Simplifying Active Memory Clusters by Leveraging Directory Protocol Threads

Dhiraj D. Kalamkar
Intel Technologies Pvt. Ltd.
Bangalore 560001
INDIA
dhiraj.d.kalamkar@intel.com

Mainak Chaudhuri
Department of CSE
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

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

Abstract

Address re-mapping techniques in so-called active memory systems have been shown to dramatically increase the performance of applications with poor cache and/or communication behavior on shared memory multiprocessors. However, these systems require custom hardware in the memory controller for cache line assembly/disassembly, address translation between re-mapped and normal addresses, and coherence logic. In this paper we make the important observation that on a traditional flexible distributed shared memory (DSM) multiprocessor node, equipped with a coherence protocol thread context as in SMTp or a simple dedicated in-order protocol processing core as in a CMP, the address re-mapping techniques can be implemented in software running on the protocol thread or core without custom hardware in the memory controller while delivering high performance. We implement the active memory address re-mapping techniques of parallel reduction and matrix transpose (two popular kernels in scientific, multimedia, and data mining applications) on these systems, outline the novel coherence protocol extensions needed to make them run efficiently in software protocols, and evaluate these protocols on four different DSM multiprocessor architectures with multi-threaded and/or dual-core nodes. The proposed protocol extensions yield speedup of 1.45 for parallel reduction and 1.29 for matrix transpose on a 16-node DSM multiprocessor when compared to non-active memory baseline systems and achieve performance comparable to the existing active memory architectures that rely on custom hardware in the memory controller.

1 Introduction

Active memory (AM) techniques [16] use address re-mapping [33] in conjunction with cache coherence protocol extensions to significantly improve the performance of a number of popular scientific computing kernels on uniprocessors as well as shared memory multiprocessors. Traditionally, such systems have relied on customized hardware support for carrying out address translation and dynamic assembly of cache lines in the memory controller. Such support not only makes the system architecture complex, but also introduces barriers to adoption of such systems in commodity multiprocessors. In this paper we show that it is possible to retain most of the performance of traditional active memory systems while eliminating all custom hardware support from the memory controller. As in the previously proposed active memory systems [16], we maintain the flexibility of adding new active memory techniques without additional hardware support by executing the cache coherence protocols in the form of software handlers. Although single-threaded as well as multi-threaded applications enjoy significant performance improvement from the AM techniques, in this paper we evaluate our proposal on a 16-node distributed shared memory (DSM) multiprocessor with nodes capable of running one or two application threads, thereby allowing us to experiment with 16- and 32-way threaded parallel applications. The nodes are kept coherent via a directory-based write-invalidate bitvector protocol extended suitably from the SGI Origin 2000 [18]. We explore four different architectures of flexible directory controllers relevant to today's multi-core and multi-threaded systems. These include dual-core processors with a dedicated coherence protocol processing core and multi-threaded processors with a dedicated protocol thread context as in SMTp [4]. Each of these architectures may or may not use custom hardware in the memory controller for address re-mapping. In this work, these architectures with custom address re-mapping hardware serve as the baseline and we show how to leverage the flexible protocol processing support already present in these DSM multiprocessors to deconstruct the custom hardware while continuing to deliver performance comparable to the baseline.

We focus on two AM techniques, namely, matrix transpose and parallel reduction which were evaluated in the past proposals of AM techniques on DSM multiprocessors, and show how to carry out a generic set of operations required by the AM techniques without any custom hardware support in the memory controller. A detailed evaluation based on execution-driven simulation shows that executing our cache coherence protocols on a hardware thread-context in SMTp achieves a speedup of 1.45 for parallel reduction and 1.29 for transpose compared to a non-AM design that does not use address re-mapping across seven 16-way threaded scientific computing benchmarks. We present a thorough comparison of performance across a set of viable architectures and show how the system scales to 32 threads. Two major contributions of this paper are as follows.

- We present the first implementation of AM techniques on flexible hardware DSM multiprocessors that do not require custom address re-mapping hardware support in the memory controller.
- We evaluate our proposal on four different flexible directory controller architectures relevant to today's multi-core and multi-threaded nodes.

In the rest of this section we briefly introduce the AM techniques that have already been developed and the custom memory controller that has been employed in the past. We also present a brief discussion on the four different flexible directory controller architectures that we evaluate. In Section 2 we present our new AM protocol that does not rely on any custom hardware support in the memory controller and discuss potential area and peak power savings that result from removal of the address re-mapping hardware. Sections 3 and 4 present our simulation results. We discuss previous work in Section 5 and conclude in Section 6.

1.1 Active Memory Techniques and the AMDU

We first present a discussion on the parallel reduction and matrix transpose AM techniques and then identify the hardware support needed by these operations. Parallel reduction maps a set of items to a single item 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 which carries out a block-row decomposition of the matrix. 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 of pid */
for j = 0 to N-1
   private_x[pid][j] = e;
   for i = pid*(N/P) to (pid+1)*(N/P)-1
      private_x[pid][j] =
                    private_x[pid][j]⊗A[i][j];
BARRIER
/* Merge phase of pid */
for j = pid*(N/P) to (pid+1)*(N/P)-1
   for i = 0 to P-1
      x[j] = x[j]⊗private_x[i][j];
BARRIER
Subsequent uses of x
```

The reduction phase does not have any remote memory accesses because we can map *private_x* in local physical memory. However, the merge phase assigns mutually exclusive index sets of the result vector to each processor and suffers from a large number of remote misses due to an inherently all-to-all communication pattern. The AM technique completely eliminates the merge phase. The transformed code is shown below.

```
/* Active Memory initialization phase */
x' = AMInstall(x, N, sizeof(long long));
/* Reduction phase */
for j = 0 to N-1
   for i = pid*(N/P) to (pid+1)*(N/P)-1
      x'[pid][j] = x'[pid][j]⊗A[i][j];
BARRIER
Subsequent uses of x
```

The `AMInstall` function belongs to the AM library kernel and informs the coherence layer of the starting virtual address of x, its size, and the size of its elements via a series of uncached stores. The need for these three pieces of information will become clear in the following discussion. The `AMInstall` function also returns the starting virtual address of a "shadow space" large enough to hold a re-mapped vector x' belonging to each thread (thus the re-mapped area in virtual memory is P times the size of x). At this point this function also establishes the virtual to physical mapping of this shadow vector. One unique property of AM is that the re-mapped vector is not backed by physical memory and hence the name "shadow vector". All the shadow physical pages of the vector x' are made contiguous. This is easy to achieve because there is no concept of page replacement in the shadow area and hence no fragmentation. The shadow physical space is identified by the upper few bits of the physical address. For example, if in a 32-bit physical address space only 1 GB memory is installed, the upper two bits of the physical address can be used to identify four different address spaces, of which three are shadow spaces not backed by physical memory. The per-thread vector x' belongs to one such space and therefore, the starting physical address of x' is a pre-defined constant. A per-thread shadow vector is needed because a single global shadow vector used in [16] introduces a coherence problem for modern shared cache architectures (i.e. SMT and CMP). The thread k performs the privatized reduction phase only to the shadow vector $x'[k]$. When a dirty cache block belonging to the shadow space is evicted it is first sent to the home node of the cache block as usual. There the custom memory controller carries out the merge operation by computing the actual physical address corresponding to the evicted shadow cache block with the help of the three pieces of information obtained from `AMInstall` and stores the result in the actual location of vector x in physical memory [9]. For example, corresponding to a requested shadow physical address p', the virtual address of x would be $(p' - a' + V)$ where a' is the pre-defined starting address of the shadow physical area and V is the starting virtual address of x. Notice that $(V - a')$ is a constant and hence can be pre-computed when `AMInstall` executes. Computing the physical address of x from this virtual address requires a TLB in the memory controller, in addition to the virtual address computation logic. Further, the memory controller should be equipped with the merge hardware, e.g., an adder if addition is the underlying operation or a multiplier if multiplication is the underlying operation. The active memory data unit (AMDU) [16] integrates this custom support in the memory controller. Figure 1 shows how a shadow writeback is handled in the AMDU with seven major steps marked. The AM 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. However, the shadow vector (x') and the actual vector (x) should be kept coherent through cache coherence protocol extensions [16]. For example, a request for a cache block belonging to x must retrieve the corresponding re-mapped cache blocks belonging to x' from all owners (that are not evicted and hence marked in the directory entry), merge them one after another with the resident memory block with the help of the custom memory controller, and send the reply containing the final result. Notice that potentially there can be P owners of a shadow cache block.

Unlike parallel reduction, in matrix transpose the performance bottleneck results from poor cache utilization due to a matrix column walk. Consider a matrix A stored in memory in row-major order. An application accesses the matrix A first row-wise, and then column-wise. 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 = pid*(N/P) to (pid+1)*(N/P)-1
   for j = 0 to N-1
      sum += A[i][j];
```

