Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet

Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, and Kenneth G. Yocum,
Duke University
Michael J. Feeley,
University of British Columbia

Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet

Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, and Kenneth G. Yocum[*]
Department of Computer Science
Duke University
{*anderson, chase, gadde, gallatin, grant*}*@cs.duke.edu*

Michael J. Feeley
Department of Computer Science
University of British Columbia
feeley@cs.ubc.ca

Abstract

Recent advances in I/O bus structures (e.g., PCI), high-speed networks, and fast, cheap disks have significantly expanded the I/O capacity of desktop-class systems. This paper describes a messaging system designed to deliver the potential of these advances for network storage systems including cluster file systems and network memory. We describe *gms_net*, an RPC-like kernel-kernel messaging system based on Trapeze, a new firmware program for Myrinet network interfaces. We show how the communication features of Trapeze and *gms_net* are used by the Global Memory Service (GMS), a kernel-based network memory system.

The paper focuses on support for zero-copy page migration in GMS/Trapeze using two RPC variants important for peer-peer distributed services: (1) *delegated RPC* in which a request is delegated to a third party, and (2) *nonblocking RPC* in which replies are processed from the Trapeze receive interrupt handler. We present measurements of sequential file access from network memory in the GMS/Trapeze prototype on a Myrinet/Alpha cluster, showing the bandwidth effects of file system interfaces and communication choices. GMS/Trapeze delivers a peak read bandwidth of 96 MB/s using memory-mapped file I/O.

1 Introduction

Two recent hardware advances boost the potential of cluster computing: switched cluster interconnects that can carry 1Gb/s or more of point-to-point bandwidth, and high-quality PCI bus implementations that can handle data streams at gigabit speeds. We are developing system facilities to realize the potential for high-speed data transfer over Myricom's 1.28 Gb/s Myrinet LAN [2], and harness it for cluster file systems, network memory systems, and other distributed OS services that cooperatively share data across the cluster. Our broad goal is to use the power of the network to "cheat" the I/O bottleneck for data-intensive computing on workstation clusters.

This paper describes use of the Trapeze messaging system [27, 5] for high-speed data transfer in a network memory system, the Global Memory Service (GMS) [14, 18]. Trapeze is a firmware program for Myrinet/PCI adapters, and an associated messaging library for DEC AlphaStations running Digital Unix 4.0 and Intel platforms running FreeBSD 2.2. Trapeze communication delivers the performance of the underlying I/O bus hardware, balancing low latency with high bandwidth. Since the Myrinet firmware is customer-loadable, any Myrinet network site with PCI-based machines can use Trapeze.

GMS [14] is a Unix kernel facility that manages the memories of cluster nodes as a shared, distributed page cache. GMS supports remote paging [8, 15] and cooperative caching [10] of file blocks and virtual memory pages, unified at a low level of the Digital Unix 4.0 kernel (a FreeBSD port is in progress). The purpose of GMS is to exploit high-speed networks to improve performance of data-intensive workloads by replacing disk activity with memory-memory transfers across the network whenever possible. The GMS mechanisms manage the movement of VM pages and file blocks between each node's *local page cache* — the file buffer cache and the set of resident virtual pages — and the network memory *global page cache*.

This paper deals with the communication mechanisms and network performance of GMS systems using Trapeze/Myrinet, with particular focus on the support for zero-copy read-ahead and write-behind of sequentially accessed files. Cluster file systems that stripe data across multiple servers are typically limited by

[*]This work is supported by the National Science Foundation under grants CCR-96-24857 and CDA-95-12356, equipment grants from Intel Corporation and Myricom, and a software grant from the Open Group.

the bandwidth of the network and communication system [23, 16, 1]. We measure synthetic bandwidth tests that access files in network memory, in order to determine the maximum bandwidth achievable through the file system interface by any network storage system using Trapeze. The current GMS/Trapeze prototype can read files from network memory at 96 MB/s on an AlphaStation/Myrinet network. Since these speeds approach the physical limits of the hardware, unnecessary overheads (e.g., copying) can have significant effects on performance. These overheads can occur in the file access interface as well as in the messaging system. We evaluate three file access interfaces, including two that use the Unix *mmap* system call to eliminate copying.

Central to GMS is an RPC-like messaging facility (*gms_net*) that works with the Trapeze interface to support the messaging patterns and block migration traffic characteristic of GMS and other network storage services. This includes a mix of asynchronous and request/response messaging (RPC) that is *peer-to-peer* in the sense that each "client" may also act as a "server". The support for RPC includes two variants important for network storage: (1) *delegated RPCs* in which requests are delegated to third parties, and (2) *nonblocking RPC* in which the replies are processed by *continuation* procedures executing from an interrupt handler. These features are important for peer-to-peer network storage services: the first supports directory lookups for fetched data, and the second supports lightweight asynchronous calls, which are useful for prefetching. When using these features, GMS and *gms_net* cooperate with Trapeze to unify buffering of migrated pages, eliminating all page copies by sending and receiving directly from the file buffer cache and local VM page cache.

This paper is organized as follows. Section 2 gives an overview of the Trapeze network interface and the features relevant to GMS communication. Section 3 deals with the *gms_net* messaging layer for Trapeze, focusing on the RPC variants and zero-copy handling of page transfers. Section 4 presents performance results from the GMS/Trapeze prototype. We conclude in Section 5.

2 High-Speed Data Transfer with Trapeze

The Trapeze messaging system consists of two components: a messaging library that is linked into programs using the package, and a firmware program that runs on the Myrinet network interface card (NIC). The Trapeze firmware and the host interact by exchanging commands and data through a block of memory on the NIC, which is addressable in the host's physical address space using programmed I/O. The firmware defines the interface between the host CPU and the network device; it interprets commands issued by the host and controls the movement of data between the host and the network link. The host accesses the network using macros and procedures in the Trapeze library, which defines the lowest level API for network communication across the Myrinet.

