Communication strategies (abstractions and protocols) to support medium/fine grain parallelism on SMP workstations clustered by high speed networks

by POKAM TIENTCHEU G. A.

Diploma Thesis

Technical University of Berlin,

and

German National Research Center for Computer Architecture and Software Technology (GMD-FIRST)

under the direction commitee of:

Prof. Dr.-Ing. Stephan Jähnichen,

Dipl-Inf. Wolf Pfannenstiel

December 1999,

Contents

List of Figures

List of Tables

“Le chercheur africain n’ a pas le droit de faire l’économie d’une formation technique suffisante qui lui ouvre l’accès direct aux débats scientifiques les plus élevés de notre temps, où se scelle l’avenir culturel de son pays. Aucune arrogance ou désinvolture (...) rien ne saurait le dispenser de cet effort. Tout le reste n’est que complexe, paresse, incapacité : l’observateur averti ne s’y trompe pas. En effet, on doit dire aux jeunes generations qui s’ouvrent à la recherche, armez-vous jusqu’aux dents.

Prof. Cheikh Anta Diop

To my beloved parents, Marie-Madeleine and Daniel POKAM

Acknowledgments

At the time I made the decision to concentrate my interest on cluster computing after having previously followed a graduate course on parallel computer architecture at the Technical University of Berlin, Professor Stephan Jähnichen was crucial in helping me by providing me with the opportunity to join the GMD-FIRST of which he is currently the director. I am deeply indebted to him for that, as well as for the advice to focus my interest on this topic, since my studies in the field led me to this Diploma Thesis.

When I joined the GMD-FIRST, Professor Wolfgang Schröeder-Preikschaft, a member of the Operating System Group, has provided me with his expertise in the field. He deserves my best thanks for his support.

During the time that I spent working on my Diploma Thesis at the GMD-FIRST, Antônio Augusto Fröhlich became my advisor. He devoted much of his time putting forth substantial efforts in order to imbue me with the meticulousness that is required in a scientific work. His patience and his advice have been invaluable. Without his support this Diploma Thesis would not have been possible. I will never forget his contribution to my education, and sincerely express to him all my gratitude.

I sincerely thank the Friedrich Ebert Stiftung for the highly competitive Fellowship arwarded me for my Diploma degree.

I would like to especially thank my brother, Christian Pokam, and my fiancée, Carine Eke, for having provided me with the support necessary for overcoming the frustrating phases of my work. I am very grateful to them for that.

I would not forget the multiple encouragements which were addressed to me by my friends all over the world. I think of Dr. rer. nat. Alain Tchouassi, Dipl. pol. Paul Simon Handy, Dipl-Ing. Simon Koundjou, Dr. Ing. David Pouhè and many others. I also particularly thank Bethanie Mills for her help with the correction of this work.

Finally and foremost, I thank my beloved parents for their permanent and unconditional support, counseling and guidance, encouragement, love and unshakable belief in me. My lovely mother has been giving me incredible support and has been displaying such a trust in me during the last phase of this work. To entirely disclose my gratitude to her will remain an unachievable task.

Chapter 1

Introduction

In recent years, systems based on massive parallel multiprocessor (MPP) have dominated the market of parallel computing. Even though the gigantic architecture exhibited by these machines includes from a hundred to a thousand microprocessors, they possess some unique benefits made possible by the great advances achieved in microprocessor technology: at great cost, they have included very highly integrated circuits, high quality pipelined processors, expensive on-chip memory and tightly coupled processor-memory/node interface; achieving low latency at high bandwidth.

Nowadays, most of the architectural and technological features used to design and build these expensive machines are commonplace. Commodity processors available on the market at present include low-latency and high bandwidth on-chip caches, they use aggressive instruction level parallelism and pipelining to achieve, on average, one instruction per clock cycle, and they implement several branch prediction algorithm to hide latencies of accessing main memory on a miss, etc. Many processor vendors also provide their product with hardware support which enables multiprocessing, and thus make SMP hardware accessible at very low cost. Furthermore, decisive improvement made in the technology of interconnection network has considerably reduced the gap between off-the-shelf networks and embedded MPP's interconnects. High speed networks of today are capable of achieving more than a gigabit per second of bandwidth.

The combined improvements in microprocessor and interconnection technology have made powerful microprocessors and high performance interconnection networks now commercially available. New cost-effective supercomputers are, more and more, being made up of clusters of off-the-shelf processors interconnected by high-speed off-the-shelf networks. A consequence of this is the increasing amount of parallel software available that will certainly favor the migration of a variety of applications from their original workstations or client/server environments to the newly available cost-effective clusters of workstations. For instance, significant softwares that now exist in the parallel computing community such as **PVM (Parallel Virtual Machine)** and **MPI (Message Passing Interface)** have already been ported on top of many cluster architectures. All this has seriously worked in favor of a re-distribution of the parallel computing market; MPP machines are no longer the only ones involved in the parallel computing field. Clusters of computers have made it possible for the emergence of a new cost-effective category of vendor composed of research laboratories, from small to large industry, and academic institutions (universities, . . .).

Although the performances achieved by these components provide good news, in reality, things are not as simple as they appear to be. Clusters of workstations are, to a large extent, akin to MPPs, in as much as they are an aggregation of stand-alone workstations interconnected by an off-the-shelf network. In the same way that processes running on different processors do have to communicate in an MPP environment in order to complete a parallel job, in a cluster of workstations, the same scenario holds between processes running on different nodes. This communication mostly takes place in the form of dedicated communication primitives (e.g "send" and "receive", remote invocation ...) exported to the application program by

means of a communication library. The way this communication library is implemented, the way it interfaces to the application program and the way it exploits the underlying features provided by the hardware (host and node interface) is of a significant importance in the performance delivered by the whole system. Thus, in most cases, applications will normally not be able to obtain too much from the potential offered by the cluster hardware. Unless the underlying communication layer is intelligent enough to overcome this software limitation, the performance of the whole system could be less efficient than that in the simplest case of the uniprocessor system.

The purpose of this work is twofold: it investigates first of all the common communication strategies used to support high performance applications in the scenario outlined above, and, secondly, provided with this knowledge, it proposes an abstraction and a design of a building block for high performance communication in a cluster environment . This building block has served us as the basis to lay **CLIP**, the high performance communication library that results from this work.

The remainder of this work is organized as follows.

Chapter two provides information about the state of current communications strategies in high performance environments. It begins with a descriptive approach to problems involving traditional operating systems. The approach then takes into account the methods implemented nowadays to solve these problems.

Chapter three introduces a discussion related to the choice of our interconnection network and provides a description of our PC platform. The chapter begins by describing the several hardware components that make up clusters of workstations. Having done this, we give a rational for the choice of our hardware network and provide a description of our experimental PC platform.

Chapter four highlights the key abstraction and design points of **CLIP**. The chapter explains the philosophy of **EPOS**, which is designed to support high performance applications. Next, we introduce the key abstraction of **CLIP**, which can be viewed as a *communication component* of **EPOS**. The last part describes the design decisions made to implement **CLIP**.

Chapter five deals with the performance of CLIP. In this chapter, we will examine two important aspects of performance; these are the latency and the bandwidth.

The last chapter presents our conclusions and discusses future issues.

Chapter 2

High Performance Communication Strategies

To handle communication involves using resources present in network hardware, which are often mastered by a low-level software implemented in the operating system. This is justified by the fact that one goal of the operating system is to hide most hardware peculiarities being acted upon by application programs. Usually, an adequate means for the operating system to carry this out, is to make a sub-system responsible for providing a transparent view of the network to user applications. Inside the operating system, this is done by encapsulating all hardware details within special purpose routines, also known as device drivers. This is the way contemporary operating systems like UNIX provide higher-level applications access to the network. Yet, when designed to target high performance applications, the approach just described is no longer suitable because of several limitations, which we will address in the next section. With the new approaches, many of the difficulties involved in the previous ones have been overcome, with the result that many new communication models are being invented. Amongst them, and to be discussed below is the user level communication approach. Before taking this up, the old approach involving traditional operating systems will once again be looked into - studying precisely the design issues that degrade the performances of communication as we attempt to go faster.

2.1 The Traditional Operating System Approach

Contemporary operating systems are often designed in a modular manner (e.g UNIX). A module in this sense represents an abstraction of a subsystem component that encapsulates a well-defined functionality (e.g memory management, filesystem, network management ...). On most machines, these operating systems extend between the application programs and the hardware ressources, the former being established atop the operating system, whereas the latter is usually laid below it. Thus, all wishes of using a particular hardware ressource are carried out by the operating system on behalf of application programs. This operation might require invoking several system components, which may in certain circumstances be a very heavy burden for high performance applications. For example, for an application program to access the network hardware, it first executes a system call which is trapped to the kernel, from there, several other system modules (e.g file system, memory management ...) may be called causing at times that data and parameters be exchanged among them.

On most Unix machines, this trapping to the kernel occurs by invoking a function of the Berkeley Sockets API, which interfaces the user level. The socket layer may in turn invoke other subsystems which corre-

spond to the layers that are effectively responsible for transferring the data to the network hardware. These could be, for example, the transport, the network or the data link layer.

Moreover, in the context of high-performance computing, most operating systems that handle communication in the way just described above doesn't have an adequate communication architecture to support parallel computing. What they require in most cases, is another level of abstraction built on top of the usual socket interface. On most UNIX machines, this additional level of abstraction is one of PVM or MPI. It is obvious that in this case, the intermediary layers of software, which extend between the parallel programming interface and the network hardware do not add anything but only overhead to a high-performance application.

In short, for at least three main reasons, traditional operating systems like UNIX are not conducive to supporting high-performance applications. First, because of the complex organization of today's operating systems that makes in-kernel processing more complicated, the numerous manipulations of data - e.g. byte swapping, copy, etc. - that sometimes results from this becomes the dominant drawback. Second, the in-kernel processing, which is also required to enforce the secure access to the network, for instance, is mainly responsible for adding more overhead. And third, the actions undertaken when trapping to the kernel, i.e context switching from user to kernel level, are on most systems a non-negligible source of overhead that penalize the applications even more.

As a result, kernel-based communication protocols that are designed with these guidelines (e.g the TCP/IP stack on most UNIX machines is implemented at the kernel level), are rarely used as a basis for communication in high-performance environments. Instead, the usual approach adopted by the scientific community consists in exploring new paradigms of communication that could overcome the drawbacks explained above, and in porting existing communication protocols or creating new ones on top of the newly established communication paradigms.

This requires first that the techniques used today to implement communication network's protocol be rearranged. Perhaps the most important methods implemented today are the so-called "User Level Communication" methods.

2.2 The User Level Communication Approach

In the same way that the RISC architecture evolved from its predecessor, CISC, the user level communication approach also evolved from the traditional operating system approaches of accessing the network hardware (see previous section). The main idea here consists in withdrawing the trapping to the kernel and the in-kernel processing, which are required in the traditional method, of the application's access path to the network. As a result, in the various implementations of this new approach, the kernel is entirely bypassed or it is referred to only for carrying out specific tasks such as providing secure access to the network or multiplexing the network hardware among several applications. Since the critical path through the kernel to the network interface has been removed, applications have now *direct access* to the network adapter card without going through the operating system. Communication is now realized via dedicated communication primitives implemented at the software level, hardware level or a combination of both; however, most of the time, it is realized through a communication library built upon them.

The hardware components (e.g registers, memory, ...) of the network adapter card can be exported to application programs in numerous ways, without or with less operating system intervention. They include, amongst others [10]:

- to map the memory region of the device into the user virtual address space,

- to directly write/read into/from the memory region of the device (e.g by dereferencing a pointer at the base address of the device in memory),
- to access the device memory region using I/O instructions.

Of course, user level communication approaches are mainly characterized by their versatility. There exist numerous implementations of this communication abstraction, and they principally differ in three of the following points. First, at the hardware level, the dedicated communication primitives provided by each implementation often have different ways of making use of the underlying network's hardware components. That is, the way they map onto the network hardware is not of a monolithic form. Second, the interface to application programs which is provided by these communication primitives is not of a standardized form. The application programs usually interface the communication layer either by using directly the dedicated communication primitives or via an API whose details are mostly implementation specific. And third, the manner by which higher-level programs use the resources of the underlying communication layer is specific from one system to the other. This constraint is due to the programming model on which application programs rely, and of which a mapping to the communication primitives is almost always built in. These are important aspects of the system performance since they participate in the realization of the programming model upon which user programs are established.

Below, we shall discuss three different approaches to realizing the user level communication paradigm. The first one is the traditional send/receive primitives that usually realizes the message-passing programming model. The second approach is the Active Messages, which embodies in its message a reference to a handler that has to be executed immediately upon message reception. Finally, we have the Asynchronous Remote Copy. It allows a local process to access the private memory of a remote process via network transactions.

It is important to note that, in some cases, the communication primitives provided by these 3 families of communication abstractions can also be efficiently combined with higher level software codes to implement a variety of runtime systems (e.g compiler, communication library,...).

2.2.1 Traditional send and receive primitives

The traditional send and receive communication primitives are usually implemented on top of message-passing architecture[1]. In its simplest form, a send operation is usually associated with a receive operation. Before a transfer of data can take place, a sending side must specify at least two or three parameters: the address of the buffer to be transmitted, its size and the address of the receiving side. In turn, at the receiving side, the address of a buffer where to place the incoming data must also be indicated. Thus, a send/receive pair allows a point-to-point message communication between a local and a remote node connected via a direct (high speed) network.

There are numerous variants of the send/receive implementation. In the preceding paragraph we have just described the synchronous send/receive pair (also known as blocking send and receive), in which a posted receive must precede a send request. Other implementations are mostly relying on the order of invocation of these primitives and on the re-utilization's policy of their related buffer after that a request has been issued (send or receive). Depending on the nature of the send/receive pair, the synchronization is done implicitly on completion of either the send, the receive or both.

Because traditional send/receive primitives rely on explicit network transactions, they often suffer from having experienced high latency. This is mostly due to the cumulative delay incurred in transporting the

[1]The send and receive message-passing communication primitives can also be implemented atop shared-memory architectures. In this case, a send operation will be interpreted by the messaging layer as a *write* to a shared memory region, whereas a receive operation will involve *reading* from a shared buffer. Synchronization events are required in order to enforce protection of shared ressources.

Interface	Latency in μs	Bandwidth in Mbps
MPI-IP	218	100
MPI/BIP-IP	100	205
BIP	5	1005

Table 2.1: Performances of MPI/IP, MPI/BIP-IP, and BIP

message into the network adapter and sending it into the interconnection network. The efforts are thus essentially focused on minimizing this delay for future implementations.

MPI (Message Passing Interface) is certainly one of the most widely used message-passing communication abstraction in the scientific community. It demonstrates, however, very high latency due to the reason cited above. Indeed, moving a message into the network adapter is enormously costly in time because MPI was originally conceived on top of the TCP/IP stack, which in turn is implemented on most systems according to the traditional operating system approach described in the previous section. This is the reason why several research projects are now providing low-level send/receive primitives intended to support efficiently MPI. BIP [**?**, **?**] has been designed with this aim. The minimal latency achieved by BIP is $5\mu s$ for messages´ sizes of less than 128 bytes. The table 2.1 below shows the latency and bandwidth achieved by MPI on top of IP compared to that achieved by MPI on top of BIP/IP. Its also shows the raw latency and bandwidth performances realized by the BIP messaging layer.

The testbed used for the measurements, presented in the table 2.1 above, consisted of four workstations based on Pentium Pro 200 MHz interconnected by an 8x8 Myrinet Switch. The host is connected to the network adapter card via a PCI bus that is able to deliver data at a throughput of about 132 MB/s. The BIP messaging layer achieves around 95% of the peak PCI bandwidth, which is, in this case the bottleneck of the Host-memory/NIC system.

Despite the efforts made to improve the performance of the traditional send/receive primitives, the buffer management required in a send or a receive operation is still a constant source of drawback on many implementations.

2.2.2 Active Messages

Active Messages is usually classified as a request-response communication. The philosophy behind this concept, and from which the name Active Messages originates, is correlated to the association of a piece of “code” that has to be executed upon the arrival of a message at its destination.

This concept is realized in Active Messages by embodying in each message a reference to a handler which executes immediately upon message reception. Thus, a handler in Active Messages provides a means of carrying out a particular action on the message that is being received. A handler in Active Messages can only be one of two types: it’s either a request or a reply handler. All communications in Active Messages are further based on this request/reply semantic.

Upon arrival of a message at its destination, the request handler whose reference is embodied in the message is executed immediately. This causes the on-going computation to be interrupted on the receiving node, and thus allows the execution of the request handler to directly interact with it. For instance, the payload portion of the received message may be introduced in the data structure of the target process on the receiving node. Additionally, a request handler may send back a message to the source process on the source node. Its reception on the source node may also cause the execution of a reply handler which will proceed similarly to the request handler. However, a restriction imposed on the reply handler is that it would not be able to issue a new request message. This measure is taken in order to remedy the deadlock/livelock problem.

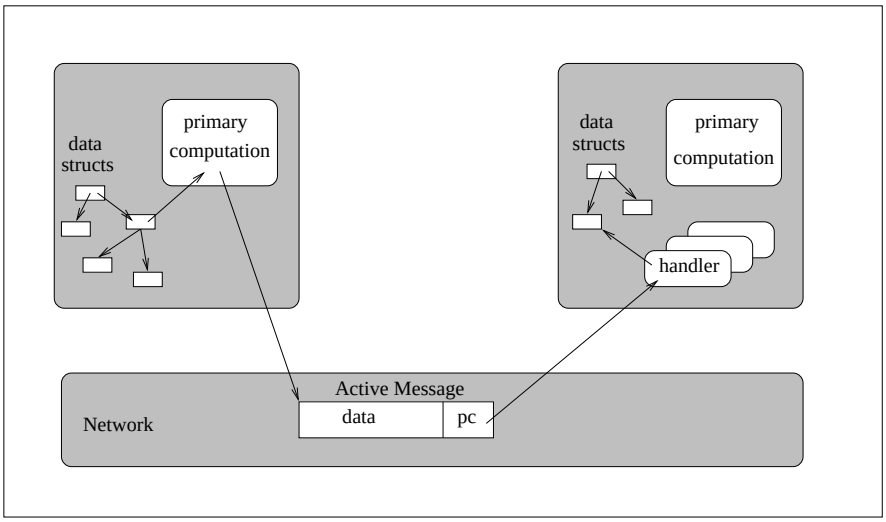

Figure 2.1: Structure of an Active Message

LogP parameters and bandwidth	Myrinet
latency in microseconds	13.8
send overhead in microseconds	5.6
receive overhead in microseconds	8.1
gap in microseconds	17.6
bandwidth in MB/s	31.7

Table 2.2: LogP parameters and bandwidth for Active Messages

In figure 2.1 we show the structure of an active message.

