NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters *

Adam Wagner[1] Hyun-Wook Jin[1]
Rolf Riesen[2] Dhabaleswar K. Panda[1]

[1]Network-Based Computing Laboratory
Dept. of Computer Science and Engineering
The Ohio State University
{wagnera, panda}@cis.ohio-state.edu

[2]Scalable Computing Systems Dept.
Sandia National Laboratories
rolf@sandia.gov

*This research is supported in part by a grant from Sandia National Laboratory (a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC004-94AL85000.), Department of Energy's Grant #DE=FC02-01ER25506, and National Science Foundation's grants #EIA-9986052 and #CCR-0204429.

// Abstract

Many of the modern networks used to interconnect nodes in cluster-based computing systems provide network interface cards (NICs) that offer programmable processors. Substantial research has been done with the focus of offloading processing from the host to the NIC processor. However, the research has primarily focused on the static offload of specific features to the NIC, mainly to support the optimization of common collective and synchronization-based communications. In this paper, we describe the design and implementation of a new framework based on MPICH-GM to support the dynamic NIC-based offload of user-defined modules for Myrinet clusters. We evaluate our implementation on a 16-node cluster using a NIC-based version of the common broadcast operation and we find a maximum factor of improvement of nearly 1.3 for our implementation versus the standard host-based implementation of broadcast. In addition, we see that this factor of improvement increases with system size, indicating that our implementation is more scalable than the default host-based approach.

1 Introduction

Many of the the interconnection networks used in current cluster-based computing systems include network interface cards (NICs) with programmable processors. Much research has been done toward utilizing these CPUs to provide various benefits by offloading processing from the host. These works have mainly focused on customizations to enhance the performance of specific operations including collective communications [5, 10] like multicast [2] and reduce [13] and synchronization operations such as barrier [3]. The common approach is to hard-code the optimization into the control program which runs on the NIC in order to achieve the highest possible performance gain.

While such approaches have proved successful in improving performance, they suffer from several drawbacks. First, NIC-based coding is much more complex and error prone due

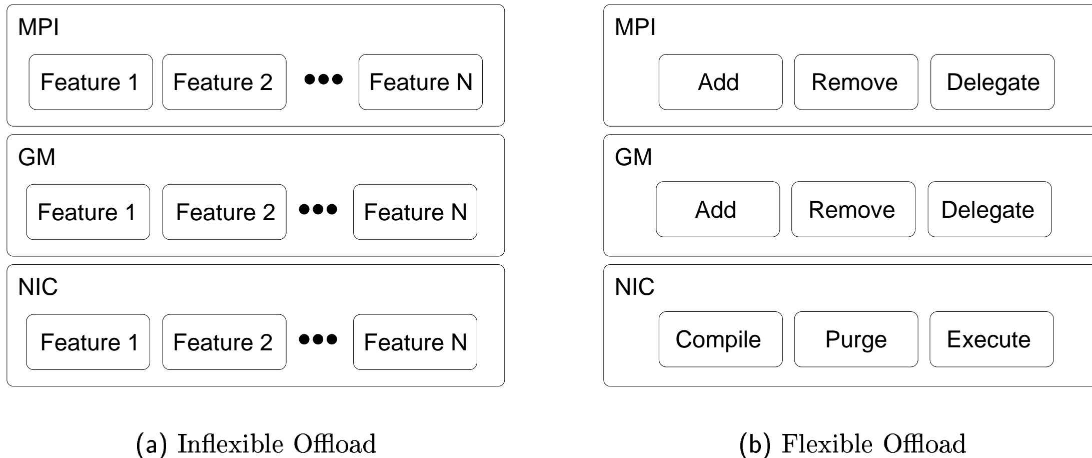


Figure 1: Inflexible, hard-coded, ad-hoc offload of features vs. flexible framework for offloading user modules to NIC.

to the specialized nature of the NIC firmware and the difficulties introduced by validating and debugging code on the NIC. Because of the level of difficulty involved in making such changes and the potential consequences of erroneous code, these sorts of optimizations may only be performed by system experts. Second, hard-coding features into the NIC firmware is inflexible. The resources available on the NIC are typically an order of magnitude less than those on the host. This means that only a limited number of features may be compiled into the firmware at a given time. Furthermore, frequent changes may be impractical on production systems which demand high levels of stability, availability and security.

Figure 1 illustrates the difference between a hard-coded approach and a more flexible approach to NIC-based offload. We can see that in the hard-coded approach, we are limited to a fixed number of features, while in the flexible approach features may be added and removed as needed. When a feature is added it is propagated down through the software layers to the NIC, where it is compiled and stored for later use. The upper layers may then delegate tasks down to the NIC for execution and incoming messages may be handled by the code on the NIC without host involvement. When a feature is no longer needed, it may be purged from the NIC to free up resources for other uses.

This paper describes our design and implementation of a new framework to support the offload of user code to the NIC in Myrinet [1] clusters. Our approach addresses many of the negative aspects associated with hard-coding features into the NIC. We accomplish this by introducing a flexible framework which we refer to as NICVM. This framework allows users to dynamically add and remove code modules from the NIC. The code is added by the user in source form and compiled into an intermediate format which is later interpreted by a special-purpose virtual machine embedded in the NIC firmware. By interpreting the code we have the benefit of complete control, and perhaps counter-intuitively we can still realize the performance benefits associated with offload. We have implemented the common broadcast operation on the NIC as a user module and observe a maximum factor of improvement of nearly 1.3 over a similar host-based implementation for 16 nodes. Furthermore, we observe that these performance benefits increase with system size.

