Efficient Block Device Sharing over Myrinet with Memory Bypass

Evangelos Koukis and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{vkoukis, nkoziris}@cslab.ece.ntua.gr

Abstract

Efficient sharing of block devices over an interconnection network is an important step in deploying a shared-disk parallel filesystem on a cluster of SMPs. In this paper we present gmbock, a client/server system for network sharing of storage devices over Myrinet, which uses an optimized data path in order to transfer data directly from the storage medium to the NIC, bypassing the host CPU and main memory bus. Its design enhances existing programming abstractions, combining the user level networking characteristics of Myrinet with Linux's virtual memory infrastructure, in order to construct the data path in a way that is independent of the type of block device used. Experimental evaluation of a prototype system shows that remote I/O bandwidth can improve up to 36.5%, compared to an RDMA-based implementation. Moreover, interference on the main memory bus of the host is minimized, leading to an up to 41% improvement in the execution time of memory-intensive applications.

1 Introduction

Clusters built out of commodity components are becoming more and more prevalent in the supercomputing sector as a cost-effective solution for building high-performance, cost-effective parallel platforms. Symmetric Multiprocessors, or SMPs for short, are commonly used as building blocks for scalable clustered systems, when interconnected over a high bandwidth, low latency communications infrastructure, such as Myrinet [1], SCI [4] or Infiniband.

To meet the I/O needs of modern HPC applications, a distributed, cluster filesystem needs to be deployed, allowing processes running on cluster nodes to access a common filesystem namespace and perform I/O from and to shared data concurrently. Today, there are various cluster filesystems implementations which focus on high performance, i.e. high aggregate I/O bandwidth, low I/O latency and high number of sustainable I/O operations per second, as multiple clients perform concurrent access to shared data. At the core of their design is a *shared-disk* approach, in which all participating cluster nodes are assumed to have equal access to a shared storage pool. The shared-disk approach is followed by filesystems such as IBM's General Parallel File System GPFS [9], Oracle's OCFS2, Red Hat's Global File System (GFS) [10], SGI's Clustered XFS and Verita's VxFS, which aim at providing a high performance parallel filesystem for enterprise environments.

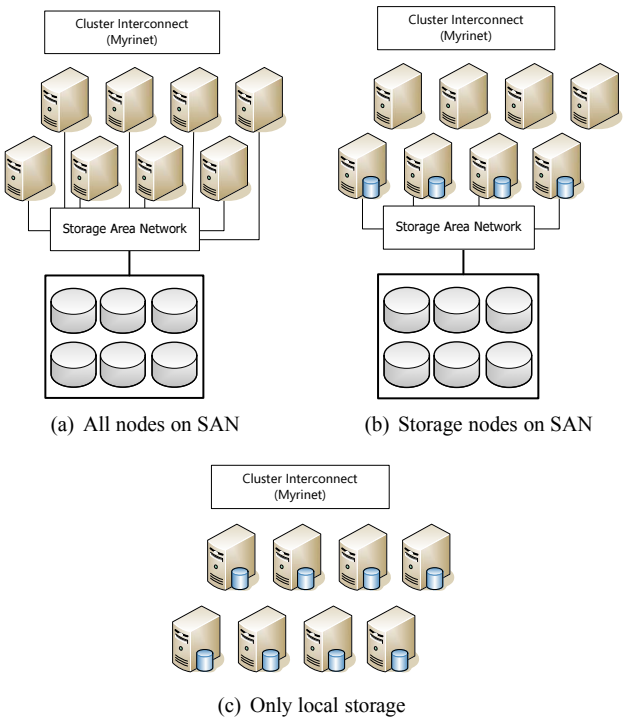
In such environments, the requirement that all nodes have access to a shared storage pool is usually fulfilled by utilizing a high-end Storage Area Network (SAN), traditionally based on Fibre-Channel (as in Fig. 1(a)). An SAN is a networking infrastructure providing high-speed connections between multiple 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 entails maintaining two separate networks, one for access to shared storage and a distinct one for Inter-Process Communication (IPC) between peer processes, e.g. those belonging to the same MPI job. This increases the cost per node, since the SAN needs to scale to a large number of nodes and each new member of the cluster needs to be equipped with an appropriate interface to access it (e.g. an FC Host Bus Adapter). Moreover, while the number of nodes increases, the aggregate bandwidth to the storage pool remains constant, since it is determined by the number of physical links to the storage enclosures. Finally, to eliminate single points of failure (SPOFs) on the path to the shared storage pool, redundant links and storage controllers need to be used, further increasing total installation costs.

