Synchronized Send Operations for Efficient Streaming Block I/O over Myrinet

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

Abstract

Providing scalable clustered storage in a cost-effective way depends on the availability of an efficient network block device (nbd) layer. We study the performance of gmblock, an nbd server over Myrinet utilizing a direct disk-to-NIC data path which bypasses the CPU and main memory bus. To overcome the architectural limitation of a low number of outstanding requests, we focus on overlapping read and network I/O for a single request, in order to improve throughput. To this end, we introduce the concept of synchronized send operations and present an implementation on Myrinet/GM, based on custom modifications to the NIC firmware and associated userspace library. Compared to a network block sharing system over standard GM and the base version of gmblock, our enhanced implementation supporting synchronized sends delivers 81% and 44% higher throughput for streaming block I/O, respectively.

1 Introduction

Building parallel platforms out of commodity SMP nodes interconnected over a high bandwidth, low latency cluster interconnect, such as Myrinet [1], Quadrics or Infiniband has become an attractive option for providing a high performance computing infrastructure in a cost-efficient manner.

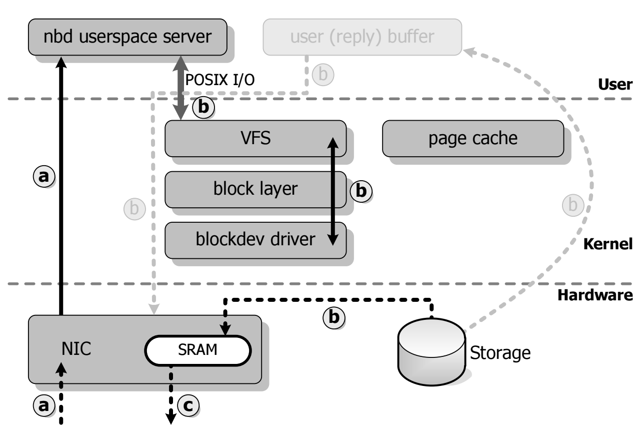
Increased computational capacity poses greater demands on the storage subsystem, which is commonly implemented either via raw access to a shared disk pool (as does the Oracle RAC cluster database, for example), or via a parallel filesystem, such as IBM's General Parallel File System (GPFS, [10]) or Red Hat's Global File System (GFS, [9]). The cornerstone of their deployment is shared, equivalent access to disks using Direct-Attached Storage protocols, for example SCSI. Traditionally, shared access to storage was made possible using a Fibre-Channel based Storage Area Network.

Recently, there is a significant trend towards networked storage, mainly for reasons of cost-efficiency (no need to maintain two distinct networks, one for storage, one for message-passing), scalability and redundancy. Cluster storage evolves to a configuration where a limited number of nodes (storage nodes) have direct access to disks over an SAN, but use a network block device sharing layer to export them for use by the rest of the cluster. Eventually, they may be eliminated altogether, with all of the cluster nodes contributing inexpensive, locally-attached storage to form a virtual shared disk pool.

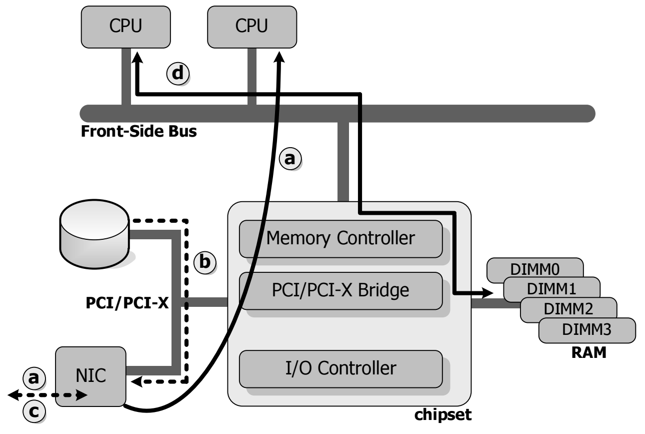
Thus, the performance of the I/O subsystem depends on the availability of an efficient network block device ("nbd") layer. Previous work [5] focused on implementing such layer over Myrinet. Our approach, called gmblock, utilizes an efficient data path on the server side, moving data directly between the storage medium and the Myrinet NIC, without requiring any involvement of the host CPU or intermediate copies of data in RAM. Its operation is block device driver agnostic, based on combining extensions to the VM infrastucture and direct I/O subsystem of the Linux kernel with custom modifications to Myrinet's GM. An initial performance evaluation of gmblock in [5] ("base implementation") showed it outperformed prevalent TCP/IP-based nbd systems and a GM-based implementation by a large margin. Still, the attained performance was lower than that expected based on disk and network throughput.

