On the Efficient Usage of High-Speed Networks for Distributed Storage in Clusters

Brice Goglin

Runtime – INRIA Futurs – LaBRI
351, cours de la Libération — F-33405 Talence cedex
France
Brice.Goglin@labri.fr

Abstract

Parallel applications running on clusters require both high-performance communications between nodes and efficient access to the storage system. We propose to improve the performance of distributed storage systems in clusters by efficiently using the underlying high-performance network to access remote storage systems. Storage and parallel computation have very different requirements. We show that it is required to modify the network programming interfaces and detail several proposals to make these interfaces interact more smoothly with distributed storage. Performance evaluations show that the integration of these ideas makes high-speed networks easy to use and very efficient in the context of storage.

Key words: Distributed storage, remote file access, cluster, high-speed network, zero-copy, unexpected messages, event notification.

1 Introduction

The increasing demand of performance in scientific computing, either for numerical simulation, particle physics, genomics, or virtual reality, requires a continuous improvement of the hardware. Massive super-computers have been replaced within the last decade by clusters of workstations which are less expensive, more generic and extensible. However, obtaining high-performance on such machines implies that the cost of communication between nodes is as low as possible.

A large amount of work has been put on first improving communication performance and secondly offloading its processing into the hardware. Nowadays, high-speed networks provide ultra-low latency (a cou-

ple micro-seconds) with a very high bandwidth (gigabytes per second). Since most usual protocols have troubles making the most out of this hardware [1], several dedicated programming interfaces have been designed, for instance BIP or MX on MYRINET, or VERBS on INFINIBAND. Message passing is nowadays the most often used programming model for parallel applications [2], especially through the MPI [3] interface. Since these softwares were implemented to efficiently support these specialized networks, communications between multiple instances of a single parallel application may easily benefit from very high performance.

Besides, high performance computing also requires an efficient access to storage. There have been multiple research projects to improve distributed file systems. First, they tried to make the client independent from the server by implementing complex caching et coherency maintenance strategies [4]. The emergence of large clusters then required the server to support a much more important workload, leading to specific research in this area [5]. Shared storage systems without server but with global synchronization were developed such as GFS [6] or XFS [7]. Then, client-server models as PVFS [8] or LUSTRE [9] were designed to support the workload of clusters by parallelizing the server and the storage system and distributing the work. However, communications in this case remain simple and do not benefit from the underlying high-speed network, even if the required bandwidth might be as high as for communications between processes in a parallel application.

Specializing network stacks to achieve best performance in parallel applications made them hard to use in other contexts where requirements are different. Therefore, it is necessary to study how to adapt these technologies to distributed storage, especially the integration of the programming model of high-speed networks in various implementations of storage. It first requires to look at data transfers between nodes, in particular a client and a server, and secondly communication control and management of the events they generate.

This article presents a study of an efficient usage of high-speed networks for distributed storage. The idea is to propose new optimizations that should be used together with existing caching and parallelizing strategies in order to improve performance by maximizing the usage of clusters' networks. We first present in Section 2 a modeling of remote storage access and locate possible optimizations. We then study in Section 3 the implementation of remote storage access within the LINUX kernel. We look both at *data*, where constraints on memory management are important, and *control*, where notifications of events and flow control matter. We propose several solutions to adapt existing programming interfaces of high-speed networks for distributed storage and evaluate their performance in Section 4.

2 Analysis of Interactions between Storage and Networking

To exhibit possible optimizations within remote storage access, we first present a model of these accesses and detail the various costs.

2.1 Modeling remote storage access

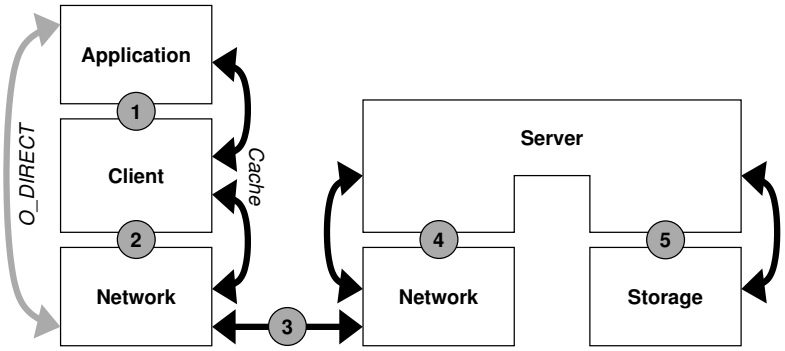


Fig. 1. Interactions involved when accessing remote files, numbered from 1 to 5. Black arrows are the regular data path while the grey arrow is an optional path that the application may choose to bypass the cache.

Figure 1 shows the steps during remote storage access. The whole access time T_{Total} has to be described as the sum of the cost of 4 steps involving the client, the network, the server, and the storage disks (Equation (1)).

$$T_{\text{Total}} = T_{\text{Client}} + T_{\text{Network}} + T_{\text{Server}} + T_{\text{Disk}} \tag{1}$$

To analyze these times more precisely, we introduce the following notations:

T_{Storage}	Time for the software to access to the local storage.
T_{Data}	Time spent in software to transfer data on the network.
T_{Control}	Time to manage and control network communications.

Times T_{Disk} and T_{Network} to access disks and the network are hardware specific (latency and bandwidth), but are fortunately well known nowadays. The time spent in the client or server, however, are software specific and depend on the implementation. On the client side, it corresponds to an access from the application to the client module managing storage (interaction #1 with cost T_{Storage1}), followed by initiating a network communication (interaction #2, which implies constraints regarding both control T_{Control2}

and data transfer T_{Data2}). On the server side, network communications have to be managed too (T_{Control4} and T_{Data4}) before an access to the local storage subsystem is done (interaction #5 with cost T_{Storage5}).

$$T_{\text{Client}} = T_{\text{Storage1}} + T_{\text{Control2}} + T_{\text{Data2}} \quad (2)$$

$$T_{\text{Server}} = T_{\text{Control4}} + T_{\text{Data4}} + T_{\text{Storage5}} \quad (3)$$

While hardware related times are fixed, software layers might allow some optimizations. Interaction #1 between the application and the client module accessing remote files has been the topic of lots of research. Some improvements have been proposed in the programming interface to suit the needs of parallel applications, especially regarding parallel, collective, vectorial or asynchronous access within MPI-IO [10]. Some specific features were added to improve the usage of the high-speed network in DAFS [11].

Time T_{Storage1} in the client remains low in case of a user-level implementation. In the kernel implementation, it might be much higher since several software layers have to be involved together with a copy of data in the cache. The overhead may be evaluated to about 15 μs for kilobytes accesses on a 2.6 GHz PENTIUM 4 XEON host. However, the actual cost may be reduced by the fact that the above cache generally avoids multiple accesses to the distant server.

On the server side, interaction #5 between the server process and its local storage subsystem has been studied a lot in the general context of high-performance storage. The corresponding time T_{Storage5} may thus vary a lot with the implementation.

In the meantime, the cost of using the network has been the target of multiple research projects in the context of distributed storage. We now describe how specific high-speed networks are and then emphasize the constraints they put on distributed storage within T_{Control} et T_{Data}.

