The Impact of Cache Coherence Protocols on Systems Using Fine-Grain Data Synchronization

David B. Glasco, Bruce A. Delagi and Michael J. Flynn

Computer System Laboratory, Stanford University, Stanford, CA 94305

Abstract: In this paper, the performance of a fine-grain data synchronization scheme is examined for both invalidate-based and update-based cache coherent systems. The work first reviews coarse-gain and fine-grain synchronization schemes and discusses their advantages and disadvantages. Next, the actions required by each class of cache coherence protocols are examined for both synchronization schemes. This discussion demonstrates how invalidate-based cache coherence protocols are not well matched to fine-grain synchronization schemes while update-based protocol are.

To quantify these observations, five scientific applications are simulated. The results demonstrate that fine-grain synchronization always improves the performance of the applications compared to coarse-grain synchronization when update-based protocols are used, but the results vary for invalidate-based protocols. In the invalidate-based systems, the consumers of data may interfere with the producer. This interference results in an increase in invalidations and network traffic. These increases limit the possible gains in performance from a fine-grain synchronization scheme.

Keyword Codes: B.3.3; C.1.2
Keywords: Performance Analysis and Design Aids; Multiple Data Stream Architectures (Multiprocessors)

1 Introduction

In shared-memory multiprocessors, data is often shared between a producer and one or more consumers. To prevent the consumers from using stale or incorrect data, the consumers must not access the data until the producer notifies them that it is available. Typically, such systems use a coarse-grain synchronization scheme to synchronize this production and consumption of data.

In such schemes, simple flags can be used to indicate that a given block of data has been produced. For example, the producer of a data block first writes the data to a shared buffer and then issues a fence instruction that stalls the processor until all writes have been performed. Finally, the producer sets the synchronization flag. The consumers, who have been waiting for the synchronization flag to be set, see the flag set and begin consuming the data. Coarse-grain schemes may also use other synchronization methods such as barriers. The coarse-grain synchronization schemes examined assume a release consistency memory model [2].

A coarse-grain synchronization scheme has two basic disadvantages. First, the scheme

requires an expensive synchronization operation such as a flag or barrier to synchronize the production and consumption of data. Second, the consumers are forced to wait until the entire data block has been produced before they are able to begin consuming the data.

An alternative to the coarse-grain synchronization scheme discussed above is a fine-grain scheme. In such a scheme, the synchronization information for each data word is combined with the word. In this case, the producer creates the data and writes it to a shared buffer. The consumers wait for the desired word to become available and then consume it.

Fine-grain synchronization has several advantages. First, the consumers are able to consume data as soon as it is available. This allows the consumption time of the available data to overlap the production time of the subsequent words. Moreover, no expensive synchronization operation is required; this means that the producer never needs to wait for the writes to be performed.

In this work, we are interested in studying the performance gains from a fine-grain synchronization scheme on scalable, cache coherent shared-memory multiprocessors. We will show that fine-grain synchronization may improve performance of applications running on systems with either invalidate-based or update-based cache coherence protocols, but that fine-grain synchronization is not a robust solution for invalidate-based systems while it is for update-based systems.

The paper is organized as follows. Section 2 describes a typical coarse-grain synchronization scheme and demonstrates the resulting work required by each class of cache coherence protocols. Next, section 3 describes a fine-grain synchronization scheme and demonstrates how the scheme overcomes the disadvantages of the coarse-grain scheme. Section 4 describes the simulated architecture, the scientific applications and the cache coherence protocols examined in this work. Section 5 presents the simulation results. These results compare the performance of fine-grain synchronization to coarse-grain synchronization, and they also compare the relative performance of the cache coherence protocols when fine-grain synchronization is used. Finally, section 6 concludes the paper.

Our work is the first to examine the performance of fine-grain synchronization on both update-based and invalidate-based cache coherent multiprocessors. The work on the Alewife [6] system examines the implementation details of a fine-grain synchronization scheme, and their work gives an excellent description of the software and hardware requirements for such a scheme. In their work, they demonstrate that a fine-grain synchronization scheme may improve the performance of the given applications running on an invalidate-based cache coherent system. Our work does not contradict their findings, but rather expands them to demonstrate the instability of the combination of invalidate-based cache coherence protocols and fine-grain data synchronization.

