Active Memory Techniques for ccNUMA Multiprocessors

Daehyun Kim and Mainak Chaudhuri
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
{*daehyun, mainak*}*@csl.cornell.edu*

Mark Heinrich
School of EECS
University of Central Florida
Orlando, FL 32816
heinrich@cs.ucf.edu

Abstract

Our recent work on uniprocessor and single-node multiprocessor (SMP) active memory systems uses address remapping techniques in conjunction with extended cache coherence protocols to improve access locality in processor caches. We extend our previous work in this paper and introduce the novel concept of multi-node active memory systems. We present the design of multi-node active memory cache coherence protocols to help reduce remote memory latency and improve scalability of matrix transpose and parallel reduction on distributed shared memory (DSM) multiprocessors. We evaluate our design on seven applications through execution-driven simulation on small and medium-scale multiprocessors. On a 32-processor system, an active-memory optimized matrix transpose attains speedup from 1.53 to 2.01 while parallel reduction achieves speedup from 1.19 to 2.81 over normal parallel executions.

1. Introduction

Active memory systems provide a promising approach to overcoming the memory wall [19] 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 scatter/gather operations invoked via address re-mapping techniques 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 two different re-mapped addresses. In our previous work [1, 9], we have shown that with the aid of a flexible active memory controller, conventional hardware cache coherence protocols can be effectively extended to transparently support a number of address re-mapping techniques and achieve significant speedup on both uniprocessor and single-node multiprocessor (SMP) systems. In this paper, we expand our previous work to multi-node hardware distributed shared memory (DSM) systems using the same active memory controller with an integrated commodity network interface, while changing only the protocol software that is run on the memory controller. We call the resulting multi-node systems Active Memory Clusters (AMC), whose architecture is introduced in [6].

We discuss *Matrix Transpose* [9, 20] and *Parallel Reduction* [4, 9] as two representative multi-node active memory techniques in Section 3. Our implementation requires novel extensions to DSM cache coherence protocols and is detailed in Section 4. We also discuss protocol design problems particular to multi-node active memory techniques, including issues like page placement and deadlock avoidance. In Sections 5 and 6 we present detailed simulation results on seven applications that demonstrate how well our active memory system scales compared to conventional parallel executions. For a 32-processor AMC system, AM-optimized applications enjoy speedup from 1.53 to 2.01 for matrix transpose and from 1.19 to 2.81 for parallel reduction over normal parallel executions.

The results presented here are the first known results for address re-mapping techniques in multi-node systems—techniques that are made possible only by our approach of employing novel extensions to DSM cache coherence protocols running on a flexible memory controller—and show that active memory clusters can effectively eliminate remote memory accesses for certain classes of applications, without sacrificing the performance of other (non-active) programs.

2. Related Work

Previous work in active memory systems can be divided into projects with data parallel PIMs such as DIVA [2], Active Pages [13], FlexRAM [8] and those with active memory controllers such as Impulse [20] and our previous work on flexible active memory controllers [1, 9]. This paper follows the latter approach.

The Impulse controller also uses address re-mapping techniques, but relies on software flushes rather than the

hardware cache coherence protocol to solve the data coherence problem associated with address re-mapping. In addition, the Impulse project has focused solely on uniprocessor systems, whereas our work leveraging cache coherence has shown improvements for both uniprocessor and single-node multiprocessor (SMP) systems, and, in this paper, on multi-node systems as well. Our parallel reduction technique was initially proposed in a non-active memory context in [4], but also used software flushes to guarantee data coherence and required changes to both the main processor and its cache subsystem. We follow the same idea, but our leveraging of the cache coherence protocol eliminates flushes and provides transparency in the programming model and scalability to multiprocessor systems without any changes to the main processor or its caches.

