Active Memory Operations

Zhen Fang
Intel Corp.
2111 NE 25th Ave
Hillsboro, OR 97124
zhen.fang@intel.com

Lixin Zhang
IBM Austin Research Lab
11400 Burnet Rd
Austin, TX 78758
zhangl@us.ibm.com

John B. Carter
School of Computing
University of Utah
Salt Lake City, UT 84112
retrac@cs.utah.edu

Ali Ibrahim
AMD
4555 Great America Pkwy
Santa Clara, CA 95054
ali.ibrahim@amd.com

Michael A. Parker
Cray, Inc.
1050 Lowater Rd
Chippewa Falls, WI 54729
map@cray.com

Abstract

The performance of modern microprocessors is increasingly limited by their inability to hide main memory latency. The problem is worse in large-scale shared memory systems, where remote memory latencies are hundreds, and soon thousands, of processor cycles. To mitigate this problem, we propose the use of Active Memory Operations *(AMOs), in which select operations can be sent to and executed on the home memory controller of data. AMOs can eliminate significant number of coherence messages, minimize intranode and internode memory traffic, and create opportunities for parallelism. Our implementation of AMOs is cache-coherent and requires no changes to the processor core or DRAM chips.*

In this paper we present architectural and programming models for AMOs, and compare its performance to that of several other memory architectures on a variety of scientific and commercial benchmarks. Through simulation we show that AMOs offer dramatic performance improvements for an important set of data-intensive operations, e.g., up to 50X faster barriers, 12X faster spinlocks, 8.5X-15X faster stream/array operations, and 3X faster database queries. Based on a standard cell implementation, we predict that the circuitry required to support AMOs is less than 1% of the typical chip area of a high performance microprocessor.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*System architectures*; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiprocessors*; B.8 [**Hardware**]: Performance and Reliability—*General*

ICS'07, June 18–20, 2007, Seattle, WA, USA.

General Terms

Design, Performance

Keywords

Memory performance, Distributed Shared Memory, Cache coherence, Thread synchronization, DRAM, Stream processing

1. INTRODUCTION

Distributed shared-memory (DSM) systems distribute physical memory across the nodes in the machine and implement coherence protocols to provide the shared memory abstraction. In the predominant directory-based CC-NUMA architecture, 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, the home node may need to send messages to additional nodes to service the request and maintain coherence. The round trip memory access time of large DSM machines will soon be a thousand processor cycles [26]. It consists of three parts: local DRAM latency, memory controller occupancy, and inter-node network latency. Cross-section bandwidth is also a limiting factor in the scalability of large DSM systems.

Caching improves memory performance and reduces remote traffic, but large caches cannot eliminate coherence misses. Coherence is maintained at the block-level (e.g., 128 bytes), and entire blocks are moved across the network or invalidated, even when the processor touches only a single word. For operations with low temporal locality or significant write sharing, moving data from (potentially remote) memory, through the cache hierarchy and into a register, operating on it, and then (optionally) writing it back to memory is highly inefficient in time and energy. In these circumstances, caches provide little benefit, and sometimes even hurt performance. The appalling truth is that the sustained performance of large DSM computers for many applications is less than 5% of peak performance, due largely to memory system performance, and this trend is expected to worsen. The unavoidable conclusion is that reducing remote coherence traffic

and inter-node data transfers is essential for DSM systems to scale effectively.

We propose to add an **Active Memory Unit** (AMU) to each memory controller in a DSM system so that operations with poor temporal locality or heavy write sharing can be executed where the data resides. We call AMU-supported operations **Active Memory Operations** (AMO), because they make conventional "passive" memory controllers more "active". AMOs are issued by processors and forwarded to the home node of their operands, which acquires a *globally coherent* copy of the data and then performs the operation in its local AMU.

Used judiciously, AMOs can eliminate cache misses, reduce cache pollution, and reduce network traffic, thereby reducing power consumption and improving performance. AMOs are best used on data with poor temporal locality or heavy write sharing, for which caching induces substantial communication while providing no benefit. Using AMOs, instead of loading data across the network and operating on it locally, short messages are sent to the data's home node, which acquires a globally coherent copy of the data (iff it is cached remotely), performs the requested operations, and (optionally) returns the result to the requesting node.

In our design, AMUs support both *scalar operations* that operate atomically on individual words of data and *stream operations* that operate on sets of words separated by a fixed stride length. All stream elements must reside in a single page of memory, so individual streams do not span multiple memory controllers. However, stream-stream operations may involve streams that reside on different memory controllers, as described in Section 3.2. Stream AMOs can be masked. All AMOs are cache-coherent; the AMU hardware performs any necessary coherency operations before it uses any data. Because AMUs are integrated with the directory controller, making AMOs coherent is fairly simple and fast (Section 3.3).

Figure 1 illustrates how AMOs can be used to implement a simple SQL query that computes the average balance of all customers in the "East" sales region. We perform three suboperations: (1) determine which records have Region fields that match the "East" attribute, (2) determine how many records matched in phase (1), and (3) calculate the sum of the Balance fields of each record that matched in phase (1). Each of these suboperations can be implemented as a single AMO. The first AMO performs a strided `stream-string cmp` against the Region field, where the stride is the size of each customer record. The outcome of this AMO is a bitmask (Bitstream), where a '1' in position N of the stream indicates that the N^{th} customer is in the "East" region. The second AMO performs a `popcount` on Bitstream to determine how many customers were in the "East" region. The third AMO adds up the "Balance" fields for each customer in the "East" region. The overall result of the query is simply the sum of the balances (result of the 3^{rd} AMO) divided by the number of customers in the east region (result of the 2^{nd} AMO). This example, while simple, illustrates a number of interesting features of our AMO implementation, which we will describe in Section 3.2.

For many operations, an AMO can replace thousands of memory block transfers. As described in Section 5, AMOs can lead to dramatic performance improvements for data-intensive operations, e.g., up to 50X faster barriers, 12X faster spinlocks, 8.5X-15X faster stream/array operations, and 3X faster database queries. Finally, based on a standard cell implementation, we predict that the circuitry required to support AMOs is less than 1% of the typical chip area of a high performance microprocessor.

2. RELATED WORK

Processor-in-memory (PIM) systems incorporate processing units on modified DRAM chips [11, 18, 24, 7, 36]. Both AMOs and PIMs exploit affinity of computation to main memory, but they differ in three important ways. First, AMOs use commodity DRAMs, which should have higher yield and lower cost than PIMs. Second, the processors in a PIM reside below the architecture level where coherence is maintained. Thus, if the data required by a PIM operation resides off-chip, PIMs effectively becomes a form of non-coherent distributed memory multiprocessor, with all of the attendant complexities. AMOs utilize existing coherence mechanisms and operate on coherent data. Third, PIMs employ merged logic-DRAM processes, which are slower than processes tuned for logic. The major benefit of PIMs is the very high bandwidth of the on-chip connections between processors and storage. However, high-performance memory controllers support a large numbers of DRAM busses and thus have raw bandwidth comparable to what is available within a single DRAM. We strongly believe that the appropriate place to perform offloaded computation is at the memory controller, not on the DRAMs, which has most of the bandwidth and power advantages of PIMs and eliminates many of the complexities that PIMs introduce.

