Using Embedded Network Processors to Implement Global Memory Management in a Workstation Cluster

Yvonne Coady, Joon Suan Ong and Michael J. Feeley
Department of Computer Science
University of British Columbia
{ycoady, jsong, feeley}@cs.ubc.ca

Abstract

Advances in network technology continue to improve the communication performance of workstation and PC clusters, making high-performance workstation-cluster computing increasingly viable. These hardware advances, however, are taxing traditional host-software network protocols to the breaking point. A modern gigabit network can swamp a host's IO bus and processor, limiting communication performance and slowing computation unacceptably. Fortunately, host-programmable network processors used by these networks present a potential solution. Offloading selected host processing to these embedded network processors lowers host overhead and improves latency. This paper examines the use of embedded network processors to improve the performance of workstation-cluster global memory management. We have implemented a revised version of the GMS global memory system that eliminates host overhead by as much as 29% on active nodes and improves page fault latency by as much as 39%.

1. Introduction

Advances in network technology are revolutionizing high-performance computing on workstation and PC networks. Modern networks have improved latency and bandwidth by two orders of magnitude compared to traditional 10-Mb/s Ethernet [3, 5]. This improvement has substantially bridged the performance gap between commodity workstation clusters and specialized multiprocessor systems.

A key difference remains, however, between the commodity and specialized approaches. Commodity workstation networks must be suitable to the broadest possible range of client applications. As a result, their network interfaces are simple and general purpose. Higher-level protocols such as reliable delivery, flow control, buffer management, group communication, and memory management are thus left to software running on host processors. In contrast, specialized systems can provide these features in hardware, at higher cost but with substantially improved performance.

This system software is now the key obstacle to the goal of high-performance cluster computing. There are two main problems. First, a gigabit network can transfer data fast enough to swamp the host IO bus with network data and the host processor with protocol overhead. Second, with hardware latencies dramatically reduced, software overhead now accounts for a significant portion of total communication latency, particularly for small messages.

Current gigabit networks present a potential solution to this important problem. While they are poised to become commodity networks of the near future, their network interfaces are implemented using a host-programmable network processor [3, 5]. It is thus possible to implement higher-level protocols partly or entirely on the network processor, thus avoiding host overhead and unnecessary data transfer over the host's IO bus.

There are two major factors to consider when dividing functionality between network and host processors. First, network processors are slower than host processors and so computationally-intensive operations are best performed on the host, trading off increased host overhead for reduced latency. Second, the processing load of both network and host processors should be balanced; overloading one makes it a bottleneck that reduces overall throughput.

This paper examines this tradeoff in the context of a global memory system, called GMS, that manages workstation RAM as a global resource. Applications that use a GMS-enabled operating system have automatic access to all memory in the network to store their data. GMS pages data to and from idle remote memory instead of disk, thus lowering page fault latency by two orders of magnitude compared to disk.

We have designed and prototyped a new version of GMS, called GMS-NP, that moves key protocol operations to the embedded network processors. We focus on the global page directory that GMS uses to locate pages in global memory and we report on two alternative designs that use the network processor to different degrees. We compare both GMS-NP prototypes to the original host-based version of GMS.

Our results show that using the network processor reduces the latency of a page-directory lookup operation by at least 50%. When the instruction and data caches on the host processor are cold, this improvement is as much as 90%. Of course, the latency of the lookup operation is only part of the total latency of a remote-memory page-fault. Its contribution, however, is significant as we were able to improve performance of a remote page-fault by as much as 39% in the cold cache state. GMS-NP also eliminates all directory overhead on the host, which can be as high as 29% under heavily loaded conditions.

1.1. Future network processors

Our work demonstrates that programmable network processors found in current gigabit networks can be used to overcome the system-software bottleneck. We thus argue that as these networks become more of a commodity, they should continue to have programmable network processors. We believe this will be the case because the economics of processor design strongly favor this outcome. Trends over the past decade show processor performance doubling every 18 months, while the price is halved each year after a processor is released. In steady state, these trends indicate that a four year old processor will perform at roughly 15% of the cutting-edge model and be priced at 6% of its original cost. It thus seems reasonable to assume that embedded network processors that lag their host counter-parts by about four years are cost effective.

1.2. Related work

The idea of performing high-level work on peripheral processors is not new. In the early 1960s, significant portions of some operating systems were coded on peripheral processors [1]. Early models of the CDC 6600 [2] were built with one CPU and 10 to 20 peripheral processing units. Many minicomputer operating systems in the 1980s used network interface processors to provide support for services (telnet and TCP/IP[17]) not included in the operating system. In contrast to today's architecture, these interfaces used costly processors which lagged a mere 1-2 years behind their host counterparts, and subsequently became extinct when operating systems began to include such support.

