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

Adam Wagner, Hyun-Wook Jin and Dhabaleswar K. Panda
Network-Based Computing Laboratory
Dept. of Computer Science and Engineering
The Ohio State University
{wagnera, jinhy, panda}@cse.ohio-state.edu

Rolf Riesen
Scalable Computing Systems Dept.
Sandia National Laboratories
rolf@sandia.gov

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 1.2 with respect to total latency as well as a maximum factor of improvement of 2.2 with respect to average CPU utilization under conditions of process skew. In addition, we see that these improvements increase with system size, indicating that our NIC-based framework offers enhanced scalability when compared to a purely host-based approach.

1. Introduction

Many of 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 [6, 11] like multicast [2] and reduce [14] and synchronization operations such as barrier [4]. The potential benefits of NIC-based offload include lowered communication latency, reduced host CPU utilization, improved tolerance of process skew [3, 17] and better overlap of computation and communication.

The common approach to NIC-based offload is to hard-code an 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 quite complex and error prone due to the specialized nature of the NIC firmware and the difficulty of 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.

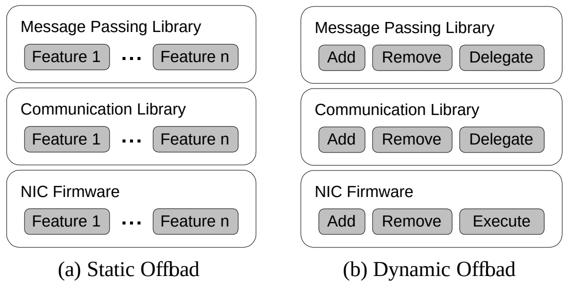


Figure 1. Static, hard-coded, ad-hoc offload of features to NIC vs. flexible framework for dynamic offload of user modules to NIC.

Figure 1 illustrates the difference between a hard-coded, static approach and a more flexible, dynamic approach to NIC-based offload. We can see that in the static approach, we are limited to a fixed number of features, while in the dynamic 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 incom-

* 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.

ing 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 (NIC-based Virtual Machine). 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 measured performance with respect to both latency and host CPU utilization. When compared to a similar host-based implementation on 16 nodes, we observe a maximum factor of improvement of 1.2 with respect to latency, and under conditions of process skew we observe a maximum factor of improvement of 2.2 with respect to CPU utilization. Furthermore, we find that in both cases these performance benefits increase with system size.

The remainder of this paper is organized as follows. In the next section, we provide brief background information. In section 3 we discuss the design challenges we encountered while implementing our framework and in section 4 we detail our implementation. In section 5 we evaluate the performance of our implementation and in section 6 we discuss related work. Finally, in section 7 we present our conclusions and discuss future work.

2. Background

GM [15] is a user-level message-passing subsystem for Myrinet networks. Myrinet [1] is a 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 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.

GM provides user-level, memory-protected network access to multiple applications at once by multiplexing the network resources between applications. The communication endpoints used by applications are called *ports*. GM maintains reliable connections between each pair of nodes and then multiplexes traffic across these connections for multiple ports. This gives applications the advantage of reliable in-order message delivery without having to explicitly establish connections.

MPI [13] is a standard interface for message passing in parallel programs. MPICH [10] is the reference implementation of MPI and has been ported to a variety of hardware platforms including GM over Myrinet. The standard implementation of MPICH over GM (MPICH-GM) does not include support for NIC-based offload techniques.

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 startup latency required to activate a given user module on the NIC. This latency includes the time to determine which module should be activated as well as the time to perform any sort of environmental setup required for module execution. The second area where performance is critical is the actual time required to execute a given module of user-code once it has been located and its execution environment has been initialized. If the startup 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 highly tuned and adding any extra delay to process user code can have a negative impact on overall performance. For example, if a user code module takes too long to execute it may cause temporary receive queue buffers on the NIC to overflow, which will result in packets being dropped and potentially even a reset of the associated communication port.