To address these problems, a shared-disk filesystem is more commonly deployed in such way that only a small fraction of the cluster nodes is physically connected to the SAN ("storage" nodes), *exporting* the shared disks for block-level access by the remaining nodes, over the cluster interconnection network. In this approach (Fig. 1(b)), all nodes can access the parallel filesystem by issuing block read and write requests over the interconnect. The storage nodes receive the requests, pass them to the storage subsystem and eventually return the results of the operations back to the client node.

This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (75%) and National Resources (25%).

1-4244-0910-1/07/$20.00 ©2007 IEEE.



(a) All nodes on SAN (b) Storage nodes on SAN

(c) Only local storage

Figure 1. Interconnection of cluster nodes and storage devices

Taken to extreme, this design approach allows shared-disk filesystems to be deployed over shared-nothing architectures, by having each node contribute part or all of its locally available storage (e.g. a number of directly attached Serial ATA or SCSI disks) to a *virtual*, shared, block-level storage pool (Fig. 1(c)). This model has a number of distinct advantages: first, aggregate bandwidth to storage increases as more nodes are added to the system; since more I/O links to disks are added with each node, the performance of the I/O subsystem scales along 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. These nodes have a dual role, both as compute and as storage nodes.

The cornerstone of this design is the *network disk sharing* layer, usually implemented in a client/server approach. 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 (Fig. 3(a)). There are various implementations of such systems, facilitating block-level sharing of storage devices over the interconnect. 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 and Redhat's GFS can also be deployed over an improved version called GNBD.

However, all of these implementations are based on TCP/IP. Thus, they treat all modern cluster interconnects uniformly, without any regard to their advanced communication features, such as support for zero-copy message exchange using DMA. Employing a complex protocol stack residing in the kernel results in very good code portability but imposes significant protocol overhead; using TCP/IP-related system calls results in frequent data copying between userspace and kernelspace, increased CPU utilization and high latency. Moreover, this means that less CPU time is made available to the actual computational workload executing on top of the cluster, as its I/O load increases.

On the other hand, cluster interconnects such as Myrinet and Infiniband are able to remove the OS from the critical path (*OS bypass*) by offloading communication protocol processing to embedded microprocessors onboard the NIC and employing DMA engines for direct message exchange from and to userspace buffers. This leads to significant improvements in bandwidth and latency and reduces host CPU utilization dramatically. As explained in greater detail in the section on related work, there are research efforts focusing on implementing block device sharing over such interconnects and exploiting their Remote DMA (RDMA) capabilities. However, the problem still remains that the data follow an unoptimized path. Whenever block data need to be exchanged with a remote node they first need to be transferred from the storage pool to main memory, then from main memory to the interconnect NIC. These unnecessary data transfers impact the computational capacity of a storage node significantly, by aggravating contention on shared resources as is the shared bus to main memory and the peripheral (e.g. PCI) bus. Even with no processor sharing involved, compute-intensive workloads may suffer significant slowdowns since the memory access cost for each processor becomes significantly higher due to contention with I/O on the shared path to memory.

In this paper, we present an efficient method for implementing network shared block-level storage over processor and DMA-enabled cluster interconnects, using Myrinet as a case study. Our approach, called gmblock, is based on direct transfers of block data from disk to network and vice-versa and does not require any intermediate copies in RAM. It builds on existing OS and user level networking abstractions, enhancing them to support the necessary functionality, instead of relying on low-level architecture-specific code changes. By employing OS abstractions, such as the Linux VM mechanism and the Linux I/O subsystem to construct the proposed data path, we maintain UNIX-like process semantics (I/O system calls, process isolation and memory protection, enforcement of access rights, etc). Moreover, application code remains portable across different architectures and only minimal code changes are necessary. Thus, gmblock can be a viable solution for implementing shared-disk parallel filesystems on top of distributed storage, and can be integrated in existing parallel I/O infrastructures as an efficient substrate for network block device sharing. The proposed extensions to the Linux VM mechanism and the GM message-passing

framework for Myrinet are generic enough to be useful for a variety of other uses, providing an efficient framework for direct data movement between I/O and communication devices, abstracting away architecture- and device-specific details.

