Architectural Support for Uniprocessor and Multiprocessor Active Memory Systems

Daehyun Kim, Mainak Chaudhuri, *Student Member*, *IEEE*,
Mark Heinrich, *Member*, *IEEE*, and Evan Speight, *Member*, *IEEE*

Abstract—We introduce an architectural approach to improve memory system performance in both uniprocessor and multiprocessor systems. The architectural innovation is a flexible active memory controller backed by specialized cache coherence protocols that permit the transparent use of address remapping techniques. The resulting system shows significant performance improvement across a spectrum of machine configurations, from uniprocessors through single-node multiprocessors (SMPs) to distributed shared memory clusters (DSMs). Address remapping techniques exploit the data access patterns of applications to enhance their cache performance. However, they create coherence problems since the processor is allowed to refer to the same data via more than one address. While most active memory implementations require cache flushes, we present a new approach to solve the coherence problem. We leverage and extend the cache coherence protocol so that our techniques work transparently to efficiently support uniprocessor, SMP and DSM active memory systems. We detail the coherence protocol extensions to support our active memory techniques and present simulation results that show uniprocessor speedup from 1.3 to 7.6 on a range of applications and microbenchmarks. We also show remarkable performance improvement on small to medium-scale SMP and DSM multiprocessors, allowing some parallel applications to continue to scale long after their performance levels off on normal systems.

Index Terms—Active memory systems, address remapping, cache coherence protocol, distributed shared memory, flexible memory controller architecture.

1 Introduction

Active memory systems provide a promising approach to overcoming the memory wall [32] 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 or address remapping operations in the memory system to either offload computation directly or to reduce the number of processor cache misses. However, both approaches introduce data coherence problems either by allowing more than one processor in memory to access the same data or by accessing the same data via more than one address.

Most active memory system approaches [5], [13], [33] require the programmer to insert cache flushes before invoking active memory operations to avoid correctness problems. Cache flush overhead on modern processors can be large—typically requiring a trap to the operating system to execute a privileged instruction or set of instructions, and grows more costly as the number of cache levels increases. Though user-level cache flushes may reduce this overhead, either compilers must conservatively insert flushes to maintain correctness or inserting flushes requires human intervention. Further, this software cache-coherent programming model via flushes does not scale well to multiprocessor systems since it must flush *all* the caches in the system to guarantee correctness.

We propose an active memory system that leverages and extends the hardware cache coherence protocol. The key to the approach is that the memory controller not only performs the active memory operations required, but also runs the cache coherence protocol and, hence, solves the resulting data coherence problems. Our system does not require cache flushes, providing transparent active memory support without changing the programming model. Further, our unique approach naturally extends to multiprocessor active memory systems. We present a flexible active memory controller architecture that merges address remapping techniques and the directory-based cache coherence protocol. Although our focus is to support address remapping techniques, our approach does not preclude the future use of active memory elements as well. While many machines employ snoopy-based coherence mechanisms, recent architectures [8], [18] have abandoned bus-based snooping in favor of directories because of the decrease in local memory access time and the electrical advantages of point-to-point links between the processor and the memory controller. Finally, our techniques work with commodity microprocessors and memory technologies because we modify only the memory controller.

With the aid of our programmable active memory controller, augmented with specialized cache line assembly and disassembly hardware, we extend a conventional hardware cache coherence protocol to transparently support a number of address remapping techniques. We

- *D. Kim, M. Chaudhuri, and E. Speight are with the Computer Systems Laboratory, Cornell University, Ithaca, NY 14853.*
  E-mail: {daehyun, mainak, espeight}@csl.cornell.edu.
- *M. Heinrich is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816.*
  E-mail: heinrich@cs.ucf.edu.

Manuscript received 20 Nov. 2002; revised 24 Apr. 2003; accepted 28 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117800.

achieve significant performance improvement on a spectrum of machine configurations, from uniprocessors through SMP multiprocessors to DSM multinode systems. Detailed simulations of our active memory system show uniprocessor speedup from 1.3 to 7.6 for three address remapping techniques. In addition to uniprocessor speedup, our system also improves the performance of parallel applications on both single-node SMP and multinode DSM systems. Our system is from 1.2 to 1.8 times faster than a normal system on a 4-processor SMP and from 1.2 to 2.8 times faster on a 32-processor DSM configuration.

The rest of this paper is organized as follows: We discuss related work in Section 2. In Section 3, we present four address remapping techniques that our system supports and explain the coherence problem that arises with each. The flexibility of our active memory controller enables us to extend this set of techniques at any time without additional hardware cost. Section 4 describes the implementation of our active memory controller as well as the details of the coherence protocol extensions required. We present our simulation environment in Section 5 and analyze simulation results for a large class of important applications on uniprocessors, single-node multiprocessors, and multinode clusters in Section 6. Finally, we draw conclusions in Section 7.

2 Related Work

Previous work in active memory systems can be divided into projects with active memory elements and those with active memory controllers. While our active memory systems approach can support active memory elements, the focus of this paper is solely on our active memory controller.

The DIVA [5], Active Pages [26], and FlexRAM [13] projects all involve active memory elements—adding processing capability to memory chips, creating so-called PIMs. Both DIVA and FlexRAM have programming models that require cache flushes when communicating between the main processor and the active memory elements. Active Pages initially required cache flushes as well, but realized the critical role of coherence in active memory systems [14] at the same time we did [20], noting that coherence was a better mechanism than flushing for Active Pages. However, the Active Pages project examined coherence only as a mechanism for ensuring the active pages acted on the latest copy of the data and not in support of the address remapping techniques discussed here.

More closely related to this work is the Impulse memory controller [33], which is a hard-wired memory controller that supports a set of address remapping techniques to improve processor cache behavior. Impulse proposed two uniprocessor techniques, namely, matrix transpose and sparse matrix, and designed specialized hardware to accelerate cache line assembly and disassembly. However, unlike our active memory approach that leverages cache coherence, the Impulse programming model requires cache flushes. The necessity of using cache flushes also complicates the use of these techniques on multiprocessors. Though we use two of the same address remapping techniques in this work, we extend those techniques to SMP and DSM multiprocessors as well as uniprocessors. Further, we present a linked-list linearization technique and a parallel reduction technique that cannot be implemented on Impulse because they require coherence support and multiprocessor support, respectively. Our active memory controller shares similar cache line assembly/disassembly functionality with the Impulse memory controller, but our memory controller implements not only the address remapping techniques, but also the cache coherence mechanism, which imposes different requirements on our memory controller design. In addition, our memory controller takes advantage of its programmability, allowing us to support multiple active memory techniques and various machine configurations without hardware modification.

Our linked-list linearization technique performs repeated linked-list traversals that are similar to those used in memory forwarding [19], complete with the "safety net." However, while memory forwarding requires both main processor and memory controller modifications, our active memory controller can extend the cache coherence protocol and perform the linked-list linearization as well as implement the safety-net without modifying the main processor.

Our parallel reduction technique, initially proposed in a nonactive memory context in [7], triggers reduction operations on writebacks. The initial proposal used software flushes to guarantee data coherence between mapped address spaces. The parallel reduction technique also has similarity to techniques used in the ReVive [27] and Memory Sharing Predictor [17] proposals in the sense that these two also trigger checkpointing/logging-related operations and sharing predictions, respectively, when the home node receives certain types of coherence messages.

The main contribution of our active memory approach is that we leverage, integrate with, and extend the existing hardware cache coherence protocol. With this approach, our active memory controller supports address remapping techniques on uniprocessor as well as multiprocessor systems transparently and efficiently.

3 Active Memory Techniques

In this section, we explain four classes of active memory techniques discussed in this paper: *Matrix Transpose*, *Sparse Matrix*, *Linked-list Linearization*, and *Parallel Reduction*. We show how active memory techniques can improve performance by reducing cache misses, TLB misses, and, in some cases, processor busy time. We also discuss why a data coherence problem arises with each technique and explain how we solve the problem.

3.1 Matrix Transpose

Consider a matrix A stored in memory in row-major order (see Fig. 1). An application accesses the matrix A first in row-wise fashion and then in column-wise fashion. The size of the matrix A is $N \times N$ and the application is parallelized on P processors.

If a processor P_{id} wants to access the matrix A column-wise, it results in poor cache behavior because the matrix A is stored in row-major order. To improve cache performance, programmers typically use a *Tiled Transpose* technique, which the above example uses to implement

```
                                             /* Active Memory initialization phase */
                                             A' = AMInitTranspose(A, N, N, sizeof(Complex));
/* Row-wise access phase */                  /* Row-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1           for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1                             for j = 0 to N-1
      sum += A[i][j];                              sum += A[i][j];
BARRIER
Transpose(A, A');                            BARRIER
BARRIER
/* Column-wise access phase */               /* Column-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1           for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1                             for j = 0 to N-1
      sum += A'[i][j];                             sum += A'[i][j];
BARRIER
Transpose(A', A);                            BARRIER
BARRIER
```

Original Code **Active Memory Optimized Code**

Fig. 1. Example code: matrix transpose.

the `Transpose` routine. Before accessing the matrix A column-wise, we transpose the matrix A into a matrix A'. Then, instead of accessing the matrix A, we can access the matrix A' row-wise. Though tiling the transpose phase reduces the number of cache misses, this software transpose technique still has some overhead. Whenever we change the access pattern from row-wise to column-wise or vice versa, we need to perform the transpose phase, which costs processor busy time, memory access time, and synchronization time. The remote and local memory accesses during the transpose phase especially become a bottleneck. Our active memory technique eliminates the transpose phase and, hence, reduces this overhead.

Instead, our active memory controller provides a remapping transpose operation, *In-memory Transpose*. An address remapping technique [33] is used to map A^T to an additional physical address space A', called the *Shadow Space*. The shadow matrix A' is not backed by any real physical memory. Instead, it is composed by the memory controller on the fly, based on information such as the starting address and the dimension and element size of the matrix A provided via the one-time `AMInitTranspose` library call. In other words, the matrix transpose is carried out by the memory controller, not by the main processor, removing the software transpose overhead. Note that the initialization phase does not perform matrix transpose—it only communicates the information used to compose the shadow matrix A' to the memory controller via a short sequence of uncached writes.

This matrix transpose technique gives rise to a coherence problem between the normal matrix A and the shadow matrix A'. Any two corresponding elements of the matrix A and the matrix A' should be identical, yet the processor may be caching them at two separate locations. One way to ensure coherence is to guarantee that only one of the two spaces is cached at any time. We extend the coherence protocol and treat the access to the two spaces by the same processor in precisely the same way a coherence protocol treats accesses to the same cache line by different processors in a multiprocessor. When returning shadow space cache lines, we invalidate the corresponding normal space cache lines from the processor caches. When a processor next references these lines in the normal space, we have guaranteed that this access will cause a cache miss and our active memory controller can undo the previous remapping operation, returning the latest copy of the data to the processor and invalidating the corresponding shadow space cache lines cached by the processors. The details of the protocol are discussed in Section 4.2.

3.2 Sparse Matrix

In this technique, the central idea is to gather scattered data that the main processor wishes to access closely spaced in time and assemble them into cache lines. As an example, in Fig. 2, we show the basic loop of Sparse Matrix Vector Multiply, using the Compressed Row Storage representation for a sparse matrix A.

The scattered accesses to the dense vector v will experience cache misses if the vector v is large. To improve cache behavior, we remap the vector v to a shadow vector v' and, in the loop, replace $v[Acol[j]]$ with $v'[j]$. When the active memory controller sees accesses to the vector v', it calculates the index j, accesses the cache line containing $Acol[j]$, assembles the corresponding elements of $v[Acol[j]]$ into a cache line, and returns the cache line to the main processor. As a result, the main processor sees contiguous accesses to the vector v' and improved cache behavior. This technique also makes it possible to prefetch accesses to the vector v', which now exhibit good spatial locality.