With this as a starting point, we perform a comprehensive evaluation of each component on the server side of the nbd system individually, and find that performance is mainly limited by latencies caused by various parts of request processing combined with the low number of outstanding requests possible, due to architectural limitations

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



(a) logical level



(b) physical level

Figure 1. Data path implemented by gmblock's server

of the Myrinet NIC and GM.

To mitigate the effect of latency for reads, we overlap disk I/O with network I/O for a single request. To achieve this, we propose the concept of *synchronized* network send operations. Their semantics enable the storage subsystem and the NIC to coordinate a peer-to-peer data transfer over the peripheral bus, without any involvement of the host CPU at all; the CPU onboard the NIC is responsible for retrieving packet data and injecting them into the network in a controlled manner, as the transfer progresses. We propose hardware changes to the Myrinet NIC in order to facilitate their implementation, and present a working prototype based on custom modifications to the GM library and firmware executing on the Lanai processor. A performance evaluation of our nbd system supporting synchronized network operations shows significant improvement compared to a Myrinet/GM-based system and the base implementation of gmblock, in terms of sustained throughput for streaming I/O.

This paper is organized as follows: we first provide a short introduction to Myrinet and gmblock (Section 2), then demonstrate the performance constraints we are targeting (Section 3) and introduce synchronized send operations (Section 4) as a solution. Section 5 describes an experimental evaluation of our approach. Finally, Section 6 presents research related to this work, while Section 7 summarizes our conclusions and explores possible future directions.

2 Myrinet/GM and the gmblock server

This section contains a short introduction to Myrinet/GM and how it is extended to support gmblock's data path. Myrinet NICs feature a RISC microprocessor, called the Lanai, which undertakes almost all network protocol processing, 2-4MB of SRAM for use by the Lanai and a number of DMA engines for access to PCI-X and the packet interfaces. Myricom's GM system provides user level networking facilities to applications. It comprises firmware executing on the Lanai, an OS kernel module and a userspace library.

A userspace application can map parts of Lanai SRAM in order to communicate with the firmware directly and issue message-passing requests. Sending a message using GM is 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.

Gmblock differs from TCP/IP and GM-based approaches, which stage data in host RAM, in that it builds a direct disk-to-NIC data path without any storage device driver-specific modifications, as follows (fig. 1): **(1)** Part of Lanai SRAM is reserved for use as GM message buffers and is mapped to the user's VM space by the GM kernel module **(2)** GM is extended to support sends directly from Lanai SRAM, bypassing the Host-to-Lanai DMA stage mentioned above **(3)** The Linux kernel VM mechanism is extended to support direct I/O from and to PCI physical addresses **(4)** The gmblock server component issues a direct I/O `read()` request, causing the storage medium to DMA block data directly to the NIC.

Thus, the steps needed to service a request when utilizing a gmblock-based nbd server (denoted by letters in fig. 1) are: **(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 SRAM on the Myrinet NIC, part of which has already been mapped in its VM space **(c)** The data are transmitted to the node that initiated the operation. Performing direct disk-to-NIC transfers minimizes the interference of block transfers with computation **(d)** which is likely to be taking place on other processors of an SMP node at the same time.

3 Motivation

In this section we perform a comprehensive performance evaluation of the base implementation of gmblock. We show that despite its use of an optimized data path, it delivers performance that lags that of the hardware units it comprises, i.e. the storage controller, the peripheral bus and the Myrinet NIC, when a low number of outstanding requests (an architectural limit imposed by limited memory on the Myrinet NIC, as explained in section 3.3) is used. We first examine each of its components in isolation, in order to deduce the constraints it imposes on overall performance, then analyze the sources of request processing latency on the nbd server.

3.1 Experimental Platform

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 features the Serverworks ServerSet III HC-SL chipset, with a Broadcom CIOB20 PCI bridge to two PCI segments: one 64bit/66MHz/3.3V with two slots and one 64bit/33MHz/5V with five slots.