This paper extends our previous active memory techniques to multi-node systems. Some initial results of running non-active memory SPLASH-2 applications [18] are presented in [6], which shows that our Active Memory Cluster system has comparable performance to hardware DSM systems for non-active memory applications (the small differences are due to the network speed and do not affect single-node systems). The main contribution of this paper over our previous work is that it investigates the scalability of the active memory techniques in multi-node DSM systems and the corresponding DSM protocol issues, while [9] focuses on the role of the cache coherence protocol in single-node active memory systems and [6] on the performance of our memory controller for non-active memory applications.

Finally, our active memory techniques take advantage of writeback triggered data operations, which is similar to techniques used in the ReVive [14] and Memory Sharing Predictor [10] 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.

3. Multi-node Active Memory Techniques

In this section, we discuss two classes of active memory operations introduced in this paper: *Matrix Transpose* and *Parallel Reduction*. Matrix transpose and parallel reduction are common operations in many parallel applications. Unfortunately, they suffer from a large number of remote memory accesses in DSM systems. We show how active memory operations can improve performance by reducing the number of remote memory accesses. We also show why a cache coherence problem arises with these operations, and explain how we solve the problem.

3.1. Matrix Transpose

Consider a matrix A stored in memory in row-major order. 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. An example code is given here.

```
/* Row-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1
      sum += A[i][j];

BARRIER
Transpose(A, A');
BARRIER

/* Column-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1
      sum += A'[i][j];

BARRIER
Transpose(A', A);

BARRIER
```

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, as shown in the above example. 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 (in the barriers). The remote memory accesses during the transpose phase especially become a bottleneck. Our active memory technique eliminates the transpose phase, and hence reduces this overhead. An example code optimized by the active memory technique is given below.

```
/* Active Memory initialization phase */
A' = AMInstall(A, N, N, sizeof(Complex));

/* Row-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1
      sum += A[i][j];

BARRIER

/* Column-wise access phase */
for i = id*(N/P) to (id+1)*(N/P)-1
   for j = 0 to N-1
      sum += A'[i][j];

BARRIER
```

Our active memory controller provides a matrix transpose via an address re-mapping technique [20] that maps 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 matrix size and element size provided via the one-time `AMInstall` library call. Thus, the matrix transpose is carried out by the memory controller, not by the main processor,

removing the software transpose overhead and eliminating a large number of cache misses. Note that the initialization phase does not perform a matrix transpose. It only communicates the information used to compose the shadow matrix A' to the memory controller.

This matrix transpose operation gives rise to a coherence problem between the original matrix A and the shadow matrix A'. Any two corresponding elements of the matrix A and the shadow matrix A' should be coherent with each other, yet the processors may be caching them at two separate locations. We solve this problem by extending the DSM cache coherence protocol. The details of the protocol are discussed in Section 4.1.

3.2. Parallel Reduction

Parallel Reduction maps a set of elements to a single element with some underlying operation. Consider an example 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. A simple parallel code is shown below. 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). In an actual implementation the i and j loops would be interchanged to get better cache behavior.

```
/* 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
```

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). Also $private\_x$ 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. The example code optimized by our active memory technique is shown below.

```
/* Active Memory initialization phase */
x' = AMInstall(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
```

The result vector x is mapped to a shadow vector x' in the initialization phase. The processors perform the reduction phase only to the shadow vector x'. The merge phase is removed as shown in the above code. In our active memory technique, the merge operations are done by the memory controller, not by the main processors. When each cache line of the shadow vector x' is written back to memory, the memory controller performs the merge operation [4]. Therefore, the active memory technique can save processor busy time by eliminating the merge phase, and remote memory access time since the writebacks are not in the critical path of execution.

This technique has a coherence problem similar to the matrix transpose discussed in Section 3.1. We solve the problem by extending the cache coherence protocol to keep the vectors x and x' coherent. The protocol is discussed separately in Section 4.2.

