PROTEUS: A High-Performance Parallel-Architecture Simulator

by

Eric A. Brewer Chrysanthos N. Dellarocas
Adrian Colbrook
William E. Weihl

September 1991

Abstract

PROTEUS is a high-performance simulator for MIMD multiprocessors. It is fast, accurate, and flexible: it is one to two orders of magnitude faster than comparable simulators, it can reproduce results from real multiprocessors, and it is easily configured to simulate a wide range of architectures. PROTEUS provides a modular structure that simplifies customization and independent replacement of parts of architecture. There are typically multiple implementations of each module that provide different combinations of accuracy and performance; users pay for accuracy only when and where they need it. Finally, PROTEUS provides repeatability, nonintrusive monitoring and debugging, and integrated graphical output, which result in a development environment superior to those available on real multiprocessors.

Keywords: Execution-driven simulation, parallel algorithm design and evaluation, parallel architecture, parallel debugging

This is an electronically reproduced version of Technical Report MIT/LCS/TR-516.

This work was supported in part by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation. Eric A. Brewer was supported by an Office of Naval Research Fellowship, Chrysanthos N. Dellarocas by a Starr Foundation Fellowship, and Adrian Colbrook by a Science and Engineering Research Council Postdoctoral Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

This paper presents the design of PROTEUS, a simulator for MIMD multiprocessors. PROTEUS is an execution-driven simulator [CMM+88]; it multiplexes a single processor among the various activities in a simulated parallel machine to provide accurate information about the timing and behavior of an application and the underlying simulated architecture. PROTEUS is fast, accurate, and flexible: it is one to two orders of magnitude faster than comparable simulators, it can reproduce results from real multiprocessors, and it is easily configured to simulate a wide range of MIMD architectures. PROTEUS' modular structure allows users to tradeoff accuracy and performance: users pay for accuracy only when and where they need it. The structure also allows easy customization of the architecture. Finally, PROTEUS provides repeatability and nonintrusive monitoring and debugging, which result in a development environment superior to those available on real multiprocessors.

We believe that simulation has a valuable role to play at all levels of the design and analysis of multiprocessor systems, from architectures to runtime systems to algorithms and applications. Many projects have used simulation during the development of new architectures to guide the design. We believe that simulation has an equally vital role to play in the development of software systems for multiprocessors.

There are two alternatives to simulation: analytical modeling and using real machines. Multiprocessor systems are sufficiently complex that analytical modeling is difficult. On the other hand, using a real machine to test, debug, and tune a program is problematic. In contrast, simulation allows nonintrusive monitoring and debugging, and also makes it easy to repeat executions so that different phenomena in an execution can be studied at a variety of levels of detail.[1] Another important advantage of simulation is flexibility. Using a simulator such as PROTEUS, we can study the behavior of a program on many different architectures. For example, alternative memory systems can be simulated, giving insight into the interactions among applications, compilers, and cache-management techniques. Similarly, the number of processors can be varied, giving insight into the scalability of a program or algorithm (perhaps well beyond the limits imposed by real machines).

For all its advantages, simulation has potential problems in two areas—speed and accuracy—that can make it less useful. First, simulators are often slow, making it impossible to run large experiments or sets of experiments. Second, simulators are often inaccurate, making it difficult to draw useful conclusions from the results of a simulation. PROTEUS is an execution-driven simulator that interleaves the execution of an application program with the simulation of the underlying architecture; this makes it possible to achieve very high accuracy. In addition, PROTEUS avoids interpreting user application code whenever

[1] Some parallel debuggers support repeatability—e.g., Instant Replay [LM86]—but at the cost of maintaining huge trace files and of introducing a significant probe effect [Gai86].

possible, thus removing the overhead of interpretation for most instructions. PROTEUS is also designed so that the entire simulation system, including the application program and the network and memory simulators, runs in a single address space. These and other factors discussed in Section 6 result in a performance improvement of one to two orders of magnitude when compared to other simulators with comparable flexibility such as Tango [DGH91].

Another important feature of PROTEUS is the ability it provides the user to control the level of accuracy of the simulation. In general, there is a tradeoff between accuracy and performance: a more accurate simulation requires more time. Since the level of accuracy desired and the amount of information needed from a simulation depend on the application, PROTEUS provides users with unprecedented flexibility in choosing or customizing the level of accuracy in the network and memory simulations. The user can also control what monitoring data is produced, both for system-level data (e.g., shared-memory traces) and user-level data (e.g., the time spent in a code section, or the size of a data structure). As discussed in more detail below, changing the level of accuracy of the simulation makes a large difference in the running time. For users who need large simulations or sets of simulations, it is important that they be able to pay only for the accuracy they need.

PROTEUS was originally designed for evaluating language, compiler, and runtime system mechanisms to support portability; thus, flexibility, accuracy, and performance are all important. We have also used it for algorithmic and architectural studies, including concurrent search trees and network and cache research [CBDW91]. In general, PROTEUS is an excellent development platform for parallel software: it supports testing and debugging, performance evaluation and tuning, and graphical output.

Section 2 provides an overview of the simulator, Section 3 discusses PROTEUS' modular structure, and Section 4 describes the use of direct execution and augmentation. Support for debugging, monitoring and graphics is discussed in Section 5, while Section 6 evaluates overall system performance. Section 7 presents evidence on the accuracy of PROTEUS: it compares simulation results to published empirical data from an nCUBE multiprocessor. Finally, Section 8 describes related work and Section 9 presents our conclusions.

2 Overview

PROTEUS is not actually a simulator; rather, it is an simulation engine that combines with architecture-specific modules and user applications to create a simulator. The resulting executable provides high-performance simulation of the user's application on the target architecture. This section presents a brief overview of PROTEUS, including the basic multiprocessor model, the programming language, and the steps involved in building and using PROTEUS simulators.

PROTEUS simulates MIMD multiprocessors in which independent processor nodes are connected via an interconnection medium. The interconnection medium can be either a bus, a direct network such as a k-ary n-cube, or an indirect network such as a butterfly. Each processor node consists of a processor, a network chip, a cache chip, and memory. Conceptually, the processor is a generic sequential processor extended with instructions for network access and cache coherence. The network chip interfaces the processor with the interconnection medium. The cache chip, which is optional, handles cache coherence and works with the network chip for remote memory accesses.

The memory at each node is divided into two sections, a *shared* section that maps to part of a global address space, and a *private* section that is not accessible from the interconnection medium. For distributed-memory machines, the size of the shared section is zero. PROTEUS can simulate hardware cache coherence for global memory and provides primitives for software coherence.

Users write applications in a superset of C. The extensions include keywords for declaring that data reside in shared memory and for controlling the placement of data structures. PROTEUS provides library routines for message passing, thread management, memory management, and data collection.

