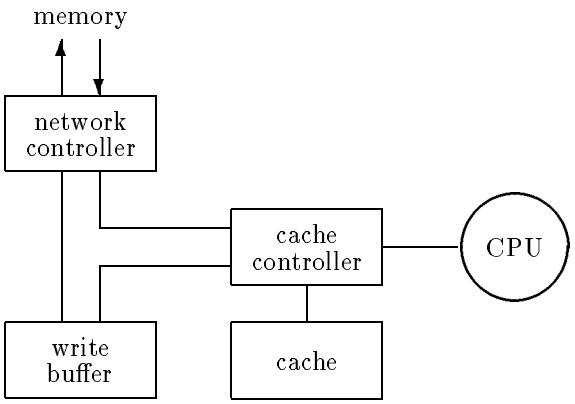


Figure 1: Block diagram of a node

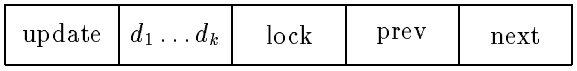
- determining how to exploit the reader-initiated coherence protocol, and the cache-based synchronization primitives from the performance point of view.

4 A Machine Architecture

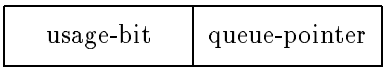
In this section, we propose a machine architecture that can be utilized by the software for implementing our buffered consistency model. Figure 1 shows the conceptual block diagram of a node in the multiprocessor architecture. Each node is equipped with a cache and a write-buffer. Each cache has a local directory referred to as *cache directory*. An entry in this cache directory contains the state of the associated cache line. The main memory is shared and all the nodes in the multiprocessor are connected to it via their respective network controllers. The main memory maintains a *central directory* for each line of the main memory. The location of the main memory and the interconnection network connecting the processors to the main memory are left intentionally unspecified since the machine architecture to be described does not depend on these details. For example, it is conceivable that the main memory is partitioned and distributed among the nodes in the multiprocessor, and the interconnection network is a multi-stage one. Table 1 summarizes the hardware primitives available to the processor. These primitives may be grouped in the following categories: read/write, buffer management, and synchronization. The next few subsections describe the features of the components in the node architecture.

4.1 Reader-Initiated Coherence

Figure 2, illustrates the structure of an entry in the cache directory and the corresponding entry in the central directory. To address the issues of false-sharing and the shared variables allocation (see Section 3) every entry in the cache directory has dirty bits, $d_1 d_2 \cdots d_k$, for each of the k words in the cache line.



a. An entry in the cache directory



b. An entry in the central directory

Figure 2: Structure of directory entries

When a cache line is replaced, only the dirty words are written back to memory. The *update* bit of cache directory entry and the *queue-pointer* field of the central directory entry are used by the read-update primitive, to be described shortly. The *lock* field is used for cache-based synchronization which will be explained in the next subsection. We chose to use a pointer-based directory structure since it is more scalable than either a full-map or limited directory structures [24].

Read and write requests are deemed for private data, and are treated as would a uniprocessor cache. Read-global bypasses the local cache and retrieves the data from the main memory[2]. The write-global request differs from the write request in that the operation is performed globally. This primitive is similar to the *post* primitive proposed by Cytron et al [6]. Read-update request is similar to read except that it requests future updates for this cache line. This is a dual to the write-update schemes [25, 17] in that the updates are receiver initiated as opposed to sender initiated. A read-update request is serviced locally by the cache if the update bit of the cache line is already set. Otherwise, this bit is turned on and this request is forwarded to the main memory. Processors that issued read-update requests for the same memory block form a doubly-linked list, and the *queue-pointer* field in the central directory contains the address of the head of the list. The linked list is constructed using the *next* and *prev* fields of each cache line corresponding to the requested memory block. When the main memory is updated, the updated block is transferred using this linked-list structure.

