A Messaging Layer for Heterogeneous Endpoints in Resource Rich Clusters[*]

Craig Ulmer and Sudhakar Yalamanchili

Critical Systems Laboratory
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, 30332-0250
Email: {ulmer, sudha}@ece.gatech.edu

Abstract

Resource rich clusters are an emerging category of clusters of workstations where cluster nodes comprise of modern CPUs as well as high-performance peripheral devices such as intelligent I/O interfaces, active disks, and capture devices that directly access the network. These clusters target specific applications such as digital libraries, web servers, and multimedia kiosks. We argue that such clusters benefit from a re-examination of the design of the message layer to retain high performance communication while facilitating the interface to endpoints for a variety of devices.

This paper describes a message layer design which includes optimistic flow control, the use of logical channels, a push-style cut-through injection optimization, and an API supporting cluster-wide active message handler management. The goal is to support a number of diverse cluster hardware configurations where communication endpoints exist in a variety of locations within a node. The current implementation has been tested on a Myrinet cluster with communication endpoints located in the host CPUs as well as Intel i960 based I_2O server cards.

1. Introduction and motivation

The current generation of clusters of workstations utilize high speed system area network (SAN) fabrics to interconnect high performance workstations. Modern low latency message layers utilize intelligent network interfaces with communication endpoints located within the host CPU memory and accessed from the user level [1]. Network communication is still largely orchestrated by the CPU with support from intelligent network interfaces. However several trends have created a need for a departure from this "CPU-centric" view of network communication in clusters.

Emerging network-based applications such as digital libraries, web servers, and data warehousing & mining are impacting cluster architectures through the inclusion of powerful peripheral devices. For example, demands for high performance web servers have resulted in programmable I/O cards that directly control arrays of disks and service network requests without host intervention. In other applications, special purpose hardware devices are being used to enhance media processing and search engine capabilities. With the migration of processing power to peripheral devices, the CPU begins to become the bottleneck as all interactions between peripherals and the network are funneled through the CPU and up and down the memory and I/O hierarchies.

Emerging I/O standards such as Infiniband [2] discard bus-based organizations of I/O devices in favor of a point-to-point switched communication fabric to provide concurrent paths from intelligent peripheral devices to (potentially multiple) network interfaces. Communication endpoints can now reside within these devices. Our goal is to provide communication layers that support customization for each peripheral device to maximize the network performance of each device. We propose a flexible API that permits devices (endpoints) to customize their interactions with the network interface without host CPU intervention or need for direct end-to-end interactions with the destination device.

Our approach is to provide a message layer that retains the properties of first generation message layers, primarily low latency, while structuring the design such that device-specific and network-specific features are separated. The

[*] This work is supported in part by a grant from the National Science Foundation under grant ANI-9876573 and by a fellowship from the NASA Jet Propulsion Laboratory's Graduate Student Research Program under grant NGT5-50146.

effect is that of providing an application programming interface to the low-level message layer that is extensible in the sense that new devices can be added by addressing device-specific functionality and a simple protocol for communicating with the local network interface. This paper describes an implementation for a Myrinet based cluster with endpoints within the host CPU and an Intel *i*960-based intelligent I/O (I_2O) card. We conclude with performance measurements that summarize the impact of the proposed message layer design.

2. Background

Communication in clusters is typically facilitated by low latency, high bandwidth system area networks (SANs). Commercial SANs such as Myrinet [3], ServerNet [4], and Scalable Coherent Interface (SCI) [5] have provided major leaps in performance over traditional LAN hardware. A number of custom message layer packages have been written for these SANs to provide low-overhead communication among host CPUs in a cluster [1]. While this “CPU-centric” approach is ideal for clusters that perform all computations at the host level, it can suffer from substantial overheads in providing communication for endpoints located outside of the host CPU.

Thus as clusters evolve we observe that node architectures are becoming increasingly heterogeneous, where powerful peripheral devices may themselves serve as sources and sinks of data. Examples include the following.

