Service Migration in a Gigabit Network
by
Margaret A.S. Petrus
B.S., Bishop's University, Canada, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia
August 1998
© Margaret A.S. Petrus, 1998
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4

Date: August 24, 1998
Abstract

Migrating services to the network adaptor makes use of the idle processing power in the adaptor that would otherwise be wasted if it was just used for sending and receiving messages. In addition, by moving some functionality to the network adaptor, decisions can be made without involving the host. This would prevent unnecessary memory copies, I/O traversals, interruptions of the host and the associated context switches. Thus the host can use its processing power, without interruption, for more useful work.

The Emu system has made possible the complete implementation of the Emerald interpreter on the network processor. The choice of an object oriented distributed interpreter makes the applications that run on the system more modular and compact because the language takes care of all the gory details. Emu has introduced runtime extension of the functionalities on the network processor and an approach for utilizing idle network processing power. However, intelligent delegation of processing to the network processor is required, because the network processor is slower than the central processor. Communication-intensive applications which require low latency and high throughput would benefit from the interpreter on the network adaptor in the Emu system.

Emu has paved the way for other similar systems to be built. It has demonstrated that it is possible to take full advantage of the programmable NIC to increase system throughput. It has shown that with intelligent choice of applications, idle network processing power can be utilized without sacrificing performance.
Contents

Abstract ii

Contents iii

List of Tables vii

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1
   1.1 Motivation ........................................ 1
   1.2 What is Service Migration? ......................... 3
   1.3 Problem Definition ................................. 3
   1.4 The Network Environment ......................... 4
   1.5 The Emerald Approach ............................. 6
   1.6 Thesis Structure .................................. 8

2 Related Work 10
   2.1 Existing Migration Systems ....................... 10
      2.1.1 Traditional Migration Systems ................ 11
2.1.2 Object Migration Systems/Languages .......................... 14
2.1.3 Heterogeneous Migration Systems .............................. 17
2.2 Moving Functionality to the Network Adaptor .................. 19
2.2.1 U-Net ............................................. 19
2.2.2 Hamlyn .......................................... 20
2.3 Other Languages Considered for the Interpreter ............... 21
2.3.1 Java - Language of the Internet ............................ 22
2.3.2 TCL ............................................. 22
2.3.3 Existing Active Networking Systems/Languages .............. 22
2.4 Contributions of this Thesis .................................. 24

3 Background ......................................................... 26
3.1 Myricom Technology and Details ................................. 26
3.1.1 LANai 4 Processor ..................................... 27
3.1.2 Myrinet API ......................................... 29
3.2 Benchmarks Used for the Choice of the Host Processors ...... 33
3.2.1 Results of bw_mem_cp and hswap Benchmarks ............... 34
3.2.2 Results of Imbench Test Suite ............................. 36
3.2.3 Results of netperf Benchmarks ............................. 37
3.3 Layout of the Myrinet Network Testbed .......................... 41
3.3.1 M2M-DUAL-SW8 Myrinet Switch ............................ 42
3.3.2 M2M-PCI32C Myrinet-SAN/PCI Interface ..................... 43
3.3.3 M2M-CB-05 Cables .................................... 44
3.4 Language Support ............................................. 44
3.4.1 Emerald Primitives to Support Mobility ..................... 44
3.4.2 Structure of the Emerald Interpreter ....................... 46
3.5 Summary ..................................................... 48
4 Design and Implementation of Emu

4.1 Key Issues for Moving the Emerald Interpreter to the LANai

4.1.1 Limited Memory

4.1.2 Library Support

4.1.3 Distribution and Mobility

4.2 Interface to the Host

4.2.1 OS-Support Protocol

4.2.2 Print/Debug Protocol

4.2.3 Shared Memory Structure Between LANai and Host

4.3 Communication Protocols

4.3.1 Design of Communication Protocol in lanai-emx

4.3.2 From LANai to the Host

4.3.3 From LANai to the Network

4.3.4 From Host/Network to the LANai

4.3.5 Reading/Writing from/to the Myrinet Network

4.4 Message Routing

4.4.1 Data Structures Used for Routing

4.4.2 Routing Information Protocol

4.5 Other Issues

4.5.1 Emerald over Myrinet Vs. Emerald over IP over Myrinet

4.5.2 Issues Concerning Hardware Resets

4.5.3 Modifications to the host-emx

4.6 Summary

5 Performance

5.1 Micro Benchmarks

5.1.1 Computation Intensive Micro Benchmarks

5.1.2 Communication Intensive Micro Benchmarks

5.1.3 Discussion
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Application 1: Kilroy</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>Application 2: Location Service</td>
<td>73</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Design of the Location Service</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Experimental Setup</td>
<td>76</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Efficiency of <em>Emu</em></td>
<td>76</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Applications that could Use the Location Service</td>
<td>80</td>
</tr>
<tr>
<td>5.4</td>
<td>Discussion</td>
<td>80</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions and Future Work</td>
<td>82</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusions</td>
<td>82</td>
</tr>
<tr>
<td>6.2</td>
<td>Effort Required in Creating <em>Emu</em></td>
<td>83</td>
</tr>
<tr>
<td>6.3</td>
<td>Future Work</td>
<td>84</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Memory Fragmentation Problems</td>
<td>85</td>
</tr>
<tr>
<td>6.3.2</td>
<td>A More Precise Debugger for the LANai</td>
<td>85</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Performing GMS lookups on the LANai</td>
<td>86</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Support for Active Networking on the NIC</td>
<td>86</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Dynamic Routing Application</td>
<td>86</td>
</tr>
<tr>
<td>6.3.6</td>
<td>HPJava Engine as Framework for <em>Emu</em></td>
<td>87</td>
</tr>
<tr>
<td>6.4</td>
<td>Final Remarks</td>
<td>87</td>
</tr>
</tbody>
</table>

**Bibliography** | 89 |
List of Tables

3.1 Results of *bw_mem.cp* and *hswap* Benchmarks 35
3.2 Results of *Imbench* Test Suite 36
3.3 UDP_STREAM Testing Using *netperf* 38
3.4 UDP_STREAM Testing Using *netperf* (turning off the checksums) 39
3.5 TCP_STREAM Testing Using *netperf* 40

5.1 Computation Intensive Micro Benchmarks 69
5.2 Communication Intensive Micro Benchmarks 71
List of Figures

1.1 Emerald Interpreter on the Network Interface Card (NIC) ........... 7

3.1 Block Diagram of the Myrinet/Host Interface ....................... 28
3.2 Layout of Myrinet Network ........................................ 41
3.3 Top View of Dual 8-Port Myrinet-SAN Switch ....................... 42
3.4 Myrinet Cluster Connections ....................................... 43
3.5 Emerald Virtual Machine .......................................... 46

4.1 Memory Layout on the LANai ...................................... 51
4.2 The Host Interface Protocol Memory Structure .................... 54
4.3 Pseudocode for Handling Requests from the LANai by the Host .... 55
4.4 The MCP Shared Memory Structure ................................. 56
4.5 Communication Protocol of the VM on the LANai .................. 58
4.6 Pseudocode for Sending Packets from LANai to Host .............. 59
4.7 Pseudocode for Sending Packets from LANai to Network .......... 59
4.8 Pseudocode for Processing Incoming Packets to the LANai ....... 60
4.9 Reading Packets from the Myrinet Network ....................... 61
4.10 Writing Packets to the Myrinet Network .......................... 62
4.11 Data Structure to Store Information for Each Switch .......... 63
4.12 switch_conn Data Structure ...................................... 63
4.13 Myrinet Routing Protocol ........................................ 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Kilroy on <em>Emu</em></td>
<td>72</td>
</tr>
<tr>
<td>5.2 The Locator Service Class</td>
<td>74</td>
</tr>
<tr>
<td>5.3 The Service Finder Class</td>
<td>75</td>
</tr>
<tr>
<td>5.4 The Location Service on <em>host-emx</em> of <em>Emu</em></td>
<td>77</td>
</tr>
<tr>
<td>5.5 The Location Service on <em>lanai-emx</em> of <em>Emu</em></td>
<td>78</td>
</tr>
</tbody>
</table>
Acknowledgements

A lot of people have influenced my work in various ways during my Masters Program at UBC. It would be impossible to remember all of them, but I will do my best.

First, sincere thanks to my supervisor, Norm Hutchinson for his invaluable support throughout my work. His constant belief in me and the freedom to explore my own ideas enabled me to gain confidence. Thank you, Dr. Hutchinson, for always listening with such patience when things were not going great.

Thanks to Alistair Veitch for answering my numerous questions while installing the Myrinet network and for reviewing the first draft of my thesis. Thanks are also due to Mike Feeley who was my second reader, and to Peter Smith for his reviews and for being a wonderful friend.

I would like to thank my family for supporting me through school, particularly two very special people, Fatima and Susai Peter. Without their help, it would have been impossible for me to venture off to Canada. I also extend my thanks to Stephen Sinnappan, Celine Stephen, Regina Simon and Pascal Simon.

Other people who have helped either technically or by just being there are Marie-France Beauchesne, Ian Cavers, Christoph Kern, Dwight Makaroff, Holly Mitchell, Gerald Neufeld, Kalpana Ramanujam, Saravanan Sanniyasi, Pradosh Sivadoss, Robert van Hulst, Dimitri Voulioris, Alan Wagner and Marcelo Walter.

Finally I would like to thank my parents and my husband Paul for being there for me through this.

MARGARET A.S. PETRUS

The University of British Columbia
August 1998
To my Mom and Dad.
Chapter 1

Introduction

1.1 Motivation

The changing technology trends for both local area networks with faster processors, and high speed networks with low latency, make the lagging I/O bus the bottleneck in achieving better throughput in the network systems. With the advent of high speed networks, this I/O bottleneck is a serious concern which limits the full potential of the network. Protocol processing on the network processor, rather than on the host’s CPU, can avoid the unnecessary use of the I/O bus which limits the speed of the processing in the network.

The network interface cards (NICs) for high speed networks should have processing capabilities to support such functionality. The new NICs have processors that remain idle except when executing code to control the flow of data between the host and the network. For example, the Myrinet [BCF+95] NIC contains a general purpose processor which can execute user programs. This is a unique feature because it allows programming the NIC of a high speed gigabit network. This feature could be exploited by moving services to the NIC which would avoid host overhead and I/O transfers.

Migrating services to the network adaptor makes use of the idle processing
power in the adaptor that would otherwise be wasted if it was just used for sending and receiving messages. In addition, by moving some functionality to the network adaptor, decisions can be made without involving the host. This would prevent unnecessary memory copies, I/O traversals, interruptions of the host and the associated context switches. Thus the host can use its processing power, without interruption, for more useful work.

There are several network intensive services that could benefit from this exploitation of the network processor. Two such examples are dynamic routing in an active networking scenario and global memory management in a clustered environment. In the latter case, low latency plays a major role in performance throughputs.

Service migration could be extended to building active networking [TSS+97, WT96] applications. Active Networking is moving executable code in the network rather than just moving passive data. The active code gets executed at routers, which support such services, as it moves in the network. For example, one of the applications that would benefit from this service on the network interface card is dynamic routing, i.e., routing decisions made by code that migrates among the routers. A virtual machine supporting active networking and running within the network adaptor would help the Internet evolve dynamically, without involving the host or constant manual intervention, during failures such as when a router goes down. More important is the introduction of a new routing functionality, where code supporting new routing protocols can be injected into the network dynamically.

Another area that will benefit from this approach is the global memory system (GMS) [Fee96] for a network of workstations. In GMS, global network-wide page replacement decisions are made. When a page-fault occurs, the replaced page is sent to global memory rather than to disk, and the globally least valuable page is discarded from memory, sending it back to disk if necessary. Page faults can thus sometimes be satisfied from the memory of a remote processor. By migrating some
code to the network adaptor, the necessary pages can be DMAed from the remote storing node's memory without OS intervention. This would result both in faster memory access and lower overhead just by migrating a small part of the service to the network interface card.

### 1.2 What is Service Migration?

Service migration involves moving some process, that may be currently active, to some other processor in the network. This technique is adopted for various reasons, the most common of which are to achieve speedup by migrating to faster processors, and for load balancing. Utilizing idle machines or not so overloaded machines leads to the completion of the task faster, while at the same time taking advantage of available idle computational power. Another advantage is reducing network bandwidth by moving two processes that are communicating a lot but are on different machines to the same machine [Smi97].

Research on service migration has led to many research-oriented or commercial systems [Che88, Dou91, BHJ+87]. *Emu*, the system implemented as part of this thesis, falls under object migration systems [BHJ+87, LJP93]. However, unlike earlier object migration systems, objects in *Emu* may also execute in the network adaptor rather than just on the central processors. This is achieved by running a fully functional interpreter in the network adaptor. It is believed to be the first such system running within a network interface. A preview of the different migration systems is given in the next chapter, with examples from homogeneous, heterogeneous and object migration systems.

### 1.3 Problem Definition

*Emu* is a system composed of two distributed object oriented virtual machines on a single host, one executing on the host processor and the other on the Myrinet NIC
which supports a general purpose programmable processor. Moving an existing interpreter to the Myrinet NIC, to be executable by the bare network processor, is quite unlike porting software for a central processor of a different architecture.

There are many challenges involved such as the RAM limit of the network adaptor, the lagging speed of the network processor when compared to the central processor, the programming environment available, handling of system calls due to the lack of an OS on the network adaptor, etc. The interpreter should also handle traffic from both host and network, each of which functions in a different manner in addition to servicing its own processes. The processing of the incoming objects may lead to the creation of new objects to be injected back into the network or to the host.

During implementation, each of the above mentioned limitations had to be handled to arrive at a fully functional interpreter. Based on the results, the following thesis statement can be derived:

*A programmable high speed NIC can service incoming network packets independently without host assistance and can improve aggregate throughput in a network of workstations if non-computation intensive jobs are offloaded to the NIC. Running a distributed object-oriented interpreter on the NIC for the *Emu* system extends this functionality further by providing dynamic migration of objects back and forth at runtime.*

This virtual machine on the network interface enables services to be migrated to the network interface as well, as the traditional migration method of moving to other processors. This work has introduced a new meaning to service migration.

1.4 The Network Environment

*Emu* is implemented for the high-speed Myrinet network. Myrinet is a full duplex 1.28+1.28 gigabit-per-second link. For distributed computing or clustered comput-
ing, a Myrinet System Area Network (SAN) or Myrinet Local Area Network (LAN) would be the ideal choice for the following reasons [Myr97c]:

1. Speed: connects workstations or PCs through full duplex 1.28+1.28 gigabit per second links and low latency cut-through switches compared to 10Mbits/sec or 100Mbits/sec Ethernet or 155Mbits/sec ATM.

2. Convenience and Robustness: allows any network topology, unlike unswitched Ethernet. Being a switched network, it supports source routing and allows dynamic reconfiguration of the network which (a) is a convenience for installation and (b) provides fault-tolerance. With the routing tables being stored in the SRAM of the network chip, they can be modified on the fly.

3. Network throughput: being a switched network, Myrinet can carry many packets concurrently, each traversing the network at 1.28 gigabit/second. However, unlike Ethernet and FDDI which share a common communication medium, the total traffic of Myrinet increases with the number of hosts since the ratio of hosts/switch is constant.

4. Versatility: like Ethernet, Myrinet can carry packets of many types or protocols simultaneously as its packets can be of varied length, and can encapsulate other types of packets such as IP without an adaptation layer, in contrast to, for example, the AAL5 adaptation layer of ATM.

