Static and Dynamic Processing Offload on Myrinet Clusters with Programmable NIC Support

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Adam L. Wagner, B.S.

* * * * *

The Ohio State University

2004

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. Mario Lauria

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

Clusters of workstations are becoming the prevalent supercomputing architecture. Many of the modern networks used to interconnect nodes in cluster-based systems provide network interface cards (NICs) with programmable processors. Substantial research has been done with the focus of offloading processing tasks from the host to the NIC processor with the goals of reducing host CPU utilization, improving overlap of computation and communication and lowering overall communication latency.

However, the research thus far has primarily focused on performing NIC-based offload in a static manner, where specific optimizations are hard-coded into the NIC for maximum performance. Because of the level of difficulty involved in such development, such optimizations may only be performed by system experts. In other words, direct utilization of the NIC-based resources by end users is not practical. Since the resources available on the NIC are typically quite constrained, only a limited number of features may be hard-coded into the NIC at a given time.

Because a fairly limited amount of processing resources are available at the NIC, there are cases where certain tasks actually perform worse when offloaded. It may make more sense to perform the majority of the processing on the host and just request a limited amount of assistance from the NIC. For instance, the NIC may only be utilized to trigger asynchronous processing on the host.

In this thesis, we discuss the design challenges we faced during the implementation of two frameworks. Both frameworks are extensions to the popular MPICH-GM message passing subsystem for clusters utilizing the Myrinet network interconnect. The first takes a NIC-assisted approach to implement an optimized version of the reduction operation, while the second supports dynamic NIC-based offload of code modules provided by end users.

We evaluate our implementations on 16 and 32-node Myrinet clusters. For the first framework, we see a maximum factor of improvement in CPU utilization of 5.1 under conditions of process skew. Process skew is unavoidable in large-scale clusters and in this case average CPU utilization is our target performance metric. We evaluate the second framework in the context of the broadcast operation. Here we observe a maximum factor of improvement of 1.4 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.

I dedicate this work to my wife Cheryl and to my parents Gary and Esterina.

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my advisor Prof. Dhabaleswar K. Panda for his support and guidance throughout the course of my graduate studies. He has always been there when I needed him and his perspective has been vital to my success.

I would like to thank Prof. Mario Lauria for agreeing to serve on my Master's examination committee, and also for his technical inspiration during our work together in research seminars.

For their generous financial support, I would like to thank Sandia National Laboratories. I would specifically like to express my gratitude to Dr. Rolf Riesen and Dr. Ron Brightwell whose ideas inspired the research in this thesis.

Finally, I would like to thank the Network-Based Computing Laboratory members, both past and present. Darius Buntinas was a major help in mentoring me through my first research project on the team, and Hyun-Wook Jin and Weikuan Wu have been very supportive of my recent work. Ranjit Noronha and Pavan Balaji were a great help with diagnosing network hardware problems. The entire team has been instrumental in brainstorming potential solutions and helping to refine both my approach to problems and my presentation of results.

VITA

September 16, 1971	Born - Butler, Pennsylvania, USA
September 1989 - May 1994	B.S. in Computer Science, Kent State University, Kent, Ohio
June 1994 - October 1996	Software Engineer, Overdrive Systems, Cleveland, Ohio
October 1996 - March 1998	Developer/Analyst, Caliber Technology, Akron, Ohio
March 1998 - June 2001	Technical Specialist, Cendant Membership Services, Dublin, Ohio
June 2001 - September 2002	Consultant, BMW Financial Services, Dublin, Ohio
September 2002 - January 2003	Graduate Teaching Associate, The Ohio State University, Columbus, Ohio
January 2003 - June 2004	Graduate Research Associate, The Ohio State University, Columbus, Ohio
June 2003 - September 2003	Student Intern, Sandia National Laboratories, Albuquerque, New Mexico

PUBLICATIONS

Research Publications

NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters, A. Wagner, H. W. Jin, Rolf Riesen and D. K. Panda, In *Proceedings of the International Conference on Cluster Computing*, to be presented, San Diego, California, September 2004.

Application-Bypass Reduction for Large-Scale Clusters, A. Wagner, D. Buntinas, R. Brightwell and D. K. Panda, *International Journal of High Performance Computing and Networking*, Cluster 2003 Special Issue.

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

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in Studies in High-Performance Computing: Prof. D. K. Panda

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Prior to the early 1990s, large-scale scientific computing was conducted primarily on massively parallel processors (MPPs). However, over the past decade the trends have shifted and clusters of commodity workstations are now replacing MPPs as the architecture of choice for supercomputing.

Clusters of workstations have several advantages over traditional massively parallel architectures. First, they can be constructed fairly cheaply and easily using off-the-shelf processing and networking components. Standard workstation-class PCs may be used for the processing nodes and the network can range from common LAN hardware to sophisticated high-performance interconnects designed specifically for clusters. Second, they can be extended and upgraded more easily than highly integrated MPP machines. For example, processing nodes or components therein may be upgraded as they become obsolete, nodes may be added as more funds become available and the entire interconnect may be upgraded or replaced. Finally, the communication software required to support scientific applications is readily, freely available for most common off-the-shelf processor and network architectures.

Until recently, the main disadvantage of clusters has been their reliance on commodity networks. While MPPs have been designed with custom networks that are tuned for the communication involved in parallel computation, the networks used in early clusters were designed for more general-purpose connectivity. Such general-purpose networks fail to deliver the latency, bandwidth and advanced features of more sophisticated interconnects.

Since clusters began gaining popularity, several major advances have been made towards improving the networks used as cluster interconnects. The first such change was the introduction of user-level communication protocols. Clusters initially utilized general-purpose communication protocols such as TCP/IP, which relies on the kernel to manage communication between an application and the network. When used for parallel computing, such protocols lead to excessive interrupts and context switches as messages are delivered, which in turn results in poor network performance and high host CPU utilization. User-level protocols take the kernel out of the critical communication path, allowing applications to communicate directly with the network and eliminating the kernel-related overhead.

Another major advance has been the introduction of network interface cards (NICs) with programmable processors. Instead of having the communication protocol logic hard-wired into the NIC, these cards provide a general-purpose processor that executes a firmware control program. This provides system architects with an extra degree of flexibility in utilizing the resources on the NIC. For example, the NIC firmware can be customized to better support the upper-level communication software stack. In addition, communication primitives and operations can be offloaded from the host processor to the NIC processor.

1.1 Problem Statement

NIC-based programming is notoriously difficult, much more so than programming on the host. This is due to various factors including the specialized nature of NIC firmware, the limited resources available on the NIC and the general lack of a robust software development and debugging environment. However, there are significant potential benefits to be gained from utilizing the processing resources of the NIC to either support the upper-level communication stack or to directly offload computation from the host. These benefits include reduced host CPU utilization, improved tolerance of process skew, better overlap of computation and communication and lowered communication latency.

Many of the collective communication operations commonly utilized in scientific applications could benefit from being split into synchronous and asynchronous components. These collective operations tend to either explicitly or implicitly introduce synchronization between the processes involved in the communication. This synchronization makes such operations highly susceptible to performance degradation due to conditions where the processes become unsynchronized or skewed. Such conditions are common in clusters, and become more prevalent as the size of a cluster grows and more opportunities for unpredictable delays are introduced. By splitting the communication into synchronous and asynchronous components, we can allow the operation to make progress independently of the application. This technique is known as application bypass.

However, in order to achieve such application bypass, we need to introduce some means to trigger the asynchronous processing. This is where the programmable features of the NIC can play a role in a limited fashion. Without extensive NIC-based programming, we can make small changes to the NIC firmware to trigger asynchronous processing on the host. The host can then handle the complexities associated with the collective operation. Such techniques are called NIC-assisted, since the NIC plays a minor role relative to the host.

With more of an investment in NIC-based development, operations may be entirely offloaded to the NIC processor. Past research in this area has mainly focused on ad-hoc customizations to enhance the performance of specific collective communication operations. The common approach has been to hard-code the optimization into the NIC firmware in order to achieve the highest possible performance gain. Because of the level of difficulty involved in such development and the potential consequences of erroneous code, these sorts of optimizations may only be performed by system experts. This makes direct utilization of the NIC-based resources by end users out of the question. Since the resources available on the NIC are typically quite constrained, only a limited number of features may be compiled into the firmware at any given time. Furthermore, frequent changes to the NIC firmware may be impractical on production systems that demand high levels of stability, availability and security.

The goal of this work is to address the following questions.

1. Can we take advantage of this NIC programmability in a limited fashion to support optimization of the host-based communication software stack?

2. Can we make these NIC-based computing resources dynamically available to the end user for general purpose processing tasks?

1.2 Our Approach

In this thesis, we discuss the design challenges we faced during the implementation of two frameworks. Both frameworks are extensions to the popular MPICH-GM message passing subsystem for clusters utilizing the Myrinet network interconnect. The first takes a NIC-assisted application-bypass approach, while the second supports dynamic NIC-based offload of code modules provided by end users.

In order to illustrate the benefits of these techniques, we have evaluated their effects on two popular collective operations. For the NIC-assisted application-bypass framework, we have implemented the reduction operation, while for the NIC-based offload framework, we have implemented the broadcast operation. The implementation of the reduction operation involved the static addition of reduction-specific application-bypass logic into the MPICH library, as well as some minor enhancements to the NIC firmware to support asynchronous processing. However, the implementation of the broadcast operation is simply an example of the capabilities that can be offloaded to the NIC as a code module using the NIC-based offload framework.

The remainder of this thesis is organized as follows. In the next chapter we provide an overview of Myrinet and MPICH-GM. In chapter 3 we discuss our implementation of application-bypass reduction. In chapter 4 we present our NIC-based framework for offload of user code modules. Finally, in chapter 5 we draw conclusions and discuss future work.

CHAPTER 2

OVERVIEW OF MYRINET AND MPICH-GM

2.1 Myrinet and GM

GM [16] 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 the multiplexes traffic across these connections for multiple ports. This gives applications the advantage of reliable in-order message delivery without having to actually establish connections.

2.2 MPI and MPICH-GM

MPI [14] is a standard interface for message passing in parallel programs. MPICH [11] 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-assisted or NIC-based offload techniques. In order to illuminate the design challenges discussed in later chapters, we first highlight some of the relevant MPICH implementation details.

One such detail is the way in which MPICH handles the receipt of messages, both those which are expected by the application and those which are not. While there are multiple functions that may be used to receive messages with different semantics, we focus on the default case in this discussion. When a process is ready to receive a message, it calls the `MPI_Recv` function, providing criteria to identify the message to be received as well as an appropriate buffer for storage of the message. If a message arrives before a matching call to `MPI_Recv` has been made, MPICH allocates a temporary buffer, copies the message into the buffer and then adds it to the *unexpected queue*. When a process calls `MPI_Recv`, MPICH first searches the unexpected queue for a matching message. If a match is found, it simply copies the message from the unexpected queue to the buffer provided by the application. Otherwise, it polls the network until a matching message is received, at which point the message is copied into the application buffer and returned.

Another notable detail relates to the way GM uses memory when sending messages. GM can only send data located in memory which has been registered for DMA transfers (*pinned*). Pinning and unpinning memory requires relatively expensive system calls. Since this sort of situation is common in user-level protocols, MPICH uses two send modes to efficiently handle both small and large messages. Small messages are sent using *eager* mode and large messages are sent in *rendezvous* mode. Basically, in eager mode message data is copied into a pre-pinned buffer for sending, while in rendezvous mode the message data is pinned in-place and sent from its original location. Eager mode eliminates the overhead of pinning for small messages at the expense of a memory copy, while rendezvous mode eliminates the overhead of copying for large messages at the expense of pinning memory.

CHAPTER 3

APPLICATION-BYPASS REDUCTION FOR LARGE-SCALE CLUSTERS

3.1 Background and Motivation

When we visualize running a parallel application on a cluster, it's common to think of all processes involved in the computation executing in a synchronous manner. For example, it's natural to assume that all processes will start at the same instant. However, in reality processes may become unsynchronized or *skewed*. This may happen for a variety of reasons including heterogeneous systems consisting of nodes with different processing capabilities, varying communication latencies between nodes, unbalanced or asymmetric code where different nodes may be assigned tasks requiring different amounts of processing resources, and random effects such as interrupts or contention for resources between multiple processes on a given node. Process skew becomes more prevalent as the size of a cluster grows and more opportunities for unpredictable delays are introduced.

Process skew is an important factor in the performance of parallel applications, especially those involving collective communications. Collective communications [12][7] often by their nature introduce implicit synchronization in the form of communication dependencies between processes. Under conditions of process skew, these dependencies can cause some processes to wait idly for other processes to catch up. This results in ineffective CPU utilization, wasting resources that might otherwise be dedicated to useful processing.

Reduction is a common example of such a collective communication. In the default MPICH [11] implementation of reduction, each process involved in the communication calls the `MPI_Reduce` function. Internally, MPICH organizes the processes into a logical tree. Processes wait to receive messages from their children before sending a result to their parent and completing `MPI_Reduce`. So `MPI_Reduce` synchronizes the participating processes, requiring each process to wait until all processes below it in the logical tree have completed `MPI_Reduce`. This synchronization is not necessary for the majority of the processes involved in the communication. It would be more efficient if the reduction operation could make progress independently of the application, allowing parent processes to continue with other work until their child processes have sent their data. This technique is known as *application bypass* [2] and is discussed in detail in the next section.

This chapter describes our design and implementation of an application-bypass version of the reduction operation in MPICH-GM [16]. We discuss the design challenges that we faced in the process of adapting the synchronous infrastructure provided by the default MPICH implementation to support our more flexible application-bypass operation. These challenges include extending the MPICH communication progress mechanism, maintaining intermediate reduction state, handling messages that arrive both earlier and later than normally expected and minimizing the overhead associated with the mechanisms that we chose to support asynchronous processing.

We have evaluated our implementation and found a factor of improvement of up to 5.1 under conditions of process skew. Furthermore, we have observed that the factor of improvement increases with system size, indicating that our application-bypass implementation is more scalable and skew-tolerant than the default non-application-bypass version.

The remainder of this chapter is organized as follows. In the next section we discuss the basic concepts of application bypass and how they can be applied to the reduction operation. The design challenges we encountered while implementing our application-bypass reduction operation are discussed in Section 3.3 and then the details of our implementation are covered in Section 3.4. In Section 3.5 we evaluate the performance of our implementation. Finally, in Section **??** we discuss related work.

3.2 Basic Concepts behind Application-Bypass Reduction

The goal in coding an application-bypass operation is to eliminate the need for applications to block while the operation makes progress. This sort of optimization is ideal for operations such as broadcast and reduction where there is no implied global synchronization between processes. It could even benefit synchronizing operations like barrier and all-reduce if they are implemented in a split-phase manner.

