The Interaction of Parallel Programming Constructs and Coherence Protocols *

Ricardo Bianchini, Enrique V. Carrera E., and Leonidas Kontothanassis†

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

{ricardo,vinicio}@cos.ufrj.br

†Cambridge Research Laboratory
Digital Equipment Corporation
Cambridge, MA 02139

kthanasi@crl.dec.com

Abstract

Some of the most common parallel programming idioms include locks, barriers, and reduction operations. The interaction of these programming idioms with the multiprocessor's coherence protocol has a significant impact on performance. In addition, the advent of machines that support multiple coherence protocols prompts the question of how to best implement such parallel constructs, i.e. what combination of implementation and coherence protocol yields the best performance. In this paper we study the running time and communication behavior of (1) centralized (ticket) and MCS spin locks, (2) centralized, dissemination, and tree-based barriers, and (3) parallel and sequential reductions, under pure and competitive update coherence protocols; results for write-invalidate protocol are presented mostly for comparison purposes. Our experiments indicate that parallel programming techniques that are well-established for write invalidate protocols, such as MCS locks and parallel reductions, are often inappropriate for update-based protocols. In contrast, techniques such as dissemination and tree barriers achieve superior performance under update-based protocols. Our results also show that the implementation of parallel programming idioms must take the coherence protocol into account, since update-based protocols often lead to different design decisions than write invalidate protocols. Our main conclusion is that protocol-conscious implementation of parallel programming structures can significantly improve application performance; for multiprocessors that can support more than one coherence protocol both the protocol and implementation should be taken into account when exploiting parallel constructs.

*This research was supported by Brazilian CNPq and CAPES.

1 Introduction

Past studies of update-based protocols for cache-coherent multiprocessors (e.g. [1, 4]) have ultimately focused on overall application performance in order to evaluate these protocols. Studies of multiprocessor communication behavior (e.g. [9, 6, 3]) also tend to concentrate on the overall application behavior, without isolating the behavior of the different parallel programming constructs and techniques used by the applications. While early studies focused on overall trends, the advent of programmable protocol processors [11, 16] and their ability to support multiple coherence protocols within the same application motivated us to examine the behavior of individual parallel constructs under different coherence protocols.

In particular we seek to understand the performance of various implementations of process synchronization and reduction operations under pure update (PU) and competitive update (CU) coherence protocols; we also present write invalidate (WI) results for comparison purposes. We use execution-driven simulation of a 32-node multiprocessor to study the performance of (1) centralized (ticket) and MCS spin locks, (2) centralized, dissemination, and tree-based barriers, and (3) parallel and sequential reduction operations. The execution time behavior of each combination of implementation and protocol is explained by the amount and usefulness of the communication generated by the combination.

Our most interesting results show that:

- the ticket lock under the update-based protocols outperforms all other protocol/implementation combinations up to 4-processor machine configurations, while the MCS lock under CU performs best for larger numbers of processors;
- the standard MCS lock is inappropriate for a PU protocol, but a slight modification of this lock can improve its performance;
- dissemination and tree barriers perform significantly better under update-based protocols than under the WI protocol;

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    my_ticket : unsigned integer :=
            fetch_and_increment (&L->next_ticket)
    repeat while L->now_serving != my_ticket

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Figure 1: The centralized ticket lock.

- the dissemination barrier under the update-based protocols is ideal for all numbers of processors;
- under the WI protocol and tightly-synchronized processes, parallel reductions outperform sequential ones, while the opposite is true under update-based protocols;
- overall, update-based sequential reductions exhibit best performance when processes are tightly synchronized;
- given the characteristics of the traffic generated by the protocols, update-based approaches are ideal for scalable barrier synchronization and reductions.

In summary, these results show that parallel programming techniques that are well-established for write invalidate protocols, such as MCS locks and parallel reductions, can become performance bottlenecks under update-based protocols. In contrast, techniques such as dissemination and tree barriers achieve superior performance under update-based protocols. These results also show that the implementation of parallel programming idioms must take the coherence protocol into account, since update-based protocols often lead to different design decisions than write invalidate protocols, as demonstrated by our reduction experiments. Our main conclusion is that protocol-conscious implementation of parallel programming structures can significantly improve application performance; for multiprocessors that can support more than one coherence protocol both the protocol and implementation of the construct should be taken into account if one is to get the best performance the system can deliver.

The remainder of this paper is organized as follows. Section 2 presents the constructs and techniques we study and their implementations. Section 3 describes our methodology and performance metrics. Experimental results are presented in section 4. Section 5 summarizes the related work. Finally, section 6 summarizes our findings and concludes the paper.