In the rest of this work, we begin with basic user level networking concepts and their implementation on Myrinet in order to gain some insight on the software and hardware environments we are interested in, then present different approaches for designing network block devices and the evolution of gmblock's design from them (Section 2). Section 3 concerns the specifics of gmblock's implementation on top of Linux and Myrinet. In Section 4 we evaluate gmblock's performance experimentally and compare it to that of existing approaches, pinpointing the various bottlenecks, which determine overall system performance. Finally, Section 5 presents previous research related to gmblock, while Section 6 summarizes our conclusions and explores possible directions for future work.

2 Design of gmblock's nbd mechanism

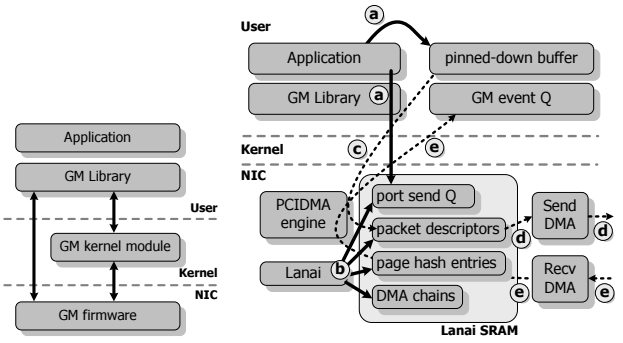
2.1 User level networking

This section contains a short introduction to the inner workings of Myrinet and its GM middleware, in order to gain some insight on how an nbd[1] system interacts with the interconnection network during the transfer of block-level data. The Myrinet software stack is displayed in Fig. 2(a). Myrinet NICs reside on the peripheral bus of a cluster node (PCI/PCI-X in the case of Myrinet-2000 used on our testbed). They feature a RISC microprocessor, called the Lanai, which undertakes almost all network protocol processing, 2-8MB of SRAM for use by the Lanai and three different DMA engines; one is responsible for DMA transfers of message data between host memory and Lanai SRAM over the half-duplex PCI/PCI-X bus, while the other two undertake transferring data between Lanai SRAM and the full-duplex 2+2Gbps 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 in order to allow direct access to the NIC from userspace, without the need to enter the kernel via system calls (OS bypass) while maintaing system integrity and ensuring process isolation and memory protection.

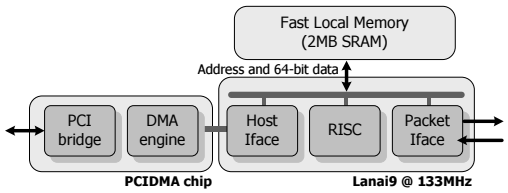
In Fig. 2(c) the main components onboard a Myrinet M3F-PCI64B-2 NIC are displayed, i.e. the DMA engines (one on the PCIDMA chip and two on the packet interface), the Lanai and its SRAM. In later versions of the NIC, such as the M3F-PCIXD-2 used on our testbed, most of this functionality has been integrated on a newer version of the Lanai, called the LanaiX chip.

Zero-copy user level communication is accomplished by mapping parts of Lanai SRAM (called *GM ports*) into the VM

[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.



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



(c) Main components on a Myrinet NIC

Figure 2. Implementation of user level networking by Myrinet/GM

address space of an application. This is a privileged operation, which is done via system calls to the GM kernel module during the application's initialization phase. Each port contains queues of send and receive descriptors, which the application can access directly. The GM firmware polls these queues periodically, in order to detect any newly posted request. In case of a send operation, it uses DMA in order to transfer the required data from host RAM to Lanai SRAM, then from Lanai SRAM to the NIC of the receiver node, while the converse happens during a receive.

Fig. 2(b) shows the steps that need to take place for a send operation to complete successfully. The solid lines lines correspond to operations involving either the host CPU (Programmable I/O – PIO) or the Lanai. The dashed lines correspond to DMA operations. A `gm_send()` operation entails the following basic steps: **(a)** The application computes the message to be sent in a pinned-down userspace buffer. The buffer must have been pinned down beforehand, using GM-specific calls to the GM kernel module, so that it resides in DMA-able host memory and is never swapped out by the kernel's VM subsystem. Then, it uses the GM user library to place a send token containing the virtual address of the buffer in an open port's send queue, in Lanai SRAM **(b)** The GM firmware notices the token as it polls the queues periodically. To be able to construct the appropriate DMA chain for the PCIDMA controller, it first needs to perform a virtual-to-physical address translation, in order to know the physical address where the data resides. This is the step that maintains process isolation and memory protection; even if the application was in the position to know the physical address