Even though Active Messages has been conceived with the claim that it best overcomes the insufficiencies of the traditional send/receive approach, its design can be, nevertheless, subject to certain questions. For instance, as already introduced in the preceding paragraph, a message in Active Messages embodies a reference to a handler (usually an integer value) which is used at the destination node to index a table of handler functions. There is absolutely no way for the process issuing the request to specify the address of the request handler on the receiving node if it doesn't know it in "advance". This further means that, either this step is done once at initialization time and thus a statically assigned table, which could likely reduce the flexibility of the design, is generated, or, it is made dynamically and the communication cost becomes dearer because of the numerous transfers that have to be done. Both cases have their advantages and disadvantages. We should also think about the way an incoming message is notified in Active Messages. That is, how does the invocation of a handler in Active Messages occur ? As far as we know, the invocation of a handler in Active Messages is usually the result of an interrupt, which causes the on-going computation to be suspended. This interrupt processing time may be non-negligible on some architectures, something that adds more overhead. Further, we should also raise the question of how a handler in Active Messages deals with memory storage ? This issue may be important, for instance, if the execution of a handler that is closely related to the on-going computation can not take place because the latter is not ready to interact with the handler at the time it is dispatched.

There exist numerous implementations of the Active Messages model. Its first implementation was realized at the University of California at Berkeley [**?**, **?**]. In the work undertaken by Lumetta et al. [**?**], the performance of an Active Messages implementation has been evaluated. We reproduce these results in the table 2.2 below.

The platform that serves as testbed for the measurements shown in the table 2.2 above consists of a cluster

of Sun Enterprise 5000 servers interconnected by Myrinet. Each host connects to the network adapter card via an SBUS, which achieves a bandwidth of 38 MB/s. The measurements were made according to the LogP model [**?**]. The LogP is a theoretical model of a distributed memory multiprocessor that specifies the performance characteristics of the interconnection network through four parameters:

- "L" is the delay incurred in sending a small message from its origin to its destination. It includes the time spent in the network interface to inject a message into the network, the delay in the network, and the time spent at the target node to consume the message off of the network.
- "o" is the overhead or the time spent by the *host processor* to transmit or receive a message.
- "g" is the gap or interval between consecutive message transmissions or receptions at a processor.
- "P" is the number of processors in the system.

Thus, from the table 2.2 above, the complete latency incurred in transferring a message from its source to its destination is 27.5 microseconds. That is, it is the sum of the send overhead, the latency time (time to inject the message in the network, time incurred in the network, time at the receiving node), and the receive overhead. The time given by the gap in the above table is dominated by the communication protocol implemented in the network adapter, since an attempt to inject a message can only take place if the previous message has already been injected into the network by the NIC's messaging protocol. The Active Messages communication layer realizes about 83% of the available peak SBUS bandwidth, which is, in this case, the bottleneck of the processor-memory/NIC system.

2.2.3 Asynchronous Remote Copy

The Asynchronous Remote Copy strategy provides a way for processes running on a local node to access the private memory of other processes running on a remote node. There are several possibilities of implementing this strategy depending on the nature of the communication architecture used. Apart from the communication architecture, the differences among the various implementations of this strategy can also be analyzed from the design of the basic operations upon which the communication primitives are built. These include the way by which a remote address is determined, how a remote write/read invocation is decoded, and how a process at the receiving node is notified of the completion of a write event. Moreover, if the memory hierarchy of the system includes a level of shared memory, these basic operations will further raise the issues of data coherency among the memories and caches owned by each node, since accessing local data physically to a given node that is possibly also locally cached, and further logically shared among other nodes, must remain coherent.

On systems in which the physical memory is shared among nodes, i.e shared physical address space, the basic operations of the remote copy semantic are essentially implemented by hardware. Principally, a write becomes visible to other nodes by making use of atomic primitives and synchronization events (e.g lock/unlock, barrier synchronization, ...). A snoopy protocol is usually implemented in order to solve the data coherency problem among caches and memories.

On software distributed shared memory systems (e.g cluster of workstations with a global address space provided by the physically distributed memory), there is much less hardware support for the basic operations of the remote copy semantic. These operations are carried out with the help of the operating system and application programs. The key design here is to rely on the shared virtual memory system, which in turn relies on the virtual memory. The shared address space is usually made up of globally accessible data, the granularity of the shared ressource varying from one implementation to the other (e.g page-based, object-based, ...). The shared ressource may be local or remote to a node, in the latter case, other nodes may

maintain a copy of it. There are several implementation's styles of the data coherency protocol which can be applied to such systems. The main operations of this protocol basically consist in retrieving a copy or copies of the missing shared ressource from other nodes, communicating with them, and possibly replacing the faulty shared ressource.

Implementations of the remote copy strategy can also be found on physically distributed memory systems made up of clusters of workstations interconnected by a high-speed network. The basic operations of the remote copy semantic are mostly implemented by software. The remote copy semantic in this case is realized by means of explicit message passing primitives. A remote read or write operation issued on a local node is interpreted within the communication layer as a network transaction. At the destination, the completion of a write event may be notified to the target process either via an interrupt, a thread invocation or polling. In the absence of a shared address space, the intra-node coherence protocol might be kept by means of the snoopy protocol, assuming the processors are sharing a common bus.

The Fast Messages (FM) developed at the University of Illinois provides an example of a (restricted) remote copy semantic implemented on a cluster of workstations interconnected by a high speed network [**?**, **?**]. Likewise the request primitive of the Active Messages (AM), a send primitive in FM also incorporates a message handler. The main difference in the AM's implementation resides in essentially 2 points:

- unlike AM, the FM handler does not interact directly with the application program. Upon message arrival, the handler executes, consuming the message off of the network. An application program that wishes to process the message should now extract it from a queue via an explicit call. A side effect of this approach, whenever compared to the implementation of the native active message, is the issue of buffer management that is emphasized at the communication layer;
- FM provides no request/reply semantic. These issues are decoupled, and it is the responsibility of the application program to provide a deadlock-free/livelock-free network.

The testbed used in FM consisted of a cluster of SPARCstations 20 interconnected via Myrinet. Each host is connected to the network adapter card via an SBUS, which achieves a peak throughput of 23.9 MB/s when using double-word write transfers. FM 1.0 achieves a latency of $25\mu s$ for packet sizes of less than 128 bytes, and a peak bandwidth of 19.6 MB/s for larger packets. Thus, FM is able to saturate about 82% of the SBUS bandwidth performance.

In general, the remote copy semantic can provide several advantages when compared to the traditional model based on paired send and receive primitives. Being able to access the memory location of a process remotely, can result to minimizing the buffer management at the destination node, and thus reduce the amount of memory copy. However, the heavy and difficult part of this approach is to be able to maintain memory and cache coherency among all the nodes of the system. The complexity of implementing this coherency protocol, and the significant overhead that can result from it, is sometimes the main reason why many designers prefer to rely on simple and carefully designed communication abstraction methods such as the traditional send and receive primitives.

Although we did not describe the various methods of communication so extensively, what is important to point out is that all tend to optimize the communication performance throughout the layers of the communication architecture. To achieve this, several research projects have adopted a simple implementation and have come up with great results for large and small packet sizes. This is, for example, the case of the BIP project [**?**], which achieves a latency of $5\mu s$ for packet sizes of less than 128 bytes and a bandwidth of

126 MB/s for larger packet size (64 KB), consuming about 95% of the bottleneck's peak bandwidth (see 2.2.1). However, this performance is not achieved without compromises. Indeed, the implementation decision in this case suffers from the lack of some important aspects such as providing secure network access, multiplexing among several applications, and end-to-end flow control. Others, conversely, have chosen to partially or completely integrate these aspects into their implementations. They consequently offer a worse performance to application programs than BIP. An example of research projects in this category includes the FM project [**?**], which achieves a minimum latency of around $25\mu s$ for packet sizes less than 128 bytes and a maximum bandwidth of 19.6 MB/s for larger packet sizes, realizing about 82% of the bottleneck's peak bandwidth (see 2.2.3). Several other implementations with more robust design decisions exist as well, e.g U-Net [**?**].

What we can note of the above analysis is that, all these various methods, although they aim to bring a solution to the problem of communication in a high-performance environment, often vary in the performance that they offer to applications. In this particular case, a legitimate question that could then be raised, would be to ask if it's even desirable to have an all-purpose communication abstraction interface common to all types of applications.

To agree with the above question could only be done by disregarding the specific nature of the application. In fact, experience has shown that applications usually demonstrate different communication requirements. For instance, a write based application will be less sensitive to latency than one based on request-response. In turn, this latter application will require less bandwidth than the former.

Thus, we believe that to provide an all-purpose communication abstraction can only partially satisfy the requirement of high performance applications. What is missing in this case is the flexibility to answer to a broad range of the application demands without being restricted to a unique communication abstraction. This can be achieved by having a communication interface that exports a repository of communication abstractions from which one has now the possibility of customizing a communication environment targeted for a specialized high-performance application. Unfortunately, this design also raises the question of the runtime system support that can deliver such a flexible communication interface, yet with the same aim of providing high performance to application programs.

This design aspect will be further discussed in chapter four, where we will introduce the philosophy of an innovative runtime system support that can provide a much more flexible communication interface. Next, we focus our attention on the hardware components of our cluster environment.

Chapter 3

Choosing an Appropriate Platform

As processors are becoming more powerful and networks even faster, clusters of workstations interconnected by a high-speed network (see figure 3.1) are now contributing significantly to increasing the influence of high performance computing. As a result of realizing such high-end workstation-based clusters, it is now possible to challenge the performance delivered by MPPs. There is certainly no doubt that these high speed networks now constitute an important element in the realization of systems targeted to deliver high performance to application programs. This means, from a system designer viewpoint, that having a better knowledge of the design trade-offs that characterize them is of a high importance. For example, a high performance interconnection hardware must be capable of transmitting packets from one end of the network to the other with as small a latency as possible. In addition to achieving excellent throughput it must provide a means to allow concurrent transfers across the network. These two performance parameters do not only depend on the nature of the interconnection network - bus-based, switch-based, etc. -, but also on other design factors such as the width of the link, the routing technique used, the type of flow control provided, etc. Furthermore, viewed from the host, the node is attached to the interconnection hardware via a so called "communication assist" (CA) or a network interface card (NIC). The nature of this interface - i.e. how tightly coupled the NIC and the processor-memory sub-system are - and its processing capability, also strongly influence the performance of the whole system.

Thus, the processor, the NIC and the high-speed interconnection hardware are forming together, the core of a cluster of workstations.

The choice of the most conducive interconnection and NIC technology for a particular platform is not always immediately apparent, as several complex architectural and technological aspects are now involved in the design of these interconnection hardware components. Nevertheless, in order to better understand some of these aspects, we will proceed in the following section to describe some popular interconnection technologies used today. Thereafter, we will give the rationale for the choice of our interconnection technology. We will then close this chapter by providing a description of the PC platform that is held together by this interconnection hardware.

3.1 Interconnection hardware for Clusters of Workstations

An interconnection hardware must provide a means for moving data from one processor to another processor, possibly a remote one. In the context of high performance computing, this singular “goal” is mostly assessed by two performance metrics: the latency and the bandwidth of the network hardware. The aim is to keep the first metric as low as possible, whereas for the second metric, the higher it is, the better

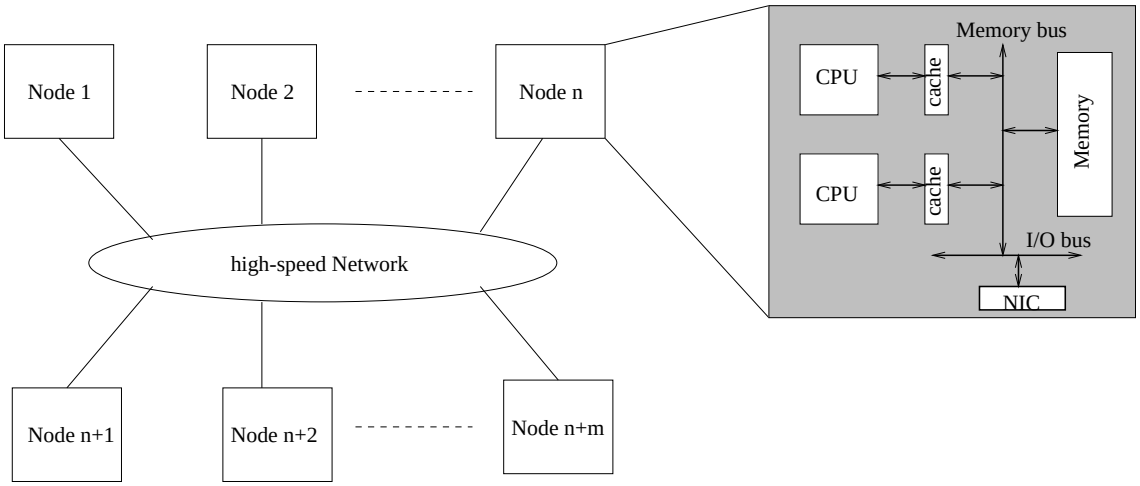

Figure 3.1: A generic organization of a network of workstations

the performance. From an architectural and technological viewpoint, this might mean, for instance, to have sophisticated routing and switching strategies, an efficient flow control algorithm, and a scalable network topology. In addition to this, the interconnection hardware interface to the host - i.e. the network adapter card - must also be able to map well onto the underlying network features and provide higher-level programs with an efficient support for communication protocol processing.

The large dimension of the components that take part in the design of a high performance, scalable interconnection hardware, does not make obvious the establishment of a taxonomy for the latter. The topology of the interconnect, as well as its routing or switching strategy, could each serve as criterion for establishing such a classification. Because this work focuses on communication strategies in high performance environments, we will thus attempt to provide a description of well-known network hardwares on top of which the communication abstractions described in the previous chapter are commonly realized. We will proceed gradually by first describing, whenever possible, the features of the network hardware and the built-in support provided by the network interface to application programs, and, finally, we will tackle the issues of high performance computing that are raised by the combination of the previous two points.

We will consider in turn the following four network hardwares: ATM, Gigabit Ethernet, Myrinet, and SCI. The first three networks are usually used to support the message passing programming model via the send and receive primitives. The last is often used to realize the shared-memory programming model via the remote copy semantic. We will close this section by giving a rationale for the choice of our network hardware environment.

3.1.1 Interconnection Hardware for message-passing

3.1.1.1 ATM

ATM is the acronym used to designate Asynchronous Transfer Mode. It was proposed by the CCITT[1] as the recommended transfer mode for implementing the B-ISDN[2] model, which defines a communication protocol for transferring video, data and voice simultaneously, at high bit rates. The ATM protocol suite is built in a layer approach (see Figure 3.2), where the ATM itself constitutes the layer 1 of the reference model.

[1] Consultative Committee, International Telegraph and Telephone
[2] Broadband Integrated Service Data Network

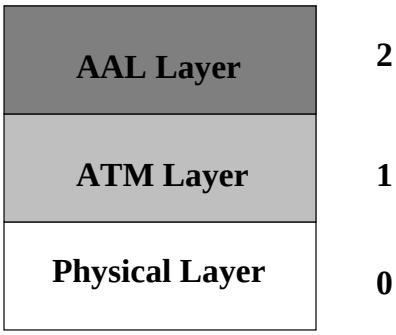

Figure 3.2: ATM protocol suite

Service Class	Timing Relation	Connection mode	Bit Rate	AAL Protocol
A	Required	Connection-oriented	Constant	Type 1
B	Required	Connection-oriented	Variable	Type 2
C	Not Required	Connection-oriented	Variable	Type 3/4, Type 5
D	Not Required	Connectionless	Variable	Type 3/4

Table 3.1: Classification of AAL services

Other layers of the protocol include an ATM Adaptation Layer (AAL) which is layered atop the ATM, and a Physical layer which is laid at its bottom. The Physical layer (layer 0) specifies a particular transmission medium for transporting fixed-size packets, called cell[3], from one node to another across an ATM network. Common data rates supported at this level include the OC-3[4] (155.52 Mbps) and the OC-12[5] (622.08 Mbps), but other data rates exist as well. The ATM layer itself (layer 1) defines the transfer protocol of cells and manages logical connections. The AAL layer (layer 2) interfaces to higher level programs, exporting to them the services of the underlying ATM layer. Among other functions, the AAL layer is responsible for performing the segmentation of higher level messages into cells and, inversely, reassembling them into messages destined for higher level application programs. The actual ATM specification defines up to 4 different AAL protocols for addressing the class of services belonging to the B-ISDN model. These are reproduced in the table 3.1.

ATM was initially intended to span over WANs[6], and was employed mainly for transferring voice and multimedia data. But, because of its ability to transport data at high bit rates and its protocol's inter-operability for various types of services, including data communications, its use as a high speed interconnection hardware for clusters of workstations is now becoming an area of intensive research in the scientific computing community [**?**, **?**, **?**].

Building such an ATM-based cluster of workstations requires one to use an ATM switch as the interconnection hardware among nodes, and an ATM network adapter card as the intermediary component between a node and the interconnection hardware.

An ATM switch is principally assigned the task of transferring cells from one input port to one output port, therefore it is often called cell relay. The cell's routing decision is made according to the routing informations stored in its header. This information identify a virtual data path that has been assigned to a given connection established between two entities prior to a transfer. The header also includes an additional piece of information intended to help the ATM switch choose a cell which will be routed towards an output port, if more than one cell competes for the same output port at the same time.

[3] A cell in ATM constitutes the smallest unit of transfer across the network. It consists of 48 bytes of valid data or payload and 5 additional bytes for the header.

[4] Optical Carrier Level 3

[5] Optical Carrier Level 12

[6] Wide-Area Networks

API	Bandwidth in Mbps	Latency in μs (one-way) for a 4 bytes message
ATM AAL type 5	31.68	869
ATM AAL type 3/4	32.56	1034
BSD Socket	16.72	1960
PVM	12.16	2766
RPC	12.72	2957

Table 3.2: Performances of accessing the ATM network from different APIs

Most ATM network adapter cards, commonly employed to connect a host to an ATM switch in a cluster configuration, implement the AAL protocol of either type 3/4 or type 5. Some of these adapter cards are capable of performing segmentation and reassembly operations, guaranteeing message delivery, handling flow and timing control, and providing multiplexing/demultiplexing functionalities. These operations are usually carried out by means of built-in hardware supports found on the adapter card. These include, for instance, an embedded processor, a DMA engine for accessing the host memory directly, and a CRC verification/generation engine, etc. To offload these operations from the host CPU not only improve the performance, but also leads to a better balanced computation-to-communication ratio.

Evaluating the performance of an ATM-based cluster of workstations in a high performance environment can be done by referring to the performance metrics introduced earlier in this chapter, namely the latency and the bandwidth. In fact, it is not only the performance of the underlying network hardware that is of significant importance, but also that of the network interface and of the host software, which allows higher level programs to access the services provided by the ATM network. Thus, to assess these metrics implies that we take into account the performances of the API, the NIC and the network itself.

Current available ATM switches require about 10 μs for performing a simple cell routing decision. If this seems to be "tolerable" at present, it's still, however, of an order of magnitude higher than the latency of most high speed networks; as we will soon see in the next sections. Lin et al. [**?**] have shown in their study that passing from one adapter card to another can severely reduce performance, due to the limited processing capability offered by some adapter card in some cases. As an example, they showed that the bandwidth of a SUN Sparc 2 to which an SBA-200 ATM adapter card of the FORE System is plugged, drops to 25% of its performance if an SBA-100 ATM adapter card of the same vendor is plugged in its stead. The study of Lin et al. also gives a deep insight into the implementation of the complete ATM protocol stack. Their methodology consisted in measuring the end-to-end performance of several APIs commonly employed for accessing the ATM network (see figure 3.3).

