INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Performance of multicomputers using high-speed communication links

Rizvi, Haider Abbas, M.S.

Rice University, 1993
RICE UNIVERSITY

Performance of Multicomputers using High-speed Communication Links

by

Haider Abbas Rizvi

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

J. B. Sinclair
Dr. J.B. Sinclair, Chairman
Associate Professor
Electrical and Computer Engineering

[Signature]
Dr. J. R. Jurek, Professor
Electrical and Computer Engineering

[Signature]
Dr. P.J. Varman, Associate Professor
Electrical and Computer Engineering

Willy Zwaenepoel
Dr. W. Zwaenepoel, Associate Professor
Computer Science

Houston, Texas
April, 1993
Abstract

Performance of Multicomputers using High-speed Communication Links

by

Haider Abbas Rizvi

This thesis presents the results of a simulation study of the performance of a message-passing multicomputer using high-speed point-to-point communication links. The multicomputer system consists of IBM RS/6000 machines linked by 220 megabits per second fiber-optic links. This system is simulated using RIOSIM, a fast, accurate, and flexible execution-driven parallel architecture simulator. An accurate timing profiler, simulating the superscalar capabilities of the RS/6000 at runtime, generates dynamically timing estimates for the instructions executed. The simulation results are validated against actual measurements on a two-processor system, using a variety of algorithms. Results show that the errors are typically around 8%.

The validated model is used to study systems with more than two processors. Simulation results indicate that this setup is suitable for coarse-grained parallel algorithms, some of which show almost linear speedups. For fine-grained algorithms, the high overhead in message passing proves to be a serious bottleneck, resulting in less than linear speedups.
Acknowledgments

I want to express my appreciation and thanks to my advisors, Dr. Sinclair and Dr. Jump, who have helped me in doing this research. They always had their doors open for me, and advised me in solving my problems.

Thanks are also due to the other members of my thesis committee, Dr. Varman and Dr. Zwaenepoel, who agreed to be on the committee. Special thanks to Dr. Sinclair, who painstakingly went through the initial drafts of this thesis, and gave a lot of feedback.

Also, thanks to my officemates, Usha Rajagopalan and Rajat Mukherjee, who have tolerated me for a long time, and have provided a friendly atmosphere in the office.

Financial support for this research was provided in part by the International Business Machines Corporation. I am grateful to the Government of Pakistan for the support of a Science and Technology graduate research scholarship.

Finally, my parents, Sakina and Muzaffar Rizvi, and sisters, Tahira, Shagufta, Rana and Masuma, deserve a special thanks for all the love and emotional support they provided me.
To my parents ...
Contents

Abstract ii
Acknowledgments iv
List of Illustrations viii
List of Tables x

1 Introduction 1

1.1 Motivation .................................................. 1
1.2 Overview .................................................... 2
1.3 Related Work ............................................... 7
  1.3.1 Message-passing systems ................................ 7
  1.3.2 Simulation systems ...................................... 9
  1.3.3 Profilers ............................................... 11
1.4 Organization of the Thesis ................................. 12

2 Design of the RISC System/6000 profiler 13

2.1 Standalone Profiling ....................................... 14
  2.1.1 Generating profiled programs .......................... 15
2.2 RISC System 6000 Processor Architecture ............... 19
  2.2.1 Instruction Cache Unit ................................ 21
  2.2.2 Data Cache Unit ...................................... 21
  2.2.3 Fixed Point Unit ...................................... 22
  2.2.4 Floating Point Unit ................................... 22
  2.2.5 Synchronization between FXU and FPU ................. 23
2.3 Profiler for the RISC System 6000 ....................... 25
3 An RS/6000-based Multicomputer

3.1 An overview of the Fiber-optic links .......................... 39
  3.1.1 Architectural overview .................................... 39
  3.1.2 Characteristics of the serial link .......................... 40
  3.1.3 Message Transmission Delays .............................. 41

3.2 Description of the Multicomputer .............................. 47
  3.2.1 Hardware .................................................. 47
  3.2.2 User programs and system services ......................... 49
  3.2.3 Message routing ......................................... 50
  3.2.4 Acknowledgment of Data Packets ........................ 50
  3.2.5 Description of the message handling system ............... 51
  3.2.6 Packets .................................................. 54
  3.2.7 Assembling Messages at the Final Destination ............. 54

3.3 Summary .................................................................. 56

4 RIOSIM: An Execution-driven Simulation System for
the Target Multicomputer .............................................. 57
  4.1 Execution-driven Simulation Technique ........................ 57
  4.2 RIOSIM: A parallel architecture simulator .................... 59
  4.3 Calculating performance of the simulated multicomputer .... 61
  4.4 Validation of Simulation Results ............................... 63
4.4.1 Algorithms used for simulation validation .................. 63
4.4.2 Results .............................................. 68
4.5 Summary ................................................. 70

5 Performance Studies of the Multicomputer .................. 72
  5.1 Basic Performance Curves .................................. 73
  5.2 Effects of changing packet size ............................. 79
  5.3 Effects of changing transmission speed ..................... 81
  5.4 Effects of changing the message overhead .................. 83
  5.5 Effects of changes in simulated architecture ............... 88
  5.6 Effects of load balancing in LU factorization .............. 91
  5.7 Conclusions ............................................ 91

6 Conclusions ................................................. 93
  6.1 Summary ................................................. 93
  6.2 Future Work ............................................. 94
    6.2.1 Reduction of message overhead ......................... 94
    6.2.2 Simulation of similar multicomputers .................. 95
    6.2.3 Extensions to the message-handling system ............ 95

Bibliography .................................................. 96
Illustrations

1.1 A shared-memory parallel computer .......................... 4
1.2 A distributed-memory parallel computer ....................... 4

2.1 Block diagram for standalone profiling ......................... 16
2.2 Native profiling on a basic block ............................. 17
2.3 Block diagram of the RISC System/6000 processor ............. 20
2.4 FXU-FPU synchronization scheme for the RS/6000 ............. 24
2.5 In situ profiling for the RS/6000 ............................ 28

3.1 Network environment using the serial links .................... 40
3.2 Maximum number of messages buffered by the serial link ....... 42
3.3 Time required for a round-trip for different message sizes ...... 44
3.4 Time required for a round-trip for message sizes from 225 bytes to
   256 bytes ................................................. 44
3.5 Time required for a round-trip for message sizes from 4075 bytes to
   4105 bytes ................................................. 45
3.6 Time required for a round-trip for message sizes from 8180 bytes to
   8204 bytes ................................................. 45
3.7 Time required for a round-trip for message sizes from 12248 bytes to
   12328 bytes ................................................. 46
3.8 Plot of the piecewise linear model of the operating system delays, and
   comparison with measured time difference ....................... 48
3.9 Flow chart of the message handler process ..................... 53
4.1 Block diagram for Execution-driven Simulator

5.1 Time to perform a 512 x 512 matrix multiply
5.2 Speedup curve for a 512 x 512 matrix multiply
5.3 Time to calculate a 512 x 512-elements SOR (200 iterations)
5.4 Speedup curve for a 512 x 512-elements SOR (200 iterations)
5.5 Time to sort 25000 elements using Quicksort
5.6 Speedup curve for sorting 25000 elements using Quicksort
5.7 Time to solve an 18-city traveling-salesman problem
5.8 Time to perform a 200-order LU factorization
5.9 Time to perform a 512 x 512 matrix multiply
5.10 Time to calculate a 512 x 512-elements SOR (200 iterations)
5.11 Time to perform a 512 x 512 matrix multiply
5.12 Time to calculate a 512 x 512-elements SOR (200 iterations)
5.13 Time to perform a 512 x 512 matrix multiply
5.14 Time to calculate a 512 x 512-elements SOR (200 iterations)
5.15 Speedup curve for a 512 x 512-elements SOR (200 iterations)
5.16 Time to sort 25000 elements using Quicksort
5.17 Time to perform a 200-order LU factorization
5.18 Speedup curve to perform a 200-order LU factorization
5.19 Time to perform a 512 x 512 matrix multiply
5.20 Speedup curve for a 512 x 512 matrix multiply
5.21 Time to calculate a 512 x 512-elements SOR (200 iterations)
5.22 Time to perform a 200-order LU factorization
Tables

2.1 Performance of the RS/6000 profiler .................................. 35

3.1 Model for the message handling delays in operating system ....... 47
3.2 Header of the data packets ............................................. 55

4.1 Validation Results for the Simulation System ....................... 69
4.2 Simulation overheads for RIOSIM .................................... 70
Chapter 1

Introduction

1.1 Motivation

Parallel processing is the wave of the future in the world of computing. Device technology places an upper bound on the speed of any single processor and technology is approaching fundamental physical limits, including the transmission speed of a signal being limited to be less than the speed of light. In order to achieve more processing power, a number of such processors must be used in parallel. Many important problems requiring huge amounts of computational power can be solved with algorithms that possess a high degree of parallelism. Such problems include three-dimensional fluid-flow calculations, real-time simulations of complex systems, dynamic air-flow calculations, and weather forecasting. Parallel processing promises to provide a way to achieve immense amounts of computing power for many applications. This motivates the development of parallel computers that can take advantage of the parallelism inherent in these applications.

In this thesis, we study a message-passing distributed-memory multicomputer which is built using high-performance node processors. The node processors or nodes are connected to each other using very high-speed communication links. Our objective is to study the performance of a multicomputer which is built using off-the-shelf high-performance workstations. We also describe a simulation system using the execution-driven simulation technique to simulate this multicomputer. The simulation system includes an accurate and efficient profiler for the base processor of the system, which gives time estimates for the execution of blocks of code. The profiler, and hence the
simulation system, allows the users to choose various levels of detailed simulation of the computer. This gives the users a choice between accuracy and performance of the simulation system.

The principal contribution of this research is the determination of the relative effects of changing different parameters of this multicomputer on the performance of the multicomputer. These parameters include the transmission speed of the communication links, overhead in sending messages over the communication link, the interconnection structure of the multicomputer, and the speed of the processor being used in the multicomputer. We study these effects for algorithms with quite different granularities of parallelism running on this multicomputer.

1.2 Overview

There are two basic types of parallel processing systems available in the market. They are

- shared-memory systems

- distributed-memory systems

In shared-memory systems, also called tightly-coupled parallel systems, multiple processors are allowed to access memory in a single global address space. The shared global memory may be a single memory system or a physically distributed memory that spans the global address space. Interprocessor communication takes place through variable sharing. In addition, each processor may contain a small local cache. In distributed-memory systems, also called loosely-coupled parallel systems, the memory is both physically and logically distributed among the processors in the form of local memories. Message passing between the processors is responsible for providing interprocessor communication. Some shared-memory systems are
Sequent Balance, KSR machines, IBM RP3 and BBN Butterfly [Hwang84, Trew91]. Examples of distributed-memory systems include the Intel iPSC hypercubes and the Thinking Machines Connection Machine-5 (CM5) [Lewis92, Stone93]. In the literature, shared-memory systems are sometimes called multiprocessor systems, while distributed-memory systems are often called multicomputers [Athas88, Seitz90].

The processors in a parallel computer (both shared-memory and distributed-memory) are connected to each other using an interconnection network. The interconnection schemes can be classified as direct or indirect [Ni93, Abraham89]. In a direct interconnection scheme, processors are connected to each other using point-to-point links. Examples of direct interconnection schemes are the ring, mesh, and binary n-cube architectures. In the indirect interconnection scheme, the interconnection network is a separate entity, with inputs and outputs. The inputs and outputs are processors in the case of message-passing multicomputers, while for the shared-memory multiprocessors, processors and memories are the inputs and outputs. Examples of indirect interconnection schemes are the indirect binary n-cube networks, omega networks, and crossbar switches [Hockney88, Siegel90].

One more criterion for distinguishing between parallel computers is the granularity of the parallelism that can be used effectively by the computer. The granularity of a system is a measure of the size of the units by which work is allocated to processors. Different types of granularities are coarse, fine and medium.

Coarse-grained parallel computers may distribute the work among the processors at the highest level in an algorithm, e.g., at the outer loop of an iterative algorithm or at the level of individual procedures. Coarse-grain parallelism implies a small number of powerful processors. Examples of such computers are the Cray 2 and the Cray XMP. Fine-grain parallelism often implies a large number of small processors. The unit of computational work in fine-grained parallel computers may be the execution
Figure 1.1: A shared-memory parallel computer

Figure 1.2: A distributed-memory parallel computer
of one statement or an iteration of an innermost loop. Examples of such systems are Thinking Machines' Connection Machine-1, and the Massively Parallel Processor by Goodyear. Medium-grain parallel computers fall between the two extremes. Most commercial parallel processing systems can be considered medium-grained parallel computers [DeCegama89]. These computers typically use inexpensive but powerful microprocessors, such as Intel's 80386 and i860, Motorola's MC68000, SPARC, MIPS, etc. Examples of such systems are the Encore Multimax, BBN Butterfly, Sequent Symmetry, Intel iPSC/2 and iPSC/860 hypercubes, and Connection Machine-5 multiprocessors.

