Hardware Assist for Data Merging for Shared Memory Multiprocessors

Alan H. Karp
Rajiv Gupta
Hewlett-Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304
{karp,gupta}@hpl.hp.com

March 13, 1994

Abstract

We describe hardware that improves on the performance of Data Merging, an efficient software cache consistency mechanism for shared memory multiprocessors. The hardware support consists of an extra bit attached to each datum in a cache line. These bits are used to control the merge operation rather than a bit mask held in the global memory. The bits can also be used to reduce the amount of network traffic by sending only modified words to memory.

We also propose some extensions to the basic model. We can reduce the cache area overhead of the extra bits by keeping a bit per word instead of a bit per byte. Subword operations are supported with the same mechanism used in the original data merging paper. We can also keep separate bit masks for readers and writers of data which allows us to report all data races, an important feature in debugging shared memory, parallel programs. Other improvements are also described.

1 Introduction

A distributed shared memory computer is one that uses software and/or hardware to make a physically distributed memory look like a physically shared memory to the programmer. Sequential consistency[6] may be the single most significant inhibitor in achieving scalability in distributed shared memory machines. The recognition of this fact has led to a number of delayed consistency schemes[1, 2, 3, 4]

One recent entry into the field is Data Merging[5]. Karp and Sarkar designed Data Merging to satisfy a number of goals.

1. The processor hardware need not have any knowledge that shared data is cached in multiple local memories.
2. The programmer and the compiler need not have any knowledge that shared data is cached in multiple local memories.
3. The solution provides efficient support for false sharing of data blocks.
4. The solution can exploit the slackness revealed by delayed consistency models.
5. The solution is effective for all data block size granularities.
6. The solution is simple to implement in hardware or software.
7. The solution can use off-the-shelf processors.

KS Data Merging is an efficient software cache consistency mechanism for shared memory multiprocessors that supports multiple writers and works for cache lines of any size. The authors claim to have met all the above stated goals except the last one; they needed a small change to existing cache designs.

We find KS Data Merging interesting because it does extra work only when a block of data is *actually* shared over the time interval it was held by a processor;

other schemes do extra work whenever the data *might* be shared during the interval. There are many applications, such as transaction processing, where all data is potentially shared but actual sharing is exceedingly rare.

Unfortunately, KS Data Merging has some shortcomings. The KS merging operations are too complicated for a simple state machine memory controller to handle. Also, there are times when memory requests to a data block must be suspended which forced Karp and Sarkar to add a time-out to allow an invalidate command to be broadcast. Another potential problem is that Data Merging generates unnecessary memory traffic because it sends the entire data block to the global memory while it needs to send only the modified data.

In this paper we describe some changes to the cache that allow us to avoid these difficulties most of the time. Some further modifications allow us to detect all data races, both write-write and read-write. We adopt the same multiprocessor memory model as did Karp and Sarkar which is illustrated in Figure 1, namely compute nodes with local memories connected to global memory servers. This dichotomy is not necessary; the compute nodes can also be used as global memory servers. We find that this model simplifies the discussion, though.

Section 2 reviews the Karp-Sarkar Data Merging scheme. Next, in Section 3 we show how a simple change to the processor cache can lead to a substantial performance improvement. A further extension presented in Section 4 shows how our proposal can be modified to detect data races.

2 Data Merging Summary

The basic KS scheme was well explained in the original Data Merging paper[5]. In this section we paraphrase their description and point out some weaknesses in their proposal.

Global memory is divided into data blocks of arbitrary size. The global memory controller maintains a counter, a bitmask, and a suspend bit for each

data block. The counter is used to keep track of how many copies of the data block are currently held by processors. The bitmask identifies the elements that have been modified in the data block, and is used to control subsequent merges into the data block. The suspend bit is explained below.