The storage medium to be shared over Myrinet is provided by a 3Ware 9550SXU-16 SATA RAID controller on a 64bit/133MHz PCI-X adapter. We built a hardware RAID0 array out of 8 Western Digital WD2500JS 250GB SATA II disks, exported as a single drive to the host OS. The RAID0 chunk size defaulted to 64KB. The nodes are connected back-to-back with two Myrinet M3F-PCIXE-2 NICs, each in a 64bit/66MHz PCI slot. The NICs use the Lanai2XP@333MHz processor with 2MB of SRAM. Linux kernel 2.6.16.5, GM-2.1.26 and 3Ware driver version 2.26.02.008 are used. In the following, 1MB $= 2^{20}$ bytes.

3.2 Performance evaluation of the base gmblock implementation

We include two Myrinet-based implementations in the initial evaluation: one is gmblock running with standard GM buffers in host RAM (`gmblock-ram`), the other is gmblock running with buffers in Lanai SRAM, using the optimized data path (`gmblock-sram`).

We start by measuring the read bandwidth delivered by the RAID controller, locally. This is done performing back-to-back requests of fixed size, in `O_DIRECT` mode, with request size in the range of $1, 2, \ldots, 1024KB$. Two different runs are done, one with destination buffers in RAM (`local-ram`), the other with buffers on the NIC (`local-sram`), providing the two bandwidth vs. request size curves of fig. 2. 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 throughput increases significantly for request sizes after 256KB-512KB (reaching 347MB/s for buffers in RAM, 388MB/s for buffers in SRAM, with 1MB request size), while performance is suboptimal for lower sizes, since the degree of parallelism achieved with RAID0 is lower and execution is dominated by overheads in the kernel's I/O subsystem. Thus, optimizing our system for use with larger request sizes is key to delivering good performance. Second, we see that doing peer-to-peer bursts over the PCI bus from storage to the NIC outperforms the storage-to-memory path by a margin of 11.8%, since they happen at the full rate of the bus without the intervention of the CIOB20 bridge.

The second component of our nbd system is the PCI bus itself. The results of a firmware benchmark on the Lanai show that its PCIDMA engine can do burst writes to RAM at 378MB/s and reads at 331MB/s.

Fig. 2 shows the throughput measured at the client side, using a userspace client submitting back-to-back requests of variable size. The `gmblock-{ram,sram}` configurations achieve 168MB/s and 211MB/s for 1024KB-sized requests, respectively, which is only 48% and 54% of the maximum read bandwidth of the RAID subsystem. To verify that the network delivers the expected bandwidth, we introduce a configuration that omits the actual disk I/O operation, returning garbage to the client: the `gmblock-bogus-{ram, sram}` curves, corresponding to buffers in host RAM and Lanai SRAM, respectively. The former exhibits maximum throughput of 330MB/s, capped by the read PCI bandwidth. The latter on the other hand, is only capped by the total bandwidth of the two fiber links on the Myrinet NIC (476.8MB/s theoretical) and reaches 462MB/s.

Since `gmblock-ram` has to cross the main memory bus, its performance is based on the available PCI and network bandwidth, capped at $\frac{1}{\frac{1}{347}+\frac{1}{331}} = 169$MB/s. Even with multiple outstanding requests, this configuration could never exceed half the rate of the PCI bus, $\sim$200MB/s. On the other hand, `gmblock-sram` reaches 211MB/s, which agrees with $\frac{1}{\frac{1}{388}+\frac{1}{462}}$ MB/s.

To break the total request processing time into distinct phases, the server monitors the state of each request as it