2 Coarse-grain data synchronization

Figure 1a shows the actions required for invalidate-based protocols using a coarse-grain synchronization scheme. After producing the data, the producer writes it to a shared buffer and waits for the writes to be performed. The writes are considered performed when the producer's cache has obtained exclusive ownership of the line (all necessary invalidations have been performed) and the data has been written into the cache. The figure assumes that the producer has already obtained exclusive ownership of the data lines. After the writes have been performed, the producer sets the synchronization flag. If the consumers have already read the flag, then this write must invalidate the consumers' copies of the flag. The consumers, who are still waiting for the flag to be set, will imme-

diately reread the flag after it is invalidated, but the producer cannot release the flag's line until all the invalidations have been performed. Once the consumers see the flag set, they can begin reading and consuming the data.

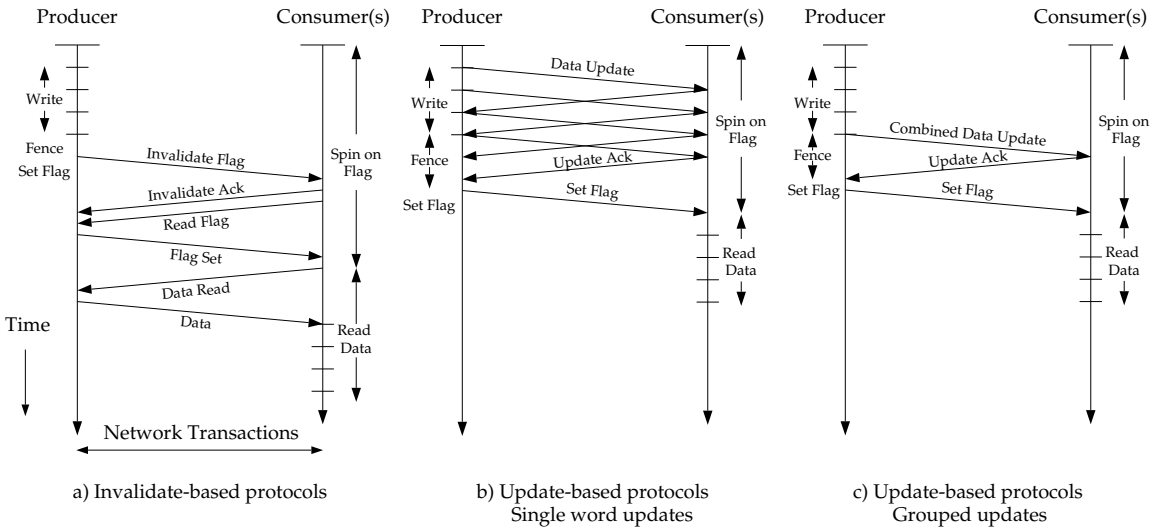


Figure 1: Coarse-grain synchronization

Figure 1a illustrates the cost of a coarse-grain synchronization scheme using a simple flag. The invalidation and reread of the flag by each consumer requires four network transactions and the transfer of a line of data. The work to transfer the synchronization information is more than the work to transfer one line of data. The size of the data block could be increased to reduce this synchronization overhead, but the larger block size also increases the waiting time of the consumers.

Figure 1b shows the actions required for update-based protocols. In this case, all of the consumers have prefetched the synchronization flag and data block before the data is written. When the producer writes the data, the consumers' caches are updated. The producer must wait for the writes to be performed (all updates acknowledged) before setting the synchronization flag. The producer's write of the flag results in the consumers' caches being updated. When the consumers see the flag updated, they can begin using the data, which is already in their caches. Figure 1c shows the resulting network transactions if a write-grouping scheme, as described in our earlier work [5, 4], is used. In this case, the data updates are grouped into larger, more efficient updates.

As in the invalidate-based case, the coarse-grain synchronization scheme has disadvantages. First, the cost of the coarse-grain synchronization operation is high. But the cost is not in transferring the synchronization information itself, it is in waiting for the updates to be performed (acknowledged). This synchronization scheme also forces the consumers to wait until the synchronization point is reached before accessing the data, but, unlike the invalidate-based case, the desired data is already in the consumers' caches as a result of the earlier updates. The coarse-grain synchronization scheme does not allow the system to take advantage of these fine-grain updates.