The library call `AMInitSparse` communicates to the memory controller the necessary information to construct the vector v' such as the starting addresses of v and $Acol$, the

```
                                             /* Active Memory initialization phase */
                                             v' = AMInitSparse(v, Acol, NZ, sizeof(double));
/* Main phase */                             /* Main phase */
for i = id*(N/P) to (id+1)*(N/P)-1           for i = id*(N/P) to (id+1)*(N/P)-1
   for j = Arow[i] to Arow[i+1]-1               for j = Arow[i] to Arow[i+1]-1
      w[i] += A[j] * v[Acol[j]];                   w[i] += A[j] * v'[j];
```

Original Code **Active Memory Optimized Code**

Fig. 2. Example code: sparse matrix.

```
ptr = A;

/* Linked-list traversal phase */
while ptr != NULL
   sum += ptr->data;
   ptr = ptr->next;
```

Original Code

```
/* Active Memory initialization phase */
AMInitLinearize(NP, LS, sizeof(Node));

/* Linked-list linearization phase */
ptr = AMLinearize(A);

/* Linked-list traversal phase */
while ptr != NULL;
   sum += ptr->data;
   ptr = ptr->next;
```

Active Memory Optimized Code

Fig. 3. Example code: linked list linearization.

total number of nonzero elements (NZ) in the sparse matrix A and the element size of the vector v. A careful comparison of the active memory optimized code with the original code will reveal that this technique saves not only the cache misses of accessing the vector v, but also processor busy cycles by removing the address calculation for `Acol[j]`.

The coherence problem arises between the normal vector v and the shadow vector v'. The solution is similar to that for matrix transpose, though this particular technique prohibits writes to the shadow vector v' because a single cache line of the vector v' may contain the same element as the vector v more than once. Therefore, if the processor writes to one element, the other element (at a different position in the same cache line) would have a stale value. This restriction applies to any active memory implementation of this operation, whether using cache flushes or leveraging the coherence protocol. However, none of the sparse matrix applications that we have seen need writes to the shadow vector v'. Again, the details of the protocol are discussed in Section 4.2.

3.3 Linked-List Linearization

Searching, inserting, or deleting items in a linked-list may require walking through the list and these linked-list traversals can exhibit poor cache behavior. Consider the following example in Fig. 3 where an application traverses a linked-list A: If the nodes of the list A are scattered in memory, the traversal of the list can result in a large number of cache misses. Our active memory technique solves this problem by packing consecutive nodes of a linked-list into a contiguously allocated memory region in a dynamic fashion.

The `AMInitLinearize` library call sends the memory controller the information needed to linearize the linked-list A such as the byte offset of the next pointer within a node (NP), the number of nodes to linearize (LS), and the size of each node in the list A. The `AMLinearize` call to the active memory controller packs a certain number (LS in this example) of nodes in the list into a contiguous region, updating the "next" pointers (based on NP) in the list as it goes. The next time the processor traverses the list, it sees contiguous memory accesses and, hence, improved cache behavior. Note that, after linearizing the list, it is possible to easily prefetch consecutive nodes of the list, which is difficult in the random linked-list structure of the original list.

Linearizing linked-lists can be done in software without the use of active memory systems. However, a correctness problem arises if, after linearization, the processor dereferences a dangling pointer that points into the "old" linked-list. Such a reference may now return stale data. Our solution to this problem is much like that of memory forwarding [19], except we can perform this optimization without processor modifications. Here, the coherence protocol implements a *safety net* by invalidating the original cache lines during the copying phase. If the processor accesses a dangling pointer, it is guaranteed to be a cache miss and can therefore be handled correctly by the active memory controller. There are some limitations of this technique such as safety net overhead and potential pointer comparison problems [19], but it is still a powerful technique that shows large benefits in many applications. The detailed discussion of the protocol is in Section 4.2.

3.4 Parallel Reduction

Parallel Reduction maps a set of elements to a single element with some underlying operation. Consider the example in Fig. 4 of reducing every column of a matrix A to a single element, thereby obtaining a single vector x at the end of the computation. The size of the matrix A is $N \times N$ and there are P processors. Processor P_0 initializes the vector x (not shown). The value e is the identity element under the operation $\otimes$ (e.g., 0 is the identity for addition and 1 is the identity for multiplication).

The matrix A is distributed row-wise as suggested by the computational decomposition in the code (i.e., the first N/P rows are placed on P_0, the next N/P rows on P_1, etc.) and $private\_x$ of each processor is placed in the local memory of that processor. Thus, the reduction phase does not have any remote memory accesses. However, the merge phase assigns mutually exclusive index sets of the result vector to each processor and, hence, every processor suffers from $(1 - \frac{1}{P})$ portion of remote misses while accessing the $private\_x$ of other processors. This communication pattern is inherently all-to-all and does not scale well. Prefetching may improve performance to some extent, but the remote read misses remain in the critical path, influencing overall performance. Our active memory technique eliminates these remote read misses by completely removing the merge phase.

Our active memory controller maps the result vector x to a shadow vector x' in the initialization phase. The processors perform the reduction phase only to the vector x' and the merge phase is removed from the main code. In our system, the merge operations are performed by the memory controller, not by the main processors. When each cache line of the vector x' is written back to memory, the memory controller performs the merge operation [7]. Therefore, the active memory technique can save processor busy time by eliminating the merge phase and remote

Original Code

```
/* Privatized reduction phase */
for j = 0 to N-1
   private_x[id][j] = e;
   for i = id*(N/P) to (id+1)*(N/P)-1
      private_x[id][j] =
             private_x[id][j] ⊗ A[i][j];
BARRIER
/* Merge phase */
for j = id*(N/P) to (id+1)*(N/P)-1
   for i = 0 to P-1
      x[j] = x[j] ⊗ private_x[i][j];
BARRIER
Subsequent uses of x
```

Active Memory Optimized Code

```
/* Active Memory initialization phase */
x' = AMInitReduction(x, N, sizeof(long long));
/* Reduction phase */
for j = 0 to N-1
   for i = id*(N/P) to (id+1)*(N/P)-1
      x'[j] = x'[j] ⊗ A[i][j];

BARRIER

Subsequent uses of x
```

Fig. 4. Example code: parallel reduction.

memory access time since the writebacks are not in the critical path of execution.

This technique requires coherence support since the application may use the result vector x before all the cache lines of the shadow vector x' are written back, leading to an incomplete merge operation. We solve the problem by extending the cache coherence protocol to keep the normal vector x and the shadow vector x' coherent. The protocol is discussed separately in Section 4.2.

3.5 Programming Model

Our active memory system provides a simple programming model. Before using an active memory technique, user applications only need to call the corresponding AMInit library provided, which communicates necessary information to the active memory controller and initializes its address remapping table. After this library call, user applications can use any of the normal or shadow data structures (described in the previous sections) without concern for coherence. At the end, user applications call the AMUninstall library to free the remapping table entry (not shown in the example codes for brevity).

Currently, our active memory system supports four different address remapping techniques, thus providing four types of AMInit and AMUninstall library functions. These techniques optimize popular computation kernels used in important applications. In addition, the programmability of our memory controller allows us to expand the set of techniques without any additional hardware cost. We only need to supply the new coherence extensions necessary as well as the associated AMInit and AMUninstall functions. The details of the active memory protocols are discussed in Section 4.2.

4 Implementation

This section details the implementation of various components of our active memory system. We discuss the memory controller architecture that is the heart of the whole system and our extended cache coherence protocols tailored to execute the active memory techniques.

4.1 Active Memory Controller

The design goal of our active memory controller is to provide flexibility in the types of active memory operations without sacrificing performance or changing the programming model. We achieve these goals by augmenting a programmable core, called the *Active Memory Processor Unit (AMPU)* with specialized hardware, called the *Active Memory Data Unit (AMDU)*. The AMPU runs software protocol handlers to implement cache coherence and control the correctness of active memory operations. The AMDU accelerates cache line assembly and disassembly, which form the datapath core of active memory techniques. By dividing our protocol execution into control and data paths (similar to the approach in [16]) and by executing them concurrently, we simultaneously achieve flexibility and performance.

4.1.1 Microarchitecture

Fig. 5 shows the microarchitecture of our active memory controller. The control flow of a memory request is divided into three stages (Dispatch, AMPU/AMDU, and Send) that operate concurrently on different requests, thereby forming a macro-pipeline. Requests arriving from the processor or the network are scheduled by the Dispatch Unit. Here, the requests get divided into header and data components, which the AMPU and the AMDU process concurrently. For active memory operations, the AMDU assembles or disassembles the cache line under the control of the AMPU that triggers necessary coherence actions. Finally, the Send Unit returns

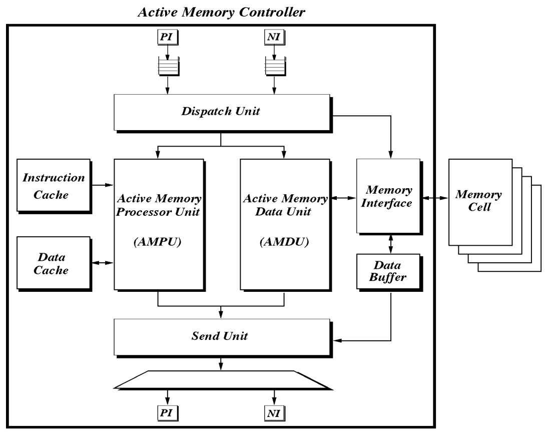


Fig. 5. Active memory controller microarchitecture.

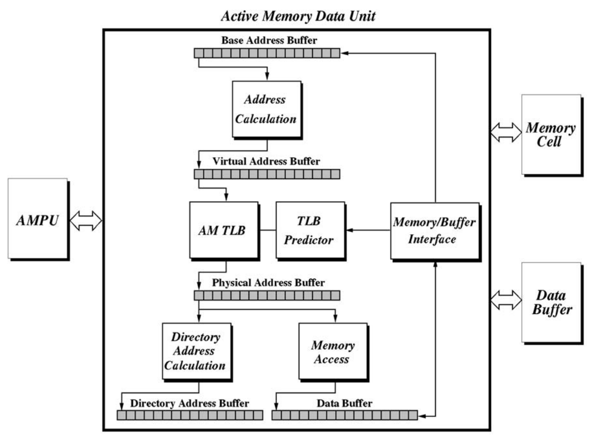


Fig. 6. Active memory data unit.

the cache line to the requester if necessary. In the remainder of this section, we describe each unit in detail.

Dispatch Unit. The Dispatch Unit schedules requests from the processor interface (PI) or network interface (NI) and initializes the AMPU and AMDU based on the address space (e.g., remapped or normal) and the type of the request.

Active Memory Processor Unit (AMPU). The AMPU is a statically scheduled, pipelined, dual-issue programmable MIPS core that executes the coherence protocol—the control portion of an active memory operation or a conventional cache miss. It does not support virtual memory, exceptions, floating-point arithmetic, or integer multiplication and division. However, it includes specialized instructions to enhance common cache coherence protocol and active memory operations. The AMPU gets its code and data from on-chip instruction and data caches, respectively. Both caches are backed by main memory.

For each memory request (normal or active), the AMPU executes the corresponding protocol handler. It checks and updates directory entries to preserve cache coherence and sends appropriate control messages to the AMDU to perform active memory operations on data. The latency of the handler is critical to overall performance. For high performance, the handler latency should be less than that of the AMDU so that it can be completely hidden by the data transfer time. In practice, we find that this is the case.

Active Memory Data Unit (AMDU). The AMDU (see Fig. 6) is a specialized hardware datapath that performs pipelined address remapping and accelerates cache line assembly/disassembly. For each cache line, it loads/stores 16 different double words (a cache line) from/to the main memory according to the addresses it generates each cycle. We follow a similar design to [33]; however, because our cache coherence mechanism demands special operations from the AMDU, it exhibits quite different behavior, as discussed in Section 4.2.