for the buffer, it wouldn't be trusted. A malicious application would provide malformed physical addresses, in order to read or write to arbitrary regions in host memory. Instead, the Lanai performs the virtual-to-physical translation on its own, based on platform-independent pagetables, which reside in host memory and are managed by the GM kernel module. Thus, an application must use GM-specific calls for memory allocation, instead of `malloc()`, which invoke the GM kernel module in order to register the region in GM's pagetables.An LRU cache of the pagetables is kept in Lanai SRAM (called the "page hash entries" region) and is updated by the Lanai via DMA whenever necessary. After consulting the page hash entries, the firmware programs the PCIDMA engine **(c)** Message data are brought via DMA into structures called Myrinet packet descriptors, kept in SRAM **(d)** The firmware programs the Send DMA engine in order to send the data to the remote NIC **(e)** An acknowledgement is received by the remote NIC. The firmware uses the PCIDMA engine in order to place an event of the appropriate type in the port's *event queue*, which resides in host memory. The application can keep polling the queue in order to discover that the send operation has completed successfully. Alternatively, it can enter the kernel and block in the GM module, waiting for an interrupt from the NIC to be received.

It is important to note that sending – and receiving – a message using GM is in fact a two-phase process:

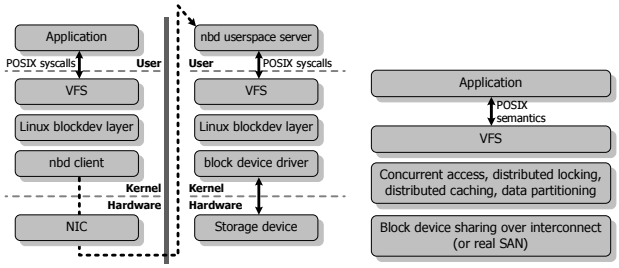
Host-to-Lanai DMA Virtual-to-physical translation takes place, the PCIDMA engine starts, message data are copied from host RAM to Lanai SRAM

Lanai-to-wire DMA Message data are retrieved from SRAM and sent to the remote NIC by the Send DMA engine.

2.2 Evolution from previous nbd designs

The main principle behind an nbd client/server implementation is portrayed in fig. 3(a). The nbd client usually 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 usually not necessary to 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.



