ISSN 2347-4289

Shared Memory Multiprocessor

Ankush Saini, Inderjeet Singh Behl, Jaideep Verma

(Dronacharya college of Engineering,Gurgaon) Computer Science and Engineering,Student, New Delhi,India;
(Dronacharya college of Engineering,gurgaon) Computer Science and Engineering, Student, Gurgaon, India;
(Dronacharya college of Engineering,Gurgaon) Computer Science and Engineering,Student,Student,Bahadurgarh,India;
Email: ankushsaini90@gmail.com, isbahl@yahoo.com, vickyjd12@gmail.com

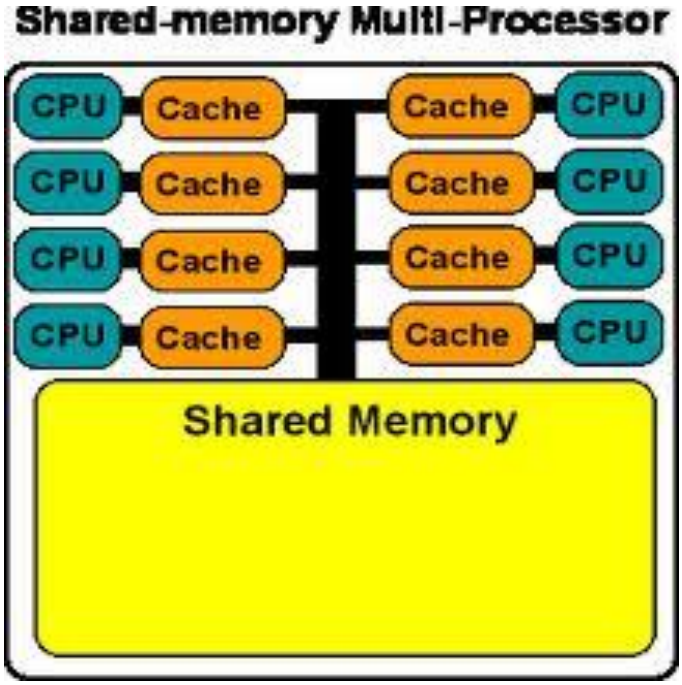
ABSTRACT: Synchronization is a critical operation in many parallel applications. Conservative Synchronization mechanisms are failing to keep up with the increasing demand for well-organized management operations as systems grow larger and network latency increases. The charity of this paper is threefold. First, we revisit some delegate bringing together algorithms in light of recent architecture innovation and provide an example of how the simplify assumption made by typical logical models of management mechanism can lead to significant performance guess errors. Second, we present an architectural modernism called active memory that enables very fast tiny operations in a shared-memory multiprocessor. Third, we use execution-driven simulation to quantitatively compare the performance of a variety of Synchronization mechanisms based on both existing hardware techniques and active memory operations. To the best of our knowledge, management based on active memory out forms all existing spinlock and non-hardwired difficulty implementations by a large border.

Keywords: Quantitatively, DRAM, RMW functionality, Conservative, Synchronization, Implemtation

