Cache-based Synchronization in Shared Memory Multiprocessors [*]

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

Joonwon Lee
Dept. of Information and Communication
KAIST, Seoul Campus
Seoul, Korea 130-650

To appear in *The Journal of Parallel and Distributed Computing.*

[*]This work is supported in part by an NSF PYI Award MIP-9058430.

Running Head: Cache-based Synchronization in Multiprocessors.

Corresponding Author: Dr. U. Ramachandran, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280. Phone: (404)894-5136. e-mail: *rama@cc.gatech.edu.*

Abstract

In shared memory multiprocessors, efficient synchronization is imperative to assure good performance. There are two aspects to the "cost" of a synchronization operation: the first is the waiting time at synchronization points; and the second is the intrinsic overhead in performing the operation. The overhead has two components. The first component deals with contention resolution for synchronization operation among competing processors. The second component deals with the shared data accesses that the processor has to perform once it enters a synchronization region. We present a mechanism to reduce the overhead of performing synchronization operations in a cache-based shared memory multiprocessor. The mechanism is based on the intuitive notion that parallel programs invariably use synchronization operations to govern the access to shared data. Traditional multiprocessor cache protocols treat synchronization accesses the same way as normal read/write memory accesses, leading to inefficiencies in performing synchronization operations which ultimately limit the scalability of such systems. The key idea in our approach is to combine synchronization with the coherence maintenance for the cached data. Each cache line maintains states for synchronization as well as for cache coherence, and the cache protocol ensures the correctness of the synchronization operations and the coherence of the data at these synchronization points. To assess the performance gain due to the proposed mechanism, simulation studies are performed using a workload model that represents a dynamic scheduling paradigm which forms the core of several parallel programs. Results from simulation studies show that the proposed cache-based synchronization performs significantly better than traditional cache coherence approaches.

Key Words:
Shared memory multiprocessors, Cache Coherence, Cache-based locks, Queue-based locks, Multiprocessor workloads, Synchronization, Simulation, Performance Evaluation

1 Introduction

Synchronization operations in a parallel program could lead to waiting times and network contention independent of the choice of the network or the availability of parallel communication paths in the network. Efficient synchronization is imperative for multiprocessors since parallel programs tend to generate repeated requests for mutual exclusion, barrier, and operations on shared data structures. The inefficiency caused by synchronization is twofold: wait times at synchronization points and the intrinsic overhead of the synchronization operations. Reducing waiting time is to some extent in the province of the programmer, but reducing synchronization overhead is a task for the computer architect. Shared memory multiprocessors usually incorporate a private cache with each processor to hide the latency for remote memory accesses, and handle the resulting cache coherence problem for shared data in either software or hardware (Section 2). It is interesting to note that even with private caches with fairly high hit ratios, the scalability of bus-based machines is limited to at most tens of processors [5, 18]. This limitation may be directly attributed to the network traffic that is generated as a result of the cache coherence and synchronization activities leading to a saturation of the bus. While the use of direct networks in shared memory multiprocessors can reduce the average latency for memory accesses if the applications are properly structured to exploit communication locality [2], the inefficiency due to synchronization effects still persist.

Usually, access to shared data is acquired via synchronization methods such as locks, semaphores, and barriers. Thus there is additional delay in accessing the synchronization variables and then acquiring the actual data. This observation leads us to design cache-based locking schemes that combine synchronization with the data coherence maintenance (Section 3). Evaluation of parallel system performance is interesting and quite difficult. Developing a workload model that is representative of parallel program behavior is a challenging task and we make an attempt at this task (Section 4).

The focus of this paper is to present our cache-based locking primitives, its hardware implementation for snoopy based and directory based caches, and their impact on performance through simulation studies. The key contributions of our research are summarized below:

1. A simple and efficient mechanism that combines synchronization with data coherence in a cache-based shared memory multiprocessor is proposed. The primary motivation for this mechanism is that parallel programs invariably use some sort of synchronization model to govern access to shared data. In a cache-based multiprocessor, this fact can be exploited to reduce both the waiting time as well as the network traffic by providing a hardware primitive that uses the synchronization information to provide coherence for shared data. The proposed mechanism constructs a hardware FIFO queue (using the cache lines of participating processors) of concurrent synchronization requests, and satisfies the requests in order trans-

ferring the data associated with the synchronization. The proposed mechanism essentially gives synchronization for free.

2. An implementation of the above mechanism in a broadcast bus as well as in an arbitrary interconnection network is presented.

3. A novel workload model that captures the salient features of shared memory parallel programming is developed. Simulation results using this workload model are presented that quantifies the performance benefits of cache-based synchronization for a bus-based and MIN-based implementation.

4. Our results indicate that an efficient handling of barrier synchronization is important for a medium size multiprocessor while an efficient parallel queue is imperative for a large scale multiprocessor.

2 Cache Coherence Protocols

There are three aspects to consistency maintenance of private caches in multiprocessors: the *protocol*, the *organization of the state information*, and the *enforcement* of the protocol either by software or hardware.

Protocol. There are two broad classifications of protocol families based on the semantics for dealing with processor writes: *write-invalidate* and *write update*. In write-invalidate schemes [21, 29, 39], a write to a cached line results in invalidating copies of this line present in other caches. In write update schemes [36, 49], a write to a cached line results in updating copies of this line present in other caches.

Bitar and Despain [8] propose a scheme in which the cache accepts lock and unlock commands from the processor in addition to the traditional read and write requests. This lock scheme combines lock-based synchronization with the line transfer, thus performing locking in zero time. It does not distinguish between read lock and write lock requests, and also does not order the processors waiting for the lock. KSR-1 [43] – a commercial shared memory multiprocessor, as well as Paradigm [13] – a research prototype from Stanford, provide primitives very similar to the one proposed by Bitar and Despain. In [22], Goodman et al. suggest a synchronization primitive, QOSB, which can be used by the programmer for constructing high level synchronization operations. Similar to the lock primitive proposed by Bitar and Despain, QOSB is an implicit exclusive lock that can be associated with any main memory block. The main semantic difference is that the data in this memory block has no significance from the point of view of the program. Goodman et al., present an efficient implementation of this primitive by constructing a queue of the processors waiting for the lock using the cache lines in the participating processors and the data field of the memory block. Since reads are a large portion of shared data accesses, all these schemes limit potential concurrency by considering only exclusive locks.

State Information. There are two ways to organize the state information (i.e. the set of active sharers of a cache line) that is needed for taking coherence actions, depending on the capabilities of the interconnection network: recording only the state (i.e. shared, private, etc.) of the cache line in the private caches and relying on a fast broadcast medium to effect the coherence actions; or keeping the active sharers centrally with the particular memory block and taking coherence actions using point to point communication. The former regime is usually referred to as *snoopy* caches and is popular in bus-based systems. The latter regime is called *directory-based* caches. Directory cache schemes maintain a centralized table associated with the shared memory for the location of the caches that have a copy of a shared data item. Thus, this information makes it possible to locate all the shared copies without the need for a broadcast capability which snoopy caches rely on. For this reason, coherence protocols based on a directory associated with the shared memory for maintaining the state information are usually preferred for large-scale shared memory multiprocessors [33, 10]. Invalidation-based protocols have been found to be preferable for large-scale multiprocessors relying entirely on hardware to provide cache coherence.

There are several ways to organize the state information in the central directory [10, 48]. One possibility is to associate a bit map with each memory block, where each bit represents the presence or absence of that memory block in a particular processor. A variant of this basic scheme is to use a limited set of pointers instead of a bit map. When the level of active sharing for a memory block exceeds the set of available pointers, the system may either emulate full map in software [11], or invalidate one of the cached copies to make room for the new request, or resort to broadcast. Another variant is to make the pointer allocation dynamic [35, 46]. Yet another variant is to maintain the state information in the form of a linked list with the memory block as the head and the set of participating cache lines forming the active chain of processors sharing that memory block [27, 32]. There are space/time trade-offs, and scalability implications depending on the particular choice of organizing the state information. The reader is referred to [48] for a more complete survey of the pros and cons of different directory based schemes.

Software Enforcement of Coherence. In software-controlled cache schemes, cache lines are invalidated or updated according to the compiler's static analysis of parallel programs. There have been several proposals [12, 16] to determine the cacheability of shared data through compile-time flow analysis for specific language constructs such as FORTRAN *doall*. Min and Baer [38] suggest a time-stamp based cache protocol that allows run-time determination of whether or not a cached data item is valid. In Beehive [45], synchronization information is used in developing marking algorithms that determine when global operations have to be performed to ensure coherence for shared data. These software assisted cache coherence schemes split the responsibility between hardware and software: the hardware provides the mechanisms needed for coherence maintenance and the software implements the policy as to when coherence actions have to be carried out.

