COMMUNICATION BOARD FOR CLUSTERING: SOFTWARE DEBUGGING AND PERFORMANCE EVALUATION

Ricardo Miguel da Costa Guapo, nº 45289, Ramo de SEC

LICENCIATURA EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES
Relatório de Trabalho Final de Curso
29/2002/M

Orientador: Prof. Doutor Leonel Augusto Pires Seabra de Sousa

Setembro de 2003
Acknowledgements

I would like to thanks the useful help of:

Prof. Leonel Sousa, Dr. Shinichi Yamagiwa and Kevin Ferreira for all support and dedication along the execution of this work, they were unstoppable in the help provided;

Signal Processing Systems Group and Maestro2 development team for supplying the computational resources used in this work;

And finally to all my colleagues and relatives that supported my work.
Abstract

Maestro2 is a high performance network equipment for cluster computing currently being developed. It will support message passing and distributed shared memory programming paradigm. This report, presents the results of static and dynamic analysis done to Maestro network systems. The static analyses involves implementing a debugging tool for the high performance network interface of Maestro2. Dynamic analysis was based on experimental results obtained by running benchmark packages. The evaluated system based on the Maestro2 consists of multiple PCs (Personal Computer) and dedicated network hardware for high performance message passing and for maintaining memory coherency. The debugging tool was implemented based on already existing debugging programs and by exploiting their capabilities. The dynamic performance of two network systems based on Maestro2 and Myrinet technologies, was measured and compared using a well-known public benchmark for obtaining more realistic and unbiased analysis. Results of the latency analysis show that Maestro2 is 3 times better than Myrinet in terms of special communication software and offers 300% of the throughput of Myrinet network communication system.

Keywords:
Clusters, high performance computing, debugging embedded hardware, communication protocols, performance evaluation
Resumo

O Maestro2 é um sistema de rede de dados de alto desempenho para equipar clusters, que se encontra actualmente em desenvolvimento e que suportará os modelos de programação de passagem de mensagens e memória partilhada. Neste relatório são apresentados os resultados das análises estáticas e dinâmicas efectuadas ao sistema. A análise estática envolveu o desenvolvimento e implementação de uma ferramenta de debug na placa de rede de alto desempenho do Maestro2. A análise dinâmica foi baseada em resultados experimentais obtidos com base em procedimentos específicos para avaliação do desempenho de sistemas paralelos. O sistema Maestro2 consiste num agregado de computadores pessoais e hardware dedicado para a função de passagem de mensagens e para manutenção da coerência da Memória. A ferramenta de debug foi implementada recorrendo a aplicações já existentes e na exploração das suas funcionalidades. A performance dinâmica de dois sistemas, o Maestro2 e a Myrinet, foi medida e comparada usando uma aplicação de benchmarking pública e conhecida, para obter resultados válidos e gerais. A análise efectuada à latência do sistema demonstra que o Maestro2 é 3 vezes mais rápido que a Myrinet ao nível da camada MPI e oferece cerca de 300% da capacidade de transmissão do sistema Myrinet.

Palavras Chave:
Clusters, computação de alto desempenho, depuração de erros em hardware embebido, protocolos de comunicação, avaliação do desempenho
Index

1 Introduction ........................................................................................................................ 1
  1.1 Background ................................................................................................................. 1
  1.2 Objectives ................................................................................................................. 1
  1.3 Work description ...................................................................................................... 1
  1.4 Report structure ...................................................................................................... 1
2 Cluster networks ............................................................................................................. 3
  2.1.1 Overview of cluster networks ............................................................................ 3
  2.1.2 Parallel applications in clusters .......................................................................... 5
  2.1.3 Components of Cluster .................................................................................... 5
  2.1.4 Layers of cluster network .................................................................................. 7
  2.1.5 Hardware layer ................................................................................................. 8
  2.1.6 Software layer .................................................................................................. 11
  2.2 Conclusions ............................................................................................................ 15
3 Debugging tool .............................................................................................................. 17
  3.1 Introduction ............................................................................................................. 17
  3.2 Debugging methods .............................................................................................. 17
  3.3 The debugging architecture and implementation aspects ....................................... 19
  3.4 Developed software on the PC .............................................................................. 20
  3.5 Developed software on the NI ............................................................................. 23
  3.6 Conclusion ............................................................................................................. 25
4 Performance evaluation .................................................................................................. 26
  4.1 Introduction ............................................................................................................ 26
  4.2 Benchmark tool ..................................................................................................... 26
  4.3 Experimental setup ............................................................................................... 30
  4.4 Experimental Results ........................................................................................... 31
    4.4.1 Experiment 1 – Software communication libraries overhead analysis .......... 32
    4.4.2 Experiment 2 - Comparative results of the protocol performance ............. 39
    4.4.3 Experiment 3 – Comparison of different software configurations ........... 45
  4.5 Conclusion ............................................................................................................. 48
5 Final Conclusions and Future Work ............................................................................. 49
6 Appendix A ................................................................................................................... 51
7 Appendix B ................................................................................................................... 53
8 Appendix C ................................................................................................................... 54
9 Appendix D ................................................................................................................... 56
10 Appendix E ................................................................................................................ 60
11 Appendix F ................................................................................................................ 69
12 Appendix G ................................................................................................................ 74
13 Bibliographic references ............................................................................................ 79
List of Figures

Figure 1: Generic distributed memory system ................................................................. 4
Figure 2: Cluster communication layers ........................................................................... 7
Figure 3: Maestro2 Cluster Network ............................................................................. 10
Figure 4: Myrinet NIC design ...................................................................................... 11
Figure 5: MPI layers interfacing program and hardware ............................................... 13
Figure 6: Cluster communication layers trend ............................................................... 15
Figure 7: Single machine debugging communication model ......................................... 18
Figure 8: Remote machine debugging communication model ....................................... 19
Figure 9: Remote processor debugging communication model .................................... 20
Figure 10: NI debugging physical model ....................................................................... 20
Figure 11: NI debugging logical model ........................................................................... 20
Figure 12: Character device driver logical model ......................................................... 21
Figure 13: Communication scheme from NI perspective ............................................... 24
Figure 14: NetPIPE benchmark algorithm ................................................................... 28
Figure 15: Example of a Fast Ethernet Throughput ....................................................... 28
Figure 16: Example of a Fast Ethernet Signature Graph ............................................... 29
Figure 17: Example of a Fast Ethernet Saturation Graph ............................................... 30
Figure 18: Myrinet network supported protocols throughput ....................................... 33
Figure 19: Myrinet network supported protocols signature graph ................................ 35
Figure 20: Myrinet network supported protocols saturation graph ................................. 36
Figure 21: Maestro2 network supported protocols throughput graph ........................... 37
Figure 22: Maestro2 network supported protocols signature graph ............................... 38
Figure 23: Maestro2 network supported protocols saturation graph ............................. 39
Figure 24: TCP/IP throughput graph ............................................................................ 40
Figure 25: TCP/IP signature graph ................................................................................ 41
Figure 26: TCP/IP protocol saturation graph ............................................................... 41
Figure 27: MMP and GM throughput graph ................................................................... 42
Figure 28: MMP and GM saturation graph ................................................................... 42
Figure 29: MPI throughput graph ................................................................................ 43
Figure 30: MPI signature graph ................................................................................... 44
Figure 31: MPI saturation graph ................................................................................... 44
Figure 32: MPI over MMP with different conditions .................................................... 46
Figure 33: One-copy operation procedure .................................................................... 47
Figure 34: Zero-copy operation procedure ................................................................... 47

List of Tables

Table 1: Stub main subroutines and description ................................................................ 23
Table 2: Stub invocation exception types .......................................................................... 25
Table 3: Latencies observed on Maestro2 system ........................................................... 39
Table 4: MPI measured values ....................................................................................... 43
Table 5: System minimum latencies in microseconds ..................................................... 45
Acronyms

ADI: Abstract Description Interface
API: Application Programming Interface
CPU: Central Processing Unit
DMA: Direct Memory Access
FFT: Fast Fourier Transform
FIFO: First In, First Out
FPGA: Field Programmable Gate Array
LAM: Local Area Multicomputer
LAN: Local Area Network
LVDS: Low Voltage Differential Signaling
MIMD: Multiple Instructions, Multiple Data
MMP: Maestro Message Passing
MPI: Message Passing Interface
MPP: Massively Parallel Processor
NetPIPE: Network Protocol Independent Performance Evaluator
NI: Network Interface (Card)
NIC: Network Interface Card
OS: Operating system
PC: Personal computer
PCI: Peripheral Component Interconnect
PVM: Parallel Virtual Machine
RSP: Remote Serial Protocol
SB: Switch Box
TLB: Translation Look-aside Buffer
WAN: Wide Area Network
1 Introduction

1.1 Background

Cluster computing offers the same computational power of super computers at a much lower cost. Due to this, cluster computers have become the vehicle of choice to build high performance computing environments. To fully exploit the computational power of these environments, one must utilize high performance network and protocol technologies, since the communication patterns of parallel applications running on clusters require low latency and high throughput. The development of these new technologies requires tools for the evaluation of their archived results on the overall cluster system.

1.2 Objectives

The evaluation of the performance of communication strategies is done thru the analysis of the system. Overall communications performance depends on the potential performance that hardware offers and from how much the software can take advantage of it. Hardware presents the maximum potential performance reachable by the system. This can be considered the static performance of the system. Communication software presents performance degrading overheads that change depending on the communication strategies. This can be considered the dynamic performance of the system.

To consider these two categories of performance this work focus methods to extract, see and analyze the characteristics of the system by the user side.

1.3 Work description

This work involved performing static and dynamic analysis to a cluster system. To execute the static analysis to the cluster system, a debugging tool was developed and implemented in a specific hardware system, Maestro2 [1].

Dynamic analysis was performed using known benchmarks to evaluate the different communication strategies in message passing libraries. Analyzed libraries were MPI [2] and MMP and the benchmark used was NetPIPE. A Myrinet based computer network [29] was also experimentally analyzed in order to compare the obtained performance.

1.4 Report structure

In the following chapter, Chapter 2, cluster computing and its necessity, pros and cons are introduced. It follows a discussion why cluster computing become so popular, what are its current flaws (weak points) and what is being done to override those that are at the communication layer. Network technologies used in this work are described in detail. After
that, message passing communications systems used in this work are described, and also why message passing is the preferred communication way in clusters. 

Chapter 3 focuses in the static analysis of the hardware layer of Maestro2. It describes in detail the implementation of the debugging tool used to extract information from the network device. 

In Chapter 4, dynamic analysis is studied at the user level. Different techniques were used to show main characteristics of the message passing libraries used and also the ones that depend from the cluster system involved. 

Chapter 5 concludes this report by focusing main analysis results retrieved from Maestro2 system with this work and indicates future areas of research in this system pointed out by the analysis results.
2 Cluster networks

2.1.1 Overview of cluster networks

The demanding for more computational power has been one of the main driving forces in the development of computers. Before the dawn of the computer era, scientists and engineers thought that the ability to carry out a few hundred arithmetic computations each second was nearly unimaginable. However, almost as soon as they found themselves with this much computational power, they began to clamor for more. They wanted to be able to carry out thousands of operations each second. And after thousands it was millions, always wanting more than the available computational power. The impetus for this rush for greater computing power is the technological advance itself. The more knowledge is acquired, the more complex questions and problems to solve become. It is not difficult to find problems requiring vastly greater computational power than it is currently available. To scientists and engineers, simulation is replacing experimenting and prototyping, because it is cheaper, faster, and there are experiments that can be simulated but not experimented.

Trying to develop one single Von Neumann computer with a extremely fast processor to solve the amount of calculations required per second is already physically impossible, being the speed of light the limit. The approach used to solve this lack is hiring more computers to increase the computing power and make them work together. A parallel computer is a computer or a collection of computers with multiple processors that can work together on solving a problem.

This definition is global enough to include super computers that have hundreds or thousands of processors, aggregates of computers, each having one or more processors or even a network of personal computers.

The configuration where the system is built by several independent computers connected by network is preferentially assumed as distributed processing, where as parallel processing is assumed for systems of computers with several processors. In both cases, the aggregate of computers working to solve the same problem is formally called cluster computing [3]. They got this name because normally the computers working together are physically packed together in a geometrical pattern to minimize distance between them.

As the need for processing power increased all around the world, the implementation of clusters become a popular solution and so research in parallel processing and programming areas became increasingly active since long time ago (in the past few years) [4].

The possibility to solve bigger problems faster is what cluster computing is offering.

At the hardware level there are many different architectures for parallel computing, being MIMD (multiple instructions, multiple data) the one that describes cluster computing the best. In MIMD each CPU (Central Processing unit) is fully autonomous and independent, each one running its own program with his set of data at its own pace.

MIMD systems are divided into two main implementation schemes: shared memory and distributed memory systems, being the first applied to super computers, and the later one to cluster computers configurations. In distributed memory systems, cluster computer configuration, each processor has its own private memory, but which is globally addressed and accessed by other processors if they need. A generic distributed memory system can be represented as in Figure 1.
Several computers were built recurring to this schemes, but they all present disadvantages at the communication level. Both systems main drawback is communication bottleneck. Each time a CPU has to access a memory segment he does not own he has to spend many idle time for the data to arrive, because the networks is not so fast as memory access, and also because there can be other transmissions occurring in the network, thus incurring in communication overhead, that will degrade performance.

Both generic schemes have been studied [5] and there are currently many variations that try to minimize these communications problems.

The idea of a process is a fundamental building block in most paradigms of parallel computing. Intuitively, a process is an instance of a program or a subprogram that is executing more or less autonomously on a physical processor. A program is parallel if, at any time during its execution, it can comprise more than one process. In order to create useful parallel programs, there must be ways that processes can be specified, created, and destroyed, and there must be ways to coordinate interprocess interaction. Cluster manufacturers offer the software tools to address these issues. Examples of these tools are OpenMP [6], High Performance FORTRAN (HPF) [7].

The advantages of cluster computing are also the impetus for the continuing research and the popularity of clusters. The main advantages are:

- Flexibility: the replacement of one of the computers by another one is an easy task, or the change of individual components like processors or disk arrays, in each case to increase processing speed or system storage capacity.
- Portability: The use of small computers assures great mobility to the system itself being possible to disassemble the cluster and reassemble it some were else with few expenses. Portability can also be reported to the software were applications can be migrated from cluster to cluster as needed. As clusters use commercial standard components, the gradual upgrade of individual components is possible due to the retro compatibility of the components.
- Small cost: Building a cluster using average processing power computers is cheaper than using a top of the art mainframe server computer with several processors which totalize equivalent computing power to the cluster. This is so because of the exponential rise of the price when looking for better performance equipment.
- Interactivity: Batch program applications execution in cluster computers is much more interactive than their run in supercomputers. In supercomputers, programmers have to cue their programs to be run on the super computers and later will check the results of the run. In cluster computing programmers can interact with other users and with the hardware system itself, because of their physical proximity.

In clusters, each individual computer is called node, as it is a concentration point for the network supporting communications.
2.1.2 Parallel applications in clusters

Engineers and scientists are gradually replacing the prototyping and experimenting by simulations, mainly by the cost but also by the flexibility and portability simulations offer. Change of parameters in simulations is almost cost less and effort less compared with the design and development of a new prototype, and there is also the case were some simulated experiments can not be realized in the real world: universe expansion, atomic or molecular reactions, etc. Many other areas already discovered the use to give this computational power: entertainment is one of them, using it to encode or render complex computer generated images [8].

The search for more computational power was originated by the need for getting results faster, so performance in cluster applications is a major key point in their evaluation. The performance in clusters is severely influenced by the needed communication between each computer due to application needs. The data throughput/latency of communication is several times less/more than the each node memory throughput/latency, making the processor spend much idle time waiting for the needed data to arrive.

So, now performance evaluation/increase of cluster computing is not that easy just like knowing/increasing the processing power of each computer in the cluster, like when there was needed to speedup execution time when running applications in a single processor computer.

2.1.3 Components of Cluster

In clusters, performance depends on many factors, and it is easy to know how fast a program runs by just looking at his execution time, but however it is difficult to know how to tune up the application’s communication strategy. When evaluating a program execution time in a cluster it is practically impossible to know in advance the effect that would have a change in the interprocess communications strategy because, they have their pros and cons that are normally severely changed by the underlying hardware where the applications are run.