1 INTORDUCTION

Barriers and spinlocks are management mechanisms commonly used by many parallel applications. A barrier ensures that no process in a group of cooperate processes advances beyond a given point until all processes have reached the barrier. A spinlock ensures atomic access to data or code protected by the lock. Their good Synchronization often limits the possible concurrency, and thus presentation, of parallel applications. The presentation of management operations is limited by two factors: (i) the number of remote access required for a management operation and (ii) the latency of each remote access. The impact of Synchronization performance on the overall performance of parallel applications is increasing due to the growing speed gap between Processors and memory. Processor speeds are increasing by approximately 55% per year, while local DRAM latency is improving only approximately 7% per year and remote memory latency for large-scale machines is almost constant due to speed of light effects [26]. For instance, a 32-Processor barrier operation on an SGI Origin 3000 system takes about 232,000 cycles, During which time the 32 R14K processors could have executed 22 million FLOPS. This 22 MFLOPS/barrier ratio is an alarming signal that conventional Synchronization mechanisms hurt system performance. Over the years, many Synchronization mechanisms and algorithms have been developed for shared memory multiprocessors. The classical paper on Synchronization by Mellor-Crummy and Scott provides a thorough and detailed study of representative barrier and spinlock algorithms, each with their own hardware assumption. More recent work surveys the major research trends of spinlocks. Both papers investigate Synchronization more from an algorithmic perspective than from a hardware/architecture angle. We feel that a hardware-centric study of Synchronization algorithms is a necessary addition to this prior work, especially given the variety of new architectural features and the important quantitative changes that have taken place in multi processor systems over the last decade. Sometimes small architectural innovation can cancel out key algorithmic scalability properties. For example, neither of the above-mentioned papers differentiate between the way that the various read-modify-write(RMW) primitives (e.g., test-and-set or compare-and-swap) are physically implement. However, the location of the hardware RMW unit, e.g., in the processor or in the communication fabric or the memory system, can have a dramatic impact on Synchronization performance. For example, we find that when the RMW functionality is performed near the memory/directory controller rather than via Processor-side atomic operations, 128-processor barrier performance can be improved by a factor of 10. Moving the RMW operation from the processor to a memory controller can change the effective time complexity of a barrier operation to O(1) network latencies from no better than O(N) in straight implementations for the same basic barrier algorithm. This examination illustrates the potential problems associated with performing conventional “pen-and-pencil” algorithmic difficulty analysis on Synchronization mechanisms. While paper-and-pencil analysis of algorithms tends to ignore many of the subtlety that make a big dissimilarity on real machines, running and compare programs on real hardware is limited by the hardware primitives available on, and the configurations of, available machines. Program outline study is hard because the hardware performance monitor counter provide limited treatment and analysis them during program finishing changes the actions of an otherwise full-speed run. Experiment with a new hardware ancient is near not possible on an accessible machine. As a result, in this paper we use execution-driven recreation to evaluate mixes of Synchronization algorithms, hardware primitives, and the physical implementations of these primitives. We do not attempt to provide a broad costing of all future barrier and spinlock algorithms. Rather, we price versions of a barrier from a marketable library and a delegate spinlock algorithm modified to several motivating hardware platforms. Detailed simulation helps us compare quantitatively the presentation of these Synchronization implementations and, when appropriate, correct preceding mis-assumptions about Synchronization algorithm performance when run on real hardware.

Copyright © 2013 IJTEEE.

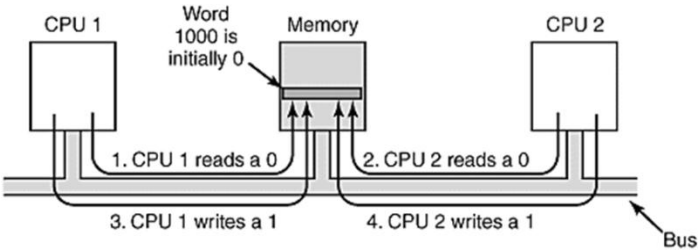


2 History of Shared Memory Multiprocessor:-

The history of shared memory multi processors goes back to the early 1970s, to two powerful research projects at Carnegie- Mellon University. The first machine, named cwmmp (from the PMS notation for ``computer with multiple mini-processors''), was organized around a crossbar switch that connected 16 PDP-11 processors to 16 memory banks. The second, cm*, also used PDP-11 processors, but connected them via the tree-shaped network shown in Figure 17 on page 63. The basic building block for this system was a PROCESSOR cluster, which consisted of four processors, each with their own local memories. The global memory space was evenly partitioned among the memories in the system. When a PROCESSOR generated a request for address i, its bus logic would check to see if i was in the range of addresses in that machine's local memory. If it wasn't, the request was transferred to a cluster controller, which would see if i belonged to any other memory within that cluster. If not, the request would be routed up the tree to another level of cluster controllers. In all, 50 PROCESSORs were connected by three levels of buses. cm* was an early example of a non-uniform memory access (NUMA) architecture. Depending on whether an item was in a processor's local memory, within the same cluster, or in another cluster, the time to fetch an item was 3 μs, 9 μs, or 27 μs, respectively. As a reference point, a PDP-11 of this era, without the cluster interconnection logic, could fetch an item from main memory in about 2 μs.