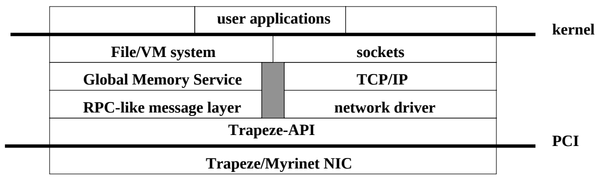

Figure 1: Using Trapeze for TCP/IP and for kernel-kernel messaging for network memory.

Like other network interfaces based on Myrinet (e.g., Hamlyn [4], VMMC-2 [13], Active Messages [9], FM [21]), Trapeze can be used as a memory-mapped network interface for user applications, e.g., parallel programs. However, Trapeze was designed primarily to support fast kernel-to-kernel messaging alongside conventional TCP/IP networking. The Trapeze distribution includes a network device driver that allows the native TCP/IP protocol stack to use a Trapeze network alongside the *gms_net* layer. Figure 1 depicts this structure. The kernel-to-kernel messaging layer is intended for GMS and other services that assume mutually trusting kernels.

2.1 Trapeze Overview

Trapeze messages are short *control messages* (maximum 128 bytes) with optional attached *payloads* typically containing application data not interpreted by the message system, e.g., a file block, a virtual memory page, or a TCP segment. Each message can have at most one payload attached to it. Separation of control messages and bulk data transfer is common to a large number of messaging systems since the V system [6].

A Trapeze control message and its payload (if any) are sent as a single packet on the network. Since Myrinet has no fixed maximum packet size (MTU), the maximum payload size of a Trapeze network is configurable, and is typically set to the virtual memory page size (4K or 8K). The Trapeze MTU is the maximum control message size plus the payload size.

Payloads are sent and received using DMA to/from aligned buffers residing anywhere in host memory. The host attaches a payload to an outgoing message using a Trapeze macro that stores the payload's DMA address and length into designated fields of the send ring entry. On the receiving side, Trapeze deposits the payload into a host memory buffer before delivering the control message.

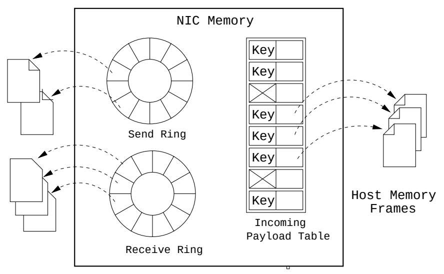

Figure 2: NIC Memory Structures for a Trapeze endpoint.

The data structures in NIC memory include an *endpoint* structure shared with the host. A Trapeze endpoint (shown in Figure 2) includes two message rings, one for sending and one for receiving. Each message ring is a circular array of 128-byte control message buffers and related state, managed as a producer/consumer queue. From the perspective of a host CPU, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring. The host sends a message by forming it in the next free send ring entry and setting a bit to indicate that the message is ready to send. When a message arrives from the network, the firmware deposits it into the next free receive ring entry, sets a bit to inform the host that the message is ready to consume, and optionally signals the host with an interrupt.

Handling of incoming messages is interrupt-driven when Trapeze is used from within the kernel. Each kernel protocol module using Trapeze (i.e., *gms_net* and the IP network driver) registers a receiver interrupt handler upcalled from the Trapeze interrupt handler.

Trapeze is designed to optimize handling of payloads as well as to deliver good performance for small messages. In a network memory system, page fault stall time is determined primarily by the time to transfer the requested page on the network. On the other hand, bursts of page transfers (e.g., for read-ahead for sequential access) require high bandwidth. The Trapeze firmware employs a message pipelining technique called *cut-through delivery* [27] to balance low payload latency with high bandwidth under load. With this technique, the one-way raw Trapeze latency for a 4K page transfer is 70μs on 300MHz Pentium-II/440LX systems with LANai 4.1 M2M-PCI32 Myrinet adapters. On these systems, Trapeze delivers 112 MB/s for a stream of 8K payloads; with 64K payloads, Trapeze can use over 95% of the peak bandwidth of the I/O bus, achieving 126 MB/s of user-to-user point-to-point bandwidth.[1]

[1]These bandwidth numbers define a “megabyte” as one million bytes. All other bandwidth numbers in this paper define 1MB as 1024*1024 bytes.

2.2 Unified Buffering for In-Kernel Trapeze

All kernel-based Trapeze protocol modules share a common pool of receive buffers allocated from the virtual memory page frame pool; the maximum payload size is set to the virtual memory page size. Since Digital Unix allocates its file block buffers from the virtual memory page frame pool as well, this allows unified buffering among the network, file, and VM systems. For example, the system can send any virtual memory page or cached file block out to the network by attaching it as a payload to an outgoing message. Similarly, every incoming payload is deposited in an aligned physical frame that can mapped into a user process or hashed into the file cache. Since file caching and virtual memory management are reasonably unified, we often refer to the two subsystems collectively as “the file/VM system”, and use the term “page” to include file blocks.

