J Supercomput (2012) 62:510–549
DOI 10.1007/s11227-011-0735-9

Active memory controller

Zhen Fang · Lixin Zhang · John B. Carter · Sally A. McKee · Ali Ibrahim · Michael A. Parker · Xiaowei Jiang

Published online: 17 January 2012

Abstract Inability to hide main memory latency has been increasingly limiting the performance of modern processors. 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 an intelligent memory and cache coherence controller (AMC) that can execute Active Memory Operations (AMOs). AMOs are select operations sent to and executed on the home memory controller of data. AMOs can eliminate a 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.

The work was done when most of the authors were at the University of Utah. The views and conclusions contained herein are those of the authors and should not be interpreted as representing those, either express or implied, of Intel, CAS, IBM, Chalmers, AMD, nVidia, or the University of Utah.

Z. Fang (✉) · M.A. Parker
nVidia Corporation, Santa Clara, USA
e-mail: fangzhen@gmail.com

L. Zhang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

J.B. Carter
IBM Austin Research Lab, Austin, USA

S.A. McKee
Chalmers University of Technology, Gothenburg, Sweden

A. Ibrahim
AMD, Sunnyvale, USA

X. Jiang
Intel Labs, Intel Corporation, Santa Clara, USA

In this paper, we present the microarchitecture design of AMC, and the programming model of AMOs. We compare AMOs' 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 50× faster barriers, 12× faster spinlocks, 8.5×–15× faster stream/array operations, and 3× faster database queries. We also present an analytical model that can predict the performance benefits of using AMOs with decent accuracy. The silicon cost required to support AMOs is less than 1% of the die area of a typical high performance processor, based on a standard cell implementation.

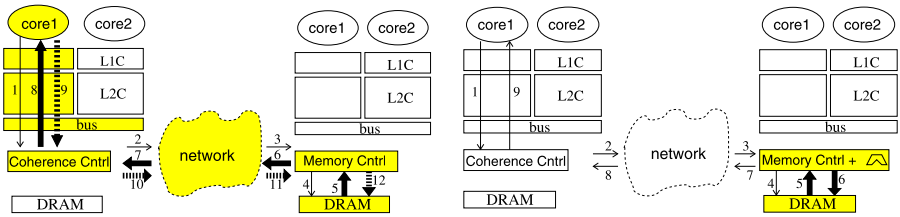
Keywords Distributed shared memory · Cache coherence · Memory architecture · Interprocessor synchronization · DRAM organization

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. The predominant scalable DSM architecture is directory-based CC-NUMA (cache-coherence nonuniform memory access), such as employed in the SGI Origin™ [43] and Altix™ lines. 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, 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 [39]. This long latency consists of three parts: local DRAM latency (often referred to as the "Memory Wall"), memory controller occupancy, and internode network latency. In addition, cross-section bandwidth is a limiting factor in the scalability of large-scale (100+ node) DSM systems.

Caching is the standard mechanism for improving local memory performance and reducing remote traffic. For operations with low temporal locality (e.g., searching a large database table looking for records that match some search criterion) or significant write sharing (e.g., updating a barrier count), 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 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. This problem is particularly serious for data-intensive scientific and commercial applications [27]. The unavoidable conclusion is that reducing remote coherence traffic and internode data transfers is essential for DSM systems to scale effectively.

For applications with low temporal locality, there are two important, but we believe avoidable, sources of performance degradation: (1) *indiscriminately moving*



(a) compute in a CPU of the issuing node (b) compute in the MC at data's home node

Fig. 1 Moving computation to data's home node memory controller

data through the intranode memory hierarchy, and (2) *indiscriminately moving data to the node that issues the computation.* The work presented here represents an effort to overcome these design flaws of conventional DSM machines and alleviate memory system bottlenecks for some of the most data-intensive applications. For such operations, it would be faster and more power efficient to ship the operations to where the data resides and (optionally) return only the result of the operation to the processor initiating the operation.

We propose to add a new hardware component, **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. Such an AMU-augmented memory and cache coherence controller is called an **Active Memory Controller** (AMC). **Active Memory Operations** (AMOs) are issued by processors, forwarded to the home node of their operands, and then get executed in the AMC.

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. Figure 1 motivates the benefits of AMOs. On a conventional system (Fig. 1(a)), when *core1* wishes to operate on a piece of remote data, the data must be loaded across the network and into *core1*'s cache, which entails interacting with two or more remote nodes. If *core1* modifies the data, it must be written back to memory at some later time. Even for locally homed data, internode communication may be needed if the data is cached remotely. Using AMOs, short messages are sent to the data's home node, which acquires a globally cache-coherent copy of the data, performs the requested operations, and (optionally) returns the result to *core1*. This is shown in Fig. 1(b).

Figure 2 depicts some common cache miss scenarios in a three-node multiprocessor system and contrasts the communication induced by AMOs versus a conventional system. Circles are nodes. Shaded nodes are where the data are processed. In each case, AMOs have a lower communication overhead and latency than loading the data across the network and performing the operation locally.

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. Stream AMOs can be masked. All AMOs are cache-coherent;

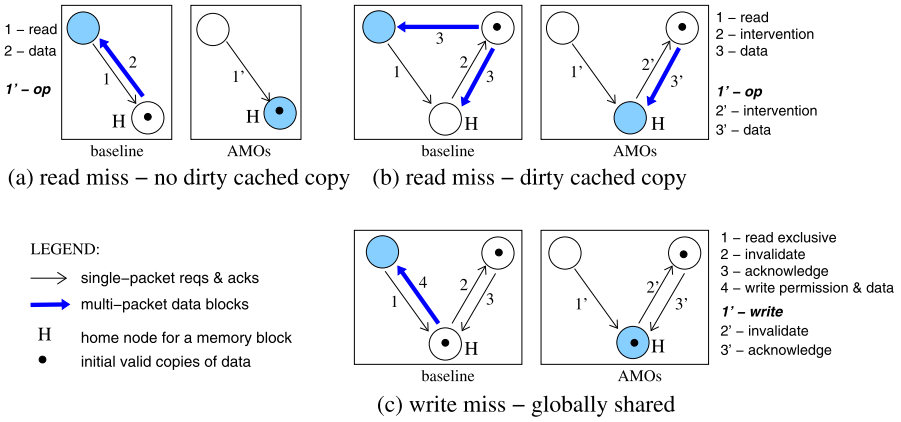


Fig. 2 Examples of cache misses in a directory-based cc-NUMA system

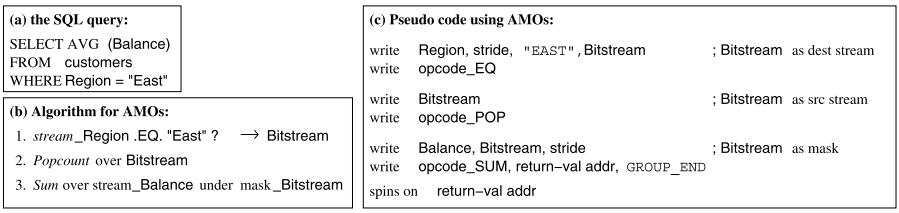


Fig. 3 Example of using AMOs for a simple SQL query

the AMU hardware performs necessary coherency operations before it uses any data. Before jumping into further detail, we illustrate application of stream AMOs using a simple example.

Figure 3 shows how AMOs can be used to implement a simple SQL query that computes the average balance of all customers in the "East" sales region. To perform the query, 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 Nth 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 3rd AMO) divided by the number of customers in the east region (result of the 2nd AMO).

For many operations, an AMO can replace thousands of memory block transfers. As described in Sect. 5, AMOs can lead to dramatic performance improvements for data-intensive operations, e.g., up to 50× faster barriers, 12× faster spinlocks, 8.5×–15× faster stream/array operations, and 3× 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. PIMs exploit the high memory bandwidth and low latency available inside the DRAM package [40]. Representative PIM systems include EXECUBE [28], IRAM [37], DIVA [18], HTMT [9], Active Pages [36], and FlexRAM [45, 51]. Both AMOs and PIMs exploit affinity of computation to main memory, but they differ in three important ways. First, AMOs use commodity DRAMs, which have higher yield and lower cost than PIMs. Second, AMOs operate on coherent data, while the processors in a PIM reside below the architecture level where cache coherence is maintained. Thus, if the data required by a PIM operation resides off-chip, PIMs effectively becomes a form of noncoherent distributed memory multiprocessor, with all of the attendant complexities. Third, PIMs employ merged logic-DRAM processes, which are slower and less power-efficient than processes tuned for logic. So, an AMO processor built using the same generation process will be somewhat faster and more power-efficient than its PIM equivalent. The major benefit of PIMs is the very high bandwidth of the on-chip connections between processors and storage. However, today's high-performance memory controllers actually have raw bandwidth comparable to what is available within DRAM. In general, the potential power and performance benefits of placing the processing units on the DRAM chip are dwarfed by the higher cost and complexity of PIM systems. Overall, we strongly believe that the appropriate place to perform offloaded computation is at the memory controller, which has most of the bandwidth and power advantages of PIMs and all of the coherence benefits of processors.

Several research projects have proposed adding intelligence to the memory controller. Examples include the Impulse memory controller [57], Active Memory [26], and memory controller-based prefetching by Solihin et al. [46]. 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 [16] was the first to implement atomic instructions in the memory controller. The FLASH [30] multiprocessor supported atomic `fetch_ops`, which were subsequently supported by the SGI Origin 2000 [31] and Cray T3E [41]. 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 noncoherent and rely on software to maintain coherence. They are used primarily to support efficient thread synchronization. Nikolopoulos [35] describes an optimized usage of the MAOs. Blelloch et al. [7] extend scalar active memory operations to support more protocols and algorithms. Kumar et al. [29] implement atomic instructions in the vector form and use them to optimize certain Recognition, Mining, and Synthesis (RMS) workloads.

Originally proposed for message passing, Active Messages [53] are the most similar in spirit to AMOs with their message-driven style of computing. 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 that AMOs avoid. Another difference between AMOs and active messages is that the active message programmer must know which node holds the desired data. In contrast, the target of an AMO is extracted by hardware based on the address of the target data. Nevertheless, active messages often perform well for the same reason that AMOs do—it is far less efficient to load data over the network to operate on it than to simply operate on it in place.

Finally, a number of researchers have proposed offloading select computation from the main processor. Several recent cluster interconnects support distributed synchronization operations [38, 49], e.g., the Quadrics™QsNet interconnect [38] supports both pure hardware barriers and network processor-based hybrid barriers. Zotov [59] supports the barrier operation using a dedicated network. Other researchers have proposed adding specialized vector operations to the memory controller to support vector scatter-add [1] or parallel reduction operations [15]. The former works well for applications that are insensitive to floating point rounding errors and whose working set can fit into the caches, but requires programmers to handle the temporarily incoherent states of the affected data. In contrast to these proposals, AMOs support a much broader set of operations and maintain full coherence.