3. Synchronization on CC-NUMA Multiprocessor:-

In this section, we describe how synchronization operations are implementing on conventional shared memory multiprocessors. We then describe architectural advance that has been future in recent years to get better synchronization show. Finally, we provide a short time difficulty approximation for barriers and spinlocks based on the more than a few of available primitives.



3.1 Background:-

3.1.1 Atomic Operation on CC-NUMA System:-

Most system supply some form of processor-centric atomic RMW operation for programmers to apply synchronization primitives. For example, the Intel Itanium TM processor chains semaphore instructions [14], while most major RISC processors [7, 16, 20] use load-linked/store-conditional orders. An LL(load-linked) order loads a block of data into the cache. A succeeding SC(store-conditional) instruction attempt to write to the same block. It succeeds only if the block has not been referenced since the previous LL. Any memory reference to the block from another computer between the LL and SC pair cause the SC to fail. To apply an atomic primitive, library routines typically retry the LL/SC pair repeatedly until the SC succeeds.

(a) naive coding

```
atomic_inc( &barrier_variable );
spin_until( barrier_variable == num_procs );
int count = atomic_inc( &barrier_variable );
```

(b) "optimized" version

```
if( count == num_procs−1 )
spin_variable = num_procs;
else
spin_until( spin_variable == num_procs );
```

A disadvantage of processor-centric atomic RMW operations is that they begin inter processor message for every atomic operation. In a directory-based write-invalidate CC-NUMA system, when a computer needs to modify a common variable, the local DSM hardware sends a message to the variable's home node to acquire select rights. In response, the home node characteristically sends cancellation mail to other nodes sharing the data. The resulting system latency brutally impacts the good organization of synchronization.

3.1.2 Barrier:-

Figure 1(a) shows a naive barrier completion. This completion is incompetent since it directly spins on the barrier variables. Since process that have reach the barrier frequently try to understand writing the barrier variables, the next increase effort by another process will struggle with these understand requirements, probably resultant in a lengthy latency for the increase action. even though process that have reach the barrier can be floating to keep away from obstruction with the consequent increase operations, the transparency of postpone and resume process is characteristically also far above the ground to be helpful. A ordinary optimization to this barrier execution is to bring in a new variables on which process spin, e.g., spin variable in Figure 1(b). Because coherence is maintained at cache line granularity, barrier variable and spin variables be supposed to not exist in on the similar cache line. by a

Copyright © 2013 IJTEEE.

separate spin variable eliminate fake distribution connecting the spin and increase operation. on the other hand, doing so introduce an additional write to the spin variables for every barrier process, which cause the abode node to drive an cancellation ask for to each computer and then each computer to refill the spin variable. yet, the advantage of using a divide spin variables often outweigh its in the clouds, which is a standard illustration of trade training difficulty for act. Nikolopoulos and Papatheodorou [23] have confirmed that using a divide spin variables improve act by 25% for a 64-PROCESSOR barrier synchronization. We divide the sum point in time necessary to carry out a barrier action into two mechanism, *collect* and *discharge* *Gather* is the time during which each filament signals its arrival at the barrier. *discharge* is the time it takes to suggest to every process that the barrier operation has completed and it is time to progress.