Such unpredictability in performance knowledge has to be reduced, and analysis of cluster networks for a closer and detailed look at each system component or module is necessary for that, because as clusters are built by so many components and layers, that only one minor problem in one of them can ruin the whole system performance.

Performance on clusters may depend at the hardware level on:

- Bus speeds, such as PCI (Peripheral Component Interconnect) bus: in each node, the communication is normally handled by dedicated hardware board, which interfaces to the outside network. These network interface cards (NIC) are connected to the computer main board thru PCI bus [9]. The conventional PCI bus interface has not suffered changes for several years and is now arising as a problematic communications performance bottleneck as processors and NIC become faster. At the moment and to face this problem, new bus architectures are being developed as PCI Express, and some, like PCI-X, which is an extension to normal PCI standard, are already being used.

- Network interface card: The ability to satisfy all the communication requests in the fastest possible time is the major performance point. If the application running on the computer that hosts the NIC, here by known as host, has to wait for the card to finish
some transfer in progress it will degrade performance by increasing execution time of application.

- Bandwidth and latency of physical link: the bandwidth of the physical link depends on the medium used, the network topology, etc, and it will be a main factor in communications completing time. More bandwidth not always means faster programs, because of the minimum latency of the link and also because information in the network can be exchanged in small packets.

- Other computer components: of course other computer components may also interfere in performance, like memory speed, CPU speed, and DMA transfer speeds.

At the hardware level, the system can be resumed to a potential performance that can be reached but that will be the limit. How much real performance will be near potential depends also on the software overhead.

Main software overheads are:
- Communication overhead: Transmitted and received data is normally encapsulated with more information that will degrade real bandwidth for which it will only count the sent and received data. In some protocols like Ethernet, a small data to be sent, like 4 bytes long would generate a packet to be transmitted at physical layer of 40 bytes as Ethernet header size is 36 bytes. So 90% of the packet size is header information that contains the addresses of sender, receiver and error correcting codes. The time spent by NIC to create or verify this header information, like the checksum, destination and origin addresses is included in this class of overheads.
- Pipeline and serialization effects: in clusters the transmissions of packets from one host to another is made in a serialized way, making packets to be sent one after another. This introduces several effects in performance that can only be known from an analysis to the system.
- Flow control overhead: creation, termination and synchronization points for processes and also communication links in the cluster are also sources of overhead time not used to main calculation. In communications the data flow control and routing causes delays in the time to complete individual communication hurting the overall communications performance.

With so much potential problems related with the performance it is necessary to address the way by which programmers can know the strategy to improve the performance. The best way is collecting all the information available about each component or part of the system, analyze it and giving it meaning, so that programmers have an easy task choosing the best approach to increase performance. Nonetheless, the system does not behave always the same. Changes done in one component may affect another in an unpredictable way, so it is necessary a continuous analysis of the system, a continuous update of the gathered information.

The analysis to be done to the system can be separated in two different tasks that at the end will provide the results needed. To retrieve system information while it is operating it is necessary to freeze it and take a snap shot of its status. The analysis to this static information would allow programmers to know better the effects of different programming strategies.
Another type of analysis would be done without stopping the system, running and looking at the output results of the application execution. This is dynamic analysis because the system is in full motion and the results only show the final status of the system.

Both static and dynamic analyses have to be performed in order to gather the required information to be provided to programmers and engineers to tune up their project strategies of future implementations. In clusters, the objective is to study system limitations and explore new strategies to improve system performance.

The following sections will describe the layers of clusters and will discuss about the overhead analysis.

### 2.1.4 Layers of cluster network

Communications in clusters are structured in several layers consisting in several stacked protocols or modules, so those clusters have flexibility in changing communicating structures. Communication in clusters is supported by several protocols, each one on top of another, forming a layers structure like the one presented in Figure 2.

![Figure 2: Cluster communication layers](image)

The communication layer includes the hardware layer and the software layer of cluster node. Hardware layer includes both network interface hardware operation and data transmission. NIC operations consist in the forming and encapsulation of packets to be sent to the network, error checking operations, routing operations, creation and destruction of communication links.

Software layer consists in a top layer for providing an interface to applications so that they can use it to use the hardware to exchange data among application programs without knowing what procedures happen to the data to travel from an application in a host to another host. The task of this interface layer is to reply to application requests, generating messages with the data to be sent, or returning to the application data requested and received from the network. Under this layer is the transport layer which is assigned for sending and receiving the messages from application to the network. This layer indicates the messages to be sent to the hardware layer that will pick them up and send them to destination. After receiving a message, hardware layer will notify software layer for it to take care of the data and deliver it to the demanding application.
Each of this structured layers have a particular effect in the overall communication performance.
The following sections will describe the functionality and the effects of hardware and software layers on the overall communication performance.

2.1.5 Hardware layer

To increase performance in communications, the most time consuming tasks inside the host, which is the transfer of the data between host main memory and NIC memory, is from the responsibility of the hardware in the NIC, leaving the host processor clear to have more time to spent on useful calculations for the application.

If this copy process was done using processor time it would degrade cluster performance very much, since cluster applications are in general intensive communications.

So the approach currently used in cluster communications layers is that NIC will be assigned for performing the copy using DMA (Direct Memory Access) transfers. Host processor only has to notify to NIC the memory zone containing the message to be sent to the receiver.

In receptions same procedures occur: host process notifies NIC of memory zone where it wants the received message to be transferred by DMA.

Host processor also notifies NIC with other information required for communications, such as: sender or receiver identifications, message identification, channel to use, and other communication intrinsic information.

Hardware layer is assigned for creating, maintaining and closing communication links between hosts. This links are created and established for the duration of the communication and are used to forward messages split in packets saving time of creating the link each time a new packet o a new message needed to be sent.

In this way, the communication performance starts to depend on many factors like host DMA capacities and less others like processor speed.

In this report two major network architectures were studied as the hardware layers for cluster communications. One was the currently being developed Maestro2 network, a high performance network equipment. Other was the Myrinet network platform that it is currently available in the market.

Maestro2

Maestro2 has been developed to solve performance disparities between the obtainable communication performance using conventional WAN or LAN based network technologies and the requirements of cluster applications.

Of importance is how to support two different aspects of communication: low latency and high throughput. The former must be emphasized on for frequent inter-processors synchronization as seen in distributed simulations, and the latter for mass data transfer, as in matrix computations. Following paragraphs will discuss about the main obstacles to obtain low latency and high throughput in (1) communication protocol software, (2) device handlers, (3) data link layer, and (4) physical layer.
1. Communication Protocol Software
The problem is the repetition of data-copying operations before messages have been transferred and made available to their receiving applications. In general, each message must be copied between the user and the kernel space, and may be further copied between the kernel space and the network device memory. Nevertheless, such copying operations increase communication latency. In addition, the user-to-kernel context switches can be considered as another overhead, especially in case of frequent inter-processors synchronization. The zero-copy communication method [10] permits underlying network devices to access messages directly in the user space, thus bypassing the kernel intervention. For zero-copy communication message areas must be pruned in the main memory so that the OS will not remove those pages to hard disk. This pruning operation is called *pindowning*.

2. Device Handlers
The virtual-to-physical address translation is an obstacle to bypassing the kernel intervention. Since communication buffers at user level are allocated in the virtual address space, their physical addresses must be identified prior to the actual transfers by underlying network devices. So the device should maintain page table and TLB (i.e. Translation Look-aside Buffer) information, so it can take care of such translations without kernel support. This also gives more CPU time for user applications and hides communication overhead further. Device handler will have as main tasks to map the NIC memory in the virtual address space, notify the NIC of the changes in page table or TLB, and also will provide a serial link interface to communicate with the NIC.

3. Data Link Layer
The first problem is conventional message framing. If the minimal frame size is too large, network latency increases relatively in transferring shorter messages. On the other hand, if the maximal frame size is too small, a longer message must be sent with multiple frames, which increase framing operation and thus degrade the total throughput. Assuming that PC clusters are used in a geographically closed environment, messages can be sent directly to their destination with less routing information. Hence, it was designed a new message frame suitable to cluster communication rather than using such as the redundant Ethernet frame, which as has minimum and maximum packet size 36 and 1500 bytes, respectively. The second problem is excessive DMA invocations. Most network devices always invoke DMA transfers to send data over network, whether or not the data size is too small to use DMA efficiently. This is regarded as overhead especially when fine-grained communication is repeated for frequent inter-processors synchronization. Therefore, network devices should include another high-speed transfer logic for a small amount of data, and switch between this logic and DMA according to the data size. The third problem is the current transfer scheme to transmit data on physical link. Provided a packet is a transfer unit dealt at physical layer, most network devices do not aggregate independent packets in one, whether or not those packets are too small. In other words, they repeat an entire transfer set-up operations including physical link arbitration for each small packet. This makes physical link idle frequently, degrading network throughput consequently. To address this problem, network devices must be capable of packet aggregation.

4. Physical Link Layer
The physical link layer must facilitate a multicast scheme that involves neither discarding nor copying messages.

Maestro2 concept hardware implementation is described next:
A simple Maestro2 network is composed of network interfaces (NI) and a switch box (SB) as shown in Figure 3. Each network interface is connected via two LVDS (Low Voltage Differential Signaling) [11] cables for transmission and reception, and is connected to a commodity processing element such as a PC or a WorkStation via a 64bit@66MHz PCI bus.

The connection between NI and SB is full-duplex, and the peak bandwidth is 6.4Gbps. Currently, the SB has eight ports for connecting NI(s). One or more ports can increase the fan out of the switch by cascading SB(s). The NI includes a NI manager, a PCI interface, network FIFO buffers, a link controller (MLC-X) and LVDS transmitter/receiver. The NI manager works as a processor element in charge of handling communications, and is composed of a PowerPC603e@300MHz and 64Mbyte SDRAM. The PCI interface maps the address space of the SDRAM and host processor’s memory into the PowerPC’s address space. The 8Kbyte network FIFO buffers store incoming and outgoing messages. The MLC-X is a full duplex link layer controller on which the continuous network burst transfer is implemented. MLC-X supports two communication channels between the network FIFO buffers. And finally, LVDS transmitter/receiver drive its physical medium under control of MLC-X. It transmits and receives data via its 6.4Gbps full duplex link. The PCI interface, network buffers, and MLC-X are implemented into the Virtex-II FPGA (Field Programmable Gate Array) chip [12]. The SB consists of four SB interfaces, an SB manager and a switch controller. Each SB interface manages two ports and includes a message analyzer. The message analyzer extracts each message header of an incoming message, and passes it to the SB manager. The SB manager consists of a PowerPC603e, 32Mbyte SDRAM, and a routing circuit that generates and writes requests to the switch controller.

**Myrinet**

Myrinet network architecture also focuses low latency and high throughput as clusters applications require.

Myrinet is a cost-effective, high-performance, packet-communication and switching technology that is widely used to interconnect clusters of workstations, PCs, servers, or single-board computers. Characteristics of Myrinet networks include: full-duplex 2+2 Gigabit/second data rate links, maintained by switch ports and interface ports. Supports flow control, error control, and continuity monitoring on every link. The switch networks can scale to tens of thousands of hosts, thru switch cascading, and that can also provide alternative communication paths between hosts. Host interfaces execute a control program to interact
directly with host processes ("OS bypass") for low-latency communication, 0-copy operation and
directly with the network to send, receive, and buffer packets.

Specifications of Myrinet NIC and hardware implementation are as follows:
Myrinet NIC is constituted by several modules depicted in Figure 4. It interfaces with the host
computer by the PCI bus being fully compatible with the several standard PCI specifications.
It accepts bus widths of 64 or 32 bits, and bus speeds of 66 or 33 MHz, and also the new PCI-
X configurations. This interface is capable of sustained PCI data rates approaching the limits
of the PCI bus (528 MB/s for 64-bit, 66MHz; 264 MB/s for 64-bit, 33MHz or 32-bit, 66MHz;
132 MB/s for 32-bit, 33MHz). However, the data rate to/from system memory will depend
upon the host's memory and PCI-bus implementation. It uses DMA transfers to copy
messages data from host main memory to NIC fast local memory. In order to support zero-
copy operations efficiently, the DMA operations can be performed with arbitrary byte counts
and byte alignments.

Figure 4: Myrinet NIC design

Control is performed by a LANai 9 RISC processor operating at 133MHz on PCI64B model,
or at 200MHz for the PCI64C interface. This control unit has a host interface module
coordinating the interaction with the host and setting up the DMA transfers, a packet interface
unit to handle packet transmission and reception.
Myrinet cards exits in the versions of 2MB, 4MB and 8MB of memory, being differentiated
by the -2, -4 or -8 suffix in model name. The local memory operates from the same clock as
the RISC, i.e., at up to 133 MHz for the PCI64B interfaces, or at up to 200MHz for the
PCI64C interfaces. Up to 1,067 MB/s (PCI64B) or 1,600 MB/s (PCI64C) of memory
bandwidth is available to support the Myrinet port, the host DMA, and the RISC processor.
Myrinet uses as physical layer optical fiber cables, one for each direction, with the physical
bandwidth limit of 2 Gbps for unidirectional data rate. An "LC" optical connector attaches to
a fiber pair up to 200m of 50/125 multi-mode fiber.

2.1.6 Software layer

The software layer is assigned for providing an interface and the means for applications to
send data to each others. The software layer near to application is the interface layer and
supplies the library calls that applications can use. This layer is just a cover for the working
layer which is ADI (Abstract Description Interface), which will do the work in converting the
information passed by application in the messages that the transport layer below recognizes.
Transport layer function is to deliver messages to its recipients and includes the data link layer
and the physical layer.
MPI

Due to the enormous amount of different manufacturers offering their own communications interface, applications were programmed specifically for the target type of clusters. This fact reduced portability to cluster applications, so several research groups have been developing programming tools to ease the parallel programming task and to uniform the interface seen by applications. Therefore applications would run in any cluster environment, provided that it offers the same interface. Some of the most known parallel programming tools are PVM [13], P4, Express, Linda and MPI. MPI is the most used and this work is focused on it.

MPI is a standard specification for message passing communication libraries. Original MPI application programming interface (API) was specified by a group of over 80 people specialists in high performance computing, from universities, government laboratories and industry in a conference series during 1993 to 1994. Motivation for MPI development as the fact that each manufacturer of cluster or parallel processing systems were, at the time creating their own API, proprietary, for message passing. In this scenario, applications would not be portable from one system to another of a different manufacturer. MPI standard has been being reviewed since its original development and has now reached his 2.0 version.

MPI was designed with the purpose of being a standard specification for message passing that each manufacturer can implement in his system. In this way, the communications interface seen by applications will be the same whatever the manufacturer and system architecture are. Manufacturers of clusters always have high performance in mind, so fast communications abilities became an objective in MPI development. MPI standard has the following main characteristics:

- point-to-point largest communication set of library calls (by far the most complete library), such as send and receive;
- collective communication operations, for communications that involve several processes at once such as all-to-all and reduce operations;
- communication context that supplies support for the development of secure software libraries;
- possibility to specify communication topologies;
- possibility to extend normal data types and use them to describe messages of non contiguous data such as transfer of structures.

MPI interface, as seen by applications, was developed with the intend of gathering all the characteristics and behaviors of the message passing systems of the several variety of clusters and massively parallel processor(s) (MPP), so that programs work correctly in any type of system. Portability achieved by MPI means that a program written for an architecture can be copied for a second architecture, compiled and executed without changes in the code. However, standard does not guarantee the correct operation in the case of mixed architectures. This comes, mainly, from the priority given to system performance. The best way of creating a fast message passing library in a specific architecture is using communication methods for which hardware has native support. Normally, they are not compatible with other hardware systems.

