

Using a self-connected Gigabit Ethernet adapter as a memcpy() low-overhead engine for MPI

Giuseppe Ciaccio

DISI, Università di Genova
via Dodecaneso, 35 16146 Genova, Italy
ciaccio@disi.unige.it

Abstract. Memory copies in messaging systems can be a major source of performance degradation in cluster computing. In this paper we discuss a system which can offload a host CPU from most of the overhead of copying data between distinct regions in the host physical memory. The system is implemented as a special-purpose Linux device driver operating a generic, non-programmable Gigabit Ethernet adapter connected to itself. Whenever the descriptor-based DMA engines of the adapter are instructed to start a data communication, the data are read from the host memory and written to the memory itself thanks to the loopback cable; this is semantically equivalent to a *non-blocking memory copy* operation performed by the two DMA engines. Suitable completion test/waiting routines are also implemented, in order to provide traditional, blocking semantics in a split-phase fashion. An implementation of MPI using this system in place of traditional memcpy() calls on receive shows a significantly lower receive overhead.

1 Introduction

Let us consider those parallel applications which tend to exchange messages of relatively large size, due to intrinsic algorithmic features or coarse run-time parallelism. Such applications incur a performance penalty when using traditional networking systems, in that they pay for the CPU-operated data movements between different regions of the host memory during data communications. This translates into inefficient use of the host CPU and cache hierarchy by all running programs, including user jobs and the Operating System (OS) itself. The ability of offloading the host CPU from the memory copy operations involved by communications has long been recognized as a key ingredient for such parallel application to scale up.

To offload the host CPU from some overhead we need additional hardware or additional features into existing hardware.

One possibility is to leverage an additional CPU for this. Assuming each PC in the cluster comes provided with two CPUs, one could think that dedicating one such CPU to computation and the other one to communication might be a good idea, as this would entirely offload the former CPU from all of the communication overhead. This however would not address the problem of a lower overall utilization of the computation resources available at the processing node.

Another possibility is to use so-called *zero-copy* communication systems, in which the CPU-consuming phases of the communication protocol like header processing, receive matching, device virtualization, and data movements, run on board of the Network Interface Card (NIC) [6, 5, 8, 11, 7, 10]. This solution is thus restricted to programmable network devices (Myrinet and some Gigabit Ethernet adapters).

Zero-copy systems based on programmable NICs can also pose some issues. For instance, if the CPU located on the NIC is significantly slower compared to the host CPU, a wrong balance between NIC-operated and CPU-operated protocol phases may lead to an increased communication delay [6]. Message delivery must be restricted to prefetched memory pages marked as unswappable, so as to let the NIC know their physical address before communications take place (like in [8, 10, 11], for instance). But a perhaps more serious drawback concerns the efficient stacking of higher-level messaging primitives atop a zero-copy communication layer. For instance, stacking MPI atop the FM messaging system [4] required modifications and extensions of the set of FM routines: features like upcalls and streamed send and receive had to be added in order for MPI to take more advantage of the zero-copy features of FM. Basically, the problem lies in the fact that stacking higher-level communication primitives usually implies additional headers on messages and an extension of the matching algorithm on receive. If these extensions are not supported natively by the low-level, NIC-operated protocol, an intervention of the receiver CPU becomes necessary *before* the message payload could be delivered to any user-space destination; as a result, the computation-to-communication overlap, which is the main goal of zero-copy systems, becomes harder to obtain.

2 Demanding memory copies to a dedicated device

2.1 Performing memory copies through a self-connected NIC

A modern NIC cooperates with the host computer using a data transfer mode called *Descriptor-based DMA* (DBDMA). With the DBDMA mode, the NIC is able to autonomously set up and start DMA data transfers. To do so, the NIC scans two pre-computed and static circular lists called *rings*, one for transmit and one for receive, both stored in host memory. Each entry of a ring is called a *DMA descriptor*.