2 Constructs and Techniques

Parallel programming for multiprocessors involves dealing with such issues as lock and barrier synchronization and reduction operations. These aspects of parallel applications can be implemented in various ways, the most important of which we describe in this section.

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil
        // queue was non-empty
        I->locked := true
        predecessor->next := I
        // Flush *pred in update-conscious MCS
        repeat while I->locked

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil
        // no known successor
        if compare_and_swap (L, I, nil)
            return
        repeat while I->next = nil
    I->next->locked := false
    // Flush *(I->next) in update-conscious MCS
```

Figure 2: The MCS lock.

2.1 Spin Locks

Spin locks are extremely common parallel programming constructs. We will consider two types of spin locks: the centralized ticket lock and the MCS list-based queuing lock [15]. We chose to study these types of locks as previous studies of several lock implementations under write invalidate protocols have shown that the centralized lock is ideal for low-contention scenarios, while the MCS lock performs best for highly-contended locks.

As seen in figure 1, the centralized lock employs a global counter that provides "tickets" determining when the processor will be allowed to enter the critical section. Another global counter determines which ticket is currently being serviced. A processor is allowed to acquire the lock when its ticket is the same as indicated by the service counter.

The basic idea behind the MCS lock (figure 2) is that processors holding or waiting for access to the lock are chained together in a list. Each processor holds the address of the processor behind it in the list. Each waiting processors spins on its own Boolean flag. The processor releasing a lock is responsible for removing itself from the list and changing the flag of the processor behind it.

Although MCS locks can be efficient under write invalidate protocols, they may generate a large amount of traffic under update-based protocols as all processors competing for a lock may end up caching all other processors' I variables and receiving an update for each modification of these variables. In order to avoid this problem, we propose a modification to the MCS lock in which a processor flushes the I's of its predecessor and successor in the list. The flush operation can be implemented using the user-level block flush instruction common to modern microprocessors such as the

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    // each processor toggles its own sense
    local_sense := not local_sense
    if fetch_and_decrement (&count) = 1
        count := P
        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense = local_sense
```

Figure 3: The sense-reversing centralized barrier.

PowerPC 604. The blocks to be flushed in the modified MCS lock are indicated with comments in figure 2.

2.2 Barriers

Just like spin locks, barriers are common parallel programming constructs. We study three different types of barriers: the sense-reversing centralized barrier, the dissemination barrier, and the tree-based barrier proposed in [15]. We chose to consider these types of barriers as previous studies of synchronization under write invalidate protocols have suggested that centralized barriers are very good for small-scale multiprocessors, while dissemination and tree-based barriers are ideal for large-scale multiprocessors.

In the sense-reversing centralized barrier (figure 3) each processor decrements a variable counting the number of processors that have already reached the barrier. The sense variable prevents a processor from completing two barrier episodes without all processors having completed the first one.

Several algorithms have been proposed to avoid the centralized nature of this barrier. An efficient one, the dissemination barrier (figure 4), replaces a single global synchronization event with $\lceil \log_2 P \rceil$ rounds of synchronizations with a specific pattern; in round k, processor i signals processor $(i + 2^k) \bmod P$, where P is the number of processors. Interference between consecutive barrier episodes is avoided by using alternating sets of variables.

Another efficient distributed barrier algorithm is the tree-based barrier proposed in [15]. This algorithm uses an arrival tree where each group of 4 processors signals barrier arrival events to their common parent, and a wake-up flag to notify the completion of a barrier episode. Pseudo-code for this algorithm is presented in figure 5.

2.3 Reductions

Reduction operations are used in parallel programs in order to produce a "global" result out of "local" arguments. Reductions usually apply a specific operator, such as *min* or *sum*, to per processor arguments to produce a machine-wide

```
type flags = record
    myflags : array [0..1][0..LogP-1] of Boolean
    partflags : array [0..1][0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags

// on proc i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is
//         false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//     allnodes[i].partflags[r][k] points to
//         allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for i : integer := 0 to LogP-1
        localflags^.partflags[parity][i]^ := sense
        repeat until
            localflags^.myflags[parity][i] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Figure 4: The dissemination barrier.

result. Sometimes these arguments are themselves produced by several local applications of the operator.

Reductions can be performed in parallel or sequentially. In a parallel reduction, all processors modify a global variable themselves inside a critical section. An example parallel reduction operation to compute overall the maximum value out of the local maximum value of each processor is presented in figure 6. A piece of code very similar to this one can be found in the Barnes-Hut application from the Splash2 suite [20]. In a sequential reduction, one processor is responsible for computing the global value sequentially as seen in figure 7.

One might wonder why sequential reductions are even reasonable. Two important aspects of parallel and sequential reductions may shed some light into this issue: (1) when processors are tightly synchronized in the parallel reduction, the critical path of the algorithm includes the sum of the critical sections of all processors that queue up for the lock; and (2) due to the manipulation of the lock variable, the sum of P critical sections of the parallel reduction is much longer than the critical path of the sequential reduction (according to our careful analysis of the code generated by gcc with -O2 optimization level).

3 Methodology

We are interested in assessing and categorizing the communication behavior under different multiprocessor coherence protocols and, therefore, we use simulation for our studies.

