# Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware

Brice Goglin

HAL Id: inria-00210704
https://inria.hal.science/inria-00210704

Submitted on 21 Apr 2008

# Design and Implementation of Open-MX:
# High-Performance Message Passing over generic Ethernet hardware

Brice Goglin

Laboratoire Bordelais de Recherche en Informatique
INRIA Bordeaux - Sud-Ouest – France
`Brice.Goglin@inria.fr`

## Abstract

*Open-MX is a new message passing layer implemented on top of the generic Ethernet stack of the Linux kernel. It provides high-performance communication on top of any Ethernet hardware while exhibiting the Myrinet Express application interface. Open-MX also enables wire-interoperability with Myricom's MXoE hosts.*

*This article presents the design of the Open-MX stack which reproduces the MX firmware in a Linux driver. MPICH-MX and PVFS2 layers are already able to work flawlessly on Open-MX. The first performance evaluation shows interesting latency and bandwidth results on 1 and 10 gigabit hardware.*

## 1 Introduction

The emergence of 10 gigabit ETHERNET hardware raises the usual question of whether it may replace dedicated hardware such as IB or MYRI-10G as a high-speed interconnect for clusters. Several advanced features such as the offloading of checksum computation, TCP segmentation (TSO) or large receive (LRO) enabled high-performance communication with TCP/IP but are still restricted to specifically tuned configurations and lead to a high CPU load.

Meanwhile, ETHERNET appears as an interesting networking layer within local networks for various protocols such as FIBRECHANNEL and ATA. The increasing importance of ETHERNET in high-performance computing is also revealed by its interoperability with high-speed interconnects such as MYRI-10G and QSNET III. However, these technologies still require dedicated interfaces on the nodes. This makes them impossible to use on regular hardware.

In this paper, we present the design and implementation of a message passing layer for any generic hardware. OPEN-MX is a port of the *Myrinet Express* [6] protocol on top of the generic ETHERNET layer of the LINUX kernel. The MX application interface and wire protocol have been designed for high-performance message passing and should provide the same abilities on top of generic ETHERNET hardware. OPEN-MX also enables interoperability between any hosts, even when running the native MXoE stack (*Myrinet Express over Ethernet*) on MYRICOM's MYRI-10G boards. This project is expected to provide the networking layer for PVFS2 [7] in the BLUEGENE/P systems. We will detail the design of OPEN-MX in Section 2, before highlighting some implementation aspects of the memory copy avoidance in Section 3 and presenting early performance evaluations in Section 4.

## 2 Design of Open-MX

### 2.1 Emulating the MX firmware

The *Myrinet Express* software stack has been designed for high-performance message passing [6]. It exposes the capabilities of MYRINET and MYRI-10G hardware and firmware at the application level while providing low-latency and high bandwidth (2 $\mu$s and 1250 MB/s data rate). To do so, OS-bypass communication is used for small messages and zero-copy for large messages. The operating system is not involved during communication, only the initialization phase and the memory registration of large messages require its assistance (see Figure 1). All the actual communication management is implemented in the user-space library and in the firmware. The MX firmware is also responsible for using ETHERNET headers when talking to regular ETHERNET switches (MXoE mode, *MX over Ethernet*), to MYRINET specific headers when talking to MYRICOM switches (native MX mode).
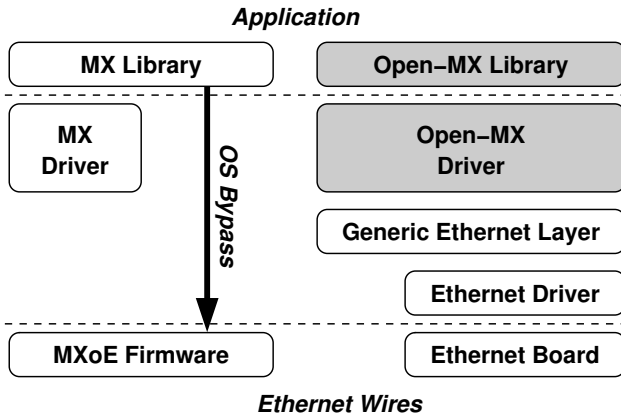
**Application**

| MX Library | Open–MX Library |
| MX Driver | Open–MX Driver |
| *OS Bypass* | Generic Ethernet Layer |
| | Ethernet Driver |
| MXoE Firmware | Ethernet Board |

