# Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX

Brice Goglin

# Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX

Brice Goglin
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. Open-MX works on all Ethernet hardware, but it suffers from expensive memory copy requirements on the receiver side due to the hardware's inability to deposit messages directly in the target application buffers.**

**This article presents the implementation of an asynchronous memory copy offload in the Open-MX stack thanks to Intel I/O Acceleration Technology. The overlapping of large message fragment copies with the processing increases the receive throughput by 30 % while reducing the CPU usage by up to 40 %. It enables Open-MX to reach 10 gigabit/s Ethernet line rate for large messages.**

**Open-MX large intra-node communication also benefits significantly from the I/OAT hardware since the performance of its one-copy-based local communication mechanism is almost doubled by using blocking I/OAT memory copies. By combining all these optimizations, the Open-MX large message performance on top of 10G hardware is now able to bridge the gap with the native Myrinet Express stack.**

## I. Introduction

The emergence of 10 gigabit/s ETHERNET hardware raises the usual question of whether it may replace dedicated hardware such as INFINIBAND [11] or MYRI-10G [12] 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 they are still restricted to specifically tuned configurations and most of the time imply a high CPU load.

Meanwhile, ETHERNET appears as an interesting networking layer within local networks for various protocols such as FIBRECHANNEL [5] and ATA [1]. 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 [16]. However, these technologies still require dedicated interfaces on the nodes. It makes them impossible to use on regular hardware.

OPEN-MX [7] is a message passing stack implemented on top of the ETHERNET software layer of the LINUX kernel. It aims at providing high-performance communication over any generic ETHERNET hardware using the wire specifications and the application programming interface of *Myrinet Express* [13]. OPEN-MX 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 [15] in the BLUEGENE/P systems.

However, as any regular ETHERNET-based protocol, OPEN-MX suffers from memory copies on the receiver side. Indeed, the hardware cannot deposit incoming packets in the destination buffer. The receive stack must copy the data after the matching. One approach to solve this problem is to rely on RDMA-enabled NICs with modified TCP/IP stacks such as iWARP [17] which lets the hardware place the data in the right receive buffer. However, such a solution would prevent OPEN-MX from being used in commodity environments with regular ETHERNET NICs.

We thus propose in this paper to keep the current OPEN-MX design which needs the aforementioned memory copies, but try to offload them thanks to the INTEL I/OAT hardware capabilities. Recent INTEL platforms [18] are indeed able to perform asynchronous copies in the memory chipset while keeping the CPU available for computation. This feature is already widely used to improve the TCP/IP stack [23] in various networking-intensive applications.

The paper is organized as follows. Section II provides a brief overview of the OPEN-MX software stack, its memory copy requirements and shows how I/OAT may solve these issues. In Section III, we discuss the implementation of asynchronous and synchronous memory copy offload with I/OAT in OPEN-MX. Section IV presents micro-benchmarks and MPI performance evaluations showing that this new implementation increases the OPEN-MX large message performance for local and network communication by up to 30 % while reducing the CPU overhead significantly on the receiver side.

## II. Memory Copies in Open-MX

In this section, we present a brief overview of OPEN-MX and how memory copies are limiting its performance on the receiver side, before introducing INTEL I/OAT as a solution to this problem.

### A. Overview of Open-MX

The OPEN-MX stack aims at providing high-performance message passing over any generic ETHERNET hardware. It exposes the *Myrinet Express* API (MX) to user-space applications. Existing middlewares such as MPICH2-MX, OPEN MPI [6] or PVFS2 [15] already successfully run unmodified on top of it. OPEN-MX is also interoperable with

This research is supported by a collaboration between INRIA and Myricom, Inc.

hosts running the native MX stack over ETHERNET (MXoE). This wire compatibility is a key feature of OPEN-MX. It is under experimentation at 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 interface running the native MXoE stack.

To achieve these goals, OPEN-MX was first designed as an emulated MXoE firmware in the LINUX kernel module [7]. This way, legacy applications built for MX benefit from the same abilities without the MYRICOM hardware neither the native MX software stack (see Figure 1).

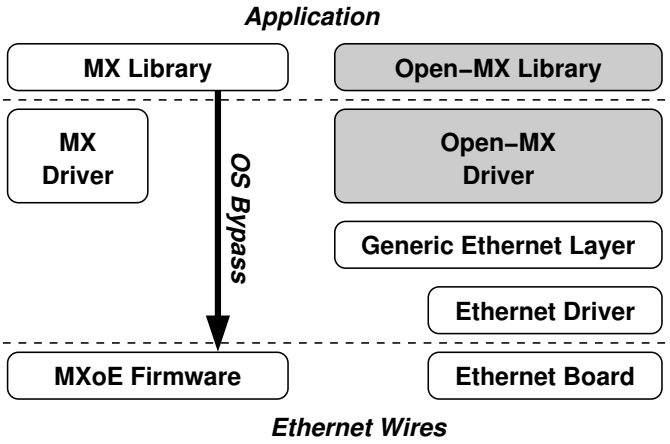


Fig. 1. Design of the native MX and generic Open-MX software stacks.

However, the features that are usually implemented in the hardware of high-speed networks are obviously prone to performance issues when emulated in software. Indeed, portability to any ETHERNET hardware requires the use of a common low-level programming interface to access drivers and NICs. The LINUX kernel exposes a generic ETHERNET access software layer which manipulates *Socket Buffers*. These *skbuffs* are allocated by upper layers for sending, and a callback is invoked to process incoming packets stored by the underlying NICs and drivers in other skbuffs.

