NIC-Assisted Cache-Efficient Receive Stack for Message Passing over Ethernet

Brice Goglin

INRIA Bordeaux – France
`Brice.Goglin@inria.fr`

Abstract. High-speed networking in clusters usually relies on advanced hardware features in the NICs, such as zero-copy. Open-MX is a high-performance message passing stack designed for regular Ethernet hardware without such capabilities.

We present the addition of multiqueue support in the Open-MX receive stack so that all incoming packets for the same process are treated on the same core. We then introduce the idea of binding the target end process near its dedicated receive queue. This model leads to a more cache-efficient receive stack for Open-MX. It also proves that very simple and stateless hardware features may have a significant impact on message passing performance over Ethernet.

The implementation of this model in a firmware reveals that it may not be as efficient as some manually tuned micro-benchmarks. But our multiqueue receive stack generally performs better than the original single queue stack, especially on large communication patterns where multiple processes are involved and manual binding is difficult.

1 Introduction

The emergence of 10 gigabit/s ETHERNET hardware raised the questions of when and how the long-awaited convergence with high-speed networks will become a reality. ETHERNET now appears as an interesting networking layer within local area networks for various protocols such as FCoE [4]. Meanwhile, several network vendors that previously focussed on high-performance computing added interoperability with ETHERNET to their hardware, such as MELLANOX CONNECTX [3] or MYRICOM MYRI-10G [11]. However, the gap between these advanced NICs and regular ETHERNET NICs remains very large.

OPEN-MX [6] 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* [12]. While being compatible with any legacy ETHERNET NICs, OPEN-MX suffers from limited hardware features.

We propose to improve the cache-efficiency of the receive side by extending the hardware IP multiqueue support to filter OPEN-MX packets as well. Such a stateless feature requires very little computing power and software support

compared to the existing complex and statefull features such as zero-copy or TOE (*TCP Offload Engine*). Parallelizing the stack is known to be important on modern machines [16]. We are looking at it in the context of binding the whole packet processing to the same core, from the bottom interrupt handler up to the application.

The remaining of this paper is organized as follows. We present OPEN-MX, its possible cache-inefficiency problems, and our objectives and motivations in Section 2. Section 3 describes our proposal to combine the multiqueue extension in the MYRI-10G firmware and its corresponding support in OPEN-MX so as to we build an automatic binding facility for both the receive handler in the driver and the target application. Section 4 presents a performance evaluation which shows that our model achieves satisfying performance for micro-benchmarks, reduces the overall cache miss rate, and improves large communication patterns. Before concluding, related works are discussed in Section 5.

2   Background and Motivations

In this section, we briefly describe the OPEN-MX stack before discussing how the cache is involved on the receive side. We then present our motivation to add some OPEN-MX specific support in the NIC and detail our objectives with this implementation.

2.1   Design of the Open-MX Stack

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. Many existing middleware projects such as OPEN MPI [5] or PVFS2 [13] run successfully unmodified on top of it.

OPEN-MX was first designed as an emulated MX firmware in a LINUX kernel module [6]. This way, legacy applications built for MX benefit from the same abilities without needing the MYRICOM hardware or the native MX software stack (see Figure 1). 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 very simple low-level programming interface to access drivers and NICs.
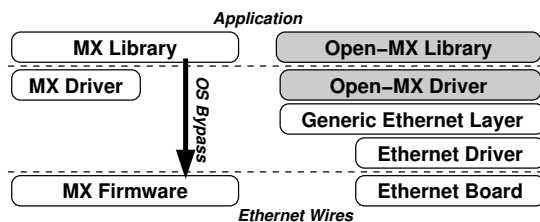

