Cache Coherence Protocol Design for Active Memory Systems

Mainak Chaudhuri, Daehyun Kim, and Mark Heinrich
Computer Systems Laboratory, Cornell University, Ithaca, NY, U.S.A.

Abstract— *Active memory systems improve application cache behavior by either performing data parallel computation in the memory elements or supporting address re-mapping in a specialized memory controller. The former approach allows more than one memory element to operate on the same data, while the latter allows the processor to access the same data via more than one address—therefore data coherence is essential for correctness and transparency in active memory systems. In this paper we show that it is possible to extend a conventional DSM coherence protocol to handle this problem efficiently and transparently on uniprocessor as well as multiprocessor active memory systems. With a specialized programmable memory controller we can support several active memory operations with simple coherence protocol code modifications, and no hardware changes. This paper presents details of the DSM cache coherence protocol extensions that allow speedup from 1.3 to 7.6 over normal memory systems on a range of simulated uniprocessor and multiprocessor active memory applications.*

Keywords: Active memory systems, address re-mapping, cache coherence, distributed shared memory, flexible memory controller.

1 Introduction

Active memory systems provide a promising approach to overcoming the memory wall for applications with irregular access patterns not amenable to techniques like prefetching or improvements in the cache hierarchy. The central idea in this approach is to perform data-parallel computations [2, 4, 7] or scatter/gather operations invoked via address remapping techniques [1] in the memory system to either offload computation directly or to reduce the number of processor cache misses. Both active memory approaches create coherence problems—even on uniprocessor systems—since there are either additional processors in the memory system operating on the data directly, or the main processor is allowed to refer to the same data via more than one address.

This paper focuses on the challenges of designing cache coherence protocols for active memory systems. The necessity of enforcing data coherence between the normal cache line and the re-mapped cache line makes the protocol behavior and the performance requirements quite different from those of a conventional DSM protocol. We propose an active memory system that supports address remapping by leveraging and extending the hardware DSM directory-based cache coherence protocol. The key to our approach is that the active memory controller not only performs the remapping operations required, but also runs the directory-based coherence protocol and hence controls which mappings are present in the processor caches.

The paper is organized as follows. Section 2 describes the protocol extensions required and protocol implementation issues unique to active memory systems. Section 3 discusses active memory protocol-related performance issues. Section 4 presents both uniprocessor speedup and the performance improvement of active memory applications on single-node multiprocessors. Section 5 concludes the paper.

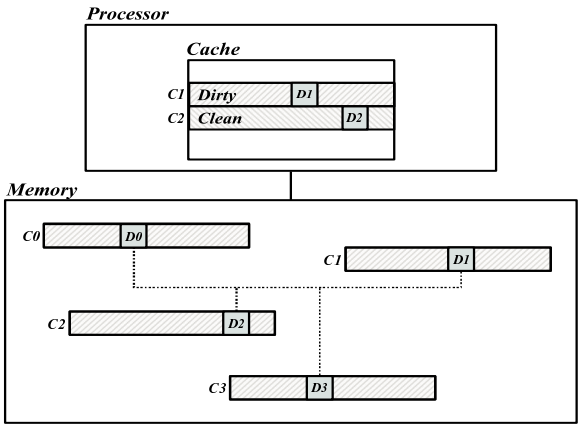

Figure 1. Data Coherence Problem

2 DSM Protocol Extensions for Active Memory Systems

Our embedded memory controller runs software code sequences to implement the coherence protocol following the same philosophy as the FLASH multiprocessor [6]. However, the controller also includes specialized hardware to speed up active memory protocol execution. Our base (non-extended) protocol is a conventional invalidation-based MSI bitvector directory protocol running under release consistency. For all normal (non-re-mapped) memory requests, our memory controller follows this base protocol. Each directory entry (per 128B cache line) is 8 bits wide. The sharer vector occupies 4 bits, so we can support up to 4 processors. This can be expanded to larger machine sizes by increasing the width of the sharer field. These four bits store a sharer list vector if the cache line is in the shared state or an owner identifier for the dirty exclusive state. Two bits are devoted to maintain cache line state information. The dirty bit indicates whether a cache line is in the dirty exclusive state. The AM bit is used for our active memory protocol extensions and is not used by the base protocol. Two remaining bits are left unused.