3.2. 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 reliable 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 which involve both performing sends as well as transferring a received message to the host via DMA. The easiest solution would be to allow the receive DMA 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 DMA to the host later. This sort of behavior is especially beneficial for collective-style communications, where the DMA can often be moved outside of the critical communication path.

3.3. 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 a user module 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 on a particular node.

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 nearly 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 heavy 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.

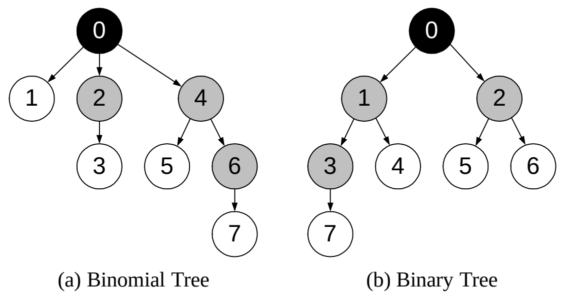


Figure 2. Examples of logical trees used to organize point-to-point communications between eight processes involved in a broadcast operation. The root node is shown in black, internal nodes are colored gray and leaf nodes are shown in white.

Broadcast is a common collective operation where a buffer of data is sent from one node (the *root* node) to all other nodes involved in the communication. In MPICH, each process calls the `MPI_Bcast` function at the application level to initiate the broadcast, with the root node supplying the outgoing buffer of data and the other nodes supplying empty buffers for incoming data. Internally, MPICH organizes the nodes into a logical tree and performs the broadcast using point-to-point communication between nodes. Figure 2 illustrates two different logical broadcast trees for eight processes. The root process is shown in black, internal processes are colored gray and leaf processes are shown in white. The arrows between processes indicate the direction of point-to-point messages associated with the broadcast.

Figure 2(a) is a binomial tree, which is the tree utilized by the default MPICH implementation of broadcast. The goal here is to maximize the amount of communication overlap. However, the logic required to construct the tree is significantly more complicated than the simple computation involved in constructing a binary tree like that in Fig. 2(b). Since the NIC has such limited processing capabilities and strict latency requirements, the simpler approach of the binary tree has the potential to offer better performance in a NIC environment.

In order to implement a NIC-based broadcast using 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. The simple module that we used for our experiments consisted of only 20 lines of code [18]. 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 behave 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 software layers or disturbance of the cluster environment.

Contrast this to the work required to perform a similar implementation without using our framework. First we would need to locate the source code for the MCP and incorporate our custom broadcast code. Even with 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.

The main components involved in our framework 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. Each layer relies on the API routines of the layer below. The functions listed inside the virtual machine are actually built into the language utilized by the user modules. Currently, we just provide basic primitives to enable forwarding messages. However, in order to make the framework more flexible we eventually plan to add primitives to support the customization of packet headers and payload.

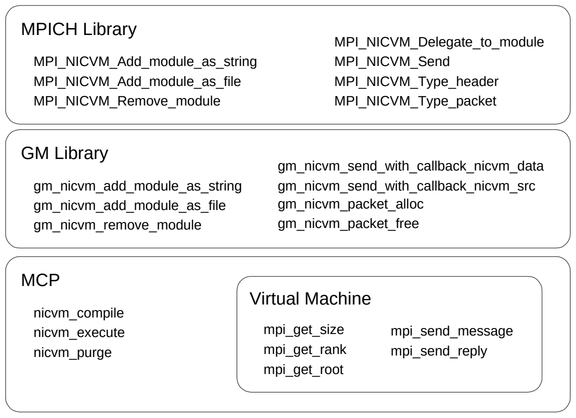


Figure 3. Various functions of the NICVM framework and where they fit in to the software layers. The functions listed inside the virtual machine are actually built into the language utilized by the user modules.

4.2. Virtual Machine

We originally began our research using a Forth interpreter named pForth [5]. 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 be more natural for programmers to learn and use.