A DMA descriptor in the transmit ring contains a pointer (a physical address) to a host memory region containing a *fragment* of an outgoing packet; therefore, an entire packet can be specified by chaining one or more send DMA descriptors, a feature called “gather”.

Similar to a descriptor in the transmit ring, a DMA descriptor in the receive ring contains a pointer (a physical address, again) to a host memory region where an incoming packet could be stored. The analogous of the “gather” feature of the transmit ring is here called “scatter”: more descriptors can be chained to specify a sequence of distinct memory areas, and an incoming packet could be scattered among them.

Now, let us consider a DBDMA-based full-duplex NIC networked to itself through a loopback wire (its output port connected to its input port).

Suppose the host CPU have to copy L bytes from source address S to destination address D ; suppose S be in kernel address space and D be in a user space, as it occurs with a message receive in a communication system implemented in the OS kernel. Virtual addresses S and D can be translated to physical addresses, say P_S and P_D ; a kernel-space address usually need not be translated at all, so the CPU overhead is just for one virtual to physical translation in this case (namely, D into P_D). At this point, the CPU can create two DMA descriptors for the NIC: one, in the transmit ring, points to address P_S , and the other, in the receive ring, points to address P_D . Both descriptors carry the same buffer size of L bytes. When the CPU commands to start a transmission, the NIC transmits L bytes of memory at address P_S to self, so receives them at address P_D . The data flow takes place at the network wire speed (supposedly lower than the DMA speed). At the end of the operation, the memory content has changed as if a `memcpy(D,S,L)` operation were issued, under the constraint that the two L byte sized memory regions P_S and P_D do not overlap.

The above procedure can be clearly extended to user-to-kernel as well as kernel-to-kernel or user-to-user cases, the latter requiring one more address translation.

Ethernet NICs usually exchange data using a MAC protocol which poses some requirements on data formatting, namely: data must be arranged into packets whose maximum size, called MTU, is fixed; and, each packet must be prepended by a header. These constraints make things a bit more complicated, but not too much. Indeed, copy operations whose size exceeds the maximum MTU size supported by the NIC can be easily broken into MTU-sized chunks. And, in order to eliminate the need for MAC packet headers, the NIC can be set up to operate in promiscuous mode so as the initial part of the packet will not be interpreted.

A number of Gigabit Ethernet adapters support non-standard MTU sizes of up to a few KBytes (to allow so called “jumbo frames”) for efficiency reasons. Using one such NIC connected to itself, long memory copies can be operated on a page-by-page basis, with only a marginal intervention by the host CPU.

2.2 Split-phase memory copies, and notification of completion

The sequence of events described in Section 2.1 gives rise to a copy of the content of a memory region to another, disjoint, memory region. In order for the events to take place, however, the host CPU must first set up two DMA descriptors, then command the NIC to start a transmission (to self). Actually, this latter operation only starts the memory copy. The operation will then proceed without any further involvement of the host CPU, but we need test/wait mechanisms to eventually synchronize the CPU to the end of a given copy operation, so as to prevent incorrect actions (e.g., reading the copied data before the copy itself has finished). This could be done by using *notification variables*, that is, memory locations whose stored value is to change as soon as the operation is complete.

Notifying completion of a copy operation by acting upon a notification variable can be done without the intervention of the host CPU. The gather/scatter features of the NIC can be exploited to this end. The idea is to extend the self-transmission trick by gathering together source data and notification *value*, then scattering them apart to destination buffer and notification *variable* on receive. To this end, we simply need to arrange a 2-way DMA descriptor in the transmit ring and a 2-way DMA descriptor in the receive ring, in place of traditional, single-entry descriptors.

At the end of the self-transmission, the presence of the notification value into the notification variable implies that the memory-to-memory copy is complete, because the notification value is delivered to memory *after* the actual data. No intervention by the host CPU is needed to accomplish such notification.

