Optimal Communication Performance on Fast Ethernet with GAMMA

Giuseppe Ciaccio

DISI, Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
E-mail: `ciaccio@disi.unige.it`

Abstract. The current prototype of the Genoa Active Message MAchine (GAMMA) is a low-overhead, Active Messages-based inter-process communication layer implemented mainly at kernel level in the Linux Operating System. It runs on a pool of low-cost Pentium-based Personal Computers (PCs) networked by a low-cost 100base-TX Ethernet hub to form a low-cost message-passing parallel platform. In this paper we describe in detail how GAMMA could achieve unprecedented communication performance (less than 13 μs one-way user-to-user latency time and up to 98% of the communication throughput of the raw interconnection hardware) on such a kind of low-cost parallel architecture.

Keywords: Active messages; Fast Ethernet; Inter-process communication; Network of workstations; Parallel processing.

1 Introduction

Personal Computer (PC) hardware technology has improved so much in terms of cost and performance that has now practically reached quite competitive levels of absolute performance with respect to scientific workstations as well as mainframes at a substantially lower cost, due to the larger economy of scale that is involved in the PC market. Therefore it does now make sense from a technological as well as a financial point of view to design and assemble low-cost, medium-to-high performance networks of PCs aiming at replacing high-performance mainframes or even massively parallel computers for computational intensive applications where a fair number of processors is required.

The idea of assembling Networks Of Workstations (NOWs) that can compete with supercomputers in terms of absolute performance as well as cost per MFLOP has been pursued by several research projects, such as the Berkeley NOW [2] and HPAM [7], Illinois Fast Messages [9], Cornell's SSAM [12] and U-Net [13], which already provided strong experimental evidence of the potentials of this approach. Fewer research projects focused on lower cost ensembles of PC-based computation nodes networked by off-the-shelf Local Area Networks (LANs) hardware, aiming at producing less ambitious NOWs in terms of absolute performance, but providing better cost/performance figures, such as the

NASA's Beowulf [11], Cornell's U-Net on Linux [15], PARMA^2 [6], and Illinois Fast Messages on Linux [8].

With low-cost PC-based NOWs the challenge is to take the best advantage of low-cost communication hardware in order to reach overall communication performance at the application level that is acceptable in absolute terms for message-passing parallel processing purposes. Some of the previous works in the field of low-cost NOWs addressed this problem by cutting down on the complexity of the communication protocols involved in LAN communications. The PARMA^2 approach consisted in replacing the industry standard TCP/IP protocol suite exploited by Unix Sockets with a lighter-weight protocol optimized for the special case of a reliable LAN. U-Net took a much more radical approach, by removing all protocol-based communication abstractions from the Operating System (OS) Kernel, and offering instead a very low level abstraction of the crude communication devices to user processes.

Our approach is different, and such difference allowed us to obtain a prototype low-cost PC-based NOW called Genoa Active Message MAchine (GAMMA) that is characterized at the same time by:

1. a cost per node (including communication devices) of less than 1.5 K US$ at the current market price;
2. message latency (12.7 $\mu s.$) as good as the leading-edge, expensive NOW prototypes, and better than many commercial messaging systems running on MPPs;
3. a maximum message throughput corresponding to 98% of the 100base-T Ethernet theoretical throughput, which in absolute terms (12.2 MB/s) happens to be higher than the one offered by Thinking Machines' CM-5 and by Transputers' channels and almost as good as the one offered by systems based on ATM and FDDI;
4. a very favourable throughput profile also for messages of short/medium size, with half the maximum throughput achieved with messages as short as 192 bytes;
5. a limited scalability up to few tens of computation nodes (16 in our current prototype);
6. a fairly high-level and convenient programming interface to develop SPMD as well as MIMD parallel applications;
7. the availability of the full Linux OS development and run-time environment for all programming aspects not related to parallel process activations and interprocessor communications (namely local and remote file systems, network services, public domain compilers, debuggers, etc.).

A comparison of GAMMA with respect to other NOW projects and commercial MPPs is sketched in Table 1.

The novelty of our approach is the consistent application of the Active Message paradigm originally proposed in [14] to the case of a low-cost NOW, taking advantage of all optimizations that we were able to devise in order to embed Active Message primitives in the architecture of a Unix-like OS Kernel, without changing the OS Kernel itself.