3 Fine-grain data synchronization

A fine-grain synchronization scheme would overcome many of the disadvantages of the coarse-grain scheme described in the last section. In a fine-grain synchronization scheme, the synchronization information is combined with the data. Such a scheme may be implemented in either hardware or software. In a hardware-based scheme, a full/empty bit is associated with each memory word [8]. Alternatively, a software-based scheme may be used in which an invalid code, such as NaN in a floating point application, is used to indicate an empty word. Currently, all the applications under study use a software-based scheme.

Figure 2 demonstrates how a producer and consumer interaction might be coded for both coarse-grain and fine-grain (using a software-based scheme) data synchronization. For the coarse-grain case, a simple flag, initialized to false, is used to synchronize the production and consumption of data. For the fine-grain case, the data is initialized to an invalid code. The consumer waits for each word to become valid and then consumes it. For iterative applications, the data must be set to the invalid code between iterations.

Coarse-Grain:

Producer:

```
shared a[Count];
shared flag = false;

/* Produce data */
for (i = 0; i < Count; i++)
  a[i] = f();
/* Wait for writes to complete */
fence();
/* Set Flag */
flag = true;
```

Consumer:

```
shared a[Count];

/* Wait for flag */
while (flag == false) ;
/* Consume a[i] */
for (i = 0; i < Count; i++)
  b[i] = f(a[i]);
```

Fine-Grain:

Producer:

```
shared a[Count] = INVALID;

/* Produce Data */
for (i = 0; i < Count; i++)
  a[i] = f();
```

Consumer:

```
shared a[Count];

/* Consume a[i] */
for (i = 0; i < Count; i++) {
  /* Spin waiting for data */
  while (a[i] == INVALID) ;
  b[i] = f(a[i]);
}
```

Figure 2: Code examples for coarse-grain and fine-grain synchronization

By their very nature, invalidate-based protocols are not well matched to fine-grain synchronization schemes. The protocols do not allow consumers to maintain copies of a data line while a producer writes to the line. This results in a very unstable solution. Figure 3a shows the ideal timing diagram for invalidate-based protocols when a fine-grain synchronization scheme is used. First, all the consumers read the first word of the data block. When the producer writes this word, all the consumers' copies must be invalidated. But since the consumers are eagerly waiting for the data, they will immediately reread the data line once it is invalidated. The invalidate-based protocols studied allow the producer to continue writing into the cache line while invalidations are pending. This prevents the producer from observing any write delay as the consumers read and reread the line. In the ideal case, the invalidation latency is greater than the producer's write time for the line. When the consumers reread the line, they will find the line completely written. The producer will not invalidate the line again; this gives the consumers all the time they need to consume the line's data for this ideal case.