The TCP/IP stack can also benefit from the unified buffering of Trapeze payloads to reduce copying overhead by *payload remapping* (similar to [11, 3, 17]). On a normal transmission, IP message data is copied from a user memory buffer into an mbuf chain [20] on the sending side; on the receiving side, the driver copies the header into a small mbuf, points a BSD-style external mbuf at the payload buffer, and passes the chain through the IP stack to the socket layer, which copies the payload into user memory and frees the kernel buffer. We have modified the Digital Unix socket layer to avoid copying when size and alignment properties allow. On the sending side, the socket layer builds mbuf chains by pinning the user buffer frames, marking them copy-on-write, referencing them with external mbufs, and passing them through the TCP/IP stack to the network driver, which attaches them to outgoing messages as payloads. On the receiving side, the socket layer unmaps the frames of the user buffer, replaces them with the kernel payload buffer frames, and frees the user frames. With payload remapping, AlphaStations running the standard *netperf* TCP benchmark over Trapeze sustain point-to-point bandwidth of 87 MB/s.[2]

Since outgoing payload frames attached to the send ring may be owned by the file/VM system, they must be protected from modification or reuse while a transmit is in progress. Trapeze notifies the system that it is safe to overwrite an outgoing frame by upcalling a specified *transmit completion handler* routine. For example, when an IP send on a user frame completes, Trapeze upcalls the completion routine, which unpins the frame and

[2]Measured Alcor (266 MHz AS 500) to Miata (500 MHz PWS 500au), 8320-byte MTU, 1M netperf transfers, socket buffers at 1M, software TCP checksums disabled (hardware CRC only): 732 Mb/s.

releases its copy-on-write protection.

However, to reduce overhead Trapeze does not generate transmit-complete interrupts. Instead, Trapeze saves the handler pointer in host memory and upcalls the handler only when the send ring entry is reused for another send. Since messages may be sent from interrupt handlers, a completion routine could be called in the context of an interrupt handler that happened to reuse the same send ring entry as the original message. For this reason, completion handlers must not block, and the structures they manipulate must be protected by disabling interrupts. Since completion upcalls may be arbitrarily delayed, the Trapeze API includes a routine to poll all pending transmits and call their handlers if they have completed.

2.3 Incoming Payload Table

The benefits of high-speed networking are easily overshadowed by processing costs and copying overhead in the hosts. To support zero-copy communication, a Trapeze receiver can designate a region of memory as the receive buffer space for a specific incoming payload identified by a tag field. When the message arrives, the firmware recognizes the tag and deposits the payload directly into the waiting buffer. Handling of tagged payloads is governed by a third structure in NIC memory, the *incoming payload table* (IPT).

GMS uses the Trapeze IPT for copy-free handling of fetched pages in RPC replies, as described in Section 3. Ordinarily, Trapeze payloads are received into buffers attached by the host to the receive ring entries; since the firmware places messages in the ring in the order they arrive, the host cannot know in advance which generic buffer will be selected to receive any given payload, and the payload may need to be copied within the host if it cannot be remapped. Early demultiplexing with the IPT avoids this copy.

To set up an IPT mapping, the host calls a Trapeze API routine to allocate a free entry in the IPT, initialize it with the DMA address of the designated payload buffer, and return a tag value (*payload token*) consisting of an IPT index and a protection key. The payload token is a weak form of capability that can be passed in a message to another node; any node that knows the token can use it to tag a message and transmit a payload into the buffer. When the firmware receives a tagged message from the network, it validates the key against the indexed IPT entry before initiating a DMA into the designated receive buffer. The receiving host may cancel the IPT entry at any time (e.g., request timeout); similarly, the firmware protects against dangling tokens and duplicate messages by cancelling the entry when a matching message is received. If the key is not valid, the NIC drops the payload and delivers the control portion with a payload length of zero, so the receive message handler can recognize and handle the error.

At present, the IPT maps only a few megabytes of host memory, enough for the reply payloads of all outstanding requests (e.g., outstanding page fetches). This is a modest approach that meets our needs, relative to more ambitious approaches that indirect through TLB-like structures on the NIC [13, 26, 7]. We have considered a larger IPT with support for multiple transfers to the same buffer at different offsets, as in Hamlyn's *sender-based memory management* [4], but we have not found a need for these features in our current uses of Trapeze.

3 Page Transfers in GMS/Trapeze

This section outlines a Trapeze-based kernel-kernel RPC-like messaging layer designed to support cooperative cluster services. The package is derived from the original RPC package for the Global Memory Service [14] (*gms_net*), extended to use Trapeze and to support a richer set of communication styles, primarily for asynchronous prefetching at high bandwidth [24]. Although the package is generic, we draw on GMS examples to motivate its features and to illustrate their use.

Since many aspects of RPC and messaging systems are well-understood, we focus on those aspects that benefit from the Trapeze features discussed in the previous section. In particular, we explain the features for transferring pages (or file blocks) efficiently within the RPC framework, and their use by the protocol operations most critical for GMS performance: page fetches (*getpage*) from the global page cache to a local page cache, and page pushes or evictions (*putpage* or *movepage*) from a local cache to the global cache.

Section 3.2 discusses the zero-copy handling of fetched pages using the Trapeze incoming payload table (IPT); Sections 3.3 and 3.4 extend the zero-copy reply scheme to delegated and nonblocking RPC variants useful in GMS and other peer-to-peer network services. We illustrate use of nonblocking RPC to extend standard read-ahead for files and virtual memory to GMS; this allows processes to access data from network memory or storage servers at close to network bandwidth.

3.1 Basic Mechanisms

The *gms_net* messaging layer includes basic support for typed messages, stub procedures, dispatching to service procedures based on message types, and matching replies with requests. The Trapeze receiver interrupt handler directs incoming messages to *gms_net* by upcalling a registered service routine; the service routine hands off incoming requests to a server thread. However, *gms_net* it is not a true RPC system: many protocol messages do not produce replies, and there is no support for automatic stub generation. The package is best thought of as a library of procedures and macros used by the messaging stubs to build and decode messages and to direct their flow through the system. It is designed for messages with relatively simple arguments and bulk data payloads (e.g., file blocks) that are not interpreted by the message handlers themselves.