**Ethernet Wires**

**Figure 1. Design of the native MX and generic Open-MX software stacks.**

We chose a similar design for the first OPEN-MX implementation while keeping the opportunity to diverge from this model later if necessary (see for instance Section 3.3). However, OPEN-MX cannot rely on any specific hardware features since only the generic ETHERNET software stack is available. OPEN-MX is thus implemented on top of the ETHERNET layer of the LINUX kernel and has to emulate the native MX firmware (see Figure 1). The OPEN-MX stack is therefore composed of a user-space library (similar to the native MX one), and a kernel driver which takes care of initialization, memory registration and also passing and receiving raw ETHERNET messages (*Socket Buffers*, skbuff).

Since these skbuffs are managed by the ETHERNET layer in the kernel, OS-bypass is impossible. While MX uses PIO from user-space, OPEN-MX relies on a system call which places the application data in a skbuff and passes it to the ETHERNET driver for sending. Section 3.1 will describe with more details how socket buffers are used and all data movements in OPEN-MX.

### 2.2 MX compatibility

OPEN-MX firstly aims at providing a high-performance message passing implementation over generic ETHERNET hardware. Secondly, it enables interoperability between this hardware and MYRI-10G hardware running the native MXoE stack. OPEN-MX interoperability with MX actually focusses on three points. Firstly, OPEN-MX offers the **same application programming interface** even though the underlying implementation differs. Any existing MX application such as MPICH or PVFS can be compiled on top of OPEN-MX.

Secondly, OPEN-MX is **wire-compatible** with MXoE

so that any generic hardware running OPEN-MX may be interconnected with a MYRI-10G hardware running MXoE (see the next Section). Finally, we are planning to have the OPEN-MX library expose the **same binary interface** so that existing MX applications may be relinked without being recompiled.

### 2.3 Wire protocol

Wire compatibility is a key feature of OPEN-MX, it is under experimentation at the Argonne National Laboratory to provide a PVFS2 transport layer between BLUEGENE/P compute and I/O nodes. The compute nodes running OPEN-MX are connected through a BROADCOM 10 gigabit ETHERNET interface to I/O nodes with a MYRI-10G interfaces running the native MXoE stack.

Apart from some control packets for connection, retransmission and rendezvous, the MX wire protocol is based on four different communication modes. *Large* messages ($>$ 32 kB) are processed through zero-copy after a rendezvous and memory pinning. The sender passes a window handle to the receiver which pulls data from it by 32 kB fragments. All other messages use a eager protocol, with either a single packet if the length is smaller than the MTU (4 kB) or up to 8 fragments. *Medium* messages (from 129 bytes) are copied in a statically pinned buffer on both sides and the hardware will transfer them using DMA. *Small* messages (below 128 bytes) are optimized on the send side by writing data in the NIC using a PIO. There also exists a ultra-optimized mode called *Tiny* where the control header (also written by PIO) contains the data (up to 32 bytes) so that a single PIO is used on the send side.

The wire protocol only enforces the message types, lengths and fragmentation. It does not force OPEN-MX to use a similar PIO or DMA implementation. We will detail in Section 3 how we translated the MX implementation into OPEN-MX.

### 2.4 Interrupt-driven model

The main specificity of the OPEN-MX design consists in the interrupt-driven model that the LINUX kernel ETHERNET layer enforces. While the native MX implementation may switch between interrupts and polling at runtime depending on the application behavior, OPEN-MX has to rely on a interrupt-driven model to receive events. Indeed, incoming packets are notified to OPEN-MX through a callback from the underlying ETHERNET drivers. No polling is possible.

This callback is responsible for reporting an event and storing the data in a user-space ring. The library then takes care of matching incoming messages with posted receives and of transferring the data to the application buffers. The

library may also sleep until the next event arrives. This sleep is however very cheap since it only adds a system call and a rescheduling of the process. It does not add any interrupt as it does on a native high-performance interconnect. It means that the latency does not vary significantly when switching from polling to sleeping on OPEN-MX while it jumps from roughly 2 to at least 5 $\mu$s with the native MX.

