Preliminary Investigation of Active Memory Operations

Lixin Zhang, Zhen Fang, John B. Carter, Mike Parker

UUCS-04-009

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

June 1, 2004

Abstract

We are rapidly approaching a time when large-scale shared memory supercomputers will have remote memory latencies measured in the thousands of cycles and cross-section bandwidth will be a limiting performance factor. For these machines to scale, mechanisms that minimize interprocessor communication will be essential. We propose one such mechanism, *active memory*, which allows operations to be sent to and executed on the home memory controller of particular data items. Performing the operations near where the data resides, rather than moving it across the network, operating on it, and moving it back, eliminates significant network traffic, introduces opportunities for additional parallelism, and hides high remote memory latencies. Active memory provides many of the benefits of PIMs without the need for non-standard DRAMs, and enables significantly better application scaling than conventional shared memory synchronization and range operations.

In this paper we investigate an active memory design that supports three classes of memory-centric operations that benefit common parallel constructs: *atomic scalar*, *range*, and *reduction* operations. We present architectural and programming models for active memory and compare its performance against a baseline conventional shared memory system implementation and a variety of optimized memory architectures. We find that active memory easily outperforms the conventional shared memory and other architectures by factors of over 10x on a collection of parallel constructs.

1 Introduction

There is an increasing demand for high performance scalable distributed shared-memory (DSM) systems from both government agencies and industrial users. DSM systems distribute physical memory across the nodes in the machine and implement coherence protocols to provide the illusion of shared memory. The predominant scalable DSM architecture is directory-based CC-NUMA (cache-coherence non-uniform memory access), such as employed in the SGI Origin™line [20]. In such architectures, each block of memory is associated with a fixed *home node*, which maintains a directory structure to track the state of all locally-homed data. When a process accesses data that is not in a local cache, the local DSM hardware sends a message to the data's home node to request a copy. Depending on the block's state and the type of request (read or write), the home node may need to send messages to additional nodes to service the request and maintain coherence. The round trip network latency of large-scale DSM machines will soon be thousands of processor cycles, as processor speeds increase while wall time network latency improves very little due to speed of light effects. In addition, cross-section bandwidth will be a limiting factor in the scalability of large-scale (100+ node) DSM systems. The unavoidable conclusion is that eliminating remote coherence traffic will be essential for DSM systems to scale effectively.

Caching is the standard mechanism for eliminating remote traffic and improving local memory performance. A serious limit to the scalability of typical CC-NUMA DSM architectures is that the amount of remote data that a node can replicate locally is limited to the size of its caches. In addition, coherence typically is maintained at the block-level (e.g., 128 bytes), so entire blocks are moved across the network or invalidated, even when the processor needs or modifies only a single word from the block. The appalling truth is that the sustained performance of large DSM supercomputers for many applications is less than 5% of peak performance, due largely to memory system performance, and this trend is expected to worsen[17]. This problem is particularly serious for data-intensive, highly parallel scientific and commercial applications. The work presented here represents an effort to alleviate memory system bottlenecks for these types of applications.

We propose to add an **Active Memory Unit** (AMU) capable of performing a select set of simple operations to DSM memory controllers. AMUs let processors operate on remote data at the data's home node, rather than loading it into a local cache, operating on it, and potentially flushing it back to the home node. We call AMU-supported operations **Active Memory Operations** (AMO), because they make the conventional "passive" memory controller more "active". The goal of AMOs is to reduce the number of cache misses, reduce cache pollution, hide the high latency of remote memory accesses, and reduce network traffic, thereby improving DSM performance. AMOs are particularly effective for operations with low temporal locality (i.e., little reuse), where caching provides little or no benefit.

Performing data operations in the memory controller provides a number of benefits. For remote memory accesses, data does not need to be moved across the network to the requesting node and, in the case of writes, moved back to the home node after modification. For local memory accesses, data does not need to be moved to/from the local processor caches via the system bus. Most AMO request/response messages are significantly smaller than a cache line, which reduces network bandwidth requirements. Finally, AMUs can implement atomic memory operations efficiently since they are co-located with the home directory of the data on which the operations are being performed. Compared to active messages [26], AMOs do not interfere with useful work being done by the processor co-located with the data, and have much lower invocation overhead than the

interrupt required to invoke an active message handler.

In the rest of the paper, we present architectural and programming models for a simple active memory unit and compare its performance against conventional shared memory and active messages. Our simple AMU supports three classes of operations: *scalar* (e.g., `fetch-and-add`), *range* (e.g., `memset`), and *reduction* (e.g., `max`) operations. We evaluate six benchmarks using an execution-driven simulator and find that active memory outperforms conventional shared memory and active messages by large margins for these benchmarks (ranging from 1.8X to over 80X).

2 Background

Avoiding remote communication has been a major focus of message passing programmers for years. Sending computation to data, as opposed to pulling data to computation, is a common technique for eliminating messages. Active Messages are an effective way to send computation to data [26]. An active message includes the address of a user-level handler to be executed upon message arrival, with the message body as its argument. This mechanism enables message passing programmers to move computation to data fairly efficiently. However, the receiving processor is interrupted to handle an incoming active message, which causes it to stop what it is doing, flush its instruction pipeline, and switch to the message handler. Moreover, the message handler must load instructions and data into the local caches, potentially evicting useful data or instructions in the process. These side effects of invoking an active message can interfere with ongoing local computation. While active messages are very effective at reducing communication traffic, using the node's primary processor to execute handlers incurs non-trivial handler startup overhead and interferes with useful work being done on that processor.

In recent years, several researchers have proposed adding intelligence to the memory controller to overcome the "Memory Wall" [28, 23]. For example, the Impulse memory controller [28] uses an extra level of physical address remapping to increase or create spatial locality for stride and random accesses. Solihin *et al.* [23] add a general-purpose processor core to a memory controller to direct prefetching into the L2 cache.