The remainder of this paper is organized as follows. In the next section, we provide a brief overview of Myrinet as well as the GM [14] and MPICH [9] software packages on which our framework is based. In section 3 we discuss the design challenges we encountered while implementing our framework. In section 4 we detail our implementation and in section 5 we evaluate its performance. We describe related work in section 6 and present our conclusions in section 7.

2 Background

Myrinet [1] is a popular low-latency, high-bandwidth interconnection network that employs programmable network interface cards (NICs), cut-through crossbar switches and operating-system-bypass techniques to achieve full-duplex 2 Gbps data rates. GM [14] is a user-level message-passing subsystem for Myrinet networks. GM consists of a lightweight kernel-space driver, a user-space library and a control program (MCP) which executes on the NIC processor. The kernel-space code is only used for housekeeping purposes like allocating and

registering memory. After taking care of such initialization tasks, the user-space library can communicate directly with the NIC-based control program, removing the operating system from the critical path.

MPI [12] is a standard interface for message passing in parallel programs. MPICH [9] is the reference implementation of MPI and has been ported to a variety of hardware platforms including GM over Myrinet. Our framework is built upon this version of MPICH (named MPICH-GM) and includes modifications to all software layers including the MPICH and GM libraries as well as the MCP.

3 Design Challenges

This section discusses the design challenges we encountered while implementing our NICVM framework. The specifics regarding our solutions to each issue will be addressed in detail in the next section.

3.1 Performance of User Code

One of our main challenges was designing the framework so that the user code could be efficiently executed. There are two different areas where performance of user code is critical. The first is the base latency required to activate a given user module on the NIC. The second is the actual time required to execute a given module of user-code. If the base latency is too high, then performance will be poor regardless of the time taken to perform the actual work associated with the module. Such startup latencies could easily outweigh the positive effect of offload-related benefits like avoiding PCI bus traffic. Of course, the complete time taken to execute the user code is important as well. The MCP is structured as a state machine with different states for sending, receiving and performing DMAs to and from host memory. The transitions between states are well tuned and adding any extra delay to process user code can have a negative impact on overall performance.

3.2 Avoiding Common-Case Impact and Interference

Another challenge involved avoiding performance impact to the common case of non-NICVM message traffic. If we were to add our support for NIC-based execution of user code in a manner that caused the basic GM or MPI message latency to increase significantly, then the end result would not be of much practical use. This issue was further complicated by the fact that GM's send and receive queues and associated flow control mechanisms are tightly shared between the host and the NIC. Our design strategy needed to include measures to avoid interference between host-based and NIC-based sends and to accommodate the fact that NIC-based sends happen asynchronously with respect to the host. At the other extreme, we needed to consider situations where the host application simply exits after loading the handler on the NIC so there are no host resources available. This could occur, for example, in the case of a NIC-based intrusion-detection code, which just needs to be loaded to the NIC and then requires no further host involvement.

3.3 Support for Multiple Reliable NIC-Based Sends

Providing an infrastructure to allow user code to initiate multiple reliable NIC-based sends proved to be another challenge. It's relatively straightforward to initiate a send from the NIC, especially if reliability is not a requirement. However, we imagined that a common scenario for user modules would be to intercept a message before involving the host and perform several sends to other nodes. Note that we wanted to avoid memory copies on the NIC, which would be prohibitively slow and would introduce scalability issues due to the lack of available NIC memory. So we needed to come up with a scheme that would support re-use of a given chunk of NIC-based memory for multiple sends and that would maintain the data associated with a given send until that send was verified complete, thus providing reliability. A related issue involved support for user modules that involve both performing sends as well as transferring a message to the host via RDMA. The easiest solution would be to allow the

host RDMA to complete and then perform the NIC-based sends. However, it would be more efficient in many cases to initiate the NIC-based sends first and then perform the RDMA to the host later outside of the critical communication path. This sort of behavior would be especially beneficial for collective-style communications.

3.4 Environmental Constraints on the NIC

When investigating the potential use of existing software packages on the NIC, we were faced with the challenge of adapting to the severely resource-constrained NIC environment. At 133-MHz and with 2-Mb of RAM, the Myrinet NICs which we used were an order of magnitude slower than the average host and contained an order of magnitude less memory. Furthermore, the NIC environment does not include many of the standard programming utilities which are taken for granted in host-based development. For instance, there is no dynamic memory allocation, C standard library routines or file system. The majority of the software packages that we initially evaluated were not sufficiently portable due to reliance on such features.

3.5 Security Concerns

Several security-related concerns also arise at the prospect of executing user code on the NIC. For example, should only the local host be able to upload code to the NIC or should it be acceptable for a remote host to do so? What happens if the user uploads code that contains an infinite loop or if a remote node sends a packet containing data that has a similar effect? Can the user execute arbitrary instructions on the NIC that might disable the NIC or allow access to memory regions belonging to other users? While we haven't addressed all of these challenges in our current implementation, they proved to be factors that influenced the decisions made in the overall design of our framework. We intend to further investigate these issues in the future.

4 Our Implementation

In this section we present the details of the implementation of our NICVM framework. We start with a high-level overview and then take a bottom-up approach to describing the details of the different framework components.

