To appear in Proceedings of HICSS '93

Data Merging for Shared Memory Multiprocessors

Alan H. Karp
Vivek Sarkar (IBM)
HPL-92-138
October, 1992

Cache coherence, delayed consistency, shared memory multiprocessors

We describe an efficient software cache consistency mechanism for shared memory multiprocessors that supports multiple writers and works for cache lines of any size. Our mechanism relies on the fact that, for a correct program, only the global memory needs a consistent view of the shared data between synchronization points. Our delayed consistency mechanism allows arbitrary use of data blocks between synchronizations.

In contrast to other mechanisms, our mechanism needs no modification to the processor hardware or any assistance from the programmer or compiler; the processors can use normal cache management policies. Since no special action is needed to use the shared data, the processors are free to act almost as if they are all running out of a single cache. The global memory units are nearly identical to those on currently available machines. We need to add only a small amount of hardware and/or software to implement our mechanism. The mechanism can even be implemented using network connected workstations.

Internal Accession Date Only

©Copyright Hewlett-Packard Company 1992

1 Introduction

The problem of cache coherence in shared memory multiprocessors has been studied almost as long as there have been such machines. A wide range of solutions have been proposed with varying degrees of hardware/software support required in the processors, cache controllers, and global memory units [7, 16, 3, 5, 4, 17, 18, 12, 8, 2, 10, 13, 1, 9, 6, 11, 15]. Our interest is in finding a solution with the following desirable properties:

1. The processor hardware need not have any knowledge that shared data is cached in multiple local memories.

   This property makes it possible to use off-the-shelf uniprocessors in an implementation of the solution.

2. The solution provides efficient support for false sharing of data blocks.

   False sharing is an important problem as more and more shared memory multiprocessors use a data block size that contains more than one data element.

3. The solution can exploit the slackness revealed by delayed consistency models.

   It has been widely observed that the strict ordering constraints imposed by sequential consistency can lead to performance inefficiencies, and that most parallel programs explicitly identify all synchronization operations thus allowing them to be executed on a weaker consistency model.

4. The programmer and the compiler need not have any knowledge that shared data is cached in multiple local memories.

   This property simplifies the programming model and makes it easier to write portable code. The programmer and compiler should only need to know the overheads associated with non-local memory references, not how the mechanism works.

5. The solution is effective for all data block size granularities.

   If the same solution can be used effectively for different data block size granularities, it will be applicable to a wide range of multiprocessors from tightly-coupled systems will small block sizes (*e.g.* a small scale bus-based multiprocessor) to loosely-coupled systems with large block sizes (*e.g.* a network of workstations). Also, the solution can be applied uniformly to multiple levels of a hierarchical shared memory multiprocessor.

6. The solution is simple to implement in hardware or software.

   This property makes it possible to build an implementation of the system in a short time; it also usually means that the overhead incurred by the solution will be smaller than that of more complex solutions.

To the best of our knowledge, none of the prior solutions satisfies all of these properties.

In this paper, we present an efficient software cache consistency mechanism for shared memory multiprocessors that supports multiple writers and works for cache lines of any size. Our solution is called Data Merging, and is based on the observation that any sharing that occurs between synchronization points during parallel execution is false. For false sharing, it is not necessary that caches be consistent; only global memory needs to be consistent. In particular, any concurrent updates to the same data block can be deterministically *merged* at global memory.

A brief summary of the data merging mechanism is as follows. The memory controller maintains a bitmask, a counter, and a suspend bit for each data block in global memory. 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 counter identifies the number of processors that currently have a copy of the data block. When the counter becomes zero, it triggers a reinitialization of the bitmask to all zeroes. For simplicity and scalability, we do not attempt to store the set of processor ids that have a copy of the data block; we just maintain the count. The suspend bit indicates whether or not a process has been suspended when attempting to access the data block. Since we do not keep track of users of data blocks, we cannot allow a processor to obtain a modified data block from global memory until its bitmask has been reinitialized. The suspend bit is needed in such a situation to suspend memory requests till the bitmask is reinitialized.

The key functional advantage of our solution is that it imposes only two minor requirements on the processor hardware/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. So, strictly speaking, our solution falls a little short of the first desirable property listed above. However, we feel that the gap is small and will most likely be overcome in future processor hardware and software.

Because our approach requires such small changes to the processing element hardware/software, our mechanism 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. The only extra processing support required is at the global memory units. This processing support can be implemented by special-purpose hardware, by programming a dedicated processor, or by time-slicing on the processor in the processing element. For a network of workstations, the appropriate data block granularity is the virtual page size. The processing support can be implemented by modifying the paging mechanism to send page requests over the network to global memory servers. The memory servers can be other workstations programmed to implement the protocols described in this paper.

The key performance advantage of our solution is that is that there is no impact to uniprocessor performance between synchronization/cache-miss points. A well-tuned parallel program should have a low cache miss rate and should have a granularity of parallelism that is well matched with synchronization overhead. Our solution is optimized for well-tuned parallel programs, and relieves the programmer from the burden of worrying about the performance impact of false sharing.

The rest of the paper is organized as follows. Section 2 contains a detailed description of the basic data merging mechanism. Section 3 discusses previous solutions to the cache coherence problem. Section 4 compares the performance of our mechanism with related solutions by a discussion of some simple codes. Section 5 discusses extensions to our basic data merging mechanism. Section 6 contains our conclusions.