The Active Switch [19] system adds a complete microprocessor to each switch in a DSM multiprocessor. Data gets preprocessed as it streams through an active switch. The functions that benefit the most from Active Switches are those that incur huge amounts of data transfers, whereas AMOs can avoid data transfers entirely. Currently, Active Switches require in-order, contiguous delivery of successive data chunks, which adds programming and/or hardware complexity to ensure or to handle when not present. Yoo et al. [54] use small processor cores near memory to speed up applications that are highly parallizable. They rely on software to maintain cache coherency. Zhao et al. [58] present the design of a copy engine (CE) that performs bulk movement of blocks. A CE-based memory copy and our `memcopy` AMO differ in two ways. First, they add a new instruction to the base machine ISA. This is an invocation method that we managed to avoid. They also modify the memory dependency check logic in the processor load/store unit. Instruction pipeline stalls are improved at the cost of increased hardware cost and power consumption of the load/store unit. The second difference is that the CE cannot copy blocks between distributed memory modules.

3 Active memory controller design

In this section, we present a description of the operations supported by AMUs (Sect. 3.1), the AMO programming model (Sect. 3.2), and the design of an AMU (Sect. 3.3).

Table 1 Currently supported AMOs

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

3.1 Supported operations

Our current AMU design supports two types of operations: *scalar operations* and *stream operations*. Table 1 lists the operations that are currently supported, which were selected based on application requirements and hardware implementation complexity. In the table, "S" indicates a stream operand, "s" indicates a scalar operand, "aop" indicates an arithmetic operation (`add`, `sub`, `mul`), and "lop" indicates a logic operation (`and`, `or`, `nand`, `eq`, `gt`, `ge`, `lt`, or `le`). Most AMOs include both integer and single precision floating point versions. All AMOs are coherent; the AMU hardware performs all necessary coherency operations before it uses any data. Because AMUs are integrated with the directory controller, making AMOs coherent is fairly simple and fast (Sects. 3.2.2 and 3.3).

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) or other processor-centric synchronization primitives, AMO-based synchronization does not require cache lines to bounce between processors as they reach synchronization points. To support scalable barriers, we provide a special option to the `inc` operation. Normally, the increment operation increments the target memory location and returns the old value. We support a variant in which the processor can specify a "test" value and request to be signaled only when the count reaches the test value. This mechanism reduces the number of network roundtrips required to implement a barrier to $2N$, all of which can be pipelined, where N is the number of threads synchronizing. AMO-based spinlocks can implement efficient signaling using a similar mechanism.

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. This will be explained in Sect. 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, `S-s-add` adds a scalar value to every element in the stream, which generates a second stream, while `S-s-.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.

3.2 Programming model

3.2.1 Invocation and completion

There are a number of ways that AMOs could be issued, e.g., ISA extensions, memory-mapped I/O operations, or address remapping.

We chose memory-mapped uncached I/O space reads and writes to communicate between the processor and AMO engine. In our design, AMOs are decoupled (asynchronous) operations. The local processor issues AMOs and then is free to perform other operations while the AMO is underway. When the local processor needs the return value or simply needs to know that the AMO operation has completed, e.g., to determine that all threads have arrived at a barrier, it spins waiting on a completion bit to be set.

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 [41].

Using physical I/O registers for the uncached memory addresses is purely a performance optimization. In the rare case that many threads on the same node use AMOs and the return value registers are used up, the OS backs the memory locations with physical memory. A process is expected to release the uncached memory locations explicitly, but it is transparent to the application whether the memory locations are mapped to registers or real memory.

3.2.2 Basic AMOs

After software has written the arguments into an AMO issue 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 that node, which is where the operation is to be performed. Each AMU has an external TLB to perform virtual to physical translation, similar to Cray T3E [41] and Impulse [57]. The home node AMU interacts with other processors to acquire a globally coherent copy of the data in the appropriate sharing mode. For example, if the operation involves reading data, the home node AMU acquires a coherent shared copy. If the operation modifies the data, the home node invalidates any remotely cached copies and acquires ownership of the data. Once the AMU has a coherent copy of the data, it performs the operation and signals completion of the AMO to the processor that issued the AMO, optionally returning a scalar result. For

streamed AMOs that span multiple cache lines within a single page, all of which are homed by the same node, 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 for consumption.

3.2.3 Grouped AMOs, temporary streams, and masks

Semantically, AMOs operate on main memory locations: data is read from main memory and results are written back to main memory. However, for operations involving several AMOs, e.g., the simple database operation presented in Fig. 3, it is inefficient to write intermediate results back to main memory, only to be immediately reread and reused as part of a subsequent AMO. As a performance optimization, we allow programmers to *group* related AMOs and specify that certain streams are *temporary streams* that will only be used as inputs to subsequent AMOs and thus 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 (i.e., otherwise invalid) addresses for their locations. Temporary streams can be used to pass intermediate values efficiently from the output of one stream operation to the input of another stream operation. For example, the Bitstream bit mask used in the database example shown in Fig. 3 would be an ideal candidate to be treated as a temporary stream. The AMU optimizes temporary streams; they 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 the data will be written back. The hardware required to support grouped AMOs and the way in which grouped AMO operations are implemented is described in more detail in Sect. 3.3.

The third AMO in Fig. 3 illustrates a *masked AMO*, which is analogous to the masked vector operations present in 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.

3.2.4 Compiling for AMOs

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 rather 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 concerns arise primarily when using stream AMOs.

Although AMOs and vector operations have drastically different data granularity and hardware implementations, stream AMOs can be considered a special class of vector operations with respect to compilation. Both break array operations into a loop of SIMD computation. 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: loop distribution, loop interchange, scalar renaming, conditional handling, index set splitting, loop peeling, idiom recognition, etc. 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, presented in Sect. 6.

3.2.5 Other programming considerations

Since primitive stream operations are restricted to operating on streams that reside entirely 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. Large reduction operations implemented without AMOs are typically parallelized also, but with AMOs the base reduction operations run much faster.

The streams in a single AMO or grouped AMOs can have different home nodes. In these cases, the AMOs 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 the group are handled by a single AMU.

3.2.6 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. Software optimizations like data privatization and array expansion are some of the programming techniques that an OpenMP programmer needs to apply on today's machines. To achieve good performance, programmers would need to tune their code to match the underlying cache hierarchy, data distribution, and possibly coherence protocols. 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.

For AMOs, computation-to-data affinity is achieved through hardware, 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.

3.2.7 Context switches, AMO register virtualization, and exception handling

In our current 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. As a result, the OS need not save or restore AMO registers during a context switch. 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. In our current design, 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. The memory controller issues an interrupt that causes the OS to be invoked to handle the fault. Currently we pin pages that are the target of AMOs in main memory, so all such exceptions represent bugs and process termination suffices. To support AMOs to pageable space, the OS could instead reload the page in question from disk and reissue the failed AMO.

3.3 Hardware organization

Figure 4 is the scalable distributed shared-memory system that we assume as a baseline platform. It is similar to the widely-deployed SGI Origin 3000 architecture [43]. The nodes are connected by a fat-tree network. Within each node, two superscalar processors share a snoopy bus. The memory and directory controller (MD Cntrl), directory cache, coherence-related structures (Coh.Struct.), network interface and I/O interface compose a hub. The components within a hub communicate with each other through a crossbar. The AMU sits alongside the memory controller in the hub. The processor(s), crossbar, and memory controller are on the same die, separate from the DRAM chips.

3.3.1 AMU microarchitecture

Figure 5(a) zooms in on the AMC. The AMU consists of three components: a scalar unit, a stream unit, and a set of auxiliary structures. The scalar unit executes scalar

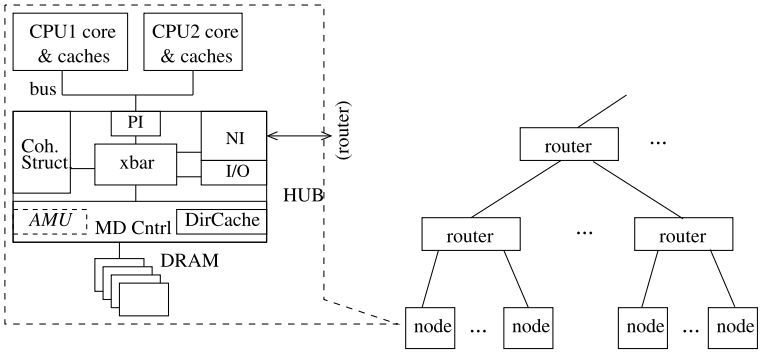
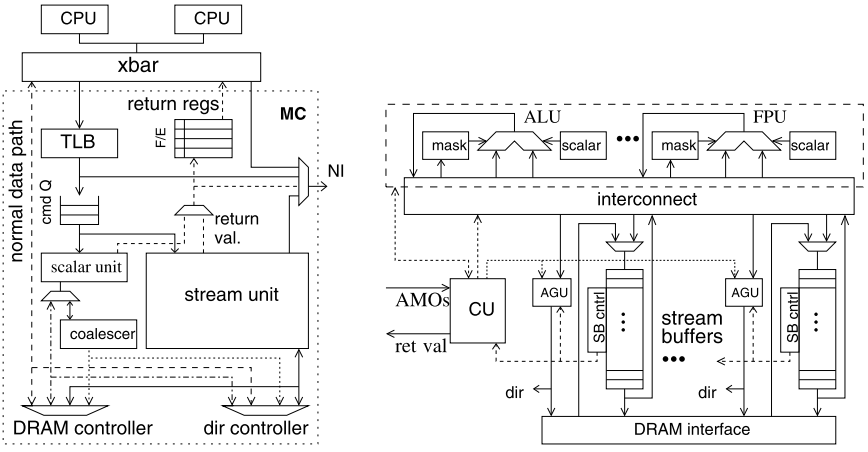


Fig. 4 The assumed DSM architecture



(a) AMC, showing components of the AMU (b) The Stream Unit

Fig. 5 The active memory controller

AMOs. The stream unit performs stream AMOs, and accounts for most of the design complexity of the AMU. The auxiliary structures include the AMU command queue, return value registers, and address translation hardware. AMO functionality is not on the critical path of normal memory references. Most of the microarchitectural design we discuss here is transparent to software.

When a processor initiates a scalar AMO or stream-scalar 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, similar to the external TLBs in Cray T3E [41] and Impulse memory controller [57]. 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 nontemporary target stream addresses and forwards the request to this node. In our current design, when there is more than one possible destina-

Step 1. The following actions are performed in parallel:
Step 1–a. For every destination stream in the group, allocate an SB from the free list.
Step 1–b. Read the mapping table for the source streams.
Step 1–c. Check the dependencies among the grouped AMOs.
Specifically, addresses of the destination streams are compared against those of the source streams of all following AMOs in the group.
Step 2. If a dependency is detected, substitute the SB allocated to the producer AMO for the SB allocation that was read from the mapping table.
Update the mapping table after dependencies are resolved.
Step 3. Allocate SBs to unmatched source streams from the free list.