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

Figure 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();
      /* Suppose it's a read */
      lseek(fd, cmd->start, SEEK_SET);
      read(fd, &reply->payload, cmd->len);
      insert_packet_headers(&reply, cmd);
      send_over_net(reply, reply->len);
}
```

Figure 4. Pseudocode for an nbd server

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 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 impementation, 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.

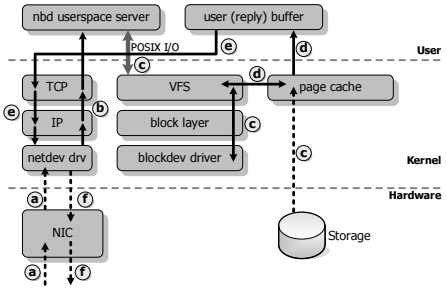
In Fig. 5(b), we can see the actual path followed by data, at the physical level. The labels correspond one-to-one with

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, in order to be sent over the network **(f)** The data cross the memory bus and the peripheral bus once again, in order to be sent over the network via DMA.

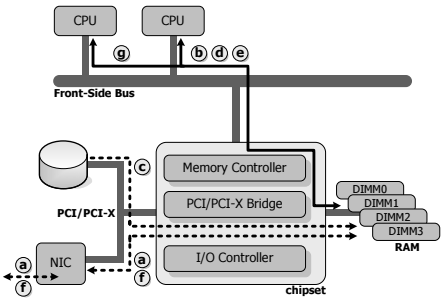
This data path is characterized by a large amount of redundant data movement. The number of copies needed to move data from the disk to the TCP/IP socket can be reduced by allowing the kernel more insight into the semantics of the data transfer; one could map the block device onto userspace memory, using `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. We must bear in mind that the actual goal is to transfer the requested block data from the storage medium to the NIC (conversely in the case of a write operation). In fact, the data being transferred are probably of no interest to the local host. Despite that, even if only one kernel copy takes place, they have to cross the peripheral bus twice, and the memory bus at least four times (one per DMA transfer and twice for the CPU-based memory copy). Thus, twice the sustained network I/O bandwidth is needed on the PCI bus and four times on the memory bus. If we take into account that the storage nodes usually function as compute nodes as well (path `(g)`), it is clear that filesystem I/O can lead to reduction in the memory bandwidth made available for computation on the CPU(s) of the system, causing memory contention and execution slowdowns. Moreover, a significant amount of CPU cycles is lost to copying block data and processing network interrupts. Having the CPU perform data copying may also have adverse cache effects: block data are read into the cache even though they will not be reused, evicting parts of the working set of the application code previously executing on this CPU.



(a) Data path at the logical level



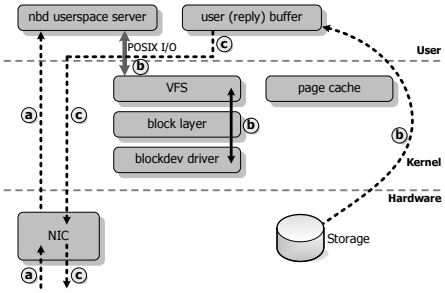
(b) Data path at the physical level

Figure 5. TCP/IP based nbd server

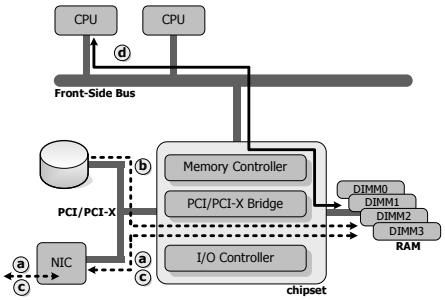
The problem is alleviated, if a user level networking approach is used. When a cluster interconnect such as Myrinet is available, the OS kernel can be bypassed during the network I/O phase, by extending the nbd server application so that GM is used instead of TCP/IP. In this case, some of the redundant 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 application uses `O_DIRECT`-based I/O so that the storage device is programmed to place block data into userspace buffers via DMA **(c)** The reponse 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`), since they contend for access to RAM.

2.3 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,

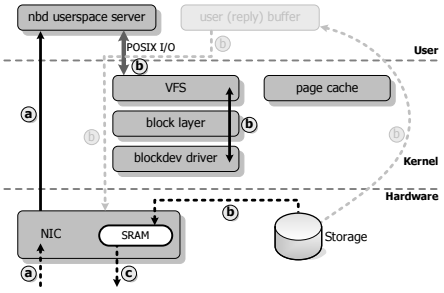


(a) Data path at the logical level

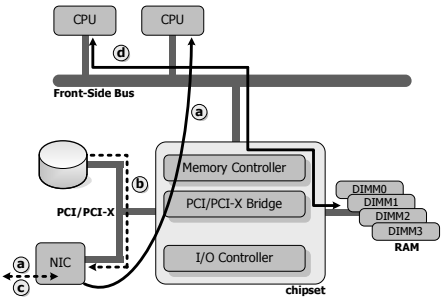


(b) Data path at the physical level

Figure 6. GM-based nbd server



(a) Data path at the logical level



(b) Data path at the physical level

Figure 7. Proposed gmblock server

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 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 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

The inclusion of RAM buffers in all previous data paths is a necessity arising from the 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 are required in the nbd server source code to support the enhanced functionality.

Let's assume a GM-based nbd server, 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 unaltered, 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. It is important to note that almost no source code changes are needed in the nbd server's source code to support this enhanced data path. The server process still issues `read()` 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.

2.4 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.