2.2 High-speed networks specificities

The design of high-speed networks for clusters is based on both high-performance (few microseconds latency and gigabytes per second bandwidth) and highly specific innovations in the hardware and in the software stack. The need to achieve optimal performance for user-level communications between multiple instances of an MPI application led to the emergence of specificities and optimizations for this area: operating system bypass, memory copy avoidance, asynchronous communications and flow

control designed for homogeneous traffic.

2.2.1 OS-Bypass communications

Strongly coupled parallel applications often exchange lots of small messages between their processes. The overall performance is thus limited by the latency of the network. The critical path between the application and the network interface has to be as short as possible. It is achieved by avoiding the traversal of multiple software layers in the operating system (*OS-Bypass*) when posting requests and receiving events.

This optimization is very efficient. It enables latencies as low as a few microseconds. However, the design of the programming interface for this kind of user-level communications makes it hard to use in other contexts, especially distributed storage which is most of the time implemented at kernel-level. It will lead to high overhead T_{Control} to manage communications.

2.2.2 Zero-copy communications

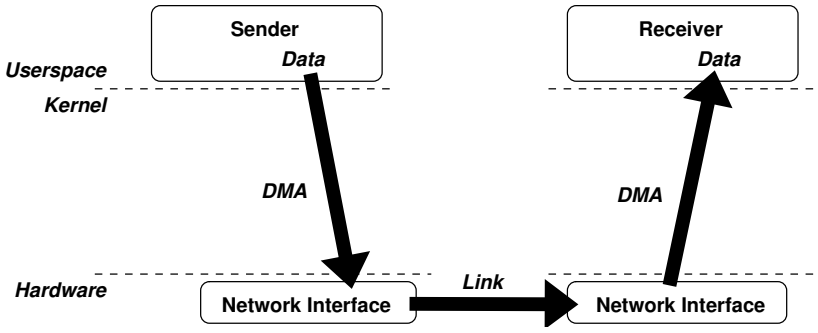


Fig. 2. Zero-copy OS-Bypass data transfer on high-speed networks.

To achieve maximum computation, it is necessary to keep the computing power available, even during communications. Copying data should thus be avoided since it consumes many CPU cycles. Nowadays, most high-speed networks are able to perform zero-copy communication between the network and user-space application buffers by using DMA (*Direct Memory Access*) initiated by the network interface (see Figure 2).

Combined with OS-bypass, zero-copy communication enabled high bandwidth and low latency communications with low CPU usage. It also raises the problem of translating the virtual addresses of the application buffers into physical addresses that the hardware may manipulate for DMA. Such a translation is usually performed by the operating system, but it is not possible in the case of OS-Bypass. High-speed network programming interfaces therefore rely on a strategy called *Memory Registration* which con-

sists in pinning memory zones and storing their physical addresses before communications [12]. However, memory registration implies an expensive advanced memory management in the application and proper preparation of memory zones during the initialization, which is not usually done in distributed storage systems. It will thus add a high overhead T_{Data} on data processing.

2.2.3 *Asynchronous programming interface*

Since the processor has to be as much available as possible for computation, parallel applications should not be blocked during communications. Processing of communications on high-speed networks is therefore offloaded into the network interface so that the host processor may still compute. The application submits asynchronous communication requests to the interface which takes care of processing them in the background. Computation may thus overlap communication until the application checks the completion of requests later.

This programming based on events is very different from the usual blocking model of the SOCKET interface. It is very important to deal with it cleverly so that communications are well overlapped. Managing this model may thus also be a large part of the communication control cost T_{Control}.

2.2.4 *Homogeneous traffic*

The network usage in parallel applications is specific to the explicitly regular distribution of the workload across the machines. Communications are most of time deterministic since the receive often knows in advance that it will receive a message before it actually receives it. Moreover, communications are distributed in a homogeneous manner across the network so that bottlenecks are avoided on the wires as in the processors. The network core is well enough dimensioned to prevent congestion in such a traffic. Control flow protocols in high-speed networks were designed for this usage, with strategies such as *stop-and-go* or *back-pressure*, without any notion of equity or fairness. This kind of protocol may not suit other models such as client-server in distributed storage, where the traffic is centralized near the servers. It will be a third part of communication control cost T_{Control}.

2.3 *Analysis of remote file access cost*

We now explain the impact of high-speed network specificities on various remote storage access implementations. User-level, kernel-level and block-

level accesses are presented.

2.3.1 User-level access

In the case of a user-level library providing access to remote files, the cost T_{Storage1} to access the client module of the distributed storage system is almost null (a function call). On the server side, most software layers will have to be traversed from the server process to the local storage, leading to a high T_{Storage5}.

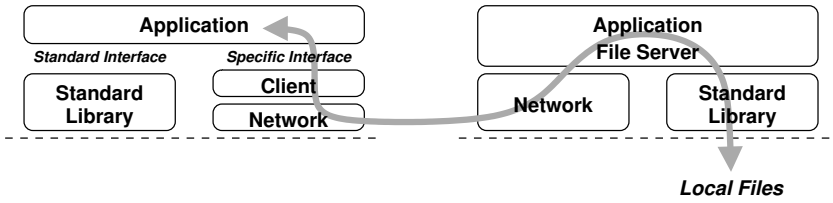


Fig. 3. User-level remote file access through a specific programming interface.

Regarding network access times T_{Control} and T_{Data}, they would be similar to the communication cost in a parallel application since the implementation is at user-level. (Figure 3). The time T_{Data1} to manage data transfer may be reduced by modifying the standard programming interface between the application and the client. Earlier work in DAFS [13] indeed shows that asking the application to provide information on its memory usage to help the network layer is a good idea. The application developer knows how he uses the memory and may thus organize and optimize it to make the network communications easier. Upto 25 % throughput improvement has been observed for database applications on DAFS during this study. Similar results were obtained with MPI-IO on DAFS [14]. In the same way, the application may be made responsible of managing communication control to reduce T_{Control}.

A specific programming interface allows remote file access optimization, but it requires to rewrite applications to respect the new interface. Such a re-development cost, especially if the interface is not portable, may be very high and could slow down software diffusion or standardization. This is the reason why several projects would rather *overload* standard library interfaces and manage memory registration and communication control transparently in the client module. We studied this model by developing a prototype named ORFA (Section 4.1).

Besides, some systems implement a cache in the client to reduce the need to contact the distant server every time the application accesses a file. The data transfer then involves the cache instead of the application memory. In this case, the storage client access time T_{Storage1} contains a memory copy between the application and the cache. However, the client may organize and

optimize its memory layout with respect to network constraints in order to reduce T_{Data}.

2.3.2 *Kernel server*

On the server side, accessing the storage may be optimized by moving the server process in the kernel so that several software layers are not traversed anymore and a memory copy is avoided (Figure 4).

