Write Grouping for Update-Based Cache Coherence Protocols

David B. Glasco
glasco@umunhum.stanford.edu
Phone: (415)336-6760
Computer System Laboratory
Stanford University

Bruce A. Delagi
bruce.delagi@Eng.sun.com
Phone:(415)336-5201
Computer System Laboratory
Stanford University

Michael J. Flynn
flynn@umunhum.stanford.edu
Phone: (415)723-1450
Computer System Laboratory
Stanford University

Abstract

In our previous work, we demonstrated the possible performance gains from update-based cache coherence protocols for a set of fine-grain scientific applications running on a scalable shared-memory multiprocessor. In this paper, we examine in detail the hardware-based write grouping scheme presented in our earlier work. First we describe both software-based and hardware-based write grouping schemes. The software-based scheme, with its perfect knowledge of the application's write pattern, is able to achieve optimal write grouping efficiency, but not without added complexity to the application's code. Nevertheless, we use the software-based scheme to determine the optimal grouping efficiency for each application studied, and then demonstrate that the hardware-based write grouping scheme is almost as efficient as the software-based scheme, but it requires little, if any, software modifications.

We also explore slight modifications to the hardware-based write grouping scheme. These modifications include varying the delay used to improve the write grouping efficiency and changing the location of the write grouping buffer.

1 Introduction

In our previous work [4], we demonstrated the possible performance gains from update-based cache coherence protocols as compared to invalidate-based protocols for a set of fine-grain scientific applications running on a scalable shared-memory multiprocessor. The previous work identified two limitations of the update-based protocols: a mismatch between the granularity of synchronization and the fine-grain data updates and the inefficiency of single word updates. We proposed a fine-grain data synchronization scheme to allow for a better match between the granularity of synchronization and the fine-grain data updates [4]. This work is discussed in greater detail in [3]. We also proposed a write grouping scheme to improve the efficiency of the write updates [4], and in this paper, we further examine write grouping for update-based cache coherence protocols.

The goal of write grouping is to group single writes destined for the same memory line into larger, more efficient write groups. Grouping is able to improve performance in two ways. First, grouping writes allows the cost of the network header to be amortized across all words in the group. This reduces network traffic. For example, figure 1 shows the number of network words required to transfer a portion of a memory line using a line transfer, single word updates or a grouped update. The y-axis is the resulting network traffic, and the x-axis is the line utilization. The line utilization is the number of modified words in the memory line that must be transferred. For line transfers, as used by invalidate-based protocols, the network traffic is constant regardless of the line utilization. For single word updates, every write update requires 3 network words: a two word header plus the data word. But by grouping the writes, the resulting network traffic is always less than or equal to the traffic required by the line transfer; the grouped updates result in

the minimal network traffic necessary to transfer the line's modified data. The second advantage of grouping is that the cost of memory updates can also be amortized across more words in a grouped update than in a single word update.

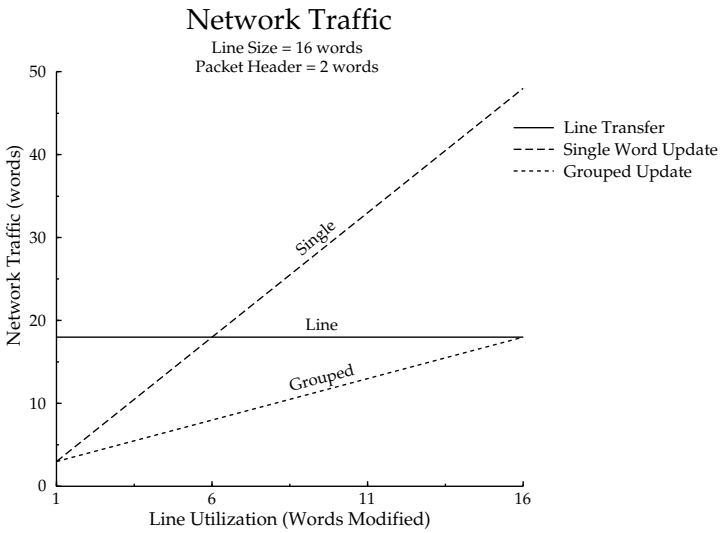

Figure 1: Network Traffic

Write grouping can be accomplished using a software-based or hardware-based scheme. In a software-based scheme, compile time analysis is used to identify writes to shared data. The compiler issues these writes with a special non-updating write and issues special write-line instructions to initiate the data updates. Section 2.1 explains the software-based scheme in more detail. In a hardware-based scheme, writes to the same memory line can be grouped into larger write groups at the write buffer, as described in section 2.2.