- *Multiple Diverse Network Substrates:* Clusters often contain multiple communication interfaces for a number of reasons. Ideally these interfaces collaborate directly and can serve as bridges between network substrates.

- *Intelligent Storage Devices:* Equipment such as the I_2O server adapter card [6] present massive storage options to peripheral devices without host intervention.

- *Hardware Accelerators:* Special-purpose co-processor devices such as FPGA cards are available for graphics acceleration, search engines, and media transformation.

- *I/O Devices:* Additional I/O devices such as cameras, video displays, and video capture devices are all common among clusters with multimedia applications.

The presence of these peripheral devices leads to the notion of *resource rich* clusters (Figure 1) where communication may be initiated not only in host CPUs, but also in peripheral devices. We argue that design trade-offs for message layers executing on a host CPU are not effective when communication endpoints are in peripheral devices.

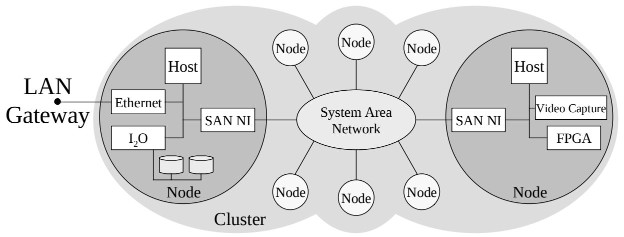

Figure 1: Resource Rich Cluster

3. Middleware for resource rich clusters

For resource rich clusters we find it necessary to re-examine the functionality of message layers to support communication originating and terminating at the peripheral devices with varying capabilities and resources. While it is possible to manage peripheral device endpoints explicitly at the host level, doing so comes at the cost of multiple traversals across the I/O and memory hierarchy. Moving away from this CPU-centric model of managing communication in favor of multiple endpoints per node produces a number of conceptual challenges:

- *End-to-End Flow Control:* Peer devices within a node generally do not have the same magnitude of memory and compute resources as the host CPU. Therefore the overheads of interacting with the NI become quite important. We argue that end-to-end flow control should be moved to the NI, thus reducing the responsibilities of the endpoint and simplifying endpoint operation. The implementation of an optimistic flow control protocol for this purpose is discussed in an earlier paper [7].

- *Multiple, Concurrent Writers:* The presence of multiple communication endpoints in a node creates the problem of multiple writers to the SAN interface. Synchronization among writing endpoints must be lightweight.

- *Receive Processing:* It is important for an endpoint to be able to specify a well-defined set of methods for processing incoming messages since operations performed on reception are specific to the receiving device. While host level endpoints can have a variety of mechanisms to process incoming messages, other devices such as cameras and disks generally have specific operations that are performed on received messages. *Customization* becomes important. This work builds on prior work in active messages [8] to construct an environment for the global registration and addressing of handlers customized for devices.

- *Simple Standardized Endpoint Interface*: In order for diverse peripheral devices to be able to communicate with a host's NI, a standard interface must be developed that can operate on a variety of cards. Since there is a wide range of capabilities for peripheral cards, it is important to design this interface so that it can be implemented on even the simplest of cards.
- *Optimizations*: This paper describes optimizations for NI-level cut-through that are controlled at the producer (rather than on the NI [18]). The result is very low latency for the first byte of the message to get to the wire. High performance demands of some endpoints can make use of such devices.

The majority of message layers are implemented in a CPU-centric fashion and would require substantial modifications to enable the features described above. BIP [9], GM [10], and FM 2.0 [11] provide excellent host-level performance. These layers are optimized for performance and implement a large portion of their functionality on the host CPU, which may not be portable to peripheral device endpoints. Other message layers such as AM II [12] and VIA [13] are attractive since they allow multiple applications to share the NI concurrently. This feature is beneficial because it can potentially be extended to support the sharing of the NI by peripheral device endpoints. The issue here would be to extend or replace the host-based context management schemes to peripheral devices. Finally, we feel that message layers such as LFC [14], PM [15], and FM 1.0 [16] are the best candidates among existing layers since they employ forms of NI based flow control and they implement network management in the NI, reducing the functionality required at endpoints. We believe that pushing as much network functionality as performance goals will permit into the network interfaces will facilitate implementation extensible message layers. Additionally, both LFC and PM provide multiple data queues in the NI that could be adapted for concurrent endpoint use. Doing so would require re-implementation of the communication endpoints for portability.