There are four steps in the creation and use of a PROTEUS simulator. First, the user specifies the architecture using an X-based configuration tool. Second, the application- and architecture-specific simulator is compiled and linked into an executable. Next, the user runs the executable to produce screen output and a trace file. Finally, PROTEUS includes a sophisticated X-based graph generator, discussed in Section 5.4, that interprets the trace file and presents the results of the simulation. [2]

3 Modules

PROTEUS was designed with a modular structure to simplify replacement and customization of specific parts of the simulator. The modular structure provides two very important abilities. First, the structure simplifies customizing the target architecture: it is very easy to experiment with part of the architecture while keeping the rest unchanged. This makes PROTEUS useful for evaluating architectural design decisions, and for simulating specific multiprocessors. Second, the modular structure promotes multiple implementations of a given module, which allows users to switch between very accurate versions and very fast versions. Users pay only for what they need; in particular, the high-performance versions greatly reduce development time. This section describes the four most important modules, uses the network module to demonstrate the effectiveness of the structure, and discusses the use of modules to tradeoff accuracy and performance.

The operating system module provides a kernel operating system for the simulated multiprocessor.

[2] All of the graphs in this paper were produced by PROTEUS' graph generator.

The kernel interface specifies procedures for thread scheduling and management, memory management, and interrupt and trap handling. In addition to the kernel interrupt handlers, users may define their own interprocessor interrupts (IPIs) and handlers; for example, user-defined IPIs are used to build dispatch routines for message-passing architectures.

The shared-memory module provides access to local shared memory, handles full-empty bits [Smi81], and provides atomic operations such as test-and-set and compare-and-swap. The shared memory of a remote processor is not accessed directly via the shared-memory module; instead, a network request is generated (usually by the cache module) that invokes the shared-memory module when it arrives at the remote node. Separating the remote access into a network portion and a local-memory portion allows the network and shared-memory modules to be replaced independently.

The cache module handles memory requests from the local processor and from the local network chip. It generates calls to both the shared-memory module (for local accesses) and the network module (for remote accesses). The primary operations provided by the cache module are read, write, and flush. In addition, the module defines operations for software coherence: *soft* read and write, and *fence* [SS87]. The intent of the soft operations is to access the currently cached, possibly stale, data. The fence operation blocks until all pending protocol transactions for the given cache line have completed and is used to ensure coherence for that cache line.

The network module, described in detail in the next subsection, simulates the movement of data within the interconnection medium.

3.1 The Network Module Interface

The network module is a good example of the modular structure of PROTEUS. It demonstrates the two key advantages of PROTEUS' modular structure: the simplicity of customization and the use of multiple versions to provide a range of accuracy and performance. The user must modify only three procedures to replace the network module. The multiple versions, which are discussed in Section 3.2, provide orders of magnitude performance differences depending on the required accuracy. Before discussing the network module, a brief discussion of the simulator engine is in order.

Instructions that affect remote nodes are implemented using simulator *requests*, which are timestamped structures stored in a central priority queue. Such a non-local instruction generates a simulator request and inserts it into the priority queue, which is sorted by timestamp. The engine repeatedly executes the request with the lowest timestamp until there are no requests left, at which time the simulation is complete. Each request type has a associated procedure: the engine executes a request by calling the associated procedure.

The network module uses three types of requests. The first is a *send* request, which signifies that the

Request Generation	Request Execution
`void Send(from, to, time, packet, mode)`	`void send_request_handler(SimRequest)`
`void Route(next, time, packet)`	`void route_request_handler(SimRequest)`
`void Receive(from, time, packet)`	`void receive_request_handler(SimRequest)`

Table 1: Interface for the network module.

processor is ready to send a packet to the network chip. The second type of request is the *route* request, which computes the next node for a packet and computes the arrival time of the packet at that node. Some versions, such as a bus, do not use this request at all. The third type is the *receive* request, which occurs when the packet reaches the target node. The receive request either interrupts the processor or notifies the cache chip depending on the packet. Only the network module generates route and receive requests; all other modules generate only send requests. Table 1 lists the procedures for generating and executing network requests. New versions of the network module only need to replace the procedures for executing network requests.

Typically, the send request generates two requests: one to resume the processor at the appropriate time and a route request to move the packet to the next node or switch. If the network chip uses DMA to get the packet, then the processor is resumed fairly quickly. Other architectures, such as the J-machine [D$^+$89], require that the processor feed the packet to the network chip word by word. In this case, the delay depends on the length of the packet. The `mode` argument is used to pass flags to the module. At the moment, the only flag determines whether or not to interrupt the processor when the DMA completes (assuming the network chip uses DMA).

The route request computes the node to which the packet should be forwarded. For example, in a k-ary n-cube the route request determines which output channel to use, based on the target node, the incoming channel, and possibly the contention on the output channels of the current node. It then computes the arrival time of the packet at the next node, using the current time and information about when the channel will be available. Only the route handler needs to know anything about channels and contention. It the next node is the target, the route handler generates a receive request.

The receive request looks at the type of the packet, which is either a memory packet or an IPI packet. Memory packets are handed to the cache module, which defines a procedure specifically for handling network packets. An IPI packet causes an interrupt of the local processor.

For a specific architecture, it is common to provide additional procedures in the network module that improve the accuracy of the module. For example, the network chip for the Alewife multiprocessor provides a way to check if the chip is busy. We added a procedure that returns true if the channel is busy; we set its cost to four cycles, which is the time it would take to load and check the busy flag.

Using this structure, most network changes, including routing algorithm and topology changes, require modifications to only the route request handler. Most detailed network modules are only a few hundred lines total, and often much of the code can be inherited from existing network modules. The nCUBE network module used for the experiments described in Section 7 took less than a day to implement.

3.2 Trading Accuracy for Performance

Depending on the end goals of a simulation, some modules may have to be very accurate while others can be less accurate. For example, users studying scheduling require very accurate costs in the operating systems module but may not need detailed network simulation. Furthermore, during development, users generally prefer to avoid the lower performance of the most accurate modules. The ability to replace modules provides a simple way to trade accuracy for performance: PROTEUS provides both a very accurate version of a module and a high-performance version with the same semantics but lower accuracy.[3] Currently, the network module and the cache module exploit this tradeoff.

The accurate version of the k-ary n-cube network module simulates the progress of each packet hop by hop. This allows complete simulation of network contention, including hot spots. It correctly simulates uni- and bidirectional edges, end-around connections, internal switch delays, and virtual channels [DS87].

The high-performance version uses an analytical model developed by Agarwal [Aga91]. Instead of simulating each hop, it computes the arrival time at the target using a formula presented in the paper and a contention factor based on a sliding-window view of recent network traffic. This version is acceptable when the traffic is mild. Although the high-performance version has limited accuracy, it is more than an order of magnitude faster than the exact version.