The paper is organized as follows. Section 2 describes the software-based and hardware-based grouping schemes in more detail. Section 3 describes the simulated architecture, the cache coherence protocols, and the scientific applications studied in this paper. Section 4 examines the performance of the grouping schemes, and section 5 examines variations in the hardware-based grouping scheme. These include varying the delay introduced to increase the opportunity

for write grouping and changing the location of the write grouping buffer. Finally, section 6 concludes the paper.

2 Write grouping

2.1 Software-based write grouping

With perfect knowledge of the application's write pattern, software-based write grouping is able to achieve optimal efficiency by grouping all shared writes to the same memory line into a single write group. Compile time analysis is used to identify writes to shared data, and the compiler would issue these writes using a special write-no-update instruction. This instruction would write the word into the cache and set a special update-pending bit for the word[1]. After all the words written to a given memory line have been issued, a special write-line instruction is issued that causes the words in the cache line with the pending-update bit set to be grouped into an update packet and forwarded to the directory. These special instructions are ignored if the line is in an exclusive state where there are no other shared copies of the line and, therefore, no updates are required.

The scheme places a large burden on the compiler. First the compiler must identify writes to shared data and issue them with a special write-no-update instruction. Next, the compiler must issue the special write-line instruction after all the writes to a give line have been issued. Note that a write line instruction is required for each memory line with pending updates. If the application is data independent, the compiler may simply insert these instructions at the proper location in the code. Alternatively, the compiler may insert code around each write to

[1]Each cache line already has a set of valid bits used to allow the protocols to write words into the line while the line is in a pending state awaiting a miss reply. These bits can also be used as the update-pending bits when the cache line is in a shared state.

generate the necessary write-line instructions at run time. In this case, the code must maintain the line address of the last shared write. When the line address of the writes change, a write-line instruction is issued for the last line address. This run-time implementation would add several instructions per write and would require the use of a register to maintain the last line address.

Figure 2 shows an example loop and illustrates how the software-based schemes may be implemented. The original code is a simple loop that computes a set of values for the shared vector a. For the compile-time grouping, the loop is unrolled into segments that are of length equal to the memory line size. The writes are issued using the special $WriteNoUpdate$ instruction that writes the values into the cache but does not initiate an update. Then, after the full line of data has been produced, the $WriteLine$ instruction is issued for the line to initiate the update. This is repeated for each memory line. For the run-time grouping, each write is again issued with the $WriteNoUpdate$ instruction, but the line address of the write must be compared with the line address of the last write. If they are not equal, a $WriteLine$ instruction is issued for the last line, and the last line pointer is set to the new line address. A final $WriteLine$ instruction must be issued to initiate the update for the last line's data.

As illustrated from the above discussion, the software-based scheme may be difficult to implement efficiently, but the scheme is able to achieve optimal grouping efficiency by grouping all writes to a given cache line. We will use the compile-time software-based scheme to compute the optimal grouping efficiency for each application studied and show that the hardware-based scheme presented in the next section can approach this efficiency.

```
shared float a[30], b[30], c[30];

for (i = 0; i < 31; i++)
  a[i] = b[i] * c[i];
```

a) Original Code

```
shared float a[30], b[30], c[30];

/* Unroll loop into line size
   segments */
for (i = 0; i < 16; i++) {
  /* Write data */
  temp = b[i] * c[i];
  WriteNoUpdate(a[i],temp);
}
/* Issue update for line */
WriteLine(&(a[0]));

for (i = 16; i< 31; i++) {
  /* Write Data */
  temp = b[i] * c[i];
  WriteNoUpdate(a[i],temp);
}
/* Issue update for line */
WriteLine(&(a[16]));
```

b) Compile-Time grouping

(Line size = 16 words)

```
shared float a[30], b[30], c[30];

/* Set line address */
LastLine = &a & 0xfffffff0
for (i = 0; i < 31; i++) {
  temp = b[i] * c[i];
  /* Write data */
  WriteNoUpdate(a[i],temp);
  /* Write to new line ? */
  if (&(a[i]) & 0xfffffff0) != LastLine) {
    /* Issue update for last line */
    WriteLine(LastLine);
    /* Set line address */
    LastLine = &(a[i]) & 0xffffff0;
  }
}
/* Issue update for last line */
WriteLine(LastLine);
```