In MPICH, each process involved in a reduction calls the `MPI_Reduce` function at the application level to initiate the operation. Internally, `MPI_Reduce` organizes the processes into a logical binomial tree and the operation is then performed using point-to-point communication between processes. Fig. 3.1 illustrates such a tree 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 reduction.

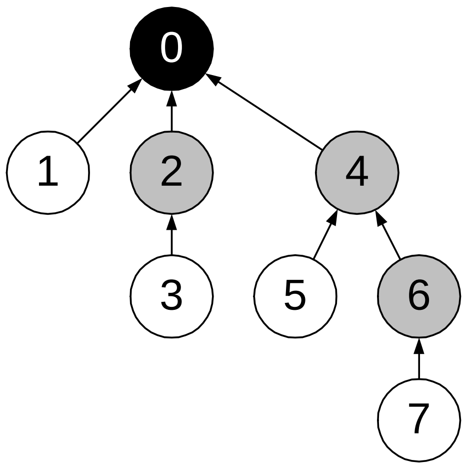


Figure 3.1: Example binomial tree used to organize point-to-point communications between eight processes involved in a reduction operation. The root node is shown in black, internal nodes are colored gray and leaf nodes are shown in white. The arrows between processes indicate the direction of messages involved in the reduction.

When calling `MPI_Reduce`, each process provides a buffer containing its input for the operation. The root process also provides an additional buffer to accept the operation results. While leaf processes simply need to send their input to their parents, all other processes must wait to receive results from their children before they can perform the arithmetic operation associated with the reduction. This organization introduces dependencies between processes. When processes become skewed, those which are parents in the tree may have to wait idly on children that are late. Application-bypass techniques eliminate the synchronous nature of these dependencies so that parent processes can proceed in spite of the late arrival of children at the `MPI_Reduce` point.

The default MPICH implementation of the reduction operation could be enhanced using application-bypass techniques. The processes that can benefit from such enhancements are the internal ones. The behavior of the leaf processes need not be optimized as their only action is to perform a send to their parent. Similarly, the behavior of the root node can not benefit from optimization. Per the MPI standard, `MPI_Reduce` is implemented in a blocking fashion, so the root process expects the function call to return only when the reduction has completed across all processes. However, a split-phase implementation would enable optimization of the root node as well.

Fig. 3.2 shows example time lines for a reduction involving four processes. Each large vertical arrow represents the progress of the operation for a given process. The portions of the large arrows shown in gray represent CPU utilization associated with the reduction. The small horizontal arrows represent point-to-point messages associated with the reduction. In this example, node zero is the root node, nodes one and three are leaf nodes and node two is an internal node. Note that the processes are slightly skewed, with nodes zero and two starting the reduction at approximately the same time, node one following shortly thereafter and node three being the last to begin.

Fig. 3.2(a) shows the default non-application-bypass implementation. We can see that node two must wait idly on node three, which is late due to process skew. Fig. 3.2(b) illustrates the application-bypass implementation. Here we can see that node two's reduction processing has been split into two components. The first portion is performed synchronously and is associated with the call to `MPI_Reduce`. Instead of

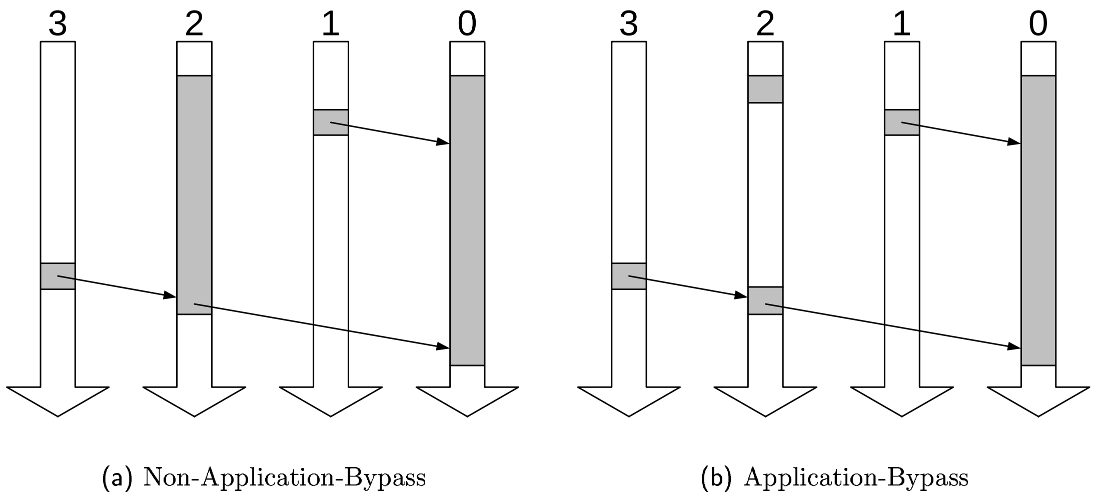


Figure 3.2: Example time line for four processes involved in a reduction operation. The large vertical arrows represent the progress of the operation for each process. The gray portions of the large arrows represent CPU utilization associated with the reduction. Each small horizontal arrow represents a point-to-point message involved in the reduction.

waiting for node three, node two returns from `MPI_Reduce` and delegates the remainder of the reduction to asynchronous processing. The reduction operation resumes only when the message from node three finally arrives, and the time in between the synchronous and asynchronous portions can be utilized for other processing.

Under conditions of process skew, application-bypass techniques can reduce both the amount of time that processes spend waiting on each other and the amount of implicit synchronization associated with collective operations. These improvements can help reduce the amount of CPU utilization associated with the operation and increase the opportunity for overlap of communication and computation. The benefits of application-bypass operations are especially relevant in large-scale clusters where skew between processes becomes inevitable.

3.3 Design Challenges

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

3.3.1 Communication Progress

In order to support splitting the processing of reduction operations into synchronous and asynchronous components, some mechanism must be used to trigger the asynchronous processing upon receipt of late messages. By default, MPICH relies on the application layer to make communication progress. When an application makes calls to functions in the MPICH library, the progress engine is triggered to check for incoming messages and either match them to posted receives or queue them for later consumption. This mechanism is clearly inadequate for our purposes as we want to decouple the application from the reduction communication.

One potential solution would involve using a dedicated thread to monitor incoming messages and activate the asynchronous processing as necessary. Another method would involve generating an interrupt upon the receipt of a late message. Both alternatives have benefits and disadvantages. The thread-based option would consume additional CPU resources while polling for late messages, but would not require the overhead of interrupts. The interrupt-based option would incur a certain amount of interrupt overhead with the arrival of late messages. However, this overhead would only occur when asynchronous processing is actually required, as opposed to the constant overhead of polling for late messages.

Based on our previous experience with the implementation of application-bypass broadcast [6], we decided to use an NIC-assisted, interrupt-based approach. Since interrupts incur a substantial performance penalty, this introduced another challenge in how to avoid the generation of unnecessary interrupts. For example, interrupts need not be generated while MPICH is already checking for receives within `MPI_Reduce`. They are also unnecessary if there are no outstanding children to be processed asynchronously. In this case, messages can be unexpected but not late. Also, note that interrupts are only required for internal nodes, as the root node must perform all of its processing synchronously and the leaf nodes have no children.

3.3.2 Maintenance of Intermediate State

Another requirement for splitting the reduction processing into synchronous and asynchronous components is the maintenance of intermediate reduction state. First, note that a parent node may have multiple children, each of which may be processed synchronously or asynchronously at different points in time. Therefore, we need to keep track of the running result of the reduction operation between the initial synchronous processing and potentially multiple periods of asynchronous processing.

Second, note that in addition to processing messages from children, internal nodes must also send their final result to their parent. However, this must not happen until all children have been processed. So we need a way to know when the processing of all children has completed and the send to the parent may be performed.

Also, if the last child processed is handled by the asynchronous portion of the code, then we need to be able to determine the appropriate parent associated with the reduction. The parent-child relationships between nodes can vary between reduction instances depending on which process is designated as the root of the reduction. A node's parent is calculated during the synchronous call to `MPI_Reduce` and must be recorded for potential use during asynchronous processing.

3.3.3 Handling Early Messages

We encountered another challenge involving the handling early or *unexpected* messages. The semantics for unexpected messages are simple in the default MPICH implementation. Because all reduction processing is performed synchronously, unexpected messages are simply those messages that arrive before the application calls `MPI_Reduce`. However, in our application-bypass implementation we need to perform some additional checking due to the asynchronous nature of the processing. First, as in the non-application-bypass case, the message must fail to match a receive associated with the synchronous processing in `MPI_Reduce`. Second, the message must also fail to satisfy a pending receive which is being managed asynchronously after exiting a call to `MPI_Reduce`. If the message matches a pending asynchronous receive, then it's actually a *late* message as opposed to an unexpected message, and must be handled appropriately as discussed below. Otherwise, the message is truly unexpected and must be saved for later processing.

3.3.4 Handling Late Messages

As mentioned above, *late* messages are those messages associated with a reduction operation that arrive after exiting a call to `MPI_Reduce`. These messages must be handled by the asynchronous component of our application-bypass implementation. So first, we need a way to differentiate these late messages from other messages and trigger the asynchronous processing. We also need to be able to match late messages to the proper reduction instance, as multiple reductions may be active concurrently and overlapped due to skew. For example, consider the eight-node case illustrated in Fig. 3.1. Assume that our application performs several reductions back to back and that process six is consistently late in performing its send to process four. Each time process six is late, process four will delegate the associated operation to the asynchronous component of the implementation. Since there are several reductions performed back-to-back, there may be several outstanding receives from process six, each associated with a separate reduction instance. So when process four finally receives a message from process six, it needs to be able to match it to the appropriate reduction instance in order to maintain correctness.

3.3.5 Reducing Frequency of Late Messages

Interrupts associated with incoming application-bypass messages are not necessary if MPICH is already checking for receives while inside `MPI_Reduce`. We explored a potential optimization involving the addition of a small delay before exiting `MPI_Reduce` in the case where all children had not been processed. By delaying, we hoped to allow receives from the outstanding children to complete within `MPI_Reduce` and thus avoid interrupts. The crucial decision here is how long to delay. If the delay is too short, then late children will not be able to catch up, but if the delay is too long, then we're basically back to a synchronous scheme.

We experimented with a simple scheme in which we calculated the delay based on the number of processes involved in the reduction. A more sophisticated scheme could be constructed by taking into account the position of the parent and child processes in the logical tree. However, such calculations become quite speculative when random skews are involved and we are still investigating these issues.

3.4 Our Implementation

In this section we present the details of our implementation of application-bypass reduction. The section is organized as follows. First we discuss the changes that we made to the MPICH infrastructure to support application-bypass processing. Next we walk through both the synchronous and asynchronous components of the processing to illustrate the associated logic.

3.4.1 Infrastructure Changes

First, we modified GM 1.5.2.1 to include the ability to generate signals from within the NIC-based control program. As previously mentioned, we took a NIC-assisted approach and made several small changes to the NIC to enhance to upper-level libraries. We added a new collective packet type for use when sending messages related to application-bypass reduction. In addition, we added the capability to disable and enable signals from within the MPICH layer via calls to the GM library. These modifications are used together to minimize the number of signals that are generated. Signals are only generated by the NIC for messages of the new collective packet type, isolating them to only those situations where they are actually required. We initialize MPICH with signals in a disabled state, as initially there can not be any outstanding reductions. We only enable signals when outstanding reductions need to be processed asynchronously, and then again disable signals as soon as all outstanding reductions have been completed. Details on exactly how and when we choose to enable and disable signals are included below. When a signal is received by the host, it triggers the activation of the MPICH communication progress engine so that asynchronous processing may be performed.

The remainder of the changes were made to MPICH-GM version 1.2.4..8a. As mentioned in Section 3.3, we needed to develop a strategy for handling both unexpected and late messages. We explored one solution which involved using the non-blocking versions of the MPICH send and receive primitives for internal point-to-point communication within the collective call to `MPI_Reduce`. The default MPICH implementation uses the blocking versions of the send and receive primitives. By switching to the non-blocking versions we hoped to gain the extra control required to support asynchronous processing, while still re-using as much as possible of the existing MPICH infrastructure. While this solution did enable reuse of the existing MPICH message matching and queuing mechanisms, it also required the allocation and management of additional buffers for use in the non-blocking receives. In addition, it introduced extra complexity associated with trying to use the MPICH infrastructure in ways other than those in which it was intended to be used.

We instead chose to implement our own *unexpected queue* specifically for application-bypass messages. This enables us to manage unexpected messages in an efficient manner, reducing the maximum number of required message copies from two to one. It also prevents our optimizations from affecting the common case of non-collective point-to-point communications, which are left to the default MPICH mechanisms. In addition to the unexpected queue, we also added a *descriptor queue* to manage descriptors containing state information for pending reductions. Each descriptor includes the intermediate result of the reduction operation, the identity of the parent process to which results should be sent and a list of children from which receives are pending. The child list is also used for matching late messages to the appropriate entry in the descriptor queue (i.e. the appropriate reduction instance). Details on how both queues fit into our implementation are provided in the following subsections.

3.4.2 Synchronous Processing

Recall that in MPICH, each process involved in a reduction calls the `MPI_Reduce` function at the application level to initiate the operation. The synchronous component of our application-bypass processing takes place within this call to `MPI_Reduce` as described below.

First, we determine whether or not to perform a given reduction in application-bypass mode. This decision is made based on the position of the node in the binomial tree as well as the size of the message. If the node is a root or leaf node, we simply perform a standard non-application-bypass reduction. As discussed in Section 3.2, application-bypass techniques only apply directly to internal nodes, so we choose to leave the processing of root and leaf nodes to the default MPICH mechanisms. We also fall back to performing a standard non-application-bypass reduction if the size of the message is beyond the limit of eager-mode processing. We have not yet investigated a rendezvous-mode implementation due to the additional complexities involved in buffer management.