2 Description of Basic Data Merging Mechanism

Section 2.1 defines the multiprocessor model assumed in this paper. The data merging mechanism is primarily concerned with the interactions between Processing Elements (PEs) and Global Memory Units (GMUs). Sections 2.2 and 2.3 described the actions performed by PEs and GMUs, respectively, in our data merging mechanism.

2.1 Multiprocessor Model

In this section, we describe the multiprocessor model assumed for describing the basic data merging mechanism. Extensions to the model and the mechanism are discussed in Section 5. In the following discussion, the terms *local memory* and *global memory* are used in a generic sense; for example, our mechanism applies to tightly-coupled architectures with hardware support for data transfers between cache and shared memory, as well as to loosely-coupled architectures with software support for data transfers among local memories.

The main components of the multiprocessor model outlined in Figure 1 are [14]:

- *Processing Element (PE)* — the processing unit that is replicated to obtain multiple processors in a multiprocessor system. Each PE consists of:
    1. *CPU* — the central processing unit which contains control units, execution units, control and data registers, control and data paths, *etc.*
    2. *Local Memory* — the memory hierarchy level in the PE that interfaces with the global memory. If the PE itself has a multi-level memory hierarchy, then "local memory" refers to the lowest level (furthest from the CPU) contained within the PE.
    3. *Local Memory Controller* — the control logic for issuing global memory requests from the PE.

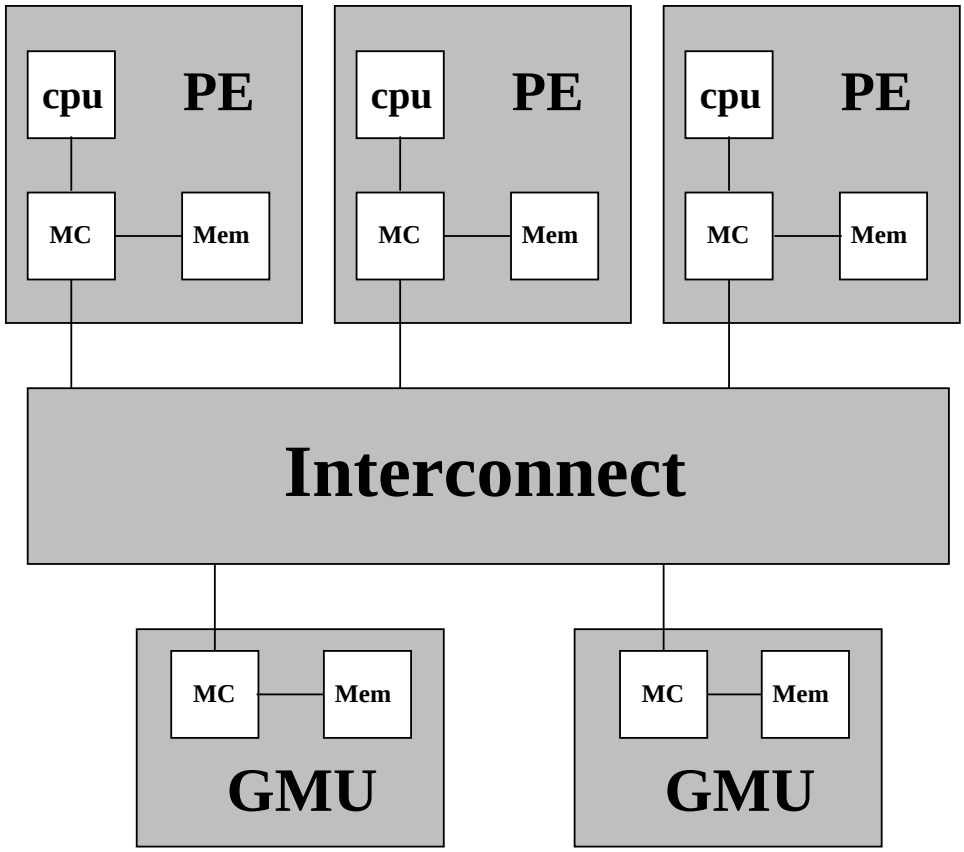

Figure 1: Multiprocessor memory model. Each processing element (PE) has a cpu, a memory controller (MC), and a local memory (Mem). Each global memory unit (GMU) has a memory controller and a global memory module. The GMU memory reserves some of its space for data structures needed to implement data merging.

Notice that we can use existing machines as our PEs.

- *Interconnect* — the communication mechanism used to transfer data between PEs and global memory, *e.g.*, a bus, crossbar switch, multistage network, or local-area network.
- *Global Memory Unit (GMU)* — the memory unit that is replicated to obtain a shared global memory which is addressable by all PEs. Each GMU consists of:
  1. *Global Memory Module* — a piece of the shared global memory. Some storage in the global memory module is reserved for GMU state information, as described in section 2.3.
  2. *Global Memory Controller* — the control logic for handling global memory requests from PEs. The global memory controller can be implemented by special-purpose hardware, or by programming a general-purpose processor.

It is possible to use the same hardware part as a PE or as a GMU. The GMUs can also be implemented on the same PEs as used for computation. In this case, the GMU memory controller would be a separate process running on the GMU, and some of the PE memory would be used to hold shared data. We could also dedicate some PEs to serve as GMUs.

