Cluster Computing
The original publication is available at www.springerlink.com, DOI 10.1007/s10586-009-0106-y

GMBlock: Optimizing Data Movement in a Block-level Storage Sharing System over Myrinet

Evangelos Koukis · Anastassios Nanos · Nectarios Koziris

Received: 5 August 2008 / Accepted: 17 September 2009

Abstract We present gmblock, a block-level storage sharing system over Myrinet which uses an optimized I/O path to transfer data directly between the storage medium and the network, bypassing the host CPU and main memory bus of the storage server. It is device driver independent and retains the protection and isolation features of the OS. We evaluate the performance of a prototype gmblock server and find that: (a) the proposed techniques eliminate memory and peripheral bus contention, increasing remote I/O bandwidth significantly, in the order of 20-200% compared to an RDMA-based approach, (b) the impact of remote I/O to local computation becomes negligible, (c) the performance characteristics of RAID storage combined with limited NIC resources reduce performance. We introduce synchronized send operations to improve the degree of disk to network I/O overlapping. We deploy the OCFS2 shared-disk filesystem over gmblock and show gains for various application benchmarks, provided I/O scheduling can eliminate the disk bottleneck due to concurrent access.

Keywords block-level storage, shared storage, memory contention, network block device, Myrinet, user level networking, SMP clusters, OCFS2

This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (80%) and National Resources (20%).

Evangelos Koukis · Anastassios Nanos · Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15780, Greece
E-mail: {vkoukis,ananos,nkoziris}@cslab.ece.ntua.gr

1 Introduction

Clusters built out of commodity components are becoming prevalent in the supercomputing sector as a cost-effective solution for building high-performance parallel platforms. Symmetric Multiprocessors (SMPs) of multicore chips (CMPs), are commonly used as building blocks for scalable clustered systems, when interconnected over a high-bandwidth, low-latency communications infrastructure, such as Myrinet [4], Quadrics [20] or Infiniband [1]. One of the most important architectural characteristics of SMPs is contention among processors for access to shared resources, most notably shared main memory and peripheral bus bandwidth.

Traditionally, processors are interconnected over a shared Front Side Bus to a memory controller, called the Northbridge in Intel-based designs. Thus, they all share bandwidth on the FSB and all their memory accesses are serialized on the Northbridge. Memory contention in SMP nodes is aggravated by peripheral devices, such as Network Interface Cards (NICs) and storage controllers transferring data from and to main memory using DMA. This increases memory pressure and has significant performance impact.

This paper explores the implications of contention in SMP nodes used as commodity storage servers. We study the data movement in a block-level storage sharing system over Myrinet and show how its performance suffers as the storage subsystem, the network and local processors all compete for access to main memory and peripheral bus bandwidth. To alleviate the problem, we explore techniques for efficient device-to-device data movement between the storage medium and the network.

A network block-level storage sharing system enables a number of clients to access storage devices which are directly connected to a storage server as if they were their own. Block read and write requests are encapsulated in network messages to the storage server, where they are passed to the local block device. When the block operation completes, the server returns the resulting data over the network. Throughout this paper, we refer to such systems also as “network block device” systems, or nbd systems; storage exported by a storage server appears to the client as a block device accessible over the network.

The availability of an efficient nbd system is often a prerequisite for the scalable, yet cost-effective deployment of various services in high-performance clustered environments. Such services include shared-disk parallel filesystems, shared-disk parallel databases, and shared storage pools for live VM migration in virtualized data centers. More information on the possible usage scenarios can be found in Section 2.2.

The ideal nbd system would be a very thin, very low-overhead layer that allows remote use of local storage media with performance close to that of local access, without imposing significant load on the storage server. At the same time, it would be scalable with the number of storage subsystems and network interfaces. To achieve that, its operation needs to take advantage of evolving features of cluster interconnects relating to network interface programmability and the possibility of offloading parts of protocol processing to dedicated cores and local memories close to the network.

However, most current nbd implementations are suboptimal in that regard; Often, they involve heavy host CPU-based processing, being based on TCP/IP for the transfer of block data. More importantly, they treat memory as a centralized resource and make use of data paths that cross the memory and peripheral buses multiple times, even when employing advanced interconnect features such as user level networking and remote DMA (RDMA). Thus, they impose high host overhead and their performance is limited due to bus saturation. The current situation is analyzed in greater detail in sections 2.2 and 8.

In this work, we present the design and implementation of gmblock, a block-level storage sharing architecture over DMA- and processor-enabled cluster interconnects that is built around a short-circuit data path between the storage subsystem and the network. Our prototype implementation uses Myrinet, allowing direct data movement from storage to the network without any host CPU intervention and eliminating any copies in main memory. This alleviates the effect of resource contention, increases scalability and achieves an up to two-fold increase in remote I/O bandwidth. When building shared storage pools by having nodes export local media to the network, remote I/O follows a disjoint path and does not interfere with computation on the local CPUs. The design of gmblock enhances existing OS and user level networking abstractions to construct the proposed data path, rather on relying on architecture-specific code changes. Thus, it is independent of the actual type of block device used, can support both read and write access safely, and maintains the process isolation and memory protection semantics of the OS.

Experimental evaluation of the base gmblock implementation shows that although it works around memory and peripheral bus bandwidth limitations effectively, its performance lags behind the limits imposed by raw disk and network bandwidth. We find this is due to the interaction between the performance characteristics of RAID storage and memory limitations of the Myrinet NIC. To better adapt gmblock to the inherent parallelism in request processing by RAID storage, we propose a new class of send operations over Myrinet, which support *synchronization*: their semantics allow the network transfer of block data to overlap disk I/O for a single block request. Sustained point-to-point throughput improves about 40% for streaming I/O compared to the base version of gmblock.

Finally, we present client-side optimizations for zero-copy scatter-gather I/O, and deploy the OCFS2 shared-disk parallel filesystem over gmblock, to study the performance of various workloads. Overall performance gains depend on application I/O patterns and the amount of read-write sharing over shared storage. We find that using the direct I/O path generally leads to performance improvement, provided incoming I/O requests can be scheduled efficiently, so that the disk subsystem does not become the bottleneck.

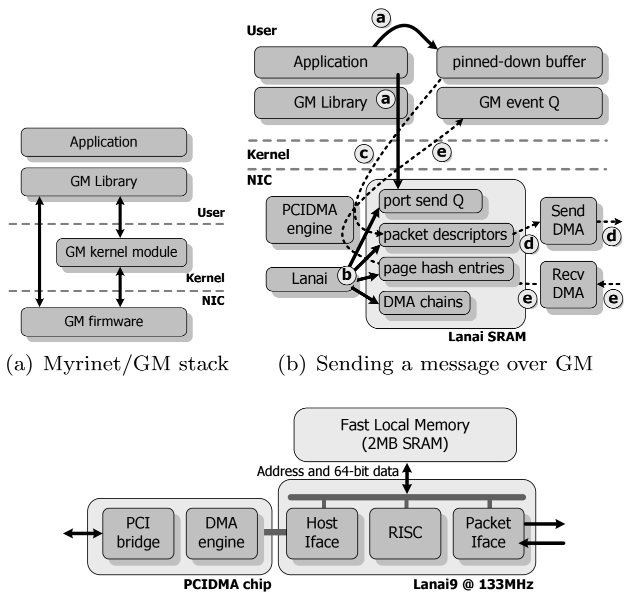
The contribution of this paper can be summarized as follows:

– We introduce gmblock, a block-level storage sharing system over Myrinet that moves data in direct I/O paths between storage devices and the network (Section 3). A prototype implementation (Section 4) shows it eliminates memory and peripheral bus contention, delivering significant improvements to remote read/write I/O bandwidth (Section 5).
– We demonstrate how bypassing main memory enables local computation to progress with negligible interference from remote I/O.
– We discover limitations in certain hardware components of the system which reduce the efficiency of peer-to-peer data transfers. We work around these limitations by employing an alternate data path us-

ing intermediate buffers on the PCI bus while still bypassing main memory.

– We propose synchronized send operations as an enhancement to the semantics of Myrinet's message-passing layer. They enable the network to adapt to the parallel, multiple-stream nature of request servicing by RAID-based storage subsystems, during peer-to-peer transfers of block data (Section 6).
– We present client-side optimizations to support end-to-end zero-copy block transfers and deploy a shared-disk parallel file system over gmblock, to test its performance with real-life application I/O patterns (Section 7).

First, however, we lay groundwork in Section 2 by presenting the essentials of user level networking and its implementation on Myrinet/GM, then discuss the importance of an efficient block-level sharing system for providing a scalable storage infrastructure to commodity clustered systems.



(a) Myrinet/GM stack (b) Sending a message over GM

(c) Main components onboard a Myrinet NIC

Fig. 1 Implementation of user level networking by Myrinet/GM

2 Background

2.1 User level networking and Myrinet/GM

This section contains a short introduction to the inner workings of Myrinet and its GM middleware, to gain insight on the communications substrate for our shared block storage system and establish the context for the proposed modifications.

Myrinet is a low-latency, high-bandwidth interconnection infrastructure for clusters and employs user level networking techniques [2] to remove the OS from the critical path of communication.

The Myrinet NICs feature a RISC microprocessor, called the Lanai, which undertakes almost all network protocol processing, a small amount (2MBs) of SRAM for use by the Lanai and three different DMA engines; one for DMA transfers between host memory and Lanai SRAM, while the other two handle data transfers between the SRAM and the network fiber link. To provide user level networking facilities to applications, the GM message-passing system is used. GM comprises the firmware executing on the Lanai, an OS kernel module and a userspace library. These three parts coordinate to allow direct access to the NIC from userspace, without the need to enter the kernel via system calls (OS bypass).

In Fig. 1(c) the main functional blocks of a Myrinet NIC are displayed. Our testbed is based the on the M3F2-PCIXE-2 version of the NIC which integrates all of the described functionality in a single Lanai2XP chip and supports two packet interfaces.

Applications map parts of Lanai SRAM (called *GM ports*), which contain send and receive queues to be manipulated directly.

GM offers reliable, connectionless point-to-point message delivery between different ports, by multiplexing message data over connections kept between every pair of hosts in the network. A "Go back N" protocol is used, trading bandwidth for reduced latency and software overhead.

The GM firmware is organized in four state machines, called SDMA, SEND, RECV and RDMA: SDMA polls for new sends or receive events in opened ports and and initiates DMA from host RAM to Lanai SRAM; SEND programs the Send DMA engine to inject packets into the network; RECV manages incoming packets and handles ACK/NACK control messages; and RDMA performs receive buffer matching, initiates DMA to host RAM and inserts receive events in the application's event queue when message reception is complete.

Fig. 1(b) shows the basic steps for a GM send operation. In all similar figures, the solid lines lines denote Programmable I/O (PIO) either by the CPU or the Lanai. The dashed lines denote DMA operations.

Sending a message `gm_send_with_callback()` entails the following: **(a)** The application computes the message in a pinned buffer and places a "send event" structure in the port send queue **(b)** SDMA performs virtual-to-physical address translation using cached pagetable entries in SRAM and initiates PCI DMA **(c)** Message data are brought into Lanai SRAM via DMA **(d)** SEND injects packets into the network **(e)** The remote side

ACKs message reception and a send completion event is posted to the application's event queue.

It is important to note that sending – conversely, receiving – a message using GM is a two-phase process:

Host to Lanai DMA: the PCIDMA engine starts, message data are copied from host RAM to Lanai SRAM

Lanai to wire DMA: the Send DMA engine fetches message data from SRAM and places them on the wire.

2.2 The need for efficient block-level storage sharing over the network

The need for shared block-level access to common storage arises often in high-performance clustered environments. Some of the most common scenarios include (a) the deployment of shared-disk parallel filesystems, (b) support of parallel databases based on a shared-disk architecture, such as Oracle RAC, and (c) virtualized environments, where disk images of virtual machines are kept in common storage, so that live migration of them among VM containers is possible.

In the first case, high performance cluster filesystems typically follow a *shared-disk* approach: all participating nodes are assumed to have equal block-level access to a shared storage pool. Distributed lock management ensures data consistency. Shared-disk filesystems include IBM's General Parallel File System GPFS [24], Oracle's OCFS2, Red Hat's Global File System (GFS) [25,21], SGI's Clustered XFS and Veritas' VxFS.

In the second case, instances of a shared-disk parallel database execute on a number of cluster nodes and need concurrent access to a shared disk pool, where table data, redo logs and other control files are kept.

Finally, in the case of virtualized environments, cluster nodes are used as VM containers. A virtual machine uses a raw storage volume as its directly connected virtual hard drive. To enable live VM migration, for reasons of load balancing and maintainability, storage must be shared among all VM containers.

Traditionally, the requirement that all nodes have access to a shared storage pool has been fulfilled by utilizing a high-end Storage Area Network (SAN), commonly based on Fibre-Channel (as in Fig. 2(a)). An SAN is a networking infrastructure providing high-speed connections between multiple nodes and a number of hard disk enclosures. The disks are treated by the nodes as Direct-attached Storage, i.e. the protocols used are similar to those employed for accessing locally attached disks, such as SCSI over FC.