4.1 Overview

To get a high-level feel for the different components of the framework and how they fit together, let's start with an example. Our framework is basically a customized version of MPICH-GM. Assume that we wish to prototype a new NIC-based feature. To match with the experiments presented later, assume that this feature is a NIC-based broadcast.

In order to perform such a task without using our framework, we would need to take the following sort of steps. First we would need to locate the source code for the MCP and craft our broadcast code into the MCP source. Without extensive experience, modifying the MCP is a difficult and error-prone process, as the code is highly optimized and quite complex. Then we would also need to, at a minimum, modify the MPICH library source code to either add a new broadcast API routine or modify the functionality of the existing routine. We would also most likely need to make modifications to the source for the GM library to support our changes to the MPI layer. Finally, after rebuilding and installing MPICH-GM, we could write an MPI program to call the new or modified broadcast routine and test it on the cluster.

Contrast this to the work required if we were to use our NICVM framework. We would actually only need to do two things. First, we would create a source code module in an easy to understand language which is similar to Pascal and C. This module would implement the logic that we wish to offload to the NIC. Assuming we want to implement our broadcast with a binary tree, the module would contain logic to initiate two sends to the appropriate child nodes upon receiving a broadcast message. Figure 2 illustrates such a module, which

```
func main()

  var size;
  var rank;
  var result;
  var child;

  size = mpi_get_size();
  rank = mpi_get_rank();
  result = NICVM_RESULT_SUCCESS;

  child = ((rank + 1) * 2) - 1;
  if (child < size) then
    if (rank == mpi_get_root()) then
      result = NICVM_RESULT_PACKET_CONSUMED;
    end if;

    mpi_send_message(GM_ST_RELIABLE_NICVM_NIC_DATA, child);

    child = child + 1;
    if (child < size) then
      mpi_send_message(GM_ST_RELIABLE_NICVM_NIC_DATA, child);
    end if;
  end if;

  return result;
end func;
```

Figure 2: Example user module for implementation of NIC-based broadcast via logical binary tree. First the rank of the left child is calculated. If valid, a send is initiated to the child. Then the same steps are taken for the right child. If the current node is the root, the module reports that the packet has been consumed to inform the MCP that no further action is necessary.

we actually used in our experiments. We would then write an MPI program in which all nodes first call an API routine to upload the source code module to the NIC. After this initialization phase the root node would call an API routine to delegate an outgoing message to the NIC-based module, while the other nodes would simply perform a receive.

At run time, the initialization phase would cause our NIC-based broadcast module to be dynamically compiled into a virtual machine running on the NIC. Upon delegation by the root node, the root node's NIC would hand off the outgoing message to the NIC-based virtual machine which would activate the broadcast module. The broadcast module would then initiate sends to the root's children. Upon receiving the message from the root, the NIC at each child would behaving similarly, handing off the incoming message to the broadcast module before involving the host. After completing the sends initiated by the broadcast module, our framework would DMA the broadcast message to the host, thus finishing the broadcast. Note that this approach does not require any modifications to the underlying components or disturbance of the cluster environment.

We can see that the main components involved our framework are are the MPICH and GM libraries, the MCP and our NIC-based virtual machine. Figure 3 details the different API routines associated with each component and how each component fits into the overall framework.

4.2 Virtual Machine

We originally began our research using a Forth interpreter named pForth [4]. This was highly portable and extensible and was invaluable in our initial proof of concept implementation. However, we decided to write a custom interpreter for two reasons. First, pForth is a general purpose interpreter for the Forth language, which is fairly extensive. Accordingly, we were unable to achieve the low latency required for our specialized NIC-based implementation. Second, the Forth language is stack-based and significantly different than what most C or Fortran programmers are use to working with. We felt that a more familiar syntax would

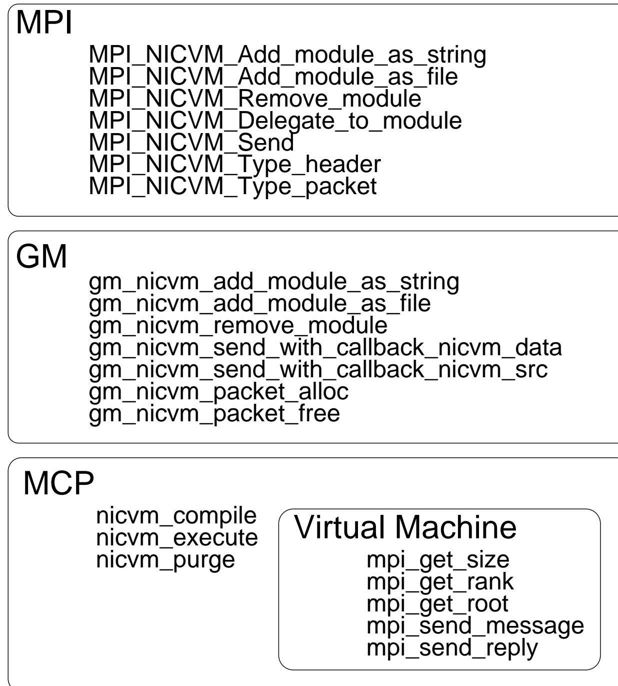


Figure 3: Various functions of the NICVM framework and where they fit in to the software layers. The functions listed inside the virtual machine are actually built into the language utilized by the user modules.

be more natural for programmers to learn and use.