```
type treenode = record
    parentpointer : ^Boolean
    havechild, childnotready : array [0..3] of Boolean
    dummy : Boolean     // pseudo-data

shared nodes : array [0..P-1] of treenode
processor private vpid : integer
processor private sense : Boolean
shared globalsense: Boolean

// on proc i, sense is initially true
// in nodes[i]: havechild[j] = true if 4*i+j < P; otherwise false
//    parentpointer = &nodes[floor((i-1)/4)].childnotready[(i-1) mod 4],or &dummy if i = 0
//    initially childnotready = havechild, sense = true, globalsense = false

procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready = {false, false, false, false}
        childnotready := havechild    // prepare for next barrier
        parentpointer^ := false       // let parent know I'm ready
        if vpid != 0  // if not root, wait until my parent signals wakeup
            repeat until globalsense = sense
        else
            globalsense := sense
        sense := not sense
```

Figure 5: The tree-based barrier.

```
shared max : integer
shared Lock : lock
shared Barrier : barrier
processor private local_max : integer

// Code that changes local_max
LOCK(Lock)
    if max < local_max
        max := local_max
UNLOCK(Lock)
BARRIER(Barrier)
    // Code that uses max
BARRIER(Barrier)
```

Figure 6: Parallel reduction operation.

```
shared max : integer
shared local_max : array [0..P-1] of integer
shared Barrier : barrier
processor private pid, i : integer