2.1 Active Memory Extensions

As an example of address remapping techniques giving rise to a data coherence problem, Figure 1 shows that the data element $D0$ in cache line $C0$ is mapped by some address remapping technique to data elements $D1, D2$ and $D3$ belonging to three different cache lines $C1, C2$ and $C3$, respectively. This means that $D1, D2$ and $D3$ represent the same data variable as $D0$ and our active memory protocol extensions must keep them coherent. As in the figure, if $C1$ and $C2$ are cached in the dirty and shared state, respectively, the processor may write a new value to $D1$, but read a stale value from $D2$. We extend the base cache coherence protocol to enforce mutual exclusion between the caching states of the lines that are mapped to each other so that only one cache line of the four mapped cache lines (as in the example above) can be cached at a time. If the processor accesses another mapped cache line it will suffer a cache miss and our protocol will invalidate the old cache line before replying with the requested cache line.

For each memory request, our protocol consults the AM bit in the directory entry for the requested cache line. The AM bit of a cache line indicates whether any data in the line has been re-mapped and is being cached by processors in the system at a different address. If the AM bit is clear, there is no possibility of a coherence violation and the protocol behaves just like the base protocol with one additional task—to guarantee coherence in the future, the protocol sets the AM bits in the directory entries of all the cache lines that are mapped to the requested line. However, if the AM bit is set, some of the re-mapped cache lines (we collectively call these lines R) are cached in the system and there is a potential data coherence problem. The caching states of R are obtained by reading the correspond-

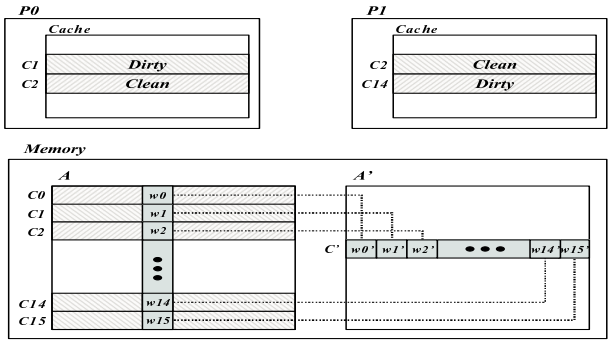

Figure 2. Example - Matrix Transpose

ing directory entries. If R is in the dirty exclusive state, the protocol sends an intervention to the owner of R to retrieve the most recent copy of the data, updates the mapped data value in the requested cache line, sends the data reply to the requester, and writes back the retrieved cache line to main memory. If R is in the shared state, the protocol sends invalidation requests to all the sharers, reads the requested cache line from memory (since it is clean) and sends the data reply to the requester. In both cases, the protocol updates the AM bits in the directory entries of all the cache lines mapped to the requested line.

2.2 Support for Active Memory Transpose

Although we have implemented four active memory operations—*Matrix Transpose*, *Sparse Matrix Scatter/Gather*, *Linked List Linearization*, and *Memory-side Merge* [5]—for space reasons we present only matrix transpose as an example. Assume the following: A is a square matrix of dimension N, A' is a transposed matrix mapped to A, and two processors $P0$ and $P1$ access them. The cache line size is 128 bytes and the data element size is 8 bytes (one double word), so one cache line contains 16 data elements. At a certain point in time, the memory snapshot is as depicted in Figure 2 and $P0$ executes the following code:

```
for i=0 to N-1
  for j=0 to N-1
    x += A'[i][j];
```

$P0$ reads a cache line C' of A' and it misses in the data cache. C' is composed of 16 double words $w0', w1', \ldots, w15'$ that are same as $w0, w1, \ldots, w15$, and the 16 cache lines $C0, C1, \ldots, C15$ of A contain them. $P0$ and $P1$ are caching $C1$, $C2$ and $C14$ in the dirty or shared states as shown in the figure. $P0$ sends a read request to the main memory and the memory controller invokes the appropriate coherence handler.