Modifying existing message layers is a non-trivial exercise when the design goals are different from the design goals governing their original implementation. Further, experience in the community has established that performance is very sensitive to the implementations within the network interfaces given the speed and power of the processors, available memory, bus architectures, and support for data movement. Thus we chose to build on the reported experiences of these message layers and incorporate the relevant concepts in a new implementation designed to meet the goals described earlier in this section. This paper describes the functionality of the design, aspects of our implementation, and some preliminary performance results.

4. GRIM: General-purpose Reliable In-order Messages

GRIM is an extensible framework for user-level messaging that is designed to facilitate the addition of multiple communication endpoints within a cluster node. Extension refers to ability to easily extend the functionality of the message layer to accommodate new endpoint features. Conceptually GRIM is designed with three specific characteristics to meet the needs of heterogeneous clusters: NI managed flow control [7], logical channels, and an active message style of packet reception.

4.1. Optimistic NI-based flow control

GRIM uses an optimistic NI-based flow control scheme to manage the reliable transmission of messages between NIs as illustrated in Figure 2. As a result the communication endpoints are simplified since message injection need only check if there is buffer space on the local NI while handling messages ejected to the endpoint by the NI. End-to-end buffer management is performed in the NI, or rather "in the network". We have observed that the increased functionality of the NI does not substantially reduce the general message layer performance and can in fact improve the gap [17] and peak point-to-point bandwidth. These effects are due to the fact that the optimistic flow control mechanism dynamically allocates buffer space as needed rather than statically pre-allocating buffer space to destination nodes as is commonly found in credit-based flow control schemes. The optimistic flow control method implemented in GRIM is described in [7]. This paper describes the remaining features of GRIM.

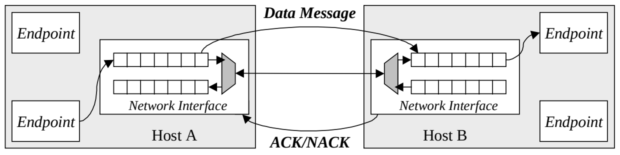

Figure 2: NI-based Flow Control

4.2. Logical channels

The use of logical channels is a simple but effective means for providing injection synchronization among a node's endpoints as shown in Figure 3. Rather than view the entire NI as a shared resource among endpoints, independent logical channels can be allocated at the NI and assigned to specific endpoints. Since a given endpoint has exclusive ownership of a set of NI logical channels, the endpoint can inject messages into these queues without contention hazards with other endpoints. Ownership of logical channels is assigned at start time by the host based on the node's specified configuration.

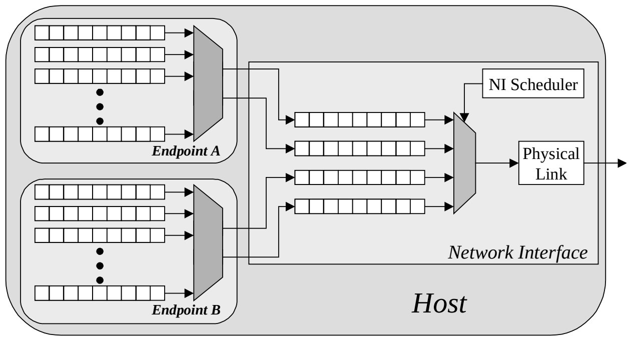

Figure 3: Endpoint Ownership of Outbound Logical Channels

While logical channels are predominantly used for injection synchronization, they can also be used for providing Quality of Service (QoS) via packet scheduling. Endpoints can be configured to control more than a single logical channel so that independent traffic streams are injected into different queues. The number of logical channels that can be implemented practically in the NI depends on the amount of buffer space in the NI and the amount of time the NI can afford to spend searching for new packets. Currently logical channels are multiplexed onto the network through a round-robin scheduler.