However, this storage architecture requires maintaining two separate networks, one for shared storage, and a distinct one for cluster communication, e.g., for MPI. This increases the cost per node, since SAN port count must scale to include all cluster nodes, which all include appropriate interfaces, e.g., FC HBAs. Moreover, aggregate storage bandwidth remains constant, determined by the number of physical links to storage enclosures. Finally, redundant links and storage controllers needed to eliminate single points of failure increase the cost further.

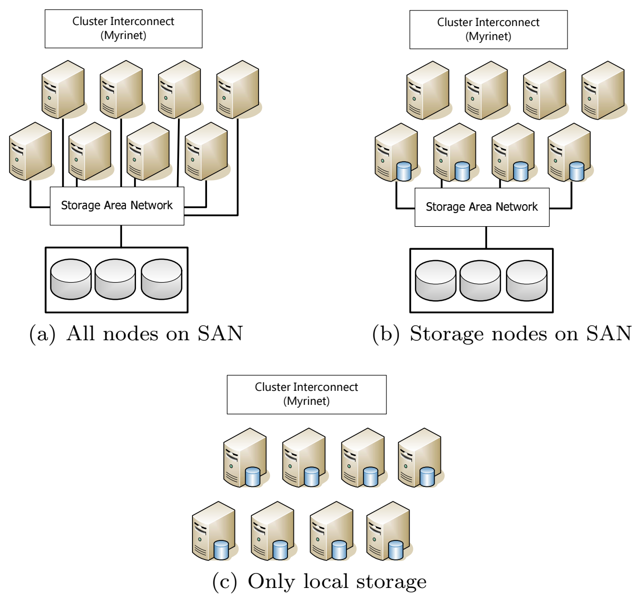
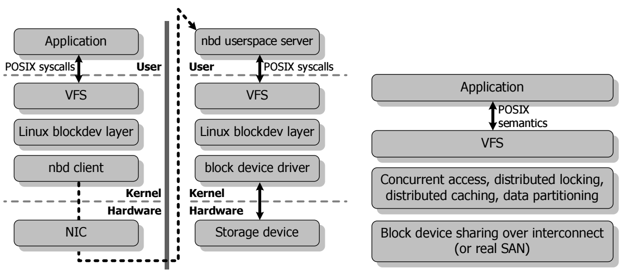

Cluster Interconnect (Myrinet)
Storage Area Network
Cluster Interconnect (Myrinet)
Storage Area Network
Cluster Interconnect (Myrinet)
(a) All nodes on SAN (b) Storage nodes on SAN (c) Only local storage

Fig. 2 Interconnection of cluster nodes and storage devices

To address these problems, a hybrid approach is commonly used, whereby only a fraction of cluster nodes is physically connected to the SAN ("storage" nodes), *exporting* the shared disks for block-level access over the cluster interconnection network, (Fig. 2(b)). Alternately, cost may be reduced further by having compute nodes contribute local resources to form a *virtual* shared storage pool (Fig. 2(c)). This model has a number of distinct advantages. First, aggregate bandwidth to storage increases with cluster node count enabling the I/O subsystem to scale with the computational capacity of the cluster. Second, the total installation cost is drastically reduced, since a dedicated SAN remains small, or is eliminated altogether, allowing resources to be diverted to acquiring more cluster nodes. However, this also means remote I/O may interfere with local computation on nodes exporting storage.

The cornerstone of this design is the *network disk sharing* layer, usually implemented in a client/server approach (Fig. 3(a)). It runs as a server on the storage nodes, receiving requests and passing them transparently to a directly-attached storage medium. It also runs as a client on cluster nodes, exposing a block device interface to the Operating System and the locally executing instance of the parallel filesystem. There are various implementations of such systems in use today: GPFS includes the NSD (Network Shared Disks) layer, which takes care of forwarding block access requests to storage nodes over TCP/IP. Traditionally, the Linux kernel has included the NBD (Network Block Device) driver [1] and Red Hat's GFS can also be deployed over an improved version called GNBD.

These implementations are TCP/IP-based. Using a complex kernel-based protocol stack increases their portability, but imposes significant protocol overhead, high CPU load and redundant data copying. Moreover, it does not take advantage of modern cluster interconnection features enabling OS bypass, zero-copy communication, such as RDMA. As explained in greater detail in the chapter on related work, there are research efforts focusing on storage sharing over RDMA. In this case, the protocol layer is simplified and copying is reduced, but data still follow an unoptimized path. For every remote I/O request, blocks are transferred from local storage to main memory, then from main memory to the NIC. These unnecessary data transfers aggravate contention on shared resources as is the shared bus to main memory and the peripheral (e.g., PCI) bus, and interfere with accesses by local CPUs.



(a) Generic nbd system (b) Parallel fs on top of nbd

Fig. 3 A parallel filesystem executing over an nbd infrastructure

```
initialize_interconnect();
fd = open_block_device();
reply = allocate_memory_buffer();
for (;;) {
      cmd = recv_cmd_from_interconnect();
      lseek(fd, cmd->start, SEEK_SET);
      switch (cmd->type) {
         case READ_BLOCK:
            read(fd, &reply->payload, cmd->len);
         case WRITE_BLOCK:
            write(fd, &req->payload, cmd->len);
      }
      insert_packet_headers(&reply, cmd);
      send_over_net(reply, reply->len);
}
```

Fig. 4 Pseudocode for an nbd server

3 Design of gmblock

This section presents the design and implementation of the gmblock block-level storage sharing system over Myrinet/GM. We begin by showing the overheads involved in standard TCP/IP and RDMA-based approaches and how gmblock's design evolves from these. Then, we discuss the necessary changes to the GM message-passing system and the Linux kernel to support a prototype implementation.

3.1 Evolution from previous nbd designs

Let's return to the generic nbd client/server implementation as portrayed in Fig. 3(a). The nbd client resides in the OS kernel and exposes a block device interface to the rest of the kernel, so that it may appear as an ordinary, directly-attached storage device. The requests being received from the kernel block device layer are encapsulated in network messages and passed to a remote server. This server is commonly not run in privileged kernelspace. Instead, it executes as a userspace process, using standard I/O calls to exchange data with the actual block device being shared.

The pseudocode for a generic nbd server can be seen in Fig. 4. For simplicity, we will refer to a remote block read operation but the following discussion applies to write operations as well, if the steps involving disk I/O and network I/O are reversed. There are four basic steps involved in servicing a read block request: `(a)` The server receives the request over the interconnect, unpacks it and determines its type – let's assume it's a read request `(b)` A system call such as `lseek()` is used to locate the relevant block(s) on the storage medium `(c)` The data are transferred from the disk to a userspace buffer `(d)` The data are transmitted to the node that requested them.

The overhead involved in these operations depends significantly on the type of interconnect and the semantics of its API. To better understand the path followed by the data at the server side, we can see the behavior of a TCP/IP-based server at the logical layer, as presented in Fig. 5(a). Again, solid lines denote PIO operations, dashed lines denote DMA operations. **(a)** As soon as a new request is received, e.g. in the form of Ethernet frames, it is usually DMAed to kernel memory, by the NIC **(b)** Depending on the quality of the TCP/IP implementation it may or may not be copied to other

[1] nbd in all small letters will be used to denote generic client/server implementations for network sharing of block devices. NBD in all capital letters denotes the TCP/IP implementation in the Linux kernel.

buffers, until it is copied from the kernel to a buffer in userspace. The server process, which presumably blocks in a `read()` system call on a TCP/IP socket, is then woken up to process the request. It processes the request by issuing an appropriate `read()` call to a file descriptor acquired by having `open()`ed a block device. In the generic case this is a *cached* read request; the process enters the kernel, which **(c)** uses the block device driver to setup a DMA transfer of block data from the disk(s) to the *page cache* kept in kernel memory **(d)** Then, the data need to be copied to the userspace buffer and the `read()` call returns **(e)** Finally, the server process issues a `write()` call, which copies the data back from the userspace buffer into kernel memory. Then, again depending on the quality of the TCP/IP implementation, a number of copies may be needed to split the data in frames of appropriate size, which are **(f)** DMAed by the NIC and transferred to the remote host.

In Fig. 5(b), we can see the actual path followed by data, at the physical level. The labels correspond one-to-one with those used in the previous description: **(a, b)** Initially, the read request is DMAed by the NIC to host RAM, then copied to the userspace buffer using PIO **(c)** The disk is programmed to DMA the needed data to host RAM. The data cross the peripheral bus and the memory bus **(d)** The data are copied from the page cache to the userspace buffer by the CPU **(e)** The data are copied back from the userspace buffer to the kernel, to be sent over the network **(f)** The data cross the memory bus and the peripheral bus once again, to be sent over the network via DMA.

This data path involves a lot of redundant data movement. The number of copies needed to move data from the disk to the TCP/IP socket can be reduced by allowing the kernel more insight into the semantics of the data transfer; one could map the block device onto userspace memory, using `mmap()`, so that a `write()` to the socket copies data directly from the page cache to the network frame buffers, inside the kernel. However, depending on the size of the process address space, not all of the block device may be mappable. Thus, the overhead of remapping different parts of the block device must be taken into account.

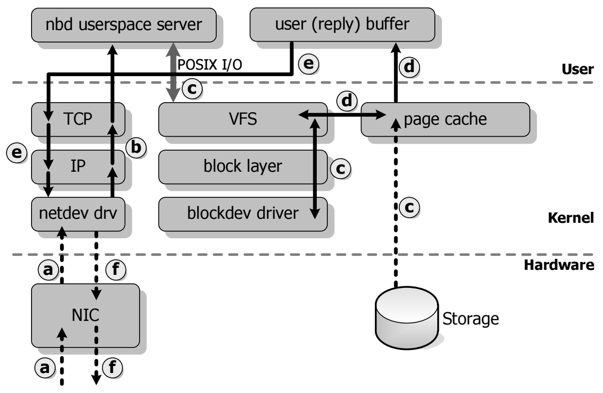
A different way to eliminate one memory copy is by bypassing the page cache altogether. This can be accomplished by use of the POSIX `O_DIRECT` facility, which ensures that all I/O with a file descriptor bypasses the page cache and that data are copied directly into userspace buffers. The Linux kernel supports `O_DIRECT` transfers of data; the block layer provides a generic `O_DIRECT` implementation which takes care of pinning down the relevant userspace buffers, determining the physical addresses of the pages involved and finally enqueuing block I/O requests to the block device driver which refer to these pages directly instead of the page cache. Thus, if the block device is DMA-capable, the data can be brought into the buffers directly, eliminating one memory copy. Still, they have to be copied back into the kernel when the TCP/IP `write()` call is issued.

The main drawback of this data path is the large amount of redundant data copying involved. If only one kernel copy takes place, data cross the peripheral bus twice and the memory bus four times. When forming virtual storage pools by having compute nodes export storage to the network, remote I/O means fewer CPU cycles and less memory bandwidth are available to the locally executing workload (path `(g)`).

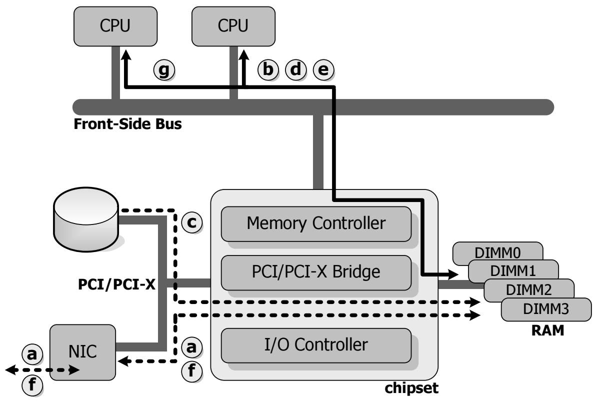
The problem is alleviated, if a user level networking approach is used. When a cluster interconnect such as Myrinet is available, the nbd server can bypass the OS kernel for network I/O, using GM instead of TCP/IP. In this case, some of the redundant copying is eliminated, since the steps to service a request are (Fig. 6(a)): `(a)` A request is received by the Myrinet NIC and copied directly into a pinned-down request buffer `(b)` The server uses `O_DIRECT`-based I/O so that the storage device is programmed to place block data into userspace buffers via DMA `(c)` The response is pushed to the remote node using `gm_send()`, as in Section 2.1. In this approach, most of the PIO-based data movement is eliminated. The CPU is no longer involved in network processing, the complex TCP/IP stack is removed from the critical path and almost all CPU time is devoted to running the computational workload. However, even when using GM for message passing, main memory is still on the critical path. At the physical layer (Fig. 6(b)), for a read operation, block data are transferred from the storage devices to in-RAM buffers, then from them to the Myrinet NIC. Thus, they traverse the peripheral bus and the main memory bus twice; pressure on the peripheral and main memory buses remains, and remote I/O still interferes with local computation (`(d)`).