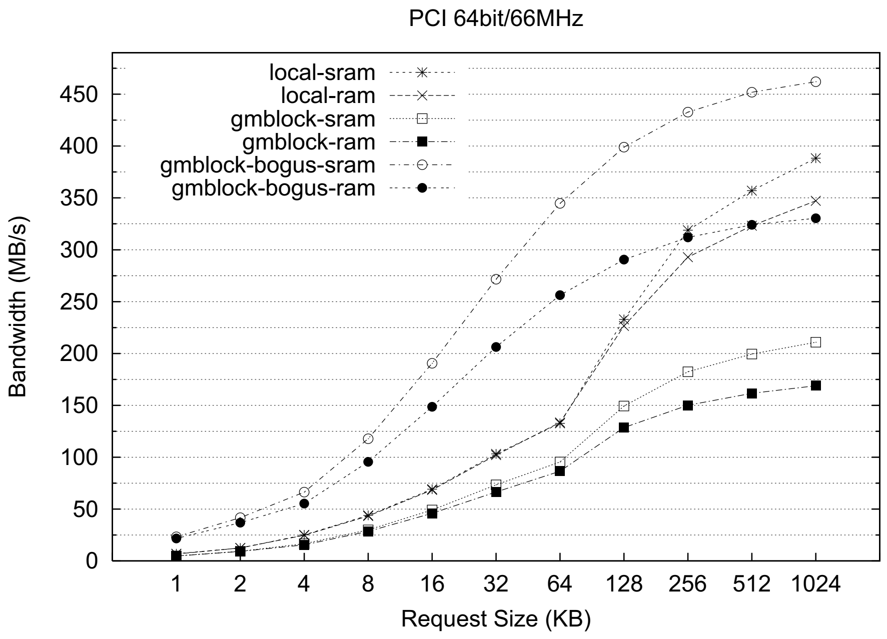


Figure 2. Sustained throughput as a function of request size (base implementation)

makes progress, using counters based on the TSC register of the Pentium. The results are presented in fig. 3. We have identified five different states each request may be in:

- `STATE_INIT`: Request received, being unpacked
- `STATE_READ`: In disk I/O phase
- `STATE_SEND_INIT`: Posting send event to the Lanai
- `STATE_SEND`: Disk I/O done, send operation in progress
- `STATE_FIN`: Returning receive token to GM

The time spent on states other than `STATE_READ` or `STATE_SEND` was found to be negligible. For smaller requests, execution time is dominated by disk I/O, since our storage delivers far lower bandwidth in the range of 20-30MB/s than for 512KB-1MB requests.

3.3 Discussion

The results presented in the previous section show that sustaining high performance for streaming I/O in an nbd system is challenging, even when utilizing an efficient, disk-to-NIC direct I/O path. The chief reason for low efficiency is idle periods and suboptimal utilization of resources. One way to attack this problem is by having a large number of requests on the fly, in various stages of processing. If the request pipeline is deep enough, the components of the nbd system are kept busy, working on distinct requests in parallel.

However, the Myrinet NIC has limited amount of SRAM (2MB to 4MB). Since gmblock moves data directly from disk to NIC, the maximum number of outstanding requests is limited by the amount of SRAM available for its use. We disabled most of GM's functionality apart from reliable send/receives (e.g. Ethernet emulation) and limited the maximum network size supported, but still no more than ~1MB was available for gmblock's use. Thus, only $1 \times 1MB$ request or $2 \times 512KB$ or $4 \times 256KB$ requests may be in flight at any moment. Using a larger number of smaller requests would improve pipelining but would move us at a lower point on the request size vs. bandwidth curve of fig. 2. So, to maintain sufficiently good performance by the storage subsystem, we are limited to one or two outstanding requests, in which case `gmblock-sram` fails to utilize the full link bandwidth, as shown in [5].

Having a limited number of outstanding requests means the attainable throughput depends on the imposed request latency; in fact, if only one outstanding request is allowed, the achieved transfer rate is $r_{net} = \frac{l}{t_{total}}$, where t_{total} is the total amount of time required to service a block request and l is its length. Thus, to improve the efficiency of our nbd system, we need to reduce t_{total}. Reducing total request latency can also be beneficial when there are data dependencies among them.

This work attempts to attack the problem of *read* request latency directly by increasing the amount of overlapped processing within each individual request itself. The results of the experimental evaluation presented in fig. 3 show that a