In this dissertation, we describe a medium-grained message-passing multicomputer based on the IBM RISC System/6000 (RS/6000) processors. These processors are interconnected via point-to-point links using the fiber-optic ports provided on the RS/6000 workstations. A message-passing system provides the necessary message-passing primitives required for message handling, message packetizing, and routing of packets. This message-passing system runs as a light-weight thread under a light-weight threads package running on top of the base Unix-like operating system. User processes also run as light-weight threads under this package and can communicate with each other by sending and receiving typed messages. The message-handling system supports asynchronous blocking and non-blocking message passing calls.

The reasons for choosing the RS/6000 computer as a basic building block for the multicomputer are its exceptionally high computing power [Bell91, IBM90b, Hall91, Oehler91], relatively low cost, and very high bandwidth interprocessor fiber optic links. The RS/6000 processor is a superscalar processor having a separate fixed-point and floating-point processing units. It also has separate instruction and data cache units. The fiber-optic links provide a mechanism for transmitting data at bit rates which are an order of magnitude faster than conventional wire networks. With
the exceptional floating-point performance of the RS/6000 processor, we expect the combination of a high-performance processor and a high-speed communication link to be good for parallel algorithms solving numerically intensive problems.

We evaluated the performance of these systems using an extension of the Rice Parallel Processing Testbed (RPPT), a set of software tools for the simulation of parallel architectures [Covington91]. The RPPT uses the execution of real parallel programs to provide execution time estimates for the programs on the simulated architecture. Since the programs are directly executed on the processor, the execution-driven simulation has relatively low simulation overhead compared to instruction-level simulation. For the execution-driven simulation of a multicomputer using RS/6000s, an accurate and efficient timing profiler has been developed for the superscalar processor architecture of the RS/6000. The profiler shows an error of less than 5% of measured time, with a profiling overhead of the order of 150. The simulation model was validated by implementing a simple two-processor multicomputer. We then used the simulation system to project the performance of similar systems with more than two processors. The simulation errors are typically around 8%, while the simulation overheads are approximately 300.

Simulations were run on different numerical algorithms (sorting, graph problems, simultaneous linear equations solver, etc.) The simulation time was compared by running these algorithms on different number of simulated processors. Also, simulations were run changing different parameters of the multicomputer system, and similar comparisons were made. Our simulation studies indicate that a large amount of time is spent in sending and receiving the messages on the fiber-optic links. This results in less than linear speedup for all the algorithms we executed. Some coarse-grained algorithms, like successive over-relaxation, show close to linear speedup while the fine-grained algorithms do not show any speedup or negative speedup because of
the high overhead in message-passing. Simulations in which the overhead is reduced by a factor of four project performance improvements of up to 45.4%. We have also studied the effects of changing certain other parameters of the system, like the maximum packet size, the speed of the fiber-optic links, the interconnection structure of the processors, etc. Changes in these parameters show little benefits, because the high message-passing overhead overshadows the effects of these changes.

1.3 Related Work

Various message-passing systems, supporting different message-passing protocols, have been developed for different platforms. In the following section, we present an overview of several message-passing systems. Efficient simulation of parallel computers is an important research topic and lot of research has been done on this subject. In Section 1.3.2, we look at different simulation systems that have been used to simulate and evaluate parallel computers. A timing profiler is an important component in an execution-driven simulation system. Previous work done in the field of profilers is described in Section 1.3.3.

1.3.1 Message-passing systems

Two types of message passing protocols have been developed: synchronous and asynchronous. In synchronous message-passing systems, if a process sends a message, the process is blocked until the receiving process has received the message. A process trying to receive a message is blocked if no suitable message has arrived at the process' node. If a process is allowed to continue immediately after sending a message, without waiting for the destination process to receive the message, it is called an asynchronous
send. An asynchronous receive call returns to the calling process immediately, along with an indication of whether or not a suitable message is available.

The V-system [Berglund86, Cheriton88, Cheriton83] is an operating system for a distributed system with processors connected by an Ethernet link. It supports a more restrictive form of synchronous message-passing primitives than that described above. A process making a send message call is unblocked only after the receiving process has received and replied to the message. A process can also forward a message, in which case the next receiving process should reply to the message.

The Intel iPSC/2 and iPSC/860 hypercubes' operating system NX/2 [Intel91] supports asynchronous message send and receive primitives. It also allows interrupt-driven send and receive calls. The send routine returns to a user provided interrupt handling routine when the send is complete. Users can also provide an interrupt handler routine for receiving messages. The message-handling system allows typed messages.

The Express kernel [Flower90, Par90a, Par90b] is a message-based multitasking system. Express lets users make typed asynchronous blocking and non-blocking message send and receive calls. It also provides some "loosely synchronous" calls. A node making a loosely synchronous call waits for all other nodes to make the same system call. Thus, all nodes are synchronized after the call completes.

PICL, a Portable Instrumented Communication Library [Geist90], developed at Oak Ridge National Laboratories, is a library of message-passing primitives available on many different machines. It permits some low- and high-level asynchronous message-passing primitives. Only blocking send and receive are available in this library. PICL is based on the low-level primitives provided on each platform for its implementation. Only one process per node is allowed in PICL.
The Cosmic Environment/Reactive Kernel (CE/RK) [Seitz90] is a message-passing operating system originally developed at CalTech. It allows the users to send and receive blocking and non-blocking messages. It does not support typed messages and also cannot distinguish among messages arriving from different nodes. This operating system is used in Ametek 2010 hypercube systems.

1.3.2 Simulation systems

Different methodologies are used for the simulation of parallel computer systems. These include instruction-level simulation, trace-driven simulation, and execution-driven simulation. To decide which methodology to use for simulating a computer system, two important factors need be considered: efficiency and accuracy. Each simulation methodology is characterized by a different tradeoff between these factors. For example, execution-driven simulation is more efficient than instruction-level simulation but it is less accurate.

An earlier version of the RPPT was used to simulate and analyze different parallel computer systems. Covington [Covington88a] developed the RPPT methodology and validated his simulation results against a 16-processor Intel iPSC/1 hypercube. Debbad [Debbad89] added interrupt handling and timeslicing capabilities to the RPPT. He used the RPPT to simulate the V operating system running on Sun-3 workstations connected using an Ethernet link.

Simon (SIMulator of Multicomputer Networks) [Fujimoto83, Fujimoto87a] is an execution-driven simulation system. Simon is able to simulate message-passing multicomputers and provides different modules to describe the architecture of the multicomputer. It supports simulation of multicast and broadcast of messages.

Mpsim [Dunigan86, Dunigan87] is a hypercube simulator developed at Oak Ridge National Laboratories. A limitation of this system is that it does not use light-weight
threads for simulating user processes; instead, it uses Unix processes. Due to this, only a small number of user processes, no more than 16, can be simulated, thus limiting the size of the simulated multicomputer. Mpsim can simulate the Intel iPSC/1 and iPSC/2 systems, and produces trace files for message events for these computers. These traces can then be analyzed using different trace analyzers.

*PSIMUL* [So87] is a distributed multiprocessor simulator running on a shared-memory multiprocessor (an IBM 3081). PSIMUL can provide performance statistics of executed instructions and memory accesses of a parallel program execution. Also, it can produce traces of instructions and data memory references.

Stunkel *et al.* [Stunkel89] use the execution-driven simulation technique to produce address traces for multicomputers. These address traces are used in performance analysis of the simulated multicomputer. They also use a profiler similar to the basic block profiler described below.

*Tango* [Davis90] is a multiprocessor simulation system developed at Stanford University that can be used as a trace-generating or execution-driven simulator. It also uses heavy-weight Unix processes to simulate user processes, and is limited to simulating a small multicomputer.

*Proteus* [Brewer91] is an execution-driven simulation system developed at MIT. It can be used to simulate shared-memory or distributed-memory multiprocessor systems. A basic-block profiling tool is used to provide an estimate of machine cycles required to execute a code segment. The profiler does not simulate the data cache and assumes uniform cache hit rates. Proteus, like the RPPT, uses light-weight threads to simulate user processes.

The *Wisconsin Wind Tunnel (WWT)* [Reinhardt92] describes an execution-driven distributed discrete-event simulator which can simulate shared-memory multiprocessors. WWT runs on a message-passing multicomputer CM-5. It directly executes
the user program on the node processors till a cache miss occurs, at which point WWT simulates a cache-coherent protocol. WWT has been validated against results by Tango/Dixie simulator for the simulation of the Stanford DASH shared-memory multiprocessor [Lenoski92].

1.3.3 Profilers

In execution-driven simulation systems, a timing profiler is used to acquire timing estimates for the execution of blocks of code. These estimates are then used to increment the simulation time, to account for time spent in the execution of those blocks of code.

Weinberger [Weinberger84] first implemented the basic block profiling technique of analyzing basic blocks in an assembly program to obtain dynamic instruction counts. The RPPT profilers use a similar technique to determine the execution time for a basic block. In a basic block profiler, a global counter is used to accumulate the total cycles spent in executing all the basic blocks to get an estimate of the execution time of the program. Basic block profilers for a variety of machines have been developed for the RPPT [Covington88a]. These include profilers for the Motorola 68020, TI 320C25 DSP, Intel 80386 and i860, Vax, Pyramid and SPARC processors.

Fujimoto et al. [Fujimoto87b] developed a basic block profiler to be used in SIMON, an execution-driven simulation system. They also developed a technique for simulating systems where the host and the target processors have different instruction sets. BKGEN [Huguet87] uses another technique for simulating a target processor which has a different instruction set than the host processor.

Stephens et al. [Stephens91] use instruction level profiling technique for the IBM RS/6000 processor. They collect statistics for dynamic instruction mix, branching behavior and resource utilization for the processor.
For this project, a timing profiler has been developed for the IBM RS/6000 processor. The profiler employs an extension of the basic block profiling technique used in the previous RPPT profilers. The new technique, called the \textit{in situ} technique, was originally developed by Carson [Carson91]. In this technique, a runtime model of the superscalar RS/6000 processor is used to generate dynamically timing estimates for the execution of a block of code. Previous research for profiling a superscalar processor for obtaining such timing estimates does not exist.

1.4 Organization of the Thesis

This thesis is organized as follows. Chapter 2 describes the design of the profiler developed for the RS/6000 processor. It discusses some features of the RS/6000 processor which were most important in influencing the development of the profiler. Chapter 3 presents an overview of the fiber-optic links present on RS/6000 machines. It also describes the model of the AIX fiber-optic links driver used in our simulation system. The multicomputer and the message-handling system developed for this computer are also explained in this chapter. The execution-driven simulation system used to simulate the multicomputer is presented in Chapter 4. This chapter also presents the validation results for the simulation system. Chapter 5 presents the performance predictions for larger multicomputers with a similar setup. Finally, some conclusions and avenues for some future work on these systems are presented in Chapter 6.
Chapter 2

Design of the RISC System/6000 profiler

An important component of the RPPT execution-driven simulation system is a family of timing profilers. The profilers are a necessary tool for implementing execution-driven simulations, since they generate the processor workload to be used in simulations.

A profiler produces an equivalent executable program which, when executed, gives an estimate of the number of cycles that would be required to execute each part of the unprofiled program on a particular machine. These estimates are used to drive the architecture simulator in the RPPT (see Chapter 4).

A parallel program for the RPPT is written in C using the function calls provided in the YACSIM (Yet Another CSIM) library [Jump93, Rizvi93]. This program is then profiled by a tprof profiler, which generates execution-time estimates for the program, thus making the program suitable for combination with the corresponding architectural model in the complete simulation.

Two styles of profiling are used in the RPPT. Standalone profiling derives its timing estimates from information that is specific to the processor on which the profiling takes place. The profiled code is then executed directly on this same processor. In certain circumstances, however, it is desirable to execute the profiled program on one processor (the host), yet allow the timing information be derived from a second processor (the target). This scheme, called cross profiling, is useful when program execution on the second processor may be expensive or inconvenient.
The performance of a profiler is measured by its ability to produce correct timing estimates for the execution of a block of code, and to keep the overhead of profiling low.

This chapter describes the \textit{in situ} profiler for the RS/6000 machines. \textit{In situ} profiling is a new technique which uses a simplified runtime model of the RS/6000 processor to obtain dynamic timing estimates. This technique was developed to simulate the superscalar nature of the processor. The profiler is used in RPPT simulations of a message-passing multicomputer consisting of RS/6000 processors connected by high speed fiber-optic links. The remainder of this chapter is organized as follows. In Section 2.1, a typical standalone profiler is described. Section 2.2 describes the main features of the RS/6000 architecture which were considered in writing the profiler. Section 2.3 describes the profiler for the RS/6000, and in particular the special considerations required for the RS/6000 profiler that differentiate it from previous profilers. It also describes the runtime model of the processor as used in the profiler. Section 2.4 discusses the performance of the profiler with regard to its accuracy and efficiency. In Section 2.5, we discuss some potential sources of error in the timing estimates produced by the profiler.