The proposed nbd system aims at minimizing the interference of remote I/O with local computation when providing a direct-attached storage abstraction to a parallel filesystem stacked on top of it, as in Fig. 3(b). In this context, nodes are assumed to interact at a much higher level, in order to coordinate concurrent accesses to the virtual shared block device and implement a distributed caching and invalidation mechanism. Thus, whenever a request finally reaches the block device layer, it has already passed through the filesystem caching layer and needs to be serviced in the least obtrusive way possible; the data can be assumed to be of no real importance to the server node and the locally executing processes, so it makes no sense to store them in the server-side page cache, or to have them cross the memory bus.

3 Implementation details

3.1 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 is done for message passing from and to userspace buffers in host RAM.

The described changes were tested on GM-2.0.22 and GM-2.0.26, running on an Intel EM64T system and an Intel i386 system respectively, under Linux kernel v2.6.16. 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.

The functionality that needs to be supported, along with the relevant parts of GM is:

- 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 using `gm_send()` and `gm_provide_receive_buffer()` from/to Lanai SRAM (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 1024KB 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 refers to Lanai SRAM, and marks it appropriately in the token passed to the Lanai. There, it is treated specially by the firmware, which skips the Host-to-Lanai DMA part, constructs the needed Myrinet packets and only performs the Lanai-to-wire DMA operation.

3.2 Linux VM support for direct I/O with PCI memory-mapped ranges

To implement gmblock's enhanced data path, the Linux VM mechanism must be extended 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_PCI` 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 managable 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.

4 Experimental evaluation

To quantify the performance benefits of gmblock's optimised data path (`gm-myri`), we compare it experimentally to two implementations. One is Red Hat's GNBD (`tcpip-gnbd`), the reworked version of NBD which accompanies GFS, the other is gmblock itself, running over GM without the proposed optimization, thus using a data path which crosses main memory (`gm-mem`).

Our experimental platform consists of two SMP nodes. One functions as the client, the other as the server. Each node has two Pentium III@1266MHz processors (16KB L1 I cache, 16KB L1 D cache and 512KB unified L2 cache, with 32 bytes per line) on a Supermicro P3TDE6 motherboard. Two PC133 SDRAM 512MB DIMMs are installed for a total of 1GB RAM per node. The motherboard is based on the Serverworks ServerSet III HC-SL chipset, which includes the Broadcom CIOB20 PCI bridge to connect the Northbridge to two different PCI segments. One is 64bit/66MHz/3.3V with two slots, the other is 64bit/33MHz/5V with five slots. A jumper on the motherboard can be closed to have the two 66MHz slots clocked at 33MHz.

The storage medium to be shared over Myrinet is provided by a 3Ware 9000S-8 SATA RAID controller, which has 8 SATA ports on a 64bit/66MHz PCI adapter. We built a hardware RAID0 array out of 8 Western Digital WD2500JS 250GB SATA II disks, which is exported as a single drive to the host OS. Array blocks are distributed evenly among the disks with a chunk size of 64KB. The nodes are connected back-to-back with two Myrinet M3F-PCIXD-2 NICs, each in a 64bit/66MHz PCI slot. The NICs use the LanaiXP@225MHz processor with 2MB of SRAM. Linux kernel 2.6.16.5, GM-2.0.26 and 3Ware driver version 2.26.02.007 are used.