The update bit of a cache line is reset when the line is replaced by the node, or by an explicit request, *reset-update* for this line from the node. Subsequent writes to this line would not result in updates being sent to this node. The reset-update request will delete the requesting node from the linked list. Since the read-update request is considered to be mutually exclusive with a lock request for the same memory block, those fields that are used for constructing a linked list can be used for

[2]This primitive was motivated by discussions with Richard Huff, Cornell University.

Instruction	Operations
READ	retrieve data without coherence maintenance
WRITE	write data without coherence maintenance
READ-GLOBAL	read data from the main memory, bypassing local cache
WRITE-GLOBAL	write data globally
READ-UPDATE	retrieve data and request the main memory to send updated value
RESET-UPDATE	cancel the request for updated value
FLUSH-BUFFER	stall the processor until all the requests in the write-buffer are globally performed
READ-LOCK	request a shared lock for a data
WRITE-LOCK	request a exclusive lock for a data
UNLOCK	release the lock

Table 1: Hardware primitives

operation	read-update	inv-I	inv-II
initial load	$\lceil n/B \rceil C_B$	$\lceil n/B \rceil C_B$	nC_B
write	$C_W + (n-1)\|\|C_B$	$\frac{1}{B}(C_R + (n-1)\|\|C_I) + \frac{B-1}{B}(2C_R + 2C_B)$	$C_R + (n-1)\|\|C_I$
read	-	$\frac{1}{B}(\lceil n/B \rceil - 1)C_B + \frac{B-1}{B}\lceil n/B \rceil C_B$	$(n-1)C_B$

Table 2: Performance of cache coherence schemes for executing a linear equation solver.

lock operations (see Section 4.3). The *usage bit* in the central directory indicates whether the linked list is for read-update or lock operations. The cache-based lock presented in Section 4.3 also uses the same linked-list structure for processors waiting for a lock.

The read-update scheme differs from a write-update scheme in several ways: In the latter, whenever a read operation is performed it is remembered forever until the line is replaced by the reader. So readers continue to receive updates even if the line is not actively used. In our scheme, readers have to explicitly specify that updates are required using the read-update primitive. Moreover, a smart compiler could selectively determine regions in the program where updates may be needed.

Given these primitives, it would be instructive to analyze their possible usage. We compare our scheme against invalidation-based cache protocols for executing a linear equation solver. A linear equation can be expressed as $Ax = b$ where A is an n by n matrix and x and b are n-element vectors. The algorithm to solve this equation is as follows:

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})/a_{ii}$$

In each iteration, $x^{(k+1)}$ is computed, and the computed x value is used in the next iteration by all the other processors. All the processors are synchronized at the end of each iteration. For simplicity of analysis, we assume dance-hall architecture with n processors. Furthermore, the analysis is focused only on the global operations of the x vector required in each iteration. Table 2 shows the overhead in terms of network traffic generated by each processor. B denotes the cache line size, and $C_B, C_W, C_I, and C_R$ denote block transfer, word transfer, invalidation, and transaction carrying no data, respectively. To show the effect of false-sharing, two cases for invalidation protocol are considered: one for colocating x vector elements (inv-I) , and another for allocating each x element in separate cache lines (inv-II). When some number of transactions (say p) can be performed in parallel, they are denoted as $p||transaction\_type$. The costs for initial loading are obvious. With read-update feature, each write of an iteration sends the updated word to the main memory, and the main memory sends the memory block to $n-1$ processors that issued read-update request for that block. For inv-I, there are B writers to the same cache line. The first writer invalidates all the other copies, and the next $B-1$ writers retrieve the cache line from the previous writer. Though separate allocation of the x vector (inv-II) reduces the overhead for write operation, read of the next iteration will incur more overhead than the inv-I scheme. All the schemes perform comparably for write operation. However, read operation of the next iteration results in a significant advantage for the read-update scheme because the invalidation-based schemes have to re-load all the elements of the x vector.