c) Run-Time grouping

Figure 2: Software-Based Write Grouping

2.2 Hardware-based write grouping

In a hardware-based write grouping scheme, writes are grouped as they are written into the write buffer as shown in figure 3. The scheme requires an additional grouping bit for each word in the write buffer. As the processor issues the writes, the line address of the write is compared with the line address of the last write inserted into the write buffer. If the write is to the same line, the additional grouping bit is set to 1, indicating that the write should be grouped with the last write inserted into the write buffer. When the cache processes the writes from the write buffer, it consumes all writes in a write group, and it is able to issue the write update, if needed, in a larger, more efficient packet.

A write group is delayed in the write buffer until either

1. the write buffer fills,

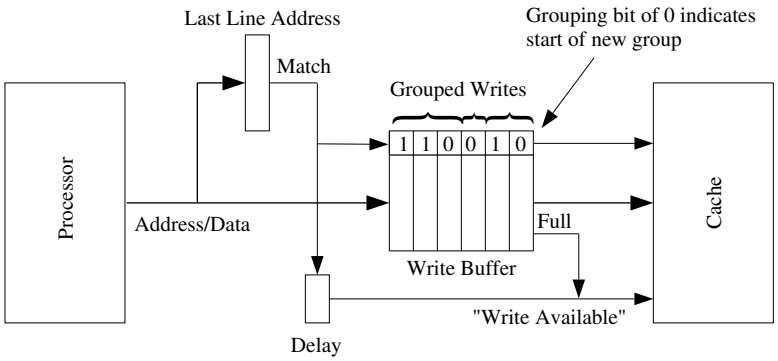

Figure 3: Hardware-Based Write Grouping at the Write Buffer

2. a write to a new line is received, or

3. the delay timer expires.

The first two conditions are obvious. Once the write buffer fills or a write to a new line is received, no more writes can be grouped with the current write group. The third condition is used to increase the grouping window for each write group. This delay prevents the cache from consuming writes as fast as the processor is able to issue them; the delay sets the minimal grouping window for each write group. The base case sets the delay count at five cycles.

Figures 4 shows an example of hardware-based write grouping. In the figure, the processor issues 7 writes labeled Wi where i is the address of the write. In the example, the line size is 2 words and the write grouping delay is 3 cycles. Writes W0 and W1 are grouped into a two word write group because they are writes to the same line. Write W5 cannot be grouped with the W0,1 group since it is a write to another line. This writes triggers the sending of the first write group, W0,1, to the cache. No writes follow W5 and the delay timeout expires resulting in W5 being sent to the cache. Writes W2 and W3 are grouped, but write W9 is a write to another line so it triggers the sending of write group W2,3 to the cache. Finally, write W0 is issued, which causes write group W9 to be sent to the cache. No writes follow W0 so the delay timeout expires

and the write is sent to the cache. Notice that the delay timeout only affects the last write in a write burst.

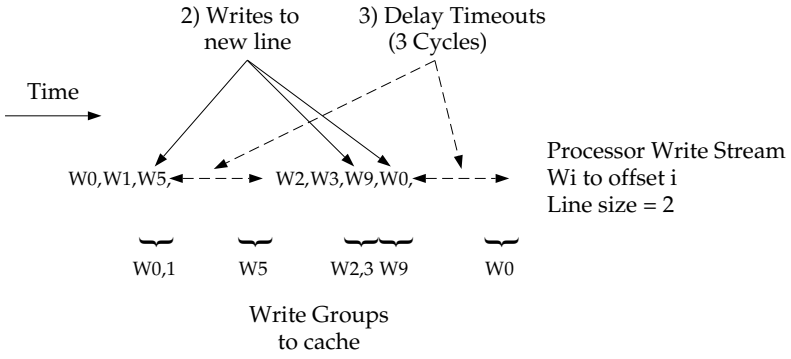

Figure 4: Hardware-Based Write Grouping Example

The grouping hardware adds an extra pipeline stage to the write buffer. Writes now take a minimum of two cycles to reach the cache, but the write buffer may still accept writes at a rate of one per cycle.