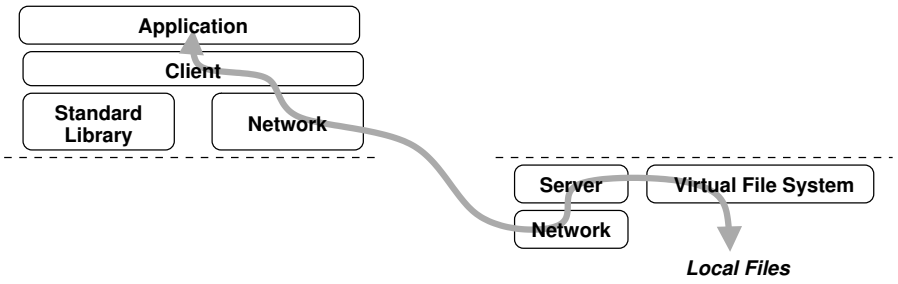


Fig. 4. User-level access to a kernel-level storage server.

However, high-speed network usage in the kernel first requires a kernel interface, which is not very common. We will present the design and implementation of the MYRINET MX kernel interface later in this article. Moreover, this interface has to be compatible with the user-space interface if it has to communicate with user-level clients. It is for instance not the case with QUADRICS QSNET [15].

Besides, memory zones that are manipulated by a kernel server actually are operating system pages where files are cached. Such pages are very specific (for instance they often do not have any virtual address). It makes them hard to use in the usual programming interface of the network, increasing T_{Data4} a lot.

2.3.3 *Kernel file-system client*

The most common strategy to access remote file is actually to place the client module in the kernel-space instead of in user-space. The access time T_{Storage1} becomes high since system calls are involved. All accesses to this file system are made transparent by the operating system which provides a virtualization layer for all file systems, local ones as well as remote ones (the *Virtual File System* in LINUX).

This model, as shown on Figure 5, also provides advanced features such as asynchronous, direct (bypassing the cache) or vectorial file access. It suits parallel applications requirements without having to modify the programming interface with a specific library. This is the reason why the most pop-

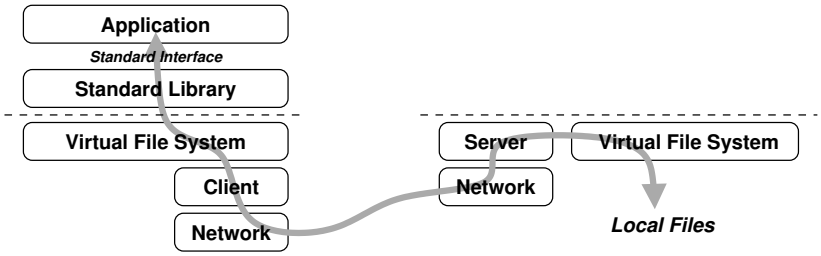


Fig. 5. Kernel-level remote file access.

ular distributed storage systems in clusters such as PVFS or LUSTRE are implemented in the kernel nowadays.

Communications in this model are initiated by the client module within the operating system. There are actually few high-speed network interfaces supporting this. Data transfer is as difficult as on the server side since the memory zones that are involved are those of the cache of the operating system.

Additionally, the application may ask the operating system to not cache any data on the client side (with the `O_DIRECT` parameter, as described by the grey arrow on Figure 1). Data then have to be transferred directly between the user-level application memory and the distant server. This kind of communication is conceptually very close to usual zero-copy communications on a high-speed network, but in our case they are initiated in the kernel instead of in the user application context. It is a very important requirement on the network programming interface to support such communications since this kind of memory usage is totally uncommon.

The usual programming model of high-speed networks is difficult to apply these two access schemes (with cache or without). The time T_{Data1} to transfer data may thus become very high.

2.3.4 Block access

The last kind of distant storage access (*Network Block Device*) consists in placing the client module at the block layer level and manipulating a remote partition or disk. Data are then exclusively transferred as blocks. This model is implemented in shared storage models for clusters such as GFS [6] or GPFS [16].

Some work has been proposed on the server side, especially in OPIOM [17] and READ [18], to optimize block transfers between the network and the disks by not going in the host anymore. However, very few similar work has been proposed on the client side.

These communications involve similar memory zones to the previous section. They have the same constraints on the network programming interface. It has to be able to manipulate special memory zones of the operating system. The costs are similar.

2.4 Summary of interactions between storage and networking

We explained in the previous section that remote storage access requirements may vary a lot with the actual implementation. The interaction between storage and network access software layers raises several difficulties.

We measured client module access times from the application (T_{Storage1}) varying from almost 0 (user-level client) to 50 μs (block-level client). On the server side (T_{Storage5}), similar timings may be observed depending on the implementation.

Data transfer cost on the client side is high and might be reduced by changing the programming interface to move memory registration problems into the application. A cache also permits to work around expensive memory registration constraints. In a kernel implementation, either in the client or in the server, the memory zones that are involved cannot be managed as usual user-space segments. It implies a high data managing cost T_{Data}, upto 100 μs for several kilobytes.

The time T_{Control} for using the asynchronous programming interface well however does not vary much with the implementation type. Most of time, several hundreds nano-seconds or a couple micro-seconds are spent.

Cost	Current value	Possible optimized value
T_{Storage}	upto 50 μs	determined by the implementation type
T_{Data}	1-100 μs	low if efficient usage
T_{Control}	100 ns-1 μs	very low if efficient usage

Table 1
Summary of software costs that are involved when accessing remote files, and possible improvements.

Table 1 summarizes various software times when accessing remote files. We showed that storage access time and some memory copies are determined by the design of the client or the server, especially if a cache is provided and whether in user or kernel-space. On the other hand, network access times, either regarding data (T_{Data}, because of memory registration) or control (T_{Control}, because of the asynchronous interface and the flow control)

remain in all models, with possibly different values. We will study these problems.

This idea is motivated by the fact that the most popular distributed file systems suffer from high-speed network interfaces not suiting their requirements. For instance, LUSTRE was until recently available on only few networks. Its first implementation used MYRINET/GM with additional memory copies since the software interface does not provide a flexible enough memory management. In the same way, PVFS2 [19] developers decided to not access the network from within the kernel client. Their implementation goes back to user-space where a dedicated process takes care of communications. This choice is justified in the documentation by the possible absence of support for network communications from within the kernel [20]. Our goal is to reduce T_{Data} and T_{Control} by providing a network programming interface that suits the needs of distributed storage.

We tried not to restrict our work to one of distributed storage systems described earlier. Choosing between a user, kernel, or block-level implementation, with or without cache, should remain the administrator or user decision depending on its needs and eventually on the machine architecture. Indeed, depending on how the target application has been programmed and behaves, the requirements regarding performance, caching or coherency might be different. Fortunately, our optimization should apply since we focus our work on the software interface between the distributed storage and network stacks.

3 Implementation of Remote File Access on Myrinet Networks

We now present a study of the problems that one may face while implementing a distributed storage system on a high-speed network. The overall goal is to reduce T_{Data} and T_{Control} by proposing modifications of existing systems, either network stacks or operating systems, to suit the requirements of distributed storage in clusters.

3.1 Hypotheses

We explained in Section 2.3.1 that the cost of using a high-speed network might be reduced by having the application taking care of the issues that are related to data transfer and communication control. This idea, which has been studied in DAFS, implies the modification of the file access programming interface that the application uses. However, such a requirement

leads to a complete rewrite of the application to deal with the constraints of the network. We chose to restrict ourself to the **standard file access interface**. It permits to apply our work to most existing legacy applications.

