NIC-Based Atomic Remote Memory Operations in Myrinet/GM

Darius Buntinas, Dhabaleswar K. Panda, and William Gropp

Technical Report
OSU-CISRC-1/02-TR01

NIC-Based Atomic Remote Memory Operations in Myrinet/GM

Darius Buntinas, Dhabaleswar K. Panda, William Gropp

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 a novel implementation and evaluation of NIC-based atomic remote memory operations which allow the implementation of locks without the use of lock manager processes.**

We modified the GM message passing system to support the NIC-based atomic remote memory operations. Our implementation gave us up to a 15.7% improvement over a host-based implementation of these operations. Furthermore, we saw up to a 62.3% improvement using the NIC-based atomic remote memory operations to implement locks when compared to implementing the locks using atomic remote memory operations implemented using a separate server process.

Keywords— **Atomic remote memory operations, Atomic operations, Remote memory access, GM, Myrinet**

I. 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 [1]. 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[2], 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[3] which uses programmable Myrinet[4] network cards. Using programmable NICs to support collective communication operations has been previously described in [5], [6], [7], [8], [9] and [10]. This paper takes this concept further and uses the programmable NICs to support atomic remote memory operations. We found a 15.7% improvement for performing a remote atomic operation using our NIC-based approach over using the best host-based implementation. When we implemented a dis-

This research is supported in part by a DOE grant #DE-F002-01ER25506 and a NSF grant #EIA-9986052.

Darius Buntinas and Dhabaleswar K. Panda are with the Network-Based Computing Laboratory, Department of Computer and Information Science at The Ohio State University, Columbus, OH 43210, ({buntinas, panda}@cis.ohio-state.edu).

William Gropp is with the Mathematics and Computer Science Division of Argonne National Laboratory, Argonne, IL 60439, (gropp@mcs.anl.gov).

tributed lock algorithm using the remote atomic operations our NIC-based implementation gave up to a 62.3% improvement over the 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. In Section II we describe the basic concept of the NIC-based atomic remote memory operation. Section III describes the implementation of the NIC-based atomic remote memory operation, and Section IV describes how to implement distributed locks using these operations. In Section V we present our experimental results, and conclude in Section VI.

II. 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 atomically on the memory at the host. 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 are performed by the NIC, not the host. Figure 1(a) shows an example of an atomic operation being performed by a thread running on the host processor (host-based atomic operation) and Figure 1(b) shows an example of an atomic operation being performed by the NIC (NIC-based atomic operation).

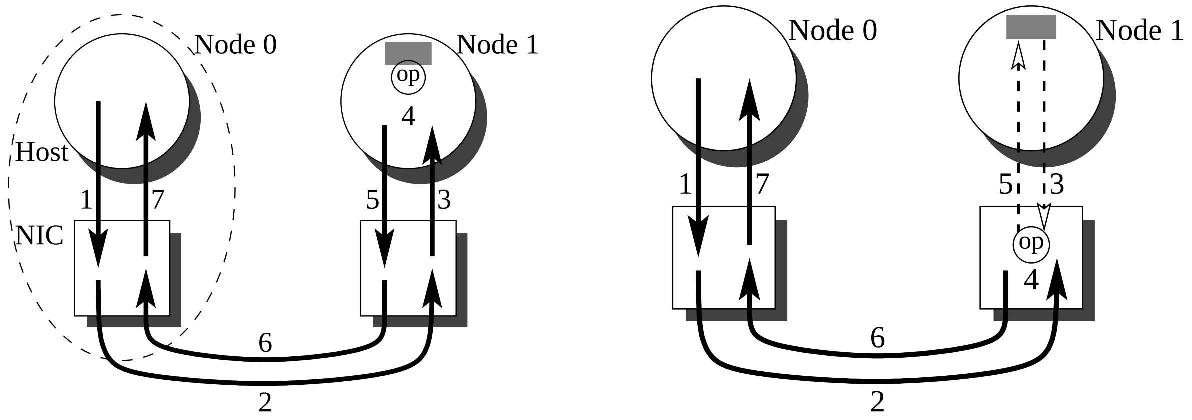
In order to perform an atomic operation on a remote memory region without using NIC-based atomic operations, the remote node would need to have a thread which receives the requests, performs the operations and returns the result. This is shown in Figure 1(a). Here, an application at node 0 sends a request for an atomic operation to a thread 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 thread which is handling the atomic operations.
4. Upon receiving the message, the thread at node 1 performs the operation specified in the message on the host's memory.
5. This thread then sends a reply message to the NIC.
6. The NIC at node 1 then 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 thread 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. This operations is performed as follows:

1. An application at host 0 sends a special atomic operation message to the NIC.
2. This NIC 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.
5. If necessary, the NIC copies the new value back to the memory location, again using DMA.
6. The NIC transmits the result back to the NIC at node 0. Note that this step can be performed concurrently with the previous step.
7. The NIC at node 0, upon receiving the result message, forwards it to the application.