2.3 Grouped update network packet

To send the grouped update efficiently, a bit vector is used to indicate the line offsets of the grouped writes. In this vector, bit b_i would be 1 if the data words in the packet included an update for offset i. This requires that the data following the packet header be arranged in ascending offset order, as shown in figure 5. To achieve this ordering for the software-based scheme, the cache simply reads the update-pending words in the proper offset order. For the hardware-based scheme, the cache must first read the write group from the write buffer. These writes may be in any offset order and may contain multiple writes to the same offset. Next, the words are written into the cache line using the valid bits, as in the software-based scheme, to indicate which words of the cache line have been modified. Finally, the words are read from the

cache line in the proper offset order, and the update packet is generated. The valid bits are then cleared.

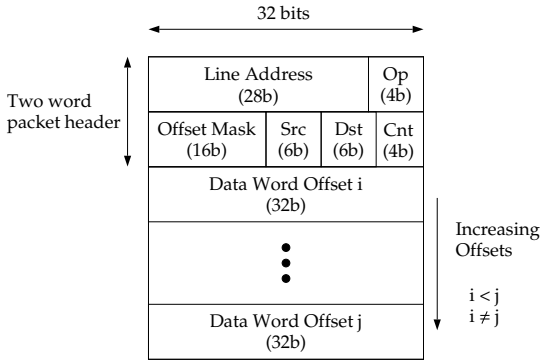

Figure 5: Grouped Update Packet

3 Simulation environment

3.1 Simulator

The Care/Simple simulation environment [6] was used to simulate the set of scientific applications described in section 3.4. The simulations were application driven rather than trace driven.

3.2 Simulated architecture

The simulated architecture consists of 64 nodes arranged in an 8 by 8 mesh, as shown in figure 6. Each node consists of a processor/memory element (PME) connected to its four nearest neighbors through a set of network queues.

A PME is shown in figure 6. The processor is a 32 bit, 100 MHz superscalar processor that is assumed to be load/store limited, and the cache is a fully associative cache with infinite size. An infinitely-size cache is used to separate the effects of a limited cache size and the actions required by the cache coherence protocols. 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

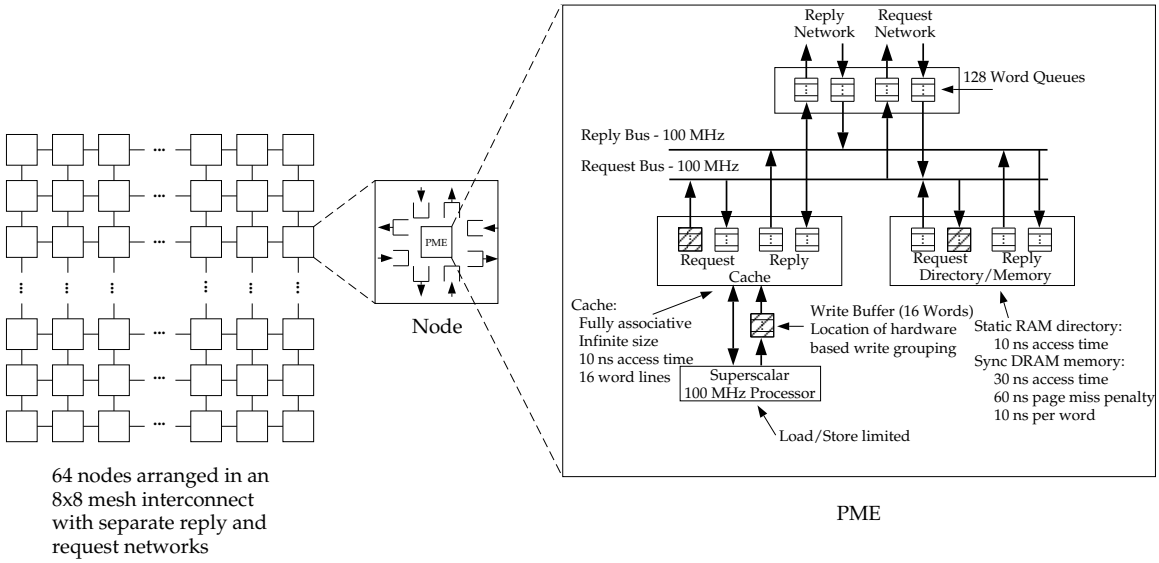

Figure 6: Simulated Architecture

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.

