An Efficient Network API for in-Kernel Applications in Clusters

Brice Goglin, Olivier Glück, and Pascale Vicat-Blanc Primet
Laboratoire de l'Informatique du Parallélisme
CNRS-ENS Lyon-INRIA-UCBL – France
{Brice.Goglin,Olivier.Gluck,Pascale.Primet}@ens-lyon.fr

Abstract

Running parallel applications on clusters with high-speed local networks requires fast communication between computing nodes but also low latency and high bandwidth file access. However, the application programming interfaces of high-speed local networks were designed for MPI *communication and do not always meet the requirements of other applications like distributed file systems.*

In this paper, we explore several solutions to improve the use of high-speed network for in-kernel applications. Distributed file systems implemented on top of the GM *interface of* MYRINET *are first examined to demonstrate how hard it is to get an efficient interaction between such applications and the network. Then, we propose solutions to simplify and improve this interaction and integrate them into the kernel interface of the new* MYRINET *driver,* MX*. Performance comparisons between* MX *and* GM*, and their usage in both a distributed file system and a zero-copy protocol show nice improvements. Moreover, we are able to improve the performance of the flexible kernel* API *we designed in* MX *that allows to remove some intermediate copy.*

1 Introduction

The emergence of parallel applications led to the success of workstation clusters which are generic, extensible and less expensive. As the application always require more computational power, high-bandwidth and low-latency local networks with intelligent interface cards have been developed, such as MYRINET [1], QUADRICS [12] or INFINIBAND [13]. Specific software optimizations have been proposed to enable on one hand the overlapping of communications with computation phases, and on the other hand a drastic reduction of communication overhead in the host.

This work is supported by the French Ministry of Research (Grid5000, ACI GRID), INRIA (RESO Project), CNRS, ENS Lyon, UCBL and MYRICOM.

Overlapping communication between different nodes is enabled by deporting a large part of the protocol stack in the network interface card. This led to high-level software programming interfaces that are based on asynchronous communication primitives, such as MPI (*Message Passing Interface* [3]). The application posts send or receive requests and gets notification of their completion later. This is very different from traditional network interfaces, especially the SOCKET interface to access ETHERNET networks through the TCP/IP stack where all communication primitives are blocking.

The reduction of the communication overhead in the host is achieved by three mechanisms. Intermediate copies have been removed (*0-copy*), the operating system has to be avoided (*OS-bypass*) and data have to be transferred directly between applications and the network through DMA (*Direct Memory Access*) initiated by the NIC (*Network Interface Card*).

Several works have targeted communication software layers to improve the performance of parallel applications. However, very few works have focussed the utilization of high-speed local networks in other contexts, especially for applications that are not implemented in user-space like MPI computations. For instance, storage requirements in clusters may benefit from an efficient usage of these networks. This includes either distributed file systems or *Network Block Devices*. Applications using the SOCKET interface may also be improved by using a zero-copy protocol on top of the native in-kernel API of a high-speed local network. All these kernel contexts have specific constraints that are very different from user-space communication requirements.

This paper presents our study and propositions for improving the interaction between such in-kernel applications and the highly-specific network programming interface on MYRINET. Section 2 presents potential in-kernel applications that may be used on clusters and their issues when interacting with the network. Then, we expose, in Section 3, how we have modified the GM interface of MYRINET to allow an efficient usage of the the underlying network in

distributed file system clients. We detail in Section 4 several ideas to improve the flexibility of network programming interfaces and how we integrated them in the new MYRINET/MX driver. Section 5 finally gives a first performance evaluation in both distributed file systems and zero-copy socket protocols.

2 In-Kernel Applications on High-Speed Local Networks

2.1 Context

Parallel applications running on clusters often want to get as much performance for storage access as for communication between computing nodes. Storage access layers may include either distributed file systems or network block devices, and were usually implemented in the operating system. The standard interface that is exposed to user application is basically based on non-vectorial non-collective blocking operations, often implemented with intermediate copies. This basic interface led cluster designers to propose new specific file access interfaces, in the same way they improved communication performance by proposing specific APIs that are suitable to high-speed network low-level software layers.

The MPI-IO interface [8] was designed to provide the same model for file access primitives than for communication primitives in MPI. Another example is DAFS (*Direct Access File System* [7]), which was designed to make the most out of the underlying network for remote file access. It thus provides a highly specific API, similar to VIA (*Virtual Interface Architecture* [14]). Both file access interfaces provide a high-performance distant file access model. But, they require user-application to be written according to their very specific requirements, for instance asynchronous requests and completion queues.

More recently, modern operating systems have integrated these parallel application requirements. Vectorial primitives were first introduced. Then, zero-copy file access was added to the LINUX kernel and asynchronous input-output in LINUX 2.6. This new support for advance features in the operating system leads to the utilization of standard interfaces instead of specific interfaces that required application rewriting. LUSTRE [2] is one of the most famous recent distributed file systems for clusters. It provides a parallel and scalable system that respects the standard file access interface.

