Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors

David B. Glasco
Computer System Laboratory
Stanford University
Stanford, CA 94305

Bruce A. Delagi
Sun Microsystems, Inc.
and
Stanford University

Michael J. Flynn
Computer System Laboratory
Stanford University
Stanford, CA 94305

Abstract

In this paper, two hardware-controlled update-based cache coherence protocols are presented. The paper discusses the two major disadvantages of the update protocols: inefficiency of updates and the mismatch between the granularity of synchronization and the data transfer. The paper presents two enhancements to the update-based protocols, a write combining scheme and a finer grain synchronization, to overcome these disadvantages.

The results demonstrate the effectiveness of these enhancements that, when used together, allow the update-based protocols to significantly improve the execution time of a set of scientific applications when compared to three invalidate-based protocols.

1 Introduction

In shared-memory multiprocessors, caches have been shown to be an effective technique to reduce memory access latency. Unfortunately, one problem that arises in such systems is the cache coherence problem. A memory line may be present in any of processor's caches; thus, to execute programs correctly, the copies of this memory line must remain consistent. When a processor modifies the memory line, other caches that have a copy of the line must be notified so that their copy may be made consistent.

The two methods for maintaining consistency on a write are invalidating or updating the copies of the memory line. Invalidating purges the copies of the line from the other caches which results in a single copy of the line, and updating forwards the write value to the other caches, after which all caches are consistent.

Cache coherence may be maintained through software, hardware or a combination of the two. The majority of scalable hardware-based systems with a general interconnect use invalidations to maintain consistency [10, 21, 9, 13]. Several of the software-based schemes use a combination of invalidations and updates [2, 14, 11, 3, 24, 23].

This paper presents two hardware-controlled update-based cache coherence protocols: one based on a centralized directory and the other based on a singly linked distributed directory. The paper considers two major disadvantages of the update-based protocols and develops approaches to overcoming them. The first disadvantage is the increased network traffic resulting from the updates. The paper demonstrates how simple hardware combining of writes can significantly reduce this traffic and the resulting congestion. The second problem is the mismatch between the granularity of synchronization and data sharing. With invalidate-based protocols, coarse-grained synchronization typically matches the inherent line-based data sharing supported by the protocol, but in the update-based protocols this tactic does not take full advantage of the fine-grain data updates. A finer grain (word) synchronization would allow shared data to be consumed as soon as it is available.

The paper is organized as follows. Section 2 describes the cache coherence design space, and in particular section 2.3 describes the two new update-based cache coherence protocols. Section 3 discusses the interaction between these update-based protocols and memory consistency models. Next, section 4 identifies the important characteristics of the shared-memory applications and describes the interactions between consumers and producers and the use of data prefetching. Section 5 introduces two enhancements to the update-based protocols: finer grain synchronization and write combining. Section 6 presents the simulated architecture, and section 7 describes the applications under study and presents the simulated results. Finally, section 8 concludes the paper.

2 Cache coherence

Cache coherence protocols can be separated by what actions are taken on a processor write. All protocols must guarantee that after the actions triggered by the write are completed, all caches in the system are consistent with each other. Copies of the memory line held by other caches may be updated or invalidated on a write, and the memory's copy may also be updated. Figure 1 summarizes the four possibilities which result in two classes of protocols: update-based (UP) or invalidate-based (INV). This paper studies the fundamental performance differences between these classes of protocols.

		Remote Caches	
Memory		Invalidated	Updated
	Not Updated	Invalidate (SCI/SDD)	Update (DD-UP)
	Updated	Invalidate (CD-INV)	Update (CD-UP)

Figure 1: Protocols Classes

2.1 Directory structure

To invalidate or update all copies of a memory line, the protocols must maintain a list of caches that have a copy of each memory line. This information, which is stored in a directory entry, can either be stored in a single, central location (centralized directory protocol) or distributed among the caches holding a copy of the line (distributed directory protocol). In both cases, the directory entries are distributed throughout the system with their respective memory lines. In the centralized directory (CD) protocols, the directory entry contains a pointer to each cache that contains a copy of the line. In the CD protocols studied in this paper, a fully mapped directory is used [19] in which there is a single bit pointer for each cache in the system. In the distributed directory (DD) protocols, a linked list structure is used to maintain a list of caches that have a copy of a given memory line. The directory entry contains a pointer to the head of this list. In the Singly linked Distributed Directory (SDD) protocol [21], a singly linked list is used to maintain the list. The Scalable Coherence Interface (SCI) [9] distributed directory protocol uses a doubly linked list. The relative scalability and performance of these directory schemes are a current research topic [4, 17, 25]; therefore, update-based protocols based on both directory structures are presented.

2.2 Invalidate-based protocols

This paper compares update-based protocols with three invalidate protocols: a centralized directory protocol (CD-INV) which is similar to DASH [10], a singly linked distributed directory protocol (SDD) and a doubly linked distributed directory protocol (SCI).

The invalidate protocols differ in how invalidation is performed. In the CD-INV protocol, the invalidates can be sent out in parallel, since the directory entry contains all the necessary information. For the distributed directory protocols, the invalidates flow down the linked list of caches; the length of the list determines the latency of the invalidations.

The source of data for a miss reply also differs among the invalidate protocols. The CD-INV protocol supplies the data from main memory if the memory line is clean. If the line is dirty in another cache, the protocol uses cache-to-cache transfers to forward the data to the requesting cache, and if the request was a read miss, the protocol writes the data back to memory. For the SDD protocol, cache-to-cache transfer is also used, but the line's data is not written back to memory. The cache at the head of the linked list always supplies the data on a miss. For the SCI protocol, the memory can supply the data until it is modified by a cache after which cache-to-cache transfers are used. For a detailed comparison of the invalidate protocols see the work by Thapar [21].