The AMDU is composed of five cache line-sized buffers: the *Base Address Buffer*, *Virtual Address Buffer*, *Physical Address Buffer*, *Directory Address Buffer*, and *Data Buffer*, and three pipeline stages: *Address Calculation*, *AMTLB Lookup*, and *Directory Address Calculation/Memory Access*. Each pipeline stage processes one double word at a time from the previous buffer and writes its result to the next buffer. Operations are fully pipelined, so one double word is processed per cycle in the absence of any pipeline stalls. Therefore, the best-case latency for the AMDU to process a cache line (16 double words) is the pipeline latency + 15 cycles, where the pipeline latency is the time it takes one double word to pass through all three stages without stalls.

The Base Address Buffer contains technique-specific values that are intended to be used for virtual address calculation. For example, the sparse matrix technique uses this buffer to store $Acol[j]$ values corresponding to the requested shadow cache line. The Address Calculation stage calculates virtual addresses from the Base Address Buffer by shift and add operations and writes to the Virtual Address Buffer. Each entry of the Virtual Address Buffer holds the virtual address of the corresponding double word. The AMTLB translates the virtual addresses to physical addresses and then the Memory Access stage performs double word load/store operations to/from the corresponding entries of the Data Buffer. The Directory Address Calculation unit helps the AMPU calculate directory entry addresses. The coherence protocol requires the AMPU to check the directory entries of the cache lines containing the double words, so the address calculation is performance-critical. By moving the address calculation to the AMDU, the latency of the AMPU is significantly reduced and, because it operates concurrently with the Memory Access stage, it does not slow down the AMDU.

Active memory techniques directly manipulate application data that cannot be accessed through physical addresses. For example, linked-list linearization traverses a list by chasing virtual addresses. The memory system is addressed with physical addresses, so the AMDU has a TLB that we call the AMTLB. Because an AMTLB miss stalls the AMDU pipeline and has a large miss penalty, the hit rate of the AMTLB is a critical determinant of performance. Therefore, our AMDU also has an AMTLB predictor to improve its performance. The AMTLB predictor prefetches the accesses to the AMTLB. It is a *Differential Finite Context Method* predictor [10] that consists of a 3 KB table and control logic. Detailed analysis can be found in [15]. It improves performance by more than 15 percent. Finally, if the AMTLB suffers a page fault, a trap is made to the kernel and the page fault handler is initiated.

Send Unit. The Send Unit is responsible for the mechanics of sending interventions or reply messages generated by the coherence protocol. The Send Unit assembles the header and the data components into a message and inserts it into the corresponding output queue (PI or NI), offloading this task from the AMPU.

Memory Interface. The Memory Interface connects the SDRAM to the other parts of the controller. It picks a request from a 16-deep request queue and performs loads or stores. It is fully pipelined and the AMDU does not stall unless the memory request queue fills.

4.1.2 Network Integration

Commodity architectures are witnessing evolutionary changes in network integration as the network connection moves from a plug-in card on the distant I/O bus to a routing chip directly connected to or integrated with the

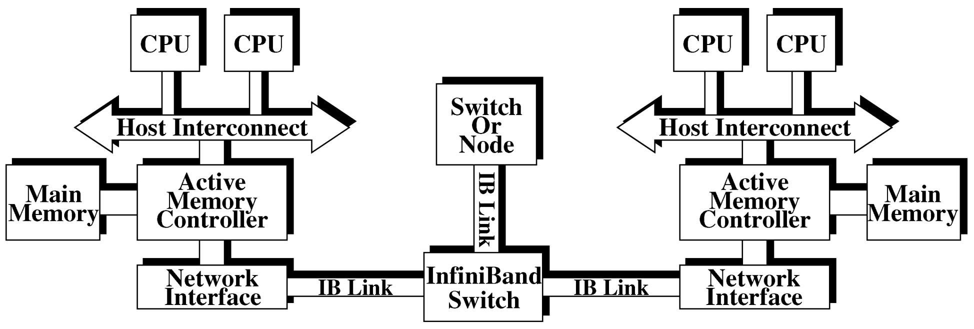


Fig. 7. Active memory cluster configuration.

memory controller (Fig. 7). Our active memory controller design uses a system area network (SAN), such as InfiniBand [12] (shown in Fig. 7) or PCI Express [30] (formerly 3GIO), as its network interface. However, the active memory controller does not use the high-level (and higher overhead) user-level protocols that run over the SAN network link when sending and receiving the messages that comprise the coherence protocol. Instead, our controller uses the fast underlying link-level performance to synthesize and handle messages to support DSM. Coherence messages travel between memory controllers only and are not forwarded up to the processor for handling via interrupts. Instead, our system handles the messages entirely in the memory controller, similar to a hardware DSM machine. The "normal" (not using active memory support) parallel performance of this *Active Memory Cluster (AMC)* configuration is detailed in [11].

4.2 Cache Coherence Protocol Support

Starting from a base protocol capable of running normal (nonactive) applications, we add support for each of the active memory techniques. In this section, we discuss the salient features of the base protocol and the extended active memory protocols. The discussion on matrix transpose and parallel reduction focuses on multinode clusters, while the explanation of sparse matrix and linked-list linearization is centered around single-node systems.

4.2.1 Base Protocol

The base coherence protocol can correctly execute any normal shared memory application. We implement two versions of our base protocol, one particularly optimized for single-node SMP systems and a similar but more scalable protocol for multinode clusters.

For single-node systems—uniprocessor and SMP multiprocessors—our base protocol is a conventional invalidation-based MSI bitvector directory protocol running under release consistency. For all normal 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-way SMP nodes. These four bits store a sharer 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 (AM and dirty-exclusive) and two remaining bits are left unused.

For multinode DSM clusters, the base protocol is an MSI write-invalidate bitvector protocol similar to that in the SGI Origin 1400 [18]. The directory entry is 64-bits wide with five state bits (AM, pending shared, pending dirty-exclusive, dirty, and local). The AM bit is used only by our active memory protocol extensions. The pending states are used to mark the directory entry busy when requests are forwarded by the home node to the current exclusive owner. The dirty bit is set when a memory line is cached by one processor in the exclusive state. The local bit indicates whether the local processor caches the line and is used to quickly decide whether an invalidation or intervention needs to go over the network interface. The sharer vector is 32 bits wide. The remaining bits are left unused for future extensions of the protocol. As in the Origin protocol, our protocol collects invalidation acknowledgments at the requester. However, we support eager-exclusive replies where a write reply is immediately sent to the processor before all invalidation acknowledgments are collected. Our relaxed consistency model guarantees "global completion" of all writes on release boundaries, thereby preserving the semantics of flags, locks, and barriers.

4.2.2 Matrix Transpose Protocol Extensions

In this section, we describe the implementation of the matrix transpose protocol extensions. Consider the example in Section 3.1. A is an $N \times N$ original matrix and A' is a shadow matrix address remapped to the matrix A. Processor P_{id} accesses the matrix A if it wants to access data row-wise or the matrix A' if the access pattern is column-wise.

Every memory request is forwarded to the home node and processed by the home node memory controller. The memory controller first consults the directory entry. A cache line can be in one of eight possible states—unowned, shared, dirty, pending shared, pending dirty-exclusive, AM, AM pending shared, or AM pending dirty-exclusive. These states are divided into two groups, normal states (unowned, shared, dirty, pending shared, pending dirty-exclusive) and AM states (AM, AM pending shared, AM pending dirty-exclusive). If a cache line is in a normal state, the meaning of the state is the same as that in the base protocol. If a cache line is in an AM state, it means that the remapped address space is being used. (e.g., if the requested cache line is in the original matrix A and it is in an AM state, the corresponding remapped cache lines in the shadow matrix A' are being cached.)

If the requested cache line from A is in any of the normal states, our memory controller executes the base protocol. Note that, for normal memory operations, our active memory protocol has only the additional overhead of checking whether the AM bit is set or not. This does not slow down conventional applications since this check is not on the critical path [11], [15].

If the requested cache line is in the AM state, there is a potential data coherence problem. Because remapped cache lines in the shadow address space are already being cached, if we simply reply with the requested cache line, it may result in data inconsistency. To solve this problem, our protocol enforces mutual exclusion between the normal and shadow address spaces. First, we set the requested cache line in the pending state so that subsequent requests to the line will be refused until the current request completes. Based on information like matrix dimension and element size provided by the one-time `AMInitTranspose` call, we calculate each remapped address and consult its directory entry. If it is in the dirty-exclusive state, we send an intervention to its owner and retrieve the most recent copy of the data. If it is in the shared state, we send invalidation requests to all the sharers and gather the acknowledgments. After we invalidate all the remapped cache lines, we can safely reply with the requested cache line. Finally, we update the directory entries of the requested cache line and the remapped cache lines. The requested cache line is set in the shared or dirty state based on the original request type and the remapped cache lines are placed in the AM state to guarantee future data coherence. During the entire procedure, if we encounter any cache line in the pending state, we send a negative acknowledgment to the requester. Our protocol ensures that the retry will eventually succeed.

The in-memory matrix transpose operation takes place in two cases: when a processor requests a shadow cache line or when a shadow cache line is written back to memory. First, for a shadow cache line request to A', our active memory controller gathers data elements from the original normal space A to assemble the requested shadow cache line. The address calculations are accelerated by specialized hardware (the AMDU in Fig. 6) and the gather operation eliminates the software transpose overhead from the main processor. Second, when a shadow cache line is written back to memory, the memory controller disassembles the shadow cache line and writes the data elements back to the original space. This operation has significant performance benefit in multinode systems. By writing back data elements from the shadow space to the normal space, it saves remote memory accesses. The next time a processor accesses the data elements in the normal space, it does not need to retrieve the possibly dirty and remotely cached shadow elements. However, the benefit depends on the percentage of shadow cache lines that are timely written back.

We have also designed an alternative protocol, called *Speculative Matrix Transpose*, that improves performance even further. Instead of carrying out the transpose operation when a shadow cache line is requested, we can perform the transpose when a normal cache line is written back and speculatively "push" the transposed cache line to the local memory of the next predicted consumer. If the speculation is correct, our technique will enjoy a reduction in remote memory accesses for the same reasons as the writeback transpose case above. Note that a read operation is in the critical path of execution, while a writeback operation is not.

4.2.3 Sparse Matrix Protocol Extensions

From the discussion of the matrix transpose protocol, it is clear that the matrix transpose technique is a kind of scatter/gather operation where the scatter/gather pattern is extremely regular. The sparse matrix technique can be seen as a generalized scatter/gather operation where the mapping function is arbitrary. For instance, in the example in Section 3.2, the mapping function is stored in the vector $Acol$. The sparse matrix protocol is similar to the matrix transpose protocol and we identify only distinguishing features of this protocol below.

This technique needs to maintain the mapping between $v[Acol[j]]$ and $v'[j]$ as the matrix transpose technique keeps the mapping between $A[i][j]$ and $A'[j][i]$. For a shadow cache line request to $v'[j]$, the memory controller first computes the index j using the starting address of v' and the size of data element passed by the `AMInitSparse` library call. This can be done since the shadow space is contiguous. The controller then loads the corresponding 16 consecutive values from $Acol$ into the Base Address buffer. These values are used to assemble the requested shadow cache line containing the 16 values of $v[Acol[j]]$. The memory controller consults the directory entry for the corresponding portion of $Acol$. If the most recent data of $Acol$ are in a processor cache, an intervention must be sent to get the correct values. Otherwise, the data may be simply copied from memory to the Base Address buffer. Finally, the virtual addresses of the requested elements of $v[Acol[j]]$ can be calculated using add and shift operations with the starting virtual address of v and the size of each data element. The corresponding physical addresses are produced by the AMTLB. After calculating the mapping, the execution of the protocol is identical to the matrix transpose protocol.

To enforce mutual exclusion between $v'[j]$ and $v[Acol[j]]$, when the memory controller gets a request to v', it invalidates the remapped cache lines of v before it replies to the request. The memory controller checks the directory entries of remapped cache lines, then sends interventions for dirty lines and invalidates shared lines. When it returns the requested cache line of v', no remapped cache lines of v reside in the processor caches.