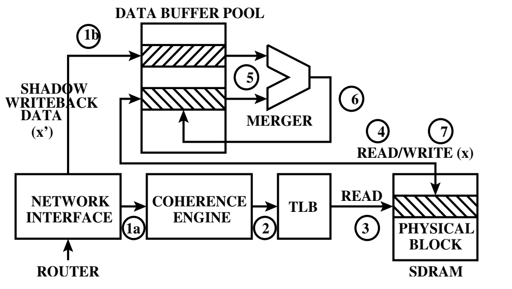

Figure 1. AM-assisted parallel reduction.

```
BARRIER
Transpose(A, A'); /* $A' = A^T$ */
BARRIER
/* Column-wise access phase */
for i = pid*(N/P) to (pid+1)*(N/P)-1
   for j = 0 to N-1
      sum += A'[i][j];
BARRIER
Transpose(A', A);
BARRIER
```

Though tiling the transpose phase reduces the number of cache misses and prefetching further reduces the cache miss latency in the critical path, 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). Especially, the remote memory accesses during the transpose phase become a bottleneck. The AM-transformed code eliminates the transpose phase and off-loads it to the AMDU, as shown below.

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

As proposed in [33], the AM technique allocates A^T in a shadow space A'. The shadow matrix A' is not backed by any real physical memory. Instead, it is composed by the memory controller on the fly by maintaining the invariant $A'[i][j] = A[j][i]$. For example, on a read request (shown as GET in Figure 2) from A', the memory controller composes the shadow cache block by gathering a number of elements from a column segment of A. To compute the virtual addresses of these words in A, it uses information such as the starting virtual address of A, the matrix dimensions and the element size provided via the one-time `AMInstall` library call. Further, for each word, one TLB lookup is needed to translate the virtual address to the corresponding physical address (for large-sized rows each word in the column segment will belong not only to a different cache block, but also to a different page). 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'. In summary, as in parallel reduction, the memory controller should be able to compute the corresponding physical addresses of the words in the original matrix A and dynamically assemble the requested cache block of shadow matrix A'. This dynamic cache line assembly/disassembly was also implemented in the AMDU [16] in the form of a three-stage pipeline where the first stage computes the virtual addresses of the cache blocks of A containing the corresponding words in a column segment, the second stage performs the TLB lookup for each of these virtual addresses, and the third stage computes the directory addresses and initiates the directory and data memory accesses. As in parallel reduction, here also the directory protocol extensions [16] should maintain the coherence between A and A'.

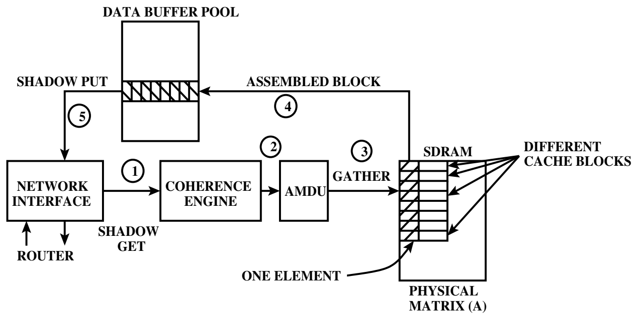

Figure 2. AM-assisted matrix transpose.

In summary, there are three basic operations offered by the custom-designed AMDU for these two AM techniques. These are shadow address to physical address translation (achieved by TLB hardware in the AMDU), dynamic cache line assembly and disassembly (achieved by issuing word, double-word, or quad-word read/write requests to the SDRAM banks), and the merge operation (achieved by the merge hardware embedded in the AMDU). These operations are done in a highly efficient manner by the hardwired AMDU pipeline [16], shown in Figure 3. The coherence protocol fills the base address buffer and initiates the AMDU pipeline. Each buffer contains 16 registers because in our simulated system a 128-byte shadow cache block can potentially be assembled from double words (64 bits) of 16 different physical cache blocks e.g., in matrix transpose. Therefore, 16 different base addresses, virtual addresses, physical addresses, directory addresses, and physical double words from application memory will be needed to assemble a shadow cache block. The central contribution of this paper (Section 2) is that we show how to move this custom support from hardware to coherence protocol software and continue to deliver most of the performance achieved by the AMDU.

1.2 Flexible Directory Controller Architecture

Flexible AM clusters employ directory controllers that can execute any cache coherence protocol. This makes introduction of new AM techniques easy. We exploit this flexibility to move the hardwired AMDU functionality to software protocol. With the removal of the AMDU, our AM cluster architecture has become even more flexible in terms of accommodating new techniques. In this section we briefly discuss

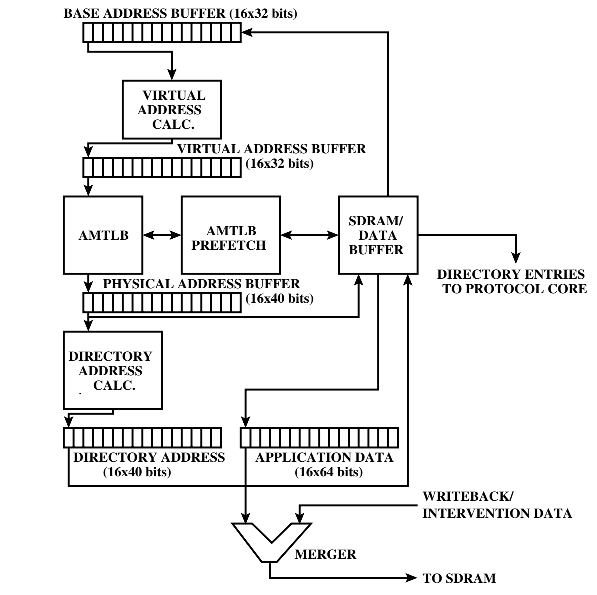
BASE ADDRESS BUFFER (16x32 bits)
VIRTUAL ADDRESS CALC.
VIRTUAL ADDRESS BUFFER (16x32 bits)
AMTLB
AMTLB PREFETCH
SDRAM/ DATA BUFFER
DIRECTORY ENTRIES TO PROTOCOL CORE
PHYSICAL ADDRESS BUFFER (16x40 bits)
DIRECTORY ADDRESS CALC.
DIRECTORY ADDRESS (16x40 bits)
APPLICATION DATA (16x64 bits)
WRITEBACK/ INTERVENTION DATA
MERGER
TO SDRAM

Figure 3. AMDU pipeline organization. The SDRAM/data buffer interface is not part of the AMDU.