Processor-in-memory (PIM) systems incorporate processing on modified DRAM chips to exploit the very high memory bandwidth and relatively low latency available inside the DRAM package [6, 14, 15, 24]. PIMs are useful in a variety of domains, e.g., systems dedicated to supporting dense streaming applications or embedded systems where all the required data resides within a single DRAM chip. However, once the data needed to perform a computation crosses a chip boundary, these systems effectively become a specialized form of distributed memory multiprocessors, with all of the attendant complexities. Much research remains to determine the extent to which PIMs can be integrated into a general-purpose computer environment. Also, the performance of PIM architectures is hindered by the fact that DRAM processes are slower than logic processes. In contrast, AMOs achieve many of the performance benefits of PIMs without their attendant complexity and cost. Systems that offload remote computation to AMUs, rather than PIMs, can use commodity DRAMs, with their higher yields, higher density, and lower per-byte costs.

The SGI Origin 2000 [10] and Cray T3E [19] implement a set of memory-side atomic operations (MAOs) in the memory controller that are triggered by writes to special IO addresses. MAOs are non-coherent and rely on software to maintain coherence. They are used primarily to support efficient thread synchronization. However, basic MAO-based spinlocks typically involve uncached loads to spin on synchronization variables, so each spin request reloads data from the home

node. To avoid this high overhead, some researchers suggest spinning on a separate cacheable variable [13]. However, with this design, when the previous lock holder frees the lock, it must send an invalidation request to every processor spinning on the lock, which must then reload the cacheable data, perform an uncached read on the MAO-based lock, and if unsuccessful resume spinning. The net result is that MAO-based locks outperform conventional atomic locks, but still require non-trivial amounts of communication.

The NYU Ultracomputer [5] implements a variety of atomic instructions in the memory controller. It uses a combining network that attempts to combine loads and stores for the same memory location within the interconnect. The hardware cost for queueing circuitry at each node is high and there is a performance penalty for references that do not exploit combining.

Off-loading the task of synchronization from the main processor to network processors is an approach taken by several recent clusters [16, 25]. For example, the Quadrics^{TM}QsNet interconnect [16] used by many current large clusters supports both pure hardware barriers and network processor-based hybrid barriers.

Garzaran *et al.* [4] propose to enhance directory controllers with execution units for parallel reduction operations. Their work focuses mainly on the *merge* phase for small to medium sized data sets, while AMO reduction operations are used primarily for the *partial reduction* phase of large data sets. The techniques are complementary and both demonstrate for the value of offloading select operations to the memory system.

3 Active Memory Operations

In this section, we present the hardware organization of an AMU (Section 3.1), a description of how AMOs can be invoked and used by software (Section 3.2), and a description of the operations supported by our current AMU design (Section 3.3).

3.1 Hardware Organization

Figure 1 presents a block diagram of the major components of a single node of our AMU-enhanced DSM architecture, which is modeled on the SGI Origin 3000 architecture [20]. A crossbar connects processors to the network backplane, from which they can access remote memory, local memory, and IO. In our simulation timing models, we assume that the processor(s), crossbar, and memory controller are all on the same die, as will be typical in near-future system designs.

To support AMOs, each MMC is extended to include a small number (4 to 8) AMU control and data registers. The AMU registers are external to the processor and are akin to IO control registers. To invoke an AMO, the local processor performs a 64-bit IO-space write to an AMU address register and then a second IO-space write to the corresponding AMU control register. These two writes fully encode the information required to perform an AMO. The second write implicitly initiates the AMO. The AMU data register is also used to store and return values associated with the AMO. The processor initiating the AMO can use IO-space reads to a DONE bit in the control register to poll whether the operation has completed. This IO-register-based interface for initiating remote memory operations is similar to that employed by the Cray T3E to implement E-registers[19]. AMO performance could be further improved if the processor ISA were extended to include AMO instructions, but this is not necessary to exploit the active memory idea.

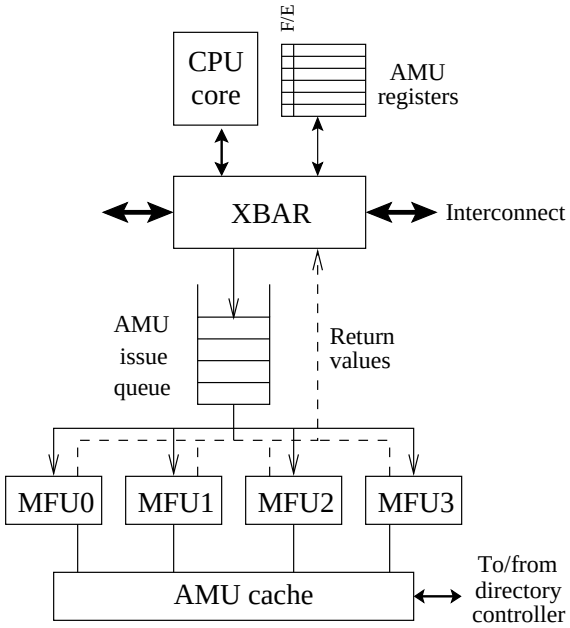

Figure 1. Organization of the Active Memory Unit

The decoupled (asynchronous) AMO model enables AMOs to be overlapped with other useful work, thereby hiding the high roundtrip message latency. The operating system is responsible for allocating AMU registers to individual programs, and saving/restoring register state as necessary to virtualize the AMU registers. A program is responsible for managing its AMU registers to ensure that the results of previous AMOs are not overwritten before they are consumed.