The main advantage of using the NIC-based approach is that the operation can be performed without the intervention of a host thread. With the host-based approach, the atomic operation requests need to be handled at the host. This can be done by having the main application periodically poll for these messages, however, this can lead to poor response time for the operation if the main application polls infrequently. Another option for the host-based approach is to have a sepa-



(a) Host-based

(b) NIC-based

Fig. 1. Steps required to perform host-based and NIC-based atomic remote memory operations. See text for a detailed description of the steps in Section II.

rate *server thread* to handle these requests. When using a server thread, unless a separate processor at the host can be dedicated to this thread, the thread should block while waiting for the requests. The main thread is then interrupted when an incoming request is received so that the server thread can process the request. When the server thread and main thread share a CPU, blocking the server thread while it is idle will lead to better utilization of the CPU by the main thread. However, when there are many such requests, the repeated interrupts can lead to poor performance of the main application. By using the NIC-based approach atomic operations can be performed without interfering with any processes at the host. The application process can be running on the host CPU, while the NIC is performing the atomic operations directly on host memory.

Just about any atomic operation can be implemented using this scheme. The only constraint is the processing power of the NIC processor. Typically, the NIC processor is much slower than the host processor. For instance, the LANai processors on the Myrinet NICs range from 33MHz to 200MHz, while host processors may range from 300MHz to 2GHz. Furthermore, NIC processors may not have floating point units, so any floating point operation would have to be simulated using integer operations. For this reason it would probably not be beneficial to perform complex operations. Another constraint is the NIC processor's access to the host memory. Most NICs do not support PIO access to the host memory from the NIC. Rather any transfers of data from host memory initiated by the NIC must be done using DMA. While DMA performs well for transferring large data, there is an overhead to setting up the DMA which makes it less efficient for performing small data transfers. So the number of data transfers between the NIC and host memory should be limited.

We implemented the following 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 I 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 *data* value in the table represents the data that is to be written to the memory location and the *compare* value in the table represents what the value stored in the target memory location is compared to.

III. 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[3] version 1.5. We will first give a brief

TABLE I
Semantics of atomic memory operations

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

overview of Myrinet[4] and GM, then describe our implementation.

A. Overview of Myrinet and GM

Myrinet is a low latency, high bandwidth, wormhole routed network. The links are full-duplex and have either 1.28+1.28 gigabits per second or 2+2 gigabits per second link rate. The newer NICs and switches provide the 2+2 Gbps link rate.

The Myrinet NIC consists of a programmable *LANai* processor, memory, one DMA engine for transferring data between the NIC memory and the host memory, one DMA engine for transmitting data from the NIC memory and the network and another DMA engine for receiving data from the network to the NIC memory. Depending on the revision of the card, the LANai processor runs at either 33, 66, 133, or 200MHz, and has between 1 and 4 MB of SRAM. The programmable processor runs a control program which allows host processes to directly interact with the NIC bypassing the operating system (OS-bypass) for low latency communication.

GM is a user-level message passing system that uses the Myrinet network. GM consists of a kernel module, a library and a Myrinet control program (MCP). The driver loads the MCP on to the NIC when it is loaded. During the execution of a program, the driver is used mainly for opening *ports*, pinning and unpinning memory, and to put a process to sleep for blocking functions. A *port* is a data structure through which a process can communicate with the NIC. Once a port is opened, the process can communicate with the NIC, bypassing the operating system and avoiding system call overhead.

Figure 2 is a block diagram of GM where a process has two ports through which send tokens and receive tokens are transferred to and from the MCP without going through the kernel. The figure also shows DMA operations which transfer data directly to and from memory regions of the process.