We ended up using a tool named Vmgen [7] 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 [9] and bison [8], 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 during the initialization phase. 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 both flex and 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 that it could be linked into the MCP. This involved breaking 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 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 it has 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. Figure 4 illustrates the integration of the virtual machine into the MCP. The MCP consists of four main software state machines associated with sending and receiving packets. The interpreter is situated on the receive path and is activated after a NICVM packet is received from the network but before the associated host DMA is initiated. The dashed arrows indicate the path exclusive to NICVM messages. Even though the interpreter is located on the receive path, it can also intercept NICVM packets delegated from the local host via a loopback path between the send and receive state machines.

One NICVM packet type contains user source code and the other contains data. When a source code packet is received, the MCP compiles it into the virtual machine. So in order to add a user module to the NIC, the host need only send a source code packet to its local NIC via the loopback path. Such details are abstracted from the user via API routines. When a data packet is received, the MCP hands off the data to the virtual machine, which invokes the appropriate user module. This processing is illustrated in detail in Figure 5. Both the source and data packets contain a name identifying the module with which they are associated. This allows the virtual machine to match data packets with the compiled version of the appropriate source module. Note that the user module may choose to *consume* the packet, indicating that the receive DMA to the host should be skipped. The receive DMA will also be skipped temporarily if the

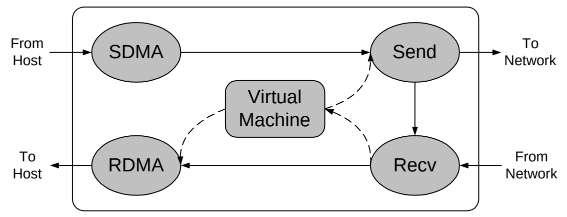


Figure 4. Integration of virtual machine into MCP. The ovals represent the different state machines which comprise the NIC logic. The solid arrows show the default path of packets through the MCP, while the dashed arrows indicate the path of packets containing NICVM source code or data. The arrow from the the Send state machine to the Recv state machine indicates loopback.

user module initiates one or more sends. In this case, the DMA is actually postponed until after the sends complete so that it occurs outside of the critical communication path.

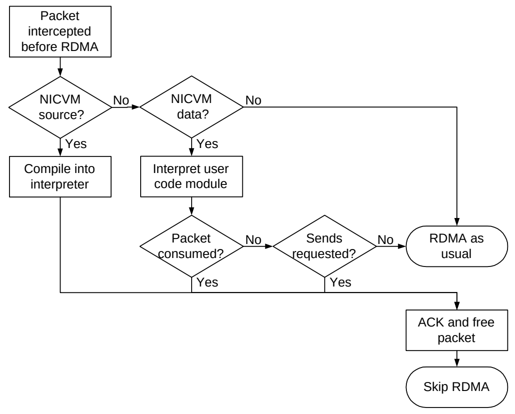


Figure 5. Synchronous component of NICVM packet processing.

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 as desired. In our case we reclaim the descriptor for re-use in subsequent NIC-based sends.

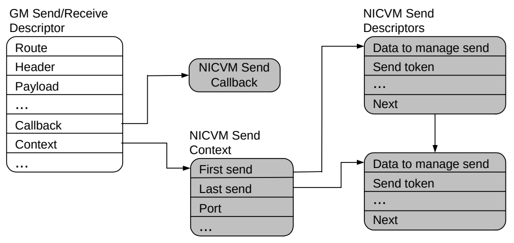


Figure 6. Relationship between a GM descriptor associated with a send or receive and the NICVM send context and NICVM send descriptors used to manage NIC-based sends. The items in white are part of the default GM implementation, while those in gray were added as part of the the NICVM framework.

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. Figure 6 illustrates these data structures for a user module which has requested two sends. 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. By *active* GM port, we mean the communication port associated with the send or receive that invoked the user module.