When a processor initiates an AMO, the local AMU examines the target virtual address (or address range), translates it to a global physical address, and sends an AMO message to the AMU on that address's home node. When the AMO message arrives at that node, it is placed in an in-order AMU instruction queue awaiting dispatch to one of a collection of identical active memory function units (MFUs). Address translation is performed via an external TLB located on the MMC, along the lines of the external TLBs supported by the Cray T3E[19] or the Impulse memory controller[28].

Each MFU contains control logic and an ALU unit that can perform the 32- and 64-bit logical, integer, and floating point operations as described in Section 3.2. To implement scalar AMOs, the MFU loads a coherent word of data, performs the requested operation, writes the result back to memory, and (optionally) returns a result to the requesting processor. For range and reduction AMOs, the MFU streams a contiguous range of coherent data through an ALU pipeline, writes the result back to memory (for range AMOs), and (optionally) returns a scalar result to the requesting processor. As described in Section 3.3, AMO range operations are restricted to ranges that fall entirely within a single memory controller's local memory.

To overlap computation with potentially high latency DRAM accesses, each MFU can track the state of multiple outstanding memory references, either from a single range/reduction AMO or multiple scalar AMOs. This support is enabled by a small queue and associated state machine in each MFU. The optimal number of MFUs per AMU and outstanding memory operations per MFU are open questions, and likely highly dependent on the performance characteristics of a particular

DSM architecture. For our experiments, each AMU has four MFUs and each MFU can service four outstanding memory requests at a time.

For certain uses, AMOs exhibit very high locality between independent requests, e.g., `fetch-and-add`s exhibit high temporal locality when used to implement scalable synchronization. To improve the performance of these operations, there is a tiny 32-entry, four-way set associative cache in the AMU. This AMU cache is banked, with as many banks as there are MFUs. The AMU cache stores intermediate result values.

AMUs generate coherent memory requests that are sent to the directory controller, which checks to see if the copy of the data in local memory can be used directly (e.g., it is a read operation and the local data is coherent or a write operation and there are no sharers). The directory controller itself uses a multi-banked directory cache to cache directory state. If necessary, the directory controller performs the necessary operations to make the data coherent (e.g., issue invalidate or flushback requests to remote sharers).

Finally, some active memory operations return a scalar value to the requesting processor, so there is a return data path back from the AMU to the crossbar.

The hardware implementation of the AMU consist of 4 MFUs and a 32-line cache. Each MFU contains a 64-bit integer adder, a 64-bit floating-point adder, a small state machine, a 64-bit pop-count circuit, and handful of MUXes and registers. In a 90nm process, the entire AMU consumes approximately 0.4mm^2, less than 0.3% of the total die area of a high-performance microprocessor with an integrated memory controller. This small chip overhead seems worthwhile given the substantial performance benefits.

3.2 Using AMO

Since an AMO can operate on a single scalar value or a range of values, it may take thousands or tens of thousands of cycles to complete. To avoid processor stalls, AMO operation issue and completion are decoupled. AMO operations are initiated using I/O writes to user-accessible AMO registers in the local memory controller. The status of outstanding AMO operations can be checked using I/O space reads. When an AMO operation issues, the triggering I/O write clears the F/E bit for the specified destination AMU register and then issues the AMO to the appropriate AMU. When the result returns, it is placed in the specified AMU register and the corresponding F/E bit is set. The initiating process can test the AMU register's F/E bit to determine when the data is available for consumption. AMU registers are like IO registers in many respects.

Because AMOs are decoupled operations, destination AMU registers must remain available even if the issuing process has been context switched off the processor. This requirement is similar to the need to pin pages that are the target DMA operations for the duration of the DMA. To provide this functionality and ensure security, AMU registers are virtualized. In our current design, there are eight architecturally visible AMU registers. A hardware table maps these virtual AMU registers to a larger number of physical AMU registers (e.g., thirty-two). When a process issues its first AMO operation, the processor traps to the operating system. In response, the OS selects a block of unallocated physical AMU registers and maps these physical AMU registers into the process's address space. If there are no free physical AMU registers, the OS can choose to "swap out" a block of physical AMU registers for any other process for which there is not an outstanding AMO. Swapping entails saving the register values to the swapped out process's control block; they will need to be swapped back in via an analogous operation when that process is rescheduled. With this

design, the target AMU register for any outstanding AMO will always be available when the data returns, and a modest number of physical AMU registers will suffice to support a large number of concurrent processes.

3.3 Supported Operations

In general, the active memory unit supports three types of operations: scalar operations that process data elements at specified memory locations, range operations that process data elements within contiguous physical memory areas, and reduction operations that operate on contiguous ranges of addresses.

Scalar operations perform atomic arithmetic, logic, or floating-point operations on individual words of data. The current design supports an atomic write/swap operation, and arithmetic operations that add, subtract, multiply, etc., a constant to an integer or floating-point value stored at the specified memory address. The AMU control register encoding includes a bit to indicate if the resulting value should be returned and (if so) the target physical AMU register where the result should be stored on the requesting processor.

To support scalable barriers, we provide a special `increment` operation that increments the target memory location and returns the old value to the requesting processor. What makes our AMO `increment` operation special is the inclusion of a "test" value, which is compared against the result of the increment. If the incremented value matches the test value, e.g., as would be the case in a barrier operation if the test value contained the total number of processors expected to reach the barrier, the AMU sends an update command along with the new value to the local directory controller. The directory controller then sends the latest value to every node that has a copy of the data. [1]

Spinlocks are often used to ensure exclusive access to critical data. AMOs can be used to implement efficient spinlocks using atomic *fetch_and_op* operations. Unlike MAO-based spinlocks, AMO-based spinlocks inherently involve local spinning (on AMU registers) and the AMO implementation allows lock signaling to be implemented efficiently via the above-described update mechanism.