3.2 An alternative data path with memory bypass

To solve the problem of redundant data movement at the server side of an nbd system, we propose a shorter data path, which does not involve main memory at all. To service a remote I/O request, all that is really needed is to transfer data from secondary storage to the network or vice-versa. Data flow based on this alternative data path is presented in Fig. 7(b): **(a)** A read request is received by the Myrinet NIC **(b)** The nbd server process services the request by arranging for block data to be transferred directly from the storage

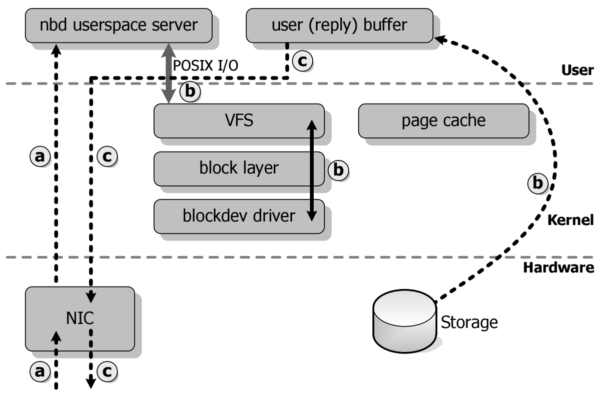


(a) Data path at the logical level

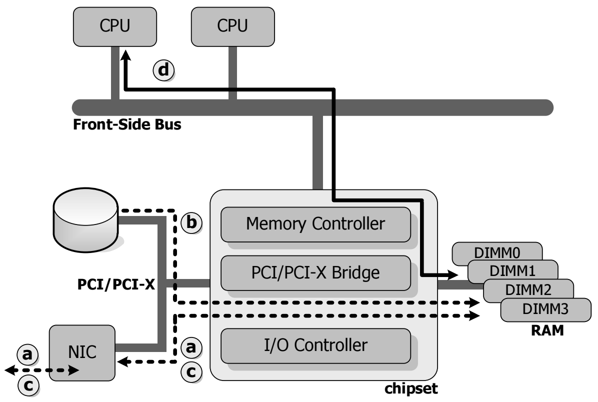


(b) Data path at the physical level

Fig. 5 TCP/IP based nbd server

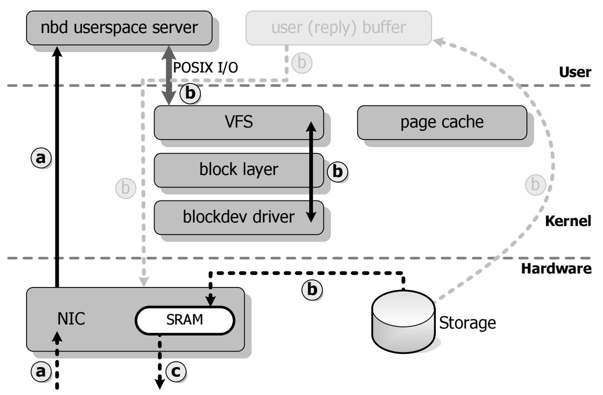


(a) Data path at the logical level

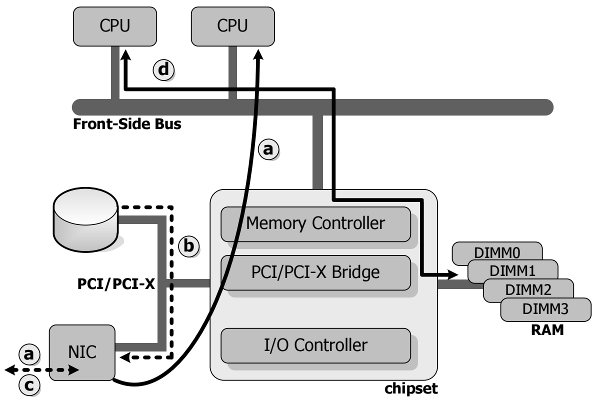


(b) Data path at the physical level

Fig. 6 GM-based nbd server



(a) Data path at the logical level



(b) Data path at the physical level

Fig. 7 Proposed gmblock server

device to the Myrinet NIC **(c)** The data is transmitted to the node that initiated the operation.

Implementing this path would solve most of the problems described above:

– The critical path is the shortest possible. Data go directly from disk to NIC or vice-versa
– The full capacity of the peripheral bus can be used, since data only traverse it once

– There is no staging in buffers kept in RAM, thus no memory bandwidth is consumed by I/O and code executing on local CPUs does not incur the overhead of memory contention

Most importantly, this design would acknowledge the fact the the remote I/O path may be disjoint from main memory. The inclusion of RAM buffers in all previous data paths is a necessity arising from the programming semantics of the mechanisms used to enable the transfer – GM and Linux kernel drivers – rather than from the intrinsic properties of remote I/O operations; GM programs the DMA engines on the Myrinet NIC to exchange data between the Lanai SRAM and RAM buffers, while the kernel programs storage devices to move data from/to page cache or userspace buffers kept in main memory. Thus, to support the proposed data path, we need to extend these mechanisms so that direct disk-to-NIC transfers are supported. At the same time, the architecture-dependent details of setting up such transfers must be hidden behind existing programming abstractions, i.e. GM user level networking primitives and the Linux I/O system call interface. In this approach, only minimal changes to the the nbd server source code will be required to support the enhanced functionality.

Let's assume a GM-based nbd server servicing a read request, similar to that of Fig. 4. In the case of GM, the server would have used `gm_open()` to initialize the interconnect, and `gm_dma_malloc()` to allocate space for the message buffer. Variable `reply` contains the virtual address of this buffer, dedicated to holding the reply of a remote read operation, before it is transferred over Myrinet/GM. If this memory space was not allocated in RAM, but could be made to reside in Lanai SRAM instead, then the `read()` system call could be used un-altered, to express the desired semantics; It would still mean "I need certain blocks to be copied to memory pointed to by `reply`", this time however referring to a buffer in SRAM, mapped onto the process's VM space at location `reply`.

However, if standard, buffered I/O was used, using this call would first bring the data into the kernel's page cache, then a CPU-based `memcpy()` would be used to copy the data from the cached page to the mapped SRAM buffer. This would still invoke PIO; the whole of Lanai SRAM is exposed as a large memory-mapped I/O resource on the PCI physical address space. Thus, every reference by the CPU to the virtual address space pointed to by `reply` during the `memcpy()` operation, would lead to I/O transactions over the peripheral bus. The situation would be radically different, if POSIX `O_DIRECT` access to the open file descriptor for the block device was used instead. In this case, the kernel would bypass the page cache. Its direct I/O subsystem would translate the virtual address of the buffer to a physical address in the Myrinet NIC's memory-mapped I/O space and use *this* address to submit a block I/O request to the appropriate in-kernel driver. In the case of a DMA-capable storage device, the ensuing DMA transaction would have the DMA engine copying data directly to the Myrinet NIC, bypassing the CPU and main memory altogether. To finish servicing the request, the second half of a GM Send operation is needed: the Host-to-Lanai DMA phase is omitted and a Lanai-to-wire DMA operation is performed to send the data off the SRAM buffer.

Conversely, in the case of a remote write operation, the DMA-capable storage device would be programmed to retrieve incoming data directly from the Lanai SRAM buffer after a wire-to-Lanai DMA operation completes.

It is important to note that almost no source code changes are needed in the nbd server to support this enhanced data path. The server process still issues `read()` or `write()` and `gm_send()` calls, unaware of the underlying transfer mechanism. The desired semantics emerge from the way the Linux block driver layer, the kernel's VM subsystem and GM's user level networking capabilities are combined to construct the data path.

3.3 Discussion

An analysis of the proposed data path at the logical layer can be seen in Fig. 7(a). There are almost no gmblock-specific changes, compared to a GM-based nbd implementation. To achieve this, we re-use existing programming abstractions, as provided by GM and the Linux kernel. By building on `O_DIRECT` based access to storage, our approach is essentially disk-type agnostic. Since the CPU is involved implicitly in the setup phase of the transfer, the server is not limited to sharing raw block device blocks. Instead, it could share block data in *structured* form, e.g. from a file in a standard `ext2` filesystem. Or, it could be used over a software RAID infrastructure, combining multiple disks in a RAID0 configuration.

Another point to take into account is ensuring coherence with the kernel's page cache. Since blocks move directly from NIC to storage and vice versa, a way is needed to ensure that local processes do not perform I/O on stale data that are in the kernel's page cache and have not been invalidated. We avoid this problem by keeping the kernel in the processing loop, but not in the data path. Its direct I/O implementation will take care of invalidating affected blocks in the page cache, if an O_DIRECT write takes place, and will flush dirty buffers to disk before an O_DIRECT read.

Using the proposed path means that server CPU and RAM are no longer in the processing path. Although this eliminates architectural bottlenecks, it means no server-side prefetching and caching is possible. Moreover, overall performance depends on the interaction of gmblock with the overlying filesystem and application, and the amount of read-write sharing that takes place through shared storage. We discuss the importance of these three factors, prefetching, caching and read-write sharing below.

Server-side buffers on storage servers may play an important role in prefetching data from the storage medium for efficiency. Data prefetching is still possible in gmblock, but has to be initiated by the client side. Experimental evaluation on an OCFS2 over gmblock setup (Section 7) shows that prefetching is indeed necessary, to support good performance for small application reads.

Regarding caching, although server-side caching is no longer possible, client-side caching still takes place: Client systems treat gmblock-provided storage as a local hard disk, caching reads and keeping dirty buffers on writes. Data need to be written back only to ensure correctness for read-after-write sharing. Moreover, the aggregate size of memory in clients can be expected to be much larger than storage server memory.

Server-side caching may also prove to be beneficial in coalescing small reads and writes, forming more efficient requests to storage. This is highly dependent on the overlying application's access patterns. Whether using the proposed data path leads to performance increase depends on the balance of memory-to-memory copy throughput, imposed CPU load and memory-to-storage transfers, relative to direct storage-to-network throughput.

Finally, server-side caching may prove important for applications with very heavy read-write sharing through the filesystem. If there is no special provision for direct client-to-client synchronization of dirty data, either by the application itself or by the filesystem, then the storage server may be used essentially as a buffer for client-to-client block transfers with many small writes followed by many small related reads. For such workloads, using gmblock's short-circuit data path is not an appropriate choice. If write coalescing is needed, then the framework can be set to cache writes, while still allowing reads to happen over the direct data path, or it can be run in fully cached mode.

On the other hand, the impact of heavy read-write sharing in application performance is a well-known problem and there are numerous efforts in the literature to attack it at the filesystem and application level. Oracle 9i uses a distributed caching mechanism called "Cache Fusion" [14] to support read and write-sharing with direct instance-to-instance transfers of dirty data over the interconnect, instead of going through shared storage. Disk I/O happens only when the needed blocks are not present in *any* of the client-side caches. Similarly, in the HPC context, heavy read-write block contention arises in MPI applications with multiple peers working on the same block set. To attack the problem, GPFS features a special data shipping mode [22], which the free ROMIO implementation of MPI-IO has also been extended to support [3]. Data shipping mode minimizes read-write block contention by assigning distinct parts of a shared file to distinct processes, so that only a single process issues read/write requests for a specific block during collective I/O.

To summarize, the applicability of gmblock's direct I/O path depends significantly on the access patterns of the I/O workload. Workloads for which it is a good fit are those with little data sharing, e.g., shared storage for live VM migration and read-write server workloads on independent data sets, or workloads where nodes can be assumed to coordinate access at a higher level, e.g., the Oracle RDMBS or MPI-IO applications with data shipping optimizations.

4 Implementation details

The implementation of gmblock's optimized data path involves changing two different subsystems: First, GM must be extended to support buffers in Lanai SRAM. Second, the Linux VM mechanism must include support for treating Lanai SRAM as host RAM, so that direct I/O from and to PCI memory-mapped I/O regions is possible.

4.1 GM support for buffers in Lanai SRAM

The GM middleware needs to be enhanced, as to allow the allocation, mapping and manipulation of buffers residing in Lanai SRAM by userspace applications. At the same time, it is important to preserve GM semantics and UNIX security semantics regarding process isolation and memory protection, as is done for message passing from and to userspace buffers in host RAM.

The described changes were tested on various combinations of GM-2.0 and GM-2.1 on an Intel i386 and an Intel EM64T system. However, the changes affect the platform-independent part of GM, so they should be usable on every architecture/OS combination that GM has been ported to.

This is the functionality that needs to be supported, along with the parts of GM that are affected:

- Allocation of buffers in Lanai SRAM (GM firmware)
- Mapping of buffers onto the VM of a process in userspace (GM library, GM kernel module)
- Sending and receiving messages from/to Lanai SRAM using `gm_send()` and `gm_provide_receive_buffer()` (GM library, GM firmware)

For the first part, the firmware initialization procedure was modified, so that a large, page-aligned buffer is allocated for gmblock's use, off the memory used for dynamic allocation by the firmware. Our testbed uses Myrinet NICs with 2MB of SRAM, out of which we were able to allocate at most 700KB for gmblock's use and still have the firmware fit in the available space and execute correctly.

The second part involved changes in the GM library, which tries to map the shared memory buffer. The GM kernel module verifies that the request does not compromise system security, then performs the needed mapping. The Lanai SRAM buffer is shared among processes, but different policies may be easily implemented, by changing the relevant code in the GM kernel module.