To avoid deadlock, each node is connected to both a reply and a request network. Request-reply deadlock is avoided by guaranteeing that the replies will eventually be consumed at their destination. Request-request and reply-reply deadlock require a timeout to break the deadlock [8]. The network is order preserving with static, wormhole routing [1] and multicast. Multicast is only used by the centralized directory protocols when there are multiple caches that must be updated or invalidated.

3.3 Update-based cache coherence protocols

The update-based cache coherence protocols examined in this work include the centralized-directory update-based protocol (CD-UP) and the singly-linked distributed directory update-based protocol (DD-UP). Both are described in our earlier work [4]. The centralized directory

protocols assume a full mapped directory [7].

3.4 Scientific applications

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)[2], sparse Cholesky factorization (SPCF), and LU decomposition.

The amount of write grouping possible for each application is determined by the application's average line utilization. The line utilization is the percentage of each memory line that is modified by the data 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. Table 1 shows the average line utilization for the applications studied. Applications with low line utilization will have little opportunity for write grouping, but in these cases the single word updates will not result in excessive network traffic, as illustrated in figure 1.

Applications	MF	PDE	SPCF	LU	3DFFT
Data Set (words)	1000x1000	32x32	1138x1138	64x64	8x8x16
Line Utilization %	93.0	50.0	11.2	59.0	50.0

Table 1: Application characteristics

The applications studied use two different types of data synchronization. The first type is the typical coarse-grain (Block) synchronization using a release consistency memory model [2].

[2]The MF application has two distinct phases of operation [5]. 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 examines only the first phase of this computation.

In this case, a flag or barrier is used to synchronize the production and consumption of data, as described in our earlier work [4]. The second type of synchronization is a fine-grain (Word) synchronization. In this case, the synchronization information is combined with the data word and, therefore, all explicit synchronization events are eliminated. This is also described in our previous work [4, 3].

4 Write grouping performance

4.1 Grouping efficiency

Table 2 shows the average write group size for the two update-based cache coherence protocols using both software-based (SW) and hardware-based (HW) write grouping schemes. The average write group size is given for both coarse-grain (Block) and fine-grain (Word) data synchronization for the five applications under study. Block synchronization requires explicit synchronization such as flags or barriers, and these variables are allocated on separate memory lines to avoid false sharing. Therefore, synchronization writes cannot be grouped with any other writes, and this results in a write group of one word. This reduces the average write group size. In contrast, word synchronization combines the synchronization information with the data word. In this case, the average write group sizes are slightly larger than in the block synchronization case; the difference indicates the relative frequency of synchronization writes to data writes.

With ideal knowledge of the applications' write patterns, the software-based scheme is able to group all writes destined for the same line into optimal write groups. The actual size of the optimal write group is determined by each application's average line utilization, which was given in table 1. The higher the line utilization, the larger the optimal write group. The maximum write group size is 16 words, a full memory line.

Applications	Sync	CD-UP-SW	CD-UP-HW	DD-UP-SW	DD-UP-HW
MF	Block	13.2	13.2	13.2	13.2
	Word	14.9	14.9	14.9	14.9
PDE	Block	4.5	4.4	4.5	4.4
	Word	8.0	7.9	8.0	7.9
SPCF	Block	1.4	1.4	1.4	1.4
	Word	1.9	1.9	1.9	1.9
3DFFT	Block	3.0	1.8	3.0	1.9
	Word	3.5	2.2	3.5	2.3
LU	Block	9.5	9.5	9.5	9.5
	Word	9.5	9.5	9.5	9.5

Table 2: Grouping Efficiency - Words Per Update Packet

As indicated in table 2, the hardware-based grouping scheme is almost able to achieve the same write grouping efficiency as the software-based grouping scheme for both update-based protocols. The one exception is the 3DFFT application. In this application, the average shared data write rate is too low to be captured by the hardware grouping scheme using a five cycle grouping window. In section 5.1, we will examine the effect of varying the length of the write grouping window.