Fig. 1. Design of the native MX and generic Open-MX software stacks

2.2   Cache Effects in the Open-MX Receive Stack

Reducing cache effects in the OPEN-MX stack requires to ensure that data structures are not used concurrently by multiple cores. Since the send side is mostly driven by the application, the whole send stack is executed by the same core. The receive side is however much more complex. OPEN-MX processes incoming packets in its *Receive handler* which is invoked when the ETHERNET NIC raises an interrupt. The receive handler first acquires the descriptor of the communication channel (*endpoint*). Then, if the packet is part of a *eager* message ($\leq 32\,$kB), the data and corresponding event are written into a ring shared with the user-space library. Finally, the library will copy the data back to the application buffers (see Figure 2).
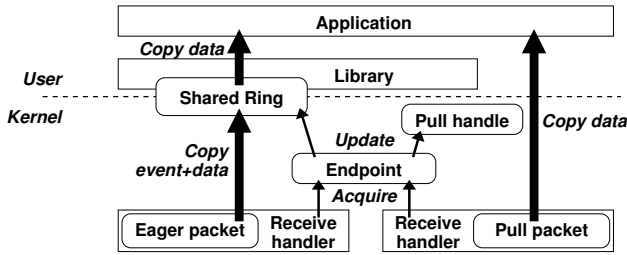

Fig. 2. Summary of resource accesses and data transfers from the Open-MX receive handler in the driver up to the application buffers

If the packet is part of a large message (after a *rendezvous*), the corresponding *Pull* handle is acquired and updated. Then, the data is copied into the associated receive buffer (Figure 2). An event is raised at the user-space level only when the last packet is received. This copy may be offloaded to INTEL *I/O Acceleration Technology* (I/OAT) DMA engine hardware if available [7].

The current OPEN-MX receive stack will most of the times receive IRQ (*Interrupt ReQuest*) on all cores since the hardware chipset usually distributes them in a round-robin manner (see Figure 3(a)). It causes many cache-line bounces between cores that access the same resources. It explains why processing all packets for the same endpoint on the same core will improve the cache-efficiency. Indeed, there would be no more concurrent accesses to the endpoint structure or shared ring in the driver. Additionally, all eager packets will also benefit from having the user-space library run on the same core since a shared ring is involved.

Large message (*Pull* packets) will also benefit from having their handle accessed by a single core. But, it is actually guaranteed by the fact that each handle is used by a single endpoint. Moreover, running the application on the same core will reduce cache effects when accessing the received data (except if the copy was offloaded to the I/OAT hardware which bypasses the cache).

2.3   Objectives and Motivations

A simple way to avoid concurrent accesses in the driver is to bind the interrupt to a single core. However, the chosen core will be overloaded, causing an availability imbalance between cores. Moreover, all processes running on other cores will suffer from cache-line bounces in their shared ring since they would compete with the chosen core. In the end, this solution may only be interesting for benchmarking purposes with a single process per node (see Section 4).

High-speed networks do not suffer from such cache-related problems since events and data are directly deposited in the user-space application context. It is one of the important features that legacy ETHERNET hardware lacks. The host only processes incoming packets when the NIC raises an interrupt. Fortunately, some ETHERNET-specific hardware optimizations have been developed in the context of IP networks. The study of cache-efficiency lead to the emergence of hardware multiqueue support. These NICs have the ability to split the incoming packet flow into several queues [17] with different interrupts. By filtering packets depending on their IP connection and binding each queue to a single core, it is possible to ensure that all packets of a connection will be processed by the same core. It prevents many cache-line bounces in the host receive stack.

We propose in this article to study the addition of OPEN-MX-aware multiqueue support. We expect to improve the cache-efficiency of our receive stack by guarantying that all packets going to the same endpoint are processed on the same core. To improve performance even more, we then propose to bind the target user-process to the core where the endpoint queue is processed. It will make the whole OPEN-MX receive stack much more cache-friendly. This idea goes further than existing IP implementations where the cache-efficiency does not go up to the application.

3   Design of a Cache-Friendly Open-MX Receive Stack

We now detail our design and implementation of a cache-friendly receive stack in OPEN-MX thanks to the addition of dedicated multiqueue support in the NIC and the corresponding process binding facility.