Platform	Processor	network	Latency	Throughput
			(μs)	(MByte/s)
GAMMA	P5 133MHz	100base-T	12.7	12.2
Cornell U-Net 100base-T	P5 133MHz	100base-T	30.0	12.1
PARMA^2 sockets	P5 100MHz	100base-T	73.5	6.6
Linux 2.0 TCP/IP sockets	P5 133MHz	100base-T	151.0	5.6
Illinois Fast Messages	PPro	Myrinet	11.5	56.3
Illinois Fast Messages	UltraSPARC	Myrinet	13.1	17.5
Berkeley HPAM	HP RISC	100Mb/s FDDI	14.0	12.0
Cornell U-Net ATM	SuperSPARC	140Mb/s ATM	35.5	14.8
Cray T3D Fast Messages	Alpha	custom	6.1	112
CM-5, CMAML	SPARC	custom	15.0	8.3
Cray T3D PVMFAST	Alpha	custom	30.0	25.1
IBM SP2 MPL	RISC	custom	44.8	34.9
IBM SP2 PVMe	RISC	custom	209.0	16.5

Table 1. Latency and Bandwidth for 2 processor communication: comparison of platforms.

.

The so called "user-level" approach followed by most NOW projects (for instance [9, 7, 13, 15]) and that consists in eliminating the Operating System from the communication path and allowing direct access from user level to the Network Interface Card (NIC), is usually considered crucial in order to achieve best communication performance. This is arguably true for some next-generation interconnection devices (for instance Scalable Coherent Interface and Memory Channel) which are more tightly coupled with the CPU than today's off-the-shelf communication devices. With low-cost commodity interconnection hardware we argue that "user-level" access is not necessary to achieve satisfactory exploitation of the raw networking devices: indeed GAMMA is able to offer best communication performance while retaining the intervention of the Operating System for protected multiplexing of the NIC among user processes.

This paper discusses in detail the optimizations carried out during the implementation of GAMMA as well as the impact of such optimizations on the overall communication performance.

2 A brief account of Active Messages

Many modern high performance messaging systems have been derived from the Active Message communication paradigm [14]. Active Messages are aimed at reducing the communication overhead and allowing communication to overlap computation. The advantage of Active Messages over other communication paradigm is that it eliminates the need of intermediate copies of messages along the communication path, thus remarkably speeding up communications. In a traditional send-receive messaging system, messages delivered to a destination node may need to be temporarily buffered waiting for the destination process to invoke a “receive” operation which will consume them. With Active Messages this is no longer true. As soon as delivered, each message triggers a function of the destination process, known as the *receiver handler*, which will consume it right away. Here “consuming” means integrating the message information into the ongoing computation of the destination process, notifying the reception to the destination process itself, and possibly setting some data structures in order to promptly “consume” the next incoming message as soon as it arrives. A well known commercial application is the Active Message Layer introduced by Thinking Machines Co. in the CM-5 platform [1], yielding user-to-user throughput and latency very close to the limit posed by the hardware devices [5].

With most current Active Messages-like messaging systems, the receiver handler is sender-based: Each transmitted message is composed of two parts, namely the message body and the explicit pointer to the destination's receiver handler that will consume the message upon reception. It is the receiver handler that extracts the message and stores it into a data structure in the address space of the destination process. Such “Remote Procedure Call (RPC)-like” communication model requires that the destination process share its own (code) address space with the sender process, a condition which is easily fulfilled only with the Single Program Multiple Data stream (SPMD) paradigm.

GAMMA exploits a more general Active Message-like model inspired to Thinking Machines' CMAML for the CM-5, where each process has a number of *communication ports* and may attach both a receiver handler and a data structure to a single communication port in order to handle all messages incoming through that port. Moving the message from the network to the destination data structure attached to the communication port may be efficiently and timely performed by the device driver. This way the sender specifies a communication port of the destination process instead of the pointer to a receiver handler, and the receiver handler must not explicitly extract and store the incoming message. This “port-oriented” variant of Active Messages is suitable for MIMD as well as SPMD programming, as it does not require a common address space among cooperating processes.

3 A sketch of GAMMA and its programming interface

The current prototype of GAMMA is composed of a set of (currently 16) autonomous PCs connected by means of two independent LANs: one isolated 100base-TX Ethernet LAN dedicated exclusively to fast inter-process communications, and one 10 Mb/s Ethernet LAN running the standard TCP and UDP protocol suites to provide UNIX network services (access to network file servers, remote login, etc.). No custom hardware device is exploited.