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

Figure 2. Ticket lock code

Assuming a best case situation where there is no outside meddling, the algorithm depict in Figure 1(b) require at smallest quantity 15 one-way mail to apply the *collect* phase in a easy 3-process barrier using LL/SC or processor-side atomic commands. These 15 mails consist first and foremost of cancellation, termination acknowledgement, and weight ask for mail. Twelve of this mail must be performing in sequence, i.e., they cannot be overlap. In the optimized barrier code, when the last process arrive, it invalidate the cached copy of spin variables in the spinning processors, modify spin variables in DRAM, and we enter *discharge* phase. Every spinning procedure needs to fetch the cache line that contains spin variables from its home node. Modern PROCESSORs use more and larger cache line size to arrest spatial area. When the number of process is large, this burst of reads to spin variables will cause obstruction at the DRAMs and network boundary. several researchers contain planned *barrier trees* [12, 27, 33], which use many barrier variables prepared hierarchically so that tiny operation on dissimilar variables be able to be done in parallel. For example, in Yew *et.al.*'s software combine tree [33], processors are leaves of a tree. The processors are prepared into group and when the last processor in a collection arrives at the barrier, it increment a answer in the group's parent node. Continuing in this fashion, when all groups of processors at a particular level in the barrier tree arrive at the barrier, the last one to arrive increments an aggregate barrier one level higher in the hierarchy until the last processor reaches the barrier. When the final process arrive at the barrier and reach the root of the barrier tree, it trigger a wave of awaken operation downward the tree to signal each to come processor. Barrier trees attain important act gains on large system by falling hot spots, but they require additional programming effort and their appearance is forced by the base (single group) barrier act.

3.1.2 SPINLOCKS:-

In this section we describe a series of via implementations for a mutual exclusion spin lock_ The first four appear in the literature in one form or another. The fifth is a novel lock of our own design. Each lock can be seen as an attempt to eliminate some decency in the previous design. Each assumes a shared memory environment that includes certain fetch and operations_ As noted above a substantial body of work has also addressed mutual exclusion using more primitive read and write atomicity_ the complexity of the resulting solutions is the principal motivation for the development of fetch and _ primitives_ Other researchers have considered mutual exclusion in the context of distributed systems but the characteristics of message passing are deferent enough from shared memory operations that solutions do not transfer from one environment to the other. Our pseudo code notation is meant to be more or less self explanatory_ We have used line breaks to terminate statements_ and indentation to indicate nesting in control constructs_ The keyword shared indicates that a declared variable is to be shared among all processors_ The declaration implies no particular physical location for the variable_ but we often specify locations in comments and/or accompanying text_ The keywords processor private indicate that each processor is to have a separate_ independent copy of a declared variable_ All of our atomic operations are written to take as their rest argument the address of the memory location to be modified. All but compare and swap take the obvious one additional operand and return the old contents of the memory location_ Compare and swap address old new is denied as if address old return false address new_ return true_ In one case algorithm . We have used an atomic add operation whose behavior is the same as calling fetch and add and discarding the result.

3.2 RELATED ARCHITECTURAL IMPROVEMENT TO SYNCHRONIZATION:-

Many architectural innovation have been future over the decades to conquer the overhead induce by cc-NUMA coherence protocols on synchronization primitives, as described in Section 2.1.The fastest hardware synchronization completion uses a pair of devoted wires connecting every two nodes [8, 29]. However, for large systems the cost and packaging complexity of running dedicated wires connecting every two nodes is too expensive. In adding to the high cost of physical wires, this move toward cannot hold up more than one barrier at one time and does not act together well with load-balancing techniques, such as process immigration, where the process-to-PROCESSOR mapping is not static. The *fetch-and-*_ instructions in the NYU Ultra computer [11, 9] are the first to implement atomic operations in the memory controller. In addition to *fetch-and-add* hardware in the memory module, the Ultra computer supports an inventive combine network that combines reference for the similar memory location inside the routers. The SGI Origin 2000 [18] and Cray T3E [28] hold up a set of memory-side atomic operation (MAOs) that are trigger by write to unique IO address on the home node of synchronization variables. However, MAOs do not work in the logical domain and rely on software to keep up coherence, which has limited their usage. The Cray/Tera MTA [17, 1] uses full/empty bits in memory to apply synchronized memory references in a cache less system. However, it require custom DRAM (an extra bit for every word) and it is not clear how it can work professionally in presence of caches. The *fetch add* operation of