We make the following assumptions about the components listed above [14]:

- The unit of transfer between local memory and global memory is a *fixed-size block*. We use the term *data block* to refer to a block in global memory, and the term *cache block* to refer to a copy of the block in a local memory. In contrast to other mechanisms, we also allow memory accesses to bypass cache.
- We assume that the storage granularity of global memory modules is a multiple of the block size, so that each data block is completely contained within a single GMU.
- Each cache block in local memory has an associated *dirty bit* that indicates whether or not the block was modified by the CPU.
- The block replacement policy used by the local memory is *write-back*, i.e., a cache block is written to global memory only when its dirty bit is set and it has been replaced or invalidated.
- If the processor stalls on a cache miss, the cache controller can still handle cache management requests.
- A *delayed memory consistency model* [1, 13] is assumed for data that is accessed by multiple processing elements. The programmer is required to use synchronization operations (*e.g.*, locks, barriers) to prevent concurrent accesses to the same memory word.
- The interconnect knows how to route a global memory request to the GMU that owns the data block. Typically, the GMU is identified by a simple hash function on the block-frame address. Also, the GMU knows which processor is associated with each global memory request.

2.2 Processing Element Actions

The local memory controller can issue the following kinds of global memory requests on behalf of its PE:

1. *Request cache block* — request a copy of a data block from the GMU that owns it.

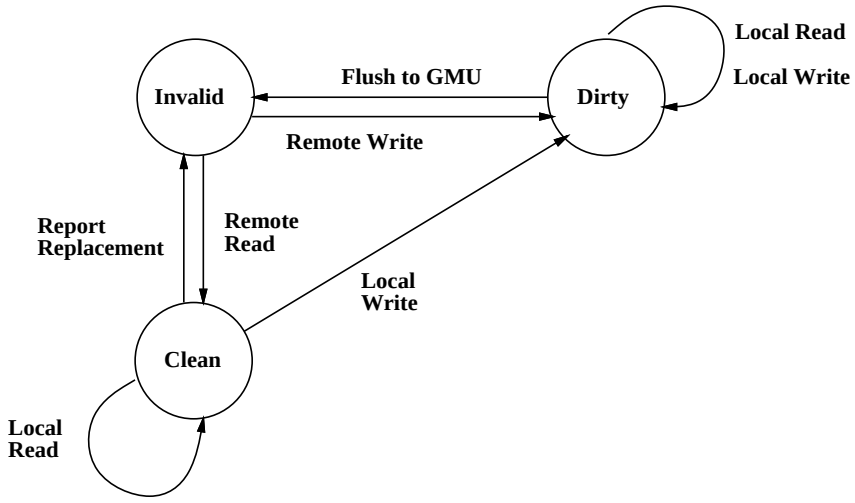

Figure 2: State transition diagram for PE local memory (cache) controller

2. *Flush data* — when replacing a dirty cache block, send its contents back to the GMU that owns the original data block.

3. *Report replacement* — when replacing a clean cache block, report its replacement to the GMU that owns the original data block, but don't send the data.

4. *Bypass-read data element* — the CPU reads a data element from global memory without storing a copy in the PE's local memory.

5. *Bypass-write data element* — the CPU stores a data element in global memory without storing a copy in the PE's local memory.

   Bypass-read and bypass-write may be used to enforce sequential consistency, rather than delayed consistency, on selected accesses to global memory. They are also more efficient for read/write accesses that have neither temporal nor spatial locality.

Since synchronization plays such an important role in parallel processing, we provide special functions accessible only through library calls to manage locks.

1. *Lock* — lock the specified data block.

2. *Unlock* — unlock the specified data block.

3. *Test and set lock* — set the lock if free and return the previous state of the lock.

The process must make sure that it has a current copy of any shared data before using it after a synchronization. The simplest scheme is for the entire local memory to be invalidated.

A better approach is to have the process calling the synchronization routine invalidate any shared data it may access. Other optimizations can be done if the programmer or compiler provides additional information on which data needs to be refreshed.

The state transition diagram for the local memory controller is shown in Figure 2. A request for global data not in the cache, either for read or write, causes a clean copy of the data block to be put into the local cache. A local write makes the block dirty. The cache block can be made invalid either by normal cache management policies or by an invalidate signal sent by the GMU. If the block is dirty it is flushed; if clean, its replacement is reported to the GMU, but no data is moved.

Note that all accesses to shared data that are hits in local memory are handled in the same way (and with the same efficiency) as in a uniprocessor *i.e.* as if the PE were the only one accessing the shared data. Further, from the PE's viewpoint, accesses to shared data that miss in local memory behave just like accesses to the next level of a uniprocessor's memory hierarchy. The only difference between the state transition diagram in Figure 2 and the state transition diagram for a uniprocessor cache controller is that we also require the GMU to be notified whenever a clean cache line is replaced.