The organization of inbound queues (messages from the network) in the NI is affected by the manner in which incoming messages are processed as they are received from the wire. One method of organization is to assign specific inbound queues to each endpoint. Messages therefore are sorted as they arrive off of the wire and isolation is provided between messages destined for different endpoints. Unfortunately this method may add a great deal of complexity to flow control mechanisms since a message can travel from any of the sender's outbound queues to any of the receiver's inbound queues. This adds to the work performed by the NI which includes ensuring that messages are received in-order into the inbound message queue.

An alternative to sorting messages at the point of arrival is to organize messages at the point at which they are ejected from the receiving NI. In this method messages are received into an inbound queue that directly corresponds to the queue from which the message was transmitted. This simplifies flow control overhead since the receiving NI can determine in-order delivery by comparing the message's sequence number to an expected value for the sender's outbound queue. This approach is similar to the concept of virtual networks where each NI queue represents an independent network plane. Given that messages are placed in inbound NI queues only if a cut-through path to the endpoint is not available, head-of-line blocking is likely to take place only at times of high loads.

4.3. Active message style packet handling

The GRIM message layer uses an active message style interface [8] for receiving messages. Active messages was first proposed as a means of reducing latency in parallel systems. In this scheme a message contains both data and information describing how the message should be handled at the destination endpoint. The active message paradigm leads to a flexible and powerful mechanism for handling messages in high performance clusters and is particularly well matched to messaging in clusters with diverse communication endpoints. In particular for I/O card-based endpoints it is often possible to abstract the card's capabilities as a set of functions. For example, an Ethernet card may be capable of transforming Myrinet formatted messages into Ethernet style packets to accomplish network bridging. By identifying this capability as a handler function for the Ethernet card, other endpoints can reference the handler to make use of the SAN/LAN bridge.

Observing that endpoints in a node can be diverse, it is important to construct an active message style interface in a manner that is usable by distinct types of endpoints. In our implementation function handlers are associated with both an integer identifier and a string name. In this interface a number of predefined handlers are available for all endpoints and user programs can dynamically define new handlers as needed. Dynamically installed handlers in GRIM are centrally registered and managed by a single cluster node in order to maintain a single global listing of all cluster handlers.

5. Implementation

In addition to host-level endpoints, we have implemented a GRIM interface for the Cyclone IQ960-RP I_2O server adapter card. This implementation provides a perspective on the communications required, including card-to-host and host-to-host transactions.

5.1. Message management

There are three types of messages used in GRIM: short, bulk, and memory. Short messages are 28 bytes long and include a logical channel id, a function handler id, and four integers that are passed to the receiving function handler. Bulk messages contain the same header information as short messages, but also include up to 48 KBytes of data as a payload. To handle MTU limits of the network hardware, bulk messages are fragmented and reassembled via predefined active message handlers. Memory messages effectively perform a block copy operation between the source and destination node memories and do not invoke message handlers.

Messages in GRIM are managed by queues at both the endpoint and NI levels. Queues consist of three items: a finite ring buffer for the message headers, a region of memory to which bulk payloads are appended, and a set of status registers for maintaining the queue. Breaking the queue into separate header and payload regions permits a large number of short messages to be stored rapidly while allowing bulk message memory allocations to be managed on-demand. The NI allocation of queues is depicted in Figure 4. The NI uses a finite number of outbound message queues to which all logical channels are directly mapped.