Assuming the reduction is being processed in application-bypass mode, we proceed as illustrated in Fig. 3.3. First we ensure that signals are disabled, as we will be explicitly making communication progress while inside `MPI_Reduce`. Next, we build

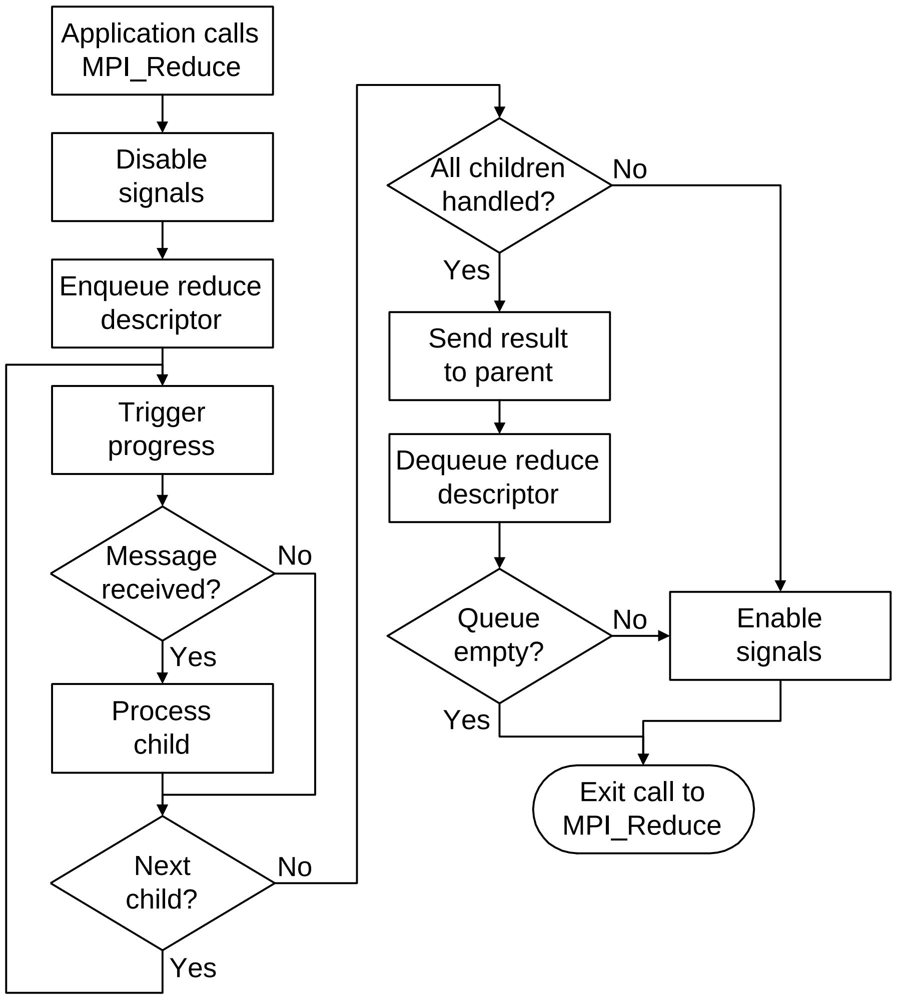


Figure 3.3: Synchronous component of application-bypass reduction processing. This logic does not apply to root and leaf nodes, which are processed using the standard non-application-bypass code. The *Message Received* test simply checks to see if an unexpected message has been received from a given child, as opposed to blocking for a receive.

a descriptor containing the intermediate state needed to manage the reduction operation or operations as well as a list of the child or children of the current process. This descriptor is added to a queue of outstanding reductions.

From this point onward, the reduction may actually be processed in parallel by both `MPI_Reduce` and our asynchronous code. The logic within `MPI_Reduce` basically walks through the list of children in the reduce descriptor, checking for unexpected messages and making communication progress if pending receives are detected. When progress is made, the asynchronous portion of the code processes expected and late messages as detailed in the next subsection. If an unexpected message from a child is encountered, the corresponding operation is performed and the associated descriptor is updated to reflect the fact that the child has been processed. If all children are processed within `MPI_Reduce`, the final result is sent to the parent and the descriptor is removed from the queue. If at the end of `MPI_Reduce` the descriptor queue is not empty, then signals are enabled.

Note that even though unexpected messages must be buffered in our unexpected queue, they are processed directly from the queue in `MPI_Reduce`, eliminating the need for a second copy to a buffer associated with a point-to-point receive as in the default MPICH implementation. This results in a 50% reduction in the number of message copies for unexpected messages.

3.4.3 Asynchronous Processing

In addition to the synchronous processing performed within `MPI_Reduce`, we also added code to enable pre-processing of incoming packets before they are examined by the MPICH matching and queuing mechanisms. This pre-processing comprises the asynchronous portion of our implementation.

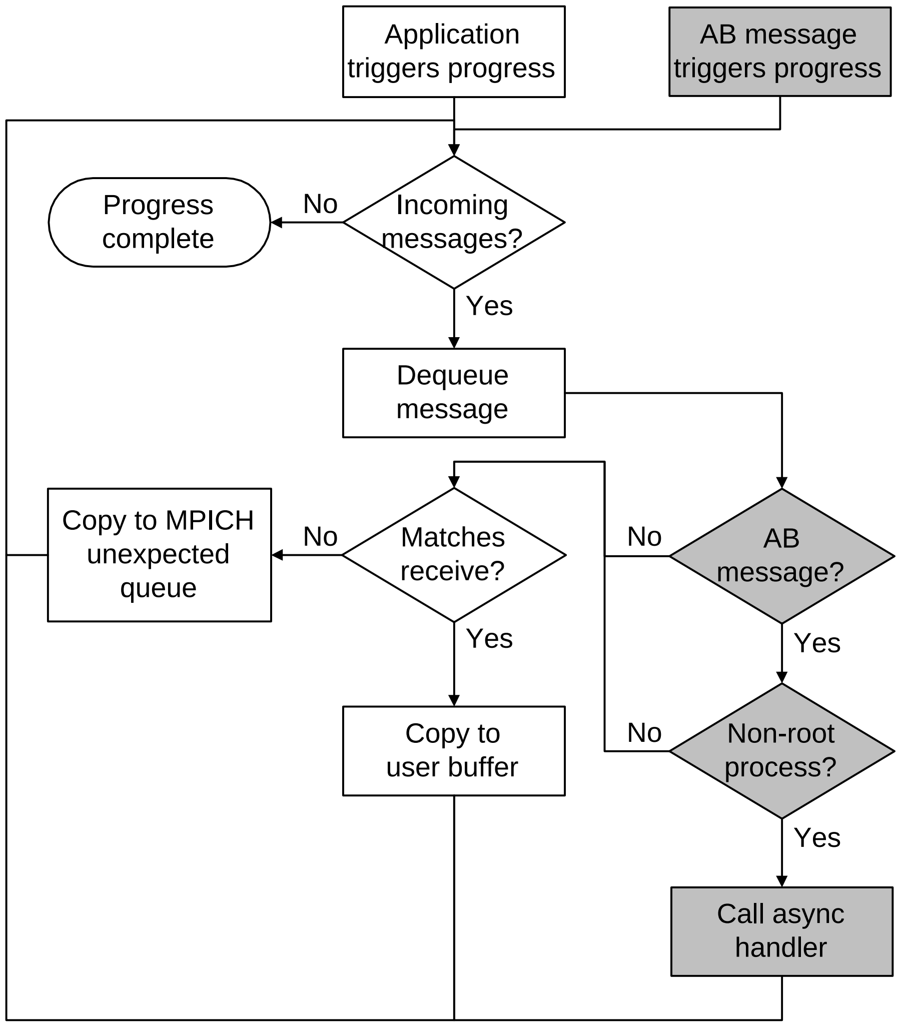


Figure 3.4: MPICH communication progress logic. The default non-application-bypass logic is shown in white, while the new application-bypass logic is colored gray.

The MPICH communication progress logic is illustrated in Fig. 3.4. Assuming signals are enabled, the arrival of an application-bypass reduction packet generates a signal that triggers communication progress. (Note that if a signal happens to occur while progress is already underway, it is simply ignored.) After the progress engine dequeues the incoming message, it checks to see whether the current process is the root of the reduction instance with which the message is associated. If so, then no extra asynchronous action is taken. This is because the behavior of the root process is necessarily synchronous, so we can utilize the default synchronous point-to-point communications. In such a case where we decide not to process a packet, it is handled by the default MPICH mechanisms.

If the current process is not the root, the progress engine hands the packet off to a routine which handles the asynchronous component of the reduction processing. This logic is illustrated in Fig. 3.5. First, the descriptor queue is searched to see if the sender of the packet matches an outstanding receive. If so, the appropriate reduction operation is performed and the descriptor is updated to reflect the fact that the child has been processed. If all children have been processed, the final result is sent to the parent and the descriptor is removed from the queue. If this action renders the queue empty (i.e. there are no outstanding reductions) then signals are disabled.

If the packet fails to match an entry in the descriptor queue, the message is added to our custom unexpected queue for later synchronous processing. Note that if the message is expected it is processed directly from the buffer associated with the packet, eliminating the need to copy it into a buffer associated with a point-to-point receive as in the default MPICH implementation. This results in a 100% reduction in the number of message copies for expected and late messages.

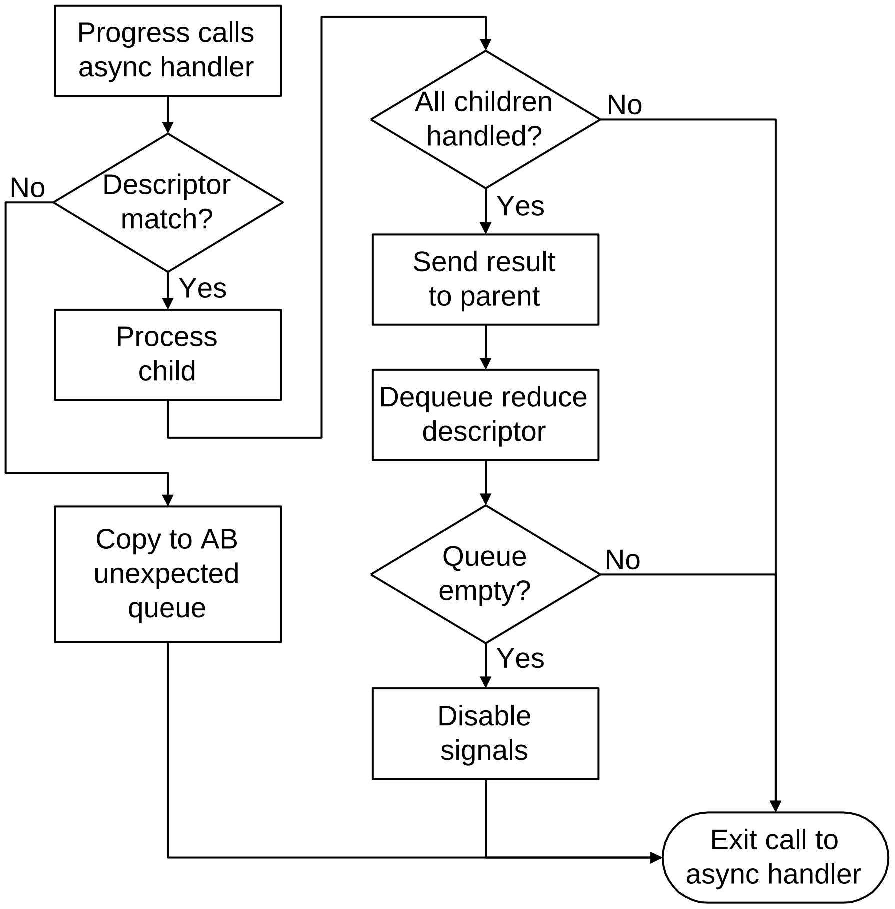


Figure 3.5: Asynchronous component of application-bypass reduction processing. The *Descriptor Match* test succeeds if an outstanding reduction instance is waiting for a message from the sender of the packet.

3.5 Experimental Results

We evaluated our implementation on a 32-node cluster consisting of 16 quad-SMP 700-MHz Pentium-III nodes with 66-MHz/64-bit PCI, and 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. Four of the 1-GHz nodes contained PCI64C network interface cards with 200-MHz LANai 9.2 processors, while the remaining 28 nodes utilized PCI64B cards with 133-MHz LANai 9.1 processors. Our application-bypass implementation is based on MPICH 1.2.4..8a over GM 1.5.2.1, and all comparisons were performed against the original, unaltered software packages of the same versions.

The MPICH-GM distribution provides a script to launch MPI applications. This script accepts a list of machines on which the application should be executed. We configured the list of machines such that the nodes from each of the two groups of 16 are interlaced, thereby ensuring a balanced mix of nodes for all system sizes used in our evaluations. Although our 32-node cluster is heterogeneous, we compared it to both of the groups of homogeneous machines separately for system sizes up to 16 nodes and observed nearly identical results. The SMP differences between the two classes of machines are mitigated by the fact that we only utilize one processor per node in our experiments. The differences in PCI and NIC capabilities are not much of a factor either, as our reduction operations involve fairly small amounts of data.

We created a pair of microbenchmarks for use in evaluating our implementation. The first microbenchmark measures the average per-node CPU utilization associated with performing a reduction under varying amounts of process skew. The second microbenchmark measures the total time (latency) to perform a reduction in the absence of process skew. As discussed in Section 3.2 and illustrated in Fig. 3.2, CPU utilization is the metric that our application-bypass implementation aims to improve. Skew will inevitably increase the overall latency, but if we can reduce the CPU utilization, additional computation may be performed while the reduction completes asynchronously.

The latency benchmark works as follows. First, we determine the one-way message latency between the root node and the node which is furthest away from the root in the logical tree (the *last node*). Next, we time a series of 10,000 reductions and take the average, using a barrier to separate iterations. We start timing just before the last node begins the reduction. Then, when the root node completes the reduction, it sends a notification message to the last node, which stops timing and subtracts off the one-way latency associated with the notification message to determine the total reduction latency. The benchmark is repeated for varying system and message sizes.

For the CPU utilization 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 so that the CPU utilization associated with asynchronous processing may be captured. Next, we perform a series of 10,000 reductions and take the average across all nodes, using a barrier to separate iterations.

Within each loop iteration, the timing measurements are taken as follows. We first start timing, then introduce a random amount of delay between zero and the maximum delay, perform the reduction, 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 reduction latency. The intent here is to be sure to delay long enough to capture all asynchronous processing in the overall time measurement.

The remainder of this section is organized as follows. First we present CPU utilization results under conditions of process skew. These are the common conditions in large-scale clusters. Our solution is designed for such scenarios and we can clearly see its benefits over the default non-application-bypass implementation. Next, we present CPU utilization and latency results without process skew. Such conditions are very optimistic for large-scale clusters. Note that this is the worst-case scenario for our implementation, where we see all of the overhead involved in the application-bypass techniques. However, even under these worst-case conditions, we begin to see the benefits of our implementation as increased system and message sizes naturally introduce process skew.