the four flexible directory controller architectures that we examine. The exact architectural parameters are presented in Section 3. The fundamental requirement of a flexible directory controller is the ability to execute the coherence protocol in software [1, 17, 19, 21]. Therefore, the design varies depending on the position and the make of this execution engine. In the first design we use an on-die protocol core with single-level private instruction and data caches. In the second design we allow this protocol core to share the outermost level of the main core's cache (in our experiments it is the L2 cache), but remove its private data cache. This design enjoys a significant area and energy advantage over the first one because there is no extra hardware dedicated to the protocol caches. In the third design we augment the second design with a private L1 data cache for the protocol core. Finally, we consider the SMTp design [4] where the coherence protocol runs on a hardware thread context of the main core and eliminates the protocol core entirely. In this design the protocol thread shares most of the pipeline resources, including the cache hierarchy, with the application threads running in the same core. However, to avoid deadlock situations the protocol thread does require some nominal reserved resources [4]. The SMTp design is probably the most attractive one in terms of metrics such as performance per transistor, but the protocol core design may be easier to validate depending on the complexity of the core. All these four architectures are shown in Figure 4. Each of these architectures may optionally have a custom designed AMDU (shown dotted in the figure).

To clarify the discussion in the next section, we would like to mention that in all these architectures the memory controller receives physical addresses from the processor side and hands them over to the coherence protocol. Depending on the address space the coherence protocol takes appropriate actions e.g., for shadow space addresses it invokes the AM protocol. For the requests from the protocol thread, instead of invoking the coherence protocol recursively, the memory controller strips off the address space bits and sends the address to either the SDRAM banks or appropriate devices in the case of protocol-initiated memory-mapped I/O requests.

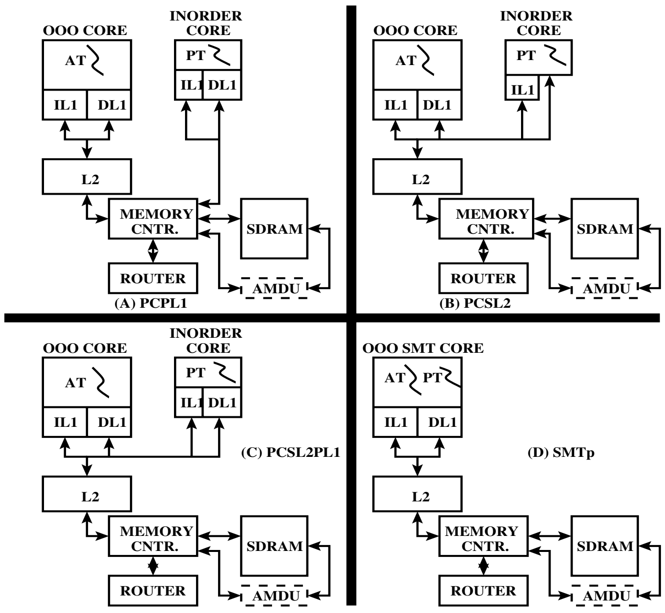

Figure 4. Flexible node architectures. AT denotes the application thread and PT denotes the directory protocol thread. The out-of-order (OOO) core may have more than one application thread, in which case it must be an SMT core. (A) PCPL1: Protocol Core with Private L1 caches, (B) PCSL2: Protocol Core with Shared L2 cache, (C) PCSL2PL1: Protocol Core with Shared L2 and Private L1 caches, (D) SMTp.

2 Deconstructing the AMDU

This section details our contribution of eliminating the AMDU hardware. We present novel extensions to the existing directory-based AM protocols [16] for carrying out address translation, merge operation, and dynamic assembly/disassembly of cache blocks. The virtual address calculation, shown in Figure 3, involves one conditional shift and one 32-bit add operation, and therefore, can be trivially done in coherence protocol software. Same applies to the directory address calculation which involves a constant amount shift followed by a 40-bit addition.

2.1 Parallel Reduction

In the AM parallel reduction technique, existing AM clusters require two pieces of hardware support. The first is an embedded TLB in the memory controller. To replace the TLB with appropriate extensions in protocol software we have two design options. The first option is to maintain a direct-mapped translation table in the protocol data area entirely managed by the protocol software. Each entry of the table has a valid bit, a tag, the translation, and the access control bits. The TLB hit/miss is determined by the protocol software by examining the valid bit and the tag field and on a miss, it issues a page table entry request to the SDRAM interface (or invokes the hardware page walker if one is implemented). The page table base address is statically allocated in a general purpose register of the protocol core/thread and is loaded with the correct value by the operating system when the application is launched. We implement this design in our experiments, although it introduces one extra complexity in the virtual memory management layer of the operating system. The operating system when modifying virtual to physical translations (due to swap or migration), in addition to shooting down the TLBs and flushing the caches, must now invalidate the potentially stale translations residing in the protocol's soft TLB area. For the sake of completeness, we mention one alternative. It is possible to allow the protocol thread in SMTp to directly access the data TLB of the core. However, this design either interferes with the functioning of the application threads or requires an extra port in an already performance-critical TLB design. Furthermore, this design puts heavy pressure on the data TLB and may re-introduce the poor TLB behavior in some of the AM techniques such as transpose. Since sharing a TLB among multiple cores is not very attractive due to floor-planning issues in a planar single-die architecture, we do not consider it in the rest of the paper.

We found moving the merge operation to protocol software much more challenging than the address translation support. The merge operation is invoked by the coherence protocol on the home node when it receives a writeback from shadow space or an intervention reply for shadow space (this intervention is usually generated by a normal space request on finding the corresponding shadow block dirty). The coherence protocol needs to read the memory-resident cache block in a memory buffer (MYB), merge it with the cache block in the message buffer (MEB) that comes with the writeback or intervention reply, and write the merge result back to memory. This process was shown in Figure 1 for the case of a shadow writeback. All data buffers are cache block sized. However, the protocol core/thread can load/store at most 64 bits of data between a data buffer and its register file. Let us assume that the L2 cache block size is 128 bytes and the reduction grain is 64 bits. A naïve software implementation would read 64-bit data at a time from two buffers into two registers, merge them into a register, and write this register back to one of the buffers. Finally, the result buffer is written back to memory. If the protocol load/store operations between data buffer and register file are uncached, this algorithm will require 32 uncached loads between MYB/MEB and register file, 16 merge instructions (add, multiply, etc.), and 16 uncached stores from register file to result buffer. Since uncached loads take a long time to complete, a large number of such instructions can severely hurt performance. We note that in the highly pipelined AMDU datapath this whole process would take only 16 cycles with one cycle per merge (assuming a simple merge operation like addition).

Allowing the already memory mapped data buffer pool to be cached in the protocol core's cache or in the shared cache of the protocol thread enables the use of cached load/store instructions, which are fast and can even be issued speculatively. For example, suppose the data buffer storage starts at the physical address 0x8000. Each data buffer holds 128 bytes of data (one cache block). Therefore, data buffer zero starts at address 0x8000 and extends up to 0x807f, data buffer one starts at 0x8080 and extends up to 0x80ff, etc. MEB and MYB are two of these buffers allocated from the buffer pool. The load to the first 64 bits of a data buffer (MYB or MEB) would miss in the cache and bring the entire cache block into the cache. We store the results of the merge back to the cache block containing MYB. At the end of the merge operation the protocol flushes the result block back to MYB buffer by issuing a writeback invalidate instruction similar to that in the MIPS R10000 [20] and then issues an uncached instruction to write the entire buffer contents back to the proper memory address where the MYB was loaded from originally. Note that we also need to invalidate the cache block holding the contents of MEB to make sure that during the next merge operation the protocol core/thread sees new data at this buffer address. For this purpose we use the invalidate instruction similar to that in the MIPS R10000. We note that the writeback invalidate as well as the invalidate instructions of the MIPS R10000 are allowed to be executed only in the privileged mode. However, since the coherence protocol software is provided by the designer and the user cannot download arbitrary protocol software in the protected protocol code/data area, we can allow this trusted protocol software to execute in the privileged mode.