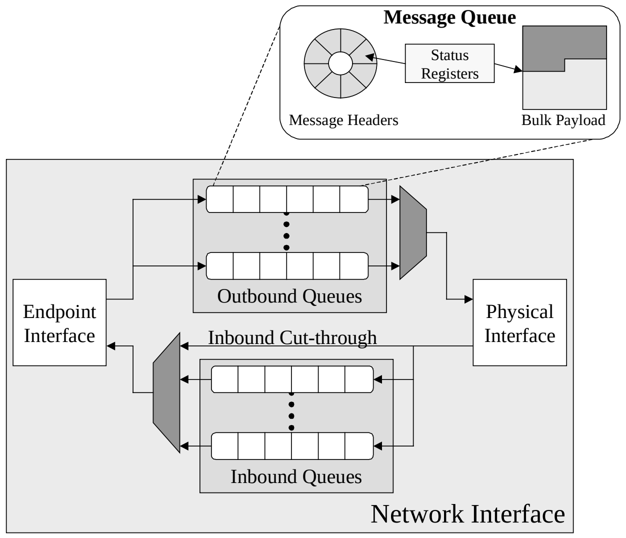

Figure 4: Network Interface Buffer Organization

5.2. Injection issues

In general, message injections in middleware are implemented using either *push* or *pull* style operations:

- *Push*: Endpoints are responsible for moving all data from the endpoint's address space into the NI's. Since the endpoint explicitly moves message data, it is a unidirectional operation that completes when all bytes are injected.
- *Pull*: Endpoints provide pointers to the NI so that the NI can use local DMA engines to pull data from the endpoint's address space to the NI's. Once the pull operation is complete the endpoint is notified so that it can release locks held on the injected message.

Pull style messaging is typical of high-performance middleware since the NI can concurrently pull data from the endpoint and push data to the wire. Because the DMA transactions are managed entirely by the NI, it is possible to implement and precisely control a high-throughput pipeline [18]. In contrast, while simpler to implement, the push style of operation can be limited in terms of performance. CPUs that push messages into the NI must use programmed I/O (PIO) which by itself has limited performance (5-40 MB/s). However, middleware developers [1] reported that the write-combining features of the Pentium Pro architecture could be used to increase PIO performance (up to 125 Mbytes/s). Given the potentially high injection rare and the inherent simplifications in endpoints resulting in the use of a push style protocol, we chose to implement GRIM using a push philosophy. To avoid consistency hazards, write-combining was enabled only for the regions of the NI that hold the bulk data queues.

The main performance challenge in using a push-based scheme is minimizing the amount of time between when an endpoint starts injecting a message and the time when the message reaches the wire. As shown in Figure 5(a), a simple store-and-forward operation can be used where the endpoint injects the entire message before the NI begins transmission.

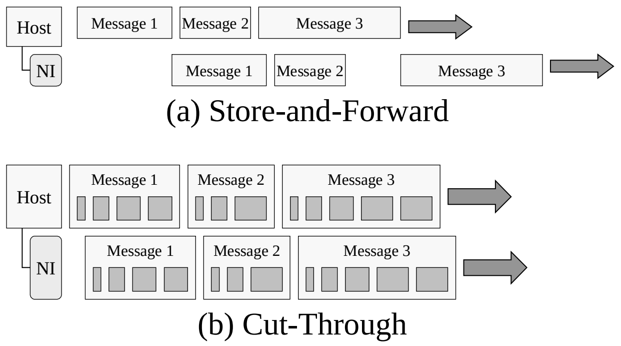

Figure 5: Timing for Message Injections

However, a better approach is to allow injected messages to “cut-through” the NI to the wire as shown in Figure 5(b). In this case a small (32-byte) segment of the message is first pushed to the NI. While the NI transmits the initial segment to the link a larger data set is pushed into the card. The transfer of successively larger data segments to the NI is overlapped with transfer from the NI to the link. At the end of each segment injection, the endpoint updates status registers on the NI to specify how much of the entire message has been pushed into the NI. The NI can therefore begin transmission of the message as soon as the first segment is available. The time to get the first byte of a message to the wire is substantially reduced.