On the other hand, high-speed network APIs remain very specific. Event-based interfaces with asynchronous primitives and completion notification generally does not raise very difficult problems in non-MPI contexts. However, the virtual memory management, and especially memory registration, still seems designed for user-space MPI applications. For instance, first LUSTRE releases used intermediate memory copies to integrate the GM interface [9] of MYRINET networks. These copies are CPU consuming while the user parallel application needs the CPU for its computations (MYRINET support has apparently been dropped now). Moreover, the kernel PVFS2 client (*Parallel Virtual File System* [6]) returns to user-space before accessing the network. This design choice is justified in the documentation through the problem of having ready access to all networking APIs from within the kernel.

We now detail the constraints of high-speed network interfaces, study their interaction with our main target application, a distributed file system client, and give several hints concerning other in-kernel applications.

2.2 Constraints of High-Speed Network APIs

2.2.1 Memory Registration

Applications manipulate virtual addresses while the hardware only knows physical addresses. The memory management subsystem is usually in charge of this translation. It is no-more involved in OS-bypass communications. QUADRICS QSNET networks require modification of the operating system so that all addressing details are transparently forwarded to the network interface. Most other network systems, especially MYRINET and INFINIBAND, prefer not modifying the operating system. Thus, they require address translation with the explicit help of the kernel.

The common strategy is based on asking the application to prepare the I/O buffers it is going to use for communications. This operation is commonly called *Memory Registration*. It uses a specific system call to pin pages in physical memory and register their address translations into the network interface card. All the following communications may then directly pass virtual addresses to the NIC which will get their associated physical addresses from its translation table. Data transfers are thus processed by a DMA engine on the NIC without any operating system help. This translation table in the NIC has been first introduced in U-NET/MM [17].

This strategy presents two drawbacks. Firstly, memory registration cost is usually so high that it can only be efficient if the application reuses registered buffers in several communications. Secondly, traditional applications were not designed to explicitly register their I/O buffers. It is thus required to either modify them or to insert a transparent layer to register on the flight.

Standard parallel computing libraries such as MPI or VIA have fortunately been implemented on top of these specific network software interfaces. This leads to parallel applications making the most out of the underlying high-speed network. However, this specificity makes their usage

difficult in a different context, especially for an in-kernel application or a client-server protocol.

2.2.2 Example of Memory Registration with GM on Myrinet networks

GM is the current official driver of MYRINET networks. It follows the message-passing paradigm and was designed for MPI applications. The user posts send, receive or remote memory access requests and gets their completion notifications in a unique event queue.

GM requires all I/O buffers used by the application to be registered in the NIC first. As the amount of page translations that may be stored in the NIC is limited, useless entries have to be deregistered. As usual, registration and deregistration have a high cost. In GM, we measured a 3 μs overhead per page registration, with the addition of a 200 μs base for deregistration (see Figure 1). Actually, this model is only interesting for large memory zones that are used several times.

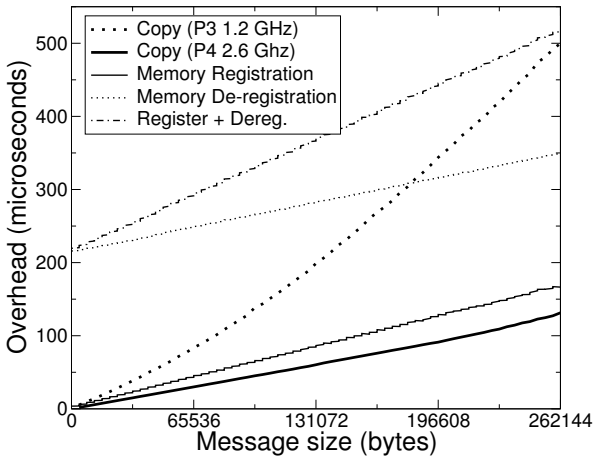

Figure 1. Comparison between copy and memory registration cost in GM.

A simple strategy consists in copying small buffers into a statically pre-registered memory zone. A small amount of CPU cycles is wasted but the large cost of memory registration is also avoided for these small buffers

The major improvement was proposed in [16]: a memory registration cache (*Pin-down Cache*). Deregistration is delayed until it is really required (when no more pages can be registered). If cached pages are re-used, registration is not needed. The major drawback is that the cache must be kept up-to-date with mapping changes. As the application is not aware of the caching of its address translations in the NIC, it might change its address space (especially through `free` or `munmap`), thus making the registered translation invalid. A common solution consists in updating the cache by intercepting all address space modification calls from the application.

This happens in a middle-ware (for instance MPI) between GM and applications that were not written to register their I/O buffers. In this case, the middle-ware transparently registers buffers on the flight and intercepts address space modifications. In a distributed file-system or zero-copy socket protocol, we will see that such an on-the-flight registration mechanism is also required.

2.3 Interaction between High-Speed Networks and Distributed File Systems

We now detail the interaction issues between high-speed network software interfaces and our main target in-kernel application, a distributed file system client.

