A High-Performance Retargetable Simulator for Parallel Architectures

by

Chrysanthos Nicholas Dellarocas

Diploma of Electrical Engineering
National Technical University of Athens, Greece (1989)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 10, 1991

Certified by ....[illegible]..............
William E. Weihl
Associate Professor of Computer Science
Thesis Supervisor

Accepted by ....[illegible]..............
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A High-Performance Retargetable Simulator for Parallel Architectures

by

Chrysanthos Nicholas Dellarocas

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1991, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The complexity of the interaction between software and hardware in MIMD machines makes experimental evaluation of parallel programs an important complement to theoretical analysis. Traditional techniques used to monitor the direct execution of programs are intrusive and may lead to inaccurate results when applied to parallel programs. Simulation allows flexible, nonintrusive and repeatable evaluation of parallel programs but usually incurs prohibitive overheads that limit its use to the simplest applications.

In this thesis, we describe Proteus, a high-performance simulation-based system for the evaluation of parallel algorithms and system software. Proteus is built around a retargetable parallel architecture simulator and a flexible data collection and display component. The simulator uses a combination of simulation and direct execution to achieve high performance, while retaining simulation accuracy. Proteus can be configured to simulate a wide range of shared-memory and message-passing MIMD architectures and the level of simulaton detail can be chosen by the user. Detailed memory, cache and network simulation is supported. Parallel programs can be written using a programming model based on C and a set of runtime system calls for thread and memory management. The system allows nonintrusive monitoring of arbitrary information about an execution, and provides flexible graphical utilities for displaying recorded data.

To validate the accuracy of the system, a number of published experiments were reproduced on Proteus. In all cases the results obtained by simulation are very close to those published, a fact that provides support for the reliability of the system. Performance measurements demonstrate that the simulator is one to two orders of magnitude faster than other similar multiprocessor simulators.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science

Acknowledgements

I would like to thank Bill Weihl, my thesis supervisor, for his encouragement and support of my research, as well as for being a great group leader. I also thank Barbara Liskov, my academic advisor during my first 12 months at MIT, for her welcome, support and counsel, as well as for suggesting this interesting thesis topic.

Eric Brewer has been my officemate ever since I came to MIT as well as my partner in making Proteus a reality. Collaborating with him has been rewarding and fun. Adrian Colbrook, Anthony Joseph and Wilson Hsieh were the first persons (un)fortunate enough to use Proteus for their research. Their skill at uncovering the most obscure and nasty bugs of the system is greatly appreciated.

The Parallel Sofware Group of MIT's Laboratory for Computer Science has been the ideal place in which to conduct this research, both in terms of human resources and group spirit, as well as in terms of technical support and organization.

I thank my roommates George Stamoulis, Stelios Smirnakis and Mike Goldstein for many pleasant moments we spent together. I also thank my good friends Helen Halkias and Maria Roussohatzaki for being there when I needed them most.

Last but not least, I would like to thank my family for always being by my side.

Contents

List of Figures

List of Tables

Reverse cannot befall
that fine prosperity
whose sources are interior

– Emily Dickinson

Chapter 1

Introduction

The performance of an algorithm on a parallel computer is heavily influenced by several factors that are not typically considered in a theoretical complexity analysis of the algorithm itself. The actual details of a parallel architecture (interconnection topology, routing method, cache coherence protocol, etc.) and those of the runtime system (scheduling, locking, communication overheads, etc.) often lead to unexpected performance degradations that are not always easily detectable by theoretical analysis. Furthermore, the interaction between software and hardware is so complex in MIMD machines that accurate analytical modelling of architectural and runtime system parameters is usually not possible. For these reasons, experimental evaluation of parallel programs is an important complement to theoretical analysis, much more than it is in the case of sequential algorithms.

Simply executing a program on a parallel machine and monitoring its performance is often difficult, due to a number of technical and economic considerations.

- Existing parallel machines are expensive and in limited supply; not all research groups can afford them.
- Because of the influence of architectural parameters in program performance [Kuck84], a reliable evaluation of a new algorithm would require testing it in a wide range of different parallel architectures.
- Even if access to the required number of parallel machines is possible, it can be difficult to obtain valid measurements from a live parallel system due to the probe effect [Gait86]: Obtaining any kind of measurement from a running program implies executing some extra monitoring code either at the program or at the microcode

level, which has the effect of distorting actual instruction execution time. For parallel machines, this distortion inevitably affects the interaction of different processing nodes and can therefore change the observable behavior of the program.

Simulation is emerging as an attractive alternative to actual experimentation in all branches of science and engineering. Simulation is especially attractive in the area of parallel program evaluation for a number of important reasons.

- Any parallel architecture can be simulated on an ordinary workstation. Researchers who rely on simulations do not need to purchase additional equipment.
- Simulation offers the ability to perform arbitrary analysis without disturbing timing relationships that may be crucial to program performance.
- Different simulations of the same program produce identical results. Therefore a problem identified during one run can be studied more thoroughly by repeating the same simulation and capturing more data related to the problem. Many real parallel machines on the other hand are nondeterministic: The same program may produce different results across different runs. Therefore an interesting phenomenon produced during one run may not be repeatable.

On the minus side, simulation has a number of limitations.

- Multiprocessor simulators slow down the execution of parallel programs considerably. High program slowdowns often limit the usefulness of the approach to only the simplest parallel applications.
- Simulators are based on models of real systems and must be thoroughly validated before any data obtained using them can be trusted to reach conclusions and make decisions.

The purpose of this thesis is to describe Proteus[1], a parallel program evaluation system that we developed as a tool for further research in parallel systems.

Proteus was born out of the need for a simulation-based platform on which to test the design of a new parallel programming language for MIMD machines. We needed a

[1]In Greek legend, Poseidon's herdsman, an old man and a prophet. Proteus was famous for his power of assuming different shapes at will. When caught, however, he was helpful to many Greek heroes by his foreknowledge. Proteus lived in a vast cave, and his custom was to count over his herds of sea-calves at noon, and then go to sleep. There was no way of catching him but by stealing upon him at this time and binding him; otherwise he would elude anyone by a rapid change in shape.

system that would be accurate enough so as to capture the important interactions of a parallel program with the operating system and the hardware. Our system should also be appropriate as a target for compilers. For those reasons, we decided to design an instruction-level simulator that simulates the behavior of a parallel machine at the level of individual machine instructions.

Since our language system is going to be portable, it is important for us to compare the performance of programs on different architectures. Hence, Proteus can be configured to simulate a wide range of shared-memory and message-passing MIMD architectures.

To allow simulations of nontrivial parallel programs in a reasonable amount of time, high-performance simulation was a major goal of this project. In addition, since different simulations have different accuracy and performance requirements, the user should be able to control the tradeoff between accuracy and performance by selecting the desired level of simulation detail.

An important goal of the system is to simplify the collection and display of execution and performance data. During simulation of a user program, the system outputs an event file that captures the important interactions between software and hardware. The system provides a number of predefined event types; the user can easily add other event types specific to his own application, or restrict the recorded event types to those relevant to his experiment. Event files are read by a sophisticated data display program that displays execution statistics in a variety of graphical forms.

Proteus is primarily used to evaluate parallel algorithms. It aims to provide its users with enough information to allow them to choose between different algorithms or implementations for a given task. It can also be used for research in operating system issues, such as scheduling and load balancing, as well as for testing hardware-related concepts, such as network routing methods or software and hardware cache coherence techniques.

Division of responsibilities

Proteus was developed jointly by the author and Eric A. Brewer as part of their S.M. thesis research at the MIT Laboratory for Computer Science. Most of the important overall system design and implementation decisions were taken jointly by the two authors. The author has been primarily responsible for the following aspects of the system:

- Design and implementation of an efficient simulation strategy that balances our performance and accuracy requirements.

- Design and implementation of all architectural simulation components.
- Design of the programming model and implementation of a preprocessor that translates our language extensions into standard C.
- Design and implementation of the runtime system.

This thesis will concentrate on the description of the system aspects for which the author has been responsible. A brief overview of the other system components will also be given.

Organization of this Thesis

The rest of this thesis is organized as follows:

Chapter 2 is a survey of different techniques that have been used to measure the performance of parallel systems. Its purpose is to describe the context in which this research is performed and to demonstrate that efficient evaluation of parallel systems is a difficult problem.

Chapter 3 provides an overview of the Proteus system. It describes the different components of the system and their interaction. It also demonstrates the system's power by explaining how the system was used to analyze and improve the performance of a simple program.

Chapter 4 is devoted to the system's design. For each of the major parts of the system, alternative designs are compared and the chosen design path is discussed in detail.

Chapter 5 presents and discusses a number of experiments performed to evaluate the accuracy and performance of our system. The results of those experiments are analyzed and compared to other similar systems.

Finally, Chapter 6 concludes the thesis by summarizing the main points made in the previous chapters.

Chapter 2

Background

In order to evaluate the performance of a program, some sort of data must be captured during the program's execution. This data can be analyzed later to provide insight into the program's behavior. This section is a survey of different techniques that have been used to gather information about program execution. Its purpose is to set the context in which the research reported in this thesis was performed. In addition, it aims to prove that efficient and reliable evaluation of parallel programs is a difficult problem. Most of the techniques that have been used with success to evaluate programs running on uniprocessors are either not appropriate or too expensive to be applied on multiprocessors. Hence, building a system that allows reliable and efficient evaluation of parallel programs is an interesting challenge. This chapter concludes with a brief overview of several systems that have similarities to our work.

Most existing evaluation techniques belong to one of two broad classes: Monitoring of direct program execution and simulation. A third class of techniques, analytical modelling, is not accurate enough to capture the detailed interaction of a program with the rest of the system and is usually used only to provide first order insights into the operation of a system. We will not consider analytical modelling in the rest of this chapter.

2.1 Monitoring of direct program execution

A large number of evaluation techniques rely on execution data gathered during the direct execution of a program on a real machine. Most of these techniques were developed for uniprocessor systems and usually do not perform well when applied to multiprocessors. In the following paragraphs, we shall examine the most important monitoring techniques and shall point out their principal strengths and weaknesses.

2.1.1 Hardware-assisted monitoring

A hardware monitor is a device that plugs onto a working backplane bus to record all activity on the bus of a system [Carpenter87, Malony89]. This technique has the advantage of nonintrusiveness. The operation of a program is usually not affected by the tracing. Captured data is therefore guaranteed to be accurate. Hardware monitoring is capable of capturing both user and operating system activity. In that way, an accurate evaluation of the interaction of a program with the operating system is possible. The primary drawbacks of this approach are its complexity, cost and lack of flexibility. Hardware monitoring can capture a fixed set of low-level machine events (bus accesses, hardware signals), but these cannot always be easily related to application events. Due to memory limitations, hardware monitors often record only counts of events, rather than the events themselves. Finally, hardware monitors are costly. In order to use them on a parallel computer, the complex hardware required would grow at least linearly with the number of processors.

2.1.2 Monitoring based on instruction interrupts

Some computer systems (e.g. some VAX machines [VAX82]) provide the capability of interrupting the execution of every machine instruction of a user program. The program then traps to a system routine that can capture all necessary information about the executed instruction. This technique does not need any extra hardware and is very flexible, since the monitoring routine can be modified to capture any kind of desired information. However, it suffers from some serious limitations. Operating system code, which is typically executed with interrupts disabled, cannot be monitored. Calling a trap routine before every program instruction slows down the execution of a program considerably (up to 100,000 times in some cases). More important, however, interrupt-based monitoring is an intrusive technique that considerably distorts the instruction execution times. For parallel machines, this distortion inevitably affects the interaction of different processing nodes and can therefore change the observable behavior of the program. Gait [Gait86] reports that when sufficiently large distortions are present, programs that contain synchronization errors can appear to execute correctly. Therefore, intrusive monitoring techniques not only produce unreliable performance data, but may also affect the correctness of parallel programs.

2.1.3 Microprogramming-based monitoring

Agarwal et. al [Agarwal86] introduced a different monitoring technique that remedies some of the shortcomings of interrupt-based monitoring. Their technique relies on modification of a machine's microcode to capture address traces of executed instructions. Microcode modifications have a much smaller overhead than single-stepping interrupts, slowing down the execution of a program only by a factor of 10-20. Also, since monitoring is done at a level "below" the operating system, accurate tracing of user and system code is possible. However, this technique is still intrusive, although less so than interrupt-based monitoring. Its most important limitation is that it is only feasible on microprogrammed systems that provide control memories large enough to accomodate the required microcode changes. Most existing multiprocessors rely on one-chip microprocessors or RISC-based processors that either do not use microcode or contain their microcode in ROM.

2.1.4 Monitoring based on program augmentation

Programs can be modified by augmenting their code with instructions that gather interesting information "on-the-fly" during their execution [Eggers90, Stunkel89]. These modifications can be special monitoring statements that must be explicitly added by the user or can be inserted automatically by a preprocessor or compiler system. Software-based monitoring has many advantages. One of the most important is that the user has full control over the kind and frequency of captured information. While all previously discussed techniques capture low-level machine events, such as address traces, when monitoring is done in software, events can be recorded in terms of abstractions used by the programmer. This makes it easier to report results in terms the user will understand and provide better insight into the behavior of the program. However, software-monitoring usually cannot monitor the execution of operating system code, or account for the effect of multiprogramming on the execution of the monitored program. Also, this technique is still intrusive; the distortion of instruction execution times is dependent on the frequency and cost of monitoring code. If the recorded events are relatively infrequent, software-based monitoring can be reliably used for the evaluation of parallel programs.

2.2 Simulation techniques

Simulation allows the behavior of a target system to be emulated on another, possibly totally different machine. Simulation is a very attractive technique for the evaluation of parallel systems. It allows the behavior of a large number of different parallel architectures to be studied on the machine where the simulator runs. It also provides immense flexibility on the nature and amount of captured information. Finally, it has two properties that are of extreme importance when evaluating parallel systems: Nonintrusiveness and repeatability.

Nonintrusiveness means that any amount of monitoring information can be captured during a simulation without affecting the operation of the simulated system. The simulator assumes that all code executed to capture monitoring data is free; it simply does not take any time to execute on the simulated machine. As was discussed in Section 2.1.1, intrusive monitoring can cause distortions in the observed behavior of parallel programs. For that reason the nonintrusive monitoring offered by simulation is especially important.

Repeatability means that the same simulation will produce the exact same results across different runs. Therefore a problem identified during one run can be studied more thoroughly by repeating the same simulation and capturing more data related to the problem. Many real parallel machines on the other hand are nondeterministic: The same program may produce different results across different runs. Therefore an interesting phenomenon produced during one run may not be repeatable.

On the minus side, simulation techniques suffer from severe performance and accuracy considerations. Systems that simulate the behavior of a parallel machine in full detail typically exhibit slowdowns of several hundred or thousand times per simulated processor. Such slowdowns can limit the usefulness of simulation to the evaluation of only the simplest, and therefore less interesting, parallel programs. In order to achieve better performance, a simulation system must sacrifice some accuracy by not simulating some parts of the system, or by simulating them in less detail. Users rarely require full simulation accuracy so such tradeoffs, if made successfully, can produce useful and practical systems.

Finally, simulators are based on models of real systems. Before any data obtained from simulations can be trusted, the simulator itself must be thoroughly validated to ensure that the results it produces indeed do correspond to reality.

The following paragraphs describe some of the most important classes of multiprocessor simulation systems in more detail.

2.2.1 Cycle-by-cycle simulators

The most straightforward and accurate multiprocessor simulation method is cycle-by-cycle simulation. In a cycle-by-cycle simulation, the behavior of an entire machine is simulated for each clock cycle before advancing the global simulation clock to the next cycle. The simulator "sweeps" the entire machine at each iteration, simulating the behavior of each machine component for a single cycle. This method has the advantage of accuracy, since the timing of all events in the simulation closely models their timing in reality. However, the simulation overhead can be severe, as the simulation of even the simplest machine events may require hundreds of instructions.

An example of a cycle-by-cycle simulator is ASIM, developed at MIT by the Alewife group in order to validate the hardware design of a shared-memory multiprocessor [CHL90]. Although very accurate, ASIM slows down the execution of a simulated program by a factor of 200-5,000 for each processor [Nussbaum91]. Therefore, a program that would take one second on a real 64-processor machine may take more than 3 days on ASIM.

Cycle-by-cycle simulators provide detailed simulations of low-level machine events and are typically developed to evaluate new hardware designs. However, their performance is too low and their level of detail too high to allow them to be used for the evaluation of large parallel programs.

2.2.2 Trace-driven simulators

Trace-driven simulators are an important class of simulators that achieve reasonable performance by simulating only a part of the target machine. Trace-driven simulators are mainly used in studies of cache memory systems [Agarwal88, Eggers89]. In such studies, the behavior of the memory system can be fully simulated if a trace of the addresses produced during execution of a program is known. On uniprocessor systems such a trace can be captured from the direct execution of a program using one of the techniques discussed in Section 2.1. Since the trace need not contain any timing relationships, any machine can be used to produce it.

Given an address trace, the simulator need only implement an accurate timing model of the memory system of the target machine. More importantly, using the same trace, many different memory systems can be accurately simulated. Figure 2-1 depicts the typical structure of a (uniprocessor) trace-driven simulator, highlighting the decomposition of tasks between the trace generation and simulation components.

When attempting to extend trace-driven simulation techniques to multiprocessor sys-

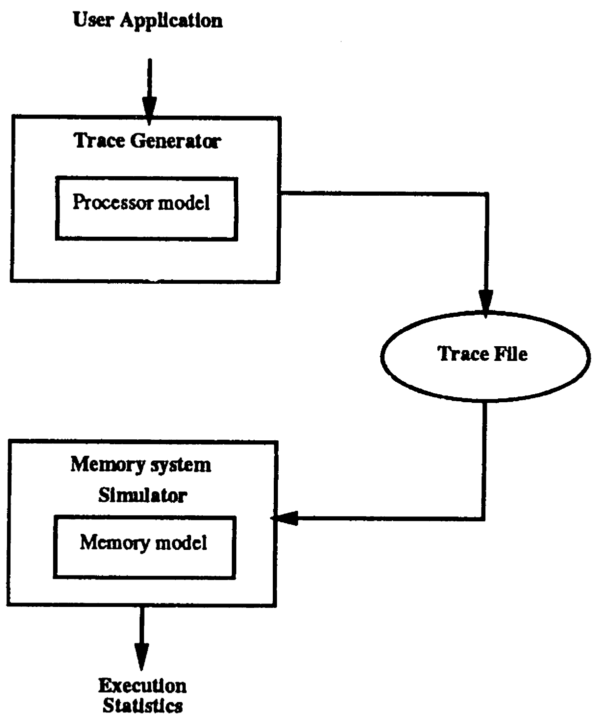


Figure 2-1: Typical structure of a uniprocessor trace-driven simulation system.

tems, several problems arise. In multiprocessor systems, each processor produces a separate address trace. Those traces are not independent; instead, they are closely interleaved in time as processors interact and communicate with each other using shared memory locations. Therefore the correct relative order of addresses contained in different traces becomes very important. As a simple example of this fact, consider a situation where the decision of processor 1 to access location *a* may depend on the contents of location *b*. Location *b* is written to by processor 2 only and serves as a signalling point between the two processors. Depending on whether a write to location *b* precedes or follows an attempt by processor 1 to read that location, an access to location *a* may or may not be issued. Therefore, a change in the relative timing of accesses by the two processors may produce different address traces.

In the uniprocessor case, address traces did not need any timing information. However, when simulating multiprocessors, processor traces must be accompanied by timestamps that will allow their correct interleaving. This, in turn, requires that the trace generator has the capability of at least partially simulating the memory model of the target architecture. The accuracy of the later memory system simulation will be limited by any inaccuracy in the timing assumptions made by the trace generator. The situation

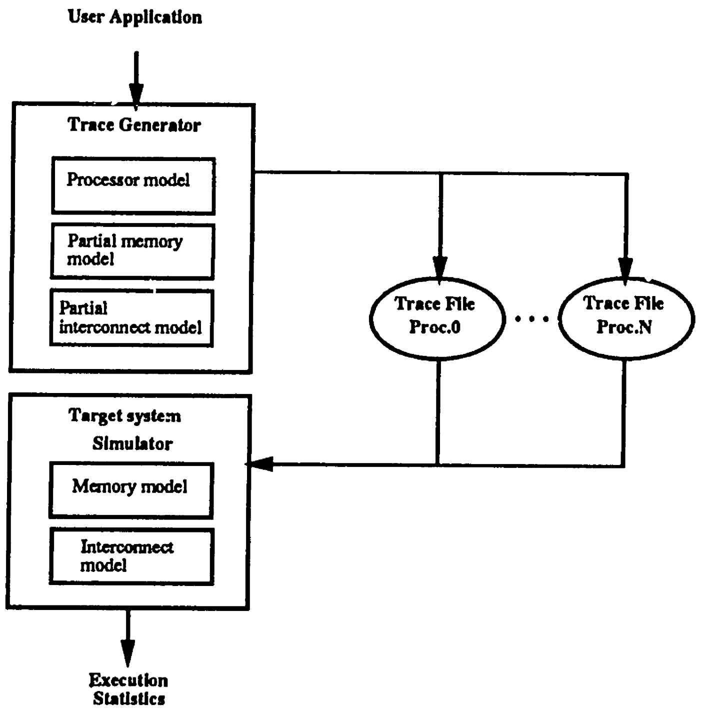


Figure 2-2: Typical structure of a multiprocessor trace-driven simulation system.

is depicted in Figure 2-2.

The special requirements of multiprocessors reduce the attractiveness of trace-driven simulation methods. In the uniprocessor case a single trace can be used to simulate several different systems. Multiprocessor traces contain timing information which is specific to the simulated system and cannot be reliably reused. Uniprocessor tracing has the advantage that traces can be produced from the direct execution of a program with relatively little extra effort. Therefore only simulation of the memory system is required. In the multiprocessor case, the trace generator must be able to simulate the target architecture in a fair amount of detail (Figure 2-2). Since the trace generator already contains a simple memory model, we argue that the separation of the simulation into a trace generation and a post-processing stage that produces execution statistics does not present any advantages any more. On the contrary, it requires the storage of large intermediate trace files, which may be a disadvantage in systems with limited disk capacity.

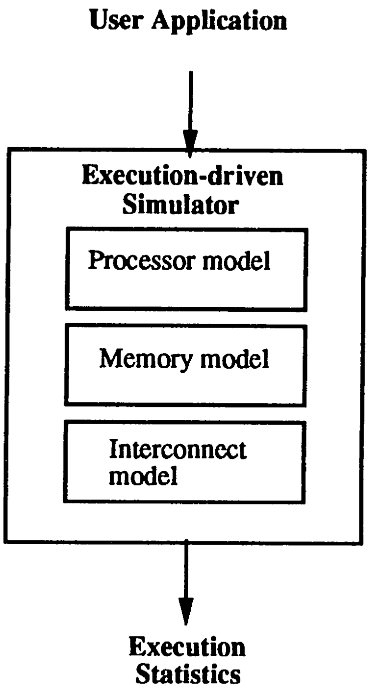


Figure 2-3: Typical structure of an execution-driven simulation system.