3 Cache-Based Lock

As we observed in Section 1, shared memory style parallel programs tend to generate frequent requests for synchronization operations such as barriers and locks. While the programmer is responsible for increasing the concurrency and consequently reducing the waiting times at synchronization points, the computer architect's task is to reduce the synchronization overhead. Hardware support for synchronization comes in various forms such as a special-purpose coprocessor (e.g. Sequent SLIC [6]), a combining network [23], and a special bus for interprocessor communication [53]. As we observed earlier, Bitar and Despain [8] were the first to propose exclusive locks (with no queuing) as a cache primitive. Goodman et al. [22] and ourselves [41] were the first to independently propose queue-based locks in hardware as a cache primitive. While our primitives support both exclusive (write) and shared (read-shared) locks, Goodman et al. support only exclusive locks. Further our primitives allow combining data transfer with the granting of the lock (similar to Bitar and Despain) thus providing synchronization for free. Subsequently, similar locking primitives have made their way into commercial machines (such as KSR-1 [43]), as well as university prototypes [33].

It is easy to show [30, 32] that a straightforward implementation of mutual exclusion locks (using test&test-and-set) and barriers (decrementing a shared counter on arrival and spinning locally on a shared flag for completion) could result in $O(n^2)$ network traffic (where n is the number of processors) irrespective of whether invalidation or write-update is used for cache coherence. Recently, there have been several software proposals for reducing the the network traffic in the implementation of locks and barriers in cache coherent shared memory multiprocessors. Anderson [4] and Graunke and Thakkar [24] have independently proposed algorithms that use software queues for ordering the lock requests from processors. These algorithms eliminate the bursty network traffic that is possible in a straightforward implementation of mutual exclusion locks under high lock contention. Hensgen et al. [25] and Mellor-Crummey and Scott [37] have proposed efficient algorithms that bring down the amount of network traffic generated during barrier synchronization operations from $O(n^2)$ (using the straightforward counter based algorithm) to $O(nlogn)$. In spite of reducing the network traffic, these software algorithms still have to pay the latency penalty (in the form of memory accesses and machine instructions for emulating the locks in software) for synchronization operations. This overhead can be a significant constant factor even with the best software algorithm for mutual exclusion locks,[1] and as much as $O(nlogn)$ even with the best software barrier algorithm.

Therefore we think that supporting synchronization primitives in hardware is a good idea. Locks are a very general synchronization abstraction with which other more special synchronization abstractions such as barriers can be efficiently built. In our design, 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. A processor blocks until its lock request is granted before generating new requests. 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.

For the purposes of this paper, simple read/write operations of a processor are assumed to be non-interfering with the locking primitives. To simplify the discussion of the locking primitives we do not discuss simple read/write operations in the rest of the paper. However, it should be clear that such locking primitives can be incorporated as part of the processor-cache interface co-existing with the read/write cache primitives (see [32] for an example). In Section 4, where we discuss performance benefits of our cache-based locks, we assume a Berkeley-style invalidation-based protocol for private accesses and shared accesses made without an explicit lock association.

3.1 Rationale for Read-lock

Read-sharing is common in most shared memory parallel applications. Further, guaranteeing that the data does not change during such read-shared phases is important in several applications. Any application that requires a 2-phase locking semantics (such as airline reservation, distributed calendar, and banking systems) will experience reduced concurrency if only exclusive locks are available for shared data structures. While read-locks can be emulated in software using exclusive locks, we have chosen to support it in hardware since the overhead for emulation may be limiting on the performance especially if this primitive is used extensively as is likely given the range of applications that need such a primitive. Besides, the hardware primitive we provide allows transferring the associated data when the lock is granted thus reducing both the number of messages as well as the latency for lock operations. In the next two subsections, we present implementations of CBL for snoopy caches and for directory-based caches.

[1]The number of machine instructions needed to implement Anderson's ticket lock [4] is roughly 50 for lock acquisition, and about 15 for lock release on the Sequent Symmetry and the KSR-1 [42].

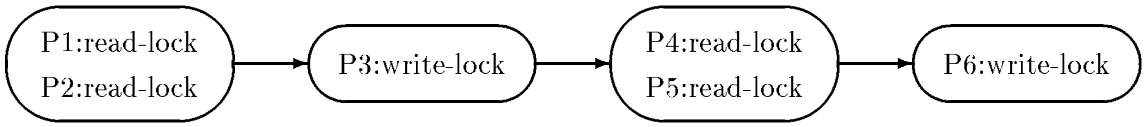

Figure 1: A waiting queue of lock requests

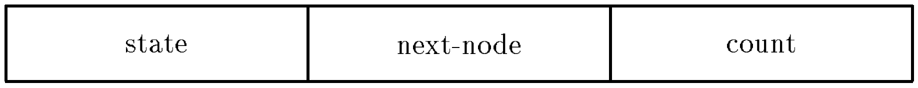

Figure 2: Cache Directory Entry Fields

3.2 Snoopy Cache Implementation

In this subsection, we present an overview of a lock-based snoopy cache protocol supporting exclusive and non-exclusive locks. Like the queueing mechanism used in QOSB [22], a distributed FCFS queue is constructed using participating cache lines.

The processor and the cache together form a node of the shared memory multiprocessor. Each node is assigned a unique id which we refer to as *node-id*.

Consider the sequence of lock requests, P1:read-lock, P2:read-lock, P3:write-lock, P4:read-lock, P5:read-lock, P6:write-lock, as shown in Figure 1. The first requester (P1) obtains a read-lock, and the following requester (P2) shares the lock since the lock type is *read*. P3 waits for the lock because its lock type is *write*. P4 and P5 wait after P3 to ensure fairness, even though the current lock held by P1 and P2 is sharable. A *peer-group* is a group of read-lock requesters who concurrently share a lock ({P1,P2} and {P4, P5} are peer-groups). To implement a queue, each directory entry of a cache line has a *next-node* field containing the node-id of the next waiting cache if any. The protocol described in this subsection assumes a single process per processor. Snoopy CBL that allows multiple processes per processor is presented in [31].

In the discussion to follow, we use lock and line interchangeably since lock acquisition is merged with the cache line transfer. Each cache line has a directory entry with the fields as shown in Figure 2. A cache becomes the *owner* for a cache line when it acquires a lock for that line (for example P1 in Figure 1). A cache at the end of the waiting queue for a lock is denoted as the *tail*, and is responsible for responding to subsequent requests on the bus for that lock (for example P6 in Figure 1). Only the first requester (denoted as *leader*) within a peer-group can be an owner and/or a tail even when the lock is shared. The next-node field (see Figure 2) in the leader points to the next peer-group (if any) in the waiting queue. For the other members of a peer-group their respective next-node fields point to the leader of the peer-group. A shared lock is released when the size of the peer-group reaches zero, so caches with read-lock ownership or awaiting ownership keep the size of the peer-group in a *count* field. The ownership persists even after the line is unlocked at the owner cache. As each reader unlocks the cache lines, the count is decremented and when it goes to zero the ownership is transferred to a waiting write-lock requester (if any, otherwise the default owner is the memory). Similarly, when a writer unlocks, the line is written back to memory and the ownership is transferred to a peer-group leader. Transfer of ownership is accomplished by broadcasting the node-id of the leader stored in the next-node field of the current owner. Since the bus is monitored by all the caches the other members of the peer-group (if it is a read-lock peer-group) simultaneously acquire the lock by matching their respective next-node field with the broadcast node-id. Assigning read-lock ownership to the first requester may result in unnecessary cache entries since it is likely that the read-lock owner will be the first to release the lock. However, the alternative choice of giving ownership to the last one in a peer-group could generate more bus traffic to transfer the count variable to the new owner. The width of the count field is determined by the number of nodes in the system. Alternatively, the width may be determined by the maximum membership we would like to allow in a peer group.

For the implementation of this protocol, we assume the existence of additional signals on the bus to indicate whether a cache line is sent by a peer cache or memory, and to signify whether a requesting cache is joining a peer-group or assuming owner- and/or tail-ship to a cache line. See Appendix A for a description of the state transitions needed for implementing this protocol in a bus-based system.

3.3 Directory-Based Caches

In this subsection we present an implementation of CBL in multiprocessors with directory-based caches. Similar to the implementation of snoopy CBL, a distributed hardware queue is constructed using participating cache lines. Since this implementation does not assume any specific interconnection network, queue maintenance is performed using message passing. The most distinct problem that is present in this protocol is the management of link information in the face of non-deterministic network delay.