Finally, to complete the integration of the SRAM buffer in the VM infrastructure and allow it to be used transparently for GM messaging, we enhance the GM library so that the requirement for all message exchange to be done from/to in-RAM buffers is removed. At the userspace side the library detects that a GM operation to send a message or to provide a receive buffer refers to Lanai SRAM, and marks it appropriately in the event passed to the Lanai. There, depending on the type of the request:

- For a send request, the SDMA state machine omits the Host-to-Lanai DMA operation, constructs the needed Myrinet packets and passes them directly to SEND, without any intermediate copies.
- Things are more complicated when incoming data need to be placed in an SRAM buffer by the RDMA state machine, since incoming packet data are placed in predefined message buffers by the hardware before any buffer matching. In the common case of receiving into RAM, they are moved to their final destination during the Lanai-to-Host DMA phase. When receiving into SRAM buffers this is replaced by a copy operation, undertaken by a copy engine on the Lanai. The memory arbitration scheme of the LanaiX ensures that the copy progresses without impacting the rate at which concurrent, pipelined packet receives occur.

4.2 Linux VM support for direct I/O with PCI ranges

To implement gmblock's enhanced data path, we need to extend the Linux VM mechanism so that PCI memory-mapped I/O regions can take part in direct I/O operations. So far, the GM buffer in Lanai SRAM has been mapped to a process's virtual address space and is accessible using PIO. This mapping translates to physical addresses belonging to the Myrinet NIC's PCI memory-mapped I/O (MMIO) region. The MMIO range lies just below the 4GB mark of the physical address space in the case of the Intel i386 and AMD x86-64 platforms.

To allow the kernel to use the relevant physical address space as main memory transparently, we extend the architecture-specific part of the kernel related to memory initialization so that the kernel builds page management structures (*pageframes*) for the full 4GB physical address range and not just for the amount of available RAM. The relevant `struct page` structures are incorporated in a Linux *memory zone*, called `ZONE_PCIMEM` and are marked as reserved, so that they are never considered for allocation to processes by the kernel's memory allocator.

With these modifications in place, PCI MMIO ranges are manageable by the Linux VM as host RAM. All complexity is hidden behind the page frame abstraction, in the architecture-dependent parts of the kernel; even the direct I/O layer does not need to know about the special nature of these pages.

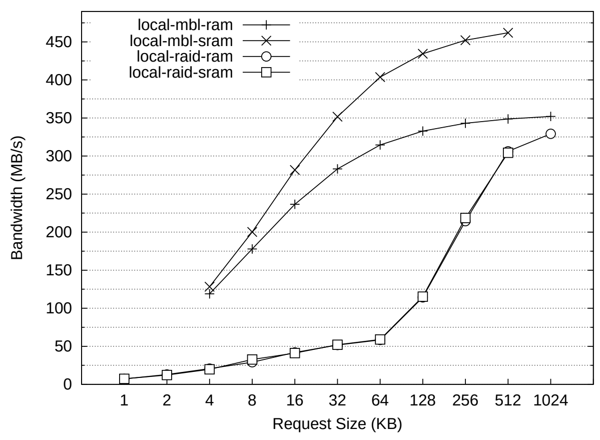
5 Experimental evaluation

To quantify the performance benefits of employing gmblock's short-circuit data path we compare three different nbd systems in a client-server block-level storage sharing configuration. The first one is a prototype implementation of gmblock with message buffers on Lanai SRAM (hereafter `gmblock-sram`) so that direct disk-to-NIC transfers are possible. The second is a standard TCP/IP-based system, Red Hat's GNBD, the reworked version of NBD that accompanies GFS (`tcpip-gnbd`). GNBD runs over the same Myrinet, with Ethernet emulation. The third one is gmblock itself, running over GM without the proposed optimization. Its performance is representative of RDMA-based implementations using a data path which crosses main memory (`gmblock-ram`).

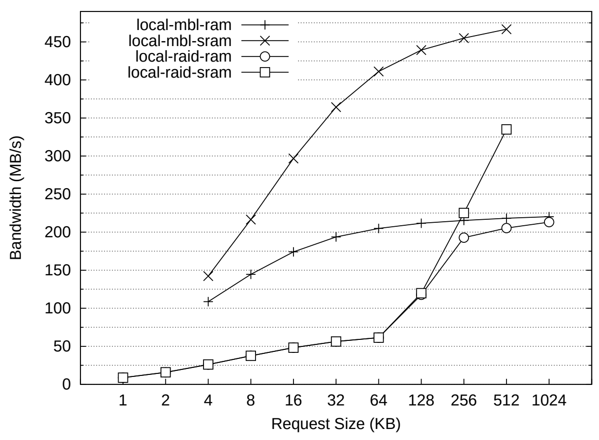
The evaluation concerns three metrics: *(a)* the sustained bandwidth for remote read operations *(b)* the sustained bandwidth for remote write operations and *(c)* the server-side impact on local executing computational workloads. At each point in the evaluation we

	Server A	**Server B**
Processor	2x Pentium III@1266MHz	Pentium 4@3GHz
M/B	Supermicro P3TDE6	Intel SE7210TP1-E
Chipset	Serverworks ServerSet III HE-SL, CIOB20 PCI bridge	Intel E7210 chipset, 6300ESB I/O controller hub
I/O Bus	2-slot 64bit/66MHz PCI	3-slot 64bit/66MHz PCI-X
RAM	2x PC133 512MB SDRAM	2 x PC2700 512MB SDRAM DDR
Disks	8x Western Digital WD2500JS 250GB SATA II	
I/O controller	3Ware 9500S-8 SATA RAID and MBL	
NIC	Myrinet M3F2-PCIXE-2	

Table 1 Technical specifications of storage servers used in experimental testbed



(a) RAID and MBL, Server A



(b) RAID and MBL, Server B

Fig. 8 Sustained local bandwidth for both storage media on Servers A and B

identify the performance-limiting factor and try to mitigate its effect, in order to observe how the different architectural limitations come into play.

We experiment with storage servers of two different configurations: Server A is an SMP system of two Pentium III processors, with a 2-slot PCI bus, while Server B is a Pentium 4 system, with more capable DDR memory and a 3-slot PCI-X bus. The exact specifications can be found in table 1.

The storage medium to be shared over Myrinet is provided by a 3Ware 9500S-8 SATA RAID controller, which has 8 SATA ports on a 64bit/66MHz PCI adapter. We built a hardware RAID0 array out of 8 disks, which distributes data evenly among the disks with a chunk size of 64KB and is exported as a single drive to the host OS. We use two nodes, one of configuration A functioning as the client, the either as the server (of either configuration A or B). The nodes are connected back-to-back with two Myrinet M3F2-PCIXE-2 NICs. The NICs use the Lanai2XP@333MHz processor, with 2MB of SRAM and feature two 2+2Gbit/s full-duplex fiber links. Linux kernel 2.6.22, GM-2.1.26 and 3Ware driver version 2.26.02.008 are used. The I/O scheduler used is the anticipatory scheduler (AS).

We also experiment with a custom solid-state storage device built around another Myrinet M3F2-PCIXE-2 NIC, which is able to deliver much better throughput even for very small request sizes. We have written a Linux block device driver and custom firmware for the card, which enables it to be used as a standard block device. Block read and write requests are forwarded by the host driver (the Myrinet BLock driver, or “MBL”) to the firmware, which programs the DMA engines on the NIC to transfer block data from and to Lanai SRAM. Essentially, we use the card to simulate a very fast, albeit small, storage device which can move data close to the rate of the PCI-X bus.

Experiment 1a: Local disk performance

We start by measuring the read bandwidth delivered by the RAID controller, locally, performing back-to-back direct I/O requests of fixed size in the range of 1, 2, . . ., 512KB, 1024KB. The destination buffers reside either in RAM (`local-raid-ram`) or in Lanai SRAM (`local-raid-sram`, short-circuit path) We repeat this experiment for the MBL device as well (`local-mbl-ram`, `local-mbl-sram`). In the following, 1MB = 2^{20} bytes. With the adapters installed on the 64bit/66MHz PCI-X bus of Server B, we get the bandwidth vs. request size curves of Fig 8(b).

A number of interesting conclusions can be drawn. First, for a given request size these curves provide an upper bound for the performance of our system. We see that the RAID throughput increases significantly for request sizes after 128KB-256KB (reaching a rate of $r_{disk\rightarrow sram}$ =335MB/s, with 512KB request size for buffers in SRAM) while performance is suboptimal for smaller sizes: the degree of parallelism achieved with RAID0 is lower (fewer spindles fetch data into memory) and execution is dominated by overheads in the kernel's I/O subsystem. On the other hand, MBL delivers good performance even for small request sizes and comes close to the theoretical 528MB/s limit imposed by the PCI-X bus itself.

We note that throughput for transfers to RAM levels off early, at ∼217MB/s. We found this is due to an architectural constraint of the Intel motherboard used on Server B: The 6300ESB I/O hub supporting the PCI-X bus is connected to the 827210 Memory Controller through a "hub link" interface which is limited to 266MB/s. Server A's PCI bridge does not exhibit a similar issue (Fig. 8(a)).

Experiment 1b: Remote read performance

We then proceed to measure the sustained remote read bandwidth for all three implementations. A user-space client runs on a machine of configuration A generating back-to-back requests of variable size in two setups, one using Server A and one with Server B. To achieve good utilization of Myrinet's 2+2Gbit/s links it is important to pipeline requests correctly. We tested with one, two and four outstanding requests and the corresponding configurations are labeled `gmblock-ram-{1,2,4}` and `gmblock-sram-{1,2,4}` for the GM-based and short-circuit path case respectively. To keep the figures cleaner we omit the curves for `gmblock-{ram,sram}-2`. In general, the performance of `gmblock-{ram,sram}-2` was between `gmblock{ram,sram}-1` and `gmblock{ram,sram}-4`, as expected.

We are interested in bottlenecks on the CPU, the memory bus, the RAID controller and the PCI/PCI-X bus. In general, GNBD performs poorly and cannot exceed 68MB/s on our platform; TCP/IP processing for GNBD consumes a large fraction of CPU power *and* a large percentage of memory bandwidth for intermediate data copying. A representative measurement is included in Fig. 9(a).

Performance for `gmblock-ram-1` and `gmblock-sram-1` is dominated by latency, since only one block read request is in flight at all times. Resource utilization is suboptimal, since the network interface is idle when the storage medium retrieves block data and vice-versa, hence the sustained throughput is low. The results are consistent with block data being transferred from the disk to buffers in RAM or SRAM (at a rate of $r_{disk\rightarrow ram}$ or $r_{disk\rightarrow sram}$ respectively), then from RAM or SRAM to the Myrinet fabric (at a rate of $r_{ram\rightarrow net}$, $r_{sram\rightarrow net}$ respectively). For example, in the case of `gmblock-sram` and the 3Ware controller on Server B (Fig 10(a)), $r_{disk\rightarrow sram}$ =335MB/s, $r_{sram\rightarrow net}$ =462MB/s and the expected remote read rate is

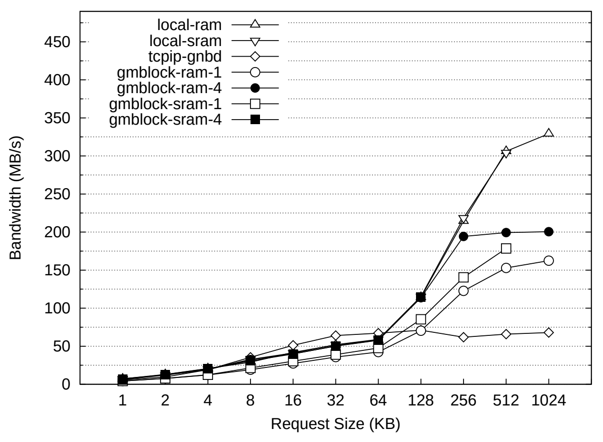
$$r_{gmblock-sram} = 1/\left(\frac{1}{r_{disk\rightarrow sram}} + \frac{1}{r_{sram\rightarrow net}}\right)$$

which is 194MB/s, very close to the observed value of 186MB/s.

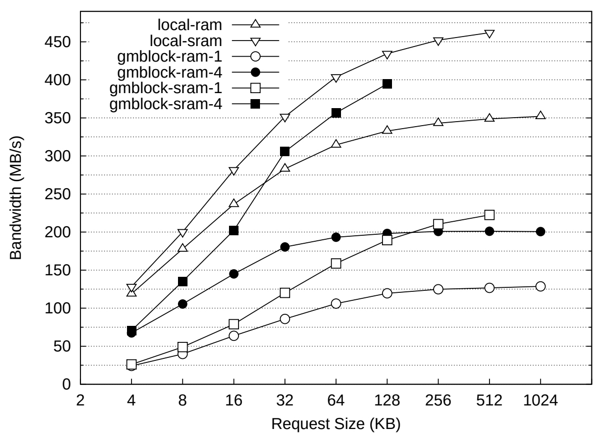
When two or four requests are outstanding, the bottleneck shifts, with limited PCI-to-memory bandwidth determining the overall performance of `gmblock-ram`. In the case of Server B (Fig. 10(b)), `gmblock-ram` has to cross the hub link (266MB/s theoretical) to main memory twice, so it is capped to half the value of $r_{disk\rightarrow ram}$ = 217MB/s. Indeed, for request sizes over 128KB, it can no longer follow the local storage bandwidth curve and levels off at ∼100MB/s. This effect happens later for Server A's PCI host bridge, since its PCI to memory bandwidth is ∼398MB/s. Indeed, with request sizes over 128KB for MBL, `gmblock-ram` is capped at 198MB/s. On the other hand, `gmblock-sram-1` has no such limitation. In the case of MBL on Server A, `gmblock-sram-{2,4}` deliver more than 90% of the locally available bandwidth to the remote node for 256KB and 128KB-sized requests respectively, a two-fold improvement over `gmblock-ram`.