Several research projects have proposed adding intelligence to the memory controller. The Impulse memory controller [40] uses an extra level of physical address remapping to increase or create spatial locality for stride and random accesses. Active Memory [16] extends Impulse to multiprocessors. Solihin *et al.* [33] add a general-purpose processor core to a memory controller to direct prefetching into the L2 cache. These systems improve the way in which conventional processors are "fed" memory, but do not actually compute on the data.

Several systems support specialized memory-side atomic operations for synchronization. The NYU Ultracomputer [9] was the first to implement atomic instructions in the memory controller. The FLASH [19] multiprocessor supported atomic `fetch_op`s, which were subsequently supported by the SGI Origin 2000 [20] and Cray T3E [29]. The SGI Origin 2000 implements 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.

Although they are performed solely in software, Active Messages [37] are similar in spirit to AMOs. Like AMOs, active messages work by moving computation to data to avoid moving data between processors. An active message includes the address of a user-level handler to be executed upon message arrival, with the message body as its argument. Because they are handled in software, active messages suffer from a number of overheads, e.g., taking an interrupt, flushing the instruction pipeline, switching to the specified message handler, and polluting the caches while the handler runs. Also, the active message programmer must know which node holds the desired data, whereas the target of an AMO is extracted by hardware based on the data's address. Nevertheless, active messages often perform well for the same reason that AMOs do - it is often far less efficient to load data over the network than to simply operate on it in place.

A number of researchers have proposed offloading select computation from the main processor, e.g., several recent cluster interconnects support distributed synchronization operations [25, 35], while Ahn et al. [2] propose adding specialized vector operations to the memory system to support vector scatter-add.

FLASH [19] and Tempest/Typhoon [28] incorporate a programmable engine on the memory/network controller. To the best of our knowledge they never explored the value of offloading user computation to the protocol processors. Rather, the protocol processors were used to support powerful coherence protocols and to facilitate high-speed message transfer.

3. AVTIVE MEMORY OPERATIONS

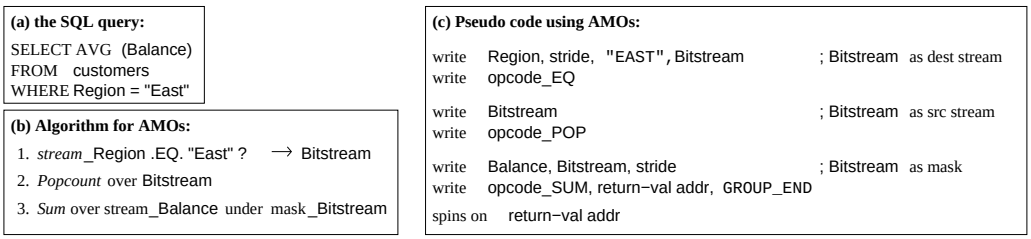


Figure 1. Example of using AMOs for a simple SQL query

Mnemonic	Result
inc, dec, cmp-swp, fetch-aop, update	scalar
memcopy, memset, pgfetch, pgsave	stream of scalars
max, min, sum, popcount	scalar (reduction)
S-s-lop, S-s-s-lop, S-string-EQ	stream of bits
S-s-aop, S-S-aop, S-S-lop	stream of scalars or bits

Table 1. Supported AMOs (*Legend: S=stream, s=scalar, aop=arithmetic_op, lop=logic_op*)

3.1 Supported Operations

Our AMU design supports two types of operations: *scalar operations* that operate on single words and *stream operations* that operate on sets of words within a page. Table 1 lists the operations supported, which were selected based on application requirements and implementation complexity. Most AMOs include integer and single precision floating point versions. All AMOs are coherent; the AMU performs all necessary coherency operations before it uses data.

Scalar operations perform atomic arithmetic operations on individual words of data, similar to the `fetch_and_op` operations of many existing architectures. Scalar operations are particularly useful for supporting efficient synchronization. Unlike synchronization operations implemented at the processors, e.g., using load-linked/store-conditional (LL/SC), AMO-based synchronization does not require cache lines to bounce between processors.

Stream operations have SIMD-like semantics: one arithmetic or logic operation is applied to every element of a stream. In our design, a stream is a set of words separated by a fixed stride length, all of which reside in a single page of memory. Thus, individual streams do not span multiple memory controllers, but stream-stream operations may involve streams that reside on different memory controllers, as described in Section 3.2. Stream operations can be optionally predicated using a bit mask, similar to masked operations in vector ISAs.

We support a mix of stream-scalar, stream-reduction, and stream-stream operations, as shown in Table 1. For example, a stream-scalar `add` adds a scalar value to every element in the stream, which generates a second stream, while stream-scalar `.GT.` compares every element in the stream against a given scalar value and creates a boolean result stream. We believe these operations are general enough to cover a large portion of the needs of data-intensive applications.

Since primitive stream operations are restricted to streams that reside entirely within a single page, stream operations that span larger ranges need to be implemented as a series of AMOs on smaller ranges, which can be performed in parallel.

3.2 Programming Model

In our current design, the basic programming model for AMOs is that of decoupled (asynchronous) operations. The local processor issues AMOs and then can perform other operations while the AMO is underway. When the local processor needs the return value or needs to know that the AMO operation has completed, it spins waiting on a completion bit to be set.

We employ memory-mapped uncached I/O space reads and writes to communicate between the processor and AMO engine. To issue an AMO, a processor writes the appropriate values (e.g., command, address(es), and scalar operand(s)) to a set of I/O space addresses that correspond to an AMO issue register on the local memory controller. To read the result of an AMO, a processor reads from an I/O space address that corresponds to an AMO return register allocated to the issuing process. Associated with each AMO return register is a full-empty (F/E) bit that is used to signal when an AMO has completed and its return value (if any) is ready. These I/O addresses are allocated and managed by the OS via special system calls invoked during process initialization. This approach is similar to that used to operate on E-registers in the Cray T3E [29].