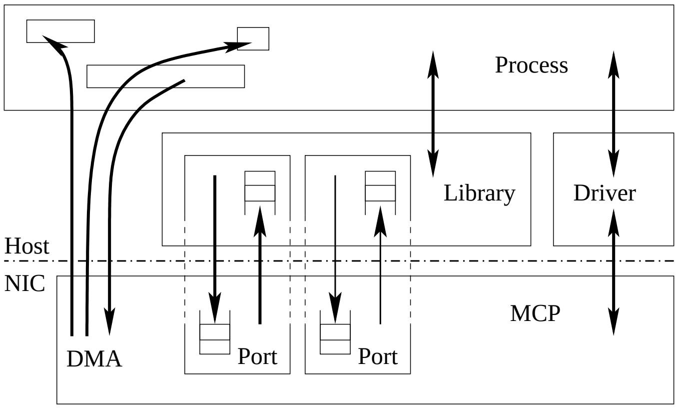


Fig. 2. Block diagram showing the components of GM.

At the host level GM is connectionless, but it provides reliability by maintaining reliable connections between NICs of different nodes. When a packet is sent by the NIC, the NIC keeps a *send record* with enough information to reconstruct the packet. The send record also has a timestamp of when the packet was sent. Until the packet is acknowledged, the NIC checks the timestamp of each send record to determine if a packet needs to be resent. If a packet times-out, using the information in the send record, the NIC DMAs the data for the packet again from host memory, reconstructs the packet and transmits it. Upon receiving an acknowledgment for a packet, the corresponding send record is deleted. Packets also have sequence numbers which are used to ensure correct packet ordering. If a packet is received out of order, or if a duplicate packet is received, it is dropped and an acknowledgment packet is sent re-acknowledging the last packet correctly received.

Flow control is used between the NIC and the host to avoid buffer overflows. To provide this flow control GM uses the concept of tokens. When a process opens a port, it has a certain number of *send tokens* and *receive tokens*. Each send token corresponds to a send event. For sending a message the process fills-in a send token describing the send event and passes it to the NIC. The message may consist of several packets. The NIC takes care of packetizing the data. Once the NIC has finished sending the message and all of the packets have been acknowledged, the NIC returns the send token to the process in a callback function.

Data can only be sent from or received into pinned memory regions. This is necessary so that the pages that contain the data are not paged out by the operating system while the data is being transmitted by the NIC. GM provides special functions which pin memory and inform the NIC of the physical address and virtual address of the pages to be used for address translation when DMAing the data. This is known as *registering* memory.

In order to send or receive a message, the process must pass a receive token describing the buffer to the NIC. Once the NIC has DMAed the data into the buffer, the receive token is returned to the process. The process can either poll to detect returned receive tokens, or block and wait for the receive tokens.

B. Design Challenges and Our Implementation

We added two functions to the GM API, and modified the MCP. The `gm_provide_atomic_buffer()` function passes a receive token to the NIC. This receive token will be used by the NIC to pass the return value of an atomic operation to the application. The application must ensure that the NIC has sufficient receive tokens to receive the return values for any atomic remote memory operation. The `gm_atomic_send_with_callback()` function builds a send token and passes it to the NIC initiating the atomic remote memory operation. The send token describes the atomic operation and includes all of the necessary parameters plus a *tag* value which the application will use to match the atomic operation request with the return value. The application checks for the completion of the operation using the `gm_receive()` function or one of its variants. When the operation is completed, the `gm_receive()` function returns a receive token with the return value of the operation along with the tag value which was provided in the corresponding call to `gm_atomic_send_with_callback()`.

When the NIC receives the send event, it transmits an *atomic operation packet* to the destination node with all relevant parameters. Upon receiving the packet, the NIC at the destination node checks for packet corruption and correct packet sequence, and when the DMA engine to the host is free, performs the operation.

The operation is performed in the following manner. The NIC DMAs the data from the target host memory location to a temporary location. The NIC then calculates the new value of the target memory location (as described in Table I), and DMAs the new value to the host memory. Because 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. The reply packet also serves as an acknowledgment for the atomic operation packet. Upon receiving the reply packet, the NIC checks for packet corruption, processes the acknowledgment and sends a *receive token* to the application. The NIC performs each of the operations using one or two DMAs. In order to ensure that the operations are atomic, the process must not directly modify any memory region onto which an atomic operation may be performed. Instead, the process should issue the operations to the NIC.

In our implementation, we had to address the following challenges: how to inform the NIC that a particular memory region should be used for atomic operations, how to notify the calling process of the return value, and how to provide reliability.