When used with a real-life RAID storage subsystem, `gmblock-sram-{2,4}` utilizes the full RAID bandwidth for any given request size up to 256KB: for Server B that is 220MB/s, a more than two-fold improvement. For Server A that is 214MB/s, 20% better than `gmblock-ram`. Since `gmblock-ram` has already saturated the PCI-to-memory link, the improvement would be even more visible for `gmblock-sram-2` if larger request sizes could be used, since the RAID bandwidth cap would be higher for 512KB requests: a 512KB request equals the stripe size of the RAID array and is processed by all spindles in parallel. However, the maximum number of outstanding requests is limited by the amount of SRAM that's available for gmblock's use, which is no more than 700KB on our platform, thus it only suffices for a single 512KB request. The small amount of SRAM on the NIC limits the maximum RAID bandwidth made available to `gmblock-sram`. We propose synchronized GM operations to work around this limitation in section 6.

In the case of the higher-performing MBL, we note that a higher number of outstanding requests is needed to deliver optimal bandwidth; moreover, a significant

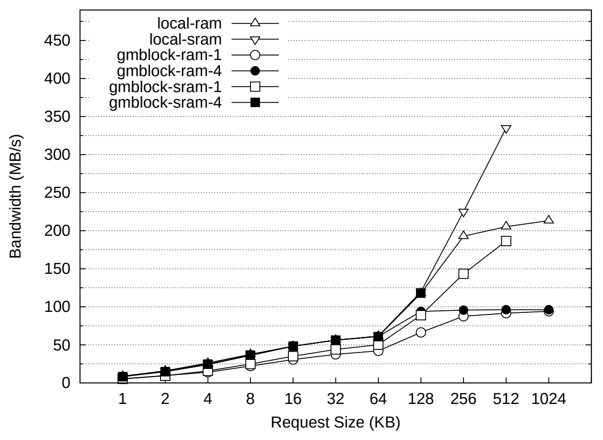


(a) RAID bandwidth

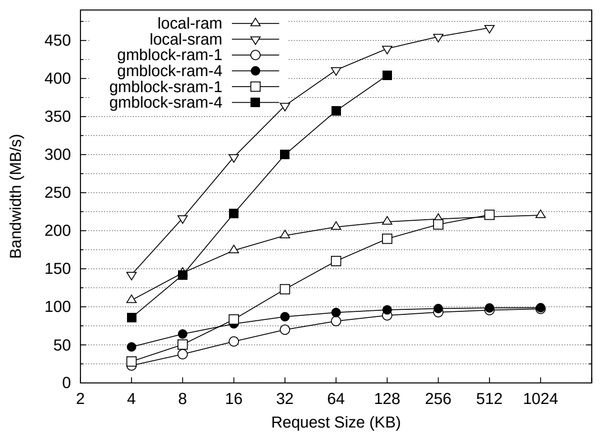


(b) MBL bandwidth

Fig. 9 Sustained remote read bandwidth, Server A



(a) RAID bandwidth



(b) MBL bandwidth

Fig. 10 Sustained remote read bandwidth, Server B

percentage of the maximum bandwidth is achieved even for small, 32-64KB request sizes.

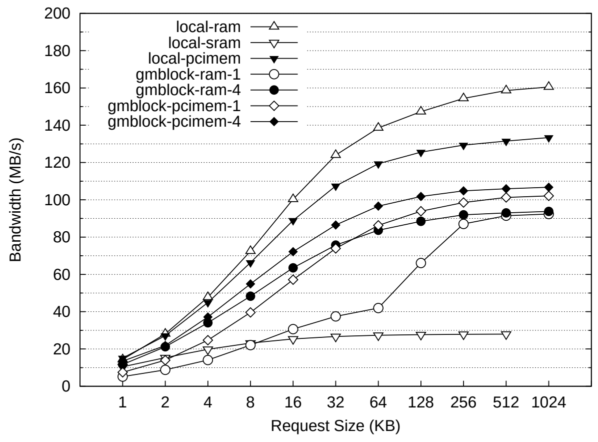
It is also interesting to see the effect of remote I/O on the locally executing processes on the server, due to interference on the shared memory bus. Although `gmblock-ram` removes the CPU from the critical path, it still consumes two times the I/O bandwidth on the memory bus. If the storage node is also used as a compute node, memory contention leads to significant execution slowdowns.

Experiment 1c: Effect on local computation

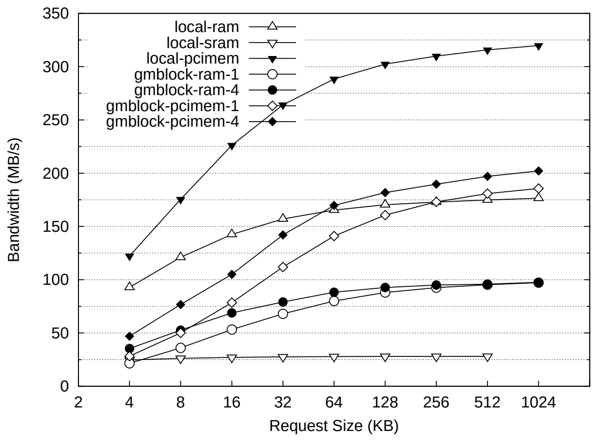
For Experiment 1c, we run `tcpip-gnbd`, `gmblock-ram-2` and `gmblock-sram-2` along with a compute intensive benchmark, on only one of the CPUs of Server A. The benchmark is a process of the `bzip2` compression utility, which performs indexed array accesses on a large working space (∼8MB, much larger than the L2 cache) and is thus sensitive to changes in the available memory bandwidth, as we have shown in previous work [13]. There is no processor sharing involved; the nbd server can always run on the second, otherwise idle, CPU of the system. In Fig. 5 we show the normalized execution time of `bzip2` for the three systems. In the worst case, `bzip2` slows down by as much as 67%, when `gmblock-ram-2` is used with 512KB requests. On the other hand, the benchmark runs with negligible interference when `gmblock-sram` is used, since the memory bus is bypassed completely and its execution time remains almost constant.

Experiment 1d: Write Performance

We also evaluate performance in terms of sustained remote write bandwidth, similar to the case of remote reads. The results for Server B are displayed in Fig. 12. Note that the maximum attained write performance of the RAID controller is much worse than the read case,



(a) RAID bandwidth



(b) MBL bandwidth

Fig. 12 Sustained remote write bandwidth, Server B

however its bandwidth vs. request size curve rises sooner due to the use of on-board RAID write buffers.

Again `gmblock-ram` is capped at ∼100MB/s due to crossing the main memory bus. However, the performance of `gmblock-sram` is *much* lower than expected based on the read results. We discovered and later confirmed with Myricom this is due to a hardware limitation of the LanaiX processor, which cannot support being the target of PCI read transactions efficiently and delivers only ∼25MB/s PCI read bandwidth.

To work around this limitation, we introduce a third configuration; we can construct a data path which still bypasses main memory but uses intermediate buffers on the peripheral bus. As buffer space we decided to use memory on an Intel XScale-based PCI-X adapter, the Cyclone 740 [6], placed in the 3rd slot of Server B's 3-slot PCI-X bus. It features an Intel XScale 80331 I/O processor and 1GB of DDR memory, on an internal PCI-X bus. A PCI-X to PCI-X bridge along with an Address Translation Unit allows exporting parts of this memory to the host PCI physical address space. By placing the message buffers of gmblock on the card, we can have the Lanai DMAing data *into* this memory, then have the storage controller read data off it, which is more efficient than reading data off Lanai SRAM directly. This path crosses the PCI bus twice, hence is limited to half its maximum bandwidth. Moreover, even after careful tuning of the Intel XScale's PCI-X bridge to support efficient prefetching from the internal bus in order to serve incoming bus read requests, the 3Ware RAID controller was only able to fetch data at a rate of ∼133MB/s compared to ∼160MB/s from main memory (the `local-{ram, pcimem}` curves). Thus, the performance of the NIC $\longrightarrow$ PCI buffers $\longrightarrow$ storage path (`gmblock-pcimem`) is comparable to that of `gmblock-ram`, however it has the advantage of bypassing main memory, so it does not interfere with memory accesses by the host processor.

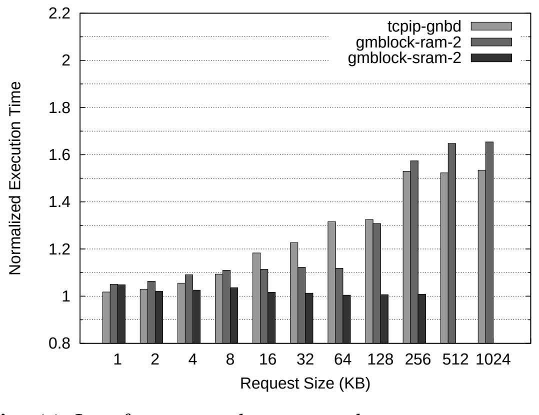


Fig. 11 Interference on the memory bus

6 Synchronized GM send operations

6.1 Motivation

Overall, gmblock delivers significant bandwidth improvements for remote read / write operations compared to conventional data paths crossing main memory. However, its performance still lags behind the limits imposed by network and local storage bandwidth; in the case of MBL on Server B, `gmblock-sram` achieves ∼400MB/s (for 128KB-sized requests in the `gmblock-sram-4` case, which is 86% of the maximum available read bandwidth available locally, while it achieves ∼220MB/s (65% of the maximum) when using a real-life RAID-based storage system. The chief reason for low efficiency lies in the interplay between a number of conflicting factors and architectural constraints: (a) The nbd system needs an outstanding request queue of sufficient depth in order to pipeline requests efficiently. (b) RAID-based storage systems deliver best performance when provided with sufficiently large requests (Fig. 8(a)). (c) Cluster interconnect NICs, such as the Myrinet NICs on our platform, feature a limited amount of memory, usually only meant to buffer message packets before injection into the network.

The proposed approach moves data directly from disk to NIC, thus the maximum number of outstanding requests is limited by the amount of SRAM available for gmblock's use. In our case, using standard Myrinet NICs with 2MBs of SRAM, we were able to reserve ~700KB for gmblock buffers. It is impossible to allocate more in productions environments, because we already had to disable a number of GM features, e.g., Ethernet emulation, decrease the maximum network size supported and reduce the amount of SRAM reserved to cache virtual-to-physical translations.

This means that only 1×512MB request or 2×256KB or 4×128KB requests may be in flight at any moment. To allow gmblock to use larger requests, while still achieving good disk and network utilization, we focus on increasing the amount of overlapped processing *within* each individual request. Our aim is to have the network sending data for a remote block read request even before the storage medium has finished serving it. This way, we can take advantage of the full bandwidth of the RAID controller while still having good overlapping of disk with network I/O.

6.2 Design

Servicing a block read request entails two steps: Retrieving data from disk to SRAM, then sending data from SRAM to the network.

To enable intra-request overlapping, the send from SRAM operation needs to be *synchronized* with the disk read operation, to ensure that only valid data are sent over the network. Ideally, this should be done with minimum overhead, in a portable, block device driver-independent way and with minimal changes to the semantics of the calls used by the nbd server for local and network I/O.

This kind of synchronization could be implemented in software, inside the nbd server, dividing each large remote read request of l bytes (e.g., 1MB) in much smaller chunks of c bytes (e.g., 4KB) then submitting them simultaneously via the POSIX Asynchronous I/O facility. However, this imposes significant software overhead.

We propose a synchronization mechanism working directly between the storage medium and the Myrinet NIC, in a way that does not involve the host CPU and OS running on top of it at all, while at the same time remaining independent of the specific type of block storage device used.

Let us consider the scenario when the server starts a user level send operation before the actual `read()` system call to the Linux I/O layer. This way, sending data over the wire is bound to overlap with fetching data from block storage into the Lanai SRAM. However, this approach will most likely fail, since there is no guarantee that the storage medium will be able to deliver block data fast enough.

To solve this problem we introduce the concept of a *synchronized* property for user level send operations. A synchronized GM operation ensures that the data to be sent from a message buffer are valid, before being put on the wire. When no valid data are available, the firmware simply ignores the send token while polling. The NIC works in lockstep with an external agent (the block device), throttling its send rate to match that of the incoming data (in our case, r_{disk}).

The NIC notices data transfer completions in chunks of c bytes. The value of c determines the synchronization grain and the degree of overlapping achieved (see Fig. 13); The NIC only starts sending after $t_1 = \frac{c}{r_{disk}}$ time units, then both the storage device and the NIC are busy for $t_2 = \frac{l-c}{r_{disk}}$, then the pipeline is emptied in $t_3 = \frac{c}{r_{net}}$ time units. Smaller values of c trade-off Lanai CPU usage for finer-grained synchronization.