Basic AMOs

To initiate an AMO, software writes the arguments to an AMO register in the local memory controller. The AMU clears the F/E bit of the associated AMO return register, determines which node is the home for the specified data, and sends a request packet to the AMU on the home node of the data on which the operation is being performed. Each AMU has an external TLB to perform virtual to physical translation, similar to the Cray T3E [29] and Impulse [40]. If necessary, the home node AMU interacts with other processors to acquire a globally coherent copy of the data in the appropriate sharing mode. Once the AMU has a coherent copy of the data, it performs the operation and signals completion to the processor that issued the AMO, optionally returning a scalar result. For streamed AMOs that span multiple cache lines within a single page, the home AMU acquires coherent copies of each cache line touched by the AMO. When the result returns, it is placed in the specified AMO return register and the corresponding F/E bit is set. The requesting process is expected to test the appropriate F/E bit periodically to determine when the operation is complete and the return value is available.

Grouped AMOs and temporary streams

Semantically, AMOs operate on DRAM values. Logically AMO data is read from main memory and results are written back to main memory, in addition to the optional scalar value that can be returned to the issuing processor. However, for operations involving several AMOs, e.g., the simple database operation presented in Figure 1, it is inefficient to write intermediate results back to main memory, only to be immediately re-read and reused as part of a subsequent AMO. To overcome this potential inefficiency, we allow programmers to *group* related AMOs and specify that certain streams are *temporary streams* whose values need not persist beyond the end of the current AMO group. Programmers mark the end of an AMO group by setting the `GROUP_END` bit in the AMO opcode. Programmers identify temporary streams by using special (otherwise invalid) addresses for their locations. Temporary streams can be used to pass intermediate values efficiently stream operations. For example, the Bitstream bit mask used in the database example shown in Figure 1 would be an ideal candidate to be treated as a temporary stream. Temporary streams are not written back to DRAM and can be *bypassed* directly from the source ALU to the destination ALU. If the Bitstream value were useful in some later computation, the programmer can specify a real

memory location where it should be written. The hardware required to support grouped AMOs and the way in which grouped AMO operations are implemented is described in more detail in Section 3.3.

Masked Operations

The third AMO in Figure 1 illustrates a *masked AMO*, which is analogous to the masked vector operations present in many vector ISAs. The Bitstream stream is used as a bit mask to indicate which elements of the strided Balance stream should be summed together. Typically the bitmask used in masked operations are generated as part of a grouped AMO and then discarded, as in this example, but bitmasks can also be stored in memory and reused.

Other Programming Considerations

Since stream AMOs operate only on streams that reside within a single page, operations on larger streams must be performed as multiple page-grained AMOs. The programming burden for doing so is modest and enables operations on different portions of the stream to proceed in parallel if the data is homed by multiple nodes.

The streams in a single AMO or grouped AMOs can have different home nodes. In these cases, the AMO(s) is(are) sent to the home node of any of the streams, and remote streams are loaded across the interconnect (coherently) from their respective home nodes. In the case of grouped AMOs, all AMOs within a group are handled by a single AMU.

Currently we manually write AMO codes using a combination of C-language libraries and macros. We argue that current compiler technology should be able to generate decent-quality AMO codes from serial or vectorized non-AMO scientific computation programs, though it is extremely challenging to automate the code transformation process for commercial applications. Using scalar AMOs is as simple as substituting an AMO wrapper function for the legacy function name (e.g., `barrier`) or instruction (e.g., `fetch-op`). The conerns arise primarily when using stream AMOs.

Stream AMOs can be considered a special class of vector operations, although with completely different hardware implementations. Both break array operations into a loop of vector computations, each of which is a SIMD operation. The difference in operation granularity and hardware implementations has significant performance implications, but the compiler technology that is required to generate both versions of the SIMD codes is largely the same. Vectorizing compilers have been successfully built for Fortran and C. Leveraging these technologies to build an AMO compiler is part of our future work. A key step in developing such a compiler is building a cost function that can predict the performance of an application if it used AMOs. We have developed an analytical model that can predict AMO performance. Details of the model are in [8].

Automatic Computation Localization

OpenMP, the *de facto* standard for shared-memory programming, allows the programmer to parallelize loops without consideration for the underlying memory distribution. This often results in unsatisfactory performance due to high remote access costs. To circumvent this problem, programmers often take great care to create computation-data affinity, i.e., to perform operations on the node where data is homed. However, if the data access pattern changes after initial data placement, which is common when the OS employs a first-touch memory allocation policy or when threads migrate between nodes, the computation-data affinity is lost. In contrast, AMOs inherently achieve computation-to-data affinity with no extra programming effort because AMOs are *dynamically* routed to the node that homes the data on which they operate. Should threads migrate or data be redistributed, AMOs will continue to be routed to the appropriate (new) nodes for execution.

Context Switches, AMO Register Virtualization, and Exception Handling

In our design, threads explicitly allocate AMO issue and return registers via system calls. Upon a context switch, the OS saves AMU issue registers so that partially-initiated AMOs can be properly restarted later. The internal states within the AMU are not part of the architectural context. A thread's AMOs continue to execute while it is context switched off the processor, and the associated AMO return register continues to be a legal and safe target for the AMU to store a return value.

Our current design limits the number of AMOs with return values that a given thread, or set of threads on a single node, can have in flight at any given time to the number of physical AMO registers present on each memory controller. A more scalable design would entail virtualizing the AMO registers so that multiple client threads could share physical AMO registers. To support virtualization, each memory controller would need to be able to map reads and writes to virtual AMO registers, identified by (ProcessID, RegNum) pairs, to either physical AMO registers or private DRAM managed by the AMU to back the physical registers. We do not support this functionality in our current design, because a modest number of AMO registers suffices for all of the applications that we consider, but it might be warranted in commercial implementations.

Arithmetic exceptions (e.g., divide by zero) can occur during an AMO. Arithmetic exceptions cause the AMO to terminate and an error value to be returned to the associated AMO return register along with enough state to identify the source of the error. A page fault can occur when a hardware pagetable walker handles an AMU TLB miss in the node where the AMO is initiated. This can happen, e.g., when a page is swapped out by the OS while an AMO is in the middle of an AMU execution, and a TLB consistency message is broadcast to the system [34]. The memory controller issues an interrupt that causes the OS to be invoked to handle the fault and the terminated AMO to restart.