It is supplied a high level of abstraction of computational resources in terms of communication topology. One example is that one process can be running in the room next door, connected by local network or it can be in the other side of the world, using internet to communicate. Hardware might be different, but it is compatible at the TCP/IP layer, allowing communication to proceed using this protocol.
A group of tasks can be organized in a specific interconnecting topology. Communication between task processes is then performed according to this new topology, which, in case of matching the physical supporting network, can improve and optimize the resources usage. Original MPI standard did not include any fault tolerance mechanism, although current MPI version 2.0 (MPI-2) does have an error notification scheme. MPI-2 includes new types of functionality:

- dynamic process: enabling the creation and termination of processes during application execution;
- one sided communications: enabling the possibility of interrupt driven receives, increasing programming opportunities;
- parallel I/O and many more.

In this work the version 1.2 of MPI standard was used, since MPI-2 is not yet common and well known.

In MPI-1, which is an alias for version 1.2 of MPI standard, task or processes and host computers are considered static. So tasks related to an application have to be initiated simultaneously. If one task or a resource fails, all application will fail and stop. The reasons for this static nature of things are based on performance as in commodity. As all tasks are always present, and also the communication links between them, there is no need for the long and time consuming group checks or domain name searches. Each task has at the starting point knowledge about all the others and so communications between them can be realized without the necessity of specific controlling process just for communications.

After this explanation, it is easy to understand why MPI implementation in a computer is structured in several layers and also depends on the hardware layer used to communicate. MPI can be split in two main layers, the interface (API) layer and the ADI layer, as depicted in Figure 5. The interface layer presents the MPI calls that programmers can use for message passing communications. As this interface is a standard, and there is a lot of documentation about how to use it [14], it presents easy-communication for the programmer that uses MPI.

<table>
<thead>
<tr>
<th>Application</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI Interface layer</td>
</tr>
<tr>
<td>Software</td>
<td>MPI ADI layer</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
</tr>
<tr>
<td>Hardware</td>
<td>Ethernet</td>
</tr>
</tbody>
</table>

**Figure 5: MPI layers interfacing program and hardware**

ADI layer is the one that is hardware dependent. There are several implementations of the MPI standard; two of the most used are MPICH [15] and MPI/LAM [16]. MPICH is a portable implementation of MPI for many environments of parallel computing, including from network of workstations to MPP. MPICH offers an ADI to be used over Ethernet, the *ch_p4*. This means that this specific MPI ADI was designed specially to be used over Ethernet.
NIC. MPI/LAM is a high quality implementation of the MPI standard, offering full support to MPI-1 and only partial support to MPI-2, and provides special features like check-pointing applications to disk and restart them at a later time.

Between ADI and the hardware itself it is the transport or link layer. This layer controls the network hardware directly and exchange data between the above layer, API, and the NIC, and its main purpose is to turn easy the interface to the hardware from above layer. This layer touches directly the hardware, influencing the communicating procedure itself, so it must be fast.

In this work two software libraries for message passing were used for analysis of the Maestro2 system: MMP, which is the developed messaging system specific to Maestro2 network hardware, and MPI, the standard message passing system.

**MMP**

To extract the performance of Maestro2 PC clusters, it has been designed and implemented a low latency and high bandwidth message passing library called MMP. MMP consists of a user level library and Maestro2 communication firmware, which exists on the network interface (NI) and switch box (SB). The two key observations in developing MMP were: first, in order for the communication not to degrade the overall performance of the parallel application, it needs to avoid unnecessary overhead; second, computation must have higher priority in accessing the CPU than communication.

In order to eliminate the data copy between user and kernel memory spaces and to eliminate issuing operating system primitives, addressing the first issue, a facility for a user memory mapped communication has been implemented. In a user memory mapped communication, the message buffer and the communication buffer managed by the network interface are mapped to the user's virtual address space. These areas are reserved as special virtual address areas, whose corresponding physical areas are never paged out, so that the PCI interface on the network interface can perform DMA. To achieve high performance communication, a user process directly accesses the network interface and polls the flag for message arrival without any system call or interrupts.

To deal with the second issue, MMP communication functions were made to be non-blocking. This allows for actual time at which the communication takes place to be flexible. In addition, complex communication operations such as the creation of data chunks, migrates to the physical network. Therefore, the communication code will be overlapped with the computation code on the host processor.

Data flow between nodes in MMP is described as follows, for message sending and receiving.

**Message sending:**
1) An application program generates a message and stores it in the message buffer allocated in the main memory of the sender node. Since the virtual address of the message buffer is mapped to the application process, the data is copied by the processor without issuing a kernel primitive.
2) The host issues a send request to the network interface.
3) The network interface dequeues the send request and invokes a DMA transfer from the message buffer to network buffer in the network interface.
4) The message is sent to the receiver node through the switchbox.

Note that the host will never wait for a completion of a send primitive. The host can continue the subsequent operation after issuing the send request. The network interface can perform
step 3) independently of the host. In case of a receive operation, issuing a receive request from the host and receiving data from network by the network interface can be performed in parallel. A message is received by the following steps.

Message Receiving:
1) NI reads a message header from the network buffer, and then allocates the area in the message buffer for the message body.
2) NI invokes the DMA function of the PCI interface to transfer the incoming message from the network buffer to the message buffer.
3) NI puts an entry, which includes the information about the incoming message, into the message management table. The entry consists of a message size, an identifier of the sender, and other required information.
4) NI dequeues a receive request issued by PC from the communication buffer. Then it will check whether the corresponding entry exists in the message management table. If it exists, NI passes it to PC.
5) PC extracts the entry from the communication buffer and copies data from the message buffer to the application- specified area.

A detailed description of MMP library functions can be found in [17] and also the detailed implementation of the entire message passing process, as sending and receiving data and creating and closing a connection between tasks.

2.2 Conclusions

For cluster computing to be portable and flexible, communications between hosts has a layered structure as seen in Figure 5 and, Figure 6. In those layers as we approach the top there is a standardization effort, so that applications are able to run on different cluster architectures but offering the same communications interface.

As we approach the lower layers they become hardware dependent and specific to the clusters NI architectures.
In the communication layers of clusters, there are two categories of performance characteristics:

![Cluster communication layers trend](image-url)
- Static performance: which is the one that it is intrinsic to the system, like, processor speed, DMA transfer speeds, which can obviously affect the communications procedures described above.
- Dynamic performance: this depends on the communication itself, it is very dependent on the application and the communicating strategy used. Dynamic performance is normally measured by how fast is the executing time of an application or how much data it can transmit to other hosts in the shortest time.

From the static analysis, it is possible to know the potential performance of the system, its maximum reached by hardware limit. Static analysis will also help in discovering problems, errors that might arise during development.

All the collected information about the system will be of major importance and very helpful for tuning of communications software layers, specially the transport layer software. From the dynamic analysis, it is possible to know characteristics for each communication of the application software. This information is very helpful for tuning the application messaging strategy such as the size of the chunk in the communications or the threshold for handshaking in MPI communications.

In this work these two strategies, static and dynamic analysis, were used to analyze the performance of Maestro2 cluster network. This report will describe the way to analyze each layer statically, focusing in the hardware layer, and dynamically in the software layer.

To the static analysis of the hardware layer, it was developed and implemented a debugging tool that would allow snap shooting of the NIC firmware status and also to make several tests to the physical system, such as DMA transfers.

In the dynamic analysis, known benchmarks applications were used to collect information about the system capabilities, to study the best approach to choose chunk sizes, buffer lengths, and also to measure protocol layers stacking overhead, information that is very useful for the programmers developing the protocol layer itself.
3 Debugging tool

3.1 Introduction

To accomplish the aims proposed, it was taken advantage of a very well-known standard debugging program, instead of building a new from the scratch. This approach benefits from the functionalities already existent in the program and also the future functionalities that can be added. The most important functionality is the possibility of remote debugging. Without this property, the debugging program would be of no interest.

The standard debugging program chosen was GNU ‘gdb’, a well known debugging program and Insight, which is a graphical version of ‘gdb’. As Insight is just the name of a graphical user interface, the debugging program will be referenced as ‘gdb’, as it is a more accurate and general name. ‘Gdb’ is considered a stable platform for the development of applications. Other reasons for its choice were that it has a big development team; it has available many graphic interfaces and has an extreme versatility of configuration and easiness to use. Due to the enormous amount of users and their feedback, it is practically a bug-free application, which is a much appreciated feature.

The GNU debugger, ‘gdb’, is an extremely powerful all-purpose debugger. Its text-based user interface can be used to debug programs written in C, C++, Pascal, FORTRAN, and several other languages, including the assembly language for all microprocessors that GNU supports.

Among ‘gdb’s many noteworthy features is its ability to debug programs “remotely,” in a setup where the platform running ‘gdb’ itself (the host) is connected to the platform running the application being debugged (the target) via a serial port, network connection, or some other support. This capability is not only essential when porting GNU tools to a new operating system or microprocessor, but it’s also useful for developers who need to debug an embedded system based on a processor that GNU already supports.

The remote debugging capability of ‘gdb’, when it has been properly integrated into an embedded system, allows a developer to step through code, set breakpoints, examine memory, and interact with the target in ways that rival the capabilities of most commercially available debugging kernels—and even some low-end emulators.

3.2 Debugging methods

In order to debug a program effectively, it is necessary to generate debugging information at the time of compilation. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request the insertion of the debugging information, it is necessary to specify the ‘-g’ option when the compiler is applied.
The GNU C compiler compiles C source in a `.c' file into assembly language in a `.s' file, which the assembler translates into a `.o' file and the linker combines with other `.o' files and libraries to produce an executable file. With the `-g' option, GCC puts in the `.s' file additional debugging information, which is slightly transformed by the assembler and linker, and carried through into the final executable. This debugging information describes features of the source file like line numbers, the types and scopes of variables, and function names, parameters and their scopes.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as stab (symbol table) directives, which are interspersed with the generated code. Stabs are the native format for debugging information in the a.out object file formats [18].

The assembler adds the information from stabs to the symbol information it places by default in the symbol table and the string table of the `.o' file it is building. The linker consolidates the `.o' files into one executable file, with one symbol table and one string table. Debuggers use the symbol and string tables in the executable as a source of debugging information about the program.

The native debugging method, which is also the most often one, occurs inside of a single machine. The machine is executing several processes, and among them there are the debugger process and the process being debugged. As separate processes they need to communicate, so that the debugging program can control and report the changes in the program being debugged. As they are running on the same machine the communication model involved uses operating system resources. Figure 7 depicts this model, with the operating system supporting the execution of both processes and the communication between them. The black arrow in Figure 7 means debug information flow, from interrupting signals to memory content dumps.

![Figure 7: Single machine debugging communication model](image)

In this debugging scheme, the program being debugged is usually launched as a child process from within the debugger process, ‘gdb’, allowing immediate creation of communication links. In this way, the debugger will also set the environment of the program to the user needs. Other native debugging scheme is debugging an already running program. For that, user specifies the intended process identification to ‘gdb’, which in turn will try to stop the process and inquire it status. Both schemes allow the use of ‘gdb’ commands and functionalities. ‘Gdb’ will use the OS to send signals in order to control the program flow.
When the debugger process and the program being debugged process are run in different machines, the communication channels changes to a level below the operating system, and the task is designated by remote debugging.

In remote debugging, it is necessary to configure the debugger application so that it understands that the process being debugged runs on a different machine (the target machine) and needs to know information on how to contact with that process. The remote debugging model can be described as in Figure 8, when both machines have an operating system and a physical link available between them.

![Figure 8: Remote machine debugging communication model](image)

For remote debugging, the program being debugged needs to be launched from a smaller version of the debugging application, which will support/provide the communication between the process being debugged and the host debugging process. This scheme normally occurs when the machine running the program to be debugged lacks the means to run the full debugger program.

Remote debugging is also applied when debugging an embedded system or when the target machine does not support running processes. For example, remote debugging might be used on an operating system kernel, or on a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger.

### 3.3 The debugging architecture and implementation aspects

In Maestro2, this work environment, the NI is a stand-alone device, which lacks an operating system supporting communication. Therefore, the particular communication model adopted is depicted in Figure 9.

As referred in a previous chapter and depicted in Figure 10, the physical connection between the NI and PC is the PCI bus. At the logical level, the access is obtained using a character device driver. Programs in the PC can use it to download new memory images to be run in the NI. The logical model for this connection is shown in Figure 11. On the NI side communication are handled by specific libraries existent in the downloaded software. Monitoring programs on the PC carry out the basic setup procedures for the NI. They are used to download the memory images.
In this debugging architecture, where there is no operating system to support communication between debugger process and program being debugged and they are run on different processor types, it is required to supply to the debugging program the type of target processor were debugging is going to take place.

### 3.4 Developed software on the PC

In this development stage, operating system used in the host PCs for NI(s) is Linux, as is a friendly environment for development of hardware drivers. Linux is an open source project and anyone can see its internals and change them to his pleasure.
The character device driver [19] used to establish communication with the network interface is of the serial type. It provides FIFO functionality to programs that use it. Before it is launched, a special file node situated in /dev/maestro2_char0 is created. When the driver is loaded, this file can be opened, written to or read from by any application. This file is a device handler for the NI; it is a virtual file to access the device from the user application. Via this file user application can be allowed to access the device. It has been designed so that character written to it is passed directly to the NI, where the communication libraries will pick it up. Multiple programs can open it for reading as for writing, but the driver does not recognize them individually. This means that we can have more than one program sending information to the NI, but only one at a time reading data from the driver. If more than one program has the driver open for reading, the data is distributed by the driver to the open applications without any predefined order, effectively garbling the information.

When a program executes a read from the file descriptor of the driver, the driver will supply the next char available sent by the communication libraries of the NI. In both communication ways, the receiving and the sending one, the FIFOs buffers are 256 characters long. If the receiving FIFO buffer, as seen in Figure 12, is full and the program running in the NI wants to send more characters, the program running on the NI will stop in an active waiting loop for the character to be read. Character will be read from the NI into the FIFO buffer only when there is free space available for it. Reading from the file descriptor of the character device driver will return the next character in the FIFO buffer, creating an available position. If the receiving FIFO buffer is empty and there are programs trying to read from the character device file descriptor, as there is no char to give to them, they will enter also in a waiting state. They will be waken by the operating system kernel when there are characters again in the FIFO buffer to be distributed.

![Figure 12: Character device driver logical model](image)

On the PC side it was necessary to install and configure 'gdb' application. The most recent version of 'gdb' was downloaded from GNU website [20]. The download package contains the source files that need to be compiled to generate the executable file. As 'gdb' has to be informed of the type of debugging required, this is done by configuring the compilation scripts prior to compilation.

To our particular case, the general purpose microprocessor used in the NI is a PowerPC603e [21]. So, the configuration of ‘gdb’ to work with this target machine is done by supplying the parameter –target=powerpc-none-elf to the configuration command. This indicates that the executable files opened by the debugging program should comply with the elf file format and will be executed in a PowerPC microprocessor. No more configurations are required and
Compilation will generate an executable file prepared to debug programs executed in the PowerPC in the NI.

Finally, so we can use ‘gdb’ to debug programs being run on the NI microprocessor, it is necessary to configure the communication. As referred previously, ‘gdb’ comes prepared to execute remote debugging on machines that do not have an operating system to support communication. In this cases, the communication is performed by using the standard ‘gdb’ remote serial protocol [22], between ‘gdb’ and a special routine inserted in the NI program called stub. The stub routine running in the NI will be responsible for handling the requests made by the debugging application ‘gdb’, and to inform it of the current execution status in the NI.

The ‘gdb’ Remote Serial Protocol (RSP) is a simple, ASCII message-based protocol suitable for use on serial lines, local area networks, or just about any other communications medium that can support at least half-duplex data exchange. RSP packets begin with a dollar sign ($), followed by one or more ASCII bytes that make up the message being sent, and end with a pound sign (#) and two ASCII hex characters representing the message's checksum. For example, the following is a complete RSP packet:

$m4015bc,2#5a

The receiver of the packet responds immediately with either a "+" or a "-" to indicate that the message was received either intact or in error, respectively. A typical transaction involves ‘gdb’ issuing a command to a debugging target, which then responds with data, a simple acknowledgement, or a target-specific error code. If the latter is returned, ‘gdb’ will report the code to the user and halt whatever activity is currently in progress. The console output message, which debugging targets use to print text on the ‘gdb’ console, is the lone exception to the typical command-response sequence. Except when another command is already in progress, this message can be sent from the debugging stub to ‘gdb’ at any time.