When a processor requesta a data block, the global memory unit sends the data and increments the corresponding counter. When the processor replaces a clean copy, either because of a synchronization or due to LRU replacement, the global memory controller decrements the counter. When the processor flushes a dirty copy, the global memory controller merges the changed words with the current version of the data block using the bit mask. The bit mask is updated to identify which data elements have been modified. When the counter reaches zero, the bitmask is cleared.

For simplicity and scalability, no attempt is made to store the set of processor ids that have requested a copy of the data block; only a count is kept. It is possible for a processor to fetch a data block, modify it, then flush it. If, before reaching a synchronization point, the same processor requests the data block and changes a word it had modified before, this change will not be recorded when the block is flushed to global memory. Hence, processors must be suspended until the merge is completed, *i.e.*, the count has gone to zero. The suspend bit is used to suspend memory requests until the bitmask is reinitialized.

The key functional advantage of Data Merging solution is that it imposes only two minor requirements on the processor hardware or software: a) the processor must also notify global memory when a clean cache line is replaced, b) all synchronizations must be performed through explicit synchronization routines (so that they can also ensure that the appropriate shared data in cache is flushed or replaced). Requirement a) is a very minor extension to processor hardware. Requirement b) is also needed by other delayed consistency mechanisms. The suspend bit can also be used to do locking in memory which can greatly reduce the need for spin locks.

In practice, data merging has the great advantage that the processors do local computation exactly as they would in a serial implementation. Between synchronization points cache lines are fetched, modified, and flushed exactly as in a uniprocessor run. Even at a synchronization point, processors do no computation; they need only flush some of their shared data. The only extra computational work done is at the global memory unit, and then only when a modified data block was *actually* shared during the interval a processor held a copy. This latter fact is particularly important when the processors also act as global memory units. The number of CPU cycles used to implement the data merging protocol will be greatly reduced over that of other schemes that use processor cycles whenever the data *might* be shared.

Because KS data merging approach requires such small changes to the processing element hardware and software, it can be efficiently implemented on tightly-coupled multiprocessors as well as on networked workstations. In a tightly-coupled system, the appropriate data block granularity is the cache line size using the memory controller to implement the protocol. For a network of workstations, the appropriate data block granularity is the virtual page size. Here, software can be made to handle the protocol by modifying the paging mechanism.

There are two potential problems with KS Data Merging. The first is the need to suspend memory requests which makes it impossible to bound the delay of a memory request. Karp and Sarkar had to add a time out to their basic scheme because of the possibility of deadlock. When a memory request has been in the memory queue beyond the time-out interval, an invalidation request for this data block is broadcast to all the processor. Karp and Sarkar claim that these broadcasts will be rare but present no application experiences to justify this belief.

The second problem is that the assumption that the processor will send the entire data block to the global memory. This procedure won't be too much of a

problem in a tightly coupled system where the data blocks are relatively short cache lines. It may be more of a problem in a loosely coupled system where the blocks are 2 KByte or 4 KByte pages. While a well-tuned program may well modify most or all of the words on a page between synchronizations, many programs may not. The increased memory traffic may be significant in some settings.

3 Hardware Assist

A very simple modification to the cache structure enables us to avoid both problems inherent in KS Data Merging. Each byte in each cache line has a modified bit associated with it. When the cache line is loaded, the associated bits are cleared. Any store into the cache line sets the affected bits. In principle, these bits should be set only when the data is actually changed, but in practice, they will probably be set on any write, even one that doesn't change the contents.

There are two ways these bits can be used when the cache line is replaced. Either the entire cache line along with the bit mask can be sent to memory, or just the bit mask and modified bytes. In a tightly coupled system, the former approach would probably be used. The memory controller would use the mask to control which elements replace those in the copy in global memory. The latter scheme might be used in a loosely coupled system in which the local memory of the processor is treated as an additional layer of caching. In this case, the local memory unit could save the bit mask for use when a data block is later moved to the global memory.

With this approach, the global memory needs only the ability to use the bit mask to control which elements get replaced. There is no counter, no bit mask in memory, no suspend bit. Unfortunately, maintaining a bit per byte in the cache is expensive in terms of area and transmitting a bit per byte to the global

memory is expensive in terms of communications bandwidth.