Our work is somewhat reminiscent of hardware distributed shared memory systems such as Dash and Flash [15, 14]. These systems implement directory-based cache coherency protocols for specialized multiprocessor systems. Our work is differentiated by the fact that we target commodity workstation clusters.

Several recent research projects have explored the benefits of using programmable network interfaces provided by current gigabit networks [6, 3]. These benefits include the lower message overhead possible when interfaces are directly accessible at user level [7, 10, 21], the lower large-message latency possible when interfaces fragment and pipeline data transfers between host memory and the network [4, 8, 22], the higher throughput possible when fragmentation and pipeline are adaptive to message size [18, 20], and the lower overheads possible when interfaces implement sender-based flow control [7]. Our work differs from each of these projects in that we focus on functionality that is higher-level than the network protocol layer.

The SPINE system from the University of Washington provides a safe and extensible environment for programming network processors [11, 12]. In contrast, our work demonstrates the benefits and tradeoffs of moving memory management functionality to a network processor. We believe that the implementation and debugging of our prototype would have benefited from the SPINE framework.

2. Background and design of GMS-NP

GMS-NP re-implements the GMS global memory system to divide functionality between the host and network processors. In GMS-NP all global directory services are executed by the network processor, while the remainder of GMS remains unchanged and on the host. This section introduces the essential functionality GMS provides, establishes the motivation for GMS-NP, and describes the key design issues.

2.1. GMS

GMS is a global memory service integrated with operating system virtual-memory and file-cache management [4, 9, 13, 19, 22]. GMS manages workstation memory to store both local and remote data. Page faults and replacements are handled globally using a distributed algorithm based on LRU. Using GMS, an active node thus has automatic access to the idle memory of other nodes in the network. When a page is replaced from a node's local memory, the algorithm moves the page to the node with the globally least-valuable page, where it replaces that page. For each page fault or file-cache miss, GMS checks its global page directory for the desired page. If the page is stored on a remote node, either due to a previous pageout or because the page is shared with a remote process, GMS transfers the page to the faulting node, thus avoiding a disk read.

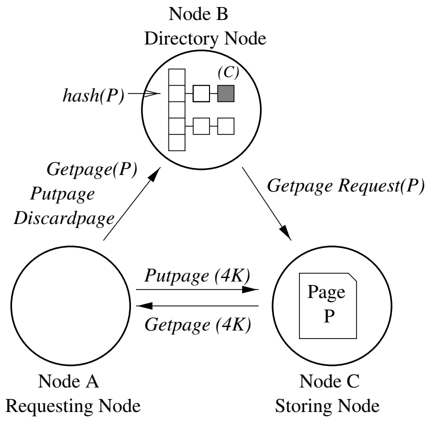

Figure 1. GMS operations.

2.2. The global page directory

The global page directory is used to locate network-memory-resident pages. The directory stores an entry for every resident page, listing the nodes that store copies of the page. Pages are assigned a globally unique name based on their permanent location on disk and this name is used to index the directory.

Although the directory is logically centralized, it is physically distributed, with each node storing an equal a portion of the total directory. This design aids scalability by sharing directory overhead evenly among all nodes in the system. It also facilitates the efficient transition of a node between active and idle states, because this transition does not change how the directory is distributed. In contrast, a design that used only idle nodes to store the directory would require a costly reconfiguration each time an idle node became active or an active node became idle. Nodes locate a page's unique directory node using a small hash table that is replicated on every node.

2.3. GMS operations

Figure 1 shows three key GMS operations: **putpage**, **discardpage**, and **getpage**. For the sake of clarity, each node in this figure has been assigned exactly one role within the system: Node A is the *requesting node*; Node B is the *directory node*; and Node C is the *storing node*. It is important to remember, however, that any node could be a requesting node, every node contains part of the distributed global directory, and any node could be a storing node.

In GMS, nodes manage their local memory in the standard way, using LRU to replace old pages to make room for new ones. When a node replaces a page, however, the GMS running on that node determines whether the page should be forwarded to remote memory and then invokes either **putpage** to forward the page, or **discardpage** to drop the page. Two messages are required for **putpage**, the first to forward the page and the second to update the directory. **Discardpage** only requires a directory update message.

When a node incurs a page fault or a file-buffer cache miss for a page stored in network memory, the local GMS uses **getpage** to locate and retrieve a remote copy of the page using three messages. First, the requesting node sends a lookup request to the page's directory node. Then, if the page is stored in remote memory, the directory node forwards the request to a storing node. Finally, the storing node sends a copy of the page back to the requesting node.