The aforementioned design offers some benefits by eliminating a large number of slow uncached operations, but it still suffers from the problem that we cannot retain the merge results in the cache for reuse by multiple merge operations. Note that there could be multiple shadow blocks mapped to the same normal cache block and owned by different threads at the same time (each thread locally accumulates its portion). Therefore, a straight-forward optimization would be to directly bring the memory block to cache, instead of going through MYB, when the first merge operation to this block is invoked, and to not flush the block back to memory at the end of each merge operation. The coherence protocol can find out from the directory state when the last merge operation is executed for a cache block (the directory maintains an owner vector of the shadow block as in the original AM reduction protocol [16]), and only at that time does it issue a writeback to main memory. However, in this case we cannot cache the block at the original application space address because that would create a new coherence problem in the shared cache of SMTp or the protocol core in the PCSL2 and PCSL2PL1 configurations. The application may generate this address while merges are in progress and get a wrong intermediate value from the cache. To solve this problem, we cache the block at a different shadow address space. For example, if the result is being generated for application cache block address 0x2345680, we cache it at address 0xc2345680 where the bits [31:30] represent the address space (application address space is 0 while result block space is 3). Note that this does not require any extra hardware support because the protocol software itself generates the load/store instructions for the cache block and therefore, can issue the appropriate shadow addresses for loading the physical blocks. When the local memory controller receives such an address, it strips off the address space bits (as the request is generated by the protocol thread) and issues the cache block read request to SDRAM. At the end of all merges to a cache block the protocol writes back the block by issuing a writeback invalidate instruction. As we have already mentioned, on receiving this writeback, the memory controller strips off the space bits and sends the write request to the SDRAM banks. This design does increase cache pressure, but overall, we found it to deliver excellent performance.

In summary, we have introduced three orthogonal design options for moving the merge operation to protocol software, namely, caching (C) or not caching (U) MEB during a merge, caching (C) or not caching (U) MYB during a merge, and caching (C) or not caching (U) a merge result block across merges. This leads to a total of eight designs (UUU, UCU,. . . , CCC), but it is clear from the above discussion that UUC and CUC are not viable design options because there is no distinction between the merge results and the contents of MYB, if the merge results are cached across merges. We found that UCC and CCC are always superior to UUU, UCU, CCU, and CUU because UCC and CCC allow the merge results to be cached across merges. Interestingly, we found UCC to be slightly better than CCC. The uncached load path of MEB in UCC is faster than the cached path in CCC the first time around because a cache miss will go through an L2 lookup, which the uncached path can bypass. Further, caching the MEB increases cache pressure, thereby hurting performance in CCC. Therefore, for brevity we will present the results of UCC only. We found that caching the merge results across merges is the most important determinant of performance.

2.2 Matrix Transpose

Matrix transpose requires support for dynamic cache line assembly/disassembly in addition to the address translation support discussed in the last section. An incoming request from shadow space needs to gather the corresponding matrix elements from a column and assemble them into a cache block-sized data buffer. This process was depicted in Figure 2. This operation requires accessing $N = B/W$ cache blocks of normal physical space where B is the size of the requested shadow cache block (i.e. L2 cache block size) and W is the size of one matrix element. We assume W to be at most a quad-word i.e. 128 bits, which is the case for a complex double matrix. We further assume that the SDRAM interface accepts quad-word sized requests. Therefore, to replace the AMDU pipeline, the protocol software can issue a series of memory requests along with the destination data buffer indices i.e. the i^{th} data element will go to the i^{th} W-sized segment of the data buffer. The valid bit of each segment is memory-mapped and the protocol software can monitor the valid bits through uncached load operations and clear them through uncached store operations. Existing multiprocessors [17] already have this support to implement aggressive data pipelining between the DIMM interface and the memory controller. After implementing these modifications in the protocol software we can completely get rid of the AMDU, as we have already discussed how to move the address translation support to protocol software in the last section.

2.3 Complexity, Area, and Power Issues

In this section, we briefly discuss the issues related to the complexity, area, and power consumption of the AMDU, which we propose to get rid of. The AMDU datapath, at a very high level, was shown in Figure 3. However, many details related to the logic that synchronizes the AMDU and the protocol thread were not shown for brevity. We first discuss some of these here so that the readers can appreciate the need for substituting the AMDU with less complex designs. The AMDU was designed for a general address re-mapping technique where a shadow cache block is composed of words from multiple physical cache blocks, as in the matrix transpose technique. Therefore, multiple registers are needed in each pipeline buffer. A shadow cache block request is first sent to its home node. On this node the base addresses of the physical cache blocks are filled by the protocol thread in the base address registers speculatively even before consulting the directory entries. Such a design favors the common case where all the contributing cache blocks are clean in the home memory. The AMDU pipeline operates on the base addresses iteratively taking one address in each pass. Thus, a multiplexer in each pipe stage selects the appropriate register entry to operate on. The selection input of the multiplexer could be driven using a simple iteration counter, but the situation is complicated by the fact that under many circumstances, depending on the directory state of certain cache blocks, the protocol thread may instruct the AMDU to operate on a few selected registers in a buffer. Thus, with each pipeline buffer we must attach a valid bit vector of length equal to the number of registers in the buffer. The valid bits of any pipe stage can be set by the protocol thread and reset by that AMDU stage at the end of processing. This essentially means that after each iteration, each of the AMDU stages must inspect the valid bits to find out which register to work on in the next iteration. However, the protocol thread must be careful while setting the valid bits and filling the pipeline register contents because one can easily envision numerous races corrupting the computation. Another array of ready bits is needed with each buffer to indicate the completion of computation in a pipe stage. This bit is tested by the protocol thread before reading any buffer contents. From our experience, we found that functional verification of the AMDU even in a high-level simulator (which ignores many gate-level details) is quite tedious. This motivated us to explore simpler ways of implementing efficient address re-mapping.

The new coherence protocol extensions proposed in this paper remove the burden of verifying the AMDU, but does introduce extra burden of verifying the protocol software itself. However, we found that there are a few unique blocks of protocol code that get re-used at many places, thereby reducing the verification effort enormously. The biggest advantage of implementing address re-mapping in protocol software is that discovery of a late bug does not require a silicon respin. In summary, we feel that there are compelling reasons for believing that the design proposed in this paper is indeed complexity-effective.

Elimination of the AMDU saves area in the memory controller. Moving the re-mapping support to the protocol software does not add any extra hardware because the new protocol continues to run on the existing protocol thread hardware in SMTp or protocol core. Using a 65 nm process technology we estimate the area and the peak dynamic power consumption (without any clock gating) of the AMDU to be respectively 1.77 mm^2 and 2.81 W. These numbers do not include wiring area and static power. The wordline spacing, bitline spacing, register cell height and width, and various capacitance values are taken from Wattch distribution [2] and scaled down appropriately to 65 nm. The supply voltage, threshold voltage, and frequency are assumed to be 1.1 V, 0.18 V, and 2.4 GHz, respectively. All the arrays have one read port and one write port. All the array ports, except for the 1024-entry direct-mapped AMTLB and the DFCM7 [11] AMTLB prefetcher, are modeled as single-ended, as sense amplifiers are not necessary in the small arrays. The ALU/adder/shifter area is derived by scaling down the data published in [3] for the MIPS R10000 at 0.35 μm technology. We assume the area of an adder to be double of a barrel shifter. Further, we assume the peak dynamic power of the adder to be double of the shifter. The power consumption of the adder is computed from Wattch. As expected, we found that big portions of the area (about 60%) and peak power (about 60%) of the AMDU are consumed by the AMTLB and the AMTLB prefetcher. The majority of the remaining area and peak power is contributed by the merger and the address calculators. By examining Figure 3 we expect a significant additional wiring overhead between the SDRAM/protocol thread/protocol core interface and the AMDU. This may even necessitate re-engineering of the SDRAM to memory controller interface. As the dynamic peak power density of the AMDU is about 1.6 W/mm^2, future efforts will develop a temperature model of the AMDU and explore if elimination of the AMDU removes a potential hot-spot from the system.