## 2.5 Retransmission and Progression

The OPEN-MX user-space library provides the same features that are available in the native MX library. It first matches incoming messages against posted receive and makes communication progress in case of *rendezvous*. Secondly, it manages retransmission by acknowledging or resending messages if necessary. This work is done when any function of the interface is invoked, which means the application or the middleware has to explicitly help OPEN-MX to progress. In the future, we also plan to add an optional thread to enforce progression and retransmission when the application does not invoke OPEN-MX often enough.

## 3 Memory Copy Avoidance

High-performance interconnects try to avoid memory copy as much as possible to reduce cache pollution, to improve bandwidth and to decrease CPU consumption. These zero-copy strategies are implemented thanks to the ability of both the hardware and driver to manipulate all the host physical memory in a very flexible way. Even if some of these advanced features have been transfered into the high-end ETHERNET hardware, it remains less flexible and thus brings several issues that are presented in this section.

### 3.1 Socket Buffer Management

The OPEN-MX stack emulates all the MX communication modes by using the ETHERNET layer of the LINUX kernel instead of the native MXoE firmware. While MX uses PIO and DMA to transfer data on the sender side, and DMA on the receiver side, OPEN-MX has to rely on the *Socket Buffers* (skbuff) that all ETHERNET drivers manipulate. The LINUX kernel provides two ways to associate some data with a skbuff: copying into a linear skbuff or attaching physical pages. We now explain how OPEN-MX tries to achieve high-performance even if the skbuff interface is limited.

**Sender side.** When the application submits a send request, the OPEN-MX stack may either copy the data or attach the buffer physical pages to a new skbuff. The underlying driver will then actually send it. This raises the question of which strategy is faster and less expensive. Since

attaching physical pages first requires to pin the application buffers into physical memory, this translates to the usual choice between copy and memory registration on high-speed interconnects.

MX switches from copy to registration when passing 32 kB buffers. We took the same decision for the first OPEN-MX implementation: small buffers are copied into a linear skbuff while large buffers are pinned and attached. Since most recent ETHERNET drivers and hardwares are able to setup DMA transfer from skbuffs, large buffers are then sent without any copy. Figure 2 summarizes the current OPEN-MX send implementation. It uses as many memory copies as the native MX, i.e. one for small and medium messages[1], and no copy for large messages.



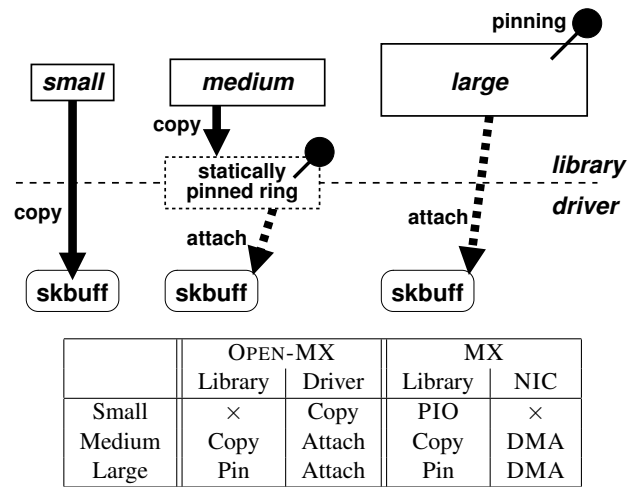| | OPEN-MX | | MX | |
|---|---|---|---|---|
| | Library | Driver | Library | NIC |
| Small | × | Copy | PIO | × |
| Medium | Copy | Attach | Copy | DMA |
| Large | Pin | Attach | Pin | DMA |

**Figure 2. Open-MX send strategies and comparison with MX.**

We actually observed that memory pinning and translating virtual addresses into physical is not very expensive in OPEN-MX. Regular LINUX routines such as `get_user_pages` are used for both pinning pages and translating addresses at once. This is far less expensive than memory registration than in native high-speed interconnects such as INFINIBAND because OPEN-MX does not have to store the address translations into the network interface. This way, pinning, translating and unpinning in OPEN-MX costs about 200 ns on an Intel XEON E5345 2.33 GHz processor while copying a page costs about 2 $\mu$s.
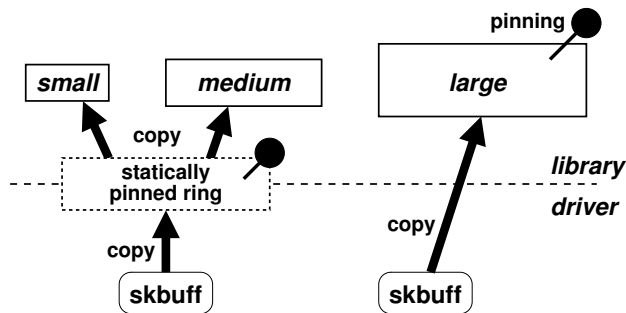
We are thus planning to diverge from the MX implementation by reducing the copy/pin threshold and start attaching pinned pages at 4 kB instead of 32 kB. Moreover, such a zero-copy strategy would enable a wider use of registration