4.2 Buffer Management

Implementation of the buffered consistency model is facilitated by the operation on the write-buffer, namely, *flush-buffer*. As we mentioned earlier, the write-global primitive requires that the operation be performed globally. However, to reduce the latency for global operations such requests are immediately buffered in the write-buffer. Depending on the availability of the interconnection network these writes are performed globally by the write-buffer without stalling the normal operation of the node. As and when an acknowledgment is received from the main memory, the associated buffer entry is deleted from the write-buffer. The flush-buffer primitive stalls the processor until all the buffered writes have been globally performed. This primitive allows the processor to wait for the completion of global operations which may be required by the program before executing a CP-Synch operation.

The read-update primitive combined with the buffered consistency model is a powerful feature. The basic functionality of the read-update primitive is similar to write-update primitive. However the fact that readers can selectively request updates to selected lines is expected to provide a significant performance advantage in implementing several parallel algorithms. For example, in parallel Fast Fourier Transform programs, readers may need access to different regions of a shared data structure during different phases of the computation. In implementing such algorithms, the program may selectively reset the update bit for certain regions of the shared data structure and request the regions to be used in the current computation phase using the read-update primitive.

4.3 Synchronization

Thus far we have proposed a buffered consistency model, and an efficient cache scheme that implements this model to address the two main problems of latency and network contention in the design of large-scale shared memory multiprocessors. An associated problem that also limits the scalability are synchronization operations that are quite heavily used in parallel program design.

For memory-based synchronization, the hardware usually provides some form of an atomic read-modify-write operation that allows higher level primitives to be built. However, such low level primitives could be quite inefficient in large-scale shared memory multiprocessors. The inefficiency caused by synchronization is twofold: wait times at synchronization points and the intrinsic overhead of the synchronization operations. If a busy-wait discipline is used, then the processors generate considerable memory traffic on the busy-wait variable. To reduce this traffic, a busy-wait on the cached copy of the busy-wait variable has been proposed for multiprocessors with coherent private caches [20]. However, when a mutual-exclusion lock is released, competition to acquire the lock results in bursty traffic to the same memory location. This contention impedes the useful computation that is being performed by the processor that has acquired the mutual exclusion lock thus prolonging the total execution time of the parallel program. In this section, we propose a cache-based lock scheme (CBL), that reduces the effects of lock contention. This scheme is similar to the one that we proposed in in our earlier work [13].

Exclusive and shared locks are common synchronization abstractions used in parallel programs. This abstraction is quite general and can be used for implementing other synchronization abstractions. In our design we have chosen to support this abstraction. The synchronization primitives provided by the cache are: *read-lock*, *write-lock*, and *unlock*. Each lock is associated with a cache line, and read-lock gives non-exclusive access to the line, while write-lock gives exclusive access to the line. When a lock request is granted the data associated with this lock is also transferred to the requester thus merging the data transfer with the synchronization request. Similar to the implementation proposed in our earlier work [13], a distributed hardware queue is constructed using participating cache lines.

Figure 3 shows the result of a sequence of lock requests generated by nodes P1, P2, and P3 for a memory location i; P1:read-lock,P2:read-lock, P3:write-lock. Only the cache lines and the memory block that contain the memory location i are shown in the Figure. The memory block is assumed to be of the same size as the cache line. A doubly-linked list is constructed using pointers of the participating cache lines and the central directory as shown in Figure 3. The *prev* and *next* pointers in each cache line denote the previous and the next node in the lock request sequence, respectively. The corresponding central directory entry has a pointer, *queue-pointer*, which points to the last lock requester.