2.2.3 Execution-driven simulators

In the previous section we argued that the practice of separating the simulation of a parallel system into two phases, one that concentrates on the functional behavior of a program and one that focuses on the timing of execution on the target machine, does not make much sense for multiprocessors, where the execution path of a program, and thus its functional behavior, is highly dependent on the timing of different system events. In *execution-driven simulators*, also known as *program-driven* or *algorithm-driven simulators* [Covington88], the execution of a program is interleaved with the simulation of the target system architecture. Therefore, a simulation run directly produces both the program results and the execution statistics (Figure 2-3).

In trace-driven simulations the obtainable types of execution statistics and the accuracy of the simulation are limited by the types and accuracy of events contained in the intermediate trace file. Execution-driven simulators do not suffer from such limitations and therefore provide added flexibility and accuracy. They are also limited, of course, by the accuracy of the models they use to model the different parts of the target architecture.

The most important issue in the design of a successful execution-driven simulator is to balance accuracy and performance, in order to build a system that provides fairly reliable results in a reasonable amount of time. Many systems use a combination of simulation and direct execution to achieve that goal: They execute directly the local parts of parallel programs and simulate only nonlocal events, that is, events that affect more than one component of the target system. If such a combination is used, care must be taken to ensure both that the execution of local program parts is accurately timed and that the simulation of global events occurs in the correct order.

Proteus belongs to the class of execution-driven simulators. It takes advantage of a combination of simulation and direct execution to achieve better performance and uses a number of elaborate techniques to ensure that the resulting system does not substantially sacrifice simulation accuracy. The design of Proteus is explained and discussed in detail in Chapter 4.

2.2.4 Related work

In this section we will briefly present several multiprocessor simulation systems that present similarities with our own work. We will refer to them in Chapter 4, when we discuss design alternatives for different parts of our system.

The CARE simulator [Delagi87] provides an elaborate graphical interface for specifying different architectures and a programming model (LAMINA) based on LISP. The programmer writes a parallel program in LISP, which is then simulated by direct execution of the LISP code. Timing is done by use of a hardware microsecond timer. The use of a timer does not provide accurate timing of directly-executed instruction blocks. Performance of the system is relatively poor, due to the high overhead of interpreting LISP code. Even more important though, the use of a high-level LISP-based programming model and the decision to base timings on the actual execution time of code written in LISP can lead to erroneous conclusions. There are many hidden costs in the high-level semantics of LISP that are counted by the CARE simulator, even though they might not exist in an efficient compiled version of the same algorithm written in another language.

Aspen is a retargetable simulator with a programming model based on C [Elshoff91]. Aspen executes code directly and uses UNIX timing routines to time the execution of local instruction blocks. The problem with this approach is that it does not have adequate accuracy to simulate a fine-grain multiprocessor. UNIX timing routines have a granularity of one millisecond, which corresponds to several thousand instructions on a modern microprocessor. Fine-grain parallel machines execute local instruction blocks that can be as short as a few tens of instructions, and cannot be accurately timed using this approach.

Rizzo [Rizzo89] presents a retargetable parallel architecture simulator design that also uses a programming model based on C. As in our system, local instructions are executed directly and a library of calls supports kernel primitives that perform nonlocal operations. The main limitation of his system is the primitive way of counting execution time. The user has to explicitly specify the estimated time requirements of local blocks by use of a `takes_time` call. Kernel primitives are assigned fixed costs by the user. Since the only user-settable architectural parameters are the number of processors and the fixed kernel primitive costs, the flexibility of the system is limited. The system described by Dubois et al. [Dubois86] suffers from similar limitations.

In the Threads simulator [Mathieson88] programs are written in the Threads programming language and translated into machine code. Dynamic instruction counting is used to time local instruction blocks and special simulated kernel operations implement nonlocal interactions. Total execution time is separated into user and kernel time to isolate the costs of the run-time system. However, the machine model used supports only shared-memory multiprocessors and provides a very simple generic model for interconnection that cannot model real-life non-uniform access machines or the possibility of hot spots.

Tango [Davis90], developed at Stanford concurrently with our work, is the system that comes closest to ours in terms of accuracy and flexibility. It uses a combination of simulation and direct execution, times local instruction blocks using code augmentation, and can be configured to simulate several different architectures. The current implementation, however, does not accurately model operating system code in terms of low-level machine instructions. Also, Tango uses UNIX processes to implement the different executing threads with the result that simulation performance is significantly reduced.

Chapter 3

Proteus Overview

This chapter gives an overview of the Proteus system. Its purpose is to give the reader an idea of what the system looks like, and thus set the stage for the discussion of the system's design that follows in the next chapter. In addition, it intends to demonstrate that the combination, in Proteus, of a high-performance simulator and a flexible, graphics-based data display subsystem has resulted in a powerful tool which makes the experimental evaluation of parallel programs easy, fast and reliable. To that end, an example of a simple but interesting parallel program is presented. It is then explained how Proteus has helped in discovering the program's performance problems, identifying their causes and coming up with improvements that eliminated them.

This chapter does *not* intend to serve as a User Guide of the system and does not contain detailed information about the system's use. A detailed Proteus User Guide is available as a separate document.

3.1 Main components of the system

Proteus is composed of a number of programs and utilities. Most of them are not visible to the user but are called indirectly from other parts of the system. From a typical user's viewpoint, Proteus consists of three main components:

- `config`, a menu-driven program through which a user can set the parameters of the simulated machine configuration and compile his application into an executable simulator.

- `parsim`, which is the executable instance of a simulator, created by linking together the user application with the simulator engine and the components of the simulated machine architecture.

- **stats**, the data display program, which reads event files generated during a simulation, and displays execution statistics in a variety of graphical forms.

The most important "hidden" parts of Proteus are catoc, a C-preprocessor that translates our extended version of C into normal C when compiling user application files, and **augment**, a program that augments user program files with code that correctly times their execution. Finally, Proteus also has a special version of the C library and a number of simple script files that perform tasks such as installation of the system and compilation of a user application into a simulator.

Proteus is implemented and runs on DECstation workstations under UNIX. Both the configuration program and the data display subsystem use the X Windows interface standard to draw graphs and to allow mouse interaction.

3.2 Using Proteus

This section describes the steps a user needs to follow in order to create, run and evaluate a parallel application using Proteus. These steps are depicted in Figure 3-1.

3.2.1 Installing Proteus

Before writing an application, Proteus must be installed in the same directory where the source files of the application will be stored. A master copy of all Proteus system files is assumed to exist in a directory that is known and accessible throughout the host machine. A simple shell script is provided to install Proteus in an empty directory. During installation, symbolic links to all relevant Proteus system files are created in the target directory. Using those links, system files will be compiled and linked together with user files to create an executable simulator.

3.2.2 Writing a parallel program

Proteus provides a high-level language programming model to allow the writing of parallel programs. The model consists of a set of low-level simulator calls, a number of extensions to the C programming language, and a runtime system interface.

The local parts of a parallel program, that is, the operations that are performed locally on a processor and do not interact with other parts of the system, are written using normal

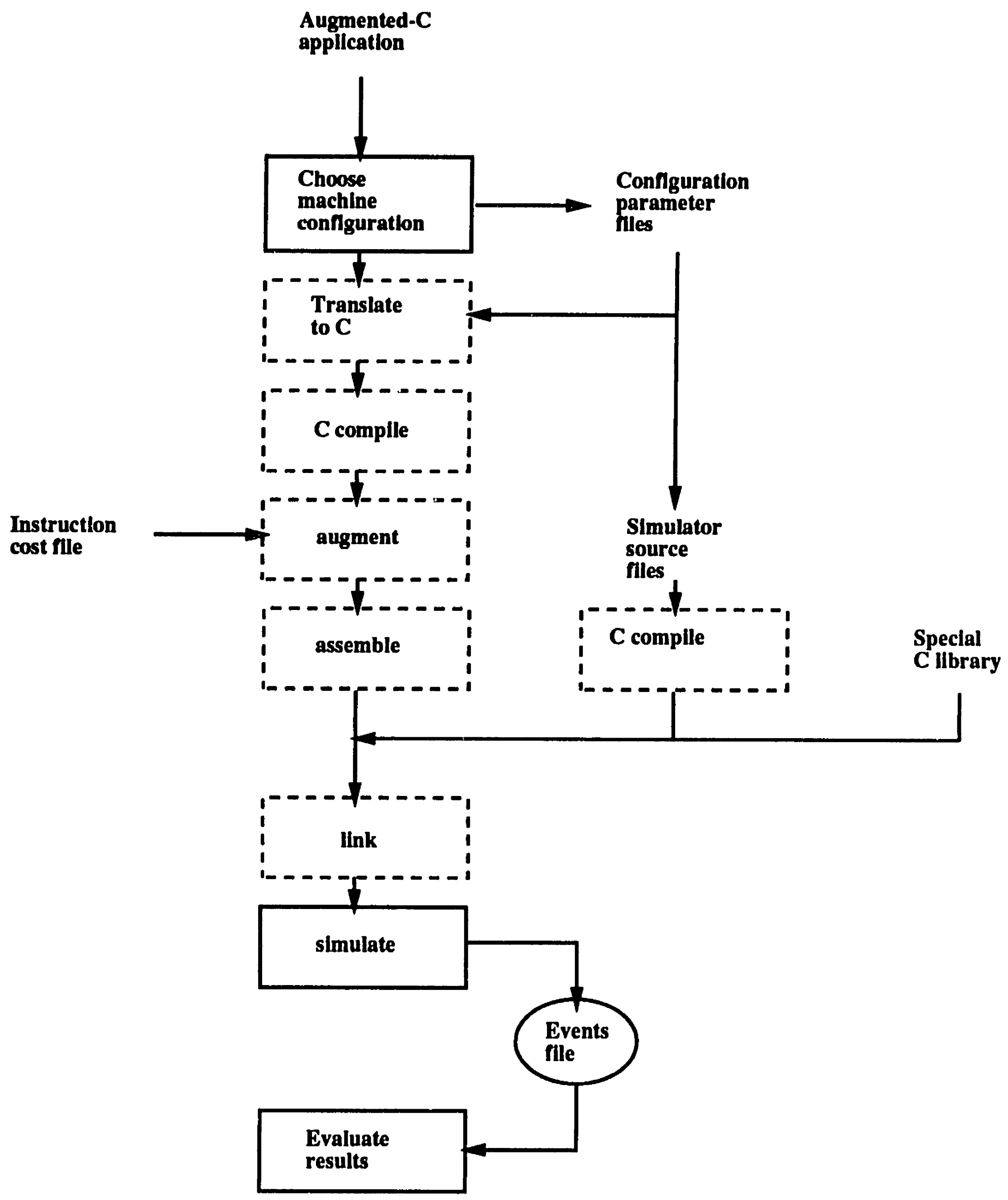
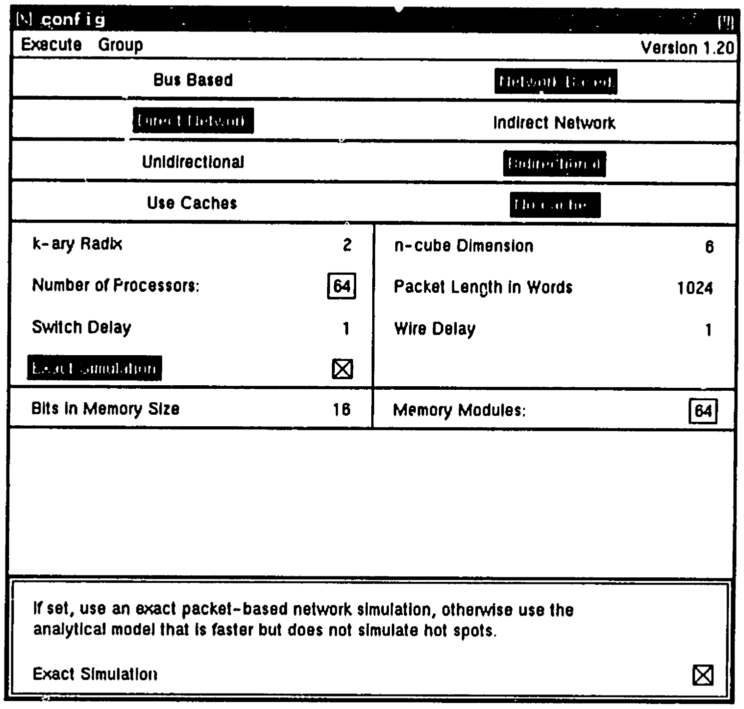


Figure 3-1: Creation, simulation and evaluation of a program using Proteus. Steps in dashed boxes are invisible to the user.



config

Execute Group Version 1.20

Bus Based	Network Based
Direct Network	Indirect Network
Unidirectional	Bidirectional
Use Caches	No Cache

k-ary Radix	2	n-cube Dimension	6
Number of Processors:	64	Packet Length in Words	1024
Switch Delay	1	Wire Delay	1
Exact Simulation	☒		
Bits in Memory Size	16	Memory Modules:	64

If set, use an exact packet-based network simulation, otherwise use the analytical model that is faster but does not simulate hot spots.

Exact Simulation ☒

Figure 3-2: Sample screen from the `config` program. Screens not shown control the simulator engine parameters, the set of events to collect, and the operating system parameters.

C statements. Operations that require interaction of multiple system components can be simulated using low-level simulator calls. We will use the term *nonlocal operations* to refer to those operations in this thesis. Low-level simulator calls are provided to support shared-memory accesses, sending of interprocessor interrupts, spinlock creation and use, and control of hardware features of the simulated processors.

C language extensions provide new storage classes and operators that support the declaration and use of shared memory variables using the normal C syntax. Language extensions are translated into low-level simulator calls during program compilation by a preprocessor (see below).

The runtime system interface provides a set of calls for thread and shared memory management. Using those calls, an application can create and control the behavior of a thread on a processor. It can also allocate and deallocate blocks of shared memory.

The programmer's interface to Proteus is described in more detail in Section 4.4.

3.2.3 Choosing the desired machine configuration

Proteus can be configured to simulate a wide range of shared-memory and message-passing MIMD machine architectures. In order to simulate a particular type of parallel machine, the user must specify several architectural and runtime system parameters to the system. These parameters are independent of the user application; the same user program can usually be simulated in a variety of architectures without any modifications. We provide a menu-driven tool (`config`) for controlling the parameters of a simulation, as well as building and executing the specified simulator. In addition, `config` allows users to add their own parameters. For example, when testing multiple variations of an algorithm, a user may add a parameter that chooses which one to include in the present simulation. Figure 3-2 gives a sample screen from the `config` program.

3.2.4 Compiling into an executable simulator

After specifying the required simulation parameters, the user can use the `Execute` menu of `config` to compile and link his application into an executable simulator. Alternatively, the `makesim` script can be used for the same purpose. During the compilation process, user files are compiled and linked together with simulator source files and configuration parameter files.

The compilation process consists of 5 steps, as depicted in Figure 3-1. User source files are first processed by `catoc`, our C preprocessor[1]. `Catoc` translates references to shared memory variables to the appropriate sequences of low-level simulator calls and produces normal C files as output. These files are then compiled by the standard C compiler and the equivalent assembler files are produced. In the next step, `augment` transforms the assembly code by inserting additional instructions that accurately track the amount of simulated time used by the code. The process of augmentation is explained more fully in Section 4.2.2. A standard assembler is used to convert the augmented code into object files. All simulator source files that are dependent on the configuration parameters modified by the user are recompiled. Finally, all user and simulator object files are linked together with a special version of the C library to produce an executable instance of the simulator. The executable simulator is called `parsim`.

[1]The obscure name means ".ca to .c", since the preprocessor transforms an augmented-C file, which has filename extension .ca, into an equivalent C file with filename extension .c

3.2.5 Running a simulation

An executable simulator created by the previous steps can be run as a normal UNIX process by simply giving its name. During its execution, `parsim` simulates the behavior of the user program on the selected machine configuration and generates a file of execution statistics that can be read and interpreted by the data display subsystem.

The simulator can be interrupted at any time during its execution by pressing the keyboard interrupt (CTRL-C) key. When simulation is interrupted, the user is transferred to the *snapshot* component of the simulator. The snapshot mode provides a simple menu that allows the user to examine in detail the state of the parallel machine and the simulator engine at the interrupt point. The snapshot menu also provides features, such as single stepping and the ability to set breakpoints, that help the difficult task of debugging parallel programs.

3.2.6 Displaying and evaluating simulation data

The primary use of a simulation is to provide insight that helps the user resolve specific questions about an algorithm or implementation. During a simulation, our system generates a file of interesting execution events and statistics. This file is then interpreted by `stats`, our data display program.

Our data collection and display system has two major goals: first, collect exactly the right data needed to answer the user's questions, and second, display the data in the form that best answers these questions.

One of the best ways for humans to interpret large amounts of data is through the use of color graphs. Thus, color graphs are the output of our data display subsystem. For a given set of data, there are many useful graphs, hence, users should be able to view their data in a variety of ways. The system provides a set of predefined graphs, some of which are listed in Table 3.1. In addition, the user can define new graphs via a simple but powerful graph language.

The system currently classifies data into two groups: time-independent and time-dependent. Time-independent data summarizes entire simulations; time-dependent data reflects the state of a simulation at a specific time. For example, the overall cache hit ratio for a simulation is a single number that is independent of time. If the ratio is recorded over time, each point in the record is time-dependent, but the mean of the points is independent of time.

Time-independent data is collected using *metrics.* Users specify the name and value

- Concurrency graph (busy processors versus time)
- Active threads (per system and per processor) versus time
- Threads waiting on a system lock versus time
- Processor lifelines
- Cache hit ratio over time
- Bus and Network contention over time
- Total number of calls to each program function and avg. number of cycles per call
- Total time spent in user and runtime system code

Table 3.1: Some predefined graphs and execution statistics

of a metric at run time. After the simulation, the metrics can be referred to by name. Users may also specify an array of metrics, which is used to collect the same data for an array of objects, such as processors or semaphores. For example, a useful array metric is the utilization of each processor. There is one name for an entire array; users reference elements by name and index.

Time-dependent data is collected using *events*. Events have types that allow all of the events of one type to be handled as a unit. For example, one type is processor usage; events of this type record the times at which processors become idle or busy. This kind of event can be used to generate a graph of the number of busy processors versus time, commonly known as a concurrency graph. Furthermore, users may define their own event types and may specify an array of events analogous to an array of metrics.

To help control the volume of data, events may be filtered as they are produced. For example, cache information is filtered to reduce the data output by a factor of one hundred. This is done by counting the number of cache hits in one hundred accesses and using this count as the value of the event. The general filtering mechanism outputs one value for every k updates. The user recalculates the event value at every update using the previous value, the update information, and any state the user wishes to keep.

The user specifies the display of data using a general graph language. The display program, **stats**, reads a file that specifies the graphs desired by the user. The user switches among these graphs using a menu. Facilities are provided for zooming in on portions of graphs, calculating the area and average value of a graph, producing a hardcopy[2], and switching between graph and table format.

The graph language currently supports six kinds of graphs: metrics and array metrics graphs, events-versus-time graphs, bar graphs, array graphs, and multi-simulation graphs. Metrics-based graphs display the values of a set or array of metrics in either a bar-graph or tabular format. Event-based graphs display a function of a set of events versus time. Array graphs consist of an array of event-versus-time graphs, and thus contain three dimensions. The y-axis is the array index, the x-axis represents time, and the color of the point indicates its value. Array graphs are particularly useful for quickly verifying behavior and for viewing the simulation as a whole. Multi-simulation graphs display one point for each member of a set of simulations; speedup graphs fit this form. Finally, groups of event or multi-simulation graphs can be displayed on the same axis, with each member of the group in a distinct color. Among the graphs of Section 3.4 below, Figure 3-6 is an example of an event graph, Figure 3-10 is an example of an array graph, while the speedup graphs (e.g. Figure 3-5) are examples of groups of multi-simulation graphs.

3.3 Typical uses of the system

Proteus is used to support research in several areas of parallel processing. It was originally conceived as a tool for the evaluation of a new parallel programming language. However, the system provides enough power and flexibility to be used in many other areas.

3.3.1 Evaluation of parallel applications

The performance of a program on a parallel machine depends on many factors that are not considered in a theoretical complexity analysis of the program and usually defy satisfactory analytical modelling. As we will observe in the next section, the interaction of a program with the runtime system, the method of assigning new threads to processors, and the details of the architecture can have a profound effect in the observed program

[2]The graphs in this thesis were created directly from **stats**, using the **Hardcopy** menu command, combined the command-line flag **-latex**.

performance. Proteus accurately models factors such as those mentioned above, and allows users to study the behavior of applications running on a variety of machine models. It can, therefore, provide a useful complement to the theoretical analysis of new parallel algorithms. Furthermore, the efficiency of the simulator component makes it feasible to use the system to predict the performance of real applications on different architectures.

3.3.2 Research in parallel programming languages

The C language, augmented by the low-level simulator call interface of Proteus, can be used as the target language of parallel language compilers. In that way, the system can be used to evaluate the performance of high-level parallel programming languages in a variety of different machine architectures.

3.3.3 Research in parallel operating systems

Proteus provides a complete runtime system which is built on top of its low-level simulator call interface and is viewed by the simulator as an application program. Time spent in runtime system code is accurately measured and, if desired, separated from time spent in user code. Detailed information about threads waiting in various system locks and queues can be plotted using the `stats` program. Therefore, the effects of the runtime system on user program performance can be observed. Runtime system routines can be partly or completely rewritten. In that way, researchers can test new operating and runtime system designs.

3.3.4 Research in architecture-related issues

Proteus provides support for research in two architecture-related areas: Cache coherence protocols and network routing algorithms.

The appearance of large-scale shared memory machines has renewed interest in research on cache coherence protocols. The shared memory component of Proteus supports the detailed simulation of cache coherence protocols. A well-defined interface of calls connects the protocol with the rest of the system (see Section 4.3.2). Researchers in that area can implement their own protocols and then use Proteus to evaluate their performance. Proteus provides a much easier and more reliable way of performing such evaluation than the currently popular method of trace-driven simulations (see Section 2.2.2).

Proteus supports research in network routing algorithms by providing an accurate

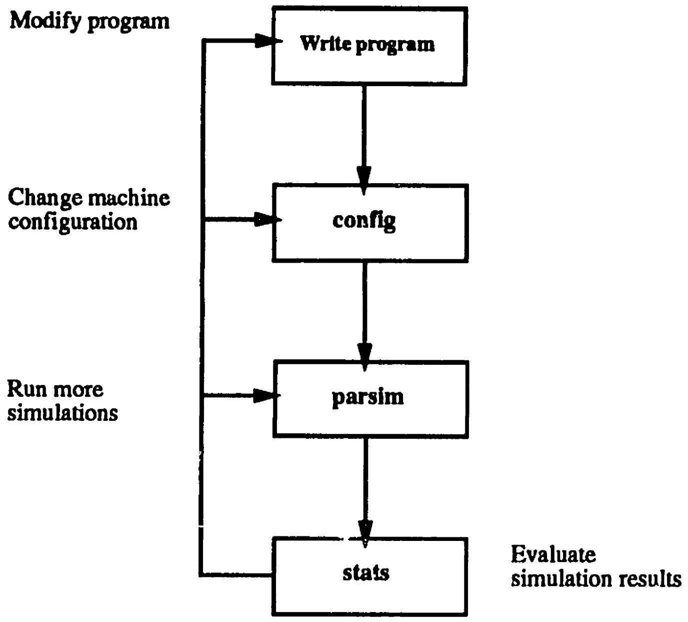


Figure 3-3: Typical parallel program development and evaluation cycle using Proteus.

network simulation component and a flexible router interface (see Section 4.3.3). The user can replace the routing algorithm with his own, and the behavior of the router during simulation of real applications can be easily monitored.