The analytical model used in the high-performance module produces incorrect arrival times both when there are hot spots and when there is no contention at all. As an example of the latter, consider a pipeline application that has high network traffic but no contention. The high traffic leads to a high contention factor, even though none of the packets contend for an edge. Thus the model-based version artificially inflates network delays when there is no contention.[4]

The accurate cache module simulates Chaiken's cache-coherence protocol for direct networks [Cha90]. It simulates all of the cache states and protocol packets. The less accurate module simply provides coherent shared memory by not caching at all: it always goes over the network for remote memory

[3] Versions with intermediate performance and accuracy are possible: the cache module currently provides three versions.

[4] Although easy to see in hindsight, the inaccuracy at zero contention was first noticed in PROTEUS simulations; it was a surprise even to the author of the model.

	Analytical Network Model	Hop-by-hop Network
Uniform Cost	1,500,000	700,000
No Caching	1,000,000	400,000
Coherent Cache	500,000	120,000

Numbers are in simulated cycles per second.

Table 2: This table shows the relative system performance of the six combinations of network and cache modules. The numbers are for the 8-queens application running on an 8x8 mesh. The simulations were run on a DECstation 5000. These numbers vary quite a bit depending on the application and the architecture, but the relative magnitudes are typical.

accesses. Although this increases network traffic, the overall system performance improves substantially. A third version runs even faster: it accesses global memory *directly*, that is, without using the network.[5] It assigns all global memory accesses a single fixed cost. Note that all three versions have the same semantics, the only difference is the cost of accesses.

Table 2 shows the relative system performance of the six combinations of cache and network modules for an 8-queens application running on an 8x8 mesh. There is more than a ten-fold difference in performance between the least and most accurate combinations. Most simulations achieve well over one million simulated cycles per second, since the accuracy is usually not needed during application development.

In summary, the modular structure of Proteus allows easy replacement and customization of individual parts of the simulator. This allows users to tailor Proteus to a particular architecture. We have exploited this ability to reproduce both the nCUBE [FJL+88], a message-passing multiprocessor, and Alewife [A+91], a shared-memory multiprocessor. (Section 7 describes the correspondence between the nCUBE version of Proteus and the real nCUBE.) The modular structure also allows selection of modules based on required accuracy, which allows users to maximize performance for a particular simulation by trading unneeded accuracy for increased performance. In particular, users can exploit more than a ten-fold gain in performance during development by forfeiting detailed simulation of the network and cache. Later, when their code is debugged, they can switch to more accurate modules without modifying their code.

[5] This is possible because Proteus runs in a single address space.

4 Direct Execution

A primary factor in the performance of Proteus is the use of direct execution to provide very low-overhead simulation of most instructions. The key idea is to execute local instructions directly and augment the code with cycle-counting instructions to time the code. This section presents an overview of direct execution with augmentation and discusses the flexibility it provides and the assumptions it requires.

Proteus directly executes *local* instructions. An instruction is local if it only affects the local processor. For example, all register-to-register instructions are local instructions. An instruction that might affect another part of the system is a *non-local* instruction. All shared-memory accesses and network instructions are non-local. Proteus simulates local instructions by directly executing the instruction on the host workstation; non-local instructions are simulated via a procedure call.

Although direct execution provides the correct functionality of local instructions, it ignores the simulated time required to execute them. Proteus uses code augmentation to count the cycles required by local instructions. For each basic block of local instructions, code is added to increment a global cycle counter by the number of cycles required to execute that block. Because the counter is incremented every time a block executes, the counter correctly tracks the required cycles for any path through the control-flow graph.

The use of direct execution with augmentation was used first by Mathieson and Francis [MF88] and by Covington et al. [CMM$^+$88]. The technique has been used in several other simulators [DGH91, Che89, SF89]. We extend the work in this area in three ways. First, Proteus provides support for nonintrusive monitoring, which is discussed in Section 5.1.

Second, profiling information, similar to the Unix tool `prof` [DECb], can be generated by using a procedure-specific cycle counter in addition to the global cycle counter. This produces very accurate counts of the simulated cycles spent in each procedure. As with `prof`, the profiling information guides tuning and aids debugging. Unlike `prof`, which uses periodic sampling to collect profiling data, Proteus profiling data is exact.

Third, we use augmentation to limit the number of cycles a single thread can execute without returning control to the simulator engine. This limit, called the *quantum*, keeps processors close together in simulated time. Normally, processors are kept close together simply because they perform non-local instructions, which always return control to the engine. However, without the quantum, loops containing only local instructions can cause a thread to get thousands of cycles ahead. This affects arriving interrupts, which may get artificially delayed thousands of cycles. The quantum also prevents infinite loops in user code from hindering debugging: since the simulator regularly regains control, the user can enter debugging mode and easily determine which processors and which procedures are in

Program	Normal Cycles	Augmented Cycles	Overhead Factor
`Queue`	65,876,631	144,581,647	2.2
`Sieve`	52,483,384	130,868,590	2.5
`augment`	11,670,316	24,578,648	2.1
Minimum ASIM overhead			200

Table 3: Measuring the overhead of augmentation. This table compares several sequential programs with and without augmentation. The cycles were determined by `pixie` [DECa], a profiling tool available on MIPS-based workstations. The overhead factor is the ratio of the `pixie` cycle count for the augmented version over that of the normal version. The overhead is consistently a small factor. ASIM is a multiprocessor simulator developed for the Alewife project at MIT [A+91, CLN90]; it is representative of instruction-interpreting simulators.

infinite loops.

The simulation overhead incurred by code augmentation is much lower than that incurred by instruction interpretation, which is used in most processor simulators. Table 3 shows the overhead due to augmentation for three sequential programs. As discussed by Davis et al. [DGH91], the overhead for augmentation is about a factor of two, which is about one hundred times lower than the overhead for instruction interpretation. Unfortunately, these numbers only apply for local instructions; non-local instructions must still be interpreted. Thus the overall performance of Proteus, which is discussed in Section 6, is rarely one hundred times faster than instruction-interpreting simulators.

The hundred-fold performance gain for local instructions does not come for free. Using direct execution with augmentation requires several assumptions that are not required by simulators that interpret every instruction. First, because Proteus determines the cost of each basic block at compile time, the cost of a block is a fixed number of cycles. In reality, the cost of an instruction depends on cache hits and sometimes on the operands. Thus, we use the expected cost of the instruction, taking into account both the expected number of cycles for the instruction and the expected delay due to cache misses. In essence, we assume uniform cache hit rates for instructions and data in private memory. (Shared-memory accesses are simulated in detail and thus avoid this assumption.) This assumption is reasonable because uniprocessor cache hit rates are very high, and because small periodic errors in instruction costs rarely affect overall simulation results.