As explained in [7], the OPEN-MX sender side does not actually suffer much from this generic hardware design since it is able to attach user-level physical pages to skbuffs in order to achieve zero-copy. *OS-bypass* is not possible since the ETHERNET access layer requires a system call for every communication. Fortunately, the cost of system calls decreased dramatically in the recent years, making OS-bypass much more negligible than before[1].

Thanks to this design, the OPEN-MX stack is able to achieve a very high throughput on the sender side, saturating 10 gigabit/s ETHERNET links. However, the receiver side is far more difficult to implement efficiently since generic hardware cannot easily deposit data directly in user-level application buffers. This problem is discussed further in the next section.

[1]The basic cost of a system call is close to 100 nanoseconds on recent INTEL processors and even less on AMDs.

### B. Open-MX Receiver Side Memory Copies

Most ETHERNET hardware implement their receive path using some sort of circular ring of buffers in the host memory. The driver keeps adding newly allocated skbuffs to the ring, while the NICs basically consumes them in order. It fills them by DMA (*Direct Memory Access*) and notifies the driver to pass them to upper layers. This design intrinsically makes zero-copy receive impossible since the driver cannot predict which packet will arrive next and thus cannot provide the NIC with the corresponding receive buffer at the right time.

This problem is not OPEN-MX specific since any other ETHERNET-based protocol suffers from the same issues. It led to the design of RDMA-enabled NICs with modified TCP/IP stacks such as iWARP [17] which lets advanced 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 above problem.

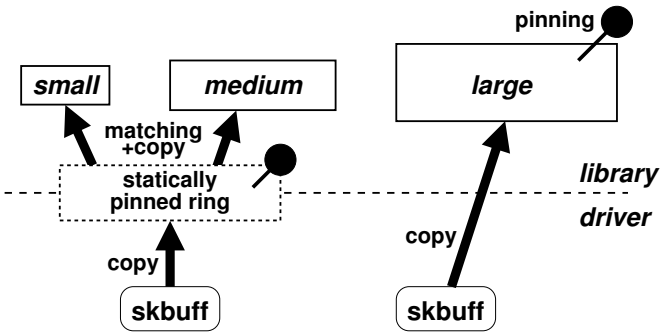


Fig. 2. Open-MX receive strategies.

Once the skbuff has been filled by a regular ETHERNET NIC, the driver passes it to the low-level networking layer of the operating system which schedules the execution of a *Bottom Half* (BH). It is the heavy part of the interrupt processing that runs after the actual short interrupt handler. It launches the OPEN-MX receive callback which needs to copy the data to its destination.

It copies small and medium messages into a statically allocated user-space ring, and let the user library move the data back to the application buffers after the matching. For large messages, the callback already knows the final destination buffer since the data transfer only occurs after a *rendezvous* between the user-space libraries. 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, in the end, the overall OPEN-MX stack requires one additional memory copy compared to MX: two for small messages and one for large messages, as summarized on Figure 2.

This model puts severe pressure on the CPU and memory bus and thus limits receiver side performance to at most 7 Gbit/s on 10G hardware [7]. Figure 3 presents a performance

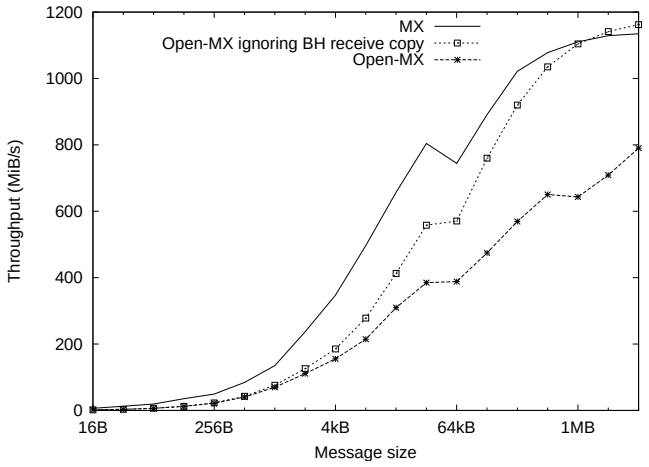


Fig. 3. Expected Open-MX performance improvement when removing the copy in the receive callback (invoked by the bottom half interrupt handler of the driver).

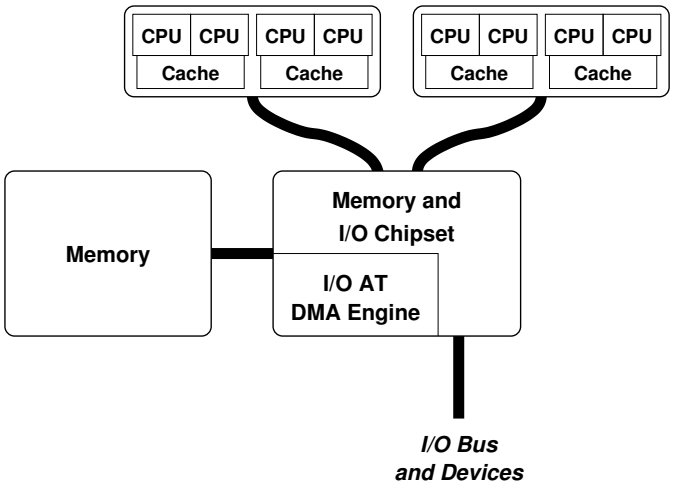


Fig. 4. Description of the I/OAT hardware architecture in our dual quad-core Xeon hosts.

comparison of native MX and OPEN-MX ping-pong performance between two MYRI-10G NICs connected without any switch. MX achieves up to 1140 MiB/s for large messages while OPEN-MX saturates near 800 MiB/s[2]. However, for performance prediction purpose, if we disable OPEN-MX copies in the receive callback invoked by the driver bottom half (BH), line rate appears to be achievable. This motivates our following work on offloading these memory copies on dedicated hardware in order to achieve line rate performance.

