NIC-Based Atomic Remote Memory Operations in Myrinet/GM*

Abstract: *Efficient implementations of synchronization operations such as locks and semaphores is important in parallel and distributed systems. These operations can be efficiently implemented using hardware atomic Read-Modify-Write (RMW) memory operations on shared memory machines, however such primitives have not been available for SAN-connected clusters. Separate lock manager processes have typically been used in such an environment which can negatively impact CPU and network utilization. In this paper we describe our implementation and evaluation of NIC-based atomic remote memory operations which allow the implementation of locks without the use of lock manager processes.*

1 Introduction

Efficient implementations of synchronization operations such as locks and semaphores is important in parallel and distributed systems. These operations can be efficiently implemented using hardware atomic Read-Modify-Write (RMW) memory operations, such as test&set, compare&swap, etc., on shared memory machines [4]. As system area network (SAN) connected clusters are becoming more cost effective and popular, other methods for implementing locks are necessary, since such atomic RMW operations have not been available which operate across SANs. Synchronization operations for clusters are typically implemented with *lock manager* process running on one or more nodes which performs the operation. Such a process serves only to handle the synchronization operations and does not directly contribute to the computation. In fact, because it uses computational resources at the node it is running on, it negatively impacts the computation because it reduces the processor utilization at that node.

By using remote atomic memory operations which are supported by the communication layer, such as those described in the InfiniBand Architecture (IBA) standard[5], locks can be performed without the intervention of the remote host. This means that lock manager processes need not be used leading to higher CPU utilization. Furthermore, because context switches at the host processor are not needed to handle the lock requests, locks implemented using communication layer remote atomic memory operations can lead to better lock performance.

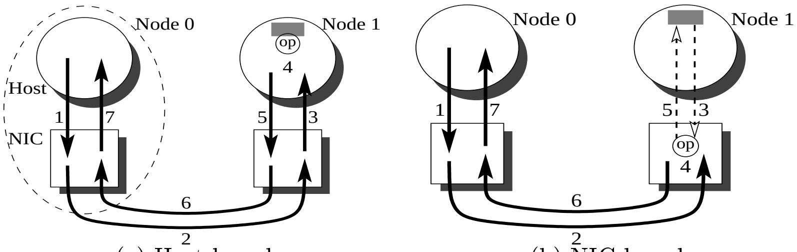
In this paper we describe our implementation and evaluation of network interface card based (NIC-based) remote atomic memory operations. We implemented these operations by modifying the GM message passing system[7] which uses programmable Myrinet network cards. Using programmable NICs to support collective communication operations has been previously described in [8], [2] and [3]. This paper takes this concept further and uses the programmable NICs to support atomic remote memory operations. We found a 15.66% improvement for performing a remote atomic operation using our NIC-based approach over using the best host-based implementation. When we implemented a distributed lock algorithm using the remote atomic operations our NIC-based implementation gave up to a 62.3% improvement over the best host-based implementation. Furthermore, we found that locks implemented with host-based atomic operations had a significant impact on host CPU utilization and network utilization, while locks implemented with NIC-based atomic operations had little to no impact.

The rest of the paper is organized as follows. Section 2 describes the basic concept of the NIC-based remote atomic operation. In Section 3 we describe the implementation, and Section 4 presents our experimental results. Finally, we conclude in Section 5.

2 NIC-Based Atomic Remote Memory Operations

The basic idea of the NIC-based remote atomic operations is to have the NIC perform the operation directly rather than dedicate a separate thread at the host to perform the operation. The application initiating the remote atomic operation would send a message to the NIC at the remote node indicating which operation to perform along with the operands. The remote NIC, upon receiving the message, would perform the operation on the memory at the host atomically. The atomicity of an operation is guaranteed by ensuring that the NIC does not perform any other operations on that memory region until that operation has completed and that any modifications on that memory region be done by the NIC, not the host. Figure 1 shows an example of an atomic operation being performed by a thread running on

*This research is supported in part by a DOE grant #DE-F002-01ER25506



(a) Host-based (b) NIC-based

Figure 1: Host-based and NIC-based atomic remote memory operations

the host processor (host-based atomic operation) and an atomic operation being performed by the NIC (NIC-based atomic operation).