First P1 sends its read-lock request to the main memory. Assuming that the memory block is currently unlocked, P1 is the only outstanding lock requester, and thus the address of P1 is stored in the queue-pointer field of the central directory entry. The memory block is sent to P1. A cache line is selected to store this memory block, the lock field of the associated cache directory entry is set to read-lock, and the prev and next pointers are set to nil. Receiving the read-lock request of P2, the request is forwarded to the current tail (P1), and the queue-pointer field of the central directory is changed to P2. Since the lock types of P1 and P2 are compatible, P1 allows P2 to share the lock. Now, the next pointer of P1 is set to P2 and the prev pointer of P2 is set to P1. Subsequently, when P3 makes a request for an exclusive access to the same memory block, P2 notifies P3 to wait at the tail of the queue. Figure 3

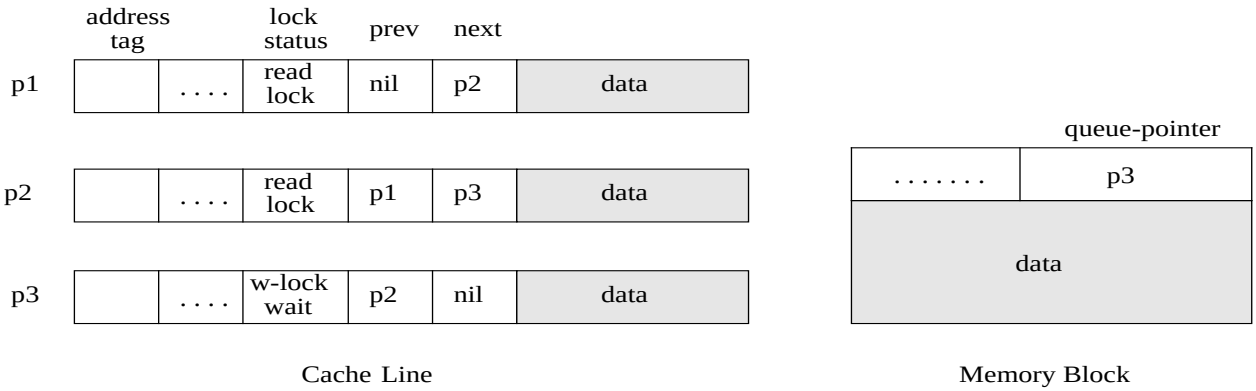


Figure 3: A waiting queue: doubly linked list

shows the final state after all these lock requests have been processed by the central directory[3].

Upon an unlock request, the cache releases the lock to the next waiting processor (if any), and writes the cache line to the main memory (if necessary). When a write-lock is released there could be more than one processor waiting for a read-lock. The lock release notification goes down the linked list until it meets a write-lock requester (or end of the list), and thus, allows granting of multiple read-locks. When a processor unlocks a read-lock and the processor is not the sole lock owner, the list is fixed up similar to deleting a node from a doubly-linked list. Note that the unlocking processor is allowed to continue its computation immediately, and does not have to wait for the unlock operation to be performed globally.

Replacing a cache line that is part of a linked-list may result in breaking the list. The most simple and straightforward solution is to disallow replacement of such cache lines. This solution would require the cache to be fully associatively mapped. Since such a mapping increases the hardware cost and introduces longer delays, this solution is unacceptable. Since a processor holds (or waits for) only a small number of locks at a time, a small separate fully-associative cache for lock variables would be an efficient method to eliminate this restriction. Limited size of the lock cache can be considered as a typical resource management problem, and should be handled as such. The lock cache is a "hardware resource" that can be judiciously used by the system software. Mapping of software locks to hardware locks is a compile time decision that can be made conservatively to ensure that there never will be the case that a hardware lock request cannot be satisfied.

With respect to the buffered consistency model, read-lock and write-lock are NP-Synch class of operations, while unlock is a CP-Synch class of operation. It is appropriate to mention a few usage notes for these primitives. When the size of the data structure to be governed by a lock fits within a memory block, acquiring the lock brings the associated data structure to the requesting processor. If the data structure spans several memory blocks, it is the responsibility of the compiler to associate locks and regulate accesses to the shared data structure. The compiler is also responsible to ensure that multiple lock variables are not allocated to the same memory block. The scheme does not prevent colocating normal read/write data with a lock variable in the same memory block.