4. Implementation

This section details the implementation of our active memory system, focusing on active memory protocol design. Our implementation satisfies two major design goals: *flexibility* and *performance*. Flexibility is achieved by executing software protocol code tailored to service different kinds of active memory operations. To achieve high performance, we run the protocol code on a customized dual-issue embedded protocol processor augmented with a special data path unit satisfying the needs of active memory operations [9]. To support a multi-node cluster architecture, introduced in [6] as Active Memory Clusters (AMC), our memory controller is equipped with a network interface in addition to a processor interface. In forthcoming network architectures such as InfiniBand [7], or 3GIO [17], the network interface will be moved closer to the main CPU, attached directly to or integrated with the memory controller.

The coherence protocol for the baseline system can correctly execute any normal (non-active memory) shared memory application. It is an MSI write-invalidate bitvector protocol similar to that in the SGI Origin 2000 [11]. 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 by our active memory protocol extensions, and is explained in the following sections. 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 the invalidation acknowledgments at the requester. However, we support eager-exclusive replies where a write reply is immediately sent to the processor even 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.

4.1. Matrix Transpose

In this section, we explain the implementation of the matrix transpose protocol. Consider the example in Section 3.1. A is an $N \times N$ original matrix and A' is a shadow matrix address re-mapped 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. We use the same directory entry as in the base protocol. A cache line can be in one of 8 possible states—unowned or invalid, shared, dirty, pending shared, pending dirty exclusive, AM, AM pending shared, and 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, 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 re-mapped 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 re-mapped 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, but this does not slow down conventional applications since this check is not on the critical path.

If the requested cache line is in the AM state, there is a potential data coherence problem. Because re-mapped 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 size and matrix element size provided by the one-time `AMInstall` call, we calculate each re-mapped 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 re-mapped cache lines, we can safely reply with the requested cache line. Finally, we update the directory entries of the requested cache line and the re-mapped cache lines. The requested cache line is set in the shared or dirty state based on the request type and the re-mapped cache lines in the AM state. This will guarantee data coherence in the future. During the entire procedure, if we encounter any cache line in the pending state, we send a negative acknowledgment to the requester. We adopt the same scheme as in Origin 2000 protocol for forward progress, so our protocol ensures that the retry will eventually succeed.

The 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. 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 in the memory controller and the gather operation eliminates the software transpose overhead from the main processors.

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 because it saves remote memory accesses. First, it is performed during a writeback operation, and hence is not in the critical path of the execution. Second, it writes back data elements from the shadow space to the normal space, so the next time a processor accesses the data elements in the normal space it does not need to access the (potentially remote) shadow space.

4.2. Parallel Reduction

In this section, we discuss the protocol extensions to support parallel reduction. Please refer to the example in Section 3.2. 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 entry 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 out the 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 to 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.

The last case is the simplest. In this case the directory entries of both the x and x' cache lines are clean. So 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 to 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 examine what happens when a read or a write 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 exactly the same as 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. Also, 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 pending state of the appropriate flavor 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.

4.3. Design Issues

In this section, we summarize implementation-specific issues particular to multi-node active memory systems. Our discussion touches on three topics, shadow page placement, cache line invalidation, and deadlock avoidance.

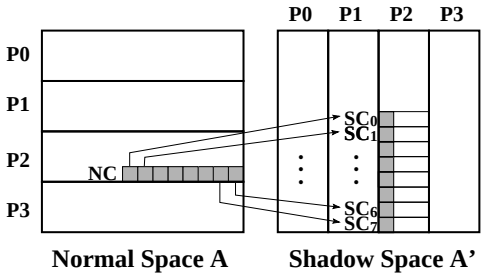

Figure 1. Page Placement for Transpose

Shadow page placement is a unique problem to multi-node active memory systems. To understand the problem, let us consider a memory snapshot from the matrix transpose protocol execution in Figure 1. 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. Figure 1 shows an example of a four node system. The normal space is partitioned row-wise and the shadow space is 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. We adopt the same method for the parallel reduction technique, although there the address re-mapping is one-to-one instead of one-to-many.