In order to perform an atomic operation on a remote memory region without using NIC-based atomic operations, the remote node would need to have a process which receives the requests, performs the operations and returns the result. This is shown in Figure 1(a). Here, an application at node 0 sends a request for an atomic operation to a process at node 1 which performs the operation. This is performed in seven steps: (1) A message is generated by the application at the host of node 0 and is sent to the NIC. (2) The NIC then transmits it to the NIC at node 1. (3) The NIC at node 1 receives the message and forwards it to the process which is handling the atomic operations. (4) Upon receiving the message, the process at node 1 performs the operation specified in the message on the host's memory. (5) It then sends a reply message to the NIC which (6) transmits it back to node 0. (7) This reply is received by the NIC at node 0 and is forwarded to the application.

Using NIC-based atomic operations, no process is needed at the remote host to handle the atomic operations. Instead, the operations are performed directly by the NIC. Figure 1(b) shows a NIC-based atomic operation on remote memory. (1) An application at host 0 sends a special atomic operation message to the NIC which (2) transmits it to the NIC at node 1. (3) Upon receiving the message, the NIC copies the value stored at the memory location specified in the message using DMA. (4) The NIC then performs the operation using this value, and if necessary, (5) copies the new value back to the memory location, again using DMA. (6) The NIC then transmits the result back to the NIC at node 0. (7) That NIC then forwards it to the application.

We provide three atomic primitives: fetch&add, fetch&write and compare&swap. The fetch&add and fetch&write operations take four parameters: the target node id, the target port id, the remote virtual memory address, and a 32 bit data word. The compare&swap operation takes one additional 32 bit parameter which is used for the compare part of the operation. Table 1 describes the semantics of the operations. For each operation, the table shows the value of the memory location before the operation and after, as well as the value returned to the caller. The d and c values in the table represent the data and compare parameters of the operation.

3 Myrinet/GM Implementation

In this section we describe our implementation of the NIC-based remote atomic operations as a modification of Myricom's message passing system GM[7] version 1.5. We added two functions to the GM API. The `gm_provide_atomic_buffer()` function provides a *receive token* to the NIC. The NIC uses this receive token to send the return value to the application when the atomic operation has completed. The `gm_atomic_send_with_callback()` function builds a *send event* describing the atomic operation and includes all of the necessary parameters plus a *tag* value which it will use to match the atomic operation request with the reply.

A call to `gm_atomic_send_with_callback()` initiates the atomic operation. In response to this call, an *atomic operation packet* will be sent to the NIC a the remote node requesting that the atomic operation be performed. The remote NIC will DMA the data from the target memory location in the host to the NIC's memory, perform the operation, then DMA the new value back to the host's memory. Because

Table 1: Semantics of atomic memory operations

Operation	Memory Contents: Before Op	Memory Contents: After Op	Return Value
fetch&add (d)	X	$X + d$	X
fetch&write (d)	X	d	X
compare&swap (d, c)	X	if $c = X$ then d else X	X

the NIC performs this operation without interruption, the atomicity of the operation is guaranteed. The return value is stored in a table and a *reply packet* is sent back to the initiating node. This reply packet also serves as an acknowledgment for the atomic operation packet.

If an atomic operation packet is lost, the packet will time-out at the sender and it will be retransmitted. If the reply packet is lost, the corresponding atomic operation packet will time-out and will be retransmitted. The NIC at the destination node will receive a duplicate packet. Because the operations are not idempotent, the NIC cannot perform the operation again. Instead, the NIC looks up the sequence number in the table of return values and re-transmits a reply packet. Because there is a limited number of entries in the table of return values, each NIC must limit the number of unacknowledged atomic operations it has pending otherwise not all return values for outstanding packets can be stored.