// Code that changes local_max[pid]
BARRIER(Barrier)
    if pid = 0
        for i := 0 until i = P-1
            if max < local_max[i]
                max := local_max[i]
BARRIER(Barrier)
// Code that uses max
```

Figure 7: Sequential reduction operation.

3.1 Multiprocessor Simulation

We use a detailed execution-driven simulator (based on the MINT front-end [19]) of a 32-node, DASH-like [13], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a 4-entry write buffer, a 64-KB direct-mapped data cache with 64-byte cache blocks, local memory, a full-map directory, and a network interface. Shared data are interleaved across the memories at the block level. All instructions and read hits are assumed to take 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into the write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. A memory module can provide the first word 20 processor cycles after the request is issued. Other words are delivered at a rate of 1 word per processor cycle. Memory contention is fully modeled. The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 2-cycle delay to the header of each message. The network datapath is 16-bit wide. Network contention is only modeled at the source and destination of messages.

Our WI protocol keeps caches coherent using the DASH protocol with release consistency [12]. In our update-based implementations, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. The writing processor only stalls waiting for acks at a lock release point.

Our PU implementation includes two optimizations. First, when the home node receives an update for a block that is only cached by the updating processor, the ack of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data written by the parent but not subsequently needed by the child.

In our CU implementation, each node makes a local decision to invalidate or update a cache block when it sees an update transaction. We associate a counter with each cache block and invalidate the block when the counter reaches a threshold. At that point, the node sends a message to the block's home node asking it not to send any more updates to the node. References to a cache block reset the counter to zero. We use counters with a threshold of 4 updates.

Our simulator implements three atomic instructions: fetch_and_add, fetch_and_store, and compare_and_swap. The coherence protocol used for the atomically-accessed data is always the same as the protocol used for all the rest of the shared data. The computational power of the atomic instructions is placed in the cache controllers when the coherence protocol is WI and in the memory when using an update-based protocol. So, for instance, the fetch_and_add instruction under the WI protocol obtains an exclusive copy of the referenced block and performs the addition locally. Fetch_and_add under an update-based protocol sends an addition message to the home memory, which actually performs the addition and sends update messages with the new value to all processors sharing the block. All the atomic instructions we implement force write buffer flushes, but in our experiments write buffers are almost always empty when these instructions are executed.

3.2 Performance Metrics

The focus of this paper is on running times and our categorization of the communication traffic in invalidate and update-based protocols. We consider the communication generated by cache misses (and block upgrade operations) under both types of protocol and the update messages under update-based protocols. The miss rate is computed solely with respect to shared references.

In order to categorize cache misses we use the algorithm described in [5] as extended in [2]. The algorithm classifies cache misses as cold start, true sharing, false sharing, eviction, and drop misses. We assume cold start and true sharing misses to be *useful* and the other types of misses to be *useless*. More specifically, the different classes of cache misses in our algorithm are:

- **Cold start misses.** A cold start miss happens on the first reference to a cache block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached, but that has been invalidated due to a write to the same word by some other processor.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not referenced by the missing processor.
- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.
- **Drop misses.** A drop miss happens when a processor references a word belonging in a block it had previously cached, but that has been self-invalidated under the competitive update protocol.