Fig. 6 The stream buffer allocation algorithm

tion node, the source AMU selects the one that results in the fewest internode stream transfers. For grouped AMOs, if the inter-node network does not guarantee in-order delivery, the hardware attaches in-group sequence numbers to the forwarded AMO messages. 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. Since stream AMOs can be issued by different threads in the system and arrive at the same AMU, each entry in the AMU command queue has a thread ID tag.

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 (e.g., data being used to implement spinlocks and barrier counts). In our study, four single-word entries in the coalescer can cache all hot scalar data.

3.3.2 The stream unit

Figure 5(b) is an enlargement of the stream unit in Fig. 5(a). When the control unit (CU) sees the `GROUP_END` bit, it allocates stream buffers (SBs) to hold the operand and result streams associated with each AMO.

Stream buffer allocation Suppose the $N-1$ AMOs at the head of the AMU command queue in Fig. 5 have their `GROUP_END` bit cleared. This means that the programmer suggests that there might be stream reuse between the first N AMOs, such as shown in the example in Fig. 3. The CU needs to allocate stream buffers to the multiple streams that are involved in the group. This process is conceptually similar to dynamic register mapping in OoO processors. Similar to register mapping, the main data structures in stream buffer allocation are a free list of the SBs, and a table that contains the current stream-to-SB mappings. We give an algorithm for the stream buffer allocation logic in Fig. 6. The small address space of stream buffers and the huge space of page numbers lead us to a reverse mapping table along the lines of a design in Alpha 21264 [25]. Specifically, the number of entries equals to the number of stream buffers and stored in each entry are the stream address and up to $N-1$ AMO ids. The mapping table lookup in step 1-b is performed by doing an associative search using the stream address as the key. For a worst case that every AMO uses four streams (two input, one output, and one mask), step 1-b requires $3 \times N$ table searches, and step 1-c involves $\frac{3\times N\times(N-1)}{2}$ stream address comparisons. Remember

that stream AMOs are page-grained. Therefore, neither timing nor energy consumption is an issue in the stream buffer allocation logic. An implication of this is that we do not need multiple ports for the mapping table which are required for single-cycle register renaming.

Stream buffer organization The SBs are key to stream AMO performance. They hide DRAM latency and provide intermediate storage between components of grouped AMOs. The SBs do not occupy address space, and they do not have tags. 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 SBs effectively exploit *non-speculative* prefetching. 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. 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. This is the reason we use an SRAM instead of a FIFO for SBs.

Flexible DRAM burst lengths For nonunit stride streams, we exploit short DRAM burst lengths. While the unit of memory accesses on most conventional memory controllers is a cache line (system block), DRAM vendors actually offer finer grained data accesses. With DDR memory, a cache line load consists of a burst of several `min-grain` transfers. A typical min-grain value of today's DDR DRAM is 4 bytes. To optimize DRAM channel usage, our memory controller generates memory accesses using the following policy:

$$\text{memory access granularity} = \begin{cases} \text{DRAM min-grain} & \text{if stride} \geq \text{system block size} \\ \text{system block size} & \text{if stride} < \text{system block size} \end{cases}$$

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 a memory request returns from DRAM or from the network interface, it is directed to the appropriate SB according to its memory address. The SB controller can then calculate the SB address to use because of the strided access pattern. 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.

Cache coherency Our AMC design and quantitative performance evaluations assume directory-based cache coherence. 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 (e.g., it is a read operation and the local data is coherent). If not, the directory controller performs the necessary operations

to make the data coherent (e.g., issuing invalidate or flushback requests to remote sharers).

For snoop-based cache coherence, we provide a qualitative discussion on the efficiency of AMOs. Suppose we add an AMU to each of the memory controllers connected to a snoopy bus. To maintain cache coherence, the AMUs need to send probes to the snoopy bus. Since a bus naturally supports broadcast, AMO-optimized barriers and spinlocks will achieve better relative performance on a bus-based system than on a directory-based system with no multicast support.

For stream AMOs, coherence actions are performed on a per-block basis. Therefore, stream AMOs generate roughly the same amount of coherence traffic compared with a non-AMO snoop-based system. However, because AMOs do not fetch data across the system bus when memory blocks are fetched from local main memory, AMOs benefit from greater data bandwidth. Reduced contention for the bus will also allow a larger share of the bus bandwidth to be used for each of the non-AMO threads that compete for the same shared interconnect. Given the current trend of building many processor cores in a single chip, we expect higher contention for the shared interconnect, and thus greater performance benefits from using stream AMOs.

3.3.3 Silicon cost

Our current AMU design includes a 1024-entry TLB, five integer ALUs, four single-precision FPUs, 16 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™ to model the SRAM structures and implemented the rest in Verilog, which we synthesized using Synopsis™ tools. We conservatively estimate the area of an AMU to be no more than 2.6 mm^2 in a 90 nm 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 [22]. This silicon area is about 30% of an on-chip memory controller that is built using *custom* layout [23].

If AMOs were integrated into the processor so that AMO addresses could be translated by the processor TLB, the chip area would shrink to 1.5 mm^2. These area estimates are conservative, since they are derived using high-level design and synthesis tools. A custom implementation would likely be less than half the estimated size. This chip area seems worthwhile given the substantial performance benefits presented in Sect. 5.

4 Experimental setup

4.1 The Simulator

We use execution-driven simulation to evaluate the performance of AMOs. Our simulator, UVSIM [55], accurately simulates large-scale ccNUMA systems, including detailed processor, cache, bus, interconnect, memory controller, IO, and DRAM models. The system model is a hypothetical future-generation SGI supercomputer and

Table 2 System configuration

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

models the complete SGI directory-based coherence protocol [44]. Each simulated node models two superscalar processors connected to a high bandwidth bus. Also connected to the bus is a hub that integrates the processor interface, memory controller, directory controller, coherence engine, network interface, and IO interface. Each node contains a DRAM backend with a maximum of 16 GB of physical memory. We simulate a micro-kernel that has realistic memory management routines, supports most common Unix system calls and directly execute unmodified, 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 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-B 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 nonleaf 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 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%.

4.2 Extensions to the base simulator

To evaluate the performance of AMOs, we extend the baseline UVSIM model with several non-AMO architecture innovations that were not part of the baseline processor. We then extended the resulting system to model AMOs.

We extended the simulator to support active messages and processor-side atomic instructions like those in the Itanium2 [21]. AMOs and ActMsg share the same programming model and are initiated the same way; the only difference is the way each

Table 3 The default configuration of AMU

Parameter	Value
Clock	hub clock (default 500 MHz)
AMU TLB	1024-page coverage, 2 AMU cycles delay
ALUs	5×64b, pipelined
FPUs	4×32b, pipelined
AMO group size	≤ 5
Stream buffers	16, 64×32b each
SB allocation	2 AMU cycles
SB-to-FU delay	2 AMU cycles

is handled by the receiving node. Most overheads of active messages are accurately modeled, e.g., interrupt handling 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. 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 ignore the operating system overhead of ActMsg on the initiating processor.

Table 3 contains some of the key AMU parameters with their default values. The minimal size of DRAM access that our memory controller can issue is 32 bytes. The Active Memory Unit sits alongside the memory and directory controller, as shown in Fig. 5. Support for AMOs does not increase the no-contention latency of normal load and store requests. However, AMO memory accesses and normal accesses compete for the same directory controller pipeline, directory cache ports, and DRAM channels.

4.3 Benchmarks

Table 4 lists the ten benchmarks we use to evaluate 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.

The *barrier* benchmark uses the barrier synchronization function of the Irix OpenMP library. To evaluate *spinlock* algorithms, we consider two representative implementations, ticket locks [34] and Anderson's array-based queue locks [4]. The *GUPS* benchmark [27] 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 [33] include *memcopy, scale, sum,* and *triad*. They 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

Table 4 Benchmarks

Benchmark	Description	From	AMOs
barrier	barrier synchronization	SGI Irix OpenMP library	scalar
spinlock	ticket lock and array-based queue lock	Mellor-Crummey and Scott [34], Anderson [4]	scalar
GUPS	random global updates	HPCS Program	scalar
STREAM	memcpy, scale, sum, triad	J. McCalpin	stream
SAXPY	$Y+=a*X$, single precision FP	BLAS level 1	stream
Info_Retrieval query	unindexed relational database query	OSDB by Compaq-HP	stream
Total_Report query	unindexed relational database query	OSDB by Compaq-HP	stream
Query 2A	for document search	Set Query Benchmark	stream
Query 3A	for direct marketing and decision support	Set Query Benchmark	stream
Query 4	for direct marketing and document search	Set Query Benchmark	stream

performance, so memory performance has emerged as the new bottleneck for many large databases [2, 5, 8]. For example, Barroso et al. [5] observed less than 5% of TPC-H's execution time is spent in kernel mode performing I/O on an Oracle server. This negligible portion includes disk I/O and context switches. We investigate the potential of AMOs to accelerate queries from Open Source Database Benchmark (OSDB) [20], maintained by Compaq-HP, and Set Query Benchmark [17], by Patrick O'Neil. DB benchmarks like TPC-H are more complete, but are hard to install and evaluate in realistic simulation time [24, 52]. While these benchmarks 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 [24, 42]. The database benchmarks that we consider are considerably more complex than the queries used in those studies.

5 Simulation results

For each of the eleven 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 (Atomic), memory-side atomic operations like those in the SGI Origin 2000 (MAOs), and active memory (AMOs), where applicable. In the paper, we define *speedup of* $code_B$ *over* $code_A$ as $Execution\_time_A / Execution\_time_B$. The conventional shared memory version serves as the baseline in all the experiments.

5.1 Interprocessor synchronization

We have published a detailed study on using AMO to optimize synchronization operations [14, 56]. Here, we would like to highlight some of the key results.