2.1 Standalone Profiling

Standalone profiling, or native profiling as it is sometimes called, is a relatively low overhead method for obtaining detailed information on a program's execution behavior. One of the first implementations of this technique was the \textit{bb} (basic block) profiler [Weinberger84] developed at Bell Labs. However, while the \textit{bb} profiler is primarily used to obtain dynamic instruction counts, the tprof profilers in the RPPT are used to generate estimates of the execution time of a program.
The `bb` profiler and the `tprof` profilers do the profiling analysis based on the basic blocks of code in the program. Basic blocks are straight-line segments of code with one entry point and one exit point. Once a basic block is entered, all the instructions in the block are executed in series before execution passes to another basic block. A basic block therefore begins with

- the first instruction in a file, or
- a labeled instruction, or
- an instruction immediately following a branch, except in the case of delayed branches.

Basic blocks end with

- the last instruction in a file, or
- an instruction immediately before a label, or
- a branch.

The typical steps in standalone profiling are shown in Figure 2.1. The profiler first identifies the basic blocks in the assembly language program. Each basic block is then analyzed to estimate the number of cycles required for its execution. After this, a few profiling instructions are inserted at the top of each basic block, which increment a global counter by the cycle count of the basic block. These steps are described in detail in the following section.

2.1.1 Generating profiled programs

The generation of a profiled program for a typical (non-superscalar) processor involves a number of steps, which are listed below.
Figure 2.1: Block diagram for standalone profiling
Figure 2.2: Native profiling on a basic block
Generation of assembly language program: An assembly language program is obtained from the C program using appropriate compiler options.

Basic block identification: The basic blocks are identified in the assembly code according to the rules described above.

Timing analysis: Each basic block is analyzed to determine an estimate of its execution time. This essentially requires a table lookup using the opcode of each instruction, taking into consideration the specific addressing modes which may be used in the instructions.

Insertion of profiling instructions: Instructions are inserted in each basic block of the assembly language code, to increment the global counter by the amount required for the execution of the (unprofiled) basic block, as determined in the previous step.

Compilation of the instrumented program: The instrumented program is assembled to get the executable program which will drive the simulation.

Figure 2.2 illustrates the difference between an unprofiled and a profiled program. The profiled program executes all the instructions in the original program and also gives an estimate of the time required to run the program on the machine. Instructions $P1$ and $P2$ in Figure 2.2 increment the global counter by the number of cycles required to execute the basic block $A$, as estimated by the profiler. Thus, after all the basic blocks have been executed (i.e., at the end of the program execution), the global counter variable will contain the aggregate of the cycles required for the execution of the whole program. This is the profiler estimate of the execution time for the particular program. Its accuracy can be checked against the measured real time of the execution of the unprofiled version of the program.
The superscalar nature of the RS/6000, the presence of a data cache, and certain features of the RS/6000 AIX compilers required this simple approach to be modified substantially. The following section describes the RS/6000 processor architecture emphasizing the features which force these major modifications in the original approach for native profiling.

2.2 RISC System 6000 Processor Architecture

The RISC System 6000 (RS/6000) processor is a superscalar, second-generation reduced instruction set computer (RISC) processor that has a short cycle time and a low cycles-per-instruction (CPI) ratio [Hennessy90, Johnson91]. In keeping with the RISC philosophy, it has a relatively simple register-oriented instruction set, a hard-wired CPU control unit, and a pipelined implementation. Unlike earlier RISC processors, it has several advanced features, including multiple instruction dispatch, simultaneous execution of fixed- and floating-point instructions, separate instruction and data caches, and zero cycle branches.

The RS/6000 processor also has a tightly-coupled floating point unit, which features a fused multiply-and-add instruction executed in two cycles, in a two-stage floating-point pipeline, which means it can produce a floating-point multiply-and-add result every cycle. Thus, the processor is capable of sustaining a peak rate of 50 MFLOPS at 25 MHz. A block diagram of the RS/6000 processor is shown in Figure 2.3.

The major components of the RS/6000 processor, and their influence on the design of the profiler are described in detail in the following sections.
Figure 2.3: Block diagram of the RISC System/6000 processor
2.2.1 Instruction Cache Unit

The instruction cache unit (ICU) contains a two-way set-associative, 8-Kbyte instruction cache with a line size of 64 bytes [Grohoski90b]. It also has the I-cache directories and a 32-entry, two-way, set-associative I-TLB. The ICU processes branches, condition register instructions, and supervisor calls, and dispatches the rest of the instructions to the fixed- and floating-point units.

Four instructions per cycle can be fetched from the I-cache arrays to the instruction buffers and dispatch unit, which can dispatch up to four instructions per cycle. Two of these are internal dispatches to the ICU (branches and condition-register instructions) and two are external dispatches to the FXU and FPU. FXU and FPU never see any branches, and in most cases they receive an uninterrupted instruction stream and do not see the effect of the branches. This is referred to as zero-cycle branches. Unconditional branches cause no delay in the pipeline. Conditional branches that fall-through also have no penalty, while taken branches may delay the pipeline by up to three cycles, depending on when the condition register was set.

2.2.2 Data Cache Unit

The RS/6000 has a four-way set-associative 64-Kbyte data cache (D-cache) divided into four identical DCU chips of 16 KB each. The cache-line size is 128 bytes and the cache is implemented as a write-back cache\(^1\). The DCU also features a cache reload buffer (CRB) and store-back buffers (SBB).

The 128-byte CRB allows the CPU to start processing as soon as the required memory datum is received into the buffers, without waiting for the whole cache-line to be brought into the cache. The cache line is loaded into the CRB in eight cycles,

\(^1\)The write-back caching protocol is also called the store-back protocol
but the first packet from the memory cards contains the datum which caused the cache miss. This allows the system to have a low cache miss penalty.

Store-back buffers are also 128 bytes long. They can accept data from the D-cache or the CRB and pass it on to main memory. An SBB improves the performance because the dirty line does not have to be written back to the memory before the new line is brought into the cache. In addition, the data-cache arrays are not kept busy during the whole write-back sequence.

2.2.3 Fixed Point Unit

The fixed-point unit of the RS/6000 processor has 32 32-bit general-purpose registers. It handles the address generation and DCU controls for both fixed- and floating-point loads and stores. The FXU also features a fixed-point multiply and divide unit: the multiply instruction takes 3 to 5 cycles, and the divide instruction takes 19 to 20 cycles. The FXU has a one-word data path from the DCU.

2.2.4 Floating Point Unit

The floating-point unit has 32 64-bit general-purpose registers. The FPU fully conforms to the IEEE 754 floating-point standard. The FPU has a double-precision-wide data path and executes all floating-point instructions with double precision only. The FPU can generate one double-precision result per cycle, and has a pipeline latency of two cycles [Montoye90]. The FPU has a double-word data path to the DCU.

The FPU can perform a multiply-add operation with the same delay as a multiply or add instruction. It also has four instruction decode buffers (IDB) which allow the FXU to get ahead of the FPU. Also, the FPU uses register-renaming to increase the overlap of the execution of floating- and fixed-point units. The FPU has six rename registers.
2.2.5 Synchronization between FXU and FPU

The scheme used for synchronizing the FXU and the FPU is explained using Figure 2.4. Both units have six instruction-prefetch buffers (IPBs), which allow the branch processing unit to get ahead of the FXU and the FPU. IPBs on both sides keep the same instructions at all times. In the FXU, there are two instruction decode registers, D0 and D1. One of these registers feeds the execute stage of the FXU. On the FPU side, there are two pre-decode registers, PD0 and PD1, which keep a mirror image of D0 and D1, respectively. PD0 and PD1 feed the rename registers, R0 and R1.

Instructions are fetched four at a time from the ICU. If these include appropriate instructions for both the FXU and the FPU, an instruction to each is issued in the same clock cycle. Both the issued instructions are kept until the rename stage because one of the instructions can be a fixed-point arithmetic operation while the other may be a floating-point load or store which is considered a fixed-point operation but needs renaming of registers. After the rename stage, the floating-point instructions go through the decode stage and the two execution cycles FP1 and FP2.

Four instruction-decode buffers (IDB0 - IDB3) are present in the FPU, which can buffer floating-point instructions if the floating-point pipeline is busy. Thus the FXU does not need to wait for the FPU to finish executing the next instruction(s) if there are no dependencies. A counter present in the FXU is used to synchronize the FXU and the FPU by keeping track of the relative positions of an instruction in the FXU and the FPU. If the FXU is working on an instruction, the permissible values of the counter allow the FPU to look down up to six subsequent instructions for renaming. Conversely, if the FPU is busy and the IDBs become full, the FXU is allowed to execute up to two subsequent instructions.
Figure 2.4: FXU-FPU synchronization scheme for the RS/6000
Note that the FPU pipeline contains seven additional stages after the rename stage. This means that the FPU can actually buffer up to seven issued floating-point instructions, while the FXU can be executing fixed-point instructions. The ability to model this overlap between the FXU and FPU is one of the most significant innovations in the RS/6000 profiler compared to profilers for other processors.

For more information on the RS/6000 processor architecture, see [Bakoglu90a], [Oehler90], [Grohoski90a].

2.3 Profiler for the RISC System 6000

The profiler for the RS/6000 uses a new technique called in situ profiling. This technique is an extension of the native profiling technique outlined in Section 2.1. In this technique, all of the profiling is done statically, but part of the timing estimation is done dynamically. This gives the profiler the ability to take into account the superscalar nature of the processor, as well as the overlap between the functional units and the D-cache.

A discussion of the problems with native profiling, and a detailed description of the new technique is given in the following sections.

2.3.1 Problems in native profiling on the RS/6000 processor

Several problems were encountered in developing a native profiler for the RS/6000 processor. The most critical of these are discussed below.

- The assembly language listing file produced by the C/FORTRAN compiler cannot be assembled. As shown in Figure 2.2, this capability is necessary for instrumenting the assembly code produced by the high-level language compiler.
• For source-level profiling of the code, there must be a close correspondence between basic blocks at the source code level and basic blocks at the assembly language level.

The AIX compilers discard the labels and comments in C source code, when generating equivalent assembly code. Use of dummy procedure calls as markers may actually cause the compilers to produce different code. Thus there are no source-code markers to establish the correspondence between the basic blocks in assembly code and the source code, making source-level profiling impossible.

• As described in Section 2.2, the RS/6000 has a superscalar architecture, with multiple operational units. Instruction executions overlap in ways which cannot be predicted at compile time. This makes the job of estimating the execution time for a basic block more difficult.

• The presence of a data cache on the RS/6000 also poses problems for standalone profiling of the processor. A data cache miss causes a maximum delay penalty of 12 cycles. This penalty may be covered in a number of ways due to the presence of other instructions around the instruction that caused the cache miss. Due to this high and unpredictable cache miss penalty, it is difficult to accurately predict the timing estimates for the execution of a program using static timing analysis methods.

2.3.2 Design of the RS/6000 profiler

Because of the problems listed above, a profiler for the RS/6000 could not be developed using the standard native profiling techniques. Instead, the in situ profiler attempts to take into account the superscalar nature of the processor in order to produce accurate estimates of the execution times for a program.
Figure 2.5 illustrates in situ profiling. First a static estimate for the execution time of the instructions in a basic block is determined. Each instruction requires some work from each processor unit. A list of this work and the processor unit required to do the work is passed to the handler for each basic block. A handler is a segment of code which invokes the runtime model of the simulated processor. The runtime model simulates the overlap of the different functional units of the processor and determines the real time required to execute the basic block at runtime. The first instruction of a basic block is replaced by a call to a handler for that basic block. After invoking the model, a typical handler executes the original instruction that was displaced from the basic block by the call to the handler, and then returns control to the next instruction in the basic block.

The runtime model simulates the fixed- and floating-point units of the processor, and also charges the delays incurred due to conditional branches. The user decides whether or not to model the data cache unit and the TLB, depending on the profiling overhead (see Section 2.4) acceptable to the user. Using separate handlers for each basic block and modeling the different processor units at runtime allows the profiler to estimate the total time spent in executing the basic blocks.

A major goal of the profiling technique used for this profiler is that we should be able to get a fairly accurate prediction for the execution time of a program, while keeping the time spent in doing so relatively small. To achieve this goal, scenarios which cause stalls in a pipeline but which are not expected to occur frequently are ignored by the profiler. Also, cases which would require substantially greater overheads to produce what we felt would be relatively small improvements in accuracy have also been neglected.

The different steps for generating a profiled program are described below.
Figure 2.5: In situ profiling for the RS/6000
2.3.2.1 Determining basic blocks

To determine the execution times for each instruction in a basic block, we need to have an equivalent assembly program. As mentioned above, the assembly code produced by the AIX C/Fortran compilers can not be assembled. To get around this problem, we use a disassembler which takes the listing file produced by the AIX compilers and uses the numerical opcodes in that file to create a list of the instructions in the program [IBM90a]. Since the RS/6000 is a RISC processor with fixed-length instructions, it is relatively easy to do a table lookup on the opcodes to determine the instructions.