A second and related assumption is that code and stacks reside in private memory. If code resides in shared memory, Proteus must simulate the cache-coherence protocol for every instruction fetch, which removes most of the performance benefit of direct execution. Likewise, if stacks reside in shared memory, every stack access must be simulated in detail, which again results in a severe loss of performance. Section 8 discusses future plans regarding this assumption.

The errors due to these assumptions are small and localized; in practice, they have had negligible effect. Section 7 compares PROTEUS results with those of real multiprocessors; for these applications, our assumptions are validated.

5 Monitoring and Debugging

In addition to performance, a primary asset of PROTEUS is its support for monitoring and debugging. PROTEUS provides *nonintrusive* monitoring and debugging: users can add monitoring code that does not affect the behavior or timing of the simulation. PROTEUS also provides repeatability: users can rerun simulations to pinpoint bugs. Real multiprocessors generally provide neither of these abilities.

Because PROTEUS runs as a single process, it works well with sequential debuggers such as `dbx` [Lin90]. This extends the power of advanced sequential debuggers to the parallel development arena. Furthermore, PROTEUS provides an internal debugging mode that allows users to examine the states of threads, processors, locks, and memory. Combining the PROTEUS debugger with a sequential debugger such as `dbx` results in a very effective development environment.

PROTEUS also provides an integrated subsystem for data collection and display. Data collection is supported by primitives for recording data to a trace file and by user-defined data types. Data display is performed by an X-based graph program that uses a simple but powerful graph language to interpret the trace file data.

This section examines PROTEUS' support for nonintrusive monitoring and discusses repeatability and nondeterminism. It then examines the primitives for data collection and concludes with a discussion of the graph-generation program.

5.1 Nonintrusive Monitoring

Nonintrusive monitoring, combined with repeatability, greatly simplifies the development of concurrent programs. Real multiprocessor systems suffer from the *probe effect*: the addition of monitoring code may cause the monitored effect to disappear [Gai86]. This prevents programmers from collecting additional data for debugging. PROTEUS allows users to add arbitrary monitoring or debugging code without changing the behavior of the simulation.

For non-cycle-counted code, the addition is trivial. Since the cost of the code is not determined by cycle counting, the monitoring code does not affect the cost, which ensures no change in behavior.[6]

[6] The monitoring code may alter costs if desired, but this is unusual since it could change the system behavior.

Thus for engine code and most architectural modules, the addition of nonintrusive monitoring code is straightforward.

Adding nonintrusive code to cycle-counted code can be more difficult. In this case, a simple addition *will* change the behavior since the cost of the code increases. To resolve this problem, PROTEUS allows users to turn off cycle counting temporarily within cycle-counted code. Thus, a typical nonintrusive addition would first turn off cycle counting, then add the extra code, and then turn on cycle counting.[7]

It is conceivable that even with cycle counting turned off, the addition may change the behavior of the application. This is because the additional code may affect the surrounding code indirectly. For example, if the additional code uses several registers, the surrounding code may spill more registers than the previous version. This would increase the cost and thus could change the behavior of the system.

We have *rarely* observed this problem in practice; the addition of monitoring code to cycle-counted code has not caused the effects being studied to disappear. Should it occur, however, it is possible to adjust the cost of the monitored code so that it matches the cost it had prior to the addition. PROTEUS provides primitives for increasing and decreasing the cycle counter by a delta, so it is easy to subtract out the extra cycles due to the monitoring code.[8] Section 8 discusses future work on nonintrusive monitoring.

5.2 Repeatability

Nonintrusive monitoring is only useful if the platform ensures repeatability: the whole point of nonintrusive monitoring is to allow repeatability in the presence of additional code. Repeatability is perhaps the single most important feature of PROTEUS; its presence provides a debugging environment that is not available on real multiprocessor systems.

Nondeterministic systems, such as multiprocessors, rarely provide any form of repeatability; some bugs may occur only once every ten thousand executions. For deterministic programs, such as PROTEUS, repeatability is the rule rather than the exception. Thus, PROTEUS simply extends the repeatability inherent in sequential programs to multiprocessor applications.

Given that PROTEUS is deterministic, it might seem reasonable to assume that it can reproduce only one of the many possible executions of a fundamentally nondeterministic application. In fact, however, PROTEUS can reproduce multiple executions of a nondeterministic application, an ability unique to PROTEUS among multiprocessor simulators. The multiple executions arise because PROTEUS chooses randomly between two requests with the same timestamp; PROTEUS views two such requests as a race

[7] Turning on and off cycle counting is done with macros that allow nesting; it is legal to embed non-cycle-counted macros into code that already has cycle counting turned off.

[8] The number of extra cycles can be determined by looking at the assembly code or by printing out the cycle counter with and without the change. Guessing a small number would probably work as well, since the cost only needs to be accurate enough to prevent the monitored effect from disappearing.

condition. A pseudo-random number generator is used to decide the race condition; this provides the determinism required for repeatability. At the same time, using pseudo-random numbers implies that changing the seed changes the outcome of some of the race conditions and thus leads to a different execution of the same nondeterministic application.

For most applications, the ability to reproduce multiple executions is not critical. However, some applications, such as concurrent branch-and-bound search algorithms, exhibit vastly different behavior depending on the outcome of race conditions. In the case of a concurrent search algorithm, the ability to investigate multiple executions allows a researcher to collect a distribution of execution times, which provides a much more accurate view of the effectiveness of the algorithm. As expected, some PROTEUS applications exhibit a wide distribution of execution times when the random number seed is varied.

5.3 Data Collection

The ability to collect exactly the desired data greatly enhances the usefulness of simulation. PROTEUS provides a framework for generating trace file data that allows users to generate their own data in addition to the statistics collected by the engine and the modules.

The simulator uses two basic kinds of data, time-dependent and time-independent. The time-dependent data records, called *events*, include a value, an index, and a timestamp. For example, a concurrency graph can be generated using events: each point is an event consisting of the number of busy processors and the timestamp.[9] Any graph that plots something versus time uses events. The index field is used when generating data for a set of event versus time graphs; Figure 2 is an example.

Time-independent data records, called *metrics*, summarize one aspect of a simulation with a single value. For example, the execution time is a metric. An *array metric* is simply an array of metrics. Processor utilization graphs, for example, use an array metric with one metric for each processor. Metrics are often used to compare the results of several simulations. For example, the nCUBE graphs in Section 7 plot execution time versus the number of processors; each point is a metric from one simulation.

In addition to several predefined data types, users can define their own event types and metrics. The interpretation of user data is specified in a simple graph language used by the graph generator. User-defined data types allow researchers to generate high-quality application-specific graphs in very little time. Typically, it takes only a few minutes to define a new event type and specify the interpretation using the graph language.