2.4. Motivation for GMS-NP

The primary goal of GMS is to use idle or under-utilized network resources to improve performance without harming local performance on any workstation. To achieve this goal, GMS was designed to minimize the overhead imposed on active nodes. Most GMS overheads are borne by the idle nodes that store remote pages.

The GMS global page directory, however, can introduce overhead to all nodes in the network, even *active* nodes. In the original version of GMS, which ran on a 100-Mb/s network, the peak overhead the directory imposed on any active node was less than 3% and was thus of little concern. Moving GMS to a gigabit/s network however, increases the peak remote-memory page fault rate by an order of magnitude which in turn can increase active-node overhead an order of magnitude, to 29% in our tests.

This significant increase in overhead is a direct result of the improvements in latency and bandwidth delivered by a gigabit network. Hence, the functionality associated with the source of this overhead, the global directory, is a good target for transfer to the network processor. Doing so not only frees active hosts from this overhead, but also reduces the latency of GMS operations such as remote page fault.

2.5. GMS-NP design overview and tradeoffs

GMS-NP implements the global page directory on network processors instead of host CPUs. The key benefit of this design is that directory operations can be completed without interrupting the host processor. Host processor overhead is eliminated at the cost of increasing network processor overhead.

To achieve an overall improvement in both latency and throughput, which is our goal, we must thus ensure that the amount of overhead added to the network processor is less than the savings achieved on the host. Our task is complicated by the fact network processors are roughly eight-times slower than host processors and have a modest amount of on-board memory (e.g., our prototype has 1-MB). We must thus carefully consider what operations to perform on the network processor and how to store the data they access.

The global page directory is a hash table. Directory operations lookup, update, insert, and delete entries from this table. Specifically, **putpage**, **discardpage**, and **getpage** perform a lookup and update an entry. In addition, **putpage** inserts an entry if one does not already exist and **discardpage** deletes an entry if the last copy of a page is being discarded. Finally, **getpage** also sends a request message to the storing node or a response message to the requesting node, if the page is not found.

This data structure presents two main challenges for moving to the network processor. First, hash-table lookup requires computation of a hash index and this computation is performed much more efficiently on the host CPU. Our design addresses this issue by moving index computation to the requesting host processor, instead of the network processor. The requesting node computes the hash index and includes it with all request messages it sends to the directory node. Directory operations then use this pre-computed index to access their hash table.

The second challenge is that the hash table may be too large to store in memory on-board the network processor. This is particularly true for multi-programmed environments in which the network processor may implement many different subsystems and applications. In this case, it will be necessary to store the table in main memory, which complicates access for operations running on the network processor and may degrade performance. To investigate this issue, we prototyped two versions of GMS-NP, one that stores the table in host memory and the other that stores it in network-processor memory.

3. Implementation of GMS-NP

In this section we describe the implementation of two GMS-NP prototypes: GMS-NP(HM), which stores the directory in host memory, and GMS-NP(NM), which stores the directory in network memory. These prototypes are implemented on a cluster of Pentium II PCs connected by the Myrinet [5] gigabit network. Each is integrated with the FreeBSD operating system running on the PCs and with the Trapeze [4, 22] network control program running on the network processors. We describe our implementation in three stages from the bottom up, beginning with GMS-NP's integration with the network layer and control program running on the network processor. The second stage focuses on the processing of GMS-NP directory operation messages common to both GMS-NP prototypes. Finally, we feature the implementation details and optimizations of GMS-NP(HM).

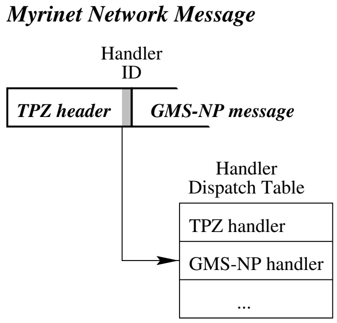

Figure 2. The Handler-ID and Handler-Dispatch Table

3.1. Integration with Trapeze network layer

Trapeze is a messaging system that includes a control program for Myrinet's network processor. In part, Trapeze was designed to minimize the latency of remote-memory page faults in GMS. Trapeze optimizes for low latency for large messages by fragmenting the messages into smaller packets and pipelining those packets through the data transfer between host memory, network-interface memory, and the network.

We modified the Trapeze Myrinet control program to add a general facility for network-processor based message handling, in the style of active messages [16]. To do this, we added a *handler-ID* field to the Trapeze message header. As depicted in Figure 2, the handler-ID indexes a *handler-dispatch table* stored in network-interface memory. Each entry of this table records the address of a handler procedure that receives control for incoming messages, based on the message's handler-ID field.