The distributed file systems we are interested in are **client-server** models, especially PVFS and LUSTRE, which are the most famous in the high performance computing world nowadays. These systems respect the standard file access interface and may be implemented in the kernel. Studying them will make us work the main types of implementations that we detailed in Section 2.3.

We use MYRICOM MYRINET network [21] which has been the mostly used high-speed network in high performance computing for a decade. Its software drivers are easily reprogrammable, making the implementation of our optimization possible. We first worked on the GM driver, whose programming interface behaves as specificly as explained in Section 2.2 and which represents well what old high-speed network interfaces are. We then proposed several solutions and applied them to the new MX driver of MYRINET networks, which was under development during our study.

3.2 Data transfer

As explained in Section 2.2.2, OS-bypass zero-copy communications provide high-performance communications for parallel applications. However, the memory zones that are involved in these communications have to be *prepared* using an expensive mechanism called **memory registration**. It is necessary to use this strategy in a clever way.

3.2.1 Memory registration

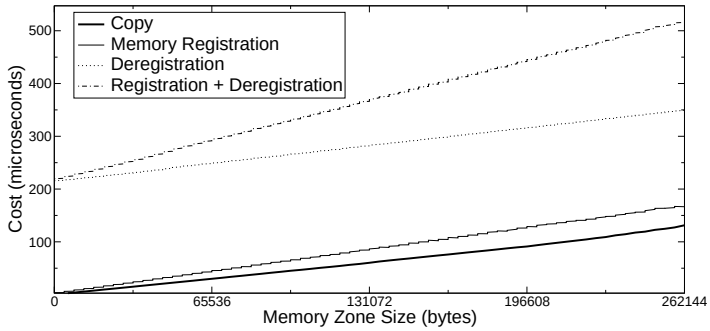


Fig. 6. Comparison between memory registration (with MYRINET-GM) and memory copy on a dual 2.6 GHz PENTIUM 4 XEON host.

Figure 6 presents a comparison of the cost of memory registration and memory copy. It shows that it might be a good idea to replace memory

registration with a copy in a statically pre-registered zone. However, this strategy consumes many CPU cycles and is thus not satisfying for parallel applications since the CPU has to be available as much as possible. A more interesting optimization named *Pin-down Cache* has been proposed in [22] to avoid the expensive cost of memory registration. It creates a registration cache by delaying memory de-registration as long as possible.

This strategy is known to be easy to implement and efficient in user-space parallel applications. It applies in the same way for distributed storage in user-space since the requirements are very similar as we explained in [23].

We now study the case of kernel-implemented file-systems, especially LUSTRE. We focus on the client side since it has the strongest requirements and constraints. Indeed, our decision to comply with the standard file access interface restricts our possibilities since we cannot forward the constraints of the high-speed network interface upto the application. We will detail in the next sections the different types of remote file access that an application may request: through the cache or directly.

3.2.2 Access through the operating system cache

Any regular file access goes through the cache in the operating system. It means that the application actually only deals with this cache, while the system takes care of maintaining the cache up-to-date by contacting the server in the background (Figure 7). The actual communications then involve memory zones of the operating system instead of user-level application buffers.

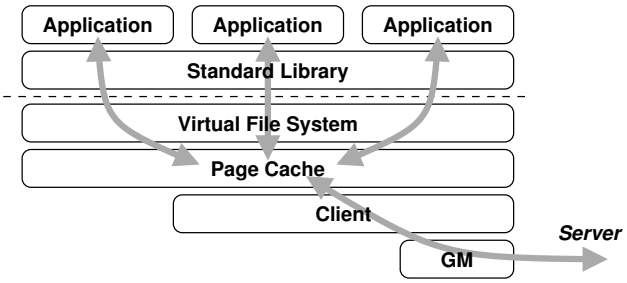


Fig. 7. Remote file access through a kernel client and the operating system cache.

There has been some research to formalize sharing and copying of memory pages between a file-system cache and a network software stack to define a clear semantics [24]. It led to some proposals towards a merge of the operating system file-system cache and the temporary buffers that are involved during network communications. It may help transferring data between the network and the file-system cache [25]. However, these systems remain based on memory registration and it looks too complicated because of the

execution within the kernel and the characteristics of the pages that are involved. Indeed, the pages of the operating system cache are very different from those of user-space application buffers. We showed in [23] that memory registration does not suit these pages at all.

Instead of adapting memory registration to these special pages, we proposed to modify the network programming interface in order to directly use the physical addresses of the cache memory. Such a strategy is actually very simple to use since no memory zone preparation is required anymore before communications: these pages are already locked and their physical address is known. We implemented this proposal in the GM interface of MYRINET networks. It required to modify the micro-program that runs in the hardware.

3.2.3 Direct access from the operating system

Accessing remote files through the operating system cache has the drawback of implying a memory copy between the application and the cache, which consumes many CPU cycles. Moreover, parallel applications often want to know that a write has actually been forwarded to the server instead of being kept in the local cache for an unknown delay. Recent operating systems therefore propose direct accesses bypassing the file-system cache. After studying the cached accessed, we worked on these direct accesses.

While memory registration does not suit cached accesses requirements, direct access looks similar to user-level communications in parallel applications since user-space memory zones are involved. The major difference is that communications are initiated from within the operating system. The network interface thus has to be able to manipulate simultaneously virtual addresses of various user-space processes from the kernel. It is a strong requirements since high-speed networks are used to associate any single communication channel to a single task. We had to modify the MYRINET board microprogram again so that the virtual addresses it manipulates also contain an identifier of the target address space [23].

Then, memory registration has to be efficiently used so that its observable cost appears to be low. The registration cache strategy that we described in Section 3.2.1 should suit direct file access needs. However, it requires the cache to be kept up-to-date with respect to modifications of the application address space. Indeed, since the application does not know that some of its pages have been registered on the fly by the underlying software layers, it could modify its address space without making the cache manager aware of it. If the cache is not invalidated when a memory zone becomes invalid, the network hardware could be using invalid address translations, possibly

causing serious problems to the system.

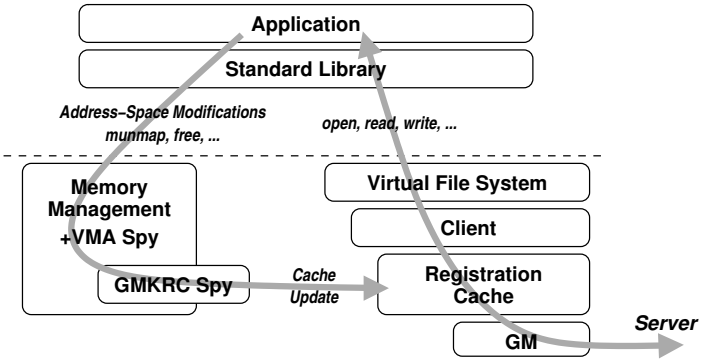


Fig. 8. Management of direct accesses within the operating system with a registration cache that the VMA SPY subsystem keeps up-to-date.

Keeping the cache up-to-date in a user-space implementation is a well-known problem. However, we showed that it is impossible in a kernel-level implementation because of missing software support. We proposed some modifications to the LINUX kernel to solve this problem. This patch, called VMA SPY, permitted the implementation of a safe registration cache between the GM driver and the remote file access client [23] (Figure 8).