[3]There are several subtle details that are intentionally elided due to space constraints. Detailed algorithms for maintaining the queue can be found in [14]

5 Performance Implications

Given the issues raised in Section 3, and the architectural features presented in Section 4 there is an interplay of hardware and software that makes the performance evaluation task quite arduous. In this section, we take a first cut at this task. Our preliminary study has two parts: The first part deals with evaluating the advantage of providing synchronization operations in hardware as opposed to simulating them in software. This evaluation is done by both analytical means as well as simulation. Analytical expressions for the cost in terms of time and network messages are developed for implementing standard synchronization scenarios using our primitives and a write-invalidation (WBI) approach. The second part deals with evaluating the performance advantage of buffered consistency as opposed to sequential consistency for processing a memory reference stream. This evaluation is done using simulation. Simulation of a large-scale system at the level of detail needed to evaluate the proposed primitives is a complex task worthy of exposition in its own right. Moreover, such large simulations take considerable computation time.

5.1 Analytical Results

The overhead in executing various synchronization

synchronization operation	WBI		CBL	
	messages	time	messages	time
parallel lock	$6n^2+4n$	$nt_{cs}+10nt_{nw}+$ $n(n+1)/2t_m+$ $5n(5n-1)/2t_D$	$6n-3$	$nt_{cs}+(2n+1)t_{nw}+$ $(n+1)t_D+t_m$
serial lock	8	$8t_{nw}+5t_D+t_m+t_{cs}$	3	$3t_{nw}+t_D+t_{cs}$
barrier request	18	$18t_{nw}+12t_D$	2	$2(t_{nw}+t_m)$
barrier notify	$5n-3$	$4t_{nw}+(2n-1)t_D$	n	$2t_{nw}+(n-1)t_D$

Table 3: Cost for executing synchronization scenarios with different cache schemes. Costs for serial lock and barrier request are for one processor.

scenarios are presented in Table 3: n is the number of processors, t_{nw} is the network transit time, t_{cs} is the time inside the critical section, t_D is the time to check the central directory or the cache directory, and t_m is the time for reading a memory block from the main memory. *Parallel lock* is the case when n processors request the same lock simultaneously. *Serial lock* is the other extreme case when all the lock requests occur serially. *Barrier request* is an operation executed by each processor participating in the barrier synchronization while *barrier notify* is an operation executed by the last processor to arrive at the barrier. Note that for a large amount of lock contention (parallel lock) the time and message complexity of our scheme is $O(n)$ while it is $O(n^2)$ for the WBI scheme. Detailed derivation of these cost functions can be found in [15].

5.2 Simulation Results

A new workload model (*work-queue* model), is used in our simulation studies. This model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [19]. The basic granularity is a task. A large problem is divided into atomic tasks, and dependencies between tasks are checked. Tasks are inserted into a work queue of executable tasks honoring such dependencies, thus making the work queue non-FIFO in nature. Each processor takes a task from the queue and processes it. If a new task is generated as a result of the processing, it is inserted into the queue. All the processors execute the same code until the task queue is empty or a predefined finishing condition is met. If there is a need to synchronize all the processors at some point, then a barrier operation is used. In the simulation, the memory modules are distributed among the nodes in the multiprocessor, and the nodes are interconnected via a multistage Ω network with two-way switches. It is assumed that each switching element in the network has infinite buffer capacity. The size of the write buffer is also assumed to be infinite.