We addressed the first challenge of how to inform the NIC that a particular memory region should be used for atomic operations by using the same method that GM uses to provide *directed sends*. In GM, directed sends are messages where the data is written directly to the receiver's memory without the receiver calling `gm_receive()`. The sender of the message provides the address of the buffer at the receiver. This type of communication is sometimes called RMA (for remote memory access) or RDMA (for remote direct memory access). With directed sends, the sender can specify any registered memory region at the receiver as the destination address. We used the same idea. Atomic operations can be performed on any 32-bit word in any registered memory region at the target node.

The number of memory locations that can be used by the atomic remote memory operations is limited only by the amount of memory that a process can register in GM, which, for GM version 1.5 on Linux, is 7/8 of the physical memory of the host. We can specify a remote memory location by a triple: a node id, to identify a particular machine on the network; a port id, to identify a particular process on that machine; and the virtual address of the memory location in the address space of that process. The method of distributing this triple to other nodes in the system is left up to the programmer (e.g., by simply sending the triple in a message).

The second challenge was how to notify the calling process of the return value. Atomic remote memory operations produce a return value which the calling process must receive. We needed a mechanism by which the calling process can receive the value. Since the receive token had space for small message data to be sent to the host, we decided to use the receive token to provide the return value to the process. When the NIC receives the return value from the NIC at the remote node, it passes the process a receive token containing the return value. In order for the calling process to match the call to the atomic operation with the return value, the process specifies a *tag* value when the operation is initiated which is then included in the receive token along with the return value.

The third implementation challenge was to provide reliability for the atomic operations messages. To do this, we used mechanisms similar to the ones used by GM with two differences. First the reply packet doubles as an acknowledgment packet to the atomic operation packet, so a separate packet is not needed. Second, we handle duplicate atomic operation packets differently. A duplicate atomic operation packet cannot be dropped because the initiating node needs the return value of the atomic operation. Furthermore, because the operations are not idempotent we cannot repeat the operation to get the return value. Instead, the NIC keeps a table of return values and sequence numbers. Upon receiving a duplicate atomic operation packet, the NIC looks up the return value and re-sends a reply packet. Because there are a limited number of entries in the table of return values, each NIC must limit the number of unacknowledged atomic operations it sends, otherwise it is possible that some of the return values for outstanding packets will not be stored. This method of using a table to record return values is similar to the one described in the InfiniBand Architecture standard[2].

IV. Implementing Distributed Locks with Atomic Remote Operations

One use of remote atomic memory operations is in distributed locks. We implemented a software queuing lock using atomic remote memory operations similar to the MCS[11] lock. The MCS lock is intended for shared memory machines, but we extended the idea for distributed memory machines using atomic remote memory operations. The algorithm creates a distributed linked list of processes waiting for the lock. The process at the head of the queue holds the lock. In this algorithm, each process has a `next` variable which points to the process which has requested the lock immediately after this process, and a boolean `locked` variable which indicates whether the node is waiting for the lock. These two variables should be stored so that atomic remote memory operations can be performed on them. The `lock` itself is a variable which points to the last node to request the lock. The `lock` variable is stored at the *home node* of the lock.

When a process, p, requests a lock, it first sets its `next` variable to NIL. Next, it performs a fetch&write(p) operation on the `lock` variable to determine which process is currently last on the queue (i.e., its *predecessor*. If the queue is empty (i.e., the predecessor is NIL), then this node has acquired the lock. Otherwise, it sets its own `locked` variable to true, then performs a remote write to write p to its predecessor's `next` variable, thereby inserting itself in the queue. It then polls on its own `locked` variable until it becomes false.

To release a lock, a process, p, first checks if its own `next` variable is NIL. If it is not, then it performs a remote write operation on its *successor's* locked variable setting it to false thereby successfully releasing the lock. Otherwise, it performs a compare&swap(NIL, p) operation on the lock variable. If this succeeds, (i.e., operation wrote a NIL to the locked variable) then process p was the last node on the queue, and has successfully released the lock. If the operation failed, this indicates that another process has begun requesting the lock, and has updated the lock variable, but has not yet updated process p's next variable. Process p should then poll on its next variable until that process updates it, at which point process p should perform a remote write setting the locked variable at that process to false.