Our prototype uses a two-entry handler table, one entry for standard Trapeze message handling and the other for GMS-NP. Messages with the Trapeze handler index are processed in the standard way, interrupting the host and transferring message data into host memory. Messages with the GMS-NP handler index are handled locally on the network processor, as described next.

3.2. Handling directory operations

As shown in Figure 1, all three GMS operations, **putpage**, **discardpage**, and **getpage**, send a message from the requesting node to the directory node. Figure 3 shows the subsequent handling of these incoming GMS-NP directory request messages by the network processor. Executing a directory operation requires the key fields highlighted in Figure 3: *operation-ID*, *hash value* and *page name*. First, the GMS-NP handler extracts the *operation-ID* from the message and indexes the GMS-NP *operation dispatch table* to

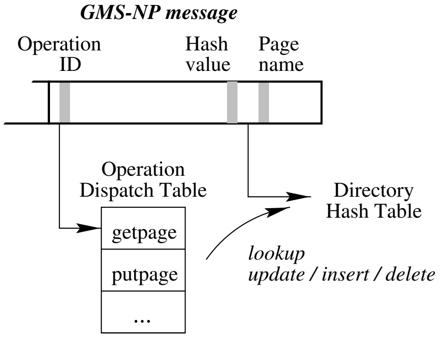

Figure 3. GMS-NP message handling.

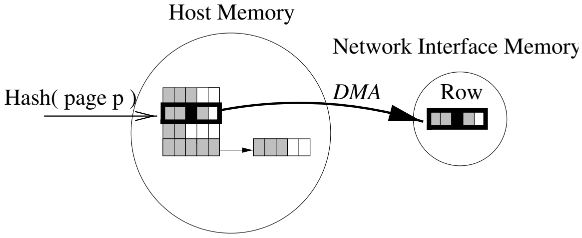

Figure 4. The GMS-NP(HM) directory hash table is stored in host memory.

select the requested operation. Control is then transferred to the appropriate GMS-NP operation. The GMS-NP operation then extracts the *hash value* and *page name* from the message in order to perform a lookup in the directory. Finally, the directory entry is either updated, inserted or deleted, depending on the operation.

Getpage also requires that an outgoing message be assembled by the network processor and forwarded to the storing node if the page is resident, or the requesting node if it is not in network-memory. Constructing this message on the network interface is accomplished by modifying the handler ID, operation ID, and destination routing information of the original request message. In effect, the original request is transformed into a new message for the Trapeze handler of the destination node, which is then sent back out to the network.

3.3. Accessing the host-resident table in GMS-NP(HM)

In GMS-NP(HM), the hash table is stored in host memory and relevant portions must be efficiently transfered to and from network interface memory. Figure 4 diagrams the basic layout of the GMS-NP(HM) hash table. Operations running on the network processor manipulate the table locally through a small buffer stored in network interface memory. The network processor uses DMA to transfer hash-table rows into this buffer from host memory as needed to satisfy table read accesses. Write accesses are handled by a DMAing a single entry from network interface memory to host memory.

Hash table positions are organized into rows so that related entries can be stored contiguously and transferred in a single DMA. Row size is an important parameter for overall performance, as discussed in Section 4.5.

Performing the lookup operation when accessing the host-resident table requires first, DMAing a row from host memory and second, scanning the row for the desired page. Our implementation optimizes this operation by pipelining the DMA and scanning components of this operation. The GMS-NP(HM) lookup operation starts scanning a row as soon as the first entry of the row arrives in interface memory. To do this, it makes use of a LANai hardware register that maintains a running counter of the number of bytes transferred by the host-DMA engine.

The following algorithm is used to scan a row, starting with the first entry (i.e., $i = 0$).

1. Spin on the LANai DMA register until entry i arrives.
2. Scan entries up to and including i.
3. Estimate a new i based on DMA throughput and the elapsed time since the DMA was initiated.
4. Repeat, starting at step 1, until the desired entry is found or the entire row is scanned.

The computation of the new i in step 4 is determined statically based on performance measurements of DMA and LANai performance presented in Section 4.4. Section 4.2 shows that this procedure achieves almost complete overlap in practice.

Finally, synchronization between network and host processors in GMS-NP(HM) was handled by requiring that all updates to the table and its meta-data be performed by the network processor. As a result, some directory lookup operations that could be completed locally in GMS, must be handled by the network processor in GMS-NP(HM). Recall that the global directory is distributed among every node in the network. This means that $1/n$ requests are for pages whose directory entry is stored on the requesting node, in a network of n nodes. In GMS, these directory lookups could be handled by a local procedure call. In GMS-NP, however, the requesting node must ask its network processor to perform the lookup, even though the desired entry stored in the requesting host's memory.