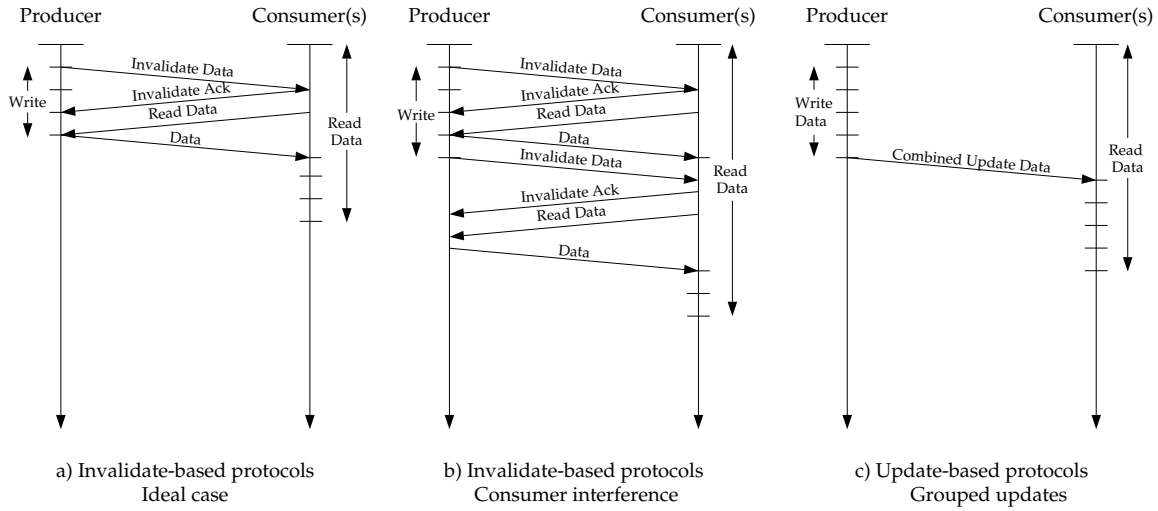


Figure 3: Fine-grain synchronization

However, figure 3b demonstrates the problem with invalidate-based protocols and fine-grain synchronization schemes. In this case, the consumers have reread the line after the initial invalidation but before the producer has completed writing the data. Now the producer is required to invalidate the consumers' copies of the line again, and the consumers are forced to reread the line. The producer is able to release the line again only after all the pending invalidations have been performed.

The relative timing of the writes and reads will have an enormous impact on the performance of the system. If the writes occur in bursts, as they often do, the producer will usually be able to produce many words of data between each reread by the consumers. But the consumers, who will reread the line immediately after it is invalidated, will only receive the data after all the consumers' copies of the line have been invalidated. The consumers will then be able to consume data only until the producer invalidates the line again, which may occur soon after the consumer receives the data if the write rate is high.

The frequency with which these invalidations and rereads occur depends on two characteristics of the application. First, the probability that the producer finishes writing the line before the consumers attempt to reread it depends on the number of words written to the line. This measure, known as the line utilization, will be used to classify the application space. The other characteristic is the number of consumers. The more consumers, the higher the probability that consumers will interfere with the producer's writing of the data.

The definition of update-based protocols offer a better match to fine-grain synchronization schemes. Unlike the invalidate-based protocols, update-based protocols allow consumers to maintain a copy of the data line while a producer writes to the line. Also, the producer's write of the data results in an update of all the consumers' caches: a proactive distribution of data rather that a reactive approach as in the invalidate-based protocols in which each consumer is responsible for refetching an invalidated line. For example, figure 3c shows the actions required for update-based protocols using fine-grain data synchronization and write grouping. First, all the consumers prefetch the desired data lines. As the producer writes the data, the writes are grouped and sent to the consumers, and the consumers can consume the data as soon as it arrives.

Update-based protocols can also take advantage of the fine-grain synchronization scheme in another way. Update-based protocols using a coarse-grain synchronization scheme require that updates be acknowledged before the synchronization flag is set, but in a fine-grain synchronization scheme, no update acknowledgment is needed because the producer is never required to wait for the updates to be performed. This has the largest impact on the distributed-directory update-based (DD-UP) protocol described in our earlier work [4].

The update-based protocols offer a much more robust solution. The data prefetch cannot degrade the performance of the fine-grain synchronization scheme as it can with the invalidate-based protocols [4]. The prefetch can be issued as early as desired by the consumers. Also, the relative timing of the producer's writes and consumers' reads can not affect performance as in the invalidate-based protocols. *The amount of work is fixed regardless of any variations in the timing of reads or writes.*

4 Simulation methodology

The Care/Simple simulation environment [1] was used to simulate a set of scientific applications running on a shared-memory multiprocessor. The simulated architecture consists of 64 nodes arranged in an 8 by 8 mesh. Each node consists of a processor/memory element (PME) connected to its four nearest neighbors through a set of network queues. A PME consists of a processor, cache, directory/memory and network interface. The processor is a 100 MHz superscalar processor that is assumed to be load/store limited, and the cache is a fully associative cache with infinite size. The cache has a single cycle access time and a line size of 16 words. Each memory consists of a single bank of 100 MHz synchronous DRAMs supporting page mode operation. The SDRAMs 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. The network is order preserving with static, wormhole routing and multicast.