We found that if the protocol enforces mutual exclusion between the corresponding lines in v and v', some applications may suffer from cache thrashing. In those applications, the access pattern is not migratory from the normal space to the shadow space or vice versa. Instead, processors access both spaces (i.e., v and v') at the same time, so the mutual exclusion results in false sharing. In a variation on our AM protocol, we relax the mutual exclusion. We allow the normal and the shadow spaces to be cached at the same time if neither is in the dirty state. If a processor tries to write one space, an upgrade request is sent to the memory controller, which then invalidates the other space. Note that, in this new protocol, the mutual exclusion is imposed only on write requests, not on read requests. This illustrates another advantage of our flexible memory controller in that it can adapt the coherence protocol to the needs of applications.

As mentioned in Section 3.2, this protocol has one limitation: It does not allow writes to the shadow vector v'. If two elements in a shadow cache line (at different positions, but in the same line) are mapped to a single element in a normal cache line and a processor writes to one of the elements, the other element will become stale. This restriction applies to any active memory system that does not modify the processor cache architecture. However, writes to v' may be introduced only if the application writes to v in a sparse manner and, in our experienc,e sparse matrix applications do not need sparse writes to the vector v.

4.2.4 Linked-List Linearization Protocol Extensions

The linked-list linearization protocol is composed of two operations—*Linearization* and *Safety-Net*. Linearization copies a linked-list into a contiguous memory space and safety-net guarantees correctness. Consider the example given in Section 3.3, where we traverse a linked-list A.

This technique provides a library call `AMLinearize` to user applications. On a linearize call to the linked-list A, the memory controller linearizes the list—it copies a certain number of nodes of the list into a contiguous physical address space. First, the linearize call communicates to the memory controller the starting virtual address of the list A, which the AMTLB translates into a physical address. Then, the memory controller checks the directory entry of the corresponding cache line. It retrieves a dirty cache line from the processor cache and invalidates any sharers, then copies data into a contiguous memory space provided by the operating system. It also updates the "next" pointer of the node to point to the new space. Now that the first node of the list is copied, the memory controller chases the next pointer to get the virtual address of the next element and repeats the same procedure above. The memory controller finishes when it reaches the end of the list or it has linearized a predefined number of nodes. It returns the starting address of the newly linearized list and the application traverses the new list instead of the list A. This linearize call can be implemented in software without any active memory support. However, our technique can achieve higher performance since our memory controller is closer to main memory and can manipulate data with word granularity. The main processor pays longer latency to access memory and must transfer entire cache lines.

More seriously, without active memory support, a correctness problem arises if a processor accesses a dangling pointer that points into the "old" linked-list. Such a reference may return stale data. Our solution is the safety-net operation that keeps data coherent between the original list and the linearized lists. We establish a mapping between the original and linearized lists and enforce mutual exclusion so that only one of the remapped elements can be cached at a time. First, we make a mapping by the "memory forwarding" chain. When the memory controller performs a linearize call, it stores pointers to the newly linearized list in the memory space of the old list. So, each node of the old list now contains a pointer to the corresponding node of the new list, instead of a data value. It also sets the AM bits of the cache lines that contain any nodes of the old list. Second, if a processor accesses a dangling pointer, the AM bit in the directory entry of the requested cache line must be set, indicating that some of the nodes in the cache line are linearized and therefore invalid. The memory controller finds these old nodes, reads the pointers to the new nodes, and follows the memory forwarding chain to the end. Note that multiple linearize calls may be performed on a single linked-list and we keep the valid data nodes (not the pointer nodes) at the end of the chain. For those valid nodes, the memory controller retrieves the most recent data through the same procedures as before—it consults the directory entries and sends interventions for dirty cache lines and invalidations for shared lines. Finally, the memory controller gathers all the valid data and returns the requested cache line. The memory controller also needs to update the memory forwarding chain and the directory entries to guarantee future correctness. It puts the requested cache line at the end of the forwarding chain and clears the AM bit. The AM bits of other cache lines in the same forward chain are also updated accordingly.

4.2.5 Parallel Reduction Protocol Extensions

As in the parallel reduction example in Section 3.4, in the first phase of execution, every processor reads and writes the shadow cache lines of x'. When a processor reads a shadow cache line in the shared state, the local memory controller immediately replies with a cache line filled with identity values e. It does not notify the home node because the algorithm guarantees that every shadow cache line will be written eventually. If the processor wishes to write a shadow cache line, the local memory controller still replies immediately with a cache line filled with values e. The main processor receives the cache line, but the write does not complete globally until all address remapped memory lines are invalidated from other caches and all the necessary acknowledgments are gathered. To do this, the local memory controller also forwards the write request to the home node. The home node memory controller consults the directory entries of the requested x' cache line as well as the corresponding x cache line. The protocol execution observes one of the following four cases.

The first case occurs when the corresponding x cache line is in the dirty state. The home node notifies the requester that the number of acknowledgments is one, sends an intervention to the owner, and sets the shadow directory entry in the pending exclusive state to indicate that the first shadow request for this cache line has been received and the intervention has been sent. Later, after the dirty line is retrieved and written back to memory, the home node sends an acknowledgment to every requester marked in the shadow directory entry and clears the pending bit. Only after the requesters receive the acknowledgments does the corresponding write complete.

The second possibility is that the x cache line is in the shared state. The home node replies with the number of sharers to the requester and sends out invalidation requests. The sharers send their acknowledgments directly with the requester.

The third case arises when the requested x' cache line is in the pending exclusive state—the first case above describes why and when the directory entry transitions to this state. In this case, the home node notifies the requester that the number of acknowledgments is one.

In the last case, the directory entries of both the x and x' cache lines are clean. The home node notifies the requester that the expected number of acknowledgments is zero. In all the cases, the home node marks the requester in the shadow directory entry and sets the AM bit in the normal directory entry.

The merge phase takes place at the home node when it receives a writeback for a shadow cache line. The home node clears the source node of the writeback from the shadow directory entry, performs the reduction operation, and writes the result back to memory. The last writeback clears the AM bit in the normal directory entry. At this point, the corresponding x cache line in memory holds the most recent value.

Finally, we discuss the case when a memory request arrives for a normal cache line of x. If the AM bit in the corresponding directory entry is clear, the behavior of our active memory controller is identical to the base protocol. However, if the AM bit is set, it means that the corresponding shadow cache line is cached in the dirty-exclusive state by one or more processors. Note that, in this protocol, there are only two stable states for a shadow cache line, namely, invalid and dirty-exclusive. Note also that, from the protocol execution discussed above, it is clear that the same shadow cache line can have simultaneous multiple writers. To satisfy the request for the x cache line, the home node sends out interventions by reading the owners from the shadow directory entry and keeping the normal directory entry in the appropriate pending state (e.g., shared or dirty-exclusive) until the last intervention reply arrives. Every intervention reply arrives at the home node and clears the source node from the shadow directory. At this time, the home node also carries out the reduction between the intervention reply and the resident memory line. The final intervention reply triggers the data reply carrying the requested x cache line to the requester.

4.2.6 Protocol Design Issues

In the following, we summarize a few implementation-specific issues in active memory protocol design. Our discussion touches on four topics, handler latency, shadow page placement, cache line invalidation, and deadlock avoidance. The latter three issues are particular to multi-node active memory systems.

The most important performance issue in our protocol design is the path length of the protocol handlers. Handler latency (the execution time of the software protocol handler) directly affects memory system performance. In our active memory controller, the AMPU and the AMDU work concurrently. The AMPU executes the protocol handlers, while the AMDU accesses main memory. In the ideal case, the handler latency is completely hidden by the memory access time, which is the case for all normal (nonactive) memory accesses. However, the active memory protocol handlers are longer since they do more work (note that this is justified because we are saving future cache misses). To optimize our active memory handlers, we implement some common functionality in hardware. For example, the AMDU calculates the directory entry

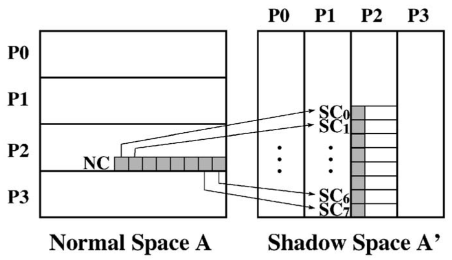


Fig. 8. Page placement for matrix transpose.

addresses for the AMPU, which it uses in every active memory protocol. We also adopt software optimization techniques like loop unrolling in our protocol handlers, which results in better code scheduling and fewer branches in the dual-issue AMPU.

Shadow page placement is a problem unique to multi-node active memory systems. Consider a memory snapshot from the matrix transpose protocol execution in Fig. 8. While servicing a request for a normal cache line NC, the home node of NC needs to consult the directory entries of the corresponding shadow cache lines, $SC_0, \ldots, SC_7$. A naive shadow page placement would necessitate network transactions to look up the shadow directory entries and this overhead can be a serious bottleneck. We solve this problem by placing the normal space and the corresponding shadow space on the same node. Fig. 8 shows an example of a four node system. The normal space is partitioned row-wise and the shadow space is partitioned column-wise. This page placement guarantees that, for any normal cache line request, the home node memory controller can locate all the directory entries of the corresponding shadow cache lines on the same node. In addition, the same is true for shadow cache line requests. Note how the underlying active memory operation (e.g., transpose) dictates a column-wise partitioning of the shadow space. We adopt the same method for the parallel reduction technique, although there the address remapping is one-to-one and both the spaces are partitioned row-wise.

The next design issue we discuss is related to invalidation messages. Since our system collects invalidation acknowledgments at the requester, the acknowledgment message typically carries the address of the requested cache line (i.e., the cache line that is being written). However in our active memory protocols, the remapped shadow cache lines have different addresses from the corresponding normal cache line addresses. So, the invalidation addresses can be different from the requested addresses. If a normal cache line request invalidates one or more shadow cache lines, a problem will arise at the requester while gathering the invalidation acknowledgments. The same problem will also happen for shadow cache line requests. Note that the invalidation acknowledgments are directly sent by the invalidating node to the requester and they carry the address of the invalidated line, not the address of the originally requested line. In normal systems, these two addresses are the same because one cache line corresponds to a system-wide unique physical address. We propose two solutions to this problem. The first solution is to pack two addresses (the invalidation address and the requested

TABLE 1
Applications and Problem Sizes

Applications	Problem Sizes
SPLASH-2 FFT	1M points
FFTW	8K×16×16 matrix
Transpose	1K×1K matrix
Conjugate Gradient	8K×8K matrix, 256K non-zeros
VecMult	8K×8K matrix, 256K non-zeros, 10 iterations
DDOTI	256K-element vector, 8K non-zeros, 10 iterations
SMVM	64K×64K matrix, 2M non-zeros, 50 iterations
Health	6 level tree, 4 children per node
MST	2K-node graph
Traverse	256 lists, 1K elements per list
DenseMMM	256×256 matrix
Spark98Kernel	64K×64K matrix, 1M non-zeros
SparseFlow	512K nodes and 1M edges
MSA	64×128K matrix

address) in the header of the invalidation request message from the home node to the invalidating node so that the invalidating node can set the requested address in the acknowledgment header. The second solution is to carry out an address remapping operation again at the invalidating node to compute the corresponding requested address. The flexibility of our design allows us to employ the second solution and therefore does not require changes to the network message header structure.

Finally, the possibility of generating multiple interventions from a single request in active memory protocols has ramifications on the deadlock avoidance strategy. Also, unlike the baseline protocol, many of the active memory protocol handlers require more than one data buffer for transferring cache-line-sized data to and from the memory system, requiring careful data buffer management. However, we solve both problems similarly to conventional DSM systems—a handler running out of any necessary resource suspends execution and reschedules itself at a later point in time instead of waiting for that resource. We enqueue the incomplete request to a reserved portion of main memory at the home node and dispatch it when appropriate resources are available. This ensures forward progress by allowing the memory controller to handle other outstanding requests and break deadlock cycles. Note that our protocols use three or four virtual lanes, obviating the need for complicated back-off mechanisms like those implemented in the SGI Origin 1400 protocol.

5 Simulation Methodology

In this section, we discuss the applications we use to evaluate the performance of our active memory system and the simulation environment we use to collect the results.