Copyright © 2013 IJTEEE.

the MTA is similar to and predates the MAOs of the SGI Origin 2000 and Cray T3E. Active message are an well-organized way to arrange parallel applications [5, 32]. An active message includes the address of a user-level handler to be executed by the receiving processor upon message arrival using the message body as the arguments. Active messages can be used to perform atomic operations on a synchronization variable's home node, which eliminates the need to shuttle the data back and forth across the network or perform long latency for remote invalidations. However, performing the synchronization operations on the node's primary processor quite than on devoted synchronization hardware entail higher summons latency and interfere with useful work being performed on that processor. In particular, the load imbalance induces by having particular node hold synchronization traffic can severely impact performance due to Amdahl's Law effects. The I-structure and explicit token-store (ETS) mechanism supported by the early dataflow project Monsoon [4, 24] can be used to implement synchronization operations in a manner similar to active messages. A token comprises a value, a pointer to the instruction to execute (IP), and a pointer to an activation frame (FP). The instruction in the IP specifies an opcode (e.g., ADD), the offset in the activation frame where the match will take place (e.g., FP+3), and one or more destination instructions that will receive the result of the operation (e.g., IP+1). If synchronization operations are to be implemented on an ETS machine, software needs to manage a fixed node to handle the tokens and wake up stalled threads. QOLB [10, 15] by Goodman *et al.* serializes synchronization requests through a distributed queue supported by hardware. The hardware queue mechanism greatly reduces network traffic. The hardware cost includes three new cache line states, storage for the queue entries, a "shadow line" mechanism for local spinning, and a mechanism to perform direct node-to-node lock transfers. Several recent clusters off-load synchronization operations from the main processor to network processors [13, 25]. Gupta *et al.* [13] use user-level one-sided MPI protocols to implement communication functions, including barriers. The Quadrics TMQs Net interconnect used by the ASCI Q supercomputer [25] supports both a pure hardware barrier using a crossbar switch and hardware multicast, and a hybrid hardware/software tree barrier that runs on the network processors. For up to 256 participating PROCESSORs, our AMO-based barrier performs better than the Quadrics hardware barrier. As background traffic increases and the system grows beyond 256 PROCESSORs, we expect that the Quadrics hardware barrier will outperform AMO-based barriers. However, the Quadrics hardware barrier has two restrictions that limit its usability. First, only one PROCESSOR per node can participate in the synchronization. Second, the synchronizing nodes must be adjacent. Their hybrid barrier does not have these restrictions, but AMO-based barriers outperform them by a factor of four in all of our experiments.

3.3 TIME COMPLEXITY ANALYSIS:-

It is habitual to assess the performance of synchronization algorithms in terms of the number of remote memory reference (*i.e.*, network latencies) required to perform each operation. With atomic RMW instructions, the *collect* stage latency of an N-process barrier includes 4N non-overlap able one-way network latencies, illustrated in Figure 3(a). To increase the barrier count, each processor must issue an limited ownership request to the barrier count's home node, which issues an invalidation message to the prior owner, which responds with an invalidation acknowledgement, and finally the home node sends the requesting processor an exclusive ownership reply message. If we continue to assume that network latency is the primary performance determinant, the time complexity of the *discharge* stage is O(1), because the N cancellation messages and successive N refill requests can be pipelined. However, researchers have reported that memory controller (MMC) occupancy has a greater impact on barrier performance than network latency for medium-sized DSM multi processors [6]. In other words, the assumption that coherence messages can be sent from or processed by a particular memory controller in negligible time does not hold. If MMC occupancy is the key determinant of performance, then the time complexity of the *discharge* stage is O(N), not O(1). Few modern processors directly apply atomic RMW primitives. Instead, they provide some variant of the LL/SC instruction pair discussed in Section 2.1. For small systems, the performance of barriers Implemented using LL/SC instructions is close to that of barriers built using atomic RMW instructions. For larger systems, however, competition between processors cause a lot of obstruction, which can lead to a large number of reverse offs and retries. While the best case for LL/SC is the same as for atomic RMWs (4N message latencies), the worst case number of message latencies for the *collect* phase of an LL/SC-based barrier is O(N2). Typical performance is somewhere in between, depending on the amount of work done between barrier operations and the average skew in arrival time between different processors. If the atomic operation functionality resides on the MMC, as it does for the Cray [28] and SGI [18] machines that support MAOs, a large number of non-overlapping cancellation and refill messages can be eliminate. In these systems, the RMW unit is strongly coupled to the local coherence controller, which eliminates the need to invalidate cached copies before updating barrier variable in the *collect* phase. Instead, synchronizing processors can send their atomic RMW requests to the home MMC in parallel,