3.5.1 Results with Process Skew

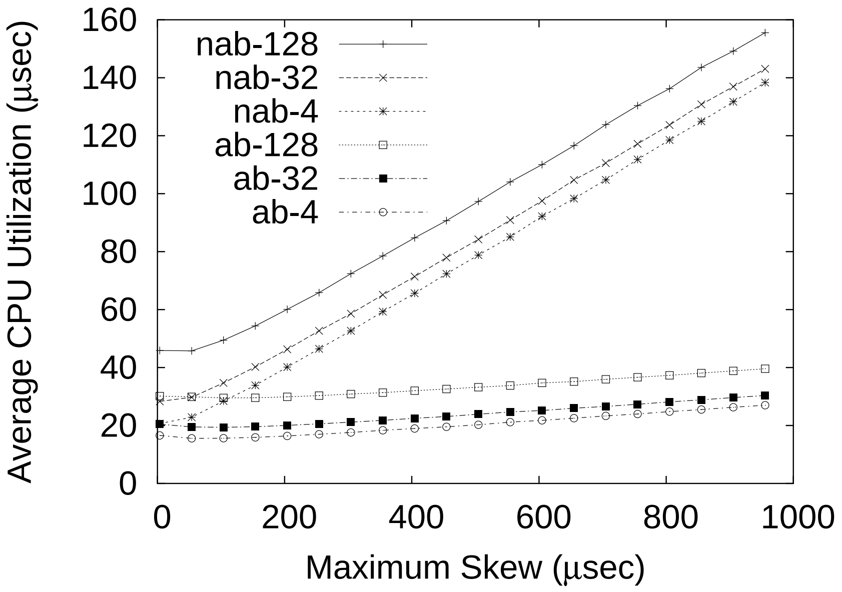
Fig. 3.6(a) shows the results of the CPU-utilization benchmark for 32 nodes with increasing amounts of process skew and double-word messages of 4, 32 and 128 elements. We can see that the application-bypass implementation consistently outperforms the non-application-bypass implementation for all combinations of skew and message size. As the amount of skew increases, the non-application-bypass implementation spends more and more time polling the network for messages from late child nodes. However, the application-bypass implementation simply notes that there are pending receives from these late child nodes and then processes the messages asynchronously whenever they finally arrive. The overhead associated with polling in the non-application-bypass implementation quickly outweighs the overhead due to signals in the application-bypass implementation. Fig. 3.6(b) shows a maximum factor of improvement of 5.1 for four-element messages when the maximum skew is 1,000 μs.

Fig. 3.7(a) shows the results of the CPU-utilization benchmark for 2, 4, 8, 16 and 32 nodes with a maximum process skew of 1,000 μs and double-word messages of 4, 32 and 128 elements. These results confirm that the trends demonstrated in Fig. 3.6 apply for varying numbers of nodes. Again, the application-bypass implementation consistently outperforms the non-application-bypass implementation, with Fig. 3.7(b) showing a maximum factor of improvement of 5.1 for 32 nodes and four-element messages. Furthermore, we can see that the factor of improvement increases with the number of nodes, demonstrating the enhanced scalability of the application-bypass implementation.

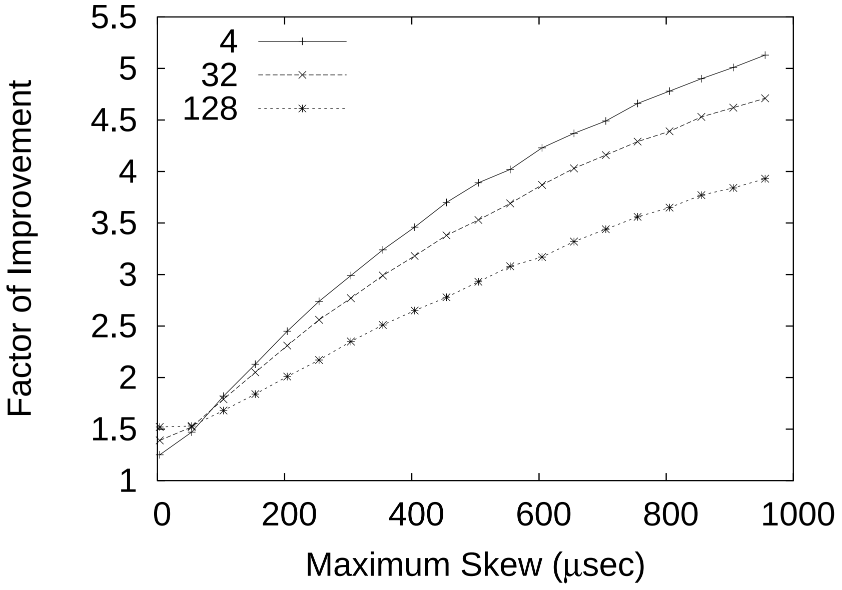
Note that in both cases, the factor of improvement is greatest for small message sizes. This is encouraging, as profiling efforts conducted by Moody et. al.[15] have shown that in typical large-scale parallel scientific applications, 95% of all reductions are performed on three or less elements and 100% typically use less than eight elements.

3.5.2 Results without Process Skew

Fig. 3.8 shows the results of the CPU-utilization benchmark without process skew for 2, 4, 8, 16 and 32 nodes and double-word messages of 4, 32 and 128 elements. We can see that as the number of nodes increases, the performance of

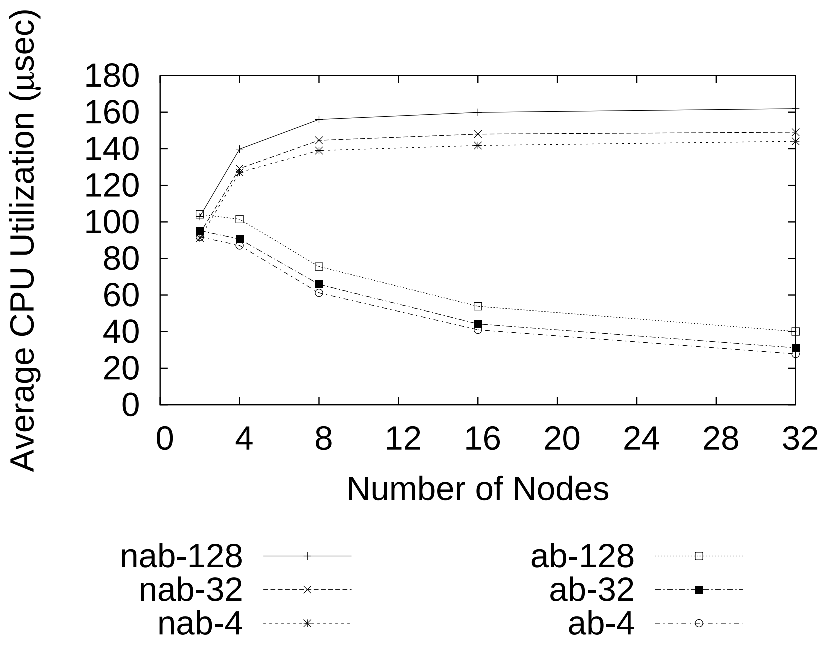


(a) Average CPU Utilization

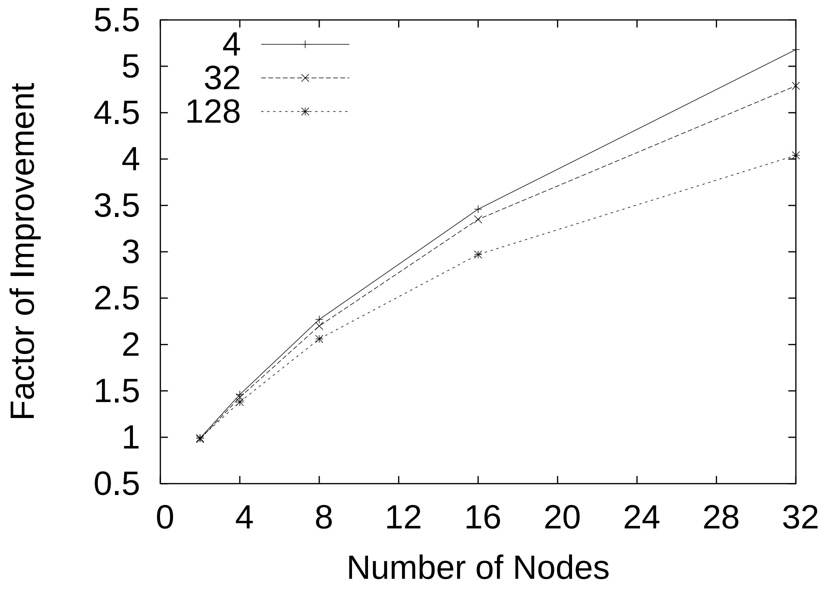


(b) Factor of Improvement

Figure 3.6: Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction for 32 nodes with varying process skew and 4, 32 and 128-element double-word messages.



(a) Average CPU Utilization



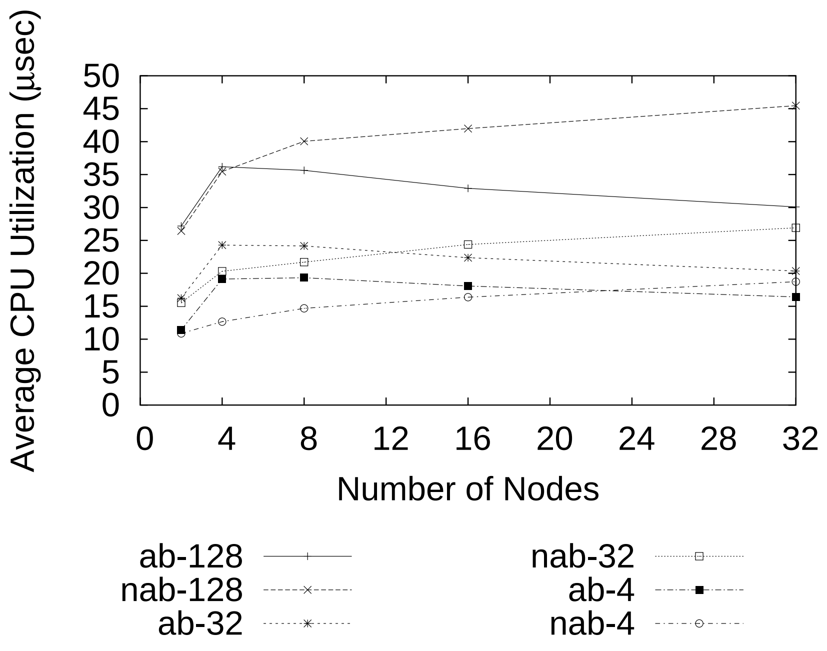
(b) Factor of Improvement

Figure 3.7: Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction for 2, 4, 8, 16 and 32 nodes with maximal process skew and 4, 32 and 128-element double-word messages.

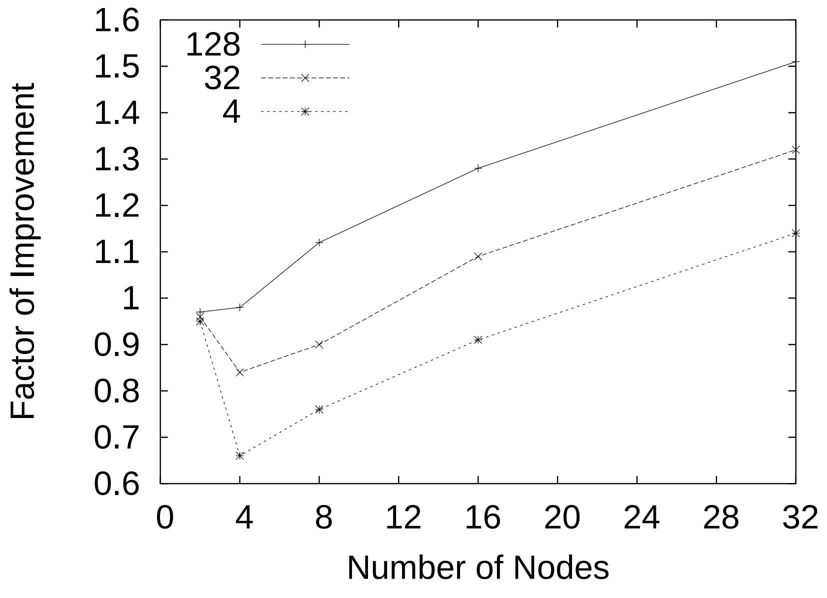
the application-bypass implementation improves. Even for our relatively small 32-node cluster, the application-bypass implementation eventually outperforms the non-application-bypass implementation for all message sizes. Fig. 3.8(b) shows a maximum factor of improvement of 1.5 for 32 nodes and 128-element messages.

Clearly, the application-bypass implementation scales with system size while the non-application-bypass implementation does not scale. Even though we are not introducing artificial process skew, the effects of naturally-occurring skew appear as the number of nodes involved in a reduction operation increases. We also see that the application-bypass implementation begins to outperform the non-application-bypass implementation at smaller numbers of nodes as the message size increases. Larger messages require more time for transmission and processing, introducing delays that accumulate corresponding to the number of descendants a node has in the binomial tree. These variations in processing requirements between nodes introduce process skew. Also, recall that the application-bypass implementation has the additional benefit of requiring less message copies than the default implementation.