3.2.4 Flexible memory addressing

We focussed on the client side and insisted on memory registration problems since this is the commonly used mechanism to implement zero-copy communications. We implemented various modifications in the GM driver of MYRINET networks and showed how distributed storage requirements may be hard to support with existing network interfaces. Indeed, making these interfaces that were designed for user-space communications in parallel application interact with the distributed storage software layers is hard. We will present an evaluation of our implementation in Section 4.

Apart from modifications in the operating system to maintain the registration cache up-to-date with respect to address space modifications, we identified three features that are required in the network interface:

- Manipulate user-space memory with communications that are initiated in the kernel;
- Share a single communication channel between several address spaces of several processes accessing remote files;
- Manipulate memory zones without having to register them when they have special characteristics such as those of a file-system cache.

We proposed in [26] a programming interface that provides all these key features. It is based on extended communication primitives that ask the user

to describe the memory addressing type of the buffers that are involved. Then, it is possible to manipulate with a uniform interface any buffer in user-space processes or in the file-system cache of the operating system.

We implemented this interface in the new generation driver of MYRINET networks, MX [**?**]. We tried to keep the core of MX generic so that user-space communications would not be favored and our extended primitives handicapped. We will present a performance evaluation of this work in Section 4.

3.3 Flow control

After presenting our study of data transfer related issues regarding distributed storage on high-speed networks, we now proceed with control issues, starting with flow control.

Generic networks and protocols were designed to enable communications between any machines in the world, without any requirements on the network topology and reliability. While it permits a high flexibility, it also implies a large overhead since we have to make multiple machines work together and react to various problems such as congestion or failure. High-performance computing clusters are based on a regular and static topology, with a well-dimensioned network backbone so that any regular traffic between end nodes does not suffer of any congestion. Transport protocols in lower network layers relies on the fact that the communication channel is reliable. Error recovery is however a rare and non-optimized case.

Parallel applications generally generate a uniform traffic across the cluster. Indeed, a good parallelization requires to distribute the workload across the machines with equity, but also to distribute the network traffic. This way, bottlenecks are avoided, regarding either the actual computing power or link capacity.

Distributed storage does not however generate such a traffic. Indeed, client-server models explicitly imply a centralized traffic around the server, creating a bottleneck. Parallel file systems have been designed to distribute the workload across multiple servers, distributing the network traffic across various links too. Even so, the system has to be dimensioned correctly since multiple clients may easily saturate any server link.

Figure 9 presents the throughput that our experimental file server may process when multiple clients simultaneously send 64 kB write requests on the GM interface of MYRINET networks. With 92 clients, 233 out of the 250 MB/s supported by the link are used. However, with 184 clients, only

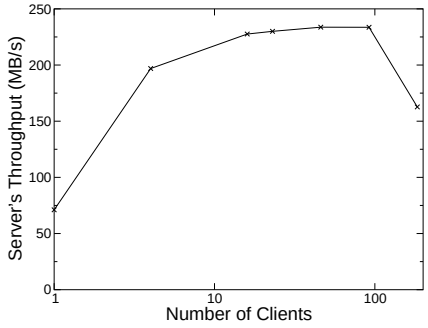


Fig. 9. Impact of the amount of clients on the effective throughput of a storage server, with 64 kB write requests and the GM interface of MYRINET networks.

163 MB/s appear to be processed by the server.

What causes this behavior is a large waste of link bandwidth due to re-send of non-received messages. In the *Rendezvous* model, the one offered by the GM interface and very often used in high-speed networks, the server has to prepare receive buffers. The number of buffers is limited by the available memory in the network interface (125 buffers in this case). If there are too many clients, some requests will not be received due to receive buffer starvation. They will have consumed some bandwidth without their data ever being used. The observable throughput of the server thus decreases with large numbers of clients, as shown on Figure 9.

Unexpected messages is a well-known problem in parallel application programming. For instance, it has been shown that the offloading features of network boards cannot always be enough to efficiently manage unexpected messages in MPI [27]. Solving this bandwidth wasting problem may be achieved by having the client wait for the server agreement before sending its data. No message would then be unexpected anymore. Regarding small messages, it is also possible to store them in a temporary buffer in receiver host memory until the application provides the corresponding actual receive buffer. This kind of protocol is nowadays often implemented in MPI layers, but not in most native low-level network interfaces.

Since implementing such a feature in the GM interface would require too many changes, we decided to adapt the application. We designed on top of the restricted low-level network interface a high-level application protocol to manage unexpected messages [28] in a client-server distributed storage model.

We proposed the combined use of *Rendezvous* and *Remote Direct Memory Access* (RDMA) models. By deporting data transfer decisions onto the server, it controls the incoming traffic to avoid any bottleneck. The server actually takes care of reading or writing data in the clients' memory when the required resources are available. Figure 10 describe our new client-server

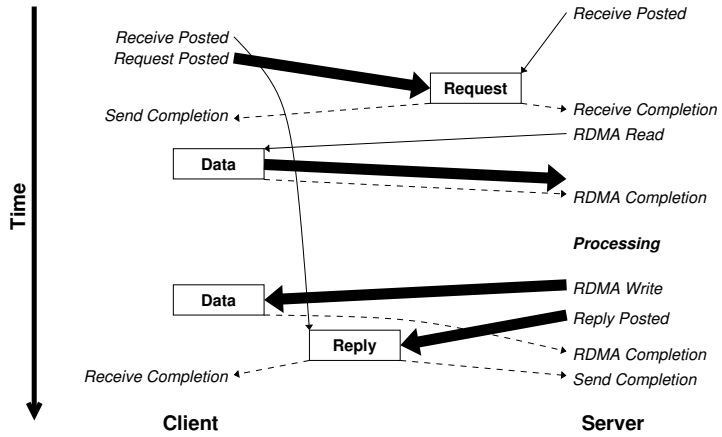


Fig. 10. Client-server protocol to access remote files when flow control is managed by the server. The client only sends a request while the server takes care of reading or writing in the client memory.

protocol to access remote files. This model is however restricted to network programming interfaces that provides both RDMA and *Rendezvous* communications, which is for instance not the case of INFINIBAND VERBS.

It is worth citing interesting results that were obtained by having the client initiate RDMA to the server in DAFS [29]. Assuming the server is exporting part of its file-system cache, there are chances that a client successfully writes or reads directly in the cache through RDMA. In case of a cache hit, the client does not require any server intervention anymore. In case of cache miss, a remote exception is sent to client, which reverts to a basic protocol with the server being the master.

3.4 Asynchronous programming interface

We explained in Section 2.2.3 that high-speed network programming is based on a very specific model. While the usual SOCKET interface offers blocking communications, high-speed network interface primitives are non-blocking and asynchronous. The application submits communication requests that the hardware processes in background while the application computes. Later, the request status may be checked and the application may block until some requests complete.

Using such a programming model makes it hard to interact between networking and storage because the needs of the storage subsystems may vary a lot. Indeed, we showed in [28] that the server generally waits for the completion on any pending request, while the client may want to wait on a single request.