The Myrinet interfaces include a LANai processor - a general purpose custom VLSI chip which has a 32-bit RISC processor. The programs it executes are called Myrinet Control Programs (MCPs) [Myr97d]. The MCPs supplied by Myricom, Inc., handle interactions with the network and host processors. Any MCP must run within 1MB of RAM which is what the LANai adaptor supports. Emu's MCP running within the LANai is a complete distributed object oriented interpreter in addition to controlling the flow of data between the host and the network with the help of the DMA engine on the Myrinet NIC.
The experimental framework consists of the Myrinet SAN network composed of 16 Pentium-II 266MHz, 128MB RAM, 512KB cache, 440FX chipset processors and 16 M2M-PCI32C Myricom boards. These boards are Myrinet-SAN/PCI interfaces, PCI short cards which support 1MB of memory. The benchmarks involved in the selection of the processor are discussed in Chapter 3.

1.5 The Emerald Approach

Emerald [BHJ+87, JLHB88, RTL+91, BHJL86], developed at the University of Washington, is an object oriented programming language that was designed to be used in a distributed environment. Hence, it inherently provides for abstraction, moving objects from one machine to another, locating those objects easily because of their unique global identifiers, and other basic distributed services. It is a complete object-migration system and so provides all the services necessary for migration and can be easily extended to provide services for active networking. The language also provides replication and transactions, thereby supporting fault-tolerant and highly available applications.

Emerald provides fine-grained mobility. It supports both small passive data objects and active process objects. In addition to the traditional advantages of migration such as load sharing, fault-tolerance, availability, etc., fine-grained mobility in Emerald also supports the following [JLHB88]:

- Provides a protocol for moving data (objects) from one machine to another without requiring the user to provide marshalling/unmarshalling of the data.
- Improves remote invocation performance by moving parameter objects to the remote node to reduce dependability on the source during the duration of the remote execution.
- It moves objects to sites where references exist in order to simplify distributed garbage collection.
Emu focussed on building an interpreter for the Emerald byte codes that could execute on the LANai (lanai-emx) and communicate with the one on the host. This interpreter provides features to support service migration and active networking via objects. It is written in the form of an MCP and is loaded by the host into the memory of the Myrinet interfaces.

In Emu, there are two separate Emerald interpreters per host: (a) one on the host (host-emx), and (b) one on the Myrinet interface (lanai-emx) as shown in Figure 1.1. The Emerald emx running within the network interface handles traffic from the network without assistance from the host. lanai-emx is not a simple MCP just processing messages from/to the network and the host. It can actually interpret executable code without involving the host. In other words, a complete virtual machine is running within the LANai network interface. The host-emx and lanai-emx communicate via a shared memory structure on the LANai. An application to illustrate its functionality was designed and implemented as discussed in Chapter 5.

Previous research on migrating functionality, other than basic mux/demux of messages, to the network adapter is discussed in the next chapter.
1.6 Thesis Structure

The layout of the following chapters are outlined below:

- Chapter 2 reviews prior research on systems that deploy service migration and migrate functionality to the network interface cards. This is followed by a discussion of the languages considered for the framework of *Emu*. The drawbacks of these languages that initially seemed a better choice than Emerald are also explained. It finally concludes the contributions of *Emu* that are different from prior work.

- Chapter 3 provides the background for understanding the design of *Emu*. It discusses the capabilities of the LANai 4 processor that runs on the Myrinet NIC, and the Myrinet API available. It explains the choice of Pentium-II for the Myrinet testbed at UBC and the benchmarks that influenced its usage. It also provides some background about the Emerald programming language that is used as the framework for *Emu*. The original model of the Emerald interpreter is reviewed to understand the complexity of the changes made in Chapter 4.

- Chapter 4 discusses the design and the implementation of *Emu*. There are two virtual engines, one on the host *host-emx* and the other on the LANai *lanai-emx* with the latter consuming most of the time spent in the construction of *Emu*. It explains the protocols designed to circumvent the limitations posed by the LANai environment, the communication protocols for the new Myrinet network, the complex mux/demux needed for *Emu*, the routing protocols, and modifications made to the *host-emx* to assist the *lanai-emx* with OS support. It also discusses some miscellaneous issues that arose during the design of *Emu*.

- Chapter 5 discusses the performance of *Emu*. Micro benchmarks that were
run on Emu help analyze the slowdown on the LANai for computation and communication intensive operations. The slow down is because the network processor is slower than the host processor by a factor of 8. This is followed by a review of the application designed to show the robustness and usefulness of the Emu model, and the efficiency of such a system.

- Chapter 6 summarizes the thesis and provides suggestions for future work.
Chapter 2

Related Work

This chapter discusses prior work related to the thesis. The first section reviews some existing migration systems, focusing on significant features relevant to this thesis.

A couple of recent systems that migrate some fixed set of functionality to the network interface card are reviewed in Section 2.2. Executing code on the NIC is a considerably new approach and not much research exists in literature.

Section 2.3 discusses the languages considered to build the virtual machine, including some active networking systems/languages as these inherently support mobility. It also draws out those features which make them an impossible choice to be used on the LANai adaptor. The chapter concludes with a summary of how this thesis is different from prior approaches.

2.1 Existing Migration Systems

Process migration research was first undertaken in the late 1970s and a considerable number of experimental systems have since been implemented and tested. To migrate a process from one machine to another, the following steps are taken:

1. Freeze the process at the source.
2. Move all the process' state to the destination host.
3. Resume execution from the point where it was stopped at the source.

This section is a brief overview of process migration in homogeneous systems, object migration systems and some examples of heterogeneous migration systems.

2.1.1 Traditional Migration Systems

Homogeneous migration is moving processes among machines of similar architecture. Most of the homogeneous process migration systems follow the general protocol, but with significantly different techniques used to address the following design goals.

- How should the virtual memory be transferred from host to destination?
- When should the migrant process' execution be stopped?
- What degree of transparency should be achieved?
- How should residual dependencies on the source, or any other intermediate hosts be reduced?
- What are the different migration policies used, i.e., which process to migrate, when to migrate it, and where to migrate it?

The V [Che88, TLC85] system developed at Stanford supports migration to allow off-loading of programs to idle workstations in order to increase available computational power. The V migration facility considers these three issues as vital to its design: (1) a network-transparent execution environment, (2) minimal interference due to migration, and (3) no residual dependencies on its previous host. In order to not increase execution time due to migration, V does pre-copying of the process' state while continuing execution. This leads to two or three copies of dirty pages before the host is finally migrated to the destination to resume execution.
The MOSIX multicomputer operating system [BSW89], developed at the Institute of Computer Science, the Hebrew University of Jerusalem, is an extension of Unix. It is designed for a network of workstations (NOW) [PCA95], and supports a comprehensive resource sharing policy. This policy depends on various factors such as load sharing, memory sharing and IPC optimization. Algorithms to calculate these factors are executed independently on each workstation. Later, they exchange results with algorithms from other workstations for information gathering and dissemination. Migration decisions are made by this policy with partial knowledge about the state of the network, and do not depend on complete information of the state of the NOW, or any particular workstation. Such a migration system where decisions are not completely centralized, especially on a low latency network such as the Myrinet [BCF+95], scales well to large configuration.

Charlotte [AF89], work done at the University of Wisconsin-Madison, makes distinct separations of policy (when to migrate which process to which destination) from mechanism (how to detach, transfer, and re-attach the migrant process). Their design of keeping the policy (in the user space) separate from mechanism (in the kernel space) allows various efficient and flexible policies like load sharing, load balancing, etc., to be used. This also obviates the need to recompile the kernel each time the policy is changed. While MOS [BL85] and Locus [WPE+83] freeze processes when selected for migration, and V freezes near the end of transfer, Charlotte freezes somewhere in between (only when the context is marshalled and transferred).

Process Migration in Demos/MP [PM83] was implemented efficiently with no system changes due to such characteristics as a uniform and location independent communication interface, and the participation of the kernel in sending and receiving operations in the same manner as a normal process. It is a message-based operating system (communication is the most basic mechanism) with messages being sent using links to specify the receiver of a message. So when processes migrate, some level of transparency is achieved using message forwarding and updating of links. Unlike V
and Charlotte, execution of processes stop when they are chosen for migration.

The bottleneck in process migration is the transfer of the possibly large virtual address space of the process being migrated. To alleviate this problem, Accent [Zay87] uses a lazy transfer copy-on-reference approach. It uses different variations of virtual memory copy: (a) Pure-copy (complete physical copy of the address spaces), (b) Pure-IOU (logical transfer of address space, which requires only portions of the address space to be physically transferred), and (c) RS (moving resident set during migration). Although Pure-IOU and RS hold a clear advantage in the address space transfer of migration, they suffer longer remote execution times when compared to Pure-copy. Hence only processes which access a small portion of their address spaces at the new site are best suited to the use of the copy-on-reference technique.

Sprite [Dou91], developed at the University of California, Berkeley, migrates processes at two particular times: (a) exec system call when a resource-intensive program is about to be initiated, and (b) eviction of foreign processes when user returns to his workstation. This is one of the few migration systems to receive extensive practical use besides Locus [WPE+83] and MOSIX [BSW89]. Sprite's designers considered trade-offs among transparency, residual dependencies, performance and complexity in the design of their system. Sprite uses a different form of lazy copying from Accent which also makes use of its existing network services. Backing storage for virtual memory is implemented using ordinary files which are stored in the network file system and are accessible throughout the network. When a process is migrated, the source freezes the process, flushes the dirty pages to the backing files and discards its address space. The process then resumes execution at its destination host with no resident pages, and pages are loaded from backing storage as needed using the standard paging mechanism.

Condor [LTBL97] is a distributed batch processing system for Unix developed at the University of Wisconsin-Madison. Condor makes use of idle workstations by
migrating processes via checkpointing of their state [LA89, LS91]. The mechanism
used is to write all of the process' state information (data, stack, information about
open files, pending signals and CPU state) into a file, and to use this information
to restore state on the destination machine at restart time. Sprite and V have been
carefully designed and implemented to accommodate migration, which is not the
case here. Hence not all processes can be migrated in Condor. For example, as
there is no way of saving all the state necessary for every kind of process, there is
no way of migrating a set of communicating processes.

A survey of systems providing process or object migration facilities is dis­
cussed by Mark Nuttall in [Nut94] and a more detailed survey is presented by
Milojicic et al. in [MDP+96]. Some of the above systems would have been ideal
choices if not for the two constraints: (a) the processes are too big to fit within 1MB
of memory and (b) they do not support heterogeneity. The 1 Mbyte memory limit
of the LANai network interface narrows down the choice of the system to use.

2.1.2 Object Migration Systems/Languages

This section discusses systems that migrate objects instead of processes. An object
is usually a part of a graph of references. Hence when an object is to be migrated,
one can either move just that object, some set of objects in the graph, or the entire
graph. Moving only the object in question may not be appropriate. By moving an
application specific set of objects, unnecessary remote invocations can be avoided.
A common problem of these systems is determining what comprises the persistent
state of the object so it can be encapsulated before migration. The most typical
solution is to provide a description of each object's interface using an Interface
Definition Language (IDL), as used by DCE++ [SM], or use routines to marshall
an object's state into a buffer. However, in spite of the various research systems, no
object migration system has achieved the commercial acceptance as that of a few
process migration systems.
Emerald [BHJ+87, JLHB88, RTL+91, BHJL86], discussed in chapter 1, has the advantage of being designed to support mobility unlike other systems which have had migration added to them after the fact. It also has additional benefits over process mobility because the overhead of an object is commensurate with its complexity. Hence mobility in Emerald provides a relatively efficient way to transfer fine-grained data from source to destination. Emerald has been designed for a LAN of not more than 100 nodes. Emerald’s designers used templates instead of tag bits (used in Distributed Smalltalk [SB88]) to distinguish data types, such as integers, from object references which make mobility easier. Performance of mobility was traded-off in favor of fundamental operations like local invocations. However, it is a very efficient design which explains the reason for its techniques being adapted in many of the systems discussed below.

The DOWL [Ach93] distributed object-oriented language, developed at the University of Karlsruhe, Germany, adopts most of Emerald’s mechanisms:

- Mobility of objects is restricted by fixing (move an object to another machine and fix it there), and unfixing them (the object is free to move again).

- Attach objects; by using the reserved word attached in front of the closely related data of the moving object, all the attached objects also move to the destination site of the moving object when it migrates. Attachment is transitive but not symmetric.

- Besides passing-by-object-reference, support for pass-by-move (move argument object permanently to the invoked object’s location) or pass-by-visit (move argument object temporarily to the invoked object’s location) reduces the number of remote invocations.

If one of the object’s methods is activated, it can still migrate by moving the invocation context along with the object. Execution resumes at the target node and operation results are transparently delivered to the caller.
DCE++ [SM], developed at the University of Karlsruhe, Germany, is an extended distributed object-oriented environment on top of DCE (The OSF Distributed Computing environment which is becoming an industry standard for open distributed computing). It supports mobility based on concepts introduced by Emerald [JLHB88], Amber [CAL+89], Arjuna [SDP91] and Amadeus [HC91]. Migrated objects can be accessed in a uniform way by clients, and concurrent migration and invocation requests are synchronized. A migrating object leaves a proxy at its former location which results in forwarding chains of proxies when it gets invoked. The location of the object gets updated on the proxies upon stepwise return of the call. However, the birth-node of the object is always updated upon every migration and is used in case the forwarding chain breaks.

Comet [MV93], developed at Katholieke Universiteit Leuven, Belgium, uses a multi-dimensional migration approach. The designers of Comet consider that location is not the only attribute whose change in value is worth considering during migration. Other attributes such as state representation, access limitations or failure handling semantics also change during the lifetime of an object. Migration of objects in Comet is considered to take place in an N-dimensional space where N is the number of different attributes or flavors, and hence object migration is expressed in terms of coordinate changes along each of the axes.

Cool [LJP93], is an object-oriented kernel implemented above the Chorus [RAA+88] microkernel which provides mechanisms to instantiate, migrate and store objects. The designers of Cool have adopted a layered approach. It is implemented as three layers (COOL-base, generic runtime (GRT) and language specific runtime) above the Chorus microkernel. It was designed with an aim to provide a generic set of abstractions for better support between existing and future object-oriented systems, languages and applications. It also supports passive objects (persistent memory) and activities (threads of execution). Activities can migrate from one object to another, and objects can contain several concurrently executing activities.
A survey of object mobility is given in [Nut94]. Many of these systems merge two or more high level language calls into a single feature (for example, the visit primitive in Emerald is a combination of locate and move), thereby reducing the probability of errors (as the underlying system takes care of the details) and the number of remote invocations.

2.1.3 Heterogeneous Migration Systems

There are some systems supporting migration of processes among processors of varied architectures or those that run different operating systems. In addition to load or resource sharing, migration among different architectures has the advantage that some computations are quite efficient in certain specific architectures.

The process-originated migration system [Shu90, DRS89], developed at the University of Colorado at Colorado Springs, is a prototype built on the existing homogeneous migration features of the V system. Their approach is to eliminate many portability problems by having a compiler resolve the intricate migration issues. Since a process requests its own migration, the request occurs at a reasonably clear point in its execution. In addition to modifications to the linker and the compiler (which take care of the translation of text, heap, data, stack, process control block, etc.), the operating system was modified to include a migration server to satisfy the process-initiated migration requests. Their data message is simplified because at the time of migration request, all the needed state information is in memory, and none is in the machine registers of the source machine. The designers of the system have traded complexity of representation, and overhead at migration or page fault time, for pure runtime speed.