Fig. 3.9 shows the results of the latency benchmark without process skew for 2, 4, 8, 16 and 32 nodes and single-element double-word messages. The results in Fig. 3.9(a) were taken on our heterogeneous 32-node cluster, while those in Fig. 3.9(b) were taken using only the 16-node cluster of 700-MHz machines. For small numbers of nodes, the latency of the application-bypass and non-application-bypass implementations are nearly identical. This is especially evident on the homogeneous cluster, where there is less potential for natural process skew. In this case, the application-bypass implementation actually does slightly better than the default implementation for a system size of four nodes. However, once the number of nodes increases past



(a) Average CPU Utilization

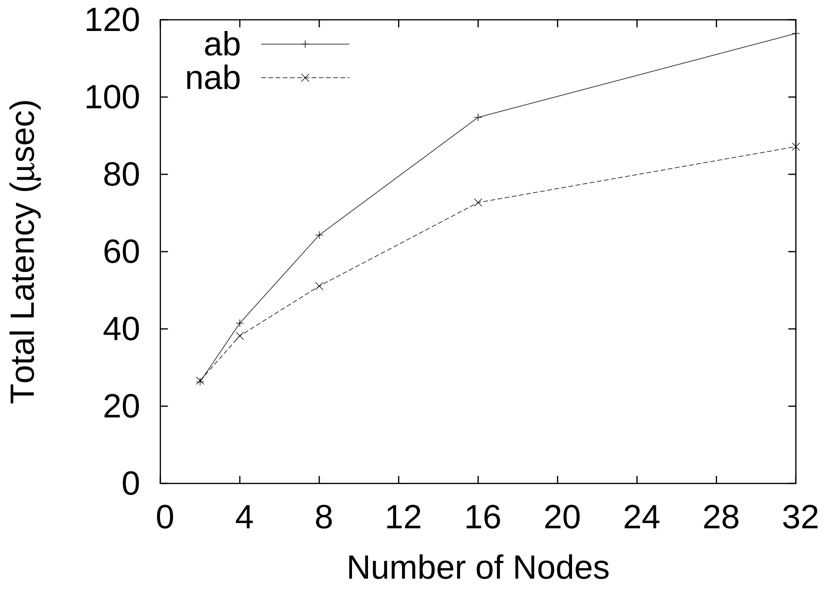


(b) Factor of Improvement

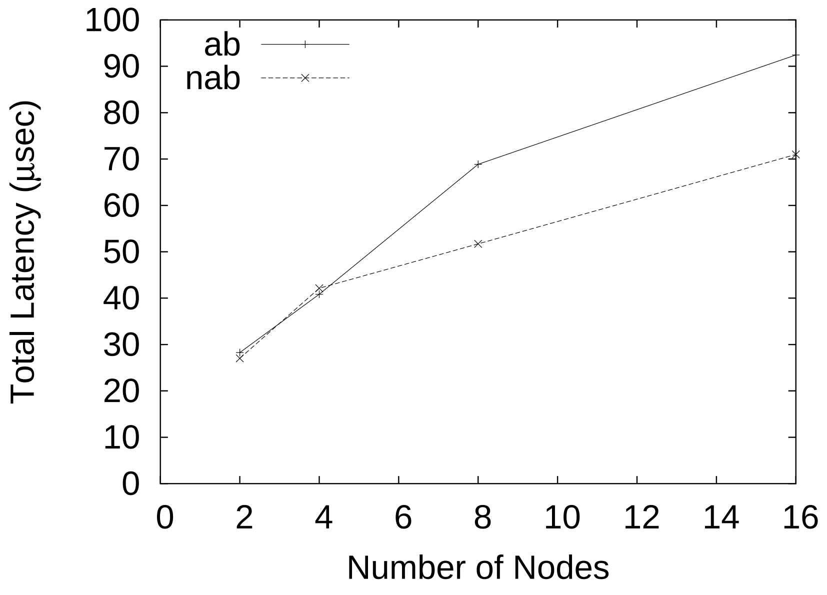
Figure 3.8: Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction without process skew for 2, 4, 8, 16 and 32 nodes and with 4, 32 and 128-element double-word messages.

four, the asynchronous component of the application-bypass implementation begins to be utilized as the processes become naturally skewed. This results in an increase in latency for the application-bypass implementation due to overhead from signals associated with late messages.

Figure 3.10 shows the results of the latency benchmark for 32 nodes with the number of message elements increasing from one to 128. Again, we see a difference in latency due to overhead from signals in the application-bypass implementation. However, note that this latency penalty stabilizes and remains fairly constant as the number of elements increases.



(a) Heterogeneous 32-Node Cluster



(b) Homogeneous 16-Node Cluster

Figure 3.9: Average latency of application-bypass (ab) and non-application-bypass (nab) reduction without process skew for 2, 4, 8, 16 and 32 nodes and single-element double-word messages.

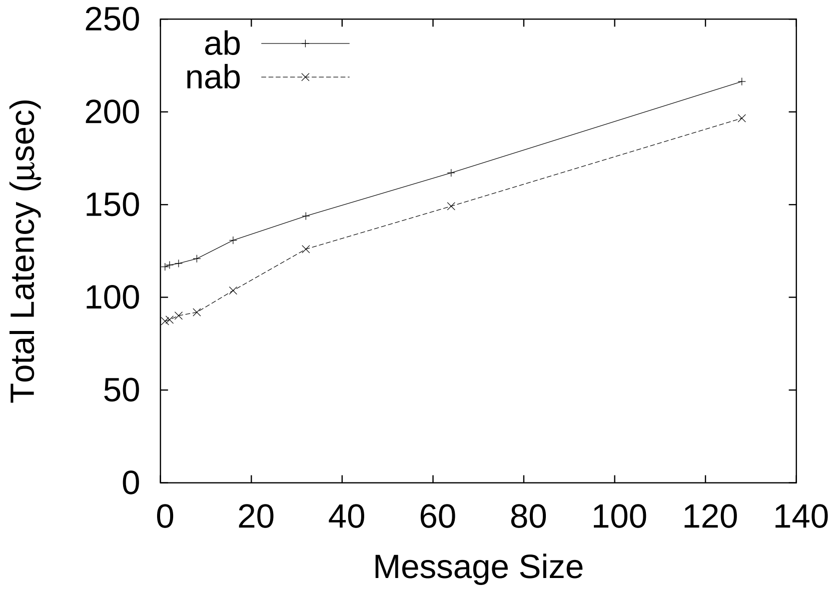


Figure 3.10: Average latency of application-bypass (ab) and non-application-bypass (nab) reduction without process skew for 32 nodes and varying-size double-word messages.

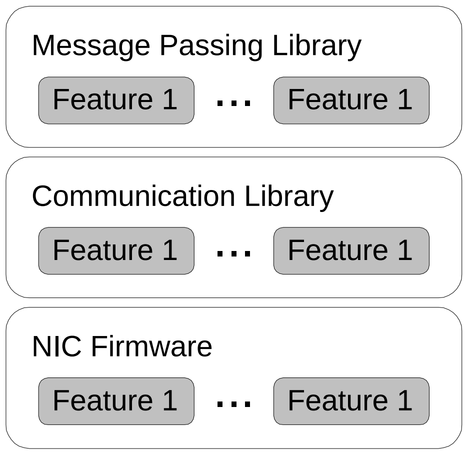
CHAPTER 4

NIC-BASED OFFLOAD OF DYNAMIC USER-DEFINED MODULES

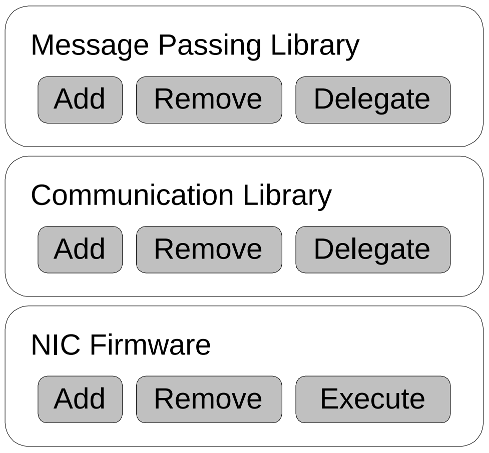
4.1 Background and Motivation

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 [7, 12] like multicast [3] and reduce [15] and synchronization operations such as barrier [4]. 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 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



(a) Static Offload



(b) Dynamic Offload

Figure 4.1: Static, hard-coded, ad-hoc offload of features to NIC vs. flexible framework for dynamic offload of user modules to NIC.

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 4.1 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 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 chapter 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 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 1.3 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 chapter is organized as follows. In the next section, we discuss the design challenges we encountered while implementing our framework and in section 4.3 we detail our implementation. In section 4.4 we evaluate the performance of our implementation. Finally, in section 4.5 we discuss related work.

4.2 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.

4.2.1 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. This base 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. 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. 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.

4.2.2 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 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 outside of the critical communication path. This sort of behavior is especially beneficial for collective-style communications.

4.2.3 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.

4.2.4 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.

4.2.5 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.

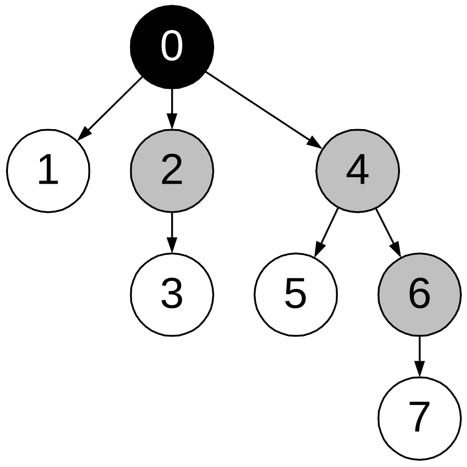
4.3 Our Implementation

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.

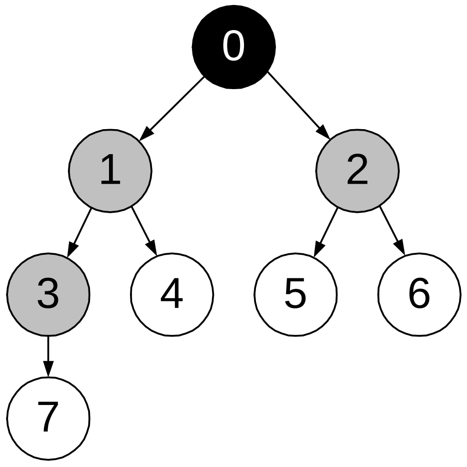
4.3.1 Overview

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.

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



(a) Binomial Tree



(b) Binary Tree

Figure 4.2: 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. The goal of the binomial tree is to maximize communication overlap, while the binary tree is much simpler to construct.