The results obtained from this comparison are reproduced in table 3.2. The table shows the accessing performances of the ATM network from the APIs depicted in figure 3.3 above. These tests were made on a variety of hosts, including the Sparc 1+, Sparc 2, and 4/690. The ATM network adapter cards and switch used for these measurements were from the FORE System.

From the above table, one can see that the AAL layer of the ATM protocol suite gives the best latency and throughput results. However, the latency incurred in sending a small message of 4 bytes is still too large. In a high performance environment, an application which is heavily based on the request-response semantic will be severely penalized by this high latency. Indeed, in most high performance environments of today, the acceptable range of latency is usually about one tenth of microsecond or even less. Moreover, the positive achievable bandwidth of the AAL layer must also be relatively considered. In fact, we can observe that this value lies below a third of the peak ATM bandwidth (155 Mbps in this case). This means that a large potential of the raw ATM bandwidth still remains unused.

Two aspects of the ATM protocol suite can help us shed light on the results of Table 3.2 discussed above. The first aspect might be inherent to the ATM protocol itself. It is due to the establishment of a virtual

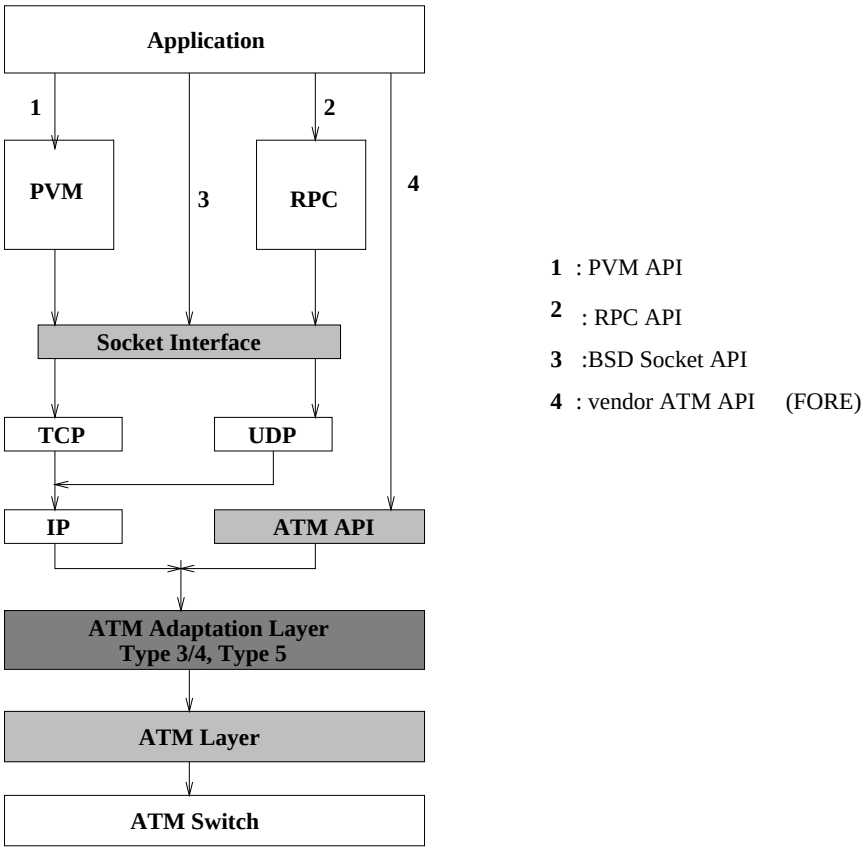

Figure 3.3: A layered protocol model for accessing the ATM network

connection before any transfer can take place. Indeed, the cost of establishing a virtual connection between two entities each time a message has to be sent, is likely to increase the overall communication time. This cost might be even excessive if the connection made between the two entities can not last long enough to compensate for the cost of having established it. The second aspect might be due to the implementation of the device driver that masters the network adapter card and which lies within the kernel. As already explained in the previous chapter, removing the whole communication processing protocol from the kernel can dramatically improve the performance. This approach has been explored by Matt Welsh et al. [36]. In the paper of Werner Almesberger [19], the efforts are essentially focused on the reduction of the number of memory copies (e.g just-in-time DMA, send locking, receive prediction, ...) and on the optimization of the number of memory accesses (e.g avoiding checksum calculation).

3.1.1.2 Gigabit Ethernet

Gigabit Ethernet is the latest product of the Ethernet technology. In the IEEE specification, the Gigabit Ethernet is defined as the 802.3z (1000BASE-X) and 802.3ab (1000BASE-T) standards. These standards prescribe the class of the physical medium on top of which the Gigabit Ethernet is expected to run. These are the Fibre Channel and the long haul copper UTP.

Gigabit Ethernet delivers a raw bandwidth of 1Gbps (1000Mbps) while remaining compatible to proven network's technologies such as the 10Mbps Ethernet or 100Mbps Fast-Ethernet. To achieve this, Gigabit Ethernet relies on a modification of the well-understood IEEE 802.3 and ANSI X3T11 Fibre Channel specifications for the physical layer.

The Gigabit Ethernet technology is widely deployed as a backbone interconnect in a campus or building-wide network area. In some cases, it can serve as a means to enhance a server-switch connection. Although yet not widely deployed under this form, the Gigabit Ethernet can also be used to build a high-end workstations' cluster. The Parallel PCs Cluster (PPCC) of the Indiana State University[7], which consists of 32 Pentium II dual-processors interconnected by a Gigabit Ethernet Switch fabric, provides an example of this last configuration.

To build a such high-end workstations' cluster based on Gigabit Ethernet, one essentially needs two different hardware components :

- a Gigabit Ethernet adapter card (NIC),
- and a Gigabit Ethernet Switch or Repeater.

The Gigabit Ethernet NIC provides a small subset of built-in hardware features intended to offload an amount of protocol specific functions from the host CPU. Costly communication operation such as checksum calculations requiring several memory accesses, packet header manipulations, and byte swapping, can now be implemented on the adapter card via an embedded processor. The memory copies to or from the network adapter that were performed by the host CPU on behalf of application programs, are now offloaded from it and reliquinshed to the NIC. A DMA engine integrated in the NIC is being given the responsibility of realizing this. The DMA engine is further capable of initiating multiple gather/scatter DMA operations to or from the host memory. Another feature of the Gigabit Ethernet adapter card consists in being able to merge several interrupt requests into one, and thus generating only a single interrupt after having received many packets. This saving in the amount of issued interrupts is intended to reduce the stealing of the host CPU cycles which, otherwise, could have been much more serious if each packet received were to cause an interrupt.

[7] http://www.indiana.edu/~uits/

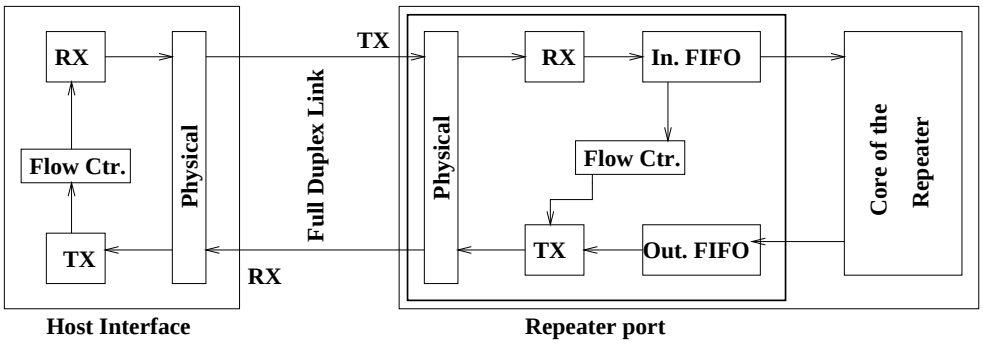

Figure 3.4: A Gigabit Ethernet Repeater

A Gigabit Ethernet adapter card is connected either to a Gigabit Ethernet Repeater or to a Switch. Optionally, a such switch or a repeater might be configured with a full or a half dupplex link. In the latter case, the switch or the repeater should have to implement the CSMA/CD algorithm inherited from the 802.3 standard for detecting and managing collisions. In the figure 3.4, we show how a generic repeater with a full-duplex link looks like. It's equipped with one input and one output FIFO queue [22] . An incoming packet is stored in the FIFO input queue before it is forwarded to all output FIFO queues but the local one. In this way, the Repeater provides a hub-like behavior.

Because Gigabit Ethernet has inherited the frame format defined by the 802.3 standard, its also has a maximum transfer unit (MTU) of 1500 bytes. The functionalities of segmentation and reassembly must then be provided either by higher level softwares or by the NIC itself, if a message's size of more than the size of the MTU has to be sent across the network.

3.1.1.3 Myrinet

Myrinet [**?**, **?**] is a high-speed interconnection network commercially produced by the Myricom Inc. It has its roots on the Caltech Mosaic massive multiprocessor project [**?**] and the USC/SI ATOMIC LAN [**?**]. Since November 2, 1998, Myrinet became the standard ANSI/VITA 26-1998.

A high-end network of workstations based on Myrinet is composed of several point-to-point links connecting nodes to the Myrinet interconnection hardware. Such a Myrinet-based cluster is essentially made up of three elements:

- the host hardware or node (corresponding to the workstation or PC),
- the Myrinet Network Adapter Card (NIC) that connects a node to the Myrinet interconnection hardware,
- and the Myrinet switch engine (the hardware interconnect).

A Myrinet NIC includes an embedded processor - the LANai Processor -, which can be programmed to control the flow of mesages between the host CPU and the network. In addition, it can also be used to offload an amount of specific communication functions from the host CPU. A Myrinet NIC also includes a small amount of on-board SRAM memory that can span from a few kilobytes (256 KB) to several megabytes in current implementations. The on-board memory usually holds the firmware program or MCP[8] (code

[8]MCP, Myrìnet Control Program, is the program that runs on the Lanai processor.

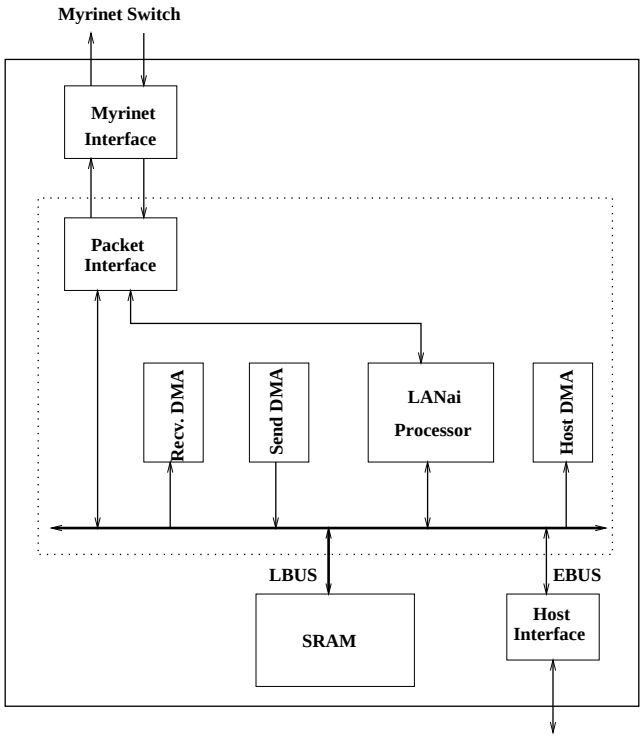

Figure 3.5: Structure of a Myrinet NIC

and data) which can be downloaded into it by the host CPU. This on-board memory also provides enough storage to serve as a staging buffer for messages en route to the network or for messages that have been consumed off of the network and are prepared to be pushed into the host memory. Other feature of the Myrinet NIC are the three DMA engines that are responsible for injecting a message into the network (send DMA), for consuming a message off of the network (receive DMA) and for initiating burst transfers directly into/from the host memory (host DMA) respectively. In the Myrinet standard, these DMA engines are designed to work fully in parallel in order to achieve their full performance. Nevertheless, the concurrent use of these DMAs is subject to a slight drawback. The three DMAs requiring one memory access per clock cycle each, and the on-board processor that requires two memory accesses per clock cycle, are competing for the SRAM whose access is granted by an on-board local bus, called the LBUS. The LBUS runs at twice the chip-clock speed, thus every memory access involving one DMA engine will stall at least one of the two others.

The interface of the NIC to the network is made up of a full duplex link interface that is able to achieve a transmission rate of 1.28 Gbps. A checksum engine is responsible for generating/verifying packet's checksum while they are transferred to or received from the network. The host bus interface of the Myrinet NIC is usually laid either on the I/O bus of the I/O sub-system, e.g PCI or S-Bus. Figure 3.5 presents the organizational structure of a generic Myrinet NIC.

A Myrinet Switch is made up of several bi-directional ports (current implementations have up to 16 ports) of 160 MB/s each. The decision of routing an incoming packet from one input port to another output port is made by consuming the first byte of the packet which contains a fixed route. This routing strategy is known as "*source routing*". Once the routing decision has been made, the remainder of the packet is spooled to the destination port. In the case where the selected destination port is already occupied by another packet, the Myrinet Switch blocks the current packet for an amount of time that is less than 50 ms. After expiration of this time-out period, the packet is simply discarded. The effect of blocking the packet in the network backs up to the source NIC and to the application program, which then stops to emit new packets. In the opposite

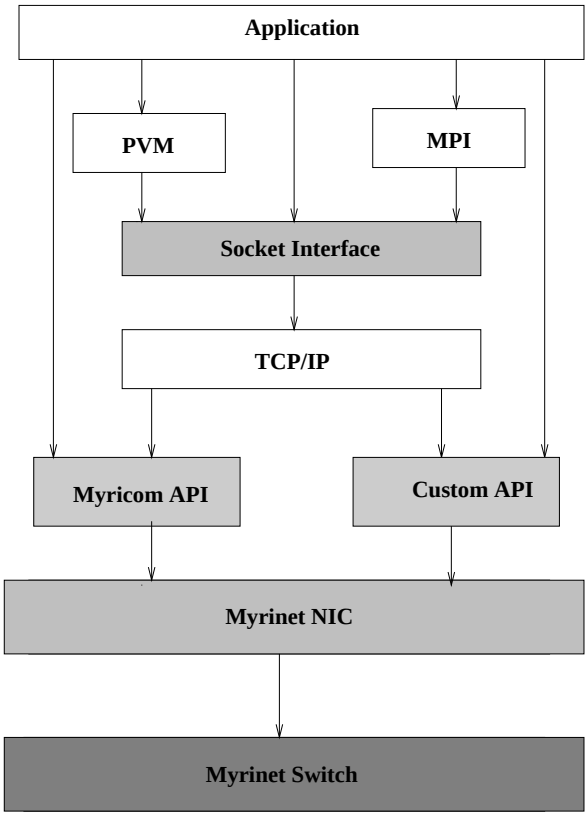

Figure 3.6: APIs for accessing Myrinet

case where the selected output port is not busy, the Myrinet Switch advances the head of the packet - the first flit - to this port while the remainder of the packet is "*cut through*" to the path from the input port to the output port. This form of switching strategy is known as "*wormhole routing*".

The raw latency and bandwidth metrics realized by the Myrinet network are very promising in terms of performance. At the interconnection hardware level, the minimum latency incurred by a message in the Myrinet switch is only about 0.5 μs. In addition, the point-to-point link that connects a Myrinet adapter card to the switch achieves 1.28 Gbps of raw bandwidth. In the same manner that we attained insight into the ATM network by measuring the performances of various APIs to transfer messages over it, we can proceed similarly in order to obtain such performances for Myrinet. As already argued in section 3.1.1.1, measuring such performances can be of significant importance if one needs to assess the overhead generated by the software running on the host and the network adapter respectively. We show in figure 3.6 possible ways of accessing the Myrinet switch from application programs.

In his work, Loïc Prylli [**?**] has evaluated the performances of some of the APIs shown in figure 3.6. Although we could not easily draw general conclusions about the results he obtained over all platforms, due to the differences in the architecture, these results provide, nevertheless, a close approximation of what performance levels the Myrinet network is able to sustain. We reproduce in table 3.3 the results of some of these tests performed on a Myrinet-based cluster of Pentium Pro 200 MHz processor. The custom API used in this case was BIP [**?**, **?**].

As in the case of the ATM network, it is the API closest to the network that yields the best results, namely BIP, which overcomes the software limitations that penalize the Myricom API. Not only does BIP have the best results, but these results are also very close to those that the hardware can export to application programs (95% of the PCI bandwidth that represents the bottleneck in this case, and only 5 μs of latency).

API	Latency in μs	Bandwidth in Mbps
MPI (Myricom-IP)	200	100
MPI (BIP-IP)	100	205
Sockets (TCP/Myricom-IP)	150	190
Sockets (TCP/BIP-IP)	70	272
API Myricom	80	120
API BIP	5	1005

Table 3.3: Performances of accessing the Myrinet network from various APIs

This tends to enforce the idea that the hardware of Myrinet as well as the software that masters it - i.e. the switch, the NIC and the Myrinet driver - are not constituting a decisive critical path in the communication process, or at least, if they are, the overhead incurred in this path is surmontable, as it has been proven by the BIP team. Also, the fact that Myrinet provides an efficient hardware flow control mechanism (backpressure) and a very low error rate (less than 10^{-5}), makes the communication very reliable and less prone to errors, thus eliminating the need for stapling additional layers of software on top of the basic hardware protocol.

3.1.2 Interconnection Hardware for shared-memory

3.1.2.1 SCI

SCI - Scalable Coherent Interface - is an IEEE (IEEE 1596-1992) standard which defines a protocol to connect up to 64K nodes. Nodes in a SCI-based cluster have access to a 64-bits wide distributed shared memory that represents the global address space of the system. The standard defines the implementation of a directory-based cache coherent protocol for maintaining cache coherency among nodes of the cluster, and a message passing communication model for realizing unidirectional point-to-point message transactions between nodes. The standard also describes remote memory accesses in the global address space provided by the SCI hardware.

Nodes in a SCI-based cluster are arranged around a ringlet. A ringlet is an unidirectional ring that can connect up to 8 nodes. In principal, it is possible to expand the ringlet to more than 8 nodes. But, since doing so would substantially increase the latency incurred by the application program that has initiated a communication on the ringlet, the amount of interconnected nodes within the ringlet is often kept to a maximum of 8. Larger configurations can be built by arranging nodes around ringlets of ringlets, or ringlets interconnected via switches (see figure 3.7).

The implementation of the SCI standard is supported at the hardware level by the definition of 3 layers of protocol abstraction: the physical layer, the packet layer and the transaction layer.

The physical layer defines the link specification. An SCI link has a bandwidth of 1 Gbps.

In the packet layer, the generic format of an SCI packet is described. A packet in SCI is made up of a sequence of several 16-bit units. Important fields of the SCI packet format include: a target ID, a source ID, data and CRC.