We ended up using a tool named Vmgen [6] to generate an interpreter which is customized for our own needs. Vmgen is a utility that basically accepts a description of an instruction set and generates C code for the corresponding virtual machine. Vmgen generates an engine which accepts as input instructions of the type recognized by the virtual machine and emulates them using C statements. The front end to this engine is a parser created using flex [8] and bison [7], which are standard scanner and parser generators. The parser accepts source code written in the language to be interpreted by the virtual machine and translates it into a sequence of instructions understood by the the engine. This compilation only happens once for a given module. The resulting instructions are then stored in the virtual machine in an optimized direct-threaded manner which supports very low-latency interpretation.

We made several changes to the default Vmgen interpreter templates to generate code that would port to the Myrinet NIC. First, we replaced all dynamic memory allocation with

code to use free lists of statically allocated structures. This is a commonly used technique in the MCP where there is no support for dynamic memory allocation. Next, we implemented our own versions of several standard C library routines on which the parsing code was dependent. A final step in porting was to build the interpreter as a library so it could be linked into the MCP. This involved converting the default executable-style flow of the interpreter code into library functions. These functions allow the MCP to compile modules into the virtual machine, execute modules and purge modules when no longer required. Also, since the original interpreter code was intended to be run as an executable, it only supported one module at a time. So as part of the conversion to a library, we also added code to manage the compilation and execution of multiple modules.

After the initial porting work, we extended the language to include several built-in functions for use by the user-provided code modules. These primitives give the user code access to MPI and GM state such as process ranks and IDs and the number of processes involved in communication. This information may then be used as input to other primitives for the purpose of initiating sends. We also extended the language to include constants for use by the user code in return values. These constants enable the user code to indicate success or failure as well as whether they have consumed a message or if the message requires further processing by the MCP.

4.3 MCP

Our first step in modifying the MCP was to define two new packet types. These allow us to efficiently differentiate between default message traffic and NICVM messages, which require the involvement of our framework. This isolates the overhead of our extensions and prevents impact to default message latency. One packet type contains user source code and another contains data. When a source code packet is received, the MCP compiles it into the virtual machine. When a data packet is received, the MCP hands off the data to the virtual machine, which invokes the appropriate user module. Both the source and data packets contain a name

identifying the module with which they're associated. This allows the virtual machine to match data packets with the compiled version of the appropriate source module.

In order to facilitate multiple reliable NIC-based sends originated by user modules, we employed a new feature of GM-2. In GM-1, there were only two *send chunks* and two *receive chunks*. Both send and receive chunks are just blocks of memory in the NIC SRAM used for staging sends and receives. The send chunks were used to overlap the transfer of data from the host to the NIC with the transfer of data from the NIC to the network. The receive chunks were used in a similar manner to pipeline the transfer of data from the network to the NIC with the transfer of data from the NIC to the host.

However, GM-2 uses send and receive free lists, each containing multiple *descriptors* which take the place of the fixed number of send chunks. Descriptors basically contain pointers to the route, headers and payload in NIC SRAM for a given packet. In addition, each descriptor contains a pointer to a callback function and an associated context pointer. Just after the MCP frees a given descriptor, if a callback function has been specified it is called and passed a pointer to the descriptor as well as the context pointer. The callback is then free to reclaim the descriptor from the free list for use it as desired. In our case we reclaim the descriptor for re-use in subsequent NIC-based sends.

We make use of this mechanism as follows. When the user module wants to initiate sends, we basically just record all of the information required to enqueue the send in a *NICVM send descriptor*. We maintain a queue of these send descriptors for a given GM send or receive descriptor. The queue is organized using a *NICVM send context* which maintains pointers to the first and last NICVM send descriptors as well as other common information such as the active GM port to be used for the sends. Figure 4 illustrates these data structures for a user module which has requested multiple sends.

After the user module terminates, we dequeue the first NICVM send descriptor and use it to start a send. A GM send token is required for each send. In order to avoid interfering with host-based sends on the same port, we use a dedicated send token included as part of

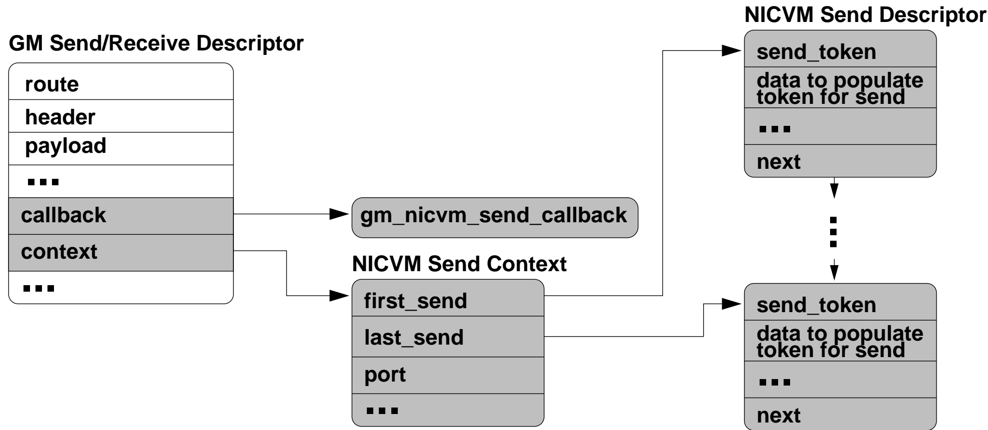


Figure 4: Relationship between a GM descriptor associated with a send or receive and the NICVM send context and NICVM send descriptors used to manage NIC-based sends. The items in white are part of the default GM implementation, while those in gray were added as part of the the NICVM framework.

the NICVM send descriptor. When the MCP finishes the send, it calls our callback which simply reclaims the GM send or receive descriptor. When the send is acknowledged by the receiver, we dequeue the next NICVM send descriptor from the context and start another send. This process repeats until all sends have been completed, at which point we DMA the message to the host if it was originally associated with a receive and has not been consumed by the user module.