2.3.1 Buffered Access to remote files and Interaction with the Page-Cache

Physical storage systems are so slow that modern operating systems have to optimize their access. The LINUX *Page-Cache* keeps copies of disk blocks in the host memory to avoids repetitive reading of same physical blocks. Writing is processed asynchronously so that the application does not wait. Data transfer involving the application are thus only memory copies between its user-space and the page-cache.

In a distributed file system using such **buffered accesses**, a protocol has to maintain consistency across cached pages on different clients and physical blocks on the server. Using a GM-like interface in such a context really differs from parallel application context. Firstly, high-speed network software layers were not designed for communication from a kernel context. Secondly, memory zones that are involved here have very different characteristics than the traditional user buffers. Pages of the page-cache are already locked in physical memory and generally not mapped in virtual memory. But, their physical address is easy to obtain since for instance, a distributed file system client runs in a kernel context (in contrary to a user application running in a user context). The assumptions for the design of the memory registration model are thus no more suitable here.

It is important to note that this is also valid for *Network Block Device* clients. The NBD client is at the bottom of the storage stack in the operating system. It allows to mount remote disks as local partitions. Such an application manipulates the page-cache as a distributed file system client does. Thus, the same conclusion about memory registration applies here.

2.3.2 Direct Access to remote files

The kernel page-cache has the drawback of preventing applications from controlling physical disk accesses. Memory consuming applications, for instance databases or out-of-core computation, keep their own memory cache in user-space. They do not want their write requests to be buffered by the operating system since the page-cache might swap out some important pages of the application to store the local copy of written data. Modern UNIX systems thus provide zero-copy disk access to bypass the page-cache (by giving the `O_DIRECT` flag when opening the file). Data transfers are then **direct** from application I/O buffers to the storage subsystem, that is local disks or a distant file-system on the server.

This strategy is very similar to zero-copy data transfer between I/O buffers of different instances of a parallel application running on a cluster. But, the implementation is very different since several operating systems, especially LINUX, do not provide any support for the high-speed network communication model while zero-copy file access is supported.

Actually, direct file access is not the only in-kernel application that require zero-copy data transfer between user-space and the networks. It is now common to implement a zero-copy socket layer on top of the native network interface. A specific zero-copy implementation is added to the support of the SOCKET interface in the kernel. This makes traditional application using sockets benefit from the underlying high-speed network without any modification. Zero-copy socket protocols have same requirements than direct file access with the `O_DIRECT` flag.

3 Experimentations with GM on Myrinet

Our goal is to improve the interaction between in-kernel applications and high-speed network APIs. We now detail of such an application, a distributed file system client, may be efficiently implemented on MYRINET networks.

3.1 ORFA, a Remote File-Access Protocol for High-Speed Networks

We developed an experimentation protocol named ORFA (*Optimized Remote File-system Access*) to optimize point-to-point communications between a client and a server in a distributed file system. Our work must then be applicable in real systems such as PVFS or LUSTRE to improve their usage of the underlying high-speed network. We focus on standard file access interfaces, especially recent LINUX interfaces, and try to tightly integrate the usage of MYRINET networks to make the most out of their performance.

ORFA has initially been developed in user-space to study the impact of high-speed networks on remote file access, without suffering of in-kernel implementation constraints. The ORFA client was a user-level library transparently intercepting all remote file access and supporting special primitives such as `fork` or `exec` [5].

We showed in [4] that MYRINET networks may transfer large amount of data with very high performance in this context. However, meta-data access (file attributes) does not benefit from the low latency of the network. We then decided to work on ORFS (*Optimized Remote File System*), the ORFA client in the LINUX kernel. This implementation benefits from VFS caches (*Virtual File Systems*) improving meta-data access, and secondly gives a much larger validation of our work since most distributed file systems in clusters are now implemented in the kernel.

We now detail both direct and buffered file access implementations in ORFS on GM. We used GM 2.0.13 on a Linux kernel 2.4.26. Our experimentation platform is composed on 2.6 GHz dual-XEON nodes with 2 GB RAM and PCI-XD MYRINET cards. This network can sustain 250 MB/s full-duplex.

3.2 Direct Access to remote files on GM

Direct remote file access implementation in ORFS (see Section 2.3.2) requires communications from application user-space memory to a MYRINET communication port, that was open in the kernel. Memory registration seems to be the simplest solution to this case. But, a registration cache (as in the ORFA client) is needed. This imposes to know address space modifications. It is possible in user-space by intercepting application requests in a shared library. But, the LINUX kernel does not provide any mechanism for such tracing in a kernel context.

Thus, we developed a generic infrastructure called VMA SPY allowing any external module to ask for notification of address space modifications. VMA SPY adds several hooks in the LINUX kernel where registered functions are called. An external module may then attach a function it to be called when an virtual memory area is unmapped or duplicated.