AMOs are far more efficient for synchronization than load-linked/store-conditional (LL/SC) instructions that are available on most modern systems [3, 8, 11]. LL instructions load a block of data into cache. Subsequent SC instructions attempt to write to the same block, but succeed only if there was no interceding write to the block since the previous LL. To achieve atomicity, library routines typically retry the LL/SC pair repeatedly until the SC succeeds.

Unfortunately, in large multiprocessor systems, there is often high contention for synchronization variables, which results in significant coherence traffic and high latency. The problem gets worse as the number of processors contending for a barrier variable increases, so high-performance barrier implementations typically employ *barrier trees* [27, 18], where only a small number of nodes (e.g., 16-32) contest for a given barrier variable at a time. Many optimized spinlock algorithms have been proposed [12]. These algorithms typically introduce local variables and complex data structures to alleviate hot spots. Even these efficient synchronization implementations suffer from frequent interference and remote memory latencies. AMU-based synchronization centralizes the atomic operations in the lock's home node, which eliminates interference. If the target of the

[1] Efficient update is enabled by the fact that our system supports a fine-grained update protocol, details of which are beyond the scope of this paper.

AMO hits in the AMU cache, the operation takes only two cycles to complete, independent of the number of nodes contending for the synchronization variable. We discuss the performance implications of this design in detail in Section 5.

Range operations operate on ranges of contiguous addresses. To eliminate the need for cross-node communication as part of an AMO, our current design requires ranges to be entirely resident on a single node. We are exploring mechanisms to eliminate this restriction should we determine that there are sufficient application needs for it (e.g., to perform an arbitrary vector-vector add or `memcpy`). Range operations perform the same scalar operation on every element in the range, e.g., vector-scalar add, vector-scalar mult, or `memset`.

Reduction operations perform a reduction operation on a contiguous range of addresses. Like range operations, the range on which the reduction is performed must be fully contained in a single memory controller. The supported operations include computing the sum (`sum`), finding the maximum (`max`) or minimum (`min`) value, counting the number of elements matching with a given key (`count`), and counting the number of bit 1's (`popcount`) in the specified memory area. These operations can also provide a stride as an argument to instruct the AMO to examine data separated by a constant stride.

Reduction operations are important in many scientific codes and commercial workloads. Examples include telecommunication logs, sensor data, financial tickers, and code breaking. Reductions over extremely large ranges can be parallelized by performing a series of partial reductions on smaller ranges, e.g., ones that fall entirely within a single memory controller, the results of which are themselves reduced to generate the final reduction result. Our experimental results show that AMOs can significantly lower the time required to do partial reductions, thereby improving overall performance.

We provide a set of library functions to give programmers access to all active memory operations. We are investigating ways to enable compiler automation. Native compiler support for AMOs are not currently available, so we manually insert AMOs into the benchmark programs using simple macros for each AMO.

4 Simulation Environment

We use an execution-driven simulator, UVSIM, in our performance study. Our simulator models a hypothetical next-generation Origin 3000 supercomputer, including a directory-based coherence protocol [21]. Each simulated node contains two MIPS next-generation microprocessors connected to a high bandwidth bus. Also connected to the bus is a future-generation hub [22] that integrates the processor interface, memory controller, directory controller, network interface, and IO interface. Each node contains a DRAM backend with a maximum of 16GB of physical memory. We simulate a micro-kernel that supports most common Unix system calls, and directly execute statically linked 64-bit MIPS-IV executables. Our simulator supports the OpenMP runtime environment, and all of the parallel benchmarks discussed in this paper are OpenMP programs.

Table 1 lists the major parameters of the simulated system. The L1 cache is virtually indexed and physically tagged. The L2 cache is physically indexed and physically tagged. The DRAM backend has 16 20-bit channels connected to DDR DRAMs, which enables us to read an 80-bit burst every two cycles. Of each 80-bit burst, 64 bits are data. The remaining 16 bits are a mix of ECC bits and partial directory state.

Parameter	*Value*
Processor	4-issue, 48-entry active list, 2GHz
Node	2 processors w/ shared hub and DRAM
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
System bus	16B CPU to system, 8B system to CPU
	max 16 outstanding references, 1GHZ
DRAM	16 16-bit-data DDR channels
Hub clock	500 MHz
Memory latency	60 processor cycles
Network latency	100 processor cycles per hop

Table 1. System configuration.

Benchmark	*Description*
barrier	multiprocessor barrier operation
spinlock	multiprocessor spinlock operation
gups	random global updates
memset	fill a range with a constant value
max	find the maximum value of a range
popcount	count the 1 bits in a range

Table 2. Benchmarks

The simulated interconnect subsystem is based on SGI's NUMALink-4. The interconnect is built using a fat-tree structure, where each non-leaf router has eight children. The minimum network packet is 32 bytes. We do not model contention within the routers, but do model port contention on the hub network interfaces.

We have extended the simulator to support active messages and processor-side atomic instructions (like the semaphore instructions available on Intel^{TM}Itanium2 [7]). AMOs and active messages share the same programming model and are initiated the same way; the only difference is the way each is handled by the receiving node. The overhead of active messages on the recipient processor is accurately modeled, but we ignore the operating system overhead of active messages on the initiating processor. The remote processor is interrupted when an active message arrives, which causes it to flush the instruction pipeline and start executing the active message handler. When the handler exits, the interrupted process is resumed.

We have validated the core of our simulator by configuring its parameters to match those of an SGI Origin 3000, running a large mix of benchmark programs on both a real Origin 3000 and the simulator, and comparing performance statistics (e.g., run time, cache hit rates, etc.). All simulator-generated statistics are within 15% of the corresponding numbers generated by the real machine, most within 5%.