Our miss categorization algorithm includes a sixth category, **exclusive** request transactions. An exclusive request operation (caused by a write to a read-shared block already cached by the writing processor under the WI protocol) is not strictly a cache miss, but does cause communication traffic.

We classify update messages according to the algorithm described in [2]. The algorithm classifies update messages at the end of an update's lifetime, which happens when it is overwritten by another update to the same word, when the cache block containing the updated word is replaced, or when the program ends. We also classify updates as useful and useless. Intuitively, useful updates are those updates required for correct execution of the program, while useless updates could be eliminated entirely without affecting the correctness of the execution. More specifically, the different classes of updates in our algorithm are:

- **True sharing updates.** The receiving processor references the word modified by the update message before another update message to the same word is received. This type of update transaction is termed useful, since it is necessary for the correctness of the program.
- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.
- **Proliferation updates.** The receiving processor does not reference the word modified by the update message before it is overwritten, and it does not reference any other word in that cache block either.
- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.
- **Termination updates.** A termination update is a proliferation update happening at the end of the program.
- **Drop updates.** A drop update is the update that causes a block to be invalidated in the cache.

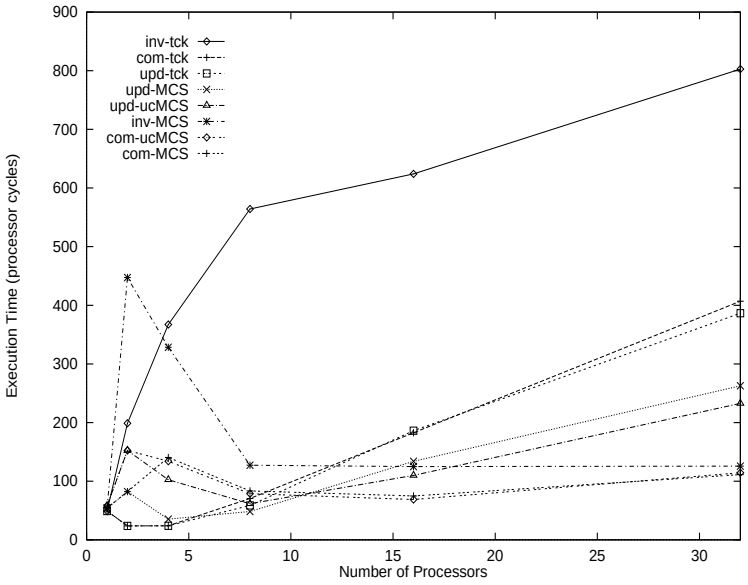

Figure 8: Performance of spin locks in synthetic program.

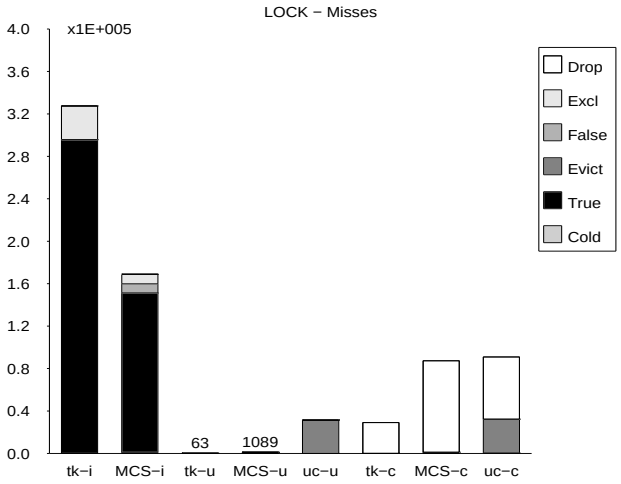

Figure 9: Miss traffic of spin locks in synthetic program.

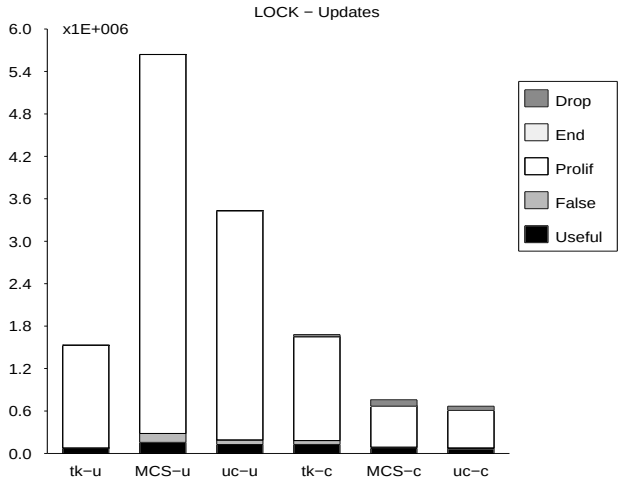

Figure 10: Update traffic of spin locks in synthetic program.

This categorization is fairly straightforward, except for our false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, our algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

4 Experimental Results

In this section we evaluate the performance of the different implementations of locks, barriers, and reductions under the three coherence protocols we consider. In all implementations, shared data are mapped to the processors that use them most frequently.

4.1 Spin Locks

In order to assess the performance of each combination of spin lock implementation and coherence protocol under varying levels of lock contention, we developed a synthetic program where each processor acquires a lock, spends 50 processor cycles holding it, and then releases the lock. All of this in a tight loop executed $32000/P$ times, where P is the number of processors.

Figure 8 presents the average latency of an acquire-release pair (in processor cycles) for each machine configuration. This average latency is computed by taking the execution

time of the synthetic program, dividing it by 32000, and later subtracting 50 from the result. Figure 9 presents the number and distribution of cache misses involved in each of the lock/protocol combinations on 32 processors, while figure 10 presents the number and distribution of update messages in the lock implementations using the update-based protocols again on 32 processors [1]. The bar labels in these figures represent the specific algorithm/protocol combinations: "tk", "MCS", and "uc" stand for ticket, MCS, and update-conscious MCS locks respectively, while "i", "u", and "c" stand for WI, PU, and CU respectively. Note that we placed the absolute numbers of cache misses on top of the bars that are not tall enough to be noticeable.

For centralized locks, figure 8 shows that PU performs slightly better than CU for 32 processors, while both protocols perform significantly better than WI for all machine configurations. As seen in figures 9 and 10, the reason for this result is that the update-based protocols exchange the expensive cache misses necessary to constantly re-load the `ticket` and `now` counters in WI, for corresponding update messages that only lead to performance degradation if they end up causing resource contention.