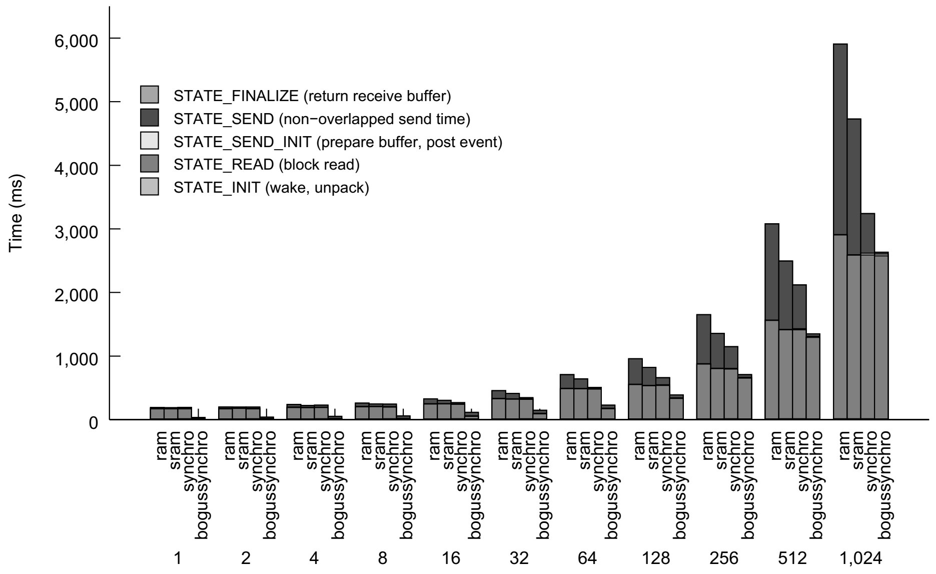


Figure 3. Latency breakdown per request size

significant fraction of the total request service time is spent waiting for network I/O (t_{send}). Servicing a block read request is a two-phase process (Storage-to-SRAM, then SRAM-to-wire); during the former phase the network interface is idle, while during the latter phase the storage subsystem is idle. Request service time would be reduced significantly if these two phases could overlap, i.e. by streaming data from disk, through Lanai SRAM, to the wire (fig. 4).

For this to be possible, the send from SRAM operation needs to be *synchronized* with the block read operation, in order 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.

Implementing this mechanism in software implies splitting up each request of l bytes (e.g. 1MB) in much smaller chunks of c bytes (e.g. 4KB) which would be then submitted simultaneously (via a facility such as POSIX (Asynchronous I/O) to the local I/O scheduler. The server would receive individual completion notifications and would invoke individual GM send calls. This approach has a number of significant drawbacks; first, it incurs the overhead of waking up the server and enqueueing GM send events very frequently; second, it discards the information that all chunks are contiguous, relying on the server-side I/O scheduler to reassemble them into to a larger I/O request to the storage medium, for efficiency.

4 Synchronized send operations

4.1 Design

What is needed is 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 is likely to fail, since the correctness of the data being transmitted essentially depends on the storage medium being able to deliver data faster than the Send DMA engine on the NIC consumes them. Of course, the storage medium may fail to keep up for a variety of reasons, such as transient disk errors, being used by applications other than the nbd server at the same time, or simply because it cannot deliver enough bandwidth to saturate the network link(s).

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. If at some point in time no valid data are available for a synchronized send token, the firmware ignores it when searching for a suitable token from which to enqueue a packet to the network. Essentially, what happens is that the NIC works in lockstep with an external agent (the block device), throttling its send rate in order 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. 4); The NIC only starts sending after $t_1 = \frac{c}{r_{disk}}$ time units, then both the storage device are 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. There is a trade-off involved, since smaller values mean finer-grained synchronization and better overlapping, but could impose significant CPU overhead on the Lanai, while bigger values lead to lower synchronization overhead but reduce the overlapping between the two phases.

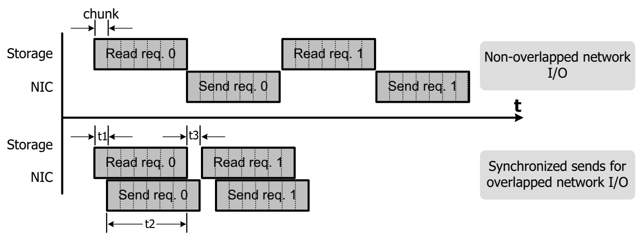


Figure 4. Intra-request phase overlap

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 CPU involvement.

4.2 Implementation issues on Myrinet/GM

There is one major implementation-specific point which has not yet been addressed. We need a way for the Lanai to be notified as an external agent places data into its SRAM. However, the Myrinet NIC does not provide such functionality in hardware. It could be implemented with a "dirty memory" bitmap describing the state of the Lanai SRAM (2MBs) divided into chunks of size c bytes. The bit corresponding to an SRAM chunk is set by the hardware whenever a value is written anywhere into it. For a value of $c = 4096$ bytes, at most 64 bytes are needed. The firmware initializes all bits corresponding to an SRAM buffer involved in a synchronized operation to zero. Then, it can verify chunk n contains valid packet data by checking if the bit for chunk $n + 1$ is set, assuming that the external agent performs DMA into the SRAM serially, in ascending order of addresses.