4 Experimental Results

In this section, we evaluate our implementation of NIC-based atomic remote memory operations. We first evaluate the individual operations and compare them to host-based implementations. Next we implement a distributed lock using atomic remote memory operations and evaluate the performance of the locks using our NIC-based implementation and the host-based implementations. Finally, we evaluate impact of using NIC-based versus host-based atomic remote memory operation on CPU and network utilization.

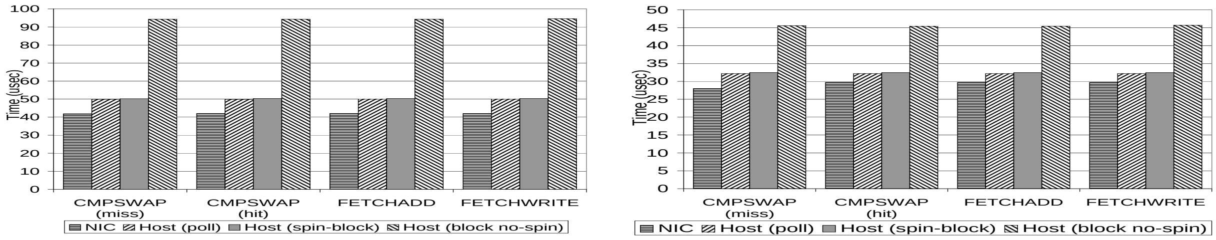
The performance results were run on two clusters. One cluster consists of 16[1] dual 300MHz Pentium II machines each with 128MB of RAM, running Linux kernel version 2.2.5. The machines are connected by a Myrinet[1] LAN network using NICs with 33MHz LANai 4.3 processors. These are connected to a 16 port switch. The second cluster consists of eight quad 700MHz Pentium III machines each with 1GB of RAM, running Linux kernel version 2.2.17. The machines are connected by another Myrinet LAN network using NICs with 66MHz LANai 7.2 processors. These are connected to an eight port switch.

4.1 Atomic Remote Memory Operations

We tested the performance of the NIC-based atomic operations and compared it to host-based implementations. A host-based implementation consists of a process which sends atomic operation request messages to a *server* process on another node which performs the operation and returns a reply message. Because a host-based implementation would most likely be on a separate thread which would be interrupt driven, we tested three different methods for checking for incoming messages. In the first case, *poll* the server process is polling for the reception of new request messages. In the second case, *spin-block*, the server process polls for a short while, then blocks waiting for the message. In the last case, *block no-spin*, the server process blocks waiting for a new message without polling. These three cases correspond to the GM functions `gm_receive()`, `gm_blocking_receive()` and `gm_blocking_receive_no_spin()`, respectively.

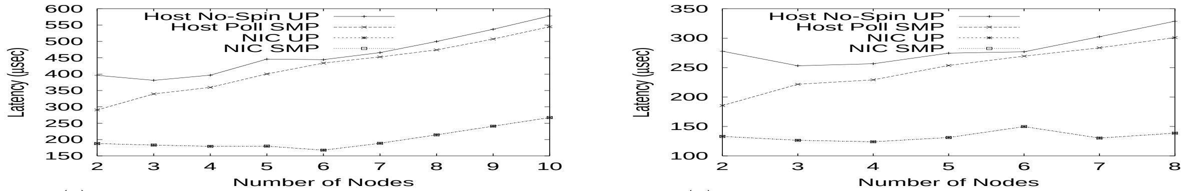
The tests consist of timing 10,000 iterations of each atomic operation then taking the averaging. The compare&swap operation was evaluated in two ways, once where the compare failed (*miss*), so that the swap was not performed, and once there the compare succeeded (*hit*). Figures 2(a) and 2(b) show the results of these tests for 300MHz processors with LANai 4.3 NICs, and 700MHz processors with LANai 7.2 NICs, respecively. Notice that in all cases the NIC-based atomic operations perform better than any of the host-based operations. Using the 300MHz hosts and 33MHz LANai 4.3 NICs, the NIC-based compare&swap (hit) operation took an average of $42.06\mu s$ as compared to the best host-based implementation, polling, which took an average of $49.87\mu s$, a 15.66% improvement. The host-based implementation which took the most time was blocking-no-spin which took an average of $94.26\mu s$ (55.38% improvement). When we used the 700MHz hosts with the 66MHz LANai 7.3 NICs the NIC-based compare&swap (hit) operation took an average of $29.73\mu s$ while the host-based polling implementation

[1] At the time when we ran the experiments only 10 of these machines were available.