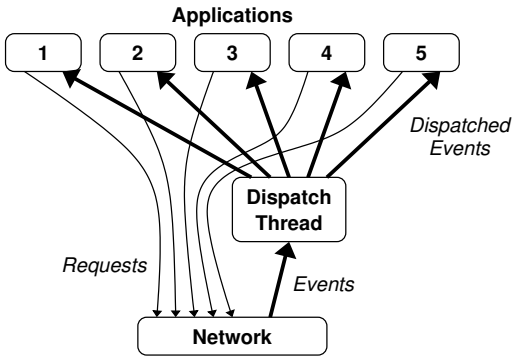


Fig. 11. Event dispatching with a thread.

Both these completion strategies are often used in MPI. However, most low-level network programming interfaces do not implement both. When only the former is implemented, as in the GM driver of MYRINET networks or in INFINIBAND VERBS, the latter may be implemented on top of it. To do so, a *Dispatch* thread takes care of receiving all completion events in a single queue and dispatching them to the corresponding task (Figure 11). Unfortunately, it increases the overall latency since there are much more context switches. We observed several microseconds of overhead in such an implementation.

When only the second strategy is available, as in the ELANLIB interface of QUADRICS networks, implementing the first strategy requires active polling on all pending requests since their completion order is often unknown. It consumes lots of CPU cycles.

Getting the distributed storage subsystem and the high-speed network stack to interact well requires both strategies to be available and efficiently implemented. This requirement actually also appears in MPI middlewares or applications.

Moreover, exploiting multiprocessor machines requires to distribute the workload across multiple processors. It is therefore common to share completion event notifications across several threads. One solution would be to distribute network events in multiple queues (*Completion Group*) and have one thread take care of each queue. This is a third kind of completion notification that is useful is distributed storage. While the first and second ones were already available in the MX driver of MYRINET networks, we had to implement the third one. It is available as `mx_test_any` and `mx_wait_any` in the recent released 1.2 version of MX and is now used in the recent port of LUSTRE and PVFS2 over MX.

We studied in this section various needs corresponding to times T_{Data} et T_{Control} involved when using high-speed networks for distributed storage. Regarding data transfer, memory management has a strong impact on performance and requires a very flexible network programming interface. We explained how to work around the limitations of existing interfaces and proposed an extension that we integrated in the MX driver of MYRINET networks.

Regarding communication control, we first presented an issue related to flow control in client-server implementations of distributed storage. We showed that unexpected messages have to be managed carefully, either in the network stack or in the application. Then, we described completion notification requirements in both storage clients and servers. We explained that it is interesting to have multiple strategies available natively in the network programming interface.

These needs for communication control are actually very similar to those of parallel applications running on MPI. It is thus a good idea to have the corresponding features in the low-level network interface so that both MPI and storage may rely on them. It is what the new MX driver of MYRINET networks now proposes. However, several other existing interfaces such as GM will still have to be used with a complex and less efficient high-level protocol at the application level.

4 Performance Evaluation

We now present an evaluation of the impact of our proposals on the high-speed network usage in distributed storage, especially regarding how easy and efficient it is to make them interact. We have implemented our ideas regarding both data transfer and communication control in the MX interface of MYRINET networks. This section will therefore compare the efficiency of MX in distributed storage to an old network programming interface, GM.

We actually had to first make MX programming interface available in the kernel so that in-kernel distributed storage modules may use MYRINET networks. Then, we had to extend its implementation to efficiently support various data transfer and memory addressing types as described earlier. When comparing with GM, we had to heavily modify both the GM programming interface and the operating system as explained in Sections 3.2.3 and 3.2.2 so that distributed storage requirements were met.

4.1 Experimentation platform

Measuring the performance of the distributed storage system is very difficult due to various advanced features such as caching or parallelization. To isolate the problem that we have been working on, the efficient usage of the underlying high-speed network, we developed a minimal prototype providing remote file access without any advanced feature. All non-required steps on the path from the application buffer to the server were therefore removed so that only the critical path was observed.

This prototype has been implemented both at user-level (ORFA, *Optimized Remote File-system Access* [30]) and in the kernel (ORFS, *Optimized Remote File System* [23]) so that various remote file access models may be evaluated. ORFA is based on the dynamic shared library exposing the standard file access interface and converting application requests into remote requests to a distant server over a MYRINET network. We showed in [23] that the registration cache gives good performance for this kind of remote file access. We now present a study of in-kernel performance with ORFS. It will allow the applying of our results in actual distributed storage systems such as PVFS or LINUX. ORFS is based on a kernel module that adds a new file-system type into LINUX.

All experimentations were run between dual 2.6 GHz PENTIUM 4 XEON hosts with 2 GB memory. They were interconnected with a MYRINET 2000 (PCIXD boards) using GM 2.0.13 and MX 0.8.8 drivers.

To illustrate remote file access over MX and GM, we published in [26] a comparison of raw performance of these interfaces. While MX bandwidth is not much higher than GM's one, its latency is much lower (4 μs against 8 μs). We also noticed that our implementation of in-kernel communications in MX offers as good performance as the regular user-level communications. It is worth noticing it in our study since most network programming interfaces, such as GM or ELANLIB, were initially designed to optimize user-level, making the in-kernel communications harder to optimize. For instance, GM latency in the kernel is 30% higher than in user-space.

4.2 Direct access to remote files

Figure 12 presents a study of direct remote file access from within the kernel over ORFS and GM. It shows that our modifications in GM and our VMA SPY infrastructure in the LINUX kernel (see Section 3.2.3) provides very good performance with an in-kernel registration cache. Fortunately, it means that the huge amount of work that has been required to support di-

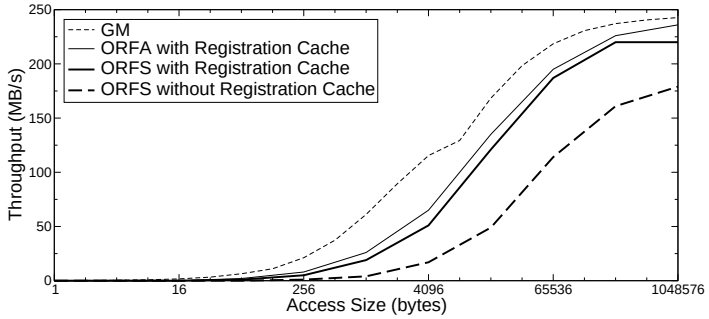
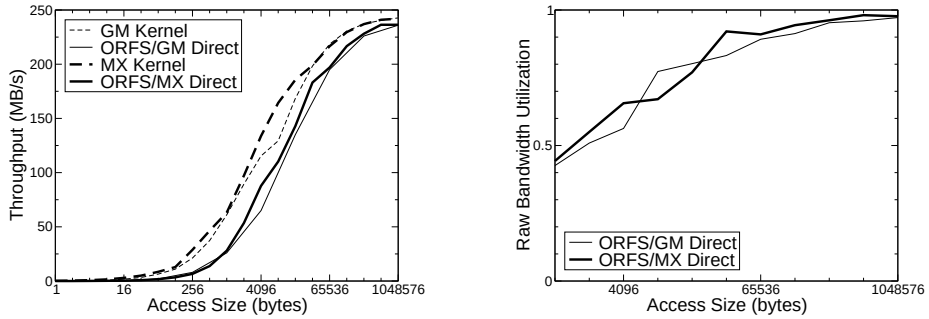


Fig. 12. Performance comparison between user-level (with ORFA) and kernel-level (ORFS) remote file access over GM.

rect access from within the kernel over GM has been useful. Performance is however still slightly lower than in user-space since the client has to traverse several software layers to reach the ORFS client in the operating system (T_{Storage1} is higher).