2.3 Update-based protocols

In this section, a brief description of the centralized directory and distributed directory update protocols is presented. Both protocols currently rely on a network that preserves order among messages.

In a centralized directory update-based protocol (CD-UP), the directory information is used to update the necessary caches on a write. Figure 2 shows the flow of data for a write update. In the figure, the processor associated with cache 0 issues a write to the cache. The cache sends a "write" signal along with the data to the directory where the directory information is used to determine which caches have a copy of the line. An "update" signal along with the new data is sent to these caches and memory is updated. (The CD-UP protocol relies on multicast to send updates

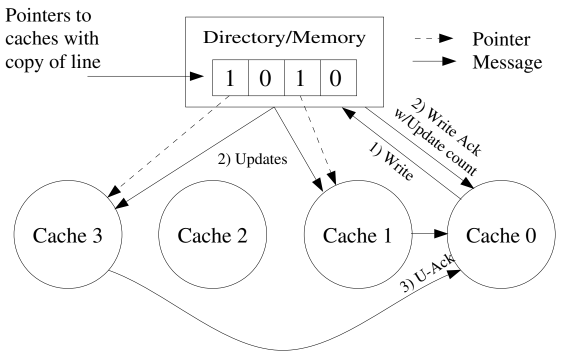


Figure 2: Centralized Directory Update Protocol

to multiple caches). The directory also sends a "write acknowledgment" signal back to the writing cache. This signal indicates the number of update acknowledgments to expect. After updating its copy, each updated cache sends an "update acknowledgment" signal to the writing cache. Once all the expected "update acknowledgment" signals and the "write acknowledgment" signal have been received, the write is considered to be performed as all caches with a copy of the line have been updated.

For misses, the memory supplies the data if it has a valid copy; that is, if no cache has an exclusive copy of the line. Otherwise, cache-to-cache transfer is used to forward the line to the requesting cache, and the line is also sent back to the memory. If no other cache in the system has a copy of a requested memory line, the requesting cache receives an exclusive copy for both read and write misses.

The distributed directory update protocol (DD-UP) is based on the directory structure and singly linked lists of the SDD invalidate protocol. Both read and write misses are serviced by the cache at the head of the list, and the requesting cache becomes the new head of the list.

For a write hit, the update must begin at the head of the list of caches. If the writing cache is not at the head of the list, it sends a "write" signal along with the data to the directory as shown in figure 3. The directory responds by sending an "update" signal with the data to the cache at the head of the list. This cache updates its copy and sends the update to the next cache in the linked list of caches. Each cache in the list receives the "update" signal, updates its copy and forwards the signal. If the cache is at the end of the list, it sends an "update acknowledge" signal to the writing cache indicating that the update has been performed. If the writing cache is at the head of the list, it simply sends an "update" signal with the data to the next cache in the list as described above; the directory is not involved.

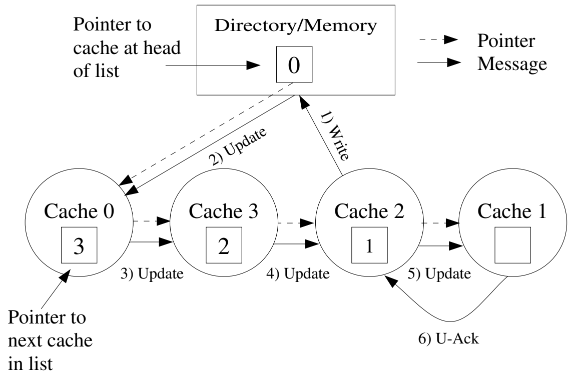


Figure 3: Singly Linked Distribute Directory Update Protocol

3 Memory consistency models

The update protocols described in section 2.3 can implement a relaxed consistency model [8]. To implement sequential consistency, a two-phase update protocol is required [23]. In this type of protocol, an update requires two phases. In the first phase, all copies of the word are marked as pending. Any processor attempting to access the word is blocked. During the second phase, the data is updated and the pending bit is cleared. The two-phase protocol prevents a processor from reading the old value while another processor is able to read the new value.

The sequential consistency model requires the writing processor to stall until the first phase of the update completes. The latency of this first phase will determine the resulting impact of the two-phase update protocol on the execution time of the application. It is interesting to note that the first phase does not require the actual data; and therefore, it may be issued earlier. Future work will study the performance of the update protocols using stronger memory consistency models and the two-phase update.

The scientific applications under study use a relaxed consistency model [8].

4 Data sharing in parallel applications

In this section, the types of shared objects in shared-memory applications are examined. First, previous classification schemes are examined, and then these classifications are refined to be more readily applicable to the current study. Finally, a brief discussion of typical producer and consumer interactions and the impact of data prefetching is presented.

4.1 Classification of data objects

The types of shared-data objects have been classified into the following types: code and read-only, migratory, mostly read, frequently read and written, and synchronization objects [22]. Since the main difference between update and invalidate protocols is what happens on a write, these types are subdivided here by the frequency of writes. Code, read-only and mostly read objects will have little impact on the relative performance of the protocols. But migratory, frequently read and written, and synchronization objects will significantly impact the overall performance of the application. The scientific applications currently under study consist of a large number of frequently read and written objects and high-contention synchronization semaphores.