Heterogeneous Process Migration by recompilation [TH91], introduces the concept of a source level program being automatically created, transferred to the destination machine; and then recompiled. When the program is executed, it restores the state of the process at the destination and then continues the migrated
process' execution from the correct location. This approach is similar to that of the previous systems in that it deals with compiled programs rather than with interpreted source programs. However, unlike the previous case, the details of machine and compiler-dependent data translation procedures are hidden in the compilers and in the debuggers by employing recompilation techniques. The main disadvantage is that migration cost is greatly increased because of source code compilation.

The Emerald system [BHJ+87, JLHB88, SJ95], supports fine-grained object and thread mobility among heterogeneous computers with threads executing native code. In Emerald, threads follow objects as they move around. The translation of machine-dependent code and data are dealt with efficiently by making changes to the compiler. These changes enable the runtime system to transform the machine-dependent code or data into a machine-independent format. However, translation of differently optimized versions of the code running on different architectures has not been dealt with. This is because the latter requires the runtime system to be able to invoke parts of the compiler at runtime. Migration of active threads was implemented with the use of bus stops [SJ95]. A bus stop represents a program counter value that is the same for all processor types, and hence can be considered as a safe migration point. This makes the code generator free to optimize all code between these bus stops.

The Heterogeneous Migration Facility (HMF) [BVW95], developed at the Department of Computer Science, Dartmouth College, requires explicit registration of the data to be migrated. The migration library should be linked with the migration program in order to provide procedures for automatically migrating processes and checkpointing data. Their system is based on a graph model for single process migration. Migration costs are based on the vertex and edge weights of the graph that require complex analysis of many factors, and this is a serious disadvantage to this model.

The Tui system [Smi97], is different from the above systems in that in all the
previous cases, the program must be written in a type-safe language or in a type-safe subset of a language. Tui supports heterogeneous migration of common languages that are not type-safe. It also allows an external agent to make a migration request in addition to process-originated migration, which is the case in most prior systems. While most of the above systems incorporate the marshalling code into the migrating process itself, Tui is a completely separate program.

2.2 Moving Functionality to the Network Adaptor

Research on migrating functionality to the network adaptor in order to increase processing speed, reduce round-trip latencies, etc. began in 1980s. The network processors supported complete implementations of protocols, such as TCP/IP, that were not provided by the operating system. But as operating systems adopted these protocols, the functionalities were dropped from the NIC. This was because the marginal improvements in reducing host overhead did not justify the high cost of running these protocols on the network processor with 10Mbits/sec Ethernet. Most of the work done was commercial or proprietary, and hence not available in literature. But with the birth of research into high speed networks, these ideas were revisited. The following two sections discuss work recently done in this area.

2.2.1 U-Net

With the advent of ATM [dKdP88], von Eicken et al., from the Department of Computer Science at Cornell University, developed U-Net [vEBBV95], a user-level network interface for parallel and distributed computing. The goal of U-Net, implemented using off-the-shelf ATM hardware, is to remove the kernel from the communication path while providing full protection, so that user processes are only limited by network capabilities. It provides a virtual view of the network interface to the user processes so that each process has the feeling that they own the network adaptor.
By removing the kernel from the communication, an extra memory copy is avoided when sending or receiving messages from the network. This is because user process memory can be mapped to the network adaptor’s memory and direct DMA can access user memory locations. U-Net not only supports traditional inter-networking protocols like TCP and UDP but also novel ones like Active Messages. The TCP and UDP protocols implemented using U-Net have latencies close to the raw minimum, and throughputs close to the raw maximum. U-Net active messages, a prototype of the Generic Active Messages (GAM) [ea95], have round-trip times which are only a little above the absolute minimum.

The U-Net model explained above was extended further, by von Eicken et al., into a U-Net/MM architecture [BWvE97]. This allows memory to be copied from and to any part of the application’s address space. In other words, memory is not physically pinned down as in the previous case. By integrating a translation look-aside buffer in the network adaptor, which coordinates with the operating system, network buffer pages are pinned and unpinned dynamically. This architecture has been demonstrated using both ATM (PCA-200/Linux implementation) and Ethernet (DC21140/Windows NT implementation). While the former performs both address translation and mux/demux of the messages in the network adaptor, the latter relies on the host for both functions. Their evaluation leads to the conclusion that while performing mux/demux of messages on the network interface avoids buffer copies on receive, it is better to perform address translation on the host to reduce communication latency and complexity.

2.2.2 Hamlyn

Hamlyn [BJM+96], developed at Hewlett-Packard laboratories, is a sender-managed interface architecture optimized for clustered computing using Myrinet. The latency (both hardware and software) is just a few microseconds and provides full protection among the different applications with little or no OS intervention. Being a sender-
managed design, messages are only sent when the sender knows that there are sufficient buffers at the receiver sites. So it needs to maintain some information about memory space availability at the destination, taking into account buffer overruns and retransmission under heavy loads.

To achieve low latency, direct access to the Myrinet interface is given to the different applications with no OS intervention. So applications can DMA memory between EBUS (External-access Bus) and LBUS (LANai Bus) without any interference from the kernel. To achieve high bandwidth, Hamlyn supports a scatter-gather direct memory access (DMA) capability that frees the host for other operations during long transfers. Since Myrinet [BCF+95] has very low (almost negligible) error rates, flow control, and a physically secure network, messages need not be held for retransmission and packet acknowledgements are unnecessary.

However, in these systems only a fixed set of functionality has been ported to the network interface, with no techniques to extend them at runtime. Unlike these systems, Emu supports a full-fledged virtual machine running within the network adaptor, thereby allowing extensions at runtime. This is believed to be the first such virtual machine on a network interface that can mux/demux messages, DMA memory, build objects, execute them, etc. within 1MB of the network interface memory with no host intervention.

2.3 Other Languages Considered for the Interpreter

Prior to choosing Emerald as the basis of Emu, other languages were studied. Initially, Java seemed to be the ideal choice since it is popular and commercially accepted. Another venue of thought was choosing an active networking language like PLAN. Such a language would support mobility, and thus can be easily modified for service migration. The following sections briefly discuss the advantages and the disadvantages of each of the languages considered:
2.3.1 Java - Language of the Internet

The features of Java [Boo96, BMKK97] that are advantageous for mobility are (a) its excellent exception handling mechanism, (b) it is object oriented, (c) it provides support for encapsulation of objects, (d) it has no pointer references, and (e) it provides marshalling and un-marshalling routines.

Unfortunately Java's interpreter is 50 times slower than C. It does not provide migration primitives or compiler runtime support on different architectures. For active networks, Java is probably much too bulky, as the virtual machine comes with a much larger overhead than C. Hence, the advantage of the gigabit network might be lost. But the main reason for not choosing Java is the massive size of the code, and the high rate of increase of its code size. It would have been almost impossible to fit the code including heap and stack size in 1MB of LANai memory. Even removing unnecessary classes might not have solved the problem.

2.3.2 TCL

TCL [Ous93], a prototyping language, supports many of the features of conventional procedural languages, including variable assignment, procedure calls, control structures, and in addition it has very easy access to graphical widgets. TCL is less strict in its handling of data than many conventional languages, and does not provide the data hiding or modularity features of languages with strong support for abstract data types, such as ML, Modula2, Modula3, or Ada. Thus it may not be an appropriate language in which to write large and complex programs, which are required to be highly reliable and robust.

2.3.3 Existing Active Networking Systems/Languages

The concept of active networking involves moving executable code around the network, unlike the traditional method of moving just passive data. This code can be executed at various routers by running an interpreter on them. Recent work on
active networking includes [TSS+97, TW96, BCZ96a, BCZ96b, TGSK96, WT96]. Active networks allow users to inject customized programs into the network. These programs can help in dynamic modification of the network. For example, active programs can assist routers and gateways in making intelligent routing decisions based on the network traffic at a particular time, as well as dealing with failures due to network partitioning without manual intervention.

Languages for active networking include Telescript [tel96b, tel96a, tel96c, tel96d], PLAN [GHK+97, Kak97, MH97, Moo97], Wave [Sap95, Sap88], Messengers [Bic95, FBDM96], and others.

**Telescript**

Everything in telescript [tel96b, tel96a, tel96c, tel96d] is an object. It is an interpreted language and has two levels: high telescript has an object oriented syntax and compiles to low telescript which has a postfix syntax for stack-based interpretation. Telescript supports preemptive prioritized multi-tasking of process objects (which is an object with a life of its own), migration, “meeting locations” for agents, persistent objects and authentication.

Telescript’s source code could not be obtained easily for experimentation owing to the fact that it has become a commercial product. Odyssey, its successor, is composed of Java class libraries to support distributed applications, and hence poses the same problems as mentioned in section 2.3.1.

**PLAN**

PLAN [GHK+97, Kak97, MH97, Moo97] is a script-like functional language similar to ML. It is designed to satisfy the following two functions in an active network: (i) Resource discovery: the location, invocation, or installation of services, and (ii) Diagnostics: analysis of network status or behavior. PLAN consists of two levels. The PLAN level which is the lower level provides services for the above two functions,
and consists of PLAN programs which are distributed to all PLAN routers. The SERVICE level, which is the higher level, includes services that are available only to certain routers. Therefore, it may require authentication, unlike services from the lower level, for their use. It usually includes a rich collection of protocols which provide functionality for an active network.

Programs are carried in the packets and executed at the different routers. However, PLAN packets do not communicate with each other directly as in Tele-script, but only through application programs at endpoints or service calls on the routers. Although PLAN was successfully installed, it could not be ported to the Myrinet. The PLAN interpreter grows to over 6MB (6776KB) of memory to handle simple programs such as ping and traceroute, and would clearly grow with more complicated services. This was because current PLAN implementations are built on top of Java or Caml, and the memory limitations of the LANai adaptor (1MB) make such sizes unacceptable.

2.4 Contributions of this Thesis

Emu has made possible the running of a heterogeneous object-oriented distributed virtual machine on a NIC. The interpreter running within the LANai network adaptor supports all the features of a regular virtual machine running on the host. In Hamlyn and U-Net, functionality migrated to the network interfaces are statically fixed, unlike Emu which can be extended at runtime. Emerald, an object oriented migration system was selected as the framework due to the limitations posed by the other migration and active networking systems discussed.

The LANai being a different environment, makes the homogeneous and heterogeneous systems discussed unsuitable for running on the network processor which is slower than the host processor by a factor of 8. Homogeneous process migration systems have introduced the fundamental concepts of migration, but these systems are course-grained because the processes are big and their state cannot be easily
marshalled as is the case with objects. It would be impossible to run one of the systems within 1MB of RAM. Also, these systems do not support heterogeneity and the LANai is quite unlike regular processors with a very unfriendly programming environment. In the case of the discussed heterogeneous migration systems, the limitation is the very slow recompilation times which would reduce throughput in a gigabit network. The usefulness of such a network would be lost.

The active networking systems, which provide the support for migrating functionality to the NIC, have the wrong languages for the LANai adaptor. The memory limitations of the LANai adaptor force the virtual machine to be small and optimized such that its code, stack and heap size can run within 1 MB. For example, PLAN did not satisfy this constraint. Moreover, a case study of PLAN implied that a more efficient active networking scenario can be deployed with a much simpler distributed language. Java’s main disadvantage was its ever-increasing code size, which would have otherwise made it the first choice due to its wide usage. C and C++ provide no support for mobility, marshalling or unmarshalling routines, and have to be programmed at the socket level, etc. These fundamental functions were inherently provided by the Emerald language which made it a better choice.

The Emerald virtual machine has been successfully ported and it runs within 1MB of memory. It is object-oriented which makes migration easier and it supports heterogeneity. Although Emu has been developed using techniques discussed in the research above, it is believed to be the first system on a network interface card supporting functionality which can expand at runtime.
Chapter 3

Background

Emu was implemented for the Myrinet network interface cards. A review of the network hardware, and the software interface between the LANai and the host is required to understand the design issues of Emu. The Emerald language which is used as the framework for Emu is also reviewed. The following is the layout of this chapter.

Section 3.1 reviews the LANai 4 processor and the Myrinet API. Section 3.2 discusses the various host processors considered for the Myrinet network. This is followed by the results of the experimental benchmarks to support the choice of Pentium-II for the Emu system. Section 3.3 explains the Myrinet layout, the Myrinet/PCI interface card, the Dual Myrinet switches and the cable support. Section 3.4 discusses the language support for mobility followed by a background on the Emerald interpreter design and implementation, and Section 3.5 summarizes.

3.1 Myricom Technology and Details

Myrinet [BCF+95] is a local-area network (LAN) designed for packet communication and switching within massively parallel processors (MPPs). The characteristics that distinguish Myrinet from other LANs are (a) high data rates owing to its full-duplex
pair of 640Mb/sec channels unlike the uni-directional Ethernet channels, (b) very low error rates (almost negligible), (c) unlike unswitched Ethernet and FDDI LANs, the Myrinet topology is scalable and the aggregate capacity grows with the number of nodes, (d) the regularity of its network topology avoids deadlock because it allows for simple algorithmic routing, (e) it uses flow control and error control on every link, and (f) it uses low-latency cut-through switches. Various parts of the following subsections contain details, about LANai processors and host/LANai interface support, which were taken from the Myrinet guide [Myr97a] and the Myricom website ("http://www.myri.com/").

3.1.1 LANai 4 Processor

Each network interface card contains a LANai 4 processor, which is a programmable communication device that provides the interface to the Myrinet LAN or SAN network. As shown in Figure 3.1, it consists of an instruction interpreting processor and a packet interface which together make up the LANai core, as well as a DMA/Checksum engine, the Myrinet-link interface and the host/IO interface. The arrows in the figure indicate the flow of data between components.

The LANai processor [Myr97a, BCF+95] is a 32-bit dual-context machine, with 24 general-purpose registers. There are two contexts. The user context can be interrupted by events that cause the processor to switch to the system context. The system context is uninterruptable but it can be explicitly switched to user context by executing a `punt` (assembly) instruction. The three special registers: (a) Interrupt Status Register (ISR), (b) Interrupt Mask Register (IMR), and (c) External Interrupt Mask Register (EIMR), control interrupt generation. Packet-interface special registers and interrupt-control special registers are all memory-mapped except for IMR.

All the memory-mapped registers can be accessed by both the LANai processor and the external-access bus (EBUS). IMR is an internal on-chip LANai register
and hence not accessible from the EBUS. The LANai bus (LBUS) operates at twice the chip-clock speed, i.e., two memory cycles for every clock cycle. The addresses of both LBUS and EBUS are byte addresses and the byte order is big-endian. Words (32-bits) and half-words (16-bits) must be aligned on the LBUS. The LANai ignores least-significant bits of the address that would make it non-aligned.

The LANai chip provides bi-directional access to the Myrinet network. The packet interface handles the injection of packets into the network and their consumption from the network. Special memory-mapped registers are to be used to access the packet interface. On the host side, the LANai core cannot access the EBUS directly and the processor has to initiate data transfer between LBUS and EBUS. In a typical mode, the LANai processor with the memory on its LBUS appears as a synchronous memory block from the EBUS. But, when the DMA engine starts the data transfer between the LBUS and EBUS, the chip becomes the master and master.
EBUS the slave. The EBUS is simple and needs extra hardware to connect to any standard bus such as PCI.