Each PC comprises the following components: Intel Pentium 133 MHz CPU, 256 KByte of 15 ns pipelined secondary cache, PCI Intel Triton II chipset, 32 MByte of 60 ns DRAM, either a 3COM 3C595-TX Fast Etherlink or a 3COM 3C905-TX Fast Etherlink XL PCI network adapter.

The 100base-TX Ethernet LAN consists of two repeater hubs, a Bay Networks' Netgear FE516 hub with 16 RJ-45 ports, and a 3Com FS 100 Linkbuilder hub with 12 RJ-45 ports. Each Fast Ethernet NIC is connected to an hub port by a class 5 UTP cable.

The GAMMA communication layer is implemented mainly at kernel level as an extension of the Linux OS kernel. Additional system calls and a custom NIC device driver implement the largest part of the protocol. Most of the GAMMA code has been written as a set of carefully optimized C functions, with a few performance-critical sections of code written in 80x86 Assembler. The GAMMA communication services are made available to user applications through a programming interface implemented as a user-level library. The implementation of the GAMMA library exploits the GAMMA additional system calls. Both C and FORTRAN programming interfaces have been developed.

In order to understand the implementation details as well as the programming interface of GAMMA, it is worth pointing out the computational model we had in mind during the design phase.

A *physical GAMMA* is the set of M PCs connected to the Fast Ethernet LAN. At the level of physical GAMMA each PC is addressed by the Ethernet address of the corresponding NIC. A *virtual GAMMA* is a set of N computation nodes, each one corresponding to a distinct PC in the physical GAMMA (hence $N \leq M$ must hold). Nodes in a virtual GAMMA are addressed by a number from zero on, similarly to instance numbers in SPMD process groups. At the kernel level each of these numbers is mapped onto the Ethernet address of the corresponding PC.

The programming paradigm of GAMMA is SPMD (Single Program Multiple Data). A *parallel program* may be thought of as a SPMD group of N processes running in parallel on the N nodes of a single virtual GAMMA and inheriting node numbers as instance numbers in the group.

GAMMA supports parallel multitasking, i.e., more than one virtual GAMMA (that is, more than one SPMD process group on behalf of potentially different users) may be spawned at the same time on the same physical GAMMA. As a consequence, any PC may be shared by more than one parallel as well as sequential program at a given time. Each virtual GAMMA and the corresponding running parallel program is identified by a number which is unique in the platform. We call such a number a *parallel PID*. Processes belonging to the same group share the same parallel PID. The parallel PID is used to distinguish among processes belonging to different parallel applications at the level of each individual PC. A maximum of 256 different parallel PIDs may be activated on a physical GAMMA.

With GAMMA, any process of a given parallel application owns, and may activate and use thereof, 255 *communication ports* through which it can send and receive messages. Useful communication ports are numbered in the range from zero to 254. Port number 255 is currently reserved to the implementation of the barrier synchronization. Prior to using any of its own ports, the process may bind it to:

- The parallel PID, node and port of the destination process, for messages that will be sent throughout the port.
- A buffer in user space for storing incoming messages.
- A program-defined function acting as receiver handler for the port.
- A program-defined function acting as *error handler* for the port. A GAMMA error handler is like a receiver handler, but it is issued in case of communication errors rather than upon successful message receptions. The purpose of error handlers is to help building application-level error recovery policies.

The destination process of a GAMMA communication as resulting from the binding of a port may belong to the same process group of the sender as well as to a different group. This enables general MIMD parallel programming in addition to SPMD.

After a port is bound, its number fully defines the destination of messages sent through the port, as well as the user-space buffer where messages incoming through the port are stored at their arrival and the actions performed by the process in order to consume them.

Since the kernel is notified the address of each destination user-space buffer where messages are to be stored in, the activity of storing incoming messages into their destination buffers is performed directly by the GAMMA device driver rather than by the user-defined receiver handlers. In order to avoid that a subsequent incoming message overlaps the previous one in the same user-space buffer, the receiver handler may bind the port with a fresh user-space buffer where to store the next incoming message. This way message overlapping is avoided, as receiver handlers are issued immediately upon each message arrival.

3.1 Synchronous receive in GAMMA

With Active Messages the receiver handlers act as independent threads of the application triggered by message arrivals rather than being mere functions invoked by “receive” operations. This means that the programmer has to spend an additional effort to ensure that receiver threads correctly cooperate with the main process thread. A very frequent problem is when the main thread needs to synchronize with a message arrival before continuing computation (e.g. when the process needs to receive data before processing them). A general solution is to use application-defined synchronization flags as follows:

1. A flag F of the application is initially reset.
2. In order to wait for one incoming message from a port P, the receiver process starts busy-waiting in a loop until F is set.
3. The receiver handler bound to port P sets F upon message arrival.