3.4 Exploring parallelism with Proteus

The flexibility and interactive nature of Proteus encourage an exploratory approach to parallel program evaluation: The user typically runs one or several simulations of an application and subsequently enters the `stats` program to evaluate their results. With the help of a number of carefully chosen predefined system graphs, as well as his own program-specific graphs, the user can study several aspects of the program's behavior and discover any areas of trouble. Once a problem has been detected, more simulations may be needed to identify its cause. The program may need to be modified, different types of events may need to be added, or different machine configurations may be tried to provide further insight into the program's behavior. At the end of every stage of experiments, `stats` is called to display the results of the latest simulations and help decide what needs to be done next. This iterative interaction among the system's components is shown in Figure 3-3.

In this section we aim to demonstrate the power of the Proteus system by studying

```
proc solve_queens() :
    stage( empty_chessboard , 0 );

proc stage( chessboard, i ) :
  if ( i == 8 )
      found_solution( chessboard );
  else
      for( j = 1 ; j <= 8 ; j++ )
         if ( safe_to_add_queen( chessboard, i+1, j ) )
            stage( add_new_queen( chessboard, i+1, j ), i+1 );
```

Figure 3-4: A recursive algorithm that generates all solutions of the 8-queens problem.

a simple but interesting program that implements a parallel algorithm for solving the 8-queens problem. Our program suffers from a common problem in parallel programming: Although the algorithm on which it is based contains a large amount of parallelism, its implementation on a real machine produced a disappointing speedup curve. By using Proteus in a systematic way, all different causes of the program's poor scaling behavior were easily identified and, whenever possible, corrected. Apart from proving the capabilities of the Proteus system, this example also demonstrates the complexity of the interaction between the hardware, the runtime system and even the simplest parallel program, and emphasizes the need for advanced parallel program measurement tools such as Proteus.

3.4.1 The eight queens problem

The eight queens problem is stated as follows: Eight queens are to be placed on a chessboard in such a way that no queen checks against any other queen. The problem has exactly 92 solutions and one way to solve it is by a recursive algorithm described by the pseudocode shown in Figure 3-4.

`Stage` receives a chessboard in which a queen has already been placed in each of the first i rows and attempts to place an additional queen in all columns j of row $i+1$ where the new queen does not check against any of the previously placed i queens. Each successful new configuration is recursively passed back to `stage`, which now attempts to place a queen in all safe columns of row $i+2$, and so on, until either no additional queen

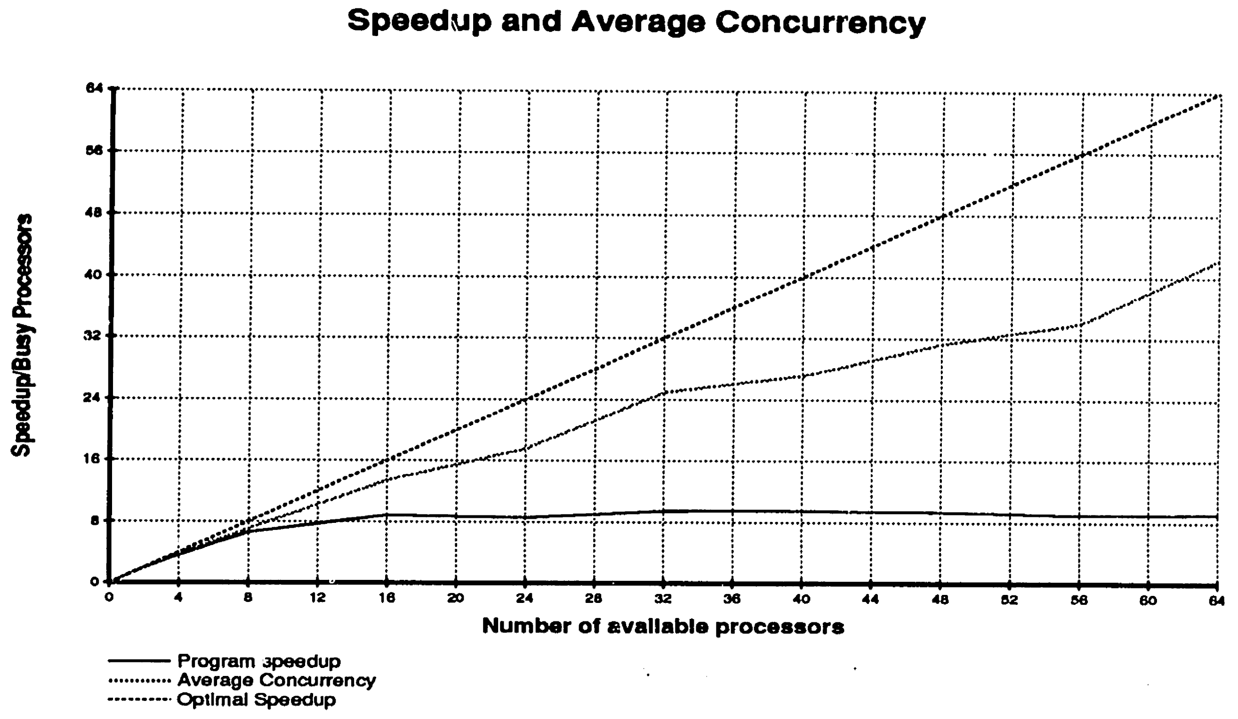


Figure 3-5: Speedup and average concurrency graphs for the 8-queens program with a single memory list.

can be safely placed in any of the columns of a given row, or a complete solution has been found. The algorithm essentially performs a depth-first traversal of the solution space and abandons further traversal of any subtree whose root is an illegal problem configuration.

It is easy to parallelize the above algorithm by transforming all calls to `stage` into invocations of new tasks that will execute `stage` on different processors from the calling tasks. This transformation causes all traversals of subtrees with a common parent to execute in parallel. Since the 8-queens solution tree has a maximum branching factor of 8, and very quickly expands to a very large number of subtrees, it is expected that the parallel version of the algorithm will exhibit a significant amount of concurrency.

3.4.2 Evaluation of the eight queens program

We implemented a version of the parallel 8-queens solution program using the programming model of our simulator. The code is listed in Appendix A and gives an idea of what a typical Proteus program looks like. Different chessboard configurations are represented using data structures stored in shared memory. `Add_new_queen` dynamically allocates an

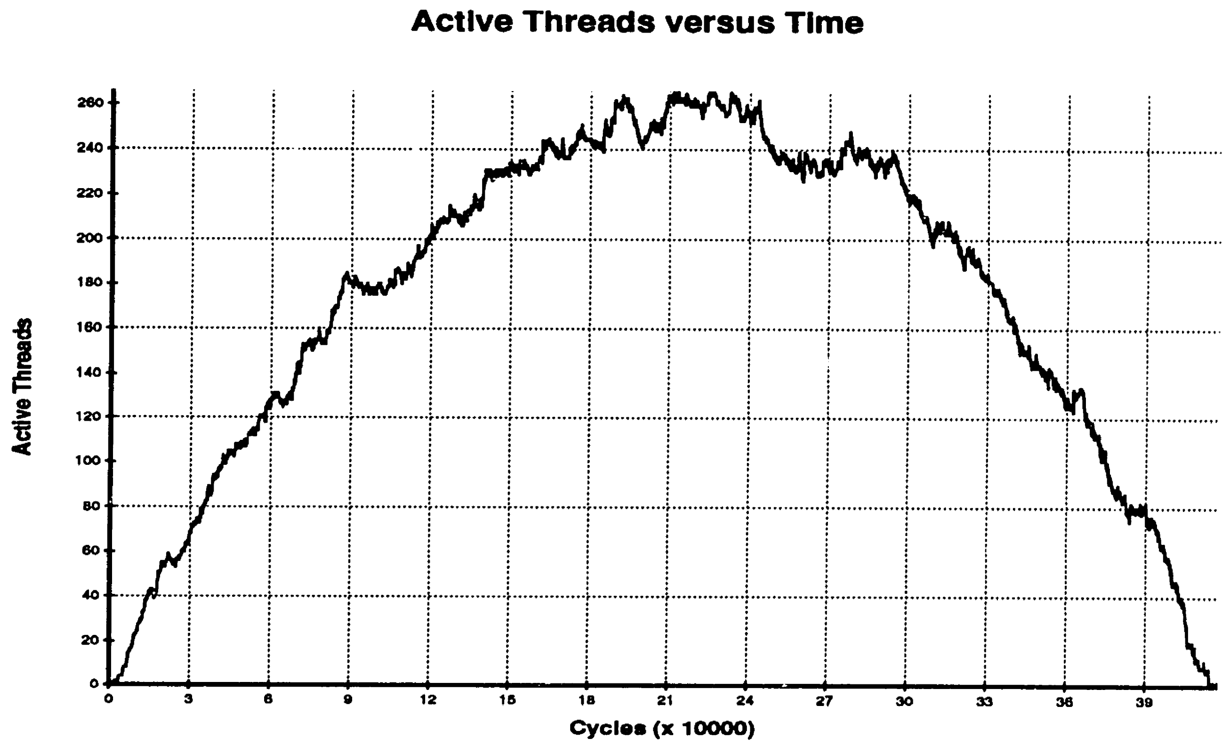


Figure 3-6: Number of active threads versus time during execution of the 8-queens program.

amount of shared memory to store a new configuration with one extra queen, and returns the shared memory address of the new configuration. Each new task then receives this new configuration and attempts to create new configurations with yet another queen placed. We can see therefore that the program makes relatively intensive use of shared memory.

We executed a number of simulations of the program on bus-based machines of different sizes. In all machines, a number of processors and a number of shared memory modules were connected together through a common bus. All shared memory accesses had to pass through the common bus. Each processor was equipped with a 64K 2-way associative cache to improve average memory latency and reduce traffic on the bus. From the results of our simulations, we produced a program speedup and average concurrency graph (Figure 3-5). Program speedup is defined as the ratio of the program execution time on one processor to the program execution time on n processors. Average concurrency gives the average number of processors that were busy during execution of the program on a given machine configuration.

The speedup graph of Figure 3-5 is disappointing. Although we expected the program to exhibit a large amount of concurrency, and therefore to scale very well on large parallel

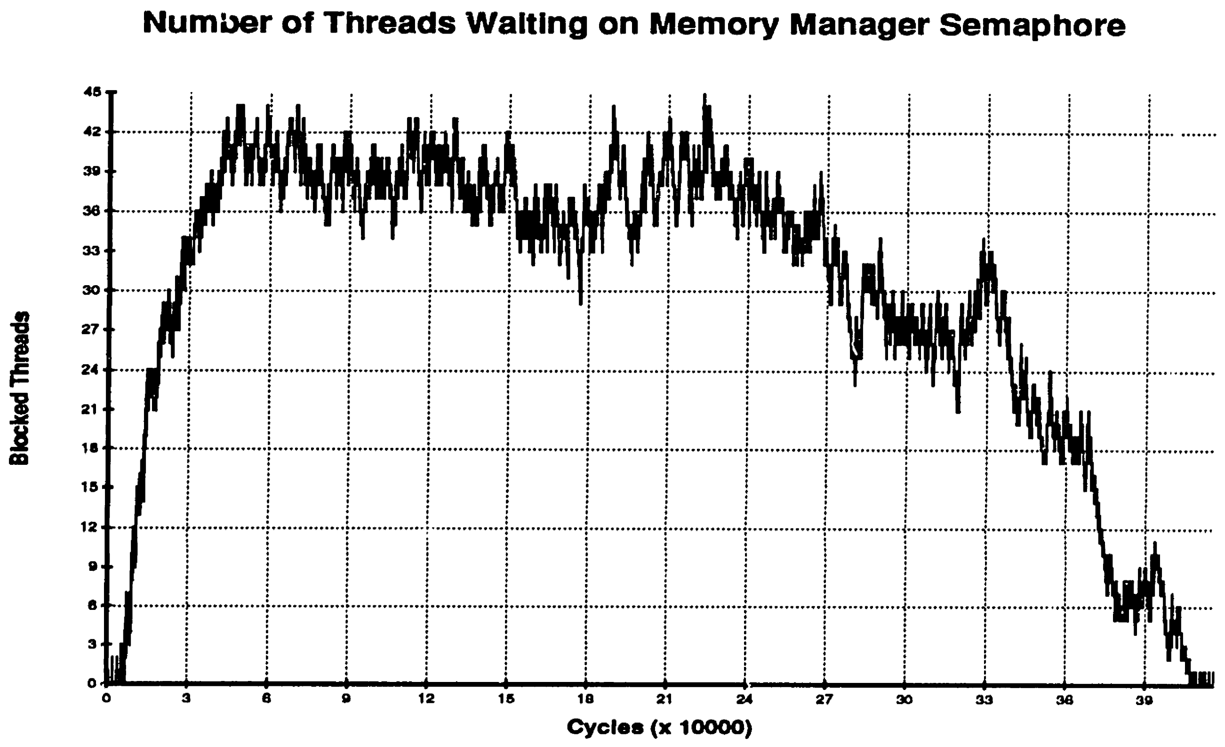


Figure 3-7: Number of threads spinning in the memory manager semaphore during execution of the 8-queens program (processors=64).

machines, speedup does not seem to be able to increase above 9 even on machines with 64 processors.

There are several factors that limit the performance of parallel programs. Some of the most common are:

- The underlying algorithm does not have enough concurrency.
- The implementation does not make good use of available resources.
- There is a bottleneck in the interaction between the program and the runtime system.
- There is a bottleneck in the interaction between the program and the hardware.

In the case of our program, we can easily rule out the first listed possible cause. Figure 3-6 gives the number of active threads versus time during the execution of the 8-queens program on a 64-processor machine. At almost all times this number is more than the number of available processors and peaks at about 260. As we expected, the algorithm has enough concurrency to keep many processors busy!

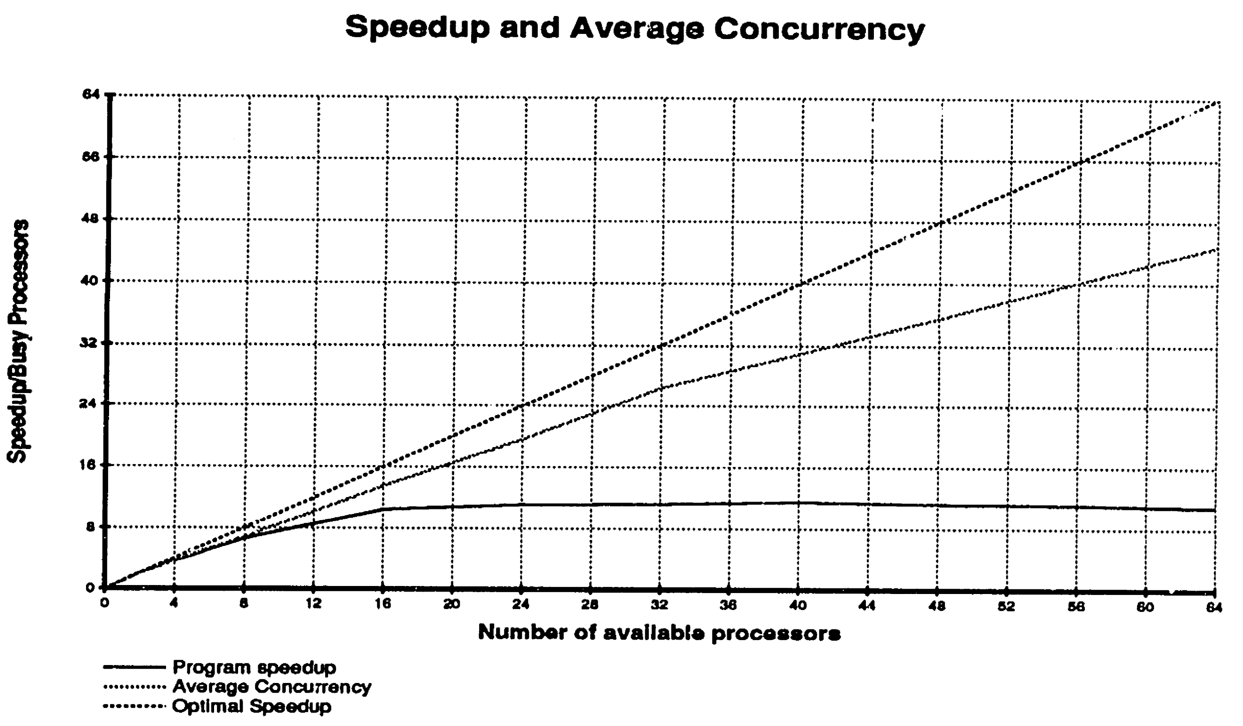


Figure 3-8: Speedup and average concurrency graphs for the 8-queens program with multiple memory lists and random thread placement.

We next investigated the interaction of the program with the runtime system. Problems in this area usually arise from shared runtime system data structures that are guarded by mutual exclusion semaphores. Such data structures can easily become bottlenecks. `Stats` can produce graphs that depict the number of threads that are blocked in any system semaphore versus time. By examining those graphs we were able to discover that the memory manager lock was in fact a serious bottleneck (Figure 3-7). It turned out that the current implementation of the dynamic shared memory allocator keeps a single free list guarded by a mutual exclusion semaphore. With several processors trying to allocate and deallocate shared memory blocks at the same time, this can indeed become a bottleneck.

We replaced the memory allocator by a different algorithm that keeps one free list per processor. We also modified our program to ensure that each processor would always allocate memory blocks from its own free list. In that way, contention for the memory manager was eliminated.

We executed a number of simulatons using the modified program and produced a new speedup and concurrency graph which appears in Figure 3-8. Although speedup has improved, it was still very low and had a peak value of about 11. Some other factor was

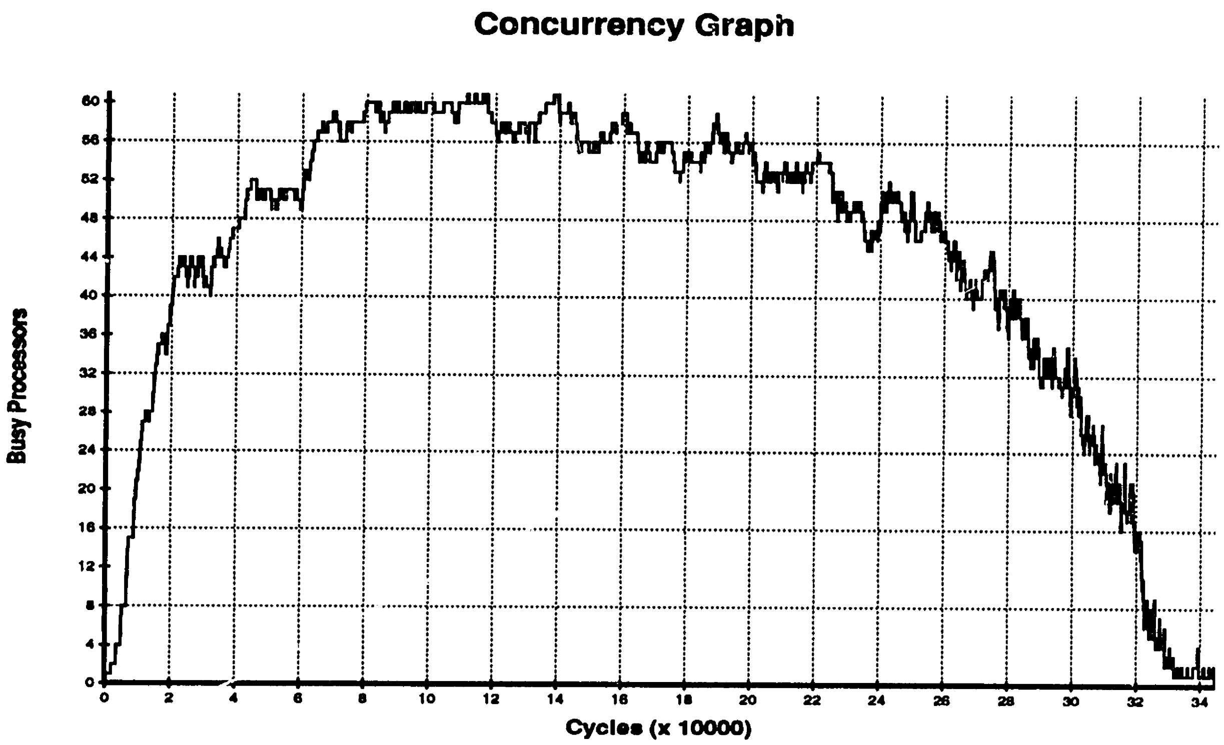


Figure 3-9: Busy processors versus time for the 8-queens program with random thread placement (processors=64).

limiting the performance of our program.

By taking a closer look at the average concurrency graph of Figure 3-8 we observed that that too was not scaling particularly well as the number of processors increased. Figure 3-9 shows the concurrency graph (number of busy processors versus time) for the 64-processor case. We observed that at all times there were some idle processors in the system. Given that the number of active threads was many times the number of active processors, this discrepancy suggested that our program did not make good use of the available processors.

A careful look at the lifeline graph of Figure 3-10 helped identify the problem. All processors were utilized at some point during the program, but many processors were having idle periods during which they were waiting to receive some work. This observation prompted us to take a closer look at the method of assigning new threads to processors in our program.