nodes into a logical tree and performs the broadcast using point-to-point communication between nodes. Figure 4.2 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 4.2(a) is a binomial tree, which is the tree utilized by the default MPICH implication 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 figure 4.2(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 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.

```
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 4.3: 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.

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 4.3 illustrates such a module, which 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 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 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 4.4 details

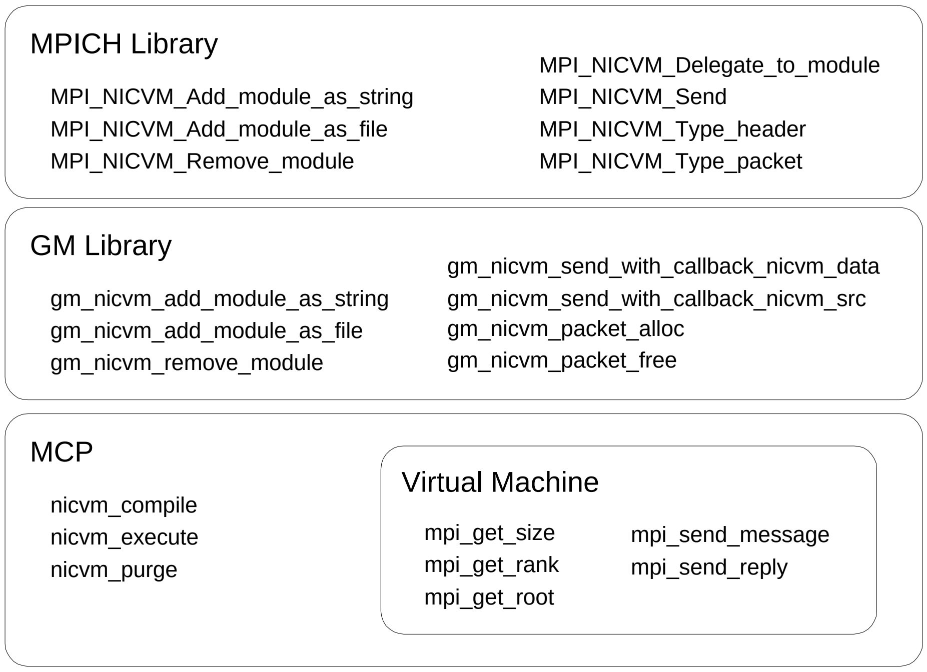


Figure 4.4: Various functions of the NICVM framework and where they fit in to the software layers. The functions listed inside the virtual machine are actually built into the language utilized by the user modules.

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.

4.3.2 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 [8] 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 [10] and bison [9], 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 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 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.

4.3.3 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 4.5 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 to and from the network and sending and receiving packets to and from the host via DMA. 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. 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 4.6. 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. Note that the user

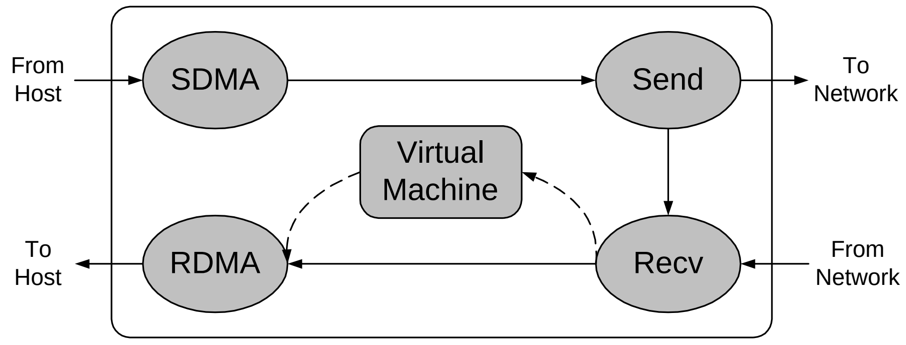


Figure 4.5: Integration of virtual machine into MCP. The ovals represent the different state machines which comprise the NIC logic. The solid arrows show the default path of packets through the MCP, while the dashed arrows indicate the path of packets containing NICVM source code or data. The arrow from the the Send state machine to the Recv state machine indicates loopback.

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 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.

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.

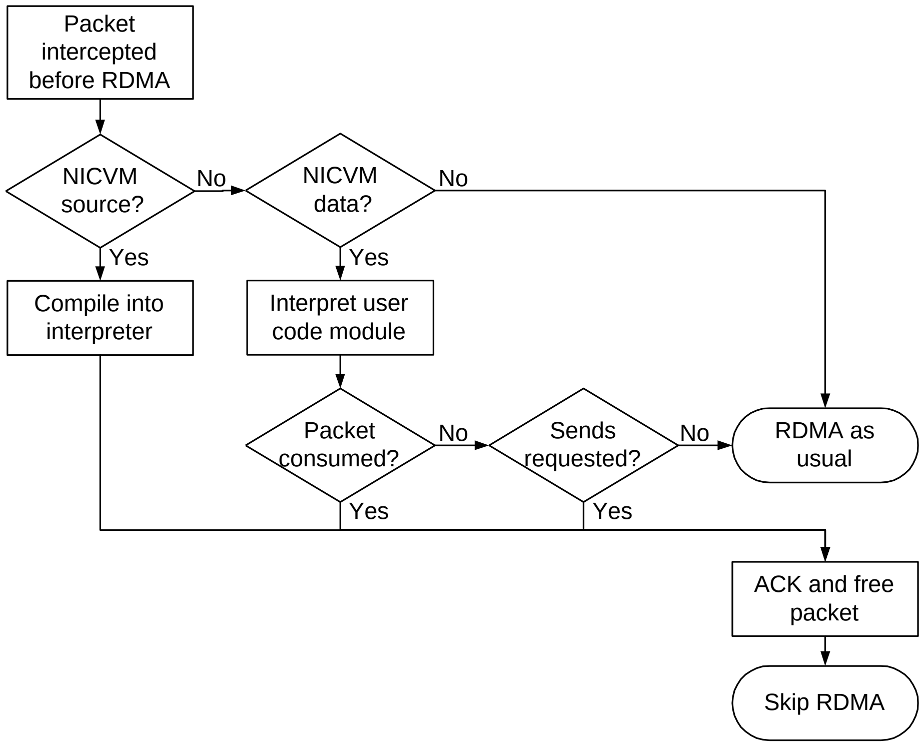


Figure 4.6: Synchronous component of NICVM packet processing. User code modules may either consume packets, request sends or both. If a packet is consumed, the receive DMA to the host is completely skipped. If one or more sends are requested, the DMA is delayed until after the sends have been completed.

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.

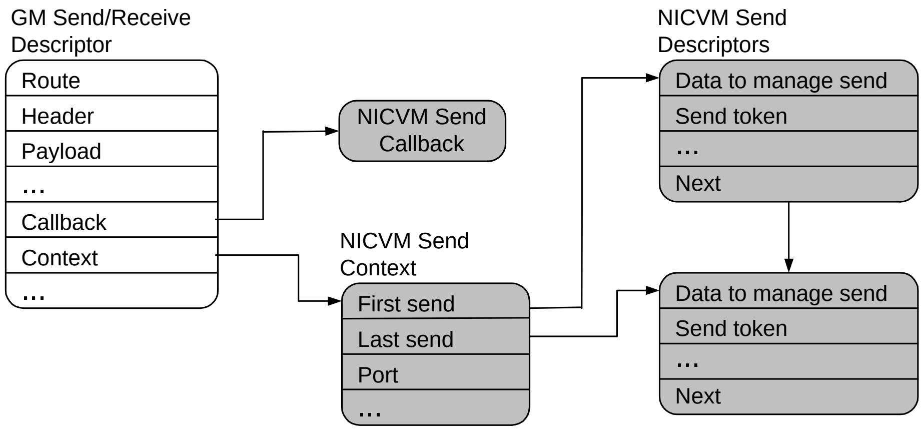


Figure 4.7: Relationship between a GM descriptor associated with a send or receive and the NICVM send context and NICVM send descriptors used to manage NIC-based sends. The items in white are part of the default GM implementation, while those in gray were added as part of the the NICVM framework.

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 4.7 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.

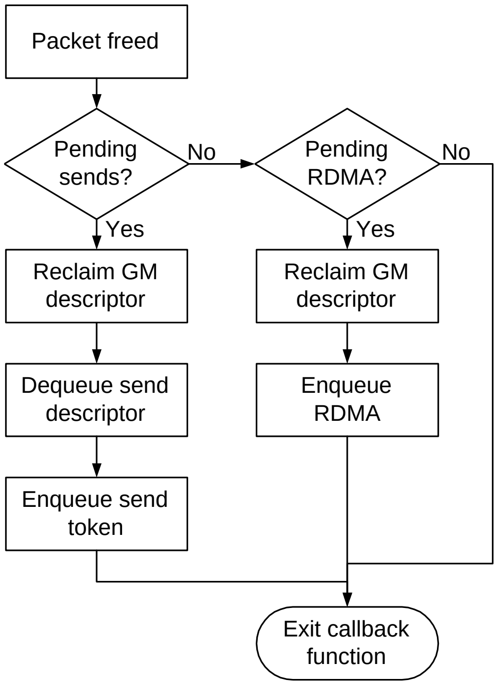


Figure 4.8: Asynchronous processing of sends requested by a user code module. This processing takes place within the NICVM send callback just after a GM send or receive descriptor is freed by the MCP.

After the user module terminates, we proceed in an asynchronous manner to perform the actual sends. This process is illustrated in Figure 4.8. Just after the GM descriptor associated with the original send or receive is freed, the MCP calls our

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.

4.3.4 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.

4.4 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 four MPI microbenchmarks for use in evaluating our framework. The first microbenchmark is a standard ping-pong test, where we measure the time required to send a message from one node (the ping node) to another (the pong node) and back. This time is then divided by two to determine the one-way communication latency. We mainly used this test to evaluate the overhead of our NICVM implementations compared to the base implementation of MPICH-GM.

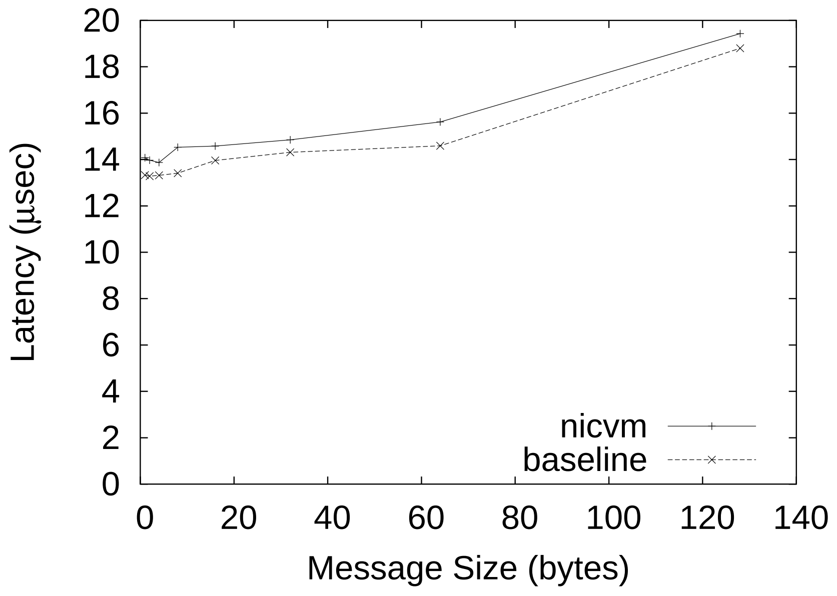
The second microbenchmark is similar to the ping-pong test but differs in a couple of major ways. We'll refer to this test as the *echo* microbenchmark to distinguish it from the standard ping-pong test. The main difference between the ping-pong test and the echo test is that the echo test measures the round-trip communication latency from one node (the initiator) to a second node (the responder). Also, in the echo test 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, a task that can easily be delegated to the NIC

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. The fourth 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 all microbenchmarks, we compare a baseline version using the standard MPI mechanisms to a customized version based on our NICVM framework.

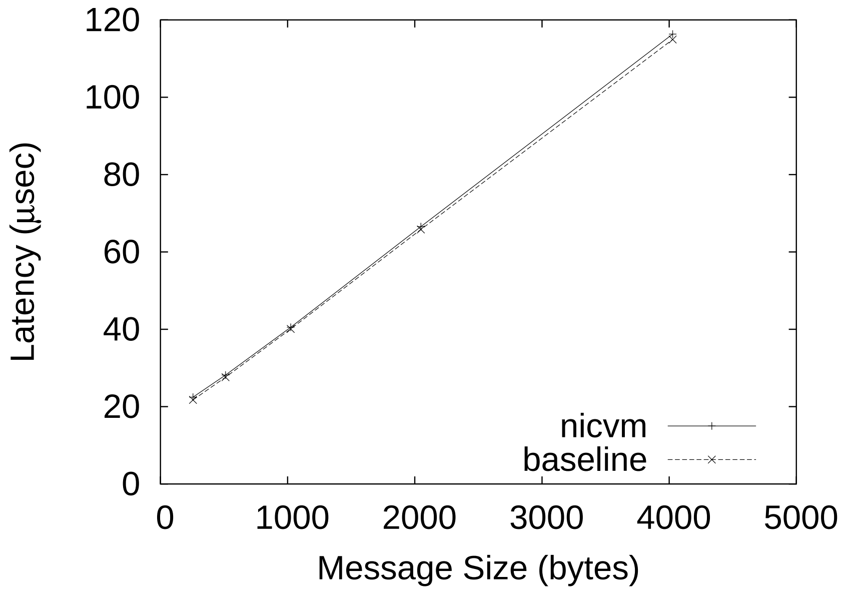
4.4.1 Overhead Results