[9] In practice, we use two events for concurrency graphs, one that indicates a processor became busy and a second that indicates a processor became idle. The index field contains the processor number. This allows us to determine exactly which processors are busy; recording the count directly hides this information.

```
Map state_map { "Compute" 0, "Send" 1, "Receive" 2 }

ArrayGraph state (p, 0, NO_OF_PROCESSORS - 1) {
     menu <- "Processor State",       ; name in menu
     usemap <- state_map,             ; name-value map
     x_axis <- "Time",                ; x-axis label
     y_axis <- "Processor",           ; y-axis label
     action {
          EV_STATE: VALUE(p)          ; use the value of events with index p
     }
}
```

Figure 1: Graph specification for a processor state graph. The `Map` statement defines a set of name-value pairs. The `ArrayGraph` keyword indicates that this is an array of event versus time graphs: the local variable `p` iterates over the valid processor numbers, and the resulting graph has one timeline for each processor. The `action` clause says to ignore all but `EV_STATE` events; the `VALUE` action sets the y value to the value of the event. The "`(p)`" notation indicates that the index field of the event must match the current value of `p`, so that only events from the relevant processor affect the timeline for this iteration. A graph with this specification appears in Figure 2.

5.4 Graphics

PROTEUS provides integrated graphics capabilities that are not available with comparable simulators and are often not available with real multiprocessors. PROTEUS' graphics capabilities make it simple to evaluate algorithms and architectures: users can quickly create graphs that answer their key questions and provide new insight. The key is a simple but powerful graph-specification language that tells the graph generator how to interpret the trace file.

The data for the graphs comes from the events and metrics stored in the trace file. An individual graph specification gives meaning to the events and metrics by determining which events and metrics are relevant and by specifying how to build a graph from the relevant elements. Figure 1 shows a typical graph specification. Like most graph specifications it is simple and very short.

The graph generator produces line graphs, bar graphs, and tables, and can combine multiple graphs onto the same axes. It can also merge data from multiple simulations; this simplifies comparison of an algorithm across a range of architectures, machine sizes, or other architectural parameter. The generator uses the X Window system and can produce PostScript hardcopy. It can also produce PostScript files for inclusion in documents such as this paper.

We have found the ability to create new graphs quickly to be an excellent debugging aid. The most effective approach is to graph the state of each processor versus time and then combine all of the timelines into one graph. Defining new event types, adding the data collection statements, and

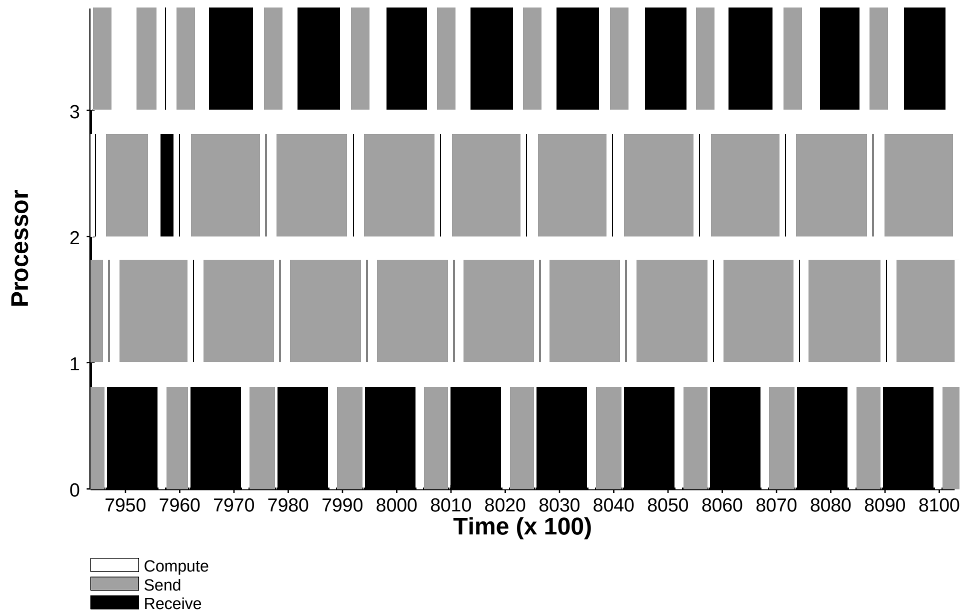

Figure 2: This graph shows the state of four processors in a pipeline search tree application [CBDW91]. The state graph is generally periodic; the width of the period reveals the throughput of the pipeline. A single operation has a "slope" of about 60 degrees from the positive time axis: the pipeline latency can be measured directly from this slope. More importantly, this graph reveals that this particular algorithm is spending all of its time performing communication: very little of the graph is white. A change to buffered asynchronous message passing resolved this problem.

specifying the graph typically require a total of about ten minutes. Many interesting effects are visible on these graphs including livelock and deadlock. Excessive lock-holding times are readily apparent, as are violations of mutual exclusion. In addition to debugging, these graphs are useful for program tuning since they indicate how long different states last. Figure 2 shows one of these graphs.

The data collection and display subsystem gives PROTEUS a unusual level of effectiveness. Users can collect and display the data they need to answer their questions. The support for user-defined data collection and user-specified graphs gives users of PROTEUS full access to the insight available through simulation.

6 Performance

PROTEUS substantially outperforms comparable multiprocessor simulators. By providing one to two orders of magnitude improvement in performance, PROTEUS allows researchers to investigate applications and machine sizes prohibited by the performance of other simulators. Table 4 summarizes the performance of three multiprocessor simulators.

Simulator	Program Slowdown Per Processor	
	Best	Typical
Proteus	**2**	**35-100**
ASIM	200	1,000-5,000
Tango	2	500-2,000

Table 4: Overall system performance for several multiprocessor simulators.

The ASIM simulator [CLN90], which was developed for the Alewife project at MIT, is a fairly representative instruction-interpreting simulator. The overhead of instruction interpretation is reflected in the "Best" column of Table 4, and limits the typical performance substantially.

Tango [DGH91] is very similar to Proteus in its use of direct execution with augmentation. Thus, its peak performance has an overhead factor of about two. The typical performance, however, is far worse than that of Proteus. This seems surprising, since Tango has similar overhead for augmentation. In practice, augmentation overhead is an insignificant part of simulation overhead; simulating non-local instructions and context switching dominate the cost of simulation. It is in these areas that Proteus outperforms Tango.