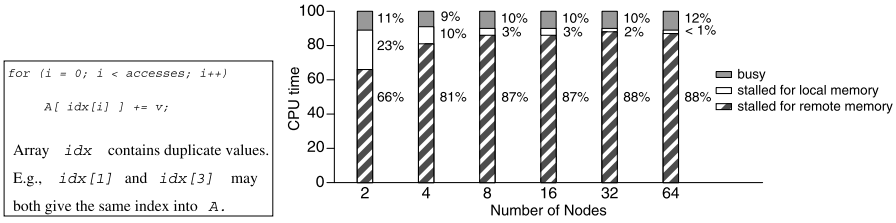
For barriers, the baseline is from the SGI OpenMP library using the LL/SC instruction pair. We hand-optimized the same algorithm for different hardware support

Table 5 Barrier performance

Nodes, CPUs	Speedup over baseline			
	Atomic	ActMsg	MAO	AMO
2, 4	1.03	0.73	1.29	**1.93**
8, 16	1.17	1.40	5.53	**12.06**
32, 64	1.19	1.62	5.46	**27.34**
128, 256	1.18	1.83	11.70	**54.82**

Table 6 Spinlocks performance, showing speedups over LL/SC-based ticket lock

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
8, 16	1.00	0.50	0.97	0.56	2.27	0.53	1.10	0.50	**2.32**	2.27
32, 64	1.00	1.66	0.87	1.69	0.67	1.47	0.67	1.51	**6.39**	5.01
128, 256	1.00	3.55	1.24	3.44	1.00	3.01	0.85	2.99	**13.58**	11.35



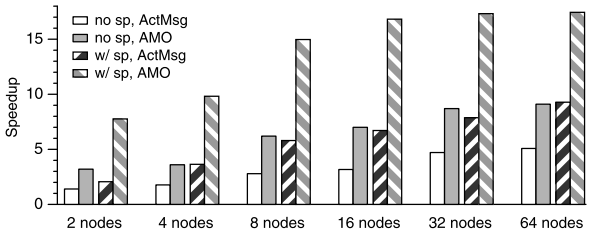
(a) code (b) Execution time breakdown

Fig. 7 GUPS code and memory performance behavior

and show their performance in Table 5. For example, processor-side atomic instruction can finish a 128-node barrier operation 18% faster than the baseline, while the AMO version is 54 times faster. For spinlocks, we implemented two popular algorithms, ticket lock [34] and array-based queue lock [4]. The results are in Table 6, e.g., AMO-based ticket lock outperforms the LL/SC ticket lock by a factor of 13× on 128 nodes. Interested readers can find more detailed information on synchronization in [56] and [14].

5.2 GUPS

Figure 7(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 [27], 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 the TLB. As the number of active processors



(a) speedup

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

(b) traffic (with sp)

Fig. 8 GUPS performance

increases and the histogram array is spread across more processors, the number of remote cache misses increases. Figure 7(b) shows memory stall time for GUPS when aggressive superpaging is employed to avoid TLB misses. With the majority of TLB misses eliminated, cache misses dominate execution time. With four processors, remote memory stalls account for 66% of execution time, which increases to 88% for 128 processors, even after we employ aggressive superpaging to eliminate all TLB misses.

Note that memory collisions due to duplicate values in `idx[]` (e.g., in histogram calculations) prevent the use of incoherent scatter-gather memory operations that are available on many vector machines. Even with coherent memory, either atomic adds or complex software techniques like segmented scan [12] are required to ensure correctness. In our baseline implementation we employ normal (nonatomic) 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 again 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 for GUPs because real applications perform nontrivial work on the data between updates, and simply disabling caching will hurt performance.

In Fig. 8(a), we report four results for every system configuration. The left two bars represent the speedups achieved by ActMsg and AMOs, respectively, compared to the baseline implementation, when neither the baseline or optimized versions employ superpaging. The right two bars represent the speedups achieved when we employ TLB misses have been eliminated by superpaging in both the baseline and optimized versions. Without superpaging, ActMsg-based GUPS outperforms the baseline version by as little as a factor of 1.4 on the two-node system and by as much as a factor of 5.1 on the 64-node system. With superpaging, the ActMsg solution outperforms the baseline version by 1.8× (2 nodes) to 8.2×. Again, the performance of the AMO implementation is dramatically better than even the optimized ActMsg version. The main reason that both mechanisms are so effective is that they eliminate substantial amounts of network traffic due to remote misses. Figure 8(b) shows the total 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 5.5×.

Table 7 Speedup of AMOs on STREAM and SAXPY with a normal system bus

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

5.3 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 multistream 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 7 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 processor cycles, respectively. In all cases, AMOs perform very well, with performance as much as 38× faster than the baseline implementation (`triad` with 200 cycle hop delays). In addition, we can see that as remote memory latencies increase, the benefits of AMOs increase.

Except for *triad*, the 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.

Another major factor that contributes to AMO's observed performance improvement is the L2 cache's write-allocate policy. On such a cache, a write miss is converted to a read request to memory (or `read exclusive` in multiprocessors). For streams that completely overwrite a cache line, as is the case for `memcopy`, `scale`, `sum` and `triad`, this extra read step is useless. AMOs do not load before writing, which saves significant memory bandwidth. Overall, we observe speedups of 1.08 to as high as 38× compared to non-AMU solutions.

The performance of stream operations on a conventional machine is sensitive to the $\frac{DRAM}{system\_bus}$ bandwidth ratio. In our default setup, DRAM and system bus bandwidths are balanced. With a more aggressive DRAM backend, even higher speedups from AMOs are obtainable (results omitted due to space limitations). In Table 8, we present results with an aggressive system bus ($\frac{DRAM}{system\_bus}$ bandwidth ratio $= 1:2$). With this model, system bus bandwidth is less of a performance bottlebeck, so we see still solid, but diminishing, benefits from AMOs. In fact, for `scale` and `saxpy`

Table 8 Speedup of AMOs on STREAM and SAXPY with an aggressive system bus

nodes, procs	memcopy		scale		sum		triad		saxpy	
	h100	h200	h100	h200	h100	h200	h100	h200	h100	h200
1, 1	1.19	1.19	1.06	1.06	1.26	1.26	2.88	2.88	1.17	1.17
1, 2	1.07	1.07	*0.91*	*0.91*	1.01	1.01	2.47	2.47	*0.94*	*0.94*
4, 8	1.13	1.17	1.17	1.20	1.65	1.77	3.09	3.88	1.09	1.22
8, 64	1.28	1.43	1.88	2.19	2.21	3.98	8.10	13.24	1.75	2.33
128, 256	1.39	1.98	2.18	3.44	3.97	5.13	11.22	20.08	2.06	3.89

AMOs exhibit slight slowdowns on 2-processor systems because of the performance gap between the processor FPU and the AMU FPU. If we increase the clock of the AMU from the conservative 1/4 processor frequency to 1/2 processor frequency, we will see positive speedups for both `scale` and `saxpy`. Nevertheless, with an aggressive system bus, we still observe good performance with AMOs in most cases in Table 8.

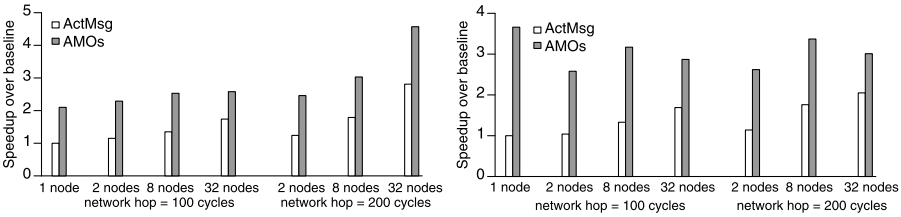
Our results in Table 8 concur with the insight into *realizable* memory bandwidth by Batten et al. [6]. That is, it is the control and buffering overhead in the processor (reorder buffer entries, physical registers, ld-st queue entries, outstanding cache miss trackers and buffering cost in caches and in the interconnect, etc.), rather than the raw memory bandwidth, that limits memory system performance. Indeed, even given a very aggressive system bus, the conventional processor is unable to fully utilize the high bandwidth offered by the DRAM backend.

5.4 Database queries

In this section, we present the performance derived by applying AMOs to a variety of database microbenchmarks from OSDB [20] and Set Query benchmarks [17]. The *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). We use a 307 MB database for OSDB. The Set Query benchmarks perform a number of condition tests on each data record. We use a 204 MB database for Set Query. These queries are representative of a variety of document search, marketing, and decision support workloads.

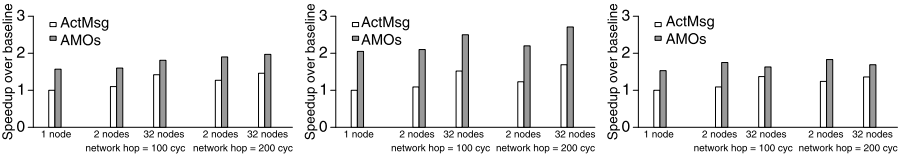
Almost all commercial relational DBMSs store and access data row-by-row *horizontally*. The AMO versions access database tables by attributes, i.e., column-wise. Each field in the database table is accessed *vertically* as a stream across all tuples.

Figures 9 and 10 present the speedups of the ActMsg- and AMO-optimized database engines compared to conventional implementations. Both processors on each node perform queries. 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 multinode settings, the L1 cache hit ratio for *Info_Retrieval* and *Total_Report* are 99.5% and 98.3%, respectively. For multinode experiments, we employ the "50% remote" distribution and hop latency variations described in Sect. 5.3.



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

Fig. 9 Optimizing the OSDB benchmark



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

Fig. 10 Optimizing the set query benchmark

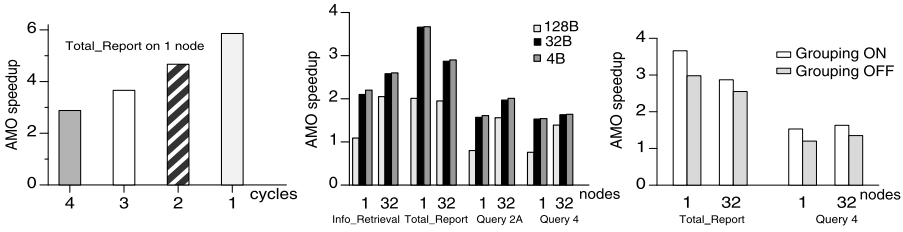
As can be seen in the figures, AMOs speed up the various database queries by factors ranging from 2.1× to 4.4× compared to their respective baselines. The ActMsg variants perform poorly at small node counts, but achieve 50–75% of the benefits of AMOs for larger configurations.

The database queries we consider do not employ any index structure, but rather perform sequential scans of the tables. However, bitmap indexing, a data access method widely used in document search and decision support systems, would actually be an ideal application of AMOs (discussed in Sect. 4.5 of [13]). We do not quantify the impact of using bitmap indexing in this study, because with a meaningful length for the index bit stream, the corresponding database would be too large for our simulation to complete in a realistic amount of time.

5.5 Quantifying the sources of AMO performance advantage

The benchmarks benefit from different aspects of our AMO design. Since space is limited, 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: (1) localizing computation by performing the work at the data's home node, (2) utilizing specialized hardware for specific operations such as `max` or `min`, (3) accessing sparse data using short DRAM bursts as opposed to full cache line bursts, and (4) exploiting stream-level parallelism in grouped 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 to each other; each one is only profitable if it alleviates a system bottleneck for a particular application.



(a) *ALU latency. Default = 3 cyc.* (b) *DRAM access granularity. Default = 32B* (c) *AMO grouping. Default = ON*

Fig. 11 Quantifying the performance contribution of various aspects of AMOs

Computation Localization: By shipping the computation to data's home node, remote memory loads are converted to local loads. Thus, network bandwidth and latency limitations are avoided for both active messages (ActMsg) and AMOs. In contrast, 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, the ActMsg result in Figs. 9 and 10, represents the computation localization benefit. As can be seen in the figures, the benefit of computation localization to AMO performance is modest for small system configurations, but substantial for larger ones.