Parameters	value
ratio of shared accesses	0.03, 0.5*
number of shared blocks	32
cache hit-ratio	0.95
read ratio	0.85
main memory cycle time	4 cache cycles
block size	4 words
cache size	1024 blocks
lock ratio	50%

* 0.03: task execution, 0.5: queue access

Table 4: Summary of parameters used in simulation

To simulate the memory reference pattern of each processor during task execution, a probabilistic model (*sync model*) similar to the one developed by Archibald and Baer [2] is incorporated into our model. Additional features in our model are synchronization primitives, differentiation of synchronization variables from other variables, and different evaluation metrics. The values of the parameters used in the simulation are summarized in Table 4. Another important parameter is the grain size of parallelism. The grain size is decided by the number of data memory references during the execution of a task. The performance metric used is completion time measured in machine cycles. Though processor utilization is measured in [2], synchronization activities may keep the processor busy without performing any useful computation.

The Figures 4 - 7 present simulation results for two different workload models. The lines with the prefix Q denote the results for the work-queue workload model, and the other lines denote the results for the sync workload model. Figures 4 and 5 show the completion time of the WBI scheme and the CBL scheme. WBI denotes the write back invalidation cache scheme. The performance of the WBI scheme with the exponential backoff for acquiring mutual exclusion is also presented (Q-backoff). These tests do not employ buffered consistency. Figure 4 shows that the WBI scheme does not scale well for the work-queue workload model above 16 nodes when the granularity of parallelism is medium-sized. Backoff method eliminates the severe performance loss but it also fails to scale to a large number of processors. The CBL scheme performs comparably with the WBI

scheme for the sync workload model (two lines at the bottom). However, for the work-queue workload model, the CBL scheme shows much better performance for a large number of processors. Increasing the task granularity reduces the proportion of time spent in synchronization activity compared to the total computation time. For coarse granularity of parallelism (Figure 5), the WBI scheme shows improved scalability for the work-queue model but its performance degrades as the size of the system increases to more than 32 nodes. These two figures illustrate the effectiveness of implementing synchronization in hardware.

The performance gain due to the buffered consistency model depends on the usage of CP-Synch operations and global writes. In the simulation model, leaving a critical section and barrier synchronization are treated as CP-Synch operations; and writes to shared data are treated as global writes. Further, it is assumed that an unlock operation that follows a lock operation performs any write to the shared data secured by the lock operation globally before relinquishing the lock. The test performed here is a comparison of CBL with buffered consistency (BC-CBL) against CBL with sequential consistency (SC-CBL). The reason why only CBL is considered for the test is because the intent is to measure the performance potential of buffered consistency with respect to sequential consistency, without interference from specific cache coherence strategy. Figures 6 and 7 show the completion times of CBL with the two memory models for the work-queue workload. These results indicate that buffered consistency improves the performance for most tested cases, but the improvement is not very impressive. This can be explained by the fact that buffered consistency reduces memory latency for global writes which happen only with a probability of $sh \times write\_ratio$, i.e., 0.0045 in the tested workload.

6 Concluding Remarks

There are two visible trends in the evolution of multiprocessor architectures, namely, message-passing and shared memory. While the former is intrinsically more scalable than the latter, the latter fits the programming paradigm that is currently popular for developing large parallel applications. In this paper we have shed some light on the issues that affect the scalability of shared memory multiprocessors and have suggested architectural solutions that address these issues. The two main issues are latency for memory accesses and the network contention stemming from these accesses. Private coherent caches alleviate these problems, and their design is well-understood in small shared memory multiprocessor systems (up to about 16 processors). However, the memory model assumed by these designs and the protocols themselves limit their applicability to large-scale shared memory multiprocessors. Further, such protocols introduce additional problems such as false sharing that hamper the performance potential of such architectures. A new memory model, buffered consistency, was developed in this paper that allows the processor to continue with its local computation without stalling for the completion of global updates. We also identified the hardware support needed for implementing buffered consistency. An important benefit of supporting the buffered consistency model is that false sharing is also eliminated. A new cache protocol based on reader-initiated coherence was proposed. We also extended our earlier work of providing synchronization support in hardware to large-scale shared memory multiprocessors. We feel that the machine architecture we have proposed in this paper has the right blend of features for scaling to large numbers of processors.