3 Simulation Environment

In this section we discuss our simulation environment including the applications we use for evaluating our AMDU-free protocol extensions. We simulate DSM multiprocessors with 16 nodes where each node contains a heterogeneous dual-core processor. One core is in-order statically scheduled dual-issue and dedicated to protocol processing in non-SMTp models. This core is modeled after the embedded protocol processor in Memory And General Interconnect Controller (MAGIC) of the Stanford FLASH multiprocessor [10, 17]. The other core is dynamically scheduled out-of-order issue and simultaneous multi-threaded (SMT) [30, 31]. This core executes one or two application threads in addition to the protocol thread in the SMTp architecture. Thus in the SMTp architecture the protocol core is unnecessary (and hence is not shown in Figure 4(D)), while in non-SMTp models the protocol thread context is idle. We would like to mention that, other than a PC, a rename table, a return address stack, a branch history table (BHT) in the branch predictor, and an active list, no extra resource is provided when adding the protocol thread context in SMTp. In addition to the cores, each node contains an on-die integrated memory controller [5, 13, 15, 27, 28] clocked at core frequency, an integrated e-cube router clocked at core frequency, and off-chip SDRAM banks connected to the memory controller. Optionally, each node may contain an AMDU attached to the memory controller. The architectures with AMDU serve as the baseline in this study and we evaluate the performance after the AMDU is removed, while executing our proposed AM-enabled coherence protocols in software and taking advantage of address re-mapping. In Table 1 we present the architecture of the SMT core including the SMTp-specific reserved resources. The L2 cache round-trip time includes a 3-cycle tag look up time determined by CACTI 3.2 [12] for 65 nm process with a four-way banked organization. The number of reserved resources for SMTp is determined through extensive simulation. Note that these are not additional resources, but are just reserved from the existing pool of resources.

The memory system architecture is presented in Table 2. We simulate a 16-way banked 400 MHz SDRAM module and explore configurations with one logical channel capable of transferring 64 bits on both edges of the clock (DDR), the critical 64 bits being the first transfer packet. This leads to an aggregate bandwidth of 6.4 GB/s per channel.

Table 1. Simulated SMT core

Parameter	Value
Frequency	2.4 GHz
Thread contexts	2 app. + 1 protocol
Pipe stages	18
Fetch policy	ICOUNT (2 threads)
Front-end/Commit width	8/8
BTB	256 sets, 4-way
Branch predictor	Tournament (Alpha 21264)
RAS	32 entries (per thread)
Br. mispred. penalty	14 cycles (minimum)
Active list	192 entries (per thread)
Branch stack	48 entries
Integer/FP Register	224/224
Integer/FP/LS queue	48/48/64 entries
ALU/FPU	8 (two for addr. calc.)/3
Integer mult./div. latency	6/35 cycles
FP mult. latency	2 cycles
FP div. latency	12 (SP)/19 (DP) cycles
ITLB, DTLB	128/fully assoc./LRU
Page size	4 KB
L1 Icache	32 KB/64B/2-way/LRU
L1 Dcache	32 KB/32B/2-way/LRU
Unified L2 cache	2 MB/128B/8-way/LRU
MSHR	16+1 for retiring stores
Store buffer	32
L1 cache hit	3 cycles
L2 cache hit	11 cycles (round trip)
Reserved (SMTp specific)	
Front-end slots	1
Branch stack slots	1
Integer registers	16
Integer queue slots	12
LSQ slots	8
Store buffer	1
Bypass buffer	16 each for instruction and data

Table 2. Memory system

Parameter	Value
Memory controller frequency	2.4 GHz
System bus width	64 bits
System bus frequency	2.4 GHz
SDRAM access time	80 ns (row buffer miss) 40 ns (row buffer hit)
SDRAM bandwidth	6.4 GB/s
Router frequency	2.4 GHz
Hop time	10 ns
Link bandwidth	3.2 GB/s
Router ports	6 (SGI Spider)
Network topology	2-way bristled hypercube
Virtual networks	4 (AM protocol uses all)

The in-order dual-issue protocol core is clocked at the same frequency as the SMT core. We simulate multiple configurations for the protocol core differing mostly in the data cache organizations. The instruction cache of the protocol core is always 32 KB direct-mapped with 128-byte line size, backed by main memory, and accessible in a single cycle. We simulate four different configurations for data accesses. In the first two configurations the protocol core has a single level of private data cache backed by main memory. These configurations correspond to Figure 4(A). We simulate 128 KB direct-mapped and 2 MB 8-way set associative caches. The first size allows a single-cycle access at 65 nm (16-way banked as determined by CACTI 3.2 [12]) while the second size requires three cycles, but matches the amount of L2 cache space that the protocol thread shares with the application thread(s) in SMTp. We call these two configurations PCPL1_128KB (Protocol Core with Private L1 of size 128 KB) and PCPL1_2MB, respectively. In the third configuration the protocol core does not have a private data cache, but shares the L2 cache with the SMT core. This will be called PCSL2 (Protocol Core with Shared L2 cache). This configuration corresponds to Figure 4(B). In the fourth configuration the protocol core has a single-cycle 128 KB direct-mapped private L1 cache and shares the L2 cache with the SMT core. This configuration corresponds to Figure 4(C) and closely resembles that of a shared L2 cache dual-core processor, although the protocol core is much simpler than the SMT core. This configuration will be called PCSL2PL1 (Protocol Core with Shared L2 and Private L1 caches). In the AMDU-enabled architectures, the protocol core/thread communicates with the AMDU through uncached load/store operations. In these architectures, we use a 1024-entry direct-mapped TLB in the memory controller accessible in a single cycle at 65 nm (determined with CACTI 3.2 [12]).

The seven explicitly parallel applications used to evaluate our proposal are listed in Table 3. The first four applications

Table 3. Applications and problem sizes

Applications	Problem Sizes
DenseMMM	256×256 matrix
Spark98Kernel	64K×64K matrix, 1M non-zeros
SparseFlow	512K vertices and 1M edges
MSA	256×128K matrix
SPLASH-2 FFT	1M complex double points
FFTW	8192×16×16 cube
Transpose	1K×1K matrix

are used to evaluate parallel reduction while the last three use matrix transpose. DenseMMM carries out the computation $C = A^T B$ on square matrices A and B which is a special case of the level-3 BLAS [6] for matrix-matrix multiplication. The modified Spark98 Kernel [22, 23] parallelizes one call to LocalSMVP. SparseFlow computes a function on the in-flow of every edge incident on a vertex and sums up the function outputs as the net in-flux at each vertex in a sparse multi-source flow graph. MSA calculates the mean square average of the elements in every column of a matrix. All four applications use addition as the underlying reduction operation. All these applications are optimized with prefetching and application-directed page placement to reduce remote misses as much as possible.