Hardware support	Gather stage	Release stage	Total
Processor-side atomic	O(N)	O(1) or O(N)	O(N)
LL/SC best case	O(N)	O(1) or O(N)	O(N)
LL/SC worst case	$O(N^2)$	O(1) or O(N)	$O(N^2)$
MAO network latency bound	O(1)	O(1)	O(1)
MAO MMC occupancy bound	O(N)	O(N)	O(N)
ActMsg network latency bound	O(1)	O(1)	O(1)
ActMsg MMC occupancy bound	O(N)	O(N)	O(N)

Table 1. Time complexity of barrier with different hardware support.

and the MMC can perform the requests in a pipelined style (Figure 3(b)). Since the operations by each processor are no longer serialized, the time complexity of the *collect* phase MAO-based barriers is O(N) memory controller operations or O(1) network latencies, depending on whether the bottleneck is MMC tenancy or network latency. Since MAOs are performed on non-coherent (IO) addresses, the release stage requires processors to spin over the interconnect, which can by bring in significant network and memory controller load. Barriers built using active messages are comparable to those build with MAOs in that the atomic increase requests on a par-

Copyright © 2013 IJTEEE.

ticular worldwide variable are sent to a single node, which eliminate the message latencies required to maintain coherence. Both completion strategies have the same time complexity as measured in network latency. However, active messages typically use interrupts to trigger the active message handler on the home node processor, which is a far higher latency operation than pipelined atomic hardware operations performed on the memory controller. Thus, barriers built using active messages are more likely to be occupancy-bound than those built using MAOs. Recall that when occupancy is the primary performance bottleneck, the *collect* phase has a time complexity of O(N). In terms of performance, spinlocks suffer from similar problems as barriers. In a program using ticket locks for mutual exclusion, a process can enter the critical section in O(1) time, as measured in network latency, or O(N) time, as measured in memory controller occupancy. A program using Anderson's array based queuing lock has O(1) complexity using either measurement method. Table 1 summarizes our discussions of non-tree-based barriers. If we build a tree barrier from one of the above basic barriers, the O(N) and O(N2) time complexity cases can be reduced to O(log(N)), albeit with potentially non-trivial constant factors. In Section 4.2.2, we revisit some of these conclusions to determine the extent to which often ignored machine features can affect synchronization complexity analysis.

4. AMU-SUPPORTED SYNCHRONIZATION:-

We are investigating the value of augment a conventional memory controller (MC) with an **Active Memory Unit** (**AMU**) capable of performing arts simple atomic operations. We refer to such atomic operations as **Active Memory Operations** (**AMOs**). AMOs let processors ship simple computations to the AMU**.** On the home memory controller of the data being processed, instead of loading the data in to a processor, processing it, and writing it back to the home node. AMOs are particularly useful for data items with poor temporal locality that are not accessed many times between when they are loaded into a cache and later evicted, e.g., synchronization variables. Synchronization operations can exploit AMOs by performing atomic read-modify-write operations at the home node of the synchronization variables, rather than bouncing them back and forth across the network as each processor tests or modifies them. Unlike the MAOs of Cray and SGI machines, AMOs operates on cache coherent data and can be programmed to trigger coherence operations when certain events occur. Our proposed mechanism augments the MIPS R14K ISA with a few AMO instructions. These AMO instructions are encoded in an unused portion of the MIPS-IV instruction set space. Different synchronization algorithms require different atomic primitives. We are considering a range of AMO instructions, but for this study we consider only amo.inc (increment by one) and amo.fetch add (fetch and add). Semantically, these instructions are identical to the corresponding atomic processor-side instructions, so programmers cause them as if they were processor-side atomic operations.