Figure 3 gives an example of how the lock algorithm works. In the figure, the circle with the L in it represents the lock variable stored at the home node. The boxes with the numbers in them represent the processes requesting the lock. The arrows coming out of the boxes represent the next variable, and the squares in the boxes represent the boolean locked variable. A filled in square indicates that locked is set to true and that the process is waiting on the lock. Step (a) shows the initial state where there are no processes requesting the lock. Step (b) shows the state after process 1 acquires the lock. In (c) we see the state after processes 2 and 3 have requested the lock, but before process 1 has released the lock. Notice that the locked variables for processes 2 and 3 are shown as true. When process 1 releases the lock, it will notice that its next variable points to process 2. It will then change process 2's locked variable to false, so that process 2 can acquire the lock, as shown in step (d). Step (e) shows the state where only process 3 is left in the queue and has acquired the lock. If process 3 releases the lock before another process requests it, the lock variable will be set to NIL, and the state will be the same as in step (a).

Because we implemented the lock algorithm for distributed memory, we cannot use simple memory pointers for the next and lock variables as used in the original MCS algorithm. As we described in Section III-B, a remote memory location is specified as a triple of the node id, port id, and virtual memory address. Instead of using memory pointers, our lock implementation uses process ranks. Each process then has an array where the ith el-

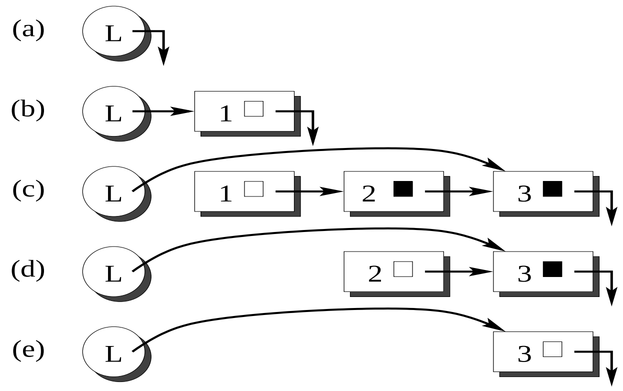


Fig. 3. Example of a distributed lock

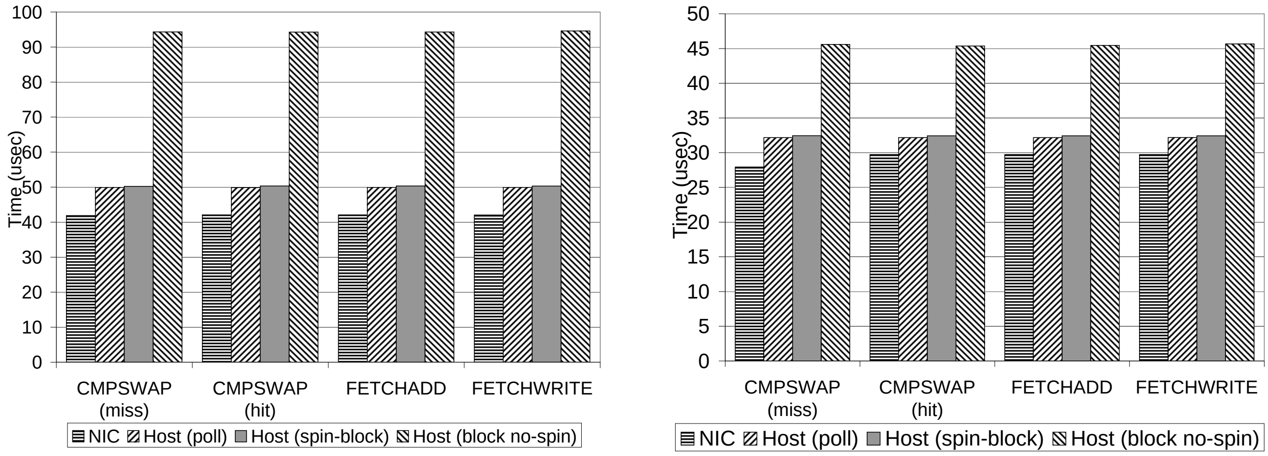
ement stores the remote memory triple describing the location of the ith process' next and locked variables. For example, when a process releases a lock, and reads a value of 4 in its next variable, it gets the remote address triple for its successors locked variable by looking up the 4th entry in the array.

V. 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[4] 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. These machines are connected by another Myrinet LAN network using NICs with 66MHz LANai 7.2 processors. These are connected to an eight port switch.

[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

Fig. 4. Latencies of atomic operations

A. Atomic Remote Memory Operations

We tested the performance of the NIC-based atomic operations and compared it to host-based implementations. A host-based test consists of a process which sends atomic operation request messages to a *server* process on another node. The server process receives the request messages, performs the operations and returns reply messages. Because a host-based implementation would most likely to be on a separate thread which would be interrupt driven, we tested three different methods that the server could use 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 immediately 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.