4.2 Execution time

Figure 7 shows the relative execution times for the applications using the two grouping schemes compared to the non-grouping case for each update-based protocol. For all applications, except 3DFFT, the hardware-based scheme resulted in about the same or faster execution time compared to the software-based combining scheme for both update-based protocols. The hardware-based scheme did not perform as well for the 3DFFT since the write grouping was sub-optimal compared to the software-based scheme, as described in section 4.1,

The software-based grouping scheme resulted in a longer execution time than the hardware-

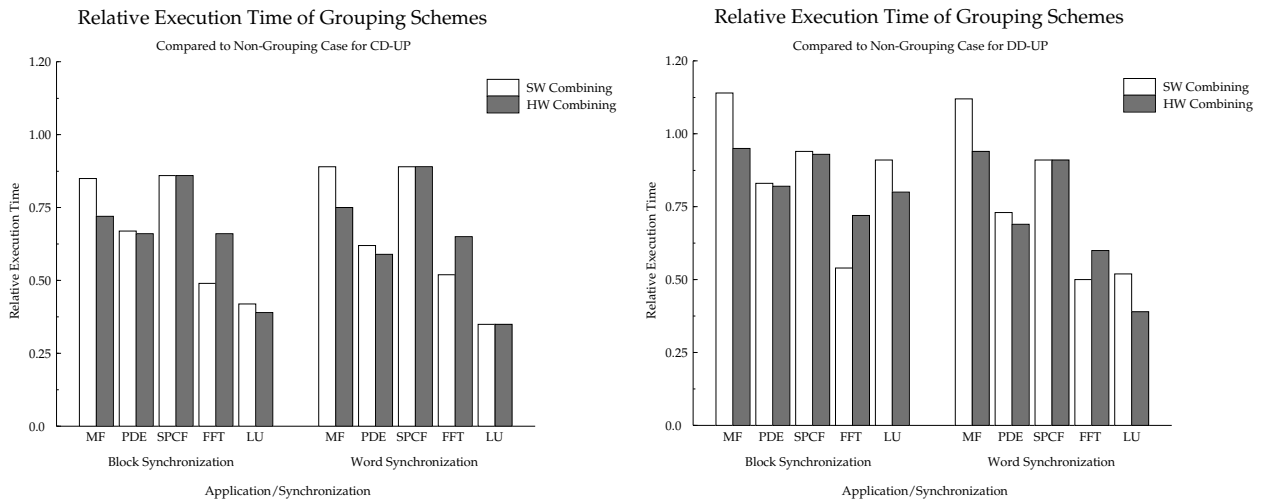

Figure 7: Relative Execution Time of Grouping Schemes compared to Non-Grouping Case

based scheme for the MF application for both protocols and the LU application for the DD-UP protocols. In the MF application, the data was not as eagerly shared as in the other applications. Most of the writes were to cache lines in the exclusive state, which did not require updates. In this case, the extra cycles introduced by the software-based scheme outweighed any performance gain from the relatively few write groupings. For the LU application, the observed write miss latency was increased because the delay to issue the writes, introduced by the software-based scheme, reduced the amount of the miss latency which could be hidden behind useful work. This resulted in a slightly longer execution time for the DD-UP protocol.

The performance improvement from write grouping is larger for the CD-UP protocol than for the DD-UP protocol. The write grouping improves the performance of the CD-UP protocol by reducing both the network traffic and the memory update overhead. The DD-UP protocol does not update memory on each write and, therefore, only benefits from the reduction in network traffic.

5 Variations in hardware-based grouping

In this section, we examine the performance of the hardware-based write grouping scheme when the grouping delay is varied and the location of the write grouping buffer is moved to the output of the cache or the input of the memory/directory.

5.1 Write grouping delay

Table 3 shows the words per write group as the write grouping delay is varied from no delay to 20 cycles of delay. With no delay, the write grouping scheme is able to group only a small fraction of the writes. This grouping only occurs when the writes are delayed in the write buffer because the cache is busy responding to a read or network request.

As the write grouping delay is increased, the grouping scheme captures many more writes. As noted in section 4.1, a five cycle delay is sufficient to group almost all writes that can be grouped. The one exception is the 3DFFT application, but as the write grouping delay is increased for this application, more writes are grouped. A ten cycle delay results in an almost optimal write grouping efficiency for all applications.