We can reduce both of these costs from 12.5% of the cost of transmitting just the data to only 3% if we store a bit per word. Since most programs modify only words (or even double words), these programs will always modify all the bytes in a word at the same time. Providing a bit per word will suffice for these programs. On the other hand, most machines available today allow the programmer to modify bytes.

In order to support those programs that involve sub-word operations we propose using a hybrid approach. Each cache line will be modified further to include a bit that says whether any sub-word modifications were done. This bit can be set as part of the store-byte or store-halfword operations. Most of the time this bit will indicate that no sub-word operations were done, and the memory unit will act as described for the bit per byte case; the mask will be used to control which elements are to be replaced.

If the sub-word flag is set, we use KS Data Merging. In other words, when such a cache line is flushed, the global memory unit will allocate a bit per byte to act as a mask and a suspend bit. Since the global memory does not know which data blocks may be modified by sub-word operations, it will have to have a counter attached to each data block. Most of the time this counter will only be incremented and decremented. Once the first sub-word operation on the data block has been detected, the counter will be used to control suspending of requests for this block that are received before the merge is complete.

This hybrid scheme has the same problems as KS Data Merging, but on a much reduced scale. Unbounded delays can occur, including deadlock, and we must send entire words from the cache to the memory even if only one byte in it was modified. However, sub-word operations are much less frequent and can be avoided entirely. Further, the maximum amount of extra data transmitted is only 3 bytes in this scheme versus nearly a full cache line in KS Data Merging.

4 Data Race Detection

Once we have accepted changes to the cache to improve the performance of Data Merging, there are more things we can do to assist the programmer. In this Section we describe a modification to our proposal that enables us to report all data races.

Debugging shared memory programs is extremely difficult, in part because it is so easy to inadvertantly modify a shared word without proper synchronization. Techniques such as instant replay are difficult to implement efficiently because the granularity of the shared objects, words, is so small. Further, there are two types of races – write-write in which two or more tasks modify the same word, and read-write in which one task reads a word written by another.

We propose modifying the cache to have two bits per data element, either byte or word. Both bits will be cleared when the cache line is loaded. One will be set, as before, when the datum is written; one will be set when the datum is read. When the cache line is flushed to the global memory, these two masks can be used to detect races using the following scheme.

As in KS Data Merging, the global memory unit will keep a counter for each data block. In addition, it will keep a write-mask and a read-mask which are initalized to zero. When a copy is sent to a processor, the counter is incremented. When a copy is returned, the data is simply stored if the counter is unity and the read and write masks are both in their initial state. Otherwise, the read and write masks from the processor will be compared with the current copies in the global memory. If there are no set write mask bits occur in the same position as set read or write mask in the global memory copy, the masks are ORed and stored in the global memory copy. If there are set bits in the same positions in the two versions, a data race has been detected and can be reported to the programmer.

Unfortunately, we must suspend requests for a data block once a merge has started even though we know which words have been modified for the same

reason that KS Data Merging must. If we don't suspend these requests, we won't know if a conflict is a true race or caused by a processor fetching a line, modifying it, flushing it, and refetching it. Since this situation is expected to occur rarely, some users may be willing to accept the occasional false detection rather than allow requests to be suspended. In the next Section, we show another way to avoid suspending memory requests.

There is one problem with the above scheme that requires programmer or compiler assistance; we don't know when to reset the race detection masks to their initial state. The only really safe place is at a global barrier. However, there are many situations where a subset of the processors coordinate their activities independently of the remaining processors. Our hardware has no way of knowing when *enough* processors have synchronized to allow us set the masks to their initial state. Hence, we ask the programmer to call library routines around a block of code suspected of having a data race. As long as this block has no synchronization points inside it, we will report only data races that occur during the run. If the block includes a synchronization of some sort, we may report some false data races.