SPLASH-2 FFT [32] and FFTW [8] are respectively 1D and 3D fast Fourier transform kernels (frequently used in multimedia and DSP applications), while Transpose is a microbenchmark which reads from and writes to a matrix and its transpose, and hence is highly memory-bound. In addition to prefetching and page placement, the parallelized versions of all these three applications are optimized with tiling and padding to improve cache utilization.

4 Simulation Results

We show results for 16 and 32-way threaded applications running on a 16-node DSM multiprocessor with each node capable of running one or two application threads in addition to the coherence protocol thread in SMTp. We present evaluations for architectures with AMDU as well as without AMDU. The architectures with AMDU run the baseline directory-based coherence protocol suitably extended with address re-mapping support as presented in [16]. The architectures without AMDU run either the baseline protocol or our proposed protocol that takes advantage of address re-mapping. In the evaluation we will focus on the following two questions.

- How much speedup does our protocol offer compared to running the baseline non-AM protocol without AMDU?
- What is the performance gap between our protocol running without AMDU and the previously proposed protocol [16] running with AMDU on otherwise similar architectures?

4.1 Parallel Reduction

We present the simulation results for MSA, DenseMMM, SparseFlow, and Spark98 in Figures 5, 6, 7, and 8, respectively. For each application we show two groups of bars, namely, one group for 16 threads and the other for 32 threads, both running on 16 nodes. The left five bars in each group (up to SMTp+AMDU) present results for existing systems while the right five bars present results for our proposal. The first two bars present the execution time for an architecture with a protocol core having a private data cache of size 128 KB and 2 MB, respectively, each running a non-AM conventional bitvector protocol. The next three bars show the results for AMDU-enabled architectures sporting either a protocol core (PCPL1_128KB and PCPL1_2MB) or a protocol thread (SMTp). These three architectures run the same cache coherence protocol extensions presented in [16] with some selective optimizations. The next five bars on the right (starting with SMTp+UCC) present the results of our best protocol UCC on different flexible directory controller architectures that do not rely on the AMDU. Recall that UCC uses uncached loads to access the data in message buffers and caches the merge results across merges. The SMTp+UCC bar shows the results of executing UCC on a protocol thread. The next two bars show performance of UCC when executed on a protocol core with a private data cache of size 128 KB and 2 MB (PCPL1_128KB and PCPL1_2MB). The last two bars present results of running UCC on a dual-core architecture with a shared L2 cache, one core being the protocol core. PCSL2 has a 3-cycle access latency for every cached load/store to the shared L2 cache while PCSL2PL1 enjoys the advantages of having a single-cycle reasonably large private L1 data cache. Each bar is broken down into four parts, namely, busy commit cycles, memory stall cycles, synchronization cycles, and other lost commit cycles. Cycle accounting is done at the commit stage of the core pipeline.

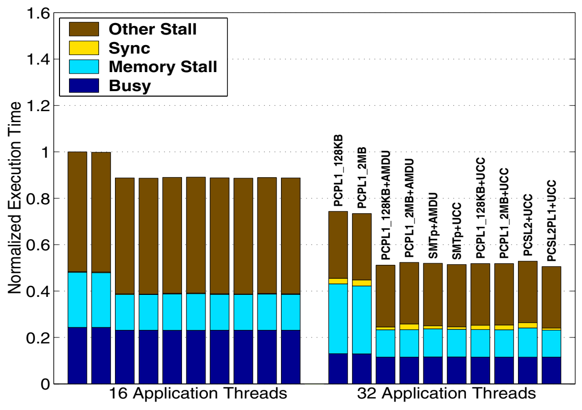

Figure 5. MSA using 16 and 32 threads.

In MSA (Figure 5) all architectures with UCC (the right five bars) are performing equally well when compared to the AMDU-enabled architectures (the left five bars, except the first two). All these architectures achieve a speedup of around 1.12 over the non-AM baseline (the leftmost two bars) for 16 application threads. Thus the removal of the AMDU in UCC does not lead to any performance loss. Further, the performance of the non-SMTp models is largely insensitive to the size of the cache provided to the protocol core. Finally, with 32 threads we observe fairly good scalability. Also, with 32 threads, the architecture with a protocol core sharing the L2 cache and having a private L1 data cache running our UCC protocol turns out to be the best (the last bar).

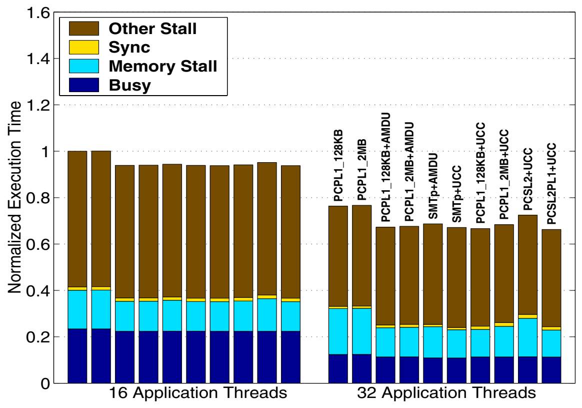

Figure 6. DenseMMM using 16 and 32 threads.

DenseMMM (Figure 6) shows similar performance variation across the design space as MSA. The architecture with a protocol core sharing the L2 cache and having a private L1 data cache running UCC (the last bar) performs the best and achieves a speedup of 1.07 on 16 threads and 1.15 on 32 threads compared to the non-AM baseline (the leftmost two bars). We also note that if the protocol core does not have a private L1 data cache, and is only allowed to share the L2 cache (as in PCSL2+UCC), the performance degrades considerably with 32 application threads due to slow cache hits (3 cycles). Interestingly, with 32 application threads, SMTp+UCC is 2.3% faster than a more complex SMTp+AMDU architecture. This is mostly due to caching of the merge results in UCC, which the existing protocol did not have [16].

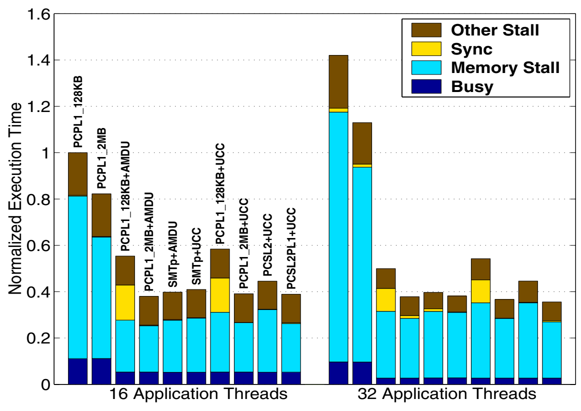

Figure 7. SparseFlow using 16 and 32 threads.

SparseFlow (Figure 7) shows wide variation in performance across the design space. The first interesting aspect of this application is that the cache size and organization of the protocol core affect performance significantly irrespective of the coherence protocol: PCPL1_2MB is 22% better than PCPL1_128KB, PCPL1_2MB+AMDU is 46% better than PCPL1_128KB+AMDU, and PCPL1_2MB+UCC is 49% better than PCPL1_128KB+UCC for 16 application threads. However, it is worth noting that SMTp+AMDU is only 2.9% better than SMTp+UCC, latter running our proposed protocol extensions. More interestingly, the architecture with a protocol core sharing the L2 cache with main core and having a private L1 data cache (the rightmost bar: PCSL2PL1) and running our UCC protocol delivers the best performance among all AMDU-free architectures. It achieves a speedup of 2.57 over a non-AM protocol core baseline (the leftmost bar: PCPL1_128KB) for 16 application threads and comes surprisingly close (within 2.4%) to PCPL1_2MB+AMDU. It is encouraging to note that PCSL2PL1 also has a lower area requirement compared to PCPL1_2MB+AMDU because PCSL2PL1 does not have an AMDU. Finally, we note that this application is not as scalable as MSA or DenseMMM. However, scalability improves significantly due to a large reduction in memory stall time after we employ the AM protocol.