Specialized Hardware: AMUs have specialized hardware designed for efficient stream processing, as opposed to a general-purpose processor. For example, to determine the `max` of two integers on a conventional CPU requires several load, store, comparison, and branch instructions, whereas the AMU can compute the max in only one cycle after the operands are available. Among the database queries presented in this paper, the *Total_Report* query from the OSDB suite benefits the most from specialized function units.

Figure 11(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 3.7× with the default 3-cycle latency 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. Exploiting short DRAM bursts is particularly helpful in the database benchmarks. For stream operations, the AMU only loads the referenced fields (e.g., Balance in Fig. 3) and not the entire cache line. In addition, the query optimizer performs predicate evaluation in order of selectivity, using masked operations to avoid subsequent computing on tuples that fail to meet a particular SELECT test. 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 (Fig. 10(c)), the first predicate evaluation filters out 80%

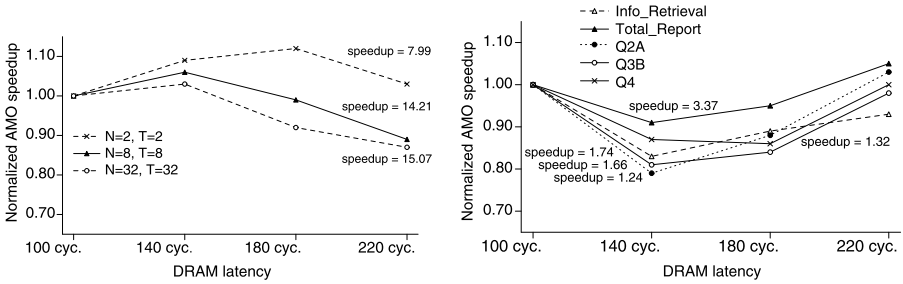
of the tuples. Thus, AMUs have ample opportunity to exploit short DRAM bursts for stream operations.

In our design, AMUs load data from DRAM in 32 B bursts, whereas the baseline system loads 128 B cache lines. DRAM vendors (e.g., Rambus) have started to enable even shorter DRAM burst lengths, which is irrelevant when loading entire cache lines but will further improve AMO performance. Figure 11(b) shows the performance of four queries as we vary DRAM burst lengths. These experiments confirm that loading data at smaller granularity (32 B) 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. However, a number of architecture trends may motivate the use of shorter burst lengths. Future memory controllers are likely to be integrated on the processor die, in which case pin limitations will constrain the number of memory channels. In addition, the current trend in processor designs is to incorporate an increasing number of processors per die, which will increase DRAM pressure. For both of these reasons, the ability to exploit DRAM burst lengths shorter than 32 bytes will likely be useful in the near future. AMOs provide one means to utilize this opportunity.

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 temporary streams allows results from one stream operation to be bypassed directly to 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 Fig. 11(c) show the performance of these queries when we ran them using serial (nongrouped) AMOs. Doing so, reduced performance by 10–20%.

5.6 Sensitivity to DRAM latency

The effect of increased DRAM access latency on the relative speedup of AMOs compared to conventional systems is not self-evident. Increased DRAM latency affects the speedup of AMOs in two conflicting ways. On the one hand, higher DRAM latency adds a constant factor to each memory access in the execution of both the baseline and AMO versions of each benchmark, which reduces AMOs' speedup over the baseline. On the other hand, higher DRAM latency reduces the actual memory bandwidth. Because of constraints posed by the various control and buffering structures in the microprocessor, slower DRAM chips can hurt effective memory bandwidth more for processor-based computation than for memory controller-based computation. This phenomenon tends to increase the AMO speedup over the baseline. The



(a) *GUPS* (b) *database queries*

Fig. 12 Varying DRAM latency

overall influence of prolonged DRAM latency on AMOs' speedup is an artifact of both effects.

To determine the sensitivity of AMO performance to increases in DRAM latency (as measured in CPU clock cycles), we repeated our earlier experiments with modeled DRAM latencies of 140, 180, and 220 cycles, versus our default 100-cycle latency. We then normalize the speedup achieved by using AMOs with different DRAM latencies against the speedup gained by using AMOs on a 100-cycle-latency DRAM system.

Figure 12(a) presents the sensitivity tests for the GUPS benchmark. As we increase DRAM latency, all three configurations show nonmonotonic curves. When DRAM latency increases from 100 to 140 cycles, the relative benefit of using AMOs grows due to AMOs' ability to deliver higher effective memory bandwidth. As we continue to increase the DRAM latency, we start to see reduced speedups (normalized values < 1.0), because the common DRAM performance component in both the AMO and non-AMO versions of GUPS increasingly compromises the advantages of AMOs. Sensitivity tests on STREAM benchmarks show similar curves (figures not included here).

Figure 12(b) presents the AMO speedups of database queries on a single node normalized against each query's respective AMO speedup at 100-cycle DRAM latency. The database benchmarks demonstrate quite different trends than those of GUPS and STREAM. While most of the latter see shrinking benefits from AMOs when the DRAM latency approaches 220 cycles, the benefit for the database queries increases in that region. This phenomenon is caused by the use of masks by the AMOs. Within each page of data, masks filter out most strides of subsequent AMOs so that data streams get sparse. Consequently, a slower DRAM system negatively affects AMO performance less than it does non-AMO performance, leading to better AMO speedups in the $>$220-cycle region.

6 Analytical modeling

To determine if the benefits justify the effort needed to port an application to AMOs, a programmer needs a method to obtain an early estimate of the potential performance benefits. Similarly, for a compiler to automate the program transformation needed to

employ AMOs, it needs some cost function to estimate the performance of various possible transformations. An analytical model can serve both purposes. There are a number of analytical models [3, 32, 47, 50] for general-purpose computers. However, the performance characteristics of the baseline non-AMO code provide little help in predicting the performance of its AMO-optimized counterpart, since the two solutions usually access and process data in rather different ways and are constrained by different system parameters.

The programming effort for using scalar AMOs is trivial, obviating the need to predict AMO performance. We will focus on modeling stream AMOs. The input to the model include an algorithmic description for using AMOs, plus microarchitecture parameters of the underlying hardware. Our task is to build a predictive model that can aid the programmer to decide whether to port a particular legacy code to use stream AMOs. This is a daunting task: we do not even have the AMO source code whose potential performance we wish to predict. All we have is an AMO algorithm and some key application parameters like the memory range, stride, number of streams, data element types, and ALU operations that need to be performed. As a result, our analytical model only attempts to predict the *mean* performance of AMO codes under certain conditions.

When a conventional program executes, the DSM hardware loads remote memory blocks across the network, whereas AMOs ships operations rather than large amount of data. However, "heroic" programming efforts including thread-conscious data distribution or data-conscious thread scheduling can mitigate or hide the cost of remote memory accesses. We assume the baseline code is already optimized to operate on data on local memory whenever possible, and thus do not estimate the potential advantage of reduced network traffic when using AMOs.

6.1 Overview of the model

The approach that we take is a throughput-based method, similar in spirit to the Flow Equivalent Server (FES) model [11]. The main points of this method are recursive decomposition and aggregation. First, the network of queues under investigation is iteratively partitioned into smaller subnetworks, until each partition can be trivially modeled. In the aggregation step, each subnetwork is replaced with a single load-dependent queue, which is *flow-equivalent* to the subnetwork. Execution of AMOs in an AMU can be abstracted as a simplified form of the FES model. A subnetwork in our case is one arithmetic/logic function unit or one memory channel. The aggregation step involves adding up the flows seen in each subnetwork.

Before we map AMOs to the FES model, we introduce the concept of "child" AMOs. We call the programmer-issued AMOs (page grain) "father" AMOs. Within the stream unit of an AMU, at each stride a father AMO spawns one memory block-grained suboperation. These suboperations are "child" AMOs. For example, a stream AMO over a 16 KB page will spawn a total of $16\,K/256 = 64$ child AMOs if stride $=$ 256 bytes.

Stream AMOs have a very simple flow of control. The memory, the stream buffers, and the memory function unit (MFU, the dashed box in Fig. 5(b)) compose a smooth,

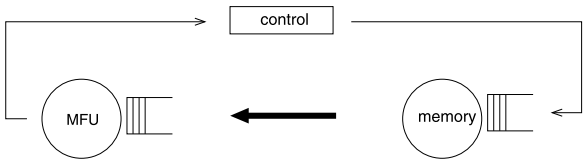


Fig. 13 Flow of stream AMOs

three-step pipeline. Except for start-up latency at the DRAM backend and trailer delay at the MFU, child AMOs go through the three-step pipeline in a streaming manner: memory access ⟶ stream buffer ⟶ MFU. As shown in Fig. 13, this is a closed loop; the AMU Control Unit sends DRAM requests as soon as slots become available in the stream buffers because the data they contained have been consumed by the MFU. Stream buffers are just a staging area with respect to timing, so they reduce to a delay center and do not appear in Fig. 13. Depending on which has a lower throughput, either MFU or memory will be the performance bottleneck throughout a given AMO's execution. The start-up latency at the DRAM backend and the trailer delay at the MFU are delay centers.

Each AMO (or AMO group) is immediately followed by a busy spin on its return value location. Thus, we can simply calculate the execution time of each individual AMO/group and concatenate them.

The analytical model consists of two steps. We first calculate the maximum throughput of the MFU and memory, and predict which one of the two is the bottleneck that throttles the flow. Then, we use the predicted performance bottleneck to estimate the absolute execution time. We do so by dividing the total number of memory blocks with the bottleneck throughput which is measured in blocks per unit time. Details of the model can be found in [13].

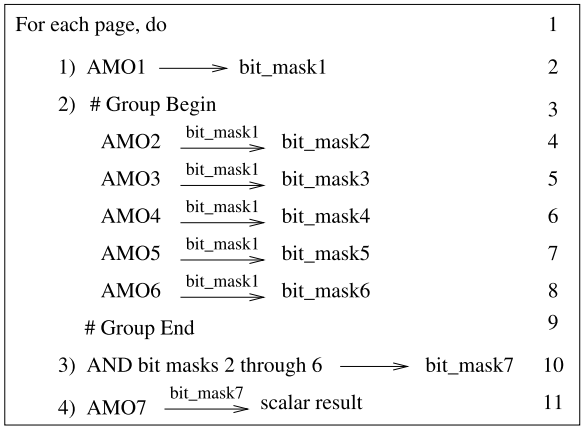
6.2 Identifying the bottleneck

Similar to IPC (Instructions Per Cycle), which measures the instruction throughput of a given piece of code, we employ an analogous metric, Blocks Per Cycle (BPC). BPC is the number of *logical* memory blocks that an AMU can finish in one clock period. A logical memory block includes all of the memory blocks that a group of stream AMOs need to read and write at a stride. The following rules clarify the concept of logical blocks.