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). But in active mem-

ory techniques the re-mapped 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 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 re-mapping operation again at the invalidating node to compute the corresponding requested address. We exploit our flexibility and use the second solution since it does not require changes to the 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, and hence careful data buffer management schemes are needed. However, we solve both the problems similarly to conventional DSM systems—a handler running out of any necessary resource (e.g. a data buffer or an outgoing network queue slot) suspends execution and reschedules itself at a later point in time instead of waiting for that resource. This ensures forward progress by allowing the memory controller to handle outstanding requests and break deadlock cycles.

5. Applications and 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 the two multi-node active memory techniques we use a range of applications—some are well-known benchmarks while others are microbenchmarks written to exhibit the potential of a particular technique. In Table 1 we summarize the applications and the problem sizes we use in simulation.

We use FFT from SPLASH-2 [18], FFTW [3], and a microbenchmark called Transpose to evaluate the performance of the matrix transpose techniques. We parallelized FFTW for multi-node shared memory systems. The microbenchmark reads and writes to a matrix and its transpose, and hence is highly memory-bound. The normal executions of all these applications are optimized with tiling and padding. 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 even larger speedup than that presented in Section 6.

Table 1. Applications and Problem Sizes

Applications	Problem Sizes
SPLASH-2 FFT	1M points
FFTW	8K×16×16 matrix
Transpose	1K×1K matrix
Dense MMM	256×256 matrix
Spark98Kernel	64K×64K matrix, 1M non-zeros
SparseFlow	512K nodes and 1M edges
MSA	64×128K matrix

To evaluate parallel reduction we use the dense matrix multiplication (Dense MMM) kernel, a modified Spark98 kernel that parallelizes one call to LocalSMVP [12], a microbenchmark called SparseFlow that 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 multi-source flow graph, and a microbenchmark called Mean Square Average (MSA) that 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. 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 and we accurately model the latency and cache effects of TLB misses. Our simulation environment is derived from that in [5], which was validated against a real DSM multiprocessor.

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 [5]. 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 access time of main memory SDRAM is fixed at 125 ns (50 system cycles), similar to that in recent commercial high-end servers [15, 16]. We assume processor interface delays of 1 system cycle inbound and 4 system cycles outbound, and network interface delays of 16 system cycles inbound and 8 system cycles outbound. We simulate 16-port crossbar switches in a fat-tree topology with a node-to-network link bandwidth of 1 GB/s, which are all typical of current system area networks. Since the latency of SAN routers is improving quickly, we present results for slow (150 ns hop time) as well as fast (50 ns hop time) routers.

6. Simulation Results

This section presents our simulation results for each of the active memory techniques. 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.

6.1. Matrix Transpose

We present the results for the three matrix transpose applications described in Section 5.1. Figure 2 shows the comparison of parallel execution time for both normal and active memory applications with two different network hop times running on a 32-processor system. Transpose shows the best speedup (relative to 32P normal executions) of 1.69 with 50 ns hop times and 2.01 with 150 ns. SPLASH-2 FFT and FFTW show speedup of 1.53, 1.82 with 150 ns hop times and 1.34, 1.69 with 50 ns hops, respectively. Recall that we optimized the normal applications with tiling and padding to avoid naive comparisons, yet active memory systems still show significant speedup. It is also interesting to note that the active memory executions with slow networks even outperform the normal executions with a three times faster network.