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 *previous* and *next* pointers in each cache line denote the previous and the next nodes in the lock request sequence,

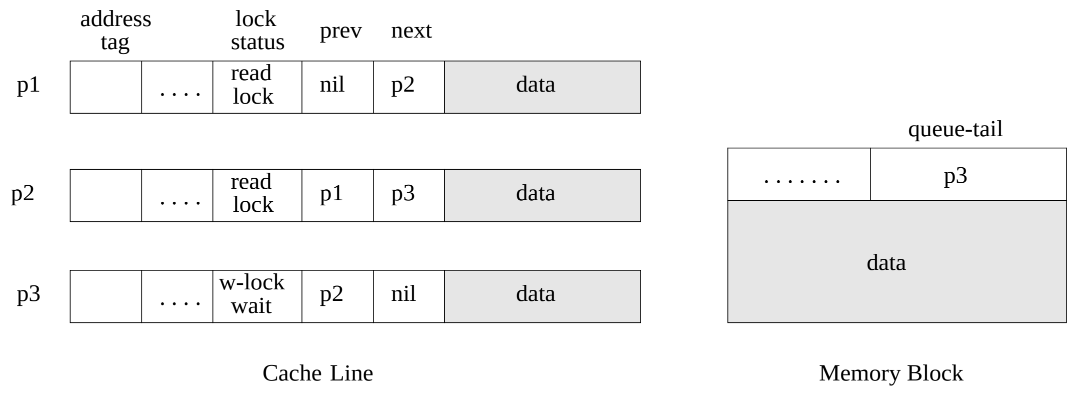

Figure 3: A queue of nodes waiting for a lock: doubly-linked list

respectively. The corresponding central directory entry has a pointer, *queue-tail*, 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-tail 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-status field of the associated cache directory entry is set to read-lock, and the *previous* 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-tail 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 *previous* 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 shows the final state after all these lock requests have been processed by the central directory.

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.

3.4 Maintenance of Distributed Queues

The linked lists are maintained by sending appropriate messages for addition and deletion of nodes on lock and unlock requests. Because of network and memory contention, a message sent from a

Figure 4: Inserting a node at the tail of a linked-list

Figure 4 illustrates the addition of a node to a linked-list. Upon a new lock request (1 in Figure 4) from the processor P_{new}, the main memory updates the queue-tail pointer in the central

Figure 5: Deleting a node at the tail of a linked-list

Deleting a node at the tail of a linked-list is shown in Figure 5. The node that initiates this operation sets its transient bit first (1 in Figure 5). The unlock message (2 in Figure 5) contains the node-id of the previous node, the node-id of the requester, and information indicating that the request is from a tail node. The main memory ensures that the requester is still the current queue-tail (otherwise the request is rejected), updates the queue-tail pointer (3 in Figure 5), and sends a message to P_{prev} informing the change of tail (4 in Figure 5). Upon receipt of this message, P_{prev} changes its *next* pointer (5 in Figure 5), and sends an acknowledgment (6 in Figure 5) to P_{tail}. Receipt of the acknowledgment resets the transient bit in P_{tail}. Until the acknowledgment is

Figure 6: Deleting a node in the middle of a linked-list

received, P_{tail} does not entertain any pointer update request from P_{prev} as we mentioned earlier. However, new lock requesters are allowed to join the linked-list while this distributed transaction is in progress and does not affect the correctness of this deletion operation.

Deleting a node from the middle of a linked-list does not involve the main memory and is straightforward as shown in Figure 6. As before the initiating node sets its transient bit (1 in Figure 6). It then sends a message to its next node (2 in Figure 6) requesting it to change its *prev* pointer. This request is honored if the next node itself has not initiated an unlock operation (3 in Figure 6). If the request is honored (4 in Figure 6), then an acknowledgment is sent (5 in Figure 6). Upon receipt of this acknowledgment the initiating node sends a message to its previous node to change its *next* pointer (6 in Figure 6). As we mentioned earlier this request is always honored thus preventing any deadlocks. Then the transient bit is cleared in the initiating node (7 in Figure 6).

The detailed algorithms for maintaining the linked-lists are given in Appendix B. Only the algorithms for read-lock and read-unlock operations are given since the ones for write-lock and write-unlock operations are similar. Though these algorithms do not require any specific interconnection network, they assume the FIFO (first-in first-out) queueing discipline for messages between any source-destination pair in the network.

3.5 Discussion

Cache based locks have some implications on data allocation and cache line replacement. Also, the requirements for an efficient implementation of CBL are worthy of exposition and are discussed in this subsection.

3.5.1 Data allocation

To exploit spatial locality of memory references, a cache line usually consists of several data items and the cache protocol treats a line as the unit of consistency maintenance. Since CBL combines synchronization with data coherence maintenance some care has to be exercised in the data allocation. Multiple shared variables that are accessed independently and simultaneously under the protection of locks cannot be co-located in the same cache line, if the intent is to use the implicit cache-based locks for governing their accesses. The compiler can do a flow analysis to determine the usage pattern of lock variables in the program. But such static analysis may not be able to track the simultaneity of lock usage through pointer variables (if in fact the programming language allows such usage). A conservative solution would be to ensure that program defined locks are mapped to distinct cache lines. However, private variables can be co-located with shared variables in the same cache line.

It is appropriate to mention a few usage notes for the CBL primitives. When the size of the shared 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 shared data structure spans several memory blocks, the implicit lock can be associated with the first memory block. The remaining memory blocks can be accessed without the protection of locks similar to private data structures. The implicit assumption is that the CBL primitives co-exist with whatever coherence protocol is in effect for data accessed without the protection of the lock. For instance, we assume in this paper that a Berkeley-style invalidation protocol is in effect for such accesses assuring data integrity for these accesses.

3.5.2 Cache line replacement

Replacing a cache line that owns a lock is a bit more complicated. The most simple and straightforward solution is to disallow replacement of a cache line that owns a lock. However, this solution may be feasible only if a fully associative or a set-associative caching strategy is in use. 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.

If a direct mapped caching scheme is used then it is necessary to modify the memory system to hold the tag and queue specific fields of the cache line in addition to the data field. We believe this can be done with a little added complexity to the cache controller, minimal change to the memory system, and support from the compiler. The compiler can allocate additional memory space for every shared read/write data structure for holding the auxiliary information. For instance, if the cache line is 4 words, one word may be reserved by the compiler for storing the auxiliary information. The memory system has a bit for every block that indicates whether the block is locked or not, which is returned with every memory request. When a cache line is replaced, the cache controller checks the status of the cache line. If it is locked, then the cache controller writes back the data field along with the auxiliary information to the memory system. When the block is reloaded from the memory, the cache controller extracts the auxiliary information returned in the data field and stores it in the appropriate fields of the cache line. When a processor makes a lock request and the block comes from the memory, the cache controller checks the lock bit. If it is set then it is an indication that some other cache currently holds the lock. Therefore, the requesting cache has to retry the lock request at a later time.

The preceding description is only conceptual. In an actual implementation, it is not necessary to modify the memory system at all to hold the lock bit information with each block; nor is there any necessity for an extra line on the interconnect to transmit this lock bit to/from memory and the caches. The cache controllers have to agree to designate a specific bit in each memory block to be the lock bit and the compiler has to reserve this bit for this purpose during data allocation. Keeping the lock bit "in band" thusly would allow the use of conventional RAMs and eliminate the need for any special purpose memory system design.

3.6 Complexity

For a bus-based implementation, the CBL scheme requires 13 states in the cache controller compared to 4 in most other protocols, and a larger cache tag memory to implement the distributed hardware queue. In [31], we quantified the hardware complexity of the state machine for CBL (through VLSI implementation with a PLA) to be roughly an order of magnitude greater than those of either the Berkeley or Dragon protocols (see Table 1). But compared to the total complexity of the cache controller (bus interface, associative hardware for matching, tag memory, etc.) irrespective of the specific coherence protocol, the absolute size of this state machine PLA is a small overhead and hence tolerable for the performance gain that it offers (see Section 4).

	Berkeley	Dragon	Snooping CBL
Input Lines	12	14	21
Output Lines	23	23	26
Product Terms	47	78	768
Total Size	35×47	37×78	47×768

Table 1: Size of PLAs for cache controllers

For snoopy caches, we have shown [31] that CBL can be implemented easily on state-of-the-art bus systems such as Futurebus [9]. For directory-based caches, our protocol assumes no specific capability in the interconnection network. In fact, the implementation can be optimized if some specific interconnect is assumed [17]. Though the CBL scheme enables efficient handling of synchronization, the hardware complexity of the scheme would be significant, especially for general interconnection networks that incur non-deterministic network delays. We have presented detailed algorithms for the CBL scheme (see Appendix B) and their correctness have been tested by extensive simulations. However, we have not identified the apportionment of responsibilities between hardware and software for a multiprocessor implementation. The minimal hardware requirements would include the directory structures in the main memory and cache lines. The rest of the functionality for implementing CBL has to be examined carefully to determine the performance loss (if any) of a software simulation. This evaluation requires the measurement of hardware complexity for each functionality and performance loss due to the software substitution, and is part of ongoing research.