As a major advantage, such a split-phase memory copy would allow the host CPU to carry out useful work in between initiation and test/wait operations, this way overlapping the memory copy with other useful computation at the price of a initiation overhead.

3 A working prototype: DAMNICK

DAMNICK (DAta Moving through a NIC connected in loopback) is a working prototype of a system for demanding memory-to-memory copies to the DMA engines of an autonomous, self-connected Gigabit Ethernet adapter, located on the host I/O bus. It is implemented as a modified Linux device driver for the supported Gigabit Ethernet adapter. Currently, only the Alteon AceNIC and its lower-cost “clones” (the 3COM 3c985 and the Netgear GA620) are supported. The driver disables all sources of IRQ in the NIC, as they are unneeded.

The DAMNICK driver implements routines for initiating copy operations, according to the ideas sketched in Section 2.1. The driver also supports the NIC-operated notification of “end of copy” described in Section 2.2. There are routines for moving data from user space to user space of the same process, with or without notification of completion (resp. `icopy_notif()`, `icopy()`). Routines to move data from kernel space to user space are provided as well (`icopy_k2u_notif()`, `icopy_k2u()`). Testing/waiting for termination of initiated copy operations is done by the invoker by inspecting the current value of the given notification variable.

These routines can be invoked directly from OS kernel, in case of internal use; in addition, a trap address is set up to allow safe invocation of DAMNICK routines from application level, through a small library of stubs.

As pointed out in Section 2.1, the NIC must be set to promiscuous mode in order for DAMNICK to work. This is done at the time of opening the device before use (using the `ifconfig` UNIX utility).

3.1 Evaluation framework, performance metrics, and testbed

To evaluate the performance advantages of DAMNICK in a realistic scenario, we decided to integrate it into the receive part of a message-passing system running

on a cluster of PCs. To this end, we decided to use the GAMMA messaging system [1], for which an implementation of MPI is available [2].

In the kernel-level receive thread of GAMMA, a traditional `memcpy()` operation delivers received message chunks from kernel buffers to application address space, followed by an “end of message” notification upon delivery of the last message chunk. The receiver CPU accomplishes the “end of message” notification by incrementing the value of a user-level variable.

The `memcpy()` has been replaced by an invocation of `icopy_k2u()` in the case of intermediate chunks of a message. For the last chunk, the `memcpy()` and the subsequent “end of message” notification have been replaced by a single invocation of `icopy_k2u_notif()`.

Our expectation was to observe a decisive decrease of the receive overhead when using DAMNICK in place of `memcpy()` on receive. Thus, next step has been to set up a proper benchmark written in MPI to evaluate the receive overhead at MPI level.

The testbed of our performance evaluation is a pair of PCs networked to each other by a dedicated, back-to-back Fast Ethernet connection. Both PCs have a single Athlon 800 MHz CPU, ASUS A7V motherboard (VIA KT133 chipset, 32 bit 33 MHz PCI bus, 133 MHz FSB), 256 MByte 133 MHz DRAM, and a 3COM 3c905C Fast Ethernet adapter. Each PC also mounts a Netgear GA620 Gigabit Ethernet adapter connected to itself through a fiber-optics loopback cable, along which the DAMNICK self-communications take place. The OS is GNU/Linux 2.4.16 including GAMMA and the DAMNICK device driver.

3.2 The benchmark

A decisive decrease in MPI receive overhead at application level would only be helpful in case of overlap between useful computation and message arrivals at a given processing node.

A suitable benchmark could be one in which a process receives a stream of tasks, each represented by a message, and has to carry out some computation to consume each of them. Ideally the process of receiving tasks and consuming them should work like a pipeline, supposing that new tasks arrive at at least the same rate at which they are consumed. Clearly, overlapping computation to message receives can be of great help in a situation like this, as it would allow to increase the servicing throughput of the process. We then developed a proper MPI benchmark of this kind, for our performance evaluations.