---

[1] *Tiny* and *Small* messages are implemented the same way in OPEN-MX since there is no notion of single or multiple PIO per send. See Section 2.3 for details about the wire protocol.

cache techniques [11], possibly reducing even more the apparent cost of pinning.

**Receiver side.** While skbuff management on the sender side remains simple, the receiver side of OPEN-MX brings several important issues. Firstly, the OPEN-MX stack cannot decide where incoming data will be stored by the hardware. The ETHERNET driver is responsible for allocating its own skbuff and passing it to OPEN-MX once the hardware has filled it. It makes zero-copy impossible on the receiver side. This problem is actually not OPEN-MX specific since any other ETHERNET-based protocol suffers from the same issues. It led to the design of RDMA-enabled NIC with modified TCP/IP stacks such as iWARP [8] which lets the hardware place the data in the right receive buffer. Unfortunately, the interesting performance improvements achieved through these stacks cannot be obtained with regular ETHERNET hardware due to the aforementioned problem.

Secondly, the interrupt-driven model causes the incoming messages to be processed in the bottom half which is outside of the application context. Indeed, the underlying ETHERNET driver takes care of running its receive path in the bottom half (possibly with multiple instances in parallel when using some advanced drivers). Once a driver bottom half completed the skbuff, it immediately calls the OPEN-MX callback in the same context. It makes memory management complex since the target page table is not available.



| | OPEN-MX | | MX | |
|---|---|---|---|---|
| | Driver | Lib | NIC | Lib |
| Small | Copy | Copy | DMA | Copy |
| Medium | Copy | Copy | DMA | Copy |
| Large | Copy | Pin | DMA | Pin |

**Figure 3. Open-MX receive strategies and comparison with MX.**

To circumvent these problems, our first OPEN-MX implementation follows the MX receive model: small messages are received into a statically allocated user-space ring (the library will copy the data back to the application buffers after the matching), while large messages are only trans-

ferred after a *rendezvous* (hence when we know where to receive them). This solution enables the copy of any incoming skbuff at a known location that is already pinned in physical memory and thus available to the receive callback. However, it involves two memory copies for small messages and one for large messages, as summarized on Figure 3.

## 3.2  Memory Copy Offload with I/O AT

Having one or two memory copies on the receiver side is an important bottleneck, we will present the corresponding performance in the next section. We plan to investigate the offloading of memory copies through the I/O AT hardware (Intel *I/O Acceleration Technology*). It is known to help the receiver side of the TCP stack [12] by reducing the CPU overhead and cache pollution. We expect it to suit the current OPEN-MX model very well since it requires memory buffers to be pinned in physical memory which is already the case on our receiver side. We thus plan to use I/O AT to offload the copy between skbuff and either the user-space ring for small messages, or pinned large receive buffers.

However, the I/O AT model will bring some new issues regarding the notification of receive events since it requires the driver to explicitly poll for offloaded copy completions. We plan to also offload the copy of the corresponding event in the user-space ring so that the library retrieves it automatically after the data has been copied. Unfortunately, the case of an application sleeping until an event occurs will remain problematic. We might need to implement some periodic polling on the I/O AT subsystem in order to wakeup the application when needed.

## 3.3  Deporting the Matching in the Driver

Having two memory copies for small messages on the receiver side (see Section 3.1) is caused by the receive callback in the driver not knowing where the data should be copied in the application. Indeed, the target buffer is decided through the matching of incoming messages in the OPEN-MX user-space library, after the receive callback was invoked in the driver. We plan to investigate the deporting of the matching in the driver. If the receive callback can match incoming messages before copying them, it will be able to copy them directly at the right location in the application (instead of in the user-space ring).