Tango uses Unix processes for each simulated thread, which results in a context switch time of 180 to 250 microseconds according to the authors. Proteus uses a custom lightweight-threads package that provides context switching times of about 3 microseconds. Even with lightweight threads, context switching accounts for several percent of the total running time; thus, using Unix processes would greatly reduce the performance of Proteus.[10]

Proteus' lightweight threads exploit "partial" context switches if the switch occurs at a procedure call boundary. Invariants hold at procedure boundaries that limit the amount of context that must be saved. Because we use procedures to implement non-local instructions, it is quite common to switch at procedure call boundaries; typically, 98% of all context switches involve the limited context.[11]

Tango uses Unix semaphores for synchronization, which further limits performance. The semaphores used in Proteus are significantly faster. In addition, Proteus simulates spinning by internally blocking the spinning thread, but still generating the correct network traffic. This allows Proteus to simulate spinlock contention without suffering from contention delays itself. Tango performance drops an order

[10] The authors of Tango are developing a version that uses lightweight threads; its performance should be much more competitive.

[11] Because of the different size contexts, we must save the size of the context to avoid excess copying when the context is restored.

of magnitude in the presence of high contention.

There are also indirect performance benefits from running in a single address space, such as reduced memory requirements and direct access to all parts of the simulator. In particular, the global priority queue, which is accessed for every non-local operation, has a tuned implementation that provides access speed that would not be possible with multiple Unix processes.

All of these decisions combine to give Proteus a level of performance that is consistently at least an order of magnitude faster than other multiprocessor simulators. During application development, the performance is typically two orders of magnitude better due to the performance-accuracy tradeoff provided by Proteus' modular structure.

7 Validation

This section compares Proteus' results with published results from a real multiprocessor. If the simulator produces valid data, then its results should match those of the real multiprocessor. We have used published results to validate Proteus several times; here we reproduce results from a comparison of sorting algorithms on an nCUBE multiprocessor.

The nCUBE is a message-passing multiprocessor with a hypercube topology; that is, there are 2^n processors with each processor connected to n other processors. Communication is in the style of CSP [Hoa85]: every send must have a matching receive. The primitives transfer data blocks via DMA over the network to the target processor. There is no cache coherence.

The algorithm comparison comes from Quinn's paper "Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge" [Qui89]. Quinn compares two sorting algorithms on a 64-processor nCUBE/7. Both algorithms mix local sorting with communication; they differ in their strategies for dividing the values among the processors. In general, quickmerge requires fewer but larger messages than hyperquicksort.

Figure 3 graphs Quinn's hyperquicksort times along with times for the nCUBE version of Proteus and a version with a generic network module. The nCUBE version provides procedures that implement the nCUBE communication primitives and uses costs adjusted to reflect the actual communication costs of the nCUBE, which are much higher than those assumed by the generic network module.[12]

Since we use direct execution, all of the local sorting compiled to MIPS code, not nCUBE code. The differences in local instructions and compilers implies that we must scale Proteus cycle counts to correspond to nCUBE seconds. For the hyperquicksort graph, we simply picked the scaling factor that

[12] Thanks to David Culler at the University of California at Berkeley for providing nCUBE timing data.

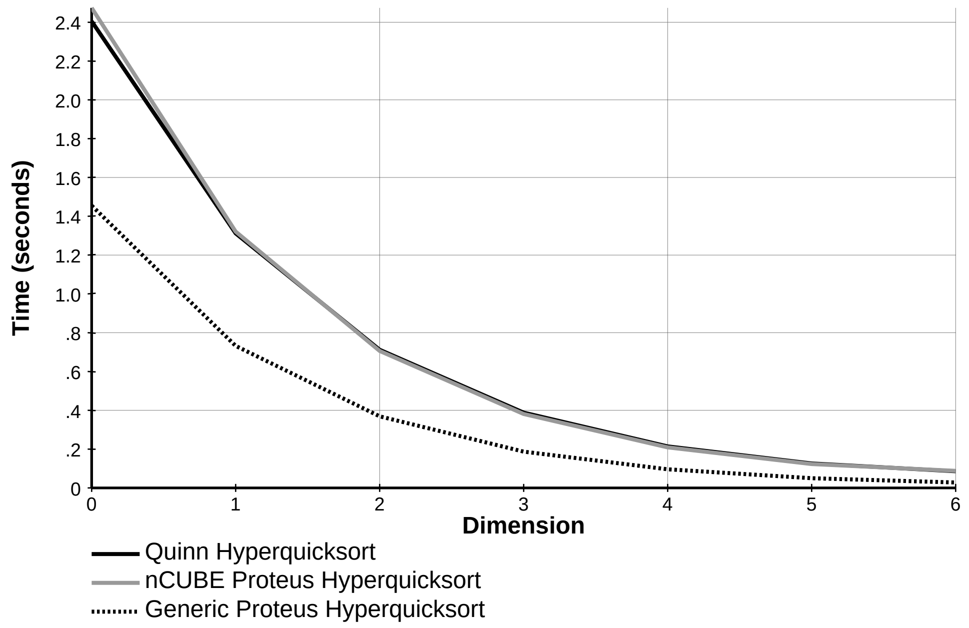

Figure 3: Hyperquicksort Times

provided the best match; thus, for hyperquicksort (only) the match between Quinn's data and our data is deceptively good.

The scaling factor, however, should be independent of the application, so we used the same scaling factor for quickmerge. Figure 4 graphs Quinn's results and PROTEUS' results for quickmerge. The key point is that although the hyperquicksort data has been scaled to fit, the quickmerge data has not: we first established the ratio of PROTEUS cycles to nCUBE seconds, *then* we ran the quickmerge simulations. The fact that the quickmerge data matches Quinn's data well validates both the scaling factor and the nCUBE version of PROTEUS as a whole. Figure 5 presents a different view of the quickmerge data; the data has been normalized to Quinn's results so that the error in individual PROTEUS points is more visible.

The nCUBE PROTEUS results match the published results extremely well, especially when compared to the generic network module. The modifications for the nCUBE version took less than one day to implement, but resulted in substantially more accurate simulations – these facts confirm the importance of the modular structure. Further refinements would improve the accuracy of the nCUBE version, but the first order modifications were sufficient to obtain results consistently within four percent.

Evidence for the accuracy of PROTEUS comes from other sources as well. In our research on concurrent search trees [CBDW91], we found that PROTEUS was able to reproduce published search tree