5.1 Applications

To evaluate our active memory system we use a range of applications—some are well-known benchmarks while others are microbenchmarks written to exhibit the potential of a particular active memory technique. In Table 1, we summarize the applications and the problem sizes we use in simulation.

We use FFT from SPLASH-2 [31], FFTW [6], and a microbenchmark called Transpose to evaluate the performance of the matrix transpose technique. The microbenchmark reads and writes to a matrix and its transpose, making it highly memory-bound. The parallelized versions of all these applications are optimized with tiling and padding for nonactive memory executions. Tiling is used to reduce cache misses, especially remote cache misses, during the transpose phase. Padding is used to reduce conflict misses in the cache. Without these optimizations, active memory techniques result in an even larger speedup than that presented in Section 6.

As sparse matrix applications, we use Conjugate Gradient from the DIS (Data-Intensive Systems) benchmark suite [3], VecMult from level-2 Sparse BLAS [23], DDOTI from Sparse BLAS [22], and a microbenchmark called SMVM. Conjugate Gradient uses the sparse matrix vector product in the main iteration loop. VecMult executes the level-2 Sparse BLAS routine `CSR_VecMult_CaABbC_double`. DDOTI carries out the inner product between a dense vector and a sparse vector stored in standard indexed format. Finally, the SMVM microbenchmark carries out the sparse matrix vector multiplication kernel. Conjugate Gradient, VecMult, and SMVM use the standard Compressed Row Storage to represent the sparse matrices.

Linked-list linearization is evaluated by running Health and MST from the Olden benchmarks [1] and a microbenchmark called Traverse. Health has a tree structure where each node contains five linked-lists. We linearize the longest two of them, but the other three can also be optimized for improved performance. MST manages a hash table, each entry of which is implemented as a linked-list that is optimized by our linked-list linearization technique. The

Traverse microbenchmark walks through an array of lists as new elements are inserted. The length of each list increases to a maximum of 1,024 nodes. We linearize the lists periodically as every 32 nodes are inserted.

To evaluate parallel reduction we use the dense matrix multiplication (Dense MMM) kernel, a modified Spark98 kernel [24], [25], a microbenchmark called SparseFlow, and a microbenchmark called Mean Square Average (MSA). Dense MMM carries out the computation $C = A^T B$ on square matrices A and B, which is a special case of the level-3 BLAS [4] matrix-matrix multiplication subroutine. The modified Spark98 kernel parallelizes one call to LocalSMVP. SparseFlow computes a function on the in-flow of every edge incident on a node and sums up the function outputs as the net in-flux at each node in a sparse multisource flow graph. MSA calculates the arithmetic mean of squares of the elements in every column of a matrix. All four applications use addition as the underlying reduction operation.

5.2 Simulation Environment

The main processor runs at 2 GHz and is equipped with separate 32 KB primary instruction and data caches that are two-way set associative and have a line size of 64 bytes. The secondary cache is unified, 512 KB, two-way set associative, and has a line size of 128 bytes. For sparse matrix applications, we scale down the cache size so that we can simulate the effect of running problems with large sparse matrices by running smaller problem sizes that we can simulate within a reasonable amount of time. This is justified since these applications have a working set size proportional to the problem size. For this class of applications, we use a 16 KB primary data cache and a 64 KB secondary cache, keeping the same line sizes and associativities. We also assume that the processor ISA includes prefetch and prefetch-exclusive instructions. In our processor model, a load miss stalls the processor until the first double-word of data is returned, while store misses will not stall the processor unless there are already references outstanding to four different cache lines. The processor model also contains fully-associative 64-entry instruction and data TLBs. For a TLB miss, we charge 65 processor cycles (miss handler latency) plus any associated cache and memory access time. The simulated page size is 4 KB.

The embedded active memory processor is a dual-issue core running at the 400 MHz system clock frequency. The instruction and data cache behavior of the active memory processor is modeled precisely via a cycle-accurate simulator similar to that for the protocol processor in [9]. Our execution-driven 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. The system bus is 64-bits wide, and its bandwidth is 3.2 GB/s, which matches the DRAM bandwidth. The memory request queue is 16 entries deep. The access time to the first 8 bytes in DRAM cells is fixed at 125 ns (50 system cycles), similar to that in recent commercial high-end servers [28], [29]. The following 8-byte requests can be pipelined, in a way similar to [21]. The input and output queue sizes in the memory controller interface are set at 16 and 2 entries, respectively. The corresponding queues in the network interface are 2 and 16 entries deep. The network interface is equipped with four virtual lanes to aid deadlock-free routing. We assume processor interface delays of one system cycle inbound and four system cycles outbound and network interface delays of 16 system cycles inbound and eight system cycles outbound. We simulate 16-port crossbar switches organized as a fat tree and present results for a slow (150 ns hop time) as well as a fast (50 ns hop time) SAN router. The node-to-network link bandwidth is 1 GB/s, typical of current system area networks.

When comparing against flush schemes, we simulate user-level complete cache flushes to minimize the flush overhead. This does not involve any kernel trap overhead, but it does model the latency incurred in the cache hierarchy to flush the whole cache. Note that systems that only support selective page flushes will see a larger flush overhead because realistic problem sizes are far bigger than the cache size and the entire data structure is flushed one page at a time.

6 Simulation Results

Our simulation results are divided into three areas: uniprocessor active memory systems, single-node multiprocessor active memory systems, and multinode active memory clusters.

6.1 Uniprocessor Active Memory Systems

We report uniprocessor results for three active memory techniques: matrix transpose, sparse matrix, and linked-list linearization. We present speedup of the active memory version over the normal application. We also show the speedup of active memory applications with software prefetches only for the shadow space accesses. Prefetching the same accesses in the normal case is extremely difficult or highly inefficient, while our active memory techniques result in easily prefetched sequential data from the shadow space. For the first two techniques, we also give the speedup of active memory applications using user-level cache flushes (to emulate the Impulse memory controller [33]) rather than our hardware coherence, while, for applications involving linked-list linearization, cache flushes have no relevance since this technique requires leveraging the coherence mechanism.

6.1.1 Matrix Transpose

Fig. 9 shows the uniprocessor speedup of three matrix transpose benchmark applications with active memory optimization (AM), with active memory and software prefetching of the shadow address space only (AM+Prefetch), and with active memory using cache flushes rather than cache coherence (Flush), measured relative to the execution time of the normal application. Table 2 summarizes the speedup of the nonprefetched AM case versus normal execution, as well as how much our active memory system reduces CPU busy time, L2 cache read miss count, L2 cache write miss count, and data TLB stall time.

All the applications show the clear success of the matrix transpose technique. Our active memory system achieves speedup from 1.28 to 2.30 over a normal memory system. The matrix transpose technique improves performance by

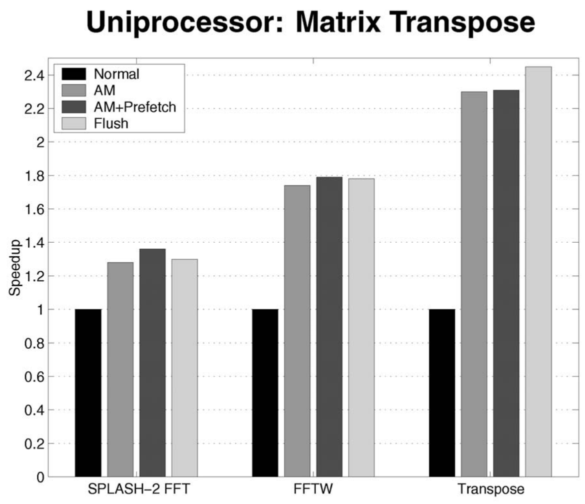


Fig. 9. Speedup of uniprocessor: matrix transpose.

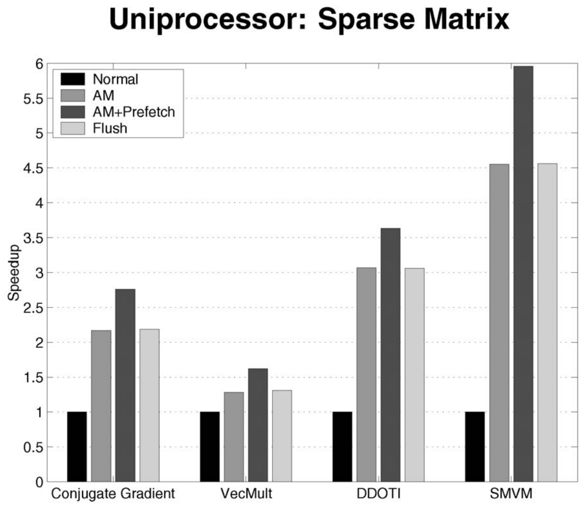


Fig. 10. Speedup of uniprocessor: sparse matrix.

eliminating the transpose overhead—read stall time, write stall time, and processor busy time. First, our technique reduces cache misses significantly. We found that the number of L2~cache read misses is reduced by 51 percent to 75 percent and write misses by 61 percent to 79 percent in our benchmark applications. We also enhance TLB performance. Long-strided column-wise memory accesses in the normal matrix hurt TLB performance, while sequential row-wise accesses in the shadow matrix yield better performance. The simulation results show that the matrix transpose technique removes almost all the TLB misses. However, caching TLB entries in the primary and secondary data caches alleviates TLB miss penalties for the normal applications; therefore, saving TLB misses is a smaller performance effect than reducing cache misses. Finally, we save processor busy time because the main processor does not need to execute the transpose phase, though this is a minor factor compared to the memory stall time savings.

With software prefetching of shadow space accesses, our technique can improve performance even further. Each prefetched application shows speedup of 1.36, 1.79, and 2.31, respectively. While both the normal and active memory executions could benefit equally from prefetching row-wise accesses in the normal matrix, here we emphasize that prefetching sequential accesses in the shadow matrix is a bonus of the active memory technique. Note that the Transpose microbenchmark shows little benefit from prefetching since it is a highly memory-bound application that does not include enough computation to hide the memory latency.

Here, our optimistic software cache flush scheme has marginally better performance than our hardware coherence-based system. However, given the advantages that our solution brings to the programming model along with its multiprocessor potential, we note that our results show that a hardware coherence-based approach can achieve these advantages at a performance level commensurate with the cache flush technique.

6.1.2 Sparse Matrix

Fig. 10 shows the uniprocessor speedup for the sparse matrix technique and Table 3 summarizes the reduction factors. We do not show the write miss count for DDOTI because the total number of writes in this benchmark is less than 14 (since it updates a single variable while calculating the inner product), so the reduction factor is meaningless.

Our active memory system achieves uniprocessor speedup from 1.28 to 4.55 across the four benchmarks. Although there is a reduction in processor busy time (from the

TABLE 2
Reduction Factors of Uniprocessor: Matrix Transpose

	FFT	FFTW	Transpose
Speedup	1.28	1.74	2.30
CPU Busy Time	12.4%	0.5%	4.7%
L2 Read Miss Count	51.0%	74.8%	74.0%
L2 Write Miss Count	60.6%	78.5%	77.8%
TLB Stall	99.4%	98.1%	98.6%

TABLE 3
Reduction Factors of Uniprocessor: Sparse Matrix

	CG	VecMult	DDOTI	SMVM
Speedup	2.17	1.28	3.07	4.55
CPU Busy Time	16.5%	25.3%	22.8%	18.0%
L2 Read Miss Count	73.2%	48.3%	85.7%	87.2%
L2 Write Miss Count	17.9%	20.7%	-	21.6%
TLB Stall	48.4%	51.3%	98.2%	98.8%

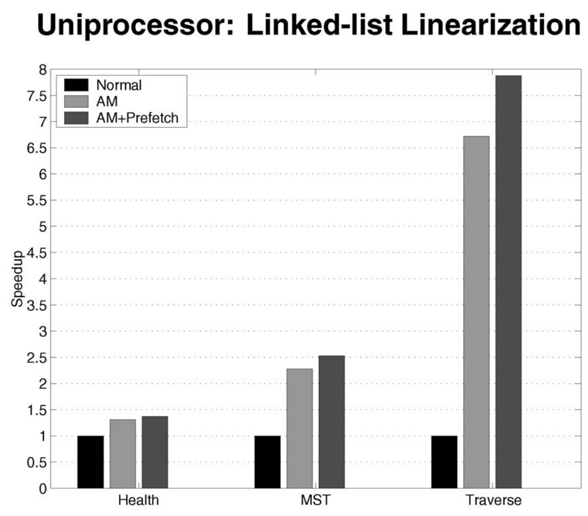


Fig. 11. Speedup of uniprocessor: linked-list linearization.

elimination of the address calculation for $Acol[j]$) and data TLB stall time (from sequential memory accesses in the shadow vector $v'[j]$), the major source of performance improvement is the significant reduction in cache misses since memory stall time occupies the major portion of the total execution time. The sparse matrix technique saves more than 50 percent of the L2 cache read misses and 14 percent of the write misses, thus reducing memory stall time by 21 percent to 81 percent. Here, the variation in memory stall time reduction across the applications is mainly due to the differing sparsity of the input matrices.