Frequently read and written objects and synchronization objects, especially semaphores, can be further classified by the number of consumers reading each object and by the line utilization, which is the fraction of each memory line that is modified by the producer. The number of consumers determines the number of invalidates or updates required and general contention for the object. The line utilization gives a general measure of the data transfer efficiency of the protocols. The higher the line utilization, the more efficient the line-sized transfers of the invalidate protocols are. As the line utilization decreases, single-word updates become more efficient. The line utilization measure is similar to the comparison metrics used in other evaluations of update-based and invalidate-based protocols [24, 7]. As will be shown in section 7, these classifications are a good predictor of the resulting performance of the cache coherence protocols.

Currently, none of these applications have migratory data. Applications with migratory data may significantly increase the number of updates required in an update-based protocol. These updates would create significant network congestion, but simple counters may be used to replace lines that receive a sequence of update without any intervening reads. The replacement of lines should reduce the number of unnecessary updates. Future work will study the impact of migratory data on update-based protocols in greater detail.

4.2 Producer and consumer interactions

In a relaxed consistency model, a binary semaphore can be used to synchronize the production and consumption of shared data. Under this type of coarse grain (block) synchronization, the synchronization and transfer of data can be divided into the basic operations shown in figure 4. The producer computes the data, writes the results to a shared buffer, and then uses a "fence" instruction to stall the processor until all the writes have been performed. Once the writes have been performed, the semaphore is set. Consuming nodes wait for the semaphore to be set before attempting to access the shared data.

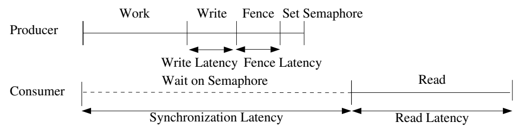


Figure 4: Block Synchronization

The performance of each protocol is dependent on how efficiently it can execute each of the components shown in the figure: the write, fence, synchronization and read latency. The write latency is simply the time to "issue" or send the writes to the write buffer. The fence latency is the latency until all writes have been performed. In the invalidate protocols, a write is considered to have been performed when the cache receives exclusive ownership of the line. For the update protocols, a write is considered to be performed when all necessary updates have been acknowledged. The synchronization latency is the time a consumer waits for the desired semaphore to be set; and finally, the read latency is the time to read the data once the semaphore has been set. The discussion of the results in section 7 will rely heavily on these latency components.

4.3 Effects of data prefetch

Data prefetch can be used to hide a portion of the data miss latency in a typical producer and consumer interaction [12]. In the invalidate-based protocols, a write prefetch can hide a portion of the producer's fence latency by issuing write requests early and overlapping multiple requests as shown in figure 5. The write prefetch allows the write miss latency to be overlapped with useful work.

A read prefetch can also be used to reduce the consumer's read miss latency, but the prefetches should be issued after reaching the synchronization point. If the prefetches are issued earlier, two scenarios are possible. In the first case, the producer has not yet written the data. In this case, the prefetched lines will be later invalidated, and the work done to prefetch

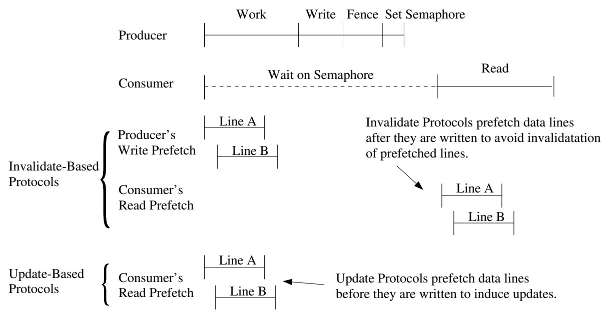

Work
Write
Fence
Set Semaphore
Producer
Consumer
Wait on Semaphore
Read
Invalidate-Based Protocols
Producer's Write Prefetch
Consumer's Read Prefetch
Line A
Line B
Invalidate Protocols prefetch data lines after they are written to avoid invalidatation of prefetched lines.
Update-Based Protocols
Update Protocols prefetch data lines before they are written to induce updates.

Figure 5: Block Synchronization with Prefetch

and invalidate the lines will be wasted. In the second case, the producer has completed writing the data before the consumer's prefetch is received. In this case, the prefetch will obtain the proper data, but the time saved compared to prefetching after the synchronization point will be minimal since the producer would have also set the semaphore and the consumer's read of the semaphore would find it set. The latency saved by prefetching early is small compared to the high cost of invalidated prefetches. In all simulated cases, prefetching before the synchronization point never resulted in a faster execution than prefetching after the synchronization point for the invalidate protocols.

In the update-based protocols, read prefetches can be used not only to hide read miss latency behind useful work, but they also can be used to induce prefetches. If the prefetch is issued before the data is written by the producer, the consumer's cache will be updated when the data is written. The prefetch allows the consumers to express an early interest in the data.

5 Update protocol enhancements

The update-based protocols have two basic disadvantages when using block synchronization and prefetching. The first is the mismatch between the granularity of synchronization and data transfer, and the second disadvantage is the inefficiency of the word updates. This section presents two enhancements to the update-based protocols that address these shortcomings.

5.1 Word synchronization

When using an update-based protocol, block synchronization prevents the consumer from using data until the synchronization point is reached, even though the desired data may already be in the consumer's cache as the result of an update. A finer grain synchronization, such as word synchronization, would allow the consumer to proceed as soon as the data is available, as shown in figure 6.

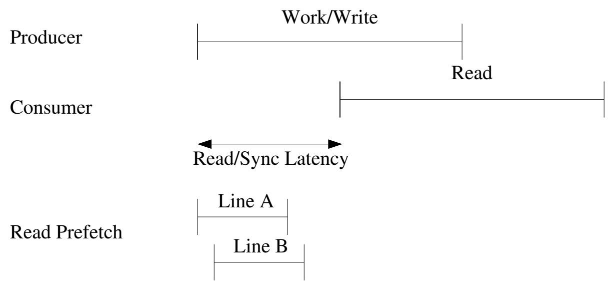


Figure 6: Word Synchronization

Word synchronization combines the synchronization and data transfer into a single operation. This combination eliminates the explicit synchronization and fence latencies and update acknowledgments.