### C. Overview of I/OAT

INTEL *I/O Acceleration Technology* (I/OAT) is a set of hardware features tailored to improve networking performance in data centers [18]. For instance, it includes *Direct Cache Access* which warms up the processor cache with network data when a new packet will have to be processed soon. The most interesting feature is the ability to perform asynchronous memory copy in the background thanks to a DMA engine integrated in the memory chipset (see Figure 4). It enables overlapped memory copies with no CPU usage neither cache pollution contrary to the regular `memcpy` method.

There are still few users of the DMA engine subsystem in LINUX since its usage is limited to restricted circumstances. Indeed, since the hardware manipulates DMA addresses, the corresponding pages have to be pinned down in memory. Therefore, the DMA engine programming interface [9] is mainly used by LINUX kernel subsystems such as the TCP/IP receiver stack. It offloads memory copies while the user process sleeps during the `recv()` system call until there is enough data to receive. The CPU usage is reduced and the network throughput is improved for various applications such as PVFS file transfers [23].

I/OAT copy offload may also be interesting to user-level application if the interface is exposed to user-space with the required memory pinning mechanism. It would enable efficient intra- and inter-process communication [22], [21].

Given the aforementioned results, I/OAT looks very interesting to improve the OPEN-MX receive stack by offloading memory copy between incoming skbuffs and the target data buffers. The requirement to pin memory buffers that would be passed to the I/OAT DMA engine is indeed already met since all source skbuffs are pinned by the kernel while OPEN-MX already pins its receive buffers (see Figure 2).

## III. DESIGN OF COPY OFFLOAD IN OPEN-MX

In this section, we discuss the design of a copy offload model in OPEN-MX on top of the I/OAT DMA engine interface. We first discuss asynchronous memory copies for large message receive before looking at synchronous copies for medium messages and shared-memory communication.

### A. Asynchronous Copy for Large Receive Fragments

The OPEN-MX receiver side processes incoming packets within its specific callback that the operating system bottom half invokes. Packet headers are first decoded to find the corresponding user endpoint and application. Then the data are copied into the target buffer, either a previously pinned large memory region, or a statically pinned shared buffer where the user-library will read data from. Once the copy is done, an event is written in a shared event ring to notify a receive completion to the user-library.

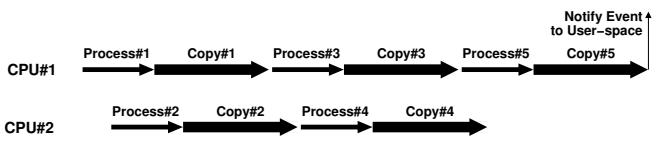


Fig. 5. Timeline of a 5-fragments message receive on 2 processors without I/OAT support. Each fragment is processed and copied before releasing the CPU and being able to process another incoming fragment. The last fragment callback notifies receive completion to user-space.

[2]The actual data rate of 10 Gbit/s ETHERNET is 9953 Mbit/s = 1244 MB/s = 1186 MiB/s.

Fortunately, large messages are entirely managed by the OPEN-MX driver and there is only an event to report to user-space when the very last fragment is received (see Figure 5). It enables asynchronous copy of all fragments except the last one. The I/OAT DMA engine is thus going to be used to receive OPEN-MX large messages as follows:

During the processing of any large message fragment, the regular `memcpy` is replaced with the submission of the corresponding I/OAT asynchronous copies. No additional memory pinning is required since the source skbuff and destination buffer are already pinned in the no-I/OAT path. If the current fragment is the last one, the OPEN-MX callback waits for all previous copies to be completed for this large message. Then, it reports a completion event to user-space as usual. If the current fragment is not the last one, there is nothing to do after submitting asynchronous copies, the CPU is released immediately.

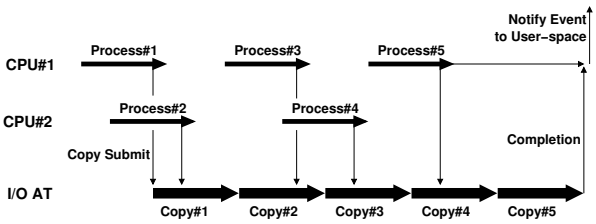


Fig. 6. Timeline of a 5-fragments message receive on 2 processors with I/OAT offload of asynchronous copies. The last fragment callback waits for the completion of all asynchronous copies before notifying receive completion to user-space. All other fragment callbacks release the CPU right after processing it and submitting the asynchronous copy.

As summarized on Figure 6, this model enables full overlap of the copies of all but the last fragment data for large messages. Processing packets with I/OAT enabled is actually slightly more expensive due to the need to submit asynchronous copies. However, the CPU is released much faster. And the I/OAT hardware performs memory copies faster as explained later in Section IV-A. The overall receive cost is thus dramatically reduced.

### B. Keeping Track of Resources

Overlapping asynchronous memory copies may lead to a large pool of skbuffs being queued for copy and not being freed before the last fragment arrives. To prevent memory starvation in case of very large messages, the OPEN-MX driver has to carefully keep track of pending fragment copies and release them periodically.

Since the MX large message model involves the periodic requesting of new fragments to the remote side[3], a resource cleanup routine is invoked when a new request is sent. This routine is also invoked when the retransmission timeout expires in case of packet loss.