To send a message, a stub allocates a message buffer with *gms_net_makebuf*, calls routines and macros to build the message, e.g., by pushing data items into the message, and sends the message to a destination with *gms_net_sendto*. In Trapeze, *gms_net_makebuf* returns a pointer to a send ring entry, and *gms_net_sendto* releases it. Messages are typed by an operation code and a request/reply bit. Incoming requests are dispatched by using the operation code to index into a vector of registered server-side stubs. Incoming replies are handled directly by the receiver interrupt handler, either by waking up a waiting thread or by calling a reply continuation procedure as described in Section 3.4.

An important function of the RPC layer is to match incoming replies with requests. If a reply is expected, the caller makes an entry in a *call record* table before sending the request message, and places a *reply token* containing a unique call record ID into the outgoing request message. After sending the request, the calling thread or process may block on the call record entry. When the server side generates a reply, it places a copy of the reply token in the reply message. When the reply arrives, the receiver interrupt handler decodes the reply token and retrieves the call record. The call record includes all information needed to process the reply, e.g., by awakening the calling thread or process.

To transfer a page or file block in a request or reply, the stub attaches the memory frame to the message buffer as a payload. The system is inhibited from reusing the frame or overwriting it until the frame contents have been transferred to the network adapter using DMA (Section 2.2). On the receiving side, Trapeze uses DMA to deposit each received payload into a memory frame designated by the receiver.

3.2 Zero-Copy Reply Handling

GMS performance depends on efficient handling of *getpage* replies containing page payloads. When the virtual memory system or file system initiates a page fetch, it first selects the target page frame according to its policies for page replacement and other factors such as page coloring. The goal of the GMS *getpage* client stub is to arrange to transfer the incoming page directly to the waiting frame using DMA.

The RPC system uses the Trapeze incoming payload table (IPT) described in Section 2.3 for this purpose. The client-side *getpage* stub calls Trapeze to allocate an IPT entry and obtain a payload token, which is added to the reply token for the call. When the server-side *getpage* stub generates a reply, it attaches the frame containing the requested page as a payload, extracts the Trapeze payload token, and tags the outgoing reply message by placing the token in its Trapeze header. Back on the client side, the Trapeze firmware recognizes the tag in the message header as the reply payload begins to arrive on the adapter; once the tag is decoded and validated, the firmware initiates DMA of the message payload into the waiting frame.

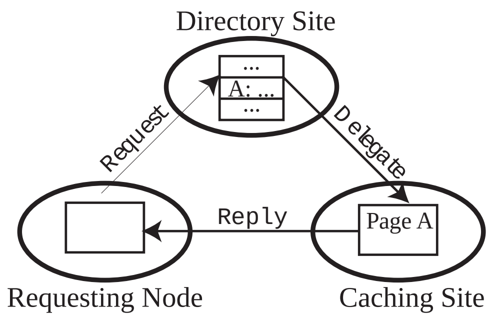

Figure 3: GMS *getpage* operation through a directory site using a delegated RPC.

3.3 Delegated RPC

Unlike traditional RPC, some GMS protocol operations involve more than one server. To fetch a remote page, for example, the *getpage* operation must first locate the page's caching site. To keep track of pages, GMS uses a distributed hash directory for pages potentially sharable by multiple nodes [14]. A requesting node locates a page's directory site by applying a globally replicated hash function to a unique page identifier. It then issues a *getpage* RPC to the directory site, which looks up the page in its portion of the directory, and forwards the request to the caching site. The caching site completes the three-way RPC by returning the page directly to the requester. We call this type of operation a *delegated RPC*.

The key idea behind delegated RPC is to allow the reply token to be passed from node to node until the call is complete; the last node in the RPC sequence then uses the reply token to complete the RPC and reply to the original requester. The delegation is transparent to the requester, which creates its reply token and includes it in the request message exactly as for an ordinary RPC.

With delegated RPC, however, a remote procedure can either return a reply or delegate the request to another peer by generating a new request message with a copy of the original reply token. Each node has the same option of either replying to the original requester or delegating the request to yet another peer. Note that each delegation call is unlike a normal RPC in that the delegated procedure never replies to its immediate caller; it either forwards the request again or replies directly to the node that initiated the delegated RPC. Most importantly, the zero-copy reply handling scheme is preserved, since the Trapeze payload token is embedded within the reply token, and so is available to the node that ultimately generates the reply.

Figure 3 shows how GMS *getpage* uses delegated RPC. The directory site for the requested page delegates the request to the caching site, forwarding the original request parameters and reply token. The caching site generates a reply, attaches the requested page as a payload, and tags the message with the payload token, as described in the previous section. It then sends the reply directly to the original requester, where it is handled in the same way as a direct reply.

3.4 Read-Ahead with Nonblocking RPC

Most file system implementations implement read-ahead to prefetch file data before it is requested. This significantly improves bandwidth for sequential file access, which is easy to detect and exploit [20]. We have extended GMS to support read-ahead in order to meet our goal of delivering files from network memory at full network bandwidth. GMS read-ahead hides fetch latency and delivers peak bandwidth by pipelining the network with a continuous stream of page fetches.

3.4.1 Nonblocking RPC

Any form of prefetching imposes new demands on the communication layer and is highly dependent on its performance. In particular, prefetching requests are RPC calls that generate replies, but the replies must be handled asynchronously and outside of the issuing thread context, so as not to block the issuing thread while the request is pending. NFS client implementations typically solve this problem by handing off read-ahead calls to a system *I/O daemon* that can wait for RPC replies without affecting user processes [22]. This solution requires a context switch to the I/O daemon for each request and response.