2.3 Global Memory Unit Actions

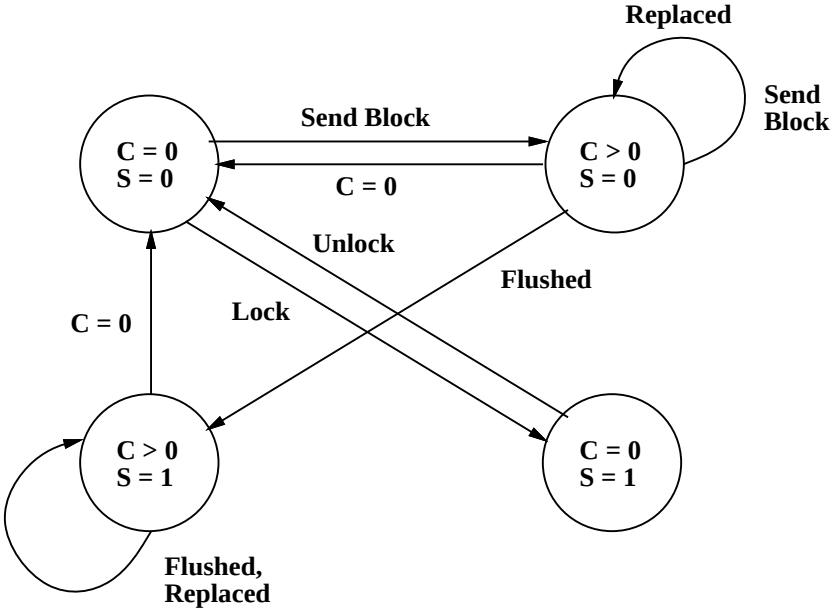

Figure 3: State transition diagram for global memory controller

The actions performed by the GMU for supporting the data merging mechanism are described below. In addition to space to hold its assigned data blocks, the global memory module also contains extra storage to store the following state information:

1. Associated with each data block assigned to the GMU are:

   (a) A *counter* (C) of size $\geq \log(P)$ bits, where P = maximum number of PEs in the shared memory multiprocessor system. The counter identifies the number of PEs that currently have a copy of the data block in their local memories.

   (b) A *suspend bit* (S) that indicates whether or not a process has been suspended when attempting to access the data block.

   (c) A *bitmask* with number of bits = number of elements in a data block. The bitmask identifies the elements that have been modified in the data block.

   The state information actually stores a pointer to the bitmask, and the bitmask is allocated only if needed *i.e.* only if concurrent accesses to the data block actually occur at runtime (this condition is identified by testing $C > 1$). A possible extension would be to store the data element size as well in the state information, thus allowing for allocation of variable-sized bitmasks and also more efficient processing of large-sized data elements.

2. A dynamically-updatable suspend queue capable of holding up to one entry per process in the multiprocessor system.

The following actions are performed by the global memory controller in response to the PE requests outlined in Section 2.2

1. *Request cache block*
   When a PE requests a copy of a data block from the GMU, there are two possible cases:

   (a) The suspend bit is in the initial state (zero).
   In this case, the global memory controller increments the counter and sends a copy of the data block to the PE.

   (b) The suspend bit is not in the initial state.
   The global memory controller inserts the data block request, requesting processor id, and current time stamp into the GMU's suspend queue.

2. *Flush data*
   In this case, the PE replaces a cache block in its local memory and sends the contents of the old cache block (C) to be merged with data block D in the GMU. There are two cases of interest for the state information associated with data block D:

   (a) Counter = 1 and suspend bit = 0.
   In this case, the global memory controller stores cache block C into data block D and resets the counter to zero.

(b) Otherwise.
In this case, the global memory controller uses the bitmask to merge selected words from cache block C into data block D as follows:

i. For each word $D[i]$ in data block D with bitmask entry = 0 do
If $C[i] \neq D[i]$ then
- set bitmask entry $i = 1$
- $D[i] := C[i]$

ii. Decrement the counter for the data block.

iii. Set the suspend bit to 1.

iv. If the counter becomes zero, then perform a *bitmask-reinitialize* operation.

3. *Report replacement*
In this case, the PE sends notification of the replacement to the GMU, without actually sending the data in the cache block. The global memory controller performs the following actions:

(a) Decrement the counter for the data block.

(b) If the counter becomes zero, then perform a *bitmask-reinitialize* operation.

4. *Bypass-read data element*
When a PE requests a data element by bypassing local memory (cache), the GMU just sends the data element to the requesting processor. No state information needs to be checked or modified.

5. *Bypass-write data element*
When a PE writes a data element by bypassing local memory (cache) the GMU updates the corresponding global memory location. No state information needs to be checked or modified.

6. *Lock*
In this case the PE has requested that the data block be locked.

(a) If the suspend bit is set, insert the request into the GMU's suspend queue along with the requesting process id and a special flag in place of the time stamp.

(b) If the suspend bit is not set, set the bit, and return its old value.

7. *Unlock*
In this case the PE has requested that the data block be unlocked. We assume that the library routine that generates these calls makes sure the process attempting to free the lock is the one that set it.

(a) Perform a *bitmask reinitialize* operation.

8. *Test and set lock*
   Set the suspend bit to 1 and return the previous value of the suspend bit to the requesting process.

Apart from the above actions in response to PE requests, the GMU also needs to perform the following actions to properly maintain its internal state:

1. *Initialize*
   Whenever a piece of global memory is allocated for use by the PEs, the global memory controller initializes the data block's bitmask pointer, count, and suspend bit to all zeros, for each data block in the allocated piece of global memory.