Word synchronization may be implemented in hardware or software. In a hardware implementation, each data word has a valid bit associated with it [18]. The valid bit is initialized to invalid, and when the producer writes the data, the valid bit is set. The consumer spins on the valid bit waiting for it to become valid. For a software implementation, an unused bit pattern may be used to represent an invalid word. For example, in a floating point application, a not-a-number, NaN, code could be used to represent an invalid number, and in a pointer based application, a null pointer could be used to represent an invalid pointer.

All applications currently under study use a software-based scheme. The data is initialized to an invalid value, and the data reads are encapsulated within a while loop. The loop waits for the data to become valid (not invalid). In iterative applications, the producer is required to clear the data, by setting each word to an invalid value, between iterations. The use of dual buffering prevents any data races in these applications.

Invalidate-based protocols may see performance improvements from the finer grain synchronization as well, but the possibility of significant line bouncing or thrashing makes word synchronization extremely unstable. For example, if the consumers all read the first word of data before the producer writes it, the producer will be forced to invalidate each consumer's copy before writing the word. Once the word is written, all the consumers will reread the line and begin consuming the first data word. Each consumer read between producer writes will result in an invalidate and reread of the line. In a worst case scenario, the consumers may attempt to read each data word before it is written, which will force the line to be transferred back and forth between the consumers and the producer for each word! The resulting increase in network traffic and latency can significantly increase the total execution time. This effect is similar to that of false sharing with invalidate-based protocols.

5.2 Write combining (grouping)

Write combining can be used to combine, or group, single writes into larger, more efficient update packets to reduce congestion and latency. To simplify the required hardware, the combining is limited to the write buffer and to sequential writes to the same memory line.

The combining scheme requires the addition of a single bit per word in the write buffer as shown in figure 7. As new writes are inserted into the write buffer, the line address is compared with the line address of the last word entered into the write buffer. If the addresses match, the additional bit is set to 1 indicating that the write is to be grouped with the previous write. If the addresses do not match, the bit is set to 0 indicating a new group. As the cache processes the writes from the write buffer, it consumes all writes in a group. For example, in figure 7 the write buffer contains three groups of writes. The first write group contains 2 words, the second, a single word and the last group contains 3 words. Any new writes could still be combined with the last group.

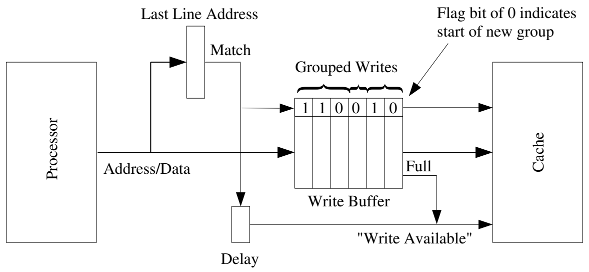


Figure 7: Hardware Combining Scheme

To improve combining of writes with modest temporal locality, write groups are delayed in the write buffer. The write group at the head of the write buffer is sent to the cache only if a new uncombinable write arrives from the processor, the write buffer fills or the delay counter expires.

The delay counter is currently initialized to 5 cycles, and is reset every time a new write is grouped with the last write. These conditions guarantee that each write group is visible to new, possibly combinable writes for a minimum of 5 cycles (a tight read, modify and write cycle).

Since writes often exhibit high spatial and temporal locality, combining at the write buffer can effectively group writes. If the spatial locality of the writes is low, then the single word updates will be more efficient than the line transfers of the invalidate protocols, and combining is not necessary. If the temporal locality is low, then the single word updates will not congest the system as they will be sufficiently spaced in time.

The combining scheme presented here is only one of many possible schemes. Any scheme that is able to combine writes with high temporal and spatial locality without introducing significant latency will work just as well.

6 Simulated architecture

The simulated architecture consists of 64 sites arranged in an 8 by 8 mesh. Each site consists of a processor-memory element (PME) as shown in figure 8.

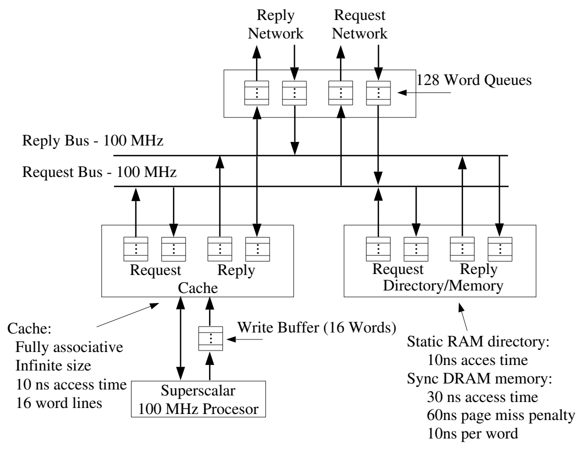


Figure 8: PME Architecture

The processor is a 32 bit, 100 MHz superscalar processor that is assumed to be load and store limited, and the cache is fully associative cache with infinite size. An infinite size cache is used to separate the effects of a limited cache size and the actions required by the cache coherence protocols. The effects of replacements will be examined in future work. The cache has a single cycle access time, a line size of 16 words and is connected to the network and memory by a 100 Mhz, 32 bit bus. Each memory module consists of a single bank of 100 Mhz synchronous DRAMs supporting page mode operation. The SDRAM have 30 ns access time for a page access with a page miss penalty of an additional 60 ns. The directory consists of a 10 ns access time SRAM for all protocols.