However, increasing the write grouping delay is not without its cost since the increase in delay also affects the total execution of the applications as shown in figure 8. The figure shows the relative execution times as the write grouping delay is varied from 0 cycles to 20 cycles. In the figure, the execution times of the applications improved with increasing delay until the delay was large enough to group a significant portion of the writes. Increasing the delay beyond this point only affected the last write of the data block. Intermediate writes are grouped as new writes are inserted into the write buffer. If these writes are efficiently grouped, then the write rate is faster than the grouping delay timeouts, and it is only the last write in the group that is delayed by

		CD-UP-HW					DD-UP-HW				
Applications	Sync	0	5	10	15	20	0	5	10	15	20
MF	Block	4.6	13.2	13.2	13.2	13.2	4.8	13.2	13.2	13.2	13.2
	Word	4.8	14.9	14.9	14.9	14.9	5.0	14.9	14.9	14.9	14.9
PDE	Block	2.9	4.4	4.5	4.5	4.5	2.8	4.4	4.5	4.5	4.5
	Word	6.8	7.9	8.0	8.0	8.0	6.4	7.9	8.0	8.0	8.0
SPCF	Block	1.2	1.4	1.4	1.4	1.4	1.2	1.4	1.4	1.4	1.4
	Word	1.2	1.9	1.9	1.9	1.9	1.2	1.9	1.9	1.9	1.9
3DFFT	Block	1.2	1.8	2.7	2.9	3.0	1.2	1.9	2.7	2.9	3.0
	Word	1.5	2.2	3.2	3.5	3.5	1.6	2.4	3.2	3.4	3.5
LU	Block	3.5	9.5	9.5	9.5	9.5	3.1	9.5	9.5	9.5	9.5
	Word	3.3	9.5	9.5	9.5	9.5	4.1	9.5	9.5	9.5	9.5

Table 3: HW Grouping Delay (Cycle) - Words Per Write Group

the full delay timeout period, as described in section 2.2.

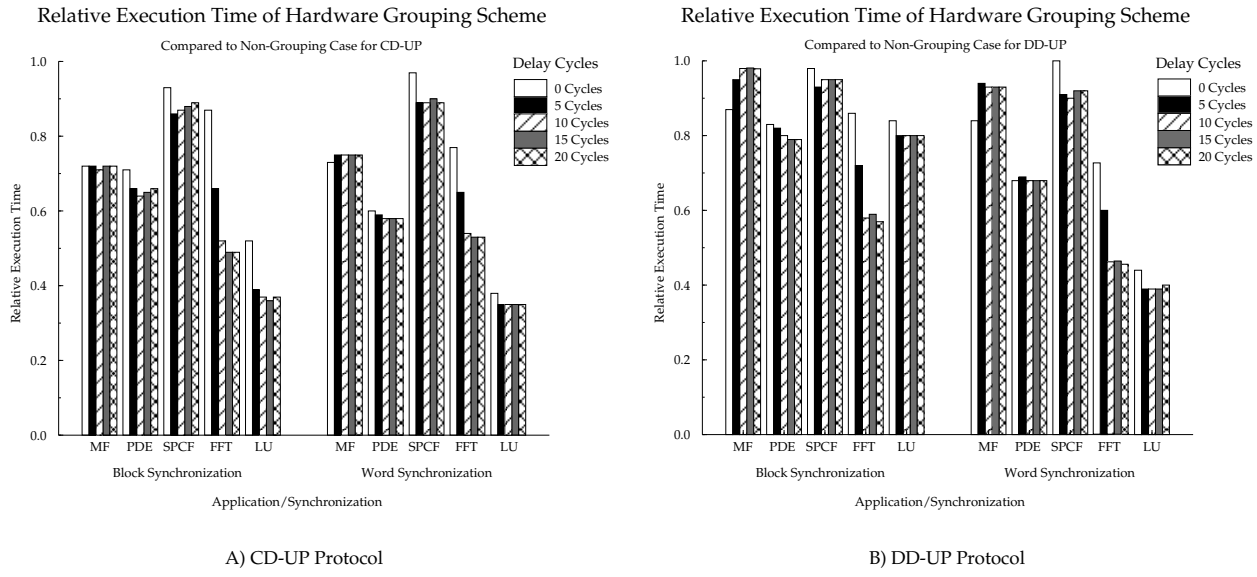

Figure 8: Relative Execution Time of Grouping Delay for Hardware-Based Grouping

The relative impact of the delay is largest for the block synchronization case. In this case, the processor must stall until all writes have been performed (all updates performed). The performance loss depends on the block size and the write completion latency. If the data block is large, the delay will be small compared to the total write time. But in applications such as SPCF

where the block size is small, the delay becomes a noticeable fraction of the total write time. The effect is clearly illustrated in figure 8 for this application using block synchronization, especially for the CD-UP protocol. For the DD-UP protocol, the write completion latency is significantly longer than in the CD-UP protocol, and it increases with the number of caches that are updated on each write. The write grouping delay is small compared to this write completion latency, and therefore, the impact of the increased write grouping delay is negligible for the DD-UP protocol.