3.4. Summary

In the original GMS, all directory request messages involve interrupting the host for processing. In GMS-NP, these same directory requests are executed by the network processor. The key issue addressed by our implementation is the balance of resource utilization and performance. We address this issue in three ways. First, we introduce the handler-dispatch table to multiplex messages between the network and host processors. Second, we implement two prototypes to compare the effects of storing the hash table in host memory versus network interface memory. Finally, we reduce the impact of the DMA latency required to access the host resident table by pipelining the the lookup operation.

System	Latency (μs)		
	NP	HP	Total
GMS (I)	6.5	20.5	27.0
GMS (C)	6.5	118.5	125.0
GMS-NP (HM)	13.5	0	13.5
GMS-NP (NM)	10.0	0	10.0

Table 1. Latency of a page lookup operation on a directory node.

4. Performance measurements

This section compares the performance of our two GMS-NP prototype implementations and standard GMS. Following a description of our experimental setup, our results are organized as follows. First we present a timeline for the **getpage** lookup latency on the directory node. Then we present microbenchmarks for page lookup and **getpage** operations, and our impact on Trapeze performance. We follow this with an examination of overheads associated with the host resident table and the tradeoffs inherent in choosing the row size for this table. Finally, we present application-level results.

4.1. Experimental setup

Our experiments were conducted on a cluster of 266-MHz Pentium II PCs with 128-MB of memory running FreeBSD 2.2.2 and with a page size of 4-KB. The PCs are connected by the Myrinet network that uses 33-MHz LANai 4.1 network processors with 1-MB of on-board SRAM. Our prototype systems are modifications of the Trapeze Myrinet control program that runs on the LANai.

Results are presented for GMS and both GMS-NP prototypes: GMS-NP(HM), which stores the hash table in host memory, and GMS-NP(NM), which stores it in network-interface memory.

4.2. Getpage timeline

Figure 5 shows a detailed timeline of a getpage operation's directory node processing for the three implementations. There is a line in the figure for each system resource involved. **Recv Wire** is the network wire time to receive a message into network-interface memory. **Network P** is network-processor compute time. **HDMA** is the host-DMA time. **Host P** is host-processor compute time. **Send Wire** is the network wire time to copy a message from interface memory into the network. The timings are for a single **getpage** request with no hash collision and no other Trapeze traffic.

For GMS, Trapeze generates the host interrupt 6.5 μs after receiving the **getpage** request message. The host processing completes a minimum of 21.5 μs later, at which time the network processor sends the resulting page-request message to the storing node.

For GMS-NP(HM), the network processor decodes the **getpage** message and readies the first hash-table DMA in 4 μs. It takes an additional 4.5 μs to locate the correct entry in the row assuming the desired entry is the first one scanned. Notice that the timeline clearly shows the overlap of Network-P and HDMA due to our host-DMA-pipelining optimization described in Section 3.3. Without this optimization, the hash lookup would be delayed until the HDMA completed.

After the entry is located, it takes another 5 μs to prepare the outgoing page-request message. Assuming the send channel is free, the message is transferred immediately to the network, as shown in the figure; otherwise, the message is copied and added to a send queue. Finally, after sending the request message, the network processor modifies the hash table entry to record the page's new state and DMAs this entry back to host memory.

For GMS-NP(NM), the DMA transfers are not needed and the scan requires 3.5 μs. This time is 1-μs less than for GMS-NP(HM), because in the host-memory case, the scan is competing with the host DMA for access to LANai memory; this is described in detail in Section 4.4. For this same reason, the time to prepare the outgoing message in GMS-NP(NM) is only 2.5 μs compared to 5 μs for GMS-NP(HM).

4.3. Microbenchmarks

Metrics for our microbenchmarks are reported for various states of the directory-host CPU cache. The configurations we consider are idle (I), active (A), and cold cache (C). In both the active and cold cache states the directory node is running a memory intensive benchmarking program. In the cold cache state, neither the L1 or L2 cache contains any instructions or data needed by the directory lookup. The numbers reported are the median of 1000 trials.