The SCI transaction layer defines 4 types of transactions: read, write, lock and move. All transactions with the exception of the “move” are completed in two network recoveries: a request sub-transaction and a response sub-transaction. A sub-transaction in turn consists of 2 phases : the request phase and the echo phase. A time-out mechanism is also provided at the initiator side to track the succees or failure of the request phase. Further, all transactions are atomic, dead-lock free and preserve memory and cache coherence along the cluster.

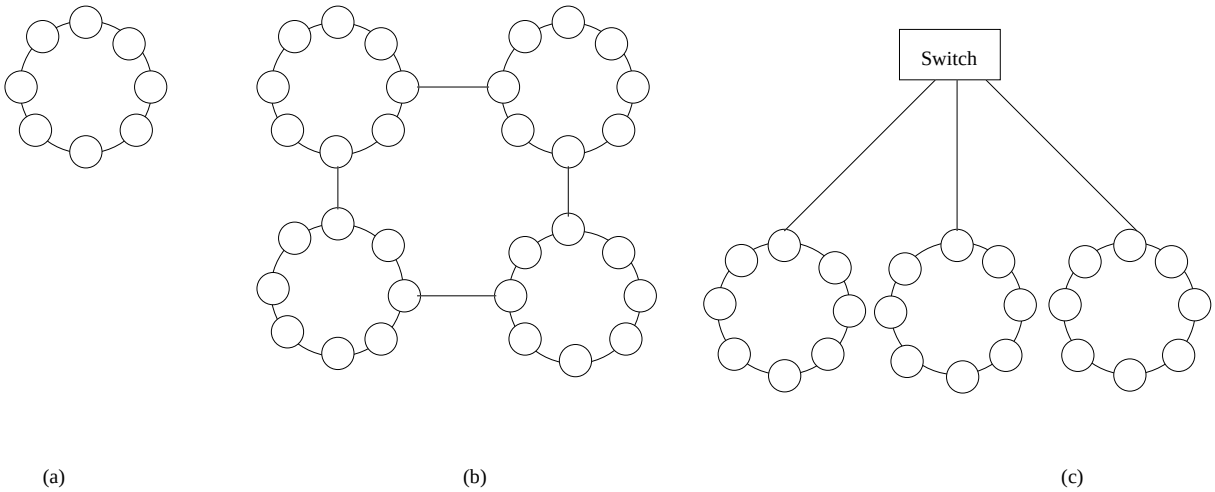

Figure 3.7: Configuration of an SCI-based cluster of workstations (a) ringlet, (b) ringlet of ringlets, and (c) ringlets interconnected by a switch

An SCI adapter card connects a node to a ringlet or a switch. On SCI-based clusters of workstations, made up of off-the-shelf components, the SCI NIC is usually attached to the I/O sub-system on the I/O bus. In this case, only a subset of the SCI standard will be provided. This is due to the fact that the cache coherent protocol recommended by the SCI standard will not be feasible on the I/O bus, because the latter doesn't provide enough means to have access to the CPU/cache data that travel across the system bus. The implementation of this protocol is sometimes atypical of custom architectures, e.g NUMA-Q [**?**]. An SCI NIC has the particularity of including an embedded processor, a DMA engine for memory-to-memory transfer without host CPU involvement, an on-board RAM, and circuits to access the I/O bus on the I/O sub-system. The on-board RAM is used to hold the Address Translation Table that provides a means to translate bus-addresses into SCI-addresses. The NIC also provides means to initiate DMA transfers from host memory.

Messages' transactions in an SCI-based cluster are entirely supported by the hardware. This required the application program to first create a shared memory segment in its address space. This is done via an API provided by the SCI NIC, e.g an ioctl() call. The created shared memory segment is then mapped in the virtual address space of remote nodes, and therefore accessible by all application programs running on them. This memory mapped segment doesn't correspond to real memory addresses (physical address). Rather, it's actually corresponding to I/O addresses belonging to the memory region assigned to the SCI NIC. Thus, accessing this address from a remote application program automatically triggers a message transaction across the SCI network. This translation process is depicted in figure 3.8.

Several research projects have adopted SCI as basis for their interconnection hardware. These include, for instance, the Scintilla project from the University of California at Santa Barbara [**?**], and the HPVM project from the University of Illinois [**?**]. However, in most of these projects, the cache coherent protocol provided by the standard has been omitted since the SCI adapter card is plugged into the I/O bus. Thus, those SCI adapter cards only provide uncached reads and writes transactions. Just as with Myrinet, the performance achieved by the SCI hardware and the basic primitives (read and write) provided by the SCI network can also be deemed as metric of performance for a SCI-based cluster. Whereas the raw bandwidth and latency provided by the SCI hardware is approximately of the same order of magnitude as most of today's high speed networks - i.e. a typical SCI adapter card connected on the I/O-bus has a bandwidth of 800 MB/s per link and achieved about 2.3 μs of latency; a SCI switch in turn provides a bandwidth of 1.6 Gbps on each port and requires about 600 ns of latency for taken a routing decision -, its application-level latency, on the other hand, depends especially on the network topology. For instance, on two direct connected nodes based on Ultra Sparc, the Scintilla team has measured a read latency of about 6.0 μs for a message of 4 bytes.

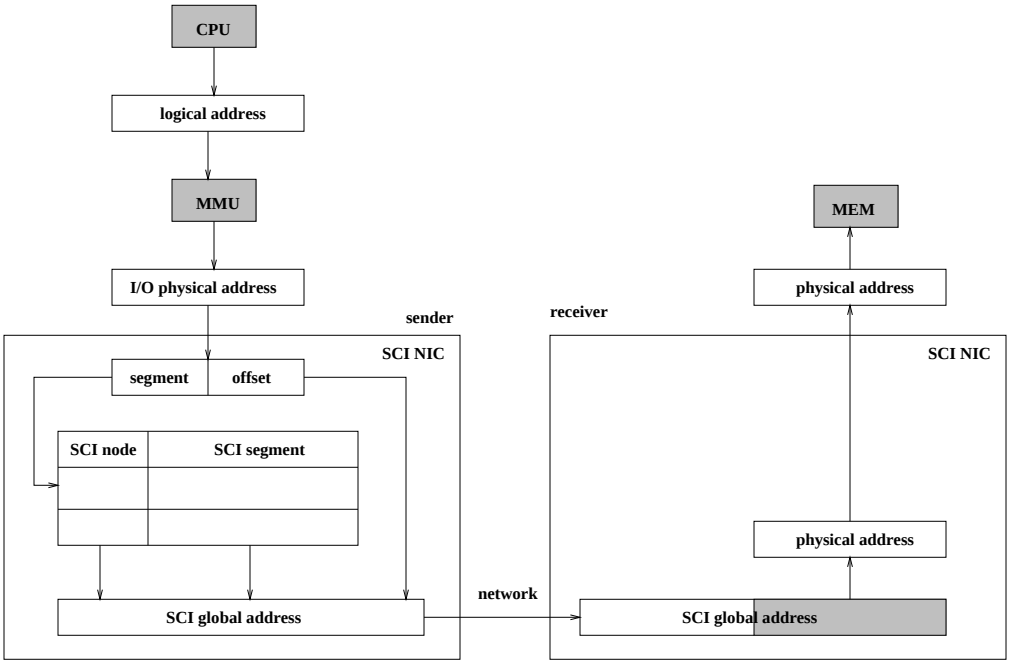

Figure 3.8: Address translation in SCI

However, as more nodes are added to the topology, the latency begins to increase because each packet has to be examined by each NIC before being consumed by the receiver on the same ringlet or forwarded to a switch or another ringlet. In addition, if a cache coherent protocol is provided by the software, the latency will be even worse due to the overhead introduced by the coherency protocol.

3.1.3 Motivation for the choice of Myrinet

As the number of interconnection networks increases, it becomes more complicated, for several reasons, to purchase a cluster of workstations based on the best available interconnection technology. First, the **performances** delivered by each network are as various as the architectural features with which they are manufactured. Second, it is obvious that the first point has a strong impact on the **affordability** of a given technology, making one more expensive than the other: the more complex a given technology is, the more it costs. Third, because some interconnection hardwares have specific design constraints, they have a restricted span regarding their **availability** on other platforms. Finally, even though available on several platforms, it might sometimes be necessary to adapt the software that controls the hardware to fit a particular design choice. Fine-tuning the software in this way is only possible if the vendor offers a certain degree of **code openness**.

Regarding the first point, i.e the performances, we believe that Myrinet belongs to the fastest interconnection technologies currently available on the market. This is well confirmed by looking at the performances achieved by Myrinet. It delivers a bandwidth of more than one gigabit per second while keeping the latency under the range of 5 μs for a carefully designed API such as BIP.

In addition to the performances achieved by Myrinet, its hardware components - switch, NIC and cables - are priced at a relative low cost in comparison to other high speed network products. In table 3.4 below, we show a comparison of the cost of purchasing an interconnection hardware based on Gigabit Ethernet against one based on Myrinet. These sources have been collected from the interconnect page of the SCL[9].

[9] http://www.scl.ameslab.gov/Projects/ClusterCookbook/interconnect.html

Gigabit Ethernet	Cost
Gigabit Ethernet Repeater FDR12 (from Packet Engine)	around $17900
Gigabit Ethernet NIC including connection cord	around $2000
Sum	$19900

Myrinet	Cost
8 x8 Myrinet switch	around $2400
Myrinet interconnection cost (NIC plus connection cord)	around $1700
Sum	$4100

Table 3.4 Cost comparisons between Gigabit Ethernet and Myrinet.

Besides the performance and cost advantages that favor Myrinet, another not less important point which speaks for it comes from the fact that Myrinet's protocols, softwares and APIs are ***open*** and ***free.*** For system designers (operating system, communication protocols), this offers the possibility of emulating a large number of new implementations. There is no doubt that this will encourage anyone interested in developing new communication abstractions, to lean in favor of Myrinet.

Furthermore, regarding the availability of Myrinet, it is entirely supported by all versions of the Linux operating system, which is the runtime substrate that runs on our platform.

3.2 Description of the testbed

Apart from the interconnection hardware that we just described in the previous section, our experimental platform consists of 8 SMP nodes. The SMPs and the interconnection hardware form our cluster of SMPs - CLUMPs. An SMP offers an alternative way of building a network of workstations - NOW - by taking advantages of both the SMP and the uniprocessor-based cluster architecture.

SMPs[10] are a kind of architecture which is very similar to that of a PC, the difference between these two architectures lying in the number of processors that the SMP combines and that share a unique subsystem of memory and I/O via a common bus. SMPs sometimes include an additional level of memory hierarchy - second level cache - intended to make them achieve a better performance than that often delivered by their counterpart single processor systems. The term "symmetric" which appears in the definition of the SMP is due to the fact that all processors inside an SMP have equal access to the memory and I/O subsets.

While SMPs often suffer from memory contention and poor scalability due to the shared bus architecture, they offer the advantages of balancing program workloads among their processors and of exploiting them in a much better manner than do single processor-based clusters. At least, this is true so far, as the amount of processors within the SMP is kept small. Advantages of single processor-based clusters are due to their capacity to scale (memory and I/O), adding more nodes to a cluster, and to their ability to benefit from any new kind of fast network technology that is arriving on the market - e.g. ATM, Gigabit Ethernet, Myrinet, SCI .

3.2.1 Hardware description of the tesbed

Our experimental platform is an aggregation of 8 SMP nodes interconnected by Myrinet for inter-cluster communications and by Fast Ethernet for all extra-cluster communications. The nodes are attached to the

[10] Symmetric Multiprocessors

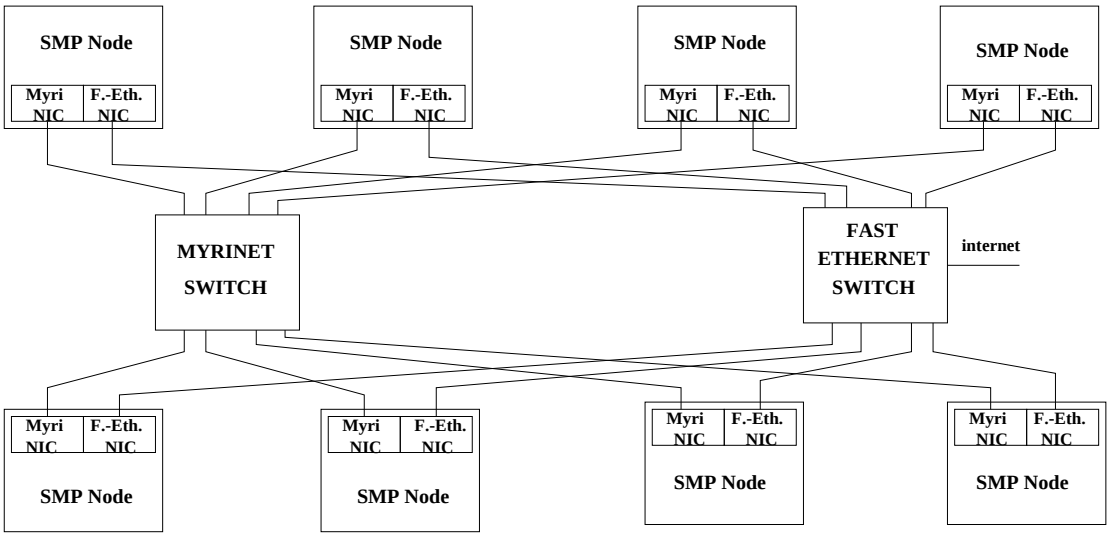

Figure 3.9: The Scalable Network of Workstations at the GMD-FIRST

network hardware either via a Myrinet NIC for Myrinet or via a Fast Ethernet NIC for Fast Ethernet (see figure 3.9).

Each SMP node is a dual Pentium II processor running at a frequency of 266 MHz. Each processor within an SMP node has a 16 KB on-chip first level cache for instruction and data respectively, and an external 512 KB unified second level cache. The processors of the SMP node share a 128 MB SDRAM memory which costs about 10ns for accessing it. Access to the I/O subsystem is made through a 32 bits PCI I/O bus that runs at 33 MHz. The organization of this SMP node is depicted in figure 3.10.

3.2.2 Software description of the testbed

Each node of our cluster is currently running a Redhat distribution of the LINUX Operating System (Linux 6.1). A version of MPI - MPICH 1.1.1 - is built on top of the TCP/IP Protocol stack. The Myrinet and Fast Ethernet drivers are also installed (see figure 3.11).

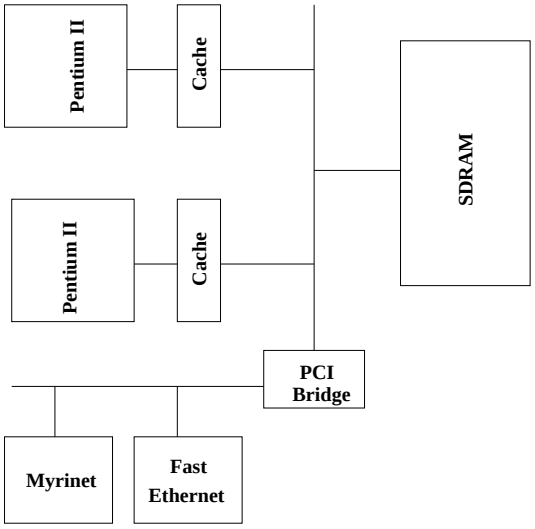

Figure 3.10: Pentium II-based SMP node

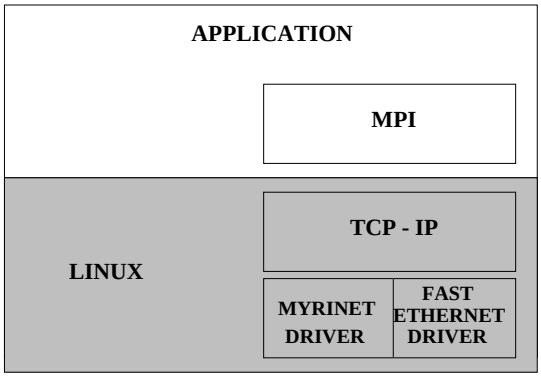

Figure 3.11: Software running on each node

Chapter 4

Abstraction and Design of CLIP

In the second chapter we studied several communication approaches employed in today´s high-performance environment. Then, in the third chapter, we looked into the architecture of various interconnection networks on top of which most communication abstractions presented in the second chapter are built. This chapter now aims at gathering together most of the issues of high-performance application that were highlighted in the previous chapters. For a better understanding of this chapter, one can view these issues as belonging to two distinct parts: the low-level part that addresses the implementation of CLIP and the high-level part that deals with the abstraction of CLIP.

At the low-level, the communication primitives provided by CLIP are designed with the aim of making use of as much of the architectural features of the interconnection hardware and the NIC as possible. Indeed, we have seen in chapter three that almost all high performance networks provide hardware features that aim at offloading a certain amount of the communication processing time from the host CPU. These hardware peculiarities will be addressed in section 4.2, that deals with the implementation issues of CLIP.

The high-level part, which addresses the abstraction of CLIP, is mainly underlain by two fundamentals that are very much related to each other. The first aspect is the abstraction's process of CLIP, whose guidelines follow the approach of object-orientation, and which ends up as an object-based implementation protocol. The second aspect is given by CLIP itself which is mainly based on EPOS. The tailor-made, application-driven nature of this runtime system, which relies heavily on a set of adaptable system components, lead us to make the major design decisions for CLIP in the early stages of the abstraction process. This further demanded that we provided careful specification of the CLIP's abstraction model and a convenient characterization of the set of operations available at each level of the abstraction.

In the next section we will extend the abstraction concepts introduced above (CLIP and CLIP within EPOS). We first begin by emphasizing the philosophy of EPOS. Having done this, we describe the design of its communication interface and then introduce the abstraction model of CLIP. The end of the section is devoted to an example of the realization of a communication interface in EPOS.

The last section of the chapter deals with the implementation decisions of CLIP that were made to address the issue of performance.

4.1 Abstraction of CLIP

It is important to note that, even at the first stages of the design, the aim of CLIP was never to provide a heavy protocol stack implementation. Rather, its principal purpose was targeted to meet the requirements

of a basic and adaptable user-level communication building block with as many limited functionalities as possible. This basic building block could then be "*extended*" by additional classes to provide more robust functionalities to higher-level application programs. The basic building block of CLIP as well as the classes that extend it are encapsulated into objects by means of well-defined key abstractions. Extensions to new functionalities are then supported via the object´s reusability or through the implementation of new classes based on existing ones. A powerful way of working this concept out is to rely on the inheritance and polymorphism techniques provided by the approach of object-orientation, whereas carrying it out in the design stage requires the use of an object-oriented programming language such as C++ [**?**].

Because CLIP's framework is based on EPOS, we naturally introduce it first before moving on throughout the abstraction layers of CLIP.