5 Extensions

Our proposed modifications to the cache remove the problems with KS Data Merging if we are willing to provide two bits per byte in the cache or keep two bits per word and forego subword operations. If we are not, there will be times when memory requests will be suspended, and we will sometimes send more than the minimum amount of data to the global memory unit. Suspended memory requests might also be a problem if we are attempting to detect data races.

We believe that memory requests will be suspended rarely, even with KS Data Merging. However, having unbounded delays and potential deadlocks is

intellectually unsatisfying. We propose a software change to eliminate the last vestiges of suspended requests by using *mini-directories*.

When a memory request arrives at the global memory that can not be satisfied because a merge is in progress that requires suspending requests, we record the id of the processor requesting the data block and the current contents of the data block including the read and write bit masks. Then, we can safely deliver the current copy to the requesting processor. Once this directory has been allocated, all subsequent cache lines returned to the global memory are checked to see if the version in memory or the mini-directory should be used to control the merge or race detection. The race detection masks will be used under the direction of user generated library routines as described in Section 4. We expect the mini-directories to be allocated rarely and to have few entries since it is unlikely that a processor will flush a line and reaccess it between synchronization points.

Another extension involves bit operations. It is possible to write a program that manipulates bits that can be parallelized to run correctly on a sequentially consistent shared memory machine. None of the proposed delayed consistancy schemes will give the correct answer if two processors modify different bits in the same byte.

We propose to handle this situation by modifying the cache and the registers. In addition to having a sub-word store flag, each cache line will have a bit-modified flag. The registers will have a bit appended to them that is cleared on a load and set whenever an operation could affect less than a byte of data. For example, shifts by amounts not a multiple of 8 bits, AND and OR, *etc.*

When a register having its bit-modified flag set is stored, the corresponding bit is set in the cache line. When this cache line is flushed to global memory, a bit-wise merge is done. Since we expect that the memory controller can only load and store bytes, we replace the current contents of memory with

```
B = (B AND NOT M) OR (C AND M),
```

where `M` is the current bit mask stored in the global memory, `C` is the data coming from the processor, and `B` is the contents of memory. We also construct the new mask by ORing in a bit mask of the newly modified bits. Clearly, bit-wise merges are expensive and will probably only be used rarely and for compatibility with other sytsems.

6 Conclusions

Delayed consistency memory systems may make it feasible to build scalable shared memory systems. However, great care will be needed to avoid a new class of bottlenecks. In particular, there are commercially important applications, such as transaction processing, that access large amounts of shared data, but any record is only rarely shared. A parallel machine running this application should be able to scale up to a large number of processors if the work done avoiding the rare conflicts can be reduced.

KS Data Merging is the first delayed consistency scheme that does extra work only when data is actually shared; all others do extra work whenever the data might be shared. For example, Munin[2] delivers a read-only copy of a page to the processor making the memory request. On the first write, a clone of the page is made. When a synchronization point is reached, the clone is compared to the current version of the page. The bytes that differ can be sent back to the global memory. This work consumes some 25% of the processor cycles on a moderately well-structured algorithm, a Fast Fourier Transform. KS Data Merging does no extra work unless another processor is holding a copy.

Another approach is taken in the Stanford DASH machine which keeps directories showing which data is held by which processor, even when implementing release consistency[4]. In this case, the complexity of the machine is increased. Because the directories are in the critical path of loads and stores that miss in

the cache, either the cycle time of the machine is larger or the latency of a cache miss is larger than without the directory.

KS Data Merging may not be scalable to large numbers of processors if requests for data frequently occur while a merge is in progress. Furthermore, KS Data Merging may saturate the communications network unnecessarily if only a small percentage of the data in a block is modified. A more serious drawback is the need for a rather sophisticated memory controller; it is unlikely that even the basic KS protocol can be implemented with the state machines used in today's workstations.