3.1   Open-MX-Aware Multiqueue Ethernet Support

Hardware multiqueue support is based on the driver allocating one MSI-X interrupt vector (similar to an IRQ line) and one ring per receive queue. Then, for each incoming packet, the NIC decides which receive queue should be used [17]. The IP traffic is dispatched into multiple queues by hashing each connection into a queue index.

The OPEN-MX multiqueue support is actually very simple because hashing its packets is easy. Indeed, the same communication channel (*endpoint*) is used to communicate with many peers, so only the local endpoint identifier has to be hashed. Therefore, the NIC only has to convert the 8-bit destination endpoint
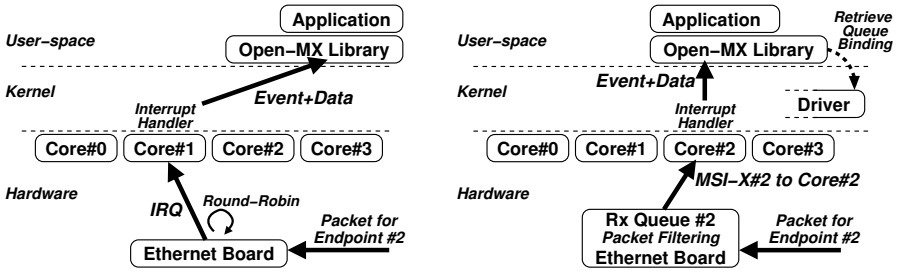
(a) Round-Robin Single-Interrupt: the interrupt goes to any core while the application may run somewhere else.

(b) Open-MX-aware Multiqueue: the NIC raises the MSI-X interrupt corresponding to the core where the application runs.

Fig. 3. Path from the NIC interrupt up to the application receiving the event and data

identifier into a queue index. It is much more simple than hashing IP traffic where many connection parameters (source and destination, port and address) have to be involved in the hash function. This model is summarized in Figure 3(b). The next step towards a cache-friendly receive stack is to bind each process to the core which handles the receive queue of its endpoint.

3.2   Multiqueue-Aware Process Binding

Now that the receive handler is guaranteed to run on the same core for all packets of the same endpoint, we discuss how to have the application run there as well. One solution would be to bind the receive queue to the current core when an endpoint is open. However, the binding of all queues has to be managed globally so that the multiqueue IP is not disturbed by a load imbalance between cores.

We have chosen the opposite solution: keep receive queues bound as usual (one queue per core) and make Open-MX applications migrate on the right core when opening an endpoint. Since most high-performance computing applications place one process per core, and since most MPI implementations use a single endpoint per process, we expect each core to be used by a single endpoint. In the end, each receive queue will actually be used by a single endpoint as well. It makes the whole model very simple.

3.3   Implementation

We implemented this model in the Open-MX stack with Myricom Myri-10G NICs as an experimentation hardware. We have chosen this board because it is one of the few NICs with multiqueue receive support. It also enables comparisons with the MX stack which may run on the same hardware (with a different firmware and software stack that was designed for MPI).

We implemented the proposed modification in the myri10ge firmware by adding our specific packet hashing. It decodes native Open-MX packet headers

to find out the destination endpoint number. Once the ETHERNET driver has been setup with one receive queue per core as usual, each endpoint packet flow is sent to a single core.

Meanwhile, we added to the `myri10ge` driver a routine that returns the MSI-X interrupt vector that will be used for each OPEN-MX endpoint. When OPEN-MX attaches an interface whose driver exports such a routine, it gathers all interrupt affinities (the binding of the receive queues). Then, it provides the OPEN-MX user-space library with binding hints when it opens an endpoint. Applications are thus automatically migrated onto the core that will process their packets. It makes the whole stack more cache-friendly, as described on Figure 3(b).

4   Performance Evaluation

We now present a performance evaluation of our model. After describing our experimentation platform, we will detail micro-benchmarks and application-level performance.