Then, we implemented a generic registration cache in the kernel named GMKRC (*GM Kernel Registration Cache*) which is kept up-to-date by VMA SPY (see Figure 2). Moreover, GMKRC is responsible for solving collisions between address spaces of the multiple processes accessing our file system. Indeed, GM assumes a *port* can only be used by a single process. Our shared port model prevents the network interface card from knowing which address space a given virtual address belongs to. We solves this problem by recompiling the card firmware with 64 bits pointers on 32 bits host. GMKRC then stores a descriptor

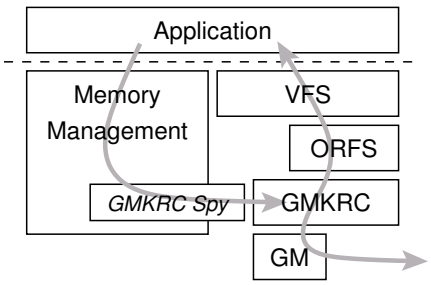

Figure 2. Direct access in ORFS with GMKRC (registration cache) and VMA Spy (notification of address space modifications).

of the address space in the most significant bits. This makes any descriptor unique and thus avoids any address space collision. This strategy is transparently implemented inside GMKRC so that in-kernel users still pass normal 32 bits pointers to the GMKRC API.

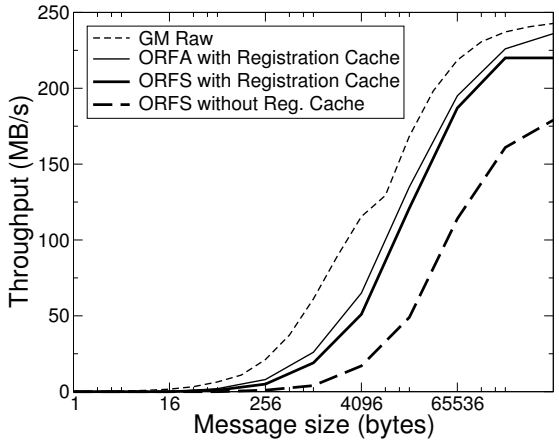

Figure 3. Performance of direct access in ORFS with and without registration cache. Comparison with raw GM and ORFA (user-space implementation).

We present on Figure 3 the bandwidth observed at the application level through traditional `read` calls on top of direct remote file accesses with the ORFS protocol and GMKRC and VMA SPY. These results, and all the following ones, were obtained by averaging 100 runs. Comparing ORFS with and without registration cache highlights the impact of the application memory utilization scheme. Without any cache hit, the performance is 20 % lower. ORFS performance is still lower than ORFA because of the overhead of system calls and of the traversal of the VFS layers.

ORFS direct file access with GMKRC and VMA SPY shows to very good performance on top of GM in the kernel since 90 % of the network bandwidth is achieved.

3.3 GM and the Page-Cache

Buffered file access requires to transfer data between the page-cache in the kernel and the distant server. From the ideas we developed in Section 2.3.1, we added to the GM kernel interface some communication primitives based on physical addresses and the required infrastructure in the MCP (*Myrinet Control Program*, the program running in the network interface card). Remote file access through the page-cache in ORFS give the physical addresses of all pages that are involved in the requested communication to the NIC. This strategy improves the latency since the NIC does not require to translate the given virtual address by looking in its translation table. We measured a 0.5 μs gain on both the sender and the receiver's side on our MYRINET cards, that is 10 % improvement.

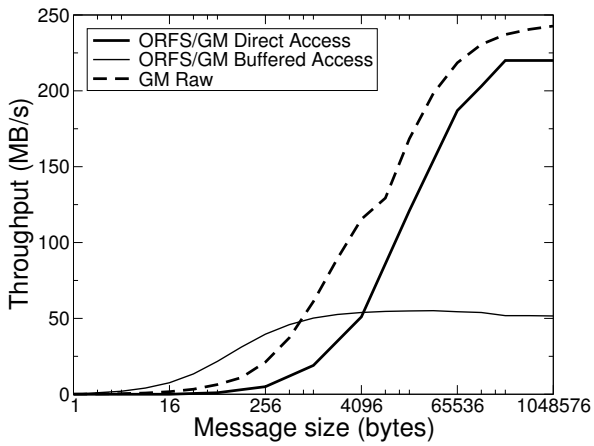

Figure 4. Performance comparison between direct access (zero-copy) and buffered access (through the Page-Cache using the physical address) in ORFS on GM.

Remote file access performance with ORFS using our GM physical address based interface is presented on Figure 4. We measure the throughput at the application level when accessing large files sequentially. Remote accesses without involving the page-cache are represented by the **direct** case, while **buffered** case is through the page-cache.

The page-cache is progressively filled by the kernel according to application requests. Data transfers are processed per page (4 kB on our architecture). This leads to

an under-utilization of the network bandwidth. However, 4 kB accesses are faster through the page-cache compared to direct accesses, even if an additional copy from the page-cache to the application is required. This shows the efficiency of our physical based interface.