A first, “naive” version of the program could be sketched as follows (receiver process only):

```

for (;;) {
    MPI_Recv(task);
    do_computation(task);
}

```

Here, the use of blocking receive allows limited overlap between computation and communication. A better solution could be obtained with non-blocking receives, by unrolling the loop twice then scheduling the activities in a proper way:

```

MPI_Irecv(task1,handle1);
for (;;) {
    MPI_Irecv(task2,handle2);
    MPI_Wait(handle1); do_computation(task1);
    MPI_Irecv(task1,handle1);
    MPI_Wait(handle2); do_computation(task2);
}

```

In this solution, activities on `task1` are data-independent from activities on `task2`, so they can occur in between initiation and termination of communication activities concerning `task2`, and viceversa. The scheduling of activities shown in the program leads to a good *potential* overlap between task receive and task processing, and even between different phases of receive (of consecutive tasks).

However, the amount of *effective* overlap depends on the implementation of MPI in use. In the ideal case in which the whole implementation of MPI is operated by the NIC, the overlap takes place and the CPU never blocks on communication¹. To the best of our knowledge, however, no fully NIC-operated implementation of MPI has been attempted so far. The best available MPI implementations are stacked atop lower-level, zero-copy, NIC-operated messaging systems, lacking significant parts of the MPI semantics, and especially the message matching (see [4, 9, 11, 2] for instance). In Section 1 we already pointed out that such architecture requires an intervention of the host CPU to match the MPI message envelope against MPI pending receives, and this must occur after the message arrival is detected but before the payload can be delivered to its destination address. As a result, it becomes difficult or impossible to achieve a good overlapping between communication and computation in spite of non-blocking receives and possibly zero-copy implementation of MPI.

With an MPI stacked atop a lower-level messaging system, the only way to get a significant computation-to-communication overlap is to extend MPI with an additional, blocking routine called `poll()` which waits for the low-level communication layer to notify a new message arrival, then performs the

¹ This is true under the aforementioned hypothesis that new tasks arrive at at least the same rate at which they are consumed. If this is not the case, the process will block on receive, waiting for incoming tasks.

MPI message matching and computes destination address for the new message, and finally asks the low-level communication layer to deliver the payload to destination, without waiting for the delivery to complete. Usually, these steps are accomplished within the `MPI_Wait()` routine. Moving them away from the `MPI_Wait()` to the `poll()` routine allows to split the blocking receive semantics into three phases rather than just two. That is, instead of accomplishing message receives by traditional split-phase pairs `MPI_Irecv(); MPI_Wait()`, we rather use triples of the kind `MPI_Irecv(); poll(); MPI_Wait()` and schedule the computation in the two slots between the three routines, which correspond to the two *separate* time slots in which the communication-to-computation overlap is allowed to take place.

The “three-phases” splitting requires unrolling the benchmark loop three times rather than just twice. Our MPI benchmark thus sketches as follows:

```
MPI_Irecv(task1,handle1);
MPI_Irecv(task2,handle2);
poll(); /* for task1 */
for (;;) {
    MPI_Irecv(task3,handle3);
    poll(); /* for task2 */
    MPI_Wait(handle1); do_computation(task1);
    MPI_Irecv(task1,handle1);
    poll(); /* for task3 */
    MPI_Wait(handle2); do_computation(task2);
    MPI_Irecv(task2,handle2);
    poll(); /* for task1 */
    MPI_Wait(handle3); do_computation(task3);
}
```

From the discussion above, it turns out that traditional, established MPI application benchmarks like the NAS NPB, which use standard split-phase send/receives to overlap computation to communication, could *never* detect the potential computation-to-communication overlap as provided by *any* MPI stacked atop a zero-copy messaging layer. This is the only reason why we could not provide any meaningful evaluation using established MPI application benchmarks.