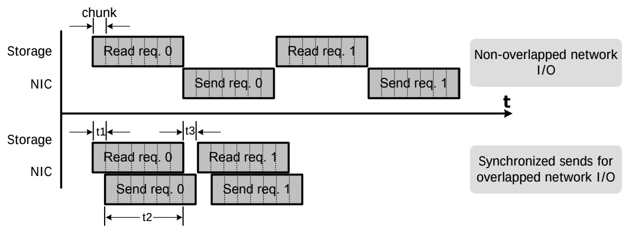

chunk
Storage
NIC
Read req. 0
Read req. 1
Send req. 0
Send req. 1
Non-overlapped network I/O
t
t1
t3
t2
Synchronized sends for overlapped network I/O

Fig. 13 Intra-request phase overlap

The semantics described above break the assumption that the whole of the message is available when the send request is issued, allowing the NIC to synchronize with an external agent while the data are being generated. However, as we discovered experimentally and explain in greater detail in section 6.4, this does not suffice to extract good performance from our platform; the design retains the assumption that the external agent places data into the message buffer as a single stream, in a sequential fashion. However, most real-world storage devices rely on *parallelizing* request processing to deliver aggregate high performance, e.g. by employing RAID techniques. Thus, incoming data comprise *multi-*

ple slower streams. We incorporate this multiplicity of streams in the semantics of synchronized send operations by allowing the sender to construct network packets from distinct locations inside the message buffer, and the receiver to support out-of-order placement of incoming fragments. Noticing DMA transfer completions *anywhere* in the message buffer is prohibitive, so we make a compromise. “Multiple stream” sends need the user to provide *hints* on the position and length of a finite of incoming streams inside the buffer. In our case, they are derived by the RAID array member count and chunk size.

Although our implementation of synchronized operations is Myrinet/GM based, it is portable to any programmable NIC which exposes part of its memory onto the PCI address space and features an onboard CPU. Synchronization happens in a completely peer-to-peer way, over the PCI bus, without any host CPU involvement.

6.3 Implementation issues on Myrinet/GM

The Myrinet NIC does not provide any functionality to detect external agents placing data directly into its SRAM via DMA, e.g., via a “dirty memory” bitmap. So, we emulate such functionality in firmware, by marking the relevant SRAM segments with 32-bit markers, and having the Lanai detect DMA completions by polling, as they get overwritten by incoming data. The probability of at least one overwritten marker going undetected because the value being sent coincides with the magic value being used is very low, e.g., for l =1MB and c =4KB, it holds: $P = 1 - \left(1 - 2^{-32}\right)^{\lceil \frac{l}{c} \rceil} \Longrightarrow P = 5.96 \times 10^{-8}$. Still, to ensure correctness, an extra marker is used right after the end of the block, and set by the nbd server when the data transfer is complete. Thus, in the worst case, with probability P no overlapping takes place.

For multiple stream support, we modified GM’s Go Back N protocol so that out-of-order construction of message packets at the sender side and placement at the receiver side is possible. Whenever a packet is injected into the network, the SDMA state machine keeps track of stream-specific state inside the associated send record based on the stream that packet data originates from; should the connection be rewound due to lost packets or timeouts, the firmware will know which of the stream-specific pointers to modify when resending. Similarly, the RDMA state machine does not assume incoming message fragments are to be placed serially inside the matched buffer; we extended the GM packet header so that an optional offset field is used to determine where to DMA incoming fragment data inside the message buffer.

The Lanai cannot address host memory directly but only through DMA. The cost of programming the PCIDMA engine to monitor the progress of a block transfer to in-RAM buffers is prohibitive, so synchronized GM operations are only available when sending from buffers in Lanai SRAM. However, this suffices for implementing an optimized version of gmblock’s data path.

6.4 Experimental evaluation

This section presents an experimental evaluation of gmblock extended to support synchronized operations versus the base version of gmblock using a direct disk-to-NIC data path. We do not include any instances of `gmblock-ram-{1,2,4}` since we have already shown how staging data in RAM-based buffers is detrimental to overall performance. The results were taken on Server B because it features a better performing, PCI-X bus.

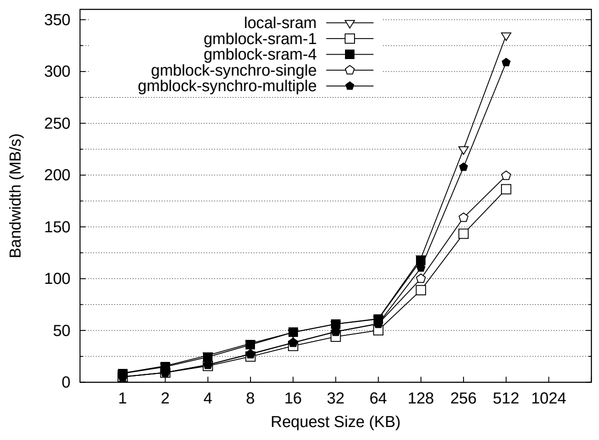
Experiment 2a: Synchronized Sends

We use two versions of gmblock: `gmblock-synchro-single` issues single-stream synchronized operations with one outstanding request, while `gmblock-synchro-multiple` supports multiple-stream synchronized operations. The number of streams is set equal to the number of disks in the RAID array. The results for both storage media are displayed in Fig 14(a), Fig 14(a) for the 3Ware Controller and the MBL device respectively.

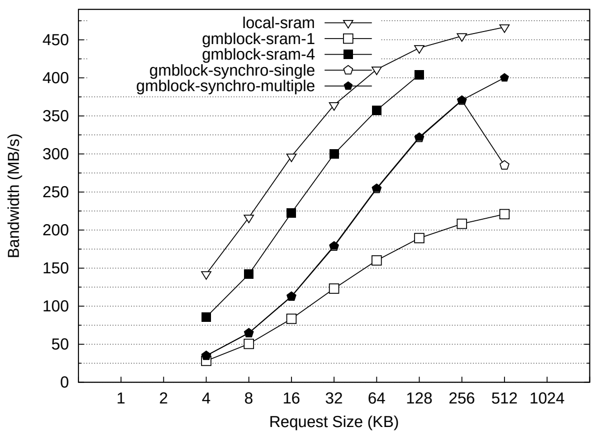
As expected using `gmblock-synchro-single` yields much better throughput over `gmblock-sram` for MBL (370MB/s, a 77% improvement, for 256KB-sized requests). Even with a single outstanding request `gmblock-synchro-single` reaches 91% of the maximum read throughput of `gmblock-sram-4`, by improving the latency of individual requests (e.g. 59% for 64KB-sized requests). This is a sharp drop-off in performance for 512KB-sized requests, which we focus on shortly. Contrary to MBL the performance gains of `gmblock-synchro-single` for the RAID configuration are marginal (7% improvement over `gmblock-sram-1`, for 512KB-sized requests).

Experiment 2b: RAID data movement

To better understand the reasons for the performance drop-off for 512KB MBL requests and the rather low performance of the RAID configuration, we need more insight on the way DMA operations progress over time. We use a custom utility, `dma_poll`, which provides data movement traces using predefined marker values (fig 16), similarly to the method described in Section 6.2. This way we can monitor when each individual chunk is DMAed into the message buffer. We find

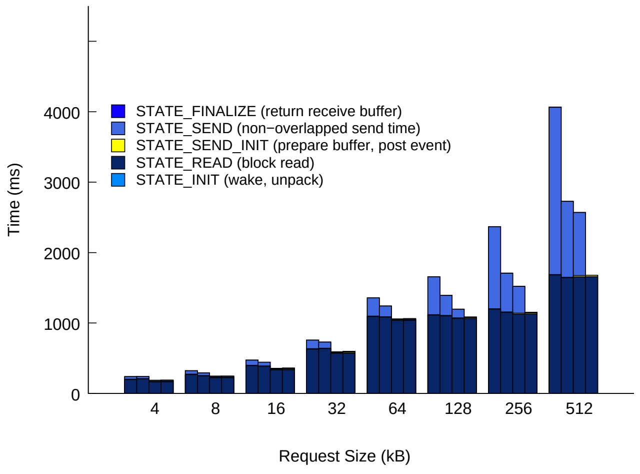


(a) RAID controller, sustained read bandwidth

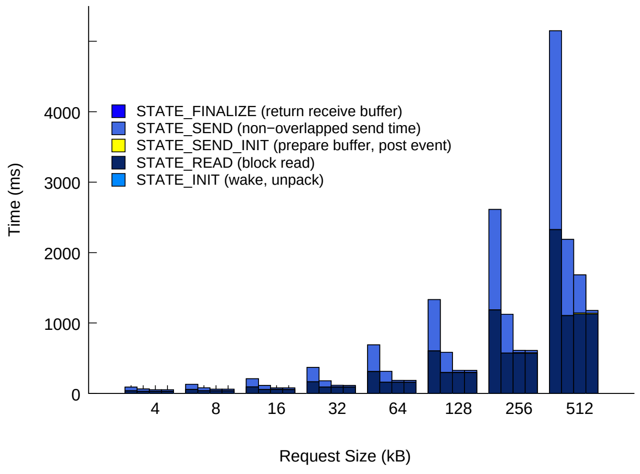


(b) MBL, sustained read bandwidth

Fig. 14 Sustained remote read bandwidth for single and multiple-stream synchronized operations

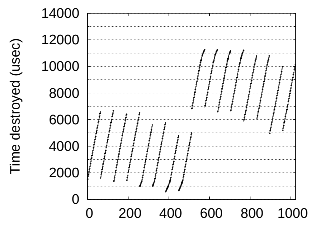


(a) RAID controller

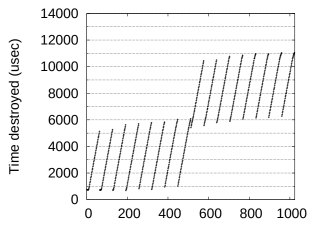


(b) MBL

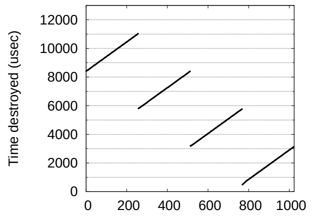
Fig. 15 Latency breakdown per request size for gmblock-{ram, sram, synchro-single, synchro-multiple}



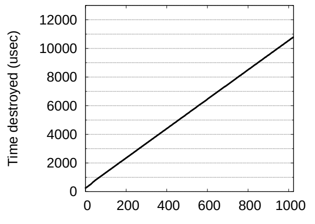
(a) RAID controller, AS



(b) RAID controller, CFQ



(c) MBL, AS



(d) MBL, CFQ

Fig. 16 DMA traces for the anticipatory and CFQ schedulers, 1024KB-sized requests

two reasons behind these results: (a) RAID data movement: `gmblock-synchro-single` ignores the fact that data are placed in different parts of the message buffer in parallel (see fig. 16(b)) and only overlaps disk and network I/O for the first chunk. (b) Linux I/O scheduling: The maximum hardware segment size for DMA operations with the MBL device is 256KB. For requests greater than that, the actual order that the segments are submitted depends on the Linux I/O scheduler. Using the anticipatory I/O scheduler leads to the two segments being reordered for 512KB requests (fig. 16(c)).

Hence the degree of overlapping for `gmblock-synchro-single` is lower.

Using `gmblock-synchro-multiple` works around these problems and achieves read rates close to that of local access. In the case of the 3Ware controller it reaches 92% of the maximum bandwidth (40% better than `gmblock-sram-4`) achieving near-perfect overlapping. This can be seen in fig. 15 where we have plotted the time spent for various steps of request processing for the gmblock server. We identify five different states: (a) `STATE_INIT`: Request received, being unpacked (b) `STATE_READ`: In disk I/O (c) `STATE_SEND_INIT`: Posting send event to the Lanai (d) `STATE_SEND`: Disk I/O done, send operation in progress (e) `STATE_FIN`: Returning receive token to GM The time spent on states other than `STATE_READ` or `STATE_SEND` was found to be negligible. `STATE_SEND` represents network I/O time that was not overlapped with disk I/O. It indicates the degree of disk to network I/O overlapping achieved and is negligible for `gmblock-synchro-multiple`.

7 Parallel filesystem deployment over gmblock

We have completed a prototype deployment of Oracle Cluster File system (OCFS2) over gmblock-provided shared storage, which allows us to evaluate gmblock with real-life application I/O patterns. This section describes the optimizations in the kernelspace gmblock client in order to support efficient end-to-end operation, and experimental results from the execution of various workloads on the parallel filesystem.

7.1 Client-side design and implementation

On the client side, gmblock runs as a kernelspace driver, presenting a standard block device interface to the rest of the system. Retaining the standard block device interface makes our framework instantly usable either directly as a raw device, e.g. by VM instances or a parallel database, or indirectly, through a shared-disk filesystem using standard POSIX I/O. Moreover, we retain the highly optimized, production-quality I/O path of the Linux kernel, which does I/O queueing, scheduling, request coalescing and mapping to physical scatter-gather lists, before presenting the requests to our driver.

The client uses Myrinet/GM for networking. GM targets user level communication, and requires all data to be in pinned-down buffers, contiguous in application VM space. To the contrary, requests handed to a block device driver involve I/O from/to physically discontiguous pages in the page cache. Thus, a straightforward implementation of the client would either have to remap those pages in the kernel's VM space and register them with GM before every message exchange – a very expensive operation – or move data through preallocated GM buffers.