To reduce context switching overhead, GMS/Trapeze implements read-ahead and prefetching using *nonblocking RPCs*. To implement nonblocking RPC, *gms_net* supplements the call record with support for *continuation* procedures invoked directly from the receiver interrupt handler to process the reply. These continuations are similar to Draves et. al. [12], but they execute at interrupt time with no associated thread context. Also, each nonblocking RPC call may have several continuations; the issuing stub pushes pointers to these continuation procedures and their arguments onto a *continuation stack* linked to the call record returned by *gms_net_makebuf*. When the reply arrives, the *gms_net* receiver interrupt handler locates the call record for the reply as before, pops the continuations from the stack, and calls them in order with their arguments. Like other interrupt handling code, continuation procedures are not permitted to sleep.

Continuations in *gms_net* nonblocking RPC are related to callbacks in Rover's QRPC [19]. In Rover, asynchronous RPC calls are used to allow applications to tolerate slow and unreliable mobile networks, whereas in GMS/Trapeze their purpose is to support pipelined RPC operations (e.g., prefetching) on a fast and reliable cluster interconnect.

3.4.2 Read-Ahead from Network Memory

GMS/Trapeze activates sequential read-ahead when the file/VM system determines that accesses are sequential, and that subsequent pages are resident in the global cache but not in the local cache. It issues nonblocking RPCs to prefetch the next N pages for some configurable depth N; these requests are issued in the context of the user process accessing the data. Each prefetch request is an ordinary *getpage* operation; to the receiver, they are indistinguishable from synchronous fetch requests. The caller allocates the target page frame before the prefetch, maps it through the IPT as described in Section 3.2, and includes a reply token in the message. The server generates a reply as described in Section 3.2, possibly delegating the request to a peer as described in Section 3.3.

A record of each pending prefetch request is hashed into the local page directory so that the frame can be located if a process references the page before the prefetch completes. If a process references a page with a pending prefetch, the process is put to sleep on a call record until the read-ahead catches up. If no process blocks awaiting completion, nonblocking RPCs do not have specific timeouts. However, call records for nonblocking RPCs are maintained as an LRU cache, so each call record eventually reports failure and is reused if no reply arrives.

When each prefetch reply arrives, Trapeze transfers the payload into the waiting frame and interrupts the host. The interrupt handler uses the reply token to locate the call record, which holds a pointer to the continuation handler for asynchronous prefetch. The interrupt handler invokes the continuation, which "injects" (hashes) the frame into the local page cache, and enters it into other structures as required, e.g., an LRU list. Note that prefetched pages are not copied.

3.4.3 Deferred Continuations

Since continuations execute from the receiver interrupt handler, they must be synchronized with any kernel code that accesses the same data structures. For example, a file prefetch "inject" continuation could corrupt the internal file/VM data structures if it interrupts a process that was operating on those structures in kernel mode. An obvious solution is to disable receive interrupts for every operation on any data structure that is shared with a continuation procedure, but this would require significant reengineering of existing kernel code that does not expect to be interrupted.

We use an optimistic approach that defers execution of continuations in the rare instances when races occur. Continuation procedures are boolean functions that validate their preconditions by probing the state of relevant kernel locks before executing. If any needed locks are held, this indicates that an operation was in progress when the interrupt was delivered, and the continuation cannot execute. In this case, the continuation returns false with no side effects, and is placed on a *deferred continuations* queue serviced by a kernel daemon thread. Deferred continuations incur higher latency and overhead, but they execute safely. This technique is similar to optimistic active messages [25].

4 Performance

This section presents performance measurements of *gms_net* and sequential file access using the GMS/Trapeze prototype in Digital Unix 4.0. We measure all the RPC variants presented in order to illustrate the costs and benefits of the *gms_net* and Trapeze mechanisms discussed in the previous sections. The file access tests are intended to show the rate at which a GMS/Trapeze client can source and sink data to network storage servers using these communication mechanisms for payload transfer and asynchronous read-ahead. A secondary goal is to show the effect of the operating system kernel interface chosen to read or write file data at these speeds, which are close to the limits of the hardware.

The systems used for these measurements are Alcor and Miata, two DEC Alpha platforms based on the 21164 CPU. Our Alcors (AlphaStation 500 and 600) are clocked at 266 MHz, and use a CIA host-PCI bridge (ASIC pass 2). Miata (PWS 500au) is a newer 500 MHz machine with a Pyxis bridge (ASIC pass 257). All systems are equipped with M2F-PCI32 Myrinet LAN adapters connected through an 8-port switch (M2F-SW8). Both the CIA and Pyxis I/O bridges deliver almost the full bandwidth of the 32-bit 33MHz PCI standard (132 MB/s) in one direction; however, Alcor receives at half-bandwidth and Miata sends at half-bandwidth. Most of the experiments in this section involve a one-way high-volume data transfer: to circumvent the bridge bottlenecks Alcor is always the sender and Miata is always the receiver. Release of the improved Miata-II is imminent, but it is not yet available at the time of this writing.

4.1 RPC Microbenchmarks

Table 1 shows latency and bandwidth results from kernel-kernel RPC microbenchmarks using 16-byte control messages and payload sizes of 0 bytes, 4K bytes, and 8K bytes. In these experiments the request message is a 16-byte control message that generates a reply with an attached payload. The client is a Miata; the server(s) are Alcors. For these experiments, Trapeze was configured to use DMA for control messages in order to reduce overhead at the cost of higher latency.