This approach is advantageous because it maintains the simplicity of push-based mechanisms while obtaining the performance of pull-based operations. By making use of host-level architectural support such as write combining, it is possible to obtain transmission bandwidths in excess of 106 Mbytes/s for bulk message.

5.3. Message ejection

The GRIM middleware was implemented to perform message ejection from the NI in a variety of styles to meet the needs of endpoints for multiple devices. Wherever possible cut-through memory transfers are used to reduce receiving latencies. For the case where an endpoint is unable to accept a new message, messages are stored in the NI’s inbound queues as depicted in Figure 4. Both short and bulk messages are pushed into queues that are in the endpoint’s address space. For host-level programs this is the pinned memory provided by the MyriAPI driver. The GRIM middleware implements a zero-copy transfer to user space for memory transfer style messages. This operation required the modification of the MyriAPI driver and provides a significant improvement in message reception performance.

5.4. GRIM API and operation

The GRIM API is a small set of commands necessary for implementing a message-passing library. The API has three categories of functions: library initialization and configuration querying, active message handler maintenance, and general-purpose message passing. The initialization function *grim_enable*() enables the GRIM library for a host and initializes all of the node’s endpoints as specified by the application’s configuration. Once enabled, endpoints can use the general-purpose message functions *grim_send_*{*short, bulk, memory*}*()* to transmit messages to cluster endpoints. The polling function *grim_poll()* is used to extract and execute active messages in an endpoint’s inbound queue.

The active message handler maintenance functions of the API allow programs to dynamically register handler functions as required. Endpoints first register handlers locally with the *grim_registerHandler()* function. This list of functions is later published to the global context when a *grim_synchHandlers()* call is made. This function transmits the local handler list to a central node in the cluster that is responsible for merging handler lists and relaying changes to all nodes. Finally, endpoints query a local copy of the global handlers list with the *grim_getHandlerByName()* function to translate a string identifier to the integer associated with the handler.

As a specific example we implemented a small set of functions for the I_2O card endpoint, including a ping/pong handler for timing and a simple bridge function to use the card’s Ethernet capabilities. By abstracting the card’s functionality into a set of handler functions, we could easily extend the capabilities of the cluster to make use of the card’s features. The extension was facilitated by the fact that end-to-end flow control is handled within the Myrinet NI and the I_2O handlers need only locally synchronize with the Myrinet card. Currently we are also studying an extension to a companion FPGA card’s resources that is suitable for GRIM. In this sense we address a problem that is similar to that being addressed at Virginia Tech in the use of FPGA cards as a collective resource via Myrinet [19].

5.5. Adding peripheral device endpoints

An important feature of GRIM is the ability to extend the functionality of the message layer to include support for new peripheral device endpoints. This was accomplished by defining both standard data structures for endpoint queue maintenance as well as specific data transactions that signify a message layer event. While adding support for a new device is non-trivial, the general process is outlined as follows.

First, a set of specific handler functions must be defined for the endpoint device that sufficiently encapsulates its capabilities. Next, the device’s physical interface with the host system must be considered. Properties such as the card’s memory, DMA engines, address translation requirements, and processing capabilities must be translated to a form in which endpoint software can be written. Equipped with this information, a designer adds card specific initialization calls to the GRIM library that instruct the node’s other endpoints how to interact with the card. Finally endpoint software is written for the card that monitors its data queues and reacts to incoming messages.

6. Performance and evaluation

GRIM was developed and tested for x86 machines running Red Hat Linux 6.1. Performance numbers reported in this paper come from a pair of Pentium III based machines with directly connected M2M-PCI32 Myrinet SAN cards. Our experiences with a larger 16-node Myrinet cluster suggest additional Myrinet switch hardware does not noticeably degrade (and can in fact improve) performance. These Myrinet cards use the older 33-MHz LANai 4 chipset and are equipped with one megabyte of memory.