We have developed custom extensions to GM to support zero-copy I/O, without any staging of block data in intermediate buffers. Since the client is already in the kernel, a block read or write request is specified in terms of a physical scatter-gather list; GM's address translation mechanism is of no value. Instead, we extend GM messaging, so that outbound and inbound message buffers can refer to scatter-gather lists. Two new primitives support this functionality: `gm_gather_send_with_callback()`, `gm_provide_receive_scatterlist_with_tag()`.

Essentially, we implement a stub of the remote storage medium on the NIC itself. This scheme, combined with the short-circuit data path on the server side, supports end-to-end zero-copy data movement from remote storage to scattered client memory segments. Scatter-gather I/O can reach all the way to user buffers for applications which implement their own data caching policies and perform direct I/O.

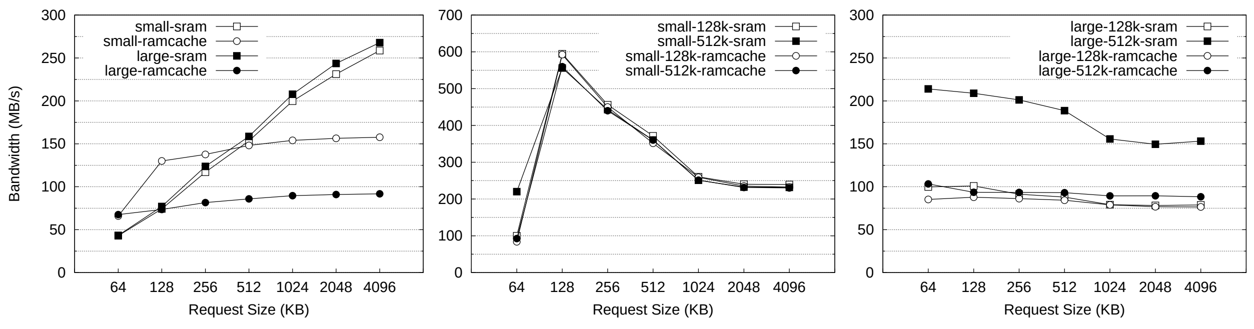
7.2 Experimental evaluation

Our testbed consists of 4 cluster nodes in configuration A and a storage server of configuration B. All cluster nodes access the 3Ware RAID0 array over virtual devices backed by the gmblock kernelspace client. They run OCFS2 1.5.0, which is part of the standard Linux kernel version 2.6.28.2. We had to patch the kernel for 16KB kernel stacks instead of the default 8KB, to work reliably with the long function call chains coming from stacking OCFS2 over gmblock over GM.

Using the proposed server-side data path means no server-side caching and prefetching is possible. To explore their effect, we compare two versions of gmblock: `gmblock-ramcache`, a version which passes both read and write data through RAM buffers using cached I/O, and `gmblock-sram`, which uses the proposed disk-to-NIC path for reads and only issues direct I/O requests. Writes still go through main memory, uncached, to work around the hardware limitation of the LanaiX which cannot support efficient peer-to-peer transfers as a read target (see Section 5).

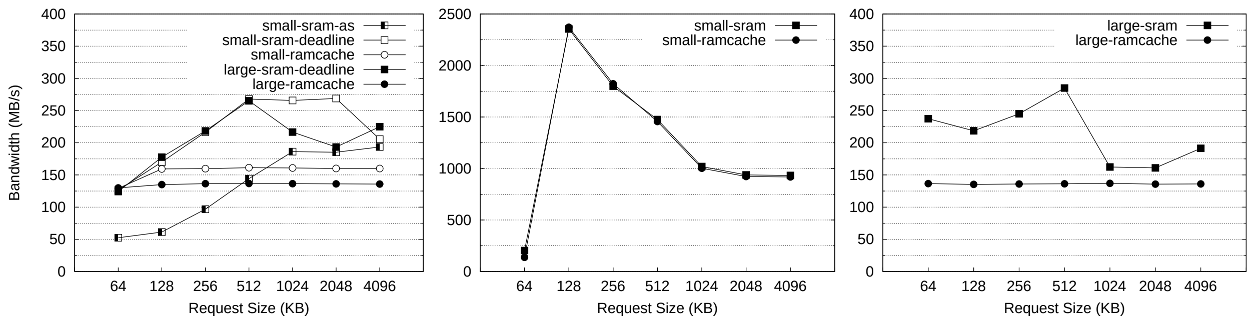
Caches are cleared on every cluster node before every experiment to ensure consistent results.

We run three different application benchmarks on the OCFS2-over-gmblock setup: IOzone [19], a server workload, and MPI-Tile-I/O [23].



(a) direct I/O (b) Small file (c) Large file

Fig. 17 IOzone, single-node performance



(a) direct I/O (b) Small file (c) Large file

Fig. 18 IOzone, multiple-node performance

Experiment 3a: Single-node IOzone performance
IOzone is filesystem benchmark generating a variety of different I/O patterns. We tested its performance in the read, re-read, and write modes. For every test, IOzone performs multiple passes varying the I/O size from 64KB to 4096KB. We used two different workloads. A "small" file of 512MB which fits entirely in a node's cache and a "large" file of 4GB. This is to demonstrate server and client-side cache effects. We show results from the read tests, since the re-read case coincides with the small file read case after the first pass, and writes follow the same data path in both implementations.

We look into the base performance of IOzone with a single client. Fig. 17(a) shows the performance of uncached (direct I/O) reads for various request sizes both for `gmblock-sram` and for `gmblock-ramcache`. We see that `gmblock-ramcache` is capped by the memory-to-PCI bandwidth at ~93MB/s and ~160MB/s for the large and small file respectively. The small file case has considerably better performance because the first 64KB pass brings the entire file in storage server RAM. Thus there is no disk-to-memory traffic competing with cache-to-NIC traffic.

The `gmblock-sram` case scales linearly with request size independently of file size since there is no client or server-side caching. It is interesting to note that `gmblock-ram` outperforms `gmblock-sram` for the initial read of 64KB requests. This is because the caching server may prefetch aggressively: the caching server takes advantage of readahead set at 512KB, so `gmblock-sram` begins to outperform `gmblock-ramcache` after the 512KB request mark and is ~1.64 and ~2.9 times better for the small and large file respectively.

Fig. 17(b) shows small file performance, for readahead settings of 128KB and 512KB. We see client-side caching in effect. For request sizes after 128KB, I/O requests are being served by the page cache on the client side, as every cluster node gets its own read lock on the shared data and caches the file independently. The steep drop as the request size increases is an artifact of processor cache behavior. As request size increases towards 512KB, IOzone's application buffer no longer fits in the L2 cache of the CPU, so RAM becomes the target of memory copy operations when hitting the local page cache. With a large enough request size, we essentially see memory copy bandwidth limitations. `gmblock-sram`'s behavior for the initial file read shows that prefetching is necessary to achieve good performance. When using the direct disk-to-NIC data path it is not possible to prefetch on the server, but client-initiated prefetching is still possible. The initial file read with 512KB readahead on the client outperforms the 128KB readahead setting by as much as 120%. Thus for all remaining experiments we continue with a readahead setting of 512KB on the clients.

Finally, Fig. 17(c) shows read performance for the large file case. This time, no cache reuse is possible. Performance of `gmblock-sram` with 128KB readahead is low because the application I/O request must complete fully before a new one can be issued by IOzone. When readahead is set at 512KB the Linux I/O layer will overlap data prefetching with page cache to application buffer copying. The bandwidth drop after 512KB is an artifact of L2 caching as in the small file case.

Experiment 3b: Multiple-node IOzone performance
We repeat the previous experiments, this time with four instances of IOzone running concurrently, one on each client node. Fig 18(a) demonstrates the aggregate attained bandwidth for direct I/O reads of various request sizes. All IOzone processes work on the same 512MB or 4GB file. This is the best possible scenario for `gmblock-ramcache`; It has consistently good performance, since it only fetches data once in memory, then all nodes can benefit from it as they read through the file at approximately the same rate. Achieving good performance with `gmblock-sram` proved more difficult. It quickly became apparent that the choice of the I/O scheduler on the server was crucial. Initial testing with the anticipatory scheduler showed poor disk efficiency. The bottleneck was the storage medium serving a seek workload. Testing with other schedulers available in the Linux kernel (deadline, noop, CFQ; only presenting results for deadline, for brevity) showed that performance varied significantly with request size. In the best case, I/O scheduling increases disk efficiency enough for `gmblock-sram` to outperform `gmblock-ram` by 66%. In the worst case, `gmblock-sram` only achieves 40% of `gmblock-ramcache`'s performance, for 64KB requests and the anticipatory scheduler.

Fig. 18(b) shows small file performance. After the initial read of the file, all clients read from the local caches concurrently at a rate of 2.4GB/s.

Fig. 18(c) shows large file performance for 512KB readahead. `gmblock-sram` consistently outperforms `gm block-ramcache`, but its performance is very sensitive to the application request size. We attribute this to the interaction of request timing with server-side I/O scheduling.

Experiment 3c: Server workload
We evaluate a web farm scenario, where all four clients run scripts simulating web server instances. Each instance serves randomly chosen files of fixed size from a single directory in the shared filesystem. We use the number of files served in a 2-minute period as a metric of sustained system throughput. The file set ranges from a small cacheable workload (70 files of 10 MBs each), to 1000 and 10000 files of 10MBs each (Fig. 19(a)).

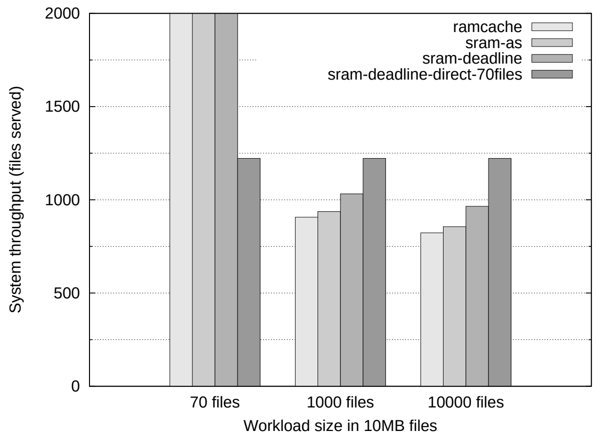
For the cacheable workload, there was no significant difference in the performance of `gmblock-ramcache` and `gmblock-sram` (results reached ∼7000files/2min and are off the chart). All clients quickly built a copy of the workload in their page caches, so the result is dominated by the memory copy rate when hitting the page cache. `gmblock-sram` performs 12% and 17% better for the 1000 and 10000-file case, respectively. It is bound by disk performance due to small file seeks. To confirm this, we repeat the test with the small 70-file workload, this time reading in `O_DIRECT` mode, to prevent any client-side caching (bar "sram-deadline-direct-70files" in the chart). With the disks performing seeks in a narrower range, performance improved by an extra 18% and 26% in the 1000 and 10000-case respectively.

Experiment 3c: MPI-IO Application
We use MPI-Tile-IO, an MPI-IO benchmark which produces a non-contiguous access workload similar to that of some visualization and numerical applications. The input file for the application is divided in a dense 2D set of tiles, with each peer process accessing a single tile. We perform the I/O needed to render a frame on a 4x4 tiled display, with 512x512, 1024x1024, or 4096x1024 tiles and 32 bytes per element. Total completion time for every configuration is shown in Fig. 19(b). In the best case, `gmblock-sram` delivers 39%, 51% and 57% the completion time of `gmblock-ramcache` for the three tile sizes respectively, although no I/O scheduler has consistently better performance.

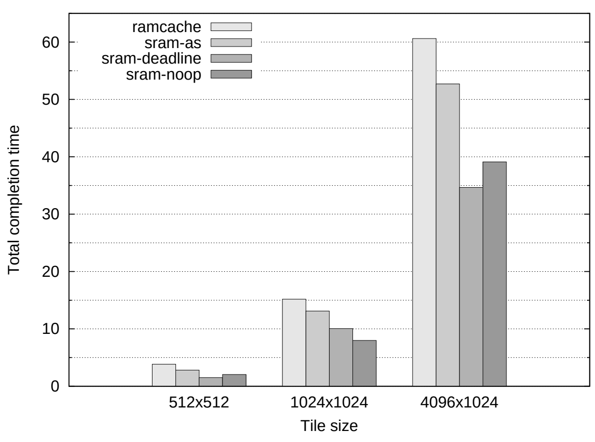
Overall, `gmblock-sram` performed better for all three workloads. However, the effect of the short-circuit data path only becomes visible when server-side I/O scheduling can remove the disk bottleneck due to concurrent access. This is why the performance increase is more pronounced for IOzone, whose access pattern comprises multiple peers streaming data concurrently, compared to the server and scientific application workloads, which lead to more frequent seeks.

8 Related work

The proposed framework for efficient remote block I/O over Myrinet has evolved from previous design approaches, as has been described in more detail in Section 3. Thus, it relates to past and ongoing research on scalable clustered storage, block-level storage sharing over



(a) Web server workload



(b) MPI-Tile-I/O performance

Fig. 19 Web server workload and MPI-Tile-I/O performance

TCP/IP and user level networking architectures, and parallel filesystems.