4.1   Experimentation Platform

Our experimentation platform is composed of 2 machines with 2 INTEL XEON E5345 quad-core *Clovertown* processors (2.33 GHz). These processors are based on 2 dual-core subchips with a shared L2 cache as described in Figure 4. It implies 4 possible processor/interrupt bindings : on the same core (SC), on a core sharing a cache (S$), on another core of the same processor (SP), and on another processor (OP).
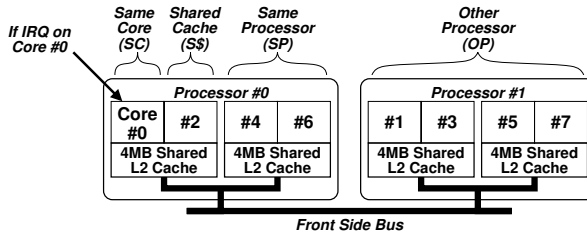

Fig. 4. Processors and caches in the experimentation platform (OS-numbered)

These machines are connected with MYRI-10G interfaces running in ETHERNET mode with our modified `myri10ge` firmware and driver. We use OPEN MPI 1.2.6 [5] on top of OPEN-MX 0.9.2 with LINUX kernel 2.6.26.

4.2   Impact of Binding on Micro-Benchmarks

Table 1 presents the latency and throughput of *Intel MPI Benchmark* [10] PINGPONG depending on the process and interrupt binding. It first shows that

Table 1. IMB PINGPONG performance depending on process and IRQ binding

Metric	Binding	SC	S$	SP	OP
0 byte latency	Round-Robin Single IRQ	$\simeq 11.4\,\mu s$			
	Single IRQ on core #0	10.96	9.34	10.32	10.25
	Multiqueue	11.71	10.10	11.09	11.25
4 MB throughput	Round-Robin Single IRQ	$\simeq 646\,\mathrm{MiB/s}$			
	Single IRQ on core #0	719	723	721	714
	Multiqueue	703	707	706	697
4 MB throughput (I/OAT)	Round-Robin Single IRQ	$\simeq 905\,\mathrm{MiB/s}$			
	Single IRQ on core #0	1056	1059	1048	1026
	Multiqueue	955	965	948	938

the original model (with a single interrupt dispatched to all cores in a round-robin manner) is slower than any other model, due to cache-line bounces. When binding the single interrupt to a single core, the best performance is achieved when the process and interrupt handler share a cache. Indeed, this case reduces the overall latency thanks to cache hits in the receive stack, while it prevents the user-space library and interrupt handler from competing for the same core.

Multiqueue support achieves satisfying performance, but remains a bit slower than optimally bound single interrupt. It is related to the multiqueue implementation requiring more work in the NIC than the single interrupt firmware.

4.3   Idle Core Avoidance

The above results assumed that one process was running on each core even if only two of them were actually involved in the MPI communication. This setup has the advantage of keeping all cores busy. However, it may be far from the behavior of real applications where for instance disk I/O may put some processes to sleep and cause some cores to become idle. If an interrupt is raised onto such an idle core, it will have to wakeup before processing the packet. On modern processors, this wakeup overhead is several microseconds, causing the overall latency to increase significantly.

To study this problem, we ran the previous experiment with only one communicating process per node, which means 7 out of 8 cores are idle. When interrupts are not bound to the right core , it increases the latency from 11 up to 15-20 μs and reduces the throughput by roughly 20 %.

This result is another justification of our idea to bind the process to the core that runs its receive queue. Indeed, if a MPI application is waiting for a message, the MPI implementation will usually busy poll the network. Its core will thus not enter any sleeping state. By binding the receive queue interrupt and the application to the same core, we guarantee that this busy polling core will be the one processing the incoming packet in the driver. It will be able to process it immediately, causing the observed latency to be much lower. All other cores that may be sleeping during

disk I/O will not be disturbed by packet processing for unrelated endpoints. This result may even reduce the overall power consumption of the machine.