The table presents measurements for ordinary request/response RPC (2-way) and delegated (3-way) RPC, for three reply-handling variants: traditional blocked caller (*wait*), nonblocking continuation (*cont*), and deferred continuation (*defer*). For the replies carrying payloads, we measured the effect of three payload handling schemes: (1) "solicited" payloads received into frames mapped by the Trapeze IPT, (2) "unsolicited" payloads received into a generic payload buffer attached to the receive ring entry (as for a received GMS *putpage* or *movepage*), and (3) "unsolicited with copy", in which the received payload is copied from a generic payload buffer into a reply buffer not mapped through the IPT. The third variant is intended to demonstrate the value of the IPT for copy-free reply handling.

What is important here is the low incremental cost and high bandwidth of pagesize payloads, and the effects of the payload handling techniques presented in Sections 2 and 3. For example, we can determine from the *2-way wait* numbers that the marginal transfer latency of a solicited payload is about 54μs for 4K and 92μs for 8K, including the cost to map the receiving frame through the IPT. With nonblocking RPCs and continuations, *gms_net* preserves 87% of the 88 MB/s of bandwidth that raw Trapeze provides with 4K payloads on this platform, and over 92% of the 105 MB/s of raw Trapeze bandwidth using 8K payloads. Interestingly, at most 7% of the remaining throughput is sacrificed by copying the payload at the receiver; this reflects the excellent memory system

Msg/Payload		16/0			16/4096		16/8192	
		Latency (μsec)	Bandwidth (msgs/sec)		Latency (μsec)	Bandwidth (MB/sec)	Latency (μsec)	Bandwidth (MB/sec)
2-way	Wait	73.7	13600	Solicited	127.6	32.1	165.9	48.8
				Unsolicited	134.0	30.6	169.6	48.0
				Unsol+Copy	167.4	24.5	217.2	37.5
	Cont	69.0	64500	Solicited	125.8	78.9	163.5	95.1
				Unsolicited	128.0	79.2	166.7	97.4
				Unsol+Copy	160.8	69.7	210.8	90.5
	Defer	73.1	64500	Solicited	131.6	76.9	169.8	94.9
				Unsolicited	135.2	70.7	170.5	96.7
				Unsol+Copy	170.0	50.4	218.3	92.9
3-way	Wait	109.5	9150	Solicited	163.4	25.1	201.9	40.3
				Unsolicited	168.8	24.3	205.9	39.7
				Unsol+Copy	201.8	20.3	254.0	32.2
	Cont	104.6	64600	Solicited	162.6	79.4	199.5	94.7
				Unsolicited	163.6	78.8	202.3	97.1
				Unsol+Copy	195.2	63.6	246.4	91.3
	Defer	108.9	64500	Solicited	166.6	77.6	205.7	95.5
				Unsolicited	169.6	65.7	206.1	96.3
				Unsol+Copy	204.0	47.9	254.4	88.1

Table 1: RPC microbenchmark results for *gms_net* on an AlphaStation/Myrinet network.

bandwidth of the Pyxis-based Miata (this is also apparent on our Intel platforms using the new 440LX chipset).

Several other points are worthy of note. Nonblocking RPCs show a modest improvement in latency because there is no process context switch to handle the reply; however, that benefit is more than lost if the continuation must be deferred. Delegated (3-way) RPCs — which are common for shared file accesses in GMS — exact a high price in latency, but have little effect on bandwidth. Solicited payloads are even cheaper than unsolicited payloads, despite the need to set up and tear down an IPT entry; this is apparently due to the cost of returning the received buffer to the VM page frame pool, and allocating a new one to replace the buffer frame lost from the receive ring entry.

4.2 GMS/Trapeze File Access Speed

We now present the performance of sequential file access on the GMS/Trapeze prototype. In these experiments, the servers are GMS network memory servers with sufficient aggregate memory to hold all the data accessed by the benchmark. Thus all disk access is removed from the critical path, reflecting the "cheating" theme of this paper. The purpose is to view the file system as an extension of the network protocol stack, and measure the bandwidth achievable through the file system interface.

For these experiments, the file system partition where the benchmark files reside is configured to use two variants of the GMS caching policies to improve delivered bandwidth. First, blocks from these files are *sticky* in the global cache: reads of these blocks from network memory are nondestructive, so that each block fetched by a client will occupy memory on both the client and the caching site. This policy uses network memory less efficiently, but duplicated blocks need not be written back to network memory when they are evicted from the client, assuming they are clean. Second, the partition is configured as a *scratch* file system that uses network memory as a *writeback* cache: dirty blocks demoted from local memory to global memory are not immediately written to disk. The writeback policy is unsafe in that file data may not survive failure of a caching site, but it allows file writes to proceed at network speeds, so it serves as a measure of the rate at which a Trapeze client can sink dirty data to a server over Myrinet.

Our results report overhead as well as I/O bandwidth. At Myrinet network speeds, file access overhead is as important as raw I/O bandwidth: it is of limited value to read files at 90 MB/s if overheads consume all of the CPU cycles or memory system bandwidth, leaving the application no resources to process the data. Many of our techniques are targeted at reducing overhead (e.g., by avoiding copies) rather than increasing bandwidth directly.

In fact, there is a complex relationship between overhead and bandwidth. One measure of overhead is system

CPU utilization — the percentage of CPU time spent in the kernel. System CPU utilization grows with I/O bandwidth due to fixed overheads for handling each page of data. For typical applications, user CPU utilization also grows with bandwidth, since the application spends time handling each page as well. As the combined effects of user and system processing push the CPU toward saturation, the user program and the system begin to issue I/O requests more slowly, and bandwidth begins to drop.