On the other hand, an application requesting large data transfers will show much better performance in the direct case. The reason is that a direct access requires only one network request while a large buffered access is split in page-sized requests. This issue should disappear with LINUX 2.6 kernels which are able to combine multiple page-sized accesses in a single request. However, this would require vectorial communication primitives, that is something GM does not provide (see Section 4.1).

We took care of optimizing both direct and buffered remote file accesses. But, it is important to keep in mind that choosing between these two types is the application responsibility.

4 Propositions to improve in-Kernel APIs of Cluster Networks

We have presented the difficulties one may face when trying to efficiently use the high-speed network in a distributed file system. We had to patch both GM and the LINUX kernel to simplify their interaction in our experimental platform, ORFS. These difficulties are actually not specific to MYRINET GM. Any software interface based on memory registration will face similar issues. That is why we propose new mechanisms to facilitate the usage of high-speed local network APIs in a non-MPI contexts.

4.1 Physical Address and Vectorial Communications

As memory registration is not the right solution for communication involving the page-cache of the kernel, and as our physical address based primitives are adapted to any communication initiated from a kernel context, it is easy to obtain the physical address of any page of the page-cache, but also any page that is mapped in kernel-space or even in user-space. **Kernel memory** is used for messages that are exchanged between the in-kernel application and distant nodes, for instance requests that are sent to a file server. Such memory zones are often already pinned. **User memory** is used for zero-copy data transfer between the application and the network (see Section 2.3.2). User memory zones have to be pinned.

In both cases, the cost of memory registration is avoided and the latency is improved (especially for small messages). Providing a physical address based interface to high-speed network software APIs thus seems recommended.

However, a virtually contiguous memory zone is generally not physically contiguous. It is especially true for user-space memory. Any multiple-page transfer (more than 4 kB on IA32 architectures) between user-space and the network using physical addresses would then be divided into multiple non-contiguous segments. Moreover, using multiple pages of the page-cache would also leads to segmented communications (see Section 2.3.1).

It is obviously possible to use one communication primitive for each physically contiguous zone. But, this might be a constraint since some interfaces (especially GM) ask the user to limit the amount of pending requests. The easier solution here is to use vectorial communication primitives to transfer several non-contiguous segments at once. These primitives are not offered by several interfaces such as GM. In the same way they are useful in user-space (for instance to implement an efficient MPI layer), they might be a very interesting feature in a kernel API. In the next Section, we detail the integration of these ideas in the MX kernel interface.

4.2 Implementation in Myrinet Express

MX (*Myrinet Express* [10]) is the next official software interface of MYRINET networks. Its main characteristics is to almost provide a MPI interface at the network level (since MPI is the most common application). Vectorial communications are thus supported. Another interesting point is the fact that physical addresses are now directly usable by the network interface card, thus no explicit memory registration is required.

We worked in collaboration with MYRICOM to, firstly move and expose the MX programming interface in the kernel, and secondly integrate our ideas to make the interaction between in-kernel applications and MYRINET networks easier. Our work has now been integrated in the official MX distribution. Its in-kernel API proposes a native and optimized support for different types of memory addressing. The application has to pass this type of address to MX:

User virtual: MX pins the target zones and translates their addresses into physical addresses.

Kernel virtual: These zones are often already pinned. MX just has to translate addresses.

Physical: The application is responsible for pinning memory if needed.

The distinction between the first two types gives a more generic support of the difference between user and kernel spaces. These spaces are generally independent and non-contiguous. This signify that they contain same virtual addresses pointing to different physical locations. It is then

impossible for the network layer to know whether a given virtual address should translated into its user or kernel corresponding physical address. Even if standard LINUX kernels do not have this issue (user and kernel spaces are contiguous), it is easy for any application to tell what kind of address it is using. This is the reason why we added this distinction in the MX API.

Looking at other kernel-level APIs such as KDAPL for INFINIBAND or KCOMM for QUADRICS, it appears that similar memory addressing solutions have been proposed, even if these APIs where not designed for same requirements. This shows how our interface may be suitable to various applications on MYRINET networks.

5 Performance evaluation

5.1 Performance of MX in-kernel interface

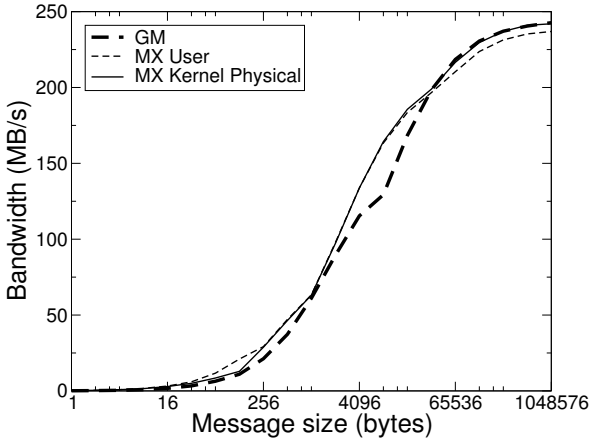

Figure 5. Comparison of bandwidth in MX and GM.