4.1 HARDWARE ORGANIZATION:-

Figure 4 depicts the architecture that we assume. A per-node crossbar connects local processors to the network backplane, local memory, and IO subsystems. We assume that the processors, crossbar, and memory controller all reside on the same die, as will be typical in near-future system designs. Figure 4 (b) presents a block diagram of a memory controller with the proposed AMU delimited within the dotted box. When a PROCESSOR issues an AMO instruction, it sends a command message to the target address' home node. When the message arrives at the AMU of that node, it is placed in a queue awaiting dispatch. The control logic of the AMU exports a READY signal to the queue when it is ready to accept another request. The operands are then read and fed to the function unit (the FU in Figure 4 (b)). Accesses to synchronization VARIABLES by the AMU exhibit high temporal locality because every participating process accesses the same synchronization variable, so our AMU design incorporates a tiny cache. This cache eliminates the need to load from and write to the off-chip DRAM for each AMO performed on a particular synchronization variable. Each AMO that hits in the AMU cache takes only two cycles to complete, regardless of the number of processors contending for the synchronization variable. An *N*-word AMU cache supports outstanding synchronization operations on *N* different synchronization VARIABLES. In our current design we model a four-word AMU cache. The hardware cost of supporting AMOs is negligible. In a 90nm process, the entire AMU consumes approximately 0.1mm2, which is below 0.06% of the total die area of a high-performance micro processor with an integrated memory controller.

4.2 FINE-GRAINED UPDATES:-

AMOs operates on coherent data. AMU-generated requests are sent to the directory controller as fine grained "get" (for reads) or "put" (for writes) requests. The directory controller still maintains coherence at the block level. A fine-grained "get" loads the coherent value of a word (or a double-word) from local memory or a processor cache, depending on the state of the block containing the word. The directory controller changes the state of the block to "shared" and adds the AMU to the list of sharers. Unlike traditional data sharers, the AMU is allowed to modify the word without obtaining exclusive ownership first. The AMU sends a fine-grained "put" request to the directory controller when it needs to write a word back to local memory. When the directory controller receives a put request, it will send a word-update request to local memory and every node that has a copy of the block containing the word to be updated. To take advantage of fine-grained gets/puts, an AMO can include a "test" value that is compared against the result of the operation. When the result value matches the "test" value, the AMU sends a "put" request along with the result value to the directory controller. For instance, the "test" value of amo.inc can be set as the total number of processes expected to reach the barrier and then the update request acts as a signal to all waiting processes that the barrier operation has completed. One way to optimize synchronization is to use a write-update protocol on synchronization VARIABLES. However, issuing a block update after each write generates an enormous amount of network traffic, which offsets the benefit of eliminating invalidation requests. In contrast, the "put" mechanism issues word-grained updates, thereby eliminating false sharing. For example, amo.inc only issues updates after the last process reaches the barrier rather than once every time a process reaches the barrier. The "get/put" mechanism introduces temporal inconsistency between the barrier variable values in the processor caches and the AMU cache. In essence, the delayed "put" mechanism implements a release 1Fine-grained "get/put" operations are part of a more general DSM architecture we are investigating, details of which are beyond the scope of this paper. Consistent memory model for barrier vari-

Copyright © 2013 IJTEEE.

ables, where the condition of reaching a target value acts as the release point.