Initially, we measure the maximum read bandwidth provided by the 3Ware controller, locally. To this end, `gmblock_srv`, gmblock's server component, is used in test mode, performing back-to-back requests of fixed size for consecutive blocks in `O_DIRECT` mode. The data are transferred either to in-RAM buffers (for `gm-mem`) or to Lanai SRAM buffers (for `gm-myri`). Our initial hardware setup involved two Myrinet NICs on the 66MHz slots and the 3Ware controller on a 64bit/33MHz slot. This test revealed that RAID-to-RAM transferres outperformed RAID-to-NIC transfers by a very large margin (maximum $\sim$380MB/s vs. $\sim$80MB/s). With trial and error, we found the problem to be with the CIOB20 bridge and the fact that the adapters were in different PCI segments. When we replaced one of the NICs with the 3Ware controller, removing the PCI bridge from the path, PCI bursts from the RAID controller to the NIC became possible and the RAID-to-NIC bandwidth reached the expected levels. The results are displayed in Fig. 8(a), as `local-mem` and `local-myri`, for request sizes 1, 2, ..., 1024KB. The bandwidth increases singificantly for request sizes $> 64KB$, since this is the chunk size of the RAID array, while a 512KB request can be processed in parallel by all 8 disks.

The first test (Fig. 8(a)) measures network bandwidth for all three implementations, for various request sizes. Since only one client is used, to achieve good utilization of Myrinet's 250MB/s link it is important to pipeline requests correctly. We tested with one, two and four outstanding requests for gmblock, both for `gm-mem` and for `gm-myri`. We are interested in bottlecks on the CPU, the RAID controller and the PCI. In general, GNBD performs poorly and cannot exceed 68MB/s on our platform. For `gm-myri` and `gm-mem`, if only one request is in flight at each time, the system is dominated by latency and the achievable bandwidth is rather low. However, for two and four outstanding requests, the effects of limited PCI bandwidth start to appear. Since the data cross the PCI bus twice for `gm-mem`, its maximum bandwidth is $\frac{1}{\frac{1}{337}+\frac{1}{397}} = 182MB/s$. Indeed, for request sizes over 256KB for `gm-mem-4`, or 512KB for `gm-mem-2` it does not exceed 177MB/s. On the other hand, `gm-myri-4` utilizes the full RAID controller bandwidth for 256KB requests (194.6MB/s, 14% better than `gm-mem`), while `gm-myri-2` exceeds 200MB/s for 512KB requests (13% better). Since `gw-mem` has already saturated the PCI bus, the improvement would be even more visible for `gm-myri-4` if larger request sizes could be used, since the RAID bandwidth cap would be higher for 512KB requests. Unfortunately, the available memory for SRAM buffers (1024KB) only sufficed for $1 \times 1024KB$ request, or $2 \times 512KB$ outstanding requests or $4 \times 256KB$ outstanding requests. Alternatively, we could have used two Myrinet NICs, so that double the amount of SRAM would be available, but this was not possible on our testbed due to the limited number of PCI slots. In any case, `gm-myri-4` is only limited by the available RAID bandwidth for a given request size and achieves good request pipelining even with only one client.

It is also interesting to see the effect of remote I/O on the locally executing processes on the server. On one hand, TCP/IP processing for GNBD consumes a large fraction of

CPU power *and* a large percentage of memory bandwidth for intermediate data copying. While `gm-mem` 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.

For the second set of experiments, we run `tcpip-gnbd`, `gm-mem-4` and `gm-myri-4` along with a compute-intensive benchmark, on only one of the CPUs. The benchmark is a process of the `bzip2` compression utility, which performs indexed array accesses on a large working space ($\sim$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 [6]. There is no processor sharing involved; the nbd server can always run on the second, otherwise idle, CPU of the system. In Fig. 8(b) we show the normalized execution time of `bzip2` along with the normalized I/O bandwidth of each nbd implementation, relative to executing on their own. In the worst case, `bzip2` slows down by as much as 70%, when `gm-mem` is used with 512KB requests (a 41% improvement for `gm-myri`). On the other hand, the benchmark runs with negligible interference when `gm-myri` is used, since the memory bus is bypassed completely and its execution time remains almost constant.

Finally, we repeated the previous experiments for a PCI 64bit/33MHz bus, by clocking the two 66MHz slots at 33MHz. In this case (Fig. 9), again `gm-mem`'s performance is bound at the expected point, the limit of the PCI bandwith, while `gm-myri` performs 36.5% better, sustaining 142MB/s vs. 104MB/s. The effects of interference on the shared memory bus are less pronounced, as expected, since the theoretical peak bandwidth of PCI is now half, thus a smaller portion of the available memory bandwidth.

5 Related work