An awk script extracts the opcodes from the listing file and determines the basic blocks. This script produces a file containing the opcodes of the instructions, and other files containing information about where the basic blocks start in the executable file.

2.3.2.2 Static timing analysis of a basic block

After determining the basic blocks in the program, the profiler determines partial timing estimates for each basic block. It performs a table lookup for each instruction to get the basic execution time for the instruction. The fixed- and floating-point execution times are recorded for each instruction according to the rules described below.

Most of the fixed-point instructions take one cycle of the FXU. Exceptions are described below.

1. Multiply (mul) may take 3 – 5 cycles, depending on the operands.

2. Divide (div) takes 19 cycles.
3. Load multiple (lm) and store multiple (stm) take \( n \) cycles if \( n \) registers are referenced.

4. The load string (ls) group of instructions take \( n \) cycles where \( n \) fullwords are referenced.

Most of the floating-point instructions take two cycles in the FPU, but the FPU has two arithmetic pipeline stages, allowing instructions to be executed at a rate of one instruction per cycle. For modeling the FPU instructions, we assign a basic delay of one cycle to each floating-point instruction. A few exceptions are listed below.

1. Divide (fd) is assigned a delay of 19 cycles.

2. Move to FP status and control register (mrtfsf) causes a three-cycle delay.

3. Store float (stf) instructions take one cycle of decode in the floating-point unit.

For the FPU, all the delays listed above are valid for normal numbers. Denormalized, infinite, or NaN numbers cause further delay which have not been taken into account.

The next step is to assign the delays in the FPU pipeline caused by data hazards, i.e., when an instruction uses a register written by the previous instruction. This causes a delay cycle. An extra delay of two cycles occurs if the previous instruction writes the register referred to first in the current instruction.

After this step, fixed-point register use delays are assigned for each relevant instruction, similar to the delays assigned for the floating-point unit.

Next, branch delays for conditional branches are calculated based on when in the instruction stream the condition register was set. The branches are checked at runtime and the delays are put in the pipeline if the branch was taken. Unconditional branches (jumps and subroutine calls) do not cause any delay.
For a detailed discussion of the execution times of the different instructions, and the delays occurring between instructions due to data dependencies, structural hazards or branches, see [Warren90, Warren91].

2.3.2.3 Runtime timing estimation

As described in Section 2.3.2, the handlers for each basic block call the runtime model of the processor. The runtime model simulates the different units of the processor by keeping track of the work required to be done by each unit. Each profiled block in the program requires a certain amount of work from each unit (fixed-point, floating-point, and data cache). The runtime model puts the static cost of the basic block in the variables representing the different pipelines. This work is retired in a parallel fashion; i.e., one unit of work can be removed from each of the three functional units on each machine cycle if it has something to do.

A number of floating-point instructions take more than two cycles to execute, while most of the fixed-point instructions take one cycle for execution. Therefore, it is more likely that the FPU lags behind the FXU in instruction execution than the other way around. The FXU pipeline is therefore modeled as a single variable, while the FPU is modeled instruction by instruction. This allows the profiler to determine when the FPU buffers become full and slow down FXU processing.

As described in Section 2.2.5, the FPU can hold up to eight issued instructions past the predecode stage. We model the FPU as an eight-deep circular queue of instructions. If the instruction buffers become full, the ICU will not send any more instructions to either unit, until some work is retired from the FPU, and FXU if there is any work in its pipeline.

We decided not to model the FXU instruction by instruction so as to keep the time overhead for profiling low. A limitation of this model is that if the FXU becomes
the bottleneck in some application, it will not be noticed and the profiler would give an optimistic prediction for execution time. For example, if there is a long fixed-point instruction like `div` executing in FXU, followed by six fixed-point instructions and then a floating-point instruction, the RS/6000 processor would finish the `div` instruction, while the FPU may be idle. The runtime model in the profiler though would execute the floating-point instruction along with the fixed-point instructions. Also, a consequence of our approach to modeling the FPU pipeline is that if there are eight consecutive floating-point instructions followed by a few fixed-point instructions, the latter would be completely covered by the former in the profiler. In reality, only one instruction can be issued to each of the two processing units in a single cycle, so only the last fixed-point instruction can be issued with the last floating-point instruction.

The error due to inaccurately simulating the overlap of the FXU and the FPU becomes negligible if the instructions are adequately mixed, i.e., the fixed-point and floating-point instructions are closely balanced. However, if such a balance does not exist in a segment of code, and there is a streak of either fixed- or floating-point instructions followed by a few instructions of the other type, the profiler would give a wrong (low) estimate of the execution time of that code segment. Our experience shows that the AIX compilers do a good job of mixing instructions of the two types and we do not expect this situation to occur often. The profiler and the runtime model can be modified, at the cost of additional runtime overhead, to account for poorly mixed FXU and FPU instructions more accurately, if this problem should become more critical.

The runtime model also simulates the data cache and the translation lookaside buffer (TLB) of the RS/6000.
The cache model simulates the 64 KB 4-way set-associative cache with least-recently-used (LRU) replacement scheme by keeping information about the line-frame addresses (tags) of the cache. Each memory address, generated at runtime by the handler, is compared against the four line-frame addresses of the set to determine if that memory access would be a cache hit or miss.

For fixed-point loads, a statically determined penalty is charged on a cache miss. For the floating-point loads, there is a statically determined penalty, but the model is checked at runtime to see if the FXU is far enough ahead of the FPU to cover it. For a cache miss on stores, a fixed penalty of 8 cycles is charged to the DCU. We do not simulate the operation of the store-back buffer on the DCU; it is assumed that every displaced line from the cache is written back to memory without any delay cycles.

The TLB is also modeled as a two-way set-associative LRU cache with 128 total entries. A miss on the TLB causes a penalty of 37 – 72 cycles. The TLB miss penalty varies by a large amount due to several reasons [IBM90b], [Bell91]. We used a fixed cost of 28 cycles for a TLB miss, since it would be time-consuming to model all of the sources of variation in TLB miss delays and because TLB misses are typically rare. TLB modeling may be turned off by the user and usually should be disabled, unless the memory accesses cover a very large range.

Taken conditional branches cause variable delay cycles. These delay cycles depend on the distance of the branch instruction from the condition-register-setting instruction, and are determined by static analysis. The runtime model checks at runtime whether the branch was taken or not. If a branch was taken, it charges the delay cycles for the branch. The runtime model determines whether a branch was taken or not by checking the order of the handlers being executed (each basic block has a separate handler, and a branch instruction separates basic blocks).
Several special cases, which provide for some unexpected delays that were discovered during the validation process, have also been added to the runtime model. For example, if a floating-point operation uses a register which was loaded by the previous instruction, the FXU pipeline is flushed before this FPU instruction can go ahead. If the last instruction in a subroutine is a floating-point instruction, it is assumed that the result of this instruction would be used in the next instruction, and therefore one cycle is added to its floating-point cost.

We do not model the instruction cache, but assume that the instruction cache always has the required instructions. The presence of the cache scanning and instruction prefetch logic make the I-cache extremely difficult to model with anything short of a cycle-level simulator, and at the same time make it highly probable that instructions are already in the I-cache when they are needed. We also do not model any virtual memory effects like page faults. Again, it would involve substantial additional overhead, with a relatively small payoff in increased accuracy of timing estimate, to model the virtual memory system. Also, the emphasis here is a high-performance parallel process which is achieved only when there is sufficient memory to hold the entire program including data.

2.4 Performance

There are two important performance criteria for a profiler.

Accuracy: Accuracy is a measure of how the time estimated by the profiler for a program compares to the measured time required to execute that program.

\[
\text{Accuracy} = \frac{\text{estimated time} - \text{measured time}}{\text{measured time}}
\] (2.1)
Profiling overhead: Profiling overhead means the extra amount of time it takes to run the profiled program. For a sequential program, it is defined as

\[
\text{Overhead factor} = \frac{\text{time to execute profiled code}}{\text{time to execute unprofiled code}}
\]  

(2.2)

The performance of the profiler for different benchmarks and small test programs is shown in Table 2.1. Because of the runtime modeling of the processor, we have a relatively high profiling overhead for the RS/6000 processor compared to overheads for other profilers [Covington88a]. The overhead ranges from 50 to 180 for both fixed- and floating-point code. The estimates produced by the profiler are quite accurate and are typically within 5 percent of the real execution time.

<table>
<thead>
<tr>
<th>Program</th>
<th>Accuracy (Percent error)</th>
<th>Profiling overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>-4.8</td>
<td>118</td>
</tr>
<tr>
<td>fft2</td>
<td>-4.6</td>
<td>59</td>
</tr>
<tr>
<td>mgsort</td>
<td>-5.3</td>
<td>171</td>
</tr>
<tr>
<td>qsort</td>
<td>-5.5</td>
<td>178</td>
</tr>
<tr>
<td>sor</td>
<td>-2.0</td>
<td>160</td>
</tr>
<tr>
<td>int</td>
<td>-2.1</td>
<td>346</td>
</tr>
<tr>
<td>float</td>
<td>0.0</td>
<td>71</td>
</tr>
<tr>
<td>mixed</td>
<td>-3.2</td>
<td>52</td>
</tr>
<tr>
<td>tlb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(tlb modeled)</td>
<td>+16.0</td>
<td>65</td>
</tr>
<tr>
<td>(not modeled)</td>
<td>-10.7</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 2.1: Performance of the RS/6000 profiler

The fft program is an implementation of the basic radix-2 Cooley-Tuckey algorithm for calculating the Fast Fourier Transforms (FFT) and is primarily floating-point code. The fft2 program is another implementation of the FFT algorithm, and consists mostly of integer code. Mgsort and qsort are basic mergesort and quicksort algorithms, and are also primarily integer code. Sor is a sequential implementation
of a successive over-relaxation algorithm and consists mostly of floating-point calculations.

The programs int, float and mixed are not real applications, but are designed to test the profiler's predictions for fixed-point, floating-point and mixed code, respectively. Each consists of a single loop containing arithmetic operations of the indicated type(s).

The int program is written such that it causes a cache miss on every cycle in the loop. The profiling overhead is high for this program since the runtime model has to perform the LRU replacement algorithm on the lines in the set to which the missing memory address belongs on every cycle in the loop.

The profiling overheads for most of the programs is around 150. For fft, mixed, and float programs, the overhead is around 60. The data set used for these examples was smaller than the D-cache size, resulting in few cache misses, and hence the low profiling overhead.

Tlb is another test program, in which the data is accessed so that it causes a TLB miss every cycle. With the TLB modeled, the profiler gives a high estimate, while if it is not modeled, we get a low estimate. It should be noted again that in practice we expect TLB misses to be rare and these errors will not significantly influence the accuracy of the results. However, if TLB misses are frequent in some application of interest, additional work on this part of the runtime model may be necessary.

2.5 Sources of errors in the profiler

As pointed out in Section 2.3.2, to keep the profiling overhead relatively low, we have opted to neglect a number of scenarios which cause delays but occur infrequently.
Only those delays have been incorporated which we believe occur often in real programs. Some sources of errors are discussed below.

- The profiler considers execution cycles for normal numbers only. Denormalized, infinite, or NaN numbers cause additional cycles delays which are not taken into account.

- Overlapping between floating-point unit and fixed-point unit is approximate only. This was discussed in Section 2.3.2.3.

- Contention between the ICU and DCU is optimistic. The runtime model assumes that there is no contention between the ICU and DCU for memory accesses.

- Performance of the ICU is modeled optimistically. It is assumed that the ICU will always have enough instructions without ever stalling for memory accesses and would always be able to provide the FXU and FPU with a constant stream of instructions. Also, the inability of the ICU to deliver four instructions when near the end of a memory page has not been taken into account.

- Data cache modeling is approximate, and does not include all the delays that might occur in the data cache unit.

Some of the sources of error could be minimized or removed by a more sophisticated run-time model, but only at the expense of larger profiling overheads. Validation results appear to indicate that further improvements in accuracy will be small except in what we believe to be extreme cases and hence larger overheads do not seem justified. However, users should be aware of the profiler limitations and sensitive to the possibility of extreme cases occurring in their applications.
Chapter 3

An RS/6000-based Multicomputer

In this project, we have developed a distributed-memory message-passing multicomputer consisting of IBM RS/6000 computers. The machines are connected to each other using the point-to-point fiber-optic links provided on the RS/6000 computers. The RS/6000 computer is well suited as the base processor for this multicomputer because of its high floating-point performance, its high-bandwidth interprocessor links and relatively low cost.