Table 1 lists how much each processor — network and host — contributes to the latency of a page lookup on the di-

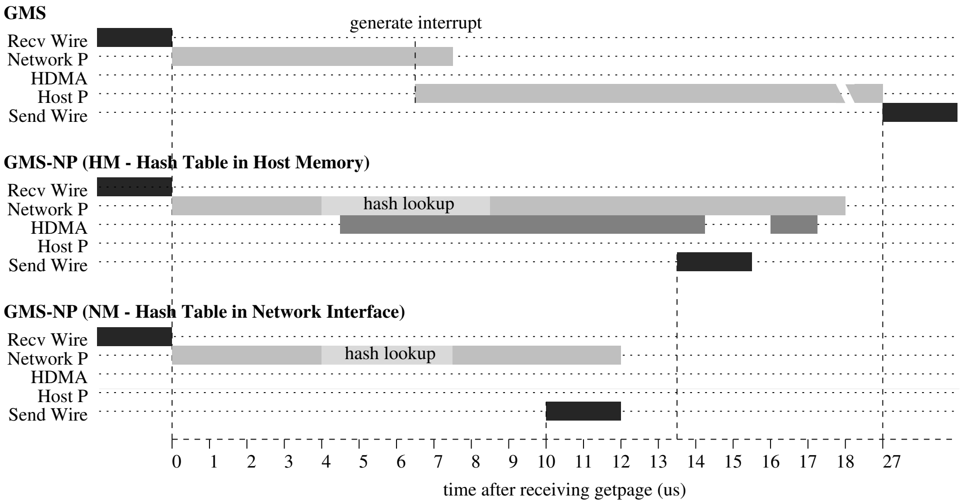

Figure 5. Detailed timeline of getpage look-up latency on the directory node for all three implementations.

State	Latency (μs)	
	GMS	GMS-NP(HM)
Idle (I)	185	166
Active (A)	191	167
Cold (C)	311	175

Table 2. Latency of getpage.

System	Latency (μs)	Throughput (MBytes/s)
Trapeze	73.7	111
Trapeze/GMS-NP	75.2	109

Table 3. Impact of GMS-NP on standard Trapeze latency and throughput.

rectory node. The table shows two GMS configurations, idle (I) and cold-cache (C); the active configuration is considered in Table 2. There is only one row in the table for each of the GMS-NP implementations, because GMS-NP performance is the same for all three configurations. The table shows that GMS-NP(HM) reduces lookup latency by 50% when compared to GMS(I), and by almost 90% relative to GMS(C). Note that the latency numbers in Table 1 are slightly lower than the overheads shown in Figure 5; this difference is due to the partial overlap between network and host processor operations.

Table 2 shows the latency of the entire **getpage** operation, which begins when the requesting node sends the **getpage** message to the directory node and ends when the requesting node receives the page from the storing node. GMS is slowed when the directory node has an active program (A, C) due to host-processing overhead and cache effects on the host. GMS-NP is slightly slowed in the cold-cache case due to host memory bus contention during the DMA of hash table rows. GMS-NP is 10% faster than GMS when the directory node is idle (I), 13% faster when the directory node is active (A), and 44% faster if the directory node's cache is cold (C).

Table 3 shows that the GMS-NP modifications on Trapeze result in a 2% slowdown in one-way latency and throughput when sending 8-KB messages.

4.4. Host DMA pipelining performance

This section provides a detailed examination of the overheads associated with fetching hash-table rows from host memory and scanning them to locate a desired page.

In Figure 6, the line labeled *fetch* is the time required for the LANai host-memory DMA engine to transfer a variable number of 32-byte hash-table entries from host memory to LANai memory. Programming the LANai DMA registers to initiate a transfer takes 0.9 μs. After that, latency is linear at about 0.25 μs per entry. We used this measurement to determine the total latency for fetching hash-table rows of various sizes. A 16-entry row, for example, requires 4.9 μs and a 32-entry row requires 8.9 μs.

The time required for our current implementation to scan a single hash-table entry to match a four word lookup key is 1.2 μs. If this scan is running at the same time as a host-memory DMA, latency increases to 1.6 μs due to competition for LANai memory between the two operations.

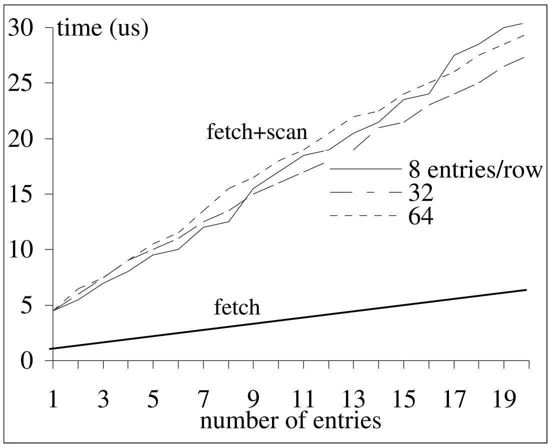

Figure 6. Hash-table *fetch* time as function of the number of entries transfered (x-axis), and *fetch + scan* time as a function of row size and number of entries scanned. Three row sizes are shown: 8 entries, 32 entries, and 64 entries; each entry is 32-bytes long.