TCP/IP-based approaches for building nbd systems are well-tested, widely used in production environments and highly portable on top of different interconnection technologies, as they rely on – almost ubiquitous – TCP/IP support. They include the Linux Network Block Device (NBD), Red Hat's Global NBD (GNBD) used in conjunction with GFS [21], the Distributed RAID Block Device (DRBD) [8] and the GPFS Network Shared Disk (NSD) layer [24]. On the other hand, they exhibit poor performance, need multiple copies per block being transferred, and thus lead to high CPU utilization due to I/O load. Moreover, using TCP/IP means they cannot access the rich semantics of modern cluster interconnects and cannot exploit their advanced characteristics, e.g., RDMA, since there is no easy way to map such functions to the programming semantics of TCP/IP. As a result, they achieve low I/O bandwidth and incur high latency.

RDMA-based implementations [12,16,15] relieve the CPU from network protocol processing, by using the DMA engines and embedded microprocessors on NICs. By removing the TCP/IP stack from the critical path, it is possible to minimize the number of data copies required. However, they still feature an unoptimized data path, by using intermediate data buffers held in main memory and having block data cross the peripheral bus twice per request. This increases contention for access to main memory and leads to I/O operations interfering with memory accesses by the CPUs, leading to reduced performance for memory-intensive parallel applications.

The work in [18,17] addresses the end-to-end performance of a a Linux 2.4 kernel-based block sharing system, over a custom 10Gbps RDMA-capable interconnect with exclusive access. The authors show that network and disk interrupt processing overhead can have major impact on the attainable performance and focus on I/O protocol optimizations in order to alleviate it. We evaluate the effect of different I/O paths on storage server performance and focus on integration in an existing HPC infrastructure: gmblock is implemented in userspace, to simplify its design and be able to access structured storage (e.g., data in a filesystem) using kernel-provided abstractions. The proposed modifications follow Myrinet/GM semantics, preserve memory protection and can co-exist with communicating applications running on the same cluster node.

Our framework is not the only one to support direct paths between storage media and the network; a number of network block sharing systems have been described in the literature supporting block transfers from the disk to the NIC. However, they impose limitations which can hinder their applicability in real-world scenarios.

The DREAD project [7] describes a mechanism of controlling SCSI storage devices over SCI-provided remote memory accesses to their PCI memory-mapped I/O space; in this setup, the SCSI controller driver runs on the client. However, only a single remote node may be running the driver and accessing the SCSI adapter, hence no data sharing between multiple clients is possible. Also, the system needs source code changes to the SCSI driver so that it performs I/O over SCI-mapped memory. This approach has significant interrupt overhead; interrupts are routed via the server CPU, causing a network transaction and an interrupt on the client, which is caught by DREAD and routed to the SCSI driver. Thus, it does not scale well as the I/O rate increases. Finally, there is little room for client- or server-side optimizations to the protocol, since it works at a very low level directly between the SCSI driver and the device, as if they were directly connected over the PCI bus.

The work on Proboscis [10,11] implements a block-level data sharing system over SCI. It builds on the idea of having nodes act as both compute and storage servers and describes a kernel-based system for exporting block devices. It also mentions the possibility of direct disk-to-NIC transfers, by exploiting hardware support specific to SCI for mapping remote memory to a node's physical address space. This reduces host overhead significantly, but may lead to problems: first, it would only make sense for disk read operations (remote memory writes), since the overhead of remote reads over SCI is prohibitive; second, there is a low limit on the number of SCI memory mappings that may be active at any time, which would interfere with processes trying to make concurrent use of the interconnect – a problem analogous to Myrinet's limited amount of SRAM on the NIC; third, referring to SCI-mapped addresses directly makes error handling in case of network failures very complicated, as there is no way for the storage device to be notified whenever a memory access operation to a physically mapped remote location fails; and finally, there is no provision for coherence with the local OS's page cache. Our framework proposes synchronized sends to work around the limited amount of SRAM on the Myrinet NIC and avoids the problem of error handling by isolating network I/O in a distinct SRAM-to-wire or wire-to-SRAM phase.

The work on Off-Processor I/O with Myrinet (OPIOM) [9] was the first Myrinet-based implementation of direct data transfers from a local storage medium to the NIC. At the server side, OPIOM performs read-only direct disk-to-Myrinet transfers, bypassing the memory bus and the CPU. However, to achieve this, OPIOM makes extensive modifications to the SCSI stack inside the Linux kernel, in order to intercept block read requests so that the data end up not in RAM but in Lanai memory. This has a number of significant drawbacks: first, since the OPIOM server uses low-level OPIOM-specific SCSI calls to make such transfers, it can only be used with a single SCSI disk. Moreover, there is no provision for concurrent accesses to the SCSI disk, both over Myrinet and via the page cache, thus no write support is possible, since there is no way to invalidate blocks which have already been cached in main memory, when a remote node modifies them. Even for the read case, it is unclear what would happen if a remote node requested data recently changed by a local process, still kept in the page cache but not yet flushed to disk. Our proposed framework is able to ensure coherence with the page cache, both for reads and for writes by exploiting the direct-I/O semantics of the Linux 2.6 kernel; it will invalidate cached blocks before an `O_DIRECT` write and will write back any relevant cached pages to disk before an `O_DIRECT` read. By integrating parts on Lanai SRAM into the VM infrastructure provided by Linux and using `O_DIRECT`-type transfers, gmblock is disk-type agnostic and can construct an efficient disk-to-network path regardless of the underlying storage infrastructure, whether it is a SATA disk, storage accessible over Fibre-Channel or even a software RAID0 device. The only requirement is that a Linux driver capable of using DMA to service `O_DIRECT` transfers is available.

A different approach for bringing storage media closer to the cluster interconnect is READ^2 [5]. In READ^2, the whole of the storage controller driver resides on the Lanai processor itself, rather than in the Linux kernel. Whenever a request arrives from the network, it is processed by the Lanai, which is responsible for driving the storage hardware directly, thus bypassing the host CPU. However, removing the host CPU (and thus the Linux kernel executing on top of it) from the processing loop completely, limits the applicability of this approach to real-world scenarios for a number of reasons. First, it disregards all the work devoted to developing stable in-kernel block device drivers; every device driver needs to be rewritten without any of the hardware abstraction layers of the Linux kernel. Second, there is no coordination with the host system, so the disk being shared is inaccessible by local kernel drivers. As a result, no enforcement of per-user rights and no process isolation is possible. Finally, even for short commands to the storage controller, the Lanai cannot do PIO, but is instead forced to setup DMA transactions from and to the controller's I/O space. Thus, the latency of control operations becomes very high.

9 Conclusions and Future Work

We have described the design and implementation of gmblock, a framework for direct device-to-device data movement in commodity storage servers. Our prototype implementation over Myrinet supports direct I/O paths between the storage subsystem and the Myrinet NIC, bypassing the host CPU and memory hierarchy completely. The system combines the user level networking features of Myrinet/GM with enhancements in Linux's VM mechanism to build the proposed data path. Its operation is block device driver agnostic and preserves the isolation and memory protection semantics of the host OS. On the client side, gmblock supports scatter-gather I/O, allowing end-to-end zero-copy block transfers, from remote storage to user memory.

Experimental evaluation of different I/O paths on our prototype system has revealed the following: (a) Resource contention on the I/O path can lead to significant performance degradation. Moving block data in a direct disk-to-NIC path eliminates memory and peripheral bus contention, leading to significant performance improvements, in the range of 20-200% in terms of sustained remote read bandwidth, depending on the storage server's architectural characteristics, (b) Staging data in main memory interferes with computationally intensive tasks executing on the local CPU, causing up to 67% execution slowdown. The proposed techniques alleviate the problem, by increasing the availability of the main memory bus to the host CPUs, (c) The performance of the optimized data path depends on support for efficient peer-to-peer transfers by various hardware components, most notably peripheral bus bridges. The system proved flexible enough to support alternate data paths, when the hardware did not have adequate support for the needed functionality, (d) The data movement characteristics of RAID-based storage subsystems combined with limited memory on the Myrinet NIC reduce remote I/O performance, even when using an optimized data path.

To work around these limitations, we have proposed enhancements to the semantics of network send operations, so that the NIC supports synchronization with external agents doing peer-to-peer transfers over the peripheral bus. To support multiple incoming data streams from RAID storage, we enhanced the Myrinet network protocol so that it supports out-of-order fragment delivery into message buffers.

We have deployed the OCFS2 shared-disk filesystem on a distributed setup on top of gmblock and evaluated its performance with a variety of workloads. We see that the applications generally have performance gains due to increased remote I/O bandwidth, although results are very sensitive to application I/O patterns. Efficient server-side I/O scheduling is needed, to eliminate the disk bottleneck due to concurrent usage.

We believe gmblock's approach is generic enough to be applicable to different types of devices. An example would be a system for direct streaming of data coming from a video grabber to structured, filesystem based storage. Moreover our approach lends itself to resource virtualization: each user may make use of the optimized data path while sharing access to the NIC or other I/O device with other users transparently.

Although the current implementation is Myrinet-based, we plan to explore porting the design to other interconnects, such as Infiniband. Essentially, the server-side data path requires a DMA-enabled network interface with a small, fast, PCI-addressable memory area close to it. Our approach may be applicable to Infiniband, given hardware modifications to include the needed memory space. Next, the relevant areas would be registered to partake in RDMA operations. We can emulate this configuration using a programmable I/O adapter like the Cyclone NIC, between the PCI interface and the Infiniband NIC. On the client side, we extend the GM firmware to offload block device functionality to the NIC and support scatter-gather I/O directly to/from the physical address space. This optimization is not possible with current non-programmable Infiniband NICs. Instead, RDMA can be used for scatter-gather, with the added cost of memory registration.

We also plan to port our system to high-end PCI Express-based systems, using multiple Myri-10G NICs, in order to explore the efficiency of peer-to-peer transfers over PCI-E switch chips. We expect the effects of resource contention on the I/O path to be even more pronounced in such setup, given the network's 10Gbps transfer rate.

References

1. I. T. Association. InfiniBand Architecture Specification, Release 1.0, 2000. http://www.infinibandta.org/specs.
2. R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *Computer*, 31:53–60, 1998.
3. J. G. Blas, F. Isaila, J. Carretero, and rossemann Thomas. Implementation and Evaluation of an MPI-IO Interface for GPFS in ROMIO. In *Proceedings of the 15th EuroPVM/MPI 2008 Conference*, Dublin, Ireland.
4. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb 1995.
5. O. Cozette, C. Randriamaro, and G. Utard. READ^2: Put Disks at Network Level. In *CCGRID'03, Workshop on Parallel I/O*, Tokyo (Japan), May 2003.
6. Cyclone. PCI-X 740 Intelligent Dual Gigabit Ethernet Controller Datasheet. http://www.cyclone.com/pdf/PCIX740.pdf.
7. M. B. Dydensborg. Direct Remote Access to Devices. In *Proceedings of the Fourth European Research Seminar on Advanced Distributed Systems*, May 2001.
8. L. Ellenberg. DRBD 8.0.x and beyond: Shared-Disk Semantics on a Shared-Nothing Cluster. In *LinuxConf Europe 2007*, Cambridge, England, Sept. 2007.
9. P. Geoffray. OPIOM: Off-Processor I/O with Myrinet. *Future Gener. Comput. Syst.*, 18(4):491–499, 2002.
10. J. S. Hansen. Flexible Network Attached Storage using RDMA. In *Proceedings of the 9th IEEE Symposium on High-Performance Interconnects*, 2001.
11. J. S. Hansen and R. Lachaise. Using Idle Disks in a Cluster as a High-Performance Storage System. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.
12. K. Kim, J.-S. Kim, and S.-I. Jung. GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux.

In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

13. E. Koukis and N. Koziris. Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 345–354, Washington, DC, USA, 2006. IEEE Computer Society.
14. T. Lahiri, V. Srihari, W. Chan, N. MacNaughton, and S. Chandrasekaran. Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 683–686, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
15. S. Liang, W. Yu, and D. K. Panda. High Performance Block I/O for Global File System (GFS) with Infiniband RDMA. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 391–398, Washington, DC, USA, 2006. IEEE Computer Society.
16. J. Liu, D. K. Panda, and M. Banikazemi. Evaluating the Impact of RDMA on Storage I/O over Infiniband. In *SAN-03 Workshop (in conjunction with HPCA)*, 2004.
17. M. Marazakis, V. Papaefstathiou, and A. Bilas. Optimization and Bottleneck Analysis of Network Block I/O in Commodity Storage Systems. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, pages 33–42, New York, NY, USA, 2007. ACM.
18. M. Marazakis, K. Xinidis, V. Papaefstathiou, and A. Bilas. Efficient Remote Block-level I/O over an RDMA-capable NIC. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 97–106, New York, NY, USA, 2006. ACM.
19. W. D. Norcott and D. Capps. IOzone Filesystem Benchmark. http://www.iozone.org/docs/IOzone_msword_98.pdf.
20. F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects 9*, Stanford University, Palo Alto, CA, August 2001.
21. K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, 1999.
22. J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, An Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 17–17, New York, NY, USA, 2001. ACM.
23. R. Ross. Parallel I/O Benchmarking Consortium. http://www.mcs.anl.gov/research/projects/pio-benchmark.
24. F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.
25. S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages 319–342, College Park, MD, 1996. IEEE Computer Society Press.