The processor can compute partial Internet checksums and has a special CheckSum (CKS) register which gets modified during EBUS-LBUS DMA transfers. It also has two real-time counters: (a) real-time clock which gets incremented every time-reference period (1 microsecond), and (b) interrupt timer which gets decremented on every time-reference period. It also has miscellaneous special registers such as the TIMEOUT register (which specifies the timeout period of the LANai watchdog timer), MYRINET register (its bit 0 NRES.ENABLE indicates whether the board is to be reset, and bit 1 CRC.ENABLE indicates whether CRC verification is to be performed on the incoming packets), etc.

### 3.1.2 Myrinet API

The Myrinet API provides functions required for programs that run on the LANai. The LANai libraries are to be initialized by certain function calls prior to their use. The topics that are discussed in the following sections are:

- Myrinet Control Program
- Channel
- Addresses
- Resets to the LANai boards
- DMA
- Sending/Receiving Messages

These are discussed with the Myricom standard MCP as an example. The two topics, channels and sending/receiving messages are very specific to the Myricom MCP. The remaining issues are relevant to other MCPs although the details might not exactly conform to them.
MCP

A program that runs on the LANai chip is called a Myrinet Control Program (MCP). An MCP is not burned into the LANai chip. It is loaded from the host into the LANai memory using the `lload` Myrinet tool. The `lload` program uses the `lanai_load_and_reset()` function from the Myrinet API library. The Myrinet API must be compiled with the LANai code to be loaded into the LANai SRAM. The device driver performs the following to load an MCP into the LANai using the Myrinet API:

1. Reset the LANai with `lanai_reset_unit()`.
2. Set the clock value with `lanai_set_clock_value()`.
3. Load zeros into the entire LANai memory.
4. Load MCP with `myriApiLoadLanai()`.
5. Set the MCP address with `myriApiSetAddress()`.
6. Set DMA burst size with `myriApiSetBurst()`.
7. Release LANai from reset mode by calling `lanai_reset_unit()` again.

An MCP can do general purpose computing, but standard MCPs typically:

- Receive messages from the network and send it to the local host.
- Send messages from the local host to the network.

There are some standard MCPs that only provide these services. Examples of such MCPs are the standard Myricom MCP, and the “Simple Message Protocol” (SMP) [SHDM96] from Mississippi State University.

There are other MCPs which provide more functionality than just the basic service of flow control. One such example is the BullDog MCP [HDMS97, HDS97],

30
also from Mississippi State University, which provides different protocols for ordered/unordered reliable message delivery. But its functionality is limited to reliable ordering and delivering of messages.

**Channel**

A channel is the software interface between the host and the MCP. Each channel consists of three queues: (a) *Send* queue, (b) *Receive* queue, and (c) *Receive Ack* queue. The number of channels in the MCP, which is configurable depending on the kind of LANai board, can be obtained using the `myriApiGetNumChannels()` API function. Channels are numbered starting from zero which is always used by the TCP/IP driver. Any channel should be used only by a single host process.

**Addresses of LANai Boards/MCPs**

Every LANai board has a unique 48-bit address written into its EEPROM which the host relays to the MCP using the LANaiDevice function calls. The MCP messages use 64-bit addresses which is of the format `0:0:a:b:c:d:e:f` where `a:b:c:d:e:f` is the 48-bit address burned into the LANai board. The static routing table stored in the LANai converts the 64-bit address into a route of the path to be traversed. If the highest bit of the MCP address is set, then it is a multicast address. Otherwise, it is a regular address. When the highest bit is set, the message is delivered to all LANai boards connected to the Myrinet network. If it is a regular address, the Myrinet address and the channel number in the header are to be considered. On reaching the LANai board with the specific Myrinet address, the message is delivered to the process on the host that holds the specified destination channel.

**Resets to LANai Boards**

To avoid deadlock, the LANai boards can be reset at anytime. There are two kinds of reset: (a) Hardware reset - this can either be generated by the host via
LANaiDevice library functions, or it can be a forward reset generated by the Myrinet network itself, (b) Software Reset- these are generated by the host by calling the `myriApiReset()` function in the API. While hardware resets cause the MCP to be restarted, a software reset only resets a single channel. After either reset, the MCP needs to re-establish communication with the host via hand-shaking. The host also needs to restore the MCP channel's list of multicast addresses, reclaim any buffers held by the MCP at the time of the reset, and set up the channel queues.

**DMA**

There are two DMAs to consider when one host communicates with another host via the Myrinet network. The DMA between host memory and LANai memory is achieved with the help of the DMA engine in Figure 3.1. The host DMA pointers must be aligned on a 4-byte boundary, and must be in the memory space specific to the particular operating system they are running on. To be considered as valid, the host DMA pointers have different requirements on different platforms and operating systems. For example, a 32-bit-aligned kernel virtual address is considered as valid for a Pentium running Linux 1.2.x. Using the same address on a different architecture may lead to problems. During the DMA, the memory must be locked so it is otherwise not disturbed. The second DMA between the LANai memory and the network is carried out with the help of the Packet Interface in Figure 3.1. The DMAs are started by the MCP executing on the network processor.

**Message Sending/Receiving**

The Send queue is a fixed size circular buffer of length `NUM SENDS`. When the hosts wants to send a message, it places it on the Send queue. The MCP removes the message and sends it over the Myrinet network. At any time, there can be only `NUM SENDS` messages in the Send queue. If the Send queue is full, the host has to wait until the MCP has removed a message from the Send queue and has sent it
Receiving messages is more complicated than sending messages since there are two queues involved: Receive queue and Receive Ack queue. When the Myrinet adaptor receives a message from the network, it DMAs the message into a single buffer (if the entire message can fit in the fixed-size buffer) or into multiple buffers in the Receive queue. Then it increments the Receive Ack queue. The host, on noticing the Receive Ack update, figures which group of Receive queue buffers contain the new message. It DMAs the message from the buffers, and puts the empty buffers back in the Receive queue. If either the Receive queue is empty or the Receive Ack queue is full, the MCP drops messages from the network.

3.2 Benchmarks Used for the Choice of the Host Processors

Several CPUs were evaluated for the Myrinet testbed. The ideal processor should not starve either the CPU or the Myrinet DMA engine of the PCI bus, but yet produce optimal throughput. The following four processors were considered for the choice of host machines:

1. 233MHz Pentium-MMX, 128MB RAM, 256KB cache, with Intel 430TX chipset and ASUS motherboard.
2. 200MHz Pentium-Pro, 128MB RAM, 256KB cache, with 440LX chipset and ASUS motherboard.
3. 266MHz Pentium-II, 128MB RAM, 512KB cache, with 440FX chipset and Intel motherboard.
4. 233MHz K6-AMD, 128MB RAM, 256KB cache, with 430LX chipset and ASUS motherboard.
Four benchmarks were used to choose the most ideal processor to build the high-speed gigabit network. The following sections discuss the relevance of each benchmark run, with the experimental results obtained on the four processors.

3.2.1 Results of \texttt{bw\_mem\_cp} and \texttt{hswap} Benchmarks

The terminology \texttt{bw\_mem\_cp} stands for bandwidth memory copy. It measures how fast the CPU can copy memory. It was implemented using \texttt{bcopy} of the Imbench test suite. The Imbench test suite is a system performance tool which can be obtained from the FreeBSD benchmarks ftp site.

\texttt{hswap} measures how fast the Myrinet card can read or write memory with its DMA engine. This benchmark comes with the Myrinet software.

Table 3.1 shows the memory bandwidth measured on each of the processors by \texttt{bw\_mem\_cp}, both when they are otherwise idle, and when the network device (Myrinet adaptor) is hammering on the PCI bus. The test runs are explained below:

1. \texttt{bw\_mem\_cp}, no competition: this is performed by just running \texttt{bw\_mem\_cp} with the \texttt{hswap} benchmark idle. By running this, the speed of the system’s memory can be determined.

2. \texttt{bw\_mem\_cp} during E2L phase of \texttt{hswap}: \texttt{bw\_mem\_cp} is first started and later \texttt{hswap} is started. When \texttt{hswap} goes through the E2L (host to LANai) phase, the value of \texttt{bw\_mem\_cp} is reported.

3. \texttt{bw\_mem\_cp} during L2E phase of \texttt{hswap}: this is similar to the second test, except that the value of \texttt{bw\_mem\_cp} is reported during the L2E (LANai to host) phase of \texttt{hswap} testing.

4. \texttt{hswap} b/w for 8k transfers, no competition: \texttt{hswap} bandwidth results are reported during E2L and L2E phases without running \texttt{bw\_mem\_cp}.
<table>
<thead>
<tr>
<th>Test</th>
<th>Pentium-MMX in MB/sec</th>
<th>Pentium-Pro in MB/sec</th>
<th>Pentium-II in MB/sec</th>
<th>K6-AMD in MB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bw_mem_cp, no competition (test 1)</td>
<td>49.5</td>
<td>53</td>
<td>51</td>
<td>47</td>
</tr>
<tr>
<td>bw_mem_cp during E2L phase of hswap (test 2)</td>
<td>5.53</td>
<td>24.34</td>
<td>23.78</td>
<td>8.75</td>
</tr>
<tr>
<td>bw_mem_cp during L2E phase of hswap (test 3)</td>
<td>11.65</td>
<td>37.35</td>
<td>36.94</td>
<td>6.54</td>
</tr>
<tr>
<td>hswap b/w for 8k transfers, no competition (test 4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2L phase</td>
<td>112.2</td>
<td>111.2</td>
<td>121.4</td>
<td>111.5</td>
</tr>
<tr>
<td>L2E phase</td>
<td>114.0</td>
<td>121.1</td>
<td>121.0</td>
<td>122.1</td>
</tr>
<tr>
<td>hswap b/w for 8k transfers while competing w/CPU for memory (test 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2L phase</td>
<td>112.2</td>
<td>50.7</td>
<td>49.9</td>
<td>111.4</td>
</tr>
<tr>
<td>L2E phase</td>
<td>111.9</td>
<td>40.7</td>
<td>38.7</td>
<td>113.1</td>
</tr>
</tbody>
</table>

Table 3.1: Results of bw_mem_cp and hswap Benchmarks

5. hswap b/w for 8k transfers while competing w/CPU for memory: hswap bandwidth results are reported during E2L and L2E phases while also running bw_mem_cp.

All four processors have approximately the same processor speeds (test 1), but Pentium-II has the highest DMA bandwidth when considering both E2L and L2E phases (test 4). On combining bw_mem_cp/hswap, both Pentium-MMX and K6-AMD maintain the same DMA transfer bandwidth which is around 111MB/sec (test 5). But in both the processors, the Myrinet card starves the CPU leading to very low (in the range of 5-11MB/sec) processor bandwidth (tests 2 and 3).

On the other hand, both Pentium-Pro and Pentium-II have significant decrease in their DMA transfer bandwidths, which is around 50MB/sec (test 5), when compared to about 120MB/sec in test 4. This is because the Myrinet card does not starve the CPU of its cycles, unlike the tests with the Pentium-MMX and K6-AMD processors. The processor speeds decrease only to about 30MB/sec (test 2) when compared to 52MB/sec in test 1.

Pentium-Pro and Pentium-II have both good processor and DMA band-
widths when running independent of each other, and also reasonable bandwidths when competing with each other. Pentium-II has slightly better DMA transfer bandwidth than Pentium-Pro. However, both Pentium-MMX and K6-AMD fail to deliver balanced performance, when the CPU and Myrinet DMA compete with each other.

### 3.2.2 Results of lmbench Test Suite

This system measurement tool was used to measure how fast the cpu can read, write and copy 8MB of memory. Table 3.2 gives the bandwidth in MB/sec for different tests on the processors.

<table>
<thead>
<tr>
<th>Test</th>
<th>Pentium-MMX in MB/sec</th>
<th>Pentium-Pro in MB/sec</th>
<th>Pentium-II in MB/sec</th>
<th>K6-AMD in MB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bw_mem_rd 8 MB</td>
<td>148.98</td>
<td>228.15</td>
<td>219.86</td>
<td>188.77</td>
</tr>
<tr>
<td>bw_mem_wr 8 MB</td>
<td>84.15</td>
<td>82.29</td>
<td>74.05</td>
<td>77.84</td>
</tr>
<tr>
<td>bw_mem_cp 8 MB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unrolled aligned</td>
<td>49.93</td>
<td>47.20</td>
<td>44.30</td>
<td>46.71</td>
</tr>
<tr>
<td>libc aligned</td>
<td>46.37</td>
<td>51.39</td>
<td>53.25</td>
<td>47.14</td>
</tr>
<tr>
<td>unrolled unaligned</td>
<td>49.91</td>
<td>46.78</td>
<td>44.90</td>
<td>45.24</td>
</tr>
<tr>
<td>libc unaligned</td>
<td>33.39</td>
<td>50.70</td>
<td>52.84</td>
<td>45.66</td>
</tr>
</tbody>
</table>

Table 3.2: Results of lmbench Test Suite

Both Pentium-Pro and Pentium-II have the highest read bandwidths. Although Pentium-II has slightly lower write bandwidths, it has almost the same copy bandwidths in all four cases as the other processors. In fact, in the libc case, it has the highest copy bandwidth (around 53MB/sec) in both the aligned and unaligned
3.2.3 Results of netperf Benchmarks

The netperf tool measures how fast a network application can run when stressing both the CPU and the Myrinet driver. It makes use of both memory copies (for example, the uio.move() in the socket layer), and the Myrinet card (via myristart() or check_receive() functions in the Myrinet driver). In short, it combines the testing of bw_mem_cp and hswap when sending data over the network. The sender sends data from its memory which is DMAed to the LANai memory where the MCP pushes it to the network. On the receiver side, the data is DMAed into LANai memory following another DMA into host memory to the particular process. In Tables 3.3, 3.4, and 3.5, N/A implies that particular experimental result is not available because more than one processor of certain processor types were unavailable for testing.

UDP_STREAM Testing Using netperf

Tables 3.3 and 3.4 are both results for a UDP_STREAM (data are sent in blocks over the network without waiting for any acknowledgement) with the latter repeating the same experiment after turning off the checksums. The experimental setup and the terminologies for tables 3.3 and 3.4 follow:

- Socket Buffer Size = 41600 bytes.
- Message Size = 8192 bytes.
- M_Sent: Number of messages that the sender sent to the receiver.
- M_Recv: Number of messages that made it all the way up to the user process listening on a socket on the receiver. It is likely that more messages were received into the receiver's kernel than were delivered all the way up to the user process.
- MS_BW: Throughput for M_Sent case in Mbits/sec
- MR_BW: Throughput for M_Recv case in Mbits/sec

<table>
<thead>
<tr>
<th>Receiver</th>
<th>MS_BW</th>
<th>MR_BW</th>
<th>Pentium-II</th>
<th>Pentium-Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>K6-AMD</td>
<td>203.89</td>
<td>203.42</td>
<td>31043</td>
<td>31043</td>
</tr>
<tr>
<td>M_Sent</td>
<td>N/A</td>
<td>31109</td>
<td>31043</td>
<td></td>
</tr>
<tr>
<td>M_Recv</td>
<td>203.85</td>
<td>203.42</td>
<td>31043</td>
<td></td>
</tr>
<tr>
<td>MS_BW</td>
<td>203.85</td>
<td>203.42</td>
<td>31043</td>
<td></td>
</tr>
<tr>
<td>MR_BW</td>
<td>203.85</td>
<td>203.42</td>
<td>31043</td>
<td></td>
</tr>
<tr>
<td>Pentium-II M_Sent</td>
<td>53196</td>
<td>52782</td>
<td>53176</td>
<td></td>
</tr>
<tr>
<td>M_Recv</td>
<td>24789</td>
<td>52706</td>
<td>53124</td>
<td></td>
</tr>
<tr>
<td>MS_BW</td>
<td>348.58</td>
<td>345.53</td>
<td>348.17</td>
<td></td>
</tr>
<tr>
<td>MR_BW</td>
<td>162.44</td>
<td>345.38</td>
<td>347.83</td>
<td></td>
</tr>
<tr>
<td>Pentium-Pro M_Sent</td>
<td>40335</td>
<td>40260</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>M_Recv</td>
<td>40279</td>
<td>40219</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>MS_BW</td>
<td>264.11</td>
<td>263.68</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>MR_BW</td>
<td>263.74</td>
<td>263.41</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: UDP_STREAM Testing Using netperf

From Table 3.3, Pentium-II is the fastest sender, about 347Mbits/sec, when compared to the sending speed of K6-AMD which is around 203Mbits/sec or Pentium-Pro which is around 264Mbits/sec. When considering the receiving bandwidth, Pentium-II and Pentium-Pro receive at the same speed (around 203Mbits/sec) when K6-AMD is the sender. Pentium-Pro and Pentium-II both receive at about 346Mbits/sec, but at twice the rate of K6-AMD receive bandwidth when Pentium-II is the sender. Pentium-II and K6-AMD receive at the same speed as the sending rate of the Pentium-Pro.