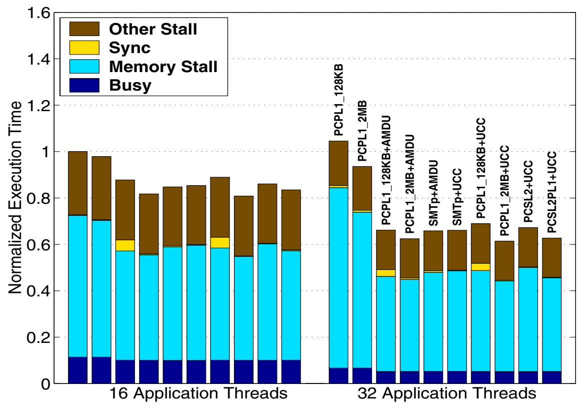

Figure 8. Spark98 using 16 and 32 threads.

Spark98 (Figure 8) repeats the trends of SparseFlow, although at a much lower variational scale across the design space. SMTp+UCC performs equally well as the AMDU-enabled design SMTp+AMDU. Also, PCSL2PL1+UCC performs close to PCPL1_2MB+AMDU. PCSL2PL1+UCC, having the large 2 MB L2 cache shared with the main core, achieves a speedup of 1.20 over a non-AM protocol core baseline (the leftmost bar) for 16 application threads. We note that this application shows better scalability than SparseFlow while moving to 32 threads.

In summary, the results are very encouraging. The architectures without an AMDU are delivering performance close (within at most 3%) to those with AMDU. The two most attractive architectural options are UCC protocol with SMTp (SMTp+UCC) and UCC protocol with a protocol core that shares the L2 cache with the main core and has private L1 caches (PCSL2PL1+UCC). Overall, these two architectures enjoy speedup of 1.45 and 1.49, respectively, with 16 application threads relative to a non-AM baseline having a protocol core with a 128 KB L1 data cache.

4.2 Matrix Transpose

Figures 9, 10, and 11 show the results for Transpose, FFT, and FFTW, respectively. The last five bars present the normalized execution time for our protocol (SoftTr), which does not require the special AMDU hardware. The first five bars are same as discussed above for parallel reduction.

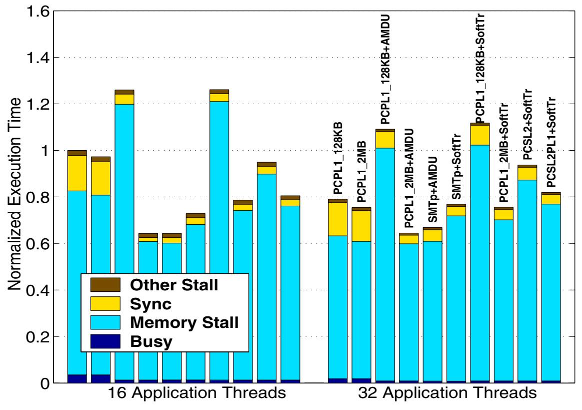

Figure 9. Transpose using 16 and 32 threads.

The first interesting observation we make in Transpose microbenchmark (Figure 9) is that even with the AMDU hardware and a private 128 KB L1 data cache, the protocol core cannot derive any benefit from the existing AM protocol (see PCPL1_128KB+ AMDU). However, with a significantly larger 2 MB 8-way dedicated cache, an AMDU delivers 51% better performance (see PCPL1_2MB+AMDU) compared to a similar non-AM baseline without AMDU (PCPL1_2MB). This is because of a large number of directory conflict misses in PCPL1_128KB+AMDU. On the other hand, SMTp+AMDU performs as well as PCPL1_2MB+AMDU because the protocol thread enjoys the advantage of having access to the large 2 MB L2 cache shared with the application threads. The most interesting result is that SMTp+SoftTr, executing our proposed coherence protocol extensions on SMTp, delivers performance within 13.2% of SMTp+AMDU. The remaining 13.2% performance gap results mostly from the absence of efficient data pipelining present in the AMDU. However, SMTp+SoftTr still achieves 1.37 speedup compared to a non-AM baseline (the leftmost bar) on 16 application threads. Interestingly, the protocol core architecture with a dedicated 2 MB L1 data cache running SoftTr (PCPL1_2MB+SoftTr) performs worse than SMTp+SoftTr, although the former has an 8-way 2 MB protocol core data cache. This mostly happens due to slow cache hits (all hits are 3 cycles). Similar trend is observed in the protocol core architecture with L2 cache shared with the main core running SoftTr (PCSL2+SoftTr), which continues to be much worse compared to SMTp+SoftTr due to the combined effect of slow cache hits and sharing of cache space with application threads. Surprisingly, SMTp+SoftTr beats the protocol core architecture with private L1 and shared L2 caches (the rightmost bar) by 10.6% on 16 application threads. This happens due to a subtle difference in LRU behavior in the L2 cache in these two configurations. In SMTp+SoftTr the L1 cache is shared among the protocol and application threads leading to a sizable number of L1 protocol misses. This keeps the L2 ways occupied by protocol data more frequently and recently used. In PCSL2PL1+SoftTr the L1 cache is private and suffers from a miss less frequently. But since the L2 cache is shared, this increases the chance of the protocol ways becoming LRU victims.

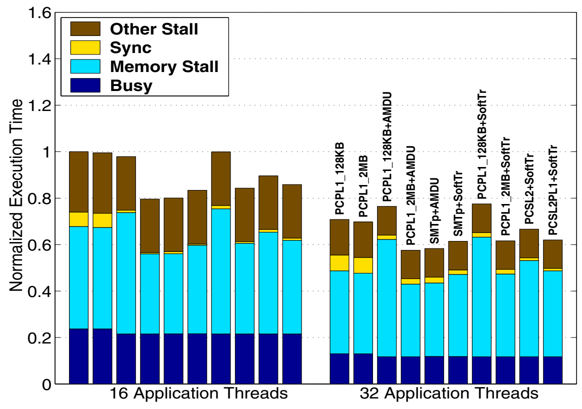

Figure 10. SPLASH-2 FFT using 16 and 32 threads.

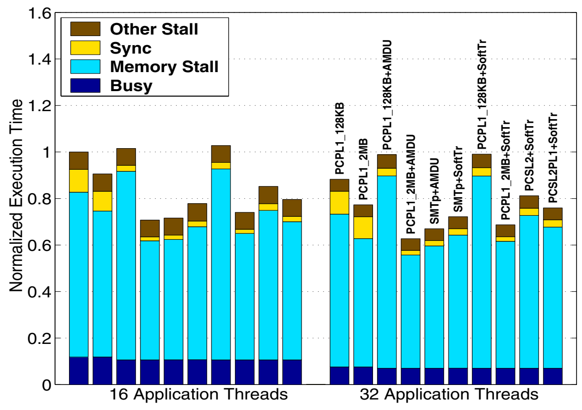

Figure 11. FFTW using 16 and 32 threads.