‘Gdb’ is prepared to use this protocol in communications with targets via the serial port. Instead of using the serial port, is necessary to configure it to use the character device driver. This is accomplished by typing in ‘gdb’ prompt the command:

`target remote /dev/maestro2_char0`

Indicating the file descriptor of the character device driver to ‘gdb’, the file node will be open for read/write by ‘gdb’ which in turn will start immediately inquiring the status of the program being debugged.

For the special file node to be considered as a serial port, a change was performed at the control level of the module, so that ‘gdb’ would identify the special file node as a serial port. The change consisted in modifying the values returned by the device driver on the response to I/O controls performed on the special file node.

‘Monitor’ application program executed in the PC allows the download of new memory images to the NI, and also serves as standard input of the program being executed in the NI. Applying the Unix ‘cat’ command to the character device driver file works as standard output for the program running in the NI. The ‘Monitor’ application serves to start and stop the execution of the downloaded program to the NI. The trick used is to enable/disable the bus.
granted signal to the PowerPC microprocessor. ‘Monitor’ also allows the screening and changing of the contents in the PCI registers of the NI and special control registers mapped in the NI FPGA.

As now the character device driver is required for the debugging communication, ‘cat’ application cannot be used to watch standard output from the NI. As the standard output is necessary, it is required to change the printing messages in the NI program to comply with specific ‘gdb’ functionality, console output message. During the compilation of the programs to be run on the NI, if the debug flag is set, printing messages are replaced by a code that sends the same message, but as packets of information to ‘gdb’. These special packets are marked as output information and when ‘gdb’ receives them it just shows the correct message information in is standard output for the user to see it.

On the way around, from PC to the NI, there is no such problem, because there is only one recipient for the information written to the driver. ‘Monitor’ application can still be used to send standard input to NI.

### 3.5 Developed software on the NI

The ‘gdb’ requests handler, also know as the stub, works under ‘gdb’ command and is responsible for supplying all the information requested by ‘gdb’ and to carry out his commands. Stub consists in one main routine designed to wait for and to respond to requests coming from the debugging application, and in several other subroutines that carry out those requests. ‘Gdb’ package contains stubs for various specific types of machines, but unfortunately, PowerPC is not one of them. So, it was necessary to port one of the existent ones to our case. It was also necessary to change the exception handlers code.

Stub subroutines do all the work required by ‘gdb’, from decoding a serial protocol packet coming from ‘gdb’ to coding memory contents to a packet. The most important ones, shown in Table 1, are both the ones that allow the access to the memory contents and also the ones that provide communication with ‘gdb’.

<table>
<thead>
<tr>
<th>Subroutine name</th>
<th>Subroutine purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>putpacket()</td>
<td>Send buffer to be transmitted to the ‘gdb’ application</td>
</tr>
<tr>
<td>getpacket()</td>
<td>Receive transmitted buffer by ‘gdb’ application</td>
</tr>
<tr>
<td>mem2hex()</td>
<td>Converts the indicated memory area binary contents to corresponding hexadecimal values and place them in a buffer to be transmitted</td>
</tr>
<tr>
<td>hex2mem()</td>
<td>Converts hexadecimal buffer contents to corresponding binary values and place them in the indicated memory area</td>
</tr>
<tr>
<td>computeSignal</td>
<td>Attempts to translate the exception code in Unix compatible SO signal type</td>
</tr>
</tbody>
</table>
The memory access is granted by means of the translation of the command received, read or write. If ‘gdb’ wants to know the value of a specific variable, it will translate the variable name in its address using the symbol table, creates a read memory packed with the address and length to be read, and send it to the ‘stub’. The stub will decode the packet, passes the address to the mem2hex() subroutine that will convert each byte of that zone in two characters representing the byte value in hexadecimal format, and places the conversion result in a buffer. Stub main code will then pass that buffer to the putpacket() subroutine that will encapsulate it and add the checksum code forming a packet that is sent to ‘gdb’. Putpacket() will only return after the reception of positive acknowledgment from ‘gdb’.

The main changes done to the stub code were at the level of specifying PowerPC processor registers and rearranging the stub subroutine calls managing input and output to use the special communication libraries. When the stub uses these communication subroutines, it interfaces with the character device driver, which in its turn interfaces with ‘gdb’. This forms the logical communication model presented in Figure 13.

![Figure 13: Communication scheme from NI perspective](image)

Communication interface with the character device driver and the NI is based on active waiting loops. When the program running in the NI wants to send a character, writes it to a special memory position dedicated to that purpose if it was empty. If it is no empty the program will wait until it is empty. That special memory position is said to be empty when its content is ASCII null char. When the program writes to that memory position, a PCI bus interrupt request is generated from the NI, which will activate the device driver to come and pick up the character. After placing the character in the receiving FIFO buffer, it clears the special memory position.

Sending a character from the PC to the NI program works the same way, but in the reverse direction. When a program successfully writes a character to the device driver, the driver keeps it in the sending FIFO buffer until the special memory position reserved for receiving characters from the PC is clear. When the memory position is clear, the driver writes the character to there, removing it from the FIFO buffer and do not care about it anymore. The program running in the NI, when reading a character from that memory position, retrieves the one that is there or will wait until there is a char present in that memory position. After copying the character for itself, it clears the contents of the receiving memory position, making it available for a new input. Due to this type of protocol, the ASCII null char has to be avoided in the communication.
The debugging process is based in the existence of exceptions. Without them it would be impossible to use this system to debug errors and trace the program execution. When 'special' exceptions occur, listed in Table 2, during normal program execution, the control is passed to the stub that inform 'gdb' of the exception and were it happened in program flow. During this process of transferring control, all possible information about the current status of execution can and has to be saved.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrementer</td>
<td>Occurs at predefined time intervals depending on the contents of the decrements register.</td>
</tr>
<tr>
<td>Program exception</td>
<td>Occurs when a breakpoint is executed or is caused by a problem with the instruction being executed.</td>
</tr>
<tr>
<td>Trace</td>
<td>Occurs after the execution of an instruction when the Trace bit of the processor status register is in the set state.</td>
</tr>
</tbody>
</table>

To save this information there is an array declared inside the stub that needs to be filled with the correct values of the microprocessor registers in the exact moment of the exception. For this to be accomplished it was created the code of the exceptions handlers that will call the stub and insert there the code that would fill the array with those values.

Due to the fact that debugging only starts when 'gdb' and stub are communicating, it became necessary to force the execution of the stub before the execution of the program to be debugged. This was achieved by calling a subroutine, immediately after the start of the main program, which will generate an exception. This way it is guaranteed that program control passes by the stub and stops there until 'gdb' gets in contact with it, preventing the precocious run of the main program.

There was also developed several extra functionalities inside the program run in the NI. These functionalities represent new commands available inside 'gdb'. Such new functionalities consist in more control of the execution flow.

### 3.6 Conclusion

The implementation of a debugging tool to help the development of the code being run on the NI microprocessor and perform static analysis was concluded with success. After its implementation, several tests were made to check the correctness of the overall system. With this debugging tool implemented it is now possible to better control the execution flow of the code being run on the NI microprocessor, resulting in a faster and better development of new programs to the NI. Using the debugging tool is now possible to regain control of programs when they were dead locked, watch correct transcriptions of the memory contents, pause the execution of a program using breakpoints, changing the context or the flow of execution while the program is being run and follow the correct execution of the code with the help of the graphical interface.

Problems were faced at the host response time to the output from the debugging tool to 'gdb'. The use of the GUI was some times impossible due to the freeze of the application when processing much debug information, forcing the use of the console version.
4 Performance evaluation

4.1 Introduction

In this chapter it is described in detail all the relevant aspects concerning the experiments realized to evaluate the performance of the Maestro2 system and the comparative results obtained using a commercial solution available on the market, i.e. Myrinet.

Benchmarks for cluster systems exist in an enormous variety of flavors; there are some to benchmark the whole system, and some for particular components like the communication network. To measure the dynamic performance of a cluster at the communication level, one must use a communications specific benchmark.

The network benchmark chosen is a well known and versatile one that was used to extract communication characteristics from the architectures under analysis. It has a same standard interface to different level architectures. Comparing the results from running the benchmark over different conditions makes no sense, such as a comparison of TCP/IP results over Ethernet with the one of ATM software on top of ATM hardware. There has to be the same conditions, like, if purpose is to compare the hardware in the example above, it would be necessary to make an IP layer to ATM, to the systems became comparable, layer by layer. To compare the software layer, it would be necessary to use ATM over Ethernet hardware. This way TCP/IP layer would be comparable to the ATM software layer.

Resuming, when the purpose is to analyze and compare the performance of a component in two different architectures, every component not directly involved should be similar in both cases.

Using a benchmark that offer access to measure the performance across several protocols means that we are using the same procedure in all experiments, leveling the different architectures and making equal the benchmark intrinsic overhead.

As in this work there is the need to analyze several different architectures and protocols, NetPIPE [23] benchmark application was used due to its enormous versatilty to run with many different protocols. Using the same application to the entire tests was an important factor, to disable biased analysis due to the use of specialized benchmarking applications for specific hardware. Using the same application, it implicates that the same routines are run the same way, and that the communication part is really the one that changes.

4.2 Benchmark tool

In recent years, much research has been directed towards evaluating the performance of high speed networks [24, 25]. The design of NetPIPE, a Network Protocol Independent Performance Evaluator, has been motivated by the need to assess the performance of communication bound applications. NetPIPE helps answer questions that surround network communications inherent to these applications. These applications include file transfer, message passing and graphical simulations for display in a virtual reality environment, such as CAVE [25] applications, which require frame transfers from a compute server. While file transfer applications allow streaming of data, a graphical simulation requires blocks of data to
be transmitted at regular intervals to maintain full-motion video. The size of each block and the number of frames per second are enough to specify a minimum network throughput required to maintain realistic animation.

With the applications in mind, several questions can be asked in reference to the network communication. For instance, how soon will a given data block of size \( k \) arrive at its destination? Which network and protocol will transmit size \( k \) blocks the fastest? What is a given network's effective maximum throughput and saturation level? Does exist a block size \( k \) for which the throughput is maximized? How much communication overhead is due to the network communication protocol layers? How quickly will a small (< 1 Kbyte) control message arrive, and which network and protocol are best for this purpose? These questions are very important when talking about message passing communication strategies.

The answers to such questions are not always straightforward and easy to obtain using other network performance tools. The two most popular tools, ttcp [25] and netperf [26], are based on the TCP/IP [26] communications protocol. While netperf has the ability to map network performance, comparing network protocols with these tools is difficult if not impossible. Finding the effective maximum bandwidth using ttcp is an exercise in delving into protocol internals. Knowledge of the appropriate buffer size, alignment address, and protocol settings is required to achieve data transfer at the effective maximum bandwidth.

NetPIPE consists of two parts: a protocol independent driver, and a protocol specific communication section. The communication section contains the necessary functions to establish a connection, send and receive data, and close a connection. This part is different for each protocol. However, the interface between the driver and protocol module remains the same. Therefore, the driver does not have to be altered in order to change communication protocols. The target protocol is specified at compilation time with an argument to the `make` command.

The driver is based on the principles presented by the HINT [27] computer performance metric. Just as a computer's performance cannot be accurately described using a single sized computation, neither can the performance of a network be described using a single sized communication transfer. NetPIPE increases the transfer block size \( k \) from a single byte until transmission time exceeds 1 second, to achieve a variable time benchmark and to scale to all network speeds.

For each block size \( c \), three measurements are taken: \( c - p \) bytes, \( c \) bytes, and \( c + p \) bytes, where \( p \) is a perturbation factor with a default value of 3. This perturbation allows analysis of block sizes that are possibly slightly smaller or larger than an internal network buffer. For each measurement, NetPIPE uses the following algorithm, depicted in Figure 14.

The variable `nrepeat` is calculated based on the time of the last data transfer. The intent is to repeat the experiment enough times such that the total time for the experiment is far greater than timer resolution. The default target time is 0.5 seconds. For most modern computers, this provides a sufficiently precise data transfer time.

NetPIPE uses a ping-pong transfer like Hockney [28] uses for each block size. This forces the network to transmit just the data block without streaming other data blocks in with the message. The result is the transfer time of a single block, thus providing the information necessary to answer which block size is best, or what is the throughput given a block of size \( k \).
/* First set T to a very large time. */
T = MAXTIME
For i = 1 to NTRIALS

t0 = Time()
For j = 1 to nrepeat
  if I am transmitter
    Send data block of size c
    Recv data block of size c
  else
    Recv data block of size c
    Send data block of size c
  endif
endFor

/* Insure we keep the shortest trial time. */
T = MIN(T, t1-t0)

endFor
T = T/(2 * nrepeat)

Figure 14: NetPIPE benchmark algorithm

NetPIPE produces a file that contains the transfer time, throughput and block size for each data point. For instance, Figure 15 presents the throughput versus the transferred block size for a typical Ethernet link. This graph is referred to as the throughput graph. From this graph, it is easy to see that the maximum throughput for this network is approximately 85 Mbps. However, it is difficult to analyze the latency, a very important statistic too.

Figure 15: Example of a Fast Ethernet Throughput
A graph that is easier to read and analyze is the network signature graph. One such graph is shown in Figure 16. It depicts the transfer speed versus the elapsed time; hence it represents a network "acceleration" graph. This graph is a simple way of viewing network performance data; the key is to use a logarithmic time scale horizontally instead of the transfer block size. In this graph, as in all graphs presented, time is plotted in seconds. It is very similar to the way computer performance is presented by the HINT performance metric. Although unconventional, this graph represents perhaps a better approach to visualizing network performance. All the necessary data are clearly visible and easy to extrapolate. The network latency coincides with the time of the first data point on the graph. The maximum attainable throughput is clearly shown as the maximum point on the graph.

![Network signature graph](image)

**Figure 16: Example of a Fast Ethernet Signature Graph**

Plotting the block size versus the transfer time on a logarithmic scale for both the x and y axis, as in Figure 17, reveals what we define as the saturation point. This is the point after which an increase in block size results in a near-linear increase in transfer time, effectively the knee of the curve. The time interval between the saturation point and the end of the recorded data is referred to as the saturation interval. In this interval, the graph monotonically increases at a constant rate i.e., the network throughput cannot be improved upon by increasing the block size.

By using the maximum effective bandwidth to compare networks (or even worse, nominal bandwidth) is much like using peak megahertz ratings to compare computers. While it may be correct for ranking certain applications, in general, its accuracy leaves much to be desired. A
given network may have a high maximum effective bandwidth but also have a high latency. So a network with a lower latency would possibly be better for small messages even though it has a much lower maximum effective bandwidth. This effect can be readily observed when comparing ATM with Ethernet, as shown below. Any ranking based on a single number does not provide sufficient insight for accurate network comparison. For network tuning and comparison, it is advisable to take the entire NetPIPE signature graph together with application specific information into consideration.

![Network saturation graph](image)

**Figure 17: Example of a Fast Ethernet Saturation Graph**

### 4.3 Experimental setup

The experimental results described in this chapter were obtained using two different sets of hardware, since it was not possible to acquire similar hardware for all research groups involved in this project.

The hardware used for the Myrinet network performance tests is better than the one used to obtain the results of Maestro2, which will bias the Myrinet results as will be shown in the next sections.

The hardware of the host machine used for the Myrinet network interface consists in a Dual 1.4GHz Intel Pentium-4 Xeon processors in a Intel 860 chipset motherboard with PCI bus slot
of 64 bits, working at 66 MHz. Each host uses 512 Mbytes of RDRAM PC800 for main memory. The Myrinet network interface card used in each node was a M3F-PCI64B-2 [29].

The hardware used in the host machine for the Maestro2 and Fast Ethernet network interfaces consists in DUAL 1.0GHz Intel Pentium-3 with Serverworks HE-SL chipset motherboard with a PCI bus of 64 bits working at 66 MHz, and using 512 Mbytes of SDRAM PC133 as main memory. The Maestro2 Network interface card used was a Y-PCI64-P [1].

The Operating System used for all experiments was Linux 2.4.7-10smp.

### 4.4 Experimental Results

This section describes the experiments performed to measure the dynamic performance of the communication systems, when used together with communication libraries.