However, once more, the receive callback is not guaranteed to be running inside the application memory context. Receive buffers will thus have to either be pinned in physical memory, or the matching and copy will have to be deported to a dedicated kernel thread. However, doing so might be bad for small message latency because of the cost of scheduling this new thread. A compromise might then be needed between additional copies for small mes-

sages and deported thread with less copies for medium and large messages.

Another problem that will have to be tackled is unexpected messages. Even if this model ensures that matching will occur in the right software layer, it cannot guarantee that a matching receive buffer has already been posted by the application. We will then have to decide between keeping the skbuff in the driver (with a possible memory starvation), passing it to a temporary buffer in user-space, or storing the receive event and letting retransmission resend the data later.

## 4 Performance Evaluation

We now present an early evaluation of OPEN-MX by looking at the MPI performance on dual quad-core machines[2]. The OPEN-MX driver was loaded into a LINUX 2.6.22 kernel.

### 4.1 Basic Performance

We measured the performance of OPEN-MX using the INTEL MPI benchmark (IMB) on various MPICH layers: MPICH-P4 for regular TCP/IP networking, MPICH-MX compiled for OPEN-MX, and the regular MPICH-MX when MX is available on the machine.
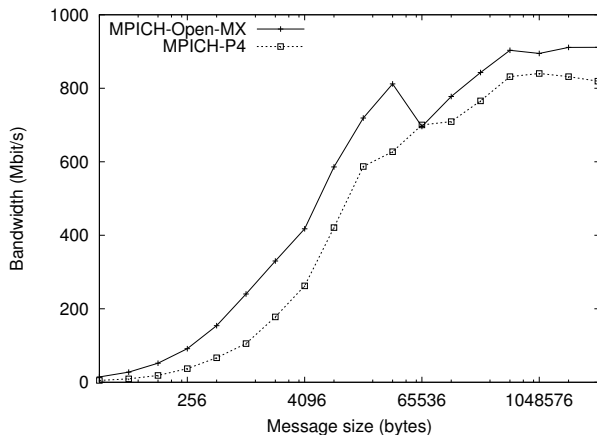


**Figure 4. Performance of MPICH-Open-MX and P4 on Gigabit Ethernet with the Intel MPI Benchmark PingPong.**

Figure 4 presents the performance of the IMB PingPong between 2 machines connected through gigabit ETHERNET interfaces[3]. Both P4 and OPEN-MX layers exhibit a good

---

[2] Two Intel XEON E5345 2.33 GHz processors.
[3] Broadcom NetXtreme II BCM5708 Gigabit, `bnx2` driver.

PingPong performance, but the latter achieves 20 % more bandwidth on average, up to roughly the line rate. On the latency side, both networks reach about 16 $\mu$s.
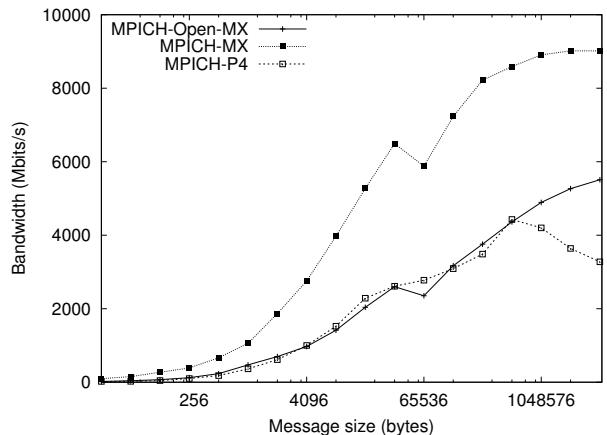


**Figure 5. Performance of MPICH-Open-MX, MX and P4 on Myricom Myri-10G Ethernet interfaces with the Intel MPI Benchmark Ping-Pong.**

Figure 5 presents the same benchmarks over a 10 gigabit ETHERNET network. We chose the MYRICOM MYRI-10G (`myri10ge` 1.3.1 driver) interfaces since it also enables comparison with the native MXoE stack. The MXoE latency is 2.4 $\mu$s, while OPEN-MX reaches 11.8 $\mu$s and P4 12.1 $\mu$s. While MXoE nearly reaches the line rate, MPICH-P4 saturates around 500 MB/s and OPEN-MX achieves about 700 MB/s.