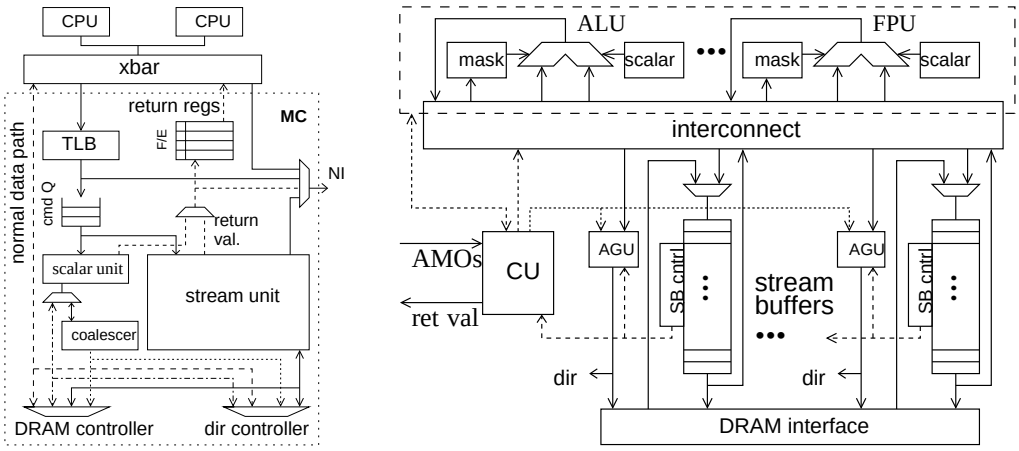
3.3 Hardware Organization

Figure 2 (a) presents a block diagram showing the major components of interest in a single node with our AMU-enhanced memory controller (MC). A crossbar connects processors to the network backplane, from which they can access remote memory, local memory, and IO. The processor(s), crossbar, and memory controller are on the same die, separate from the DRAM chips. AMO functionality is not on the critical path of normal memory references. Each MC is extended to include a modest number of AMU issue and return value registers. Most of the microarchitectural design we discuss here is transparent to software.

When a processor initiates an AMO, the local AMU translates the target virtual address to a global physical address and sends an AMO message to the AMU on the corresponding home node. Address translation is performed via an external TLB located on the memory controller. For stream-stream AMOs, the local AMU selects one of the target addresses and forwards the AMO request to the corresponding AMU. For grouped AMOs, the local AMU selects one node from any of the non-temporary target stream addresses and forwards the request to this node. In our current design, when there is more than one possible destination node, the source AMU selects the one that results in the fewest inter-node stream transfers. When an AMO message arrives at its destination, it is placed in an AMU command queue to await dispatch. If the queue is full, the target AMU sends a NACK to the requesting node, which must reissue the request.

The scalar unit handles scalar AMOs. It incorporates a tiny *coalescer* cache used exclusively for synchronization variables. The coalescer eliminates DRAM accesses when the same word of data is the target for frequent AMO operations, which is common for heavily contested synchronization variables. In our study, four single-word entries can cache all hot scalar data.

Figure 2 (b) is an enlargement of the stream unit in Figure 2 (a). When the control unit (CU) sees the `GROUP_END` bit, it allocates



(a) An AMU-augmented MC

(b) The Stream Unit

Figure 2. The Active Memory Unit

stream buffers (SBs) to hold the operand and result streams associated with each AMO. The stream addresses used in the group are compared to identify producer-consumer relationships and common-source sharing. The CU uses a small reverse mapping table similar to a conventional register mapping table to store these stream-to-SB mappings.

The SBs are key to stream AMO performance. They hide DRAM latency and provide intermediate storage between components of grouped AMOs. Each SB is a 1-word $\times$ 32-entry dual-ported SRAM managed as a circular queue. Associated with each SB is a simple Address Generation Unit (AGU). AGUs generate a stream of memory requests to load stream values into the appropriate SB entries. The addresses generated correspond to the head of the stream followed by a series of addresses, a stride apart, until either the end of the stream is reached or the SB is filled. In the latter case, the AGU pauses until an entry is freed and then issues a new request. For non-unit stride streams, we exploit short DRAM burst lengths. Reads from a given SB are consumed in-order by the ALUs, but writes into the SB can occur in random order since main memory and remote caches can respond out of order, which is why we use an SRAM instead of a FIFO for SBs.

Associated with each SB entry is a F/E bit, which denotes whether a particular operand has been returned from main memory or a remote cache. When the F/E bits of the head of each operand SB for an AMO indicate that the first operands are ready, they are fetched to the function unit (FU) that has been allocated to this AMO by the CU. Results from each FU are directed to the proper result SB by the CU. Read and write pointers associated with each SB are used to determine which word to read (write) as data is consumed (produced) by AMOs.

To ensure global coherence, the AMU issues DRAM requests via the local directory controller, which checks to see if the copy of the data in local memory can be used directly. If not, the directory controller performs the necessary operations to make the data coherent (e.g., issuing invalidate or flushback requests to remote sharers).

Our current AMU design includes a 1024-entry TLB, five integer ALUs, four single-precision FPUs, sixteen 32×32 bit stream buffers, sixteen AMO issue and result registers, control logic, and wires. To determine the amount of silicon required to implement the proposed AMO mechanisms, we use a commercial memory generator by Artisan^{TM}to model the SRAM structures and implement the rest in Verilog, which we synthesized using Synopsis^{TM}tools. We conservatively estimate the area of an AMU to be no more than 2.6 mm^2 in a 90nm process, which is less than 1.9% of the total die area of a high-volume microprocessor or 0.8% of a high-performance microprocessor [14]. If AMOs were integrated with the processor core so that AMO addresses could be translated by the processor TLB, the chip area would shrink to 1.5 mm^2. These estimates are conservative, since they are derived using high-level design and synthesis tools. A custom implementation would likely be less than half of the estimated size.

Parameter	Value
Processor	4-issue, 128-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 system to CPU, 8B CPU to system
	max 16 outstanding references, 1GHz
DRAM	4 16B-data DDR channels
Hub clock	500 MHz
Memory latency	$>$120 processor cycles
Network latency	100 processor cycles per hop

Table 2. System configuration.

4. EXPERIMENTAL SETUP

Simulation framework

We use execution-driven simulation to evaluate AMOs. Our simulator [38] accurately simulates large-scale ccNUMA systems, including detailed processor, cache, bus, interconnect, memory controller, IO, and DRAM models. The system model is a hypothetical next-generation SGI supercomputer and models the complete SGI directory-based coherence protocol [30]. Each simulated node models two superscalar processors connected to a high bandwidth bus. Also connected to the bus is a hub [31] that integrates the processor interface, memory controller, directory controller, coherence engine, network interface, and IO interface. Each node contains a DRAM backend with 16GB of physical memory. We simulate a micro-kernel that has realistic memory management routines, supports most common Unix system calls and directly execute statically linked 64-bit MIPS-IV executables. The simulator supports the OpenMP runtime environment.