Each PME is connected to both a reply and request network to avoid deadlock. Request-reply deadlock is avoided by guaranteeing that replies will eventually be consumed at their destination. Request-request and reply-reply deadlock requires a timeout to break the deadlock [20]. The network is order preserving with static, wormhole routing [5] and multicast. Multicast is only used by the CD protocols when there are multiple caches to update or invalidate.

7 Results

7.1 Description of applications

To compare the cache coherence protocols, a set of applications must be specified which represent an important domain for large scale shared-memory multiprocessors. One such domain is the scientific and engineering domain. The applications studied here include a simple, iterative partial differential equation solver (PDE) and three different methods of factorizing a matrix into triangular matrices: a multifrontal solver (MF)[1], sparse Cholesky factorization (SPCF), and LU decomposition.

As described in section 4, the important characteristics of these applications for this study are the number of consumers per block and the line utilization. These are summarized in table 1.

Benchmark	MF	PDE	SPCF	LU
Data Set	1kx1k	32x32	1138x1138	64x64
Data Blocks	332	1120	1138	2016
Block Size	78.5	8	1.80	31.5
Consumers	1	1	1.89	31.5
Line Util. %	85.0	50.0	11.2	80.1

Table 1: Benchmark Characteristics

In all four applications, the producer's data blocks are allocated at the producer's local memory. In the SPCF, LU and MF applications, the processes are allocated to processors using a wrap-mapping, where column or node i is assigned to processor i modulo P where P is the total number of processors. In the PDE application, the processes are randomly allocated to processors to simulate a more complicated mesh of application elements.

[1]The MF application has two distinct phases of operation [15]. Initially, the application exhibits large amounts of parallelism in the computation of independent submatrices. As the computation continues, the number of independent submatrices decreases at which time each submatrix can be computed using a parallel LU technique. This study of the MF application studies only the first phase of this computation.

The Care/Simple simulation environment was used to simulate the performance of the applications [16]. The simulations were execution driven rather than trace driven.

The next four sections describe the simulation results for the four applications studied. This discussion begins by examining applications with a single consumer and modest to high line utilization: MF and PDE. Next, the impact of a small number of consumers will be examined by studying the SPCF application, and finally, the impact of both high line utilization and multiple consumers is examined in the LU application. For each application, a graph of the relative execution time of each protocol compared to the base CD-INV invalidate-based protocol is presented. On each graph, four differences are labeled (one set for each update protocol). Difference BS represents the difference in the update-based and CD-INV protocol using block synchronization without write buffer combining, and difference BS-C represents the improvement in the update protocols when write buffer combining is added. Difference WS represents the improvement in the update protocols using word synchronization, and difference WS-C represents the improvement from the addition of write buffer combining to word synchronization.

7.2 MF application

For the block synchronization case without combining, the CD-UP protocol increases execution time by 26% compared to the CD-INV protocol, and the DD-UP protocol has similar performance to the CD-INV protocol as shown by difference BS in figure 9. The update protocols were able to reduce the read latency by updating the consumers' caches, but because of the high line utilization of the application, these single word updates were very inefficient compared to the line transfers of the invalidate protocols. These inefficient updates increased the congestion in the system which resulted in longer fence and synchronization latencies.

With the addition of combining, the efficiency of the updates improved, and the congestion decreased. The combining improved the execution time of the CD-UP and DD-UP protocols by 29% and 5% respectively as shown by difference BS-C in figure 9. Combining was more effective in the CD-UP protocol as the larger

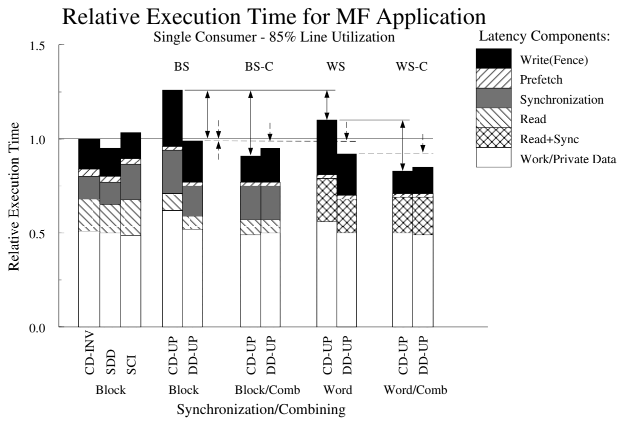


Figure 9: Relative Execution Time for MF

update packets reduced the average overhead of each memory update. The update protocols were now able to outperform the CD-INV protocol by a slight margin.

With single word updates significantly congesting the system, word synchronization was able to hide only a small portion of the update latency which resulted in a minor improvement of the execution time for the update protocols as indicated by difference WS in figure 9. Word synchronization improved the performance of the CD-UP protocol by 13% and the performance of the DD-UP protocol by 8%.

The use of both word synchronization and combining allowed the update protocols to improve performance by 25% and 6% for the CD-UP and DD-UP protocols respectively compared to the non-combining case as shown by difference WS-C in figure 9. As in the block synchronization case, combining reduced the congestion, which decreased the write latency compared to the word synchronization case without combining.

Overall, the update protocols with the enhancements were able to outperform the invalidate protocols. For the CD-UP protocol, the block synchronization case with combining performed better than the word synchronization case improving the performance of the application by 10% compared to the CD-INV protocol. For the DD-UP protocol, the performance of the word synchronization case was slightly better than the block synchronization case with combining improving the execution time by 9% compared to the CD-INV protocol. The use of both word synchronization and combining allowed the update protocols to improve performance of the application by 18% and 14% for the CD-UP and DD-UP protocols respectively compared to the CD-INV protocol.