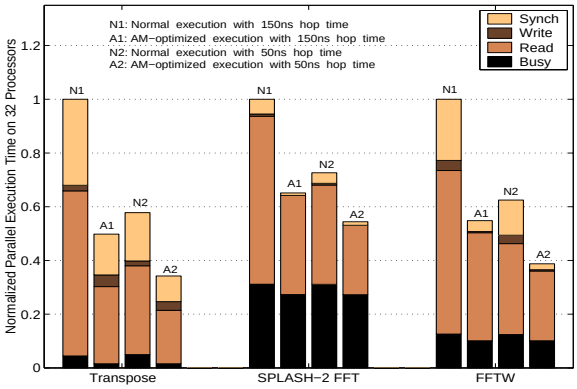

Figure 2. Matrix Transpose on 32 Processors

As explained in Section 3, the matrix transpose technique improves performance by eliminating the transpose phase, significantly reducing read stall time by eliminating cache misses—both local and remote. We found that the number of second level cache misses is reduced by 51%, 49% and 40% in Transpose, SPLASH-2 FFT and FFTW, respectively. Our technique also enhances TLB performance. Long-strided memory accesses in the normal applications hurt TLB performance, while shorter stride accesses in the active memory system yield better TLB performance. The simulation results show that the active memory system removes more than 95% of TLB misses. However, caching TLB entries in the primary data cache and L2 cache alleviates TLB miss penalties for the normal applications, and therefore saving TLB misses is a smaller performance effect than saving cache misses. We also save processor busy time because the main processors do not need to execute the transpose phase. Each application shows a 65% (Transpose), 12% (SPLASH-2 FFT) and 19% (FFTW) reduction in busy time. Finally, we reduce synchronization stall time by 52%, 83% and 82% with 150 ns hop times and 46%, 67% and 83% with 50 ns hops, in Transpose, SPLASH-2 FFT and FFTW, respectively. There are several reasons for this. First, normal applications have more barriers because they need barriers before and after the transpose phase. Second, while the active memory technique distributes memory accesses over the entire program execution, normal applications generate bursts of memory accesses (especially remote accesses) during the transpose phase, which results in system-bus and network congestion. Congested memory systems give rise to load imbalance and result in high synchronization stall times.

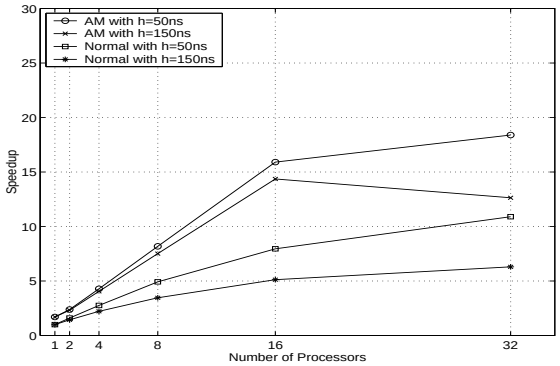

Figure 3. Scalability of Transpose

Next we show the scalability of the in-memory transpose technique. Figures 3, 4 and 5 show the speedup of Transpose, FFTW and SPLASH-2 FFT 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 nor-

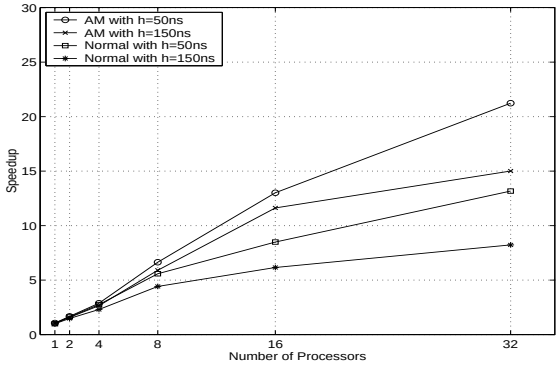

Figure 4. Scalability of FFTW

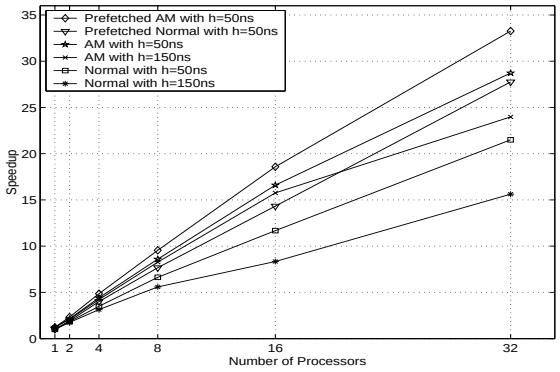

Figure 5. Scalability of SPLASH-2 FFT

mal systems widens as the number of processors increases. For instance, for FFTW with a 150 ns hop time, our system is 6% faster than the normal system on 1 processor, but 82% faster for 32 processors. The scalability of our active memory system comes mainly from saving remote memory accesses. Here, the ratio between local miss saving and remote miss saving is important. 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 might be the same, we can get better speedup on a 2-processor system because we save the larger remote cache miss penalties.