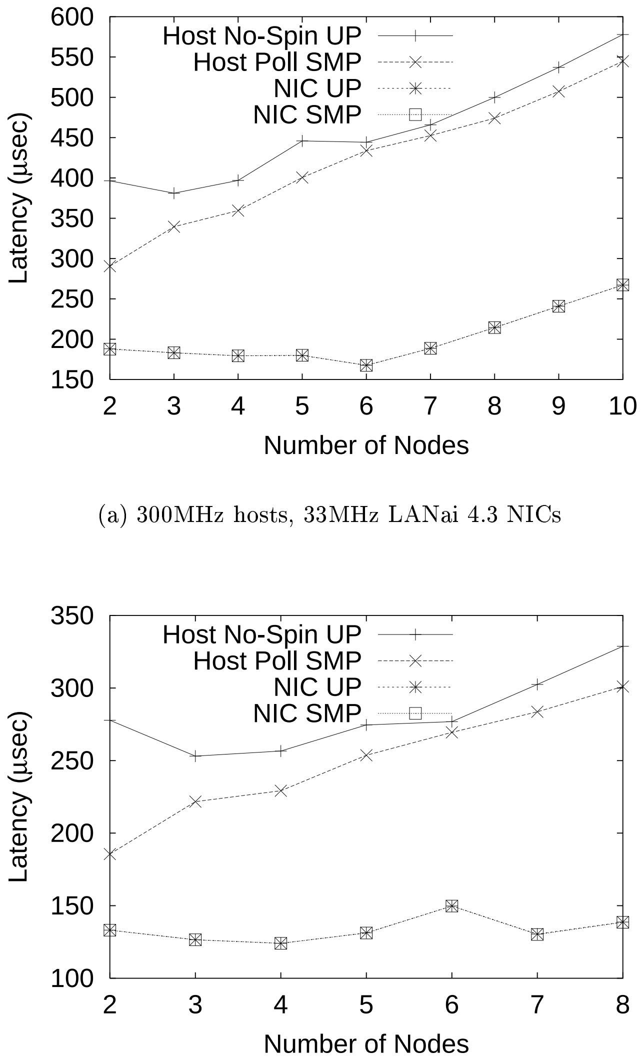
The tests consist of taking the average time of 10,000 iterations of each atomic operation. The compare&swap operation was evaluated in two ways, once where the compare failed (*miss*) so that the swap was not performed, and once there the compare succeeded (*hit*). Figures 4(a) and 4(b) show the results of these tests for 300MHz host processors with 33MHz LANai 4.3 NICs, and 700MHz host processors with 66MHz LANai 7.2 NICs, respectively. 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 42.1μs as compared to the best host-based implementation, polling, which took an average of 49.9μs. This is a 15.7% improvement. The spin-block host-based implementation took an average of 50.2μs. The NIC-based implementation showed a 16.1% improvement over this host-based implementation. The blocking-no-spin host-based implementation took an average of 94.3μs. The NIC-based implementation showed a 55.4% improvement over this host-based implementation. When we used the 700MHz hosts with the 66MHz LANai 7.3 NICs the NIC-based compare&swap (hit) operation took an average of 29.7μs while the host-based polling implementation took 32.2μs, the host-based spin-block took 32.4μs and the host-based blocking-no-spin implementation took 45.4μs. This is a 7.6%, 8.3% and 34.5% improvement respectively.

One should note, that these tests represent the best-case configurations. At each node there is only one process running. In this situation, for the host-based implementation, the polling and spin-block versions of the server thread perform much better than the block-no-spin version. However, these versions would typically not be used in a situation where the server thread was sharing a single processor with another thread. In such a situation, if a server thread uses polling receives, the CPU will be under-utilized whenever the server thread is scheduled and no messages are coming in. Using blocking-no-spin receives releases the CPU when no messages are waiting to be received, and will not schedule the process until a message comes in. This leads to better performance of the main thread because it get scheduled more often. Using spin-block receives works in a similar manner as using blocking-no-spin receives, except that when there are no messages to receive, the operation polls for a while before blocking. This works well when incoming messages are bursty, so that many messages can be handled with one interrupt. Otherwise, if no new message is received, the time that the server thread spends polling is wasted. This again would lead to poor main thread performance.