Software prefetching further improves performance since the sparse matrix technique enables prefetching of the shadow vector $v'[j]$ while it is difficult to prefetch the normal vector $v[Acol[j]]$ because of its sparse access pattern. For all four applications, our hardware coherence-based approach performs as well as or better than software coherence via flushes.

6.1.3 *Linked-List Linearization*

Fig. 11 shows the uniprocessor speedup for three linked-list linearization benchmarks and Table 4 summarizes the reduction factors. Please note that there are no results for the software flush mechanism because cache flushes alone cannot efficiently implement this technique.

The linked-list linearization technique does not reduce processor busy time. The busy time actually increases depending on how frequently it performs linearization since, whenever an application linearizes a linked-list, it calls the `AMLinearize` routine, which consumes processing time. However, as a result of the linearization, we can save a large number of cache misses and the reduction in memory stall time more than compensates for the increased processor busy time. In the Traverse microbenchmark, we achieve speedup of 6.72 from an 85 percent reduction in L2 cache read and write misses. In MST, the reduction of L2 cache read misses dominates and we achieve a speedup of 2.28. In Health, we optimize only two linked-lists out of the possible five lists per tree node, but still achieve a uniprocessor speedup of 1.4 over the normal execution.

TABLE 4
Reduction Factors of Uniprocessor: Linked-List Linearization

	Health	MST	Traverse
Speedup	1.31	2.28	6.72
CPU Busy Time	-2.6%	-91.0%	-57.0%
L2 Read Miss Count	21.3%	75.6%	84.8%
L2 Write Miss Count	42.6%	-30.7%	85.2%
TLB Stall	33.9%	14.7%	98.8%

Although only a minor contribution, our technique also saves TLB stall time since the sequential access pattern of linearized linked-list traversals improves TLB performance. In addition, software prefetches from the linearized linked-lists give additional performance boost, as seen in Fig. 11, due to the contiguous access pattern exhibited by the linearized linked-lists. Note that it is not efficient to prefetch the original linked-lists due to their random access pattern.

6.2 Single-Node SMP Active Memory Systems

In this section, we present single-node multiprocessor results for matrix transpose and sparse matrix techniques. We give the results for one, two, and four processor SMP systems. We show the speedup of our active memory techniques and analyze execution time in detail for the 4-processor executions. Note that we do not include multiprocessor results for the cache flush approach as cache flushes become unwieldy and their overhead increases as the system scales.

6.2.1 *Matrix Transpose*

Fig. 12 shows the speedup of three benchmark applications for 1, 2, and 4-processor SMP systems and Fig. 13 presents the execution time breakdown of both normal and active memory applications on a 4-processor SMP. Our coherence-based matrix transpose technique continues to improve performance on SMP multiprocessors. In all the applications and the system configurations, our active memory system outperforms the normal memory system. The performance benefit of the active memory system comes from three factors—read stall time savings, write stall time savings, and busy time savings, as explained in Section 6.1.1.

An important issue in our active memory architecture is memory controller occupancy. In SMP systems, one memory controller serves multiple processors. As the number of processors increases, the workload for the memory controller also increases. This problem is mitigated somewhat in normal memory systems because the latency of each protocol handler is small. However, as mentioned in Section 4.2.6, in our system, active memory handler latencies are larger, so the occupancy of the memory controller may become a bottleneck in an SMP. For example, in SPLASH-2 FFT, the average AMPU occupancy for one, two, and four processors in normal execution is 7.5 percent, 21.3 percent, and 43.2 percent of the total

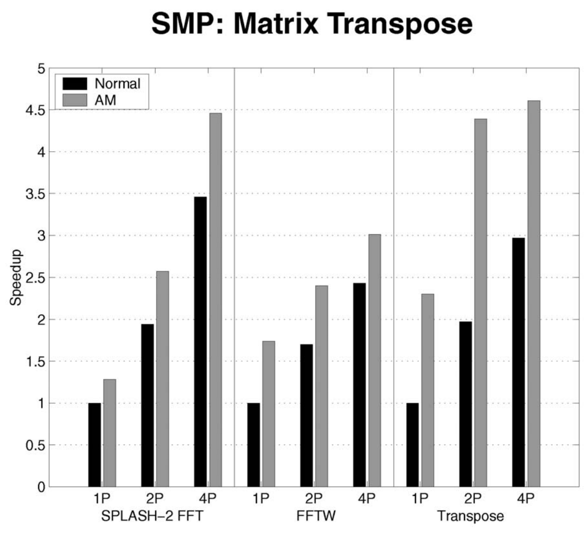


Fig. 12. Speedup of SMP: matrix transpose.

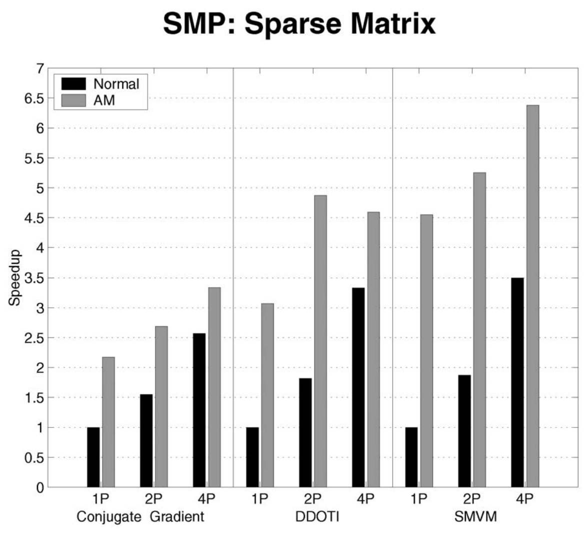


Fig. 14. Speedup of SMP: sparse matrix.

execution time, respectively, while the corresponding occupancy for the active memory executions is 15.5 percent, 29.4 percent, and 47.8 percent, though one must remember that the active memory execution time is smaller. This problem is even worse in FFTW and Transpose, where the occupancy approaches 90 percent on a 4-processor system, eliminating the possibility of a reduction in read stall time in the 4-processor FFTW, in spite of the 71 percent reduction in L2 cache read misses. However, the above simulation results show that our active memory architecture scales to a practical number of processors in single-node systems and still outperforms all normal executions. In addition, as logic speeds continue to outpace memory access time, occupancy concerns diminish. We also note that this concern arises only in SMP systems, not in the single-processor systems explained in Section 6.1 or the multinode systems described in Section 6.3.

6.2.2 *Sparse Matrix*

Fig. 14 shows the speedup of three sparse matrix benchmark applications and Fig. 15 presents the execution time breakdown on a 4-processor SMP. As can be seen in the figures, our sparse matrix technique outperforms the normal memory system on SMPs. Every active memory application is faster than normal execution for the same number of processors. Conjugate Gradient optimized by our technique is 2.17 (1P), 1.74 (2P), and 1.30 (4P) times faster than the corresponding normal execution. In DDOTI and SMVM, the active memory system has relative speedup of 3.07 (1P), 2.67 (2P), 1.38 (4P) and 4.55 (1P), 2.80 (2P), 1.82 (4P), respectively. The performance improvement of the sparse matrix technique mainly stems from the reduction in cache misses. We also reduce synchronization time in Conjugate Gradient by improving load balance.

As in the matrix transpose technique, the occupancy of the memory controller becomes a bottleneck in SMP systems. The occupancy in the active memory executions is 82.1 percent of the total execution time in Conjugate

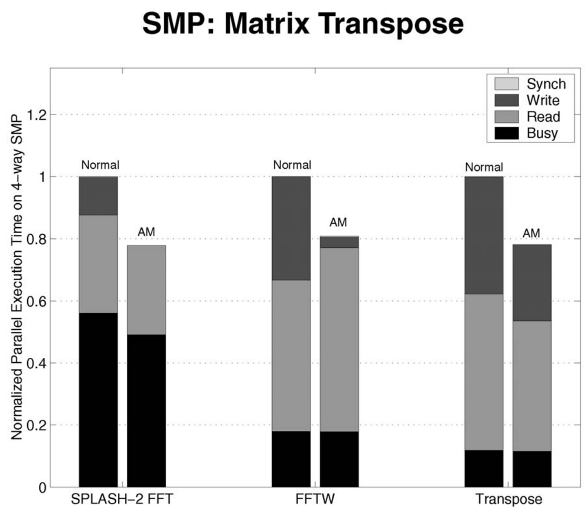


Fig. 13. Execution time on four processors for SMP: matrix transpose.

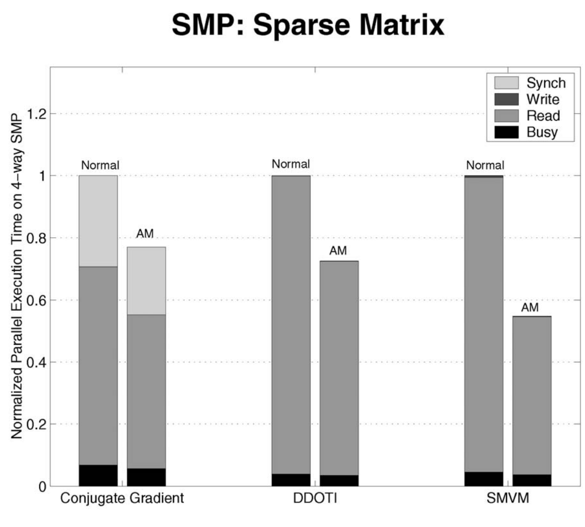


Fig. 15. Execution time on four processors of SMP: sparse matrix.

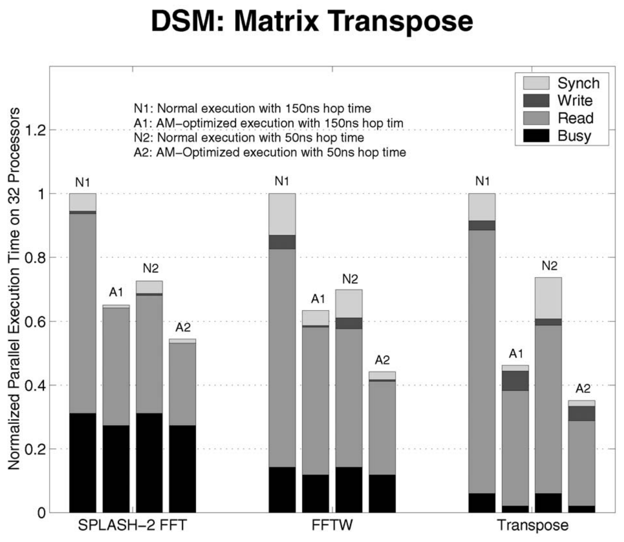


Fig. 16. Execution time on 32 processors for DSM: matrix transpose.