During the development of the MX kernel interface, we designed a very generic core infrastructure so that kernel communications would not suffer of a user-oriented design. Consequently, the MX performance (see Figure 5) does not differ between user and kernel communications. The large message bandwidth is even higher with the kernel interface since the page locking overhead is lower.

GM does not provide such an efficient kernel interface. Its small message latency is 2 us higher in the kernel compared than user-space. Moreover, GM user latency is more than 50 % higher than with MX (6.7 us against 4.2 us for 1-byte message). GM large message bandwidth is the same than MX. But, GM benefits here from a 100 % reuse of the application buffers while MX does not.

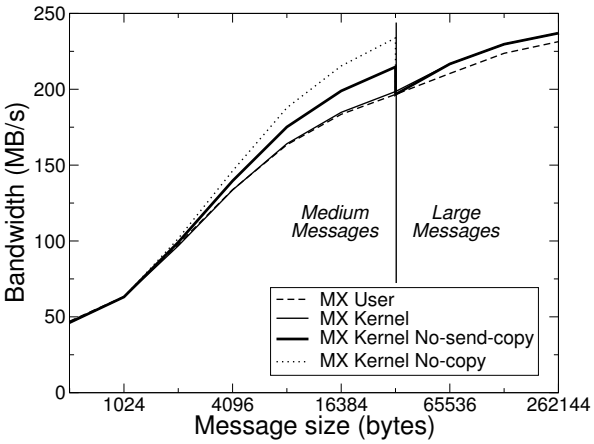

Figure 6. Measured impact of the removal of the copy of medium messages (from 128 bytes to 32 kB) on the sender side and predicted impact of the removal of both side copies in MX.

Our specific kernel interface enables optimizations according to the type of addressing that the application passes. Indeed, this addressing characteristic may be used to avoid locking or even segmentation of physically contiguous zones. The standard MX implementation uses a copy on both sides when processing medium side messages (from 128 bytes to 32 kB). Larger messages are pinned internally while small messages use *Programmed I/O*. These internal intermediate copies correspond to what common application do when trying to avoid explicit memory registration (see Section 2.2.2). MX does not ask applications to register memory but uses copy or registration internally.

This copy might actually be avoided for physical address based communications. As a proof of concept, we removed the copy on the send side and show the resulting bandwidth on Figure 6. It leads to 17 % bandwidth improvement for 32 kbytes messages. This optimization is possible since the network card interface does only manipulate physical addresses in MX. It is then easy to pass the application given addresses, even if a translation is needed (since we are in the kernel).

We also predicted the impact of removing the copy on the receive side (see the dashed graph on Figure 6). It gives another 15 % bandwidth improvement for medium messages. It is currently impossible to implement this optimization since the NIC does not know the address of the receive buffer. The receives are processed by the host. Thus, we can not avoid the receive copy by directly passing our physical address to the NIC. This issue might be solved when

future MX development will move receive processing into the NIC.

Such an improvement might lead to increase the medium message maximal size in this context since large message bandwidth looks lower. Fortunately, large message processing in MX is still under strong development. The current performance difference might disappear soon.

Anyway, our kernel interface allowed the removal of one intermediate copy for physically contiguous messages. Non-contiguous messages will require the upcoming vectorial communication support in MX. For now, our optimization gives an interesting improvement when sending up to 8 physically contiguous pages on IA32 architecture. This is not a usual case in a distributed file-system environment. The most common case would be a single-page transfer. In this case, our optimization gives a 9 % improvement.

5.2 Application to Distributed File systems

We now study the performance of our MX kernel interface in real applications. The copy-removal optimization that we just presented was not used during the following tests. We first compare remote file access performance with ORFS on GM and MX.

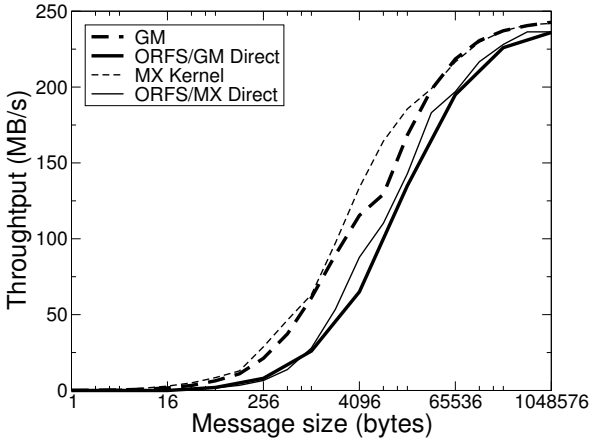

Figure 7. Performance comparison of direct file access in ORFS on MX and GM.

Direct file accesses on MX are slightly better than over GM. The difference is similar to their raw bandwidth difference. However, we have to keep in mind that GM benefits here from 100 % hits in the registration cache while MX does not use such a strategy. We showed in Section 3.2 that ORFS performance on GM can be reduce from 20 % with much less cache hits.