4 Performance Evaluation

Several techniques can be applied for evaluating multiprocessor performance: hardware measurement, simulation, and analytical modeling. While hardware measurement provides an accurate estimate of multiprocessor performance for the measured benchmarks, the feasible tests are limited to a small number of case studies. Therefore, hardware measurement is more often used with other techniques such as analytical modeling and simulation [14, 28, 26].

4.1 Analytical Models

Analytical modeling provides initial estimates of performance but has limitations owing to the assumptions that are often made to make the model tractable. Vernon et al. [50] report on the performance of snoopy cache protocols using the MVA (mean value analysis) technique. The results from this analysis are surprisingly close to the simulation results of Archibald and Baer [5] while the required computation time is significantly smaller than simulation. Though their equations are very intuitive and easy to explain in terms of the mechanics of the architecture, they need to be validated through simulations when applying to other domains due to the intrinsic property of MVA approaches in considering only average conditions for the parameters instead of probability distributions. One main limitation of analytical methods to multiprocessor evaluation is capturing synchronization activity reasonably in a workload. The bursty memory requests caused by synchronization cannot be easily modeled using the available approaches. In our study, simulation is mainly used for performance evaluation.

4.2 Simulation

Simulation is computationally intensive compared to analytical modeling but it has the advantage that the interactions that cannot be mathematically modeled can be captured very well. Trace-driven, execution-driven, and probabilistic are the three methods of generating the workload for the simulation. Trace-driven simulation has some validity concerns as observed by several researchers [3, 19, 20, 7] due to the distortions that may be introduced due to the instrumentation code that is inserted for collecting the traces. These distortions include non-uniform slowdown of the parallel processes due to varying amount of tracing code in each process; and overall slowdown in the execution speed of all processes owing to tracing. Since the execution path of a parallel program depends on the ordering of the events in the program, both these distortions have the potential of completely changing the execution path unless timing dependencies are carefully eliminated from the traces [20]. Program startup effects may also distort the results especially if the trace length is not long. Further the traces obtained from one machine may not represent true interactions in another machine. Lastly, the traces usually do not capture OS related activities (such as interrupts, context switches, and I/O) unless hardware instrumentation is available [3]. Execution-driven simulation is closest to capturing the true interactions between the program and the underlying architecture being simulated and in this sense may be the best technique if simulation is to be used. But in general this technique is more difficult since it requires collecting/developing several real programs. Also it involves simulating all aspects (such as instruction-sets) of the underlying architecture and not just the aspects one is interested in studying. Augmentation [15] is an attractive approach to supplement execution-driven simulation so that only interesting aspects of the target machine are actually simulated and others are executed on the host machine. In any event, it should be noted that both trace-driven and execution-driven simulations necessarily represent performance specific to the traces/programs that are actually studied and cannot predict the trends for other program behavior (sharing pattern, synchronization pattern, task granularity, etc.).

In this study we have chosen a probabilistic approach for the following reasons:

- it allows the input parameters to be more carefully controlled than either of the other two approaches.
- it is a good technique for getting a more accurate (than analytical modeling) prediction of performance, and yet is less demanding in terms of storage, and computation resources compared to the other two simulation approaches.
- it can be done without a huge investment in program/trace collection and/or development that is involved in the other two simulation approaches.

A criticism often leveled against such a probabilistic model is that, the workload may not be representative of any real parallel programs. However, we address this criticism by developing a workload model (to be presented next) that captures the basic computational paradigm of shared memory parallel programs. Given such a general framework, if specifics of an application (such as sharing pattern, etc.) are known, then they can be plugged into this framework as parameters of the simulation. We believe this approach gives a more controlled simulation environment for studying the impact of the proposed features on the performance of parallel systems.

4.3 Workload Model

We present a new two-level workload model for the evaluation of multiprocessors. This model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [40]. Data parallel applications consist of several phases of computation, and each phase uses the results generated by the previous phase. Therefore, when such an application is implemented using the dynamic scheduling paradigm, all the processors (executing the parallel threads of the application) would need to synchronize using a barrier at the end of each phase. Examples for this class of applications include FFT (fast fourier transformation), linear equation solver, and numerous CAD programs [18]. There are other classes of applications that do not need such a barrier. For example, the traveling salesman problem can be implemented as a tree search problem. Executing a node in the search tree may generate several children nodes. Since the execution of each node is independent of other nodes in the queue, the processors need not synchronize with one another after executing a node. Most search problems in artificial intelligence belong to this class.

The higher level of the workload is a *work queue* model. The basic granularity of parallelism 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 (process scheduling) 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. Figure 7 shows the pseudo code that each processor executes, and represents the higher-level model of the workload.

```
finished := false;
new-task := empty;
loop
    /* Queue operations in a critical section */
    lock(queue);
    if (new-task $\neq$ empty) then
        /* previous loop execution generates a new task */
        insert-queue(new-task);
    end if;
    task := delete-queue();
    unlock(queue);
    new-task := execute(task);
    if (need-synchronization()) barrier();
    finished := check-if-finished();
until(finished)
```

Figure 7: Higher level work-queue model

The lower level of the workload is the *sync* model that captures the memory reference pattern of each processor during the execution of a task ("execute(task)" in Figure 7). The sync model is a probabilistic model similar to the one developed by Archibald and Baer [5]. The main enhancements of the sync model, over that of Archibald and Baer, are the separation of synchronization accesses from normal read/write accesses and the use of overall completion time as the evaluation metric. Each processor generates a memory reference every w processor cycles, where w is a uniformly distributed random variable between 1 and 5. The reference stream of each processor contains references to shared data and references to private data. With probability sh, each memory reference is to a shared block. If the request is to a private block, it is serviced by cache memory with a probability specified by the *hit ratio*. For references to shared blocks, the block number is generated from a stack with higher probability of referencing blocks near the top of the stack. Once a block is referenced, it is placed at the top of the stack. Another parameter affecting the performance of a cache protocol is nsh, the number of shared blocks per processor. As this number increases, less

Parameters	value
ratio of shared accesses	0.03, 0.5*
number of shared blocks	32
cache hit-ratio	0.95
read ratio	0.85
memory cycle time	4 processor cycles
cache cycle time	1 processor cycle
block size	4 words
cache size	1024 blocks
lock ratio	50%

* 0.03: task execution, 0.5: queue access

Table 2: Summary of parameters used in simulation

sharing activities are expected because it is less probable that multiple processors would access the same block for a given period of time.

Many parameters are fixed not only because their effects are well studied in [5] but also our primary concern was to measure the effect of various synchronization mechanisms on protocol performance. The values of the parameters used in the simulation are summarized in Table 2. The degree of active sharing is expected to be fairly low during the execution of a task than during queue operation. Since the sync model generates references for each processor independently, it is impossible to control the degree of active sharing directly. We simulate this increased level of sharing by tweaking the ratio of shared accesses parameter. With respect to our two-level workload model (see Figure 7), we set the ratio of shared accesses to total accesses to be 0.03 during the "execute" phase of a task, and 0.5 during insertions of items into and/or deletions of items from from the queue. Real traces [18, 51] show that the read ratio ranges from 0.6 to 0.9 depending on the application of traced programs. In [5] simulation was done varying the read ratio from 0.7 to 0.85. In our simulation the read ratio is set to 0.85.

Shared accesses are secured by lock primitives with a probability *lock-ratio*. The nature of the lock can be sharable (read) or exclusive (write) depending on the type of shared access (i.e. this is also governed by the parameter read-ratio). The ratio of shared accesses under the control of a lock to all shared accesses varies from 50% to 70% in some applications [1, 34] and below 10% in other applications [18]. In the results reported in this section this ratio (lock-ratio in Table 2) is set at 50%. Varying the lock-ratio produces only a small difference in performance because the degree of sharing is quite low (0.03, see Table 2) during task execution. For invalidation and write update schemes, the lock primitive is implemented as test&test-and-set. 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.