For all the measurements it was used the Ping-Pong strategy, which consists in the measurement of the time a packet of $S$ size takes to travel from host A to host B and back to host A, defined as the *round-trip* time. The packet size varies between a minimum of 1 byte to a maximum of 8.388.608 bytes (8Mbytes).

The following text shows the categorization of the experiments:

**Experiment 1** is focused on protocol stacking overhead analysis. This experiment has taken place to allow a better perception on how problems can cascade from lower layers to upper layers when stacking communication protocol layers. For that, each protocol analysis on this experiment is performed looking to each architecture independently, forming subexperiments (A) and (B) and in each, results are categorized in three main points: First point show the results of the throughput for the different protocols, focusing special points noticed and causes involved. In second it is shown the results of network signature analysis, were latency is taken into consideration. Third and last point refers to network saturation point analysis.

SubExperiment (A) focused on the Myrinet architecture, and performed several tests to the most important communication protocols existent on that architecture, performing an extensive and separated throughput analysis and latency analysis to each:
- GM, special message passing system of Myrinet network, performance characteristic was measured and analyzed, for realizing the minimum latency achievable by this message passing and also the highest throughput of the system.
- TCP/IP over Myrinet performance was measured and analyzed, to retrieve its throughput characteristic and minimum latency and to conclude the overhead imposed above the lower layer, GM.
- MPI performance was measured and analyzed to provide information of the minimum latency and maximum throughput offered to applications that use it.

Subexperiment (B) was performed on Maestro2 Network and consisted on extracting the performance characteristics of the several supported protocols on two different configurations, first an Ethernet configuration:
- TCP/IP over Maestro2, running as lower communication layer over an Ethernet compatible configuration of the hardware
- MPI over TCP/IP, to observe how MPICH can take advantage of the performance offered by TCP/IP
Then the special configuration specific for Maestro2 network was loaded and were taken the results of:
- MMP over Mestro2, to observe the performance of the lower communication layer offered by Maestro2 system.
- MPI over MMP, to extract the performance characteristic offered to applications that use MPI.

The analyses of the experiments were made extensive to detect and show strange behaviors in each protocol communication, like abnormal performance degradation for specific packets sizes.

Experiment 2 consists on the direct comparison between the same protocols for the different architectures. So this experiment is categorized for each similar protocol and performance comparisons were performed at the following levels:
- TCP/IP implementation on both architectures
- Specialized communication software for both architectures, GM for Myrinet and MMP for Maestro2
- MPI implementation overall system performance, the one that is offered to MPI applications.

With this experiment, it is shown the differences in the performance offered by the two systems in each layer.

The Experiment 3 measures MPI performance over Maestro2 using two different configurations. It is intended with this experiment to show that performance varies according with the change in the internal configuration of the MPI implementations.

In all cases, the metrics measured were latency and throughput. Latency is calculated by dividing by two the round-trip time for small messages. Throughput is calculated by dividing the packet size by the round-trip time. NetPIPE application does these calculations automatically supplying these values in an output file, each line containing the message size, the throughput, and half the round-trip time for that message size, when in Ping-Pong mode.

**4.4.1 Experiment 1 – Software communication libraries overhead analysis**

This experiment consists on several runs of the NetPIPE benchmark using each time a different protocol supported by the cluster system and underlying network, to compare the performance of Myrinet and Maestro2 at different levels of software overhead.
The objective is to analyze the information provided from the run from each layer and analyze possible potential problems in communication system.
As also the protocol layers are normally implemented one on top of another it is possible to conclude, when crossing the plots from the different protocols of the same architecture, the performance degradation due to the protocol software.
(A) Myrinet

(1) Throughput Analysis

In Figure 18 it is plotted the throughput versus the transfer block size curves of the three different protocols, and were obtained using NetPIPE with off-the-shelf configuration.

Analyzing the average of the GM curve, it is observable that bandwidth increases until a maximum of approximately 1300 Mbit/s. For block sizes of 4096 Bytes and larger it can be seen a teeth shaped graph, due to the visual effect of the union of consecutive points by the plotting tool. Vertical fall lines mark the place of the normal measurement, central to the ones that differ by the increment. In all these curves the increment value used was 3 (Bytes). Graph shows this shape because the result of the measurements for block sizes of central and central less increment gives a high bandwidth, but for the measurement of block size of central plus increment there is 8% decrease in the result.

As an example, for a message block with 16384 bytes, the bandwidth observed is 987 Mbit/s, but increasing this block size by only 3 bytes, the bandwidth observed drops to only 900 Mbit/s.

The drop happens when the size of the block, due to the perturbation increment, exceeds a Page size, having the host to “allocate” one more page.
As a result of this analysis, applications using GM as communication layer should align their messages to the page size, in order to improve performance, for messages with size larger than 4KBytes (page size for most Linux systems).

Figure 18 also shows the TCP/IP protocol layer results. As it runs over GM layer it has a poorer result than GM due to TCP/IP operations such as window operations, fragmentations and copy operations between IP and TCP, but as it work on a different level, it is not affected by the gap differences on the GM curve.

MPI layer plot on Figure 18 shows a curious bandwidth curve filled with two special cases of interest, the spikes and the bump. Spikes begin to appear to message sizes bigger than 1024 bytes, but are very notorious in the interval between 32 KBytes and 128 KBytes. The graph shows the spikes because the bandwidth values for both of the measurements with perturbation are much lower than the central measured bandwidth. A case example, at the message size of 32 KBytes, the result bandwidth is approximately 670 Mbit/s but the bandwidth measured for the messages with perturbation do not go better than 567 Mbit/s.

The believed explanation for this is that MPI suffer a lot with packet size misalignment and when the size of the message is not a multiple of the internal MPI/LAM buffer sizes, performance degrades, and in this case, performance degradation reaches 15%. Normal MPI implementation possesses many internal memory buffers to perform send and receive operations, and the size of these buffers can be defined at the compilation time of the MPI package.

Another special point of interest is the performance degradation after message size reaches 128 KBytes. The expected graph curve should have the bandwidth points at the same level for messages bigger than 64 KBytes. The most likely reason for this to happen is that above 64KBytes, MPI starts using a large message rendezvous mode that requires a handshake before data is sent, so after that point there is no more improvement in bandwidth performance. Might also be because MPI/Lam is on top of TCP layer that has a limitation for transmission when the packet size is 64KBytes or more: after 64KBytes, TCP will have much flow processes to achieve the order of the packets. So the main reason for performance degradation is the fragmentation and window processing.

As a result to the analysis of this curve, programmers of applications running on top of MPI should align the packets or use aligned packets as much as possible.

(2) Network signature analysis

Next figures will take time in consideration.

Figure 19 shows the signature graph for the three protocols. As already stated this graph allows the detection of idiosyncrasies in the communication system. As it is not a univalued function of time, it is possible the turn to the left of the spikes in the MPI curve.

As already mentioned, this type of graphic allows the immediate perception of the two most important measurements, minimum latency and maximum bandwidth. Concerning latency, given by the time value for the first points in each curve, the following can be stated: for GM, minimum latency observed is approximately 10 microseconds, with a maximum bandwidth peak of 1350 Mbit/s for message sizes of 8 MBytes and bigger.
About TCP, it presents minimum latency of approximately 38 microseconds and a maximum reachable bandwidth of 1060 Mbit/s.

On MPI curve, spikes are slightly turned to the left meaning that the bigger packets taken less time to be transmitted than smaller ones, representing a bigger bandwidth associated to the measurement of the point coinciding with the peak of the spike, in relation to the points coinciding with the base of the spike.

![Network signature graph](image)

Figure 19: Myrinet network supported protocols signature graph

MPI latency is of approximately 48 microseconds, which is a reasonable value due to the layer stacking.

In regarding to bandwidth, as MPI does not present a standard curve as the other two, its value is different from the maximum peak observed in the plot of Figure 18. Respecting the bandwidth definition that is defined as the maximum throughput for very large messages, it states that MPI bandwidth is 650 Mbit/s, for aligned messages.

(3) Network saturation analysis

For the saturation graph in Figure 20, saturation points for each curve will be spotted, considered as the point when the line starts to have an approximate linear increase. In these graphs, that point does not correspond to the knee of the curves, but the knee of the curves is still a good approximation. For GM, saturation occurs for messages bigger than 64KBytes and for TCP saturations happens at a block size of 128 KBytes. On Figure 18 This points represents when the curve enters the 90-95% value of the final bandwidth.
For MPI and due to its strange behavior, saturation points occur after the performance peak, at 512 KBytes.

![Network saturation graph](image)

**Figure 20: Myrinet network supported protocols saturation graph**

One of the advantages of this type of plot is that it shows how small packets behave and until which packet size we can expect a transmit time similar to the latency. This information is particularly important when deciding about control message sizes, which normally are small. GM packets until 32 Bytes have the same order of magnitude than latency, the same happens for packets until 64 Bytes on TCP layer and on MPI to packets until 128 Bytes.

**(B) Maestro2**

To compare the performance of Maestro2 at different levels of software overhead, NetPIPE was run using each time a different protocol module and the plotted results are shown in Figure 21, Figure 22 and Figure 23.

The experiment involved two separate runs, on the first one an Ethernet-like firmware and corresponding driver were loaded in the Maestro2 cluster. Using that software and hardware two measurements were taken using NetPIPE: Maestro2 TCP characteristic and using MPICH message passing library, the MPICH over TCP characteristic.

On a second run, the special communication software libraries and firmware were loaded, and using NetPIPE measurements were taken of raw MMP performance and also from the special MPI implementation over MMP, currently under development. Both run results are plotted in Figure 21, Figure 22 and Figure 23.
(1) Throughput analysis

Figure 21 presents throughput versus the transfer block size for the four protocols. By beginning to analyze the TCP performance, as the lowest layer for the Ethernet firmware, it offers a maximum bandwidth of approximately 1000 Mbps. The dip at 64KB-128KBytes is of special interest and is caused by the limitation in the transmission size when it is 64 KBytes or greater. When dealing with packets bigger than 64 KBytes, TCP has much work to rearrange the order of packets. And also, this is the limitation of IP packet size after which causes copy operation(s) to TCP layer.

About the MPICH plot, as it runs over TCP it suffers from its ups and downs. The maximum bandwidth for MPICH over TCP is at the same packet size than for TCP. After packet size of 128 KBytes, MPI becomes stagnated with small variations, exposing a routing or pack regrouping overhead problem.

The MPICH dip at 64KBytes it is influenced by the TCP dip but mainly for the MPI threshold, resulting in a bandwidth of 450 Mbit/s for transfers greater than 128KBytes.

So the most striking aspect of these two curves is the impact of the MPICH software overhead that leads to performance decrease greater than 50%.

By analyzing the MMP curve, it presents a maximum bandwidth of approximately 2000Mbit/s. The teeth shape part observed for packets bigger than 4KBytes, is due to the size of the block to be transferred exceeds the size of a page, having the firmware to execute more
operations to complete the DMA transfer. That difference is amortized for larger messages transfers.
There is also a slight decrease on the performance after the peak of 1875 Mbit/s at 512KB-2MB, but since the difference between the peak and the bandwidth value (1860Mbit/s) is only 0.8%, it is ignored.

The MPI over MMP, is a currently under development layer, so this graph shows preliminary results and problems that need to be solved. There are two main dips in the MPI over MMP curve. The first one at 2 KBytes and the second is at 4KBytes block sizes. Both dips are explained in another section of this chapter. Even so, after both dips the MPI over MMP curve recovers to normal values and offer a maximum bandwidth of 1560 Mbit/s. It corresponds to a 20% loss regarding to the bandwidth offered by the layer below (MMP).

(2) Signature graph analysis

In Figure 22, the teeth like shape on the MMP curve is more noticed as well as the immense time gap between MMP and MPI over MMP. It is this excessive time loss that makes bandwidth to reduce. For an easy reference, latencies are shown in Table 3.

![Network signature graph](image)

There is an increase in latency for both MPI implementations of 36%. On strict comparing the MPI implementations, the MPICH over MMP is 3.6 times faster than MPICH over TCP, in terms of latency.
Table 3: Latencies observed on Maestro2 system

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMP</td>
<td>11</td>
</tr>
<tr>
<td>MPICH over MMP</td>
<td>15</td>
</tr>
<tr>
<td>TCP</td>
<td>40</td>
</tr>
<tr>
<td>MPICH over TCP</td>
<td>54</td>
</tr>
</tbody>
</table>

(3) Saturation graph analysis

On Figure 23, the MPI over MMP temporal problem is even more noticed. Doubtless, the MPICH over MMP software layer should be improved. The maximum packet size that still has near latency time is of 192 Bytes for MMP, 256 Bytes for MPI over MMP, 512 Bytes for TCP and 32 Bytes for MPICH over TCP.

Figure 23: Maestro2 network supported protocols saturation graph

4.4.2 Experiment 2 - Comparative results of the protocol performance

In this experiment it will be compared the performance of the same or equivalent layers from both architectures Maestro2 and Myrinet, using NetPIPE results.
(1) TCP/IP layer:

Figure 24, Figure 25 and Figure 26 clearly state that TCP performance for both architectures is almost the same, with the distinction that Myrinet results are slightly biased due to the experimental environment in which they were obtained.

Nevertheless, from the user application point of view, if for an application the main communication message sizes were less than 64 KBytes it would be recommended the use of Maestro2, due to the bigger bandwidth with the same latency. If application mainly uses bigger message sizes >64 KBytes, the use of Myrinet would be recommended.

(2) Lowest communication layer

Figure 27 compares raw performance of the special communication libraries of Maestro2 and Myrinet. Curves in Figure 28 show that latencies are virtually the same for both technologies. From Figure 27 it can be stated that, in average terms, Maestro2 MMP throughput is 45% higher than Myrinet.
Figure 25: TCP/IP signature graph

Figure 26: TCP/IP protocol saturation graph
Figure 27: MMP and GM throughput graph

Figure 28: MMP and GM saturation graph
Figure 29, Figure 30 and Figure 31 compare MPI performance on both systems. If there was no dip in Maestro2 curve it would be clearly the better one for any block size. But due to the dip, caution is needed. The only plausible conclusion is that the nominal bandwidth offered at the moment by MPI over MMP exceeds MPI/LAM over GM throughput by almost 200%. MPI/LAM bandwidth is only on average 600Mbit/s where as MPICH over MMP bandwidth is approximately 1560Mbit/s. Precise measured values can be seen in Table 4. Maestro2 also provides better (smaller) latency for messages with size smaller than 1KByte.

![Figure 29: MPI throughput graph](image)

<table>
<thead>
<tr>
<th>Packet size</th>
<th>Maestro2</th>
<th>Myrinet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>½ round trip time µs</td>
<td>Bandwidth Mbits/s</td>
</tr>
<tr>
<td>32</td>
<td>15,26</td>
<td>16,00</td>
</tr>
<tr>
<td>64</td>
<td>15,38</td>
<td>31,75</td>
</tr>
<tr>
<td>128</td>
<td>15,38</td>
<td>63,50</td>
</tr>
<tr>
<td>256</td>
<td>16,56</td>
<td>117,94</td>
</tr>
<tr>
<td>512</td>
<td>19,58</td>
<td>199,51</td>
</tr>
<tr>
<td>1024</td>
<td>23,11</td>
<td>338,12</td>
</tr>
<tr>
<td>1048576</td>
<td>5.506,56</td>
<td>1.452,81</td>
</tr>
<tr>
<td>2097152</td>
<td>10.584,75</td>
<td>1.511,61</td>
</tr>
<tr>
<td>8388608</td>
<td>40.958,17</td>
<td>1.562,57</td>
</tr>
</tbody>
</table>
Figure 30: MPI signature graph

Figure 31: MPI saturation graph
Table 5 clearly shows that MPI is much better implemented on Maestro2, in what respects the latency. So very small MPI messages (<1KByte) transmits 3 times faster in Maestro2 system than in Myrinet system. That would be of major importance if application programmer was interested in the best latency, because the program just exchanges small messages. In this case it would be optimal to use a Maestro2 system.