On turning off the checksums, in Table 3.4, both the send and receive bandwidths increase on some of the processors. However, the relative order of the processors in Table 3.4 is similar to that in Table 3.3, except that K6-AMD receives
Table 3.4: UDP_STREAM Testing Using netperf (turning off the checksums)

<table>
<thead>
<tr>
<th>Receiver</th>
<th>K6-AMD</th>
<th>Pentium-II</th>
<th>Pentium-Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>K6-AMD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_Sent</td>
<td>N/A</td>
<td>33622</td>
<td>33904</td>
</tr>
<tr>
<td>M_Recv</td>
<td>33615</td>
<td>33904</td>
<td></td>
</tr>
<tr>
<td>MS_BW</td>
<td>220.34</td>
<td>222.02</td>
<td></td>
</tr>
<tr>
<td>MR_BW</td>
<td>220.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium-II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_Sent</td>
<td>52125</td>
<td>52766</td>
<td>52274</td>
</tr>
<tr>
<td>M_Recv</td>
<td>52125</td>
<td>52631</td>
<td>52256</td>
</tr>
<tr>
<td>MS_BW</td>
<td>341.58</td>
<td>345.79</td>
<td>342.54</td>
</tr>
<tr>
<td>MR_BW</td>
<td>341.58</td>
<td>344.90</td>
<td>342.42</td>
</tr>
<tr>
<td>Pentium-Pro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_Sent</td>
<td>42889</td>
<td>42794</td>
<td>N/A</td>
</tr>
<tr>
<td>M_Recv</td>
<td>42877</td>
<td>42781</td>
<td></td>
</tr>
<tr>
<td>MS_BW</td>
<td>280.82</td>
<td>280.36</td>
<td></td>
</tr>
<tr>
<td>MR_BW</td>
<td>280.74</td>
<td>280.27</td>
<td></td>
</tr>
</tbody>
</table>

better from Pentium-II in the latter case. It is clear that Pentium-II can send data faster than both Pentium-Pro and K6-AMD, and receive slightly faster than or at the same rate as the latter two processors.

**TCP_STREAM Testing Using netperf**

Table 3.5 are the results for a TCP_STREAM where the data is sent over an established stream between the two ends. The experimental setup for TCP_STREAM testing are:

- **Recv buffer Size = 65536 bytes**
- **Send buffer Size = 65536 bytes**
- **Send Message Size = 65536 bytes**
Both Pentium-II and Pentium-Pro perform similarly when K6-AMD is the sender. Both Pentium-Pro and Pentium-II perform significantly better than K6-AMD (about 80% better) when Pentium-II is the sender, as with UDP_STREAM testing when checksums are not turned off. Pentium-II also performs about 63% better than K6-AMD with Pentium-Pro being the sender, which is different from the results with UDP_STREAM testing.

**Evaluation**

*netperf* was run focusing on the UDP_STREAM. Many applications which actually process data are going to have throughput numbers similar to what one gets with a UDP_STREAM. The better the UDP_STREAM number, the better a system is at processing data, and running a Myrinet card at the same time.

Both PCs and alphas (using results from Duke University) have problems with optimally executing the UDP_STREAM while running the Myrinet card. Alphas have limited *hswap* numbers, PCs have bad *bw_mem_cp* numbers. Because of this, neither is an ideal choice for a platform on which to run Myrinet. An ideal platform will have excellent *hswap*, *bw_mem_cp* and combined *hswap/bw_memory_cp* bandwidths. A platform like this is capable of running the Myrinet card and processing data at the same time, and will indicate this capability with good UDP_STREAM send and receive numbers.

Cost constraints of this project limited the choice to PCs. Both Pentium-II

<table>
<thead>
<tr>
<th>Receiver</th>
<th>K6-AMD in Mbits/sec</th>
<th>Pentium-II in Mbits/sec</th>
<th>Pentium-Pro in Mbits/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>K6-AMD</td>
<td>N/A</td>
<td>193.83</td>
<td>193.13</td>
</tr>
<tr>
<td>Pentium-II</td>
<td>175.71</td>
<td>312.41</td>
<td>317.45</td>
</tr>
<tr>
<td>Pentium-Pro</td>
<td>178.62</td>
<td>290.57</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.5: TCP_STREAM Testing Using *netperf*
and Pentium-Pro have equivalent \texttt{bw\_mem\_cp/\_hswap} results. Pentium-II has slightly better copy bandwidths than the other processors. Although Pentium-Pro receives at the same rate as Pentium-II, the latter sends at a much faster rate than the former, as per the results using the network performance evaluation tool. These results lead Pentium-II to be the best choice.

### 3.3 Layout of the Myrinet Network Testbed

The complete Myrinet layout is shown in Figure 3.2. Each of the 16 Pentium-IIs, with 128MB RAM, 512KB cache and 440FX chipset, numbered 1 through 16 in the figure have a M2M-PCI32C LANai interface. The LANai boards on the computers access the Myrinet network via M2M-CB-05 SAN cables. These are 5-foot length cables which connect the LANai interfaces to M2M-DUAL-SW8 dual 8-port SAN switches via M2M-Y splitters.

![Figure 3.2: Layout of Myrinet Network](image)

The specifications of the Myrinet switches, interfaces, cables and splitters are
briefly reviewed here, but their details can be found in [Myr97b, Myr96, Myr97e].

3.3.1 M2M-DUAL-SW8 Myrinet Switch

This switch is designed for an in-cabinet cluster to be deployed with M2M-PCI32C and Myrinet SAN cables. A single dual 8-port Myrinet SAN switch houses two 8-input/output, full-crossbar, Myrinet-SAN switches as shown in Figure 3.3. Each channel operates at 1.28 Gb/s, providing a total bisection data rate of 10.24 Gb/s for each switch.

The 8 ports support 1.28+1.28 Gb/s data rate on each of the A and B links. The A and B links may be split onto separate cables using the M2M-Y splitters as in ports 4, 5, 6, and 7 of Figure 3.2. The switches A and B can also be connected using a M2M-X. This is called an A-B link because it is used to create a link, between the A and B switches at the SAN connectors of the M2M-Dual-SW8 switch, as in ports 0, 1 of switch 0 and ports 2,3 of switch 1, as shown in Figure 3.2.
In the network shown in Figure 3.2, computers \(1,3,5,7\) are connected to the internal switch A of switch 0, \(2,4,6,8\) to the internal switch B of switch 0, \(10, 12, 14, 16\) to the internal switch A of switch 1, and \(9, 11, 13, 15\) to the internal switch B of switch 1. The cluster connection can be viewed as shown in Figure 3.4.

3.3.2 M2M-PCI32C Myrinet-SAN/PCI Interface

This host interface contains the LANai 4 processor which runs the MCP. Its main characteristics are:

- 32-bit PCI-bus interface operating at 33MHz. Its DMA performance at 33MHz, with large blocks, exceeds 120MB/s peer mode and with good PCI implemen-
tations to/from system memory.

- Interface memory is 1MB operating at double the PCI-clock rate.
- Interface processor is a LANai-4 RISC, operating at the PCI-clock rate (33MHz), explained in detail in section 3.1.1.

### 3.3.3 M2M-CB-05 Cables

These 5-foot cables are used to connect the interfaces to the switches. Each SAN-cable carries two channels which are full-duplex and have 1.28Gb/sec data rate each.

### 3.4 Language Support

This section discusses Emerald’s features that support mobility and the original structure of its interpreter before changes were made to its design for running within the LANai.

#### 3.4.1 Emerald Primitives to Support Mobility

Mobility in Emerald [JLHB88] is provided by a small set of language features which are discussed below:

1. **Move**: move X to Y, moves the object X to Y if Y is a node, or moves X to the node where Y is located if Y is an object.

2. **Locate**: locate X, returns the node where object X is located.

3. **Fix**: fix X at Y, moves X to Y and fixes it there, i.e., makes it non-mobile.

4. **Unfix**: unfix X makes a fixed object mobile again.

5. **Refix**: refix X at Y unfixes the object from where it is currently located, moves it to Y and fixes it there.
These routines are a complete list of functions that one would need to move objects around. A main concern in the move routine is deciding how much of the object’s state, in addition to its activation records, to move to the target when the object migrates. If the object’s state has an array of other objects, (a) should the entire array be moved, or (b) should the individual objects of the array be moved only when invoked on the remote site. The designers of Emerald have introduced the attached qualifier in the language, to be used by the programmer, in order to make explicit requests of moving certain data objects. For example if the object:

```plaintext
object myobject
  attached var myDirectory: Directory
  var myOtherDirectory: Directory
end object
```

is moved, myDirectory has to be moved as well because it is declared to be attached, as opposed to myOtherDirectory which may or may not be moved depending upon the compiler’s decisions.

Parameter passing is implemented by the call-by-object-reference technique. Moving an argument object when an object moves depends on (i) the size of the argument object, (ii) whether it will be referenced frequently enough at the remote site, and (iii) how costly it is to move it to the remote site. The Emerald compiler tries to make these decisions. For example, small immutable objects such as strings or integers should be moved rather than being referenced remotely. The application program can assist the compiler in arriving at the appropriate decisions by specifying whether it would like a particular object to be moved to the remote site.

The above review, though does not do complete justice to all the features of Emerald, supports the idea that mobility or service migratibility can be easily
achieved using it as a base framework.

3.4.2 Structure of the Emerald Interpreter

The host-based Emerald interpreter (named host-emx) is a multi-threaded distributed object-oriented virtual machine. It consists of one big C program which uses the standard libc, as well as other libraries to handle system calls and communication over the network via sockets. When it starts, the virtual machine does basic initialization, loads all the built-in types from files, and builds objects of the loaded-in types. It then creates a worker pool of threads and a single listener thread as shown in Figure 3.5.

The listener thread listens for new connections. On receiving a request from a new node, it creates a new reader thread which has its own TCP socket stream connection to the new node. The listener creates a new reader for every connection. For example, if there are four nodes in the Emerald world, each node will have three reader threads.
The reader threads listen for any incoming packets on their stream. They put the incoming messages in the process queue which uses the FIFO technique to send them to the worker pool. At any time, there should be at least one worker thread so as to avoid deadlock.

When the worker thread receives a new message, it checks whether the object to which the message is to be delivered is resident. If the object has moved, the worker redirects the message to where the local node thinks it has moved to. Otherwise it processes the incoming message, and if needed, also handles sending out new messages.

The virtual machine also has a garbage collector to clean up any stale objects. The virtual machine divides its memory area into three main blocks to handle malloc and garbage collection:

- **malloc memory**: This area is used to allocate space for data that is not garbage collected, i.e., for thread control blocks, stacks, tables, sockets, file descriptors, etc. for the different threads running in the virtual machine.

- **GC memory**: Emerald uses the generational garbage collection mechanism. There are two main blocks (a) NEWGC, and (b) OLDGC. The NEWGC memory area is further divided into two blocks NEWGC1 and NEWGC2. When objects are first created, they live in one of these blocks. When the garbage collector is called, it handles either NEWGC1 or NEWGC2. It cleans up unreferenced objects and moves the recently created referenced objects to the other block. After a few garbage collections the old objects which have moved back and forth between NEWGC1 and NEWGC2, and are still being referenced are promoted to the OLDGC block. Objects moved to OLDGC may be garbage collected if they remain unreferenced, but at a less frequent rate than NEWGC1 or NEWGC2 because they are less likely to die.
3.5 Summary

This chapter reviewed the Myricom details necessary for understanding the design and implementation of lanai-emx (running on the LANai), with particular focus on the LANai 4 processor and the Myrinet API. This was followed by the results of the experimental benchmarks to support the choice of Pentium-II for the Myrinet cluster. The Myrinet layout illustrates the network that was constructed and the different connectors and switches involved. The overall structure and the high level implementation of the host-emx was discussed to explain the changes necessary for porting it to the LANai.

The discussion of the interpreter will help understand the changes necessary for execution on the LANai. The next chapter discusses the design and implementation changes needed for running within the 1MB of memory on the LANai. The main changes include the multi-threaded distribution package of the interpreter, communication protocols and the host/LANai interface due to the lack of an OS on the LANai among other intricate implementation details.
Chapter 4

Design and Implementation of

*Emu*

*Emu* consists of two implementations of the Emerald virtual machine: *lanai-emx*, running on the LANai; and *host-emx*, running on the host. The *lanai-emx* on the network adaptor can communicate with other interpreters, either on the local host via EBUS <-> LBUS transfer, or on some other host via the Myrinet network. In other words, the *lanai-emx* running on the network adaptor can do anything that the *host-emx* can. For example, it can process messages, build objects, execute the active objects, and garbage collect. The layout of the chapter follows.

Section 4.1 discusses the key issues to be considered for running the Emerald interpreter within the LANai. This is followed by Section 4.2 which discusses the data structures and protocols designed and implemented in *lanai-emx* to get the support of the host due to lack of an OS. Sections 4.3 and 4.4 discuss the communication protocols and new routing information protocol in the *lanai-emx* respectively. Section 4.5 discuss various miscellaneous issues in the design of *Emu* and Section 4.6 concludes.
4.1 Key Issues for Moving the Emerald Interpreter to the LANai

This section discusses the main issues involved in running the Emerald virtual machine within 1MB of SRAM on the LANai. The first two issues could be handled by implementation changes. The distribution package had to be completely redesigned, because the environment of the LANai is different from what one would expect of a regular processor. Each of the issues are concerned with resolving incompatibilities with the new environment.

4.1.1 Limited Memory

The Emerald interpreter on the LANai must execute within 1MB of SRAM. This is possible only if the Emerald code, shared memory between the host and the LANai, stack, heap and garbage collection memory space fit in this 1MB. The memory layout of the lanai-emx on 1MB of RAM on the LANai is shown in Figure 4.1. The problems faced in the mapping of the various parts of the interpreter to the LANai SRAM, due to the limited memory, will now be described.

The LANai environment does not provide any debugger, such as gdb, or even printing to standard output such as simple printf stubs. Hence, initial mapping of the different parts of the interpreter to LANai SRAM was done by trial and error. When the host loads the MCP to the LANai, handshaking between the host and the LANai must successfully complete. This is necessary for the LANai and host to communicate via shared memory or for any sort of debugging protocol to be built. A considerable amount of time was spent in successfully completing the startup calibration between the LANai and the host, which could have been saved if some debugging support was provided.