2. *A data block has a null bitmask pointer, and its counter becomes* > 1
   Allocate storage for the bitmask in the GMU's memory module, and set the bitmask pointer to the allocated storage.

3. *Bitmask-reinitialize*
   Reinitialize the bitmask and suspend bit to all zeroes. Check if this event affects any processor in the suspend queue. If so, handle the request of the suspended processor as in action 1a above (GMU action when PE requests a data block and the suspend bit is in its initial state).

4. *Scan suspend queue*
   Every so often scan the suspend queue. For each data block request that is older than some age threshold, perform the following actions (the scan frequency and age threshold are system tuning parameters):

   (a) Broadcast an invalidate command for the data block to all PEs. (This invalidate will result in a replacement report or a flush request from all local memories that own a copy of the data block, which will eventually cause the suspended process to be awakened).

The state transition diagram for the global memory controller is shown in Figure 3. The data block in the global memory unit can be in one of four states. In its initial state, both the counter of outstanding copies, C, and the suspend bit, S, are zero. A request for the data makes $C = 1$. Further requests increment C; clean blocks replaced in the processor caches result in the counter being decremented. Dirty blocks replaced in the processor caches result in the suspend bit being set and the counter being decremented. When the counter reaches zero, the cache block returns to its initial state.

A special state, with $C = 0$ and $S = 1$ is used for data blocks used as synchronization variables. This state can only be reached using a library call to set the lock. Among our proposed extensions is a provision to use this state to implement fetch-and-op in the GMU.

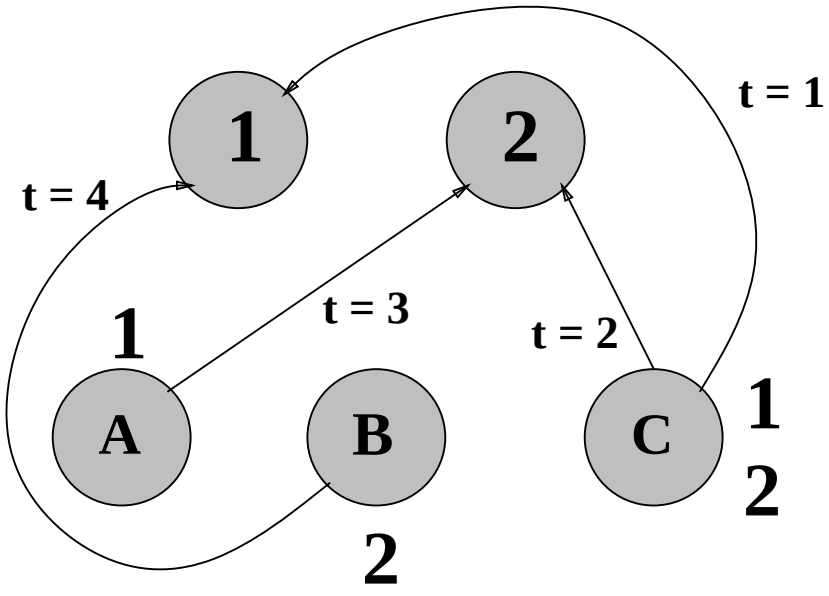

Figure 4: Deadlock example in the absence of time-out

It is possible to deadlock the machine with our mechanism unless we make provisions for requests to time out by having the GMU periodically scan the suspend queue. In Figure 4 we show a deadlock. We have data blocks 1 and 2 and processors A, B, and C. At time $t = 0$, A holds data block 1, B holds 2, and C holds both 1 and 2. Assume that at time $t = 1$ C flushes a dirty copy of data block 1 and at time $t = 2$ C flushes a dirty copy of data block 2. At time $t = 3$ A requests a copy of data block 2, and is suspended. At time $t = 4$ B requests a copy of data block 1 and is suspended, too. We now have a deadlock.

Sooner or later, perhaps after a context switch, the data will be flushed from one of the caches, and the job will proceed. However, in stand-alone operation the flush might not happen for a long time. Hence, we periodically check the age of requests in the suspend queue. In this example, the system will broadcast an invalidate signal for data block 2 (since A asked first). B will then flush block 2 allowing A to proceed. Once A is running, it will flush data block 1 allowing B to proceed.

3 Related Work

The problem of cache coherence in shared memory multiprocessors has been studied almost as long as there have been such machines [7, 16]. This early work concentrated on maintaining *sequential consistency* [16] in which the result of a run corresponds to some sequential ordering of the data accesses. A simple approach is to bypass cache on all accesses to shared data. Unfortunately, the time to get to main memory is so large on today's machines, that this approach is not viable. Another simple approach is to keep each shared data element in

a distinct cache block [5]. This approach can waste a lot of memory and destroy the spatial locality of reference in shared data.

Two mechanisms have been proposed that maintain sequential consistency for cached data when the cache block is allowed to contain multiple shared data elements [7]. With *invalidate on write*, the processor writing a cache block sends an invalidate signal to the other processors. With *update on write*, the modification is sent to the other processors. The problem is that every modification of shared data results in messages even when the other processors have no need to know the update was made. Performance can be improved by processing the invalidate requests before forwarding them to the cache [3] or by buffering them [10]. No matter what is done to reduce the impact of these messages on the cache, the large amount of unnecessary network traffic makes it unlikely that these schemes can be used to build scalable systems.