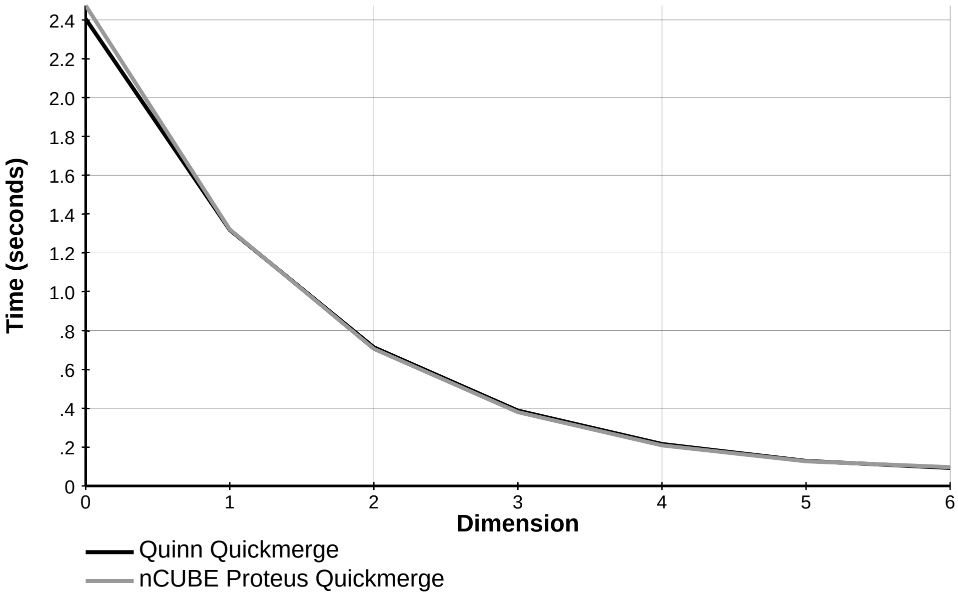

Figure 4: Quickmerge Times

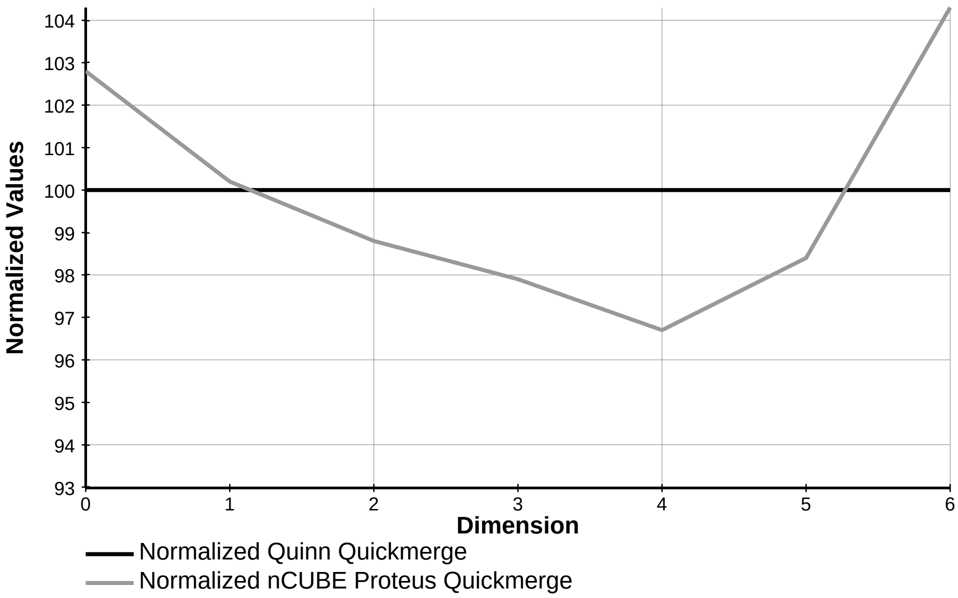

Figure 5: PROTEUS error in quickmerge times. The data has been normalized to Quinn's data to clarify the error in the PROTEUS results.

results [CS90] that were measured on a Supernode multiprocessor [Nic88].

PROTEUS also reproduced the results published in "Synchronization without Contention" by Mellor-Crummey and Scott [MCS91]. This paper compared locking algorithms on both a Sequent Symmetry and a BBN Butterfly.

In general, any effect that we expected to see has actually appeared. More importantly, all unexpected results have (so far) proven to be real effects rather than inaccuracies introduced by PROTEUS. For example, we noticed excessive communication problems in David Chaiken's cache-coherence protocol that severely hindered performance [Cha90]. In his thesis, Chaiken predicted the possibility of cache thrashing, but he did not know if it would be a problem in practice. The solutions he suggested resolved our problem, confirming that the excessive communication was due to thrashing in the cache. The thrashing problems and solutions were confirmed by Chaiken's own simulations using ASIM [CLN90].

8 Related Work

Augmentation was first used to profile sequential programs by Weinberger [Wei84]; direct execution with augmentation for multiprocessor simulation was developed by Mathieson and Francis for their *Threads* simulator, and by Covington et al. for the Rice Parallel Processing Testbed (RPPT) [CMM+88], and is used in several simulators [DGH91, Che89, SF89]. Section 4 discusses our extensions to this work.

Among these simulators, only the RPPT provides substantial support for debugging. It provides some form of "parallel debugger/tracer" that interprets and controls the simulation. In contrast, PROTEUS was designed to work well with sequential debuggers in addition to providing a debugging mode that interprets the state of the simulation and allows single stepping. Debugging in PROTEUS is simple and straightforward, primarily because we support sequential debugging techniques.

The support for integrated data collection and display is unique to PROTEUS among execution-driven simulators, although Tango provides some form of general monitoring. The CARE simulator [DSNB87], which simulates LISP code using direct execution and a hardware timer, provides integrated monitoring and graphics. The TESS simulator [Sta85], a commercial discrete-event simulation system, provides very general data collection and display abilities, but is not very useful for multiprocessor simulation.

The modular structure of PROTEUS extends the separation of functionality introduced by Tango. In Tango, it is easy to replace the memory system simulator as a whole, but the cache, network, and memory systems cannot be replaced independently. The RPPT provides several architectural models, but does not seem to support customization or independent replacement.

The ability to trade accuracy for performance is exploited to a small extent by Tango, which provides multiple versions of the memory system. PROTEUS makes this tradeoff a fundamental part of the simulator. It provides multiple implementations of modules, and also provides several parameters, such as the quantum, that tradeoff accuracy and performance.

The ability of PROTEUS to reproduce published results provides a level of confidence in simulation results that is absent in published results about comparable simulators. The execution-driven simulation literature makes no attempt to reproduce results from real multiprocessors.

PROTEUS also extends the performance of execution-driven simulation by combining simulation and analytical models. The use of Agarwal's network model as the base of one of the network module implementations provides more than an order of magnitude increase in performance in network simulation. Although simulation is always based on some model, our use of analytical models is novel in that we make no attempt to simulate what actually occurs in the network. Instead, we merely attempt to compute the correct costs for network operations. We believe that the explicit use of analytical models has an important place in the tradeoff between performance and accuracy: when used *within their limits* they provide tremendous performance and sufficient accuracy.

8.1 Future Work

One of the primary limitations of PROTEUS is the restriction that code and stacks reside in private (local) memory. This assumption prevents PROTEUS from having to simulate cache effects for every instruction fetch and stack access. Although removing this restriction would greatly reduce the performance of PROTEUS, we would like to offer the increased accuracy as an option.