4.1 Programming With Amos:-

With AMOs, atomic operations are perform at the memory controller without invalidate shared copy in processor caches. In the case of AMO-based barriers, the cached copy of the barrier adds up are efficient when the final process reach the barrier. Consequently, AMO-based barriers can use the inexperienced algorithm shown in Figure 1(a) by simply replace atomic inc(&barrier variable) with amo inc(&barrier variable, num procs), where amo inc() is a wrapping function for the amo.inc instruction. The amo.fetch add instruction adds a designated value to a specified memory location, updates the shared copies in processor caches with the new value, and returns the old value. To implement spinlocks using AMOs, we replace the atomic primitive fetch and add in Figure 2 with the corresponding AMO instruction, amo fetch add(). We also use amo fetchadd() on the counter to take advantage of the "put" mechanism. Note that using AMOs eliminates the need to allocate the global variables in different cache lines. Like Act Msg and MAOs, the time complexity of AMO-based barriers and spinlocks is O(1) measured in terms of either network latency and O(N) in terms of memory controller occupancy. However, AMOs has much lower constants than Act Msg orMAOs, as will be apparent from the detailed performance evaluation. The various architectural optimizations further reduce the constant coefficients. Using conventional synchronization primitives often requires significant effort from programmers to write correct, efficient, and deadlock-free parallel codes. For example, the Alpha Architecture Handbook [7] dedicates six pages to describing the LL/SC instructions and restrictions on their use. On SGI IRIX systems, several library calls must be made before actually calling the atomic op function. To optimize performance, programmers must tune their algorithms to hide the long latency of memory references. In contrast, AMOs works in the cache coherent domain, do not lock any system resources, and eliminate the need for programmers to be aware of how the atomic instructions are implemented. In addition, we show in Section 4 that AMOs negate the need to use complex algorithms such as combining tree barriers and array-based queuing locks even for fairly large systems. Since synchronization-related codes are often the hardest portion of a parallel program to code and debug, simplifying the programming model is another advantage of AMO saver other mechanisms.

5 Conclusion

To improve synchronization effectiveness, we first identify and analyze the deficiency of conventional barrier and spinlock implementations. We exhibit how apparently innocent simplify assumptions about the architectural factors that impact performance can lead to incorrect analytical results when analyzing synchronization algorithms. Based on these experimental problems, we strongly support complete system experiments, either via in-situ experiments or highly detailed simulation, when estimating synchronization performance on modern multiprocessors. To improve synchronization performance, we propose that main memory controllers in multiprocessors be augmented to support a small set of active memory operations (AMOs). We show that AMO-based barriers do not require extra spin variables or complicated tree structures to achieve good performance. We further show that simple AMO-based ticket locks out perform alternate implementations using more complex algorithms. Finally, we show that AMO-based synchronization outperforms highly optimized conventional implementations by up to a factor of 62 for barriers and a factor 10 for spinlocks. In conclusion, we have established that AMOs overcome many of the deficiency of existing synchronization mechanisms, enable extremely efficient synchronization at rather low hardware cost, and reduce the need for cache coherence-conscious programming. As the number of processors, network latency, and DRAM latency grow, the value of fast synchronization will only grow in time and among the synchronization implementations we studied, and only AMO-based synchronization appears able to scale sufficiently to handle these upcoming changes in system performance.

Acknowledgment

The authors would like to thank Silicon Graphics Inc. for the technical documentations provided for the simulation, and in particular, the valuable input from Marty Deneroff, Steve Miller, Steve Reinhardt, Randy Passint and Donglai Dai. We would also like to thank Allan Gottlieb for his feedback on this work.

References

[1] http://www.wikipedia.org/

[2] http://www.studymode.com/

[3] https://www. books.google.co.in/

[4] T. Anderson. The performance of spin locks alternatives for shared-memory multiprocessors. *IEEE TPDS*, K.

[5] https://www.usenix.org/

[6] https://*www.phy.ornl.gov/*

Copyright © 2013 IJTEEE.