Table 2 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

Benchmark	Description	From
barrier	barrier synchronization	SGI Irix OpenMP library
spinlock	ticket lock and array-based queue lock	Mellor-Crummey and Scott [22], Anderson [4]
GUPS	random global updates	HPCS Program
STREAM	memcpy, scale, sum, triad	J. McCalpin
SAXPY	Y += a * X, single precision FP	BLAS level 1
Info_Retrieval query	unindexed relational database query	OSDB by Compaq-HP
Total_Report query	unindexed relational database query	OSDB by Compaq-HP
Query 2A	for document search	Set Query Benchmark
Query 3A	for direct marketing and decision support	Set Query Benchmark
Query 4	for direct marketing and document search	Set Query Benchmark

Table 3. Benchmarks

4 20-byte channels connected to DDR DRAMs, which enables us to read an 80-byte burst per channel every two cycles. Of each 80-byte burst, 64 bytes are data and the remaining 16 bytes are a mix of ECC bits and partial directory state. The simulated interconnect subsystem is based on SGI's NUMALink-4. The interconnect is a fat-tree, where each non-leaf router has eight children. The minimum network packet is 32 bytes. We do not model contention within routers, but do model port contention on the hub interfaces. We have validated the core of the 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. All simulator-generated statistics (e.g., run time, cache hit rates, etc.) are within 15% of the corresponding numbers generated by the real machine, most within 5%.

We extended the simulator to support active messages and processor-side atomic instructions like those in the Itanium2 [13]. Most overheads of active messages are accurately modeled, e.g., interrupt handling, instruction pipeline flush, and cache pollution effects, but some are not, e.g., OS overhead on the issuing processor, so our results for active messages are somewhat optimistic.

Benchmarks

Table 3 lists the ten benchmarks we use to evaluate AMOs. The first three only use scalar AMOs while the rest of the benchmarks mainly use stream AMOs. All are compiled using the MIPSpro Compiler 7.3 with an optimization level of "`-O3`". Native compiler support for AMOs is not currently available, so we manually inserted AMOs using simple macros. All results presented in the next section represent complete simulations of the benchmark programs, including kernel and application time, and the direct and indirect overheads resulting from the use of AMOs and active messages. Throughout the paper, we define the *speedup* of $code_A$ over $code_B$ as $Execution\_time_B$ / $Execution\_time_A$.

The *barrier* benchmark uses the barrier synchronization function of the Irix OpenMP library. To evaluate *spinlock* algorithms, we consider two representative implementations, ticket locks [22] and Anderson's array-based queue locks [4]. For both *barrier* and *spinlock*, we insert small, random delays similar to what Rajwar *et al.* [27] did. The *GUPS* benchmark [17] performs random updates to a large array to determine the number of global updates per second (GUPS) that a system can sustain. It represents the key access pattern in molecular dynamics, combustion, and crash simulation codes. The *STREAM* benchmarks [21] are often used to measure the effective bandwidth of parallel computers. *SAXPY* is from the Basic Linear Algebra Subprograms (BLAS) suite. It is representative of the many BLAS library functions that map effectively to AMOs.

Historically, database applications were disk I/O bound. However, memory density has increased and disk optimizations have reduced the impact of I/O on database performance, so memory performance has emerged as the new bottleneck for many large databases [3, 5, 6]. We investigate the potential of AMOs to accelerate queries from Set Query Benchmark [10] and the Open Source Database Benchmark (OSDB) [12] suites. DB benchmarks like TPC-H are more complete, but are hard to install and evaluate in realistic simulation time [1, 15]. While the database benchmarks we use are simpler than benchmarks such as TPC-H, researchers from both the database and computer architecture communities have found that greatly simplified microbenchmarks capture the processor and memory system behavior of TPC workloads quite well [15, 32]. The database benchmarks that we consider are more complex than the queries used in those studies.

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 4. Barrier performance (2 cpus per node)

5. SIMULATION RESULTS

For each of our ten benchmarks, we compare their performance when implemented using conventional shared memory instructions (e.g., loads, stores, and LL/SCs), active messages (ActMsg), processor-side atomic instructions like those in Intel Itanium (Atomic), memory-side atomic operations like those in the SGI Origin 2000 (MAOs), and AMOs, where applicable. All results are scaled with the conventional shared memory version serving as the baseline.

5.1 Barrier

The SGI OpenMP barrier implementation uses LL/SC instructions. We created Atomic, ActMsg, MAO, and AMO-based variants. Table 4 shows the results of running these various barrier implementations on 4 to 256 processors (2 to 128 nodes). Active messages, MAOs, and AMOs all achieve noticeable performance gains, but AMO-based barriers achieve by far the best performance for all processor counts, ranging from a 1.9X speedup on two nodes to 54.8X on 128 nodes. The reason for these results follows.

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 need to retry. 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 LL/SC-based implementation.

Using atomic instructions eliminates the failed LL/SC attempts, but each increment causes a flurry of invalidation messages and data reloads, so the benefit of using Atomic is marginal.

The ActMsg barrier implementation sends an active message to the home node for every increment operation. The overhead of invoking the message handler dwarfs the time required to run the handler itself, but the benefit of eliminating remote memory accesses outweighs the high invocation overhead.

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, we spin on a local cacheable variable, an optimization similar to Nikolopoulos *at al.* [23]. However, as in the Atomic implementation, each increment causes a flurry of invalidation messages and data reloads.

The AMO version achieves much higher performance by eliminating the large numbers of serialized coherence operations present in the other barrier implementations. When each processor arrives at the barrier, it performs an AMO `inc` operation and then spins on the corresponding AMO result register. To optimize barrier performance, we can specify that no result should be returned until barrier count matches some trigger value, e.g., the number of threads expected to arrive at the barrier. When the barrier count reaches the trigger value, the AMU sends an update to every sharer of this cacheline as indicated by the sharing vector. [1] Upon seeing the AMO completion, each spinning thread will proceed beyond the barrier. 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 QsNet^{TM} used by the ASCI Q supercomputer [25]).

Barriers take great advantage of the small coalescer cache present in each AMU. All subsequent AMOs after the first `inc` operation will find the data in the coalescer and thus require only two cycles to process. Thus, for reasonable system configurations, the per-processor latency of AMO-based barriers is almost constant, since the typical roundtrip message latency dwarfs the time to process N increment operations at the home AMU. We do not assume that the network can physically multicast updates; performance would be even higher if the network supported physical multicast.

5.2 Spinlocks

Ticket locks [22] employ a simple algorithm that grants locks in FIFO order. Anderson's array-based queue locks [4] use an array of spinlocks to alleviate the interference between readers and writers, which is severe under contention. We pad the array to eliminate false sharing. Both algorithms still suffer from serialized invalidate-permission-write delays when implemented using conventional shared memory.

Table 5 presents the speedups of different ticket and array-based queuing locks compared to LL/SC-based ticket locks. For traditional mechanisms, ticket locks outperform array locks on fewer than 32 processors (on 16 nodes), while array locks outperform ticket locks for larger systems. This result confirms the effectiveness of array locks at alleviating hot spots in large systems.

Our results show that using AMOs dramatically improves the performance of both types of locks and negates the difference between ticket and array locks. In contrast, the other optimized lock implementations do not have clear advantage over their LL/SC-based counterparts on large systems. For more detailed discussions on applying AMOs to synchronization operations, please refer to Zhang *et al.* [39].