A message-handling system which provides reliable message-delivery has been developed for the multicomputer. The message-handling system allows asynchronous blocking and non-blocking message send and receive calls. Also, the users can send and receive typed messages. A blocking message send call returns to the user after the message has been copied into buffers kept by the message-handling system. A non-blocking send returns to the user immediately, but the user is responsible for not changing the message buffer because it is not known when the message-handling system would copy the message. At the receiving end, a blocking message receive suspends the receiving process until a message of the desired type and sender has arrived. A non-blocking message receive returns with the desired message or an indication that no suitable messages are available. The users can also check to see if a message of an appropriate type and sender is available at a processor.

We describe the setup of the multicomputer in this chapter. Section 3.1 describes the features and characteristics of the fiber-optic links architecture as present on the RS/6000. The multicomputer setup is presented in Section 3.2. It also describes in detail the message handling system developed for this multicomputer.
3.1 An overview of the Fiber-optic links

The 220 Mbps fiber-optic links \(^1\) can be used for attachment of disks and other high-speed peripherals, and they are also suitable for inter-processor message and data transfers in a multiprocessor configuration [Irwin90].

The base operating system (AIX version 3.2) [IBM92] provides the basic facilities to use the fiber-optic communication system through system calls which can read from and write to the fiber optic ports and can also check the status of the ports.

3.1.1 Architectural overview

The protocol AIX-3.2 uses in communicating over the optical links provides error detection and flow control. This protocol incorporates error-detection mechanisms at the byte, frame, and message levels. It also defines a flow control mechanism that permits data to be transferred at a sustained high rate when conditions allow.

An operational environment using the serial fiber-optic channels for connecting several processors is shown in Figure 3.1. Each physical connection consists of separate transmit and receive optical fibers, which comprise a single bidirectional serial link. The protocol allows full-duplex operation to achieve low latency.

The serial I/O architecture uses an 8/10 transmission code. Each 8-bit byte is converted to a 10-bit code for transmission on the optical fiber. Special 10-bit codes which do not match any of the 8-bit codes are used as control codes (e.g., frame delimiters and the idle sequence).

Information is continuously transmitted over the link whenever the link is active. The information transmitted may be frames of data or the idle sequence. The

---

\(^1\)The terms fiber-optic links and serial links are used interchangeably throughout this document.
idle sequence is present to allow the serial link to maintain synchronization between connected nodes.

Each RS/6000 computer (model 9xx) can have two serial link adapters, and each adapter allows two links [Bakoglu90b]. Thus a maximum of four other computers can be connected to an RS/6000 using these serial links.

3.1.2 Characteristics of the serial link

The maximum size of messages that can be sent over the serial links is 64 KB. After receiving any messages, the serial link driver buffers the received messages. Users can get the messages copied into their buffers by using the read system call. The amount of data available can also be checked by making system calls provided by AIX. The amount of buffering by the AIX device driver is limited. The number of messages that are buffered by AIX is finite and decreases with the size of the messages. Any messages arriving on the serial optical link when the buffers are full are discarded by the operating system.
To determine the maximum amount of data the serial link system would buffer for a particular message size, we performed a series of tests on the serial link. Figure 3.2 shows the results of the tests. The horizontal axis shows the number of memory pages sent in each message (one memory page on the RS/6000s is 4096 bytes). The vertical axis shows the number of messages buffered on the receiving side for a given message size. The curve shows an inversely proportional relationship between the message size and the number of buffered messages up to a message size of 16 pages.

This limited buffering of arrived messages forces a constraint on the message-handling system for message-based communications between two or more computers. The message-handling system on these computers must provide a protocol for sending and receiving messages that will buffer user messages and send only a limited number of messages to the other computer. The protocol used in the multicomputer in this study is described in Section 3.2.5.

3.1.3 Message Transmission Delays

The time elapsed in sending a message from one end and receiving it on the other end on the fiber-optic ports consists of the following components.

- The time spent in the operating system to send the message.
- The time spent in the transmission of the message over the fiber-optic link.
- The time spent in the operating system to receive the message.

The time needed for transmission of the message over the link is calculated assuming that the fiber-optic link transmits at its rated capacity of 220 megabits per second. As described in Section 3.1.1, the link uses an 8/10 coding. Thus the actual
Figure 3.2: Maximum number of messages buffered by the serial link
rate of transmission of user data is

\[ 220 \times \frac{8}{10} = 176 \, \text{Mbits/second} \]  

(3.1)

We ran some tests on the links to determine the time required to send messages from one processor to another. The results from these tests are used to calculate the time spent in the operating system for sending and receiving messages of different sizes. This information is required for simulating the message-handling system. We do not have access to the AIX source code, which prevents us from profiling the operating system code controlling the serial links to obtain this timing information.

For the tests, we first established a connection between two processors (call them A and B). Then we sent one message from A to B. B received the message and returned it to A. This process was repeated a number of times in a tight loop and the time spent between the sending of a message and its subsequent return was measured for each iteration. The minimum of the measured times was determined and accepted as the time needed for a round-trip of that message size. Figures 3.3 - 3.7 show the measured times for a few message sizes.

The time required for a round-trip has the following components.

- The time required for the two-way transmission of the message over the serial link.

- The time needed by the operating system to send the message. This overhead is incurred twice, once at each end of the link.

- The time needed by the operating system to receive the message and return it to the receiving process. This is also incurred twice during each round-trip transmission.
Figure 3.3: Time required for a round-trip for different message sizes

Figure 3.4: Time required for a round-trip for message sizes from 225 bytes to 256 bytes
Figure 3.5: Time required for a round-trip for message sizes from 4075 bytes to 4105 bytes

Figure 3.6: Time required for a round-trip for message sizes from 8180 bytes to 8204 bytes
Figure 3.7: Time required for a round-trip for message sizes from 12248 bytes to 12328 bytes

Using the data transmission rate calculated in Equation 3.1, we can calculate the time spent in the movement of data from one end to the other and back. Subtracting the time spent in data transfer from the measured time, we are left with a value that corresponds to the time spent in operating system overhead for sending and receiving of data. For our simulation of the fiber-optic links, we will assume that for a particular data size, equal time is spent in the operating system for sending and receiving of data.

Figures 3.3 - 3.7 also show the difference in timings after the transmission time has been subtracted. These figures reveal that the time spent in the operating system for message handling (operating system overhead) is piecewise linear. The operating system delay is almost constant until a message size of 236 bytes. This number corresponds to the size of an mbuf structure in the AIX operating system code,
which is 256 bytes long with a 20 bytes header. If the message size is smaller than 236 bytes, the \textit{mbuf} structure is used by the OS to send the message on the serial links. If the message is greater than 256 bytes long, the system uses one \textit{mbuf} structure for information on how long the message is, and the message is sent on a per-page basis. Therefore, the rest of the jumps in the operating system overhead are at page boundaries, i.e., at 4096 bytes, 8192 bytes and so on.

Using this observation, we modeled the operating system delay in the handling of messages as a piecewise linear function of the message size. The model is summarized in Table 3.1. This model is plotted in Figure 3.8 along with the measured timings. The plot matches the measured timings very closely. Because of considerations of acknowledging data packets, described in Section 3.2.4, we have developed the model for message sizes up to 12288 bytes.

<table>
<thead>
<tr>
<th>Message Size (x bytes)</th>
<th>Delay Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 236</td>
<td>0.4425</td>
</tr>
<tr>
<td>237 - 4096</td>
<td>0.6335 + 8.9000e-06 x</td>
</tr>
<tr>
<td>4097 - 8192</td>
<td>0.7931 + 1.1038e-05 x</td>
</tr>
<tr>
<td>8193 - 12288</td>
<td>0.9434 + 1.1276e-05 x</td>
</tr>
</tbody>
</table>

Table 3.1: Model for the message handling delays in operating system

3.2 Description of the Multicomputer

3.2.1 Hardware

The target multicomputer used for the performance studies is a network of IBM RS/6000 machines connected by point-to-point high-speed fiber-optic links. All nodes of the multicomputer run a lightweight threads package, on top of the base operating system (AIX). The message handling, message routing and user code run as
Figure 3.8: Plot of the piecewise linear model of the operating system delays, and comparison with measured time difference
lightweight processes under this package. The message handling and routing code
uses operating system calls provided by AIX to read, write and check the status of
the serial links in sending and receiving data on the links.

As described in Section 3.1.1, any computer can be connected to four other com-
puters through its fiber-optic links. This gives us several ways to connect these
computers in different structures. For example, we could construct a system using a
2-dimensional mesh, a 2-dimensional torus, or a hypercube of up to four dimensions.

3.2.2 User programs and system services

The user code is written in C using a subset of the library calls provided in the discrete
event simulator YACSIM [Jump93]. YACSIM provides library routines to create user
processes and events, assign function bodies to the processes and events, and allow
processes to interact with each other using message passing calls.

Messages sent by a user are broken up into fixed-size packets, which are trans-
mitted over the serial links. Multihop routing is provided by the message-handling
code, which relieves the user of doing the same. The user provides routines describing
packet routing in the multicomputer. This gives the user the option to try different
routings to get the best performance on the system. The users are also allowed to set
the packet size up to a certain limit.

The users can send, receive and check for typed messages. The message-passing
calls may be blocking or non-blocking. A blocking message send would not return
until the message has been copied by the message handling system, so after return the
user can change the data in the buffer holding the message data. A non-blocking send,
on the other hand, returns immediately, and the user is responsible for not changing
the message buffer or else the receiver may receive a corrupted message. At the
receiving end, a blocking receive would suspend the receiving process until a message
of the desired type and sender has arrived at the processor to which the receiving
process is attached. A non-blocking receive is provided which returns immediately,
signaling the non-availability of an appropriate message when necessary. The user
can also check to see if a message of an appropriate type and sender is present on the
processor.

A detailed description of the message-routing and message-handling system is
given in the following sections.

3.2.3 Message routing

Automatic routing relieves the user of the responsibility to route messages if it has
to be sent to a destination processor that is not directly connected to the sending
processor. The user must specify the routing function which describes the routing
algorithm used for the multicomputer. This function takes the current processor id
and the final destination processor id and returns the processor id for the next hop.
This way, the user may try different routing algorithms depending on the setup of
the multicomputer and the algorithms he is running.

3.2.4 Acknowledgment of Data Packets

As shown in Figure 3.2, only a limited number of messages are buffered by the op-
erating system at a node and the number of buffered messages is dependent on the
size of the messages. In order to have a reliable message delivery system, which does
not loose any messages, every node can send a certain maximum number of packets
to another node, and every receiving node needs to acknowledge the receiving of a
packet.

From Figure 3.2, we can see that if we choose a packet size of 4096 bytes or less,
150 of these packets can be stored at any processor. As discussed earlier, a maximum
of four processors may be connected to any processor. Thus four processors may be sending messages to any processor at any given time. We limit the number of messages a node can send to another node to avoid losing any messages because of limited buffering. This decision requires our protocol to return acknowledgment messages so that more messages can be sent once the limit is reached, and the previous messages have been received at the other side.

These considerations lead to a limit of \( \left\lfloor \frac{150}{4} \right\rfloor = 37 \) packets from each connected node. Since we need one acknowledgment packet for each data packet, we can only send \( \left\lfloor \frac{37}{2} \right\rfloor = 18 \) packets from one end at a time before we receive an acknowledgment packet back from the destination node.

Using the figure of 18 data packets and 18 acknowledgment packets maximum sent to any particular node from any node, a maximum of 144 packets may accumulate at any time at a node. Since the fiber-optic port driver in AIX can buffer a minimum of 144 packets for packet sizes up to 12288 bytes (3 memory pages), the message-handling system allows the users to set the packet size up to 12288 bytes. The default maximum size of a packet is 4096 bytes.

3.2.5 Description of the message handling system

The message handling system provides facilities for the user to send and receive messages without worrying about the packetization of messages, routing and buffering of packets at intermediate nodes for multihop messages, or re-assembly of messages at the final destination.

The message handling system is implemented by creating a lightweight process under a lightweight threads package. This lightweight thread configures the system by establishing connections with the other nodes of the system. After establishing
a connection, the process continuously checks to see if there are any messages to be sent or received. A flowchart for the message-handling system is shown in Figure 3.9.

The message-handling process first checks to see if there are any packets to be received. If there are any packets available, it reads a packet into its buffer, and checks whether the packet is a regular data packet or an acknowledgement packet. If the received packet is a regular data packet, the process checks to see if this processor is the last destination for this packet. If so, the packet is put in a list of received packets and it is checked to see if the message of which this packet is a part can be reassembled. A message can be reassembled if all the component packets have reached the processor. If any user process is waiting for a message of this type from the source processor of the message, the message is handed over to that process. The receiving process is also unblocked. If this processor is not the final destination for this packet, the packet is put in a to-be-sent packets list. Also, an acknowledgment packet is sent to the node that sent the packet. Acknowledgment packets are only sent for data packets. If the received packet was an acknowledgement packet, the message-handling system decrements the count of packets sent to the processor from where the packet came. This procedure is repeated for all the available packets.

After checking for any arrived packets, the message-handling system checks to see if any messages have to be sent. If so, it packetizes the messages and puts the packets for the messages in a to-be-sent packet list.