Since such functionality was not available on our NICs, we emulated it in software, with 32-bit markers in the SRAM itself. The Lanai polls the markers, which get overwritten as the data are DMAed in. 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. For instance, if $l = 1MB$ and $c = 4KB$, it holds: $P = 1 - \left(1 - 2^{-32}\right)^{\lceil \frac{l}{c} \rceil} \implies P = 5.96 \times 10^{-8}$.

Still, to ensure that the system never fails and the Lanai does not loop infinitely around a marker, an extra one is used right after the end of the block, which is set by the nbd server application when the data transfer into SRAM is complete and all of the data is valid. The worst-case scenario is that with probability P, no overlapping takes place and the network transfer starts after the block read operation is complete.

The implementation of synchronized operations on Myrinet/GM comprises three phases:

- *Initialization phase:*
  Function `gm_synchro_prepare_buffer()` writes a 32-bit value, `GM_SYNCHRO_MAGIC`, aligned with the end of each chunk, once every `GM_SYNCHRO_MTU` bytes in the SRAM buffer. Initialization is done with PIO inside the GM library, since the host CPU is an order of magnitude faster than the Lanai. Send events passed to the firmware are flagged as synchronized by `gm_synchro_send_with_callback()`.

- *Transmission phase:*
  This step lies in the critical path of transmission and is executed whenever a new packet of at most `GM_MTU = 4096` bytes is about to be injected into the network. To minimize the overhead of synchronization, each synchronized send token features a counter which holds the number of bytes known to be valid, initialized to zero. Whenever it is zero, the Lanai checks how many of the markers immediately after the send pointer have been overwritten and updates the counter appropriately. If it is nonzero then data may be consumed, and its value decreased otherwise the token is bypassed. The firmware never blocks on a synchronized operation, ensuring fairness between tokens. By setting `GM_SYNCHRO_MTU` to a multiple of `GM_MTU`, the normal send path is taken most of the time and the firmware has to stall only once every $\frac{GM\_SYNCHRO\_MTU}{GM\_MTU}$ packets.

- *Finalization phase:*
  The nbd server notifies the GM firmware that all of the data are valid by calling `gm_synchro_finalize_buffer()`, which sets the marker right after the end of the block buffer to a magic value.

The Lanai cannot address host memory directly but only through DMA. The cost of programming the PCIDMA

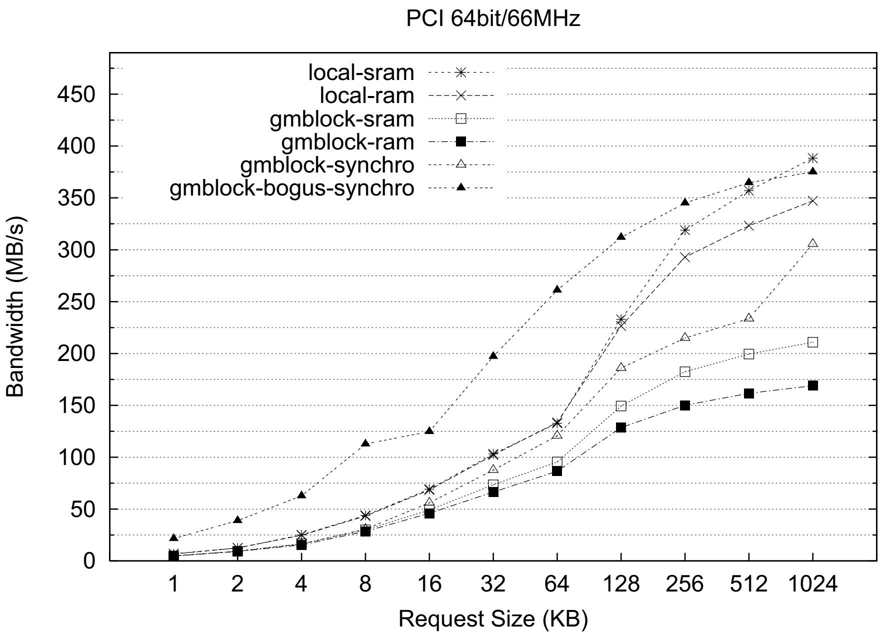


Figure 5. Sustained throughput as a function of request size (synchronized sends)

engine in order 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. This suffices for implementing an optimized version of gmblock's data path, however.