The cleanup routine polls the I/OAT DMA hardware once for copy completions and releases the corresponding skbuffs. This way, resources are freed early and the number of pending skbuff copy is bounded.

### C. Offloading Synchronous Copies

While OPEN-MX large messages are processed by the driver after a rendezvous handshake in the library, small and medium messages are matched and reassembled directly in the user-space library. It means that every fragment of these messages requires the driver to notify a receive event to user-space, making all of their copy explicitly synchronous. We plan to rework the OPEN-MX receiver side by moving the matching into the driver so that a single receive event will be reported to user-space for each message, instead of one per fragment. It would enable the overlapping of multi-fragment copies as already done for large messages.

For now, small and medium message data have to be copied synchronously. It removes the ability to overlap these copies but given the expected high-performance of the I/OAT hardware and its ability not to pollute the cache, OPEN-MX has been made able to optionally offload these copies as well.

Surprisingly, such high-performance synchronous copies may also be applied to OPEN-MX shared-memory subsystem. Indeed, OPEN-MX local communication is based on a system call where a direct copy is performed between the source process address space into the target. A comparable model has been presented in [21] as an extension to the MVAPICH MPI middleware. This design is actually nicely integrated into the OPEN-MX stack since all communications, either local or through the network, are managed by the driver through the same commands, and they return the same events to the user-space library. Offloading these local copies, which may be very large, introduces a large room for improvement thanks to I/OAT DMA engine performance.

## IV. PERFORMANCE EVALUATION

We now present a performance evaluation of our copy offload implementation in the OPEN-MX receive stack. We ran our experiments on machines based on two quad-core INTEL processors[4] and the INTEL 5000X chipset which provides an I/OAT DMA engine. MX communication relies on MYRI-10G NICs with MXoE 1.2.4 on LINUX 2.6.23 kernel, while OPEN-MX uses these boards in native ETHERNET mode using the myri10ge driver 1.4.1.

We first present a brief performance analysis of the I/OAT asynchronous copy to explain how the OPEN-MX stack should be tuned to make full benefit from it. Then we detail the impact of our asynchronous copy offload on the performance of OPEN-MX large message receive as well as on the corresponding CPU usage. We then look at synchronous copy for network and local communication performance. And finally, we present the overall OPEN-MX performance improvement thanks to the above optimizations by looking at their impact on the INTEL MPI Benchmarks.

[3]Two pipelined blocks of 8 fragments are outstanding for each large message under normal circumstances.

[4]2.33 GHz XEON E5345 "Clovertown".

### A. I/OAT Micro-benchmarks

Offloading memory copies on the I/OAT hardware requires the actual management of offloaded copies to be cheaper than a usual synchronous `memcpy` performed by the processor. To evaluate the minimal chunk length to offload a copy, we first measured the submission time on our machine to about 350 nanoseconds. This is caused by the necessity to provide a copy descriptor to the hardware.

Then, since the completions are reported by the hardware in host memory in order, it is actually very cheap to check for the completion of all pending copies through a simple memory read. The per-copy completion cost appears to be negligible.

Given that the processor copy rate is about 1.6 GiB/s rate, 600 bytes may be copied with `memcpy` (2 kB if in the cache) before I/OAT copy offload becomes interesting. These first results imply that OPEN-MX should not offload small chunk memory copies.

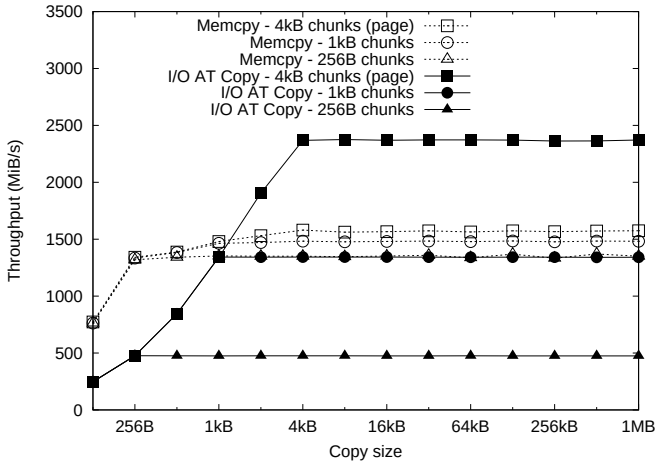


Fig. 7. Comparison of pipelined memcpy and I/OAT copy performance using 256 bytes, 1 kB and 4 kB chunks.

Given that the I/OAT hardware manipulates DMA addresses, most copies are actually split into page-aligned chunks. Figure 7 presents the raw performance of I/OAT offloaded copy and `memcpy` when splitting data streams into various chunk sizes. This splitting does not imply much `memcpy` performance degradation since the initialization time is very small. But it has a huge impact on I/OAT performance since more copy descriptors have to be submitted to the hardware. It first shows that even large copies should not be offloaded to the I/OAT hardware unless the chunk size is about 1 kB. Fortunately, when using larger chunks, such as full 4 kB pages, I/OAT is able to sustain about 2.4 GiB/s while `memcpy` saturates near 1.5 GiB/s. Actually, if the data fits in the cache, the `memcpy` performance may reach up to 12 GiB/s. I/OAT does not benefit from the cache anyhow but has the advantage of not polluting the cache for very large copies.

The conclusion of these micro-benchmarks is that OPEN-MX should not offload memory copies unless the whole message is large (smaller copies may be handled better through the local cache) and unless all its fragments are at least about one kilobyte long (the submission of smaller fragments to the I/OAT hardware would be too expensive). Fortunately, such very small fragments may actually only be involved in OPEN-MX if the application uses highly-vectorial buffers.