4.1.1 The EPOS Model

EPOS [**?**] is an Embedded Parallel Operating System that is better seen as an extension of the PURE family based operating system [**?**]. EPOS's main purpose is to bring object oriented operating system closer to high performance parallel applications. The gap between them originates from the complexity of assembling an operating system out of a complex collection of complex classes. EPOS aims to deliver, whenever possible automatically, a customized runtime support system for each application. In order to achieve this, EPOS introduces the concepts of *scenario-independent system abstractions*, *scenario-adapters* and *inflated interfaces*. An application designed and implemented following the guidelines behind these concepts can be submitted to a tool that will proceed syntactical and data flow analysis to extract an operating system blueprint. This blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a set of *selective realize keys* that will support the generation of the application-oriented operating system.

4.1.1.1 Overview of the Design of EPOS

In order to generate a tailor-made, application-oriented operating system, EPOS relies on a complex set of basic building blocks provided by PURE. To carry out the generation of this application-oriented operating system, EPOS defines three new key abstractions :

- *scenario-independent system abstractions* : they define a bulk of application-ready classes that, in their essence, must be implemented as independent from the execution scenario as possible;
- *scenario-adapters* : they provide a means for *adapting* an existing system abstraction to a given execution scenario;
- *inflated interfaces* : they provide to application programmers a set of automatic tools that will help them in specializing their applications' requirements. Inflated interfaces embrace most of the consensual definitions for a system abstraction and bring together most of its usual representations.

4.1.1.2 The Network Abstraction in EPOS

The network abstraction in EPOS conforms with the guidelines described above of tailoring a customized operating system. A communication channel between two interconnected nodes in an EPOS-based environment is built by giving to higher-level application programs, by the means of a network interface,

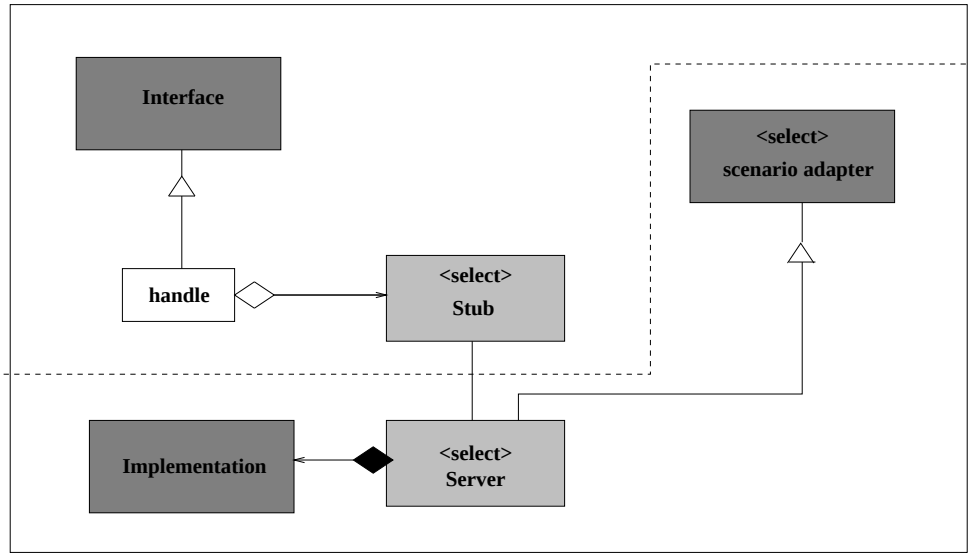

Figure 4.1: Network abstraction in EPOS

a set of low-level communication abstractions that are better viewed as system components or scenario-independent system abstractions. By binding an existing communication abstraction to a given scenario-adapter, it is possible to deliver a specialized communication channel that fulfils the requirement of a particular execution scenario implemented in the application program. This step is carried out by a "handle" that is able to understand the application's needs and invoke the right scenario-adapter whenever it's necessary. This is well understood by looking at figure 4.1 which shows the relationship between the classes of the network abstraction layer.

For a definition of our OMT-based notation (Object Modeling Technique) we refer the reader to Appendix A at the end of this work.

4.1.2 The Network Model of CLIP

4.1.2.1 Network Abstraction

CLIP's approach to delivering an adaptable, object-based implementation protocol as basis for communication, begins at the early stages of the design by taking a position on two fundamental design choices:

- the subset of services provided by CLIP (e.g reliable vs. non-reliable message delivery, sharable vs. non-sharable communication . . .), and
- the way CLIP maps its basic communication primitives onto the network and host capabilities.

Our approach regarding the first point is straightforward. We choose not to encumber our communication abstraction with costly services in favor of the benefits of a much less expensive communication protocol, thus giving us much more flexibility. Let us illustrate this with an example. We are very often confronted with a situation where a process, belonging to a parallel application that is distributed among several nodes, would like to entirely monopolize the communication network interface in order to increase its raw performance. In such a case, a heavy communication protocol that virtualizes the network interface by multiplexing it among several applications would only add overhead and nothing else to a similar

parallel application. Therefore, we are of the viewpoint that an application should establish first to which type of services it would like to adhere. Think of these services as a kind of scenario-adapter since they represent more a pattern of communication than they express something about the intrinsic nature of the communication.

Regarding the second point, we believe that the way by which a communication abstraction maps its basic communication primitives onto the capabilities of the network and the host should be of the least monolithic nature as possible. Interrupt-based applications, as well as those requiring DMA or polling, should all be able to use the communication ressources provided by the communication abstraction. Thus, we believe that the underlying communication protocol should give the application program the possibility of specializing its requirements, regardless of its nature.

When the key abstractions described above are fulfilled, the design of CLIP ends up with a class hierarchy that is well understood by looking at the figure 4.2.

4.1.2.2 Network Interface

As illustrated in the class hierarchy depicted in figure 4.2, all accesses to the network are mediated through the network interface class called "CLIP_Network_Interface". The design of this class and its association to the core of the network abstraction can be the subject of several design choices. We point out two of them. There are the granularity of the class and the way by which services of the communication layer are granted to users.

We opted for a fine grain granularity of the class's size, meaning that the interface should merely provide higher-level programs with nothing other but a "*mean of*" accessing the underlying communication primitives. This also justifies our reason for deciding to hide the implementation details of the communication abstraction from the users's perspective; decoupling in this way the network interface from the implementation of the network abstraction. A method invoked on an instance of the network interface is now "*delegated*" to (an instance of) the target class of the network abstraction layer. This means that the interface must have a knowledge of the object's type on which the method is invoked.

For instance, a way for the interface to know about the type of the class it has to instantiate, is to have at initialization time a variable set to a given type by default, e.g in our case the variable is called NETWORK_TYPE. Then, by means of this variable, we could let the interface maintain a reference to an instance of the targeted network class by using a named idiom constructor accessible via a public interface and by hiding possible default constructors from other programs behind a protected interface, i.e only friend and derived classes will be aware of it. Thus, invoking a method on an object of the network interface will result in delegating it to the method of the target network class whose reference is held in the network interface class (via the idiom named constructor). A simplified sample code of the CLIP_Network_Interface is shown in the figure 4.3.

4.1.3 Example of a Network Interface in EPOS

As an illustration of a network abstraction in EPOS, we show in this section how a common network interface that addresses the major keys described above can be designed.

Such a network interface should provide higher-level software with a means of specializing their communication requirements. Thus, the interface must be able to export a subset of functionally distinct communication abstractions. In this way, user programs can now select the type of communication abstraction

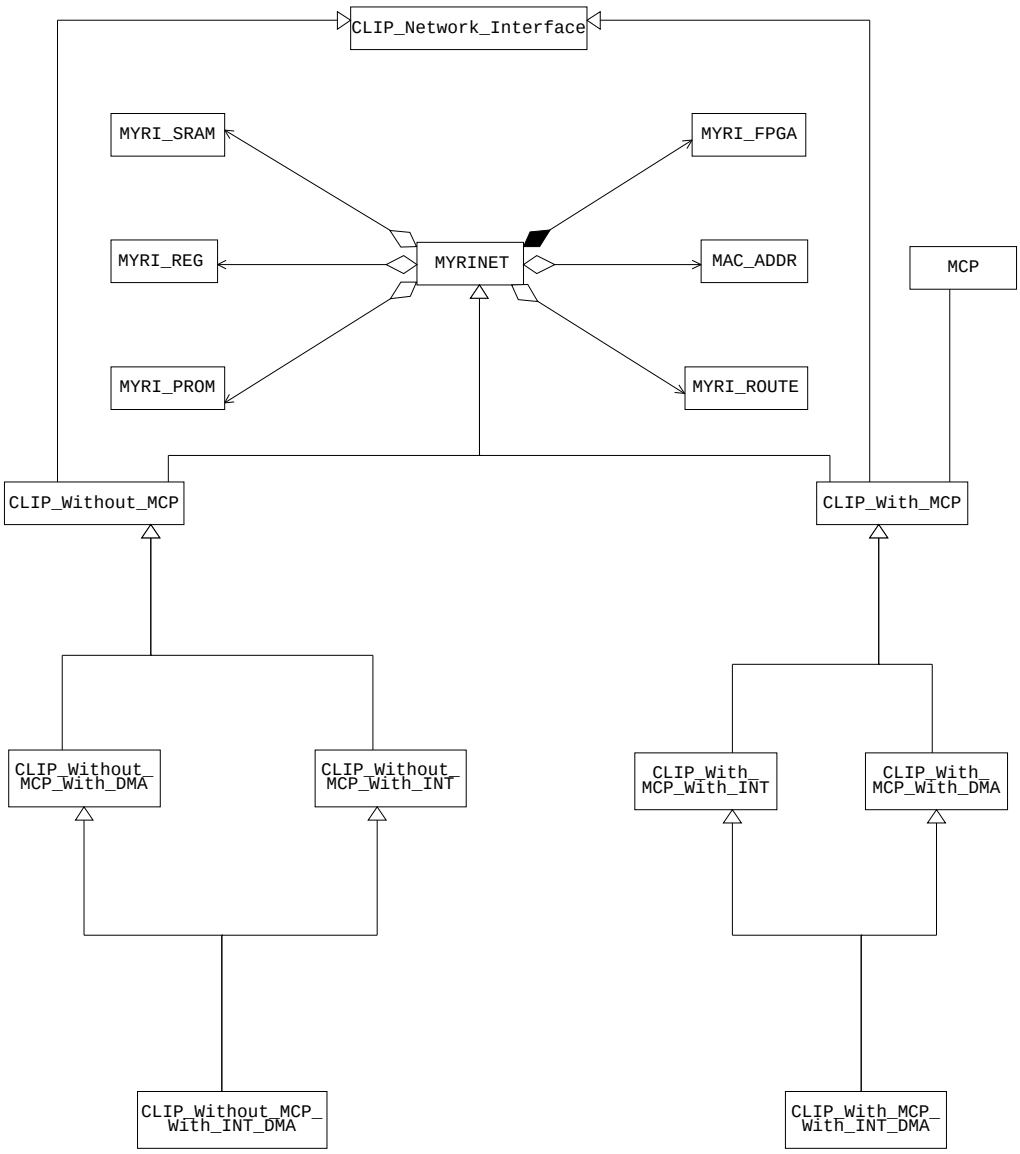

Figure 4.2: Class hiearchy of CLIP

```
Class CLIP_Network_Interface {

    public :
      static CLIP_Network_Interface* Create_Network_Instance( )
              { return new NETWORK_TYPE; }

    protected :
       Network() {}
      ~Network() {}

    public :

        send(void* buffer, int dest, int len)
        {
              return ((static_cast<NETWORK_TYPE *>(this))->send(buffer,dest,len));
        }

        recv(void* buffer, int maxlen, int* len)
        {
              return ((static_cast<NETWORK_TYPE *>(this))->recv(buffer,maxlen,len));
        }

        reset(unsigned long phys, unsigned long mmap, unsigned long recv)
        {
              return ((static_cast<NETWORK_TYPE *>(this))->reset(phys,mmap,recv));
        }
};
```

Figure 4.3: Sample code of the network interface in CLIP

that they need, and eventually code a specialized behavior in their program, which will then be wrapped to match this specific requirement.

The subset of communication abstractions available via this network interface can be of various sizes. As an example, we show in figure 4.4 how it could be designed to address the communication abstraction realized by the implementations of Active Message [**?**], BIP [**?**], PM [**?**] and CLIP together.

As we showed in figure 4.1 of this section, each call to a communication abstraction available via the EPOS network interface will be bound at runtime to its corresponding implementation.

4.2 Design of CLIP

This section addresses the implementation decisions made during the design of CLIP.

CLIP is designed and implemented on a Myrinet-based cluster of workstations running a Redhat distribution of the LINUX operating system (Linux 6.0, kernel version 2.2.0). While referring to the approach of traditional operating systems discussed in chapter two of granting network access to application programs, our goal is to provide a similar network service implemented entirely at the user level.

To achieve this goal, CLIP basically relies on the underlying hardware capabilities offered by the interconnection network and the NIC. The intervention of the kernel software that runs on the host is reduced at startup time in initializing and mapping the NIC into the user address space. It also establishes convenient addressable memory regions for hosting send and receive data. The network hardware is responsible for transferring messages from one node to another. It offers a raw bandwidth of 1.28 Gbps on each link. In addition, the network hardware guarantees highly reliable communication in that its delivers a very low error rate (less than 10^{-5}) and implements flow control on each link. However, the most significant capability

```
Class EPOS_Network  {

public:
   EPOS_Network();
 ~EPOS_Network();

public:
    reset();

    /* Active Message */
   int AM_RequestM(ep_t request_endpoint,     ,
                           int reply_endpoint, handler_t handler, ...);
  int AM_ReplyM(void* token, handler_t handler, ...);

    /* BIP */
   int bip_send( int dest, int* buf, int length);
   int bip_recv( int* buf, int maxlength);

    /* CLIP */
    int clip_send( void* buf, int dest, int taille);
    int clip_recv( void* buf, int maxlen, int* len);

     /* PM */
    int pmSend(pmCtx pmc, int dest);
    int pmGetSendBuf(pmCtx pmc, caddr_t buf, size_t length);
    int pmReceive(pmCtx pmc, caddr_t buf, size_t length);
    int pmPutReceiveBuf(pmCtx pmc);
}
```

Figure 4.4: Sample code of a network interface in EPOS

upon which CLIP relies still remains the NIC component. Its design, which has been already presented in chapter two, includes an embedded processor that is responsible for executing the core of the CLIP implementation protocol (the MCP program). CLIP's LANai code and data are hosted in the on-board SRAM memory and accessed by the LANai processor which fetches them sequentially at a clock speed of 33 MHz. The on-board memory is also used as a stage buffer and holds internal and shared data structures that are private to the MCP or common to both the MCP and the host. By the means of the three DMA engines introduced previously, the NIC can concurrently access a DMAable host region and transfer data to or receive data from the network. On top of this, the NIC also includes a checksum hardware support that automatically generates and verifies data CRC as they are transmitted to or received from the network.

How CLIP combines all of these hardware features with the aim of enhancing the communication performance, is shown in the next two sections.

4.2.1 Overview

4.2.1.1 The Software components

The software that makes up CLIP, and thus implements the communication abstraction introduced previously, is organized basically around a collection of functionally distinct software components that interact with each other in a reliable manner . We briefly describe them here.

A kernel loadable module called CLIP_modpa is responsible for managing a send and receive buffer in the system memory. The set of functionalities provided by this module includes, among others, the ability to dynamically create a communication segment in kernel space, to memory map this latter segment in the virtual address space of a user process, and to manage read/write accesses to this segment.

API	description
CLIPsend_wmd	send with MCP and DMA
CLIPrecv_wmd	receive with MCP and DMA
open_lanai	open the LANai adapter card
load_mcp	load the MCP of CLIP
clipGetBuffer	create a system buffer
clipMmap	memory map the system buffer in user space

Table 4.1: A subset of the API of CLIP

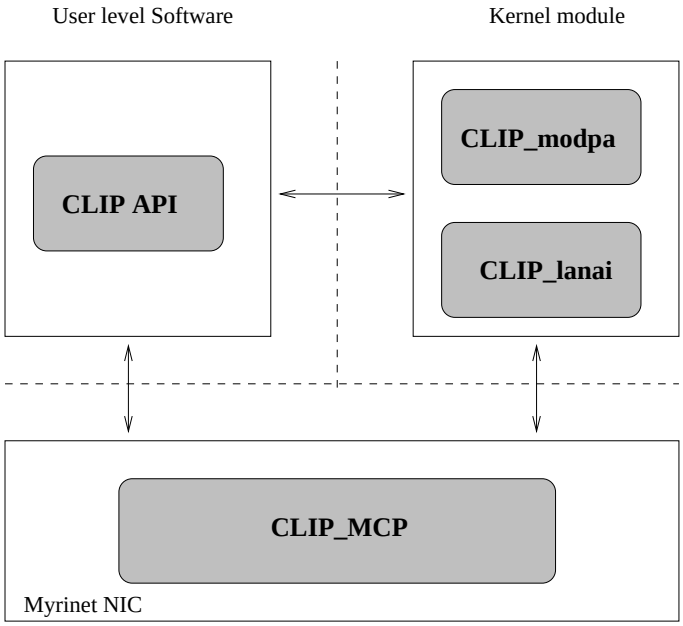

Figure 4.5: Software components interaction in CLIP

The CLIP_lanai software component implements the driver part of the Myrinet NIC and is available as a kernel module. The set of functionalities provided by CLIP_lanai includes the initialization of the NIC, the memory mapping of the whole board (memory, special registers, control registers) into the user address space and a way to overwrite the default caching policy assigned to the memory region of the NIC.

The CLIP_MCP is our Myrinet firmware that implements the most significant part of our communication abstraction. The CLIP_MCP runs on the NIC and is assisted by the user-level software.

The user-level part of CLIP consists of an API that provides a means of sending and receiving messages over Myrinet (CLIP_Network_Interface), of creating a communication segment in system memory (clipGetBuffer), of opening the NIC (open_lanai) and of loading an MCP program into the NIC memory (load_mcp). Table 4.1 gives a brief overview of some of these functions.

The interaction among the various software components described above is again illustrated in figure 4.5.

4.2.1.2 Sending and receiving messages

Prior to a send or a receive operation, a user program must have initialized a *communication endpoint*. A communication endpoint in CLIP is an aggregation of information that describes the actual communication

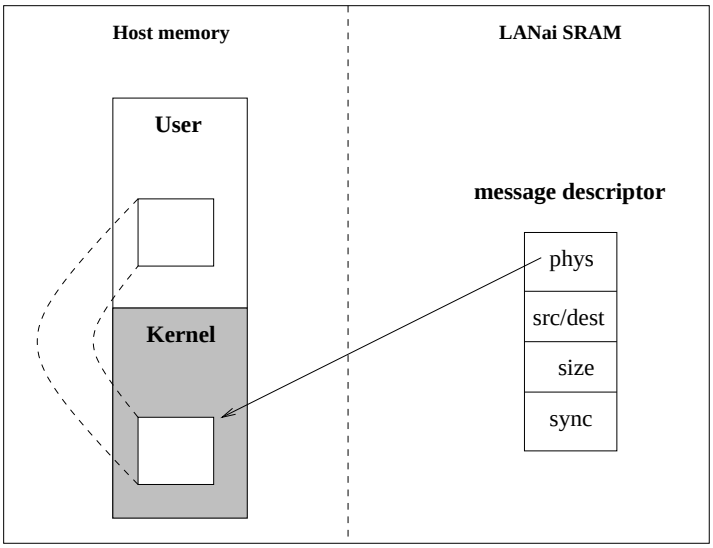

Figure 4.6: Structure of a communication endpoint in CLIP

address space of a given user process. Such an endpoint is split into two fields: a host-side and a LANai-side.