4.4 Cache Misses

Table 2 presents the percentage of cache misses observed with PAPI [2] during a ping-pong depending on interrupt and process binding. Only L2 cache accesses are presented since the impact on L1 accesses appears to be lower.

Table 2. L2 cache misses (Kernel+User) during a ping-pong

Length	Round-Robin IRQ	IRQ on S$	IRQ on SC
0 B	29.80 %+13.79 %	29.70 %+8.34 %	11.47 %+**0.13 %**
128 B	26.80 %+17.90 %	25.06 %+23.05 %	14.15 %+**0.51 %**
32 kB	28.77 %+17.71 %	23.17 %+29.32 %	24.49 %+22.56 %
1 MB (I/OAT)	27.7 %+6.28 %	36.9 %+7.97 %	25.20 %+5.96 %

The table first shows that the cache miss rate is dramatically reduced for small messages thanks to our multiqueue support. Running the receive handler (the kernel part of the stack) always on the same core divides cache misses in the kernel by 2. Binding the target application (the user part of the stack) to the same core reduces user-space cache misses by a factor of up to 100.

Cache misses are not improved for 32 kB message communication. We expect this behavior to be related to many copies being involved on the receive path. It causes too many cache pollution, which prevents our cache efficiency from being useful.

Very large messages with I/OAT copy offload do not involve any data copy in the receive path. Cache misses are thus mostly related to concurrent accesses to the endpoint and pull handles in the driver. We observe a slightly decreased cache miss rate thanks to proper binding. But the overall rate remains high, likely because it involves some code-paths outside of the OPEN-MX receive stack (*rendezvous* handshake, send stack, ...) which are expensive for large messages.

4.5 Collective Communication

After proving that our design improves cache-efficiency without strongly disturbing micro-benchmark performance we now focus on complex communication patterns by first looking at collective operations. We ran IMB ALLTOALL between our nodes with one process per core. Figure 5 presents the execution time compared to the native MX stack, depending on interrupt and receive queue binding. It shows that using a single receive queue results in worse performance than our multiqueue support. As expected, binding this single interrupt to a single core decreases the performance as soon as the message size increases since the load on this core becomes the limiting factor.
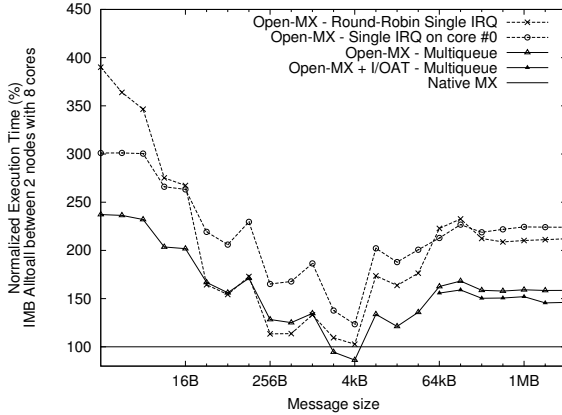
Fig. 5. IMB Alltoall relative execution time depending on interrupt binding and multiqueue support, compared with the native MX stack

When multiqueue support is enabled, the overall Alltoall performance is on average 1.3 better. It now reaches less than 150 % of the native MX stack execution time for very large messages when I/OAT copy offload is enabled. Moreover, our implementation is even able to outperform MX near 4 kB message sizes[1].

This result reveals that our implementation achieves its biggest improvement when the communication pattern becomes larger and more complex (collective operation with many local processes). We think it is caused by such patterns requiring more data transfer within the host and thus making cache-efficiency more important.

4.6 Application-Level Performance

Table 3 presents the execution time of some *NAS Parallel Benchmarks* [1] between our two 8-core hosts. Most programs show a few percents performance improvement thanks to our work. This impact is limited by the fact that these applications are not highly communication intensive. IS (which performs many large message communications) shows an impressive speedup (8.5 for class B, 2.6 for class C). Thanks to our multiqueue support, IS is now even faster on Open-MX than on MX. We feel that such a huge speedup cannot be only related to the efficiency of our new implementation. It is probably also caused by poor performance of the initial single-queue model for some reason.