- If N AMOs are grouped to proceed in parallel, and each of them reads/writes S_i ($i = 1, 2, \ldots, N$) streams to/from memory, then a logical block consists of $\sum_{i=1}^{N} S_i$ physical blocks. In Fig. 14, lines 4 through 8 are in a group. A logic block for this group consists of the physical blocks read/written by AMOs 2 through 6.
- If there is producer-consumer dependency between two grouped AMOs, the logical memory block consists of the blocks of both AMOs.
- A min-grain access (when stride > system block size, explained in Sect. 3.3.2) counts as one physical block.
- Temporary streams (see Sect. 3.2.3) are not counted.

First, we calculate the effective logical blocks per cycle that the MFU can process, BPC_{mfu}. Then we estimate the effective logical blocks per cycle that the DRAM

Fig. 14 An AMO algorithm for *Info_Retrieval*



backend can support, BPC_{memory}. Both *BPCs* reflect the *maximum* throughput that we can get for a given AMO group. The smaller of the two *BPCs* points to the system bottleneck.

6.2.1 BPC_{mfu}

BPC_{mfu} denotes the maximum throughput of the arithmetic and logic function units (FUs) in the AMU for a particular stream AMO or a group of AMOs. It depends on the number of FUs, the latency of the first arithmetic/logic operation on a loaded block, and how many computations need to be performed per loaded block.

When the memory blocks associated with a child AMO arrive from memory, the data operands are stored in the stream buffers. When the first element in a stream is present, i.e., entries at the head of the input stream buffers have their F/E bits set, they are fetched into the input latches of the FUs. The following details help to drive the equation for BPC_{mfu}:

- If several AMOs in a group share a single memory stream as one of their inputs, these AMOs obtain their operands serially, since there is only one read port per stream buffer in our design.
- When there is a producer-consumer dependence between two AMOs in a group, the operations are pipelined if both AMOs use different FUs. The AMO with the lower throughput determines the overall throughput. If both AMOs compete for a single FU, the two operations need to be serialized.
- If independent AMOs are grouped (to exploit parallelism), and they use different FUs, these AMOs proceed in parallel. However, since they need to synchronize at the end of the page as group members, the AMO with the smallest throughput determines the throughput of the group.

Expressing the above rules in a mathematical form, the BPC_{mfu} of N grouped AMOs (either chained or parallel) is

$$BPC_{\text{mfu}} = MIN_{i=1}^{N}\left(\frac{number\_of\_FUs \times MFU\_speed\_factor}{FU\_latency_i + (ops\_per\_blk_i - 1) + (number\_of\_sharers_i - 1)}\right) \quad (1)$$

in which

$$ops\_per\_blk = \frac{cache\ line\ size}{MIN(cache\ line\ size,\ stride)} \times computations\ per\ stride$$

Suppose there are *number_of_FUs* function units available to perform the operation type of a given stream AMO. The arrival of the operands for this AMO drives one of these function units. The *ops_per_blk* computations on the loaded block are performed in a pipelined fashion by this function unit. Only *ops_per_blk - 1* additional cycles are exposed per block after the operation on the first element completes. Thus *FU_latency + (ops_per_blk -1) + (number_of_sharers - 1)* is the number of MFU cycles needed to process one logical block of the AMO group. *MFU_speed_factor* is the ratio of the ALU clock frequency compared with processor clock frequency.

According to (1), N AMOs that proceed in parallel and N AMOs that are dependent on each other have the same BPC_{mfu}. This result is somewhat counterintuitive. The explanation lies in our definition of logical memory blocks. Suppose each of the AMOs uses one input stream and generates one output stream. In the case of dependent AMOs, the output stream of one AMO serves as the input of another AMO. We use a temporary stream to pass values between them, which does not count toward our definition of logic blocks. Then one logical block of the N chained AMOs consists of only two real memory blocks, while one logical block of the N parallel AMOs is composed of $2 \times N$ real blocks. Therefore, although both cases have the same BPC_{mfu}, they need different DRAM bandwidths to sustain this MFU throughput. In other words, they need different BPC_{memory} to feed the MFU, as is explained in the next section.

6.2.2 BPC_{memory}

The difficulty of analytically predicting memory access time rises from the complex interplay between the many memory system parameters and the memory requests. What makes the task particularly hard is that DRAM latency is not only load-dependent, but also heavily address-dependent. DRAM addresses are not known until runtime. Another uncertainty is value-dependency, which prevents us from getting outcome of branches and number of loop iterations at compile time.

A simplistic estimate of memory throughput We start with an optimistic estimate of BPC_{memory} that does not take into account many of the complexities involved in

a DRAM access. Then, using empirical coefficients, we factor in some additional overheads and refine the initial BPC_{memory} estimate.

Modern DRAM cell arrays are organized into a large number of banks to exploit internal DRAM array access parallelism, while the raw latency of each individual DRAM structure has not improved much. On the other hand, pin limitations will grow as memory controllers are integrated on to the processor die. Thus, the number of DRAM channels between the memory controller and DRAM chips will be severely limited. Large numbers of internal DRAM banks can reduce bank conflicts rather effectively in modern DRAMs, so address channels and data channels are the primary bottlenecks for sustained memory bandwidth.

The per-channel BPC_{memory} for one AMO group can be expressed as

$$\frac{directory\_overhead}{cycles\_per\_access \times number\_of\_streams \times DRAM\_speed\_factor} \tag{2}$$

Directory-overhead represents the percentage of DRAM accesses that are for real data, as opposed to directory loads and directory cache writebacks. In our target system, every evicted directory cache (dc) line is written back to DRAM regardless of whether it is dirty or not. Therefore,

$$directory\_overhead = \frac{1}{1 + dc\_miss\_rate \times 2} \tag{3}$$

We can predict *dc_miss_rate* using the stack-distance model [10] driven by the directory cache line size, the number of bytes needed to encode the sharing information of a memory block, and other application characteristics as inputs.

Cycles-per-access is the average nonoverlapping memory cycles needed to deliver a block of data. A "block" is either min-grain bytes for large strides or system block size for small strides. It reflects the shortest possible interval between two memory blocks returning from a same DRAM channel. The value of *Cycles-per-access* is primarily determined by the DRAM chip specifications and memory channel width. *DRAM-speed-factor* reflects the difference between clock frequency of the DRAM core and that of the memory controller. *Number_of_streams* is the number of streams that compose a logical block (defined at the start of Sect. 6.2) derived from the provided AMO algorithm.

When T threads in a node run AMO codes simultaneously, the T active AMO groups share a single AMU and a single set of memory channels, which effectively reduces the memory bandwidth to serve each thread to $1/T$ of the total capacity. This situation is different than the BPC_{mfu} case, where different AMO groups executed on the same AMU use different function units and thus do not interfere with each other.

Different memory channels work independently. When T AMO groups compete for *channels* DRAM channels, a simple formula for memory throughput is

$$BPC_{\text{memory}} = \frac{directory\_overhead \times channels \times DRAM\_speed\_factor}{T \times cycles\_per\_access \times number\_of\_streams} \tag{4}$$

In our architecture of two processors per node, T is equal to either 1 or 2.

Refining BPC_{memory} The simplistic BPC_{memory} derived above ignores at least three factors that can limit memory throughput: (i) DRAM bank conflicts, (ii) the order that directory information and data values become available, and (iii) in-order drain of data elements from, and out-of-order arrival of data elements at, the AMU stream buffers.

DRAM bank conflicts happen when two accesses to different DRAM rows hit the same bank. The two accesses need to be serviced serially, contradicting our earlier assumption of bank parallelism.

To handle a coherent data access, the home node must consult the corresponding directory entry to determine the data's sharing state. If the directory entry is not present in the directory cache (dc), then a read is issued for the directory entry in parallel with a speculative read of the data, both from local memory. The (speculative) data access and dc refill usually do not complete on the same cycle. The coherent access is not over until the slower of the two has completed, which limits the effective throughput of the memory system. Normally, we expect to receive the directory information before the data because of spatial locality in the directory accesses. However, if a directory load falls to a piece of busy DRAM module, its data counterpart can finish first. The earlier the data arrive, the more time is wasted. On the other hand, a dc refill that completes too early is also undesirable, since it often comes at the cost of delaying critical-path data accesses or directory access of another transaction. In general, the more heavily loaded the DRAM backend is, the more performance loss we suffer from this effect. For example, in `sum` operation of the STREAM benchmark, about 11% of speculative data loads complete before their directory information is available. The average interval between when the data are available and when the directory information is available is about 5 memory cycles for our setup. The harm caused by early dc refill returns is harder to quantify because of the coalescing effect of the MSHRs in dc lines.

The third factor that limits delivered memory throughput is similar to the second. Remember that we drain each stream buffer in-order (Sect. 3.3.2), but stream data elements arrive from memory out-of-order. If a data element near the end of a stream returns before an element early in the stream, it cannot be consumed until all preceding elements are consumed.

Expressing these effects analytically is challenging because they depend on the sequence of addresses generated by the stream unit. Hence we introduce an **empirical** coefficient into our formula for BPC_{memory}. This coefficient, η, is a function of the DRAM configuration and the application's memory pressure. That is, $\eta = f\,(BPC_{\text{mfu}}, \textit{memory configuration}, \textit{stride}, \textit{threads})$.

Through extensive experiments using synthetic codes, we find that in practice, on our specific platform the following formula is able to approximate the influence of bank conflicts and DRAM return orders on performance:

$$\eta = \frac{1}{[1 + f_1(BPC_{\text{mfu}}) + f_2(\textit{channels}) + f_3(\textit{banks}) + f_4(\textit{stride}) + f_5(\textit{threads})]} \tag{5}$$

in which

$$f_1(BPC_{\text{mfu}}) = BPC_{\text{mfu}} \times \textit{number\_of\_streams} \tag{6a}$$

$$f_2(channels) = 1.0/channels \tag{6b}$$

$$f_3(banks) = 3.0/banks \tag{6c}$$

$$f_4(stride) = \begin{matrix} 0.00, & if\ stride \geq cache\ line\ size \\ 0.05, & if\ stride < cache\ line\ size \end{matrix} \tag{6d}$$

$$f_5(threads) = \begin{matrix} 0.00, & if\ threads = 1 \\ 0.10, & if\ threads = 2 \end{matrix} \tag{6e}$$

Intuitively, greater numbers of memory channels and internal banks tend to pull η closer to 1.0, while higher memory pressure (due to larger MFU throughput, more concurrent threads, larger DRAM burst sizes, etc.) causes η to decrease. The empirical formula for η must be tuned on the target platform before it is used.