Recall from Section 3.3 that we overlap the scanning of a row with the DMA that fetches it. In order to provide an optimal implementation of this pipelining we need to know the ratio of fetch to scan time. That is, we need to ensure an entry will be present in network memory before we attempt to scan it. From our measurements, we can now see that in the 1.6 μs it takes to scan a single entry, the LANai host-DMA engine will have transferred 6.4 32-byte entries. Therefore, after waiting for the arrival of the first entry and scanning it, the next six entries should have arrived in interface memory. The scan loop thus checks exponentially less frequently for more entries: before the first, after the first, after the seventh, after the thirty-seventh, and so on.

4.5. Choosing the row size

In GMS-NP(HM) row size is an important parameter for overall performance, because the key overhead in each of the directory operations is the DMA transfer of the hash-table row from host memory to the network processor. Figure 6 shows the time required to *fetch+scan* a hash table entry. The x-axis indicates the number of entries scanned to locate the desired entry and the y-axis the total latency of the lookup including both fetch and scan times. The graph has three lines that correspond to the latency for three different row sizes: 8, 32, and 64 entries of 32-bytes each.

The graph shows that when nine or fewer entries are scanned, the small 8-entry row performs modestly better than the others. If more entries must be scanned, however, the larger 32-entry row yields lower lookup latency due reduced DMA-initiation overheads. Not shown in Figure 6, when more than 96 entries must be scanned the larger 64-entry rows perform better. However, if more than 96 entries are scanned by the network processor, the scanning overhead is high enough to negate all of the benefits of running there instead of on the host. Our prototype thus uses a row size of 32 entries.

State	**Latency (μs)**	
	GMS	**GMS-NP(HM)**
Idle (I)	209	203
Active (A)	219	206
Cold (C)	343	210

Table 4. Latency of a remote page fault at the application level.

System	**Utilization (5236 gp/s)**		**Utilization (10000 gp/s)**	
	NP	**HP**	**NP**	**HP**
GMS (I)	4%	13%	7%	22%
GMS (A)	4%	18%	7%	29%
GMS-NP (HM) (I)	10%	0%	19%	0%
GMS-NP (HM) (A)	12%	0%	22%	0%

Table 5. Peak directory-node network- and host-processor utilization.

4.6. Application-level results

Table 4 shows remote-memory page fault latencies seen by a user-mode program that accesses a large file mapped into its address space (i.e., using **mmap()**). Remote-memory page faults in GMS-NP are 4% faster than GMS when the directory node is idle (I), 6% faster when the directory node is active (A), and 39% faster when the directory node's cache is cold (C).

Table 5 shows the peak network- and host-processor utilizations due to directory operations for rates of 5236 **getpage**/s and 10000 **getpage**/s. The first rate corresponds to the maximum **getpage** rate a single-threaded program could generate based on the latency measurements in Table 4. Assuming that directory lookups are spread evenly across every node, this represents the maximum rate seen by any directory node. At this rate, GMS imposes an overhead of 18% on an active directory node, while GMS-NP imposes no overhead at all. Furthermore, if lookups are not spread evenly or if a multi-threaded program generates a higher **getpage** rate, the overhead will be higher. If, for example, a directory node received 10000 **getpage**/s, GMS would impose an overhead of 29%.

Of course, GMS-NP is able to eliminate this host-processor overhead by increasing the overhead on the network processor. At the 5236-**getpage**/s rate, the 18% host overhead of GMS(A) is eliminated by increasing network-processor utilization from 4% to 12%. At the 10000 **getpage**/s, GMS-NP(A)'s network processor utilization scales linearly to 22%, replacing GMS's 29% host utilization.

This tradeoff favors GMS-NP for two reasons. First, the overall overhead of GMS-NP is significantly lower than GMS. Second, the host processor is roughly eight-times more powerful than the Myrinet network processor. As a result, the 29% host-processor utilization that GMS-NP eliminates represents approximately ten-times the work compared to the 15% network-processor utilization that GMS-NP adds.

Finally, the fundamental issue for host overhead in GMS and GMS-NP is the impact it has on application programs running on the directory node. To quantify this factor, we measured the slowdown of a program running on the directory node while it is handling page faults generated by another node. Our results show that for 5236 **getpage**/s, this program runs 17% slower with GMS than with GMS-NP and for 10000 **getpage**/s, this program slows by 28%.