The message-handling process then goes through the list of to-be-sent packets. For every packet in this list, it determines if more packets can be sent to the destination of the packet. If so, the packet is sent on the fiber-optic link using the write system call for the fiber-optic ports. Otherwise, the packet remains on the list of to-be-sent packets.
Figure 3.9: Flow chart of the message handler process
3.2.6 Packets

The message handling code breaks up a message into packets of a standard size. A standard packet header, shown in Table 3.2, is created for each packet. The packet header contains required fields for the proper identification of the packet, the message to which it belongs, the source, the destination and the intermediate routing node to which the packet goes next. It also has a flag which tells whether the packet is a data packet or an acknowledgment packet.

After packetizing the message, the message handling process releases the sending process in case of a blocking send. Several different protocols regarding the release of a blocked process have been used by different message-handling systems. Some options for this decision are:

1. wait until the message has been divided into packets.
2. wait until the message leaves the sending node.
3. wait until the message arrives at the destination node.
4. wait until the message is received by a process at the destination node.

Our message-handling system waits until a message is packetized before releasing the process which sent the message. If the process is to be released after it has reached the destination (options 3 and 4), we will have to return some information regarding the acceptance of message in the acknowledgment packets.

3.2.7 Assembling Messages at the Final Destination

When a message handling process decides that its processor is the final destination for a packet, it tries to assemble the message sent from the sender. For this purpose, it
<table>
<thead>
<tr>
<th>Next destination processor id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next destination net id</td>
</tr>
<tr>
<td>Current processor id</td>
</tr>
<tr>
<td>Final destination id</td>
</tr>
<tr>
<td>Flag</td>
</tr>
<tr>
<td>Message id</td>
</tr>
<tr>
<td>Message type</td>
</tr>
<tr>
<td>Message size</td>
</tr>
<tr>
<td>Message source</td>
</tr>
<tr>
<td>Packet sequence no</td>
</tr>
<tr>
<td>Packet size</td>
</tr>
</tbody>
</table>

Table 3.2: Header of the data packets

First has to check if all the packets from this message have arrived at the node. Since the messages have distinct identification numbers for every processor, the message ids and the source processor ids can be compared to determine if the packets belong to the same message. Once it has been decided that all packets of a message have arrived, the original message is assembled from the packets' data buffers.

Then the message handling process checks to see if there is any process waiting to receive a message of this type and/or sender. This check is done on two fields, the message type and the source processor id. If the receiving process wants to receive a message which has the same type and source id, the message is handed over to the receiving process, and it is also released if it is blocked on the message receive. The receiving process can also elect to receive a message of any type from a particular processor, or a typed message from any processor. It may also receive a message of any type from any processor, in which case the first message to be assembled on the node will be handed over to the receiving process.
3.3 Summary

This chapter described a distributed-memory message-passing multicomputer built using the IBM RS/6000 computers. These computers are linked together using point-to-point 220 Mbps fiber-optic links. A message-passing system has been developed considering the particular characteristics of the fiber-optic links on the RS/6000 computers. The message-handling is performed by a thread running under a light-weight threads package, which also allows the user processes to be run as light-weight threads. The message-handling system supports blocking and non-blocking asynchronous message-passing calls, and ensures reliable message delivery.

The simulation system presented in the next chapter simulates the multicomputer described in this chapter. This multicomputer is used for validating the simulator, and is the base setup with which the rest of the performance graphs are compared.
Chapter 4

RIOSIM: An Execution-driven Simulation System for the Target Multicomputer

The multicomputer consisting of IBM RS/6000 machines is simulated using RIOSIM, an execution-driven parallel architecture simulator. RIOSIM is a part of the RPPT. RPPT consists of various parallel architecture simulators, including RIOSIM, NETSIM, and PARCSIM. These simulators provide different capabilities for the user.

RIOSIM can simulate the execution of an appropriate parallel program on the target machine. Currently RIOSIM uses the communication model developed for the fiber-optic links described in the previous chapter.

In this chapter, we describe the execution-driven simulation technique in detail. Also, we discuss the parallel architecture simulator. The validation of the simulation system’s results against real timing measurements is also presented along with a description of the algorithms used for the validation.

4.1 Execution-driven Simulation Technique

Execution-driven simulation is a simulation technique which is nearly as accurate as instruction-level simulation but is more efficient since the individual instructions are not emulated. In execution-driven simulations, the execution of the parallel programs and the simulation of the simulated architecture is interleaved. A block diagram of an execution-driven simulation system is shown in Figure 4.1.

In this simulation technique, a process is selected and executed on the host machine until a process interaction point is reached. A process interaction point is a point in the execution of the process code where the future behavior of the process depends
Figure 4.1: Block diagram for Execution-driven Simulator
on other processes, or where other processes may be influenced by the next action of the process. Examples of these points include calling message passing primitives for simulating distributed-memory machines, or calling synchronization primitives like barriers or locks for shared-memory machines. At each process interaction point, a timing profiler is used to give an estimate of the time spent in executing the program from the last interaction point. This time is charged to the processor if the process is running on a simulated processor. At this point, any data movements in the parallel computer are simulated and the simulation time is advanced accordingly. A global event list is maintained for all the processes which are ready to execute. Processes are taken from the event list and executed in simulation time order.

An advantage of the direct execution of the user processes over instruction emulation is that there is less overhead in direct execution. The profiling of the program introduces a small overhead if it is a standalone profiler. The in situ profiler for the RS/6000 introduces overheads of the order of 150 because of the runtime modeling of the superscalar processor (see Chapter 2). Simulating the movement of data introduces some overhead, too. This overhead depends on the type of the network being simulated and the level of detail to be simulated.

For further reading on execution-driven simulation technique, see [Covington88a, Covington88b, Covington89, Covington91].

4.2 RIOSIM: A parallel architecture simulator

RIOSIM is a parallel architecture simulator using the execution-driven simulation technique described above. RIOSIM has been used to simulate the RS/6000-based multicomputer described in Chapter 3.
To simulate a target multicomputer, the user provides the parallel program which will drive the simulation model. The parallel program is the same as that would run on the multicomputer nodes and is written in C using a parallel programming library. The user also must include code to create target (simulation model) processors, to define how these processors are interconnected and how messages are routed among them, to create processes that execute the parts of the parallel program, and to attach processes to processors. These steps are explained in detail in [Rizvi93].

RIOSIM has been designed to easily allow changes in the simulated architecture to study the performance of the simulated multicomputer by varying different parameters of the multicomputer. The parameters that can be changed are

- the maximum packet size for the system.
- the transmission speed of the communication links.
- the time delays in sending and receiving packets on different nodes.
- the speed of the simulated processor.
- the presence of a data cache.

To simulate the RS/6000-based multicomputer, RIOSIM uses the following default values for the above parameters.

- The maximum packet size is set at 4096 bytes.
- The transmission speed of the serial links is simulated at 176 Mbps.
- The time delays in the operating system are calculated using the model of the fiber-optic links described in Section 3.1.3.
- The RS/6000 processor is simulated to be running at 25 MHz.
The standard 64-KB 4-way set-associative cache for the RS/6000 computer is simulated. The size of the cache can be easily changed by redefining a variable and recompiling the application.

The architecture of the simulated multicomputer can also be easily changed to describe a different architecture of the system. For this purpose, the user needs to redefine the routing function for the system and the interconnection structure of the node processors of the multicomputer.

4.3 Calculating performance of the simulated multicomputer

The single most important metric of the performance of a multicomputer is the execution time of a parallel program running on the multicomputer. This section describes how RIOSIM computes the execution time for the execution of the parallel programs on the simulated target multicomputer.

The total time spent in executing a program on a multicomputer has two components.

- Computation time
- Communication time

The computation time is the time spent in executing the processes running on the different node processors. In RIOSIM, to account for the time spent in the user processes, we use the profiler for the RS/6000 computers. The profiler, which is typically accurate to within 5% of the measured time, is described in Chapter 2. At every process interaction point, the profiler is used to get an estimate of the time
required to execute the block of code from the last process interaction point. This
time is charged to the processor on which the user process is running.

The communication time for the multicomputer used in this study is composed of
the following components.

- Time spent in the message-handling code
- Time spent in the operating system in sending and receiving packets
- Time spent in the transmission of data over the fiber-optic links

The time spent in the message-handling code is accounted for by profiling the
message-handling code. Again, at every process interaction point, the time reported
by the profiler is charged to the processor on which the message-handling process is
running.

The time spent in sending and receiving packets is charged every time the message-
handling process makes a write or read system call. To account for the time spent
in the operating system in writing and reading packets on the fiber-optic links, we
use the model of the fiber-optic ports. The model produces estimates of the time to
delay the processor of the process calling a write or read routine.

A write system call is simulated by first delaying the message-handling process
by the time spent in the operating system, and then handing over the packet to
a MoveOutHandler process. The MoveOutHandler process is not attached to any
processor, and is used to charge the time spent in the movement of data over the
fiber-optic links to the system. The time spent in this data transfer is calculated
using Equation 3.1. The MoveOutHandler process is delayed by the calculated time,
so that the data is only available at the other processor after the calculated amount
of time representing the data transfer time has elapsed. After the delay, the process
puts the packet in a list of available packets at the other processor.
A read system call is simulated by delaying the message-handling process that made the call by an appropriate amount of time. This time is again calculated using the fiber-optic ports model. The packet on top of the available packets list is handed over to the message-handling process, which then tries to re-assemble the message to which the packet belongs.

4.4 Validation of Simulation Results

Our goal is to create a simulation system that produces accurate performance measurements with reasonable efficiency. We studied the accuracy of the simulator's performance predictions by comparing the timings projected by the simulation system against the measured times on a two-processor system. We are limited to a two-processor system because of available computer resources. However, validation on a two-processor system is sufficient to judge the accuracy of the serial optical links model, which is independent of the number of processors in the system.

4.4.1 Algorithms used for simulation validation

To validate the performance of the simulation system, we wrote several parallel programs for our message-passing system, and the timings obtained from the real two-processor system were compared with the estimates produced by the simulation system. The programs are: matrix multiplication, finite differencing, a traveling salesman problem solver, singular value decomposition, quicksort, and LU factorization. The algorithms are described in the following sections.
4.4.1.1 Matrix Multiplication

The matrix multiplication problem is to multiply an $m \times n$ input matrix by an $n \times p$ input matrix to obtain a resultant $m \times p$ matrix. This work is easily parallelizable by giving each processor a share of the resultant matrix and providing the processor with the required rows and columns from the input matrices. The processors work on their parts of the matrices, and on completion send the results to a manager node, which compiles all the results to obtain the final resultant matrix. In our experiments, we obtained the input matrices using a random number generator.

4.4.1.2 Finite Differencing

Finite differencing approximations form the basis of many iterative methods commonly used in solving partial differential equations. These equations occur in the study of natural phenomena such as weather forecasting and determining temperature gradients over an area as well as in commercial problems like aircraft wing design, econometrics, etc. The basic finite differencing algorithm is an iterative algorithm. The area being modeled is divided into a grid of points depending on the required granularity. The function value at each point is a function of the previous value and some weighting of the values of the neighboring grid points.

Finite differencing can be parallelized by dividing the area into subarrays and assigning a worker process to each. After each iteration, the workers exchange the boundary values for each subarray they are working on. For our implementation, the area is divided by rows. If $N$ is the problem size and $P$ is the number of processors available, each worker is assigned $\left\lfloor \frac{N}{P} \right\rfloor$ rows, with extra rows being divided equally among the worker threads. A manager process creates the array and sends it to the worker threads. For a two-processor implementation, one manager process and
a worker process run on one processor while a worker process runs on the other processor.

4.4.1.3 Traveling Salesman Problem

The traveling salesman problem (TSP) is a graph search problem. The object is to find a minimum length tour for a graph, where a tour is defined as a path traversal which starts at one node of the graph, passes once through each node in the graph and comes back to the starting node. Trying all possible path permutations to determine the minimum length tour has a prohibitive cost. However, some simple optimizations are possible which can significantly improve performance. A basic optimization is that any (sub)tour that is longer than the current minimum length tour already discovered cannot be the minimum length tour, nor can any tour created by extending it. This process of removing subtrees from the search space is known as pruning.

In our implementation, a priority queue of partially evaluated tours is maintained by a master process. The remaining sub-tours are ordered in the inverse order of their total length. Thus, the sub-tours that are "most promising" are evaluated first, making it likely that the shortest tour will be found early in the search, and thus allowing quick pruning of the rest of the tours. Slave processes are started at other nodes, which send requests to the master for sub-tours to be solved. The master returns a sub-tour or an indication that no more work remains to be done. The slaves tell the master if they find a new global minimum, and the master is responsible for propagating it.