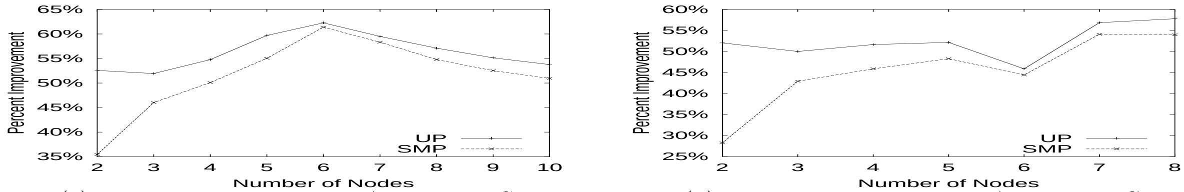
(a) 300MHz hosts, 33MHz LANai 4.3 NICs (b) 700MHz hosts, 66MHz LANai 7.2 NICs

Figure 2: Latencies of atomic operations



(a) 300MHz hosts, 33MHz LANai 4.3 NICs (b) 700MHz hosts, 66MHz LANai 7.2 NICs

Figure 3: Latency of locking and unlocking with multiple nodes contending



(a) 300MHz hosts, 33MHz LANai 4.3 NICs (b) 700MHz hosts, 66MHz LANai 7.2 NICs

Figure 4: Percentage improvement of lock benchmark

took 32.18μs and the host-based blocking-no-spin implementation took 45.36μs. This is a 7.61% and 34.46% improvement respectively.

4.2 Distributed Locks

We also evaluated the performance of a distributed lock algorithm built using the NIC-based atomic operations and compared it with host-based implementations. The algorithm creates a distributed linked list of nodes waiting for the lock. The node at the head of the queue holds the lock[6]. Our host based application consists of the *main thread*, a *server thread* and a shared memory region which both threads could read from and write to. The *main thread*, requests and releases locks using remote atomic operations. Remote atomic operations coming into a node are handled by the *server thread* which performs the operations on the shared memory region. Two versions of the server thread were used for various tests: polling and blocking-no-spin. The NIC-based application consists of just a single thread which requests and releases locks using NIC-based remote atomic memory operations.

In the test each node repeatedly locks and unlocks the same lock. We took the average time it takes a particular node to perform the lock-unlock procedure. The number of nodes was varied to vary the load. We ran the test using both an SMP enabled kernel, so that multiple CPUs were available for the threads, and using a uniprocessor kernel (UP), so that all threads shared the same processor. Figures 3(a) and 3(b) show the results of this test. We show the results for the host-based blocking-no-spin using the UP kernel, and the host-based polling using the SMP kernel and compare them to the NIC-based implementations on each kernel. Notice that in both graphs the NIC-based implementation outperforms the host-based implementations.

Figures 4(a) and 4(b) show the percentage improvement of the NIC-based implementation over the host-based implementation for the SMP and uniprocessor (UP) kernels. Notice that for the 300MHz machines, we see up to a 62.3% improvement for the UP case and up to a 61.4% improvement for the SMP case. For the 700MHz machines, we see up to a 57.8% improvement for the UP case and up to a 54.0% improvement for the SMP case.

4.3 CPU and Network Utilization

To test the impact of using the atomic operations on CPU utilization we performed a test where the home node of a lock performs a loop for 1,000μs and counts how many iterations of the loop it was able to perform. While the home node of the lock is doing that another node repeatedly locks and unlocks the lock at a certain rate. The number of iterations that the home node is able to perform in the alloted time is an indication of the amount of CPU time that is used in processing incoming atomic operation requests. In this test we used the uniprocessor kernel and the blocking-no-spin version of the server thread for the host-based implementation.