5.3 GUPS

Figure 3(a) contains the core loop of the GUPS microbenchmark, which atomically increments random fields of a large histogram array. We use a 256-megabyte histogram array, which is tiny compared to the real workloads that GUPS models [17], so our results are conservative for AMOs, whose performance is independent of array size. GUPS exposes the memory system bottlenecks of current computer architectures. Specifically, TLB and cache hit rates are extremely low. In real-world applications, most histogram accesses miss in the TLB. As the number of active processors increases and the histogram array is spread across more processors, the number of remote cache misses increases. With four processors, remote memory stalls account for 66% of execution time, which increases to 88% for 128 processors, even when we employ aggressive superpaging to eliminate all TLB misses.

Note that using the incoherent scatter-gather memory operations that are available on many vector machines can lead to race conditions and program errors, because multiple threads may attempt to increment the same histogram field at the same time. Even with coherent memory, either atomic adds or complex software techniques like segmented scan are required to ensure correctness. Our baseline implementation employs non-atomic adds, without locking, so the baseline performance results are optimistic. In contrast, our ActMsg and AMO implementations use atomic adds. To filter out the interference effect of active messages, we reserve one processor per node to handle active messages, so ActMsg results are optimistic. In the baseline and AMO-based implementations, the second processor on each node sits idle. MAOs are not considered because they work in uncacheable memory space, and simply disabling caching for GUPs will hurt performance.

In Figure 3(b), we report speedups of ActMsg and AMOs over baseline on different system configurations. ActMsg is very effective for GUPs. The performance of the AMO version is about twice of the ActMsg version with and without superpages. The main reason that both mechanisms are so effective is that they eliminate substantial network traffic induced by remote misses. Figure 3(c) shows the number of network packets (in thousands) sent for some test cases with superpaging. On average, active messages reduce network traffic by a factor of 4.3, while AMOs reduce network traffic by 5.5X.

5.4 STREAM and SAXPY

The performance of stream operations is dependent on how data is distributed across nodes. Since there is no accepted "typical" data distribution, we distribute data such that 50% of the operand data in multi-stream operations must be loaded from a remote node. For example, half of the data traffic for *memcopy* is within the local node and half is copied between different nodes. For *scale*, half of the processes use local memory and the other half fetch data from a remote node.

Table 6 shows the performance of AMOs on these benchmarks for different system sizes using the default setup of Table 2. *h100* and *h200* denote systems with a network hop delay of 100 and 200 procesor cycles, respectively. In all cases, AMOs perform very well, with performance as much as 38X faster than the baseline implementation. In addition, we can see that as remote memory latencies increase, the benefits of AMOs increase.

Except for *triad*, the MIPSpro compiler is able to perform aggressive loop unrolling and inserts near-optimal prefetch instructions. As a result, the baseline versions of these benchmarks suffer very few cache misses. However, since the processor can process data faster than the system bus can transfer it, the system bus becomes a performance bottleneck. In contrast, AMOs execute below the system bus, so bus bandwidth is not a bottleneck. AMOs exploit the high bandwidth within the memory controller and saturate the DRAM backend.

5.5 Database Queries

In this section we present the performance derived by applying AMOs to database queries from the OSDB and Set Query benchmark suites. *Info_Retrieval* and *Total_Report* are representative queries from the OSDB benchmarks suite. *Info_Retrieval* uses six fields from each database record, while *Total_Report* uses three. *Info_Retrieval* performs one aggregate operation (`count`) while *Total_Report* performs seven (`min`, `max`, and `count` on different attributes). The Set Query benchmarks perform a number of condition tests on each data record. They are representative of a variety of document search, marketing, and decision support workloads.

Figures 4 and 5 present the speedups of the ActMsg- and AMO-optimized database engines compared to conventional implementa-

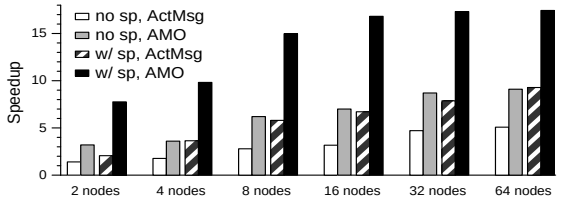
[1] AMOs keep sequential consistency except for the barrier, which is release consistent because of the delayed update mechanism. Since this operation is used exclusively for barriers, this semantic is completely acceptable.

Nodes, CPUs	LL/SC		Atomic		ActMsg		MAO		AMO	
	ticket	array	ticket	array	ticket	array	ticket	array	**ticket**	array
2, 4	1.00	0.41	0.91	0.52	1.12	0.50	1.01	0.41	**2.09**	1.24
4, 8	1.00	0.46	0.86	0.54	1.70	0.46	1.05	0.50	**2.35**	1.74
8, 16	1.00	0.50	0.97	0.56	2.27	0.53	1.10	0.50	**2.32**	2.27
16, 32	1.00	0.55	0.99	0.63	2.37	0.53	1.07	0.51	**2.38**	1.95
32, 64	1.00	1.66	0.87	1.69	0.67	1.47	0.67	1.51	**6.39**	5.01
64, 128	1.00	2.80	1.14	2.68	0.89	2.44	0.79	2.52	**11.00**	10.99
128, 256	1.00	3.55	1.24	3.44	1.00	3.01	0.85	2.99	**13.58**	11.35

Table 5. Speedup of various spinlocks compared to LL/SC-based spinlocks.

```
for (i = 0; i < accesses; i++)
    A[ idx[i] ] += v;
```

Array `idx` contains duplicate values. E.g., `idx[1]` and `idx[3]` may both give the same index into `A`.



Nodes	baseline	ActMsg	AMO
2	1358	302	262
8	2519	577	459
32	2931	688	516

(a) code (b) speedup (w/ and w/o superpaging) (c) traffic (with sp)

Figure 3. GUPS code and performance (using a dedicated ActMsg handler processor).

tions. The baseline queries have been highly optimized, including evaluating high-selectivity and low-cost predicates first, and manually inserting aggressive data prefetches. For example, in both single-node and multi-node settings, the L1 cache hit ratio for *Info_Retrieval* and *Total_Report* are 99.5% and 98.3%, respectively. For multi-node experiments, we employ the "50% remote" distribution and hop latency variations described in Section 5.4.

As can be seen in the figures, AMOs speed up the various database queries by factors ranging from 2.1X to 4.4X compared to their respective baselines. The ActMsg variants perform poorly at small node counts, but achieve 50-75% of the benefits of AMOs for larger configurations.

5.6 Sources of AMO Performance Gain

The benchmarks described in Sections 5.1 through 5.5 benefit from different aspects of our AMO design. Since space is limited and the database queries are much more complex than the other benchmarks, our sensitivity analysis is limited to the database query benchmarks.