The host-side communication endpoint consists of :

- a system communication buffer that holds a copy of the user data and that resides in kernel memory;
- the memory map address of the system communication buffer in the user address space.

The LANai-side communication endpoint (in the NIC memory) consists of a message descriptor that holds, amongst other things, the following information:

- the location of the system communication buffer;
- the size of the message that is to be sent or has been received;
- the destination or the source of the message;
- a synchronization flag .

Figure 4.6 illustrates the structure of a communication endpoint in CLIP.

After a host-side communication endpoint has been initialized, i.e a system communication buffer has been allocated and mapped in the virtual address space of the user process, a user-level "send" proceeds in two different ways. For short messages, the user data residing in the user communication buffer are directly copied into the NIC memory via programmed I/O. A LANai-side communication endpoint (send message descriptor) is then made available on the NIC memory. After the MCP has been notified of this event, it then immediately injects the data into the network. In the case of long messages, the data must first be copied in small chunks into the pre-allocated system communication buffer. This operation is carried out by a wrapped version of the memcpy() function which takes as its destination address the memory map value of the system communication buffer computed during the creation of the host-side communication endpoint.

At the same time, the "send" operation fills a LANai-side communication endpoint (send message descriptor) in the NIC memory and notifies the MCP of the presence of a send request via the synchronization flag of the LANai-side endpoint. The MCP is then expected to peak and forward the message, i.e it accesses the message in the system communication buffer and pushes it into the network. This action is taken in parallel with the copying of the message into the system buffer carried out by the user-level send. Thus, the data path from the user space to the network is pipelined and involves both the user-level software and the MCP, which then have to be synchronized accordingly. A send operation is completed only when the last byte of the message has been successfully injected into the network. This issue will be further addressed in the next section.

Similar to the send process, a receive process must have also initialized a host-side communication endpoint before invoking a receive operation. In addition, the physical address of the system buffer in the LANai-side communication endpoint must have been set up previously. Hence, an incoming message can be directly copied into the application address space, since a system communication buffer must have been assigned and memory mapped in the user address space during the creation of the host-side communication endpoint. This operation is improved by pipelining the data path of the message from the network to the system buffer. The completion of the receive operation is indicated by the presence of the last byte of the incoming message in the system buffer.

Our implementation of the send/receive primitives complies with a synchronous semantic, i.e a posted receive operation must precede a send request.

4.2.2 Surveys of the Implementation

4.2.2.1 The direct network access and the *copy semantic*

We already discussed in chapter two how traditional operating systems such as UNIX handle communication at the kernel level (e.g by implementing costly system calls, enhancing memory copies, ...), and thus, how they contribute to the degradation of the performance that they deliver to higher-level applications. In fact, it isn't worth transferring data over a high-speed network at gigabit speed, if it can only trickle into the user address space at a much slower rate due to the communication protocol. The fact that the memory sub-system even now becomes the bottleneck of the processor-memory/network-interface exacerbates this situation further. For instance, in our testbed system, even if it is theoretically possible to transfer the data over the Myrinet network at a raw speed of 160 MB/s, the available CPU/memory bandwidth for copy which is of the same order of magnitude as that of the Myrinet network, is only around 140 MB/s, the read bandwidth about 340 MB/s and the write bandwidth 143 MB/s. Thus, if for any transfer we need to access a word in memory n times, the best throughput we can achieve will be at least n times worse than the available bandwidth at which the data enters the memory. This can rapidly move down behind the expected peak bandwitdh of the network.

To alleviate the effect of the copy semantic on the achievable end-to-end application throughput, we have chosen to directly access the NIC from the user address space. This is mainly done in two or three different ways. They are the PIO, the DMA or the DMA with special purpose hardware or software that runs on the NIC, the host CPU or a combination of the two. First, we will briefly describe some hardware limitations and requirements of these methods before introducing the details of our implementation.

A common hardware limitation to these methods is the I/O subsystem on which the NIC is plugged and which usually interfaces the processor-memory subsystem by the means of a bridge or a bus adapter. The I/O subsystem sometimes places a severe limitation on the peak performance that can be delivered to applications. In our testbed platform for instance, the PCI I/O bus can deliver a bandwidth of about 128 MB/s when using DMA and 44,4 MB/s when the PIO is used instead. Thus, even if the network can deliver

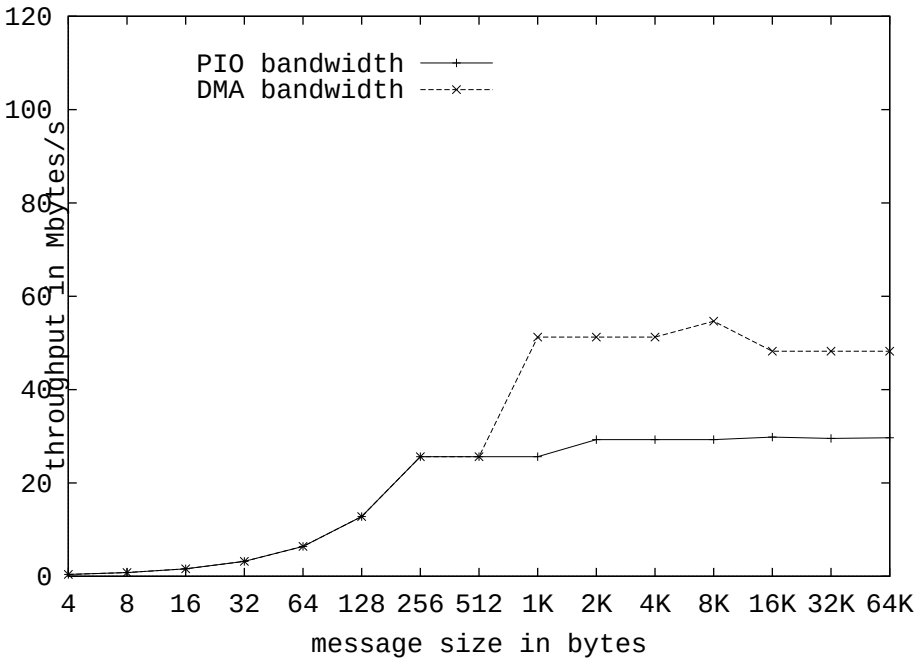

Figure 4.7: Comparaison of PIO and DMA bandwidth

data at 160 MB/s, they can only trickle into the host memory at a peak throughput of 128 MB/s. In fact, for the PCI-bus, this peak performance can only be achieved if the NIC supports the burst transfer mode that enables higher throughput over the I/O-bus. In this case only, the effective end-to-end application throughput is limited by the HOST-to-DMA or DMA-to-HOST engine of the attached network device. Fortunately, the host DMA engine of our Myrinet-based network is capable of achieving slightly more than that ($\approx$130 MB/s[1]).

The Programmed I/O (PIO) doesn't require any special hardware or software features. The host CPU usually sits in a tight loop where it is involved in the transfer of the data from the application space to the NIC memory or vice-versa. The unit of transfer is often a few bytes or sometimes an entire cache line. Because the processor must repeat the same operation each time that it has to transfer a single word, the per-word cost for big messages will, in this case, penalize the application performance. In contrast, the DMA is optimized for big messages. A DMA transfer can be done only from or into a contiguous memory segment of the host physical memory (real RAM). This has two implications. Firstly, before a DMA transfer can take place, the user´s virtual address must be translated into a physical address. Secondly, because most of the operating systems of today run in a multi-task environment that is based on paging, a selected cluster of pages in memory must first be locked before being accessed by the DMA engine. It is not unusual that, for the transfer of small messages, the initialization of the DMA transfer raises the communication cost; this is because the DMA has mostly a high per-transfer cost compared to the per-word cost of the PIO. We analysed this behavior by measuring the throughput rate incurred by sending various sizes of messages over the Myrinet network. The same communication abstraction (wihtout any enhancement) has been used for the DMA and the PIO transfer. The figure 4.7 shows the results we obtained.

The third method of accessing the user space, involving the DMA, implements the virtual to physical address translation differently. It has many variations depending on the communication architecture. Some designs have custom hardware on which the NIC is integrated, and thus shares the TLB of the host CPU, easing, in this way, the virtual to physical address translation. Other designs provide no special hardware

[1] Result obtained from the Myricom PCI DMA Performance page

support but shift the address translation process into the NIC by implementing a kind of TLB that collaborates with the MMU of the host CPU [**?**]. In both cases, the user program doesn't need to translate its virtual address anymore as this is now done automatically in the NIC. Although these implementations really seem to pay off, they do have, however, the disadvantage of involving extra efforts, and thus of increasing the cost of a system involving custom hardware support, or of adding to the overhead due to the software implemented in the NIC.

We decided to implement a PIO-based transfer for small messages and switch to a DMA-based protocol for big messages. The DMA-based protocol relies on the CLIP_modpa module. This module is responsible for allocating a contiguous region of memory in kernel space and for pinning/unpinning it consequentially in the kernel. The user interface to this module is currently the call of the clipGetBuffer() function. It takes as arguments the size of the system buffer that is to be allocated, specified in a power of two, and an indicator of the type of the buffer, i.e send or receive buffer. The last argument of the clipGetBuffer() function is essentially needed to differentiate between write and read accesses to system buffers. A side effect of this call is the return of the physical address and the file descriptor of the assigned system buffer. The function clipCloseBuffer() that takes a file descriptor as unique argument, frees the kernel ressources (the pages are naturally first unpinned) allocated previously by clipGetBuffer(). When it happens that an application will frequently reuse its communication buffer, we allocate such a kernel buffer only once.

However, if we need to move a message from user space to system buffer before sending it into the network, we will introduce a copy stage. Although this goes counter to our argumentation of eliminating memory copies made above, we will see in the following section how we deal with the performance penalty incurred in adding this extra copy stage.

4.2.2.2 Pipelining issues

The direct network access strategy discussed above has introduced a copy stage that should theoretically contribute to increase the end-to-end latency experienced by an application. Moreover, from the viewpoint of a programmer, the communication path of a message from its local memory to the memory of a remote application, can be viewed as a series of store-and-forward stages. Each stage that is visited by a user message en route towards a remote node, participates in aggravating the overall end-to-end communication delay.

Let us illustrate this by examining our case of communication in Myrinet. A Myrinet switch, as described in the previous chapter, implements a wormhole switching strategy. Thus, this eliminates the need for storing a complete message in a storage device before deciding to route it to the appropriate output port, as is the case for the store-and-forward switchting strategy. However, looking at the network from "*outside*", the communication path of a message can best be modelled as a sequence of several store-and-forward stations. Indeed, before a message can be sent, it must first be copied into the memory of the NIC (store) via its on-board host-DMA engine. Then, from there, the embedded LANai processor instructs its send-DMA engine to inject the message into the network (forward). At the other end, the message is again copied into the NIC memory of the target node (store) via the receive-DMA. It is definitively moved into the address space of the target process (store) via the host-DMA.

Clearly, if a message must travel through all these stages, the resulting performance would be poor. But since these stages are apparently unavoidable, what we must do is try hiding the latency of communication that results from it. Thus, increasing the degree of parallelism of the network protocol, by overlapping some of the stages that do not depend on each other, can decrease the overall latency. The aim becomes then to keep all stages active at the same time. This is best achieved by pipelining the various steps of the data path involved in the communication. Such a data path in CLIP includes the following stages (see figure 4.8):

Figure 4.8: Steps involved in the transfer of data in CLIP

- *copy stage*: a message is composed in the local memory of the user process before being moved later to a system buffer;
- *host DMA stage*: the message is moved from the kernel memory of the host to the memory of the NIC;
- *send DMA stage*: the message is sent into the network;
- *receive stage*: the message is consumed off of the network and stored in the local memory of the NIC;
- *host DMA stage:* the message is moved from the memory of the NIC to the memory of the target process;
- *copy stage*: the message is transported into the memory of the process at the destination node.

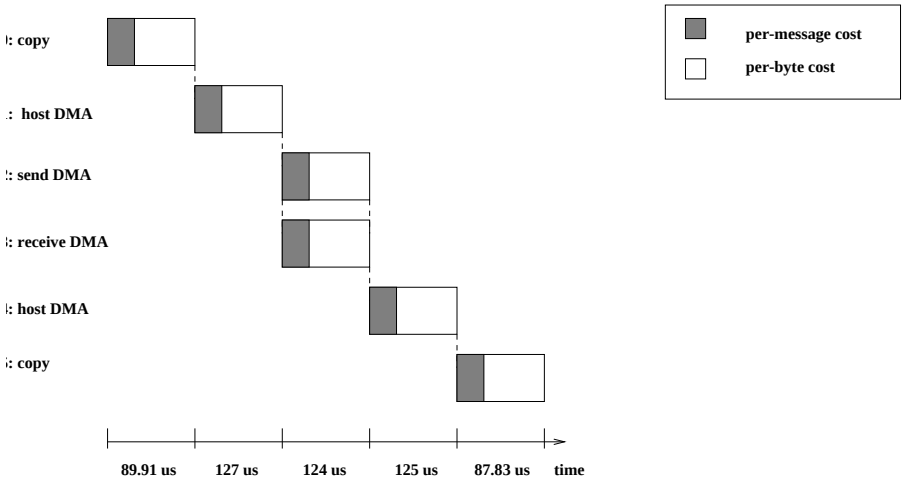

Figure 4.9: Measured time of a 16KB message sent into the network

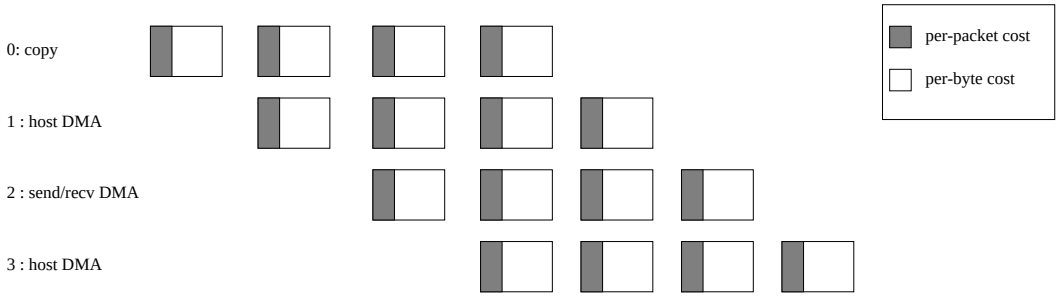

Figure 4.10: Pipeline model of CLIP

The design of our pipeline model begins by analyzing the time spent in each stage of the communication data path described above by messages of various sizes sent through the pipeline into the network. We show in figure 4.9 the timing of such a message containing 16KB of data. The time spent by the message at each stage of the pipeline includes the time involving the host CPU or overhead (per-message cost) and the effective transfer time (per-byte cost). The former value can be different from one stage of the pipeline to the other. We will refer to it later in this section.

From the figure 4.9, we can note that stages 2 and 3 of the communication data path occur in parallel. This is explained by the wormhole switching strategy implemented by Myrinet. A design choice was then made to merge these two stages together into a single pipeline stage. We thus obtained the pipeline model for the figure 4.10.

Before a message is now sent into the network, it is split into several packets of fixed size. The packets progress through the stages of the pipeline according to the following rules :

- **Rule 1** : As soon as a packet n exits from a given pipeline stage i it can enter the next stage i+1;
- **Rule 2** : A packet n enters stage i of the pipeline as soon as packet n-1 exits from stage i of the pipeline.

With the above pipelined model, two requirements suffice in order to obtain the best performance. The number of packets must be at least equal to the depth of the pipeline, and each packet must have the optimal

length to minimize the total transmission time of the entire message. Since the first requirement depends on the second (with the optimal packet length one easily computes the number of packets), we have chosen to devote the remainder of this part to the development of an analytical model for the realization of the second requirement. Related works on the computation of the optimal packet length can also be read in [**?**] and [**?**].

When the transfer is carried out sequentially, i.e with no splitting of the message into several packets, a message in CLIP experiences an overall latency equal to the sum of the latency spent in each pipeline stage of the communication data path. The contribution of each pipeline stage to the overall latency of a message can be expressed using a linear cost model [**?**] in which, a message that travels across a given pipeline stage i, experiences a delay T_i formulated as follows:

$$T_i = \beta_i + P\tau_i \text{ (1)}$$

where β_i is a fixed per-message cost of stage i, P the size of the message, and τ_i the inverse bandwidth of stage i. Thus, for the pipeline model of figure 4.10, the latency of a packet n in a pipeline stage m can be formulated as follows:

$$T_m^{(n)} = \beta_m + P\tau_m \text{ (2)}$$

For our model, we made the following simplifications:

- for each packet that travels across stage 0 or stage 4, $\beta_0 \cong \beta_4$ and $\tau_0 \cong \tau_4$ since these stages involve the same ressource (copy);
- for each packet that travels across stage 1 or stage 3, $\beta_1 \cong \beta_3$ and $\tau_1 \cong \tau_3$ since these stages involve the same hardware ressource (host DMA);
- from the given hardware specification, we know the bandwidth of copy ($\approx$ 140MB/s), the peak bandwidth of the host DMA of the NIC ($\approx$ 130MB/s) and the bandwidth of the Myrinet network (160MB/s). Thus, it follows that the inequality (3) holds:

$$\tau_1 > \tau_0 > \tau_2 \text{ (3)}$$

- From the inequality (3) and from the rules described above, we can deduce an upper bound of the time experienced by a packet travelling through the stage 1 of the pipeline model of figure 4.10 and also that of the last packet at stage 2. We express this in the inequality (4).

$$\beta_0 + \beta_1 + \beta_2 + P\tau_1 > \beta_2 + P\tau_2 \text{ and } \beta_0 + \beta_1 + \beta_2 + P\tau_1 > \beta_0 + P\tau_0 \text{ (4a,4b)}$$

$$\beta_1 + \beta_2 + P\tau_1 > \beta_2 + P\tau_2 \text{ (4c)}$$

Let now T be the overall latency of an entire message that is transmitted in a pipeline fashion across the stages of the pipeline model shown in figure 4.10, further, let L and P be the sizes of the entire message and of each packet respectively (we assume that the fraction $\frac{L}{P}$ is rounded to an integer value) then we can express T as follows:

$$T = \sum_{k=0}^{1} T_k^{(1)} + \sum_{i=1}^{\lceil \frac{L}{P} \rceil} T_1^{(i)} + \sum_{j=3}^{4} T_j^{(\lceil \frac{L}{P} \rceil)} \text{ (5)}$$

From the inequalities (3) and (4), and from the rules 1 and 2 above, the terms $\sum_{i=1}^{\lceil \frac{L}{P} \rceil} T_1^{(i)}$ and $\sum_{j=3}^{4} T_j^{(\lceil \frac{L}{P} \rceil)}$ of (5) can be approximated as follows:

stage i	β_i in μs	τ_i in $\frac{\mu s}{KB}$
0	3.0	5.9
1	0.6	7.6
2	1.3	7.5
3	0.5	7.6

Table 4.2: Computed values for the per-packet cost β and inverse bandwidth τ of the CLIP's pipeline model

$$\sum_{i=1}^{\lceil \frac{L}{P} \rceil} T_1^{(i)} \leq \lceil \tfrac{L}{P} \rceil (\beta_0 + \beta_1 + \beta_2 + P\tau_1) \text{ (6)}$$

$$\sum_{j=3}^{4} T_j^{(\lceil \frac{L}{P} \rceil)} = \Sigma_{k=0}^{1} T_k^{(1)} \leq \beta_0 + \beta_1 + P\tau_1 + \beta_0 + P\tau_0 \text{ (7)}$$

Taken the right part of the inequalities (6) and (7) as upper bound of the two terms of equation (5) mentioned above, a parameterized expression of T in L and P can be given by (8) below.

$$T(L,P) \cong 2(2\beta_0 + \beta_1) + 2(\tau_0 + \tau_1)P + \lceil \tfrac{L}{P} \rceil (\beta_0 + \beta_1 + \beta_2 + P\tau_1) \text{ (8)}$$

The optimal packet size is then computed by solving equation (9), and its solution is given by (10).

$$\frac{dT(L,P)}{dP} = 0 \text{ (9)}$$

$$P_{optimal} = \sqrt{\frac{L(\beta_0+\beta_1+\beta_2)}{2(\tau_0+\tau_1)}} \text{ (10)}$$

By applying a linear regression approximation (see appendix B) for each stage of the pipeline on the set of predictor variables that we measured by sending a non-pipelined message of various size into the network, we obtained approximatively the values for β and τ given in table 4.1.