3.3 Results

Figure 1 shows curves of MPI/GAMMA receive overhead as measured by the benchmark discussed before, as functions of the message size. Two distinct scenarios are reported about, namely, “hot cache” and “cold cache”. In the “hot cache” case, all buffers touched during the receive operations are in cache; this is not the case with the “cold cache” scenario.

In all cases, the curves show the expected linear dependence of receive overhead on the message size. Non-data-touching receive overhead is reported as a

lower bound, to highlight the portion of overhead where DAMNICK is expected to show its effects.

The `memcpy()` operation is clearly expected to be faster in the “hot cache” case compared to the “cold cache”; thus, it is no wonder that, when the messaging system uses a `memcpy()` to deliver data to final user-space destinations, the receive overhead is lower in the former case compared to the latter. It is no wonder as well, that the MPI/GAMMA receive overhead becomes insensitive to the caching state when `icopy_k2u_notif()` is used in place of `memcpy()` to deliver data to final destinations, as DAMNICK exploits DMA engines to move data, and DMA works in the host RAM, not cache.

The significant results come from comparing the MPI/GAMMA receive overhead, using `memcpy()` vs. using `icopy_k2u_notif()`. A clear advantage of the latter mechanism over the former one is shown in the two pictures. With “hot cache”, the use of DAMNICK reduces the receive overhead by 31% with long messages. The difference is even more evident with “cold cache”, with a decrease of 42%. If we are only concerned with the data-touching fraction of receive overhead, with DAMNICK this is reduced by 39% and 50%, respectively with “hot cache” and “cold cache”.

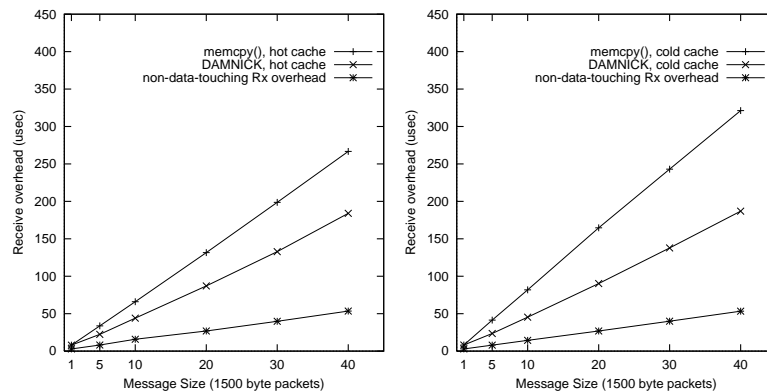


Fig. 1. Comparison of MPI/GAMMA receive overhead, using `memcpy()` vs. using `icopy_k2u_notif()` to deliver data to final destinations.

4 Related work

A problem which poses similar challenges to the ones addressed in this paper is that of intra-node communication, that is, message exchanges between processes running on the same processing node. In intra-node communication, two processes usually exploit a shared memory region to pass information to each

other, at the cost of one memory copy performed by each. One of the two memory copies can be avoided in some cases, by running OS kernel privileged code which can directly access the virtual address spaces of both processes and thus can copy data from sender to receiver directly [3]. However, as an alternative, the two partner might pass messages to each other through the NIC, as they were remote to each other, provided that the interconnection switch be able to route data to the same NIC which originates them (loopback through first switch). Myrinet provides such a feature, which was exploited for intra-node communication by the Myricom GM communication library. Poor performance of such network-assisted intra-node communication is claimed in [3]. We could not carry out any experiments on Myrinet, but we suspect the poor performance claims of [3] actually refer to end-to-end communication delay, rather than CPU overhead.

5 Conclusions and open points

Our experiments demonstrate the practical feasibility of a system which is able to offload a substantial fraction (up to 42%) of the total receive overhead of MPI from the host CPU to a separate device. This is entirely based on cheap, *non-programmable* commodity components. By connecting a generic Gigabit Ethernet NIC back to itself with a loopback cable, and designing a suitable software to drive the NIC in a non-trivial way, it is possible to exploit the DMA capabilities of the NIC to perform memory-to-memory operations. This way, a larger fraction of host CPU time is delivered to user applications and OS tasks, without leveraging any custom hardware devices (expensive, due to their narrow marketplace segment) or programmable devices (expensive, due to their high complexity).