GAMMA offers a more flexible and reliable solution in the form of two semaphore-oriented library functions, namely `gamma_wait()` and `gamma_signal()`. Such functions give safe access to per-port semaphores embedded into the GAMMA library. The example above becomes as follows:

1. In order to wait for one incoming message from port P, the receiver process issues `gamma_wait(P,1)`
2. The receiver handler bound to port P issues `gamma_signal(P)` upon message arrival.

For performance reasons the current version of `gamma_wait()` is implemented as a busy-waiting with explicit polling of the NIC in order to reduce message latency.

3.2 GAMMA broadcast services

GAMMA supports group-based broadcast: the destination node may be set to the constant `BROADCAST`, meaning that outgoing messages will be sent to every process in the group identified by the destination parallel PID. Efficiency of the GAMMA broadcast service is guarenteed by the direct use of the native Ethernet hardware broadcast service.

4 Optimizations along the communication path

Various attempts to address the inefficiency of standard inter-process communication mechanisms in NOW architectures were carried out in the last few years. Having most of them to be committed to efficiency, at least to a certain degree, most of them share three typical features, namely:

- Communication protocols are as simple as possible. Long protocol functions are time-consuming, and their low degree of locality in the access to huge data structures generates a large number of cache misses and pollutes the working sets of user processes, thus leading to a poor exploitation of cache with degradation of overall system performance.
- Host addressing in the network is as simple as possible. Indeed addressing conventions in LAN might be much simpler than in WAN.
- The number of intermediate copies of messages along the communication path has been minimized. Making temporary intermediate copies of information during the execution of complex protocols is a time-consuming task and pollutes the working sets of user processes as well.

Other two frequently encountered optimizations are the use of *light-weight system calls*, which save only a subset of machine registers and do not invoke the scheduler upon return, and the use of a *fast interrupt path*, that is an optimized code path to the interrupt handler of the network device driver. Another common optimization is to allow *polling the communication device* for incoming messages in order to save the overhead of interrupt launch and service. The use of polling by applications is possible only during explicit receive phases, and hence implies a proper structure of the parallel program.

GAMMA uses both light-weight system calls and a fast interrupt path. Moreover a low-level polling policy has been implemented in the communication layer. A polling activity is executed by the `gamma_wait()` function in order to anticipate as much as possible the reception of incoming messages during the waiting phases of the receiver process.

The port-oriented version of Active Messages implemented by GAMMA allows an additional optimization, namely: the header for Ethernet frames may be pre-computed and statically associated to a communication port as soon as that port is bound to a destination. This way the overhead of the GAMMA protocol for segmenting long messages into a sequence of frames during send operations is greatly reduced. Moreover, a process sending a message has only to specify a previously bound output port, besides a pointer to a message buffer and the message size, with a slight saving in latency time due to fewer parameter passing.

4.1 Minimal "zero copy" communication protocol for reliable LANs

Any messaging system based on the "send-receive" paradigm, as well as any implementation of Active Messages which does not address issues related to the schedule state of the receiver process (see [9] for instance) must provide at least one level of temporary message buffering on the receiver side of a communication. The reason for this is that message consumption may not promptly occur at message arrival: the receiver process may not be running at that very time and thereof the execution of a "receive" operation or a receiver handler may have to be deferred. Usually such intermediate buffering is hidden into the system-level communication software layers.

However Active Messages would in principle eliminate the need for *all* intermediate message buffering, even on the receiver side if the execution of the program-defined receiver handler could be triggered right away upon message arrival, regardless of the schedule state of the receiver process. Indeed in this case the incoming message would be promptly consumed and temporary buffering on the receiver side would be no longer needed. Such optimization would allow a so called true "zero copy" communication path. With such organization of Active Messages, that we call *real time* Active Messages, both the user-space receiver buffer and the program-defined receiver handler are notified to the kernel before communications take place. At message arrival, the NIC's device driver stores the message directly in the user-space receiver buffer and activates the receiver handler on behalf of the (possibly not running and even swapped out) receiver process, possibly issuing a temporary context switch.