(a) Absolute throughput. (b) Relative throughput.

Fig. 13. Performance comparison of direct access to remote files within the kernel with MX and GM.

Figure 13 then compares direct remote file access over GM and MX. Even if raw MX performance is slightly higher than GM, ORFS performance shows that MX may be used even better than GM since the relative throughput is often higher. It has to be noted that GM performance depends a lot on the same memory zones being reused by the application to maximize hits in the registration cache. MX performance does not have such a requirement since no explicit memory registration is needed.

Apart from the performance, MX is actually much easier to use than GM in these implementations. While MX may be immediately used by ORFS, GM requires an external registration cache, some modifications of the kernel (VMA SPY) to keep this cache up-to-date, and some modification in the network programming interface and microprogram (see Section 3.2.3).

4.3 Access to remote file through the cache

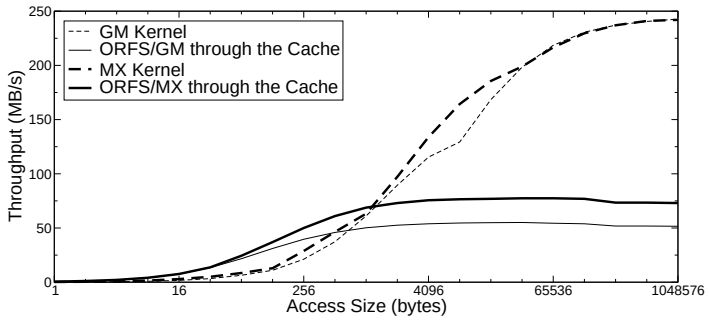


Fig. 14. Performance comparison of remote file access through the kernel cache with ORFS over MX and GM.

Figure 14 then presents the performance of remote file access through a file-system cache (Section 3.2.2). Having the cache on the path to the server leads to an explicit splitting of any request into page-sized accesses (4 kilobytes on our machines). Recent LINUX kernels enable multi-page access but it makes results harder to analyze from the network usage point of view.

Remote file access through the cache over MX have a 40% higher throughput than over GM. It corresponds to a 54 μs processing time per page on MX and 76 on GM. Within the 22 μs difference, we measured that 13 are caused by MX raw performance being better than GM. This is related to the GM implementation of kernel communications suffering from the user-space design, while our implementation of kernel communications in MX is generic enough to keep performance as high as possible. The remaining 9 μs improvement is caused by the avoidance (thanks to our flexible MX interface) of the complex management of ORFS communications over GM, especially event notification strategies that required a dispatched thread (as described in Section 3.4).

Measuring the impact of the new features that MX proposed remains hard since they mostly have an influence on the latency that the application observes. However, having a file-system cache hides the latency behind request delaying and aggregating. Nevertheless, we showed in [26] that our work led to a large improvement, especially in latency, in another application that has similar requirements than ORFS, a zero-copy SOCKET protocol.

4.4 Block access

Remote block access (*Network Block Device*, see Section 2.3.4) have very similar requirements to the file access through a cache that we presented above.

Thus, we expect this application to make the most out of our MX interface. We implemented a *Network Block Device* called MXBD using our MX programming interface in the LINUX kernel.

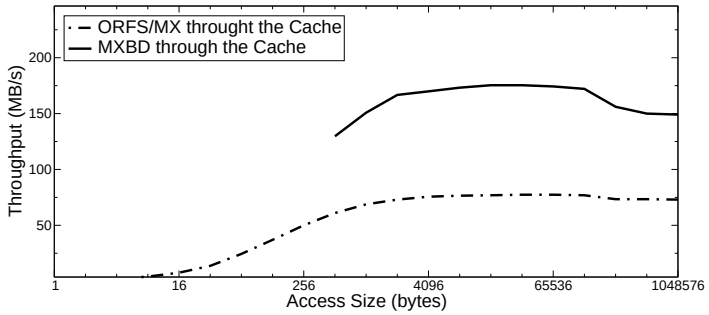


Fig. 15. Performance of remote block access (with MXBD) and remote file access through the cache (with ORFS) over MX.

Figure 15 presents a performance comparison between remote block access (MXBD) and file access through the cache (ORFS). The former model only transfers data blocks without any internal structure while the latter transfers file attributes and pages. The requirements over the network interface are very similar, but the block layer has been designed to deal with hardware storage controllers, which can handle asynchronous requests. Asynchronous communications are thus easier to implement efficiently and immediately pipelined. It leads to twice the throughput of file access through the cache. Moreover, when enabling vectorial communications with multiple non-contiguous segments, the whole link capacity may be used.

Combining such an implementation on the client side and a strategy such as OPIOM or READ on the server side should permit end-to-end optimized remote block access.

5 Conclusion

This article presents an analysis of the interaction between the operating system software layers managing the storage and high-speed networks in clusters. These networks and their application programming interfaces have been designed and optimized for user-space communications within parallel applications. While the performance that may be achieved for this kind of communication is very high, distributed storage performance in these computing systems cannot benefit from the network as well.

Previous works in distributed storage targeted caching and parallelizing strategies. In this paper, we propose an additional approach focussing on

improvements of distributed storage system performance by efficiently using the underlying high-speed network. By being so specific to user-space applications, these networks became hard to use in another context.

We presented a detailed analysis of various remote file access models, and we identified the limitations of current high-speed network programming interfaces in this area. We implemented multiple modifications in the GM interface of MYRINET networks and in the LINUX kernel to permit an efficient interaction during data transfer. Moreover, we showed that communication control in distributed storage systems has several requirements that are actually not very different from those of MPI middlewares and applications.

Several ideas were then proposed to make the use of new network interfaces easier for distributed storage. First, the needs of both distributed storage and MPI applications have to be met by the network stack by proposing both **multiple event notification strategies** and a **smart unexpected messages management**. Secondly, the in-kernel network interface has to be flexible enough to manipulate various **memory addressing types**. We implemented these ideas in the new MX driver of MYRINET networks. This work is now available within the official MX distribution.

We finally exposed a performance evaluation of this implementation through an optimized remote file access protocol. We focussed on point-to-point communication between a client and a server since this is where our work matters. Nevertheless, we expect our optimizations to work as well in multiple-nodes systems since we took care of not decreasing the scaling capabilities of existing systems in our implementation.

Raw latency from within the Kernel:
- **– GM:** 8 μs (6 in user-space).
- **+ MX:** 4 μs (4 in user-space).

Remote file access from the Kernel through the Cache:
- **– GM:** Microprogram recompiling to use physical addresses; Dispatcher thread.
- **+ MX:** 40 % bandwidth improvement over GM.

Direct remote file access from the Kernel:
- **– GM:** Expensive memory registration; Kernel patch for registration cache; Microprogram modification to support multiple address spaces; Dispatcher thread.
- **+ MX:** At least as good as GM; Easy to use.