After the user module terminates, we proceed in an asynchronous manner to perform the actual sends. This process is illustrated in Figure 7. Just after the GM descriptor associated with the original send or receive is freed, the MCP calls our NICVM send callback. We reclaim the GM descriptor, dequeue the first NICVM send descriptor and enqueue the associated 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 the NICVM send descriptor. When the MCP finishes the send, it again frees the GM descriptor and calls our callback. This time the callback just reclaims the descriptor but doesn't initiate the next send. Instead, we wait until the previous send has been acknowledged by the recipient and then proceed. This cycle repeats until all sends have been completed, at which point we DMA the message to the host if necessary.

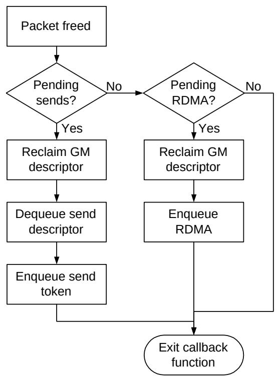


Figure 7. Asynchronous processing of sends requested by a user code module.

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 two MPI microbenchmarks for use in evaluating our framework. The first 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. The second microbenchmark is similar in that we evaluate the broadcast operation. However, in this case we measure the average per-node host CPU utilization associated with performing the broadcast under varying amounts of process skew. For both microbenchmarks, we compare a baseline version using the standard MPI mechanisms to a customized version based on our NICVM framework.

5.1. Latency Results

The broadcast latency benchmark works as follows. For the baseline version we use the broadcast primitive provided by MPI. As described in Section 4, the MPICH implementation organizes the broadcast communication into a logical binomial tree. 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. When a non-root completes the broadcast, it sends a notification message to the root node. The root node stops timing after receiving notification messages from all other nodes. The notification messages may be received by the root node in any order so as to avoid introducing unnecessary serialization of receives. This process is repeated for varying system and message sizes.

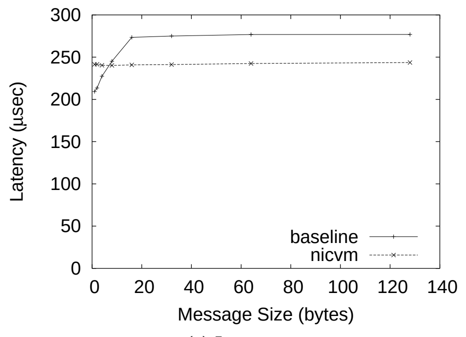
In the NICVM version, a user-provided module is uploaded to the NIC at all nodes during the initialization phase. This module implements a broadcast by organizing communication into a logical binary tree. The root constructs a NICVM packet targeted for the module installed on each NIC and delegates the 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 broadcast 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.

Figures 8 and 9 show the results of the broadcast latency 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 1.2 at large message sizes. The NIC-based implementation performs better for larger messages due to the fact that for internal nodes we avoid a trip across the PCI bus associated with a send DMA from the host to the NIC. Another factor in the improved performance is that for internal nodes, the DMA to the host associated with the received broadcast message is delayed until after the broadcast message is propagated to the node's children. This takes the receive DMA out of the critical path with respect to the entire operation and allows the broadcast to progress more quickly overall.

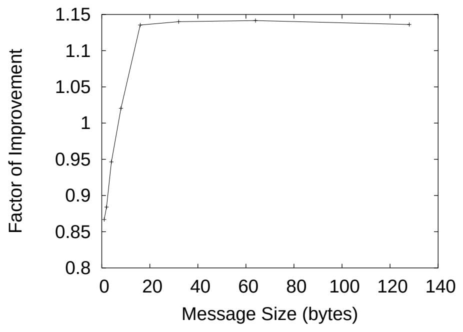
Figure 10 shows the results of the broadcast latency benchmark for varying system size. Here we can see that the factor of improvement increases with system size, indicating the enhanced scalability of the NIC-based approach.