In the regular case, most user buffers are virtually contiguous while most incoming skbuffs are page-based. So, on average, most OPEN-MX copies should consist on one or two chunks per page, causing the I/OAT hardware to be helpful as soon as a large messages are processed. We have empirically chosen to offload memory copies of fragments larger than 1 kB for messages larger than 64 kB.

### B. Overlapped Asynchronous Copy Performance

We now look at the basic OPEN-MX receiver side performance improvement thanks to asynchronous copy offload for large messages implemented as detailed in Section III-A.

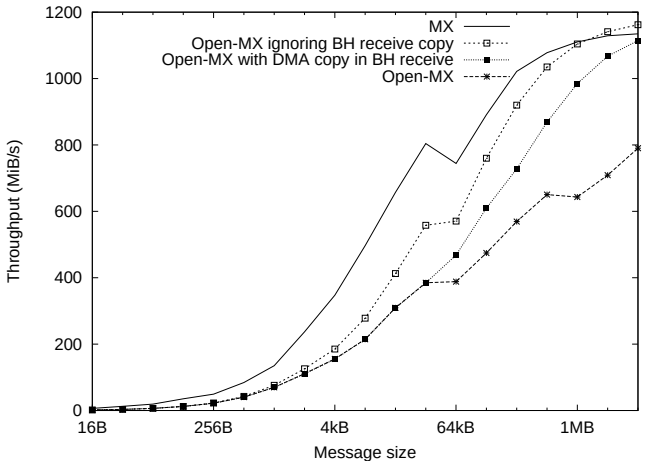


Fig. 8. Comparison of a ping-pong performance improvement using I/OAT and the expected performance with bottom half copy ignored.

*1) Throughput:* Figure 8 presents the ping-pong performance and compares it to the expected one as explained earlier on Figure 3. We observe up to 50 % higher throughput for large messages (more than 32 kB) with I/OAT asynchronous copy offload. OPEN-MX is even able to saturate the network link with multi-megabyte messages since it now reaches 1,114 MiB/s while the line rate is 1,186 MiB/s.

The throughput remains up to 26 % below the expected level with receiver-side copy ignored for 256 kB messages. This is caused by the I/OAT management cost not being negligible and its raw copy performance is not totally exploited for such message sizes. However, even for these relatively small message sizes, OPEN-MX performs more than 20 % better thanks to the overlap of asynchronous copy of large message fragments.

*2) CPU Usage:* Figure 9 presents the CPU usage of the OPEN-MX stack receiving a unidirectional stream of large messages. It shows that the regular `memcpy`-based large receive implementation saturates one of the 2.33 GHz XEON cores up-to 95 %. The user-space CPU load is mostly negligible since it only consists of posting requests to the driver and

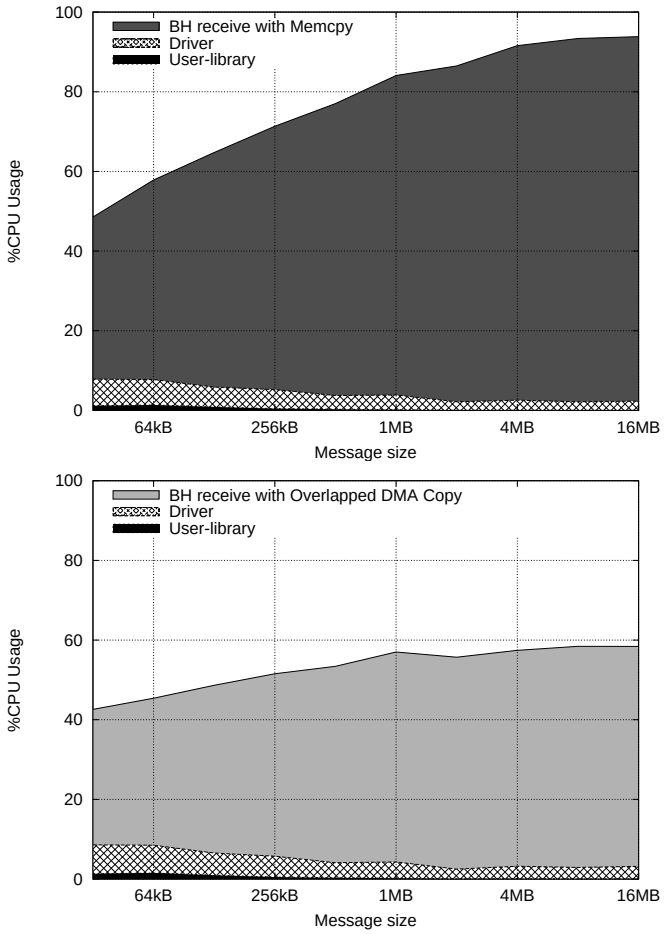


Fig. 9. CPU usage of the Open-MX library, driver command processing, and bottom half receive processing while receiving a stream of synchronous large messages with and without overlapped asynchronous copy offload.

then waits for a completion event. The driver time is higher because it involves memory pinning during a system call prior to the data transfer. Both user and driver times do not depend on I/OAT being enabled but the corresponding CPU usage percentages are higher with I/OAT enabled since the overall communication is faster. The actual small CPU availability comes from the rendezvous handshake before the large data transfer occurs since the processors are idle during the round-trip on a network.