The protocol handler reads the directory entry of C' and checks the AM bit. In this case the AM bit is set because $C1$, $C2$ and $C14$ are mapped to C' and they are cached. Therefore, instead of using the base protocol our extended protocol is invoked. From the physical address of C' the address remapping hardware (the detailed micro-architecture of the controller can be found in [5]) calculates the addresses of the cache lines mapped to C', which in this case are $C0, C1, \ldots, C15$. Since C' belongs to the re-mapped address space and the re-mapped physical address space is contiguous, from the physical address of C' the protocol can calculate the position of C' in the transposed matrix (i.e. A') if it knows the dimensions of the matrix and the size of each element of the matrix in bytes. This information along with the starting virtual address of the matrix (i.e. A) is stored in a table that is initialized at the beginning of the application via a system-level library call. Using the position of C' in A' and the starting virtual address of A the protocol calculates the virtual addresses of $C0, C1, \ldots, C15$ and looks up a TLB resident in the memory controller to compute the corresponding 16 physical addresses. Next, the protocol reads the directory entries for each of these cache lines and consults the dirty bit and the sharer vector. For $C1$, we find that it is cached by $P0$ in the dirty exclusive state. The protocol sends an intervention to $P0$ for this line because at this point $w1'$ has a stale

value and the most recent value is $w1$ residing in $P0$'s cache. On receiving the reply from $P0$, the protocol updates $w1'$ with the correct value and also writes back $C1$ to memory. For $C2$, we find that it is cached by $P0$ and $P1$ in the shared state, so the protocol sends invalidations to $P0$ and $P1$ for this line. In this case, the protocol can read the correct value of $w2'$ directly from main memory. The case for $C14$ is similar to $C1$ except that $P1$, instead of $P0$, is caching this line. For the other cache lines that are clean in main memory, the protocol need not do anything. Now that the protocol has evicted all the cached lines remapped to C' from the caches of $P0$ and $P1$ and updated the data of C' with the most recent values, it is ready to reply to $P0$ with C'. Finally, the protocol updates the AM bits of all the directory entries of the cache lines re-mapped to C'. Because C' is now cached, the AM bits of $C0, C1, \ldots, C15$ are set and that of C' is clear. This guarantees correctness for future accesses to any of these cache lines.

We have described how our Matrix Transpose protocol enforces mutual exclusion between the caching states of normal and re-mapped lines. However, this is overly strict since it is legal to cache both normal and re-mapped lines provided both are in the shared state. We find though that for Matrix Transpose, enforcing mutual exclusion achieves higher performance because all transpose applications we have examined have the following sharing pattern: a processor first reads the normal space cache lines from the portion of the data set assigned to it, eventually writes to it and then moves on to read and eventually update the re-mapped space cache lines. Therefore, accesses tend to "migrate" from one space to another. When the active memory controller sees a read request for normal or re-mapped space it knows that eventually there will be an upgrade request for the same cache line. Further, there will be no access from the other space between the read and the upgrade. So our cache coherence protocol extensions choose to invalidate all the cache lines in the other space mapped to the requested line at the time of read. This keeps the upgrade handler much simpler because it does not have to worry about invalidating the shared cache lines. However, we found that Sparse Matrix Scatter/Gather does not exhibit a migratory sharing pattern, and therefore we relax the mutual exclusion constraint in that case. This illustrates an advantage of flexible memory controllers that can adapt the coherence protocol to the needs of the application.

2.3 Multi-node Protocol Extensions

This section discusses issues related to multi-node extensions of our single-node active memory protocol. Our base multi-node protocol is an MSI invalidation-based bitvector running under release consistency with a directory entry size of 8 bytes (per 128B of cache line). To reduce the occupancy at the home node, invalidation acknowledgments are collected at the requester. To reduce the number of negative acknowledgments in the system, the home node forwards writebacks to a requester whose interventions were sent too late, and the dirty third node buffers early interventions that arrive before data replies.

Our initial findings on multi-node active memory protocol extensions suggest that special care should be taken when assigning pages to home nodes. All cache lines mapped to each other should be co-located on the same node; otherwise, a request for a local memory line may require network transactions to consult the directory states of other remote lines that are mapped to it. This would complicate the protocol handlers as well as degrade performance. Further complications can arise while gather-

ing invalidation acknowledgments on multi-node systems. The active memory protocol needs to invalidate cache lines that are mapped to the requested line and cached by one or more processors. But the requested line and the lines to be invalidated have different addresses. Therefore, invalidation acknowledgment and invalidation request messages should carry different addresses or at the time of gathering invalidation acknowledgments a mapping procedure has to be invoked so that the invalidation requests get matched to the corresponding acknowledgments. Finally, we also need to give special consideration to remote interventions. While conventional protocols may have to send at most one intervention per memory request, the active memory protocol may have to send multiple interventions whose addresses are different but mapped to the requested line. Therefore, the intervention reply handler must gather all the intervention replies before replying with the requested line.