A few drawbacks may affect real time Active Messages, namely:

- The memory regions corresponding to both the user-space receiver buffer and the code of the program-defined receiver handler must be marked as non-swappable and must be pre-charged into RAM before any message arrival.
- Receiver handlers must be really short pieces of code. If receiver handlers take too long a time for execution, they could become the bottleneck of the messaging system and message overflow would occur, unless explicit flow control is provided.

GAMMA implements real time Active Messages, with a true "zero copy" communication path. The communication protocol is very simple, thanks to the reliability of modern 100base-TX technology which ensures a negligible rate of communication error in case of high-quality cabling. With good cabling, frame corruptions arise only from collisions when using shared Ethernet. In such case frame retransmission is automatically performed by the NICs up to sixteen times, after which an error condition is raised by the NIC. On the receiver side, each collided frame makes the NIC raise a CRC error. Therefore only simple test on send failures as well as CRC errors is mandatory at the level of the GAMMA network driver in order to immediately retry a failed transmission as well as discard corrupted or collided frames. Out-of-order frames cannot occur at all. The only concerning issue is flow control, the lack of which may in principle cause frame losses due to receiver overflow. Given that the throughput of RAM-to-NIC as well as NIC-to-RAM DMA transfers are greater than the 100base-T throughput (about 30 MByte/s RAM-to-NIC and 14 MByte/s NIC-to-RAM DMA transfer rates, against 12.5 MByte/s Fast Ethernet throughput) the flow can be implicitly controlled by the device driver of the sender NIC by simply checking for room availability in the transmit FIFO queue before starting a DMA transfer for transmission. Such check ensures flow control as long as the execution time of the receiver handler is small enough not to reduce the receive throughput below 12.5 MByte/s. In the current prototype of GAMMA it is up to the programmer to guarentee that the completion time of receiver handlers is small enough. However we are currently enhancing the GAMMA protocol with flow control features based on fixed-size sliding windows, in order to prevent message overflow even with slow receiver handlers.

The communication performance of the earliest prototype of GAMMA [4], called GAMMA 0.1, was quite good although not excellent. One-way user-to-user latency, measured by a simple "ping-pong" GAMMA program, was 33.5 μs and asymptotic throughput was 9.9 MByte/s, significantly better than Linux TCP/IP sockets but not optimal. GAMMA 0.1 exploited Programmed I/O instead of DMA for moving messages between RAM and NIC.

The next version of GAMMA, namely GAMMA 0.2, exploited DMA transfer plus pre-computed headers and more care to code and data alignments. GAMMA 0.2 yielded 12.1 MByte/s asymptotic throughput, a great performance improvement for large messages. One-way user-to-user latency was 23.5 μs, about six times smaller that Linux TCP/IP sockets, and yet in the range of many messaging systems running on NOWs and MPPs. The throughput curve of GAMMA 0.2 is reported in Figure 1, curve number 2.

Based on the DMA version of the first GAMMA prototype we have implemented a "ping-pong" application where incoming messages are received in one buffer then moved to a different buffer in order to simulate a temporary copy of each message on the receiver side. The throughput curve of such simulated buffered messaging system is reported in Figure 1, curve number 1. The performance degradation is clearly apparent: a throughput loss of more than 3 MByte/s (24% of the raw 100base-T bandwidth) is experienced by large messages, and the throughput curve is much closer to the one of Linux TCP/IP. This is a measure of how better a “zero copy” solution may perform on 100base-T LAN with respect to more traditional, buffered solutions adopted by most industry standard protocols.

4.2 Filling up the communication pipeline

From curve 2 of Figure 1 the behaviour of communication performance of GAMMA 0.2 with short messages is as follows: the throughput increases quite slowly for messages of increasing size up to 1500 bytes, where throughput raises the value of 6.6 MByte/s. However for messages of increasing size beyond 1500 bytes the throughput increases much faster.

The reason for such behaviour is simple: the communication path between the sender and the receiver nodes is a pipeline comprising the following three stages:

1. The RAM-to-NIC path on the sender node, where data flow through a DMA engine.
2. The hardware path from the sender NIC to the receiver NIC, where data flow along the physical LAN according to the Ethernet protocol.
3. The NIC-to-RAM path on the receiver node, where data flow through another DMA engine to reach their destination.

In the GAMMA protocol, messages longer than 1500 bytes are segmented on send and reassembled on receive in order to fit the maximum allowed size of Ethernet frames. A long, segmented message counting many Ethernet frames fills up the pipeline better than a short, unsegmented one. Consequently the throughput increases fastly for increasing message size beyond 1500 byte due to pipeline filling. We could also say that as messages become longer than 1500 bytes the behaviour of the communication path turns from “store-and-forward” to “cut-through”, due to message fragmentation.