We believe that the hardware changes we propose present a reasonable trade-off between cost and benefit. The extra bit per word in the cache increases the chip area by a negligible amount; the extra path lengths through the cache affect stores, not loads, and so have a minimal impact on cycle time. The benefit is that we can use almost exactly the same memory controller as on a standard workstation and still implement the merging. We also eliminate the problem of needing to suspend memory requests, and we can reduce the amount of memory traffic by sending only modified data if that will improve system performance.

Acknowledgements

We would like to thank Dennis Brzezinski for helping us to understand the various memory consistency models. David Worley provided encouragement and support.

References

[1] Sarita Adve and Mark Hill. Weak Ordering — A New Definition. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 1–14, May 1990.

[2] J. B. Carter, J. K. Bennet, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, May 1991.

[3] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared Memory Multiprocessors. *IEEE Transactions on Software Eng.*, 16(6):660–674, June 1990.

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

[5] Alan H. Karp and Vivek Sarkar. Data Merging for Shared Memory Multiprocessors. In *Hawaii International Conference on System Science 26*, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[6] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

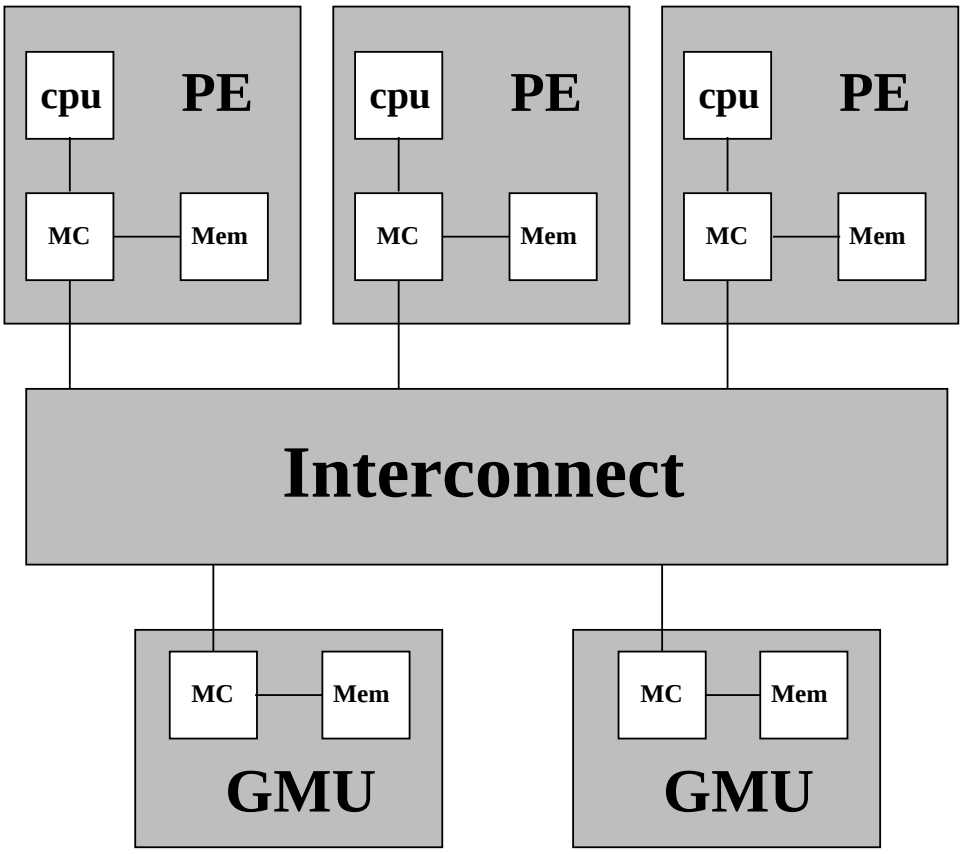

Figure 1: Multiprocessor memory model. Each processing element (PE) has a cpu, a memory controller (MC), and a local memory (Mem). Each global memory unit (GMU) has a memory controller and a global memory module. The GMU memory reserves some of its space for data structures needed to implement data merging.