For the MCS lock, the CU protocol outperforms all other combinations for 16 and 32 processors; trends indicate that for larger numbers of processors the WI protocol should become best. The MCS lock exhibits terrible performance under PU; the implementation using this protocol is worse than the ones with WI and CU by a factor of 2 for 32 processors. The main problem with the MCS lock under PU protocols is that it increases the amount of sharing (by sharing the global pointer to the end of the list and pointers to list predecessors and successors) with respect to a centralized lock, without reducing the frequency of write operations on the shared data. This increased sharing causes intense messaging activity (proliferation updates mostly) that degrades performance, as seen in figure 10.

Our modification to the MCS lock significantly alleviates the sharing problem of the standard MCS lock under PU protocols, as seen by the 39% reduction in update messages the modification produces. However, much of the effect of this reduction is counter-balanced by an increase in cache miss activity from 1089 to 31588 misses. The outcome of this tradeoff is 18% and 11% performance improvements for 16 and 32 processors, respectively. Note that the extent to which the reductions in traffic provided by our update-conscious MCS lock lead to performance improvements depends on the architectural characteristics of the multiprocessor: performance improvements are inversely proportional to communication bandwidth and latency.

Overall, we find that within the range of machine sizes we consider ticket locks with update-based protocols achieve best performance up to 4 processors, while MCS locks under CU are ideal for larger numbers of processors. Our update-conscious implementation of MCS locks improve the performance of this lock algorithm, but not enough to justify its use when a choice is available. Finally, we also find that, independently of the lock implementation, the vast majority of updates under an update-based protocol is useless.

These experiments (and all the others in the paper) were purposedly made similar to the ones used by Mellor-Crummey and Scott in [15] for comparison purposes. We also performed experiments with a slightly modified synthetic program where, instead of releasing the lock and immediately trying to grab it again, processors waste a pseudo-random (but bounded) amount of time after the release. This modified synthetic program provides for reduced lock contention. The results of these modified experiments are qualitatively the same as the ones presented in this section. In a more controlled experiment, we made the ratio of work outside and inside the critical section be equal to the number of processors (+- 10%). Again, the results of this modified experiment are qualitatively similar to the ones discussed in this section.

4.2 Barriers

In order to assess the performance of each combination of barrier implementation and coherence protocol, we developed a synthetic program where processors go through a barrier in a tight loop executed 5000 times.

Figures 11 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 5000. This time is thus the average latency of a barrier episode for each machine configuration. Figure 12 presents the cache miss behavior of each of the barrier/protocol combinations on 32 processors, while figure 13 presents the update behavior of the barrier implementations using the update-based protocols again on 32 processors. The bar labels in these figures represent the specific algorithm/protocol combinations: "cb", "db", and "tb" stand for centralized, dissemination, and tree-based barriers respectively.

Figure 11 shows that for centralized barriers the WI protocol outperforms its update-based counterparts, but only for large machine configurations, as one would expect. Figures 12 and 13 show that even though the number of misses of the centralized barriers under the update-based protocols is negligible, the amount of update traffic these protocols generate is substantial and mostly useless. The vast majority of this useless traffic corresponds to changes in the centralized counter of barrier arrivals.

For the dissemination barrier CU and PU perform equally well, significantly outperforming WI for all numbers of processors. The reason for this result is WI causes a relatively large number of cache misses on accesses to the `myflags` array, while the update behavior of the dissemination barrier under CU and PU is very good (as can be seen by their lack of useless update messages).

For the tree-based barrier PU and CU again (and for the same reasons) perform equally well and much better than WI for all numbers of processors.