Private accesses are handled probabilistically, while shared accesses are simulated exactly commensurate with each particular cache protocol. Synchronization operations including locks (using test&test-and-set and test&test-and-set with exponential backoff) and barrier (using a central counter and a global wakeup flag) are also simulated exactly. In the case of CBL, we assume a Berkeley-style invalidation-based protocol for shared accesses made without the protection of a lock. The simulator is written in CSIM [44]. A simple FIFO queue is used to simulate shared bus service for snoopy CBL. For the directory based CBL, a packet switched Ω network is assumed as the interconnect (owing to its generality) with infinite buffering in the switches which are 2 X 2 crossbar. The bus as well as the links in the MIN are assumed to be 32 bits wide.

4.4 Simulation Results

Though processor utilization is often measured in other studies [5], we measured the *completion time* (in processor cycles) for executing a given workload because synchronization sometimes makes processors busy even when they are not doing any useful computation. The CSIM simulation tool generates various statistics including average queue length and server utilization. We present those numbers only when it is necessary to justify the completion times. Since the performance studies of different cache configurations are available in [5, 52, 51], we restrict our study to the understanding of the effects of different synchronization scenarios.

4.4.1 Snoopy Cache Protocols

Snooping cache schemes evaluated are the Berkeley protocol [29] as an invalidation scheme, the Dragon [36] protocol as a write update scheme, Bitar and Despain's method [8], and our lock-based protocol. The problem size is 1024 tasks, and the results are the average of 5 runs.

Figure 9 shows the completion time (in units of 10,000 cycles) of each cache scheme for the grain size 100 (fine to medium grain) without the barrier synchronization. BD denotes the Bitar and Despain's scheme, and CBL is our cache-based locking scheme. The Berkeley protocol shows an anomalous loss of efficiency as the number of processors n increases beyond 8. This loss of efficiency happens because the bus is already saturated (the measured average length of the bus queue supports this argument) and thus, useful work is delayed by queue access activity of each processor. This effect is the multiprocessor equivalent of thrashing induced by the greedy scheduling discipline. All the other schemes also show slowdown with more than 8 processors. With 8 processors CBL completes in 20480 cycles while BD, the second best one next to CBL, completes in 25149 cycles. This performance gap grows as n increases. The performance gap between BD and CBL is due to increased concurrency provided by read-locks of CBL during the task execute phase of the simulation model (Figure 7).

Figure 11 depicts the performance with a large grain size of 500. The steep loss of efficiency with the Berkeley protocol disappears in this case because the long processing time outweighs the overhead of queue access. When n is 8 the performance gap between CBL and BD is 14303 cycles, which is more than three times the gap when the grain size is fine to medium. The experimental results (on the Sequent Symmetry) reported in Reference [47] corroborates our simulation results.

The effect of barrier synchronization is shown in Figures 10 and 12. Irrespective of the specific cache protocol, the net effect of the barrier is to synchronize the queue access of all the processors thus aggrevating the contention for this shared resource (see Figure 7). Hence for a given grain size, the inclusion of the barrier results in longer completion times for all the protocols (compare Figures 9 and 10, and Figures 11 and 12). With respect to Figures 11 and 12, the Dragon protocol outperforms the Berkeley protocol by larger gaps with the barrier than when there is no barrier. The same is true in Figures 9 and 10, except for the anomalous behavior of the Berkeley protocol beyond 8 processors in Figure 9. Even though the barrier itself does not lead to significant performance gap, simultaneous lock requests (a potential $O(n^2)$ network traffic) after the barrier entails considerable expense for the Berkeley protocol as compared to the Dragon protocol, especially with large n. With $n = 8$ and fine to medium grain parallelism, Dragon is better than Berkeley by 8% when the barrier is not used, and by 32% when the barrier is used. The case when the grain size is large (500) shows a similar trend. CBL outperforms BD by a larger margin with the barrier synchronization. When the grain size is fine to medium (100), CBL is better than BD by 4669 cycles if the barrier is not required, and by 9249 cycles with the barrier. The efficiency of CBL for the barrier operation comes from the sharable lock that enables the notification to be done in one bus transaction.

These simulation results indicate the shared bus cannot be used for more than 8 processors without an efficient synchronization scheme. Our proposed snoopy cache CBL scheme shows significant performance gain across all the measured size of multiprocessors. This performance gain is conservative in the sense that each lock is assumed without an argument. When lock requests are used with arguments, the CBL scheme combines synchronization with the actual data transfer in one operation while other cache protocols would generate two separate global requests. The CBL scheme is expected to give even more performance advantage under such circumstances.

4.4.2 Directory Cache Protocols

Completion times for executing the given workload with directory cache schemes are presented in Figures 13 and 14. We do not consider a system with less than 8 processors because connecting a small number of processors with MIN makes little sense. WBI means a cache protocol with a write-back policy and invalidations for a shared write. The performance of a WBI scheme with exponential backoff for acquiring mutual exclusion is also presented. In this scheme, each processor doubles its mean delay after each failure to acquire the lock. There needs to be a maximum bound on the mean delay to eliminate an unnecessary long delay when there remain only a few processors in a lock competition [4].

Figure 13 shows the performance of each cache scheme for a medium grain parallelism. The upper three lines are for executing the work-queue workload model. The WBI protocol is slightly better then the backoff scheme when there is little lock contention (for less than 16 processors). This is because in the backoff scheme there is a possibility of the lock being idle between transfer of lock ownership due to the inherent nature of the backoff scheme. However, as the number of processors increases, the WBI protocol shows a steep loss of performance that is absent in the backoff scheme. Like the Berkeley snoopy cache scheme, the performance loss of the WBI can be explained by the network saturation. The performance gap between CBL and the other two schemes becomes larger as the number of processor increases. Similar to the performance gap observed in snoopy caches, as more processors are involved for the lock contention, the WBI scheme experiences $O(n^2)$ overhead compared to $O(n)$ overhead of the CBL scheme. The bottom two lines represents the case when the workload model is without the work-queue and without the barrier synchronization. This situation essentially amounts to using static scheduling with the lower-level sync model serving as the memory reference pattern generator. The performance gap between the WBI protocol and the CBL scheme becomes less significant as the lock contention disappears.

The case for coarse grain parallelism is shown in Figure 14. Overall trends are similar to those for medium grain size. With all other things being the same, it can be seen that the absolute completion times using the directory schemes (top three lines of Figure 14) are larger than those for the snoopy caches schemes (Figure 12) for a multiprocessor with 8 and 16 processors. This is mainly due to the multi-hop network delay of MIN that is used in the evaluation of directory cache schemes. However, while snoopy caches show little speedup beyond 8 processors, directory caches show substantial speedup until 32 processors.

Figure 15 and 16 present the performance implication of barrier synchronization and queue accesses, respectively. For a small number of processors (8 and 16), barrier synchronization is more detrimental than the queue contention. However, as more processors are added to the system, the queue contention becomes a major factor limiting the performance gain. These results indicate that an efficient handling of barrier synchronization is important for a medium size multiprocessor while efficient parallel queue is imperative for a large scale multiprocessor.

4.5 Discussion

From the simulations studies we observe that the major factor limiting the performance of shared memory multiprocessors is the overhead for handling synchronization requests. The scalability shown by the CBL scheme confirms this fact. The snoopy cache protocols fails to scale beyond 8 processors even with a workload that does not include queue contention and barrier synchronization (see Figures 9 and 11) mainly due to bus saturation. However, the directory-based cache protocols are scalable to a large number of processors with such a workload (bottom two lines in Figures 13 and 14) due to the multiple communication paths available in the MIN.

More efficient software synchronization algorithms [37] are not compared in this simulation study. First of all, it is shown in [37] that for snoopy caches a central counter based barrier does the best in terms of measured performance for a modest number of processors. In a sense, the performance of CBL gives an upper bound for the performance of synchronization algorithms which use efficient queue maintenance in software *a la* the MCS algorithm in [37]. The scalability of such software synchronization algorithms is presented in [37]. Our work puts the results reported by [37] in context by considering a workload that contains shared and private accesses in addition to the synchronization accesses. As we mentioned in Section 3, software synchronization algorithms incur an overhead for queue maintenance in software (despite reducing the network traffic) that is not present in CBL. Further, to be fair to the other cache protocols which we compared CBL against, we deliberately did not consider combining data transfer with the granting of the lock that is possible with CBL. This feature of CBL cannot be obtained with any of the software synchronization algorithms. Quantifying this performance benefit is part of future research.