4.4 GM and MPI Libraries

Our modifications to the GM library consisted mainly of the addition of API functions to support adding and removing user modules from the NIC and sending data packets. We also included API functions to abstract the process of allocating and freeing NICVM packets. In order to make MPI state information available to the user modules, we also extended the GM port data structure and added a related API function for internal use by MPI in recording state data in the port. We modified the port to record the size of the MPI communicator as well as the mappings from MPI node ranks to the GM node IDs and subport IDs required to enqueue sends in the MCP.

The API routines that we added to the MPI library mostly map onto the underlying GM routines. The main exceptions include a function to explicitly delegate a message to the local NIC and helper routines to abstract the creation of MPI data types for NICVM packets.

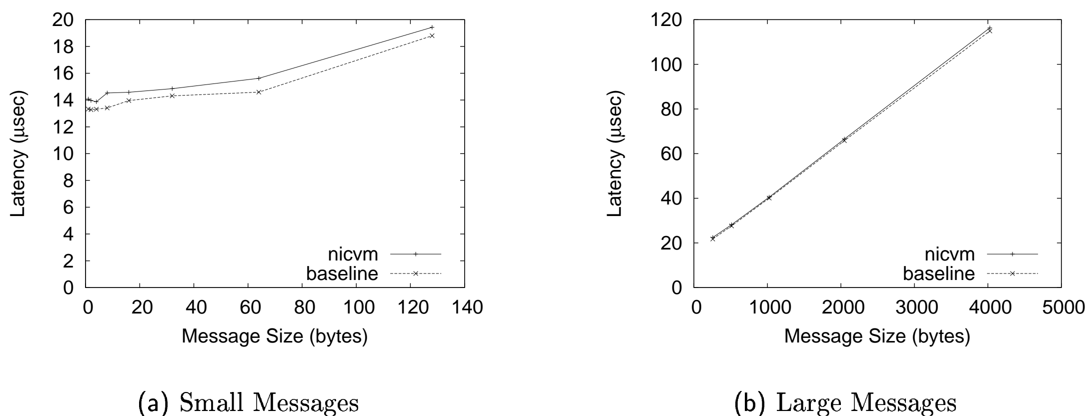
5 Experimental Results

We evaluated our framework on a cluster of 16 dual-SMP 1-GHz Pentium-III nodes with 33-MHz/32-bit PCI. The nodes were connected via a Myrinet-2000 network built around a 32-port switch. Each node contained a PCI64B network interface card with a 133-MHz LANai9.1 processor and 2 MB of SRAM. Our framework is based on MPICH 1.2.5..10 over GM 2.0.3, and all comparisons were performed against the original, unaltered software packages of the same versions.

We created three MPI microbenchmarks for use in evaluating our framework. The first microbenchmark is a standard ping-pong test, where we measure the latency to send a message from one node (the ping node) to another (the pong node) and back. We mainly used this test to evaluate the overhead of our NICVM implementations compared to the base implementation of MPICH-GM. The second microbenchmark is similar to the ping-pong test in that it measures the latency to send a message from one node (the initiator) to another node (the responder) and back. However, we'll refer to this test as the *echo* microbenchmark to distinguish it from the standard ping-pong test. The echo test differs from the ping-pong test in that the message need not be delivered to the host at the responder. The initiator is only concerned that the message is reliably echoed back by the responder. The third microbenchmark measures the total time (latency) to perform a standard broadcast operation, where a message is sent from one node (the root) to all other nodes. For all microbenchmarks, we compare a baseline version using the standard MPI mechanisms to a customized version based on our NICVM framework.

5.1 Overhead Results

We used the ping-pong microbenchmark to measure the overhead of our implementation. The benchmark works as follows. The ping node starts a timer, sends a message to the pong node and waits for a reply. When the reply is received, the timer is stopped. We perform a series of 10,000 iterations and take the average, repeating the process for varying message sizes.



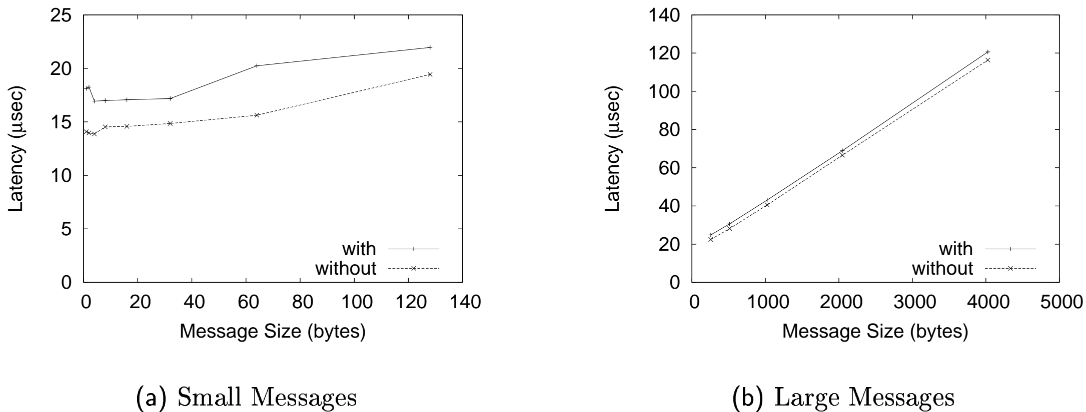
(a) Small Messages

(b) Large Messages

Figure 5: Latency of ping-pong microbenchmark for NICVM version of MPICH-GM (nicvm) vs. default implementation (baseline). These results capture the base overhead of the NICVM modifications to the MPICH-GM software package.