A better exploitation of the pipeline structure of the communication path would lead to better throughput for small to medium size messages. A good communication throughput for small messages is of great importance as it would allow to exploit a finer grain of distribution in parallel applications.

One possible approach to improve the exploitation of the communication pipeline is to implement a finer-grain message fragmentation, by allowing even messages shorter than 1500 bytes to be disassembled into small frames (as already proposed in [3]). Another possibility is to properly program the NIC driver so as to enable what we call the “early send” and “early receive” capabilities that most modern NICs offer. With “early send” enabled, a NIC starts transmitting an Ethernet frame as soon as the first few bytes of a packet are uploaded to its transmit FIFO queue, rather than waiting for the whole packet to be completely uploaded. With “early receive” enabled, a NIC raises an IRQ on frame reception as soon as the first few bytes of the frame entered its receive FIFO queue, rather than waiting for the whole frame to enter. By enabling “early send” on the sender side and “early receive” on the receiver side, the behaviour of the whole communication path turns from “store-and-forward” to “cut-through” already with very short packets, thus the throughput curve is expected to improve especially for small to medium size messages. Moreover latency time is expected to decrease as well.

The communication performance of GAMMA sensibly improved after enabling the "early send/receive" capabilities of the network devices. With the next version of GAMMA, namely GAMMA 0.3, one-way user-to-user latency dropped from 23.5 to 16.8 μs. Curve 3 in Figure 1 reports the GAMMA 0.3 throughput curve. The improvement of throughput over GAMMA 0.2 for small to medium size messages due to the enabling of "early send/receive" is clearly apparent. For instance, throughput for 1500 byte long messages increased from 6.6 to 10.9 MByte/s, an improvement of more than 65%. The so called "half-power point", that is the message size at which half the asymptotic throughput is achieved, shifted down from 670 to 192 bytes.

4.3 Polling the network device for incoming messages

Even when using low-overhead fast interrupts (see Section 4), the time overhead resulting from IRQ management is quite high. In most off-the-shelf PC architectures the hardware latency for IRQs is not negligible, in the order of (few) microseconds. Such so called "IRQ latency" adds to communication latency. With proper arrangement of the parallel program, such additional latency may be hidden by overlapping it to the ongoing computation on the receiver process. However there are some cases in which this is not possible, especially when a synchronous receive phase must occur.

In order to avoid IRQ latency the only possible technique is to explicitly poll the communication device for incoming messages instead of relying on IRQ-based asynchronous mechanisms. Of course polling the NIC may be done only on synchronous receive phases, which however are the only cases when interrupt latency may affect overall performance.

The delay in IRQ service due to IRQ latency results into buffering the initial part of the incoming packet into the NIC's receive FIFO queue. However the DMA transfer started by the IRQ service routine of the NIC driver may consume such buffered chunk at the full speed allowed by the DMA engine. If the maximum DMA transfer rate is higher than the physical LAN throughput and the "early receive" feature of the NIC is enabled, the initial delay due to the IRQ latency is rapidly recovered by the fast DMA transfer already with short messages. This is the reason why by implementing a polling mechanism in GAMMA we were expecting only a latency saving, with a very limited influence on the overall throughput characteristic.

Our latest version of GAMMA, namely GAMMA 0.4, implements a polling mechanism part at kernel level and part at library level. At kernel level, the polling loop runs uninterruptably but only for a short (50 μs) time interval. Such short-lasting uninterruptable polling loop is issued by invoking an additional GAMMA system call. The `gamma_wait()` library function, issued by applications on synchronous receive phases, invokes the polling system call repeatedly in a library-level and hence interruptable loop.

As expected, the impact of polling on the throughput curve has been negligible. The most important achievement is that one-way user-to-user latency has dropped from 16.8 to 12.7 μs, due to saving the IRQ latency (about 4 μs). With such last optimization GAMMA becomes the fastest messaging systems currently running on 100base-T Ethernet, with unprecedented performance in terms of latency as well as throughput. The closeness of the throughput curve to the optimal 100base-T Ethernet throughput curve (curve 4 in Figure 1) computed assuming a minimum one-way user-to-user latency of 7 μs accounts for the extreme degree of efficiency in the GAMMA communication layer.