We would probably simulate the cache effects on a basic-block basis; that is, each block would be augmented with calls that simulate the cache effects for the instruction fetches and stack accesses in that block. The implementation is complicated by the dynamic nature of the addresses: some of the addresses cannot be determined statically.

We would also like to provide some form of virtual-memory simulation. Although most research multiprocessors do not use virtual memory, many of the smaller commercial machines do.

Finally, we hope to implement fully nonintrusive debugging. As described in Section 5.1, there are some cases in which the "nonintrusive" code indirectly affects the monitored code, usually by changing register allocation. We can eliminate these effects by automatically setting the cost of the monitored code to its value before the monitoring was added. Thus, the augmentation program would read the previous version of the monitored code to obtain the correct costs, then it would adjust the cost of the new version to be identical, which makes the monitoring code truly nonintrusive. Since the current approach works most of the time, and users can adjust the costs themselves in the cases that fail, this change has lower priority than the others.

9 Conclusion

PROTEUS provides a unique combination of flexibility, performance, and accuracy. Its modular structure simplifies customization and independent replacement of individual parts of the simulator; this promotes modules for particular architectures and multiple implementations that provide a variety of performance and accuracy combinations. The division into independent modules also clarifies and simplifies each module, which makes it easier to tune performance.

The overall performance of PROTEUS is typically an order of magnitude higher than comparable simulators; this is due primarily to the use of direct execution, a high-performance lightweight-threads package, and efficient simulation of synchronization. When the high-performance versions of modules are in use, which is typical during development, the system performance increases an *additional* order of magnitude over other multiprocessor simulators.

The accurate versions of modules allow PROTEUS to reproduce published results; we have performed such validations several times in addition to the experiment described in Section 7. The validation experiments provide a significantly increased level of confidence in PROTEUS' results. In general, every effect that we expected to see has actually appeared, and every unexpected effect turned out to be real.

The primary use of PROTEUS so far has been the design and implementation of a portable parallel language and runtime system. It has also been used for research on concurrent algorithms, operating system network overhead, and fault tolerance. The fault tolerance application consists of roughly 10,000 lines and runs for hundreds of millions of cycles.

PROTEUS provides several key features that make it an exceptional platform for research on parallel systems:

Flexibility: PROTEUS can simulate a wide variety of MIMD multiprocessors, including both shared-memory and message-passing machines.

Performance: PROTEUS' performance allows simulation of applications and machine sizes that are prohibited by other simulators.

Performance/Accuracy Tradeoff: By providing only the required accuracy, PROTEUS maximizes performance; this allows exceptional performance during development since users can simply switch to accurate modules when needed.

Repeatability: PROTEUS provides repeatability, which is critical to quality debugging, but rarely available on real multiprocessors. It lets users rerun simulations until they have pinpointed a problem.

Nonintrusive Monitoring: To ensure repeatability despite the presence of additional monitoring code, PROTEUS allows users to add nonintrusive monitoring code. This allows users to gain more information without causing an effect of interest to disappear due to changes in timing.

Use of Sequential Debuggers: PROTEUS is designed to work well with standard debuggers such as `dbx`; this brings the power of advanced sequential debuggers to parallel software development.

Data Collection: Users can collect exactly the data they need, including user-defined data types.

Graphical Output: A simple but powerful graph-specification language allows users to create application- or architecture-specific graphs quickly and easily.

Availability: PROTEUS allows parallel-systems research to take place on standard workstations, thus avoiding the cost and limitations of real multiprocessors.

We believe that these advantages will make PROTEUS (and tools like it) a fundamental part of parallel-systems research—the flexibility and the ease of development are not available on real machines. PROTEUS' high-quality development environment, combined with its flexibility, accuracy and performance, produce not only a high-performance simulator, but a powerful tool for parallel research and development in general.

References

[A+91] A. Agarwal et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared-Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[Aga91] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[Bre91] E. A. Brewer. Aspects of a high-performance parallel-architecture simulator. Master's thesis, Massachusetts Institute of Technology, December 1991.

[CBDW91] A. Colbrook, E. A. Brewer, C. N. Dellarocas, and W. E. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, pages III138–III141, August 1991.

[Cha90] D. Chaiken. Cache coherence protocols for large-scale multiprocessors. Technical Report MIT/LCS/TR-489, MIT Laboratory for Computer Science, September 1990.

[Che89] D.-K. Chen. MaxPar: An execution-driven simulator for studying parallel systems. Technical Report CSRD 917 and UILU-ENG-89-8013, University of Illinois, October 1989.

[CLN90] D. Chaiken, B.-H. Lim, and D. Nussbaum. ASIM Users Manual. ALEWIFE SYSTEMS MEMO #13, August 1990.

[CMM+88] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988.

[CS90] A. Colbrook and C. Smythe. Efficient implementation of search trees on parallel distributed-memory architectures. In *IEE Proceedings Part E*, volume 137, pages 394–400, 1990.

[D+89] W. J. Dally et al. The J-machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[DECa] Digital Equipment Corporation. *pixie(1)*. Ultrix 4.0 General Information, Vol. 3B (Commands(1): M-Z).

[DECb] Digital Equipment Corporation. *prof(1)*. Ultrix 4.0 General Information, Vol. 3B (Commands(1): M-Z).

[Del91] C. N. Dellarocas. A high-performance retargetable simulator for parallel architectures. Technical Report MIT/LCS/TR-505 (Master's Thesis), Massachusetts Institute of Technology, June 1991.

[DGH91] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, pages II99–II107, August 1991.

[DS87] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[DSNB87] B. A. Delagi, N. Saraiya, S. Nishimura, and G. Byrd. An instrumented architectural simulation system. Technical Report KSL 86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[Lin90] M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 USENIX Summer Conference*, pages 211–220, June 1990.

[LM86] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. Technical Report TR194, University of Rochester, Computer Science Department, September 1986.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 269–278, April 1991.

[MF88] I. Mathieson and R. Francis. A dynamic-trace-driven simulator for evaluating parallelism. In *Proceedings of 21st Hawaii International Conference on System Sciences*, volume 1 (Architecture), pages 158–166, January 1988.

[Nic88] D. A. Nicole. Esprit Project 1085: Reconfigurable transputer processor architecture. In *Proceedings of CONPAR '88*, pages 81–89, September 1988.

[Qui89] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[SF89] C. B. Stunkel and W. K. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proceedings of ACM Sigmetrics 1989*, pages 70–78, May 1989.

[Smi81] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[SS87] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. Technical Report 58037, Courant Institute, July 1987.

[Sta85] Charles R. Standridge. Performing simulation projects with The Extended Simulation System (TESS). *SIMULATION*, 45(6):283–291, December 1985.

[Wei84] P. J. Weinberger. Cheap dynamic instruction counting. *AT&T Bell Laboratories Technical Journal*, 63(8):1815–1826, October 1984.