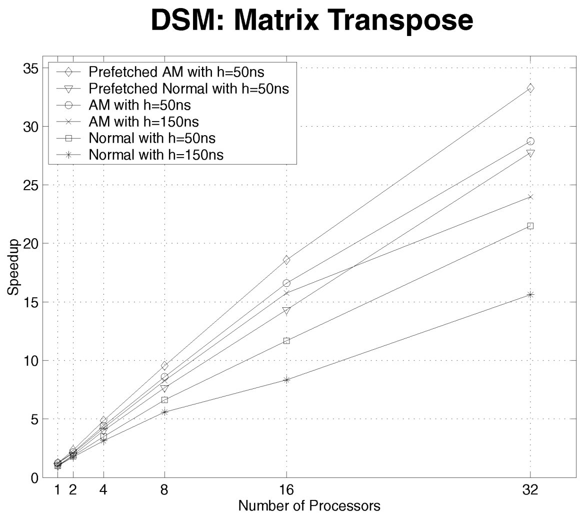


Fig. 17. Scalability of prefetched FFT for DSM: matrix transpose.

Gradient, 92.6 percent in DDOTI, and 98.7 percent in SMVM on a 4-processor system, which is 30.6 percent (Conjugate Gradient), 5 percent (DDOTI), and 17.1 percent (SMVM) higher than in the normal executions. The higher occupancy in our system explains why the relative speedup of the active memory applications over the normal executions gets smaller as the number of processors increases. It is also the reason that 4-processor active memory speedup in DDOTI is smaller than the 2-processor one (though still higher than the 4-processor normal speedup).

6.3 Multinode Active Memory Systems

This section presents simulation results for our matrix transpose and parallel reduction techniques on multinode active memory systems. We analyze parallel execution time and scalability for both normal and active memory systems. We also explore the effects of network latency on the achieved speedup.

6.3.1 Matrix Transpose

We present the results for three matrix transpose applications. Fig. 16 shows the comparison of parallel execution time for both normal and active memory applications with two different network hop times running on a 32-node (uniprocessor node) DSM system. The benchmarks show speedup of 1.54 to 2.16 with 150 ns hop times and 1.34 to 2.09 with 50 ns, respectively. Recall that we optimized the normal applications with tiling and padding to avoid naive comparisons.

As explained in Section 6.1.1, the matrix transpose technique improves performance by eliminating the transpose phase, which leads to reduction in memory access time, processor busy time, and TLB stall time. Saving cache misses improves performance even more in multinode systems through the elimination of many longer-latency remote memory accesses. In addition, we also reduce synchronization stall time in multinode systems. While this active memory technique distributes memory accesses over the entire program execution, the normal applications generate bursts of memory accesses (especially remote accesses) during the transpose phase, resulting in system-bus and network congestion and, hence, load imbalance and high synchronization stall time. Further, normal applications require barriers both before and after the transpose phase, but the active memory system needs only one (see Section 3.1).

Figs. 17, 18, 19, and 20 show the speedup of three benchmarks relative to uniprocessor normal execution with two different network hop times as the number of processors increases. These results show that our active memory system scales significantly better than normal memory systems. In all configurations—for different numbers of processors and network hop times—our active memory system always outperforms the normal system. Further, the performance gap between our system and the normal system widens as the number of processors increases. For example, for FFTW with 150 ns hop times, our system is 6 percent faster than the normal system on one processor, but 58 percent faster for 32 processors. The scalability of the active memory system stems mainly from reducing remote memory accesses. For instance, while we save only local misses in a 1-processor system, we can save half local misses and half remote misses in a 2-processor system. Though the total number of reduced cache misses is the same, we can get better speedup on a 2-processor system because we save the larger remote cache miss penalties.

Fig. 17 includes results for a software prefetched SPLASH-2 FFT. To the extent that it can hide remote miss latencies, software prefetching has a similar effect as our active memory technique. The benefit of active memory systems is reduced if software prefetching is used in normal applications. However, note that our active memory system still gives better performance than normal memory systems. First, our system can also take advantage of the prefetch optimization. The active memory speedup of the prefetched version is 32.10, while the nonprefetched active memory speedup is 28.72 in a 32-processor system. Second, though software prefetching can tolerate cache miss penalties, it still generates the same number of memory accesses. Our technique actually reduces the number of memory accesses

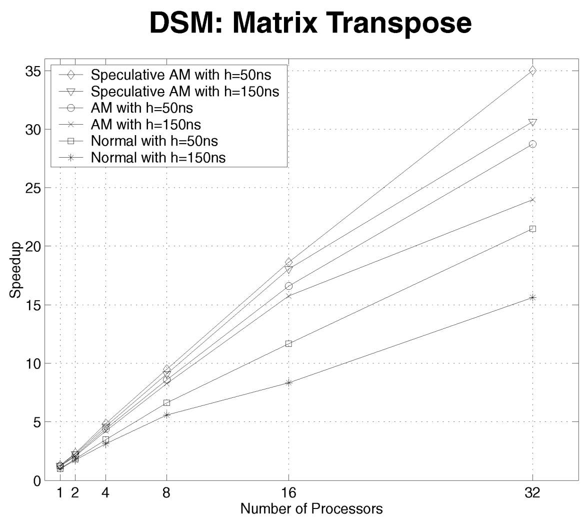


Fig. 18. Scalability of speculative FFT for DSM: matrix transpose.

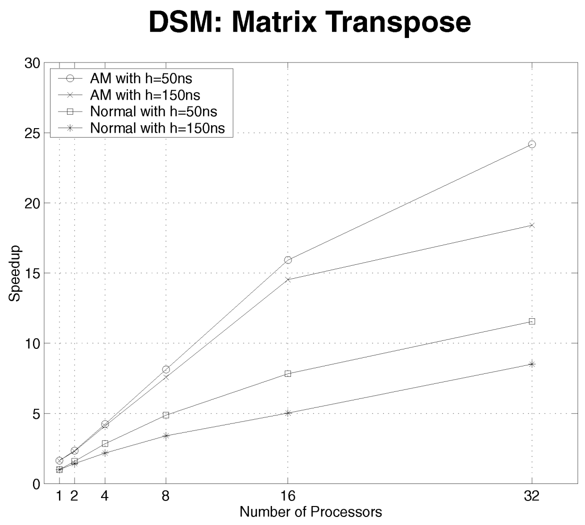


Fig. 20. Scalability of transpose for DSM: matrix transpose.

by improving locality. This difference results in lower memory system congestion and smaller synchronization stall times. The simulation results show a 79 percent reduction in synchronization stall time in our system.

Fig. 18 presents results for the speculative matrix transpose protocol discussed in Section 4.2.2 and shows the success of our technique. The speculative AM execution with a 150 ns hop time is faster than any other nonspeculative AM or normal execution (even with faster 50 ns hop times) and the speculative AM system with a 50 ns hop time always shows the best performance. In addition, the performance gap widens as the number of processors increases. Correct speculation changes remote memory accesses into local accesses, which, in conjunction with our AM-accelerated transpose operations, improves performance even further. Note that AM speculation can be more accurate than non-AM speculation. For a cache line write-back, the memory controller predicts the next consumer and pushes the cache line speculatively. In a non-AM context, the next consumer can be any node in the system, so the prediction must be made from the entire set of nodes. However, the prediction domain of the AM system is limited to a binary choice between either the original matrix or the shadow matrix. The synergy in the combination of our active memory technique and our protocol speculation technique allows for latency hiding of AM operations and improved prediction accuracy.

6.3.2 Parallel Reduction

We present results for four parallel reduction applications. Fig. 21 shows the comparison of parallel execution time on a 32-processor system. The active memory system achieves speedup of 1.19 to 2.81 with 150 ns hop times and 1.15 to 2.71 with 50 ns, respectively. All the applications (normal and AM) use software prefetching to hide remote memory latency.

AM parallel reduction benefits from eliminating the merge phase, which leads to processor busy time savings (by executing the merge phase in the memory controller)

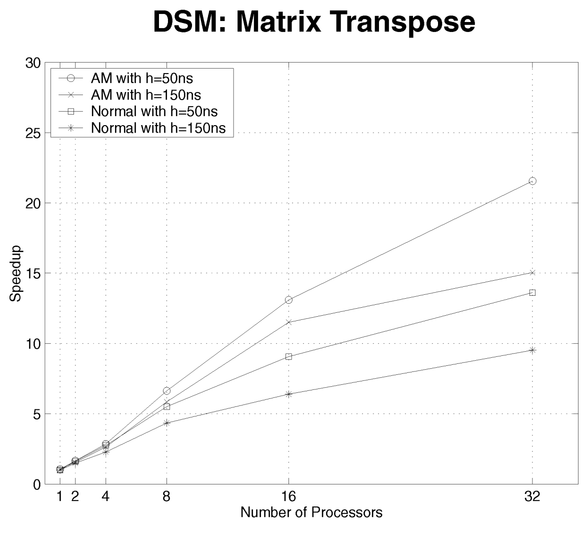


Fig. 19. Scalability of FFTW for DSM: matrix transpose.

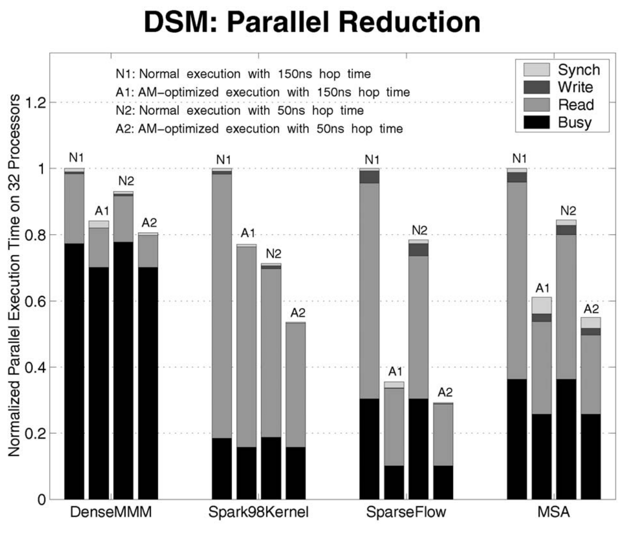


Fig. 21. Execution time on 32 processors for DSM: parallel reduction.

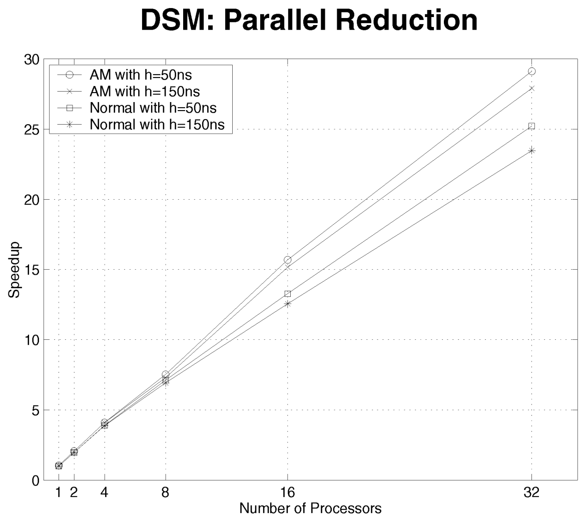


Fig. 22. Scalability of DenseMMM for DSM: parallel reduction.