Buffered file access in ORFS on MX shows a 40 % improvement over GM. Network requests are page-sized in this context. But, MX raw performance is not better than GM for such messages. The ORFS/MX performance improvement is thus caused by our improved kernel interface (which makes the ORFS implementation much more efficient). It also has to be emphasized that GM was designed for user-level applications and thus lacks an efficient in-kernel communication implementation.

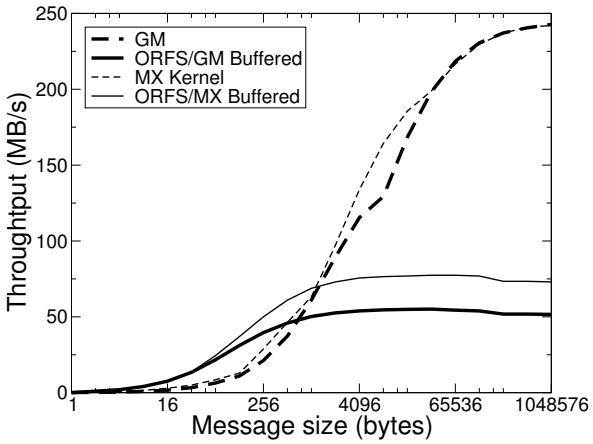

Figure 8. Performance comparison of buffered file access in ORFS on MX and GM.

In both file access types, the MX kernel interface was much easier to work with than the GM one. Firstly, no kernel patch is required since memory registration is only used internally. Our kernel API provides all primitives that our implementation requires, especially when dealing with user space for 0-copy data access or with physical addresses in the page-cache for buffered accesses. Secondly, the MX API makes event notification more flexible, for instance by allowing the application to wait on a single or any pending request. For instance, this makes the implementation of both synchronous and future asynchronous file requests easier.

5.3 Application to Zero-Copy Socket Protocol

MYRICOM offers another software which is using MX in the kernel, SOCKETS-MX. It allows existing applications in binary format to benefit from the high-speed MYRINET network when using TCP/IP socket function calls. It adds a new SOCKET protocol to the LINUX kernel where data is directly passed onto the MYRINET network bypassing TCP/IP. With the fully asynchronous send functions in MX the overhead is significantly lower than when the full TCP/IP stack needs to be traversed (with fragmentation and checksum computation). As a matter of fact, TCP/IP is known to use 50 % of the overall transaction cost [15].

We compared SOCKETS-MX performance with the same implementation using GM. SOCKETS-GM offers the same capabilities but lacked two major skills. Firstly, limited completion notification mechanisms in GM require the use of an extra (dispatching) kernel thread which increases the latency. Secondly, memory registration problems are similar to ORFS direct file access troubles on top of GM, as expected in Section 2.3.2.

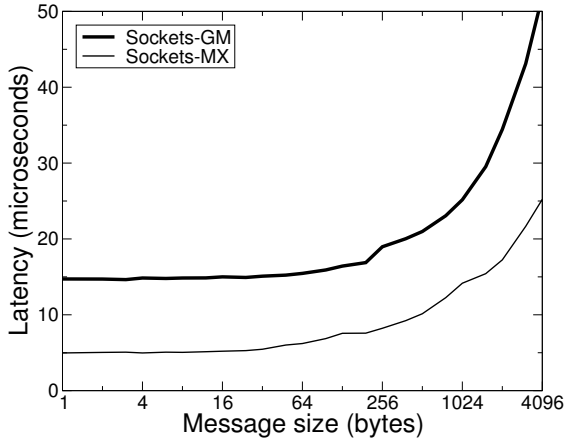

Figure 9. Sockets-MX and Sockets-GM small message latency comparison.

We used NETPIPE [11] ping-pong to compare SOCKETS-MX and SOCKETS-GM performance on PCI-XE MYRINET cards (these cards can sustain 500 MB/s full-duplex by using two links). We measured a 5 μs one-way latency at the application level for 1-byte messages with SOCKETS-MX (see Figure 9). This is only a 1 μs overhead over raw MX latency. It is very good since a system call is involved (about 400 ns). SOCKETS-GM gets 15 μs one-way latency. A common GIGA-ETHERNET network might get much more.

SOCKETS-MX bandwidth is always higher than SOCKETS-GM (see Figure 10). Medium message bandwidth improvement is up to 100 % (for 4 kB) while large message is up to 50 % (for 1 MB). This shows how the MX kernel interface and SOCKETS-MX behaves well. The different anomalies on the graphs are caused by the fact that SOCKETS-MX and SOCKETS-GM implementation in the LINUX kernel are still under development.