Figure 5 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 to our active memory technique. The benefit of active memory systems is reduced if software prefetching is used in normal applications. The speedup of our technique over the normal application in a 32-processor system with 50 ns hop time is 1.17 in the prefetched version, and 1.34 in the non-prefetched case. However, note that our active memory system still shows better performance than normal memory systems. First, our system also takes advantage of the prefetch optimization. Note that the active memory speedup of the prefetched version is 32.10 while the non-prefetched active memory speedup is 28.72 in a 32-processor system. Second, though software prefetching can tolerate remote cache miss penalties, it still generates the same number of memory accesses. Our technique actually reduces the number of memory accesses. This difference results in lower memory system congestion, and smaller synchronization stall times than normal systems. The simulation results show a 79% reduction of synchronization stall time in our system.

6.2. Parallel Reduction

We present the results for the four parallel reduction applications described in Section 5.1. All the applications use software prefetching to hide remote memory latency as much as possible. Figure 6 shows the comparison of parallel execution time with two different hop times for both normal and active memory applications running on a 32-processor system. As we have already mentioned, the active memory technique benefits by saving both processor busy time and read stall time, the latter being the dominant factor. With a 150 ns hop time, MSA achieves a speedup of 1.64 over the 32-processor normal execution while SparseFlow, Dense MMM, and Spark98Kernel enjoy speedup of 2.81, 1.19, and 1.30, respectively. The reduction in busy time for the four applications is respectively 27.6%, 63.7%, 8.61%, and 13.8%. We also found that for each application the reduction in L2 cache remote read stall time is 93.77% (MSA), 89.22% (SparseFlow), 93.98% (Dense MMM), and 24.49% (Spark98Kernel).

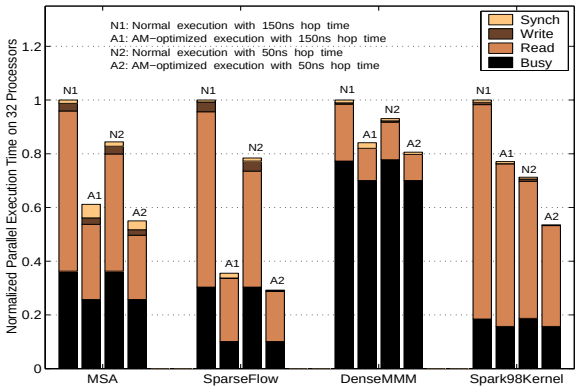

Figure 6. Parallel Reduction on 32 Processors

The surprisingly high reduction in the execution time 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 anything to the final reduced vector, every data point is visited in the merge phase. This is necessary since the reduction takes place entirely in software and at the software level it is impossible to know (especially when the reduced vector is sparsely written) which cache lines ultimately contribute to the final reduced value. On the other hand, in the active memory technique the reduction is exposed to the memory controller and the memory controller touches only those cache lines that con-

tribute to the final reduced value because the shadow cache lines requested by the processors correspond only to these "useful" normal cache lines. Dense MMM has a dominant busy time and the reduction phase forms a small portion of the total execution time. As a result, Amdahl's Law limits the achievable performance gain. Spark98Kernel mostly suffers from load imbalance due to its sparse read pattern that cannot be handled efficiently with static partitioning. Still, our technique is able to reduce some of its busy and read stall time. Even with a faster network, this active memory technique continues to achieve a substantial reduction in parallel execution time for all applications except the compute-bound Dense MMM. The speedup over normal 32-processor executions for MSA, SparseFlow, Dense MMM, and Spark98Kernel are 1.55, 2.71, 1.15, and 1.33.