When enabling our I/OAT overlapped copy model using I/OAT, the overall CPU usage drops from 50 to 42 % for 32 kB messages, and from 95 % to 60 % for multi-megabyte messages. So, in addition to largely improving the OPEN-MX throughput for large messages up to the 10 gigabit/s ETHERNET line rate, the I/OAT hardware also dramatically reduces the host load, eliminating the CPU as the bottleneck.

### C. Synchronous Copy Performance

We now look at the impact of synchronous copy offload on the I/OAT hardware. This model is not expected to improve CPU usage a lot since the OPEN-MX stack needs to wait for the completion of all submitted copies and the I/OAT hardware cannot raise an interrupt and wake up a waiting task. Since the I/OAT performance looks reasonably predictable, we plan to evaluate the idea of going to sleep until we expect the offloaded copy to complete soon. This way, the CPU being released may enable better overlap with similar copy performance. However, we did not implement this strategy yet and rely on busy polling of the I/OAT hardware with no overlap for now.

Since the raw I/OAT performance for large copies is almost twice higher than regular copies (see Section IV-A), a performance improvement of OPEN-MX might still be observed. We implemented synchronous copies in the medium message path as explained in Section III-C and noticed a performance degradation. The reason relies in OPEN-MX requiring all 4 kB medium fragment copies to be synchronous and I/OAT performance for such small copies not being interesting. We plan to evaluate the deporting of the OPEN-MX matching from the user library into the kernel in order to be able to overlap most medium message fragment copies as we do for large messages.

Fortunately, as explained in Section III-A, I/OAT copy offload may help for synchronous copies during shared-memory communication in OPEN-MX. We implemented this model in our one-copy based local communication within the OPEN-MX driver. Figure 10 shows the observed performance during a ping-pong between two local processes. As expected the OPEN-MX I/OAT-based performance increases dramatically after the large message threshold (32 kB) since the whole data transfer may then be copied at once. It then reaches 2.3 GiB/s.

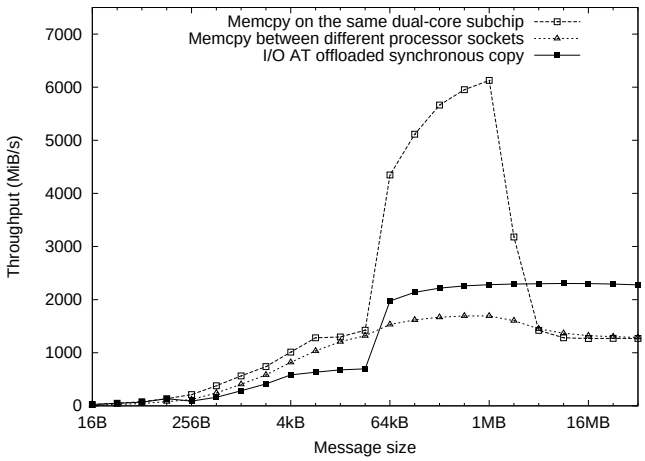


Fig. 10. Performance of Open-MX one-copy-based shared-memory communication with I/OAT offload of synchronous copies.

The usual OPEN-MX shared-memory communication with `memcpy` may reach up to 6 GiB/s for messages smaller than 1 MB thanks to the same buffer being reused and the XEON L2 cache being shared by the processes. However, when the processes do not shared any cache or when the message size increases above the shared cache size, the performance drops to 1.2 GiB/s while the I/OAT based implementation keeps the same 80 % higher throughput.

The OPEN-MX stack thus now enables I/OAT copy for shared-memory communication for large messages (beyond 1 MB). We are planning to reduce this threshold if the processes do not share any cache. We are also looking at reducing the large message threshold for local communication since their performance may be higher as soon as the message size exceeds 8 or 16 kB, even with I/OAT disabled.

### D. Overall Performance

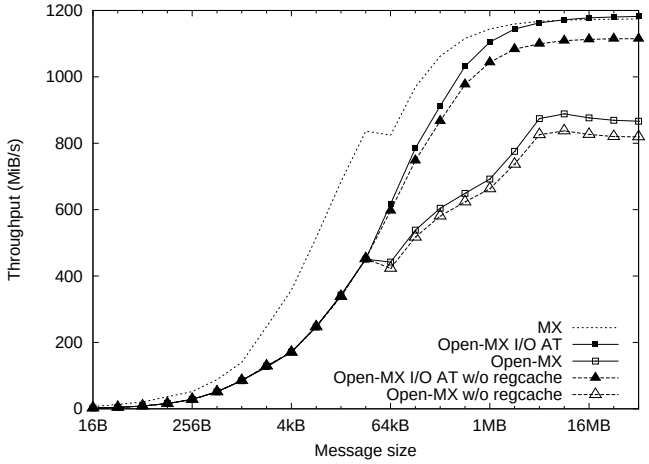


Fig. 11. Intel MPI Benchmarks PingPong throughput with MXoE and Open-MX, with I/OAT and registration cache enabled or not.

We now take a wider look at the impact of I/OAT copy offload of the OPEN-MX performance using the INTEL MPI Benchmarks (IMB [10]). We use the MPICH-MX layer which builds directly on top of OPEN-MX thanks to the API compatibility, and compare it to MPICH-MX over the native MXOE on the same MYRI-10G hardware. Figure 11 presents the IMB PingPong performance and shows that OPEN-MX is now able to reach the same performance than MX for large messages, close to the 10G ETHERNET line rate. Our I/OAT implementation brings an average 30 % throughput increase while dramatically decreasing the CPU load as explained earlier.