6 Conclusion and Perspectives

In this paper, we have presented the issues that may be raised when using high-speed network software programming interfaces in a non-traditional context. These soft-

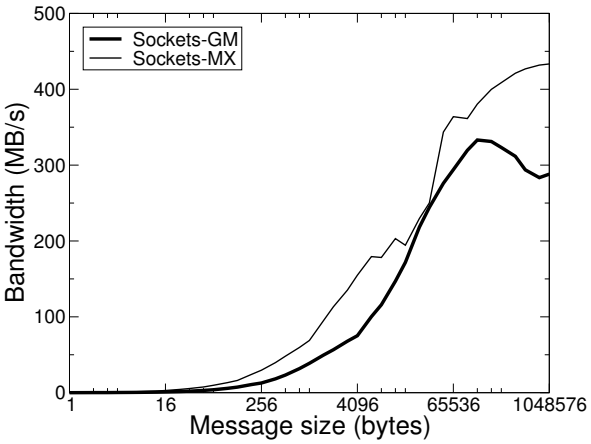

Figure 10. Sockets-MX and Sockets-GM bandwidth comparison.

ware interfaces were designed for MPI applications in user-space. Their specificity makes their usage difficult in our very different target contexts. We focussed on in-kernel applications and especially detailed the interaction between MYRINET networks and a distributed file system client.

We first exposed modifications of the GM interface of MYRINET networks. Firstly, using physical address based communications seems very useful when dealing with the page-cache. Secondly, the implementation of GMKRC (GM registration cache in the LINUX kernel) and VMA SPY (generic infrastructure to notify of addressing modification) leads to efficient direct remote file access. Performance evaluation on our experimental distributed file system ORFS show how it is important to make the network programming interface and storage access layers interact well.

We proposed several ideas to improve software interfaces for cluster networks. Physical address based primitives and vectorial communications seem very important to improve the way in-kernel applications benefit from the high-speed local network. Moreover, application help (by saying the address type) looks useful to efficiently handle different types of memory addressing. We integrated our ideas in the upcoming MYRINET driver, MX.

The MX kernel interface behaves at least as the traditional user-level interface, with a more flexible support for different types of memory that an in-kernel application may manipulate. We presented performance evaluation of two in-kernel applications, a distributed file system client and a zero-copy socket protocol. Both benefit from MX small message latency and high bandwidth without requiring any memory registration cache mechanism. Performance eval-

uations of distributed file systems with the MX kernel interface show a nice improvement for buffered file accesses. Direct file accesses are slightly better than over GM with 100 % cache hits in its registration cache. SOCKETS-MX shows that both latency and bandwidth get a large improvement compared to the GM implementation.

The kernel API appears to be much more easy to use and flexible in our context. Moreover, several kernel specific optimizations in the MX implementation are possible due to the advanced knowledge of memory type given by the application. As a proof of concept, we introduced a 15 % bandwidth improvement by removing a copy on the sender's side when dealing with physically contiguous medium message in MX.

We expect that our third target in-kernel application, a *Network Block Device* client, will also largely benefit from our improved kernel software interface. This client transmits low-level block device accesses to a remote server, allowing remote partition mounting such as with ISCSI. Such a client manipulates the page-cache in a similar way a distributed file system client does. Our physical address based interface should thus be suitable in this context.

We also plan to compare our work with other kernel-level APIs such as QUADRICS KCOMM and KDAPL on INFINIBAND since they propose similar solutions.

7 Software Availability

ORFA and ORFS implementations are distributed under GPL licenses. They may be downloaded from `http://perso.ens-lyon.fr/brice.goglin/work`.

The MX kernel interface is available in the main MX distribution through MYRICOM.

8 Acknowledgments

We would like to thank MYRICOM for its collaboration, especially Markus Fischer for providing SOCKETS-MX performance evaluation results, and Andrew Gallatin and Loïc Prylli for their precious help during the development of MX kernel interface.

References

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

[2] Cluster File Systems, Inc. Lustre: A Scalable, High Performance File System, Nov. 2002. `http://www.lustre.org`.

[3] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[4] B. Goglin and 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*, Santa Fe, New Mexico, Apr. 2004. IEEE Computer Society Press.

[5] B. Goglin and L. Prylli. Transparent Remote File Access through a Shared Library Client. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*, volume 3, pages 1131–1137, Las Vegas, Nevada, June 2004. CSREA Press.

[6] W. Ligon. Next Generaton Parallel Virtual File System. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, Newport Beach, CA, Oct. 2001.

[7] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proceedings of USENIX Conference on File and Storage Technologies 2003*, San Francisco, CA, Mar. 2003.

[8] MPI-IO: I/O Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/mpi-20-html/node172.htm`.

[9] Myricom, Inc. *GM: A message-passing system for Myrinet networks*, 2003. `http://www.myri.com/scs/GM-2/doc/html/`.

[10] Myricom, Inc. *Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet*, 2005. `http://www.myri.com/scs/MX/doc/mx.pdf`.

[11] NetPIPE: A Network Protocol Independent Performance Evaluator. `http://www.scl.ameslab.gov/netpipe/`.

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

[13] G. F. Pfister. Aspects of the InfiniBand Architecture. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, pages 369–371, Newport Beach, CA, Oct. 2001.

[14] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *International Conference on Supercomputing*, pages 184–192, 1999.

[15] S. Sumimoto. *A Study of High Performance Communication for Parallel Computers Using a Commodity Network*. PhD thesis, Keio University, 2000.

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

[17] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Stanford, Aug. 1997.