Fig. 16. Summary of GM problems and MX improvements for distributed storage from within the kernel.

Figure 16 summarizes the results. Our work in MX is proved more efficient than traditional interfaces. Moreover, it has to be noted that its usage is

much simpler. The huge amount of modifications that we had to implement in GM proves that it does not suit distributed storage requirements.

Our work is now used in large production centers within the LUSTRE distributed file system. LUSTRE is able to achieve more than 90% of the link capacity on MYRI-10G networks [31]. Moreover, our results also apply to zero-copy SOCKET protocols as well.

The LINUX kernel modifications that we proposed to keep the registration cache up-to-date over GM are not required anymore on MX. However, they could still improve the performance even more. Lots of interesting support for high-speed network specificities in the LINUX kernel is still missing. We discussed this issue with kernel developers and QUADRICS and INFINIBAND software stack maintainers. While no consensus has been found yet, it seems clear that a support similar to VMA SPY (Section 3.2.3) should be included in the LINUX kernel in the future. Indeed, the emergence of 10 Gbit ETHERNET interfaces raises similar problems than those we faced in high-speed networks. We are planning to look at these issues in the near future.

References

[1] A. Barak, I. Gilderman, I. Metrik, Performance of the Communication Layers of TCP/IP with the Myrinet Gigabit LAN, Computer Communication 22 (11).

[2] R. Brightwell, A. Maccabe, Scalability limitations of VIA-based technologies in supporting MPI, in: Proceedings of the Fourth MPI Developer's and User's Conference, 2000.

[3] M. P. I. Forum, MPI: A message-passing interface standard, Tech. Rep. UT-CS-94-230 (1994).

[4] M. N. Nelson, B. B. Welch, J. K. Ousterhout, Caching in the Sprite Network File System, ACM Transactions on Computer Systems 6 (1).

[5] P. B. School, File Systems for Clusters from a Protocol Perspective, in: Second Extreme Linux Topics Workshop, 1999.

[6] S. R. Soltis, T. M. Ruwart, M. T. O'Keefe, The Global File System, in: Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems, IEEE Computer Society Press, College Park, MD, 1996, pp. 319–342.

[7] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, R. Y. Wang, Serverless network file systems, ACM Trans. Comput. Syst. 14 (1) (1996) 41–79.

[8] W. Ligon, R. Ross, An Overview of the Parallel Virtual File System, in: Proceedings of the 1999 Extreme Linux Workshop, 1999.

[9] P. Schwan, Lustre: Building a File System for 1,000 Node Clusters, in: Proceedings of 2003 Linux Symposium, Ottawa, Canada, 2003, pp. 401–408.

[10] R. Bordawekar, J. M. del Rosario, A. Choudhary, Design and Evaluation of primitives for Parallel I/O, in: Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing, ACM Press, New York, NY, USA, 1993, pp. 452–461.

[11] K. Magoutis, Design and Implementation of a Direct Access File system (DAFS) Kernel Server for FreeBSD, in: Proceedings of USENIX BSDCon 2002 Conference, San Francisco, CA, 2002.

[12] M. Welsh, A. Basu, T. von Eicken, Incorporating Memory Management into User-Level Network Interfaces, in: Proceedings of Hot Interconnects V, Stanford, 1997.

[13] M. Rangarajan, L. Iftode, Building a User-level Direct Access File System over Infiniband, in: 3rd Workshop on Novel Uses of System Area Networks (SAN-3), 2004.

[14] J. Wu, D. K. Panda, MPI-IO over DAFS over VIA: Implementation and Performance Evaluation, in: Workshop on Communication Architecture for Clusters (in conjunction with IPDPS), Fort Lauderdale, FL, 2002.

[15] F. Petrini, E. Frachtenberg, A. Hoisie, S. Coll, Performance Evaluation of the Quadrics Interconnection Network, Journal of Cluster Computing 6 (2) (2003) 125–142.

[16] F. Schmuck, R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, in: Proceedings of the Conference on File and Storage Technologies (FAST'02), USENIX, Berkeley, CA, Monterey, CA, 2002, pp. 231–244.

[17] P. Geoffray, OPIOM: Off-Processor I/O with Myrinet, PPL - Parallel Processing Letters 11 (2-3) (2001) 237–250.

[18] O. Cozette, C. Randriamaro, G. Utard, READ^2: Put disks at network level, in: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), IEEE Computer Society, Tokyo, Japan, 2003.

[19] W. Ligon, Next Generaton Parallel Virtual File System, in: Proceedings of the 2001 IEEE International Conference on Cluster Computing, Newport Beach, CA, 2001.

[20] PVFS2 Development Team., Parallel Virtual File System, Version 2, `http://www.pvfs.org/pvfs2/pvfs2-guide.html` (Sep. 2003).

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W.-K. Su, Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro 15 (1) (1995) 29–36.

[22] H. Tezuka, F. O'Carroll, A. Hori, Y. Ishikawa, Pin-down cache: A Virtual Memory Management Technique for Zero-copy Communication, in: 12th International Parallel Processing Symposium, 1998, pp. 308–315.

[23] B. Goglin, L. Prylli, O. Glück, Optimizations of Client's side communications in a Distributed File System within a Myrinet Cluster, in: Proceedings of the IEEE Workshop on High-Speed Local Networks (HSLN), held in conjunction with the 29th IEEE LCN Conference, IEEE Computer Society Press, Tampa, Florida, 2004, pp. 726–733.

[24] V. S. Pai, P. Druschel, W. Zwaenepoel, IO-Lite: a unified I/O buffering and caching system, ACM Transactions on Computer Systems 18 (1) (2000) 37–66.

[25] J. Wu, P. Wyckoff, D. Panda, R. Ross, Unifier: Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand, in: IEEE International Symposium on Cluster Computing and the Grid (CCGrid2004), 2004.

[26] B. Goglin, O. Glück, P. V.-B. Primet, An Efficient Network API for in-Kernel Applications in Clusters, in: Proceedings of the IEEE International Conference on Cluster Computing, IEEE Computer Society Press, Boston, Massachussets, 2005.

[27] R. Brightwell, K. D. Underwood, An Analysis of NIC Resource Usage for Offloading MPI, in: Proceedings of the 2004 Workshop on Communication Architecture for Clusters, Santa Fe, New Mexico, 2004.

[28] B. Goglin, Réseaux rapides et stockage distribué dans les grappes de calculateurs : propositions pour une interaction efficace, Ph.D. thesis, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, 194 pages (Oct. 2005).

[29] K. Magoutis, The Optimistic Direct Access File System: Design and Network Interface Support, in: Proceedings of Workshop Novel Uses of System Area Network 2002, Cambridge, MA, 2002.

[30] B. Goglin, L. Prylli, Performance Analysis of Remote File System Access over a High-Speed Local Network, in: Proceedings of the Workshop on Communication Architecture for Clusters (CAC'04), held in conjunction with the 18th IEEE IPDPS Conference, IEEE Computer Society Press, Santa Fe, New Mexico, 2004, p. 185.

[31] Lustre - MX-10G Performance Measurements, `https://mail.clusterfs.com/wikis/lustre/MX-10G` (2006).