We used the ping-pong microbenchmark to measure the base overhead of our modifications to MPICH-GM. In other words, without actually using the features of our framework we want to see whether or not their addition has affected base latency. 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 and the result is divided by two. We perform a series of 10,000 iterations and take the average, repeating the process for varying message sizes.

Figure 4.9 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. The average overhead observed across all message sizes is 0.8 μs. 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.



(a) Small Messages



(b) Large Messages

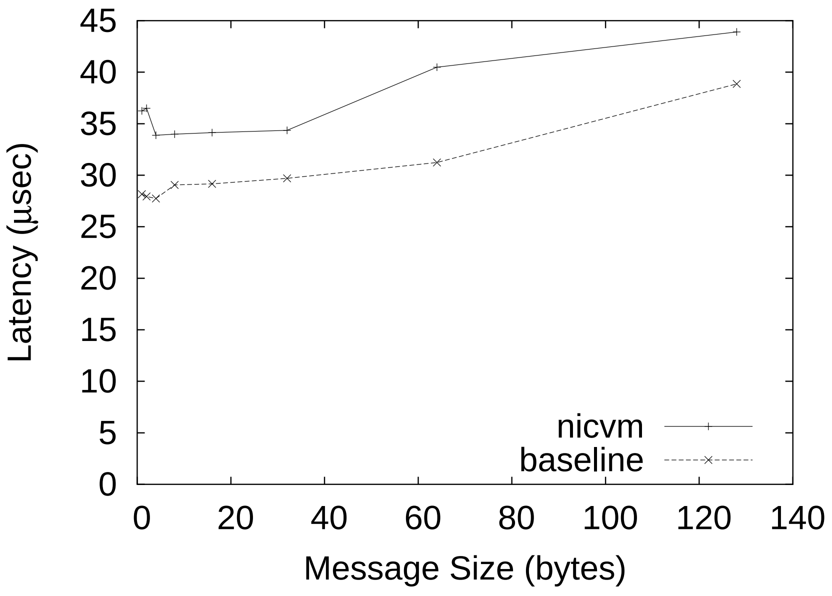
Figure 4.9: Latency of ping-pong microbenchmark for NICVM version of MPICH-GM (nicvm) and the default implementation (baseline). These results capture the base overhead of the NICVM modifications to the MPICH-GM software package.

4.4.2 Echo Results

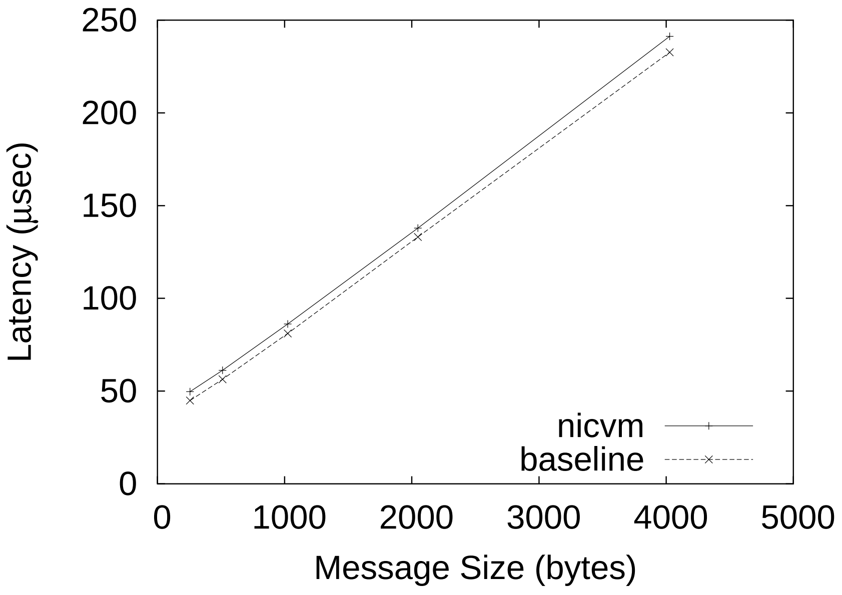
We use the echo microbenchmark to measure the base overhead of our mechanism to execute a user code module on the NIC. The echo microbenchmark works as follows. The initiator node starts a timer, sends a message to the responder node and waits for a reply. When the reply is received, the timer is stopped. We perform a series of 10,000 iterations and take the average, repeating the process for varying message sizes.

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 as usual. 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. We implemented two different NICVM versions of the test. The first version just uses a dummy no-op user module at the responder which passes the incoming packet through to the host. The host is then responsible for initiating the response, just as in the baseline case. This measures the base overhead of the NICVM mechanism to execute a user module. However, in the second version of the test, the user module actually takes responsibility for generating the echo reply without disturbing the host. This demonstrates the benefit of delegating the echo functionality to the NIC at the responder. All versions of the test were built against our customized NICVM implementation of MPICH-GM

Figure 4.10 shows the base overhead of the NICVM module-execution mechanism in action. One version of the benchmark has a no-op user module in the critical path



(a) Small Messages



(b) Large Messages

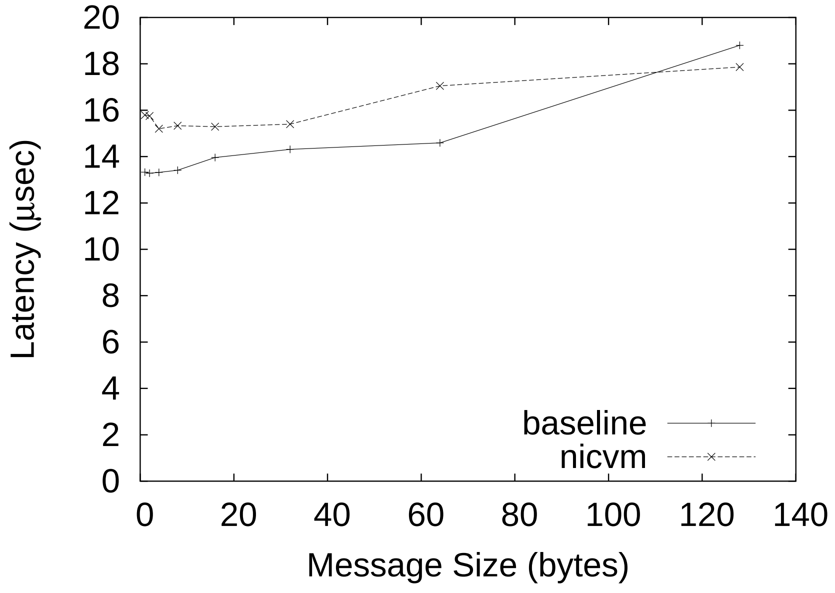
Figure 4.10: Latency of echo microbenchmark with (nicvm) and without (baseline) execution of no-op user module in critical path at pong node. These results capture the base overhead of the NICVM mechanism to execute a user module.

at the responder (nicvm) while the other does not (baseline). We can see that the overhead remains within several microseconds of the default implementation for all message sizes. More specifically, the average overhead across all message sizes is 7.6 μs.

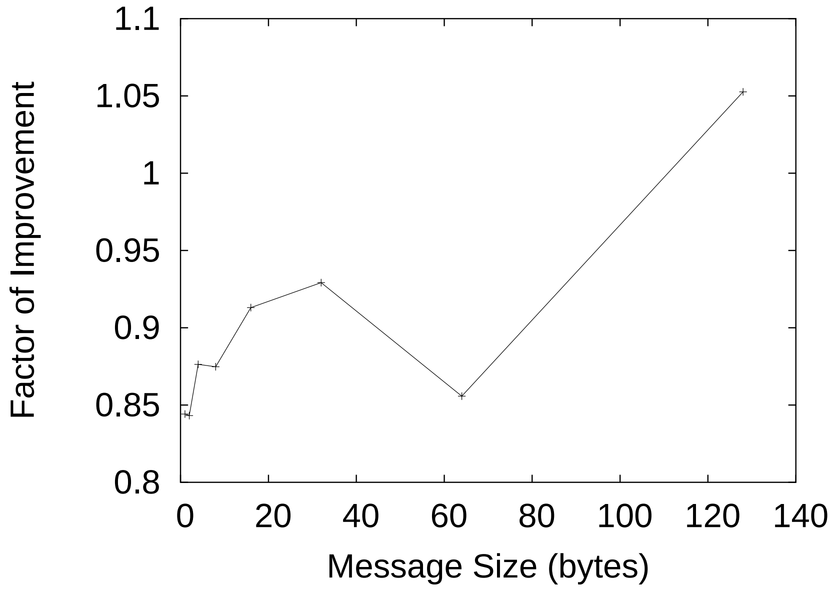
Figures 4.11 and 4.12 compare the performance of the NIC-based version of the echo microbenchmark (nicvm) to the host-based baseline version (baseline). For small messages we can see that the host-based implementation still outperforms the NIC-based implementation. This is because the overhead of user 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. Figure 4.12(b) shows a maximum factor of improvement of 1.5.

4.4.3 Broadcast Latency Results

The broadcast latency 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

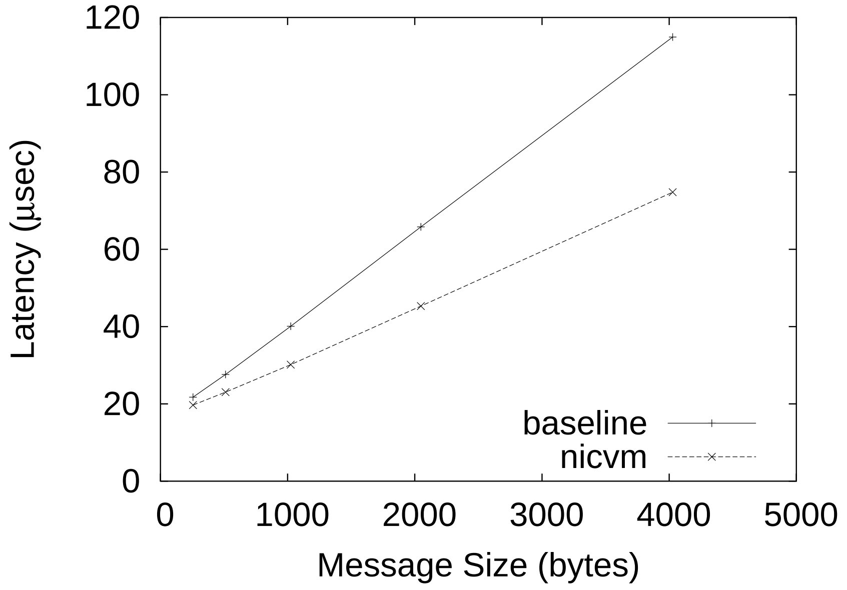


(a) Latency

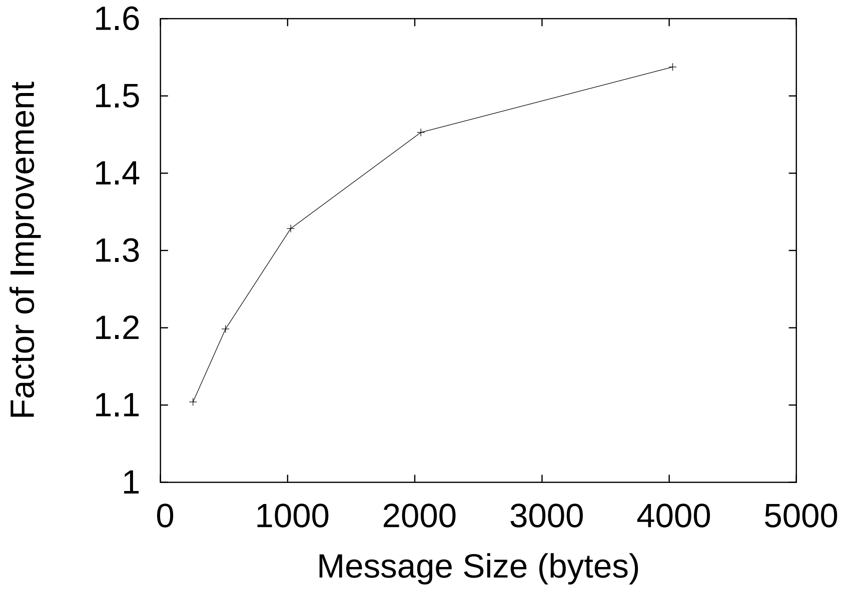


(b) Factor of Improvement

Figure 4.11: Latency of NIC-based echo (nicvm) and host-based echo (baseline) for small message sizes.



(a) Latency



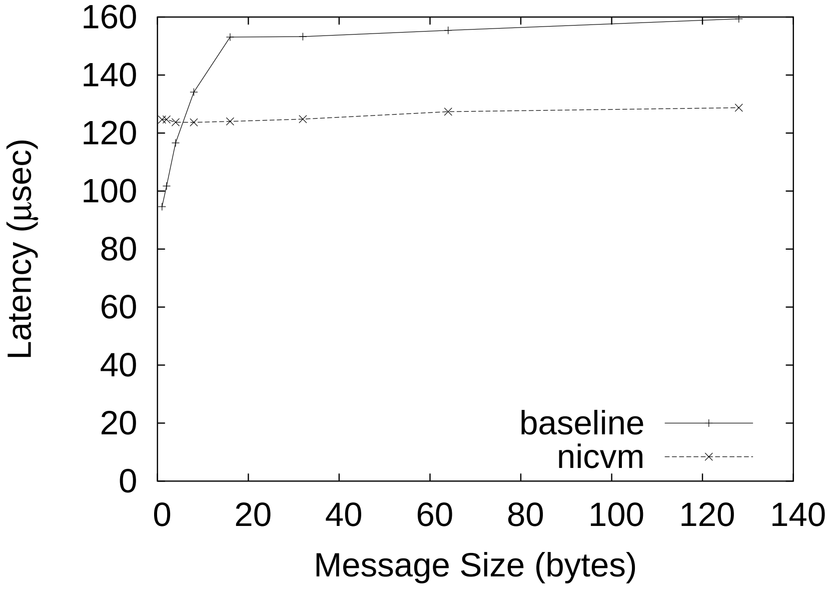
(b) Factor of Improvement

Figure 4.12: Latency of NIC-based echo (nicvm) and host-based echo (baseline) for large message sizes.

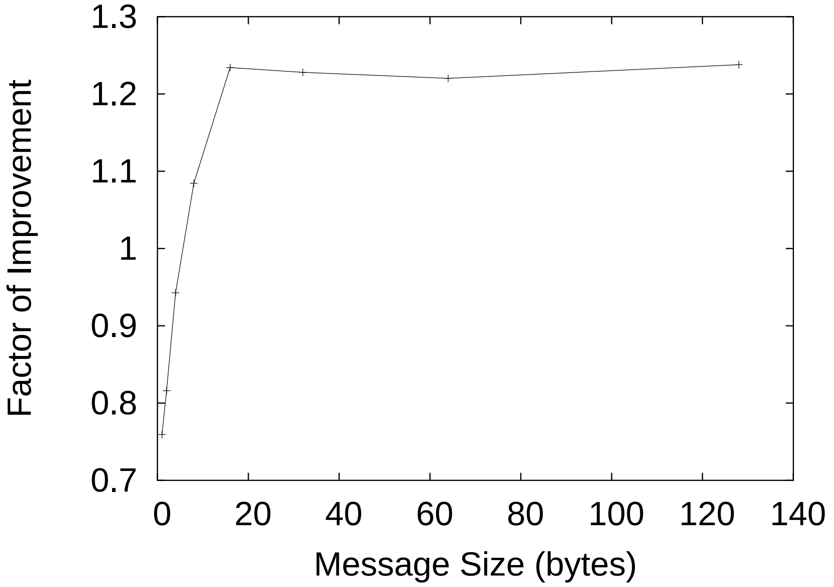
message to determine the total broadcast latency. This process is repeated for varying system and message sizes.

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 4.3 from Section 4.3. 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 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.

Figures 4.13 and 4.14 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 nearly 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 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.

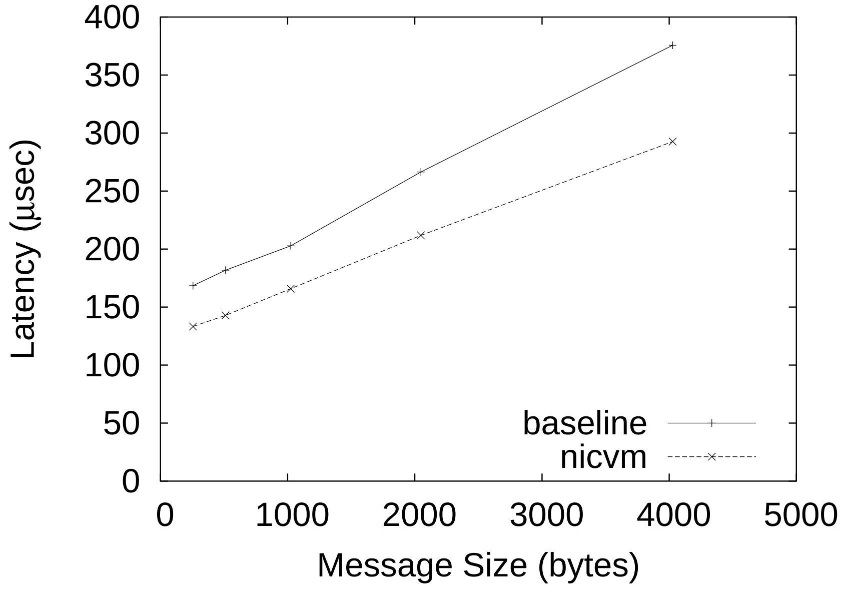


(a) Latency

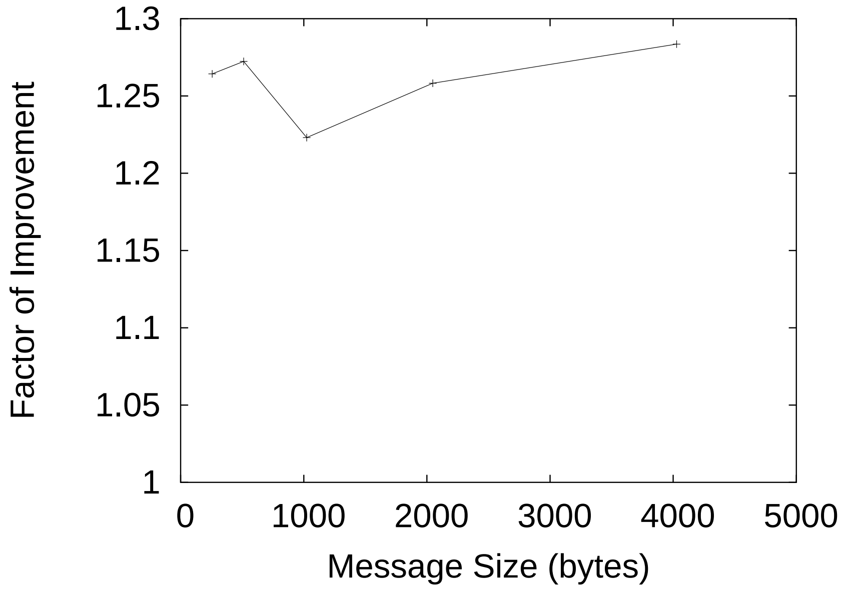


(b) Factor of Improvement

Figure 4.13: Latency of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 16 nodes and small message sizes.

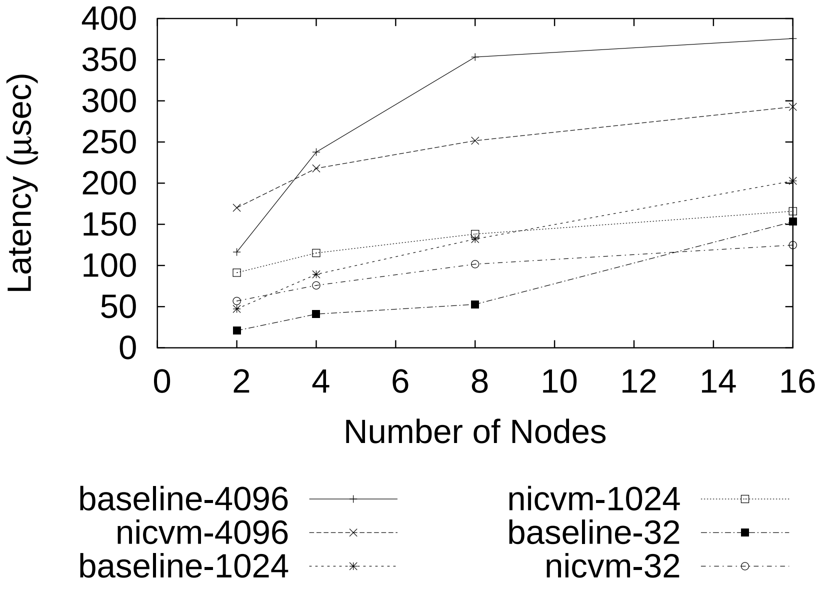


(a) Latency

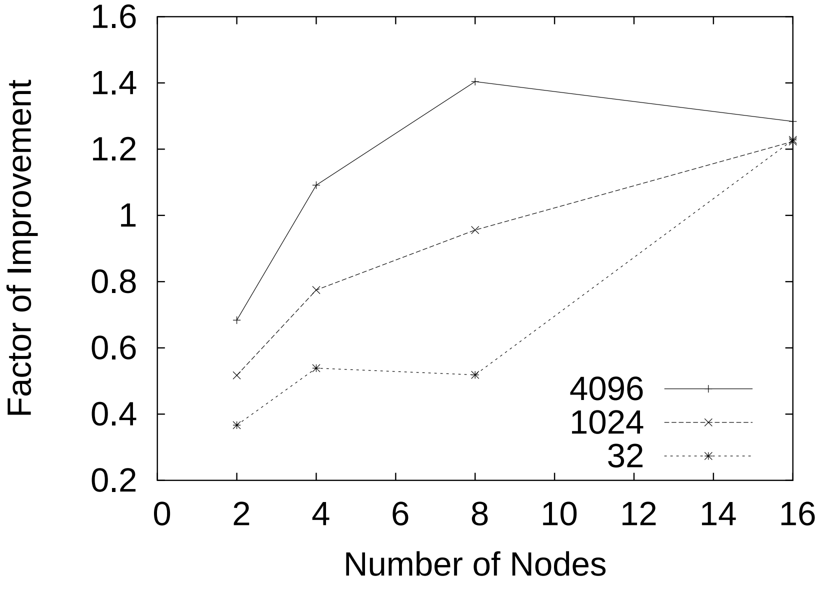


(b) Factor of Improvement

Figure 4.14: 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 4.15: Latency of NIC-based broadcast (nicvm) and host-based broadcast (baseline) for 2, 4, 8 and 16 nodes with 32, 1024 and 4096-byte message sizes.

Figure 4.15 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. In this case, the maximum factor of improvement of 1.4 was actually achieved at 8 nodes instead of 16.

4.4.4 Broadcast CPU-Utilization Results

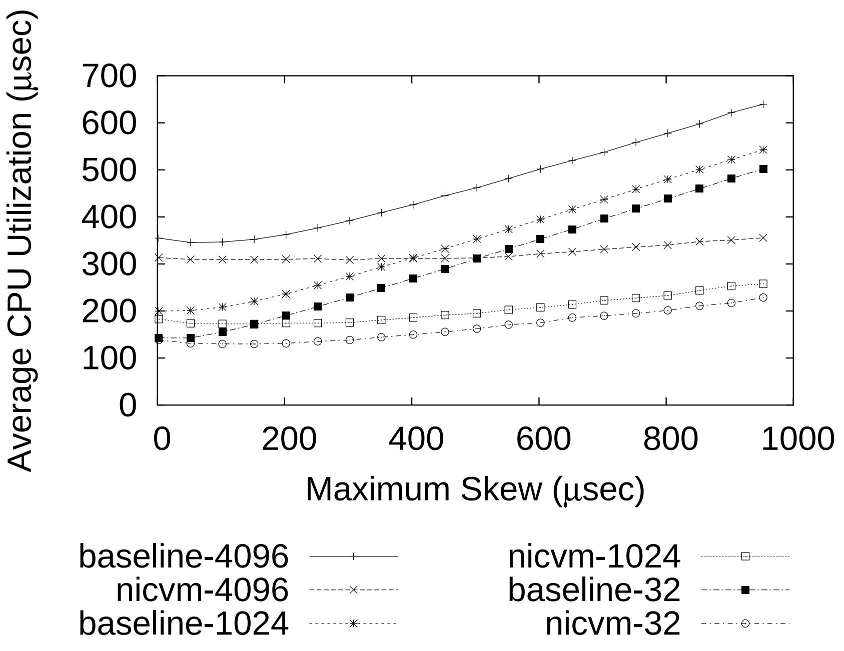