7.3 PDE application

The update protocols using block synchronization were able to decrease the total execution time of the PDE application by 3% and 20% compared to the CD-INV protocol for the CD-UP and DD-UP protocols respectively as shown by difference BS in figure 10. The main source of the improvement was a reduction in the synchronization latency resulting from the update of the semaphores. The update of the shared data also reduced the read latency, but the overall impact was small since the invalidate protocols were able to effectively prefetch the necessary data.

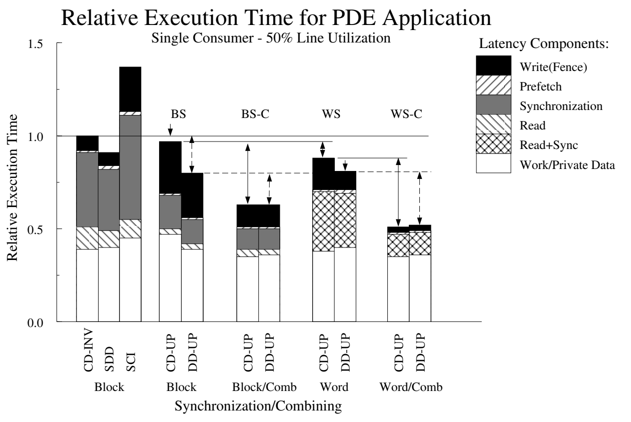


Figure 10: Relative Execution Time for PDE

Combining was able to group the 8 data writes destined for each neighboring node into single, efficient update packets. These fewer, larger packets reduced the overhead and congestion in the network and at the caches. This reduction in congestion resulted in a decrease in the fence latency which in turn reduced the synchronization latency. The sum of these latency reductions resulted in an improvement in the execution time of 36% and 21% for the CD-UP and DD-UP protocols respectively compared to the non-combining case as shown by difference BS-C in figure 10.

The improvement in execution time of the update protocols using word synchronization was minimal compared to the block synchronization case without combining as shown by difference WS in figure 10. In this iterative application, the producer was required to clear the data between iterations. This extra traffic limited the improvement in performance from word synchronization.

As in the block synchronization case, combining was able to combine all writes destined for each consumer into a single update packet. The resulting synchronization and read latency was reduced to half of that of the update protocol using block synchronization. This reduction in latency along with the elimination of the fence latency reduced the execution times of the update protocols by 47% and 35% for the CD-UP and DD-UP protocols respectively as shown by difference WS-C in figure 10.

Overall, the update protocols were able to significantly improve the execution time of this iterative application. The update protocols performed well in the block synchronization case, and the addition of combining to the block synchronization case improved the execution time by about 35%. The use of word synchronization and combining allowed the update protocols to improve the execution time by 51% and 53% for the CD-UP and DD-UP protocols compared to the CD-INV protocol.

The difference between the CD-UP and DD-UP protocols arises from the difference in the path of the updates. In the CD-UP protocol, updates are sent to the directory where they are forwarded to the consumer. In the DD-UP protocol, the path of the update depends on the producer's position in the cache list. If the producer is at the head of the list, the update is sent directly to the consumer, but if the producer is not at the head of the list, the update must be sent to the directory first and then forwarded to the consumer at the head of the list. In this particular application, each update required an average of 1.5 update hops indicating that the producer's cache was at the head of the list half the time. The reduction in update hops allowed the DD-UP protocol to perform slightly better than CD-UP for the non-combining cases. When combining is introduced, the reduction in congestion reduced the cost of the extra half hop which minimized the difference in performance.

7.4 SPCF application

When using block synchronization, the CD-UP and DD-UP protocols were able to reduce the total execution time of the SPCF application by 22% and 3% respectively compared to the base CD-INV protocol as shown by difference BS in figure 11. The main source of this reduction was a decrease in the synchronization latency which accounted for a significant portion of the execution time. The reduction in read latency was minimal as the invalidate protocols were able to use prefetch to hide a large portion of the read latency.

Since the application has a low line utilization, the combining scheme had little opportunity for combining. In both update protocols, the combined update packet contained an average of only two words, which resulted in an improvement of the execution time for both protocols of only a few percent as shown by difference BS-C in figure 11.

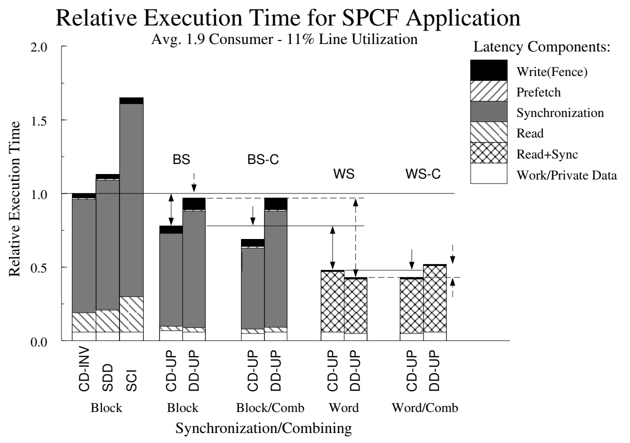


Figure 11: Relative Execution Time for SPCF

In this application, word synchronization allowed for the elimination of the costly explicit synchronization. With each consumer only consuming a fraction of the data produced by each producer, the cost of an explicit synchronization semaphore was large. As a result of the elimination of this explicit synchronization and the fence operations, the execution time of the CD-UP protocol decreased by 39% and the execution time of the DD-UP protocol decreased by 42% as shown by difference WS in figure 11.