Table 2 lists the six benchmarks that we use during our performance evaluation. The benchmarks are compiled using the MIPSpro Compiler 7.3 with an optimization level of "`-O2`".

Nodes	Speedup over baseline			
	Atomic	ActMsg	MAO	AMO
2	1.03	0.73	1.29	**1.93**
4	1.13	1.57	4.55	**8.68**
8	1.17	1.40	5.53	**12.06**
16	1.06	1.28	4.50	**14.16**
32	1.19	1.62	5.46	**27.34**
64	1.21	1.74	7.51	**37.43**
128	1.18	1.83	11.70	**54.82**

Table 3. Barrier performance, 2 processors per node.

Nodes	Cycles per processor				
	OpenMP	Atomic	ActMsg	MAO	AMO
2	176	171	242	137	**91**
4	473	418	301	103	**54**
8	961	818	685	173	**79**
16	1028	967	802	228	**72**
32	1679	1409	1034	307	**61**
64	1941	1610	1114	258	**51**
128	2261	1915	1233	193	**42**

Table 4. Number of cycles per process for one barrier.

5 Simulation Results

We compare AMOs against conventional shared memory instructions (e.g., loads, stores, and LL/SCs), active messages (ActMsg), processor-side atomic instructions (Atomic), and memory-side atomic operations (MAOs) from the SGI Origin 2000, where applicable. It is difficult to derive useful analytical estimates for the time complexity of even relatively simple operations like barriers and spinlocks on modern shared memory systems with complex cache hierarchies and coherence protocols. Thus, we evaluate the performance of the various benchmarks via execution-driven simulation. We treat the conventional shared memory implementations as our performance baseline.

5.1 Barrier

Our **barrier** benchmark is the barrier synchronization function in the OpenMP library. The original OpenMP barrier implementation on the Origin 2000 uses LL/SC instructions. We created variants that used Atomic, active messages, MAOs, and AMOs. Table 3 presents the results of running these various barrier implementations on 4 to 256 processors (2 to 128 nodes). The first column contains the numbers of nodes being synchronized. Columns 2 through 5 present the speedups of the various barrier implementations compared to the baseline LL/SC implementation. Active messages, MAOs, and AMOs all achieve noticeable performance gains over the baseline. AMO-based barriers achieve the best performance speedup for all processor counts, ranging from a 1.9X speedup on a two-node system to 54.8X on a 128-node system.

In the baseline LL/SC-based barrier implementation, each processor loads a barrier count into its

local cache using an LL instruction before incrementing it using an SC instruction. If more than one processor attempts to update the count concurrently, only one will succeed, while the others will have to retry. After each successful increment, the barrier variable will move to another processor, and then to another processor, and so on. As the system grows, the average latency to move the barrier variable from one processor to another increases, as does the amount of contention. As a result, barrier synchronization time increases superlinearly as the number of nodes increases for the baseline LL/SC-based implementation. This effect can be seen particularly clearly in Table 4, which shows the per-processor barrier synchronization cycles for each implementation.

Using atomic instructions eliminates the failed LL/SC attempts. However, a flurry of invalidation messages followed by the data must still be transferred over the network in response to each increment, so the benefit of using atomic instructions is marginal.

The ActMsg barrier implementation sends an active message for every increment operation. The overhead of invoking the message handler dwarfs the time required to run the handler itself. Nevertheless, the benefit of eliminating remote memory accesses outweighs the high invocation overhead, which results in the active message barrier implementation outperforming the baseline version for all systems with more than two nodes. For the two-node system, the amount of contention for the barrier variable is negligible and the invocation overhead of the handler is not amortized. In addition, the high latency of invoking active messages leads to a significant number of timeouts and message retransmissions.

MAO-based barriers send a command to the home memory controller for every increment operation. Instead of naively using uncached loads to spin on the barrier variable, which generates huge network traffic, we spin on a local cacheable variable, an optimization similar to what is proposed by Nikolopoulos [13].

The AMO version eliminates the extra network traffic and serialized roundtrip delays, and achieves much higher performance. When each processor arrives at the barrier, it initiates an AMO `increment` operation and then spins on a local AMO register. When the barrier count indicates that every process has arrived at the barrier, the AMU sends a return value to each requesting processor, which informs the spinning thread that it may continue.

When the first `increment` operation arrives at a barrier variable's home memory controller, the barrier count is loaded into the AMU cache. All subsequent AMOs on this variable will find the data in the AMU cache and thus require only two cycles to process. Typically, the time to perform an AMO-based barrier roughly equals $(t_o + t_p \times N)$, where t_o is an overhead close to the round-trip latency of the longest network path in the system, t_p is a fixed value related to the time of performing an increment operation and sending an update request, and N is the number of nodes. The expression implies that AMO-based barriers scale as processor counts increase, which can be seen in Table 4. The per-processor latency of AMO-based barriers is almost constant. In fact, it drops off slightly as the number of processors increases because the fixed overhead is amortized over more nodes.

Running a process on each available processor forces the ActMsg-based barrier implementation to interrupt a processor doing useful work whenever an active message arrives. Since all ActMsgs will be sent to the same node for processing, this could be a significant bottleneck.

Nodes	Speedup over baseline			
	Atomic	ActMsg	MAO	AMO
2	0.96	0.62	0.88	**1.80**
4	1.02	1.69	1.48	**5.95**
8	1.02	1.86	2.20	**9.43**
16	1.05	2.18	3.67	**16.98**
32	1.09	2.24	5.71	**29.60**
64	1.05	2.20	8.40	**40.37**
128	1.08	2.16	13.92	**81.83**

Table 5. Barrier performance assuming a dedicated ActMsg handler processor per node.

```
acquire_ticket_lock( ) {
  int my_ticket = fetch_and_add(&next_ticket, 1);
  spin_until(my_ticket == now_serving);
}
release_ticket_lock( ) {
  now_serving = now_serving + 1;
}
```

Figure 2. Ticket lock pseudocode.