It is again worth noticing that using a single interrupt bound to a single core sometimes decreases performance. As explained earlier, this configuration should only be preferred for micro-benchmarking with very few processes per node.

---

[1] Open-MX mimics MX behavior near 4 kB. This message size is a good compromise between smaller sizes (where the big Ethernet latency matters) and larger messages (where intensive memory copies may limit performance).

Table 3. NAS Parallel Benchmark execution time and improvement

	Single IRQ Round-Robin	Single IRQ on Single core	Multiqueue	Performance Improvement	MX
cg.B.16	34.62 s	34.32 s	33.68 s	+2.8 %	32.23 s
mg.B.16	4.14 s	4.19 s	4.02 s	+2.9 %	3.93 s
ft.B.16	22.80 s	23.06 s	21.34 s	+6.8 %	19.61 s
is.B.16	11.84 s	10.83 s	1.25 s	×8.5	1.33 s
is.C.16	14.69 s	14.17 s	5.62 s	×2.6	6.30 s

5   Related Works

High-performance communication in clusters heavily relies on specific features in the networking hardware. However, they are usually very different from the multiqueue support that we presented in this paper. The most famous hardware feature for HPC remains zero-copy support. It has also been added to some ETHERNET-based message passing stacks, for instance by relying on RDMA-enabled hardware and drivers, such as iWARP [14]. This strategy achieves a high throughput for large messages. But it requires complex modifications of the operating system (since the application must be able to provide receive buffers to the NIC) and of the NIC (which decides which buffer should be used when a new packet arrives).

Nevertheless, several other important features are still only available in HPC hardware, for instance application-directed polling for incoming packets. Indeed, high-speed network NICs have the ability to deposit events in user-space buffers where the application may poll. This innovation helps reducing the overall communication latency, but once again it requires complex hardware support. Regular ETHERNET hardware does not provide such features since it relies on a interrupt-driven model. The host processes incoming packets only when the NIC raises an interrupt. It prevents applications from polling, and implies poor cache-efficiency unless proper binding is used. Our implementation solves this problem by binding the whole receive stack properly.

Several ETHERNET-specific hardware optimizations are widely used, but they were not designed for HPC. Advanced NICs now enable the offload of TCP fragmentation/reassembly (TSO and LRO) to decrease the packet rate in the host [8]. But this work does not apply to message based protocols such as OPEN-MX. Another famous recent innovation is multiqueue support [17]. This packet filtering facility in the NIC enables interesting receive performance improvement for IP thanks to better cache-efficiency in the host. The emerging area where multiqueue support is greatly appreciated is virtualization since each queue may be bound to a virtual machine, causing packet flows to different VMs to be processed independently [15]. We adapted this hardware multiqueue support to HPC and extended it further by adding the binding of the corresponding target application. This last idea is, to the best of our knowledge, not used by any popular message passing stack such as OPEN MPI [5]. They usually just bind a single process per core, without looking at its affinity for the underlying hardware.

6   Conclusion and Perspectives

The announced convergence between high-speed networks such as INFINIBAND and ETHERNET raises the question of which specific hardware features will become legacy. While HPC networking relies on complex hardware features such as zero-copy, ETHERNET remains simple. The OPEN-MX message passing stack achieves interesting performance on top of it without requiring advanced features in the networking hardware.

This paper presents a study of the cache-efficiency of the OPEN-MX receive stack. We looked at the binding of interrupt processing in the driver and of the library in user-space. We proposed the extension of the existing IP hardware multiqueue support which assigns a single core to each connection. It prevents shared data structures from being concurrently accessed by multiple cores. OPEN-MX specific packet hashing has been added into the official firmware of MYRI-10G boards so as to associate a single receive queue with each communication channel. Secondly, we further extended the model by enabling the automatic binding of the target end application to the same core. Therefore, there are fewer cache-line bounces between cores from the interrupt handler up to the target application.