Placement of threads in the original version of our program was done by choosing a target processor at random. Very often, the chosen target processor would be busy executing some other thread, while at the same time other processors would be sitting idle. Our naive placement method was apparently leading to less than optimal machine

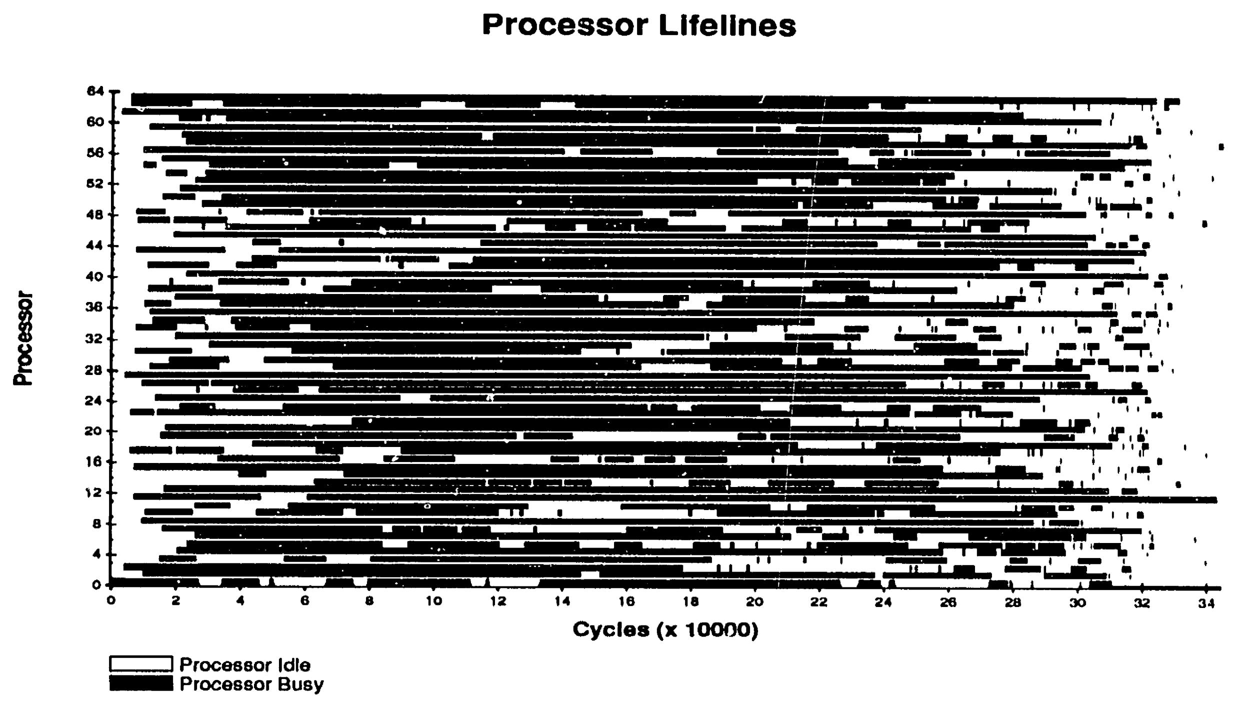


Figure 3-10: Processor lifelines for the 8-queens program with random thread placement (processors=64).

utilization.

We modified our program to place threads using a more sophisticated placement algorithm: The program kept a circular queue of idle processors in shared memory. Whenever a processor became idle, it inserted its index at the tail of the queue. Whenever a processor was ready to create a new thread, it removed a processor index from the head of the queue and created the new thread on that processor. If the queue was empty, a target processor was chosen at random. The head and tail pointers of the queue were read and incremented simultaneously, using atomic shared memory operations (fetch-and-add). Therefore, multiple threads could access the queue at the same time without the need for a mutual exclusion semaphore that would make the queue a bottleneck.

The new placement algorithm substantially improved processor utilization. Figure 3-11 shows processor utilization on a 64-processor machine using the new algorithm. The reader should compare it with Figure 3-9, which depicts processor utilization with random placement.

Unfortunately, although average concurrency increased by almost 25% in the 64-processor case, even with the new placement algorithm there was only a marginal improvement in speedup (Figure 3-12). At first this looked like a paradox because now more

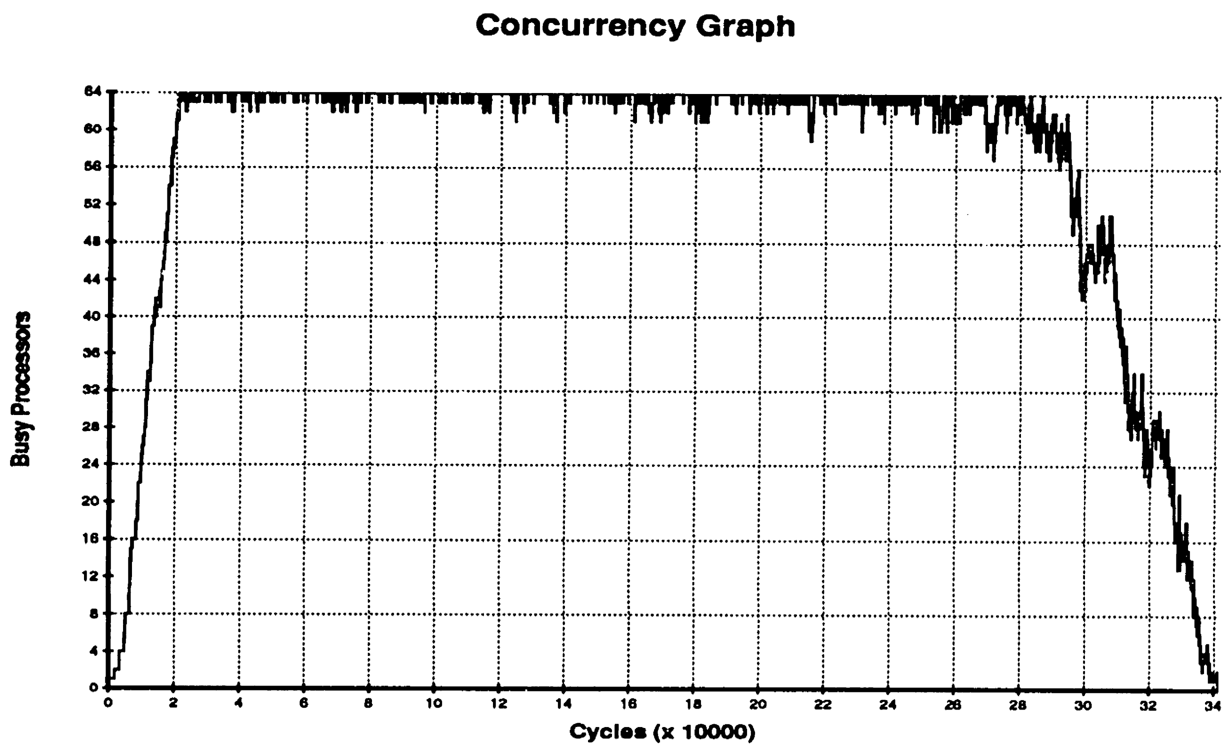


Figure 3-11: Busy processors versus time for the 8-queens program with improved thread placement (processors=64).

processors were busy doing useful work and therefore the program should have better performance.

Trying to identify the cause of our performance problems, we finally examined the interaction of the program with the hardware. It is well known that, in bus-based machines, the common bus very quickly becomes a performance bottleneck and, in fact, that was the main cause for the poor scaling behavior of our program[3]. Figure 3-13, which depicts the bus contention latency per shared memory access for the 64-processor case, is ample proof of that fact. As long as all shared memory requests have to go through the common bus, no program improvement can significantly improve the performance of the system. In fact, the better the machine utilization, the larger the number of processors that compete for the common bus. Thus, any performance gains obtained by an increase in concurrency are offset by a corresponding increase in delays due to bus contention. Caching reduces the number of memory accesses that have to go through the bus, but the remaining number of accesses is still high enough to completely saturate the bus.

[3]This fact was, of course, obvious to us from the start, but we left it for the end in order to make the discussion of all the previous performance issues more interesting.

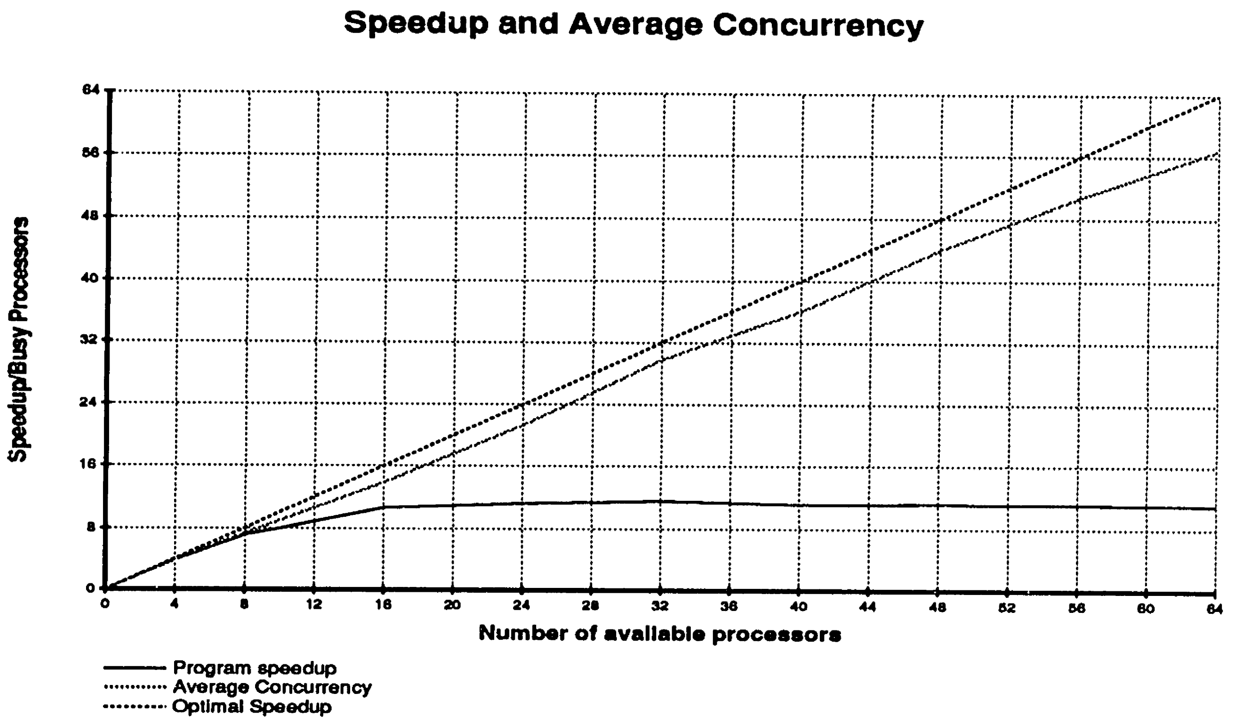


Figure 3-12: Speedup and average concurrency graphs for the 8-queens program with multiple memory lists and improved thread placement.

Figure 3-14 shows the average bus contention latency for different machine sizes. It can be observed that this latency grows faster in the case of the improved placement algorithm. This observation explains the apparent paradox of Figure 3-12 where an increase in machine utilization had almost no effect on speedup.

So how can we increase the performance of our program without rewriting it completely? Since we are working on a simulator we simply changed the machine configuration to a network-based machine. Network-based machines exhibit much better scaling behavior since they do not restrict all nonlocal requests to pass through the same exclusive access medium. Figure 3-15 shows the speedup curve of the 8-queens program on hypercubes of dimensions 1 to 6.

We observed that the program now exhibits a much better scaling behavior. Speedup remained very close to average concurrency (average number of busy processors) even for large machine sizes. However, average concurrency was lower in the case of hypercubes than it was in bus-based machines (see for comparison Figure 3-12).

By studying the concurrency graph for several sizes of hypercubes, we were able to determine that the lower average concurrency was due to the larger amount of time required to create the first few threads that are needed to fully utilize a hypercube machine.

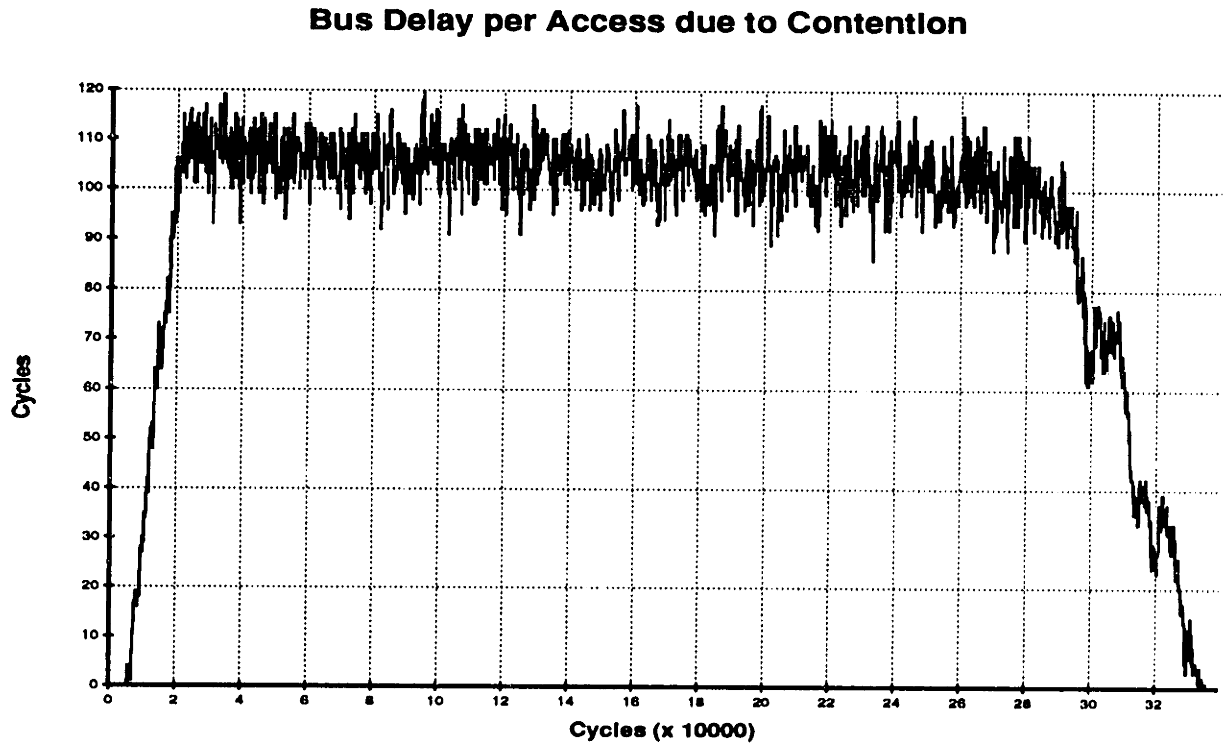


Figure 3-13: Bus latency due to contention versus time for the improved 8-queens program (processors=64).

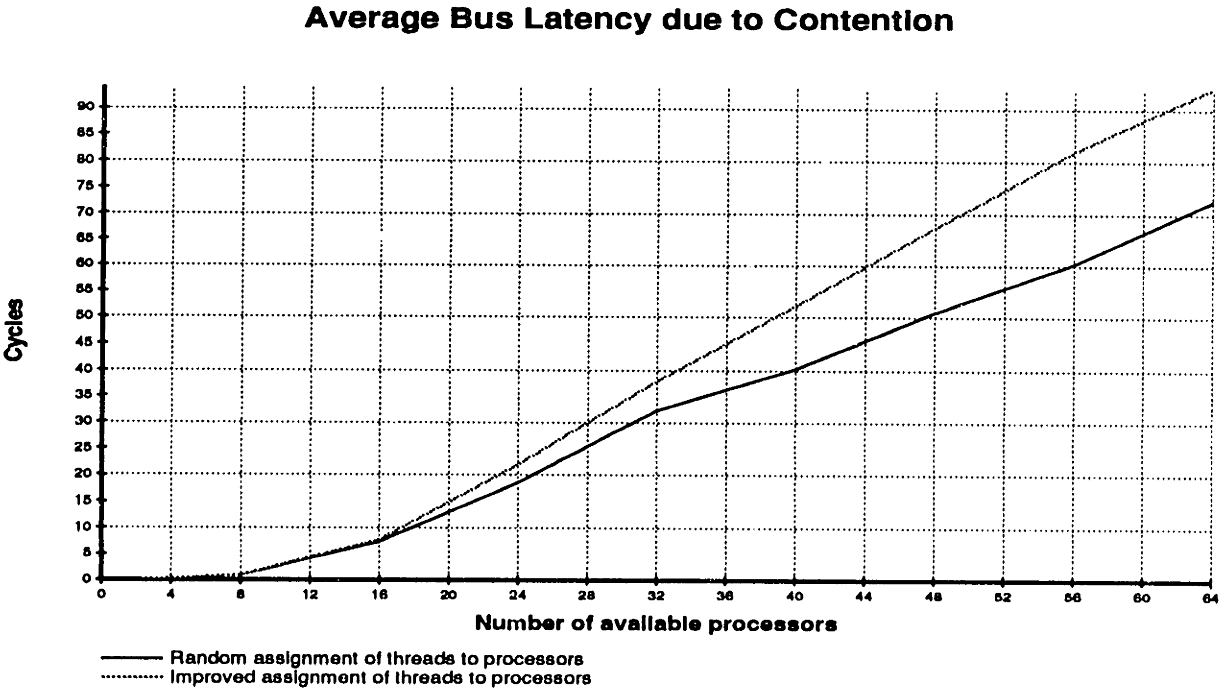


Figure 3-14: Average bus contention latencies for various machine sizes.

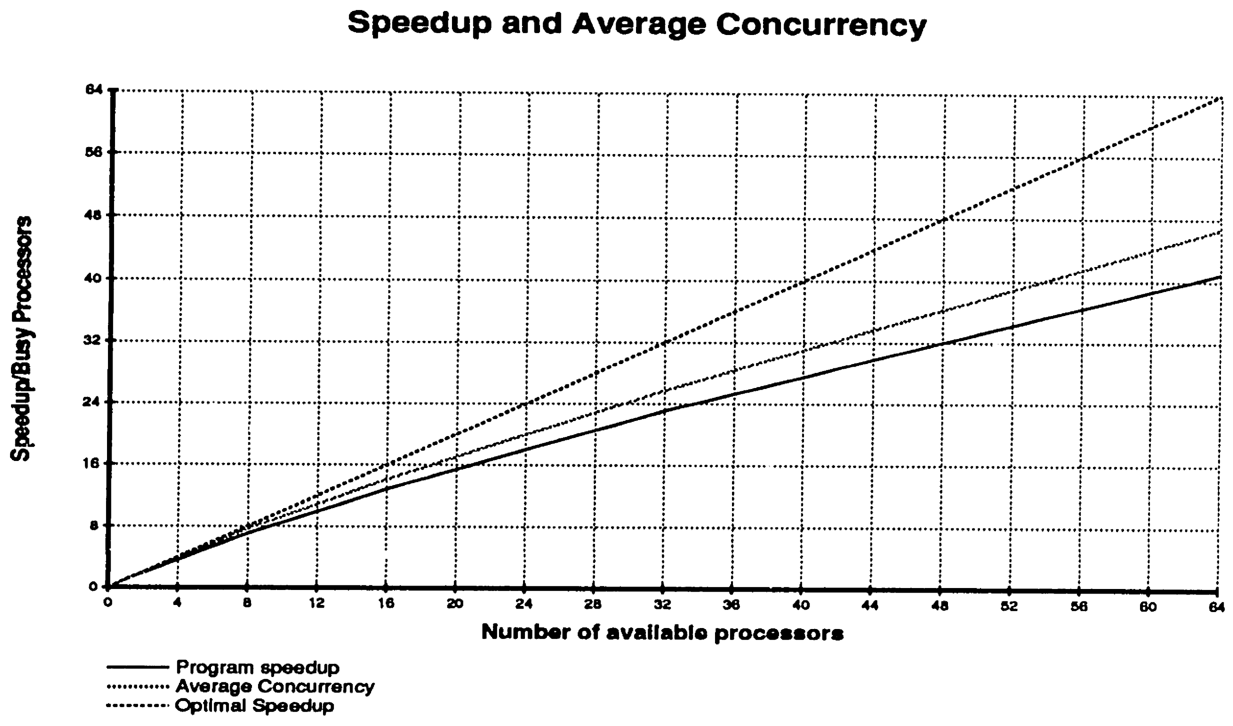


Figure 3-15: Speedup and average concurrency graphs for the 8-queens program on hypercube machines.

By comparing hypercube concurrency graphs with Figure 3-11, which shows concurrency versus time for a 64-processor bus-based machine, we found that the first, rising part of the graph was less steep in hypercube machines. This phenomenon can be explained by observing that, before it creates a new thread, `stage` first calls `add_new_queen` to create a new chessboard configuration. `Add_new_queen` allocates a block of shared memory to store the new configuration, copies the old configuration to the new block, and places an additional queen at the new configuration. The old configuration is usually stored on a different node from the new one. On a network machine, the longer memory latencies when accessing the remotely stored old configuration are responsible for longer delays between creation of new threads. This explains the longer start-up times until the machine reaches full utilization. This delay, in turn, has the effect of reducing the average concurrency over the entire execution of the program. One way to fix this problem and improve the efficiency of our program is to rewrite it so as to store chessboard configurations in processor local memory and transmit entire configurations to other processors using interprocessor interrupts.

Chapter 4

System Design

The design of any complex computer system is an exercise in balancing several, often conflicting, user requirements with the available computer resources. In this chapter, we shall describe the design of the Proteus system, concentrating on the parts of the system for which the author was primarily responsible: simulation strategy, simulation of architectural components, programming model and runtime system interface. We shall also briefly discuss the general requirements for the entire system and the overall design decisions that gave Proteus its present form.

For each system part, we shall start by describing our requirements and goals. Following that, we shall discuss the alternative design paths we considered and shall explain why we chose the design we adopted in the system.

4.1 Design goals

This section outlines the requirements and describes the overall design decisions that defined the general shape and scope of the system.

Our goal was to create a simulation-based system that would allow us to test the efficiency of portable system software for MIMD architectures. Since we were interested in writing portable programs, we did not require detailed simulations of specific parallel machines. We did require, however, a system that could capture all essential interactions of a parallel program with the other components of a MIMD machine. Such interactions can generally be classified into the following categories:

- Interactions with the runtime system (bottlenecks caused by exclusive access system data structures, effects of scheduling algorithm, effects of placement policy, overhead of different runtime system functions, etc.)

- Interactions with the hardware (type and properties of interconnection medium, effects of caching and cache coherence protocols, effects of network routing algorithm, efficiency of available synchronization mechanisms, etc.)

We decided to build a *machine-level simulator* to satisfy the above requirements. Our simulator would simulate the effects of individual machine instructions on the different parts of the system and would include detailed simulation of the major interactions that affect the performance of a program on a parallel machine. Since we were not interested in simulating actual machines, it would only be necessary to simulate the *effects* of machine instructions, not their low-level implementation in terms of registers, data paths, etc.

Our system should enable us to compare the performance of our portable programs on different architectures. Hence, our simulator would be designed from the start to be *retargetable* and able to simulate a wide range of different MIMD machines. Ideally, configuration of the simulator would not require any modifications of the user code.

We recognized very early that the purpose of simulation is to provide information that helps the user evaluate his implementation and choose between different design alternatives: A simulation is only as good as the insight it provides. The complexity of the interaction between different components of a parallel system means that simulations of even the simplest program will produce a huge amount of potentially interesting information. We decided that our system would provide a dedicated data collection and display component that would help users collect exactly the right data needed for their experiments and display data collected during a simulation in a variety of graphical forms. Section 3.2.6 contains an overview of the data collection and display component of Proteus. For a full description, the reader is referred to Eric Brewer's thesis [Brewer91].