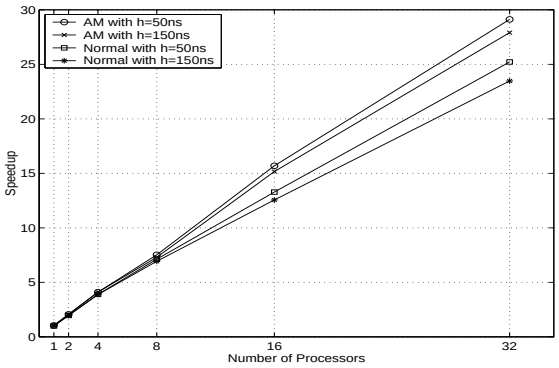

Figure 7. Scalability of Dense MMM

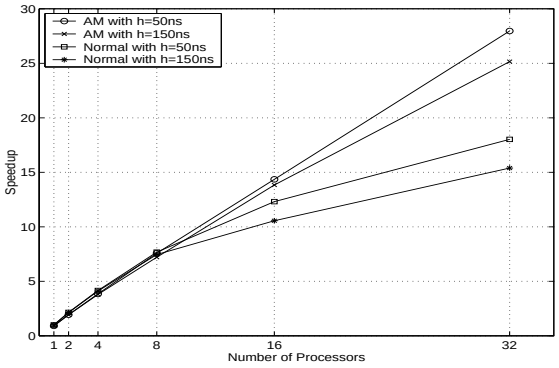

Figure 8. Scalability of MSA

Figures 7 and 8 show the speedup of Dense MMM and MSA relative to uniprocessor normal execution as the number of processors vary. Figures 9 and 10 show similar curves for Spark98Kernel and SparseFlow. It is clear that the active memory optimization achieves significantly better scalability than the normal applications. For 150 ns hop times on 32 processors, AM-optimized MSA, Dense MMM, Spark98Kernel, and SparseFlow enjoy speedup of 25.16, 27.91, 14.18, and 21.51, respectively, while the normal applications (not using AM optimization) achieve speedup of only 15.39, 23.47, 10.93, and 7.64. Decreasing the hop time to 50 ns boosts the speedup of the AM-