<table>
<thead>
<tr>
<th></th>
<th>Maestro2</th>
<th>Myrinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest Layer</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>MPI layer</td>
<td>15</td>
<td>48</td>
</tr>
</tbody>
</table>

### 4.4.3 Experiment 3 – Comparison of different software configurations

Figure 32 depicts two plots of MPI over MMP on Maestro2 using different NetPIPE and MPI configurations. Curve 1 was obtained using a perturbation value of 32 Bytes, creating aligned messages, and by changing MPICH to use a threshold of 4 KBytes for the transition of one-copy to zero-copy system. Curve 2 was obtained using a perturbation value of 3 Bytes creating misaligned packets and a threshold of 16 KBytes for the MPICH transition of one-copy to zero-copy.

As it can be noticed, there is an enormous degradation in performance when messages are of the misaligned type. When messages sizes are different from a multiple of 32, firmware incurs in many alignment procedures that result in the increase of the time spent to conclude the transmission.

Using misaligned messages, the average value for the maximum bandwidth stays at only 350 Mbits/s, 22% of the maximum bandwidth achievable by using aligned messages.

To explain both dips it is necessary to explain the MPICH implementation. MPICH implementation has several defined values on configuration files used during compilation to create final executable file. These values are used to define internal buffer sizes for message passing buffering. In *ch_maestro2*, the ADI layer for MPICH over MMP, there are two defined values that cause these dips.

First dip corresponds to the value of MPID_PKT_MAX_DATA_SIZE constant, which defines the maximum size for control messages. Messages with size smaller than this value are immediately sent to destination, because receiver has a buffer of this size prepared to receive these short messages. When messages are bigger than this value, their transmission is preceded by a handshake operation mechanism that degrades performance.

As the defined value of MPID_PKT_MAX_DATA_SIZE was set to 1952 bytes during these measurements, it creates the first dip in both curves. Changing this value not only changes the size at which the dip is verified but also the minimum latency and maximum throughput achieved.
To explain the second dip it is necessary to explain another MPI implementation detail. For messages smaller than a threshold value, they are copied using CPU to an internal MPI buffer, Figure 33 b), and then from that buffer to NIC buffer, Figure 33 a) using DMA transfer, Figure 33 c). These MPICH internal buffers are already pindowned by the NIC, so the process is just copy operations, and as there is one copy involving the CPU, this procedure is called 1-copy technique. When messages are bigger than the threshold, which is the value for the size of the buffer, the zero-copy procedure is used, which includes pindowning the message buffer memory area, the DMA transfer of the message to the NIC, Figure 34 a) and the release of the pindowned area.

The size of this internal MPI buffer, which also defines the threshold value, is given by CH_MAESTRO2_ZEROCPY_THRESHOLD constant. Both constants are defined in chconfig.c file of ch_maestro2.

In the MMP curves shown on Experiments 1 and 2, the measurements did not include the timing of pindowning, but that now it is taken into account in the 0-copy procedure in MPICH over MMP. The problem is the time taken to pindown a page in the main memory, because it takes many kernel calls and the transmission of addresses to the NIC. This time is so great compared with the communication time that it creates this second dip. For example, and recurring to Figure 31, the message with size 4KBytes less 32Bytes takes 143 μs to be transmitted where the time for the transmission of a message with size 4KB plus 32Bytes is
490 µs. As previous static analysis shows that the pindown operation takes approximately 130µs per page, and the second message in the example corresponds to two pages, it means that more than 50% of the reported transmission time is due to the pindown operation overhead.

As the time to pindown a page is constant, when transmitting very big messages, the final percentage of the pindown time overhead in the overall transmission time is very small, and so the curve easily rises to the final normal values.
cross. The cross happens because 1-copy operation saturates much faster than 0-copy operation, for messages bigger than 64 KBytes.
Concerning the best value for the MPIO_PKT_MAX_DATA_SIZE constant, the study showed that, increasing this value, latency would also increase, which is bad, but throughput for average message sizes increases, which is good. Study suggested value is 4KBytes, because latency remains the same at 15 µs and throughput increases 250% at message sizes from 2KB to 4KB, compared to the use of the default value of 1952 bytes.

4.5 Conclusion

Analysis performed to the results of the experiments show that stacking of protocol layers can cascade performance degradation, as in the case of TCP/IP layer on Myrinet. One relevant result is the required alignment in the messages size to sustain a good performance on the MPI layer on both architectures, indicating that is necessary to review and change the communication libraries or the demand of an extra care for the application programmer.

Concerning MPI over MMP on Maestro2 system, the results presented show that the system is operational, but it still needs improvement at the software layer, to eliminate the presence of performance drops for certain message sizes. In relation to latency, it already offers 15 µs, which will result in fast MPI control operations, like MPI_Sync and MPI_Barrier.

Analysis also concludes that dynamic performance is influenced by inherent system capabilities, such as DMA transfer speed.
5 Final Conclusions and Future Work

In this work it was possible to develop and integrate on the Maestro2 system a debugging tool for static analysis of the network, which also helps the development of new software, one of the main objectives of this work.

At the dynamic performance analysis, Maestro2 shows promising results as a future competitor on the market as a dedicated network system for clusters. At the moment it is not expected more improvements on the communications lower layer performance without changing the potential performance offered by the physical components of the network. Although buffer sizes on the MPI library affect the performance on the transmission of messages with sizes bigger than 2KBytes, it already provides very good latency for the message sizes bellow 2Kbytes: Maestro2 offers an MPI latency of 15µs compared with the 48µs offered by the MPI/LAM implementation over Myrinet. This is a very good result for the dynamic analysis of the system.

The next steps in the static analysis should be the development of a tool that allows a better and faster output to user. One temporary solution might me the reduction of information on the screen. It is also encouraged the development and extend of ‘gdb’ functionalities to have system performance evaluation tools particular to our system.

At the dynamic performance, it is expected to overcome the MPI problematic falls in throughput, thru the change of the library code. It is also recommended a more extended and exhaustive test to the MPI library of Maestro2. For comparing the performance of Maestro2 with other systems it would be recommended in the future that the tests take place on equal host platforms for unbiased results.

As soon as MPI library start showing improved results, tests should advance to the use of real applications to determine how fast will be the execution time.

Tests should be performed using several hosts interconnected by the SB, to determine the real loss of performance by the introduction of the SB on the communication path.

The maximum charge that a switch box can handle should also be determined, without loss of performance to each individual communication, by trying to overload the SB with several hosts connected to the SB with intensive communication.
Appendix
Appendix A

The following code consists of the file debugutils.c, and contains the macros for the conditional replacement of the printing messages. It also contains the function that sends the encoded messages to ‘gdb’ and the function that creates the decrementer exception at the start of execution.

```c
#ifdef DEBUG_ON /*debug flag, if set all printf() code is replaced*/
char temporary_text[4000]; /*Maximum chars for each printf*/
#endif
#define BREAKPT() printf("This is a TRAP, do not CONTINUE or STEP\n");
#define MAX_CHARS 10 /*limit size for chars coded to hex in each packet going to gdb*/
#define printf(...) {sprintf(temporary_text, __VA_ARGS__); putout(temporary_text);}
int putout(char * text_to_send)
{
    char very_big_string[4001], small_string[150]; /*Strings used for temporary storage of chars*/
    char * pointer_tmp; /*pointer used to break big strings in to smaller ones*/
    /*breaks the printf string in smaller ones, code them to hex, and send to gdb*/
    for(pointer_tmp=text_to_send;pointer_tmp<(text_to_send+(strlen(text_to_send))));pointer_tmp+=MAX_CHARS)
    {
        strncpy(small_string,pointer_tmp,MAX_CHARS);
        small_string[MAX_CHARS] = '\0';
        very_big_string[0] = 'O';
        mem2hex(small_string,(very_big_string+1),strlen(small_string));
        putpacket(very_big_string);
    }
    return 0;
}
#define create_dec_exp() create_decrementer_exception()
void create_decrementer_exception()
{
    register unsigned int zero_tmp=0;
    asm("mtdec %0 ::r" (zero_tmp));
    printf("\n Passed Decrementer exception \n");
}
#define brlock() wait_forever_fnc()
void wait_forever_fnc()
{
}
```
printf("please type 'restart' in db monitor application\n");
for(;;);

#else    //ifdef debug_on

#define create_dec_exp()
#define BREAKPT()
#define brlock()

#endif //ifdef debug_on
# Appendix B

This is the code of the program debugged, referenced in chapter 5

```c
#include <stdio.h>
#include <stdlib.h>
#include "debugutils.c"
#define SIZEVAL 100

int example_global_var=0;

/* simple function example*/
volatile int calc(int received)
{
    return (received+1);
}

/*quicksort compare function*/
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/*prime number example*/
int main(){
    int divisor, dividend;

    /*Communications libraries initialization*/
    maestro2_ni_initialize();

    /*force timer exception*/
    create_dec_exp();

    /*initial value for prime number candidate*/
    dividend=99999999;

    /*endless loop*/
    while((1))
    {
        divisor= dividend>>1;
        while (divisor>1){
            if (!(dividend % divisor)) break ;
            divisor--;
        }
        if (divisor==1) printf("%9d is prime\n",dividend);

        /*endless loop wayout*/
        if (dividend>999999999) break;
        dividend +=2;
    }

    printf("End of Program\n");

    /*force program execution stop*/
    brlock();

} /* main*/
```
8 Appendix C

This appendix contains a log of a debugging session using ‘gdb. The program debugged is the one in Appendix B. Comments to the performed commands and resultant output are indicated in italic. Lines consisting in ‘(gdb) text ‘are commands inserted in ‘gdb’ prompt. It is intended with this description to understand how debugging takes place. Debugging functionality demonstrated is the change of context in middle of execution of the program.

guapo@orthos:~/debugger> TGDB ./test.s4

File test.s4 consists in the memory image downloaded to the NI memory
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=powerpc-none-elf".
...

Command indicating serial connection with the target

(gdb) target remote /dev/maestro2_char0
Remote debugging using /dev/maestro2_char0
0x03f00100 in hreset ()
(gdb) info b
No breakpoints or watchpoints.

Setting breakpoint at start of main() function

(gdb) b main
Breakpoint 1 at 0x3fd4: file test.c, line 35.
(gdb) c
Continuing.
Can't send signals to this remote system. SIGILL not sent.

Program received signal SIGUSR1, User defined signal 1.
0x00003180 in clr_tlb ()
Decrementer exception occured
(gdb) c
Continuing.
Can't send signals to this remote system. SIGUSR1 not sent.

Breakpoint 1, main () at test.c:35
35      while((1))
(gdb) c
Continuing.

Program received signal SIGUSR1, User defined signal 1.
0x00003f30 in create_decrementer_exception () at debugutils.c:37
37      printf("\n Passed Decrementer exception \n");

Forced decrementer exception occurrence
Program output

Passed Decrementer exception
100000007 is prime

End of program output

Program received signal SIGUSR1, User defined signal 1.
main () at test.c:39
39                      if (!(dividend % divisor))  break ;
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGUSR1 not sent.

100000037 is prime  Program output

Program received signal SIGUSR1, User defined signal 1.
0x00003ff4 in main () at test.c:39
39                      if (!(dividend % divisor))  break ;
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGUSR1 not sent.

100000039 is prime  Program output
100000049 is prime  Program output

Program received signal SIGUSR1, User defined signal 1.
main () at test.c:39
39                      if (!(dividend % divisor))  break ;

Context change by altering local variable contents

(gdb) set dividend=999999990
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGUSR1 not sent.

Program output

End of Program
please type 'restart' in db monitor application

End of Program output and end of program execution
9 Appendix D

This appendix contains one of altered handlers assembly code, run when exception occurs. As example, only one of the handler codes is shown, the one for “Program Exception”, the others are similar. Each handler calls the special code sections that perform the backup of the PowerPC registers before calling the stub code, and consequent restore after the stub returns. Those codes are also included on this appendix.

Program Exception Handler for PowerPC603e, Assebly code:

```
.org 0x600
.globl program
program:
###########################################################
#RG code 9-12-2002                                        #
# this code checks if debug flag is ON                    #
# if it is, then run gdb stub                            #
###########################################################
.ifdef DEBUG_ON
mtsprg 3, r1                      #backup r1, user stack pointer
lis  r1, tempsave2@ha             #get tempsave adress
addi r1, r1, tempsave2@l         #to backup other registers
stw  r0, 0(r1)                   #backup r0
addi r1, r1, 4
mfsprg r0, 3                    #backup r1
stw  r0, 0(r1)
addi r1, r1, 4
stmw r2, 0(r1)                   #backup r2 to r31
lis  r8, remote_debug@ha         #go get debug flag
addi r8, r8, remote_debug@l
lwz  r9, 0(r8)
cmpwi r9, 0                      #test debug flag
beq  _no_debug_0x700             #if not to debug, restore regs and continue
mf1r  r28                        #save lr
mfsrr1 r3                       #special code for program exception
     #ins not supported
lis r4, 0x001f                   #retreive bits indicating type of exception
and r3, r3, r4
oris r3, r3, 0x700              #indicate address of exception
bl  save_registers_stub         #save registers for stub use
lis r1, (intstk+INTSTK_SIZE)@ha
addi r1, r1, (intstk+INTSTK_SIZE)@l    #stub will use except stack
bl  handle_exp_stub             #go into stub
bl  restore_registers_stub      #restore from stub use

_no_debug_0x700:

mtlr r28                         #restore lr
lis  r1, tempsave2@ha             #get tempsave adress
addi r1, r1, tempsave2@l         #to recover registers
lwz  r0, 4(r1)                   # get r1
mtsprg 3, r0                     #punch it
lwz  r0, 0(r1)                   #get r0
addi r1, r1, 8                  #advance to r2 backup position
lmw  r2, 0(r1)                   #restore all other GPR
mfsprg r1, 3                    #put back r1 value, what ever it as.

.endif
###########################################################
#end of RG code 9-10-2002       #now it will run Yama normal exception code
###########################################################
rfi
```
Registers of PowerPC603e Backup, Assebly code:

```assembly
#RG code 8-10-2002
#
# save registers for gdb use
# r28 comes with lr
#
ifdef DEBUG_ON

save_registers_stub:

mfmsr r22             # enable FP instructions
oris r22, r22, 0x0000   # because they are disabled
ori  r22, r22, 0x2000   # due to the exception cleaning msr bits
mtmsr r22

lis  r21, tempsave2@ha  # get tempsave address
addi r21, r21, tempsave2@l
mfsrc0 r10
lis  r11, registers@ha  # get global variable seen by stub
addi r11, r11, registers@l
stw  r10, 0x180(r11)    # store address of program counter
mfsrc2 r12, 3
stw  r12, 0x184(r11)   # store user stack pointer
mfcr r13
stw  r13, 0x188(r11)   # store cr

# saving registers for debugging the debugger
mfmsr r13
stw  r13, 0x80(r21)
mfsrc1 r13
stw  r13, 0x84(r21)
mfsrc2 r13
stw  r13, 0x8c(r21)

stw  r28, 0x18c(r11)    # store lr
mfcr r14
stw  r14, 0x190(r11)   # store ctr
mfexc r15
stw  r15, 0x194(r11)   # store xer
addi r21, r21, -4      # prepare for copy cycle
addi r11, r11, -4      # prepare for copy cycle
xor  r16, r16, r16
addi r16, r16, 32      # counter

_load_registers_from_tempsave:

lwzu r17, 4(r21)
stwu r17, 4(r11)
addi r16, r16, -1
cmpwi r16, 0
bne _load_registers_from_tempsave

addi r11, r11, -4      # adjust for update instruction
stfdu f0, 8(r11)
stfdu f1, 8(r11)
stfdu f2, 8(r11)
stfdu f3, 8(r11)
stfdu f4, 8(r11)
stfdu f5, 8(r11)
stfdu f6, 8(r11)
stfdu f7, 8(r11)
stfdu f8, 8(r11)
stfdu f9, 8(r11)
stfdu f10, 8(r11)
stfdu f11, 8(r11)
```