5.2. CPU-Utilization Results

The broadcast CPU-utilization benchmark is implemented slightly differently than the corresponding latency benchmark. In addition to varying the number of nodes and the message size, we also introduce a variable amount of delay at each node to simulate process skew. First, we convert a given maximum amount of delay from microseconds to busy-loop iterations at each node. All delays are then generated using busy loops as opposed to absolute timings. This enables us to capture the CPU utilization associated with the broadcast operation. Next, we perform a series of 10,000 broadcasts and take the average across all nodes, using a barrier to separate iterations.



(a) Latency

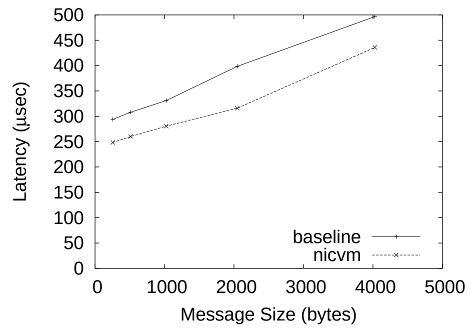


(b) Factor of Improvement

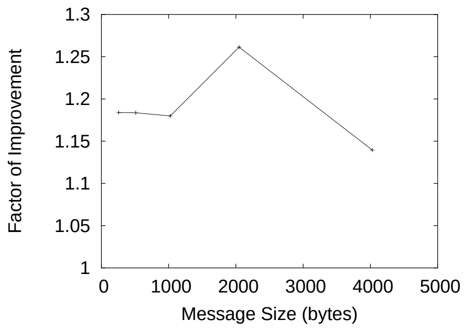
Figure 8. Latency of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 16 nodes and small message sizes.

Within each loop iteration, the timing measurements are taken as follows. At each node we first start timing, then introduce a random amount of delay between zero and the maximum delay, perform the broadcast, introduce a catchup delay and finally stop timing. The skew delay as well as the catchup delay are then subtracted from the measured time at each node to calculate the CPU utilization. The catchup delay is equal to the maximum skew delay plus a conservative estimate of the maximum broadcast latency. The intent here is to be sure to delay long enough to capture all asynchronous processing in the overall time measurement.

Fig. 11 shows the results of the broadcast CPU-utilization benchmark for 16 nodes with increasing amounts of process skew and message sizes of 4096 and 32 bytes. We can see that the NICVM implementation consistently outperforms the default implementation for all combinations of skew and message size, with Figure 11(b) showing a maximum factor of improvement of 2.2. As the amount of skew increases, internal nodes in the default implementation spend more and more time waiting on the broadcast message from their parent so that

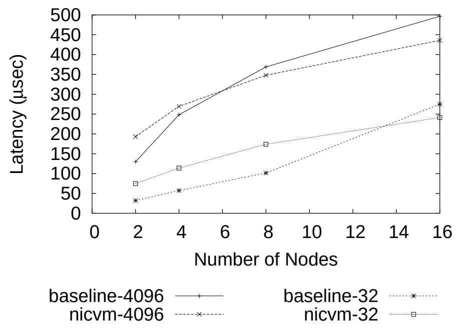


(a) Latency

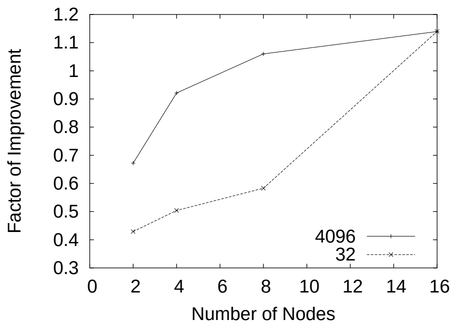


(b) Factor of Improvement

Figure 9. Latency of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 16 nodes and large message sizes.



(a) Latency



(b) Factor of Improvement

Figure 10. Latency of NIC-based broadcast (nicvm) and host-based broadcast (baseline) for 2, 4, 8 and 16 nodes with 32 and 4096-byte message sizes.

they can propagate the message to their children. However, in the NICVM case all non-root nodes simply perform a receive at the host level and delegate all of the intermediate broadcast processing to the user code module on the NIC. The artificial process skew still causes each host to be delayed, but the overall broadcast operation is less affected as the NIC takes care of forwarding broadcast messages to the children.