To test the interface to multiple endpoints we included the Cyclone IQ960-RP I_2O development card as an example of a card-based endpoint that occupies another slot on the PCI bus. The Cyclone card features an *i*960 processor, 4 megabytes of RAM, dual Ultra-SCSI ports, and dual 100Mbps Ethernet. Firmware for the card was written using VxWorks and an in-house Linux driver.

6.1. Short message performance

The performance of GRIM in the context of short messages was reported in [7] but is reproduced here for completeness. Using round-trip timing measurements we observed host-to-host latencies for short messages to be 13 µs. The overhead for a host to inject a message was very low, approximately 1 µs. Due to the dynamic usage of NI buffer space through NI-based flow control, the minimum gap between successive short messages is negligible for bursts less than 200 messages in number. For message bursts larger than 200, GRIM's minimum gap slowly grows to a steady state value of 20 µs for bursts larger than 1,000 messages.

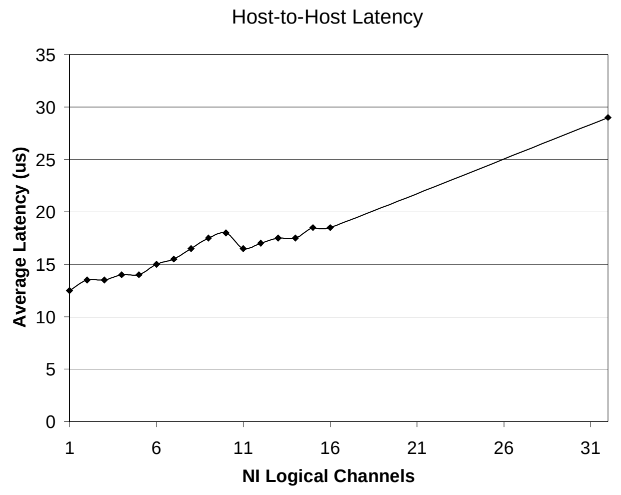

Figure 6: Latency for NI Logical Channels

As a means for evaluating the effects of logical channels, we examined the host-to-host latency performance of short messages while varying the number of NI logical channels. As shown in Figure 6, small numbers of logical channels implemented in the NI do not significantly impede performance. As the number of logical channels increases, the NI must spend more time searching for queues with pending jobs. This meets our expectation that large numbers of NI logical channels are impractical because they come at the cost of general performance. However, a small number of logical channels proves to be both useful and at low cost to performance.

6.2. Large message performance

Host–to-host round-trip timing measurements were also used to analyze the large message performance of GRIM. The tests were run on both store-and-forward and cut-through injection versions of GRIM to observe the effects of injection mechanisms and are presented in Figure 7. Cut-through does in fact obtain a much higher peak bandwidth performance (106 MBytes/s) over the store-and-forward method (74 MBytes/s).

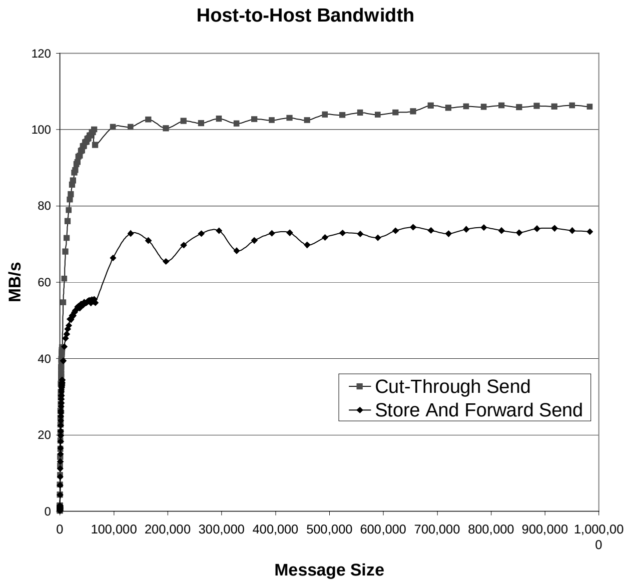

Figure 7: Bandwidth Performance

6.3. Myrinet to I_2O performance

Performance for communication with the I_2O development board was comparable to host-level performance. In our tests we transmitted messages from a host to reach a remote host's I_2O board using the Myrinet