4.4.1.4 Singular Value Decomposition

The Singular Value Decomposition (SVD) of a matrix is an important, computationally complex algorithm used extensively in signal and image processing applications.
A singular value decomposition of an $m \times n$ matrix $A$ is given by

$$A = U\Sigma V^T$$  \hspace{1cm} (4.1)

where $U$ and $V$ are orthogonal matrices of order $m \times m$ and $n \times n$ respectively and $\Sigma$ is an $m \times n$ diagonal matrix.

In the Brent, Luk and Van Loan algorithm [Brent90], which we have implemented, the matrix is divided into $2 \times 2$ sub-matrices. Each worker is allocated one such sub-matrix. A Givens' rotation [Modi88] is then applied to all sub-matrices in the diagonal to obtain diagonal $2 \times 2$ sub-matrices. The rotation angles thus obtained are propagated systolically to the other workers. Also, the workers exchange matrix data elements diagonally, after the diagonal neighbor has received and applied the necessary rotation angles.

This algorithm is originally written for VLSI-based systolic arrays which have a small communication time for neighboring processors. In our implementation, each worker is represented by a process and the total processes are divided equally between two processors.

4.4.1.5 Quicksort Algorithm

The quicksort algorithm is a recursive sorting algorithm which works by partitioning a given input list into sublists of elements with increasing or decreasing magnitude. The algorithm recursively works on these unsorted sublists unless all the lists are sorted. The algorithm uses the bubblesort method when the size of a list reaches a particular size determined by the user (called the threshold). The quicksort algorithm is parallelized by using a work queue that contains the unsorted lists. The worker threads can remove unsorted lists from the queue to sort.
A master process runs on one of the processors, while worker processes may run on other processors. For the two-processor case, a master and a worker process run on one processor, and a worker process runs on the second processor. The master process maintains a work queue and performs the partitioning of the unsorted list. The slaves send request messages to the master to obtain an unsorted list, and the master responds by sending either an unsorted list, if any is present, or a done message if no lists are present. The workers work on the unsorted lists and return them to the master on finishing the sorting.

4.4.1.6 LU Factorization

The LU factorization of a matrix $A$ decomposes it into a lower triangular matrix $L$ and an upper triangular matrix $U$. After obtaining the LU factorization for a matrix, the system of equations

$$Ax = b$$

(4.2)

can be easily solved by first solving $Ly = b$ for $y$ and then solving $Ux = y$ for $x$.

The algorithm implemented for our study uses pivoting in order to avoid round-off errors and also load-balancing techniques. The parallel algorithm is based on a system with a hypercube architecture, and is described in [Geist88]. The algorithm divides the matrix among the available processors using wrapping (i.e., assigning row $i$ to processor $i \mod p$). The algorithm works by finding a pivot row for each variable in succession. For determining the pivot row, a minimal spanning tree is embedded in the hypercube network, which allows effective global communication among the different nodes.

A process is assigned to every processor. Each process finds its local maximum from the pivot column and then sends it to its parent in the spanning tree. The parents
compare these received maxima with their own local maximum and forward the new maximum up the tree. The pivot row is thus determined by the root process in the tree, and it sends this information down the tree. The process holding the pivot row is responsible for sending it to every process. Load-balancing is implemented by only allowing a processor to have a maximum of \( \left\lfloor \frac{n}{p} \right\rfloor \) pivot rows. If a processor has more, it exchanges the pivot rows with some other processor. The rest of the algorithm is the basic LU factorization algorithm described in any numerical algorithms book.

4.4.2 Results

For the algorithms mentioned above, tests were conducted on a two-processor multiprocessor computer first. The tests were run by rebooting the RS/6000 computers in standalone mode, so that there was no other process running along with this program. Time was measured using the AIX command time for a number of runs of the programs, and the minimum of these data was accepted as the final measurement. The reported measured time is the sum of the user time and the system time as given by time. The simulated time is the sum of the following two values. First, the programs were run under RIOSIM simulating a two-processor system, and the simulated time was noted for this run. Then, we added the time spent in the light-weight context switches between the user process(es) running on a processor and the message-handling process running on that processor. To obtain the number of context switches, we ran the programs on the two-processor system and put hooks in the light-weight context switching code to give the total number of context switches. The results of the tests are shown in Table 4.1. Error is calculated as follows.

\[
Error = \frac{Simulated \ Time - Measured \ Time}{Measured \ Time} \times 100 \tag{4.3}
\]
Positive error means that the simulation system projected a higher estimate than the real system, while negative error implies a lower estimate.

The results for matrix multiply are for multiplying two $128 \times 128$ matrices. LU decomposition times are for solving simultaneous linear equations in 20 variables. Quicksort algorithm is run with 8192 random numbers with a threshold of 256. SOR applies the iterative algorithm on a $512 \times 512$ elements grid and performs 20 iterations. The SVD program diagonalizes a $32 \times 32$ matrix and does 6 iterations. TSP solves an 18-city tour problem.

Most of the projections for simulated time are within 10% of the measured time. The LU factorization and the SOR algorithms have a higher percent error. We think the reason for the high error is the large amount of message-passing needed for the two algorithms. In LU algorithm, for every variable, three messages are transferred between the two processors for determining and distributing the pivot row for that variable. Similarly, for the SOR algorithm, two messages are exchanged per iteration between the two processors. The simulated time estimates are functions both of the profiler and the architecture model. Several potential causes of profiling errors are discussed in Chapter 2.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Measured Time (sec)</th>
<th>Simulated Time (sec)</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiply</td>
<td>0.67</td>
<td>0.6142</td>
<td>-8.3</td>
</tr>
<tr>
<td>LU decomposition</td>
<td>4.01</td>
<td>3.3226</td>
<td>-17</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.95</td>
<td>0.8786</td>
<td>-6.1</td>
</tr>
<tr>
<td>Successive over-relaxation</td>
<td>4.01</td>
<td>3.4396</td>
<td>-14</td>
</tr>
<tr>
<td>Singular value decomposition</td>
<td>3.76</td>
<td>3.4039</td>
<td>-9.5</td>
</tr>
<tr>
<td>Traveling salesman problem</td>
<td>12.96</td>
<td>12.5493</td>
<td>-3.2</td>
</tr>
</tbody>
</table>

Table 4.1: Validation Results for the Simulation System
Table 4.2 shows the time spent in running a program on a two-processor system (measured time), and the time required by RIOSIM to simulate the multicomputer running the program (simulation time). Simulation time was measured using AIX command time. Simulation overhead is calculated by

\[
\text{Simulation overhead} = \frac{\text{Simulation time}}{\text{Measured time}}
\]  

(4.4)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Measured Time (sec)</th>
<th>Simulation Time (sec)</th>
<th>Simulation Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiply</td>
<td>0.67</td>
<td>237</td>
<td>354</td>
</tr>
<tr>
<td>LU decomposition</td>
<td>4.01</td>
<td>2829</td>
<td>705</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.95</td>
<td>200</td>
<td>210</td>
</tr>
<tr>
<td>Successive over-relaxation</td>
<td>4.01</td>
<td>926</td>
<td>231</td>
</tr>
<tr>
<td>Singular value decomposition</td>
<td>3.76</td>
<td>947</td>
<td>252</td>
</tr>
<tr>
<td>Traveling salesman problem</td>
<td>12.96</td>
<td>3913</td>
<td>302</td>
</tr>
</tbody>
</table>

Table 4.2: Simulation overheads for RIOSIM

For most of the programs, the simulation overhead is around 300. For the LU decomposition, the simulation overhead is around 700. A reason for this high overhead is the large amount of message-passing between the two processors. This requires the simulation of the packetizing of the messages, routing of the packets, and the movement of data, which adds to the overhead. The overhead for SOR is low in spite of the large number of messages transferred because the computation-to-communication ratio for this algorithm is large, and the profiling overhead is lower compared to the overhead for simulating message-passing.

4.5 Summary

RIOSIM is a parallel architecture simulator using the execution-driven simulation technique. Execution-driven simulators utilize the direct execution of programs on
the target processor to reduce the simulation overheads, and are more efficient than instruction-level simulation systems. RIOSIM is a flexible simulator and can be used to simulate any different multicomputers with different architectures.

RIOSIM is used to simulate the multicomputer consisting of IBM RS/6000 computers. The simulation results were compared against measured time on a two-processor multicomputer for a variety of algorithms. The simulator projections are uniformly low ranging from 3% to 17%. Simulation overheads are usually around 300, which supports our case for an efficient simulator. This simulator is used in projecting performance curves for different multicomputer settings by changing different parameters of the current setup. These parameters include the maximum packet size, the speed of the communication link, the message overhead, and the simulated architecture of the multicomputer.
Chapter 5

Performance Studies of the Multicomputer

The multicomputer described in Chapter 3 was simulated using the execution-driven simulator described in the previous chapter to study the performance of the multicomputer. We use the following programs for the performance study: matrix multiplication, successive over-relaxation algorithm, a traveling salesman problem solver, quicksort, singular value decomposition, and LU factorization. These programs are described in detail in Chapter 4. We chose these programs because they represent a wide variety of algorithms that are typically used in scientific and industrial applications, which include grid-point computations (SOR), linear algebra problems (matrix multiplication, LU factorization, quicksort), signal processing problems (SVD) and graph computations (TSP).

We simulated the multicomputer using these programs and obtained performance data for a variety of system configurations. The base setup of the multicomputer included 176 Mbps fiber-optic links\(^1\), a packet size of 4096 bytes and overhead in sending a message as in Table 3.1. Each program is set to run on a different multicomputer architecture, which is described when we discuss the performance of the multicomputer for the program. We ran the simulations and obtained speedup curves as a function of the number of processors in the multicomputer for the following cases.

- For the algorithms simulating the architecture for which the algorithm has been developed. These data use the base setup of the multicomputer for the simulations.

\(^1\)The rated speed of the serial link is 220 Mbps, but it uses an 8/10 coding for every byte [see Section 3.1.2]
- By increasing the maximum size of the packets into which a message is broken.

- For a reduced transmission rate of the serial link. We have made our studies with a 10 Mbps link, which corresponds to the speed of an Ethernet link.

- With a reduced overhead in transmitting a message on the serial link.

- Using different architectures for the algorithms. The other parameters of the system are maintained same as for the base setup.

- By making changes in the original algorithms to suit our system.

### 5.1 Basic Performance Curves

This section presents simulation results for the different algorithms. The algorithms were run with the base setup of the multicomputer.

The matrix multiply algorithm was executed to calculate the product of two $512 \times 512$ matrices. Figure 5.1 shows the simulated time required for the execution of the program. The simulation models a hypercube connection of the nodes in the multicomputer. We have simulated hypercubes up to four dimensions. The speedup curve is shown in Figure 5.2. The algorithm shows far less than linear speedup. The reason for the non-linear speedup is some serial code present in the root process' code. This serial code sets up the input matrices for sending them to the other nodes at the start, and to compile the results sent by the other nodes at the end. The situation is aggravated by the large amount of time required for packets to reach destinations which are more than one hop away due to the store-and-forward nature of the interprocessor communications.

The basic timing and speedup curves for the successive over-relaxation algorithm are shown in Figures 5.3 and 5.4, respectively. These figures are for calculating SOR
Figure 5.1: Time to perform a \(512 \times 512\) matrix multiply

Figure 5.2: Speedup curve for a \(512 \times 512\) matrix multiply
Figure 5.3: Time to calculate a 512 x 512-elements SOR (200 iterations)

Figure 5.4: Speedup curve for a 512 x 512-elements SOR (200 iterations)
on a $512 \times 512$ matrix; the number of iterations used in the simulation was 200. The algorithm was developed for a one-dimensional mesh, in which all processors are connected to two processors each except the first and the last processors which are connected to one processor each. The speedup curve shows some speedup, 3.8 for a 6-processor network. The non-linear speedup is because of the high number of messages transferred between neighboring nodes. This is confirmed when we run experiments with a lower message overhead, as described in Section 5.4.

Time measurements for the quicksort algorithm are shown in Figure 5.5. The reason for the behavior shown is that the algorithm uses a master process (see Section 4.4.1.5) and a slave process on one processor, while the other slaves are distributed on other processors, one on each processor. The processors are connected in a star-shaped pattern, with the processor running the master process in the middle. Due to a large amount of time spent in sending and receiving a message, most of the work is done by the slave process running on the root processor, and therefore almost no speedup results.

Figure 5.7 shows the time required to solve an 18-city traveling salesman problem. The curve does not show a speedup. The algorithm was run with one master process and several slave processes (see Section 4.4.1.3). The connection pattern for the processors for this experiment was also star-shaped, with the master process running on the processor in the middle. Simulations were run for sorting a 25000-element array with a threshold of 256 elements. We think the reason for a bad speedup curve is similar to the one for the quicksort program.