For word synchronization, the added delay for the last word written has little impact on the total execution time of the application, as shown in figures 8. When an application used word synchronization, the consumption time of earlier data words overlapped the latency of subsequent data updates.

5.2 Location of the grouping buffer

Write grouping could be done at any request buffer in the system. Two other candidates for write grouping are the request output buffer of the cache (CO) or the request input buffer of the memory (MI). Both buffers are shaded in figure 6. Table 4 shows the write group sizes when write grouping is introduced at these buffers. Grouping writes at either of these buffers is not as efficient as grouping at the write buffer (WB), and increasing the grouping delay beyond 5 cycles had little impact on the grouping efficiency at these other buffers. The write buffer grouping efficiency was always the best.

Grouping at these other locations does not work well since other network packets tend to disrupt the flow of writes from the caches to the directory/memory. The grouping scheme works by attempting to group a new write packet with the last packet inserted into the buffer. The complexity of the buffers could be increased to allow writes to be grouped with any appropriate write

		CD-UP-HW			DD-UP-HW		
Applications	Sync	WB	CO	MI	WB	CO	MI
MF	Block	13.2	7.0	6.6	13.2	7.1	7.0
	Word	14.9	6.8	6.2	14.9	7.3	7.3
PDE	Block	4.4	3.5	1.1	4.4	3.3	1.3
	Word	7.9	5.3	1.1	7.9	1.2	1.2
SPCF	Block	1.4	1.2	1.2	1.4	1.2	1.2
	Word	1.9	1.4	1.2	1.9	1.3	1.3
3DFFT	Block	1.8	1.8	1.1	1.9	1.8	1.1
	Word	2.2	2.2	1.2	2.4	2.3	1.2
LU	Block	9.5	8.2	1.0	9.5	7.8	1.1
	Word	9.5	7.5	1.0	9.5	2.3	2.3

Table 4: Grouping Location - Words Per Write Group

packet currently in the buffer. This might improve grouping, but, as discussed in section 4.1, the write grouping efficiency at the write buffer is almost optimal. The additional cost of increasing the buffer complexity would result in no performance gain compared to the inexpensive write buffer grouping.

6 Conclusions

In this paper, we have explored the performance gains from write grouping in update-based cache coherent systems. Two types of grouping schemes were discussed: a software-based and a hardware-based scheme. The software-based scheme required compiler and programmer support, but the scheme was able to optimally group writes. The hardware-based scheme was shown to almost achieve this optimal write grouping efficiency if a grouping delay window was introduced, and the scheme required only a small amount of hardware support and little, if any, software support.

The write-buffer write grouping improved the performance of the CD-UP protocol the most.

The improvements in execution time ranged from 13% to over 50%. The gains came from two sources. First, the write grouping significantly decreased network traffic resulting from the write updates. Second, the larger write groups decreased the average memory update latency per word as the memory access time could be amortized across more data words. The DD-UP protocol also benefited from write grouping. The improvement in execution time ranged from only a few percent to over 50%. Since the DD-UP protocol does not update memory, the gain came from a reduction in write update network traffic.

Increasing the write grouping delay beyond the base 5 cycles had little impact on the efficiency of the write grouping. The one exception was the 3DFFT application in which a 10 cycle delay was required to achieve near optimal write grouping efficiency. Also, increasing the delay beyond 5 cycles had a minor affect on the total execution time of the application. Moving the write grouping to the cache output buffer or the memory input buffer resulted in poor write grouping and, therefore, little improvement in total execution time compared to the non-grouping case.

Overall, write grouping is essential for improving the performance of update-based cache coherent system with a general interconnect, and a write grouping delay window of 5 cycles was sufficient to achieve efficient write grouping in most applications without adversely affecting non-grouped writes.

References

[1] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[2] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip. Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multi-

processors. 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. The impact of cache coherence protocols on systems using fine-grain data synchronization. Technical Report CSL-TR-94-611, 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] 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.

[6] 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.

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

[8] Manu Thapar. Cache coherence for scalable shared memory multiprocessors. Technical Report CSL-TR-92-522, Computer Systems Laboratory, Stanford University, May 1992.