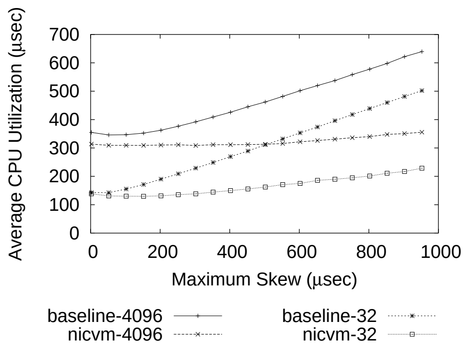
Fig. 12 shows the results of the broadcast CPU-utilization benchmark for for 2, 4, 8 and 16 nodes with a maximum process skew of of 1,000 μs and message sizes of 4096 and 32 bytes. These results confirm that the results demonstrated in Fig. 11 hold for varying system sizes. Once the system size increases past the unrealistic two-node scenario, the NICVM implementation outperforms the default implementation for all message sizes. Furthermore, we can see that the factor of improvement increases with system size, demonstrating the scalability benefits of offloading computation to the NIC.

Note that in both of the previous cases, the greatest factor of improvement occurs for smaller message sizes. This is because small messages are the most vulnerable to the effects of process skew since the effects of factors such as transmission time, copy time and DMA time are less prevalent then they are for larger messages.

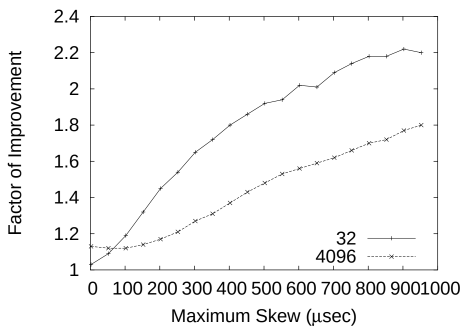
Fig. 12 shows the results of the broadcast CPU-utilization benchmark without process skew for for 2, 4, 8 and 16 nodes and message sizes of 4096 and 32 bytes. Here we can see that even without the introduction of artificial process skew, the NICVM implementation eventually outperforms the default implementation for all message sizes beyond the fairly modest system size of eight nodes. This is due to the fact that process skew is naturally introduced as the number of nodes involved in the broadcast increases and there are more opportunities for the nodes to become unsynchronized.

6. Related Work

The U-Net/SLE [19] 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 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

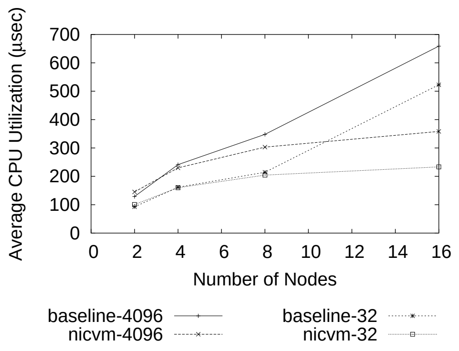


(a) Average CPU Utilization

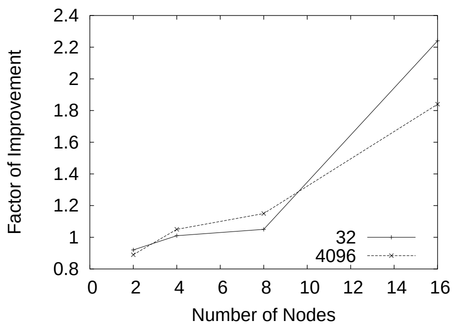


(b) Factor of Improvement

Figure 11. Average CPU utilization of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 16 nodes with varying process skew and 4096 and 32-byte messages.



(a) CPU Utilization



(b) Factor of Improvement

Figure 12. Average CPU utilization of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 2, 4, 8 and 16 nodes with maximal process skew and 4096 and 32-byte messages.