B. *Distributed Locks*

The lock test for our host-based implementation consists of the *main thread*, a *server thread* and a shared memory region which both threads can read from and write to. The *main thread* requests and releases locks by sending remote atomic operation requests to the *server threads* at the target node. The server thread performs the operations on the target memory location in the shared memory region. The NIC-based implementation consists of just a single thread which requests and releases locks using NIC-based remote atomic memory operations.

In this test we took the average time it takes for one process at one particular node to repeatedly acquire and release a remote lock. To vary the load, we added more nodes also repeatedly locking and unlocking the same lock. The tests were run using both an SMP enabled kernel, so that one CPU was available for each thread, and using a uniprocessor kernel (UP), in which case both threads shared the same processor. For the tests run on the SMP kernel, the server thread for the host-based implementation used polling receives, since this performed better than the other receive methods when there was no other thread contending for the CPU. While for the tests run on the uniprocessor kernel, the server thread used polling-no-spin receives, because this performed



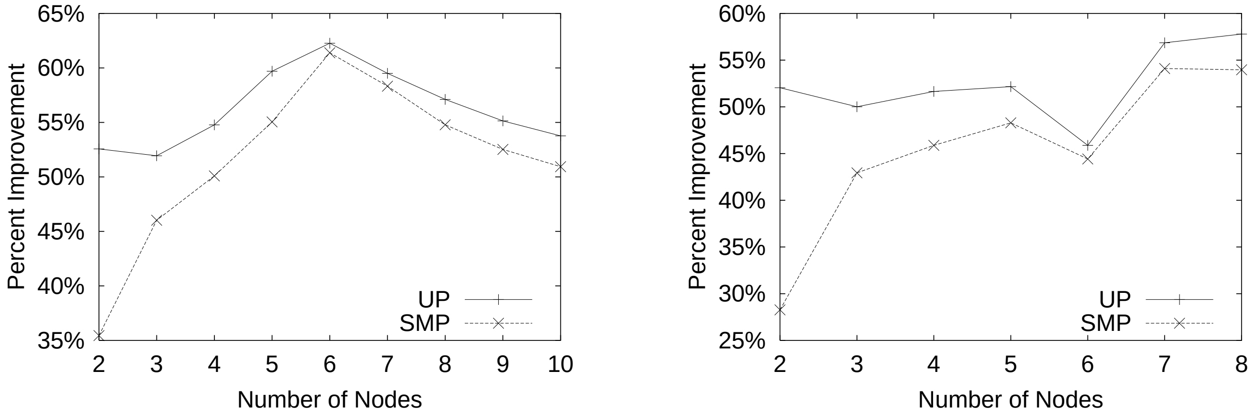
(a) 300MHz hosts, 33MHz LANai 4.3 NICs

(b) 700MHz hosts, 66MHz LANai 7.2 NICs

Fig. 5. Latency of locking and unlocking with multiple nodes contending

better than the other receive methods when the server thread was sharing the processor with the main thread.

Figures 5(a) and 5(b) show the results of this test. We show the results for the host-based blocking-no-spin using the uniprocessor (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. Notice also that because there is only one process necessary for the NIC-based implementation, the NIC-based implementation gives the same performance



(a) 300MHz hosts, 33MHz LANai 4.3 NICs

(b) 700MHz hosts, 66MHz LANai 7.2 NICs

Fig. 6. Percentage improvement of lock benchmark

using either kernel. As expected the host-based polling implementation on the SMP kernel outperforms the host-based blocking implementation using the uniprocessor kernel because of the lack of context switching overhead in the SMP case.

Figures 6(a) and 6(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.

C. *CPU and Network Utilization*

To test the impact of using the atomic operations on CPU utilization we performed a test where a process (the *counter process*) at 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 counter process is doing that a process at another node repeatedly locks and unlocks the lock. We inserted a delay just after the lock operation and another just after the unlock operation. By varying these delays, we can alter the rate at which the lock-unlocks are performed. We also performed the test where no lock or unlock operations were performed to serve as a baseline (*idle case*). When running a uniprocessor kernel, the number of iterations that the counter process is able to perform in the alloted time gives an indication of the amount of CPU time that is used in processing incoming atomic operation requests. The more time that the CPU is spending processing incoming requests, the fewer iterations the counter process is able to complete. In this test we used the uniprocessor (UP) kernel and the blocking-no-spin version of the server thread for the host-based implementation. This way, when atomic operation requests are received, the counter process will be interrupted by the server process, and the impact can be measured.