The applications studied here include a simple, iterative partial differential equation solver (PDE), a 3-D iterative partial differential equation solver using FFTs (3DFFT), and three different methods of factorizing a matrix into triangular matrices: a multifrontal solver (MF), sparse Cholesky factorization (SPCF), and LU decomposition.

Table 1 summarizes the important characteristics of the applications studied. The number of consumers for each data block gives a measure of general contention for each object and the maximum number of invalidates or updates that might be needed when the data is modified. The line utilization is the percentage of each memory line that is modified by the producer. For example, if the data is a dense vector, the producer is likely to modify all the words in the given line. This would result in a line utilization of 100%. If the data is a structure, the producer might only modify a few words, which would result in a low line utilization. In the invalidate-based protocols using fine-grain synchronization, these measures give an indication of the possible interference between the consumers and producer of a line of data. The larger the line utilization or number of consumers, the higher the probability of interference and extra invalidation and reread cycles. The table also indicates the type of the coarse-grain synchronization: a simple flag or barrier. The table specifies which applications are iterative and shows both the number of synchronization events in each application and the average number of words protected by each synchronization event.

The update-based protocols examined in this paper include the centralized-directory update-based protocol (CD-UP) and the distributed directory update-based protocol

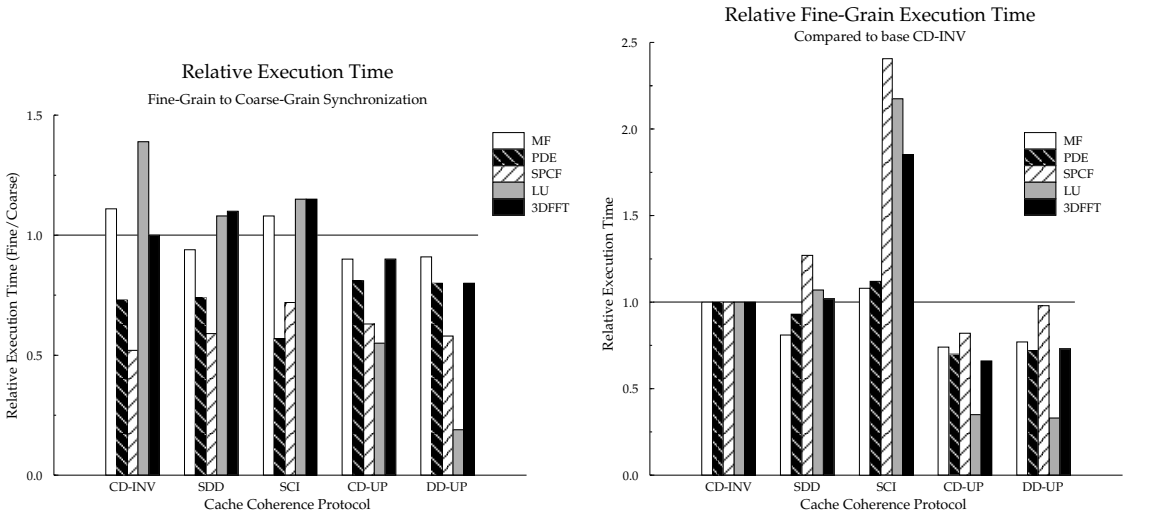
Applications	MF	PDE	SPCF	LU	3DFFT
Data Set (words)	1000x1000	32x32	1138x1138	64x64	8x8x16
Consumers	1	1	1.89	31.5	4
Line Utilization %	93.0	50.0	11.2	59.0	50.0
Sync Type	Flag	Flag	Flag	Flag	Barrier
Iterative	No	Yes	No	No	Yes
Sync Events	332	1120	2118	2016	678
Words/Sync Event	49.8	8.0	4.9	44.4	18.6

Table 1: Application characteristics

(DD-UP) described in our earlier work [4, 3]. The update-based protocols use the write-buffer grouping scheme also described in our earlier work [5]. The invalidate-based protocols examined include a centralized directory invalidate-based protocol (CD-INV), which is similar to DASH [7], a singly-linked distributed directory invalidate-based protocol (SDD) [9] and a doubly-linked distributed directory invalidate-based protocol (SCI), which is the IEEE standard protocol.