5 Conclusions

Our investigation in implementing an efficient inter-process communication layer to implement a network of PCs suitable for parallel processing has demonstrated that excellent results may be achieved even with low-cost LAN technology like 100base-T Ethernet.

Moreover we have demonstrated that a performant messaging system may be implemented even following a traditional software architecture, namely embedding most of the communication software at kernel level into a industry standard OS and building the programming interface of the messaging system as a user-level programming library. By such traditional software organization GAMMA allows a smooth integration with the existing multi-tasking environment and provides parallel multi-programming features supporting SPMD as well as MIMD applications.

The superiority of GAMMA compared to other much more radical approaches, like U-Net for instance, proves that in order to efficiently exploit off-the-shelf communication devices it is not necessary to eliminate the OS from the communication path; rather, the key issue is to choose the right communication paradigm and mechanisms, and to implement them efficiently by properly exploiting the features of modern interconnection devices.

Many of the optimization techniques that we have discussed before are of general interest in implementing communication layers and NIC drivers. Indeed techniques like a finer segmentation of messages as well as the exploitation of "early send/receive" features of the network devices help increasing the communication performance of any messaging system running in a LAN environment, especially for small to medium size messages. The adoption of light-weight communication protocols, made possible by the reliability and the simplified device addressing of modern LANs, has been already explored by many authors (see [6, 10] for instance). So called "zero copy" protocols are not a new idea, although they require suitable low-level communication mechanisms in order to be effectively implemented and work in a multi-tasking OS.

It may be argued that the programming interface of GAMMA is too low-level. We have written a number of parallel applications on GAMMA for benchmarking purposes and we found the GAMMA programming interface fairly flexible though simple once we became familiar with the Active Message paradigm.