Finally, we realized that our system would be used both for the evaluation of existing programs and for the development of new ones. Therefore, we decided to make it easy to use in an interactive manner. Configuration of simulation parameters, generation and execution of a simulator would be done from a simple, menu-driven program. The data display system itself would be menu-driven. In addition, the simulator would provide support for the difficult task of debugging parallel programs.

4.2 Simulation strategy

Section 2.2 discussed the merits of simulation as a method for evaluating parallel programs. The flexibility and non-intrusive nature of simulation-based systems, however, is usually offset by their severe performance overhead. Systems that are accurate enough to capture most interesting interactions between the different parts of a system typically require execution times that are several thousand times longer than running a program directly on a parallel machine. This limitation prohibits the use of simulation for longer, and therefore more interesting, applications.

In order to reduce the overhead of simulation, a system designer may choose not to simulate some of the components and interactions of a real system, or to simulate them in less detail. In that way, the system will present a tradeoff between performance and full accuracy. Usually, such a tradeoff is acceptable because the user rarely requires a complete simulation of all events that take place in the simulated system. On the other hand, different users (or the same user at different times) have different requirements. For example, one user may want to test a new network routing algorithm and thus requires an exact simulation of a packet-switched network. However, he may not care about accurate shared memory modelling. A different user, who wants to test the performance of a new cache coherence protocol, requires accurate modelling of shared memory accesses, but may not require detailed network simulation. A successful system should allow the user to choose the desired performance/accuracy tradeoff himself.

A principal goal in the design of our simulator was to achieve high performance that would allow us to simulate large applications. In order to achieve that performance, we made a number of tradeoffs. The most important tradeoffs are reflected in our simulation strategy that is described in this section. Other tradeoffs in the simulation of individual machine components are described in Section 4.3. The user has the power to control simulation parameters that control both the efficiency/accuracy tradeoff of the simulation strategy and the detail of simulation of machine components.

4.2.1 Combining simulation with direct execution

In order to design an efficient simulation strategy for our system, we examined cycle-by-cycle simulators (see Section 2.2.1) and tried to determine some of the reasons for their very high simulation overheads. Two of the most important were the following:

- Individual instructions are simulated. Hundreds of machine instructions are typically required to simulate a single user instruction.

- Even machine components that are idle are simulated, since the simulator "sweeps" the entire machine before advancing to the next cycle.

At the same time, we observed that a multiprocessor CPU spends most of its time executing operations on local data stored in processor registers or private memory. Local operations are similar to operations on uniprocessor CPUs. Operations that require interaction with other parts of the system, such as shared memory accesses, are performed using a small set of special machine instructions. We use the term *nonlocal instructions* to refer to those special instructions in this thesis. Apart from the support of a number of nonlocal machine instructions, uniprocessor and multiprocessor CPUs are quite similar from a programmer's viewpoint.

By combining the previous observations, we decided to design a system that does not simulate local instructions; instead it executes them directly. We make the assumption that local instructions on the parallel machine's processors are identical to instructions on the uniprocessor where the simulator runs. Special instructions for nonlocal interactions are implemented using function calls that transfer control to the simulator engine. The simulator incorporates a collection of routines that simulate the functionality of nonlocal instructions and account for their performance costs. In that way, local instructions incur no overhead and only nonlocal instructions are simulated.

There are several problems we had to solve to make the above method accurate and efficient enough for our purposes:

- Accurately measure the time taken to execute local instructions.
- Efficiently implement the threads that would be created on different processors on a real parallel machine.
- Interleave the simulation of the threads that would execute concurrently on different processors on a real multiprocessor, so that the timing and sequence of events on the simulator closely matches their timing and sequence on a multiprocessor.

In the rest of this section, we describe how we solved each of these problems.

4.2.2 Timing local instruction blocks

A parallel simulation system keeps a clock for the entire machine or a number of clocks for each processor. These clocks are incremented to reflect the amount of time consumed by instructions executed on each processor. When simulating instructions, it is easy to increment the processor clock to account for the time taken by each instruction. However, when executing instructions directly, accounting for the amount of time consumed by a local block of instructions becomes tricky.

Existing multiprocessor simulators solve this problem in various ways. Some systems [Delagi87, Elshoff91] use the host processor's real-time clock to get an approximate estimate for the time taken by local instructions. This method has limited accuracy, especially since local blocks executed between two nonlocal instructions are usually smaller than the interval between two real-time clock ticks. Other systems [Rizzo89] require the user to explicitly give an estimate of the time taken by the local parts of his code, using a special `takes_time` function call. This method is no more accurate and also puts an added burden on the user of the system. The best method to time local instruction blocks is *code augmentation*, first described by Weinberger [Weinberger84]. Code augmentation adds special code to the assembly version of user code to count executed instructions dynamically.

Augmentation is the third step in the compilation of Proteus user source files (Figure 3-1) and is performed by a special program called `augment`. During augmentation, the assembly version of user code is broken into *basic blocks.* The defining property of a basic block is that its instructions execute as an atomic unit. That is, it is not possible to execute only a subset of the block's instructions. Three types of statements determine block boundaries: labels, branch instructions and call instructions. A label requires a new block since otherwise a program could jump into the middle of a block, breaking the atomicity requirement. Likewise, a branch instruction ends a block to prevent jumping out of the middle of a block. The reason for ending a block after a call is more subtle. The called procedure may give up the processor because it contains a nonlocal instruction that calls the simulator engine. Therefore, it is important to ensure that the cycles for this block prior to the call have been counted. Thus code in the basic block up to and including the call instruction is always counted before the call, and code after the call instruction is always counted after the call returns.

After locating basic blocks, the augmentation program consults an *instruction cost file* that contains the cost, in cycles, of every local instruction type. Based on that information, `augment` calculates the cost of each basic block and inserts before each block special code that updates the cycle counter by the number of cycles consumed by the block. Figure 4-1 gives a simple example of the augmentation process.

```
wordlength :
  x = -1;
  bits = 0;
loop :
  x <<= 1;
  bits++;
  if ( x != 0 ) goto loop;
  print bits;
```

(a) A simple program fragment that calculates the wordlength of the machine.

```
wordlength :
  x = -1;
  bits = 0;
-----------------------------
loop :
  x <<= 1;
  bits++;
  if ( x != 0 ) goto loop;
-----------------------------
  print bits;
```

(b) The same program fragment separated into basic blocks.

```
wordlength :
  cycles += 2;
  x = -1;
  bits = 0;
-----------------------------
loop :
  cycles += 3;
  x <<= 1;
  bits++;
  if ( x != 0 ) goto loop;
-----------------------------
  cycles += 1;
  print bits;
```

(c) The original program fragment augmented with cycle-counting code.

Figure 4-1: An example of code augmentation. In this example, we assume that each statement costs one cycle.

4.2.3 Efficiently implementing user threads

During execution of a parallel program on a real multiprocessor, a large number of different tasks or threads would be executing concurrently on different processors. Our simulator should provide a way to implement these threads efficiently on a uniprocessor. Efficiency considerations require a fast way to create and destroy threads and a low-overhead mechanism for switching control between threads.

Some existing systems [Davis90] have made use of the existing UNIX system call interface to implement user threads using UNIX processes. This approach has the advantage that it does not require writing extra code. However, it has two serious limitations. UNIX processes reside in separate address spaces and contain much more state than is needed for our purposes and therefore all operations (creation, destruction, context switch) on them incur severe overheads. Also, the maximum number of UNIX processes is limited by the size of system tables to about one hundred. On a large-scale parallel machine with hundreds or thousands of processors, several thousand threads may well be active at the same time.

We decided to implement our own threads package that would multiplex a large number of lightweight threads on top of a single UNIX process. In our package, threads have a minimum of state: Each thread is assigned an entry in a *threads information table* and a stack for local data and procedure invocations. All threads share the same global data, which provides a cheap way for communicating values across threads.

A *thread information block* (TIB) contains space for storing the processor context when a thread passes control to another thread, as well as a small amount of additional system information. Stack blocks are allocated from a static stack block pool. To avoid page faults when searching the stack pool for free blocks, the stack free list is implemented as a separate compact table.

The threads package provides calls to create new threads and discard terminated threads. Transfer of control between threads is done using a coroutine mechanism. This requires simply storing the processor context of the old thread in its TIB and loading the context of the new thread from its TIB. On the MIPS R3000 processor where our system is implemented the above procedure requires 135 instructions or 5.4 microseconds.

4.2.4 Interleaving different simulated processors

On a real parallel machine, many processors execute different instructions concurrently, and several events take place at the same time. A uniprocessor simulating a parallel machine can execute or simulate only one instruction or event at a time. In any multiprocessor simulation it is important to provide an interleaving of the concurrently executing threads that will ensure that the timing and sequence of simulated events is as close as possible to reality.

The most accurate interleaving is achieved by allowing each thread that is executing on a processor to execute a single instruction before passing control to the next thread. This approach, used in cycle-by-cycle simulators, ensures that all processors advance together in time and all events are simulated in the correct order. However, it results in very high simulation overhead.

We observe that in a parallel machine simulation, nonlocal operations, that is, operations that are visible to more than one system components, are of the greatest interest. Local operations can be performed in any order that ensures the correct timing and sequence of nonlocal operations. Therefore, to achieve better performance, in our system we allow processors to drift slightly apart from each other in time. Each processor now has its own clock, which reflects the number of cycles the processor has executed since the beginning of the simulation. Processors are synchronized whenever one of them executes a nonlocal operation.

On our simulator, every task spawned by the user program on a processor is implemented as a thread. The simulator engine itself is also a thread. As it was mentioned, transfer of control between threads is done using a coroutine mechanism.

Whenever control is transfered to a user thread, the thread is allowed to execute a block of local instructions (augmented by cycle-counting code) uninterrupted. As soon as a thread needs to perform a nonlocal operation, however, the following events take place:

- A request for the nonlocal operation is placed on a simulator priority queue, ordered by timestamp.

- Control passes to the simulator engine thread.

The simulator engine maintains a single priority queue where threads that are ready to continue execution of local instructions, as well as threads that are waiting for the completion of nonlocal operations, are ordered by their timestamps. Events with identical timestamps can be optionally placed in the queue in a random order. This feature can be used to simulate the nondeterminism of real parallel machines: Using different random seeds, slightly different request orderings are expected to produce slightly different program behavior, just as it happens across different executions of the same program on a real machine.

When it gains control, the engine always removes and services the earliest thread request in the queue. If the earliest thread is ready to continue execution of local instructions, control is transfered from the engine to the thread. Otherwise the engine calls a simulator routine that implements the requested nonlocal operation. Implementation of a nonlocal operation may cause the generation of additional, intermediate simulator requests. When a nonlocal operation is complete, a request is placed to resume the requesting user thread.

This scheme ensures that a synchronous nonlocal operation is started only when all earlier nonlocal operations have also been started. The timing and sequence of synchronous nonlocal operations on the simulator is identical with their timing and sequence on a real machine. This includes all shared-memory accesses and message send operations.

4.2.5 Advantages and limitations of our approach

The combination of simulation and direct execution used in our simulator has resulted in a system that is two orders of magnitude faster than other systems of comparable flexibility (see Section 5.2.4). In a typical program, a parallel machine processor spends most of its time executing simple, local instructions. Our approach allows simulation of those local parts with the minimal overhead of the cycle-counting code inserted during augmentation.

Our system does not spend any time simulating idle processors. The simulator is driven by requests generated by threads running on simulated processors. Whenever a processor becomes idle, there is no thread on it to generate any more requests and the simulator simply forgets about it. Whenever an event (typically an interprocessor interrupt) makes that processor busy again, the clock of the processor is set to the timestamp of the event that has woken it up and its simulation is resumed.

Our simulation method is a tradeoff between complete accuracy and good performance. There are only two areas where our method introduces some small inaccuracies. The direct execution of local instruction blocks assumes that all instruction and local data accesses can complete in a fixed number of cycles. This assumption does not permit modelling the effects of instruction and local data caching, which would make the cost of local fetches smaller on cache hits and larger on cache misses. These effects, however, tend to be quite small in modern systems with large, efficient caches.

The other source of inaccuracies is in asynchronous interrupt reception times. On real multiprocessors, interrupt reception can occur at any time. On our simulator, local instruction blocks are executed uninterrupted and received interrupts can only be acknowledged and serviced at points where threads make nonlocal requests. In most cases this results in negligible inaccuracies, since nonlocal requests are very frequent. However, in infrequent cases, a user thread may enter a very long block of local instructions and thus cause an excessive delay in the service of pending interrupts. Worse still, as a result of a program bug or otherwise, a thread may enter an infinite loop and never transfer control to the simulator engine again.

To avoid these problems, the simulator limits the number of local instructions that a user thread can execute uninterrupted before it is forced to transfer control to the simulator engine. This number is called the *simulator quantum*; its value is set by the user. The simulator quantum is an upper bound on the number of cycles that two simulated processors can be apart from each other at any time. It is also an upper bound on the inaccuracy in interrupt reception times. In applications where accuracy in interrupt reception times is important, the user can increase accuracy at the expense of performance by setting low quantum values.

4.3 Architectural model

Many research studies (see for example [Kuck84]) have pointed out the important effect of the underlying architecture on the performance of a parallel program. Even in the simple example we presented in Section 3.4, going from a bus-based to a hypercube machine had a dramatic impact on the program speedup. For that reason, any reliable evaluation of a parallel program must test its performance on a variety of different architectures. One of our major goals in Proteus was to design a system that could be easily configured to simulate a wide range of MIMD architectures. We were not interested in simulating specific machines in full detail, but wanted to be able to configure the system so as to simulate the important interactions between the architecture and a parallel program on any given machine.

Our architectural model consists of a number of *architecture simulation components* that simulate the three main parts of a parallel computer system: processing elements, memory and interconnection medium. The user can choose among several different components for each machine part. In that way, from the combination of a small number of components, a large number of different architectures can be constructed. In addition, each component has several *architectural parameters* that can be set to fine-tune our simulation to a given target architecture.

Different simulation components usually simulate different architectures. For example, there is a bus and a network interconnection component. Different components may also simulate the same machine part at different levels of detail (and performance). For example, we provide two network simulation components, an exact hop-by-hop simulator, and a faster but less detailed simulator based on an analytical model of network contention. By choosing less detailed simulation components users can gain some extra performance at the expense of full accuracy.

In the rest of this section, we will describe the components available in the current version of the system.

4.3.1 Processor model

A single, generic processor model is used for all machine architectures. Our processor architecture combines features found in processors for both shared memory and message passing machines.

The local instruction set of our processor is identical to the instruction set of the host machine where the simulator runs. As explained in Section 4.2.2, the simulation method used in our system times local instruction blocks by augmenting user code with cycle-counting instructions during compilation. This method assumes that each instruction completes in a fixed number of cycles. In architectural terms, this can be translated to the assumption that our processor has an amount of fast, private memory where code and local data can be stored. An approximately equivalent assumption is that each processing element is equipped with large instruction and local data caches that guarantee a very high hit ratio. The cost of each instruction type, in cycles, is defined in an *instruction cost file*, used by the augmentation program. This file can be modified to fine-tune the simulation of different existing processors.

Each simulated processor supports two methods of interacting with other parts of the system:

- Shared-memory accesses
- Interprocessor interrupts

Shared-memory accesses are handled by the memory model, which is described in the next section. Interprocessor interrupts (IPIs) are the principal method of communication

- Local instruction costs in cycles
- Context switch latency
  (cycles to save and restore processor state)
- Interrupt latency
  (cycles needed to save state and branch to interrupt handler)
- Number of available interrupt types

Table 4.1: Architectural parameters of the processor component

in message-passing machines. In our system, an IPI is a packet of data that is transmitted through the interconnection medium and causes an asynchronous interrupt upon reception at the target processor. Reception of an interrupt results in the execution of an interrupt handler routine in the context of the thread that was executing when the interrupt arrived. Interrupts have priorities and their servicing can be delayed by temporarily disabling a processor's interrupt mask. Each processor also has a countdown timer that produces a local interrupt upon timeout.

Table 4.1 lists the architectural parameters of the processor component.

4.3.2 Memory model

Shared memory is divided into memory modules of equal size. Memory modules can either be connected to the interconnection medium, or each can be associated with a processor to form a processor-memory node. Address to module translation can be based either on the lower or on the upper bits of a shared memory address. Basing the translation on the lower bits provides memory interleaving, which reduces memory module contention when several processors concurrently access the same region of shared memory. Basing the translation on the upper bits, on the other hand, enables the allocation of contiguous blocks of memory from the same block. Memory module contention is accurately modelled.

The system provides different components that allow the user to specify architectures with or without caching. When caching is selected, cache coherence is provided by full simulation of an appropriate protocol. We have selected Goodman's snoopy-bus

- Number of memory modules
- Memory module size
- Cache line size
- Cache set size
- Number of sets per cache
- Cache access latency
- Memory module access latency
- Address to module translation mechanism (lower or upper bits)

Table 4.2: Architectural parameters of the memory components

coherence protocol [Goodman83] for bus-based machines and Chaiken's limited directory protocol [Chaiken90] for network-based machines. Different cache coherence protocols can easily be added by the user, as explained below. The system also allows shared memory accesses without cache coherence. In that case every cache behaves as if it is the only cache accessing main memory. This option is useful to simulate machines in which cache coherence is achieved by software means (e.g. the IBM RP3 [Pfister85]).

Every byte of shared memory has an associated full/empty bit [Agarwal90, Smith81], which can optionally be checked or set by shared-memory accesses. Full/empty bits are useful for synchronization, and can also serve as general-purpose tags.

Table 4.2 lists the architectural parameters of the memory components.

Cache coherence protocols

Proteus supports detailed simulation of cache coherence protocols. Users can replace the supported cache protocols with their own. The interface between cache protocols and the rest of the memory component consists simply of two routines `processor_request_pre` and `processor_request_post`. The first routine is called before a read or write takes place, to fetch a location into a local cache. The second routine is called after a local

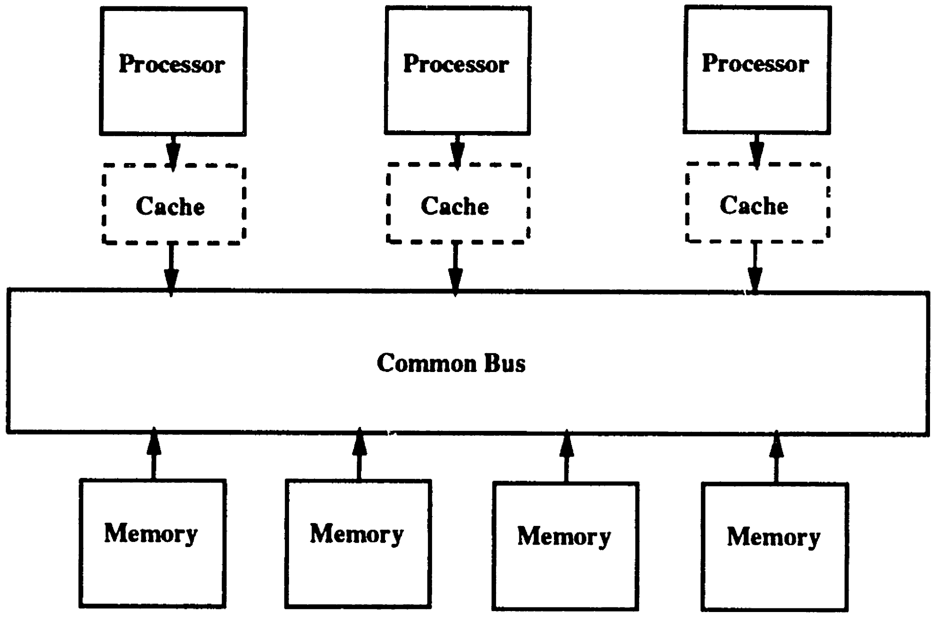


Figure 4-2: Typical bus-based architecture supported by the simulator.

cache has been read or written, in order to perform write-back, or any other bookkeeping needed by the protocol. Cache coherence protocols must support five operations: `READ` and `WRITE` for cache coherent memory accesses, `READSOFT` and `WRITESOFT` for memory accesses without cache coherence, and `FLUSH` for explicit invalidation of addresses from local caches. In addition, a cache coherence protocol must supply a `protocol_fence` function, which stalls the calling processor until all pending flushes have been completed.

4.3.3 Interconnection model

Our simulator supports two kinds of interconnection: Bus and network.

When bus is selected, all processing elements and all memory modules are connected together through a common bus (Figure 4-2). All nonlocal interactions (shared memory accesses and IPIs) have to pass through that bus. Uniform shared memory access is assumed, that is, access of any memory module from any processor takes the same amount of time (ignoring delays due to bus contention). Bus contention is accurately modelled.

The supported network-based architectures connect a number of processor-memory nodes using a variety of direct and indirect networks (Figure 4-3). *Direct networks* directly connect two processing nodes through point-to-point links. In such networks there is a variable "distance" between two processing nodes, equal to the number of individual

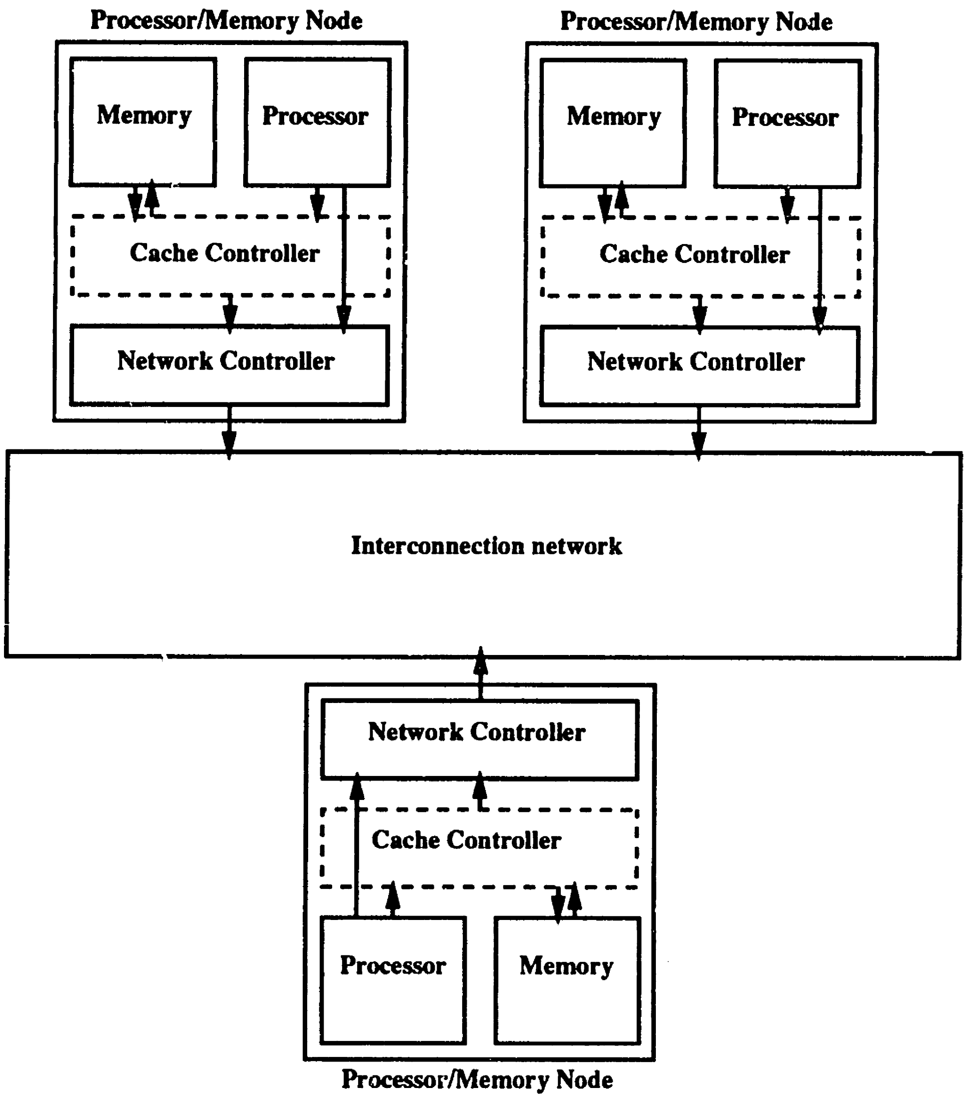


Figure 4-3: Typical network-based architecture supported by the simulator.