5 Results

The relative performance of the fine-grain synchronization scheme compared to the coarse-grain scheme and to a common base (CD-INV) is illustrated in figure 4. Table 2 gives the ratio of invalidations (updates) required and the relative change in total network traffic for the fine-grain synchronization case compared to the coarse-grain case for the invalidate-based (update-based) protocols.



a) Fine-grain execution time compared to coarse-grain

b) Execution time compared to common base (CD-INV)

Figure 4: Performance of fine-grain synchronization

For the invalidate-based protocols, the performance of the fine-grain synchronization scheme varies. For the applications with small blocks and few consumers (PDE and

	Invalidate-Based Protocols						Update-Based Protocols			
	Ratio of invalidations			Ratio of network traffic			Ratio of updates		Ratio of network traffic	
	CD-INV	SDD	SCI	CD-INV	SDD	SCI	CD-UP	DD-UP	CD-UP	DD-UP
PDE	0.93	0.93	0.93	0.67	0.55	0.51	1.85	1.91	0.87	1.09
SPCF	0.94	0.97	0.95	0.41	0.51	0.50	0.73	0.73	0.40	0.44
MF	10.7	7.08	4.07	0.81	0.81	0.80	0.98	0.84	0.84	0.84
3DFFT	1.00	1.08	1.14	0.98	1.07	0.99	1.38	1.38	0.84	1.13
LU	4.42	3.67	3.42	1.24	1.32	1.59	0.40	0.41	0.34	0.71

Table 2: Ratio of invalidations/updates and network traffic

SPCF), the fine-grain synchronization scheme improves the performance of the applications compared to the coarse-grain synchronization case. In these applications, the coarse-grain synchronization operation is costly, as each synchronization point protects only a small block of data: 8 words for the PDE application and 4.9 words for the SPCF application, as summarized in table 1. Therefore, the elimination of this coarse-grain synchronization operation outweighs any extra invalidations or network traffic generated by the fine-grain scheme. The fine-grain synchronization has actually reduced the total number of invalidations compared to the coarse-grain case for these two applications, as illustrated in table 2 for all the invalidate-based protocols. With the small block sizes of these applications, the producer was able to write the full block before the consumers could reread the line after the initial invalidation. The single invalidation per block acted as a synchronization or triggering event. The elimination of the explicit synchronization also significantly reduced the network traffic for these applications, as shown in table 2. This resulted in an improvement in execution times for the PDE and SPCF applications, as shown in figure 4a for the invalidate-based protocols.

As the line utilization increases, the consumer and producer interference also increases. For the MF application, the number of invalidations was small for the coarse-grain synchronization case. This indicates that the consumers were not always eagerly consuming the data. The fine-grain synchronization increased the number of invalidations significantly, but the overall traffic was reduced since the extra invalidation traffic was less than the traffic eliminated by the elimination of the explicit coarse-grain synchronization events. The actual execution time increased for the CD-INV and SCI protocols, but decreased for the SDD protocol. The difference in execution time between the invalidate-based protocols arises from the particular producer-consumer interaction that was interfered with. For the SDD protocol, the interference was off the critical timing path of the application and in the critical path for the other two invalidate-based protocols.

For the 3DFFT application, the iterative nature of the application required approximately twice the number of shared writes for the fine-grain synchronization case as compared to the coarse-grain case because the data must be set to an invalid code between iterations. As shown in table 2, these extra writes created at least as many invalidations as were eliminated by the fine-grain synchronization, and the resulting network traffic remained almost constant for the same reason. The small number of consumers increased the execution time of the distributed directory invalidate-based protocols (SDD and SCI) slightly more than the centralized-directory protocol (CD-INV). Overall, the fine-grain synchronization scheme did not improve the execution time for the 3DFFT application when invalidate-based protocols were used.

As the number of consumers and the line utilization increased, the consumer and producer interference also increased. For the LU application, fine-grain synchronization

increased the number of invalidations and network traffic, as illustrated in table 2. With relatively inexpensive coarse-grain synchronization in this application, the increase in invalidations and network traffic outweighed any performance gains from the elimination of the coarse-grain synchronization operations. Again, fine-grain synchronization offered no improvement in execution time for systems with invalidate-based protocols.