3 Protocol Evaluation

This section discusses protocol memory overhead and protocol-related performance issues. Additional memory overhead in the active memory protocol stems from an increase in protocol handler code size, directory space overhead for re-mapped cache lines, and the space required to store mapping information in a table accessible by the embedded protocol processor. As an example of the increase in handler code size, the base protocol code size is 20KB, but adding the protocol extensions to support the active memory Matrix Transpose discussed in Section 2 yields a protocol code size of 33KB. Sparse Matrix Scatter/Gather, Linked List Linearization and Memory-side Merge have protocol code sizes of 24KB, 24KB, and 26KB respectively. The directory space overhead depends on the size of the re-mapped address space. Finally, the mapping table is only 128 bytes in size. This additional memory overhead is independent of the number of nodes in the system, except that the small mapping table must be replicated on every node of the system.

There are many performance issues for active memory protocols that do not impact conventional DSM protocols. We briefly discuss some of these here. One major potential performance bottleneck is the occupancy of the memory controller. To service a request for a particular cache line, the protocol may have to consult the directory entries of all the re-mapped cache lines (16 in the previous example) and may have to take different kinds of actions based on the directory states. Performing all these directory lookups in the software running on our programmable memory controller was too slow. Instead, our controller has a specialized pipelined hardware address calculation unit that computes 16 re-mapped cache line addresses, loads directory entries in special hardware registers and initiates any necessary data memory accesses. Our software protocol handlers tightly control this active memory data unit, striking a balance between flexibility and performance. As an example of the average controller occupancy, in a single-node system for 1, 2 and 4 processors for the normal executions controller occupancy is 7.7%, 21.3% and 43.2% of the total execution time respectively, while for the active memory executions it is 15.5%, 29.4% and 47.8%, though one must remember that the active memory execution time is smaller, and therefore occupancy percentages are naturally higher.

Another important performance issue is the behavior of the directory data cache, which is accessed only by the programmable embedded processor core on the memory controller. Since an access to one directory entry may necessitate accesses to multiple re-mapped directory entries (16 in the

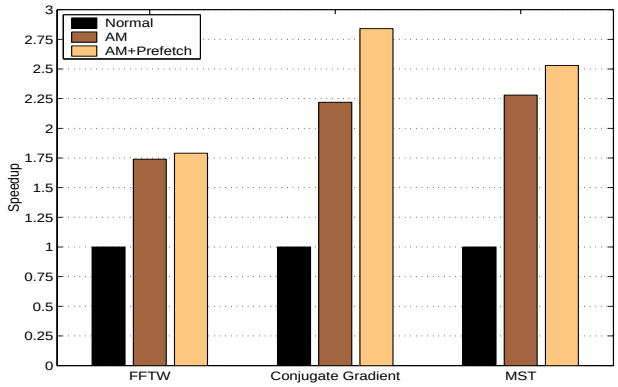

Figure 3. Active Memory Uniprocessor Speedup

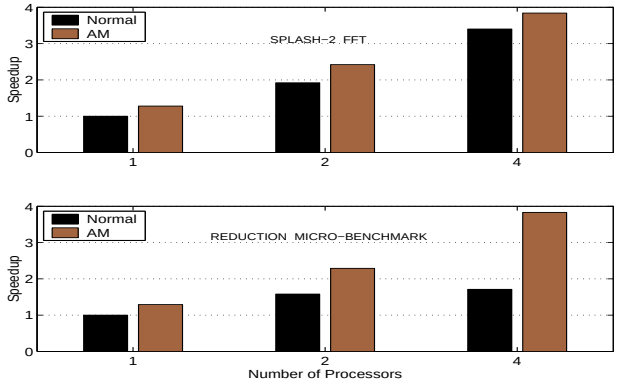

Figure 4. Active Memory Multiprocessor Speedup

above example) that may not correspond to directory entries for contiguous cache lines, the directory data cache may suffer from poor locality and hence large numbers of misses. The choice of byte-sized directory entries mitigates this problem in uniprocessor or single-node multiprocessor active memory systems. However, directory data cache performance may become an issue for extremely large problem sizes on small machines as the directory width increases for multi-node active memory systems.