We observed similar behaviors with the IMB SendRecv benchmark (bidirectional communication scheme). It has to be noted that OPEN-MX is slightly slower than P4 on medium messages (especially near 32 kB). This is probably caused by the additional memory copy on the receiver side (see Section 3.1). We expect to largely improve this by reworking the receive model as explained earlier.

The OPEN-MX performance is limited by the fact that the underlying `myri10ge` driver uses a single bottom-half and thus enforces the processing of all OPEN-MX incoming packets on a single core, which is overloaded. We are planning to look at multi-slices ETHERNET drivers where multiple instances of the bottom half will enable parallelization of the OPEN-MX receive path.

The current OPEN-MX design already supports such a parallelization since very few locks are involved on the receive path. Indeed, apart from the fast RCU locking [5] that protects against early endpoint closing or interface removal, at most two locks are taken for each incoming packet. The first one is used to serialize the in-order notification of

events to the user-library. The second one is associated with large messages handles (whose retransmission is taken care of by the driver) to manage the set of pending and missing fragments. We are in the process of reducing their holding time as much as possible to provide a good scalability when OPEN-MX runs on top of multi-slices drivers with parallel bottom halves.

## 4.2 Ethernet driver tuning

The interrupt-driven model that the LINUX ETHERNET layer enforces for OPEN-MX (see Section 2.4) raises some important latency issues since it prevents active polling from the application. Moreover, due to the cost of interrupt processing (a couple microseconds), most ETHERNET drivers use *Interrupt Coalescing* to defer packet processing in the host by notifying multiple events through a single interrupt. While most IP applications do not care about latency, OPEN-MX suffers from this and thus requires to configure interrupt coalescing to achieve low latency.

| Interrupt coalescing | On | Off |
|---|---|---|
| MXoE | 2.4 | 2.4 |
| P4 | 19.5 | 10.1 |
| OPEN-MX | 75.7 | 9.2 |

**Table 1. Interrupt coalescing influence on MPI latency (in microseconds) on Myri-10G interfaces.**

Table 1 presents the MPI latency (measured with IMB PingPong) between 10 gigabit interfaces. It shows how disabling interrupt coalescing is important for ETHERNET networking, while MXoE avoids the problem thanks to the application being able to busy poll for receive events.

Unfortunately, disabling coalescing entirely is generally not a good idea since it increases the host load. One satisfying configuration seems to be setting the coalescing to the network raw latency. It prevents the OPEN-MX latency from increasing, and also keeps some coalescing enabled to reduce the load. In the future, we plan to investigate the implementation of a protocol-dependent coalescing in the ETHERNET firmwares and drivers so that OPEN-MX requirements do not impact on other networking stacks such as TCP/IP. To go further, we even imagine disabling coalescing only for latency-sensitive packets (OPEN-MX small packets).

## 5 Related Works

Several research projects did target high-performance message passing over ETHERNET in the past. The most popular one is GAMMA [2] which only works on limited hardware since it uses a modified driver. MULTIEDGE [4] uses a similar design on recent 1 and 10 gigabit hardware and thus achieves good bandwidth, but it remains limited to a pretty high latency. EMP [9] goes even further by modifying the firmware of some programmable boards to achieve better performance. However, these implementations do not support any 10 gigabit hardware so far while OPEN-MX relies on the generic ETHERNET layer of LINUX and may thus use any hardware.

MPI/QMP [1] uses a OPEN-MX-like model, based on M-VIA, to achieve large bandwidth over multiple regular ETHERNET connections. PM/ETHERNET-HXB [10] offers a similar design and supports trunked ETHERNET connections. They both achieve interesting performance thanks to multiple underlying ETHERNET connections, but are not designed for single high-performance connections such as MYRI-10G. OPEN-MX is designed to efficiently use high performance ETHERNET hardware and may also transparently use a trunked connection to aggregate multiple connections.

iWARP has been designed for high-performance TCP/IP networking, not only for clusters but also for long-distance connections where the CPU overhead should be as low as possible. It uses an IB-like model to provide RDMA semantics at the application level within the TCP stack. iWARP achieves very good performance on RDMA-enabled NICs [8]. A software-only implementation is also available for generic hardware [3] but its performance is seriously limited since iWARP has been designed mostly for RDMA-enabled NICs and also because it suffers from memory copies as OPEN-MX does. We expect future OPEN-MX implementations to achieve better performance since it does not suffer from the intrinsic limitations of the TCP stack and RDMA-enabled NIC model, and can be optimized for cluster-only environments.