In this subsection, we quantify the extent to which each of the potential sources of AMO performance improvement affect the database query results. The four potential benefits of AMOs that we consider and isolate are:

- localizing computation by performing the work at the data's home node
- utilizing specialized hardware for specific operations such as `max` or `min`
- accessing sparse data using short DRAM bursts as opposed to full cache line bursts
- exploiting stream-level parallelism in grouped AMOs

To better understand how AMOs are able to improve database engine efficiency, we perform a number of experiments that isolate the impact of the various features of AMOs. The first factor, localizing computation to avoid remote memory accesses, benefits both AMOs and ActMsgs; the other three are unique to AMOs, and explain the performance difference between ActMsgs and AMOs. The four factors are not orthogonal; each one is only profitable if it alleviates a system bottleneck for a particular application. Alleviate a bottleneck beyond the point where other system components have become the primary performance bottleneck and we will see diminishing returns. For space limitations, we only included experiments that highlight the influence of the variable that is under investigation. Figure 6 presents the results of our sensitivity analysis. The metric is AMO speedup over baseline. The black bars are the ones already shown in Figures 4 and 5 using the default configuration.

Computation Localization

By shipping the computation to data's home node, remote loads are converted to local loads. Thus, network bandwidth and latency limitations are avoided for both active messages (ActMsg) and AMOs, while half of the data in the baseline version is fetched from across the network. Since only a single active message is sent per page in the ActMsg variant of each benchmark, message handler invocation overhead is negligible. Thus, the speedup achieved by using active messages shown in Figures 4 and 5 represents the benefit of computation localization. As can be seen in the figures, the contribution of computation localization to AMO performance is modest for small configurations, but substantial for larger ones.

Specialized Hardware

AMUs have specialized hardware designed for efficient stream processing, e.g., an AMU can compute the max of two integers in a single cycle after the operands are available, whereas doing so on a conventional CPU requires several load, store, comparison, and branch instructions. Of the database queries discussed here, *Total_Report* benefits the most from specialized function units.

Figure 6(a) shows how *Total_Report* performance changes with varying latencies for `max`, `min`, and `count`. For example, if it takes the AMU 4 memory controller cycles (20 processor core cycles) to perform a `max/min/count` operation, the AMO code can achieve a speedup of 2.9 on single node, down from 3.7X with faster ALUs. This result motivates the addition of common simple stream operations (e.g., `min`, `max`, and `popcount`) when the area overhead is small and the performance impact is significant.

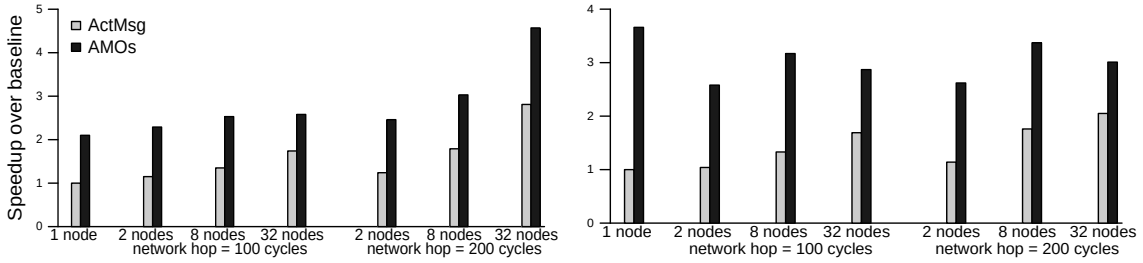
Fine-grained DRAM Accesses

Rather than loading entire cache lines (e.g., 128 bytes) from DRAM, the AMU loads as few bytes as the DRAM backend allows. For stream operations, the AMU only loads the referenced fields and not the entire cache line. For masked stream operations, the AMU only loads stream elements that correspond to a '1' bit in the mask stream. In the *Info_Retrieval* query, the first predicate evaluation results in a bit mask that eliminates 90% of the tuples, so subsequent AMOs need only operate on 10% of the tuples. In Q4 of Set Query (Figure 5(c)), the first predicate evaluation filters out 80% of the tuples. Thus, AMUs have ample opportunity to exploit short DRAM bursts for stream operations.

In our design, AMUs load data from DRAM in 32B bursts, whereas

nodes, procs	memcopy		scale		sum		triad		saxpy	
	h100	h200	h100	h200	h100	h200	h100	h200	h100	h200
1,1	1.33	1.33	1.08	1.08	2.01	2.01	4.27	4.27	1.58	1.58
1,2	1.17	1.17	1.12	1.12	2.05	2.05	4.54	4.54	1.09	1.09
4,8	1.19	1.22	1.29	1.73	4.17	4.44	7.97	12.09	1.24	1.45
32,64	1.31	1.68	2.11	2.99	9.34	12.23	17.74	25.72	2.04	3.36
128,256	1.49	2.13	2.58	3.74	11.94	19.13	21.50	38.37	2.47	4.82

Table 6. Speedup of AMOs on STREAM and SAXPY



(a) *Info_Retrieval* (b) *Total_Report*

Figure 4. Optimizing the OSDB Benchmark

the baseline system loads 128B cache lines. DRAM vendors (e.g., Rambus) have started to enable even shorter DRAM burst lengths. Figure 6(b) shows the performance of four queries as we vary DRAM burst lengths. These experiments confirm that loading data at smaller granularity (32B) for strided accesses greatly improves memory performance. In particular, using AMOs for the two Set Query benchmarks on single node systems hurts performance (speedup < 1) if load elements via 128-byte DRAM bursts. Decreasing the minimum DRAM burst length from 32 bytes to 4 bytes provides a marginal benefit for our experiments, because with 32-byte bursts memory bandwidth is not a bottleneck when there are two processors and four DRAM channels per node.

Greater Parallelism

In the baseline implementations, stream operations are performed serially. Each query predicate is evaluated on the entire database table in its entirety before the next predicate is evaluated. Although there is substantial instruction-level parallelism, performance is limited by processor reorder buffer size, the number of physical registers, the number of MSHRs, and other factors. In contrast, AMOs exploit stream-level parallelism. Each AMO in a group of AMOs can proceed in parallel using different stream buffers, thereby fully exploiting the available DRAM bandwidth. Further, the use of temporary streams allows results from one stream operation to be bypassed directly to the the FU where they will be consumed, which minimizes SB accesses. OSDB `Total_Report` and Set Query `Q4` both exploit this form of stream parallelism. The gray bars in Figure 6(c) show the performance of these queries when we ran them using serial (non-grouped) AMOs. Doing so reduced performance by 10-20%.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a mechanism *Active Memory Operations* (AMOs), which allow programmers to ship select computation to the home memory controller of data. Doing so can eliminate a significant number of remote memory accesses, reduce network traffic, and hide the access latency for data with insufficient reuse to warrant moving it across the network and/or system bus. AMOs offer an efficient solution for an important set of computation patterns. Through simulation, we show that AMOs can lead to dramatic performance improvements for data-intensive operations, e.g., up to 50X faster barriers, 12X faster spinlocks, 8.5X-15X faster stream operations, and 3X faster database queries.