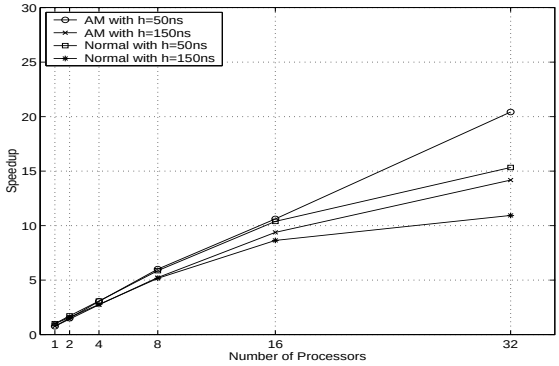

Figure 9. Scalability of Spark98Kernel

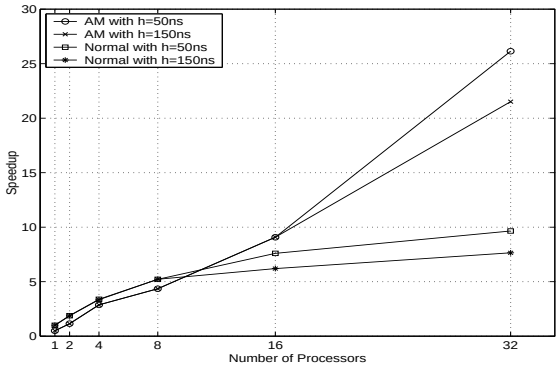

Figure 10. Scalability of SparseFlow

optimized applications to 27.97, 29.12, 20.42, and 26.15, respectively, while the normal applications achieve speedup of 18.02, 25.22, 15.33, and 9.65. As the number of processors increase, the performance gap between active memory optimization and normal execution widens. Since the total volume of memory accesses in the merge phase remains constant as the number of processors varies, with P processors $\frac{1}{P}$ fraction of the accesses remain local while the remaining $\frac{P-1}{P}$ fraction are remote memory accesses. It is clear that this remote memory fraction increases with increasing P. The merge phase of the parallel reduction operation therefore suffers from an increasing number of remote memory accesses as the system scales. With active memory optimization, however, all these accesses are moved from the critical path of execution to the writeback messages. 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

Certain classes of scientific applications written for cache-coherent distributed shared memory multiprocessors suffer from all-to-all communication patterns that ultimately get translated into remote cache misses even with proper page placement algorithms. In this paper we focus on two such classes of parallel kernels, namely, ma-

trix transpose and parallel reduction. We adopt the address re-mapping technique, previously used to improve locality of cache accesses in uniprocessor and single-node multiprocessors (SMPs), to reduce the remote miss overhead in multi-node systems using these two kernels.

This paper shows that our single-node active memory controller can be used in multi-node active memory clusters without any hardware modifications by designing the appropriate extensions to the DSM cache coherence protocol. We detail the novel cache coherence protocols that solve the data coherence problems inherent in address re-mapping techniques. We also discuss issues unique to multi-node re-mapping techniques such as shadow page placement and deadlock avoidance. Our simulation results show that the multi-node active memory techniques we present scale significantly better than the normal applications. For a 32-processor system, AM-optimized applications enjoy speedup from 1.53 to 2.01 for matrix transpose and from 1.19 to 2.81 for parallel reduction over normal executions that do not use active memory optimization. The end result is a completely transparent and highly scalable system that can efficiently support otherwise non-scalable parallel operations without introducing new hardware cost over single-node active memory systems and without affecting the performance of non-active memory programs. Our system is also flexible enough to support new multi-node active memory operations—the designer needs only to write the necessary coherence protocol extensions.

Acknowledgments

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

References

[1] M. Chaudhuri, D. Kim and M. Heinrich. Cache Coherence Protocol Design for Active Memory Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 83–89, June 2002.

[2] J. Drapper et al. The Architecture of the DIVA Processing-In-Memory Chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 14–25, June 2002.

[3] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, 1998.

[4] M. J. Garzaran et al. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[5] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[6] M. Heinrich, E. Speight, and M. Chaudhuri. Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters. In *Proceedings of the 4th International Symposium on High-Performance Computing, Lecture Notes in Computer Science (vol. 2327)*, pages 78–92, Springer-Verlag, May 2002.

[7] InfiniBand Architecture Specification, Volume 1.0, Release 1.0. InfiniBand Trade Association, October 24, 2000.

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

[9] D. Kim, M. Chaudhuri, and M. Heinrich. Leveraging Cache Coherence in Active Memory Systems. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 2–13, June 2002.

[10] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 172–183, May 1999.

[11] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[12] D. R. O'Hallaron, J. R. Shewchuk, and T. Gross. Architectural Implications of a Family of Irregular Applications. In *Fourth IEEE International Symposium on High Performance Computer Architecture*, pages 80–89, February 1998.

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

[14] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[15] SGI 3000 Family Reference Guide. http://www.sgi.com/origin/3000/

[16] Sun Enterprise 10000 Server–Technical White Paper. http://www.sun.com/servers/white–papers/

[17] Third Generation I/O Architecture. http://www.intel.com/technology/3GIO/

[18] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[19] Wm. A. Wulf and S. A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In *Computer Architecture News*, **23**(1):20–24, March 1995.

[20] L. Zhang et al. "The Impulse Memory Controller", *IEEE Transactions on Computers, Special Issue on Advances in High Performance Memory Systems*, pages 1117–1132, November 2001.