The broadcast CPU-utilization benchmark is implemented slightly differently than the corresponding latency benchmark and is very similar to the method we used to evaluate our implementation of application-bypass reduction. In addition to varying the number of nodes and the message size, we also introduce 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.

Within each loop iteration, the timing measurements are taken as follows. 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.

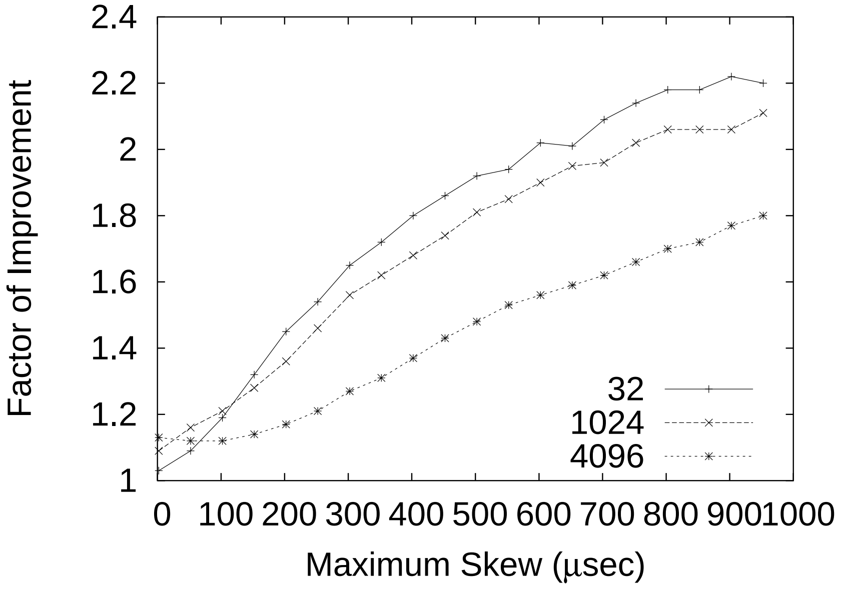
Fig. 4.16 shows the results of the broadcast CPU-utilization benchmark for 16 nodes with increasing amounts of process skew and message sizes of 4096, 1024 and 32 bytes. We can see that the NICVM implementation consistently outperforms the default implementation for all combinations of skew and message size, with Figure 4.16(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 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 less affected as the NIC takes care of forwarding broadcast messages to the children.

Fig. 4.17 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, 1024 and 32 bytes. These results confirm that the results demonstrated in Fig. 4.16 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

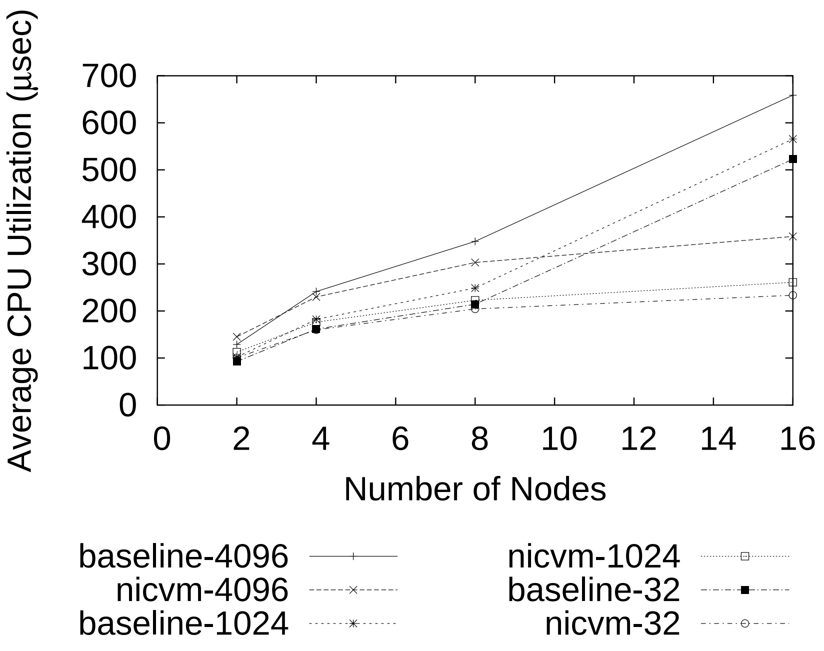


(a) Average CPU Utilization

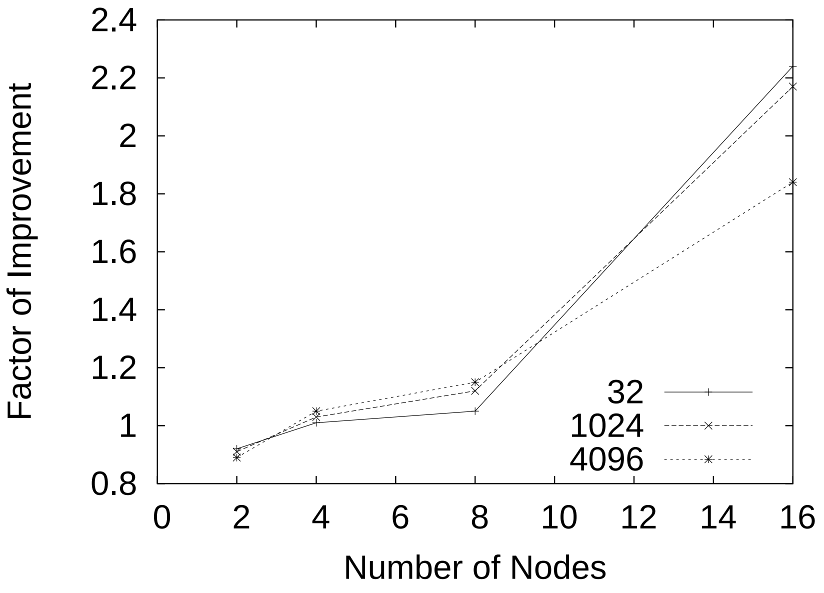


(b) Factor of Improvement

Figure 4.16: Average CPU utilization of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 16 nodes with varying process skew and 4096, 1024 and 32-byte messages.



(a) CPU Utilization



(b) Factor of Improvement

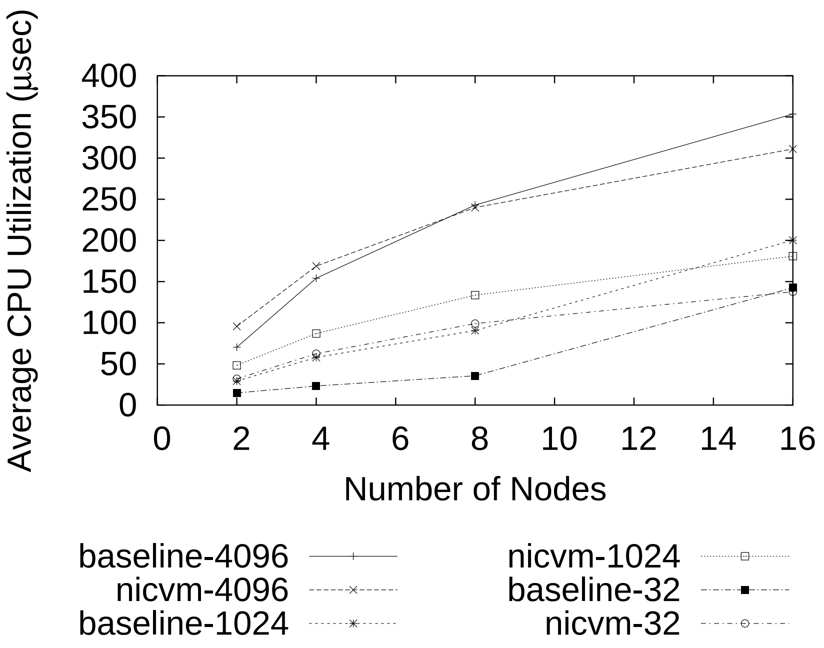
Figure 4.17: Average CPU utilization of NIC-based (nicvm) broadcast and host-based broadcast (baseline) for 2, 4, 8 and 16 nodes with maximal process skew and 4096, 1024 and 32-byte messages.

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.

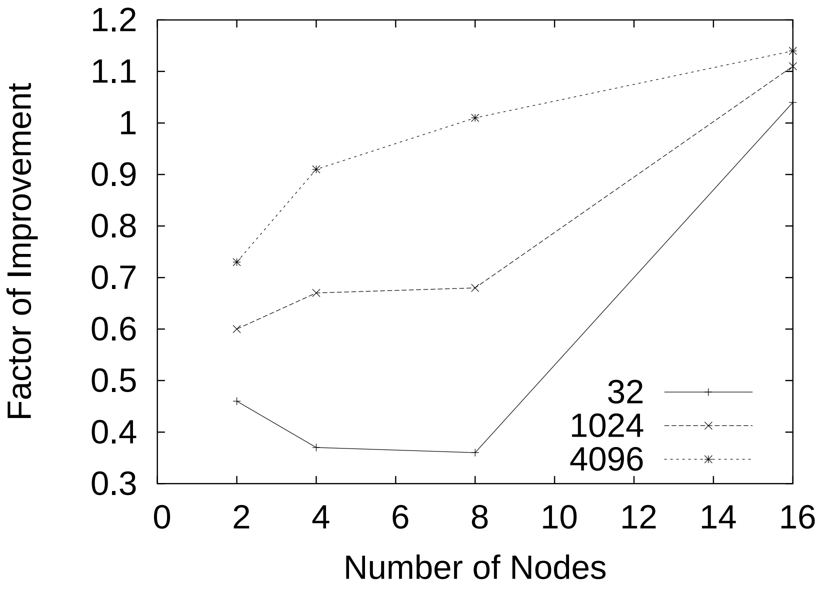
Fig. 4.17 shows the results of the broadcast CPU-utilization benchmark without process skew for for 2, 4, 8 and 16 nodes and message sizes of 4096, 1024 and 32 bytes. Here we can see that even without the introduction of artificial process skew, the NICVM implementation eventually outperforms the default 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.

4.5 Related Work

The U-Net/SLE [18] 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



(a) CPU Utilization



(b) Factor of Improvement

Figure 4.18: 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.

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 [13] 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) [17] 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.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

We have described the design challenges and implementation details of both static NIC-assisted and dynamic NIC-based offload frameworks for Myrinet clusters. Upon evaluation of our NIC-assisted application-bypass implementation of the reduction operation, we found a factor of improvement of up to 5.1 when compared to the default non-application-bypass MPICH implementation under conditions of process skew.

As an example of the capabilities of our framework for dynamic NIC-based offload of user code modules, we implemented a NIC-based version of the broadcast operation. With respect to overall latency, 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 observed a factor of improvement in CPU utilization of up to 2.2 under conditions of process skew. However, note that while performance improvement is desirable, the main focus of this work has been to enable end users to dynamically offload computation to the NIC.

For both frameworks, we note that the factor of improvement increases with system size, indicating that thee benefits of our implementations will lead to improvements in performance and scalability on larger clusters.

5.1 Future Work

In the future, we intend to evaluate the performance of both application-bypass operations and our NICVM framework on large-scale clusters and the latest NIC hardware. We also intend to perform application-based evaluations to better understand how these solutions perform in real usage scenarios.

Another area of investigation which we plan to pursue is the integration of the two frameworks. The NICVM framework could be expanded 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. Depending on the complexity of the reduction operator, an integrated framework could decide whether to perform the reduction on the host in application-bypass mode or to completely offload the reduction to the NIC.

In an effort to further enhance performance and usability of our NICVM framework, 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.

BIBLIOGRAPHY

[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] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[3] 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.

[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] D. Buntinas and 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.

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

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

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

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

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

[18] 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.