Registers of PowerPC603e Restore, after stub run, Assembly code:

```assembly
restore_registers_stub:
    xor     r8,r8,r8  # lets check if we need to
    lis    r8,stepping@ha  # change the trace bit
    addi   r8,stepping0l  # in MSR
    lwz   r9,0(r8)
    cmpwi   r9,0
    beq     __not_step
    mfsrr1  r0
    ori    r0,r0,0x0400  # set SE
    mtsrr1  r0
    b      __escl
__not_step:
    mfsrr1  r0
    # andi    r0,r0,0xfbff  # instruction not supported!!!  # reset SE
    lis    r5,0xffff
    ori    r5,r5,0xfbff
    and    r0,r0,r5
    mtsrr1  r0
__escl:
    lis    r8,tempsave2@ha
    addi   r8,r8,tempsave20l
    xor    r9,r9,r9
    lis   r9,registers@ha
    addi   r9,r9,registers0l
    addi   r8,r8,-4  # prepare for update
    addi   r9,r9,-4  # "   "
    xor    r11,r11,r11
    addi   r11,r11,32

____store_GPR_from_REGPOOL:
    lwzu  r10,4(r9)  # copy from registers to tempsave
    stwu  r10,4(r8)
    addi   r11,r11,-1
    cmpwi   r11,0
    bne    ____store_GPR_from_REGPOOL
```

The code above shows how to restore registers after a stub run. It includes checks for stepping and setting the SE bit, copying registers to tempsave, and handling branch unconditional to (lr).
addi     r9, r9, -4  # adjust for update instructions
1fdu     f0, 8(r9)  # restore fp regs
1fdu     f1, 8(r9)
1fdu     f2, 8(r9)
1fdu     f3, 8(r9)
1fdu     f4, 8(r9)
1fdu     f5, 8(r9)
1fdu     f6, 8(r9)
1fdu     f7, 8(r9)
1fdu     f8, 8(r9)
1fdu     f9, 8(r9)
1fdu     f10, 8(r9)
1fdu     f11, 8(r9)
1fdu     f12, 8(r9)
1fdu     f13, 8(r9)
1fdu     f14, 8(r9)
1fdu     f15, 8(r9)
1fdu     f16, 8(r9)
1fdu     f17, 8(r9)
1fdu     f18, 8(r9)
1fdu     f19, 8(r9)
1fdu     f20, 8(r9)
1fdu     f21, 8(r9)
1fdu     f22, 8(r9)
1fdu     f23, 8(r9)
1fdu     f24, 8(r9)
1fdu     f25, 8(r9)
1fdu     f26, 8(r9)
1fdu     f27, 8(r9)
1fdu     f28, 8(r9)
1fdu     f29, 8(r9)
1fdu     f30, 8(r9)
1fdu     f31, 8(r9)
xor     r9, r9, r9
lis     r9, registers@ha
addi    r9, r9, registers@l
lwz     r10, 0x180(r9)  # recover PC
mtsrr0  r10
lwz     r10, 0x184(r9)  # recover SP (PS)
mtsprg  3, r10
lwz     r10, 0x188(r9)  # recover cr
mtcr    r10
lwz     r10, 0x18c(r9)  # recover lr
mr      r28, r10
lwz     r10, 0x190(r9)  # recover ctr
mtctr   r10
lwz     r10, 0x194(r9)
mtxer   r10

# Data Cache flush and invalidate
li      r11, 0xc00
mfspr  r12, HID0
or     r13, r12, r11
mtspr  HID0, r13
mtspr  HID0, r12
isync

bclr   0x14, 0  # return

#end of restore regs


59
This appendix contains the stub code. Stub consists in one main routine designed to wait for and to respond to requests coming from the debugging application, and in several other subroutines that carry out those requests.

```c
#ifdef DEBUG_ON

/* ppc32-stub.c -- debugging stub for the PowerPC

NOTE!! This code has to be compiled with optimization, otherwise the function inlining which generates the exception handlers won't work.
*/

/********************************************************************************
THIS SOFTWARE IS NOT COPYRIGHTED

HP offers the following for use in the public domain. HP makes no warranty with regard to the software or it's performance and the user accepts the software "AS IS" with all faults.

HP DISCLAIMS ANY WARRANTIES, EXPRESS OR IMPLIED, WITH REGARD TO THIS SOFTWARE INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

********************************************************************************/

/* Remote communication protocol.

A debug packet whose contents are <data>
is encapsulated for transmission in the form:

$ <data> # CSUM1 CSUM2

<data> must be ASCII alphanumeric and cannot include characters
'$$' or '#'. If <data> starts with two characters followed by
':', then the existing stubs interpret this as a sequence number.

CSUM1 and CSUM2 are ascii hex representation of an 8-bit
checksum of <data>, the most significant nibble is sent first.
the hex digits 0-9,a-f are used.

Receiver responds with:

  +  - if CSUM is correct and ready for next packet
  -  - if CSUM is incorrect

<data> is as follows:
All values are encoded in ascii hex digits.

Request  Packet
----------------------------------
read registers  g   Each byte of register data
   reply     XX....X   is described by two hex digits.

or ENN

write regs GXX..XX Each byte of register data
   reply     OK   is described by two hex digits.
               for success
               for an error

ENN

```
write reg  Pn...=r... Write register n... with value r..., which contains two hex digits for each byte in the register (target byte order).

reply  OK for success
       ENN for an error
       (not supported by all stubs).

read mem  mAA..AA,LLLL AA..AA is address, LLLL is length.
reply  XX..XX XX..XX is mem contents
       or ENN NN is errno
       Can be fewer bytes than requested if able to read only part of the data.

write mem  MAA..AA,LLLL:XX..XX
AA..AA is address, LLLL is number of bytes, XX..XX is data
reply  OK for success
       ENN for an error (this includes the case where only part of the data was written).

cont  cAA..AA AA..AA is address to resume
      If AA..AA is omitted, resume at same address.

step  sAA..AA AA..AA is address to resume
      If AA..AA is omitted, resume at same address.

last signal  ? Reply the current reason for stopping. This is the same reply as is generated for step or cont : SAA where AA is the signal number.

There is no immediate reply to step or cont. The reply comes when the machine stops. It is SAA AA is the "signal number"
or...
      TAAn...:r...;n:r...;n...:r...;
      AA = signal number
      n... = register number
      r... = register contents

or...
      WAA The process exited, and AA is the exit status. This is only applicable for certains sorts of targets.

kill request  k

toggle debug  d toggle debug flag (see 386 & 68k stubs)
reset  r reset -- see sparc stub.
reserved  <other> On other requests, the stub should ignore the request and send an empty response ($#<checksum>). This way we can extend the protocol and GDB can tell whether the stub it is talking to uses the old or the new.

search  tAA:PP,MM Search backwards starting at address AA for a match with pattern PP and mask MM. PP and MM are 4 bytes. Not supported by all stubs.

general query  qXXX Request info about XXXX.
general set  QXXX=yyyy Set value of XXXX to yyyy.
query sect offs  qOffsets Get section offsets. Reply is Text=xxx;Data=yyy;Bss=zzz
console output  Otext Send text to stdout. Only comes from remote target.

Responses can be run-length encoded to save space. A '*' means that the next character is an ASCII encoding giving a repeat count which stands for that many repetitions of the character preceding the '*'. The encoding is n+29, yielding a printable character where n >=3 (which is where rle starts to win). Don't use an n > 126.

So
"0* " means the same as "0000". */

/*
 * BUFMAX defines the maximum number of characters in inbound/outbound
 * buffers. At least NUMREGBYTES*2 are needed for register packets.
 */
#define BUFMAX 8192

/*
 * Number of bytes for registers
 */
#define NUMREGBYTES (4*32+8*32+4+4+4+4)
#define NUMREGBYTES (4*32+4+4+4+4+4)

/*
 * Forward declarations
 */

static int hex (char);
static char *hex2mem (char *, char *, int);
static int hexToInt (char **, int *);
static unsigned char *getpacket (void);
void handle_exp_stub (int exceptionVector);
void init_serial();

void putDebugChar (char);
char getDebugChar (void);
char *mem2hex (char *, char *, int);
void putpacket (char *);

int stepping ;
#define SYS_RESET           (0x01000000)
#define MACHINE_CHECK       (0x02000000)
#define DSI_EXCEPTION       (0x03000000)
#define ISI_EXCEPTION       (0x04000000)
#define EXTERN_INT          (0x05000000)
#define ALIGNMENT           (0x06000000)
#define PROGRAM_EXCEPTION   (0x07000000)
#define FP_UNABLE            (0x08000000)
#define DECREMENT           (0x09000000)
#define SYSCALL             (0x0c000000)
#define TARECE              (0x0d000000)
#define BP_EXCEPTION         (0x13000000)

int remote_debug=0xdd;/* debug flag set*/

/* PowerPC UISA */
enum regnames
{
  r00, r01, r02, r03, r04, r05, r06, r07,
  r08, r09, r10, r11, r12, r13, r14, r15,
  r16, r17, r18, r19, r20, r21, r22, r23,
  r24, r25, r26, r27, r28, r29, r30, r31,
  f00, f01, f02, f03, f04, f05, f06, f07,
  f08, f09, f10, f11, f12, f13, f14, f15,
  f16, f17, f18, f19, f20, f21, f22, f23,
  f24, f25, f26, f27, f28, f29, f30, f31,
  pc, ps, cr, lr, ctr, xer
};

int registers[NUMREGBYTES / 4];
const char hexchars[] = "0123456789abcdef";
char remcomInBuffer[BUFMAX];
char remcomOutBuffer[BUFMAX];

char highhex(int x)
{
  return hexchars[(x >> 4) & 0xf];
}

char lowhex(int x)
{
  return hexchars[x & 0xf];
}
/*
 * Routines to handle hex data
 */

int hex (char ch)
{
    if ((ch >= 'a') && (ch <= 'f'))
        return (ch - 'a' + 10);
    if ((ch >= '0') && (ch <= '9'))
        return (ch - '0');
    if ((ch >= 'A') && (ch <= 'F'))
        return (ch - 'A' + 10);
    return (-1);
}

/* convert the memory, pointed to by mem into hex, placing result in buf */
/* return a pointer to the last char put in buf (null) */
char *
mem2hex (char *mem, char *buf, int count)
{
    int i;
    int ch;

    // for( i=count ; i>=0 ; i-- ){  ####### generated extra byte beeing moved to array
    for( i=count ; i>0 ; i-- ){
        ch = *mem++;
        *buf++ = highhex (ch);
        *buf++ = lowhex (ch);
    }
    *buf = '\0' ;
    return (buf);
}

void putDebugChar(char c)
{
    char temp;
    temp=c;
    write(0,&temp,1);
}

char getDebugChar()
{
    char temp;
    read(0,&temp,1);
    return temp;
}

/* convert the hex array pointed to by buf into binary, to be placed in mem */
/* return a pointer to the character after the last byte written */
static char *
hex2mem (char *buf, char *mem, int count)
{
    int i;
    unsigned char ch;

    for (i = 0; i < count; i++)
    {
        ch = hex (*buf++) << 4;
        ch = ch + hex (*buf++);
        *mem++ = ch;
    }
    return (mem);
}

/*****************************/
/* WHILE WE FIND NICE HEX CHARS, BUILD AN INT */
/* RETURN NUMBER OF CHARs PROCESSED */
/*****************************/
static int
hexToInt (char **ptr, int *intValue)
{
    if (*ptr == NULL)
        return 0;

    while (*ptr != NULL)
    {
        *intValue = (*intValue << 4) + (*ptr - '0');
        *ptr = NULL;
    }

    return (*intValue);
}
int numChars = 0;
int hexValue;

    *intValue = 0;
    while( **ptr != '\0' ){
        hexValue = hex (**ptr);
        if (hexValue >= 0) {
            *intValue = (*intValue << 4) | hexValue;
            numChars++;
        } else {
            break;
        }
    }

    return (numChars);
}

/*
 * Routines to get and put packets
 */

/* scan for the sequence $<data>#$<checksum> */

unsigned char *
getpacket (void)
{
    unsigned char *buffer = &remcomInBuffer[0];
    unsigned char checksum;
    unsigned char xmitcsum;
    int count;
    char ch;

    while (1){
        /* wait around for the start character, ignore all other characters */
        while ((ch = getDebugChar ()) != '$') ;
        retry:
        checksum = 0;
        xmitcsum = -1;
        count = 0;
        /* now, read until a # or end of buffer is found */
        while (count < BUFMAX){
            ch = getDebugChar ();
            if (ch == '$') 
                goto retry;
            if (ch == '#')
                break;
            checksum = checksum + ch;
            buffer[count++] = ch;
            }
        buffer[count] = '\0' ;
        if (ch == '#')
            ch = getDebugChar ();
        xmitcsum = hex (ch) << 4;
        ch = getDebugChar ();
        xmitcsum += hex (ch);
        if (checksum != xmitcsum){
            putDebugChar ('-'); /* failed checksum */
        } else {
            putDebugChar ('+'); /* successful transfer */
        } /* if a sequence char is present, reply the sequence ID */
        if (buffer[2] == ':'){
            putDebugChar (buffer[0]);
            putDebugChar (buffer[1]);
            return &buffer[3];
        } else {
            return &buffer[0];
        }
    }
```c
/* send the packet in buffer. */

void
putpacket (register char *buffer)
{
    unsigned char checksum;
    int count;
    unsigned char ch;

do{
    putDebugChar('$');
    checksum = 0;
    count = 0;
    while (ch = buffer[count]){
        putDebugChar(ch);
        checksum += ch;
        count += 1;
    }
    putDebugChar('#');
    putDebugChar(hexchars[checksum >> 4]);
    putDebugChar(hexchars[checksum & 0xf]);
}
    while (getDebugChar() != '+');
}

/*
 * this function takes the SH-1 exception number and attempts to
 * translate this number into a unix compatible signal value
 */
static int
computeSignal (int exceptionVector)
{
    int sigval;

    switch((exceptionVector&0xff000000)) {
    case SYS_RESET:
        sigval = 1; /** SIGHUP **/
        break;
    case MACHINE_CHECK:
        sigval = 1; /** SIGHUP **/
        break;
    case DSI_EXCEPTION:
        sigval = 11; /** SIGSEGV **/
        break;
    case ISI_EXCEPTION:
        sigval = 11; /** SIGSEGV **/
        break;
    case EXTERN_INT:
        sigval = 2; /** SIGINT **/
        break;
    case ALIGNMENT:
        sigval = 10; /** SIGBUS **/
        break;
    case PROGRAM_EXCEPTION:
        if( (exceptionVector & 0x00100000) == 0x00100000 ){
            sigval = 8; /** SIGFPE **/
        }else if( (exceptionVector & 0x00080000) == 0x00080000 ){
            sigval = 4; /** SIGILL **/
        }else if( (exceptionVector & 0x00040000) == 0x00040000 ){
            /** used Supervisor Inst on usermode **/
            sigval = 4; /** SIGILL **/
        }else if( (exceptionVector & 0x00020000) == 0x00020000 ){
            sigval = 5; /** SIGTRAP **/
        }else{
            sigval = 4; /** SIGILL **/
        }
        break;
    case FP_UNABLE:
        sigval = 4; /** SIGILL **/
        break;
    case DECREMENT:
```
```c
sigval = 30; /** SIGUSR **/
bcase SYSCALL:
    sigval = 12; /** SIGSYS **/
bcase TARCE:
    sigval = 5; /** SIGTRAP **/
bcase BP_EXCEPTION:
    sigval = 31; /** SIGBP **/
default:
    sigval = 32; /* "software generated"*/
}
}

/*
This function does all exception handling. It only does two things -
it figures out why it was called and tells gdb, and then it reacts
to gdb's requests.

When in the monitor mode we talk a human on the serial line rather than gdb.
*/

void handle_exp_stub (int exceptionVector)
{
    int sigval;
    char *addr;
    int length;
    char *ptr;

    /* reply to host that an exception has occurred */
    sigval = computeSignal (exceptionVector);
    if (sigval==5) stepping=0;
    remcomOutBuffer[0] = 'S';
    remcomOutBuffer[1] = highhex(sigval);
    remcomOutBuffer[2] = lowhex (sigval);
    remcomOutBuffer[3] = '\0';
    putpacket (remcomOutBuffer);

    while(1)
    {
        remcomOutBuffer[0] = '\0';
        ptr = getpacket();

        switch (*ptr){
            case '?':
                remcomOutBuffer[0] = 'S';
                remcomOutBuffer[1] = highhex (sigval);
                remcomOutBuffer[2] = lowhex (sigval);
                remcomOutBuffer[3] = '\0';
                break;
            case 'd':
                remote_debug = !(remote_debug); /* toggle debug flag */
                break;
            case 'g': /* return the value of the CPU registers */
                mem2hex ((char *) registers, remcomOutBuffer, NUMREGBYTES);
                break;
            case 'G': /* set the value of the CPU registers - return OK */
                hex2mem (ptr, (char *) registers, NUMREGBYTES);
                strcpy (remcomOutBuffer, "OK");
                break;
            case 'H': /* set the value of the CPU registers - return OK */
                strcpy (remcomOutBuffer, "OK");
                break;
            case 'm': /* mA..AA,LLLL Read LLLL bytes at address AA..AA */
                
                
            
        }
    }
```
while( *ptr!='\0' ){
    if( *ptr==',' ){ ptr++ ; break ; }
    i=hex(*ptr) ;
    if( i>=0 ) j=((j<<4) | i) ;
    else       break ;
    ptr++ ;
}
addr=(char*)j ;
length=0 ;
while( *ptr!='\0' ){
    if( *ptr==',' ){ ptr++ ; break ; }
    i=hex(*ptr) ;
    if( i>=0 ) length= ((length << 4) | i) ;
    else       break ;
    ptr++ ;
}
if( length>0 ) mem2hex ( addr, remcomOutBuffer, length);
else       strcpy (remcomOutBuffer, "E02" );
break;