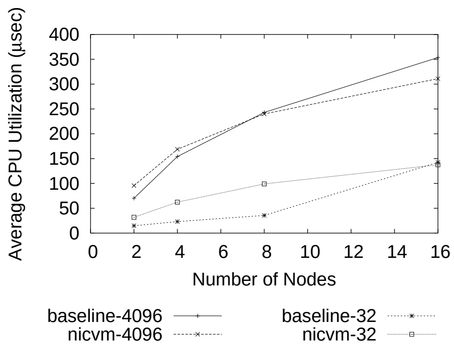
is equivalent to a port in GM in that it abstracts an application's connection to the network. Once associated with an endpoint, methods in the class are called to process all incoming and outgoing messages. In contract, NICVM allows multiple user modules to be added to the NIC and does not make any 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 [12] 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.

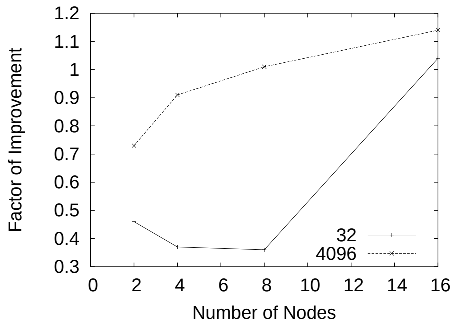
Active Messages (AM) [16] also provides packet driven handler invocation. The AM packet, however directly specifies the address of a handler routine to be used in processing the packet, making it less flexible than the dynamic NICVM framework where the loaded source modules may vary from NIC to NIC. Moreover, the AM handler actually executes on the host, so it can't 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. With respect to overall latency, we found a maximum factor of improvement of 1.2 for NIC-based broadcasts when compared to a similar host-based implementation. Furthermore, we observed a factor of improvement in CPU utilization of up to 2.2 under conditions of process skew. We observe that in both cases the factor of improvement increases with system size, indicating that the benefits of our implementation will lead to improvements in scalability on larger clusters. However, note that while performance improvement is de-



(a) CPU Utilization



(b) Factor of Improvement

Figure 13. Average CPU utilization of NIC-based (nicvm) broadcast and host-based broadcast (baseline) without process skew for 2, 4, 8 and 16 nodes and 4096, 1024 and 32-byte messages.

sirable, the main focus of this work has been to enable end users to dynamically offload computation to the NIC.

In the future, we intend to port our framework to the latest NIC hardware and perform application-based evaluations as well as evaluations on more contemporary 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. Another essential addition to the interpreter and the associated language is support for floating-point operations. The current Myrinet NICs do not include hardware support for such operations, so such an effort will require software emulation.

We also intend to investigate the security issues that we were unable to fully explore during the development of this version of the framework. Because of the fact that we chose the virtual machine approach to NIC-based execution of user code, we have complete control over the user code modules and should be able to address such issues as necessary.

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 R. Brightwell. Application-Bypass Broadcast in MPICH over GM. In *Proceedings of the Cluster Computing and Grid Conference (CCGrid)*, May 2003.

[4] 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.

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

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

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

[8] F. S. Foundation. Bison - GNU Project. http://www.gnu.org/software/bison/bison.html, 2004.

[9] F. S. Foundation. Flex - GNU Project. http://www.gnu.org/software/flex/, 2004.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

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

[12] Q. S. W. Ltd. QsNet high performance interconnect. http://www.quadrics.com/website/pdf/qsnet.pdf.

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

[14] 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.

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

[16] 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.

[17] A. Wagner, D. Buntinas, D. K. Panda, and R. Brightwell. Application-Bypass Reduction for Large-Scale Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, December 2003.

[18] A. Wagner, H. W. Jin, R. Riesen, and D. K. Panda. NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters. Technical Report OSU-CISRC-5/04-TR31, 2004.

[19] 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.