For the update-based protocols, fine-grain data synchronization always improved the performance of the applications, as illustrated in figure 4a. The fine-grain synchronization decreased both the number of updates and the network traffic for non-iterative applications (SPCF, MF and LU), as shown in table 2. For the iterative applications (PDE and 3DFFT), the extra writes to clear the data between iterations increased the number of updates for both update-based protocols. The network traffic was reduced for the CD-UP protocol, but it increased slightly for the DD-UP protocol.

The fine-grain synchronization scheme had the largest impact on the DD-UP protocol when the number of consumers was greater than one (SPCF, 3DFFT and LU). In these applications, the coarse-grain synchronization scheme required the producer to wait for the updates to be propagated down the list of caches and then acknowledged. This limited the performance of the DD-UP protocol; the fine-grain synchronization scheme removed the need for these acknowledgements.

Figure 4b shows the relative execution time of the applications using fine-grain synchronization compared to the centralized directory invalidate-based protocol (CD-INV). As shown in the figure, the update-based protocols always perform better than any of the invalidate-based protocols when a fine-grain data synchronization scheme is used.

For the invalidate-based protocols using fine-grain data synchronization, the SDD protocol performed better for the applications with a single consumer. In these cases, the memory write backs of the CD-INV protocol were unnecessary since the data was not read from memory again because cache-to-cache transfers were used to transfer the data to the single consumer. But as the number of consumers increased, the invalidation latency of the distributed directory protocols resulted in longer execution times compared to the base CD-INV protocol.

For the update-based protocols, the performance of the two protocols was almost identical except for the SPCF application. The improvement in performance compared to the base CD-INV protocol was best for applications with high line utilization and a large number of consumers (LU). As the number of consumers decreased, the improvement from the update-based protocols also decreased compared to the CD-INV protocol. Compared to the SDD protocol, the improvement was less for applications with a single consumer, but it was more for applications with multiple consumers.

6 Conclusions

In summary, the fine-grain data synchronization scheme, when used with invalidate-based cache coherent systems, did not offer a robust solution. It only improved the performance for applications with a small number of consumers (less than 2) and a low line utilization. These application characteristics tended to avoid consumer interference. On the other hand, systems with update-based protocols could take advantage of the fine-grain synchronization scheme. The resulting execution times were always less than the coarse-grain synchronization case, and they were always less than the corresponding execution times for the invalidate-based systems using fine-grain synchronization.

The performance of fine-grain synchronization was unstable when invalidate-based cache coherence protocols were used because the definition of the invalidate-based pro-

tocols prevented the producer and consumers from actively sharing a memory line. The producer was required to obtain exclusive ownership of the line before writes could be performed. Consumers who might be consuming data from the line were forced to give up their copy of the line. The simulated results of the fine-grain scientific applications demonstrate the performance loss that can occur as the producer and consumers interfere with each other.

The definition of update-based protocols offered a much better match to the fine-grain data synchronization. The protocols allow multiple producers and consumers to maintain copies of a given memory line at the same time; the producer and consumers of data can not interfere with each other. Overall, our earlier work and this work have demonstrated the performance gains that can be achieved with update-based cache coherence protocols.

References

[1] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. Instrumented Architectural Simulation. In *Proceedings of the Third International Conference on Supercomputing*, pages 8–11, March 1988.

[2] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip. Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[3] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Design and Validation of Update-Based Cache Coherence Protocols. Technical Report CSL-TR-94-613, Computer Systems Laboratory, Stanford University, March 1994.

[4] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 534–545, January 1994.

[5] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Write Grouping for Update-Based Cache Coherence Protocols. Technical Report CSL-TR-94-612, Computer Systems Laboratory, Stanford University, March 1994.

[6] David Kranz, Beng-Hong Lim, David Yeung, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. In *International Symposium on Computer Architecture*, 1992.

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

[8] Burton J. Smith. Architecture and Application of the HEP Multiprocessor Computer System. *Real Time Signal Processing IV*, 298:241–248, August 1981.

[9] Manu Thapar, Bruce A. Delagi, and Michael J. Flynn. Linked list cache coherence for scalable shared memory multiprocessors. In *7th International Parallel Processing Symposium*, April 1993.