The LU factorization program solved a 200-variable simultaneous linear equations system. This algorithm was developed for a hypercube-type architecture. Figure 5.8 shows the time required to solve this system on hypercubes of different dimensions. The curve shows a negative speedup for increasing number of processors. The time
Figure 5.5: Time to sort 25000 elements using Quicksort

Figure 5.6: Speedup curve for sorting 25000 elements using Quicksort
Figure 5.7: Time to solve an 18-city traveling-salesman problem

Figure 5.8: Time to perform a 200-order LU factorization
needed to solve the system increases linearly with the number of processors used. As explained in Section 4.4.1.6, the algorithm requires the processors to exchange data for determining the pivot rows, which happens once every iteration. Also, for load balancing, the processors exchange rows in case of bad pivot rows distribution. This fine-grained message-passing is not suitable for our system because of a high overhead in sending small-sized messages. We also ran the algorithm with a lower message overhead; the results of this experiment are described in Section 5.4.

5.2 Effects of changing packet size

This section presents the study of the effects of changing the maximum packet size on the performance of the system. The default maximum packet size for the system is 4096 bytes. If a message is bigger than this size, it is divided into packets of 4096 bytes each and these packets are then sent to the destination. The user is allowed to change the maximum packet size up to 12288 bytes (this limitation is explained in Section 3.2.4).

We ran the matrix multiplication algorithm with a packet size of 8192 bytes on a hypercube of up to four dimensions. The simulated times for different number of processors is shown in Figure 5.9. The performance of the system improves slightly since most messages are big and utilize 8192-bytes packets, and sending/receiving one 8192-bytes packet incurs less overhead than sending/receiving two 4096-bytes packets. The performance improvement ranges from about 1.0% for a two-processor system to about 6.2% for a 16-processor system.

Similarly, for the SOR algorithm, we see modest performance gains by increasing the maximum packet size to 8192 bytes. Figure 5.10 shows the time spent to execute the algorithm for maximum packet sizes of 4096 and 8192 bytes. The improvement
Figure 5.9: Time to perform a 512 x 512 matrix multiply

varies from 0.7% for a two-processor system to 4% for a six-processor system. The reason for a small change is that only the initial messages containing the parts of the matrix are big enough to need 8192 bytes. The rest of the messages exchanged between the processors are smaller than 4096 bytes and therefore the time spent in sending and receiving these packets does not change for the two cases.

The rest of the algorithms are such that they do not exchange big messages among the processors. We do not expect these algorithms to show any performance improvement by increasing the maximum packet sizes.

A smaller maximum packet size would not be very beneficial for this system. From Table 3.1, the first jump in operating system delay is for 236-bytes packets. If a message is 1000 bytes long, it will be broken into five packets for a 236-bytes packet size setup, while with a packet size of 4096 bytes, only one packet is required. From the
Figure 5.10: Time to calculate a 512 x 512-elements SOR (200 iterations)

timing model in the table, the former would incur a delay of 0.4425 x 5 = 2.2125 msec while the latter would incur 0.6424 msec. For messages smaller than 236 bytes, both would incur the same delay for sending the messages. An extremely large packet size is not advisable because of memory allocation requirements. The message handling system allocates memory equal to the packet size every time it tries to receive a packet.

5.3 Effects of changing transmission speed

An important component of the system is the high-speed fiber-optic link which has a transmission rate of 176 Mbps. In this section, we present results for simulating a system with a different transmission rate (10 Mbps) and compare the performance of the two systems.
Figure 5.11: Time to perform a 512 x 512 matrix multiply

Figure 5.12: Time to calculate a 512 x 512-elements SOR (200 iterations)
Figure 5.11 shows the simulated time for running the matrix multiply algorithm on the two systems. The increase in simulated time is 3.7% for two-processors and 2.4% for a 16-processor system. For the SOR algorithm (Figure 5.12), the time to perform 200 iterations is 5.53% higher for the two-processor system. For a 6-processor system, the execution time increases by 18.25% compared to the base system.

Because of the high message overhead in the OS in sending and receiving data, the transmission speed of the link does not play a significant role for small-sized messages. When the data sent is very large, the time spent in transmission of messages becomes comparable to the time spent in the OS, and changes in transmission speed affect the overall performance by a significant amount. Two messages are exchanged between all neighboring nodes for the SOR algorithm. At high transmission speed, the total time spent in the transmission of all the messages is negligible compared to the overhead in sending/receiving the messages, but at a lower speed, it becomes significant and results in a large increase in the execution time.

5.4 Effects of changing the message overhead

The time spent in the operating system and the message-handling system for sending and receiving packets is another important parameter. We believe that the performance of the multicomputer is reduced because of the large amount of time spent in the operating system in sending and receiving a message. The performance of the system can be improved considerably by reducing this overhead. The message-handling system and the operating system both do certain tasks like copying of messages into their own buffers, providing for acknowledgement, etc. This redundancy can be avoided if the message-handling system uses a custom-written device driver for the
serial links. This would result in a smaller overhead in sending and receiving every message.

We ran simulations with the operating system overhead reduced by a factor of four, and compared the results with the measurements for the base setup. We also ran simulations with no overhead in sending and receiving a message (by setting the operating system overhead to zero and not profiling the message-handling process' code). The results of these experiments are described below.

Figure 5.13 shows the results of the test for the matrix multiply algorithm. For the lower message overhead, this algorithm shows a maximum reduction in execution time of 10.9% for 16 processors. With no message overhead, the reduction is 16.6%. This algorithm still does not show a linear speedup because of the presence of some serial code described earlier. The SOR algorithm (Figure 5.14) shows an improve-
Figure 5.14: Time to calculate a 512 x 512-elements SOR (200 iterations)

Figure 5.15: Speedup curve for a 512 x 512-elements SOR (200 iterations)
Figure 5.16: Time to sort 25000 elements using Quicksort

describing the results of the experiment ranging from 3.6% for two processors to 16.7% for 6 processors for the reduced overhead. With no message overhead, the SOR shows an almost linear speedup curve with improvements from 7% for two processors to 34.3% for 6 processors. Quicksort results are shown in Figure 5.16, showing a maximum improvement of 23.5% in execution time for the first experiment, while it shows a 29.9% improvement for the second experiment. Though the LU factorization algorithm does not show any speedup using a lower message overhead (Figures 5.17 and 5.18), the algorithm shows a lot of improvement in execution time. For two processors, it is 45.4%, while for 16 processors, the change is 16.6%. When the message overhead is set to zero, the LU algorithm shows some speedup. Since the LU factorization is a fine-grained algorithm, a reduction in the high overhead of sending a message should improve the performance considerably.
Figure 5.17: Time to perform a 200-order LU factorization

Figure 5.18: Speedup curve to perform a 200-order LU factorization
As expected, all of the experiments showed a significant improvement in performance when the message overhead was reduced or set to zero. Thus, if we could reduce the total amount of time spent in the message-handling system and the operating system for sending and receiving messages, we would get good performance from the multicomputer.

5.5 Effects of changes in simulated architecture

We have studied the effects on performance of changing the architecture of the simulated multicomputer. First, we changed the architecture for the matrix multiply algorithm from a hypercube connection to a star-shaped connection. In this configuration, the master processor is connected to every other processor. The timing and speedup curves are shown in Figures 5.19 and 5.20. This setup shows a good performance with speedup of 2.15 for five processors.

For the SOR algorithm, we changed the original setup of a one-dimensional mesh to a ring (a one-dimensional torus). The routing algorithm was changed to send the packets for a shortest-length path. Thus if a packet has to be sent from processor 0 to processor 4 in a 6-processor system, it will go from processor 0 to processor 5 to processor 4. The results of this test are illustrated in Figure 5.21, showing a maximum improvement of 4.3%. A low improvement figure is expected since the only multihop messages (which would benefit from this setup) are for sending and receiving of the matrix at the beginning and end of the algorithm. The rest of the message-passing is between neighboring processors and is not affected by this change. An algorithm which has frequent message-passing between non-neighbor nodes would benefit more from this setup.
Figure 5.19: Time to perform a 512 x 512 matrix multiply

Figure 5.20: Speedup curve for a 512 x 512 matrix multiply
Figure 5.21: Time to calculate a $512 \times 512$-elements SOR (200 iterations)

Figure 5.22: Time to perform a 200-order LU factorization
5.6 Effects of load balancing in LU factorization

The LU factorization algorithm used in the above experiments has load-balancing properties. If the pivot rows are badly distributed, i.e., some processors have more than one pivot row while others do not have any, the processors exchange their rows to get the load balanced. This pivoting strategy is called *dynamic pivoting* [Geist88]. We ran the LU factorization algorithm both with and without dynamic pivoting. The results are shown in Figure 5.22. For two processors, the algorithm without load-balancing performs 6.8% faster than the version with load-balancing. For 16 processors, the former is 3.8% faster than the latter. The effects of this change in algorithm on the performance of the system are very data dependent, and would vary with a different set of data.

5.7 Conclusions

In this chapter, we presented performance studies of the multicomputer based on the RS/6000 processors. Using RIOSIM, we have simulated the execution of a variety of algorithms of different granularity and belonging to different classes of algorithms.

The simulations of these algorithms on the base setup of the multicomputer indicate that this system can be used to an advantage for coarse-grained algorithms. For fine-grained algorithms, the high overhead in message-passing proves to be a bottleneck. This theory is confirmed when we simulated the execution of the same programs by setting the message overhead to be lower and later to zero. Some coarse-grained algorithms show almost linear speedup, while the fine-grained algorithms show a significant improvement in execution time. Changes in packet sizes affect the execution time if the message sizes are big. Transmission speed changes affect the performance significantly if the total amount of data exchanged between the node processors is
large. We have also simulated some algorithms for a different architecture than the one set up originally. A star-shaped architecture for the matrix multiply shows a lot of improvement in execution time.
Chapter 6

Conclusions

6.1 Summary

The goal of this research was to determine the performance of a message-passing coarse-grained parallel computer built using low-cost commercially available workstations. The primary performance metric we were interested in is the speedup for different algorithms shown by the system as we increase the number of processors in the system. A multicomputer consisting of IBM RS/6000 computers linked by high-speed fiber-optic links was used for this study. The RS/6000 computers are well-suited as the base processors because of a high performance processing unit, especially the floating-point unit, and the availability of 220 Mbps fiber-optic links.

We developed a message-handling system, that ensures reliable messages delivery, to run on these computers under a light-weight threads package. This message-handling system was developed after analysis of the characteristics of the fiber-optic ports on the RS/6000 to provide maximum performance. The user code is written in C using a parallel programming library, and it also runs as a thread under the light-weight threads package.

An execution-driven parallel architecture simulator, RIOSIM, was developed which accurately simulates the multicomputer. The simulation system was validated by comparing its projected results against measured times on a two-processor multicomputer. The simulator was found to be projecting low within the range of 3 to 17%. The simulation overhead for the system were found to be around 300. For the execution-driven simulation, we also developed a timing profiler for the RS/6000,
which uses an innovative technique for simulating the superscalar RS/6000 processor. Standalone tests for the profiler give errors of up to 5% with profiling overheads of around 150.

The validated simulation model was used to study systems with more than two processors. We ran several experiments to determine the performance of these multicomputers, and also experiments to determine the bottlenecks in the systems by changing different parameters of the systems. We simulated the execution of several algorithms with different granularity on different types of interconnection of the processors. These experiments indicate that the system shows good speedups for coarse-grained algorithms, and hence can be utilized beneficially. For fine-grained algorithms, the high overhead for message-passing renders the use of multiple processors inefficient.

6.2 Future Work

This section describes some future avenues for extending the work described in this thesis.

6.2.1 Reduction of message overhead

As discussed in Section 5.4, the message-passing overhead has a big effect on the performance of the multicomputer, and if we could reduce the message overhead in the operating system by a factor of four, we would get a reduction in execution time up to 45.4%.

One way to reduce the operating system overhead is by writing a device driver for the fiber-optic ports customized to the message-handling system developed for this multicomputer. This would reduce the overall time spent in the message-passing be-
cause there are certain redundancies in the current setup. Both the message-handling system and the operating system device driver do similar tasks, like copying of messages into buffers, providing for acknowledgements, etc. The custom device driver should complement the message-handling system, and would remove the redundancies which are present right now.

6.2.2 Simulation of similar multicomputers

RIOSIM can be used to simulate any direct-connect multicomputer. Also, we have added features to it to be able to simulate switches. IBM makes a non-blocking switch for the fiber-optic ports. You can connect 14 ports to it and have 7 simultaneous 200 Mbps conversations [Rader92]. The switch makes dynamic connections and adds about 8-bit times to the transfer. This switch could be used to connect RS/6000 computers in a fully-connected system, or as part of a larger interconnection network.

A future improvement to RIOSIM could be to extend the fiber-optic ports model to include the switch model, and study the performance of systems built using these switches.

6.2.3 Extensions to the message-handling system

Currently, there are no multicast or broadcast facilities in the message-handling system. If an algorithm requires a multicast or broadcast, the user has to implement this in the user code by receiving messages on intermediate nodes and then sending them on to other nodes. A multicast (or broadcast) facility implemented in the message-handling system would be more efficient, since it would reduce the number of message copies, and the user would not need worry about this.
Bibliography