The lanai-emx running within the LANai is a highly optimized version, with code size of about 209 KB without debugging options (emx), and about 285 KB
with debugging help (\textit{emx.debug}), as shown in Figure 4.1. Shared memory starts at 308 KB to provide for proper execution of either versions of the \textit{lanai-emx} MCP. With the help of a short program, it was found that the stack grows to about 3 KB to 4 KB and the shared memory between the host and the LANai is about 56 KB. An additional 20 KB was added to provide for nested activation records, hence 76 KB was allocated in total for the shared memory and the stack. The remaining space was partitioned to about 256 KB for the heap, 64 KB for the new generation garbage collection, and 320 KB for the old generation garbage collection, as shown in Figure 4.1.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{memory_layout.png}
\caption{Memory Layout on the LANai}
\end{figure}

### 4.1.2 Library Support

Emerald makes extensive use of the standard C library, \textit{libc}. The LANai, being a different environment with no OS or debugging support, is an unfriendly programming environment. It does not support the standard \textit{libc} routines. The original
host-emx was implemented using many functions from libc. The simple approach to solve this problem is to include all libc source files along with the VM code without modification. However this could not be done for two reasons. First, the entire VM including its code, heap and stack must run within 1MB of RAM, and including unnecessary libc routines would make this impossible. Second, many of the libc files use system calls for a regular OS. Hence, including the entire libc unmodified will not work. The major libc issues to be considered for the lanai-emx were:

- Malloc routines such as malloc(), realloc(), free(), etc. had to be rewritten to work in the LANai environment. The malloc routines work in the SRAM area that is allocated for the heap/malloc, as shown in Figure 4.1.
- Stdout calls were handled by using the print/debug protocol discussed in Section 4.2.2.
- Stdin, system calls and file opens, reads, etc. were handled using the host interface protocol discussed in Section 4.2.1.
- Although some libc routines could be worked around, sources for frequently occurring routines such as strcmp(), memmove(), strcpy(), etc. had to be provided.
- Basic structures, defines and typedefs had to be provided in addition to routines from libraries.

4.1.3 Distribution and Mobility

In the original Emerald interpreter, distribution implementation depends on user-level threads. But there is no builtin thread utility for the LANai. It seemed better to reimplement the multi-threaded package as a single thread in the LANai considering the memory limit of the network adaptor. Redesigning distribution from a multi-threaded model to a single processor model was the most difficult feature of porting the interpreter to the LANai.
The original multi-threaded Emerald interpreter, explained in Section 3.4.2, provides a conceptual understanding which facilitated coding. However, restructuring this multi-threaded system as a single process required a completely different understanding of the interpreter in order to make the different calls at the proper times. Achieving mobility and distribution for the VM on the LANai completed the implementation and porting of the lanai-emx.

4.2 Interface to the Host

In order to execute on the network processor, the lanai-emx needs assistance from the host. The protocols implemented provide OS services, due to the lack of an OS, and debugging support on the LANai. This is followed by a discussion of the shared memory structure for communication between the host and LANai and the different services that rely on it. Each of the issues discusses the support provided by the LANai with the help of the host.

4.2.1 OS-Support Protocol

The Emerald interpreter was designed to run in a Unix environment. It uses files and system calls. On initialization, Emerald interpreter loads a huge (about 120 KB) Builtins file to build the standard objects (note that everything in Emerald is an object). It also uses system calls to get the ipaddress of the local host which is a part of the object identifier of all objects resident on the local machine.

The LANai does not support inodes or vnodes as in a Unix environment. There is no file system on the LANai. So the OS-Support protocol was designed and implemented to handle opening/reading/closing of files with the help of the host. But the Builtins file is too big to be loaded all at once into the LANai memory. Therefore, parts of the file are read and objects are built at every stage before reading the next part of the file.
The OS-Support protocol was extended to handle system calls such as `gethostname()`, `gethostbyname()`, etc., due to the lack of OS on the LANai. But the MCP running on the LANai has the illusion that the calls are satisfied by the LANai processor. It was also modified to handle basic information exchange between the LANai and the host.

In order to make it possible for the host to assist the LANai in satisfying the above mentioned calls, the OS-Support protocol on the `lanai-emx` and the `host-emx` of the local host communicate via a shared structure shown in Figure 4.2. In short, this protocol can be extended as necessary to enlist the host's help in any future needs where the LANai is unable to do so by itself.

```c
typedef volatile struct _MCP_E2L_info {
    int req_flag;   /* query to be satisfied */
    int fd;         /* file descriptor for file calls */
    unsigned int nbytes; /* number of bytes */
    int result;     /* result of the call */
    char name[50];  /* name of file or hostname, etc. */
    char buffer[MAX_SIZE]; /* buffer to return the reply */
} MCP_E2L_info;
```

Figure 4.2: The Host Interface Protocol Memory Structure

The routine in Figure 4.3, which is added to `host-emx`, assists the `lanai-emx` in handling various calls due to the lack of OS support on the LANai.

1. File requests: Cases 1, 2 and 3 handle opening, reading and closing of files respectively for the `lanai-emx`.

2. `gethostname()`/`gethostbyname()` requests: When an Emerald machine comes up, it tries to get its hostname and other details to set or define its node address. Due to the inability of the LANai interpreter to satisfy system calls, such as `gethostname()` / `gethostbyname()`, it enlists the help of the host interpreter in order to successfully complete those requests. The host VM periodically calls the function whose pseudocode is shown in Figure 4.3 to check whether
void handleReqFromLANai() {

    access the MCP.E2L.info shared structure of the LANai;
    switch (req_flag of the shared structure) {

        case 0 : no request to satisfy;
        case 1 : open file specified and returns its file descriptor;
        case 2 : read from file as specified;
        case 3 : close file;
        case 4 : satisfy gethostname() request;
        case 5 : satisfy gethostbyname() request;
        case 6 : satisfy cases 4 and 5 in one go;
        case 7 : get rootnodeid;

    }
    update req_flag;
}

Figure 4.3: Pseudocode for Handling Requests from the LANai by the Host

these system requests (cases 4, 5 and 6) have been issued from the LANai.

3. Information exchange: Basic information about rootnodeid, etc. for the lanai-emx are handled by case 7.

4.2.2 Print/Debug Protocol

The LANai does not provide any support for printing to standard output. Debugging support was essential to complete porting of the stand-alone interpreter to the LANai. A new data structure MCP.L2E.info was introduced in the shared memory, between the LANai and the host, to assist with writing to the standard output. It is in the form of a circular array. No locking is needed to access this shared block of memory. This is because the LANai only writes to the buffer and updates the tail of the circular queue, and the host only reads from the buffer and updates the head of the queue.
4.2.3 Shared Memory Structure Between LANai and Host

The structure shown in Figure 4.4 is the memory on the LANai which can be directly accessed by the host. It plays the major role in handling any type of communication such as sending/receiving messages, handshaking, etc. between the LANai and the host. It consists of data structures for different services such as debugging, host interface support, etc.

```c
typedef volatile struct _MCP.Sharedjnem_ {
  int handshake;
  int address;
  MCP_Route route;
  MCP_Send_info send_info;
  MCP_Recv_info recv_info;
  MCP_L2E_info debug_info;
  MCP_E2L_info hostInterfaceInfo;
  int dma_burst;
  int staging_buffer;
} MCP_Shared_mem;
```

Figure 4.4: The MCP Shared Memory Structure

The different services that this shared memory structure helps support are discussed below:

- Handshaking between the host and the LANai is achieved by setting the `handshake` attribute.
- Routing is done using `MCP_Route` which consists of the hop length and switching information in an array.
- `MCP_Send_info` is used to send messages to the LANai from the host. It consists of a pointer to the buffer, the buffer length and a `ready_bit`.
- `MCP_Recv_info` is used by the host to receive messages from the LANai. Its structure is similar to that of `MCP_Send_info`. 
• Debugging is done with the help of \textit{MCP.L2E.info}. This structure handles messages from the LANai to the host, hence the name. This shared structure played an important role in porting the interpreter as there was no other means of debugging in the LANai.

• \textit{MCP.E2L.info} is the structure that handles file requests, \texttt{gethostname()}, and \texttt{gethostbyname()} requests, etc. The host does the appropriate work based on the \texttt{req.flag} value as explained in Section 4.2.1.

### 4.3 Communication Protocols

Communication protocols of the original Emerald interpreter were redesigned in order to properly execute within the LANai. \texttt{lanai-emx} listens for messages not only from the network but also from the \texttt{host-emx}. Also, the original Emerald interpreter was designed for TCP/IP using sockets. This was redesigned for the Myrinet network due to the lack of socket support in the LANai environment. The following topics are covered to give an understanding of the different factors involved in building the communication protocol for \texttt{lanai-emx}.

• Design of communication protocol in \texttt{lanai-emx}

• Communication from LANai to Host

• Communication from LANai to Network

• Communication from Host/Network to LANai

• Reading/Writing data to the Myrinet network

#### 4.3.1 Design of Communication Protocol in \texttt{lanai-emx}

The design of the communication protocol for the \texttt{lanai-emx} is portrayed in Figure 4.5. The \texttt{lanai-emx} supports message exchanges between (a) the LANai and the
network, (b) the host and network, and (c) the host and the LANai. This adds to the complexity of the lanai-emx when compared to the original Emerald interpreter.

4.3.2 From LANai to the Host

The function sendToLocalhost() handles writing out messages from the LANai to the host. It sends messages using MCP_RECV_INFO of the shared memory structure discussed in Section 4.4. Its pseudocode is shown in Figure 4.6.
void sendToLocalhost(void *buffer, int message_len) {
    wait till send_bit is ready;
    setup DMA of buffer to the host;
    wait for DMA to complete;
    inform host that there is a message waiting for it;
}

Figure 4.6: Pseudocode for Sending Packets from LANai to Host

4.3.3 From LANai to the Network

The function sendToNetwork() is called by processIncomingMessages(), discussed in Section 4.3.4, to route packets from the host to the network. It is also used by vmThreadSend(), which handles outgoing messages for lanai-emx from the LANai address space. vmThreadSend() of lanai-emx, has been modified to write out the messages to the Myrinet interface (via sendToNetwork()) from its original sending of objects using the Ethernet interface. The pseudocode of sendToNetwork() is given in Figure 4.7.

void sendToNetwork(void *buffer, int message_len, in_addr dest_ip) {
    (Note that the ready_bit of LANai->network need not be checked,
    as this function returns only when the DMA from LANai->network completes)
    find the route from me to dest_ip;
    if (no route exists)
        return error;
    else {
        write the route to dest_ip to network;
        write the packet header type to network;
        write message_len to network;
        setup the DMA of buffer to network;
        wait for the DMA to finish;
        return;
    }
}

Figure 4.7: Pseudocode for Sending Packets from LANai to Network
4.3.4 From Host/Network to the LANai

The function \texttt{processIncomingMessages()} handles all incoming messages from both network and the host. This function gets called periodically by the main loop of the \texttt{lanai-emx}. Its pseudocode is outlined in Figure 4.8.

```c
void processIncomingMessages()
{
    /* From host to the LANai */
    if (the ready bit of host->LANai is set) {
        receive message from host;
        extract its header;
        if (the message is for me)
            store it in the receive buffer for processing;
        else /* it is for other VM in the network */
            DMA message to the network (Figure 4.7);
        clear the ready bit of host->LANai transfer;
    }

    /* From network to the LANai */
    if (the ready bit of network->LANai is set) {
        DMA message from network;
        extract its header;
        if (the message is for me)
            store it in the receive buffer for processing;
        else /* it is for the host */
            DMA the message to the host (Figure 4.6);
        clear the ready bit of network->LANai transfer;
    }
}
```

Figure 4.8: Pseudocode for Processing Incoming Packets to the LANai

4.3.5 Reading/Writing from/to the Myrinet Network

The pseudocode for reading a packet from the Myrinet network is shown in Figure 4.9 and writing a packet to the Myrinet is shown in Figure 4.10. Understanding of the LANai registers and flag were necessary in writing code for accessing the Myrinet network. However, low level details are beyond the scope of this section.
void readPacketFromMyri() {
    /* the following is executed only after checking
       if a packet has arrived */
    read RH register for packet header;
    if (it is not my header type) {
        consume packet and throw it away;
        return;
    }
    read RW register for the packet length;
    /* a packet should have at least 1 word */
    if (length <= 0) {
        consume it and throw it away;
        return;
    }
    use RMP and RML registers to setup DMA
    from network to LANai memory;
    wait for DMA to complete;
    read the CRC; /* read the tail of the packet */
    report CRC errors if any;
    if (tail bit was not received)
        consume extra bytes and continue;
}

Figure 4.9: Reading Packets from the Myrinet Network

The function readPacketFromMyri() is used by processIncomingMessages() discussed in Section 4.3.4. The function sendToNetwork, discussed in Section 4.3.3, uses writePacketToMyri() to write packets out to the network.

4.4 Message Routing

A standard MCP, or a program executing on a network adaptor, sends messages from the host to the network, or from the network to the host. The lanai-emx performs processing of the messages, in addition to just routing them between the host and the network. Hence, the routing function of the lanai-emx is required to be more intelligent than that of the original Emerald interpreter, because messages from the network or from the host could be for the objects in its own address space.

As discussed in Section 4.3.4, the router part of the lanai-emx extracts head-
void writePacketToMyri() {
    wait until the packet interface says we can send;
    send the packet header which consists of the route
determined plus a tag; {
        use the SB registers to write the route->path
calculated one hop at a time;
        /* this is the tag */
        write out the packet header in the SH register;
    }
    write out the packet length in the SW register;
    setup LANai->myrinet DMA using SMP and SMLT registers;
    wait for DMA to complete;
}

Figure 4.10: Writing Packets to the Myrinet Network

rs from incoming packets, either from the host or from the network, in order to
properly route the message to the indicated destination object. The destination ob-
ject might be in its address space in which case the message is placed in its incoming
buffer. Otherwise, the message is routed to the network or the host appropriately.
Similarly, handling of outgoing packets from the LANai is handled appropriately,
depending on the route, as discussed in Sections 4.3.2 and 4.3.3.