As an example, for a message of 64 KB, we found that, analytically, the optimal packet length that best minimizes the latency of the message is around ≈ 3490 bytes. In practice we got 4096 bytes, which gives us an error rate of less than 15%. Further, we also noted that sizes of message belonging to various ranges can lead to different packet lengths. Thus, by determining the range of message sizes for which the length of packets given by equation (10) yields the best latency given by (8), we obtained an adaptive pipeline implementation that switchs automatically to the appropriated length of packet, which minimizes the latency of the message being proceeded.

4.2.2.3 Issues regarding the crossing of the PCI

When we discussed the issues of the direct network access, we saw that the I/O-bus also contributed to the degradation of the end-to-end application throughput provided by the underlying communication abstraction. We saw that the main reason for this was that, on most systems, the I/O-bus is still of an order of magnitude lower than that of the bandwidth of the attached network hardware. The approach which we opted for then, was to rely on the burst transfer mode in order to exploit the full bandwidth of the I/O-bus. This has proven to be worthwhile. However, there are still some situations where this is not always the case. There are cases where small messages have to be transferred over the bus. Indeed, for posting a send request or waiting for its completion, the host CPU must access the memory on the NIC, i.e it must cross the PCI-bridge. Just as for the send, for the notification of a receive operation, the host CPU should also continuously poll a flag in the NIC memory until it changes its state. For this kind of synchronization messages involving only a small amount of data, the burst transfer mode is not a panacea.

In order to alleviate the drawback generated by each crossing of the PCI-bridge, we decided not to poll the NIC memory on a receive. Under these circumstances, a user process in our implementation needs only to

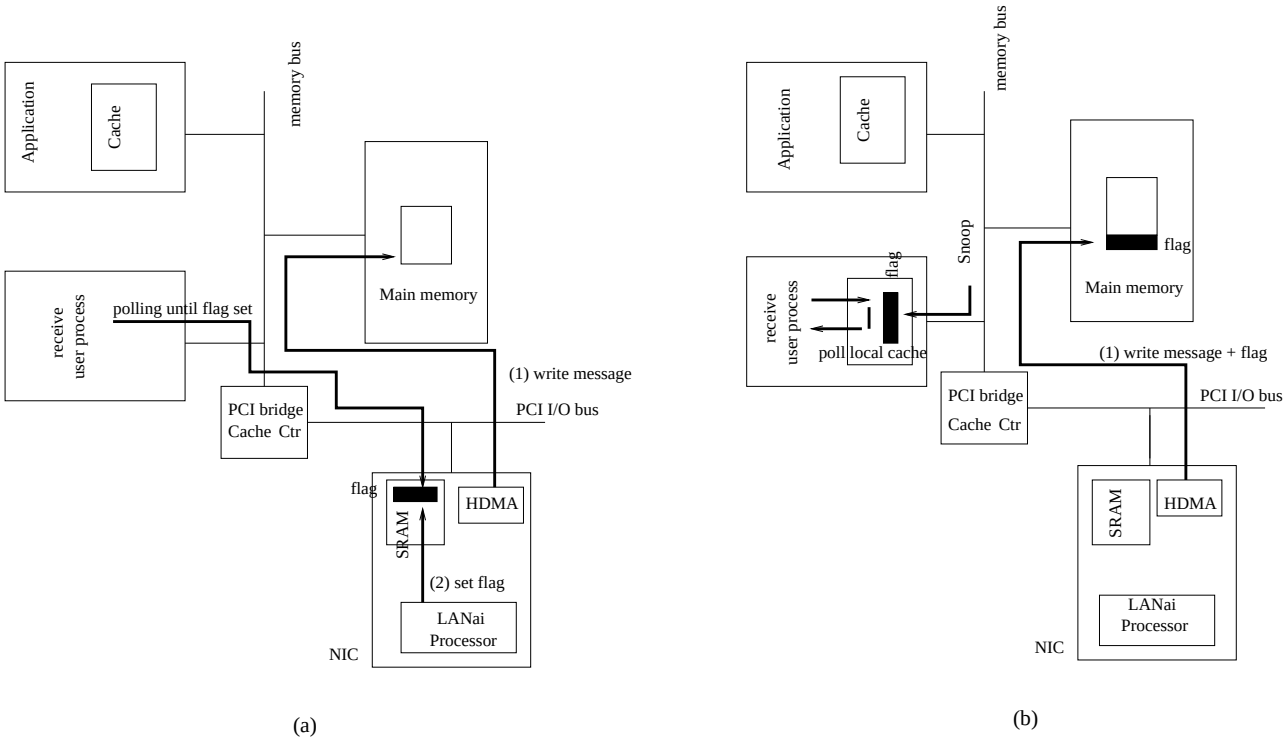

Figure 4.11: Notification of a receive completion in CLIP: (a) without snooping, (b) with snooping

poll its local cache for the completion of a receive event. This approach is essentially motivated by the fact that our SMP node supports a coherent cache protocol implemented by the snoopy logic on the I/O-bus. Each time a bus transaction occurs, e.g a write initiated by any bus master (for example our Myrinet NIC) in any memory location of the host, the bus-side controller of the cache must capture the target address of the memory on the bus and then perform a tag check on it. In the event of a "snoop miss" nothing occurs. However, if a "snoop hit" is reported instead, i.e the data is cached in the cache and in the main memory and its state has been modified, the cache controller must then have to update its state according to the coherency protocol that it implements. In our case, the entry of the related line in the cache will be invalidated, forcing the host CPU to process a cache line fill the next time it accesses the data of this cache line.

The result of these two different approaches is showed in figure 4.11 below. In (a) we represent the normal case where the host CPU must poll the NIC memory for the notification of a receive event completion. It consumes more cycles due to the arbitration for the bus and experiences an extra latency in accessing the memory in the NIC. It may also contribute to consuming the bandwidth of accessing the memory on the NIC. In (b) we show how we improved the notification of the receive completion presented in (a). Just before the last packet of the received message enters the last stage of the pipeline (host DMA stage), we append at its tail a structure that contains useful information such as a flag for the notification of the receive completion.

As soon as the host DMA of the NIC completes its task, i.e the data are now resident in the main memory of the host and the state of the flag has changed, the host CPU will be aware of it via the snoopy logic. It will then perform a cache line fill of the corresponding invalidated cache line, and exit from its tight loop.

We haven’t applied this technique on a “send” because we would have been obliged to re-initialize a new DMA transfer, the first DMA transfer being carried out in the opposite direction (host-to-NIC) requiring that the synchronization flag be written from the NIC to the memory of the host.

4.2.2.4 Data alignment issues

Data misalignments can lead to annoying drawbacks if we aren´t careful enough. The Pentium II processor that we're using places several constraints on the addresses of data and code in host memory. For instance, a misalign data access in the data cache unit (DCU) costs up to 12 clock cycles. In a tight loop, this could really raise the overall communication cost. Thus, in order to improve the cache performance, we have to make sure that data structures and arrays are all aligned on a cache line boundary, i.e 32 bytes. This is realized by padding structures and arrays to make their size an integer multiple of a cache line size. We also make sure that dynamic allocated buffers are aligned on a cache line boundary by wrapping the allocator function with a macro that carries out this work.

4.2.2.5 Caching issue

The issue of caching in CLIP intervenes when short messages have to be transferred by means of PIO into the network.

In the default case, the region of system memory that hosts the Myrinet NIC implements a Write-back caching policy. In this regime, a memory-to-memory transfer initiated by the host CPU is performed at the granularity of a word. The Pentium II processor offers a way to improve this type of transfer. By remapping the memory type of the Myrinet NIC (in system memory) into a "write combining" caching policy, we can force the host CPU to defer all memory-to -memory transfers into a "write combining buffer". Once this buffer becomes full, its content is then transferred in a single burst transaction to the target memory. However, the Pentium II specification doesn't guarantee that the buffer will be flushed if it isn't full. Thus, in order to have this certitude, we must explicitly flush the write combining buffer after each memory-to-memory transfer. We carry it out by introducing the following assembly code :

asm("cpuid" : "=d" (dummy) : : "%eax", "%edx", "%ecx", "%ebx");

Remapping the memory type of the Myrinet NIC into a write combining caching policy is made in CLIP at module loading time. This is done by the CLIP_lanai module via its clip_mtrr interface. This function accesses the memory type range registers of the Pentium II processor and reconfigures them accordingly to figure the write combining caching policy. It is then unset at module unloading time.

This enhancement has already been used in many cases. For the transfer of data between the host memory and a video frame buffer on a video card for example, the write combining buffer is often used to improve the achievable memory bandwidth. In BIP [**?**] for instance, the write combining buffer is also employed to enable high data throughput for short messages.

4.2.2.6 Miscellaneous

Since we spend a lot of our time copying data, i.e user memory to system memory for long messages and user memory to memory in the NIC for short messages, we decided to look at ways that might optimize the most common of cases.

Fortunately, we found that the Pentium II processor has special features that enhance the copy, whenever large blocks of data have to be transferred. These features include the utilization of the "rep movs" assembler instruction. We thus provide an inline assembly code for copying large blocks of data instead of using the memcpy() function provided by the string.h interface. We show this in figure 4.12.

ADDR is a macro that disables possible gcc optimizations.

```
#define REP_PREFIX  "rep ;"

static inline void _memcpy_ (volatile void* dest, volatile void* src, int taille)
{
   asm  volatile ("cld\n\t"
                  "movl %2, %%ecx\n\t"
                  "lea %1, %%edi\n\t"
                  "lea %0, %%esi\n\t"
                  REP_PREFIX "\tmovsb"
                  : /* no output */
                  :"m" (ADDR(src)), "m" (ADDR(dest)), "g"  (taille)
                  : "ecx", "edi", "esi" ); /* clobber registers */
}
```

Figure 4.12: Inline assembly code for copying large blocks of data in CLIP

Chapter 5

Performance

In the previous chapter, we dealt with the design keys of CLIP. In particular, we looked into several ways of addressing the issues of performance introduced in the first two chapters. So far, our approach regarding these issues has mainly been to focus on pipeline, direct memory access with the means of DMA, and the crossing of the PCI. In order to evaluate the design tradeoff of our implementation, we now turn our attention to the performance realized by the communication primitives of CLIP. In this chapter, our main interest is to address two important aspects of communication performance. Firstly, we are interested in knowing the latency contribution of our messaging layer in sending a few bytes of message into the network. Secondly, we would like to know what percentage of the available peak bandwidth the CLIP's communication layer is able to cover when sending large messages into the network.

Since these two aspects of the communication performance are relying on accurate values of the host CPU clock, we will first discuss issues related to the timing. Next, we will introduce the issues of the latency and bandwidth in this order.

5.1 Timing issues

For the latency and bandwidth benchmarks, we measure the elapsed time in two of the following ways :

- the first method consists in reading the value of the Pentium II Time Stamp Counter (TSC) register by executing the RDTSC assembler instruction. A side effect of executing this instruction is to load the 64-bit value of the TSC register into the EDX:EAX registers of the CPU. The following macro shows how we coded this:

  ```
  #define ix86_get_tsc(tsc) \
  __asm__("rdtsc" : "=a" (*((long *)tsc)), "=d" (*((long *)tsc + 1)) :: "eax","edx")
  ```

- in the second method we read the user and system times of the host CPU via the times() interface of the times.h header.

Both methods have been applied in order to obtain accurate values of the elapsed time.

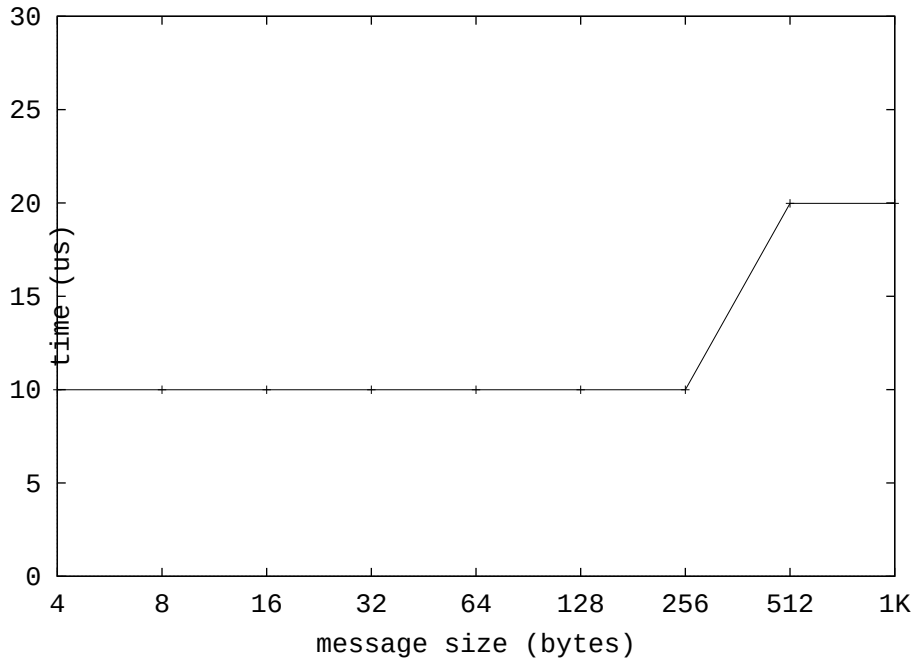

Figure 5.1: One-way latency for short messages

5.2 Latency benchmark

With the basic communication operation of CLIP being “send”, we have measured the elapsed time needed to transfer a few bytes of data from one node to the other. This experience is repeated 1000 times, then the average elapsed time is computed. We show in figure 5.1 the results we obtained for messages´ sizes up to 1024 bytes. The figure 5.2 extends these results to sizes of messages going up to 64KB.

For messages´ sizes of up to 256 bytes, the incurred latency in CLIP is about 9.9 μs. For these sizes, we used a PIO-based transfer, i.e the data are copied into the LANai memory via our wrapped memcpy() function. For messages greater than 256 bytes, we used a DMA-based transfer instead (see chapter 4, section 4.2). Messages in the range of 512 bytes to 1024 bytes have a latency of about 19.98 μs.

5.3 Bandwidth benchmark

For the bandwidth benchmark, we proceeded similarly to the latency benchmark. However, instead of restricting the measurements to messages of small sizes, we measure the elapsed time for sending messages of up to 64 KB into the network. The experiences are repeated 1000 times, then the average value is computed. The result is shown in figure 5.3.

The peaks observed in the figure correspond to the sizes of messages at which the algorithm automatically adapts the size of the packet to its optimal value.