To determine whether the performance bottleneck of the ActMsg-based barrier implementation was processor occupancy, we ran a second set of experiments in which we dedicated one processor per node to handle active message requests. The other barrier implementations (baseline, Atomic, MAO and AMO) use one processor per node, leaving the other node completely idle. The results of these experiments are shown in Table 5. Comparing the results presented in Table 5 with those presented in Table 3, we can see that not interrupting the busy processor improves the speedup of active messages by up to 38%, but ActMsg-based barrier performance still lags far behind AMO-based barriers. This result indicates that the low invocation overhead and smaller bandwidth requirements of dedicated hardware AMOs are important for fully exploiting the potential value of offloading barrier operations to the node where the barrier variable resides. For the configurations that we test, AMO-based barriers outperform even the expensive pure hardware barriers present in several of current high-end interconnects (e.g., the Quadrics^{TM}QsNet used by the ASCI Q supercomputer [16]).

5.2 Spinlock

Different spin lock algorithms require different atomic primitives. We do not consider every proposed spinlock implementation, but instead focus on two representative algorithms: ticket locks [12] and Anderson's array-based queue locks [1].

Ticket locks employ a simple algorithm that grants locks in FIFO order. Figure 2 presents a typical implementation that uses two global variables, a sequencer (`next_ticket`) and a counter (`now_serving`). The performance of LL/SC-based ticket locks degrades rapidly as the number of participating processors increases due to races on the sequencer, and thus SC failures and retries, and the slow propagation of new counter values caused by invalidate-reload storms.

Array-based queue locks [1] use an array of flags and a counter that serves as an index into

CPUs	LL/SC		Atomic		ActMsg		MAO		AMO	
	ticket	array	ticket	array	ticket	array	ticket	array	**ticket**	array
4	1.00	0.41	0.91	0.52	1.12	0.50	1.01	0.41	**2.09**	1.24
8	1.00	0.46	0.86	0.54	1.70	0.46	1.05	0.50	**2.35**	1.74
16	1.00	0.50	0.97	0.56	2.27	0.53	1.10	0.50	**2.32**	2.27
32	1.00	0.55	0.99	0.63	2.37	0.53	1.07	0.51	**2.38**	1.95
64	1.00	1.66	0.87	1.69	0.67	1.47	0.67	1.51	**6.39**	5.01
128	1.00	2.80	1.14	2.68	0.89	2.44	0.79	2.52	**11.00**	10.99
256	1.00	3.55	1.24	3.44	1.00	3.01	0.85	2.99	**13.58**	11.35

Table 6. Speedup of various spinlocks compared to LL/SC-based spinlocks.

the array. Every process spins on a locally cached copy of its own array entry (flag). To signal a waiting process, the process releasing a lock sets the corresponding array entry, which invalidates the cached copy from the next process's cache. The next process then suffers a cache miss and loads the new value via a remote fetch. Even though selectively signaling one processor at a time noticeably improves performance in large systems, contention for the sequencer remains a hot spot.

For AMO-based spinlocks, we replace atomic `fetch_and_add` primitives with corresponding AMO operations. Further, waiting processes spin on an AMO register rather than a shared variable so that we can exploit the ability of AMOs to perform PUTs.

Table 6 presents the speedups of different ticket locks and array-based queuing locks compared to the LL/SC-based ticket lock. For traditional mechanisms, ticket locks outperform array locks when less than 32 processors (on 16 nodes) are involved. Array locks outperform ticket locks for larger systems, which illustrates the effectiveness of array locks at alleviating hot spots in large systems.

Our results show that AMOs greatly improve the performance of both types of locks and negate the difference between ticket locks and array locks. In particular, on 256 processors, AMO-based ticket locks outperform LL/SC-based ticket locks by a factor of 13.58 and LL/SC-based array locks by a factor of 3.8. In contrast, the other optimized lock implementations (using processor-side atomic operations, active messages, and MAOs) perform almost identically to their LL/SC-based counterparts, if not slightly worse. This advantage persisted even when we ran the benchmark on only one processor per two-processor node, leaving the second processor free to service active message requests without interrupting the main processor. Active message performance did not improve when a second processor per node was dedicated to message handling because performance was limited more by the higher overhead of invoking an active message handler and by network bandwidth on/off the home node than by processor occupancy. AMO-based ticket locks consume significantly less network and remote node occupancy than the other mechanisms studied.

5.3 Gups

The **gups** (a.k.a., the **table toy**) [9] microbenchmark performs random updates to a large array. It measures the number of GUPS (Global Updates Per Second) that a system can perform.

```
for (i = 0; i < accesses; i++)
```

```
    A[idx[i]] += v;
}
```

Gups is the principal benchmark of interest for the National Security Agency, and is important in many other high performance computing applications. We use a 256-megabyte array, which is tiny compared to a "real" NSA workload. We choose this array size because it can be simulated in a reasonable amount of time, yet is large enough to put significant pressure on the memory system and demonstrate the effectiveness of AMOs. A larger array results in higher cache miss rates for the baseline version, but has no effect on the AMO version. We believe that the performance improvement that AMO will be able to achieve on real workloads is no less than what is presented in this paper.