These results indicate that the dissemination barrier under either PU or CU is the combination of choice for all numbers of processors. In addition, our barrier results demonstrate that update-based protocols perform extremely well for scalable barriers, as shown by the absence of useless update messages in these barriers.

[1] The bars in the figure do not include the replacement updates category since this type of updates was never observed in our experiments.

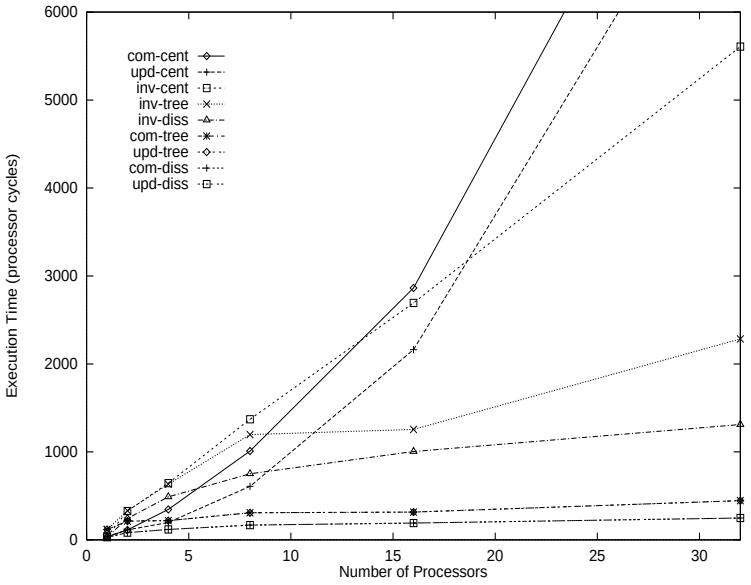

Figure 11: Performance of barriers in synthetic program.

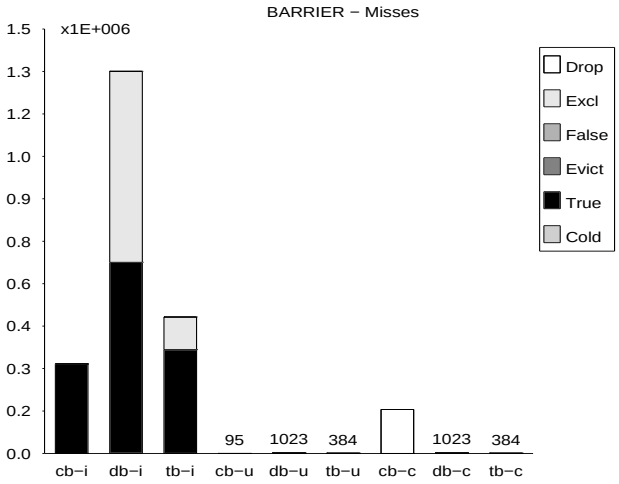

Figure 12: Miss traffic of barriers in synthetic program.

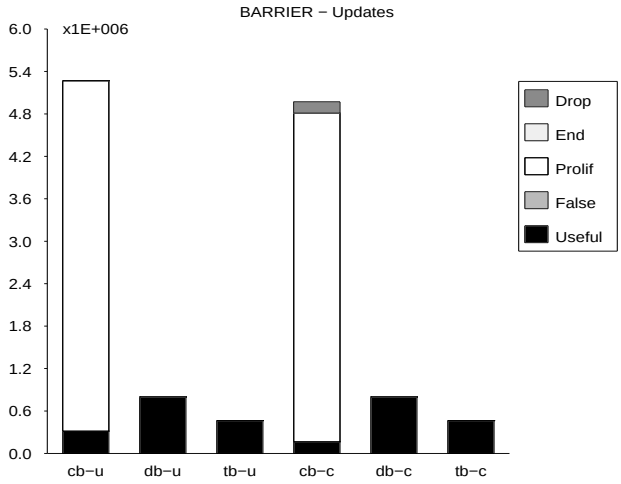

Figure 13: Update traffic of barriers in synthetic program.

4.3 Reductions

In order to assess the performance of each combination of reduction implementation and coherence protocol, we developed a synthetic program where each processor executes 5000 reductions in a tight loop. To avoid disturbing the results by the synchronization overhead involved in the reductions, we simulated locks and barriers that synchronize without generating any communication traffic.

Figure 14 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 5000. This time is thus the average latency of a whole reduction operation for each machine configuration. Figure 15 presents the cache miss behavior of each of the reduction/protocol combinations on 32 processors, while figure 16 presents the update behavior of the reduction implementations using the update-based protocols again on 32 processors. The bar labels in these figures represent the specific algorithm/protocol combinations: "sr" and "pr" stand for sequential and parallel reductions respectively.

Figure 14 shows that under the WI protocol, parallel reduction outperforms its sequential counterpart; the former strategy leads to significantly fewer cache misses (on accesses to `max`) in its critical path than the sequential reduction (on accesses to `max` and `local_max[pid]`). However, for update-based protocols sequential reduction is the ideal strategy, since there are no cache misses in the critical path of the two algorithms and, for our tightly-synchronized synthetic program, the critical path of the parallel reduction is longer

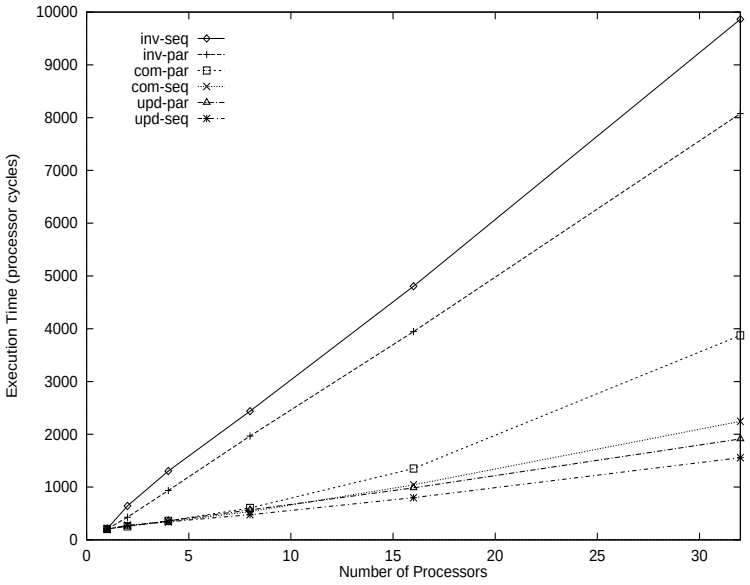

Figure 14: Performance of reductions in synthetic program.

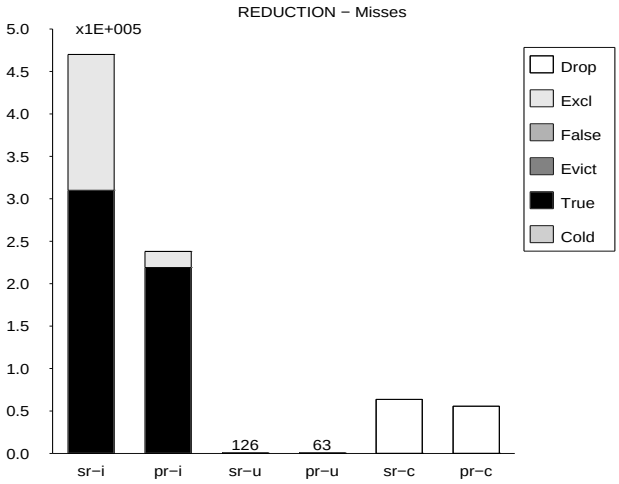

Figure 15: Miss traffic of reductions in synthetic program.

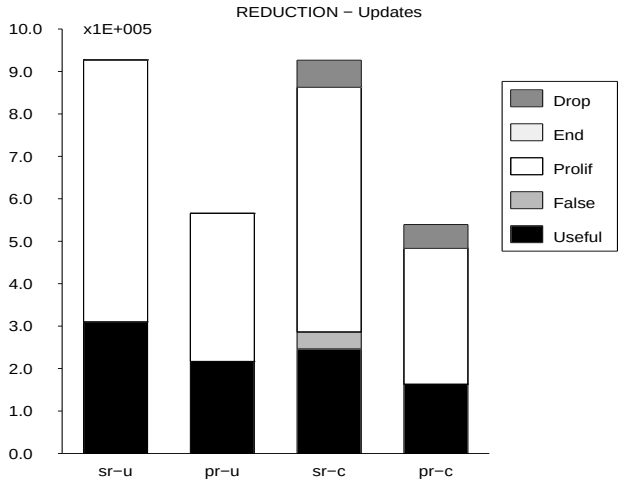

Figure 16: Update traffic of reductions in synthetic program.

(as explained in section 2.3). Figure 16 shows that both parallel and sequential reductions exhibit a large percentage of useful updates, indicating that update-based protocols are appropriate for this type of operation, just as they are for scalable barriers.

A comparison between update-based and invalidate-based reductions is also interesting. Overall, update-based sequential reductions always exhibit better performance than parallel reductions under WI. The reason for this result is that the critical path of a parallel reduction under WI is significantly longer than the critical path of a sequential reduction under the update-based protocols due to the cache misses involved in the former algorithm/protocol combination.

Although interesting, these experiments only model the case where processes are tightly synchronized and most processors end up contending for lock access. We also performed experiments with a slightly modified synthetic program to generate some load imbalance and consequently reduce lock contention. The results of these experiments show that parallel reductions become more efficient than their sequential counterparts, but still parallel reductions with PU and CU perform better than parallel reductions with WI.

5 Related Work

As far as we are aware, this study is the first to isolate the performance of important parallel programming constructs and techniques under PU and CU protocols. This analysis led to a number of interesting observations that challenge

previously established results. In addition, our study is the first to relate these constructs and techniques to their communication behavior under invalidate, update, or competitive protocols. Some related pieces of work are listed next.