Figure 5 shows the overhead of the addition of our NICVM mechanisms to MPICH and GM. For this test we used two different versions of the ping-pong microbenchmark. One version (baseline) was built against the original implementation of MPICH-GM and the other (nicvm) was built against our customized NICVM implementation. Note that the NICVM build did not actually make use of any new features of the NICVM framework. We can see that even with the incorporation of our interpreter into the MCP, the overhead is minimal and remains fairly constant with increased message size. This is due to the fact that we isolate our NICVM messages into separate packet types to minimize the interference with the default, common-case message traffic.

Figure 6 shows the base overhead of the NICVM module-execution mechanism in action.



(a) Small Messages

(b) Large Messages

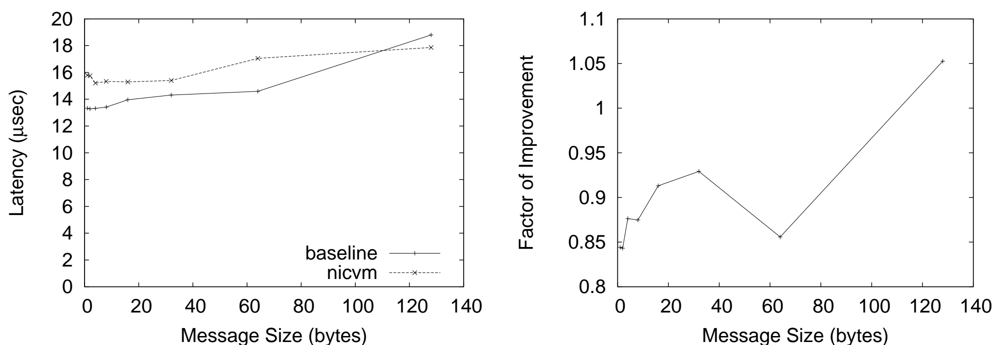
Figure 6: Latency of ping-pong microbenchmark for NICVM implementation of MPICH-GM with and without execution of no-op user module in critical path at pong node. These results capture the base overhead of the NICVM mechanism to execute a user module.

For this test we again used two different versions of the ping-pong microbenchmark. This time both versions were built against our customized NICVM implementation of MPICH-GM. However, one version (with) included the NIC-based execution of a no-op user module in the critical path on the pong node, while the other version (without) did not. For the version with the user module, during the initialization phase a no-op module is uploaded to the NIC at the pong node. Then, instead of sending a default packet type as in the other ping-pong tests, the initiator sends a NICVM packet type which triggers the execution of the no-op module at the pong node. The no-op module just exits and allows the packet to continue to the host. This captures the base overhead of our mechanism for user module execution. We can see that the overhead remains within several microseconds of the default implementation for all message sizes.

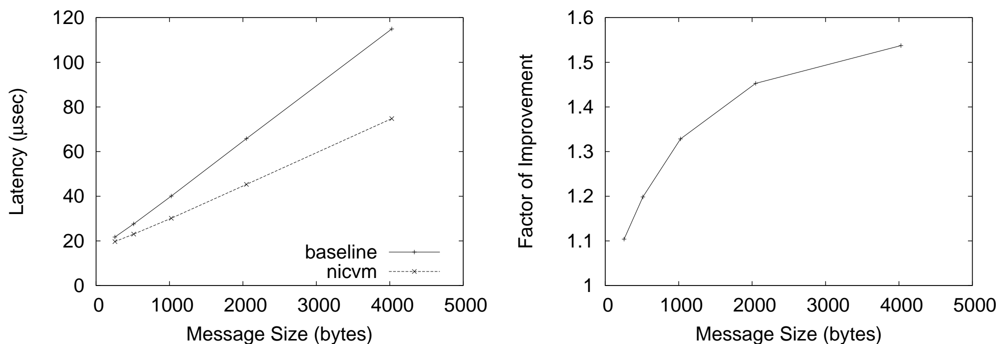
5.2 Echo Results

The echo microbenchmark works as follows. In the baseline version, the benchmark simply uses the send and receive primitives provided by MPI. Note that since the baseline version uses the default MPI send and receive primitives, the message is delivered to the host at

the responder. Therefore, it is basically equivalent to the standard ping-pong test. In the NICVM version, a user-provided module is uploaded to the NIC at the responder during the initialization phase. The initiator then constructs a NICVM packet and targets it for execution by the module installed on the NIC at the responder. The module at the responder simply turns the packet around and sends a reply back to the initiator. For both versions, the timing is performed identically to the ping-pong microbenchmark.



(a) Small Messages



(b) Large Messages

Figure 7: Latency of NIC-based echo (nicvm) vs. host-based echo (baseline) for varying message sizes.