## 6 Conclusion

In this paper, we presented the design and implementation of the OPEN-MX software stack. It aims at providing a high-performance message passing layer over any generic ETHERNET hardware. OPEN-MX also enables interoperability between such hardware and the MYRICOM's native MXoE stack.

Our first OPEN-MX implementation[4] is based on the MX stack with the firmware emulated in a driver in the LINUX kernel. It already provides interesting performance and is able to run MPICH-MX, MPICH2-MX and PVFS2 applications thanks to the application level com-

---

[4]The OPEN-MX source code is available for download from its homepage at `http://open-mx.org/`.

patibility. We expect most existing applications that were designed for MX to work flawlessly on OPEN-MX soon.

The current OPEN-MX design leaves room for large performance improvements. We are profiling the stack under various benchmarks to locate the bottlenecks such as memory copies and lock contention. The current sender side remains very simple and enables zero-copy for large messages at in MX. We firstly plan to diverge from the MX design for smaller messages to achieve better performance by relying more on memory pining since it is not as expensive as on high-performance dedicated interconnects. Secondly, we will investigate the deporting of the matching of incoming messages into the driver to reduce memory copies on the receiver side. I/O AT will also be studied since it exhibits an interesting model that should suit our receiver side very well. Thirdly, we are also implementing a pin-down cache [11] to reduce pinning overhead and increase large message performance.

We are also working with MYRICOM on the next MX wire specifications so that OPEN-MX can match line rate performance more easily. For instance, when wire compatibility with MXoE is enabled, the OPEN-MX bandwidth currently drops from 700 to about 400 MB/s due to the limited usage of the MTU (only 4 kB data frame while 9 kB are available).

Finally, we also plan to look at how to improve the OPEN-MX latency. A configurable per-ETHERNET-type or per-packet-type interrupt coalescing should prevent from increasing the host load uselessly if only OPEN-MX small packets requires it. In the long term, a challenging idea would be to modify both software and hardware sides to add a generic way to poll ETHERNET in order to fill the latency gap with high-speed interconnects.

## References

[1] J. Chen, W. Watson III, R. Edwards, and W. Mao. Message Passing for Linux Clusters with Gigabit Ethernet Mesh Connections. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, Denver, CO, 2005.

[2] G. Ciaccio and G. Chiola. GAMMA and MPI/GAMMA on gigabitethernet. In *Proceedings of 7th EuroPVM-MPI conference*, Balatonfured, Hongrie, Sept. 2000.

[3] D. Dalessandro, A. Devulapalli, and P. Wyckoff. Design and Implementation of the iWarp Protocol in Software. In *Proceedings of PDCS '05*, Phoenix, AZ, Nov. 2005.

[4] S. Karlsson, S. Passas, G. Kotsis2, and A. Bilas. Multi-Edge: An Edge-based Communication Subsystem for Scalable Commodity Servers. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, page 28, Long Beach, CA, Mar. 2007.

[5] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *Ottawa Linux Symposium*, July 2001. Available: `http://www.linuxsymposium.org/2001/abstracts/readcopy.php`.

[6] Myricom, Inc. *Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet*, 2006. `http://www.myri.com/scs/MX/doc/mx.pdf`.

[7] The Parallel Virtual File System, version 2. `http://www.pvfs.org/`.

[8] M. J. Rashti and A. Afsahi. 10-Gigabit iWARP Ethernet: Comparative Performance Analysis with Infiniband and Myrinet-10G. In *Proceedings of the International Workshop on Communication Architecture for Clusters (CAC), held in conjunction with IPDPS '07*, page 234, Long Beach, CA, Mar. 2007.

[9] P. Shivam, P. Wyckoff, and D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceeding of Supercomputing ACM/IEEE 2001 Conference*, Denver, CO, Nov. 2001.

[10] S. Sumimoto, K. Ooe, K. Kumon, T. Boku, M. Sato, and A. Ukawa. A Scalable Communication Layer for Multi-Dimensional Hyper Crossbar Network Using Multiple Gigabit Ethernet. In *ICS '06: Proceedings of the 20th International Conference on Supercomputing*, pages 107–115, Cairns, Australia, 2006.

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

[12] K. Vaidyanathan and D. K. Panda. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, San Jose, CA, Apr. 2007.