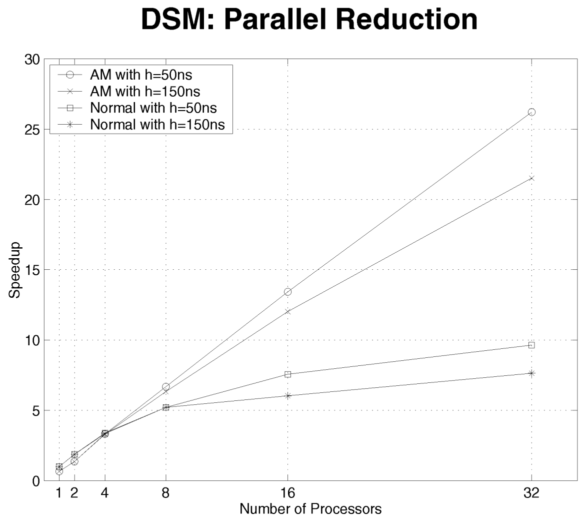


Fig. 24. Scalability of SparseFlow for DSM: parallel reduction.

and read stall time savings (by removing cache misses, especially remote misses, during the merge phase), the latter being dominant. We found that the busy time reduction in each benchmark is 8.6 percent (DenseMMM), 13.8 percent (Spark98Kernel), 63.7 percent (SparseFlow), 27.6 percent (MSA), and the L2 cache read stall time savings is 94 percent, 24 percent, 89 percent, and 94 percent, respectively. DenseMMM has a dominant busy time and the merge phase forms a small portion of the total execution time. As a result, Amdahl's Law limits the achievable performance gain. The surprisingly high speedup of SparseFlow stems from the sparse structure of the write operations to the reduction vector. In normal executions, even if a cache line does not contribute to the final reduced vector, every data point needs to be visited in the merge phase since it is hard to know (especially if the reduced vector is sparsely written) which cache lines ultimately contribute to the final reduced value. However, in the active memory case, the reduction is exposed to the memory controller and the memory controller touches only those cache lines that contribute to the final reduced value because the processors request only these shadow lines.

Figs. 22, 23, 24, and 25 show the active memory speedup of the four benchmarks as the number of processors varies. It is clear that the active memory optimization achieves significantly better scalability than the normal applications. Active memory applications are always faster than normal executions and, as the number of processors increases, the performance gap widens. The scalability of the parallel reduction technique results from the memory access pattern of the merge phase. Since the total volume of memory accesses in the merge phase remains constant, with P processors, $\frac{1}{P}$ fraction of the accesses remain local while the remaining $\frac{P-1}{P}$ fraction are remote memory accesses. In normal memory systems, this remote memory fraction increases with increasing P, therefore the merge phase suffers from an increasing number of remote memory accesses as the system scales. With active memory optimization, however, these accesses are moved from the critical path of execution to the writeback messages.

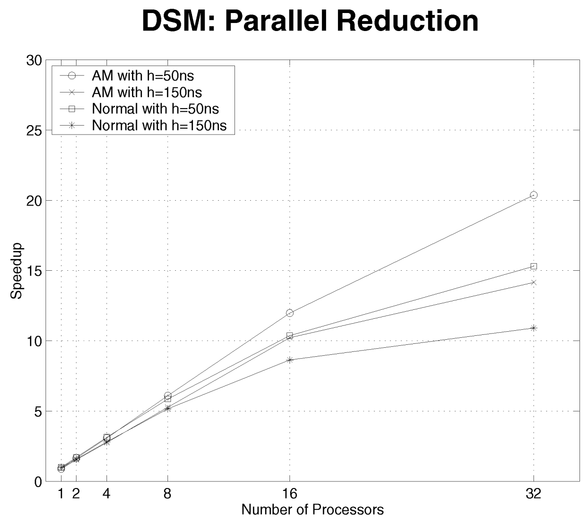


Fig. 23. Scalability of Spark98Kernel for DSM: parallel reduction.

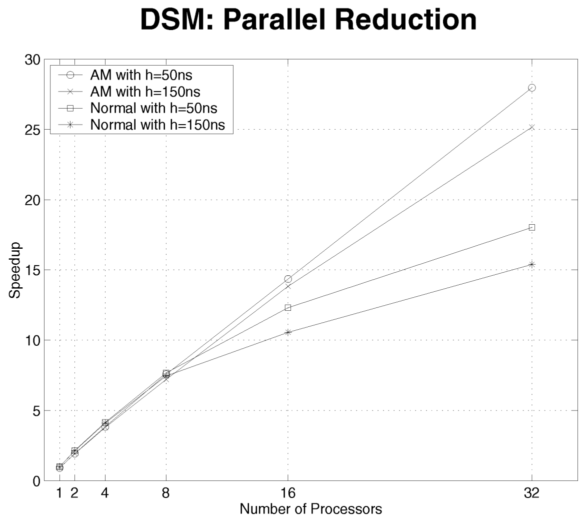


Fig. 25. Scalability of MSA for DSM: parallel reduction.

Therefore, the scalability of the merge phase is greatly enhanced, resulting in the widening performance gap between active memory optimization and normal execution as the system scales.

7 Conclusions

Our active memory controller architecture and coherence protocol extensions improve the memory system performance of applications that exhibit poor cache behavior on a spectrum of machine configurations, from uniprocessors through single-node multiprocessors to distributed shared memory systems. We have detailed the microarchitecture of our novel active memory controller that extends the cache coherence mechanism to implement address remapping operations without requiring cache flushes. The address remapping techniques improve spatial locality, enhancing cache performance in both uniprocessor and multiprocessor systems. However, they create data coherence problems that our programmable active memory controller solves by executing augmented cache coherence protocols. In addition, the same cache coherence mechanism allows us to efficiently extend traditional uniprocessor active memory techniques to multiprocessor systems. The end result is a completely transparent and highly scalable memory system.

The heart of our memory system is the active memory controller that runs active memory protocols merging the data coherence mechanism and active memory operations. Our active memory controller provides flexibility without sacrificing performance. The programmability of the AMPU supports various active memory techniques without any additional hardware cost and the AMDU accelerates address remapping operations with its specialized cache line assembly and disassembly hardware.

Through detailed simulation, we have shown that our active memory system achieves significant uniprocessor speedup and that the same memory architecture accomplishes remarkable scalability in SMP and DSM multiprocessors. A class of applications that takes advantage of our active memory techniques enjoy speedup from 1.3 to 7.6 on uniprocessors and super-linear scalability in multiprocessors, without sacrificing the performance of conventional nonactive applications.

Acknowledgments

This research was supported by US National Science Foundation CAREER Award CCR-9984314.

References

[1] M.C. Carlisle and A. Rogers, "Software Caching and Computation Migration in Olden," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 29-38, July 1995.

[2] M. Chaudhuri, D. Kim, and M. Heinrich, "Cache Coherence Protocol Design for Active Memory Systems," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications,* pp. 83-89, June 2002.

[3] "DIS Benchmark Suite" http://www.aaec.com/projectweb/dis/, 2002.

[4] J.J. Dongarra et al., "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software,* vol. 16, no. 1, pp. 1-17, Mar. 1990.

[5] J. Drapper et al., "The Architecture of the DIVA Processing-in-Memory Chip," *Proc. 16th ACM Int'l Conf. Supercomputing,* pp. 14-25, June 2002.

[6] M. Frigo and S.G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proc. 23rd Int'l Conf. Acoustics, Speech, and Signal Processing,* pp. 1381-1384, May 1998.

[7] M.J. Garzaran et al., "Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors," *Proc. 10th Int'l Conf. Parallel Architectures and Compilation Techniques,* Sept. 2001.

[8] K. Gharachorloo et al., "Architecture and Design of AlphaServer GS314," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 13-24, Nov. 2000.

[9] J. Gibson et al., "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 49-58, Nov. 2000.

[10] B. Goeman, H. Vandierendonck, and K.D. Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* pp. 147-216, Jan. 2001.

[11] M. Heinrich, E. Speight, and M. Chaudhuri, "Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters," *Proc. Fourth Int'l Symp. High-Performance Computing,* pp. 78-92, May 2002.

[12] "InfiniBand Architecture Specification, Volume 1.0, Release 1.0," InfiniBand Trade Assoc., 24 Oct. 2000.

[13] Y. Kang et al., "FlexRAM: Toward an Advanced Intelligent Memory System," *Proc. Int'l Conf. Computer Design,* pp. 192-141, Oct. 1999.

[14] D. Keen et al., "Cache Coherence in Intelligent Memory Systems," *Proc. ISCA 2000 Solving the Memory Wall Problem Workshop,* June 2000.

[15] D. Kim, M. Chaudhuri, and M. Heinrich, "Leveraging Cache Coherence in Active Memory Systems," *Proc. 16th ACM Int'l Conf. Supercomputing,* pp. 2-13, June 2002.

[16] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.

[17] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 172-183, May 1999.

[18] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 241-251, June 1997.

[19] C.-K. Luk and T.C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 88-99, May 1999.

[20] R. Manohar and M. Heinrich, "A Case for Asynchronous Active Memories," *Proc. ISCA 2000 Solving the Memory Wall Problem Workshop,* June 2000.

[21] B.K. Mathew et al., "Design of a Parallel Vector Access Unit for SDRAM Memory Systems," *Proc. Sixth Int'l Symp. High Performance Computer Architecture,* pp. 39-48, Jan. 2000.

[22] "Netlib Sparse BLAS," http://netlib.bell-labs.com/netlib/sparse-blas/, 2002.

[23] "NIST Sparse BLAS," http://math.nist.gov/spblas/, 2002.

[24] D.R. O'Hallaron, "Spark98: Sparse Matrix Kernels for Shared Memory and Message Passing Systems," Technical Report CMU-CS-97-178, Oct. 1997.

[25] D.R. O'Hallaron, J.R. Shewchuk, and T. Gross, "Architectural Implications of a Family of Irregular Applications," *Proc. Fourth IEEE Int'l Symp. High Performance Computer Architecture,* pp. 80-89, Feb. 1998.

[26] M. Oskin, F.T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *Proc. 25th Int'l Symp. Computer Architecture,* pp. 192-143, June 1998.

[27] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. 29th Int'l Symp. Computer Architecture,* pp. 111-122, May 2002.

[28] "SGI 3000 Family Reference Guide," http://www.sgi.com/origin/3000/, 2002.

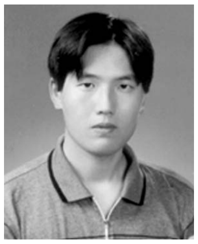
[29] Sun Microsystems, "Sun Enterprise 10000 Server—Technical White Paper," http://www.sun.com/servers/white-papers/, 2002.

[30] "Third Generation I/O Architecture," http://www.intel.com/technology/3GIO/, 2002.

[31] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

[32] W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News,* vol. 23, no. 1, pp. 14-24, Mar. 1995.

[33] L. Zhang et al., "The Impulse Memory Controller," *IEEE Trans. Computers,* special issue on advances in high-performance memory systems, vol. 50, no. 11, pp. 1117-1132, Nov. 2001.



Daehyun Kim is a PhD student in electrical and computer engineering at Cornell University. He received the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology in 1999 and the BS degree in electronic engineering from Yonsei University, Korea, in 1997. His research interests include memory systems design and performance analysis, especially multithread and multiprocessor memory systems. He also has interests in multiprocessor architecture, cache coherence protocols, hardware/software codesign, and simulation methodology.

Mainak Chaudhuri received the BTechn degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, in 1999, and the MS degree in electrical and computer engineering from Cornell University in 2001, where he is currently working toward the PhD degree. His research interests include micro-architecture, parallel computer architecture, cache coherence protocol design, and cache-aware parallel algorithms for scientific computation. He is a student member of the IEEE and the IEEE Computer Society.

Mark Heinrich received the PhD degree in electrical engineering from Stanford University in 1998, the MS degree from Stanford in 1993, and the BSE degree in electrical engineering and computer science from Duke University in 1991. He is an associate professor in the School of Electrical Engineering and Computer Science at the University of Central Florida and the founder of its Computer Systems Laboratory. His research interests include active memory and I/O subsystems, novel parallel computer architectures, data-intensive computing, scalable cache coherence protocols, multiprocessor design and simulation methodology, and hardware/software codesign. He is the recipient of a US National Science Foundation CAREER Award supporting novel research in data-intensive computing. He is a member of the IEEE and the IEEE Computer Society.

Evan Speight received the PhD degree in electrical and computer engineering at Rice University in 1999, the MS degree from Rice in 1994, and the BS degree in electrical engineering from Stanford University in 1991. He is an assistant professor in the School of Electrical and Computer Engineering at Cornell University and a member of the Computer Systems Laboratory. His current research interests include distributed computing, parallel processing, computer architecture, location-independent data access, operating systems research, and fault-tolerant computing. He is a member of the IEEE and the IEEE Computer Society.