FFT (Figure 10) and FFTW (Figure 11) continue to show similar trends as Transpose. However, the performance gap between SMTp+AMDU and SMTp+SoftTr is much smaller in FFT (within 4.1%) and FFTW (within 8.7%) compared to

Table 4. Summary of results for 16-threaded executions

AM Technique	Average AMDU-free speedup		Max. gap	Mean gap
	SMTp (Figure 4(D))	PCSL2PL1 (Figure 4(C))		
Reduction (UCC protocol)	1.45	1.49	3.0%	1.5%
Transpose (SoftTr protocol)	1.29	1.23	13.2%	8.7%

Transpose (within 13.2%). This is a good news because Transpose represents the extreme memory-bound behavior with practically no computation, which in reality should be rare. Overall, across these three applications SMTp+SoftTr delivers the best performance among all AMDU-free architectures and achieves a speedup of 1.29 on average compared to a non-AM architecture employing a protocol core with a 128 KB direct mapped data cache (the leftmost bar) for 16 application threads. The dual-core PCSL2PL1+SoftTr design with shared L2 cache and private L1 data cache comes close (within 2.3% in FFT and FFTW, and 10.6% in Transpose) to this performance. Finally, with 32 application threads the performance trends are very similar to those with 16 application threads. We find that Transpose is the least scalable of the three benchmarks (as expected) while FFT shows very good scalability with FFTW following closely.

We summarize the salient results for 16-way threaded executions in Table 4. In this table we answer the two key questions that we started with for each of the two AM techniques. We present the speedup of two most attractive AMDU-free architectures running our protocols when compared to the non-AM baseline PCPL1_128KB. We also present the maximum and average performance gaps between the AMDU-free architecture and a similar architecture with AMDU. We conclude that our proposal is highly successful in reducing the complexity of the active memory systems while delivering performance comparable to the existing complex systems that use the AMDU.

5 Related Work

The Impulse memory controller [33] used the address remapping technique and showed how to improve cache locality of matrix transpose and sparse matrix-vector product on single-threaded systems. This architecture employed a custom-designed memory controller for doing address remapping e.g., address translation and dynamic cache line assembly/disassembly, and relied on application-directed cache flush calls for maintaining coherence between shadow and physical spaces. Subsequently, the Active Memory architecture [16] made the coherence maintenance completely transparent to the compiler and extended conventional cache coherence protocols to handle this while continuing to rely on a custom-designed active memory data unit (AMDU) embedded in the off-chip memory controller. This naturally enabled seamless extension of address re-mapping to symmetric multiprocessors (SMPs) and DSM clusters. Further, this design employed a flexible off-chip coherence protocol processor to handle an array of AM techniques e.g., matrix transpose, sparse matrix-vector product, linked list linearization, and parallel reduction. The parallel reduction technique employed in this paper was first proposed in a non-AM context [9] and that architecture relied on application-directed cache flush calls for maintaining coherence between the two spaces. In this paper we have shown how to efficiently carry out coherent address re-mapping without any custom hardware in the memory controller.

The DIVA [7], Active Pages [24], and FlexRAM [14] projects, instead of using address re-mapping, implement active memory elements to improve memory performance. These architectures add processing capabilities to memory chips, thereby creating so-called PIMs. The MAUI effort [29] integrates the concepts of DIVA, Active Pages, and user-level memory threads [26] into a single design. All these designs are very different from our proposal which employs flexible address re-mapping to improve locality of cache access.

Flexible directory controller architectures have been explored in academia (Stanford FLASH multiprocessor [17], Wisconsin Typhoon [25]) as well as in industry prototypes (Compaq Piranha [1], Sequent STiNG [19], Sun S3.mp [21]). However, none of these studies explore the performance impact of on-die integrated protocol engine and the cache organizations of the protocol engine as we do in this paper in the context of the AM protocols. A cost-effective architecture for carrying out flexible coherence processing was presented in the SMTp study [4], which does not require extra cache area or protocol core area. In this paper we have explored an array of flexible directory controller organizations in the context of today's multi-threaded and dual-core environments and presented a thorough quantitative evaluation of these organizations.

6 Summary

For the first time, we have shown how to achieve flexible and hardware-coherent address re-mapping without custom-designed hardware in the memory controller. We have evaluated our protocol extensions for parallel reduction and matrix transpose kernels. Our experiments show that for parallel reduction, our best coherence protocol running on a hardware thread context in SMTp achieves, on average, 1.45 speedup relative to a non-AM baseline design and delivers performance within 3% of an SMTp design with custom AM hardware support with 16 application threads. Our experiments also discover efficient low-complexity alternatives. Specifically, an architecture with a simple dedicated in-order static dual-issue protocol core having a private L1 data cache and sharing the L2 cache with main core achieves a speedup of 1.49 relative to a non-AM baseline. However, this alternative may not be as attractive as SMTp, given the extra area overhead of the protocol core. What is worth noting is that none of these architectures require custom AM support in the memory controller, as the existing proposals do.

We have evaluated our matrix transpose protocol extensions on a microbenchmark and 1D and 3D fast Fourier transform kernels. The results show that SMTp, running our protocol on a hardware thread context, achieves, on average, a speedup of 1.29 over a non-AM baseline and comes within

13.2% of SMTp equipped with custom AM memory controller. The architecture with a protocol core having a private L1 date cache and sharing the L2 cache of main core comes within 5%, on average, of SMTp performance.

In summary, we have presented AM protocol extensions that do not require any custom hardware support in the memory controller. Executing these protocols on either a thread context in SMTp or a core with private L1 and shared L2 caches in a CMP delivers competitive performance for parallel reduction and matrix transpose, two widely used application kernels. We believe that this study opens up new opportunities for complexity-effective active memory implementations on single-threaded as well as multi-threaded systems.

References

[1] L. A. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282–293, June 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[3] J. Burns and J-L. Gaudiot. SMT Layout Overhead and Scalability. In *IEEE Transactions on Parallel and Distributed Systems*, **13**(2): 142–155, February 2002.

[4] M. Chaudhuri and M. Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 124–135, June 2004.

[5] Z. Cvetanovic. Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 218–228, June 2003.

[6] J. J. Dongarra et al. "A Set of Level 3 Basic Linear Algebra Subprograms". *ACM Transactions on Mathematical Software*, **16**(1):1–17, March 1990.

[7] 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.

[8] 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, May 1998.

[9] 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.

[10] 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.

[11] B. Goeman, H. Vandierendonck, and K de Bosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[12] HP Labs. CACTI 3.2. Available at *http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html*.

[13] R. Kalla, B. Sinharoy, J. M. Tendler. IBM Power5 Chip: A Dual-core Multithreaded Processor. In *IEEE Micro*, **24**(2): 40–47, March-April 2004.

[14] Y. Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the International Conference on Computer Design*, pages 192–201, October 1999.

[15] C. N. Keltcher et al. The AMD Opteron Processor for Multiprocessor Servers. In *IEEE Micro* **23**(2):66–76, March-April 2003.

[16] D. Kim et al. Architectural Support for Uniprocessor and Multiprocessor Active Memory Systems. In *IEEE Transactions on Computers*, **53**(3):288–307, March 2004.

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

[18] 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.

[19] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–317, May 1996.

[20] MIPS/SGI. R10000 Microprocessor User's Manual.

[21] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, Vol. 1, pages 1–10, August 1995.

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

[23] 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.

[24] 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*, pages 192–203, June 1998.

[25] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34–43, May 1996.

[26] Y. Solihin, J. Lee, and J. Torrellas. Using a User-level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 171–182, May 2002.

[27] Sun Microsystems. An Overview of UltraSPARC III Cu. White Paper, September 2003. Available at http://www.sun.com/processors/whitepapers/USIIICuoverview.pdf.

[28] Sun Microsystems. UltraSPARC IV Processor Architecture Overview. White Paper, February 2004. Available at http://www.sun.com/processors/whitepapers/us4_whitepaper.pdf.

[29] J. Teller, C. B. Silio, and B. Jacob. Performance Characteristics of MAUI: An Intelligent Memory System Architecture. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Memory Systems Performance*, June 2005.

[30] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[31] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

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

[33] 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.