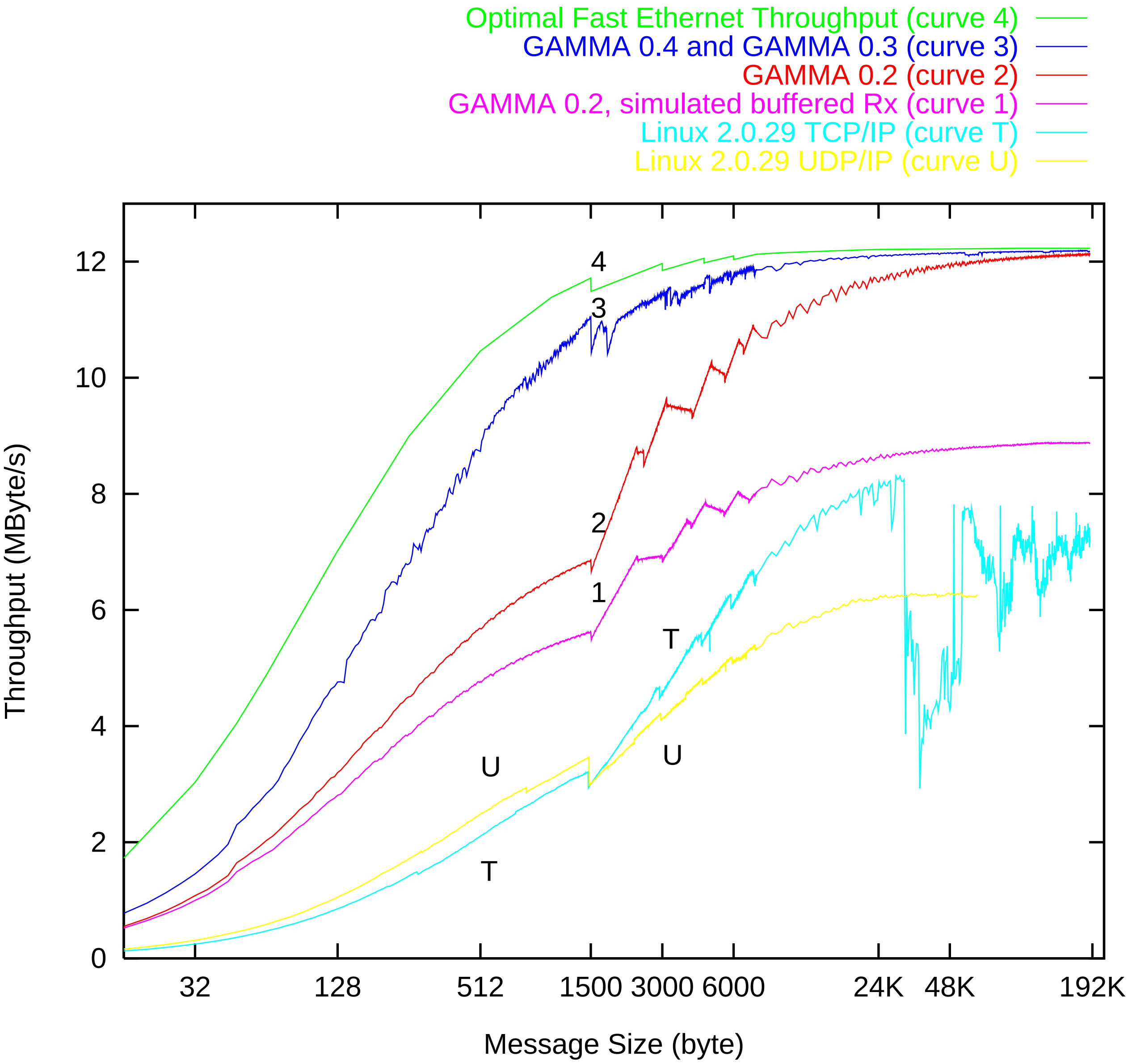

Fig. 1. "Ping-pong" throughput curves of various versions of GAMMA compared to Linux TCP/IP and Linux UDP/IP on the same hardware platform.

It may be also argued that GAMMA provides non-standard communication mechanisms. Indeed this is a severe obstacle to the portability of GAMMA parallel applications. For this reason one of our next goals is to implement a more standard, high-level parallel programming interface like MPI atop GAMMA. The challenge posed by this task is not to introduce any significant software overhead, at least for a small yet sufficiently complete subset of MPI mechanisms.

6 Acknowledgements

We are grateful to Donald Becker for having written the driver of the 3COM 3c595-TX Fast Etherlink 10/100base-T PCI network adapter for the Linux operating system. That driver was our only documentation about the assembly-level interface of the NIC and indeed the custom driver we developed for GAMMA is a slightly modified and optimized version of that one.

We are also very grateful to Paolo Marenzoni, Giovanni Rimassa and Michele Vignali (PhD and undergraduate students of University of Parma, Dept. of Information Engineering) for their invaluable practical suggestions and the "tricks of the trade" they revealed to us in the early stages of our project, without which this work would not even have started. Also Luca Landi and Alessandro Polverini (undergraduate students at DISI) contributed to the installation of the systems and provided useful information and suggestions on some practical implementation aspects. Most of the basic ideas and optimizations described here, as well as a substantial contribution to the implementation phase, are due to Giovanni Chiola (full professor at DISI).

The hardware for GAMMA has so far been kindly provided by our Department in order to allow this project to start without any specific financial support from external institutions.

References

1. Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, 1992.
2. T. Anderson, D. Culler, D. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), February 1995.
3. G. Chiola and G. Ciaccio. GAMMA: a low cost Network of Workstations based on Active Messages. In *Proc. Euromicro PDP'97*, London, UK, January 1997. IEEE Computer Society.
4. G. Chiola and G. Ciaccio. Implementing a Low Cost, Low Latency Parallel Platform. *Parallel Computing*, (22):1703–1717, 1997.
5. L.T. Liu and D.E. Culler. Measurement of Active Message Performance on the CM-5. Technical Report CSD-94-807, Computer Science Dept., University of California at Berkeley, May 1994.
6. P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte, and P. Rossi. An Operating System Support to Low-Overhead Communications in NOW Clusters. In *Proc. of the 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97), LNCS 1199*, pages 130–143, February 1997.

7. R. P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. In *Proc. of Hot Interconnect II*, August 1994.
8. S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997 (to appear).
9. S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet Computation. In *Proc. Supercomputing '95*, San Diego, California, 1995. ACM Press.
10. S. Rodrigues, T. Anderson, and D. Culler. High-performance Local-area Communication Using Fast Sockets. In *Proc. USENIX'97*, 1997.
11. T. Sterling, D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proc. 24th Int. Conf. on Parallel Processing*, Oconomowoc, Wisconsin, August 1995.
12. T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–64, February 1995.
13. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, December 1995. ACM Press.
14. T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, Gold Coast, Australia, May 1992. ACM Press.
15. M. Welsh, A. Basu, and T. von Eicken. Low-latency Communication over Fast Ethernet. In *Proc. Euro-Par'96*, Lyon, France, August 1996.

This article was processed using the LaTeX macro package with LLNCS style