Figure 7 shows the results of this test. The figure shows the number of iterations that the counter process was able to complete while lock-unlock operations were happening at a certain rate. As expected, for both the 700MHz and 300MHz machines, the NIC-based implementation is unaffected by the number of atomic operations being performed, because the operations are performed completely by the NIC and require no intervention of the host processor. However, for the host-based implementation we see that the CPU utilization of the counter thread at the home node decreases as the number of atomic operations the server node is processing increases. In the host-based implementation, the main thread must be interrupted so that the server thread can process the incoming request leading to decreased CPU utilization by the counter process.

We also evaluated the impact of handling atomic remote memory operations on communi-

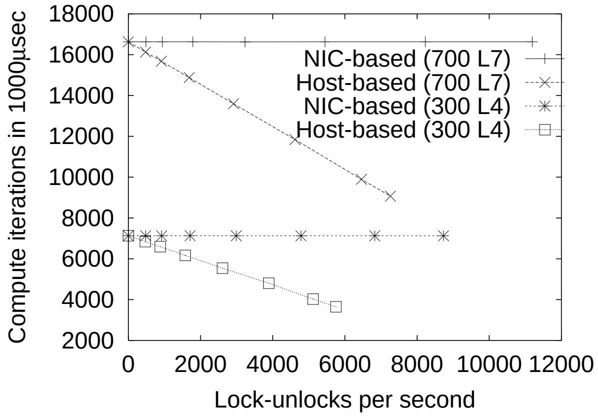


Fig. 7. The number of iterations a process at the home node of a lock can perform in 1,000μs while a process at another node is locking and unlocking the lock at a certain rate for 300MHz hosts with 33MHz LANai 4.3 NICs (300 L4), and 700MHz hosts with 66MHz LANai 7.2 NICs (700 L7).

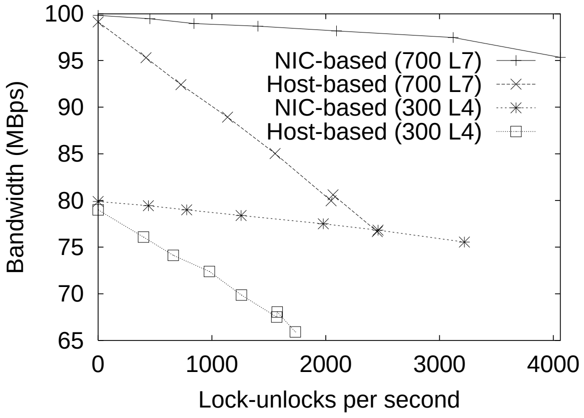


Fig. 8. The send bandwidth achievable by a process at the home node of a lock while a process at another node is locking and unlocking the lock at a certain rate for 300MHz hosts with 33MHz LANai 4.3 NICs (300 L4), and 700MHz hosts with 66MHz LANai 7.2 NICs (700 L7).

cation bandwidth. This test is similar to the CPU utilization test except that the process at the lock home node is performing a bandwidth test with a process at a third node. The process at the lock home node measures the bandwidth by timing how long it takes to send 1,000 16MB messages to the process at the third node and receive a reply from it. Figure 8 shows the bandwidth achievable by the process at the home node in the presence of lock-unlock operations. Notice that for both clusters, the impact of handling the atomic operations is greater on the host-based implementation than on the NIC-based implementation. In both NIC-based and host-based 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 bandwidth process can pass the send commands to the NIC.

VI. Conclusions

In this paper, we presented an implementation of NIC-based atomic remote memory operations. We added support for atomic remote memory 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 15.7% improvement for the compare&swap operation when comparing the NIC-based implementation to the best host-based 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 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.

Additional Information

Additional papers related to this research can be obtained from the following Web pages: Network-Based Computing Laboratory (http://nowlab.cis.ohio-state.edu) and Parallel Architec-

ture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html). If you are interested in using this software, please contact Dr. D. K. Panda at panda@cis.ohio-state.edu.

References

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

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

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

[4] 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.

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

[6] R. Kesavan and D. K. Panda, "Optimal Multicast with Packetization and Network Interface Support," in *Proceedings of International Conference on Parallel Processing*, Aug 1997, pp. 370–377.

[7] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal, "Efficient Multicast on Myrinet Using Link-Level Flow Control," in *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, August 1998, pp. 381–390.

[8] 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.

[9] 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.

[10] D. Buntinas, D. K. Panda, and P. Sadayappan, "Performance benefits of NIC-based barrier on Myrinet/GM," in *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[11] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.