Figure 7 compares the performance of the NIC-based version of the echo microbenchmark (nicvm) to the host-based baseline version (baseline). For small messages we can see that the host-based implementation still outperforms the NIC-based implementation. This is because the overhead of user module execution at the NIC overshadows the benefit of avoiding the

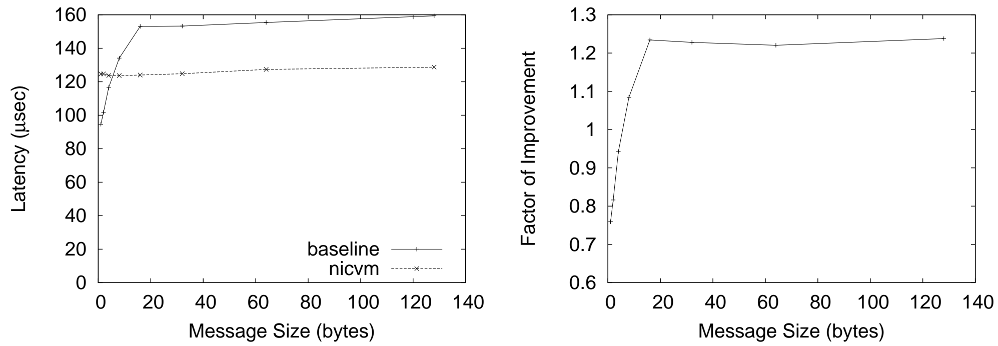
round trip to the host across the PCI bus. However, as the message size increases, we can see the benefit of the NIC-based approach, which outperforms the host-based implementation for all message sizes greater than 128 bytes.

5.3 Broadcast Results

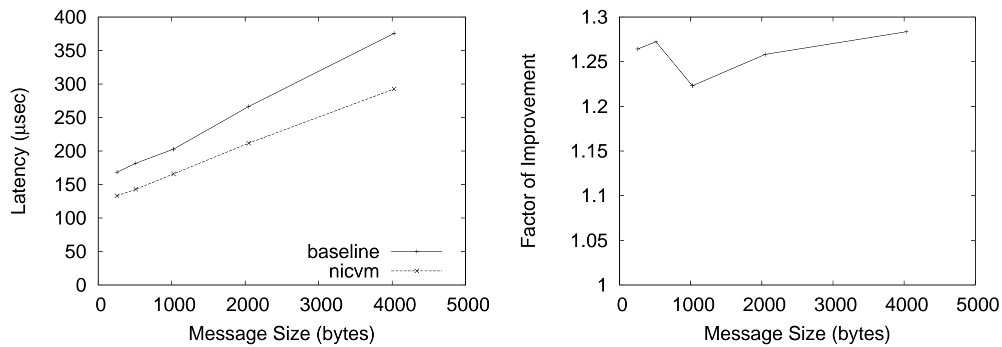
The broadcast benchmark works as follows. As with the echo benchmark, for the baseline version we use the broadcast primitive provided by MPI. The MPICH implementation organizes the broadcast communication into a logical binomial tree. We first determine the one-way latency between the root node and the node which is furthest away from the root (the *last* node) in the logical tree. Next, we time a series of 10,000 broadcasts and take the average, using a barrier to separate iterations. We start timing just before the root node initiates the broadcast. Then, when the last node completes the broadcast, it sends a notification message to the root node, which stops timing and subtracts off the one-way latency associated with the notification message to determine the total broadcast latency. This process is repeated for varying system and message sizes.

In the NICVM version, a user-provided module is uploaded to the NIC at all nodes during the initialization phase. This module implements a broadcast by organizing communication into a logical binary tree as illustrated in Figure 2 from Section 4. The root constructs a NICVM packet targeted for the module installed on the each NIC and delegates the packet to its local NIC. All other nodes simply perform a standard MPI receive. The NIC at the root node then assumes responsibility for initiating the first two point-to-point sends associated with the broadcast. As the NICs at the other nodes receive the packet, the associated module decides whether or not to perform additional sends based on the position of the node in the logical tree. The timing is performed identically to the baseline version.

Figure 8 shows the results of the broadcast microbenchmark for 16 nodes. We can see that the NIC-based implementation consistently outperforms the host-based implementation for all but the smallest message sizes. We see a maximum factor of improvement of nearly



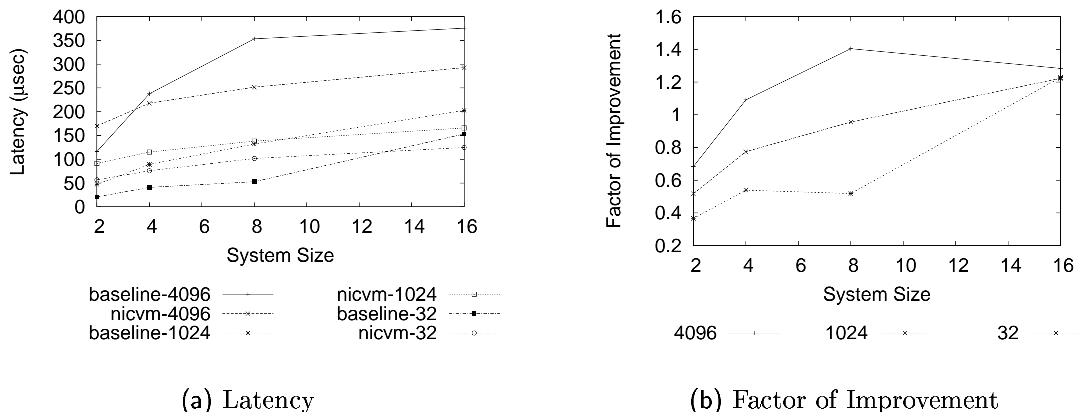
(a) Small Messages



(b) Large Messages

Figure 8: Latency of NIC-based (nicvm) broadcast vs. host-based broadcast (baseline) for 16 nodes and varying message sizes.