Since false sharing accounts for a large fraction of these messages in many programs, the performance of the system suffers. If the system invalidates on write, cache blocks can "ping-pong" between two processors modifying distinct words in the block. If the system updates on write, both sending and receiving processors will be slowed because they must process irrelevant messages. Some systems, such as DASH [13], use an invalidate on write protocol and make no special provisions for false sharing.

A number of systems have been designed to maintain sequential consistency while reducing the performance impact of false sharing. Ivy [17] uses a write invalidate protocol on page size blocks. Since large blocks are more likely to be falsely shared, it is left to the compiler or programmer to effectively align the data. Clouds [18], which uses objects as its unit of consistency, allows the processor to lock the object to avoid cross invalidates. Mirage[12] automatically locks pages, its unit of consistency, for a certain amount of time. Orca[4] uses reliable broadcast for both invalidate and update protocols. Amber[8] avoids the problem entirely by requiring the programmer to move data between processors. APRIL[2] guarantees sequential consistency, hiding the unavoidable delays with very fast context switching.

There has recently been a great deal of interest in *delayed consistency* protocols. In these schemes, data update messages can be delayed till synchronization points thus making it possible to overlap more useful work with message latencies. The *weak ordering* model [1] can be summarized by three rules: a) accesses to synchronization variables are sequentially consistent, b) no global access can start till any prior access to a synchronization variable has completed, and c) all global accesses must complete before the next access to a synchronization variable is issued. In the *release consistency* model [13], accesses to synchronization variables are categorized as acquire/release operations to further relax the ordering constraints. Rule b) is only applied to acquire operations, and rule c) is only applied to release operations. These delayed consistency solutions require modifications to the cache controller of each processor and that extra work be done. In addition, since synchronization variables control when the updates are propagated, they must be treated specially by the memory

system.

Munin [6] is a distributed shared memory system that allows shared memory parallel programs to be executed efficiently on distributed memory multiprocessors. Munin implements a release consistency model by placing a data object directory in each processor. On the first modification to an object with multiple writers, a "twin" copy is made. When the next synchronization event occurs, the copy is compared to the modified object. The differences are encoded and sent to the other processors holding the object. Good performance is obtained if the programmer or compiler tags each object with its usage pattern, *e.g.*, write once, write shared, reduction, producer consumer, *etc.* A similar scheme was used by the Myrias SPS with the copy on write and comparison being done in hardware [9].

Both the receive delayed and the send-and-receive delayed protocols need additional hardware support so the cache can know which data is *stale* [11]. A cache block becomes stale when an invalidate command is received from another processor. Any write to a stale block sends an invalidate to all other holders of the block and gets a fresh copy from memory. Performance is improved by sending invalidate commands only once per object between synchronizations. All stale blocks become invalid at a synchronization point. Additional hardware is needed for an invalidation send buffer and support for partial updates to memory.

More recently, there has been work done on *lazy release consistency (LRC)* [15], a new algorithm for implementing release consistency that only performs modifications as needed. This algorithm is an extension of the release consistency algorithm in Munin's write-shared protocol [6]. Unlike previous implementations of release consistency that used the weaker ordering constraints only to hide latency, the LRC algorithm can also reduce the number of messages and the amount of data transferred for global memory accesses. Modifications are not made globally visible each time a release occurs; instead, the modifications are forwarded to only the processor that performs a corresponding acquire operation.

Of all the related work discussed here, the LRC algorithm is the most closely related to the data merging mechanism described in this paper. Table 1 contains a summary comparison of the two mechanisms.

4 Examples

Any realistic memory subsystem makes compromises. These compromises mean that some programs run well and others poorly. In this section we compare the estimated performance of our mechanism and some others on a variety of simple codes.

On a machine with multiword cache lines, any cache coherence protocol that does not allow multiple writers will perform poorly on the loop

Table 1: Comparison between Lazy Release Consistency and Data Merging mechanisms

	Lazy Release Consistency	Data Merging
State information	Data object directory. Each node maintains the following directory information for each shared data object (page): set of remote processors that have a copy of the object, start address and size, protocol parameter bits, object state bits, access control semaphore for directory entry, miscellaneous pointers.	Data block state. Each GMU maintains the following information for each shared data block: one counter, one suspend bit, one bitmask pointer, and a dynamically-allocatable bitmask. No information is maintained to identify the set of processors that have a copy of the data block.
Write notification	A write-notice is propagated on each message that causes a release-acquire synchronization to occur. A write-notice is an indication that a page has been modified in a particular interval.	The PE notifies the GMU of a write only if a flush occurs or if a bypass-write operation is performed.
Cache miss	A copy of the page is retrieved as well as a number of diffs which have to be merged into the page before the page can be accessed.	If the suspend bit is zero, the GMU sends a copy of the data block to the PE. Otherwise, the request is placed on a suspend queue until the counter becomes zero or a timeout occurs.
Support for multiple writers	A duplicate copy (twin) of the data object is made on each writer node. A word-by-word comparison of the object and its twin results in a "diff", which is sent to all nodes requiring updates.	The bitmask maintained by the GMU identifies the elements that have been modified in the data block and thus controls the merging of different cache blocks into the same data block. A word-by-word comparison is done between a cache block and the data block to update the bitmask.
Invalidate protocol	An acquiring processor invalidates all pages in its cache for which it received write-notices.	A synchronizing processor invalidates all potentially shared data in its local memory.
Update protocol	An acquiring processor updates all pages in its cache for which it received write-notices by obtaining diffs from all intervals that are concurrent last modifiers.	The basic data merging mechanism has no support for direct update of local memory.

```
do i = me(), n, ntasks()
   a(i) = function(i)
enddo
```