We also look at the usual *registration cache* optimization which defers deregistration to avoid the cost of pinning for each communication [20]. Such a cache is often considered as a very important optimization but we observe that it is actually here much less important than I/OAT copy offload. This may be related to OPEN-MX having a cheap registration as it does not require the registered address translations to be stored in any NIC as high-speed networks do. But it also emphasizes the importance of our copy offload design.

Figure 12 finally presents the normalized performance of all IMB tests. We observe a 24 % overall performance improvements for 128 kB messages, reaching an average 68 % of the MXOE performance. We are still trying to understand why our I/OAT strategy appears to be slowing down ReduceScatter on 2 processes per node.

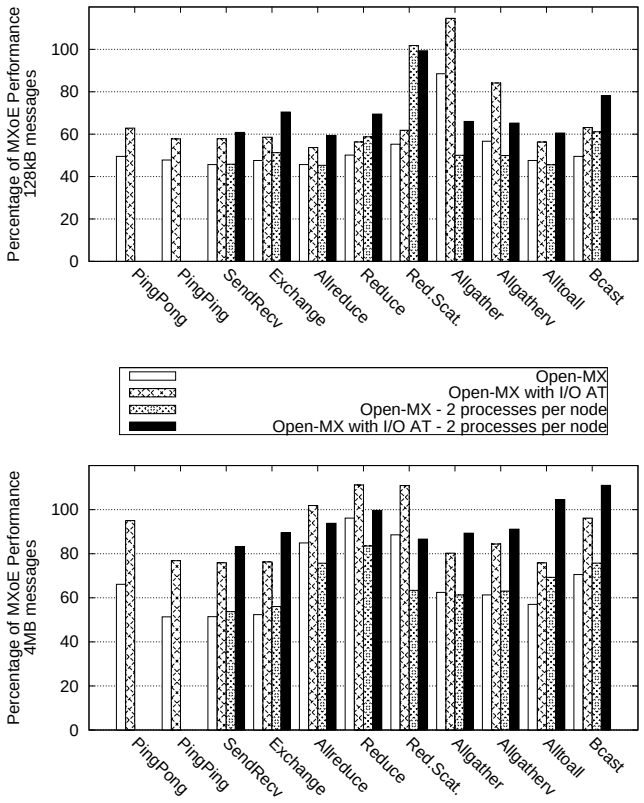


Fig. 12. Intel MPI Benchmarks performance on top of Open-MX (normalized to the performance on top of MXoE), with I/OAT being enabled or not, with 2 nodes and 1 or 2 processes per node.

For 4 MB messages, the improvement is on average 32 % for 1 process per node (achieving 90 % of MXOE) which shows the impact of our I/OAT usage over large messages. For 4 MB messages with 2 processes per node, the improvement is even higher (41 % on average, up to 94 % of MXOE) thanks to the addition of our I/OAT-based shared-memory communication.

Finally, it has to be noted that OPEN-MX is now able to even pass the native MXOE performance on several IMB tests. We also observed up to 10 % performance increase on the NAS parallel benchmarks [2], especially on IS which relies on large messages.

## V. DISCUSSION AND RELATED WORKS

The I/OAT hardware is available in most modern INTEL servers but few software subsystems actually use it. The TCP receive stack in the LINUX kernel was the primary user and enables better performance and reduced CPU overhead for various workload such as PVFS file transfers [23].

To the best of our knowledge, no other networking stack uses I/OAT copy offload yet. In HPC, most protocols are designed to avoid memory copies from the beginning, for instance by relying on RDMA-enabled hardware and drivers, such as EMP [19] or recently iWARP [17], which let the hardware place the data in the right receive buffer. Several other specific stacks such as GAMMA [4], MULTIEDGE [14]

rely on regular hardware with modified drivers to achieve a similar goal. However, these implementations only support very few 10 gigabit/s hardware while OPEN-MX relies on the generic ETHERNET layer of the LINUX kernel and may thus use any hardware.

The network receive stack is however not the only target application for I/OAT hardware. In the context of high-performance computing, I/OAT improves inter-process communication such as shared-memory MPI implementations on multicore nodes [21]. Our I/OAT based local communication model is very similar but has the advantage of being transparently integrated into the OPEN-MX stack since the driver automatically switches from regular to local communication without needing any specific support in user-space. It makes the OPEN-MX shared-memory implementation very simple to administrate and exploit, instead of being an add-on kernel module to a user-level MPI implementation.

Using multiple DMA channels[5] simultaneously improves copy throughput by up to 40 % [22]. However, OPEN-MX assigns a single channel per message and only relies on multiple channels to handle multiple outstanding messages. This strategy reduces the management cost without much decreasing the overall performance since we expect modern multicore machines to use many endpoints at the same time and thus have many outstanding messages use all the available DMA channels.

Another way to reduce memory copies is to use virtual memory tricks to remap the source buffer in the target virtual address space. Such a strategy has been studied for a long time to propose zero-copy socket implementations [3]. But it has multiple corner-cases caused by modern operating systems heavily relying on multiple page table states, pages being shared, miss-alignment, or memory pinning. It makes remapping difficult and expensive in many cases while it is indeed very interesting for performance and CPU load reduction purposes.

Virtual memory management is actually also involved in I/OAT copy offload since the hardware manipulates DMA addresses and thus requires pages to be pinned in physical memory. This problem appears in the vast majority of HPC network stacks whenever trying to reduce memory copies. It is often worked around using a registration cache [20] to reuse formerly pinned pages. This model however requires tracing of address space modifications to maintain the cache validity, which means some system calls have to be intercepted in user-space [22] or the kernel has to be modified [8] to do so. We plan to work around this important technical issue by making the need for a registration cache much less important. Indeed, the OPEN-MX model is compatible with an overlapping of memory registration with the actual beginning of the network communication. While not reducing the CPU load, such a strategy should dramatically reduce the impact of the registration cache on performance.