NI. The one way latency for Host-Myrinet-Myrinet-I_2O was observed at 21 μs. Specific characteristics of the I_2O board affect performance. The DMA engines on the I_2O board are optimized for large multi-stage transactions and as such are not optimal for short message bursts. Additional card-specific hardware (such as chained DMAs or doorbells) was not utilized in these tests to preserve generality.

7. Conclusions

Resource rich clusters are an emerging category of computational platforms where cluster nodes have CPUs as well as high-performance peripheral devices that directly access the network. This paper proposed an implementation for message layers that facilitated the interface to a variety of such endpoint devices. The approach is based on the use of active message style of communication, coupled with NI-based flow control and NI cut-through for low latency to the wire. We have verified that this approach can be realized for an I_2O based card without significant degradation in general performance.

Acknowledgements

The authors gratefully acknowledge the assistance of Ivan Ganev, Robert Goldman, and Emily Crawford for their help with the linux drivers, the I_2O card, and the Myrinet card. We would also like to thank Bob Felderman of Myricom for his invaluable assistance with Myrinet. The authors would like to acknowledge the generous support of NASA JPL under the NASA Graduate Student Research Program and the Advanced Networking Infrastructure program of National Science Foundation.

References

[1] R. Bhoedjang, T. Ruhl, and H. Bal. *User-Level Network Interface Protocols*. IEEE Computer, Vol.31, No.11, P53-60, 1998.

[2] Infiniband Trade Association Website http://www.sysio.org/home.html

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. *Myrinet: A Gigabit-per-second Local Area Network*. IEEE Micro, Vol.15, No.1 1995.

[4] R. Horst and D. Garcia. *Servernet SAN I/O Architecture*. In Hot Interconnects Symposium V, August 21-23 1997

[5] *Scalable Coherent Interconnect,* IEEE Standard 15961992, 1992

[6] Cyclone Microsystems Website: http://www.cyclone.com

[7] C. Ulmer and S. Yalamanchili, *An Extensible Message Layer for High-Performance Clusters.* In Proceedings of PDPTA 2000, June 2000.

[8] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation*. Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA). May 1992.

[9] L. Prylli and B. Tourancheau, *BIP: A New Protocol designed for High-Performance Networking on Myrinet*. In Proceedings of PC-NOW IPPS/SPDP98, 1998.

[10] Myricom, Inc. The GM message layer, http://www.myri.com

[11] S. Pakin, V. Karamcheti, and A. Chien. *Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors*. IEEE Parallel and Distributed Technology, 1997.

[12] B. Chun, A. Mainwaring, and D. Culler. *Virtual Network Transport Protocols for Myrinet*, In Hot Interconnects'97, Stanford, CA, April 1997.

[13] P. Buonadonna, A. Geweke, and D. Culler. *An Implementation and Analysis of the Virtual Interface Architecture*, in Proceedings of Supercomputing '98. Orlando, FL, November 1998.

[14] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. Journal of Parallel and Distributed Computing, 40(1):49-64, February 1997.

[15] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. *Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication*. In 12th Int. Parallel Processing Symposium, pages 308-314, Orlando, FL, March 1998.

[16] S. Pakin, M. Lauria, and A. Chien. *High Performanc Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet* in Supercomputing '95, 1995

[17] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a Realistic Model of Parallel Computation*, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993.

[18] K. Yocum, D. Anderson, J. Chase, A. Gallatin, and A. Lebeck. *Cut-Through Delivery in Trapeze: An Exercise in Low Latency Messaging.* IEEE Symposium on High-Performance Distributed Computing (HPDC), Portland OR, August 1997.

[19] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott. *Implementing an API for Distributed Adaptive Computing Systems.* FCCM 99