Using the coefficient η to refine (2), we have

$$BPC_{\text{memory}} = \frac{\eta \times directory\_overhead \times channels}{T \times cycles\_per\_access \times number\_of\_streams \times DRAM\_speed\_factor} \tag{7}$$

This equation gives the per-thread BPC_{memory} when T simultaneous AMO threads compete for *channels* DRAM channels.

Using (1) and (7), we can predict the performance bottleneck of a stream AMO.

6.2.3 Adjusting BPC_{mfu} and BPC_{memory} for sparse streams

BPC_{mfu} and BPC_{memory}, as we have defined so far, represent the maximum throughput that the MFU or the memory system can provide for a given AMO group. We can utilize a maximum throughput only if the control unit in Fig. 13 generates enough memory requests to saturate the component that is under investigation. In some cases, the MFU or memory may consume/supply data at a rate lower than its capacity. This typically happens when a high-selectivity bit mask is used, because most of the strides have been filtered out. We refine our equations for BPC_{mfu} and BPC_{memory} in this section.

Let P_{unmask} denote the percentage of ALU operations or memory strides that are not masked off in a given stream AMO. A higher value for P_{unmask} corresponds to a denser stream. Note that P_{unmask} can take different values for the MFU and memory, because for the AMU to skip loading a DRAM block, all of the operations that correspond to stream elements residing in the block need to be masked. Therefore, P_{unmask} for memory is often greater than P_{unmask} for the MFU. They are equal if and only if the stride is equal to the DRAM access granularity.

Adjusting $\mathbf{BPC_{mfu}}$ for sparse streams

FU_latency + (ops_per_blk -1) is the number of cycles that it takes to process one memory block. After this period of time, the function unit can be reused to process another block. When a mask is sparse, only up to *[FU_latency + (ops_per_blk -1)]* $\times$ P_{unmask} FUs need to be active at a time to service the mask-filtered blocks for a selectivity of P_{unmask}. We call this number *effective_FUs*. That is,

$$effective\_FUs = [FU\_latency + (ops\_per\_blk - 1)] \times P_{\text{unmask}} \tag{8}$$

Effective_FUs represents the number of FUs that are actually needed. If *number_of_FUs* > *effective_FUs*, some of the FUs will be idle and do not contribute to the execution. We substitute *effective_FUs* for *number_of_FUs* in (1). If P_{unmask} is extremely small, *effective_FUs* can be less than 1.0. This indicates that even one FU is more than enough for the task, and that we should base our estimate of execution time on BPC_{memory}.

Adjusting $\mathbf{BPC}_{\text{memory}}$ for sparse streams

Cycles_per_access is the number of memory cycles that a DRAM transaction occupies a channel before this channel can be reused for another transaction. To service K outstanding memory transactions during the *cycles_per_access* period, $K \times cycles\_per\_access/(directory\_overhead \times \eta)$ channels are sufficient.

For $K = number\_of\_streams \times T \times P_{\text{unmask}}$, the number of channels that we need is

$$effective\_channels = \frac{number\_of\_streams \times T \times P_{\text{unmask}} \times cycles\_per\_access}{directory\_overhead \times \eta} \quad (9)$$

η in this equation is calculated using *number_of_FUs* = 1 if *effective_FUs* < 1.0.

If $1.0 \leq effective\_channels < channels$, we use *effective_channels* for *channels* to calculate BPC_{memory} (7), because more channels than *effective_channels* are unnecessary. The calculation of η is always based on the actual number of DRAM *channels* present in the system.

If *effective_channels* < 1.0, the memory channels downgrade from a queuing center to a delay center, thus we can estimate the execution time as:

$$logical\ blocks\ per\ page \times P_{\text{unmask}} \times DRAM\ latency \quad (10)$$

6.3 Estimating total execution time

If $BPC_{\text{mfu}} > BPC_{\text{memory}}$, memory throughput determines stream AMO performance, otherwise MFU throughput does. After using (1) and (7) to identify whether MFU or memory throughput is the bottleneck for a given AMO stream operation, we estimate the absolute execution time for an AMO (operating on a single page) using the following simple calculation:

$$per\text{-}page\ time = \sum_{serial\ groups} \left(DRAM\ latency + \frac{logical\ blocks\ per\ page}{BPC_{\text{mfu/memory}}} \right) \quad (11)$$

For a thread that iterates over a large range,

$$Execution\ time = pages \times per\_iteration\ time \quad (12)$$

where *per_iteration time* is typically a sum of *per-page time* calculated with every AMO group within the loop body, plus non-AMO code between the AMO groups. For example, the *per_iteration time* of the algorithm in Fig. 14 is a sum of execution time of the four steps.

To summarize the analytical model, we go through the following steps to estimate the total execution time of T stream AMO threads simultaneously running on a node.

(1) Calculate *effective_FUs* and *effective_channels* (8) and (9).
  (1.1) If *effective_FUs* < 1.0, go to step 1.3).
  (1.2) If 1.0 ≤ *effective_FUs* < *channels*, set *number_of _FUs* = *effective_FUs*.
  (1.3) If *effective_channels* < 1.0, use (10) for the execution time of the AMO/group. Go to step 3).
  (1.4) If 1.0 ≤ *effective_channels* < *channels*, set *channels* = *effective_channels*.
(2) Calculate BPC_{mfu} and BPC_{memory} using (1), (7), and (9).
  $BPC = MIN(BPC_{\text{mfu}}, BPC_{\text{memory}})$.
(3) Calculate per-page execution time using (11).
(4) Calculate total execution time using (12)

This description of our analytical performance prediction method includes the adjustments to BPC_{mfu} and BPC_{memory} calculation in Sect. 6.2.3 as the first step. In practice, a typical AMO algorithm uses several serialized groups to accomplish certain selections. The very first group often has a P_{unmask} equal to 100%, and generates a (high-selectivity) mask to be used by subsequent groups for the same page. These AMO groups work on the strides that are not masked off by the mask. In practice, execution time of each page is often dominated by the first group because of the filtering effect of the mask.

6.4 Using the analytical model: an example

As an illustration of how to use the analytical model that we developed, this section goes through an OSDB query, *Info_Retrieval*. The AMO algorithm that we have designed consists of three AMO groups for each page of data. Steps 1) and 4) are two groups with only one AMO each, shown in the pseudocode in Fig. 14.

We assume the system configuration presented in Tables 2 and 3. For this application, AMOs 2, 3, and 4 have a computation latency of 2 AMU cycles, and AMOs 5 and 6 have a latency of three cycles. Each of the database tuples is padded to 128 bytes. We have chosen AMO1 to be performed first because it results in a mask with high selectivity: *bit_mask1* has 5% of its bits set, so only 5% of the memory blocks in AMOs 2 through 6 need to be loaded. In the following calculations, we note the relevant equation(s) in parentheses.

Step 1) AMO1:

$P_{\text{unmask}} = 100\%$ for both the MFU and memory.
effective_FUs = 3.0 (8)
$BPC_{\text{mfu}} = 0.2500$ (1)
$\eta = 0.5904$ (5)
effective_channels = 4.0 (9)
$BPC_{\text{memory}} = 0.0472$ (7)

Because $BPC_{\text{memory}} < BPC_{\text{mfu}}$, memory bandwidth is the bottleneck. Apply BPC_{memory} to (11) and we get

$TIME_{AMO1} = 2910$ *cycles per page*

Step 2) AMO2 through AMO6:

Using a $P_{\text{unmask}} = 5\%$ for both the MFU and memory, we have

effective_FUs = 0.15 (8)
$\eta = 0.5375$ (5)

Table 9 Analytically predicted cycles versus simulated cycles

Benchmark	Predicted	Simulated	Error
STREAM memcopy	565.1 K	606.2 K	−7%
STREAM scale	585.7 K	638.5 K	−8%
STREAM sum	792.8 K	941.1 K	−16%
STREAM triad	816.6 K	935.4 K	−13%
SAXPY	856.5 K	946.6 K	−10%
OSDB Info_Retrieval	28.1 M	24.5 M	+15%
OSDB Total_Report	11.4 M	14.1 M	−19%
Set Query 2A	38.8 M	41.5 M	−7%
Set Query 3A	30.2 M	31.7 M	−5%
Set Query 4	20.9 M	24.1 M	−13%

$effective\_channels = 4.0$ (9)
$BPC_{\text{memory}} = 0.0086$ (7)
$effective\_FUs < 1.0$ and $effective\_channels > 1.0 \Rightarrow$
$TIME_{\text{AMOs 2 to 6}} = 944$ *cycles per page* (11)

Step 3) the bit operations:
Insert a 400-cycle delay for this non-AMO code block. This can be derived from existing analytical models [3, 48].
Step 4), AMO7:
The high-selectivity predicates have masked off almost all of the data elements. Therefore, the cost of this AMO reduces to a processor-DRAM roundtrip. We simply insert a 250-cycle delay.
The aggregate step:
In our experiment, two threads run on each node. Each of them processes half of a 196 MB database.

$pages = 6250$
Total execution time
$= pages \times (TIME_{\text{AMO1}} + TIME_{\text{AMOs 2 to 6}} + TIME_{\text{bit ops}} + TIME_{\text{AMO7}})$
$= 28.1$ *million cycles* (12)

Our execution-driven simulation experiment determined that the execution time of this sample code is 24.5 million processor cycles, so the error of our analytical model is about +15% in this example.

6.5 Validation of the analytical model

To validate the analytical model we derived, we apply the model to the stream benchmarks in Table 4. Table 9 presents results of these tests. Error is defined as *(predicted cycles − simulated cycles)/simulated cycles*.

For all of the benchmarks we consider, the execution time predicted by our analytical model is within ± 20% of the execution time observed through simulation. Given the complexities in interactions between the dynamic memory references and the DRAM backend organization, it is difficult to isolate sources for the disagreement

between predicted and simulated results in Table 9. Here are some example sources of the observed errors:

- DRAM page open time: dependent on address sequence at runtime
- DRAM bank conflicts: dependent on address sequence at runtime
- distribution of values for a certain field: significantly impact the database query time
- correlation between different table fields: significantly impact the database query time

Given the large speedups of AMOs that we observed, the analytical model can tolerate a fair degree of inaccuracy. For example, although our analytical model overestimates the efficiency of AMOs on *Total_Report* by 19%, the simulated speedup is 360% (Fig. 9(b)), dwarfing the analytical error. Thus, we claim that our analytical model allows the programmer to make good qualitative decisions on code porting effort.

7 Conclusions and future work

In this paper, we propose a mechanism called *Active Memory Operations* (AMOs) that allows programmers to ship select computation to the home memory controller of the data on which the computations operate. 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 50× faster barriers, 12× faster spinlocks, 8.5×–15× faster stream operations, and 3× faster database queries.

These results motivate us to continue this line of research. For example, we plan to determine the power benefits of using AMOs. Since AMOs bypass processor caches, minimize network traffic, and greatly reduce program execution time, we expect significant power savings. AMU-enabled systems are also good candidates for using adaptive frequency and voltage scaling techniques.