In general, evaluation of architectural features is an arduous task. Especially, in our architecture we have identified a division of responsibility between the hardware and the software. This division coupled with the choice of primitives leads to a complex interplay making performance evaluation that much more difficult. We have made some initial attempts at performance evaluation through analytical means and simulation and have presented the results. There are several directions for extending our work. Evaluating the performance advantages of eliminating false sharing, and reader-initiated coherence are important and are being currently pursued. Trace-driven simulation is another alternative to probabilistic simulation and is also being investigated. Our architecture provides a range of primitives for use by the compiler. Investigating compilation techniques, and/or programming language extensions that exploit these primitives are also part of our future research.

References

[1] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–11, May 1990.

[2] J. Archibald and J. Baer. Cache coherence protocols: evaluation using a multiprocessor model. *ACM Transactions on Computer Systems*, pages 278–298, Nov. 1986.

[3] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, pages 49–58, June 1990.

[4] K. Charachorloo, D. Lenoski, J. Laudon, and A. Gupta. Memory consistency and event ordering

in scalable shared-memory multiprocessors. Technical Report CSL-TR-89-405, Stanford University, Computer Systems Laboratory, Nov. 1989.

[5] D. A. Cheriton, H. A. Goosen, and P. D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.

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

[8] J. Edler, A. Gottlieb, Cl. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Telen, and J. Wilson. Issues related to MIMD shared-memory computers : the NYU Ultracomputer approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, June 1985.

[9] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–382, June 1988.

[10] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. Technical Report TR-814, Univ. of Wisconsin at Madison, Jan. 1989.

[11] Mark D. Hill and James R. Larus. Cache consideration for multiprocessor programmers. *Communication of ACM*, 33(8):97–102, August 1990.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[13] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, May 1990.

[14] Joonwon Lee. *Architectural Features for Scalable Shared Memory Multiprocessors.* PhD thesis, College of Computing, Georgia Institute of Technology, 1991.

[15] Joonwon Lee and Umakishore Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. Technical Report GIT-CC-91/10, College of Computing, Georgia Institute of Technology, 1991.

[16] K. Li and R. Schaefer. Shiva: An operation system transforming a hypercube into a shared-memory machine. Technical Report 217-89, Computer Science Dept., Princeton University, 1989.

[17] E. McCreight. *The Dragon Computer System: An early overview.* Xerox Corp., Sept. 1984.

[18] G. F. Pfister and V. A. Norton. Hotspot contention and combining in multistage interconnection network. *IEEE Transactions on Computers*, C-34(10), Oct. 1985.

[19] C. D. Polychronopoulos. *Parallel Programming and Compilers*, pages 113–158. Kluwer Academic Publishers, 1988.

[20] L. Rudolph and A. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[21] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June. 1987.

[22] Charles L. Seitz. The cosmic cube. *CACM*, 28, January 1985.

[23] G. S. Sohi, J. E. Smith, and J. R. Goodman. Restricted fetch-and-ϕ operations for parallel processing. In *International Conference on Supercomputing*, June 1989. Crete, Greece.

[24] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–25, June 1990.

[25] C. P. Thacker and L. C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Oct. 1987.