5 Concluding Remarks

An important issue in the design of shared memory multiprocessors is scalable solutions for synchronization operations. We presented a cache-based lock scheme which combines data transfer with synchronization. Lock operations are used as the underlying primitives for cache coherency. As a waiting mechanism, a distributed queue is constructed in hardware using the cache line of each lock requester. The protocol distinguishes between sharable lock and exclusive locks thus allowing increased concurrency for simultaneous readers. Implementations of these protocols in a snoopy cache and directory-based cache system were discussed. The performance of the proposed primitives is presented through simulation. For the simulation studies, we developed a new workload model that represents the high-level structure of parallel programs. The simulation results indicate that an efficient handling of synchronization is crucial to shared memory multiprocessors both in a snoopy cache as well as a directory-based cache system. In particular, we observed that an efficient handling of barrier synchronization is important for a medium size multiprocessor while an efficient parallel queue is imperative for a large scale multiprocessor. Determining the split between hardware and software for directory-based CBL protocols, a more thorough performance evaluation of the proposed synchronization primitives, the incorporation of these primitives with the normal cache coherence mechanism in shared memory multiprocessors, and the compiler work needed to exploit these primitives are the areas of ongoing research related to this study.

Acknowledgements

The feedback from the arch-beer meetings at Georgia Tech was a key to improving the ideas presented in this paper. In particular, we wish to thank Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Dr. H. Venkateswaran for their comments on earlier drafts of this paper. The authors would also like to thank the anonymous referees for their comments, which helped us clarify the ideas presented in this paper and improve the quality of the presentation.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under MACH. In *ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 215–225, May 1988.

[2] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, Oct. 1991.

[3] Anant Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 186–195, 1986.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

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

[6] B. Beck, B. Kasten, and S. Thakkar. VLSI assist for a multiprocessor. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–20, 1987.

[7] P. Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. *ISCA '89 Workshop: Cache and Interconnect Architectures in Multiprocessors*, 1989.

[8] 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.

[9] P. Borrill. IEEE 896.1: the Futurebus. *Electron*, 33(10):628–631, Oct. 1987.

[10] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[11] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A scalable cache coherence scheme. In *Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[12] 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 138–145, Aug. 1988.

[13] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2):33–48, February 1991.

[14] D. W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1983.

[15] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[16] 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.

[17] Martin H. Davis, Jr. *Using Optical Waveguides in General Purpose Parallel Computers*. PhD thesis, College of Computing, Georgia Institute of Technology, November 1993.

[18] 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.

[19] Richard M. Fujimoto and William C. Hare. On the accuracy of multiprocessor tracing techniques. Technical Report GIT-CC-92/53, College of Computing, Georgia Institute of Technology, November 1992.

[20] Stephen R. Goldschmidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, May 1993.

[21] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[22] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[23] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Trans. Computers*, C35(2):175–189, 1983.

[24] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[25] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, February 1988.

[26] Mark D. Hill and Alan J. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 158–166, 1984.

[27] IEEE P1596 - SCI Coherence Protocols. *Scalable Coherent Interface*, March 1989.

[28] Rajeev Jog, Phillip L. Vitale, and James R. Callister. Performance evaluation of a commercial cache-coherent share memory multiprocessor. In *ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 173–182, 1990.

[29] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[30] 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.

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

[32] Joonwon Lee and Umakishore Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[33] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[34] Z. Li and W. Abu-sufah. A technique for reducing synchronization overhead in large scale multiprocessor. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 284–291, 1985.

[35] David J. Lilja and Pen-Chung Yew. Combining hardware and software cache coherence strategies. In *ACM International Conference on Supercomputing*, pages 274–283, 1991.

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

[37] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[38] S. L. Min and J. Baer. A timestamp-based cache coherence scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:23–32, 1989.

[39] M. Papamarcos and J. Patel. A low overhead solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

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

[41] U. Ramachandran and J. Lee. Processor initiated sharing in multiprocessor caches. Technical Report GIT-ICS-88/43, Georgia Institute of Technology, Nov. 1988.

[42] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the International Conference on Parallel Processing*, pages I–237–240, August 1993.

[43] Kendall Square Research. Technical summary, 1992.

[44] Herb Schwetman. *CSIM Reference Manual.* MCC Corp., March 1990.

[45] Gautam Shah and Umakishore Ramachandran. Towards exploiting the architectural features of beehive. Technical Report GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.

[46] R. Simoni and M. Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[47] Anand Sivasubramaniam, Gautam Shah, Joonwon Lee, Umakishore Ramachandran, and H. Venkateswaran. Experimental evaluation of algorithmic performance on two shared memory multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 13–24, April 1991.

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

[49] 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.

[50] Mary Vernon, Edward D. Lazowska, and John Zahorjan. An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988.

[51] W-D Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, 1989.

[52] W-D Weber and A. Gupta. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Workshop on Scalable Shared Memory Architecture*, May 1990. Seattle, WA USA.

[53] W. A. Wulf and C. G. Bell. C.mmp - a multi-mini processor. In *Proceedings of the Fall Joint Computer Conference*, pages 765–777, Dec. 1972.

A State Transitions for Snoopy CBL

This appendix describes the state transitions for CBL implementation in a bus-based environment.

States	Meaning
INVALID	The line is invalid
WO	Write lock owner.
WOT	WO at the tail.
WOV	Waiting for a write lock.
WOVT	WOV at the tail.
R	Read lock holder.
RV	Waiting for a read lock.
RO	Read lock owner.
ROT	RO at the tail.
ROV	Waiting for a read lock ownership.
ROVT	ROV at the tail
O	Unlocked, but still an owner.
OT	O at the tail.

Table 3: States of a cache line

The possible states of a cache line are summarized in Table 3, where **R,W** are used to specify the lock type, **T** to signify the tail of the queue, **V** to indicate waiting state, and **O** for the ownership. State transitions are triggered by processor requests and/or bus activities. The states in Table 1 apply only for shared lines obtained through lock requests. In the discussion to follow, we use lock and line interchangeably since lock acquisition is merged with the cache line transfer.

An owner cache has the latest copy of the line, so it provides the line to the other caches when requested. The line is written back to memory when a write-lock owner releases the lock. There is at most one owner of a lock even when the lock is shared. A lock state with a **T** suffix denotes that the cache is at the tail of the waiting queue and should respond to subsequent requests for that lock. Only the first requester within a peer-group can be a tail or an owner. A shared lock is released when the size of the peer-group reaches zero, so caches with read-lock ownership or awaiting ownership keep the size of the peer-group in a *count* field. The ownership persists even after the line is unlocked at the owner cache. Assigning read-lock ownership to the first requester may result in unnecessary cache entries since it is likely that the read-lock owner will be the first to release the lock. However, the alternative choice of giving ownership to the last one in a peer-group could generate more bus traffic to transfer the count variable to the new owner. The width of the count field is determined by the number of nodes in the system. Alternatively, the width may be determined by the maximum membership we would like to allow in a peer group. Each cache line has a directory entry (tag) with the fields as shown in Figure 2.

Figure 8: State Transitions

Figure 8 illustrates the state transitions. When a processor requests a read-lock, the cache broadcasts it on the bus resulting in one of following responses:

- The block came from the main memory (denoted as *hit(M)* in the state transition diagram). It means that the line is not locked by any cache. The memory system provides the line, and the cache changes the state to ROT since it is the first requester.

- The block came from another cache (denoted as *hit* in the state transition diagram). The current read-lock owner sends the cache line allowing the requester to share the lock. The receiving cache changes its state to R.

- A *wait* signal is detected on the bus. A peer cache in state ROVT sends this signal with its own node-id. Since the *wake* signal (to be discussed shortly) is addressed to the first requester in a peer-group, this node-id is necessary for the waiting nodes to receive the signal correctly. The receiving cache stores this node-id in the next-node field, and changes its state to RV.

- A *wait(T)* signal is detected on the bus. The signal comes from a cache in state WOVT or WOT, and signifies that the tail state is transferred to the requester. Therefore, the receiving cache changes its state to ROVT.

When a cache receives *read-unlock* from the processor, the state of the line is one of R, ROT, or RO. A cache line in the R state is simply changed to the state INVALID, and a read-unlock signal is broadcast on the bus to inform the owner to decrease the count. If the state is ROT or RO, it is changed to OT or O respectively after decrementing the count. The cache is still the owner even after its own processor releases the lock and is responsible for sending a wake signal when the count goes to zero. Even though for this discussion we assume a single process per processor, a processor may request a lock after releasing a lock, i.e, it may request a lock when the state of the line is OT or O. This case is not shown in Figure 8 since it is treated as a sub-case of multiple processes per processor (details in [31]). Another simple solution without increasing hardware complexity is to allow the processor to be an owner again. In this case, the fairness between processors cannot be guaranteed.

In case of a write lock, it is not necessary for the owner to keep the count since only one writer is allowed at a time. If a wait(T) signal is detected after broadcasting a write-lock, the state is changed from INVALID to WOVT. However, it ceases to be at the tail when any subsequent request for a lock is observed on the bus. On receiving an appropriate wake signal, the cache controller changes waiting states, WOV, WOVT, to owner states, WOT, WO respectively, and allows the processor to use the line. It is not necessary to broadcast a write-unlock. On receiving a write-unlock request from the processor, the cache changes the state of the line to INVALID, sends a wake signal enclosing the cache line to the next requester (if any) as indicated by the next-node field, and writes the line back to memory.