- Maximum packet length
- Direct/Indirect network
- Unidirectional/Bidirectional links
- Number of stages (for indirect networks)
- Network radix (k) and dimension (n) (for k-ary n-cubes)
- Switch delay
  (cycles needed to send one word through a network switch)
- Wire delay
  (cycles needed to send one word through a point-to-point link)

Table 4.3: Architectural parameters of the network components

point-to-point links that form the shortest path between those nodes. *Indirect networks* do not directly connect any two processors by point-to-point links but rather have a number of internal switching stages that automatically route a packet to its destination. In these networks, all pairs of processing nodes have the same "distance", equal to the number of internal stages plus one.

Our network components support the complete family of k-ary n-cube direct networks [Seitz84] with either unidirectional or bidirectional connections. This family includes most usual types of network encountered in MIMD machines, such as rings, meshes and hypercubes. Our indirect network model can emulate Butterfly networks [BBN87], Omega networks [Lawrie75], Delta networks [Patel81], etc.

Two different network simulation components are provided:

- An approximate simulator that calculates network latency based on analytical formulas that take into account network contention [Agarwal89]. This component is very fast and reasonably accurate; however, it assumes an uniform distribution of network load and it is unable to model hot spots.
- An exact, hop-by-hop packet-switched direct network simulator for cases where

high accuracy is important. Static lowest-dimension-first wormhole routing is provided by default. The routing algorithm can be redefined by the user, allowing experimentation with other routing methods. User-defined routing algorithms can also be used to implement irregular network topologies that do not belong to any of the supported network families.

Table 4.3 lists the architectural parameters of the network components.

4.3.4 Modelling contention

All architectural components use the same simple method to model resource contention. Our method takes advantage of certain properties of the simulation strategy described in Section 4.2 to eliminate the need for busy-waiting or separate resource buffers and queues.

In real machines, when a request arrives to a resource that is already busy servicing other requests, the new request is either placed in a buffer or the requesting processor busy-waits until the resource is ready to service it. A network routing switch is an example of a resource that uses buffers to store pending input and output packets. A common bus is an example of a resource that causes requesting processors to busy-wait (enter wait states or continuously retry the request) until they gain exclusive access to its data paths.

Our system splits all requests to shared resources into two phases: A *request* phase and a *grant* phase. The request phase simply calculates the time the resource will be made available to the requesting thread and generates a *grant request* with that timestamp. The grant phase actually accesses the resource and performs the requested operation. The advantage of this approach is that it eliminates the need for busy-waiting. However, the method relies on the ability to correctly calculate the timestamp of the grant request.

We calculate when a resource will be granted to a requesting thread with the help of an integer variable kept for each shared resource. This variable, will be referred to as *resource_free_again* in this section. If a resource is busy, *resource_free_again* stores the earliest time when that resource will become free again. If a resource is free, the variable stores the timestamp of the cycle that immediately follows the last period of time during which the resource was busy. *Resource_free_again* is initially set to zero for all resources and is updated every time a resource is utilized, as explained below.

If a resource is free when it is requested, it is granted immediately. Otherwise it is granted at the first cycle on which it becomes free again. This is modelled by the formula:

bus_free_again	Request		
$bus\_free\_again = 0$	Processor 0	time = 100	REQUEST BUS
$bus\_free\_again = 110$	Processor 0	time = 100	BUS GRANTED
$bus\_free\_again = 110$	Processor 1	time = 104	REQUEST BUS
$bus\_free\_again = 120$	Processor 1	time = 110	BUS GRANTED

Figure 4-4: A simple example of bus contention modelling

$$resource\_granted = max(request\_time, resource\_free\_again)$$

Our system uses *resource_granted*, computed by the previous formula, as the timestamp of the grant request. The system now knows that some thread will access the resource at that time. It also knows the operation this thread will perform on the resource and presumably its latency (this information is supplied during the request phase). With this information, *resource_free_again* can be updated as follows:

$$resource\_free\_again = resource\_granted + operation\_latency$$

Requests for access to shared resources are always implemented as simulator calls that cause a simulator request to be placed in the simulator engine's queue. Simulator requests are extracted from the simulator queue and serviced in their exact timestamp order (see Section 4.2.4). Therefore, a request for a shared resource will be serviced exactly when all previous requests have already been serviced by the simulator. By induction, this means that *resource_free_again* always contains the earliest time when the resource will be available.

Figure 4-4 gives a simple example of how this method works to model bus contention. Processor 0 requests the bus at time 100. At that time the bus is free and is granted immediately. We assume that all bus accesses take 10 cycles. Therefore, after the bus is granted to processor 0, it will become available again at time 110. At time 104, processor 1 also requests access to the bus. The system knows that the bus will be available again at time 110 and grants it to processor 1 at that time. It also updates *bus_free_again* to 120. Using *bus_free_again*, the system knows exactly when a bus request will be granted. Thus, there is no need for processors to busy-wait in the simulator.

Our strategy services requests to shared resources in a FIFO manner. In reality, FIFO servicing is not always the case. In fact, many shared resources (especially buses) may not even guarantee fairness. We can extend our scheme to model resources that are not fair by maintaining a list of pending requests for each shared resource. The request phase schedules a grant request, as before. In addition, it inserts the requesting thread in the list of pending requests for the resource. In this case, the grant request simply means that the request will be granted to *some* thread at the calculated time; it does not specify which thread. When serviced, the grant request picks a thread from the pending request list and grants the resource to it. Since any algorithm can be used to pick a requesting thread, this scheme is able to model any resource arbitration mechanism.

4.4 Programming model

Our system provides a programming model for writing parallel programs. There are two conflicting requirements from the programming model:

- It must be low-level enough to enable the progr nmer to control the behavior of the simulated machine at the level of the simulation, which is the machine instruction level. A low-level interface is also desirable if the system is to be used as a target for experimental compilers of new parallel programming languages.
- It must be simple enough to be used directly by programmers.

An assembly-level interface would allow full control of the instruction-level behavior of each processor. However, it would make direct coding of new programs difficult and time-consuming. In our system, we have chosen to base our interface on the C programming language, augmented with a set of calls and language extensions that implement the additional instructions present in multiprocessor CPUs. We chose C because it is a rather "low-level" high-level language that has minimal run-time support and does not introduce significant distortions to the underlying processor model. On the other hand, it hides tedious low-level details such as register allocation, data placement, data type conversion, etc. C is simple enough to be used directly by programmers and can also function as a high-level assembly language for parallel language compilers. Compilers can perform high-level optimizations and produce C code as output. A standard optimizing C compiler can then translate this code into assembly, performing low-level, machine dependent optimizations.

Our programming model augments the standard C language with a set of low-level simulator calls and a number of language extensions.

4.4.1 Low-level simulator calls

The standard C language is used as a high-level assembly in which local parts of user programs are written. A set of simulator calls is provided to implement the functionality of special instructions that perform nonlocal interactions on a parallel machine. These calls are listed in Table 4.4. Low-level calls are used to perform shared memory accesses, to send interprocessor interrupts, to define and use spinlocks, and to control hardware features of our processor model, such as the interrupt flag and the countdown timer.

The low-level simulator call interface is useful as a target language for parallel language compilers or as a building block for higher-level interfaces.

4.4.2 Language extensions

Low-level simulator calls are at the level of individual machine instructions. For example, shared memory calls require an explicit memory address and number of bytes to access. The use of low-level simulator calls may become difficult in user programs whose other parts are written in C, since C does not deal with individual memory addresses but rather with named variables and structures. In our low-level model, accessing a field of a structure stored in shared memory would require explicit calculation of the exact address and length of that field. To avoid burdening the user with such tedious calculations, our programming model provides extensions to the C language that allow definition and use of shared memory variables using the normal C syntax. We also provide a preprocessor, `catoc`, that translates all references to shared variables or structures to the appropriate sequence of simulator calls. `Catoc` is called in the first step of the compilation process described in Section 3.2.4.

A new `shared` storage class has been added to C to declare variables that are placed in shared memory. Shared memory variables can be optionally qualified as `tagged`, `untagged` and `soft`. `Tagged` variables are accessed using full/empty bit synchronization (see Section 4.3.2). `Soft` variables are accessed without cache coherence. The default is cache coherent accesses without full/empty bit synchronization. Shared variable declarations can be optionally accompanied by placement directives using the new `placeat` keyword.

A new `tag` unary operator can be used to read or write the full/empty bits of the first location where a shared memory variable is stored. Finally, C pointers can now point to data that is stored both in private and in shared storage. To distinguish pointer accesses to shared-memory locations from accesses to private locations, a set of new operators

Shared Memory Support
• Access an integer value from a shared memory location *value = Shared_Memory_Read(address, mode)* *value = Shared_Memory_Write(address, value, mode)* • Access a floating point value from a shared memory location *value = Shared_Memory_Read_Fl(address, mode)* *value = Shared_Memory_Write_Fl(address, value, mode)* • Access the tag (full/empty bit) of a shared memory location *value = Shared_Memory_ReadTag(address, mode)* *value = Shared_Memory_WriteTag(address, value, mode)* • Flush an address from a local cache *Flush(address)*
Spinlock functions
• Create and discard a spinlock on a given memory module *spin_address = sem_open(value, name, module)* *sem_close(spin_address)* • Acquire a spinlock *sem_P(spin_address)* • Release a spinlock *sem_V(spin_address)*
Interprocessor Interrupts
• Send a interrupt to another processor *send_ipi(processor, priority, ipi_type, length, argc, args)*
Control of hardware features
• Access the countdown timer *value = ReadTimer(processor)* *WriteTimer(processor, value)* • Enable/Disable interprocessor and timeout interrupts *value = ReadInterruptFlag(processor)* *WriteInterruptFlag(processor, value)*

Table 4.4: Summary of low-level simulator calls

has been added for pointer accesses to shared-memory locations. The standard pointer access operators have been reserved for accesses to private memory locations.

4.5 Runtime system interface

In order to run parallel programs, every real machine requires an operating or runtime system that provides basic services, such as thread and memory management. Many simulator systems (e.g. [Rizzo89, Davis90]) chose not to simulate the operation of the runtime system in detail, but rather provide a set of simulator calls that approximate the functionality of the real runtime system calls.

Proteus provides a detailed implementation of a simple runtime system, which is built on top of the programming model discribed in the previous section and is viewed by the simulator as a user program. We believe that a detailed implementation of a runtime system is essential for the following reasons:

- The interactions between the runtime system and a program can have significant effects on the program's performance. When unexpected execution behavior is observed, it is important to be able to measure which part of it is due to the program, and which part is due to the runtime system.

- Runtime system design for parallel computers is an interesting area of current research. By supporting detailed runtime system modelling, our system can be used to evaluate new ideas in many runtime-system issues, such as scheduling, placement, migration, etc.

The cost of our runtime system routines is accurately calculated by cycle-counting their code. Time spent in runtime system code can be separated from time spent in user code to highlight the effects of runtime system functions on overall algorithm performance. If desired, the costs of selected runtime system functions can be set to a fixed value (including zero), to observe the effects on overall performance.

Since runtime system routines are not part of the simulator engine, they can be partially or completely replaced by user-defined routines. Thus, the system can be used for experimentation with different scheduling algorithms and other runtime-system issues.

Our runtime system model assumes that every processor has an identical copy of the runtime system code. The user can choose between two versions of the runtime system: *Single-ready-list* (SRL) and *multiple-ready-lists* (MRL). In the SRL version, there is a global pool of ready threads and each processor picks and executes one of these threads

Thread Management

- Spawn a new thread on a processor
  tid = OS_spawn(function, stacksize, threadname, processor, priority, mode, argc, args)
- Kill a thread
  OS_kill(tid)
- Join with a previously created thread
  value = OS_join(tid)
- Change a thread's scheduling priority
  old_priority = OS_set_priority(tid, new_priority)
- Temporarily suspend a thread or resume a suspended thread
  OS_suspend(tid)
  OS_resume(tid)
- Sleep for a specified amount of time
  OS_sleep(cycles)
- Enable or disable interprocessor and timeout interrupts
  old_state = OS_interrupt_ctrl(tid, new_state)
  old_state = OS_set_preempt(tid, new_state)

Memory Management

- Dynamically allocate or deallocate a block of shared memory
  addr = OS_getmem(nbytes)
  addr = OS_getmemfrommodule(nbytes, module)
  OS_freemem(addr)

Table 4.5: Summary of supported runtime-system functions

whenever it becomes idle. In the MRL version, threads are created on specific processors. When there is more than one thread on a processor, the processor divides its time between them according to a local scheduling algorithm. Finally, there is a version of the runtime system that does not make use of shared memory, to be used when simulating message-passing architectures.

Any thread can invoke a runtime system call that affects any other thread. Threads have globally unique identifiers that also identify the processors where they reside. When a target thread is on another processor, the call is transmitted to the target processor using an interprocessor interrupt and the service is executed there.

After system initialization, the runtime system creates a thread on processor 0 that executes function **usermain** of the user program. That thread is responsible for dynamically creating all the remaining threads of the program.

Table 4.5 lists the supported runtime system calls.

Chapter 5

Experiments

This chapter summarizes a number of experiments that were performed to evaluate two important aspects of our system: accuracy and performance.

Since simulation is based on models of real systems, before any data obtained from simulations can be trusted, the simulator itself must be validated and proven to provide accurate results. Thus, our first goal in this chapter is to provide experimental evidence which demonstrates that Proteus is accurate enough to allow the reliable evaluation of parallel programs.

Multiprocessor simulators typically incur high simulation overheads that slow down the execution of simulated programs considerably. Our second goal in this chapter is to analyze the simulation overhead and measure the overall simulation performance of Proteus. These performance measurements allow us to compare our system to other similar multiprocessor simulators and prove that the simulation strategy discussed in Section 4.2 has indeed resulted in a system with superior efficiency.

5.1 System validation

A simulation system must be accurate enough to produce results that do not lead to erroneous conclusions. Simulators are generally very complex systems and formal validation of their accuracy is in practice very difficult. In this section, we do not attempt to prove our system correct in a strict mathematical sense. We report experiments, however, that support the claim that the system is reliable enough to be used in drawing conclusions and making decisions.

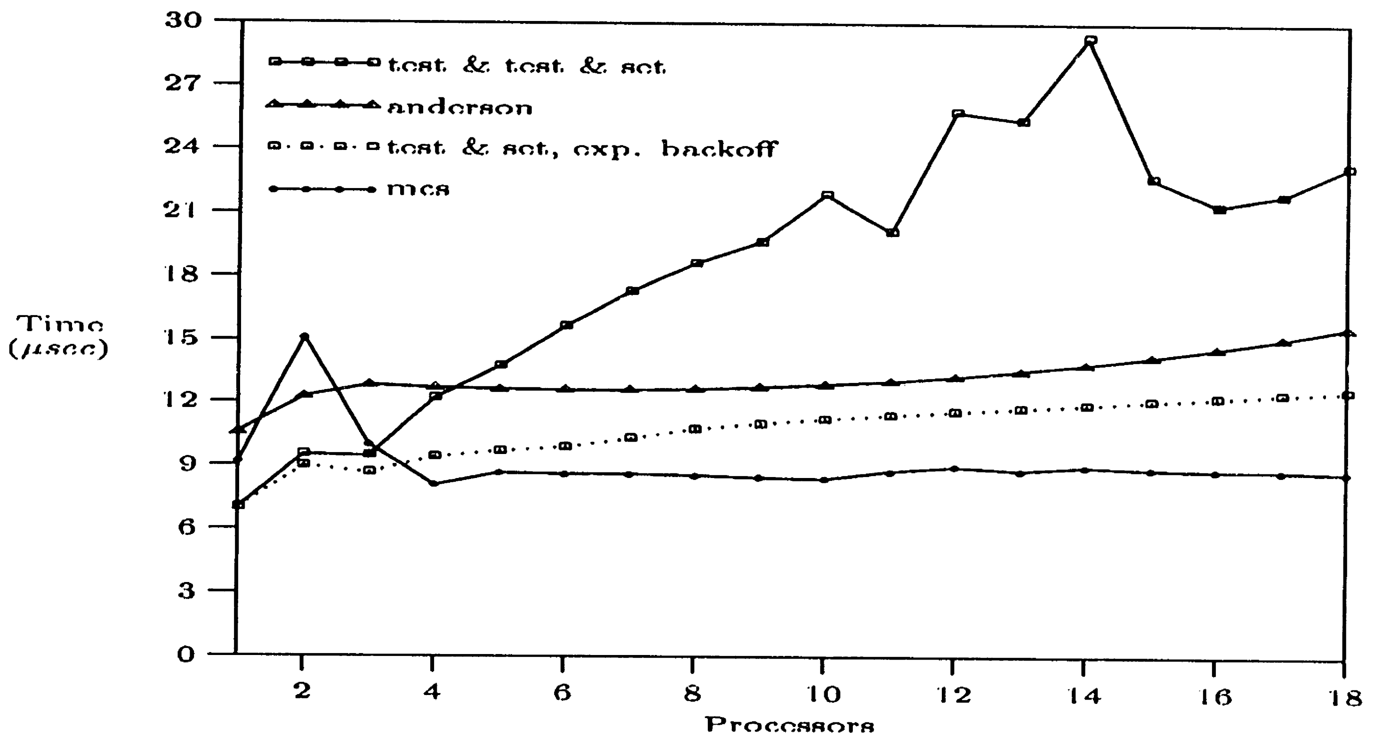
When simulating an existing computer system, simulation accuracy can be validated by comparing the results of a simulation run to those of a direct execution of a program on the target system. Proteus is not meant to simulate any particular machine in full detail but rather to provide insight into the behavior of a program on a wide range of different machines. In order to validate our system, we reproduced a number of experiments described in published papers and compared the results we obtained from Proteus to those reported on the papers. We did not expect absolute simulated running times to match those reported in the literature, as that would require an accurate simulation of a particular machine model. What we were mainly interested in was to determine whether our system correctly simulated the qualitative behavior of a program on a given machine (program concurrency versus time, processor lifelines, resource contention versus time, spinlock activity versus time, etc.), the relative behavior of an application on a family of related machines (program speedup, average concurrency, etc.), or the relative behavior of several applications on the same machine. These are the measures that are most important for the users of our system.

Adrian Colbrook used Proteus to reproduce experiments previously performed on a Supernode system[1] [Colbrook90]. The simulator produced results that were very close to those obtained experimentally. The complete set of experiments was implemented on the simulator in under two weeks, compared to the six weeks required to implement the same experiments on the Supernode. Colbrook et al. also used Proteus to evaluate a new algorithm for maintaining a balanced search tree [Colbrook91].

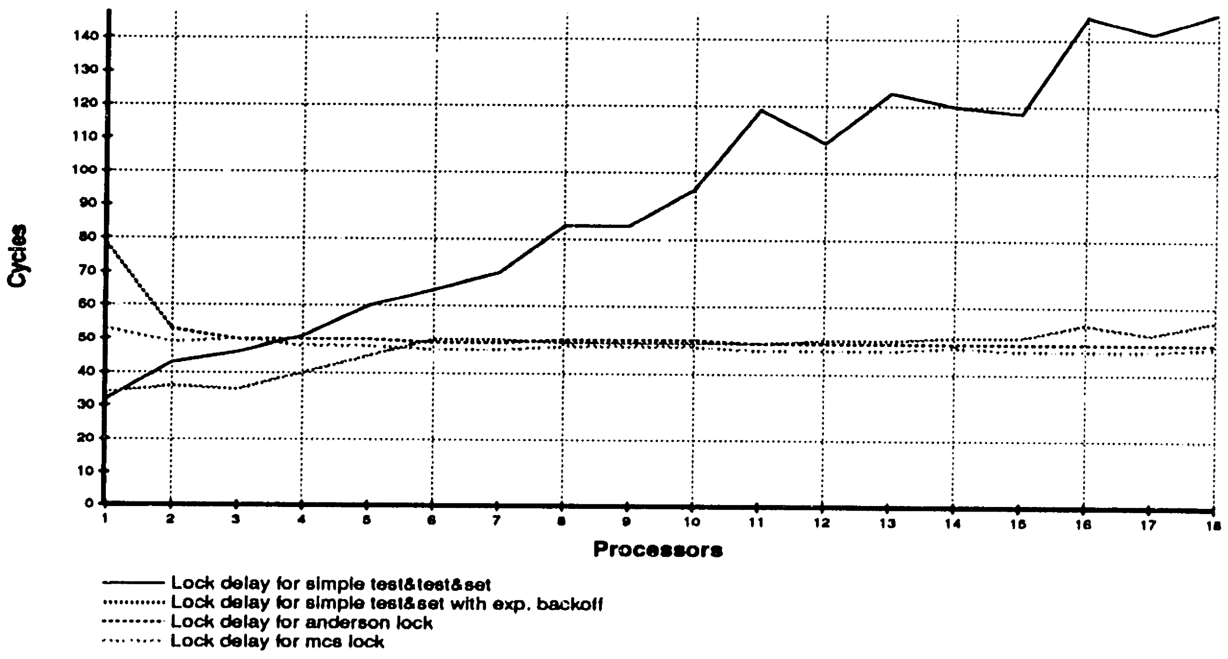
In his thesis, Chaiken [Chaiken90] introduces a new directory-based cache coherence protocol for large-scale multiprocessors. Chaiken describes a number of thrashing conditions that can arise during the protocol's operation and proposes some solutions. Proteus supports an exact implementation of Chaiken's protocol. Before the solutions proposed by Chaiken were implemented, we were able to observe all the thrashing conditions he described.

Mellor-Crummey and Scott [MCS90] report an extensive set of experiments that test the scaling behavior of several spinlock implementations on bus-based and network-based machines. We obtained the code that was used to run the reported experiments and modified it to run on Proteus. All necessary modifications were completed in just one day, which is an indication of the power and convenience of our programming model. We then configured our system to match the systems on which the reported experiments were performed (the Sequent Symmetry and the BBN Butterfly) as closely as possible, and repeated the same experiments on Proteus. The latency curves obtained by our system

[1] a distributed-memory MIMD machine based on Transputers and programmed using a parallel dialect of C.