4.2.1 File Access Interfaces

Our highest bandwidths and lowest overheads are achieved using the file mapping *mmap* system call, rather than the traditional read/write interface. Differences among these interfaces have a negligible effect on delivered bandwidth at disk speeds, but the effect is substantial at gigabit-per-second network speeds. This is true even on modern platforms whose memory systems have sufficient bandwidth to serve the CPU and the I/O system simultaneously (e.g., Alpha Miata or Intel Pentium-II/440LX). The effect can be dramatic on platforms with lower memory system bandwidth (e.g., Alcor).

The experiments use three different file access schemes. The *stream* option uses *read* and *write* system calls. The *mmap* experiments use a single *mmap* system call to map the entire file into virtual memory, avoiding the copying inherent in the *read* and *write* interface. The *memory-mapped block* (MMB) experiments use a hybrid scheme that combines the benefits of the *stream* and whole-file *mmap* access policies. MMB is an an attractive interface for high-volume file access when performance is important and the uniform addressing of whole-file *mmap* is not required.

MMB uses the *mmap* system call in a block-oriented fashion, repeatedly mapping N regions of virtual memory of size B to different ranges of offsets in the file. Our MMB experiments use $B = 256K$ and $N = 2$ (double buffering). Like whole-file *mmap*, MMB is a zero-copy file access scheme; it incurs slightly higher system call overhead than *mmap*, but our results show that this is insignificant with sufficiently low N and sufficiently large B, and is overshadowed by other benefits. Like *stream*, MMB allows the application to explicitly specify its accesses to the kernel. This information can be exploited by the kernel to improve performance for applications that use MMB. Moreover, MMB is an asynchronous interface, allowing the application to specify accesses early in order to overlap file access with computation.

We have modified our Digital Unix kernels to detect MMB accesses and respond with the following policies:

- When an MMB *mmap* call is issued, the kernel immediately initiates an asynchronous prefetch of all pages in the newly mapped region, superseding the usual read-ahead policy.
- The memory frames that previously backed the remapped region are released, discarding the mapped-over file blocks from the local file cache, and pushing them to the network if necessary. This allows the application to control the local cache replacement policy by selecting the region to remap, and reduces the overhead of the paging daemon and LRU eviction code.
- We have extended the *mmap* interface with a MAP_OVERWRITE flag that allows the application to specify that the mapped file region will be overwritten without reading it; in this case, the kernel may simply leave the existing frames mapped, but change their identity in the file cache. The new flag fixes an inherent flaw in *mmap*: when a process references a page in a mapped region, the kernel does not know if the process will read from that page, so it must initialize the frame by zeroing it or reading it from the file. Our approach will expose corrupt data to a process that mistakenly reads from a region mapped with MAP_OVERWRITE, but it will never violate security by leaking data belonging to another process.

We emphasize that with the exception of the new flag, these policies do not change the *semantics* of the standard *mmap* interface, but only its performance. We leave a detailed study of the MMB interface and its usage to future work.

4.2.2 Sequential Benchmark Results

Figure 4 compares bandwidths delivered to a user process reading and writing files sequentially through the *stream*, *mmap*, and MMB interfaces. To show the effect of user program activity, we report bandwidths and CPU utilizations as the test program touches varying amounts of the data on each page. The benchmark repeatedly accesses a file that overflows the local file cache, varying the size read or written for each page from one word up to the 8K page size. We averaged ten iterations with 120,000 page accesses, moving just under a gigabyte of data to or from the process for each test. Variance is negligible for all tests.

The read tests show that bandwidth starts high and decreases as the test program accesses more of the data. The highest bandwidths are delivered at the left end of the graph; these are sparse read tests in which the application reads and loads only one word of each fetched page. Given the excellent memory system bandwidth on

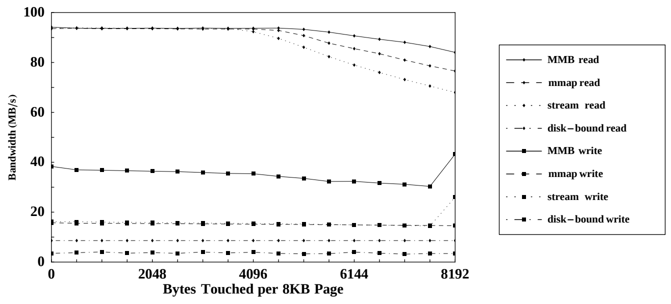

Figure 4: File access bandwidths for sequential read and write tests.

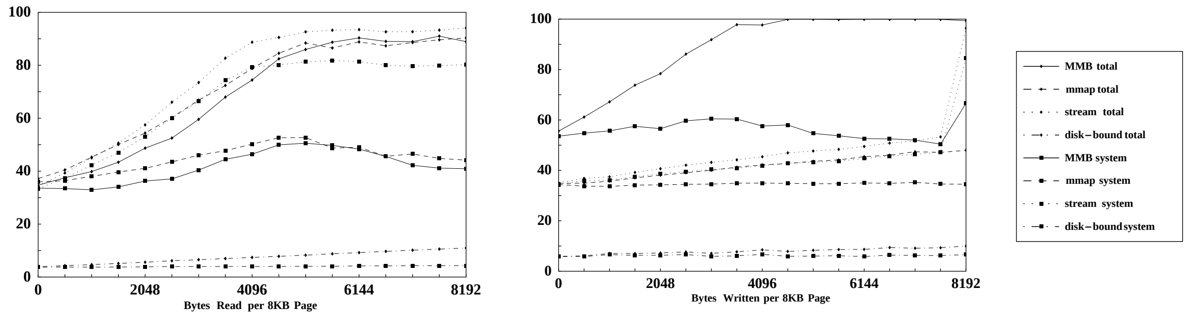

Figure 5: System CPU utilization (%) and total CPU utilization (%) for the read and write tests.

the Miata, the three interfaces deliver the same bandwidth (96 MB/s) until about half the data is touched. Until this point, every access stalls waiting for data to arrive and interface overheads are masked by read-ahead. In comparison, NFS reads from server memory at 13.5 MB/s on the same platform (using standard Myrinet firmware and sufficient I/O daemons); GMS with read-ahead enabled delivers 25 MB/s using IP/Myrinet with the standard firmware rather than Trapeze.