Bus signals include: read-lock, write-lock, read-unlock, hit, hit(M), wait, wait(T), and wake. The wait signal is sent from the tail cache to the lock requester when the lock is unavailable. The wait(T) signal additionally transfer the tail state to the requester. The wake signal is sent to notify that the lock is released to a cache whose node-id was stored in the next-node field of the tag entry for the line. A more detailed description of the protocol can be found in companion publications [31, 30].

B Algorithms for Queue Maintenance

```
read_lock(my_id);
local variables:  tail_node, next_node;
cache.transient := 1;
if (memory.lock = unlocked) then
      /* The memory is currently unlocked.  */
      memory.lock := read_lock;
      /* This node becomes the queue tail.  */
      memory.queue_tail := my_id;
      cache.data := memory.data;
      cache.state := read_lock;
else
      tail_node := memory.queue_tail;
      tail_node.next := my_id;
      /* Set the connection to the previous lock requester.  */
      /* Note that the transient bit of tail_node is ignored.  */
      cache.previous := tail_node;
      if (tail_node.state = read_lock)
            cache.state := read_lock;
            cache.data := tail_node.data;
      else
            cache.state := read_lock_wait;
      endif;
endif;
/* Reset the transient bit.  */
cache.transient := 0;
/* Check if another lock request has occurred after this node.  */
if (cache.next != NULL) then
      next_node := cache.next;
      if (next_node.state = read_lock_wait) then
            /* read_lock is shared */
            next_node.state := read_lock;
            next_node.data := cache.data;
      endif;
endif;
end read_lock;
```

```
read_unlock(my_id);
local variables:  tail_node, next_node;
if (cache.next = NULL) then
      /* This node is the tail of the queue.  */
      if (cache.previous = NULL) then
      /* This node is the only one in the queue.  */
            if (memory.queue_tail != my_id)
            /* There has been new lock requesters while this */
            /* transaction is coming to the memory, so abort it.  */
                  abort();
            else
                  /* reset the queue tail and release the lock */
                  memory.queue_tail := NULL;
                  cache.state := unlocked;
            endif;
      else
            /* This node is the tail of the queue, thus prevent new lock*/
            /* requesters from updating the next pointer of this node */
            cache.transient = 1;
            if (memory.queue_tail != my_id);
            /* other requests changed queue-tail before */
            /* this message arrives at the memory */
                  cache.transient = 0;
                  abort();
            else
                  memory.queue_tail := cache.previous;
            endif;
            /* make the previous node a new tail */
            cache.previous.next := NULL;
            cache.state := unlocked;
            cache.transient := 0;
      endif;
/* the else part is given in the following page */
```

(continued)

```
else        /* cache.state is not NULL */
      /* This node is not a tail */
      if (cache.previous = NULL) then
            /* This node is the head of the queue */
            /* Note that transient bit is NOT set.  */
            next_node := cache.next;
            if (next_node.transient = 1) then
                  abort();
            endif;
            next_node.previous := NULL;
            if (next_node.cache.state = wait) then
                  next_node.cache.data := cache.data;
                  wake_up(next_node);
            endif;
            cache.state := unlocked;
            cache.next := NULL;
      else
            /* This node is in the middle of the queue */
            cache.transient := 1;
            next_node := cache.next;
            if (next_node.transient = 1) then
                  abort();
            endif;
            next_node.previous := cache.previous;
            cache.previous.next := cache.next;
            cache.state := unlocked;
            cache.transient := 0;
      endif;
endif;
end read_unlock;
```

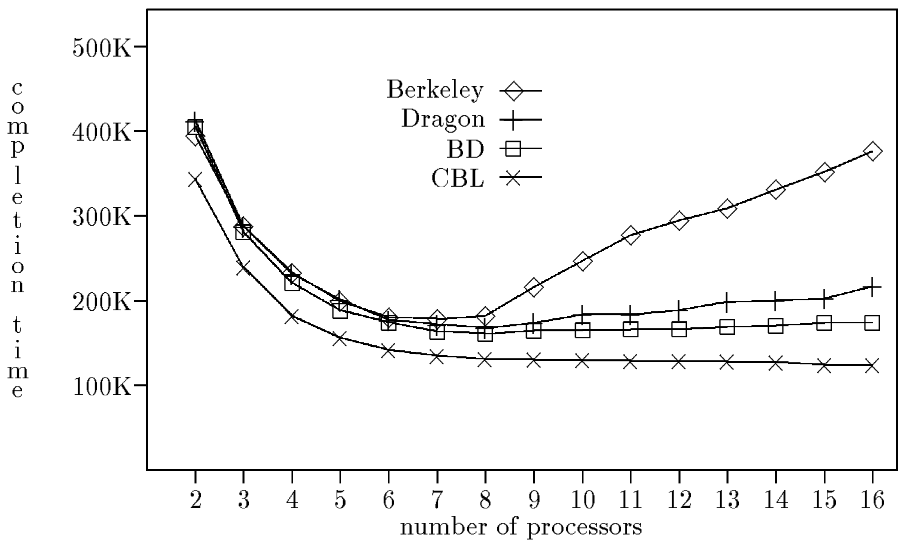

Figure 9: Efficiency of snooping cache protocols for grain size=100

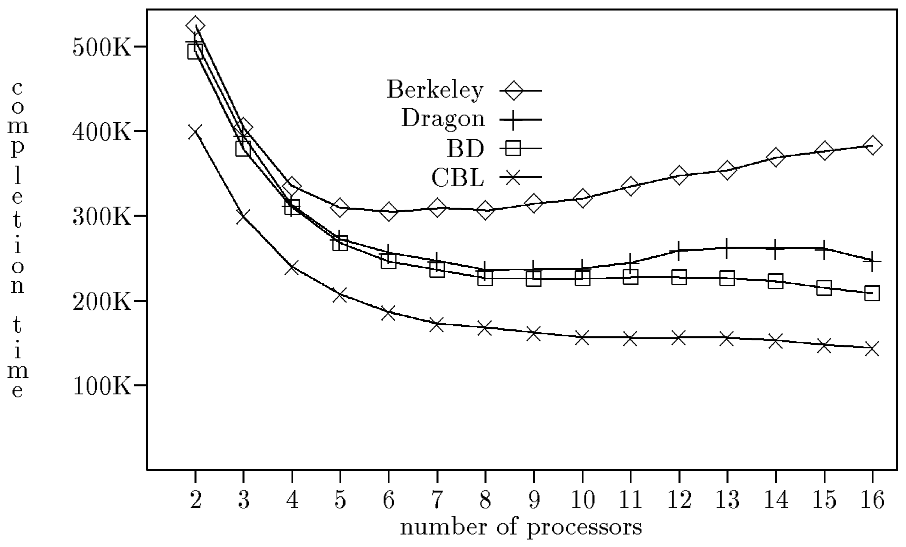

Figure 10: Efficiency of snooping cache protocols for grain size=100 with barrier synchronization