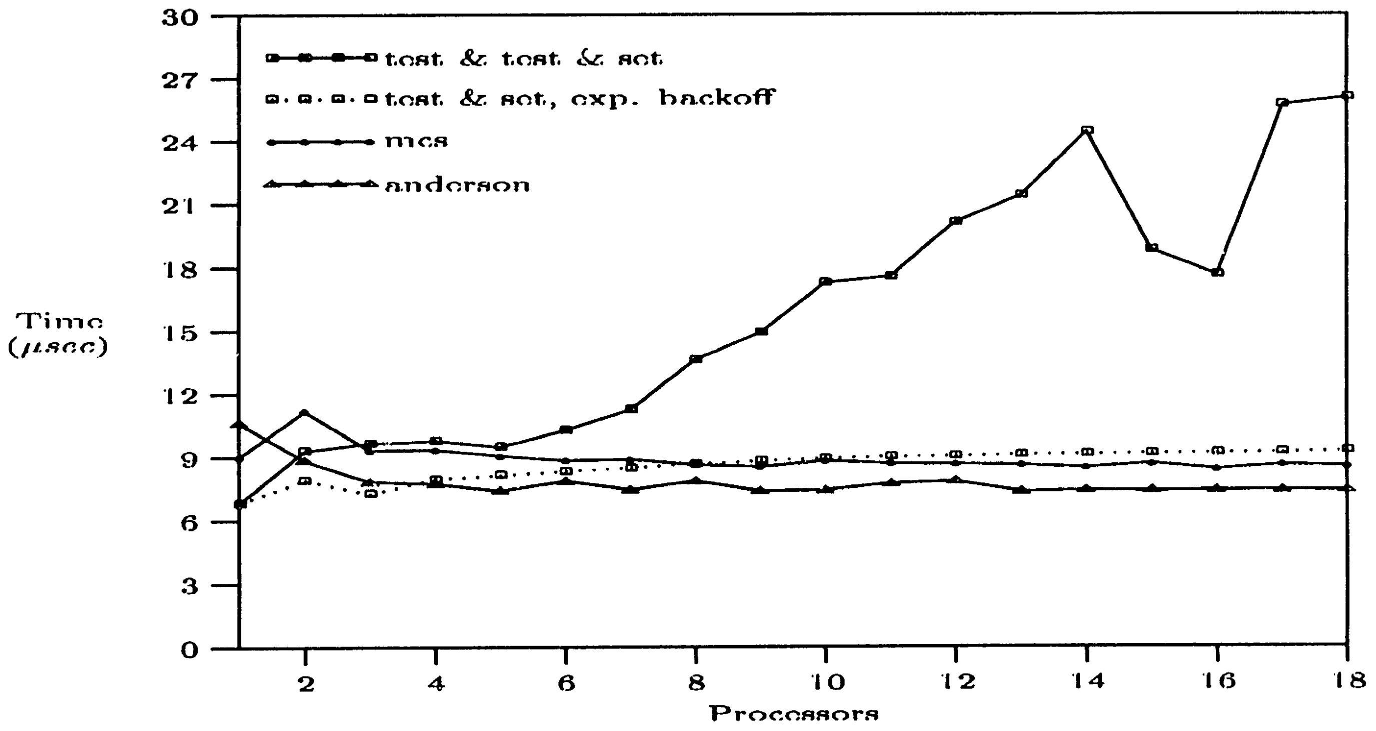


(a) Original graph, as published in [MCS90]

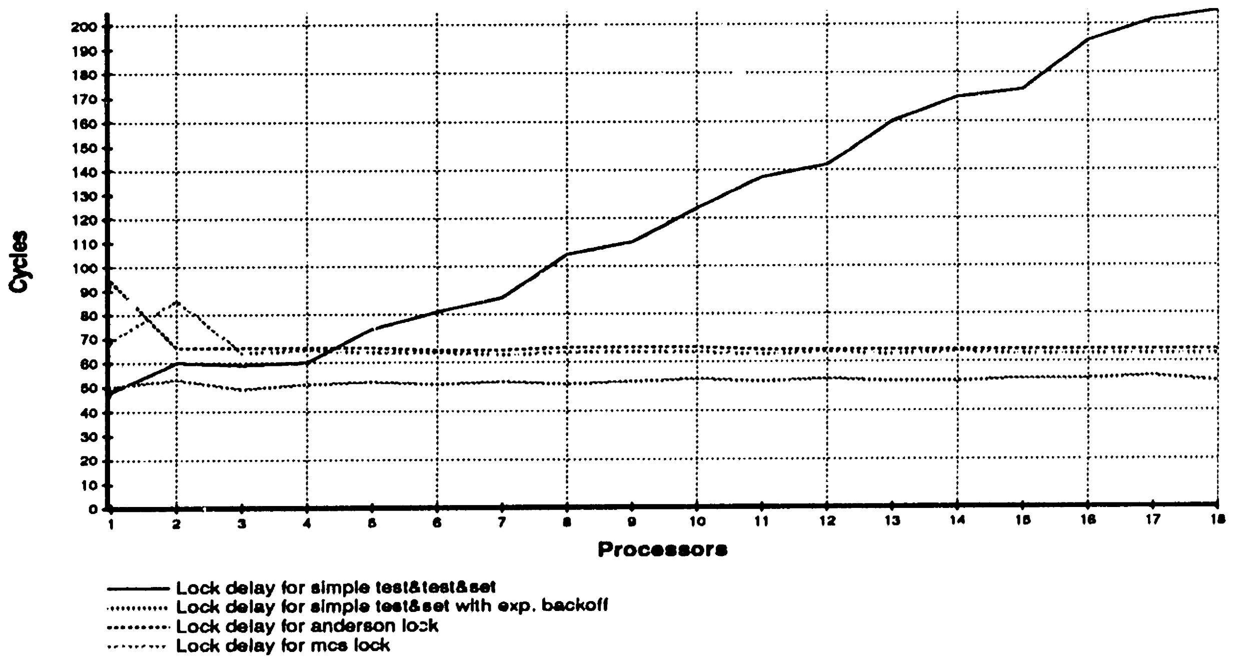


(b) Graph produced by running same experiment on Proteus

Figure 5-1: Performance of spin locks on a bus-based machine (empty critical section)



(a) Original graph, as published in [MCS90]



(b) Graph produced by running same experiment on Proteus

Figure 5-2: Performance of spin locks on a bus-based machine (small critical section)

are remarkably similar to those published in the paper (Figures 5.1 and 5.2).

5.2 Performance measurements

The viability of simulation as an approach for the evaluation of parallel programs is highly dependent on the program slowdown caused by various simulation costs. Typical multiprocessor simulator systems exhibit program slowdown of several thousand times per simulated processor. When simulating large-scale parallel machines with hundreds or thousands of processors, such slowdowns limit the use of simulation to the simplest, and therefore less interesting, parallel applications. One of our goals in Proteus was to design a simulator that is efficient enough to allow simulations of nontrivial parallel programs. In Section 4.2 we described the design of our simulation component. This section describes several experiments we performed to measure its performance. It also compares the performance of Proteus to other similar systems.

5.2.1 Sources of simulation overhead

Several factors contribute to reducing the performance of parallel programs that are executed using our simulator. Generally, these factors can be classified in one of the following four categories:

- Overhead due to code augmentation
- Overhead due to context switching
- Overhead due to statistics logging
- Overhead due to simulated nonlocal operations

Code augmentation inserts extra cycle-counting instructions in the user program code to accurately calculate the cost of local instruction blocks (see Section 4.2.2). Brewer [Brewer91] reports that code augmentation increases the number of locally executed instructions by a factor of 2 to 2.5. However, since local instruction blocks usually amount to a very small fraction of the total simulation time, the effects of code augmentation on performance are negligible.

Our simulator multiplexes a large number of threads on top of a single UNIX process. On a real multiprocessor, these threads would execute concurrently on different processors. During a simulation, control is passed from one thread to another using a context switch. Context switches are typically performed at the end of a local instruction block, after a thread issues a simulator request for a nonlocal operation. In an application with many threads and many nonlocal operations, context switches are very frequent. Time spent to do a context switch is pure simulation overhead.

In our system, a context switch saves the contents of all processor registers for the old thread and loads the values of all registers for the new thread. In some cases, only a subset of the registers needs to be saved or loaded. On a R3000[2], a full context switch saves and restores 65 registers and requires 135 cycles or 5.4 microseconds. By profiling the execution of the simulator with several test programs, we observed that 5% to 15% of the total simulation time for a typical application is spent doing context switches.

A simulation run generates an event file that contains several types of execution statistics. Those statistics are generated "on-the-fly" during program simulation and are written to disk using the `putw` library call. An event file can be subsequently read and interpreted by the data display subsystem (see Figure 3-1). Using `config` (see Section 3.2.2) the user can select the types of statistics to be logged in the event file. After measuring a number of programs with statistics logging turned on and off, we did not observe a significant difference in the total simulation time. Therefore, we conclude that statistics logging incurs a negligible overhead.

The most significant source of overhead is due to the simulation of nonlocal operations. Several thousands of instructions are typically required to simulate an operation that would take only a few cycles on a real machine. Since accurate measurements of the overhead of nonlocal operations are important, we devote the next section to them.

5.2.2 Overhead of nonlocal operations

Our processor model supports two classes of nonlocal operations: Shared memory accesses and interprocessor interrupts (see Section 4.3.1). In this section we describe a number of experiments we performed to measure the overhead of each of these operation classes for different machine configurations and sizes.

Overhead of shared-memory accesses

Shared memory accesses are relatively costly to simulate for a number of reasons. On machines with caches, each memory access requires simulation of the appropriate cache

[2]The MIPS R3000, a RISC processor, is the building block of the DECStation workstation family where Proteus was developed.

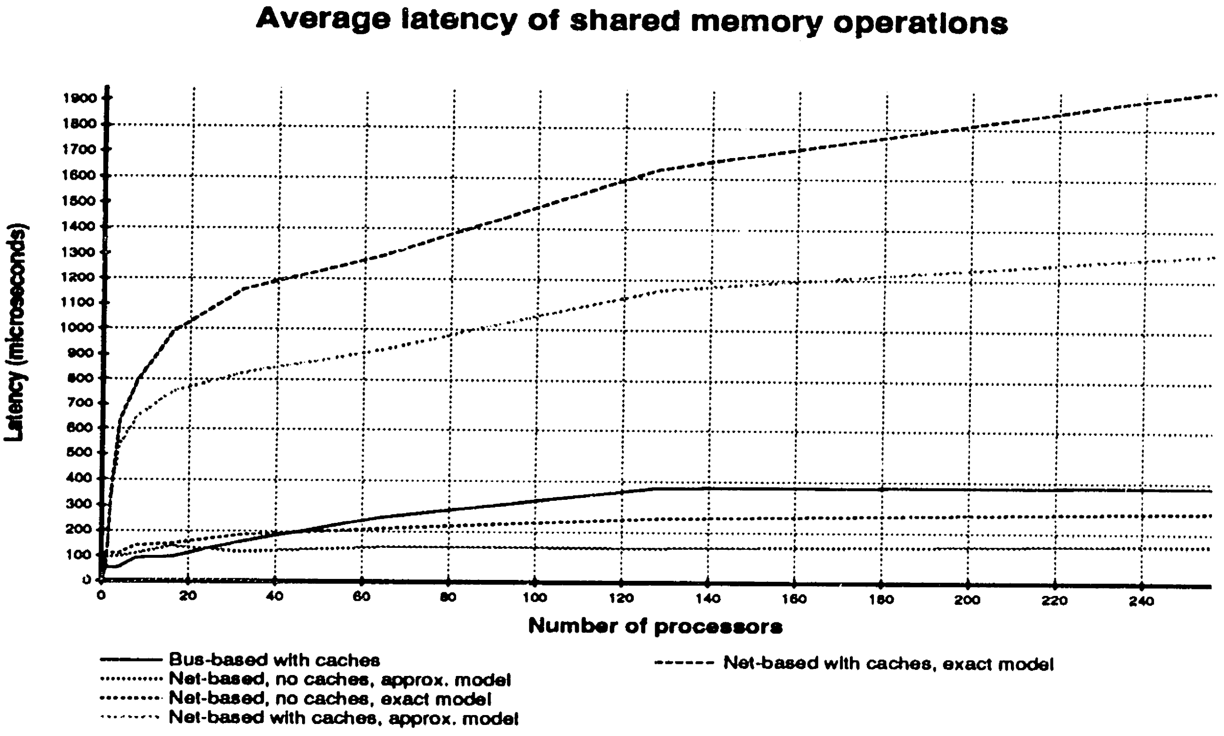


Figure 5-3: Average latency of shared memory operations for various machine configurations and sizes.

protocol. On net-based machines, a memory access may require creation and transmission of a packet through the interconnection network. On bus-based machines access of the common bus must be simulated.

Figure 5-3 plots the average latency of shared memory operations versus machine size for various simulated architectures. The results were obtained by timing the simulation of a program that performed a random mix of 100,000 shared memory operations. The same program, with shared memory accesses removed, was also run to determine the overhead caused by system initialization and other program statements. The difference of the two timings gives the total simulation time needed to perform 100,000 shared memory operations. By dividing that number by 100,000 we got an estimate of the average latency per shared memory operation.

From Figure 5-3, it can be observed that the latency of shared memory operations on bus-based machines increases with machine size. This is due to the nature of the snoopy cache coherence protocol used [Goodman83], which requires checking the state of all other caches on each memory operation and therefore takes time proportional to the number of processors. The original implementation of the protocol exhibited a very bad scaling behavior and latency increased almost linearly with the number of processors.

Since then, the implementation of the protocol has been improved by adding an internal directory for each memory block, which stores the processor indices that currently cache that block. Using the directory, a simulated memory access need only check those caches that are contained in the directory of the accessed block.

As expected, latency for network-based machines without caching scales very well as the size of the machine is increased. When the exact network model is used, a slight increase in latency for larger machines is due to the larger average distance that remote memory access packets have to travel through the network.

Memory accesses on network-based machines with caches are significantly more expensive than on the other three architecture classes. This difference is due to the significant complexity of the simulated cache coherence protocol [Chaiken90]. In this case, a single memory access may cause the generation and transmission of several packets through the network. The number of packets is proportional to the number of processors that have loaded the accessed memory block in their caches. This number increases as the machine size increases. Furthermore, the average distance each packet has to travel through the network increases with the size of the machine. These observations explain the rising slope of the latency curves.

Overhead of IPI operations

Overhead of ipi operations is due to three factors. First, the simulator copies user-supplied ipi packets to internal simulator data structures. Second, each ipi packet has to be routed through the network. Third, upon arrival at the target processor, the simulator creates an interrupt structure based on the packet and inserts it in a queue of pending interrupts for the processor.

Figure 5-4 plots the average latency of ipi operations versus machine size for various simulated architectures. The results were obtained by timing the simulation of a test program that operated as follows: Each processor repeatedly chose a target processor at random and sent it a fixed-size dummy ipi packet. The ipi handler at the other end simply returned the packet to the sender. The total number of packets exchanged in each case was equal to 32,768. As we did in the case of shared memory operations, we subtracted the time needed to initialize the system and execute the other instructions of the program from the timing of the test program, and divided by 32,768 to get an estimate of the average latency per ipi operation.

From Figure 5-4, it can be observed that the latency of ipi operations increases with packet size. This is due to the extra time needed to copy the used-supplied packets

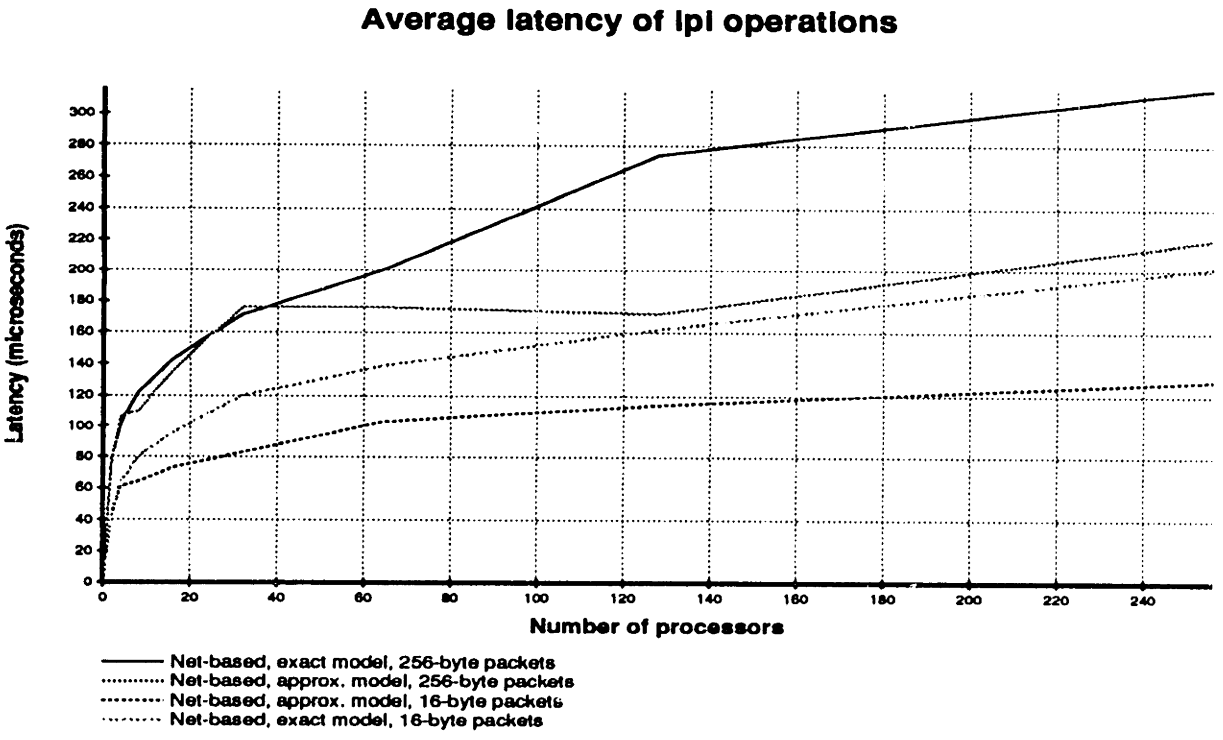


Figure 5-4: Average latency of ipi operations for various machine configurations and sizes.

to internal simulator data structures. It is also interesting to observe the differences in latency between the analytical and the exact network models. The differences are negligible for small machine sizes but increase as the size of the machine increases. On small machines, the average ipi will only travel a small distance through the network. For small distances, the overhead of floating point latency calculations used in the analytical model does not present any advantage over the explicit hop-by-hop routing algorithm of the exact model. For larger machines however, the analytical model exhibits better performance.

5.2.3 Program slowdown

Program slowdown (PS) is defined as the ratio of the time taken to run a program on the simulator to the time needed to execute the same program on a real machine. Program slowdown is the ultimate performance measure of a program simulation system, as it determines which applications can be realistically simulated in a reasonable amount of time. Program slowdown is highly dependent on the specific program being simulated and the selected machine configuration. Programs that make heavy use of the costliest

Application name	Total cycles executed ($\times 10^6$)	Number of shared memory accesses	Number of ipis
Qsort	21-27	0	4-768
Fib	41.5-58	5,000-25,400	0-3,150
8Queens	4-40	152,000-171,000	0-2,000

Table 5.1: Characteristics of test applications used to measure program slowdown. All three measures depend on the simulated machine configuration and typically increase with the size of the machine. Listed ranges are for machines with 1-64 processors (2-64 for Qsort).

operations will exhibit a much worse slowdown than programs that do not.

Program slowdown per processor (PSPP) is defined as the ratio of the total simulation time, in machine cycles, to the total number of cycles executed on all processors of the simulated system (the sum of the number of cycles executed on each processor). PSPP gives a notion of how much an average machine cycle costs on the simulator. PS and PSPP are related according to the formula:

$$(Program\ Slowdown) = (Program\ Slowdown\ per\ Processor) \times (Average\ Concurrency)$$

In order to characterize the performance of our simulator, we measured program slowdown per processor as a function of machine size for three representative programs with different characteristics. Table 5.1 presents a summary of some of the most important test program characteristics.

`Qsort` is an implementation of the Hyperquicksort sorting algorithm developed for hypercube machines [Wagar86]. `Qsort` does not make any use of shared memory and processors communicate relatively infrequently in a nearest-neighbor fashion. The program spends most of its time executing local instruction blocks. `Qsort` belongs to the class of programs that are simulated most efficiently in our system.

Figure 5-5 plots PSPP versus machine size for a program that uses hyperquicksort to sort 65,536 random words. As expected, slowdown is very low (between 4 and 6) and remains relatively stable as the size of the machine increases. It is also interesting to observe that in this particular program, the analytical network model performs slightly worse than the exact hop-by-hop network simulator. This is a consequence of the nearest-neighbor nature of communication in the program. All messages travel exactly one hop. For such small distances, the floating point calculations of latency performed by the

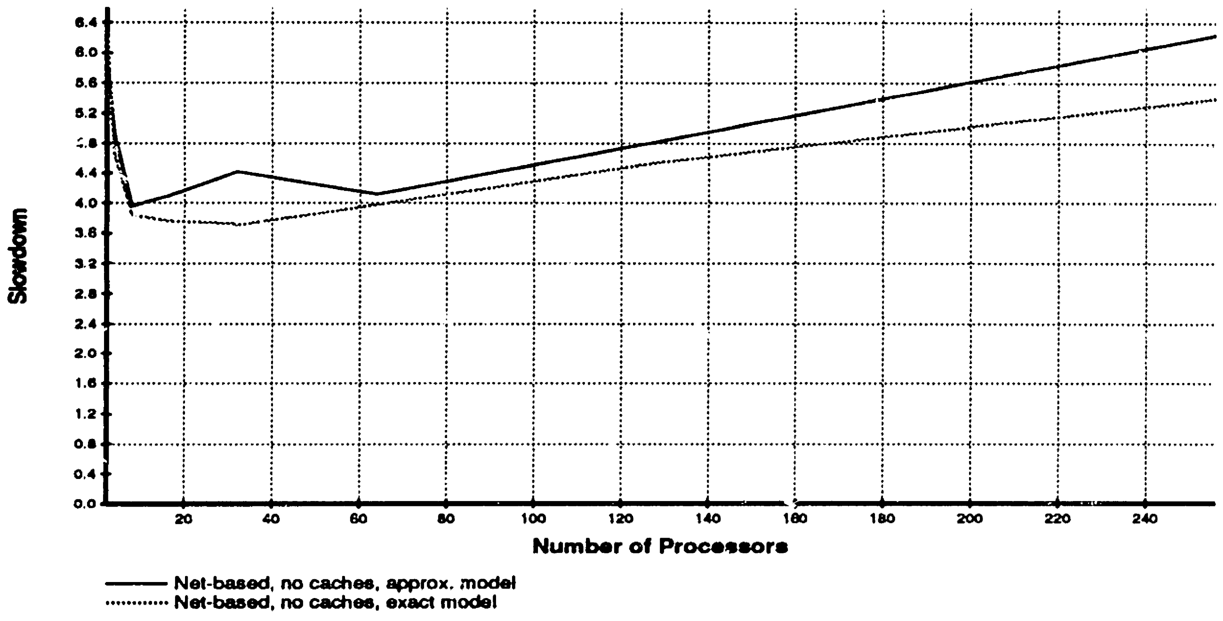


Figure 5-5: Program slowdown per processor for the Qsort program.

analytical network model cost more than the routing algorithm used by the exact network model.