The routing protocol implemented is a modified version of the routing code
obtained from the BDM package [HDMS97] at Mississippi University. The original
version would read the connections of the computers to the switch ports from a static
file. As the OS-Support protocol implemented requires the assistance of the host,
using it would require host assistance for routing every single packet. The purpose of
the entire porting of the interpreter to the LANai, to offload the host and save CPU
cycles, would be lost. The OS-Support protocol had to be designed to support file
handling, in order to load the Builtin file initially. However, other protocols which
use files frequently, and hence need constant host assistance were redesigned. The
following sections discuss the data structures and routing function implemented to
route packets by the shortest path into the network to the appropriate destination.
4.4.1 Data Structures Used for Routing

The static file information, describing the network connections, were stored in a static array. The modified `switch_info` data structure which was used to store the information for each switch is shown in Figure 4.11 and the `switch_conn` structure it uses, follows in Figure 4.12.

```c
typedef struct
{
    int num_ports;
    unsigned int hosts[MAX_PORTS];
    switch_conn switches[MAX_PORTS];
    int visited;
} switch_info;
```

Figure 4.11: Data Structure to Store Information for Each Switch

```c
typedef struct
{
    int switch_num;
    int port_num;
} switch_conn;
```

Figure 4.12: `switch_conn` Data Structure

4.4.2 Routing Information Protocol

The `lanai-emx` calls the function in Figure 4.13 to find the routing path to the destination node. On return, `route->path` stores shortest routing path from `host.ip` to `dest.ip`, and `route->length` stores the number of hops to traverse to reach the destination. The routing function uses some basic routing code from the BDM package, but a lot of functions had to be modified for the `lanai-emx`.
void Map (MCP_Route *route, unsigned int host_ip, unsigned int dest_ip)
{
    int my_switch, my_entry_port, hops, i;
    int tmp_route[MAX_ROUTE];
    use the static information array to find the switch and
    and the port to which the local host is connected;
    if ( (my_switch == -1) or (my_entry_port == -1) ) {
        my local host_ip is not in routing table;
        report error;
    }
    else {
        hops = Find_route (tmp_route, my_switch, my_entry_port, dest_ip);
        if ( hops == (MAX_ROUTE + 1) ) {
            could not find connection from host_ip to dest_ip;
            report error;
        }
        /* tmp_route is backwards, so access it from end to front */
        for (i=0; i<hops; i++)
            route->path[i] = ((tmp_route[hops-(i+1)]&0x3f)|0x80);
        route->length=hops;
    }
}

Figure 4.13: Myrinet Routing Protocol

4.5 Other Issues

The following are miscellaneous issues encountered while designing Emu, and the
changes made to the host-emx, both to assist the lanai-emx by providing OS support,
and to communicate with other machines via the Myrinet network.

4.5.1 Emerald over Myrinet Vs. Emerald over IP over Myrinet

Normally, an Emerald machine communicates with another machine via a TCP
stream, using sockets for sending, receiving or listening for messages. But in a
Myrinet cluster, the messages need to be packaged as Myrinet packets. The original
Emerald packet headers did not need to specify the destinations since the TCP/IP
protocol took care of copying the destination address in their packet headers.

Emerald over IP over Myrinet would result in three levels of traversal when
compared to the two levels in the case of Emerald over Myrinet. To satisfy the
latter case, the Emerald packet headers had to accommodate an extra field to store
the destination node. Changing the Emerald packet headers resulted in changing
different routines like \texttt{startMsg()}, \texttt{sendMsg()}, \texttt{forwardMsg()}
which handle outgoing
packets. Marshalling routines for assembling an outgoing packet and unmarshalling
routines on an incoming packet had to be modified to support the added destination
attribute.

4.5.2 Issues Concerning Hardware Resets

If the LANaI is being bombarded with a lot of messages from the network, the
board is sometimes hard reset by the network. The two least significant bits of
the TIMEOUT register (refer to Section 3.1.1) determine the timeout period by
which the LANaI chip must consume packets from the network. If the LANaI chip
fails to do so within this period, the NRES signal is asserted. This causes the
NRES\_ENABLE of the MYRINET register to the reset the LANaI chip. So far, in
our application, this has not occurred as high priority is given to messages from the
network.

4.5.3 Modifications to the \textit{host-emx}

The Emerald interpreter on the host had to be modified to communicate with the
lanai-emx. The \textit{host-emx} is a modified version of the model described in Section
3.4.2. The following items discuss the changes to the original interpreter running
on the host:

- Communication medium support:

  The \textit{host-emx}, of the \textit{Emu} system, was modified to listen for messages on
  the Myrinet interface rather than the Ethernet interface. It checks whether
  the \texttt{ready.bit} of the \texttt{MCP\_Recv\_info} shared structure is set for it to receive
  messages. If so, it copies the message to its incoming buffer and clears the
  \texttt{ready.bit}. For sending messages, \texttt{vmThreadSend()} of the \textit{host-emx} is modified
to set the \textit{ready.bit} of the \texttt{MCP.Send.info} shared structure, in order to inform the interpreter on the LANai that a new message has to be DMAed from the host.

- Functionality to support \textit{lanai-emx}:
  Support for \textit{lanai-emx} with files, debugging, system calls and other basic information exchanges were discussed in Section 4.2.

- Modification to Packet Headers:
  As in the case of the \textit{lanai-emx}, the headers had to be changed to accommodate the destination field for communicating with the Myrinet interface. The routines that use the packet headers had to be modified to support the new format.

- Byte-order:
  LANai uses the big-endian format whereas the PCs use the little-endian format for the representation of integers. This initially caused a lot of handshaking problems between the LANai and the host when trying to read/write the \textit{handshaking} variable. It was however difficult to pinpoint this problem as there was no debugging tool at the commencement of the project.

- Heterogeneous Processor Environment:
  Initially, the Emerald interpreter, on the host, only had to process packets from other nodes which are also running in a similar environment (Unix), while it now has to assist the LANai as well. Coordinating all these interactions required proper interleaving of function calls. For example, the \textit{host-emx} has to assist with handling file support, debugging support, etc. for the \textit{lanai-emx} in addition to coordinating with the LANai in sending and receiving messages via the Myrinet interface.
4.6 Summary

This chapter discussed the various issues involved in the design and implementation of *Emu*. It discussed the problems with the LANai environment in running the Emerald interpreter and the protocols implemented to resolve the different processing environment conflicts. Second, the communication issues for the new network, and the complexity of the mux/demux function was explained. Third, the new routing functionality introduced by *lanai-emx* was discussed. The new routing information protocol implemented was discussed at a high level, explaining the technique of routing involved in the Myrinet interfaces and switches. Finally, miscellaneous issues that arose during the design of *Emu* were discussed including the changes needed to *host-emx* to communicate with the *lanai-emx*.

The Myrinet interfaces can now run an Emerald interpreter, thereby executing Emerald packets. Emerald objects circulate within the Myrinet network and can be processed either by the *lanai-emx* or the *host-emx*. Thus, an interpreter on a network adaptor can do so much more than just a simple routing program as is the traditional case.
Chapter 5

Performance

This chapter is concerned with the performance of the Emu system and the applications that benefit from the interpreter running on the LANai. Section 5.1 discusses the micro benchmarks run on Emu to measure the slow down of certain operations on the network processor when compared to the central processor of the computer. The results of the micro benchmarks support the expected claim that the LANai processor is better for communication intensive processing while the host processor is better for computation intensive processing. Two applications were used to test the Emu system, or to be more explicit, the usefulness of migrating functionality to the LANai. The first application, kilroy, is a standard Emerald program, and the second application, locator, is designed to locate services among the workstations in the cluster. The final section concludes with a discussion of the usefulness of Emu.

5.1 Micro Benchmarks

Two categories of micro benchmarks were used to compare the performance between the lanai-emx and the host-emx. The first category dealt with computation-intensive operations and the second concentrated on communication-intensive functions. The experimental framework of the system is described in detail in Section 3.3.
### Table 5.1: Computation Intensive Micro Benchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>host-emx runtime in microsec</th>
<th>lanai-emx runtime in microsec</th>
<th>LANai Slow down factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (2)</td>
<td>2.7</td>
<td>15.8</td>
<td>5.9</td>
</tr>
<tr>
<td>multiply (7)</td>
<td>2.7</td>
<td>24.5</td>
<td>9.1</td>
</tr>
<tr>
<td>divide (436 / 17)</td>
<td>2.9</td>
<td>25.2</td>
<td>8.7</td>
</tr>
<tr>
<td>invoke</td>
<td>3.6</td>
<td>20.6</td>
<td>5.7</td>
</tr>
<tr>
<td>create</td>
<td>3.4</td>
<td>21.5</td>
<td>6.3</td>
</tr>
</tbody>
</table>

#### 5.1.1 Computation Intensive Micro Benchmarks

The micro benchmarks under the first category are listed in Table 5.1. Each of the operations were run one million times and the averages are reported for accuracy. The LANai timings are measured using the host's clock because the LANai's clock does not report time accurately. As each of the operations are repeated one million times, the slight increase in contacting the host for the runtimes on the LANai can be ignored.

The second and third columns of Table 5.1 report the runtimes of the reported operation on the host processor and the network processor respectively. The fourth column is the ratio of the runtime on the LANai to that on the host. The operations in the first column are discussed in detail below:

1. add (2): adding 2 to an integer object.
2. multiply (7): multiply an integer object by 7.
4. invoke: invoking another object.
5. create: creating an object.

Note that the reason for the arguments 7 and 436 / 17, in the multiply and divide operations respectively, is to avoid any hardware support provided for arguments such as 1 or 2.
The slow down in the last column of Table 5.1 is almost as expected. The clock rate or the speed of the host (Pentium-II) processor is 266MHz, and that of the LANai processor is 33MHz [BCF+95]. So the network processor is slower than the host processor by a little more than a factor of 8. Even though the clock rate ratio of the host to the LANai is 8, the measured slow down ratio is in the range of 5.7 to 9.1 for the different operations. The slowdown on the network processor cannot be avoided due to the lagging speed of the LANai processor compared to the host processor.

5.1.2 Communication Intensive Micro Benchmarks

Table 5.2 contains the results of communication intensive operations. Each of the operation were run one thousand times and the averages are reported. In the table, the acronym RTT stands for Round Trip Time. The explanation of the communication intensive operations, and the results reported are discussed below.

Communication times along the four paths: (i) from one host to another host via the Myrinet network, (ii) from host to the local LANai, (iii) from host to a remote LANai, and (iv) from one LANai to another LANai, are reported for each of the different operations in the first four rows. The fifth row reports the slow down on the LANai, which is the ratio of the communication on the LANai to LANai path to that on the host to host path. The description of the operations executed in all the four possible paths are discussed below.

1. Move Obj RTT: move an object back and forth.

2. Move Self RTT: move the executing process object back and forth.


It is clear from the results, that as expected, communication intensive operations fare better than computation intensive operations. The slow downs reported in the last row are all less than a factor of 6. In fact, in the cases of moving another
<table>
<thead>
<tr>
<th>Path</th>
<th>Move Obj RTT in microsec</th>
<th>Move Self RTT in microsec</th>
<th>Remote Invoke in microsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host to Host (HH)</td>
<td>1854.7</td>
<td>2020.5</td>
<td>1041.2</td>
</tr>
<tr>
<td>Host to locLANai</td>
<td>936.8</td>
<td>5076.9</td>
<td>965.9</td>
</tr>
<tr>
<td>Host to remLANai</td>
<td>2418.2</td>
<td>6855.8</td>
<td>1721.3</td>
</tr>
<tr>
<td>LANai to LANai(LL)</td>
<td>3064.2</td>
<td>11827.4</td>
<td>2336.9</td>
</tr>
<tr>
<td>Slow down (LL/HH)</td>
<td>1.7</td>
<td>5.9</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 5.2: Communication Intensive Micro Benchmarks

object or remote invocations, the slow downs are hardly noticeable considering the measurements are in microseconds. In such cases, avoiding host interrupts can be beneficial if the host is involved in a heavy computation intensive workload.

It is difficult to give the exact reasons for this almost negligible slow down in the two cases **move obj RTT** and **remote invoke**. For a network processor that is slower than the host processor by a factor of 8, the slowdown in the range of 1.7 to 5.9 was quite unexpected. A reasonable explanation, for the **move self RTT** slow down to be greater than the other two cases, is because it is more computation intensive. In the case of moving the current executing process object, all its activation records also need to be moved to the remote node. This extra computation, of figuring out the state of the executing object to be migrated, is not required in the other two cases.

### 5.1.3 Discussion

The micro benchmarks support the thesis statement in Chapter 1 that communication intensive components of an application are better off on the network processor, if extensive computation intensive processing is also needed which can occur on the host processor. The lagging clock rate of the LANai processor makes it unsuitable for computation intensive processing.
5.2 Application 1: Kilroy

*kilroy* is a simple Emerald program which tests whether an object that is started on one node can migrate to all other nodes in the Emerald world and return to the originator successfully. This application is useful for testing whether each node has state about every other node in the Emerald world.

The scenario, shown in Figure 5.1, is just one particular path taken by *kilroy* in traversing the *Emu* world. For clarity, it is limited to three nodes in the figure. There were two reasons why *Kilroy* was run in the *Emu* system.

![Figure 5.1: Kilroy on Emu](image)

First, *kilroy* tests the robustness of *Emu*. In the *Emu* world, there are two interpreters on each host, one on the host and the other on the NIC. So, *kilroy* on *Emu* will run on 32 Emerald virtual machines in the 16-node Pentium-II cluster. This process object successfully visited all nodes in the *Emu* system and validated the communication protocols used.
Second, a process object which migrates from one node to another provides the framework needed for active networking applications. The application can be easily modified to visit only the lanai-emxs of Emu. As the Myrinet network is dynamically reconfigurable, kilroy can be modified to do dynamic routing in the LANai without host intervention. As the network increases in size and complexity, kilroy can modify the routing tables in the SRAM. In this way, it extends the static routing function in the LANai for dynamic routing. Emu makes this possible because it is designed to be extended at runtime, unlike systems such as Hamlyn [BJM+96] or U-Net [vEBBV95].

5.3 Application 2: Location Service

This application was designed and implemented to provide a generic location service to different types of applications. The following sections discuss the design of the service, the experimental setup, the efficiency of migrating the locator to the LANai and examples of applications that might benefit from it.

5.3.1 Design of the Location Service

The location service provides an API for applications that need to locate services in some part of the distributed environment. The location service is provided by two classes, locator and dataFinder, listed in Figures 5.2 and 5.3 respectively. Locator is responsible for maintaining information about the services available at the local node. dataFinder is the class that is invoked by applications to find the location of particular services either on the local node or on remote nodes. This requires dataFinder to maintain a list of all of the locators existing in the cluster.

The locator is a monitored class which provides mutual exclusion to protect its shared state when different dataFinders try to access or modify the state of a particular locator at the same time. It provides the basic operations to:
const locator <- monitor class locator
attached const local <- Set.of[Service].create[320]

export operation add[ind: Service]
local.add[ind]
end add
export operation remove[ind: Service]
local.remove[ind]
end remove
export operation objReq[ind: Service]->[result: Boolean]
result <- local.contains[ind]
if result then
local.remove[ind]
end if
end objReq
export operation isitHere[ind: Service] -> [result: Boolean]
result <- local.contains[ind]
end isitHere
end locator

Figure 5.2: The Locator Service Class

- add a service
- remove a service
- check whether a particular service is available at the local node
- be remotely invoked by other objects when requesting a remote service

The dataFinder is the class that is called by applications wishing to locate services. When requested to find a particular service, it checks whether it is available locally. If it is locally available, no remote operations are required. If the requested service is not locally resident, dataFinder serially multicasts requests to other locators in the cluster until it finds the service. It reports failure to the application if none of the nodes provide the requested service.

The dataFinder class which exports one operation could have been easily merged with the locator class, as this was the initial design of the location service. The reason for separating the location service into two classes is to avoid deadlock if one monitored locator tries to access another locator which is in turn trying to
const dataFinder <- class dataFinder
    attached const field locators : Vector.of[locator] <-
        Vector.of[locator].create[16]
    attached field count : Integer <- 0
    export operation get[ind: Service] --> [result: Boolean]
        result <- locators[0].isItHere[Service]
        if !result then
            for i : Integer <- 1 while i < count by i <- i + 1
                result <- locators[i].objReq[ind]
                if result then
                    locators[0].add[ind]
                    exit
                end if
            end for
        end if
    end get
end dataFinder

Figure 5.3: The Service Finder Class

access the former at the same time. In the final version, no monitored operation
invokes another monitored operation. This confirms the avoidance of deadlock in
this service.