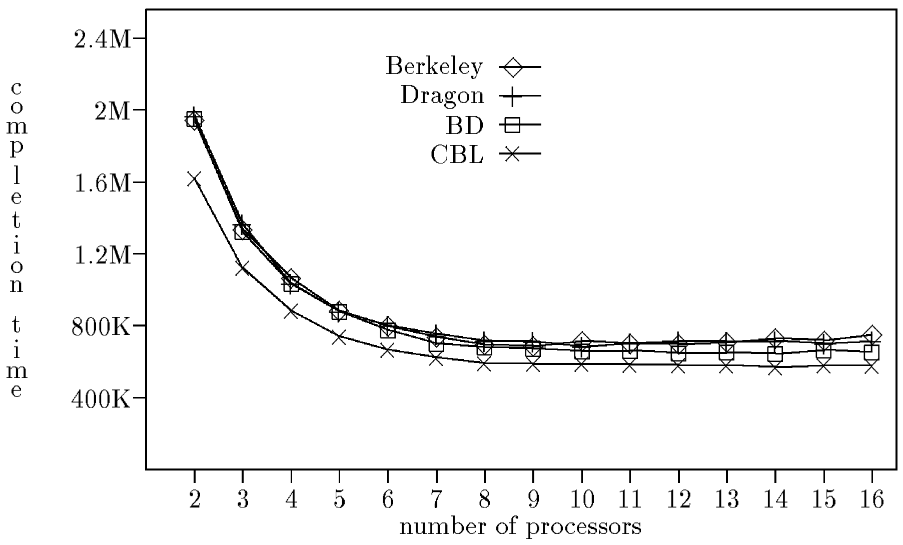

Figure 11: Efficiency of snooping cache protocols for grain size=500

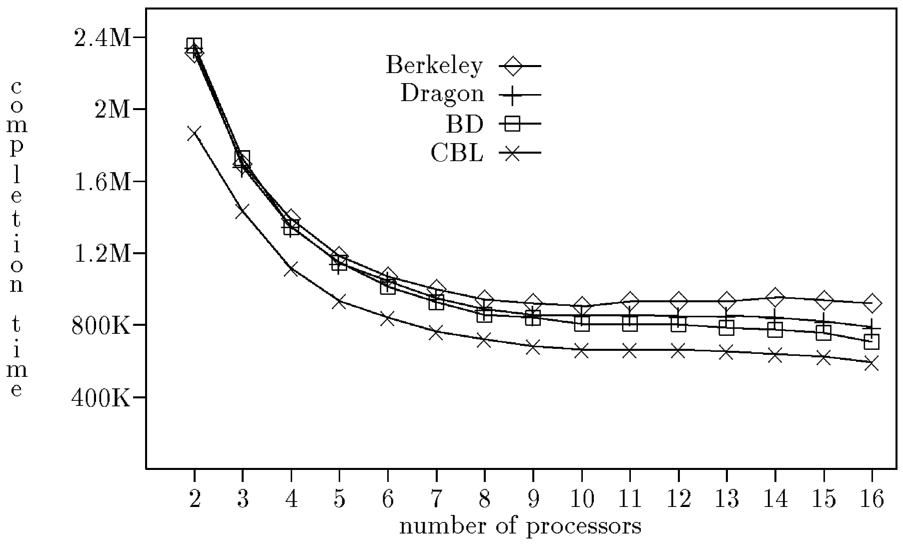

Figure 12: Efficiency of snooping cache protocols for grain size=500 with barrier synchronization

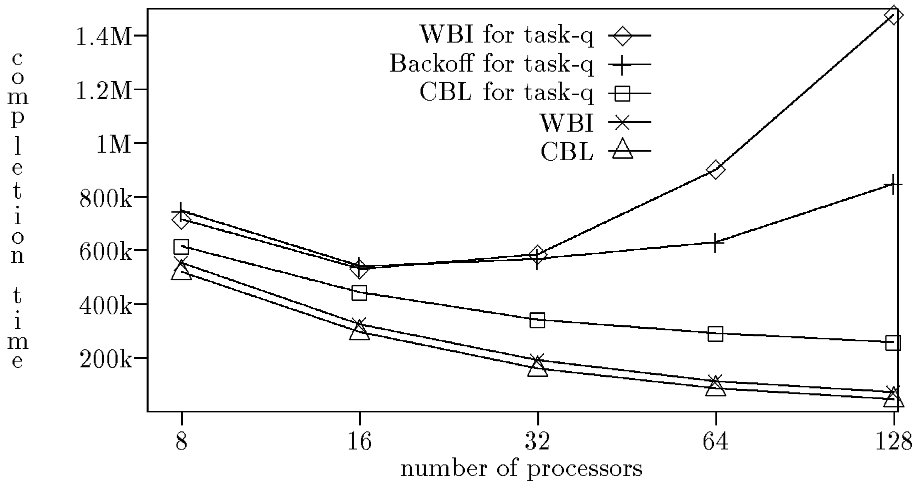

Figure 13: Efficiency of directory cache protocols for grain size=250

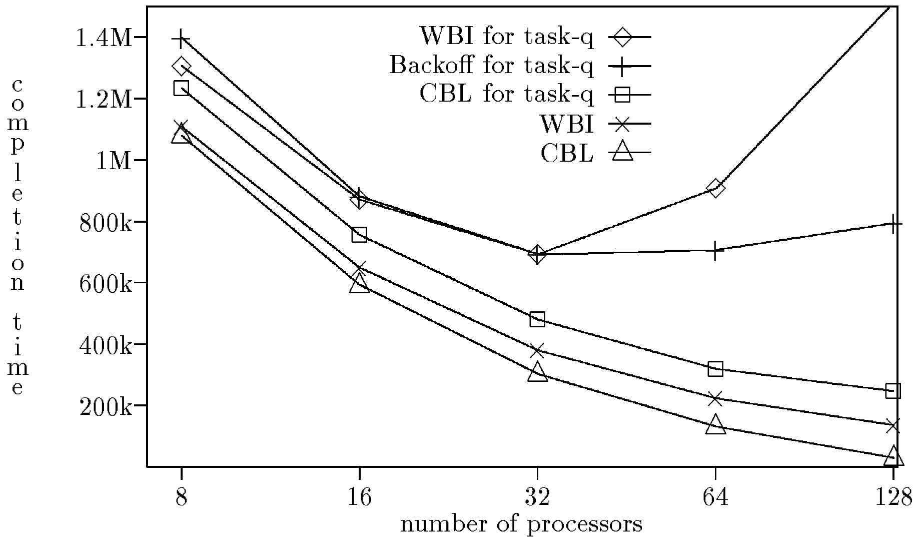

Figure 14: Efficiency of directory cache protocols for grain size=500

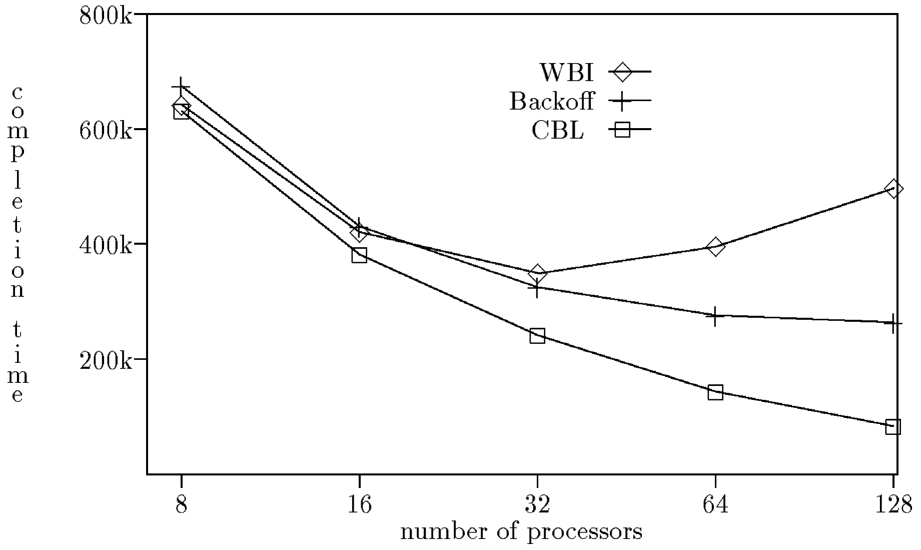

Figure 15: Performance implication of barrier synchronization

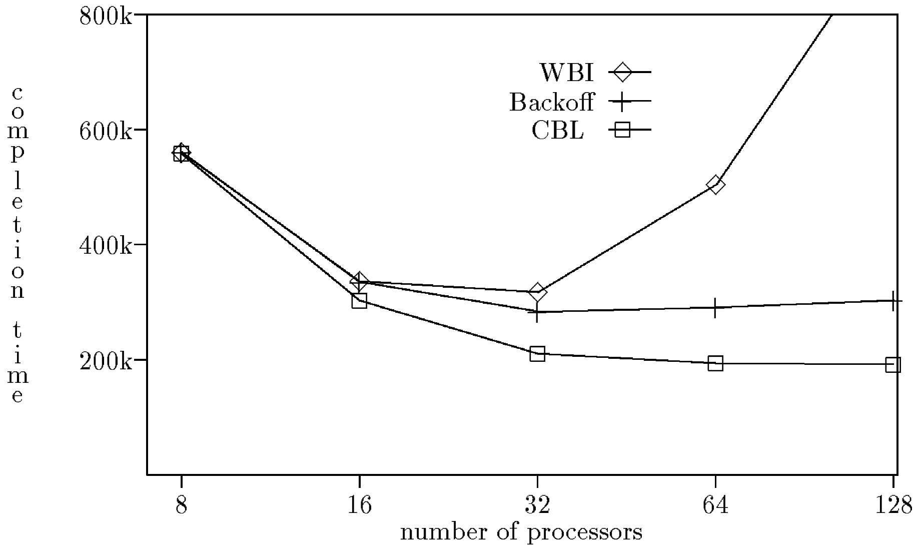

Figure 16: Performance implication of a task queue