where `me()` returns a unique task identifier between 1 and `ntasks()`, the number of tasks in the parallel job. In this loop each cache line is shared by several tasks. Performance will be poor because a message will be sent every time an element of `a` is modified. More importantly, such a machine will probably perform poorly on a self-scheduled variant of this loop. Any system that allows multiple writers to a cache line should perform well on this example. No network traffic will be generated until the tasks synchronize at the implicit barrier at the end of the loop.

Since there is such a large difference between schemes that impose sequential consistency and those that delay consistency, our comparison will be between our method and the delayed consistency scheme most similar to it, Munin.[6] The newer version of Munin using *lazy release consistency*[15] is much more complicated. The only fair comparison would be to our method with some of the extensions described in Section 5. Since we have not fully studied these extensions, we limit our comparison to be between the original Munin proposal and our simplest variant. A functional comparison of our method with lazy release consistency is given in Section 3.

In order to make an *apples-to-apples* comparison, we make two assumptions. The first, which penalizes Munin, is that neither the programmer nor the compiler has inferred how the data is shared. This assumption is likely to be reasonable for the first example discussed. If such information is available, our mechanism could use it, too. The second assumption, which penalizes our mechanism, is that the GMUs are implemented as processes running on the PEs. Also, although we talk about cache blocks, the same statements are true if we substitute the word `pages` as used by Munin.

Now let's look at two programs. We will show that our mechanism is better for a popular method for solving partial differential equations while Munin does better on a common data base function.

Domain decomposition is a popular method for solving partial differential equations on parallel processors. The problem domain is divided into a region for each processor. The processors compute the solution in the interior of their regions independently. They then exchange information with their neighbors and update their boundary regions. The only synchronization is in the data exchange process. The procedure

1. Process my interior points
2. Synchronize with my neighbors

3. Exchange data

4. Process my boundary points

is repeated for each step of the solution scheme. It is common practice to declare the entire array to be shared. Hence, the usage pattern hints used to improve performance when using Munin are ineffective.

In our mechanism, each process will access the part of the shared array corresponding to its interior points. The global memory controller will deliver the data and increment the counter for each cache block accessed. When the data is flushed from the cache by normal cache management protocols, the modified cache block will be sent to the GMU. Since the count for the block is one, and the suspend bit is zero, the data will be stored, and the counter reset to zero. In other words, negligible extra work is done for data corresponding to interior points.

A synchronization will be done before processing the boundary points. In our simplest implementation, the entire cache will be invalidated before proceeding. The loss in performance will be small since we expect the problem size to be so large that the data being invalidated by synchronizations would be flushed anyway by normal cache block replacement procedures before it could be used again. Furthermore, only two processes need access to the boundary elements, one for read and one for read/write. Hence, processors should be put into the suspend queue only rarely.

Munin does not do as well on this example. Since the entire array was declared as a unit, Munin must assume that all cache blocks can have multiple writers. Every piece of data will be cloned and `diff`ed. The `diff` file will be as large as the cache blocks, larger if we count the extra control information needed to encode the sparse updates Munin assumes. These updates will be transmitted to the global memory when the code reaches one of the synchronization points. It is not clear how Munin deals with normal cache block replacement events.

Clearly, Munin has done much more work than our mechanism. First, Munin was forced to clone and `diff` blocks that never could have more than one writer. If it had such information, Munin could have avoided much of this work, but our mechanism could have used the information to decide which cache lines to invalidate at synchronization points. Secondly, Munin's messages are longer than ours because Munin is designed to send sparse updates to a page and must encode what data is to be modified. Our method sends only the data with very little control information.

This example illustrates one key difference between our mechanism and Munin. Munin must do extra work on any cache block that *might* be shared, even if the sharing did not actually occur. Our mechanism, by doing the merge operation in the GMU, does the extra work only when the data is actually shared.

Now look at a case that favors Munin. Assume we have a database with a record for each passenger flight of a major airline. The record for a flight is updated each time a seat is assigned and each time a crew member is assigned to the flight. Assume we have two tasks, one to update the seat assignments and the other to update the crew. Since these two tasks are modifying different parts of the record, they may execute in parallel without any synchronization until they need to commit the changes.

Munin should perform well in such an environment. The first update to the data in a cache block will cause Munin to clone the data. Subsequent updates will be made efficiently. Each commit operation requires Munin to do the `diff`. The cost of the cloning and `diff` operations is amortized over the number of changes made between commits. We expect the `diff` file to be quite small, so Munin will generate relatively little network traffic. Munin will also be able to use its directory information to do direct cache to cache transfers of the data from the processor that modified it to those holding copies.

Our mechanism, at least in its simpler implementations, will not do as well. Each commit will cause us to invalidate the cache. Even if we invalidate only those cache blocks holding modified data, the entire block must be sent to the GMU. If there are readers holding the cache blocks for a long time ("Does your brother want the kosher meal or the low salt meal, Sir?"), the suspend bit will be set for a long time and lots of processors will end up in the suspend queue.