The proposed framework for network block I/O has evolved from previous design approaches, as has been described in Section 2. Thus, the work both on TCP/IP-based nbd solutions, such as NBD, GNBD and NSD as well as on RDMA-based implementations is relevant to gmblock.

TCP/IP-based approaches are well-tested and highly portable on top of different interconnection technologies, as they rely on – almost ubiquitous – TCP/IP support. 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, they cannot explot the advanced characteristics of modern cluster interconnects, e.g. RDMA, since there is no easy way to map such functions to the programming semantics of TCP/IP, thus achieving low I/O bandwidth and having high latency.

RDMA-based implementations [8, 5, 7] 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 on Off-Processor I/O with Myrinet (OPIOM) [3] is very similar in spirit with gmblock and has similar goals. At the server side, OPIOM performs 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, it is SCSI-specific and can only be used with a single SCSI disk. Second, there is no provision for coherence with the page cache, thus no remote write support is possible. Even so, it is unclear what would happen if a remote node requested data recently changed by a local process, still kept in the page cache.

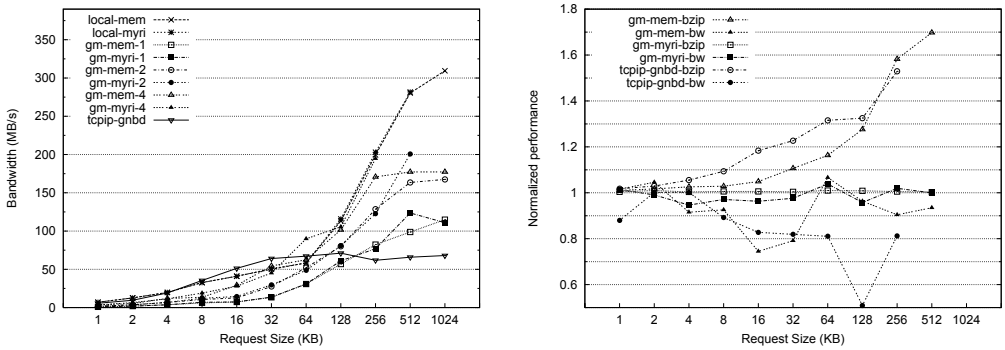
A different approach for bringing storage media closer to the cluster interconnect is READ^2 [2]. In READ^2, the whole of the storage controller driver resides on the Lanai processor itself, rather than in the Linux kernel. Whenever a request arrives from the network, it is processed by the Lanai, which 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, for each different block device, its driver needs to be rewritten to run on the limited resources of the Myrinet NIC, which provides none of the hardware abstraction layers of the Linux kernel. Second, the shared storage medium is not accessible by the host, since it is controlled directly by the Lanai. Finally, the Lanai can only access the storage device through DMA transactions from/to its PCI space, thus even very small operations have very high latency.

6 Conclusions - Future work

We have described and evaluated experimentally gmblock, an nbd system over Myrinet which uses an optimized disk-to-NIC data path. The data path is constructed combining the user level networking characteristics of Myrinet with Linux's VM mechanism, without compromising the security and isolation features of the OS.

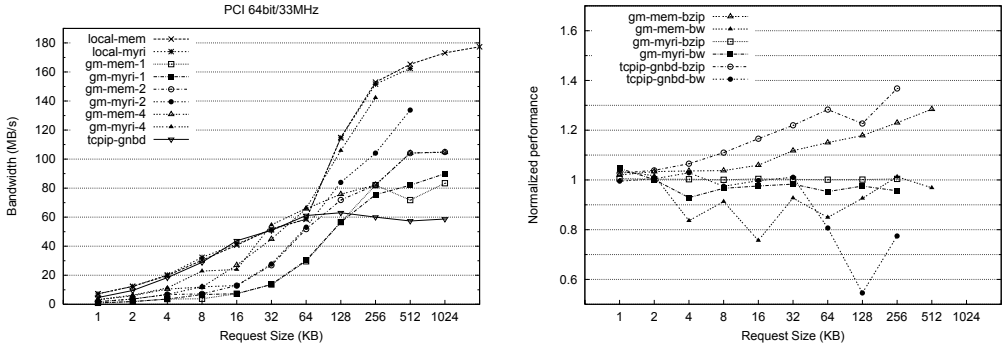
We believe this approach is generic enough to be applicable to different types of devices. An example would be a system for direct streaming of data coming from a video grabber to structured, filesystem based storage. Moreover our approach lends itself to resource virtualization: each user may make use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

In the future, we plan to evaluate the performance of gmblock on higher-end systems, equipped with PCI-X/PCI Express buses and more capable storage devices. We also plan to integrate it as a storage substrate in an existing GPFS/NSD installation, in order to examine its interaction with the parallel filesystem layer.



(a) Sustained bandwidth (b) Interference on the memory bus

Figure 8. Experimental evaluation on a PCI 64bit/66MHz bus



(a) Sustained bandwidth (b) Interference on the memory bus

Figure 9. Experimental evaluation on a PCI 64bit/33MHz bus

References

[1] 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.

[2] 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.

[3] P. Geoffray. OPIOM: Off-Processor I/O with Myrinet. *Future Gener. Comput. Syst.*, 18(4):491–499, 2002.

[4] H. Hellwagner. The SCI Standard and Applications of SCI. In H. Hellwagner and A. Reinefield, editors, *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*, pages 3–34. Springer-Verlag, Sep 1999.

[5] K. Kim, J.-S. Kim, and S.-I. Jung. GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[6] E. Koukis and N. Koziris. Memory Bandwidth Aware Scheduling for SMP Cluster Nodes. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 187–196, 2005.

[7] 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)*.

[8] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most Out of Direct-Access Network Attached Storage. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 189–202, Berkeley, CA, USA, 2003. USENIX Association.

[9] 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.

[10] 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.