These results motivate us to continue this line of research. One limitation of the current AMO design is that it supports only a small set of operations. Another direction of future work is using one or more imple in-order processor cores to implement the AMU. Such a design would significantly complicate the programming model, but provide richer opportunities for a variety of applications.

7. References

[1] TPC-D, Past, Present and Future: An Interview between Berni Schiefer, Chair of the TPC-D Subcommittee and Kim Shanley, TPC Chief Operating Officer. available from `http://www.tpc.org/`.

[2] J. H. Ahn, M. Erez, and W. J. Dally. Scatter-add in data parallel architectures. In *HPCA-11*, pp. 132–142, Feb. 2005.

[3] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a modern processor: Where does time go? In *VLDB-25*, pp. 266–277, Sept. 1999.

[4] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE TPDS*, 1(1):6–16, Jan. 1990.

[5] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proc. of the 25th ISCA*, pp. 3–14, 1998.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB-25*, pp. 54–65, 1999.

[7] D. Patterson *et.al.* A case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2):34–44, Apr. 1997.

[8] Z. Fang. Active memory operations, Ph.D thesis, University of Utah. 2006.

[9] 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.

[10] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems, Chapter 6*. 1993.

[11] M. Hall, et al. Mapping irregular appilcations to DIVA, a PIM-based data-intensive architecture. In *SC'99*, Nov. 1999.

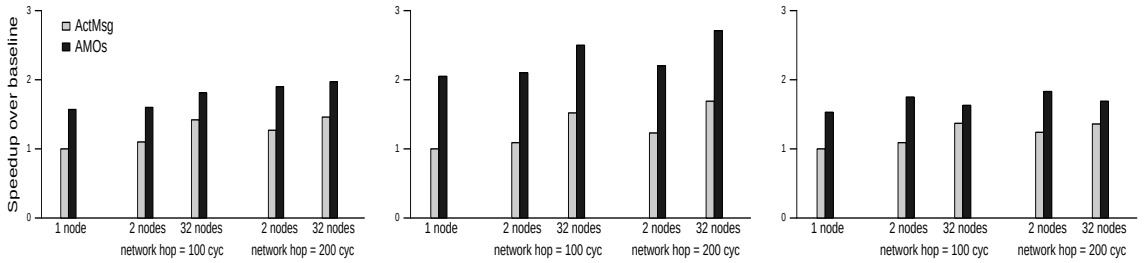
[12] Hewlett-Packard Inc. The open source database benchmark.

[13] Intel Corp. Intel Itanium 2 processor reference manual.

[14] International Technology Roadmap for Semiconductors.

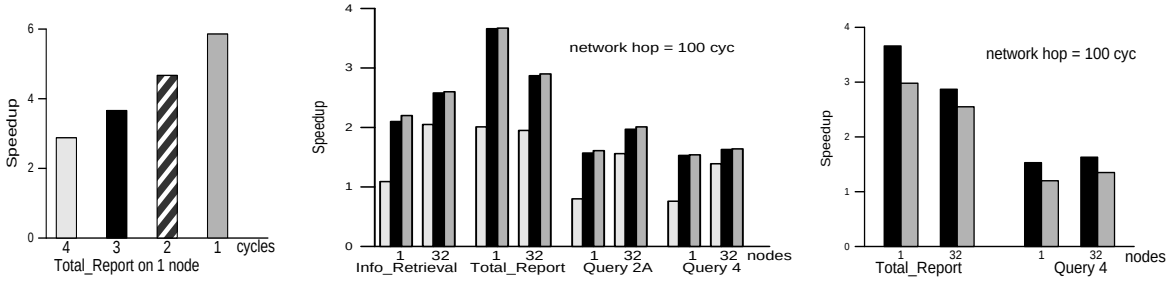
[15] K. Keeton and D. Patterson. *Towards a Simplified Database Workloads for Computer Architecture Evaluation*. 2000.

[16] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight. Architectural support for uniprocessor and multiprocessor active memory systems. *IEEE Trans. on Computers*, 53(3):288–307, Mar. 2004.



(a) *Query 2A* (b) *Query 3B* (c) *Query 4*

Figure 5. Optimizing the Set Query Benchmark



(a) *ALU latency* (b) *DRAM access granularity (128/32/4 bytes)* (c) *AMO grouping (on/off)*

Figure 6. Quantifying the performance contribution of various aspects of AMOs

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

[18] P. Kogge. The EXECUBE approach to massively parallel processing. In *International Conference on Parallel Processing*, Aug. 1994.

[19] J. Kuskin, et al. The Stanford FLASH multiprocessor. In *Proc. of the 21st ISCA*, pp. 302–313, May 1994.

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

[21] J. McCalpin. The stream benchmark, 1999.

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

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

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

[25] F. Petrini, et al. Scalable collective communication on the ASCI Q machine. In *Hot Interconnects 11*, Aug. 2003.

[26] T. Pinkston, A. Agarwal, W. Dally, J. Duato, B. Horst, and T. B. Smith. What will have the greatest impact in 2010: The processor, the memory, or the interconnect? HPCA8 Panel Session, 2002.

[27] R. Rajwar, A. Kagi, and J. R. Goodman. Improving the throughput of synchronization by insertion of delays. In *Proc. of the Sixth HPCA*, pp. 168–179, Jan. 2000.

[28] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st ISCA*, pp. 325–336, Apr. 1994.

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

[30] SGI. *SN2-MIPS Communication Protocol Specification*, 2001.

[31] SGI. *Orbit Functional Specification, Vol.1*, 2002.

[32] M. Shao, A. Ailamaki, and B. Falsafi . DBmbench: Fast and accurate database workload representation on modern microarchitecture. TR CMU-CS-03-161, Carnegie Mellon University, 2003.

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

[34] P. J. Teller, R. Kenner, and M. Snir. TLB consistency on highly-parallel shared-memory multiprocessors. In *21st Annual Hawaii International Conference on System Sciences*, pp. 184–193, 1988.

[35] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proc. of IPDPS*, page 84a, Apr. 2003.

[36] J. Torrellas, A.-T. Nguyen, and L. Yang. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Proc. of the 7th HPCA*, pp. 15–25, Jan. 2000.

[37] 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.

[38] L. Zhang. UVSIM reference manual. TR UUCS-03-011, University of Utah, May 2003.

[39] L. Zhang, Z. Fang, and J. B. Carter. Highly effi cient synchronization based on active memory operations. In *IPDPS*, Apr. 2004.

[40] 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.