The ease of designing locator clearly exemplifies the advantage of using Emer­
ald for a distributed system such as Emu. Each of the consistency, remote invocation
and migration services required by the location service are inherently provided by
Emerald. This saved considerable time in implementing locator whose naive version
is less than 100 lines of Emerald code.

The real difficulty in successfully completing the application was in deter­
mining the data structures that had to be used because of the memory limitations
of the LANai. For example, using an array to keep track of locators in the Emu
world failed when the location service was migrated to the LANai. The lanai-emx
was unable to malloc memory for the array data structure. Hence, the vector was
used instead as shown in Figure 5.3. Vector is the most primitive mechanism in
Emerald for indexable storage. This builtin object is used in the implementation of
the array in Emerald. Vector is a mutable object and supports the basic subscripting
operations. Using arrays would have been more appropriate, because they can dynamically grow, unlike vectors, and hence can be extended at runtime appropriately as the number of nodes in the cluster increase.

5.3.2 Experimental Setup

The experimental framework is described in Section 3.3. A dataFinder and a locator is created for each application on the host. The application invokes the dataFinder to locate a service. The dataFinder maintains state about other locators, by invoking the getlocationServer operation on all the remote host nodes, in the Emu world. The dataFinder checks its local locator and if it receives a negative reply, invokes the remote locators. Experiments were performed for two configurations.

- Configuration 1: In this case, locator, dataFinder and the application execute on the host-emx of Emu as shown in Figure 5.4. In this situation, the lanai-emx of Emu is involved only in the routing of messages.

- Configuration 2: In the second case, the locator and dataFinder are migrated to run on the lanai-emx with the application running on host-emx as shown in Figure 5.5.

The two configurations can be compared with the help of Figures 5.4 and 5.5. In these figures, only three hosts in the cluster are depicted for clarity. The number of I/O crossings, interrupts and context switches avoided in Figure 5.5, when compared to Figure 5.4, suggest that there are significant advantages in migrating the location service to the LANai. The next section discusses the benefits achieved, and the overhead avoided by running the location service on the LANai.

5.3.3 Efficiency of Emu

Migrating the location service to the LANai alleviates the I/O bottleneck discussed in Chapter 1. The advantages of the Emu system are clearly demonstrated in the
scenario where only 3 hosts are considered.

- Traversal of the I/O bus is reduced as all the remote invokes and their replies are handled by the lanai-emx. In configuration 1, a maximum of 6 traversals of I/O bus is reduced to a constant 2 traversals in configuration 2.

- Unnecessary main memory copies across the I/O bus are avoided. All remote invocations in configuration 2 use the LANai SRAM.

- There is always only 1 interrupt on the host for any invocation of the location service.

- The reduction of interrupts avoids context switches which consume a significant amount of the CPU time.

Each of the benefits, listed above, result in lower demands placed on the host CPU, freeing more of its resources for useful work. In this evaluation, maximum
of 2 context switches is achieved for any request on dataFinder even in the worst case scenario where all remote locators have to be invoked. This constant is valid if the number of nodes is 3 or 100. This evaluation can be generalized to $n$ nodes of the cluster. The following equations estimate the time spent in locating a service in the worst case where all $n$ locators have to invoked for the two configurations.

$$CT(n) = n(t_{CP}) + n[2(t_c + t_{pci} + t_{mem}) + t_{int}] + 2(n-1)(t_{net})$$  \hspace{1cm} (1)

$$NT(n) = n(t_{NP}) + [2(t_c + t_{pci} + t_{mem}) + t_{int}] + 2(n-1)(t_{net})$$  \hspace{1cm} (2)

where $CT(n)$ and $NT(n)$ are the times to locate a service, in the worst case, for Figures 5.4 and 5.5 respectively. The factors involved are:

- $n$ is the number of hosts in the cluster.
- $t_c$ is the time spent in context switching in the kernel.
- $t_{int}$ is the kernel interrupt time.

- $t_{pci}$ is the PCI traversal time for DMA between the host and the LANai which is approximately 15.8 microseconds for 256 bytes.

- $t_{mem}$ is time spent in memory copies in the host which is approximately 1.02 microseconds for 256 bytes, assuming the data is in the cache.

- $t_{net}$ is time spent for each DMA from/to LANai to network which is a little less than 15.8 microseconds for 256 bytes.

- $t_{CP}$ is processing time on the CPU which is about 10.3 microseconds.

- $t_{NP}$ is processing time of the network processor which is approximately 59.4 microseconds (about 5.7 times that of $t_{CP}$).

The times reported for $t_{pci}$ and $t_{mem}$ are for messages of size 256 bytes as the request messages are small. There is only one final response message of size 6 KB for any service request, and hence that is not taken into calculation.

Moving the location service to the LANai saves about $n(15.8 + 1.02)$ microseconds in host/LANai DMAs and memory copies on the host. The measured network processing time, however, is on average about 5.7 times slower than the measured host processing time. Thus, the host overhead savings is accompanied by an increase in the processing times on the LANai.

From the equations it is clear that context switches, memory copies and interrupts grow with the number of nodes in equation (1), but remain a constant in equation (2). For a sample run of 256 random service requests in a 6-node cluster, for example, the savings in context switches amounted to about 892 and interrupts to about 446, just by moving the location service to the LANai. Thus, when the location service runs on the LANai, this savings in processing time can be spent by the $n$ hosts to do useful work.
5.3.4 Applications that could Use the Location Service

Most applications that are built for a distributed environment need some kind of location service if the services are distributed and no central state information is maintained. Even if state is maintained, services could migrate and correct state information update might be impossible. In such a case, multicasting is often done to locate the requested service. The location service, implemented, is a generic service demonstrated by using integers. However, Emerald code could be easily modified to use any other type defined by the programmer. It could also be optimized to perform in a more intelligent manner. Examples of systems that might use this service are:

- Distributed directories where the data type would be strings or files.
- A global memory system which would use pages as the service data types.
- Parallel matrix multiplication where the data type would be matrices.

5.4 Discussion

The main motivations for Emu, discussed in Chapter 1, were to alleviate the I/O bottleneck and to offload all unnecessary processing from the host. The locator implemented clearly demonstrates that Emu has satisfied the goals, stated in Chapter 1, which are: avoiding context switches, interrupts, unnecessary memory copies and I/O traversals, and utilizing the processing power on the network processor for more useful work than basic flow control.

Emu is an attempt to focus on reducing the overhead involved in message transmissions as suggested by the analysis in [MVCA97]. In [MVCA97], Martin et al. state that applications demonstrate strong sensitivity to overhead involved in sending or receiving a message and the gap between transmissions. But, they are surprisingly tolerant of increased latency and lower per-byte bandwidth. Their
results obtained using a wide variety of chosen applications verify their claim. The Emu system reduces the overhead by eliminating host processing as much as possible. The system would work well if the right applications are chosen for migration to the network processor.

Applications that are computation intensive are not going to benefit running on the lanai-emx because the network processor is slower than the host processor by a factor of 8. Emu provides support for Emerald object processing on the network processor. It does not decide whether to execute on the central processor or the network processor. It is the responsibility of the programmer to make that decision. Components of applications that are communication intensive can benefit from Emu’s support of processing on the NIC. Intelligent offloading of communication-intensive components, from the host to the LANai, will save host processing power for computation-intensive components that are better off executing on the host-emx.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

The Emu system has made possible the complete implementation of an interpreter on the network processor. The choice of an object oriented distributed interpreter, such as Emerald, makes the applications that run on the system more modular and compact because the language takes care of all the gory details. Emu has introduced runtime extension of the functionalities on the network processor, and an approach for utilizing idle network processing power.

Emu has satisfied all the thesis goals stated in Chapter 1: first, migration of the Emerald interpreter to the stand-alone embedded environment of the LANai has been completed; second, the interpreter running on the central processor has been modified to communicate via the Myrinet interface; third, the I/O bottleneck problem has been attacked successfully. Unnecessary host interrupts, associated context switches, memory copies and CPU cycles can be avoided with the help of the interpreter running within the network processor. Emu can improve the throughput of the system if intelligent decisions are made in choosing those components of applications that will be run (optimally) on the network processor. Emu does not make migration decisions, and leaves that responsibility to the end application.
However, it provides all the support necessary for the actual migration process.

Performance of the *Emu* system in satisfying these goals is supported by the locator service. A formula has been derived which compares the total processing time in locating a service, when processing is done on the central processor, to that obtained by moving the processing to the network processor. The *Emu* system only requires a single interrupt, when the location service executes on the LANai, in locating a service in a cluster with increasing number of nodes, even in the worst case scenario when all the nodes have to be invoked.

*Emu* has proved that processing on the network processor can increase or at least maintain similar overall system throughput, while at the same time saving host processing, for more useful work, during heavy computational workload. However, intelligent delegation of processing to the network processor is required, because the network processor is slower than the central processor by a factor of 8. Communication-intensive applications which require low latency and high throughput would benefit from the interpreter on the network adaptor in the *Emu* system.

*Emu* has paved the way for other similar systems to be built. It has demonstrated that it is possible to take full advantage of the programmable NIC to increase system throughput. It has shown that with intelligent choice of applications, idle network processing power can be utilized without sacrificing performance.

### 6.2 Effort Required in Creating *Emu*

Implementing *lanai-emx*, in the *Emu* system, was writing programs for the bare hardware, with not much support for programming. This required learning the intricate details of the NIC registers and Myrinet network. Although the design was simple, the actual implementation was a painful learning experience. This can be attributed to the different environment of the LANai processor and the existence of bugs in the Emerald interpreter code which became more pronounced when the communication interface was modified to use the Myrinet network.
The main limitations of the LANai processor were the lack of an OS, lack of standard C library support in the lanai3-gcc compiler, 1MB of SRAM memory and no debugging tools. Protocols were built to provide OS and libc support, and debugging tools. The 1MB of SRAM forced a smaller and a more compact version of the Emerald interpreter. In fact, the optimized version of the interpreter code, with no debugging support, is only about 209 KB of compiled C code.

The implementation of Emu helped make Emerald more robust. First, it helped find the dormant bugs that became active with the introduction of the new interface. Examples of not completely correct code were found in move and remote invocation implementations. Second, in the original version there were tables for mapping oids (these are numeric object IDs) to objects and objects to oids. Due to the memory limitations, the implementation was modified to have both the mappings in a single table with a more complex mapping function but with a 50% savings in memory usage. Third, in the original version, the distribution package was implemented using user level threads. Due to lack of a thread utility, and space constraints in the LANai 4 processor, the current version of Emerald supports a single process distribution package for the LANai as well the original package with threads.

Emu would significantly reduce system creation for any future research that would make such extensive use of the network processor. It has demonstrated the difficulties that one would encounter while implementing such systems. The debugging tools and the protocols implemented, to overcome the limitations of a network adaptor with a different environment, would help in providing the steps to take for building a similar system. Thus, Emu not only shows that such a system is possible but also would be useful research for future systems on the NIC.

6.3 Future Work

Emu is a system that provides support for network processing on the LANai pro-
cessor. But it could be improved to perform better with some optimizations and modifications to certain parts of the Emerald interpreter. The following sections discuss some features to help improve the Emu system and suggests some additional applications that can benefit from this system.

6.3.1 Memory Fragmentation Problems

The malloc and free routines, implemented to allocate and free space from the LANai heap, do not provide for the defragmentation of memory when it is freed. This causes fragmentation of memory which creates havoc in the already stringent heap space allowed which is about 256 KB of SRAM.

Hence, even though pages are freed as expected, malloc reports failure if space is requested for a large number of pages. This is because there is no one contiguous block of memory available for the request to be satisfied. If some sort of defragmentation is implemented for the heap area, more efficient utilization of memory will be possible.

6.3.2 A More Precise Debugger for the LANai

The debugger implemented for the lanai-emx part of Emu is the simple approach using printf stubs. It has helped solve the memory problems in Emu but was not easy to use. For example, there is no way to read the point of execution in the program, such as is possible with the where and stacktrace commands in gdb, or to display the values of the variables when the program on the LANai dies due to segmentation faults.

A more precise debugging tool would avoid unnecessary printing during the process of debugging memory faults. The debugger could help with providing only the necessary information when the program fails.
6.3.3 Performing GMS lookups on the LANai

One of the applications that would benefit from Emu is the global memory system [Fee96]. It has two main operations `getpage()` and `putpage()`. GMS makes extensive use of the communication bandwidth. If these requests are directed to an engine on the LANai which can DMA pages from/to host memory without consuming CPU cycles, the host processor need not be bothered with remote requests. In this way, it lends its memory to trusted hosts but is not disturbed in order to process any remote messages.

If the hashtable of the GMS is moved to the LANai, the `getpage()` and `putpage()` requests can be successfully handled by the LANai. It could make use of the location service which was explained in the previous chapter. Moreover, most of the processing is in the transferring of pages which makes these requests communication intensive. Hence, the lower speed of the network processor when compared to the central processor can be ignored.

6.3.4 Support for Active Networking on the NIC

Kilroy, in chapter 5, demonstrated the capabilities of Emu to provide support for Active Networking. Applications such as this could be created, following kilroy's pattern, which run only on the NICs and do useful work as they migrate, such as checking state and modifying it appropriately. For example, an active object migrating among the NICs can be responsible for maintaining up to date information in the hash tables of the GMS application described above.

6.3.5 Dynamic Routing Application

The concept of supporting an intelligent interpreter engine on the NIC, can be extended to a local area network. But in a local area network, the unpredictable latency in traversing the network might mean that traditional paging to disk is faster than paging to global memory. However, other applications might benefit from the
Emu concept.

If the engine on the NIC is modified to make intelligent routing decisions based on (i) network partitioning, (ii) heavy traffic on the standard route, and (iii) a new shorter route becoming available, then a dynamic routing scenario is established. First, the engine on the NIC need not bother the host, thereby avoiding interrupts and context switches among other things. Second, manual intervention is unnecessary because the network can automatically reconfigure itself to changes, almost instantaneously, which is not the case otherwise.

6.3.6 HPJava Engine as Framework for Emu

The HP Java Virtual Machine [Vij98] is designed for embedded systems with resource constraints such as the Myrinet NIC, with memory limitations of 1MB of SRAM. It complies with Sun’s Java Virtual Machine specifications, but with its 0.5MB of code, provides more flexibility. Using HP Java as the framework for Emu instead of Emerald would be an interesting future work. With Java being so popular, using it as the basis for Emu would make the latter more easily accepted as a new programming model for a cluster.

6.4 Final Remarks

The Emu system is a model that supports migration of active objects to the network processors, thereby offloading the hosts for other useful work. The runtime migration of services to the network processor is made possible by the interpreting engine designed for the NIC. It provides all the features to support active networking applications, as stated above. Emu has proved that processing on the network processor is possible. But good throughput in such a system depends on proper delegation of responsibilities to the NIC, keeping in mind that the network processor is much slower than the host processor. Unless the network processors are made faster, the performance in such a system would be limited by the speed of the network proces-
sor. *Emu* has provided a model to avoid the I/O bottleneck by utilizing idle network processing power, but the new network processing bottleneck introduced has to be tackled for more complex processing to occur on the network interface cards.
Bibliography


[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed


94


Alexander B. Schill and Markus U. Mock. DCE++ : Distributed Object-Oriented System Support on top of OSF DCE. Technical report, Institute of Telematics, University of Karlsruhe, Germany.


[TW96] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26, Nr 2, April 1996. An earlier version of this paper was delivered during the keynote session of Multimedia Computing and Networking, San Jose, CA, January 96.