From these examples we can characterize the situations that favor our proposal and those that favor Munin. If the data usage pattern is unknown, but the data is most often accessed by a single processor at a time, our mechanism will do better. If the data is sparsely written and the synchronizations occur frequently, Munin will do better. Both methods will outperform sequential consistency schemes if there is a lot of false sharing and will do worse if false sharing is negligible.

5 Extensions to Basic Data Merging Mechanism

Having described the basic data merging mechanism in Section 2, we now point out a number of enhancements that can be made. For example, we can do global reductions, such as fetch-and-add, directly in the memory unit using the memory locations in the data block protected by a lock. We can also use this space to carry data to be transmitted to other processors testing the lock. For example, we might want to put in information to be used by a trace tool.

A more interesting possibility is having a debug mode that would report access anomalies. This mode could be implemented by cloning the original data block at the GMU when it is first accessed. Comparing the modified cache block to both the original data and the partially modified data block will enable us to determine that two processes did modify the same word between synchronizations. Depending on how much additional information we

wish to store, we can report that the anomaly occurred and which processes participated.

Our approach can even work for processors that do not have the ability to notify the GMU when they replace a clean cache block. Instead of counting the number of processors holding the data, we count only those that write to it. The GMU sends out the data with read-only permission and does not increment the counter. When a process attempts to modify the corresponding data block, it will trap. The trap handler can tell the GMU to increment its counter. When the cache line is replaced, it will be flushed to the GMU. Eventually the counter will reach zero. We need not keep track of readers if we are willing to trap on the first write to a cache block.

The performance of our scheme would improve markedly if we could modify the cache to keep a dirty bit per element. Not only would we not need to send the entire cache block on an invalidate, but we could do without the suspend queue. We need the suspend queue only for the case where a process modifies a word twice, once before a flush and once after. Our basic mechanism has no way of knowing that the second change occurred so that change is lost. A dirty bit per element solves this problem.

The scalability of systems that use broadcasts is subject to question. Our scheme broadcasts only when processors have been in the suspend queue too long which we don't think will happen too often. However, if we keep a directory in the GMU, we will be similar to Munin in that we only need to send the invalidate message to processors holding the data.

One problem we have is that we invalidate all or most of the cache at a synchronization. We could reduce the number of invalidates if the programmer or compiler told us which data would be used before the next synchronization. Sometimes this is easy to do. For example, in those situations where an optimizer could move a data reference across the synchronization call, the programmer must explicitly list the variables to be protected.[19]

Memory hot spots are always a problem in massively parallel systems. If a lot of processors request the same data block simultaneously, the GMU that owns that data could be tied up for a long time. Other processors needing other data blocks held by that GMU will also be delayed. Our solution to the problem is the one most often proposed – use a combining network so that the GMU only needs to send the data block a few times.

6 Conclusions and Future Work

Our data merging scheme comes from a common question asked by people designing scalable multiprocessors. Is it possible to maintain the appearance of cache consistency without modifying the basic processor hardware? We believe we have shown that it is. In fact, the system we have described has the additional desirable properties outlined in the Introduction.

We do not yet have an implementation on which to make performance measurements. The simplest approach would be to change the paging mechanism of some existing Unix work-

stations to do their paging over the network. Other workstations could be programmed to implement the GMU controller functions. While such an implementation would suffer from the well-known problems of high latency and limited bandwidth, it would permit us to test the approach quickly. If data merging in the GMU works as well as we believe it will, it should be relatively easy to produce a more tightly coupled machine.

Acknowledgements

We would like to thank Rad Olson, Ray Bryant, and Henry Chang of IBM, Kourosh Gharachorloo of Stanford, Radhika Thekkath of U. Washington, and Martin Fouts, Dennis Brzezinski and Rajiv Gupta of HP for their comments and suggestions.

7 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] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] B. M. Bean *et al.* Bias Filter Memory for Filtering out Unnecesssary Interrogations of Cache Directories in a Multiprocessor System. U.S. Patent #4,142,234, February 1979.

[4] H. E. Bal and A. S. Tanenbaum. Distributed Programming with Shared Data. In *Proc. IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, 1988.

[5] P. Bitar and A. Despain. Multiprocessor Cache Synchronization: Issues, Innovations, Evolution. In *Proc. of the 13th Annual Int. Symp. on Comp. Architecture*, pages 424–433, June 1986.

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

[7] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multilevel Caches. *IEEE Trans. on Computers*, C-27(12):1112–1118, December 1978.

[8] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[9] Myrias Corporation. System Overview. Edmonton, Alberta, 1990.

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

[11] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *Proc. Supercomputing '91*, pages 197–206, November 1991.

[12] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, Decmeber 1989.

[13] 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.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[15] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

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

[17] K. Li and P. Hudak. Memory coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[18] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proc. of the 1989 Conference on Parallel Processing*, pages II–160–II–169, June 1989.

[19] Leslie J. Toomey, Emily C. Plachy, Randolf G. Scarborough, Richard J. Sahulka, Jin F. Shaw, and Alfred W. Shannon. IBM Parallel Fortran. *IBM Systems Journal*, 27(4):416–435, 1988.