In Gups, the array `idx[]` contains random values, so the loop exhibits no regularity in its access stream to `A[]`. Conventional microprocessors perform poorly on this loop when the array `A[]` is large. Vector machines like the Cray XMP [2] and T3E [19] can run this loop efficiently using scatter-gather memory operations that efficiently load non-contiguous data into vector registers. However, scatter-gather is allowed only if there are no duplicated values in `idx[]`, which is not the case for some input sets, e.g., ones that model histogram calculations. For example, if both `idx[1]` and `idx[3]` equal `1` and a vector machine gathers `A[idx[0]]` to `A[idx[3]]` into a single vector register, `A[1]` will be incremented only once while it should have been incremented twice. To ensure correctness in such cases, the increment must be performed atomically. To the best of our knowledge, no existing machines handle a large number of random atomic operations efficiently.

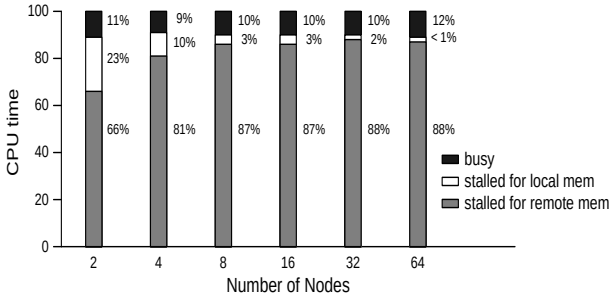

Figure 3. Breakdown of execution time.

The simulated system employs aggressive superpaging to avoid TLB misses. The TLBs in each node are designed to handle peta-byte memory space and therefore can easily handle 256MB datasets. Thus, TLB behavior has no significant impact on performance. Cache behavior is the limiting performance factor. In the baseline version of `gups`, almost every access to the array `A[]` suffers a cache miss and the CPU stalls frequently waiting for memory. Figure 3 shows that the CPU is stalled on memory accesses the majority of the time. As the number of active nodes increases, `A[]` is spread across more processors and thus the number of remote misses increases. With four processors, remote memory stall cycles account for 66% of execution time, while 88% of execution time is spent waiting for remote memory accesses on 128 processors.

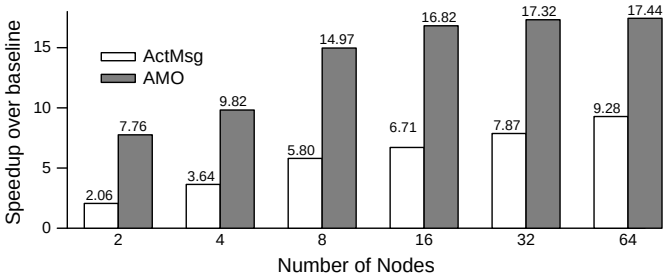

Figure 4. GUPS **Performance**

Gups performance can be optimized using either active messages or AMOs. For our experiments, we assume there are no duplicate values in `idx[]`, so normal additions, instead of atomic additions, are performed. Using atomic additions would make the baseline version of Gups slower, but would have no effect on the AMO version, which would further increase AMO speedup. The performance results are shown in Figure 4. For ActMsg-based `gups`, we reserve one processor per node to handle active messages, providing a kind of best case performance for active messages. For baseline and AMOs, the second processor per node sits idle.

ActMsg-based `gups` outperforms the baseline version by as little as a factor of 2.06 on the two-node system and by as much as a factor of 9.28 on the 64-node system. The speedup of the AMO-version is about twice of that the ActMsg-based version, ranging from 7.76 to 17.44.

nodes	2	4	8	16	32	64
baseline	1358	2096	2519	2739	2931	3071
ActMsg	302	402	577	643	688	744
AMO	262	393	459	494	516	558

Table 7. Total number of network packets (in thousands).

The main reason that both mechanisms are so effective is that they eliminate substantial amounts of network traffic. Table 7 shows the number of network packets sent for each test case. On average, active messages reduce network traffic by 4.3X, while AMOs reduce network traffic by 5.5X. Another factor that causes AMOs to outperform active messages is that each AMU can handle four AMOs simultaneously while a processor can handle only one active message at a time.

5.4 Range and Reduction Operations

We use three microbenchmarks to evaluate the effectiveness of AMO range and reduction operations: *memset*, *max*, and *popcount*. The *memset* benchmark comes from the standard C library. It counts the number of bit 1's in a 16MB memory area. We exploit the potential of adding a specialized popcount circuit to the AMU. [2] We assume a computation latency of 128 processor cycles for doing a popcount of 128 bytes in an MFU. The baseline version of this benchmark uses a 64KB

[2]Though there are ISAs that include a *popcount* instruction (e.g., SPARC), we are not aware of any existing processor that actually supports the instruction. And even if they do, the system bus will not be able to deliver data from memory to the processor-side *popcount* circuit in a timely fashion.

	memset	max	popcount
speedup	2.78	5.67	16.69

Table 8. Speedups of range and reduction active memory operations.

jump table to optimize the naive bit-shift implementation. A 64KB table is small enough to fit in the L2 cache, yet allows us to count the number of 1 bits in a two-byte region using a single table access.

For these tests, we show single node performance. Techniques like partitioning and dynamic scheduling have been used extensively to perform efficient parallel range operations on supercomputers. For instance, reduction operations supported by the MPI library create tree structures to first perform partial reductions locally in the leaf nodes and then merge partial results in the parent nodes. Because AMOs are always executed at the home node, in a parallelized version, we will automatically achieve the goal of performing range operations on the nodes where the data resides without complex partitioning algorithms.

Table 8 shows the speedups of AMOs on these benchmarks.

We disassembled the baseline version of these benchmarks and found that the compiler inserted nearly-optimal software prefetch instructions. As a result, the baseline versions see very few cache misses. However, because the processor can process data faster than the system bus can transfer it, the system bus becomes a performance bottleneck. The AMO version of these benchmarks issues one AMO for each page, thus the system bus bandwidth is no longer a bottleneck. Instead, AMOs exploit the high bandwidth within the memory controller and speed up *memset* by a factor of 2.78 and *max* by a factor of 5.67.