case 'M': /* MAA..AA,LLLL: Write LLLL bytes at address AA.AA return OK */
{
    int i,j;
    j=0 ;
    while( *ptr!='\0' ){
        if( *ptr==',' ){ ptr++ ; break ; }
        i=hex(*ptr) ;
        if( i>=0 ) j=((j<<4) | i) ;
        else       break ;
        ptr++ ;
    }
    addr=(char*)j ;
    length=0 ;
    while( *ptr!='\0' ){
        if( *ptr==',' ){ ptr++ ; break ; }
        i=hex(*ptr) ;
        if( i>=0 ) length= ((length << 4) | i) ;
        else       break ;
        ptr++ ;
    }
    if( length>0 ){
        hex2mem (ptr, (char *) addr, length);
        strcpy (remcomOutBuffer, "OK" );
    }else{
        strcpy (remcomOutBuffer, "E02" );
    }
break;
}

case 's':
    stepping = 1;
    return ;
    break ;

case 'c':
    return ;
    break ;

case 'k': /* kill the program */
/*in our case, will be the same as restart command from MONITOR*/
    switch( *ptr ){  
    case 'C' : /** thread ID **/
        /*...*/
        strcpy( remcomOutBuffer, "QC0000" ) ;
    case 'q': /* query */
        switch(*ptr){
            case 'C' : /** thread ID **/
                strcpy( remcomOutBuffer, "QC0000" ) ;
                break; /* do nothing */
            case 'q': /* query */
                /*...*/
                break; /* do nothing */
        }  
}  
/*...*/
break;
/*in our case, will be the same as restart command from MONITOR*/
strcpy( remcomOutBuffer, "RESTART COMMAND ACTIVATED, restart debugging session is necessary!!" ) ;
putpacket (remcomOutBuffer);
asm(" lis 10, __reboot_point@ha");
asm(" ori 10, 10, __reboot_point@l");
asm(" mtspr 26, 10 ");
asm(" rfi");
break; /* do nothing */

break;
case 'O' : /** Offset **/
    strcpy( remcomOutBuffer, "Text=0;Data=220000;Bss=230000" ) ;
    break ;
}  
break;}
/* reply to the request */
putpacket (remcomOutBuffer);
}

char* strcpy(char *dst, const char *src)
{
    int i ;

    for( i=0 ; ; i++ ){
        dst[i]=src[i] ;
        if( src[i]=='\0' ) break ;
    }
    return( dst ) ;
}

#endif
Appendix F

This appendix consists in the code of the developed module for NetPIPE application program. Using this module NetPIPE is able to carry out message passing using the special message passing library of Maestro2, MMP. The results of using this module are shown on Chapter 4.

```c
#include    "netpipe.h"
#include    <math.h>
#include    <mmp.h>

#define MMSERVER 1
#define MMCLIENT 0
#define MAXLINE 20
#define MAXBUFF 20

struct MMP_connect *my_conn;
struct MMP_connect *your_conn;
struct MMP_send_handle *s_hand,*time_s_hand,*rpt_s_hand;
struct MMP_recv_handle *r_hand,*time_r_hand,*rpt_r_hand;
struct MMP_PNA_table *PNA_table;
FILE * fich;
int status/*clint ,server*/,stat/*status from send/recieve operation*/;
int my_network_addr, your_network_addr, r_addr1, r_addr2;
int *tmp;
int recvPosted = 0;

/* Initialize vars in Init() that may be changed by parsing the command args */
void Init(ArgStruct *p, int* pargc, char*** pargv)
{
    //debug  printf("=====>>>> void Init(ArgStruct *p, int* pargc, char*** pargv) .......
    p->tr = 0;     /* The transmitter will be set usin g the -h host flag. */
    p->rcv = 1;
    //debug  printf("=====>>>> END OF Init(ArgStruct *p, int* pargc, char*** pargv) ......
}

void Setup(ArgStruct *p)
{
    char line1[MAXLINE], line2[MAXLINE];
    int j;

    //debug printf("=====>>>> void Setup(ArgStruct *p) ........\n");
    p->port=1;/*over ride port number*/
```
PNA_table = MMP_alloc_PNA_table(2); // 2¤Ä¥Í¥Ã¥È¥ï¡¼¥¯¤ò»È¤¦¤³¤È¤òÀë¸À
if(PNA_table == NULL){
    printf("ERR: cannot allocate PNA table!
"); exit (1);
}

if ((fich = fopen("mfile.txt", "r")) == NULL){
    printf("ERR: machine file does not exist or is read protected \n"); exit (1);
}

fgets(line1, MAXLINE, fich);
fgets(line2, MAXLINE, fich);

if((line1[0]!='0') || (line1[1]!='x') || (line2[0]!='0') || (line2[1]!='x')) {
    printf("File format Error\n"); exit (1);
}

r_addr1=0;
r_addr2=0;
for (j=2; j<10; j++) {
    r_addr1+=(line1[j] - '0')*pow(16,9-j);
    r_addr2+=(line2[j] - '0')*pow(16,9-j);
}

//printf(" j=%d value=%d  x %d\n",j,line[j] - '0', pow(16,9-j) );

// printf(" Li %s e %s  =0x%x  =0x%x   
”,line1, line2,r_addr1, r_addr2);
fclose(fich);

MMP_add_PNA_to_table(PNA_table, SERVER_PNA); //MMP_add_PNA_to_table(PNA_table, CLIENT_PNA);

MMP_add_PNA_to_table(PNA_table, r_addr1);
MMP_add_PNA_to_table(PNA_table, r_addr2);

printf("making connection ------\n");
my_conn = MMP_Make_connect(0, p->port);

printf("making initialization ------\n");

if((my_conn = MMP_initialize(my_conn, PNA_table)) == NULL){
    perror("connection error"); exit (1);
}

my_network_addr = my_conn->network_addr;
if (my_network_addr == PNA_table->PNA[0])
    status = MMSERVER;
else {
    status = MMCLIENT;
    p->tr=1; /* set transmitter*/
    p->rcv=0;
}

//info*/
if (status == MMSERVER) printf("I am a server\n");
else printf("I am a Client \n");

//create outgoing connection*/
if (status == MMCLIENT) your_network_addr = PNA_table->PNA[0];
else your_network_addr = PNA_table->PNA[1];
your_conn = MMP_Make_connect(your_network_addr, p->port);

//REPEAT INTEGER BUFFER ALLOCATION*/
//debug printf("REPEAT BUFFER allocation\n");

if ((tmp = (int *)MMP_Pindown_malloc(MAXBUFF, my_conn)) == NULL){
    fprintf(stderr,"couldn't allocate memory for TMP buffer\n");
    perror("pindown&malloc error");
}
void Sync(ArgStruct *p)
{
  //debug    printf("Going to sync\n");
  MMP_Sync(1, 2);
  //debug    printf("Leaving sync\n");
}

void PrepareToReceive(ArgStruct *p)
{
  //debug    printf("=====\\>>> void PrepareToReceive(ArgStruct *p) .........\n");
  /*not using this functionality yet*/
  if (recvPosted)
  {
    printf("Can't prepare to receive: outstanding receive!\n");
    exit(-1);
  }
  if((r_hand = MMP_Recv(p->r_ptr, p->bufflen, 3,your_conn, my_conn)) == NULL){
    perror("recv error");
    exit(1);
  }
  recvPosted = -1;
}

void SendData(ArgStruct *p)
{
  //debug    printf("=====\\>>> void SendData(ArgStruct *p) .........\n");
  if((s_hand = MMP_Send(p->s_ptr, p->bufflen, 3, my_conn, your_conn)) == NULL){
    perror("send error");
    exit(1);
  }
  stat = MMP_Wait_send(s_hand);
}

void RecvData(ArgStruct *p)
{
  //debug    printf("=====\\>>> void RecvData(ArgStruct *p) .........\n");
  if (recvPosted)
  {
    stat = MMP_Wait_recv(r_hand);
    recvPosted = 0;
  }
  else
  {
    if((r_hand = MMP_Recv(p->r_ptr, p->bufflen, 3,your_conn, my_conn)) == NULL){
      perror("recv error");
      exit(1);
    }
    stat = MMP_Wait_recv(r_hand);
  }
}

void SendTime(ArgStruct *p, double *t)
{
double *timeptr;

//debug    printf("==========> void SendTime(ArgStruct *p, double *t) .......
");

timeptr=(double *)tmp;

*timeptr=t;

if((time_s_hand = MMP_Send(tmp, MAXBUFF, 2, my_conn, your_conn)) == NULL)
{
    perror("send error");
    exit(1);
}

printf("==========> Send time process iniiciated.......\n");
stat = MMP_Wait_send(time_s_hand);
printf("==========> Send time process DONE.......\n");

}

void RecvTime(ArgStruct *p, double *t)
{

double *timeptr;

//debug    printf("==========> void RecvTime(ArgStruct *p, double *t) .......
");

if((time_r_hand = MMP_Recv(tmp, MAXBUFF, 2, you_r_conn, my_conn)) == NULL){
    perror("send error");
    exit(1);
}

printf("==========> Recv time process iniiciated.......\n");
stat = MMP_Wait_recv(time_r_hand);
printf("==========> Recv time process DONE.......\n");

timeptr=(double *)tmp;
*t=timeptr;

}

void SendRepeat(ArgStruct *p, int rpt)
{

//debug    printf("==========> void SendRepeat(ArgStruct *p, int rpt) .......
");

(*tmp)=rpt;

if((rpt_s_hand = MMP_Send(tmp, MAXBUFF, 1, my_conn, your_conn)) == NULL){
    perror("send error");
    exit(1);
}

stat = MMP_Wait_send(rpt_s_hand);

}

void RecvRepeat(ArgStruct *p, int *rpt)
{

//debug    printf("==========> void RecvRepeat(ArgStruct *p, int *rpt) .......
");

if((rpt_r_hand = MMP_Recv(tmp, MAXBUFF, 1, you_r_conn, my_conn)) == NULL){
    perror("recv error");
    exit(1);
}

stat = MMP_Wait_recv(rpt_r_hand);
*rpt=tmp;

}

void CleanUp(ArgStruct *p)
{

if(tmp != NULL)
MMP_Pindown_free(tmp, my_conn);

if(my_conn != NULL){
    MMP_finalize(my_conn);
}

//debug printf("=====> yourconn finalize process iniciated.......\n");

void FreeBuff(char *buff1, char *buff2)
{
    if(buff1 != NULL)
        MMP_Pindown_free(buff1, my_conn);
    if(buff2 != NULL)
        MMP_Pindown_free(buff2, my_conn);
}

void MyMalloc(ArgStruct *p, int bufflen)
{
//debug printf("=====> void MyMalloc(ArgStruct *p, int bufflen) .......
");
    if((p->r_buff = (char *)MMP_Pindown_malloc(bufflen, my_conn)) == (char *)NULL)
        perror("pindown&malloc error");
        exit(1);

    if(!p->cache) /* Allocate second buffer if limiting cache */
        if((p->s_buff = (char *)MMP_Pindown_malloc(bufflen, my_conn)) == (char *)NULL)
            perror("pindown&malloc error");
            exit(1);
}

void Reset(ArgStruct *p)
{
}

void AfterAlignmentInit(ArgStruct *p)
{
}

void InitBufferData(ArgStruct *p, int nbytes)
{
    memset(p->r_buff, 'a', nbytes);
    if(!p->cache)
        memset(p->s_buff, 'b', nbytes);
}
12 Appendix G

This appendix is focused on the study performed on MPICH implementation over MMP in Maestro2 architecture. It presents the results obtained and discusses them.

As indicated on Chapter 4, Experiment 3, in MPICH configuration files there are two constant values that control the behavior of message passing execution:

**CH_MAESTRO2_ZEROCPY_THRESHOLD**: This value defines from which size transfer procedure changes from one-copy to zero-copy mode. One-copy mode implies the use of CPU to copy the message from user address space to a transmission buffer in kernel memory space. Zero-copy mode represents an immediate copy between the user memory space to the NI transmission buffer. Detailed explanation of one-copy and zero-copy method can be found in the text of Chapter 4.

Previous to the study, it was expected that by changing the threshold value, only the point were the performance stops following one of the curves and start following the other changes, as there is no other change in the system.

Figures G1, G2 and G3 show the final results of the study.

![Throughput graph from threshold study](image)
For these measurements MPID_PKT_MAX_DATA_SIZE remained at its default value of 1952 Bytes, as the study only wanted to find the best value for threshold.

Figure G2: Signature graph from threshold study

Figure G3: Saturation graph from threshold study
The one-copy mode curve was obtained using 16Mbytes for the value of the threshold. This way, one-copy operation mode is always used, because packet sizes used in these measurements vary from 1 byte to 8Mbytes. The zero-copy mode curve was obtained using 0 (zero) for the value of the threshold. This way, zero-copy is always used whichever the size of the message.

These figures show, as expected, that the final plot is defined by the transition of one curve to another, exactly at the point defined by the threshold value. So the better value for the threshold is the value were the lines cross, so that there is no dip. Value chosen was 64KBytes, because as can be observed, is the message size were the curves of one-copy and zero-copy cross.

This conclusion is accepted as valid, just for this hardware configuration (were the measurements were performed). Another hardware configuration can have a different cross point for the zero-copy and one-copy curves, and so adjustments will be necessary.

**MPID_PKT_MAX_DATA_SIZE**: This value will define the maximum size for control messages exchanged between MPI applications. As can be observed in figures G4, G5 and G6, changing this value will dramatically change the overall performance.

When the value is increased, throughput increases for message sizes below that value, but system latency also increases (figure G6), fact that was not expected. This performance degradation on latency starts to be so important, that for buffer size equal to 64KBytes, there is just no gain in throughput, when compared with the performance using the default buffer size).
Figure G4: Throughput graph from maximum buffer size study

Figure G5: Signature graph from maximum buffer size study
For these measurements CH_MAESTRO2_ZEROCPY_THRESHOLD remained at its new value of 16 KBytes.

The explanation for the loss of performance for message sizes bigger than the buffer size is straightforward: the message has to be divided in smaller ones, incurring in routing and regrouping overhead. This explains the gain observed on throughput for messages that are smaller than the buffer size.

The disturbing fact is the increase on the latency as response to the increase on the buffer size. That fact was not expected previously to the study, and so the value suggested by this study to be adopted for this buffer size is not yet assumed as a final value.

Having to conjugate these two factors, the value for this constant must be chosen in a way that latencies are still acceptable, but that allow a better throughput for some message sizes than the default value of 1952 Bytes.

The value for the buffer size suggested by this study is 4KBytes:

- latency is the same for the default value, approximately 15 µs,
- throughput increases by 250% for messages in the interval 1952B to 4KBytes, and by 5 to 10% for sizes bigger than 4 KBytes.
13 Bibliographic references