In order to emphasize the advantages of applying the adaptive pipeline method and the snoopy strategy introduced in the previous chapter, we show in figure 5.4 a comparison of the bandwidth obtained for

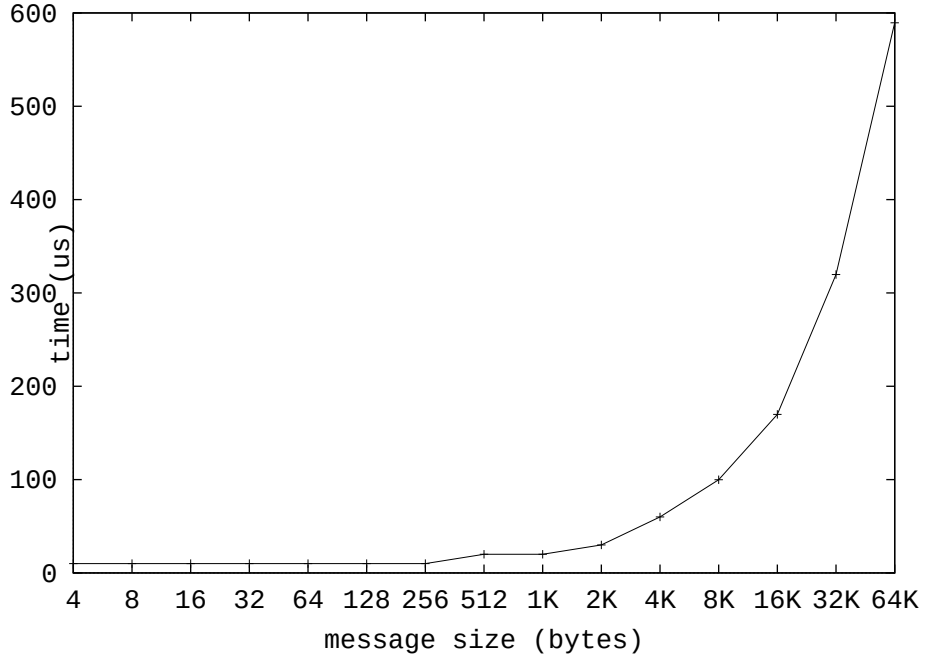

Figure 5.2: One-way latency for big messages

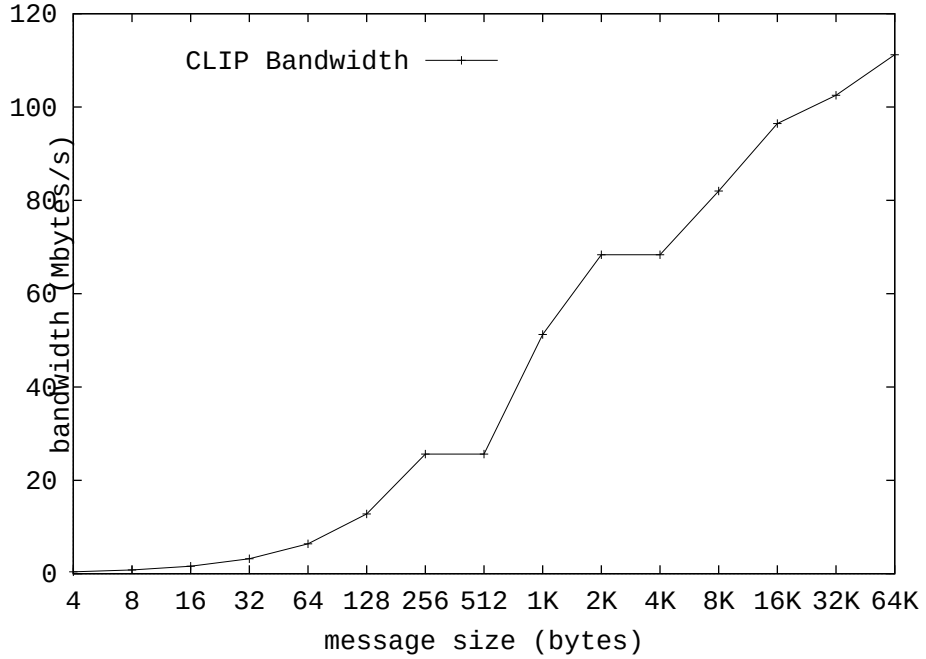

Figure 5.3: Bandwidth of CLIP

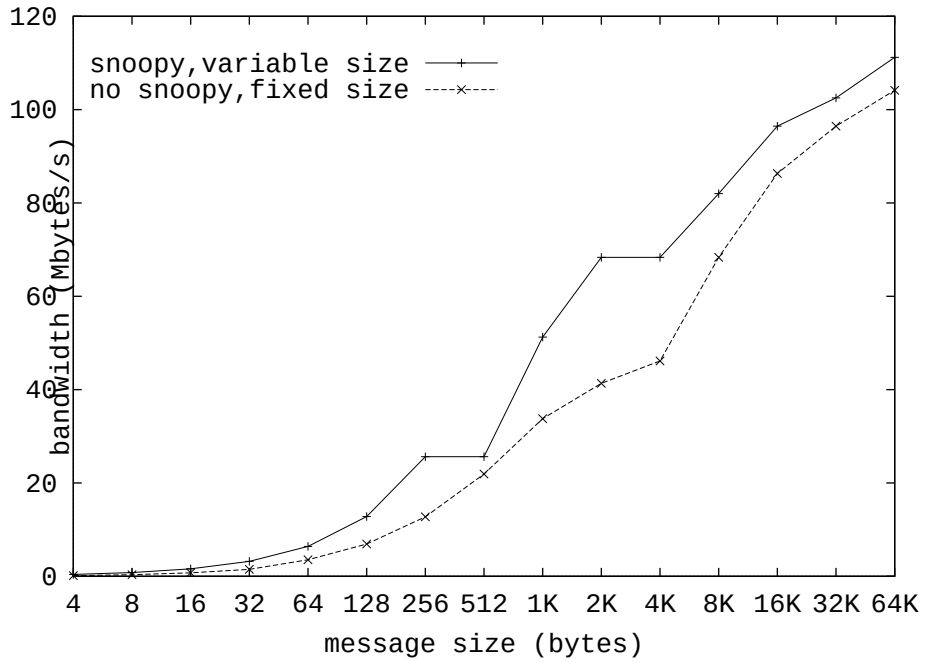

Figure 5.4: Bandwidth of CLIP (a) with snoopy and a variable packet size, (b) without snoopy and with packets of fixed sizes

size in bytes	Bandwidth in MB/s	Bandwidth in MB/s
	snoopy and variable size	no snoopy and fixed size
1024	51.25	33.77
2048	68.33	41.33
4096	68.33	46.15
8192	82.00	68.33
16384	96.47	86.32
32768	102.50	96.47

Table 5.1: Bandwidth comparison

sending messages of fixed-size into the network without applying the snoopy strategy with the bandwidth achieved by sending messages of variable sizes into the network with the snoopy strategy.

As Figure 5.3 illustrates, the peak bandwidth achieved in the former case is about 104 MB/s, whereas for the latter case, a maximum of 111,2 MB/s is reached. For all sizes of messages the latter case performs better than the former. As an example, table 5.1 presents some comparisons done for sizes of messages from 1 KB up to 64 KB.

Chapter 6

Conclusion

Building clusters of workstations out of commodity components such as high speed networks and powerful microprocessors, alone, do not suffice if one wants to achieve a similar performance to the one attained by massive parallel multiprocessors (MPPs). We have seen throughout this work that many things must be done in order to realize the potential performance offered by these commodity components. This starts by re-thinking the manner by which the communication protocol is usually proceeded in traditional operating systems. As highlighted in chapter two, reducing the software overhead generated by the communication protocol architecture represents one of the biggest challenges in writing high performance messaging layer. We have seen that many solutions have been applied to overcome the obstacle of the traditional approach. Some of them have tried to substitute new communication paradigms to existing ones that are based on the message-passing or the remote copy semantic. These are the Active Message [**?**, **?**], or, to a certain extent, the FM [**?**, **?**] too. Others have chosen to rely on traditional communication abstraction methods. They have come out with fine-tuned messaging layers. This is for instance the case of BIP [**?**], whose implementation belongs to the message-passing communication abstraction. Our design approach follows the idea of BIP. We entirely relied on the usual message-passing communication abstraction while keeping in mind the issue of performance. This has led us to establish a carefully designed communication abstraction that exports as much of the underlying hardware potential to application programs as possible, achieving a peak bandwidth of 111.2 MB/s for messages of 64 KB while keeping the latency under the range of 10 μs (actually $\approx$9 μs) for messages of 4 bytes. However, unlike most solutions suggested by others, we do not claim with our approach to provide the best solution for the communication problem introduced above. We are far from this. Rather, our contribution to this subject is to provide a communication building block that can be viewed as a component of a flexible communication interface through which an application program has now the possibility to customize its communication environment for fitting a particular requirement. We showed in chapter four how this has to be taken up in CLIP by introducing the concept of EPOS [**?**] and its three design keys (the scenario-adapters, the inflated interfaces, and the scenario-independent system abstractions). At the time we are writing this thesis, some preliminary results regarding the performance achieved by a communication channel implemented in EPOS were already available. The tests that were carried out consisted in sending an increasing size of message between two machines interconnected by Myrinet. The same communication abstraction was employed in two distinct environments: a single-task and a multi-task environment. The results obtained in both contexts have showed a difference, in favor of the single-task configuration, of about 22% for messages of 16 bytes and 46% for 64 KB messages. Thus, this demonstrates that it really pays off having application programs specializing their environment to fit a specific requirement.

Obtaining the potential performance of the underlying hardware with the aim of exporting it to application programs also requires one to have a better understanding of the interaction between the components that participate in the design of a high performance cluster of workstations. We gave a synoptic description of

some of these components in the chapter three where we described some popular interconnection networks. From the experience that we won by designing the communication layer of CLIP discussed in chapter four, we learned some lessons for a better understanding of the interaction between the various hardware elements that participate in the realization of a high performance communication library. These are listed below.

- we must avoid all unnecessary memory accesses during a transfer because the memory system is now becoming a critical path. One must have in mind that each memory access forcefully leads to a decrease of the end-to-end performance of the system;
- the performance of today's I/O buses is usually behind or on the same level as that achieved by high speed networks. Thus, all transfers that pass over the I/O bus must be optimized to use special hardware features such as burst transfer mode which yields better performance. In the other cases, transfers over the bus must be done only when necessary;
- knowing that NICs of today have a processing capability lower than that of the host CPU, the issue regarding the amount of communication protocol which must be implemented on the NIC is of significant importance. Carefully balancing the protocol among them may lead to a better performance;
- latency hiding techniques such as pipelining, might be sometimes unavoidable if one wants to compensate the drawback introduced by other elements of the design. Whenever possible one should try to model such aspect of the system, looking for ways to tune it, before starting effectively to implement it.

However, exporting a complete ready-to-run high performance communication layer to application programs requires more than what has been done so far. Filling the gap of the design that was addressed within the scope of this thesis to delivering a powerful messaging layer to application programs will require to look into some other important issues. These issues may be provided at the scenario-adapter level introduced in the chapter four, or they could even be directly integrated into the design. For instance, supporting the issue of SMP might require that concurrent access onto the network hardware be provided. If the cost of binding such an execution scenario to an application program might seem dear, then one should better decide to implement it directly on the adapter card. Guaranteeing message delivery may also fall into the class of scenario-adapters.

Even though the knowledge in the cluster computing field seems to have progressed much during this decade, so that a great number of powerful communication abstractions available in the scientific community have succeeded in considerably reducing the effect of the software overhead, we believe that the progress realized in the hardware would be of a decisive contribution in the advances that are likely to be made in this field of knowledge in a near future. For instance, having larger I/O bus would certainly contribute to reduce the performance gap between high speed network and actual I/O bus which represents a bottleneck on most today's systems and thus, leading to a better achievable end-to-end performance. As an example, a 64-bits PCI bus currently available on the market achieves a peak throughput of 528 MB/s. This is 3 times greater than the throughput achieved by the Myricom network we are using. Fast memories may also contribute to decrease the latency of accessing data in memory. However, the cost of the system might also raise proportionally to the price of purchasing such memory modules. Providing network adapter cards with much more powerful on-board processor and even more specialized built-in hardware features can also considerably increase the processing capability of the NIC and thus, leading to a much faster communication protocol processing.

Chapter 7

Appendix A : OMT-Notation

We have used diagrams throughout chapter four in order to illustrate the relationship between classes. These relationships complie with the semantic of the object modelling technique.

In OMT, a class diagram is usually represented by a box with the class name appearing in its middle (see Figure A.1).

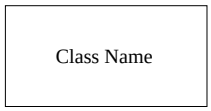

Figure A.1 : OMT Class Representation.

The inheritance relationship specifies the type of association between a parent class and its subclass (see Figure A.2). A Subclass that inherits from its parent class includes all the definitions of data and operations declared in his parent class.

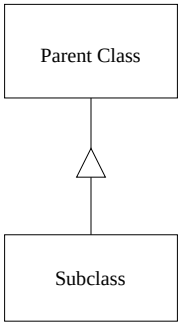

Figure A.2 : Class Inheritance

The aggregate relationship is depicted in Figure A.3. A class A which generates an aggregate instance of another class B is responsible for the instantiated object of the latter class. Another way to express this relation is to say that class A “owns” an instantiated object of class B.

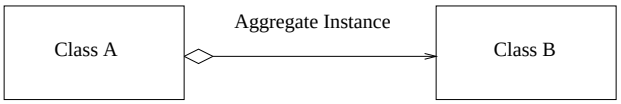

Figure A.3 : Aggregate Relationship

The acquaintance relationship is also expressed as the “using” relationship and simply emphasizes the fact that an instantiated object of class A “uses” another instantiated object of class B. This relation is shown in Figure A.4 below.

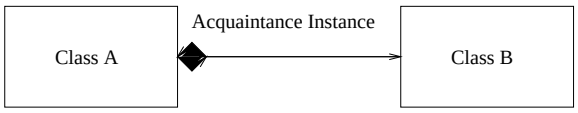

Figure A.4 : Acquaintance Relationship

Chapter 8

Appendix B : Simple Linear Regression Model

Given the linear model described in chapter four (see 4.2.2.2, equation (1)) for computing the elapsed time experienced by a message or packet sent through the network, a simple linear regression approximation [**?**] can be applied in order to estimate the value taken by the per-packet or per-message cost β_i and the inverse of the bandwidth τ_i respectively. These two parameters are called regression parameters, and the variables used to determine them are known as predictor and response variables.

By measuring the response variable T_i given in equation (1) of chapter four (4.2.2.2) for different sizes P_i of message (predictor variable) at each stage i of the pipeline model of figure 4.10, one computes the regression parameters β_i and τ_i for each stage i of the pipeline as follows:

$$\tau_i = \frac{\sum TP - n\overline{TP}}{\sum P^2 - n(\overline{P})^2} \text{ (B.1)}$$

$$\beta_i = \overline{T} - \tau_i \overline{P} \text{ (B.2)}$$

where

$$\overline{P} = \text{mean of the values of predictor variables} = \frac{1}{n}\sum_{i=1}^{n} P_i \text{ (B.3)}$$

$$\overline{T} = \text{mean of the values of response variables} = \frac{1}{n}\sum_{i=1}^{n} T_i \text{ (B.4)}$$

$$\sum TP = \sum_{i=1}^{n} T_i P_i \text{ (B.5)}$$

$$\sum P^2 = \sum_{i=1}^{n} P_i^2 \text{ (B.6)}$$

Bibliography

[1] Gregory F. Pfister [1995]. In Search of Clusters - The Coming Battle for Lowly Parallel Computing. Prentice Hall, Inc.

[2] David E. Culler and Jaswinder Pal Singh [1999]. Parallel Computer Architecture - A Hardware/Software Approach. Morgan Kaufmann Publishers, Inc.

[3] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpassi-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwarning, Richard Martin, Chad Yoshikawa and Frederic Wong [1997]. Parallel Computing on the Berkely NOW. In proceedings of the 9th Joint Symposium on Parallel Processing - JSPP'97, Kobe, Japan.

[4] John L. Hennessy and David A. Patterson [1996]. Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kaufmann Publishers, Inc.

[5] Peter Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. In IEEE Computer, March 1994.

[6] Alan Mainwaring and David Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Draft Technical Report, Computer Science Division, University of California at Berkeley.

[7] T. von Eicken [1993]. Active Messages: An Efficient Communication Architecture for Multiprocessors. PhD thesis, University of California at Berkeley.

[8] Steven S. Lumetta, Alan M. Mainwarning and David E. Culler [1997]. Multi-Protocol Active Messages on a Cluster of SMP's. In Proceedings of SC97.

[9] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. SMP PCs : A Case Study on Cluster Computing. In Proceedings of the Firstsup Euromicro Workshop on Network Computing, Västeras, Sweden, August 1998, p. 953-960, ISBN 0-8186-8646-4.

[10] Alessandro Rubini [1998]. LINUX Device Drivers. O'Reilly & Associates, Inc.

[11] Scott Pakin, Vijay Karamcheti and Andrew A. Chien [1997]. Fast Messages (FM) : Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. In IEEE Concurrency, vol. 5, no. 2, April-June 1997, pp. 60-73.

[12] Scott Pakin, Mario Lauria and Andrew Chien [1995]. High Performance Messaging on Workstations : Illinois Fast Messages (FM) for Myrinet. In Supercomputing '95, San Diego, California.

[13] Hiroshi Tezuka, Atushi Hori and Yutaka Ishikawa [1996]. PM : A High-Performance Communication Library for Multi-user Parallel Environments. Technical Report TR-96015, RWC, November 1996.

[14] Loic Prylli and Bernard Tourancheau [1997]. BIP : A New Protocol Designed for High Performance Networking on Myrinet. In Workshop PC-NOW, IPPS/SPDP98, Orlando, USA, 1998.

[15] Loic Prylli [1998]. Réseaux et Systèmes de Communication. Etude de logiciels de base. PhD thesis, Ecole Normale Supérieur de Lyon.

[16] Thorsten von Eicken, Anindya Basu, Vineet Buch and Werner Vogels [1995]. U-Net : A User-Level Network Interface for Parallel and Distributed Computing. Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado.

[17] Mengjou Lin, Jensey Hsieh, David H. C. Du, and Joseph P. Thomas [1995]. Distributed Network Computing over Local ATM Networks. IEEE Journal on Selected Areas in Communications Special Issue of ATM LANs: Implementations and Experiences with an Emerging Technology.

[18] Eric Dillon [1996]. De l'Utilisation de ATM pour le calcul distribué. Technical Report, INRIA, France.

[19] Werner Almesberger [1995]. High-Speed ATM Networking on Low-End Computer Systems. Technical Report 95/147, DI-EPFL, August 1995.

[20] J. de Rumeur [1994]. Communications dans les Réseaux de Processeurs. Etudes et Recherches en Informatique, Masson.

[21] White paper of the LANQUEST Laboratory [1998]. A Competitive Performance Benchmark Tests of Gigabit Ethernet Adapters. http://www.lanquest.com

[22] Bernard Daines [1996]. Full Duplex Repeater, IEEE Standards. Slide presentation made to the 802.3z task force, ftp://stdsbbs.ieee.org/pub/802_main/802.3/gigabit/presentations/sep1996/BDreptr.pdf.

[23] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic and Wen-King Su [1995]. Myrinet: A Gigabit-per-Second Local-Area Network. In IEEE Micro.

[24] Robert E. Felderman, Annette DeSchon, Danny Cohen and Gregory Finn [1994]. Atomic: A High Speed Local Communication Architecture. Journal of High Speed Networks.

[25] Charles L. Seitz, Nanette J. Boden, Jakov N. Seizovic and Wen-King Su [1992]. The Design of the Caltech Mosaic C Multicomputer. Technical Report, California Institute of Technology.

[26] Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman and Manmfred Weis [1997]. High-Performance Cluster-Computing using SCI. In Proceedings of Hot Interconnects V, Stanford, CA, USA.

[27] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini and J. Prusakova [1997]. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing.

[28] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson [1996]. "NetPIPE: A Network Protocol Independent Performance Evaluator," IASTED International Conference on Intelligent Information Management and Systems.

[29] F. Schön, W. Schröeder-Preikschaft, O. Spinczyk and U. Spinczyk [1998]. Design Rationale of the PURE Object-Oriented Embedded Operating System. In proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems, Paderborn, Germany.

[30] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-Oriented Operating Systems - the EPOS Approach. In Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing, Natal, Brazil, September 1999, p. 3-9.

[31] Bjarne Stroustrup [1999]. Le Langage C++. CampusPress France.

[32] Matt Welsh, Anindya Basu, and Thorsten von Eicken [1994]. Incorporating Memory Management into User-Level Network Interfaces. Presented at Hot Interconnects V, August 1997, Stanford University.

[33] Randolph Y. Wang, Arvind Krishnamurthy, Richard P. Martin, Thomas E. Anderson, and David E. Culler [1998]. Modeling and Optimizing Pipeline Latency. SIGMETRICS Conference on the Measurement and Modeling of Computer Systems , Madison, Wisconsin , 6/24/98 - 6/26/98 .

[34] Raj Jain [1991]. The Art of Computer Systems Performance Analysis. JOHN WILEY & SONS, INC.

[35] Richard P. Martin, Amin M. Vahdat, David E. Culler and Thomas E. Anderson [1994]. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In ISCA 24, Denver, Co , June, 1997 .

[36] Matt Welsh, Anindya Basu, and Thorsten von Eicken [1997]. ATM and Fast Ethernet Network Interfaces for User-Level Communication. IEEE, Proceedings of the Third International Symposium on High Performance Computer Architecture, San Atonio, Texas, USA.

[37] R. Felderman, A. DeSchon, D. Cohen, and G. Finn [1994]. A high speed local communication architecture. Journal of High Speed Networks, Volume 3,1994.