The use of combining with word synchronization had a negligible impact on execution time for both protocols as shown by difference WS-C in figure 11. For the DD-UP protocol, the extra latency introduced by the combining scheme actually increased the total execution time as very few writes were combined. For both protocols, the early consumption permitted by word synchronization was able to hide a majority of the latency of subsequent updates.

The difference between the CD-UP and DD-UP protocol was due to the increased latency introduced by the longer sharing lists of caches in the DD-UP protocol. On the average, the list of caches was only 2.3 caches long, but the maximum reached 10 caches. The longer list increased the fence latency as updates were required to traverse the entire list before being acknowledged, and the synchronization latency also increased as caches at the end of the list had to wait longer before receiving updates.

7.5 LU application

For the LU application with a high line utilization and a large number of consumers, the single word updates were extremely inefficient. The updates created significant congestion in the system, and the large number of consumers increased the fence latency since each update had to be acknowledged. The resulting execution times of the update protocols using block synchronization were over twice that of the CD-INV protocol as shown by difference BS in figure 12.

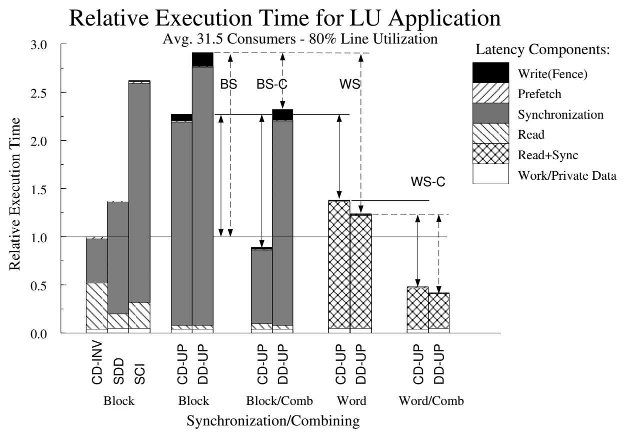


Figure 12: Relative Execution Time for LU

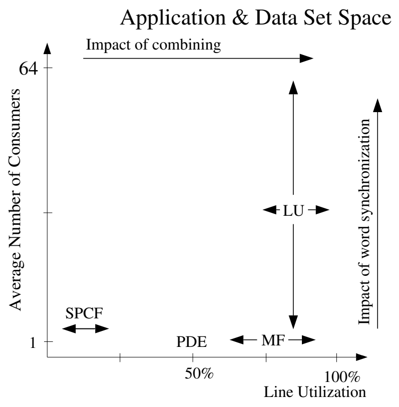
The high line utilization allowed for ample combining. The fewer, larger combined update packets decreased the fence and synchronization latencies. The resulting improvement in performance for the update protocols is shown by difference BS-C in figure 12. For the CD-UP protocol, the resulting reduction in execution time was significant indicating that congestion dominated the execution time in this case. But for the DD-UP protocol, these reductions did little to improve the execution time indicating that the latency from the longer list of caches dominated the execution time.

The addition of word synchronization eliminated the fence latency and allowed consumption of early data words to hide the update latency of subsequent words. In the CD-UP protocol, the early consumption helped to hide a large portion of the congested update latency and improve the performance of the protocol improved by 39% as shown by difference WS in figure 12. For the DD-UP protocol, the word synchronization improved the performance by 57%, and the elimination of the fence latency reduced the performance impact of the longer lists of caches. Caches began consuming data as soon as it arrived. Caches at the end of the list only experienced a long wait for the first word of the data to arrive; the subsequent words followed closely behind the first.

The use of combining with word synchronization improved performance even more. Combining improved the performance of the update protocols by 65% and 66% for the CD-UP and DD-UP protocols respectively as shown by difference WS-C in figure 12. In both protocols, the combining decreased congestion in the network and caches. In the CD-UP protocol, the larger update packets reduced the average memory update latency, as the memory access overhead per word decreased with increasing update packet size.

8 Summary and conclusions

The relative performance of the update protocols was dependent on the line utilization and the number of consumers. Figure 13 summarizes these characteristics for the applications that were studied.



Application & Data Set Space
Impact of combining
64
1
Average Number of Consumers
Impact of word synchronization
LU
SPCF
PDE
MF
50%
100%
Line Utilization

Figure 13: Application Space

The line utilization determined the efficiency of the updates. In the applications with a low line utilization, the updates were more efficient than the line transfers of the invalidate protocols, but as the line utilization increased the efficiency of the updates decreased. These inefficient updates tended to congest the network, caches and memories. Combining was introduced to address this problem.

Combining was able to group single updates into larger update packets, which improved the efficiency of the updates and decreased congestion. This improvement in efficiency was dependent on the line utilization as indicated in figure 13. When the line utilization was low, the possibility of combining was small, but as the line utilization increased the combining scheme was able to combine a significant number of updates.

These new update packets were as efficient as the line transfers of the invalidate protocols, even for the applications with line utilizations approaching 100%. The improvement in the execution time of the update protocols when combining was added ranged from 5% to 61% for the block synchronization case; the improvement was largest for the applications with the highest line utilization.

Word synchronization eliminated the explicit synchronization semaphores from the applications and allowed the consumers to begin consuming the data as soon as possible. The elimination of the semaphores removed the need for update acknowledgements and for the fence operation. In applications with few consumers, this had little impact on performance, but as the number of consumers increased, the impact of the elimination of the update acknowledgements increased, especially in the DD-UP protocol. In this protocol, the latency of the update acknowledgements and the fence operation was dependent on the number of the consumers, which determined the length of the list of caches that each update had to traverse before being acknowledged. The improvement in execution time of the update protocols ranged from 10% to 40% for the CD-UP protocol and from no improvement to over 50% for the DD-UP protocol for the block synchronization case.