Finally, an active memory protocol may have to send multiple interventions (a maximum of 16 in the above example) and every local intervention requires a data buffer that is filled by the processor-bus interface when the processor sends the intervention reply. This puts heavy pressure on the number of data buffers needed on the memory controller. We decided to keep four buffers reserved for this purpose and recycle them if a handler needs to send more than four interventions.

Table 1. L2 Cache Read Miss Count

App.	Normal	AM	Reduction Factor
FFTW	5644644	1421816	3.97
CG	13886869	3628477	3.83
MST	48582789	11829608	4.11

4 Simulation Results

We present representative simulated performance results for uniprocessor as well as single-node multiprocessor active memory systems in Figure 3 and Figure 4, respectively. FFT and FFTW use Matrix Transpose, Conjugate Gradient (CG) uses Sparse Matrix Scatter/Gather while Minimum Spanning Tree (MST) uses Linked List Linearization. The Reduction microbenchmark uses parallel reduction and shows the speedup for Memory-side Merge. Our simulator models contention in detail within the active memory controller, between the controller and its external interfaces, at main memory, and for the system bus. Further details on the simulation environment and the simulated applications can be found in [5]. In Figure 3 the "AM+Prefetch" bars correspond to the speedup achieved by our AM techniques along with exploiting new prefetching opportunities created by our AM optimizations (e.g. for Matrix Transpose it is now possible to prefetch re-mapped rows that were columns in the original matrix). Tables 1 and 2 show a comparison of L2 cache misses for Normal and non-prefetched AM executions, while Table 3 compares the data TLB miss penalty seen by the main processor for the two executions on a single processor corresponding to the results shown in Figure 3. All the tables show the reduction factor achieved by AM over normal execution for uniprocessor simulations.

For the parallel reduction microbenchmark (shown in Figure 4) the speedup of the normal application flattens out beyond two processors while the AM technique continues to achieve good speedup (3.83) for

Table 2. L2 Cache Write Miss Count

App.	Normal	AM	Reduction Factor
FFTW	5369071	1156683	4.64
CG	211323	136731	1.55
MST	12544	8947	1.40

Table 3. Data TLB Miss Penalty in a Million Processor Cycles

App.	Normal (% of t_{exec})	AM (% of t_{exec})	Reduction Factor
FFTW	721.51 (15.72%)	13.42 (0.51%)	53.76
CG	26.63 (0.46%)	12.56 (0.48%)	2.12
MST	838.58 (4.47%)	714.03 (14.22%)	1.17

a quad-processor node. This clearly shows that even for a quad-processor node the controller occupancy does not become a bottleneck. Both uniprocessor and multiprocessor results demonstrate the clear success of our coherence-leveraged active memory technique. Further, the multiprocessor speedup shows that our active memory protocols gracefully scale as the number of processors increases.

5 Conclusions

Active memory techniques, while improving application data access patterns, introduce a data coherence problem. Since the memory controller handles every cache miss and runs the coherence protocol, it has complete control over which memory lines can be cached by a particular processor in the system and in what state. Our approach enforces the mutual exclusion between the caching states of the remapped memory lines by naturally extending the conventional DSM coherence protocol, thereby efficiently solving the coherence problem. However, the design of active memory protocols raises some unique issues that are quite different in nature from a conventional DSM coherence protocol. The advantage of using software coherence protocols over hardware finite state machines is that the former can support new active memory techniques without changes to the memory controller hardware. This paper presents representative results on uniprocessors and single-node multiprocessors that confirm that our approach scales and performs well. Further, this protocol extension naturally lends itself to the research and development of multi-node active memory systems that we call Active Memory Clusters [3], which have the ability to attain hardware DSM performance on commodity clusters.

Acknowledgments

This research was supported by Cornell's Intelligent Information Systems Institute and NSF CAREER Award CCR-9984314.

References

[1] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, January 1999.

[2] M. Hall et al. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. *Supercomputing*, Portland, OR, Nov. 1999.

[3] M. Heinrich, E. Speight, and M. Chaudhuri. Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computing, Lecture Notes in Computer Science*, Springer-Verlag, May 2002.

[4] Y. Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. *International Conference on Computer Design*, October 1999.

[5] D. Kim, M. Chaudhuri, and M. Heinrich. Leveraging Cache Coherence in Active Memory Systems. In *Proceedings of the 16th ACM International Conference on Supercomputing*, New York City, June 2002.

[6] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[7] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.