[5]There are 4 independent DMA channels on current INTEL I/OAT hardware.

Apart from improving performance and reducing CPU overhead, offloading memory copies with I/OAT also has the advantage of reducing cache pollution. This is very important for large messages since copying several megabytes may easily pollute the entire cache of the processor when using regular `memcpy` strategies. Indeed, it has been proven that offloaded copy avoids decreasing the performance of concurrent memory operations [22]. However, this cache "pollution" cannot always be consider as bad since it may actually load data that the receiver user application will soon read. For this reason, it may actually be interesting to use `memcpy` for small messages, or for the beginning of larger messages, and then switch to I/OAT. But it requires that `memcpy` occurs on the same core that shares a cache with the target application, which is hard to predict with the current OPEN-MX stack since the NIC may send interrupts to any core.

## VI. CONCLUSION AND FUTURE WORKS

In this paper, we presented the design of a copy offload strategy in the OPEN-MX receive stack to reduce memory copies. OPEN-MX aims at providing a high-performance message passing layer over any generic ETHERNET hardware, enabling interoperability between such hardware and the MYRICOM's native *Myrinet Express over Ethernet* stack. It is already able to run many existing applications that were designed for the native MX stack, such as MPICH-MX, OPEN MPI and PVFS2. This work targets the emergence of ETHERNET as an interesting networking layer in local area networks and already enables message passing with promising performance in this context.

The early OPEN-MX implementation suffered from CPU and memory pressure on the receive side due to one or two memory copies being required [7]. We designed an asynchronous copy offload strategy enabling the overlap of large message copy on the receive side. It increases the throughput by 30 % while reducing the CPU overhead by 40 % on the receive side. Copy offload also enables a 80 % throughput improvement for large message in the OPEN-MX single-copy-based local communication.

This implementation is available for download from the OPEN-MX homepage at http://open-mx.org/. It achieves 10G line rate performance for large messages at the MPI level. This work shows that the now widely available I/OAT hardware enables high performance message passing over ETHERNET without requiring RDMA-enabled NICs or any other hardware or driver specific feature in the network.

We now plan to look at auto-tuning the thresholds that control the enabling of our I/OAT copy offload. Indeed, the I/OAT and `memcpy` performance depend on the data size and on various hardware characteristics. Benchmarking the I/OAT hardware and `memcpy` in the cached and uncached cases on startup may thus help configuring our thresholds. Moreover, we are looking at copying the beginning of messages with `memcpy` to warm up the cache of the target user application if it shares a cache with the core that processes the packet.

Besides, the current I/OAT subsystem also does not support interrupts to wake up tasks waiting for copy completion. It reduces the overlap capability of our implementation, especially for large shared-memory communication since the receiver has to busy poll for copy completion. We plan to benchmark the I/OAT hardware on startup to predict the duration of the submitted copies, and auto-tune it at runtime using the actual completion times as a feedback. This way, synchronous copies in OPEN-MX should be able to sleep instead of busy poll until the actual completion is expected to arrive. Such a feature would enable better overlap of synchronous copies on I/OAT hardware.

This work mainly targeted the OPEN-MX throughput. It successfully removed a large part of CPU and memory copy bottleneck for large messages. The performance for smaller messages (below 32 kB) could not be improved as well since the I/OAT startup is too expensive and the current OPEN-MX model prevents the overlapping of multiple synchronous fragment copies. We are now working on deporting the matching from user-space into the driver so that a single completion event per medium message will be needed, making the aforementioned overlapping possible. We are also looking on improving small message latency, for instance by reducing cache effects between interrupt handlers and user-space applications when reporting OPEN-MX events from one core to another one. The combination of these ideas and the new copy offload model presented in this paper is expected to bring a highly competitive message passing implementation for all generic ETHERNET hardware.

## REFERENCES


[1] Coraid: The Linux Storage People. http://www.coraid.com/.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–264, San Diego, CA, 1996.

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

[5] FCoE (Fibre Channel over Ethernet. http://www.fcoe.com/.

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[7] Brice Goglin. Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, April 2008. IEEE.

[8] Brice Goglin, Loïc Prylli, and Olivier 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*, pages 726–733, Tampa, Florida, November 2004. IEEE Computer Society Press.

[9] Andrew Grover and Christopher Leech. Accelerating Network Receive Processing (Intel I/O Acceleration Technology), July 2005.

[10] Intel MPI Benchmarks. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm.

[11] Infiniband architecture specifications. InfiniBand Trade Association, 2001. http://www.infinibandta/org.

[12] Myricom Myri-10G Index. http://myri.com/Myri-10G/.

[13] 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.

[14] Stavros Passas, George Kotsis, Sven Karlsson, and Angelos Bilas. Exploiting spatial parallelism in Ethernet-based cluster interconnects. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, April 2008. IEEE.

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

[16] Quadrics. http://www.quadrics.com.

[17] Mohammad J. Rashti and Ahmad 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, March 2007.

[18] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):48–58, November 2004.

[19] Piyush Shivam, Pete 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, November 2001.

[20] 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, April 1998.

[21] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*, Austin, TX, September 2007.

[22] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *Proceedings of the International Workshop on Communication Architecture for Clusters (CAC), held in conjunction with IPDPS '07*, page 234, Long Beach, CA, March 2007.

[23] 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, April 2007.