Figure 5 shows the results of this test. Notice that for both the 700MHz machines (labeled "700 L7" in the figure) and the 300MHz machines (labaled "300 L4"), the NIC-based implementation is unaffected by the number of atomic operatione being performed, while with the host-based implementation, the CPU utilization of the main thread at the home node decreases as the number of atomic operations the server node is processing increases.

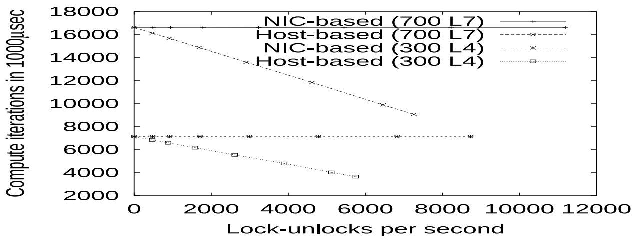


Figure 5: The number of iterations an application at the home node of a lock can perform in 1,000μs while another node is locking and unlocking the lock at a certain rate.

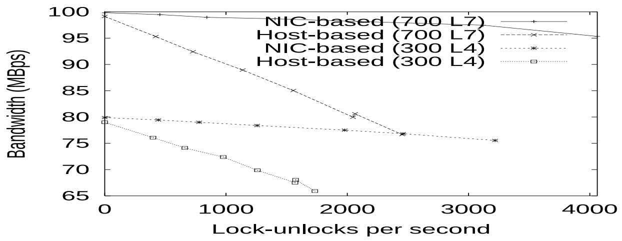


Figure 6: The send bandwidth achievable by an application at the home node of a lock while another node is locking and unlocking the lock at a certain rate.

We also evaluated the impact of handling atomic operations on communication bandwidth. This test is similar to the CPU utilization test except that the main thread at the lock home node is performing a bandwidth test with a third node. The lock home node measures the bandwidth by sending 1,000 16MB messages to the third node and waits for a reply from it. Figure 6 shows the results of this test. Notice that the impact of handling the atomic operations is greater on the host-based implementation than on the NIC-based implementation. In both implementations, atomic operation request and reply messages need to be sent and received which affect the bandwidth at the home node. However, in addition to this, in the host-based implementation, the process performing the bandwidth test is being interrupted to handle the incoming atomic operation requests. This has an effect on the rate at which the application can pass the send commands to the NIC.

5 Conclusions

In this paper, we presented an implementation of NIC-based atomic remote memory operations. We added support for the atomic operations to GM version 1.5. We then evaluated the performance of the NIC-based atomic operations and compared them with atomic operations implemented at the host level. We found a 15.66% improvement for the compare&swap operation when comparing it to the best host implementation. Using these atomic operations to implement a distributed lock, we saw up to a 62.3% improvement when using NIC-based atomic operations. Because the NIC-based atomic operations do not use the host processor they gave us better CPU utilization and a smaller impact on communication bandwidth than when using the host-based implementations.

By using the NIC-based remote atomic memory operations along with remote memory access methods (RMA) provided by some communication layers such as VIA and GM, applications can reduce the number of messages that need to be handled by the application. This means that for applications which currently use server threads, the number of interrupts can be reduced, or that the server thread can be eliminated altogether. This would lead to better CPU utilization and performance of the main thread. Such an approach demonstrates potential for designing high performance system area networks for next generation clusters and servers.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - a gigabit per second local area network. In *IEEE Micro*, February 1995.

[2] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Proceedings of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, 2000.

[3] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, (IPDPS)*, April 2001.

[4] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kaufmann, 1998.

[5] InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0. http://www.infinibandta.com.

[6] A. Kagi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *International Symposium on Computer Architecture*, June 1997.

[7] Myricom. Myricom GM Myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[8] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the International Conference on Parallel Processing*, pages III:156–165, Aug 1996.