The use of both enhancements allowed the update protocols to significantly improve the performance of the applications when compared to the CD-INV protocol; the improvements in execution times ranged from about 15% to over 50%. The applications with both high line utilization and a larger number of consumers benefited the most from the enhancements. Even in applications with high line utilization and a single consumer, which tend to favor the invalidate protocols, the update protocol was able to improve the execution time.

With both enhancements, the difference between the CD-UP and DD-UP protocols was small. Combining improved the performance of the CD-UP more than the DD-UP protocol, as the overhead of updating each memory word decreased with the larger packet size. Word synchronization had the largest impact on the DD-UP protocol. It eliminated the need for update acknowledgments which reduced the impact of the length of the sharing list on the performance of the protocol. The choice of which update protocol to use will be dependent on other issues such as the scalability of the directory structure.

Overall, this paper has demonstrated that the performance of the applications studied here can be significantly improved by update-based protocols. The efficiency of the updates, which became a problem in applications with higher line utilization, could be improved by combining. The use of a finer grain (word) synchronization was shown to be effective in improving performance by both eliminating the congestion and latency of explicit synchronization and update acknowledgments and by hiding the update latency behind consumption of earlier data words. Together, these latency reduction and hiding techniques allow the update protocols to significantly improve the execution time of the applications studied when compared to the three invalidate-based protocols examined.

References

[1] Sarita Adve and Mark Hill, "Weak ordering - A new definition," *Proc. 17th International Symposium on Computer Architecture*, pages 2-14, 1990.

[2] John Bennett, John B. Carter and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," Technical Report Rice COMP TR90-108, Department of Electrical and Computer Engineering and Department of Computer Science, Rice University, 1990.

[3] Roberto Bisiani and Mosur Ravishankar, "PLUS: A Distributed Shared-Memory System," *Proc. 17th International Symposium on Computer Architecture*, pages 115-124, 1990.

[4] David Chaiken, John Kubiatowicz and Anant Agarwal, "limitLESS Directories: A Scalable Cache Coherence Scheme," *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224-234, 1991.

[5] William J. Dally and Charles L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, no. 5, pages 547-553, May 1987.

[6] Iain S. Duff, "Parallel Implementation of Multifrontal Schemes," *Parallel Computing*, vol. 3, pages 193-204, 1986.

[7] S. Eggers and R. Katz, "Evaluating the Performance of Four Snooping Cache Coherence Protocols," *Proc. 16th International Symposium on Computer Architecture*, pages 2-15, May 1989.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors" *Proc. of the 17th International Symposium on Computer Architecture*, pages 15-26, 1990.

[9] IEEE "Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications, P1596" IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, November 1991.

[10] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta and John Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th International Symposium on Computer Architecture*, pages 148-159, May 1990.

[11] K. Li and P. Hudak, "Memory Coherence in shared virtual memory systems", *ACM Transactions on Computer Systems*, vol. 7, no. 4, pages 229-359, November 1989.

[12] Todd Mowry and Anoop Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, pages 87-106, 1991

[13] Bill Nitzberg and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", Technical Report CIS-TR-90-26, Department of Computer and Information Science, University of Oregon, January 1991.

[14] U. Ramachandran, M. Ahamad and M. Y.A. Khalidi, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfers", *Proc. 1989 International Conference on Parallel Processing*, pages II160-II169, August 1989.

[15] Edward Rothberg and Anoop Gupta, "A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results," Technical Report CSL-TR-90-416, Computer Systems Laboratory, Stanford University, February 1990.

[16] Nakul P. Saraiya, Bruce A. Delagi and Sayuri Nishimura, "SIMPLE/CARE an Instrumented Simulator for Multiprocessor Architectures" Technical Report KSL-90-66, Knowledge Systems Laboratory, Stanford University, 1990.

[17] Richard Simoni and Mark Horowitz, "Modeling the Performance of Limited Pointers in Directories for Cache Coherence," *Proc. 18th International Symposium on Computer Architecture*, pages 309-318, May 1991.

[18] Burton J. Smith, "Architecture and application of the HEP multiprocessor computer system," *Real Time Signal Processing IV*, SPIE vol. 298, pages 241-248, August 1981.

[19] Per Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, vol. 23, no. 6, pages 12-24, June 1990.

[20] Manu Thapar, "Cache Coherence for Scalable Shared Memory Multiprocessors," Technical Report CSL-TR-92-522, Computer Systems Laboratory, Stanford University, May 1992.

[21] Manu Thapar, Bruce Delagi and Michael J. Flynn, "Linked List Cache Coherence for Scalable Shared Memory Multiprocessors", *7th International Parallel Processing Symposium*, April 1993.

[22] Wolf-Dietrich Weber and Anoop Gupta, "Analysis of cache invalidation patterns in multiprocessors", *Proc. 3rd International Conference on Architectural Support for Programming Languages and Systems (ASPLOS III)*, pages 243-256, April 1989.

[23] Andrew W. Wilson Jr. and Richard P. LaRowe Jr., "Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture," *Journal of Parallel and Distributed Computing*, vol. 15, pages 351-367, 1992.

[24] Andrew W. Wilson Jr., Richard P. LaRowe Jr. and Marc J. Teller, "Hardware Assist for Distributed Shared Memory", *13th International Conference on Distributed Computing Systems*, May 1993.

[25] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla and Steven K. Reinhardt, "Mechanisms for Cooperative Shared Memory,", *Proc. 20th Annual International Symposium on Computer Architecture*, May 1989.