Fib is an implementation of a recursive algorithm to calculate the first 15 Fibonacci numbers according to the formula:

$$fibb(n) = \begin{cases} 1 & \text{if } n < 3 \\ fibb(n-1) + fibb(n-2) & \text{otherwise} \end{cases}$$

Fib spawns a new task to execute each recursive call to *fibb* and uses OS_join to collect the results of the spawned tasks. Fib does not make any explicit shared memory accesses. However, the implementation of OS_join uses a spinlock located in shared memory to synchronize the parent and child thread. Therefore Fib is a program with a large number of very short threads and a lot of spinlock activity.

Figure 5-6 plots PSPP versus machine size for Fib. We observe that all different simulated configurations exhibit comparable slowdown, which ranges between 12 and 22.

8Queens is an implementation of a recursive algorithm to produce all 92 solutions to the 8-queens problem. The algorithm is discussed in more detail in Section 3.4. 8Queens spawns a large number of relatively short-lived threads. It makes heavy use of shared memory and also uses interprocessor interrupts to transmit spawn requests to the target

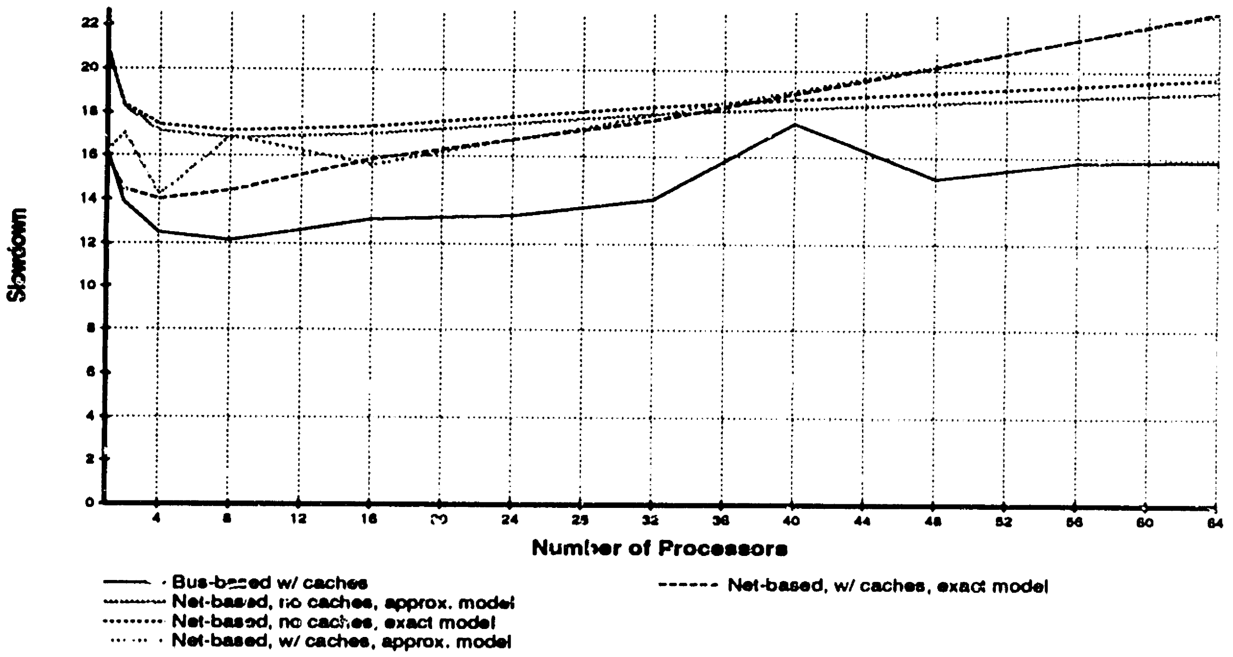


Figure 5-6: Program slowdown per processor for the Fib program.

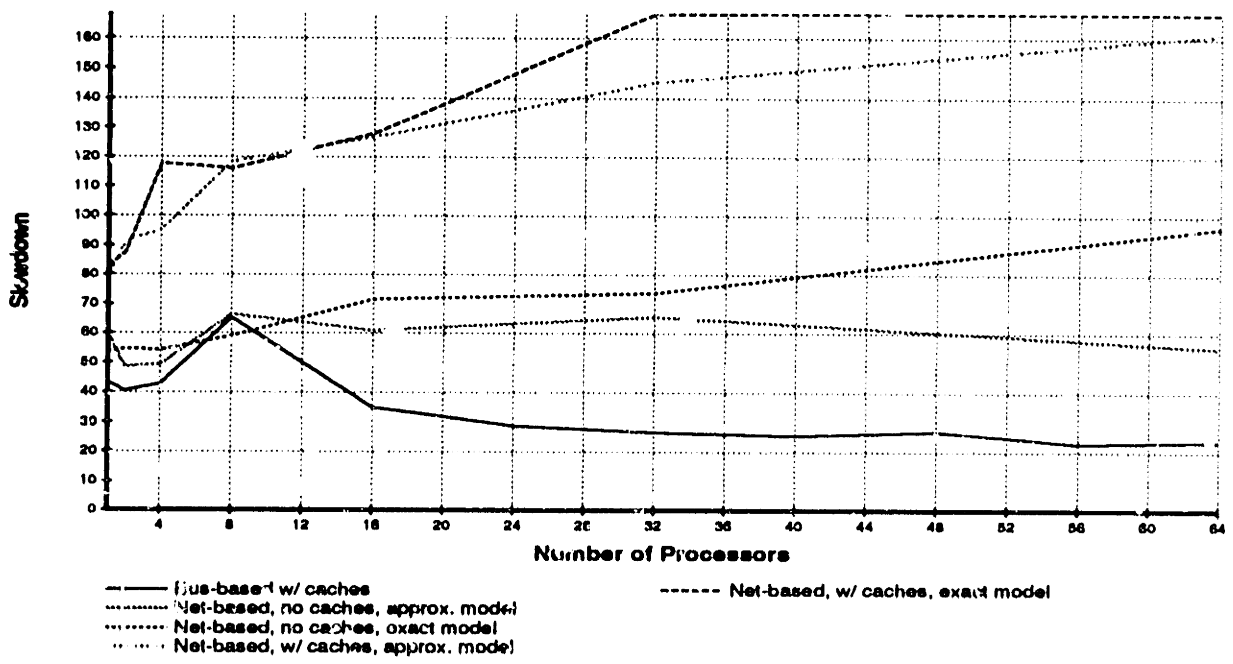


Figure 5-7: Program slowdown per processor for the 8Queens program.

processors. Since it makes intensive use of the operations that are costliest to simulate, 8Queens is representative of the class of programs that are simulated least efficiently in our system.

Figure 5-7 plots PSPP versus machine size for 8Queens. We observe that different machine configurations exhibit large differences in performance. In general, program slowdown correlates strongly with shared memory access latency (Figure 5-3). Thus, network-based machines with caching exhibit the worst slowdown (between 100 and 170), network-based machines without caches come next (slowdown between 50 and 90) and bus-based machine configurations perform best (slowdown between 25 and 60). An apparent paradox is the declining slowdown in bus-based machines as the number of processors increases. This does not correlate well with the results shown in Figure 5-3, where we observed that bus-based shared memory access latencies increase with the size of the machine. To explain the paradox, we recall from Section 3.4.2 that in the case of bus-based machines, bus contention increases rapidly as the number of processors increases. For larger machines, most of the program cycles are wasted waiting to acquire the bus. In our system, contention is modelled very efficiently (see Section 4.3.4). Therefore, in larger bus-based machines a large percentage of the program is simulated very cheaply, which has the effect of reducing overall slowdown.

5.2.4 Comparison with Other Systems

The performance of our system compares very favorably to that of other similar multiprocessor simulation systems. A brief discussion of the systems mentioned in this section can be found in Sections 2.2.1 and 2.2.4.

The authors of CARE report a latency of a couple of milliseconds per simulation event [Delagi90]. Nonlocal operations (message exchanges, shared memory accesses, etc.) typically involve a dozen simulation events. Therefore, nonlocal operations typically take 24 milliseconds to complete. In Proteus, the average cost of a shared memory access is between 100 microseconds and 2 milliseconds (Figure 5-3). This is 12 to 240 times faster than CARE. IPI operations cost between 60 and 300 microseconds (Figure 5-4). This is between 80 and 400 times faster than CARE. There was no information available about average slowdown of programs simulated using CARE.

ASIM [CHL90], the simulator system developed by MIT's Alewife research group, can operate in two modes: The fast mode provides an accurate simulation of processing elements, but does not simulate in detail caching and network transactions. The slow mode provides an accurate simulation of all machine components. When simulating 9Queens, a program similar to **8Queens** on a 64-processor network-based machine, slowdown factors of between 200 and 500 per processor were reported for the fast mode. The slow mode gave slowdown factors of between 1,000 and 5,000 per processor [Nussbaum91]. Depending on the selected machine configuration, Proteus exhibits a slowdown of between 50 and 170 per processor (Figure 5-7). This is a one order of magnitude improvement over ASIM.

Tango [Davis90] reports a fairly comprehensive set of experiments used to measure the performance of the system. As in our system, program slowdown on Tango is highly dependent on the frequency of nonlocal operations issued by the simulated application, the chosen simulator configuration and the size of the simulated machine. Typical PSPP values reported for an efficiently simulated application (Mp3d) were in the range of 2-500. Slowdown times for a less efficiently simulated application (Mincut) were in the range of 700-18,000. In each case, Proteus exhibits two orders of magnitude better slowdown. Furthermore, in Tango, contention simulation is done on a cycle-by-cycle basis. Proteus on the other hand has a very efficient way of simulating contention (see Section 4.3.4).

Chapter 6

Conclusions

The complexity of the interaction between software and hardware in MIMD machines makes experimental evaluation of parallel programs an important complement to theoretical analysis. The performance of parallel programs is subject to significant influence from a large number of implementation, runtime-system and architectural details. Those details are usually hard to predict or model analytically.

In order to test new ideas, evaluate new parallel algorithms, or choose between different implementations of the same algorithm, researchers need a reliable tool that provides them with information and insight about the behavior of their programs on various parallel machines. Our aim in designing Proteus was to provide exactly such a tool.

Proteus evaluates parallel programs by simulating their execution on a variety of parallel architectures. Simulation is an attractive alternative to monitoring the execution of programs on real parallel machines for a number of reasons. Real parallel machines are not always easily available to researchers In addition, traditional techniques used to monitor the direct execution of programs are intrusive and may lead to inaccurate results when applied to parallel programs. Simulation allows flexible, nonintrusive and repeatable evaluation of parallel programs. On the minus side, it usually incurs prohibitive overheads that limit its use to the simplest parallel applications.

Our simulator uses a combination of simulation and direct execution to achieve very high simulation performance. High-performance makes it feasible to evaluate large parallel applications. Local parts of a parallel program, that is, parts that do not require interaction with other components of a parallel machine, are translated into native code and executed directly on the uniprocessor where the simulator runs. Only nonlocal program parts, whose execution requires interaction with other components of the machine, are simulated.

In order to allow users to compare the behavior of a program on different architectures, Proteus can be configured to simulate a large variety of different MIMD architectures. Our architectural model allows users to define target architectures by specifying the type, properties and level of simulation detail for the three main parts of a MIMD machine: processing elements, memory and interconnect. Allowing users to choose the level of simulation detail gives them power to control the simulation accuracy/tradeoff themselves.

An important goal of the system is to provide users with insight that will help them draw conclusions and make decisions about parallel programs. For that reason, a lot of attention has been given to the collection and display of simulation data. During simulation of a user program, the system outputs an event file that captures the important interactions between software and hardware. The system provides a number of predefined event types; the user can easily add other event types specific to his own application, or restrict the recorded event types to those relevant to his experiment. Event files are read by a sophisticated data display program that displays execution statistics in a variety of graphical forms.

In the few months between the release of the first version of Proteus and the time this thesis was written, a number of published experiments were reproduced on our system. In all cases the results obtained using Proteus were very close to those published, a fact that provides evidence for the reliability of the system. Performance measurements demonstrated that our simulator is one to two orders of magnitude faster than other similar multiprocessor simulators.

Several members of our research group are currently using Proteus to evaluate their ideas. They found that Proteus has reduced the time it takes them to set up and perform experiments. Our group has already published a paper on a new algorithm for maintaining concurrent search trees, which was evaluated entirely using Proteus [Colbrook91].

Appendix A

Eight Queens Problem Solution

This appendix contains the listing of the 8-queens solution program that was discussed in Section 3.4. 8Queens is an example of a simple Proteus program. It can give the reader an idea of what the current Proteus programming model looks like.

```
/****************************************************************************
 *                                                                          *
 *  Parallel solution to the eight queens problem                           *
 *  C. N. Dellarocas 1990                                                   *
 *                                                                          *
 ****************************************************************************/

/* user.h contains common definitions of types and constants */
#include "user.h"

#define N 8             /* Number of queens */

/* Target processor to which new threads are assigned is randomly chosen */
#define TARGET_PROCESSOR (random() % NO_OF_PROCESSORS)

/* Chessboard configuration structure */

typedef struct {
          int x[N];     /* x contains queen column positions for each row */
          int a[N];     /* a[i] is FALSE if there is a queen in column i  */
          int b[2*N];   /* b[left_diag(i,j)] is FALSE if there is a       */
                        /* queen at the left diagonal crossing (i,j)      */
          int c[2*N];   /* c[right_diag(i,j)] is FALSE if there is a      */
                        /* queen at the right diagonal crossing (i,j)     */
          } Solution;
```

```
/* Space for the empty chessboard is statically assigned
 * in shared memory module 0
 */

shared Solution NullSolution placeat 0;

#define left_diag(i,j)  ((i)+(j))
#define right_diag(i,j) (N+(i)-(j)-1)

/* It is safe to place a new queen at (i,j) if there is no other queen
 * already placed in column j, nor in either the left or the right
 * diagonal crossing (i,j).
 * @> is equivalent to the standard -> C operator and is used for
 * pointers that point to structures stored in shared memory.
 */

#define safe_to_add_queen(s,i,j)   \
                          ( (s) @> a[j] && \
                            (s) @> b[left_diag(i,j)] && \
                            (s) @> c[right_diag(i,j)] )


/* Create an empty solution configuration */

Solution *empty_chessboard()
{
  int i;

  for(i=0; i<N; i++)
    {
      NullSolution.x[i] = 0;
      NullSolution.a[i] = TRUE;
      NullSolution.b[i] = NullSolution.b[i+N] = TRUE;
      NullSolution.c[i] = NullSolution.c[i+N] = TRUE;
      }

  return( &NullSolution );
}
```

```
/* Create a new configuration with an additional queen */

Solution *add_new_queen(s, row, col)
     Solution *s;
     int row;
     int col;
{
  Solution *news;
  int i;

  /* Allocate a block of shared memory for the new configuration */

  if ( (news = (Solution *)OS_getmemfrommodule(sizeof(Solution),
                                                 CURR_PROCESSOR))
      == (Solution *)NULL )
    fatal("Out of shared memory in function add_new_queen.\n");

  /* Copy old configuration to new */

  for(i=0; i<N; i++)
    {

      news @> x[i]   = s @> x[i];
      news @> a[i]   = s @> a[i];
      news @> b[i]   = s @> b[i];
      news @> c[i]   = s @> c[i];
      news @> b[i+N] = s @> b[i+N];
      news @> c[i+N] = s @> c[i+N];
    }

  /* Add new queen at (row, col) */

  news @> x[row]                  = col;
  news @> a[col]                  = FALSE;
  news @> b[left_diag(row,col) ] = FALSE;
  news @> c[right_diag(row,col)] = FALSE;

  return(news);
}
```

```
/* Choose a TARGET_PROCESSOR and spawn a new thread that executes stage */

#define SPAWN_STAGE( chessboard, i )   \
      { \
        if( OS_spawn( stage, 4, "stage", TARGET_PROCESSOR, USER_PRIORITY, 0, \
                      2, chessboard, i) \
             == ERROR ) \
           fatal("Cannot spawn thread at STAGE i = %d\n", i); \
      }


/* Get a chessboard with i queens and try to place the (i+1)th in all
 * safe columns of row i
 */

stage(chessboard, i)
     Solution *chessboard;
     int i;
{
  int j;

  if (i == N)
    found_solution( chessboard );
  else
    for(j=0; j<N; j++)
      if ( safe_to_add_queen( chessboard, i, j ) )
         SPAWN_STAGE( add_new_queen( chessboard, i, j ), i+1 );
}

/* All Proteus user programs begin execution from function usermain,
 * which is executed by a thread spawned on processor 0
 */

usermain()
{
  stage( empty_chessboard(), 0 );
}
```

Bibliography

[Agarwal86] A. Agarwal, R. L. Sites and M. Horowitz. "ATUM: A New Technique for Capturing Address Traces Using Microcode", *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ACM-IEEE, June 1986.

[Agarwal88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ACM-IEEE, June 1988.

[Agarwal89] Anant Agarwal. "Limits on Network Performance", MIT VLSI Memo 1989.

[Agarwal90] A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz. "APRIL: A Processor Architecture for Multiprocessing", *17th Annual International Symposium on Computer Architecture*, Seattle WA, MAy 1990.

[BBN87] *Inside the Butterfly Plus*, BBN Advanced Computers Inc., Cambridge, MA, 1987.

[Brewer91] Eric A. Brewer. "Aspects of a High-Performance Parallel-Architecture Simulator", Master's Thesis, MIT Laboratory for Computer Science, expected 1991.

[Carpenter87] Robert J. Carpenter. "Performance Measurement Instrumentation for Multiprocessor Computers", Technical Report NBSIR 87-3627, Institute for Computer Sciences and Technology, National Bureau of Standards, August 1987.

[CHL90] David Chaiken, Beng-Hong Lim, and Dan Nussbaum. "ASIM Users Manual", Alewife Systems Memo #13, MIT Laboratory for Computer Science, August 1990.

[Chaiken90] David Chaiken. "Cache Coherence Protocols for Large-Scale Multiprocessors", MIT Laboratory for Computer Science MIT/LCS/TR-489, September 1990.

[Colbrook90] A. Colbrook, C. Smythe. "Efficient implementations of search trees on parallel distributed memory architectures", *IEE Proceedings*, Vol. 137, Pt. E, No.5, pages 394-400 (Sept. 1990).

[Colbrook91] A. Colbrook, E. A. Brewer, C. N. Dellarocas and W. E. Weihl. "An Algorithm for Concurrent Search Trees", To be published in *Proceedings of the 1991 International Conference on Parallel Processing.*

[Covington88] R. C. Covington, S. Madala, V. Mehta, and J. B. Sinclair. "The Rice Parallel Processing Testbed", *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, ACM, May 1988.

[Davis90] H. Davis, S. R. Goldschmidt and J. Hennessy. "Tango: A Multiprocessor Simulation and Tracing System", Computer Systems Laboratory Technical Report CSL-TR-90-439. Stanford University. July 1990.

[Delagi87] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. "An Instrumented Architectural Simulation System" Knowledge Systems Laboratory Technical Report No. KSL 86-36. Stanford University. January 1987.

[Delagi90] Bruce Delagi. *Private Communication.*

[Dubois86] M. Dubois, F.A. Briggs, I. Patil and M. Balakrishnan. "Trace-Driven Simulations of Parallel and Distributed Algorithms in Multiprocessors", *Proceedings of the International Conceference on Parallel Processing*, pages 909-917, August 1986.

[Eggers89] S. J. Eggers. "Simulation Analysis of Data Sharing in Shared-Memory Multiprocessors", Technical Report UCB/CSD 89/501, University of California, Berkeley, 1989.

[Eggers90] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor", *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*, ACM, May 1990.

[Elshoff91] I. J. P. Elshoff. *Aspen.* Ph.D. dissertation to appear early 1991, Computer Science Department, University of Arizona.

[Gait86] J. Gait. "A Probe Effect in Concurrent Programs", *Software - Practice and Experience*, 16(3), pages 225-233 (March 1986).

[Goodman83] J. R. Goodman. "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983, Stockholm, Sweden, pages 124-131.

[Kuck84] D.J. Kuck et. al. "The effects of program restructuring, algorithm change, and architecture choice on program performance", *Proceedings of the Parallel Processing Conference*, August 1984.

[Lawrie75] D. H. Lawrie. "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers*, C-24 (12), pages 1145-1155, December 1975.

[Malony89] A. D. Malony, D. A. Reed, R. A. Aydt, J. W. Arendt, D. Grabas and B. K. Totty. "An integrated performance data collection, analysis, and visualization system.", Technical Report TTR11, University of Illinois at Urbana-Champaign, March 1989.

[Mathieson88] I. Mathieson and R. Francis. "A Dynamic Trace-Driven Simulator for Evaluating Parallelism", *Proceedings of the 21st Hawaii International Conference on System Sciences - Volume 1 (Architecture Track)*, pages 158-166, January 1988.

[MCS90] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", TR90-114, Department of Computer Science, Rice University, May 1990. Also Technical Report 342, Computer Science Department, University of Rochester, April 1990.

[Nussbaum91] Dan Nussbaum. *Private Communication*

[Patel81] Janak H. Patel. "Performance of Processor-Memory Interconnections for Multiprocessors", *IEEE Transactions on Computers*, C-30 (10), pages 771-780, October 1981.

[Pfister85] G. F. Pfister et. al. "The IBM Research Parallel Processor Architecture (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771.

[Rizzo89] Luigi Rizzo. "Simulation and performance evaluation of parallel software on multiprocessor systems", *Microprocessors and Microsystems*, Vol.13 No.1, Jan/Feb.1989, pages 39-46.

[Seitz84] Charles L. Seitz. "Concurrent VLSI Architectures", *IEEE Transactions on Computers*, C-33 (12), December 1984.

[Smith81] B. Smith. "A Pipelined, Shared Resource MIMD Computer", *Proceedings of the 1978 International Conference on Parallel Processing*, Belaire MI, pages 6-8.

[Stunkel89] Craig B. Stunkel and W. Kent Fuchs. "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation", *Proceedings of ACM Sigmetrics 1989*, May 1989, pages 70-78.

[VAX82] *VAX-11 Architecture Reference Manual.* Digital Equipment Corporation, Bedford, MA, 1982.

[Wagar86] Bruce Wagar. "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes", *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 292-299, SIAM, 1987.

[Weinberger84] P. J. Weinberger. "Cheap Dynamic Instruction Counting", *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984.