In order for such a system to work properly, the memory pages involved in the copy operation must be already loaded in RAM. Pages belonging to the OS kernel space are usually marked as non-swappable. For user space pages, it is the communication system that should check for page presence and possibly enforce page loading before use. This is already common practice with zero-copy messaging, and does not introduce any additional penalty compared to classical systems, based on memory copies with page fault management.

This system requires dedicating a Gigabit Ethernet NIC to memory copies; such a NIC could not be used for anything else. Needless to say, in a cluster of PCs we would rather prefer to find suitable DMA devices on each PC's motherboard, instead of adding a Gigabit Ethernet NIC to each node. Indeed, most motherboards come equipped with DMA devices for EIDE disk operation, but these have no descriptor queue; as a consequence, the CPU will block waiting for completion of the current DMA operation before submitting a new one, what makes this device unsuitable as an alternative to CPU-operated memory copies.

The addition of a Gigabit Ethernet adapter to each node in a cluster could be a questionable idea if the adapters were expensive. This is no longer the case, however, especially after the recent deployment of the Gigabit-over-copper low-cost technology. Indeed, modern high-end motherboards often come equipped

with on-board Gigabit Ethernet controllers, whose price has become marginal compared to the cost of the entire motherboard.

Performance evaluation of DAMNICK as a support to MPI led us to a deeper understanding of the reason behind the relative insensitiveness of MPI parallel benchmarks to the receive overhead. From the arguments in Section 3.2, it follows that the classical split-phase approach to non-blocking send/receive routines is inappropriate whenever MPI is implemented in software; such an architecture cannot deliver a good computation-to-communication overlap to user applications on receive, regardless of it being zero-copy or not, because in this case the “overlapping window” offered by a `MPI_Irecv()`; `MPI_Wait()` pair is actually made up of two separate parts. Running the entire MPI in firmware on a programmable I/O device would be a satisfactory answer, but has not been attempted so far, probably due to excessive resource requirements.

Demanding memory copies to an I/O device forces data to travel across the PCI bus, which might show a higher bit error rate compared to the system bus. Extensive tests are to be carried out to investigate this aspect.

References

1. G. Chiola and G. Ciaccio. Efficient Parallel Processing on Low-cost Clusters with GAMMA Active Ports. *Parallel Computing*, (26):333–354, 2000.
2. G. Ciaccio. MPI/GAMMA home page, <http://www.disi.unige.it/project/gamma/mpigamma/>.
3. P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High performance message passing over a cluster of commodity SMPs. In *Proc. of 11th IEEE - ACM High Performance Networking and Computing Conference (SC99)*, 1999.
4. M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.
5. Myricom. GM performance, <http://www.myri.com/myrinet/performance/>, 2000.
6. S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing '95*, San Diego, California, 1995.
7. I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proc. Infocom 2001*, Anchorage, Alaska, April 2001. IEEE.
8. L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, number 1388 in Lecture Notes in Computer Science, pages 472–485, Orlando, Florida, April 1998. Springer.
9. L. Prylli, B. Tourancheau, and R. Westrelin. The design for a high performance MPI implementation on the myrinet network. In *Proc. EuroPVM/MPI'99*, number 1697 in LNCS, pages 223–230, Barcelone, Spain, 1999.
10. P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proc. 2001 International Conference on Supercomputing (SC01)*, Denver, Colorado, November 2001.
11. T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In *Proc. of High Performance Computing and Networking (HPCN'97)*, number 1225 in Lecture Notes in Computer Science, pages 708–717. Springer-Verlag, April 1997.