The impact of coherence protocols on application performance is an active area of research. Early work by Eggers [8] studied the relative performance of invalidate and update protocols on small bus-based cache-coherent multiprocessors. More recent work [4, 18] has looked at the impact of update-based protocols on overall program performance on larger scale machines.

Other researchers have taken an applications-centric view and have focused on the communication patterns induced by applications, mostly in the context of write-invalidate protocols. Gupta and Weber [9] looked at the cache invalidation patterns in shared-memory multiprocessors, and determined that for most applications the degree of sharing is small. Holt et al. [10] also looked at the communication behavior of applications in the context of large-scale shared-memory multiprocessors and identified architectural and algorithmic bottlenecks. Dubois et al. [5], Torrellas at al. [17], and Eggers et al. [7] have looked at the usefulness of communication traffic generated by real applications in the context of write-invalidate protocols. Bianchini et al. [3] and Dubois et al. [6] have looked at the usefulness of communication traffic under both invalidate and update-based protocols.

Parallel programming constructs and in particular synchronization algorithms have also received a lot of attention, however always in the context of either non-coherent machines or machines based on write invalidate protocols. Mellor-Crummey and Scott [15], for instance, have presented the set of synchronization algorithms that we have used in our evaluation of synchronization primitives.

Finally Michael and Scott [14] have studied the performance impact of different implementations of atomic instructions in scalable multiprocessors. However, their study focuses on the atomic primitives rather than on the algorithms that use them.

6 Conclusions

In this paper we have studied the running time and communication behavior of several lock, barrier, and reduction implementations on top of invalidate and update-based protocols. Our analysis indicates that locks can profit from update-based protocols, especially for small to medium contention levels. In addition, our results show that scalable barriers can benefit greatly from these protocols, independently of the number of processors, as their associated miss rates are low and their update traffic is light and mostly useful. Our reduction results demonstrate that sequential reductions also benefit from protocols based on updates, but their performance is only competitive under heavy contention. Finally, this study shows that the implementation of parallel programming idioms must take the coherence protocol into account, since invalidate and update-based protocols often lead to different design decisions. Programmers of update-based multiprocessors and machines with protocol processors should then carefully implement their constructs if applications are to avoid unnecessary overheads.

References

[1] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[2] R. Bianchini and L. I. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.

[3] R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors. In *Proceedings of the International Parallel Processing Symposium '96*, April 1996.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[5] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, May 1993.

[6] M. Dubois, J. Skeppstedt, and P. Stenstrom. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, Sept 1995.

[7] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[8] 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 International Symposium on Computer Architecture*, pages 373–383, May 1988.

[9] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transaction on Computers*, 41(7):794–810, July 1992.

[10] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 134–145, May 1996.

[11] J. Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

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

[13] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[14] M. M. Michael and M. L. Scott. Implementation of General-Purpose Atomic Primitives for Distributed Shared-Memory Multiprocessors. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 222–231, January 1995.

[15] 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.

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

[17] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[18] J. E. Veenstra and R. J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[19] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, May 1995.