One limitation of the 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 implementing the AMU using one or more simple in-order processor cores. As transistor density increases, a number of these cores can easily fit on an external memory controller. Such a design would significantly complicate the programming model, but provide richer opportunities for a variety of applications.

Acknowledgements The authors would like to thank Silicon Graphics Inc. (SGI) and the National Security Agency (NSA) for their support. We are thankful for Randy Passint, Donglai Dai, Steve Miller, and Marty Deneroff for their valuable technical feedback. We are also indebted to Dr. Juliana Freire for her help on the database experiments.

References

1. Ahn JH, Erez M, Dally WJ (2005) Scatter-add in data parallel architectures. In: Proceedings of the eleventh annual symposium on high performance computer architecture, Feb 2005, pp 132–142
2. Ailamaki A, DeWitt D, Hill M, Wood DA (1999) DBMSs on a modern processor: where does time go. In: Proceedings of the 25th VLDB conference, Edinburgh, Scotland, Sept 1999, pp 266–277
3. Albonesi DH, Koren I (1995) An analytical model of high performance superscalar-based multiprocessors. In: Proceedings of the 1995 international conference on parallel architectures and compilation techniques, Sept 1995, pp 194–203
4. Anderson T (1990) The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 1(1):6–16
5. Barroso LA, Gharachorloo K, Bugnion E (1998) Memory system characterization of commercial workloads. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, Spain, pp 3–14
6. Batten C, Krashinsky R, Gerding S, Asanovic K (2004) Cache refill/access decoupling for vector machines. In: Proceedings of IEEE/ACM 37th international symposium on microarchitecture, Dec 2004, pp 331–342
7. Blelloch G, Gibbons P, Vardhan S (2008) Combinable memory-block transactions. In: Proceedings of the 20th international symposium on parallel algorithms and architectures, June 2008, pp 23–34
8. Boncz PA, Manegold S, Kersten ML (1999) Database architecture optimized for the new bottleneck: memory access. In: Proceedings of the 25th VLDB conference, Edinburgh, Scotland, Sept 1999, pp 54–65
9. Brockman JB, Kogge PM, Sterling TL, Freeh VW, Kuntz SK (1999) Microservers: a new memory semantics for massively parallel computing. In: Proceedings of the 1999 international conference on supercomputing, June 1999, pp 454–463
10. Cascaval C, Rose LD, Padua DA, Reed DA (1999) Compile-time based performance prediction. In: Proceedings of the 12th international workshop on languages and compilers for parallel computing, pp 365–379
11. Chandy KM, Herzog U, Woo LS (1975) Approximate analysis of general queuing networks. IBM J Res Dev 19(1):43–49
12. Chatterjee S, Blelloch G, Zagha M (1990) Scan primitives for vector computers. In: Proceedings of supercomputing '90, June 1990, pp 666–675
13. Fang Z (2006) Active memory operations. PhD thesis, University of Utah
14. Fang Z, Zhang L, Cheng L, Carter J, Parker M (2005) Fast synchronization on shared-memory multiprocessors: an architectural approach. J Parallel Distrib Comput 65:1158–1170
15. Garzaran M, Prvulovic M, Zhang Y, Jula A, Yu H, Rauchwerger L, Torrellas J (2001) Architectural support for parallel reductions in scalable shared-memory multiprocessors. In: Proceedings of the 2001 international conference on parallel architectures and compilation techniques, Sept 2001, pp 243–254
16. Gottlieb A, Grishman R, Kruskal C, McAuliffe K, Rudolph L, Snir M (1983) The NYU multicomputer—designing a MIMD shared-memory parallel machine. ACM Trans Program Lang Syst 5(2):164–189
17. Gray J (ed) (1993) The benchmark handbook for database and transaction systems, Chap 6, 2nd edn. Morgan Kaufmann, San Mateo
18. Hall M, Kogge P, Koller J, Diniz P, Chame J, Draper J, LaCoss J, Granacki J, Brockman J, Srivastava A, Athas W, Freeh V (1999) Mapping irregular appilcations to DIVA, a PIM-based data-intensive architecture. In: Supercomputing'99, Nov 1999
19. Hao M, Heinrich M (2003) Active I/O switches in system area networks. In: Proceedings of the ninth annual symposium on high performance computer architecture, Feb 2003, pp 365–376
20. Hewlett-Packard Inc (2011) The open source database benchmark
21. Intel Corporation (2011) Intel Itanium2 processor reference manual
22. International Technology Roadmap for Semiconductors (2011) Executive summary 2003 edition. http://public.itrs.net/Files/2003ITRS/Home2003.htm
23. Kalla R, Sinharoy B, Tendler JM (2004) IBM Power5 chip: a dual-core multithreaded processor. IEEE MICRO 24(2):40–47
24. Keeton K, Patterson DA (1999) Towards a simplified database workloads for computer architecture evaluations. In: Workshop on workload characterization, Austin, TX, USA, Oct 1999
25. Kessler RE (1999) The Alpha 21264 microprocessor. IEEE MICRO 19(2):24–36

26. Kim D, Chaudhuri M, Heinrich M, Speight E (2004) Architectural support for uniprocessor and multiprocessor active memory systems. IEEE Trans Comput 53(3):288–307
27. Koester D, Kepner J (2003) HPCS assessment framework and benchmarks. MITRE and MIT Lincoln Laboratory, Mar 2003
28. Kogge P (1994) The EXECUBE approach to massively parallel processing. In: International conference on parallel processing, Aug 1994
29. Kumar S et al (2008) Atomic vector operations on chip multiprocessors. In: Proceedings of the 35th annual international symposium on computer architecture, June 2008, pp 441–452
30. Kuskin J et al (1994) The Stanford FLASH multiprocessor. In: Proceedings of the 21st annual international symposium on computer architecture, Chicago, IL, USA, May 1994, pp 302–313
31. Laudon J, Lenoski D (1997) The SGI Origin: a ccNUMA highly scalable server. In: ISCA97, Denver, CO, USA, June 1997, pp 241–251
32. Marin G, Mellor-Crummey JM (2004) Cross-architecture performance predictions for scientific applications using parameterized models. In: Proceedings of the international conference on measurement and modeling of computer systems (Sigmetrics '04), June 2004, pp 2–13
33. McCalpin J (1999) Stream: sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/
34. Mellor-Crummey JM, Scott ML (1991) Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans Comput Syst 9(1):21–65
35. Nikolopoulos DS, Papatheodorou TA (2001) The architecture and operating system implications on the performance of synchronization on ccNUMA multiprocessors. Int J Parallel Program 29(3):249–282
36. Oskin M, Chong F, Sherwood T (1998) Active pages: a model of computation for intelligent memory. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, Spain, June 1998, pp 192–203
37. Patterson D, Anderson T, Cardwell N, Fromm R, Keaton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for Intelligent RAM: IRAM. IEEE MICRO 17(2):34–44
38. Petrini F, Fernandez J, Frachtenberg E, Coll S (2003) Scalable collective communication on the ASCI Q machine. In: 11th symposium on high performance interconnects, Stanford, CA USA, Aug 2003
39. Pinkston T, Agarwal A, Dally W, Duato J, Horst B, Smith TB (2002) What will have the greatest impact in 2010: the processor, the memory, or the interconnect? HPCA8 Panel Session, Feb 2002
40. Saulsbury A, Pong F, Nowatzyk A (1996) Missing the memory wall: the case for processor/memory integration. In: Proceedings of the 23rd annual international symposium on computer architecture, May 1996, pp 90–101
41. Scott S (1996) Synchronization and communication in the T3E multiprocessor. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, Cambridge, MA, USA, Oct 1996, pp 26–36
42. Shao M, Ailamaki A, Falsafi B (2003) DBmbench: fast and accurate database workload representation on modern microarchitecture. Technical Report CMU-CS-03-161, Carnegie Mellon University
43. Silicon Graphics, Inc (2001) SGI™Origin™3000 Series Technical Report, Jan 2001
44. Silicon Graphics, Inc (2001) SN2-MIPS Communication Protocol Specification, Revision 0.12, Nov 2001
45. Solihin Y, Lee J, Torrellas J (2001) Automatic code mapping on an intelligent memory architecture. IEEE Trans Comput 50(11):1248–1266
46. Solihin Y, Lee J, Torrellas J (2002) Using a user-level memory thread for correlation prefetching. In: Proceedings of the 29th annual international symposium on computer architecture, May 2002, pp 171–182
47. Sorin DJ, Lemon J, Eager DL, Vernon MK (2003) Analytic evaluation of shared-memory architectures. IEEE Trans Parallel Distrib Syst 14(2):166–180
48. Sorin DJ, Pai VS, Adve SV, Vernon MK, Wood DA (1998) Analytic evaluation of shared-memory systems with ILP processors. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, Spain, June 1998, pp 380–390
49. Tipparaju V, Nieplocha J, Panda D (2003) Fast collective operations using shared and remote memory access protocols on clusters. In: Proceedings of the international parallel and distributed processing symposium, Apr 2003, p 84a
50. Torrellas J, Hennessy JL, Weil T (1990) Analysis of critical architectural and program parameters in a hierarchical shared memory multiprocessor. In: Proceedings of the international conference on measurement and modeling of computer systems (Sigmetrics '90), May 1990, pp 163–172

51. Torrellas J, Nguyen A-T, Yang L (2000) Toward a cost-effective DSM organization that exploits processor-memory integration. In: Proceedings of the seventh annual symposium on high performance computer architecture, Jan 2000, pp 15–25
52. TPC-D, Past, Present and Future: An Interview between Berni Schiefer, Chair of the TPC-D Subcommittee and Kim Shanley, TPC Chief Operating Officer. (2011). available from http://www.tpc.org/
53. von Eicken T, Culler DE, Goldstein SC, Schauser KE (1992) Active messages: a mechanism for integrated communication and computation. In: Proceedings of the 19th annual international symposium on computer architecture, Gold Coast, Australia, May 1992, pp 256–266
54. Yoo J, Yoo S, Choi K (2011) Active memory processor for network-on-chip based architecture. IEEE Trans Comput Apr 2011
55. Zhang L (2003) UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, May 2003
56. Zhang L, Fang Z, Carter JB (2004) Highly efficient synchronization based on active memory operations. In: International parallel and distributed processing symposium, Apr 2004
57. Zhang L, Fang Z, Parker M, Mathew B, Schaelicke L, Carter J, Hsieh W, McKee S (2001) The impulse memory controller. IEEE Trans Comput 50(11):1117–1132
58. Zhao L, Iyer R, Makineni S, Bhuyan L, Newell D (2005) Hardware support for bulk data movement in server platforms. In: Proceedings of the 23th international conference on computer design, Oct 2005, pp 53–60
59. Zotov I (2010) Distributed virtual bit-slice synchronizer: a scalable hardware barrier mechanism for n-dimensional meshes. IEEE Trans Comput 59(9):1187–1199