1.3 for 4096-byte messages. As with the echo benchmark, the NIC-based implementation performs better for larger messages due to the fact that we avoid the trip across the PCI bus associated with a DMA to the host.



(a) Latency (b) Factor of Improvement

Figure 9: Latency of NIC-based broadcast (nicvm) vs. host-based broadcast (baseline) for 2, 4, 8 and 16 nodes with 32, 1024 and 4096-byte message sizes.

Figure 9 shows the results of the broadcast benchmark for varying system size. Here we can see that the factor of improvement increases with system size, indicating the enhanced scalability of the NIC-based approach.

6 Related Work

The U-Net/SLE [16] project ported a Java virtual machine to the NIC on a Myrinet network. There are several major differences between this work and our NICVM framework. First, U-Net/SLE utilizes a Java virtual machine while we take a more customized approach, building an interpreter from scratch specifically for use on the NIC. Even though the Java virtual machine used by U-Net/SLE has been stripped of non-essential Java language features, it still incurs a high amount of base overhead. This overhead makes the NIC-based approach slower than similar host-based approaches for all but the simplest tests. Second, in U-Net/SLE a single Java class file may be associated with a given U-Net user endpoint. A U-Net endpoint

is equivalent to a port in GM in that it abstracts an application's connection to the network. Once associated with an endpoint, methods in the class are called to process all incoming and outgoing messages. In contract, NICVM allows multiple user modules to be added to the NIC and does not make any association between a module and an application or port. In fact, NICVM modules may even be left on the NIC for utilization after a user application terminates. Also, NICVM packets are differentiated from standard GM packets so that the overhead of the mechanism for executing user modules may be avoided unless actually required. Finally, to the best of our knowledge no high-level API such as MPI has been ported to U-Net/SLE. As part of the NICVM framework, we provide extensions to both the GM and MPI layers, making our offload features easily accessible to both user applications and API developers.

Recent versions of Quadrics [11] have included a feature that enables end users to compile a code module and load it into the NIC at runtime. This code is then executed by a dedicated thread processor on the NIC. While this approach enables offload of processing to the NIC, it also has some minor drawbacks. First, although more than one module may be added to the NIC, there is no published way to remove a module. Also, a module is only active as long as the user program is alive, so extra effort is needed to offload persistent code to the NIC.

Active Messages (AM) [15] also provides packet driven handler invocation. The AM packet, however directly specifies the address of a handler routine to be used in processing the packet, making it less flexible than the dynamic NICVM framework where the loaded source modules may vary from NIC to NIC. Moreover, the AM handler actually executes on the host, so it can't actually provide the benefit of offloading computation to the NIC.

7 Conclusions and Future Work

We have described both the design challenges and implementation details of our framework for offload of dynamic user-defined modules to the NIC on Myrinet clusters. Upon evaluation of our implementation, we found a maximum factor of improvement of nearly 1.3 for NIC-based broadcasts when compared to a similar host-based implementation. Furthermore, we note that the factor of improvement increases with system size, indicating that the benefits of our implementation will lead to improvements in scalability on larger clusters.

In the future, we intend to perform application-based evaluations as well as evaluations on large-scale clusters. We also plan to extend the framework to support NIC-based reduction using user-provided operator modules. We feel that this would be a natural extension to the existing MPI capabilities which allow users to define their own host-based reduction operators. In an effort to further enhance performance and usability, we plan to investigate the feasibility of letting users compile and perform basic validation of their source modules on the host. This would eliminate the need to perform the compilation on the NIC, further lightening the NIC-based virtual machine. It would also make basic debugging tasks easier for users. We also intend to investigate the security issues that we were unable to fully explore during the development of this version of the framework.

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 and D. K. Panda. NIC-Based Reduction in Myrinet Clusters: Is It Beneficial? In *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2003.

[3] 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.

[4] Phil Burk. pForth - portable Forth in 'C'. http://www.softsynth.com/pforth/, 1998.

[5] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[6] M. Anton Ertl. Vmgen Interpreter Generator. http://www.complang.tuwien.ac.at/anton/vmgen/, 2004.

[7] Free Software Foundation. Bison - GNU Project. http://www.gnu.org/software/bison/bison.html, 2004.

[8] Free Software Foundation. Flex - GNU Project. http://www.gnu.org/software/flex/, 2004.

[9] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[11] Quadrics Supercomputers World Ltd. QsNet high performance interconnect. http://www.quadrics.com/website/pdf/qsnet.pdf.

[12] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[13] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable NIC-based Reduction on Large-scale Clusters. In *Proceedings of the SuperComputing Conference (SC)*, November 2003.

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

[15] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[16] M. Welsh, D. Oppenheimer, and D. Culler. U-Net/SLE: A Java-based User-Customizable Virtual Network Interface. In *Proceedings of the Java for High-Performance Network Computing Workshop held in conjunction with EuroPar '98*, September 1998.