4.7. Impact of future network hardware

The performance of GMS-NP is directly affected by specific features of our network interface. If we had been presented with a less restrictive set of hardware constraints we anticipate that the performance improvement of GMS-NP compared to host-based GMS would be more dramatic.

One hardware constraint that significantly impacts performance is contention for the memory bus shared between the network processor and the DMA engines on the Myrinet network interface. As a direct result of this bus contention, program execution slows by as much as 50% when one DMA is active and stops completely when both host and wire DMAs are active. If this contention was eliminated, or at least reduced by the presence of a CPU cache or faster host DMA using a 64-bit PCI bus, we anticipate GMS-NP would have lower latencies than those reported here. Additionally, if host DMA overhead was significantly reduced, our decision to pipeline the fetch and scan operations may need to be revisited.

5. Conclusions

Modern gigabit networks can deliver data at rates that swamp the host IO bus and CPU. As a result, host overheads now dominate the performance of many network operations. This paper investigates a potential solution to this problem that utilizes programmable network processors to offload selected protocol operations from the host.

GMS-NP implements a global page directory entirely on embedded network processors in a Myrinet gigabit network. Compared to GMS's host-based approach, GMS-NP eliminates host-processor overhead and modestly increases network-processor overhead to perform the same operations. Taking into account the differences in processor performance, GMS-NP performs directory operations on the network processor roughly ten-times more efficiently than GMS does on the host.

GMS-NP also improves directory lookup latency compared to GMS. While lookup latency improved significantly by 50% to 90%, the improvement in page-fault latency was more modest, because page-fault latency is dominated by the network transfer of the requested 4-KB page and not by the directory lookup time. Nevertheless, GMS-NP improved remote-memory page fault latency by at least 4%. If the directory host processor instruction and data caches are cold, this improvement increases to 39%.

Technology trends favor approaches such as GMS-NP that move higher-level operations to commodity network processors. The economics of processor design indicate that network-processor performance can increase at the same rate as host-processor performance, remaining at roughly 15% of cutting-edge processors. As network processors increase in power, the benefits of performing selected operations there also increases. As we have demonstrated, doing so can significantly improve resource utilization and thus lower latency and host-overhead.

References

[1] Control Data Corporation (CDC) Systems. *http://www.tno.nl/instit/fel/museum/computer/6400pps.html*.

[2] The First Commercial Computers. *http://kandor.isi.edu/aliases/PowerPC_Programming_Info/intro_to_risc/irt2_history4.html*.

[3] Next generation adapter design and optimization for gigabit ethernet. *www.alteon.com/products/white_papers/adapter.html*.

[4] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] G. Buzzard, D. Jacobson, M. Mackey, S. Marovitch, and J. W. H. Labs). An implementation of the Hamlyn sender-managed interface architecture. In *2nd USENIX Symposium*

on Operating System Design and Implementation (OSDI), pages 245–60, October 1996.

[8] J. S. Chase, A. J. Gallatin, A. R. Labeck, and K. G. Yokum. Trapeze messaging API. *Technical Report CS-1997-19, Duke University, Department of Computer Science*, November 1997.

[9] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[10] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnick, L. Iftode, and K. Li. Early experience with message-passing on the Shrimp multicomputer. In *Proc. of the 23rd International Symposium of Computer Architecture*, May 1996.

[11] M. E. Fiuczynski, R. P. Martin, B. N. Bershad, and D. E. Culler. SPINE: An operating system for intelligent network adapters. *UW TR-98-08-01*, 1998.

[12] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. SPINE - a safe programmable and integrated network environment. *Eighth ACM SIGOPS European Workshop*, September 1998.

[13] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. E. III, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 258–67, October 1996.

[14] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *In Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[15] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, and et al. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1992.

[16] A. M. Mainwaring and D. E. Culler. Active messages: Organization and applications programming interface. *http://now.CS.Berkeley.EDU/Papers/am-spec.ps*, 1995.

[17] G. Powers. A front end telnet/rlogin server implementation. In *UniForum 1986 Conference Proceedings*, pages 27–40, February 1986.

[18] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on Myrinet. In *Workshop PC-NOW, IPPS/SPDP98*, 1998.

[19] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally managed memory system. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98)*, June 1998.

[20] R. Y. Wang, A. Krishnamurthy, R. P. Martin, T. E. Anderson, and D. E. Culler. Modeling and optimizing communication pipelines. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems*, 1998.

[21] M. Welsh, A. Basu, and T. von Eiken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects V*, Aug 1997.

[22] K. G. Yokum, J. S. Chase, A. J. Gallatin, and A. R. Labeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–52, August 1997.