The *popcount* benchmark shows the potential benefit of providing support for specialized operations at the memory controller. The AMO-based popcount outperforms the optimized (table-based) baseline version by a factor of 16.69. This large performance improvement motivates the inclusion of a popcount circuit, or other specialized circuit, at the MC for select operations if there is sufficient demand.

6 Conclusions and Future Work

As the relative penalty of remote memory accesses increases, eliminating remote memory accesses and hiding their latency have become critical to achieve a high performance in large scale DSM systems. In this paper, we propose to ship computation to the home memory controller of select data as a means of doing so. Such active memory operations (AMOs) can significantly reduce network traffic and hide the latency of accesses to data with insufficient reuse to warrant moving it across the network or system bus. Like active messages, AMOs are efficient at overlapping communication with computation. Unlike active messages, AMOs do not need to interrupt the remote processor.

We have demonstrated the potential of the active memory unit for some important operations. Our simulation results show that AMOs significantly increase the scalability of applications like

synchronization and GUPs that do not scale well on conventional shared memory systems. AMOs also greatly improve single node performance of range and reduction operations.

These encouraging results motivate us to continue this line of research. We are extending AMOs to a richer set of functions. One extension we are considering is support for vector-vector operations, which are often seen in scientific and engineering applications. We plan to compares the potential benefits of AMO-based vector operations compared to or in conjunction with dedicated per-node vector units.

One limitation of our current AMO design is that it supports only a small set of operations. Another direction of future work that we plan to pursue is investigating the value of replacing MFUs with simple in-order processor cores, like an MIPS R4000. The relatively poor performance of the various ActMsg-based benchmarks compared to their AMO-based counterparts provides a strong indication that the overhead of initiating an AMO operation on such a general purpose processor must be minimized. As transistor density increases, a number of these cores can easily fit in an AMU. Such a design would significantly complicate the programming model, but provide richer opportunities for performance improvement.

Acknowledgments

The authors would like to thank Silicon Graphics Inc. (SGI) for the technical documentation used to configure our simulation models, and in particular, for the valuable technical feedback provided by Marty Deneroff, Steve Miller, and Steve Reinhardt. We would also like to thank the Defense Advanced Research Projects Agency (DARPA) and National Security Agency (NSA) for their support. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements of SGI, DARPA, NSA, or the U.S. Government.

References

[1] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[2] T. Cheung and J. Smith. A simulation study of the Cray X-MP memory system. *IEEE Trans. on Computers*, C-35(7):613–622, July 1986.

[3] Compaq Computer Corporation. Alpha architecture handbook, version 4, Feb. 1998.

[4] M. Garzaran, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas. Architectural support for parallel reductions in scalable shared-memory multiprocessors. In *Proc. of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 243–254, Sept. 2001.

[5] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU multicomputer - designing a MIMD shared-memory parallel machine. *IEEE TOPLAS*, 5(2):164–189, Apr. 1983.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, and V. Freeh. Mapping irregular appilcations to DIVA, a PIM-based data-intensive architecture. In *SC'99*, Nov. 1999.

[7] Intel Corp. Intel Itanium 2 processor reference manual. `http://www.intel.com/design/itanium2/manuals/25111001.pdf`.

[8] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[9] D. Koester and J. Kepner. *HPCS Assessment Framework and Benchmarks*. MITRE and MIT Lincoln Laboratory, Mar. 2003.

[10] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *ISCA97*, pp. 241–251, June 1997.

[11] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture: A Specification for a New Family of Processors, 2nd edition*. Morgan Kaufmann, May 1994.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

[13] D. S. Nikolopoulos and T. A. Papatheodorou. The architecture and operating system implications on the performance of synchronization on ccNUMA multiprocessors. *International Journal of Parallel Programming*, 29(3):249–282, June 2001.

[14] M. Oskin, F. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proc. of the 25th ISCA*, pp. 192–203, June 1998.

[15] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keaton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2), Apr. 1997.

[16] F. Petrini, J. Fernandez, E. Frachtenberg, and S. Coll. Scalable collective communication on the ASCI Q machine. In *Hot Interconnects 12*, Aug. 2003.

[17] R. Graybill. High productivity computing systems. `http://www.darpa.mil/DARPATech2002/presentations/ipto_pdf/speeches/GRAYBILL.pdf`, 2002.

[18] M. Scott and J. Mellor-Crummey. Fast, contention-free combining tree barriers for shared memory multiprocessors. *International Journal of Parallel Programming*, 22(4), 1994.

[19] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. of the 7th ASPLOS*, Oct. 1996.

[20] Silicon Graphics, Inc. *SGI™Origin™3000 Series TR*, Jan. 2001.

[21] Silicon Graphics, Inc. *SN2-MIPS Communication Protocol Specification, Revision 0.12*, Nov. 2001.

[22] Silicon Graphics, Inc. *Orbit Functional Specification, Vol. 1, Revision 0.1*, Apr. 2002.

[23] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. of the 29th ISCA*, pp. 171–182, May 2002.

[24] T. Sunaga, P. M. Kogge, et al. A processor in memory chip for massively parallel embedded applications. *IEEE Journal of Solid State Circuits*, pp. 1556–1559, Oct. 1996.

[25] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proc. of the International Parallel and Distributed Processing Symposium*, page 84a, Apr. 2003.

[26] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th ISCA*, pp. 256–266, May 1992.

[27] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. on Computers*, C-36(4):388–395, Apr. 1987.

[28] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee. The Impulse memory controller. *IEEE Trans. on Computers*, 50(11):1117–1132, Nov. 2001.