Figure 5 shows CPU utilizations for the same experiments. The system overhead of *stream* rises quickly as the program touches more of its data and the read and write system calls copy more data in and out of the user process. In contrast, *mmap* and MMB avoid the copy, and the system overhead stays relatively flat (the hump at 3-5KB appears to be due to the combined effects of high bandwidth and memory system contention from the user process). In the dense read experiments at the right end of the graphs, bandwidth delivered through *stream* drops to 68 MB/s, as the saturated 500 MHz Alpha CPU spends 80% of its time executing I/O code in the kernel. In contrast, under MMB GMS/Trapeze still delivers almost 84 MB/s, leaving 59% of the CPU time free for the application to process the data. However, simply loading each word up to the CPU saturates the system at these speeds, due to memory system delays. For all three interfaces this experiment is limited by CPU and memory bandwidth rather than the network.

We ran the write tests with Alcor as the client, since its I/O system delivers full send bandwidth. While all tests benefit from zero-copy asynchronous writes (write-behind), file write bandwidths are much lower than read bandwidths for three reasons. First, in the partial-write tests, the kernel must fetch each page (or zero it) before modifying it. Second, these reads do not benefit from read-ahead, since partial sequential writes are rare in practice. Third, Alcor has a slower CPU, its I/O system can receive at only 66 MB/s, and its memory system exacerbates overheads: an Alcor transmitting at full speed delivers less than 25% of its memory system bandwidth to the CPU. Using raw Trapeze, an Alcor can send raw 8KB payloads at 105 MB/s, but the bandwidth drops to 58 MB/s if the sender overwrites each payload buffer before sending it.

For partial writes, MMB delivers the highest bandwidth because it prefetches implicitly on each block access. The bandwidth/overhead spike for the dense write tests at the right end of the graphs occurs because the test program overwrites all of the data, and it is no longer necessary to read each page before writing it. While *stream* and MMB (using the MAP_OVERWRITE flag) recognize this case, *mmap* cannot detect the full-block write in advance, and continues to read before writing. *Stream* delivers 26 MB/s for dense writes on Alcor, while MMB delivers the peak of 46 MB/s (79% of the platform maximum for this test) since it does not copy the data and also avoids fetching or zeroing the pages before they are overwritten.

5 Conclusion

This paper focuses on features of the Trapeze messaging system that support data-intensive cluster OS services, and their use in the GMS network memory system. The paper makes three contributions:

- It describes useful techniques for copy-free handling of page and block transfers in a network storage system, including an *incoming payload table* (IPT) on the NIC, RPC stub support for zero-copy replies using the IPT, and unified buffering in the network, file, and virtual memory subsystems.
- It shows how to implement useful RPC variants for peer-peer OS services, including delegated RPC and nonblocking RPC using continuations.
- It illustrates use of these techniques in a network memory system that meets aggressive performance goals on a gigabit Myrinet network, using the file system interface to source and sink data at close to network and I/O bus speeds. The GMS/Trapeze prototype features integrated support for high-speed network storage at three levels of the system: (1) the network interface firmware, (2) file/VM and networking subsystems, and (3) the system call interface, with optimizations for the memory-mapped block file access scheme.

6 Acknowledgments

Chandu Thekkath and Geoff Voelker contributed to the original *gms_net* implementation, and some of the ideas in this paper originated with them. Hank Levy and Anna Karlin have contributed to all aspects of the GMS project. We thank Bob Felderman at Myricom and Mark Shand, Don Rice, Marc Viredaz and Lance Berc at Digital for help with the hardware, and Gretta Bartels for help preparing the document.

7 Availability

Trapeze is free software, and a GMS/Trapeze port to FreeBSD is in progress. For information about availability see the Trapeze web site:

```
http://www.cs.duke.edu/ari/trapeze
```

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.

[3] José Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 277–291, Seattle, WA, October 1996. USENIX Assoc.

[4] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, October 1996. USENIX Assoc.

[5] Jeffrey S. Chase, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Trapeze messaging API. Technical Report CS-1997-19, Duke University, Department of Computer Science, November 1997.

[6] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[7] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual network transport protocols for Myrinet. In *Hot Interconnects Symposium V*, August 1997.

[8] D. Comer and J. Griffioen. A new design for distributed systems: the remote memory model. In *Proceedings of the 1990 Summer USENIX*, June 1990.

[9] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP 97)*, 1997.

[10] Michael D. Dahlin, Randolph Y. Wang, and Thomas E. Anderson. Cooperative caching: Using

remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 267–280, November 1994.

[11] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proceedings of IEEE Infocom*, 1997. WUCS-96-12 technical report.

[12] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 122–136, December 1991.

[13] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Hot Interconnects Symposium V*, August 1997.

[14] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[15] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[16] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[17] Hsiao-Keng and Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[18] Hervé A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans III, Anna R. Karlin, Henry M. Levy, and Mary K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 258–267, October 1996.

[19] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 156–171, December 1995.

[20] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Unix Operating System*. Addison Wesley, Reading, MA, 1996.

[21] Scott Pakin, Vijay Karamcheti, and Andrew Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Parallel and Distributed Technology*, 1997.

[22] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[23] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[24] Geoff M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, June 1998.

[25] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[26] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects Symposium V*, August 1997.

[27] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.