Performance evaluations first shows that the usual single-interrupt based model may achieve very good performance when using a single task and binding it so that it shares a cache with the interrupt handler. However, as soon as multiple processes and complex communication patterns are involved, the performance of this model suffers, especially from load imbalance between the cores. Using a single-interrupt scattered to all cores in a round-robin manner distributes the load but it shows limited performance due to many cache misses.

Our proposed multiqueue implementation distributes the load as well. It also offers satisfying performance for simple benchmarks. Moreover, binding the application near its receive queue further improves the overall performance thanks to fewer cache misses occurring on the receive path and thanks to the target core being ready to process incoming packets. Communication intensive patterns reveal a large improvement since the impact of cache pollution is larger when all cores and caches are busy. We observe more than 30 % of improvement for ALLTOALL operations, while the execution time of the communication intensive NAS parallel benchmark IS is reduced by a factor of up to 8.

These results prove that very simple hardware features enable significant performance improvement. Indeed, multiqueue support is becoming a standard feature that many NIC now implement. Our implementation is *Stateless* and does not require any intrusive modification of the NIC or host, contrary to usual HPC innovations. We now plan to further study hardware assistance in the context of message passing over ETHERNET. We will first look at multiqueue support on the send side. Then, designing an OPEN-MX-aware interrupt coalescing in the NIC may lead to a better compromise between latency and host load. Finally, the prefetching of incoming packets in processor caches with INTEL DCA (*Direct Cache Access*) [9] is another feature that may be combined with multiqueue

support for improving performance. Such features, as well as other stateless features than can be easily implemented in legacy NICs, open a large room for improvement of message passing over ETHERNET networks.

Acknowlegments

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. The International Journal of Supercomputer Applications 5(3), 63–73 (Fall 1991)
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. The International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
3. Mellanox ConnectX - 4th Generation Server & Storage Adapter Architecture, `http://mellanox.com/products/connectx_architecture.php`
4. FCoE (Fibre Channel over Ethernet), `http://www.fcoe.com`
5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004, pp. 97–104 (2004)
6. Goglin, B.: 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 Computer Society Press, Los Alamitos (2008)
7. Goglin, B.: Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In: Proceedings of the IEEE International Conference on Cluster Computing, Tsukuba, Japan, September 2008, pp. 223–231. IEEE Computer Society Press, Los Alamitos (2008)
8. Grossman, L.: Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In: Proceedings of the Linux Symposium (OLS 2005), Ottawa, Canada, July 2005, pp. 195–200 (2005)
9. Huggahalli, R., Iyer, R., Tetrick, S.: Direct Cache Access for High Bandwidth Network I/O. SIGARCH Computer Architecture News 33(2), 50–59 (2005)
10. Intel MPI Benchmarks, `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm`
11. Myricom Myri-10G, `http://myri.com/Myri-10G/`
12. 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`
13. The Parallel Virtual File System, version 2, `http://www.pvfs.org`

14. Mohammad, J.R., Afsahi, A.: 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 2007, Long Beach, CA, March 2007, p. 234 (2007)
15. Santos, J.R., Turner, Y., Janakiraman, G(J.), Pratt, I.: Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In: Proceedings of USENIX 2008 Annual Technical Conference, Boston, MA, June 2008, pp. 29–42 (2008)
16. Willmann, P., Rixner, S., Cox, A.L.: An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In: Proceedings of the USENIX Technical Conference, Boston, MA, pp. 91–96 (2006)
17. Yi, Z., Waskiewicz, P.P.: Enabling Linux Network Support of Hardware Multiqueue Devices. In: Proceedings of the Linux Symposium (OLS 2007), Ottawa, Canada, June 2007, pp. 305–310 (2007)