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

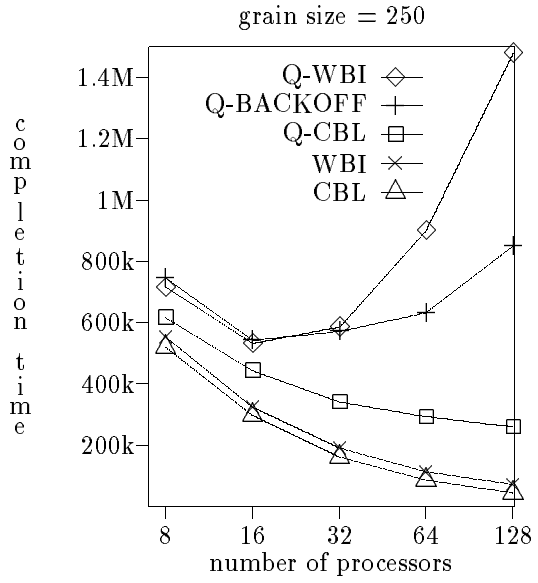


Figure 4: Performance of cache schemes with medium-granularity parallelism

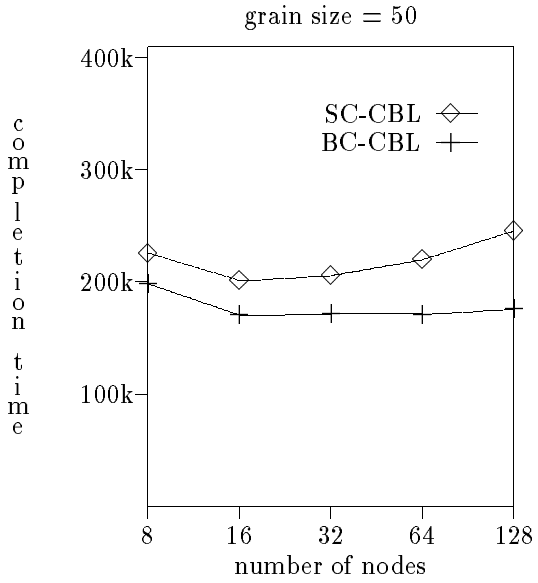


Figure 6: Performance implication of the buffered consistency model for fine-granularity parallelism

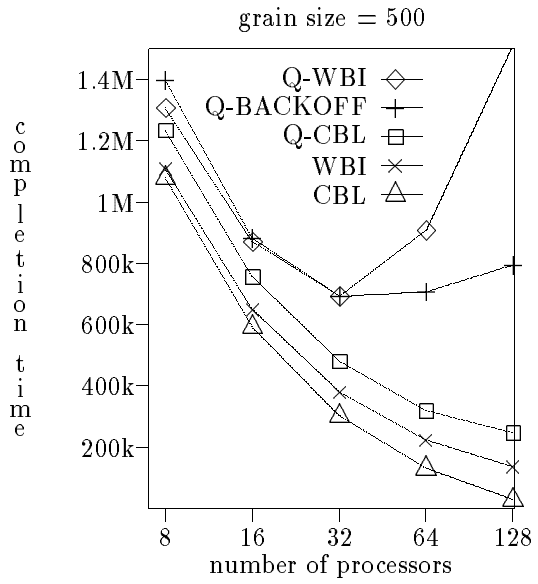


Figure 5: Performance of cache schemes with coarse-granularity parallelism

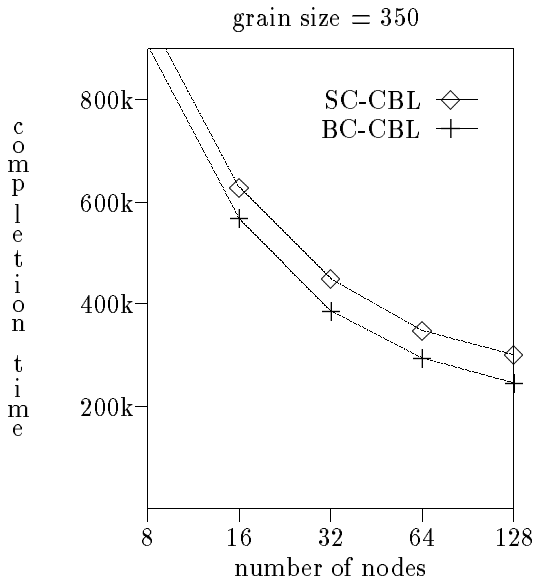


Figure 7: Performance implication of the buffered consistency model for medium-granularity parallelism