5 Experimental evaluation

We repeated the experiments of section 3.2 for a gmblock server incorporating synchronized sends and the results are displayed in fig. 5, for a chunk size of $c = 16KB$. There is significant improvement (305MB/s, 44% better than `gmblock-sram`, 81% better than `gmblock-ram`), with performance reaching 79% of the maximum RAID read bandwidth. However, there is still network I/O time that is not overlapped with disk I/O, as shown in fig. 3. This time is much larger than the expected non-overlappable duration ($t1 + t3$ from fig. 4); we attribute this to the initial RAID chunk-sized block being DMAed by just one disk.

To verify this hypothesis, we use a relaxed configuration, `gmblock-bogussynchro`, which omits the actual disk read. Instead, it simulates a storage device DMAing data at a constant rate (same as that expected from the RAID controller), by writing into the Lanai SRAM buffer. Now that the buffer is being filled at a constant rate from the beginning, gmblock achieves 375MB/s, with almost all of disk I/O overlapping with network I/O.

6 Related work

This work builds on gmblock, presented in [5]. TCP/IP-based approaches to building nbd systems, such as Red Hat's GNBD and the Network Shared Disk component of IBM's GPFS [10] are well-tested and highly portable but exhibit poor performance, perform multiple data copies per block transferred and have high CPU utilization. On the other hand, RDMA-based implementations [7, 4, 6] relieve the CPU from network protocol processing but still feature an unoptimized data path by staging data in main memory and crossing the peripheral bus twice per request. Thus, their performance would be comparable to that of `gmblock-ram`, since using 1-sided RDMA operations would not have any impact on the main performance-limiting factor, which is the path followed by data itself. Crossing main memory also leads to I/O interfering with the execution of memory-intensive applications on the host CPUs.

The work in [8] addresses the end-to-end performance of a kernel-based nbd system, over a custom 10Gbps RDMA-capable interconnect. It focuses on I/O protocol modifications for improving performance for a large number of small outstanding requests. This work focuses on mitigating the effect of server-side architectural bottlenecks for larger, streaming I/O requests. It aims to provide support for synchronized sends with as few changes as possible to the semantics of existing userlevel networking protocols (specifically Myrinet/GM). That is why gmblock has been implemented in userspace, based on userlevel mappings and standard POSIX system calls.

The work on Off-Processor I/O with Myrinet (OPIOM) [3] is very similar in spirit to gmblock and has similar goals, performing direct disk-to-Myrinet block transfers on the server side. However, it is SCSI-specific, relies on modifications to the SCSI stack of the Linux kernel and does not address the problem of coherence with the Linux page cache, which gmblock solves by exploiting the kernel's direct I/O mechanism.

READ^2 [2] brings storage closer to the network by having the storage controller driver residing on the Lanai processor itself. This however has limited real-world applicability, because it requires porting each individual driver to the limited environment of the Lanai and makes the storage device unavailable to the host.

7 Conclusions - Future work

Motivated by the discrepancy between the locally available storage bandwidth and the throughput attained by our prototype nbd server implementation, we enhanced user level send operations as provided by Myrinet/GM in order to support synchronization semantics. Our implementation enables the Myrinet NIC to coordinate in a peer-to-peer manner with an external agent, in our case a RAID storage controller without any device driver changes, bypassing the host CPU and OS running on top of it. This reduces the latency of individual requests by allowing the block read phase to overlap with the network I/O phase, leading to an 81% improvement in streaming block I/O throughput compared to a standard GM-based nbd implementation and a 44% improvement compared to our base implementation.

We believe the system would behave similarly on high-end hardware, such as PCI-X/PCI Express-based systems with DDR2 memory. We expect our conclusions would still hold, since performance when using an unoptimized data path would be limited by the available bandwidth to main memory; its effects would be even more pronounced given the emergence of modern 10Gbps cluster interconnects.

In the future, we plan to experiment with combining synchronized operations and multiple outstanding requests, using NICs equipped with 4MBs of RAM. Also, we plan to use a second Myrinet adapter with custom versions of the GM firmware and the GM module in order to emulate a high performance solid state storage device on a higher clocked PCI-X interface. Finally, we intend to study the behaviour of our nbd system with real application I/O patterns, by deploying GPFS on top of gmblock.

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] 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.

[5] E. Koukis and N. Koziris. Efficient Block Device Sharing over Myrinet with Memory Bypass. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*, page 29, 2007.

[6] 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)*.

[7] 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.

[8] 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.

[9] 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.

[10] 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.