The Performance of an Object-Oriented, Parallel Operating System*

DAVID R. KOHR, JR.[1,†], XINGBIN ZHANG[1,†], MUSTAFIZUR RAHMAN[2,†], AND DANIEL A. REED[1,†]

[1]*Department of Computer Science, University of Illinois, Urbana, IL 61801*
[2]*Department of Computer Science, University of Massachusetts at Amherst, Amherst, MA 01003*

ABSTRACT

The nascent and rapidly evolving state of parallel systems often leaves parallel application developers at the mercy of inefficient, inflexible operating system software. Given the relatively primitive state of parallel systems software, maximizing the performance of parallel applications not only requires judicious tuning of the application software, but occasionally, the replacement of specific system software modules with others that can more readily respond to the imposed pattern of resource demands. To assess the feasibility of application and performance tuning via malleable system software and to understand the performance penalties for detailed operating system performance data capture, we describe a set of performance instrumentation techniques for parallel, object-oriented operating systems and a set of performance experiments with *Choices*, an experimental, object-oriented operating system designed for use with parallel systems. These performance experiments show that (a) the performance overhead for operating system data capture is modest, (b) the penalty for malleable, object-oriented operating systems is negligible, but (c) techniques are needed to strictly enforce adherence of implementation to design if operating system modules are to be replaced.

1 INTRODUCTION

Striking advances in device technology have made high-speed processors and large primary memories both ubiquitous and inexpensive. With these advances have come parallel systems whose peak performance can be scaled across a wide range simply by adding processor/memory building blocks. However, high hardware performance peaks are not synonymous with high achievable performance across a wide range of scientific or commercial applications; many parallel systems exhibit performance instability, with a high variance in observed performance on different applications. The root cause for performance instability is rarely simple, but is most often due to the interactions of the hardware, the operating system software resource management policies, and the application resource demands. Minimizing performance instability on parallel systems is crucial to achieving substantial fractions of peak performance for scientific application codes.

Application software developers normally view the hardware and operating system software as an

Received April 1994
Revised May 1994

* Based on "A Performance Study of An Object-Oriented, Parallel Operating System" by D. R. Kohr, X. Zhang, D. A. Reed, and M. Rahman, which appeared in the *27th Hawaii International Conference on System Sciences*, Wailea, Hawaii, January 4–7, 1994, pp. 76–85, © 1994 IEEE.

† Supported in part by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, and by an equipment grant from the Digital Equipment Corporation External Research Program.

© 1994 by John Wiley & Sons, Inc.
Scientific Programming, Vol. 3, pp. 301–324 (1994)
CCC 1058-9244/94/040301-24

integrated "black box" that cannot be modified. Instead, they must adapt the application code to the existing configuration and maximize application performance subject to these constraints. Although this approach is well suited to mature systems with well-understood features, the nascent and rapidly evolving state of parallel systems often leaves parallel application developers at the mercy of inefficient, inflexible operating system software. Simply put, the evolution of system software and resource management algorithms has not kept pace with dramatic changes in parallel architectures.

Given the relatively primitive state of parallel systems software, maximizing the performance of parallel applications not only requires judicious tuning of the application software, but occasionally, the replacement of specific system software modules with others that can more readily respond to the imposed pattern of resource demands. Lowering the barrier between the application and the operating system increases the opportunity for optimization—one can adjust the system software to more efficiently support the application. Two requirements are implicit in this approach: detailed performance data and malleable operating system infrastructure. Detailed performance data are a prerequisite for informed performance optimization. The second, flexible operating system infrastructure, provides the mechanism for experimentation. Unless it is easy to replace existing operating system components with new components, the intellectual burden will preclude experimentation. Instead, a building block approach is needed that allows one to assemble operating system modules in a variety of ways to accommodate specific application needs.

In this article, we describe a set of performance experiments with *Choices* [1], an experimental, object-oriented operating system designed for use with parallel systems. Succinctly, our research goals were to:

1. Explore performance instrumentation techniques for parallel operating systems.
2. Measure the performance penalty, if any, imposed by an object-oriented operating system implementation.
3. Study the interaction of parallel operating system components by capturing a trace of operating system service demands.
4. Assess the feasibility of performance optimization by operating system customization.

The remainder of the article is organized as follows. In Section 2, we begin with a brief overview of the *Choices* operating system design philosophy and the implications of an object-oriented design for parallel operating systems. This is followed in Section 3 by a description of an object-oriented approach to capturing operating system performance data and the lessons learned from building operating system performance instrumentation. In Sections 4–6 we describe the experimental environment, a set of independent performance measurements used to validate our instrumentation software, and a detailed analysis of the behavior of *Choices* when supporting members of the Stanford SPLASH (Stanford Parallel Applications for Shared Memory) benchmark set. In Section 7, we examine the issue of system software malleability and the feasibility of operating system reconfiguration to improve application performance. Finally, we conclude in Section 8 with observations on the feasibility of reconfigurable operating systems and the value of dynamic performance data.

2 THE CHOICES OPERATING SYSTEM

Historically, operating systems research has addressed two basic issues, although rarely in concert: policy (i.e., algorithms for effective resource management) and mechanism (i.e., the logical organization of operating system components). During the early years, resource management policies (e.g., virtual memory and backing store, disk arm scheduling, and process scheduling) were the primary research focus. Later, the focus shifted to the logical organization of single processor operating systems (e.g., kernels, modularization, and process hierarchies) and then to distributed system models (e.g., remote procedure calls and client/server models).

Choices [1] is a research operating system designed to promote experimentation with new operating system design mechanisms and with new resource management policies. By separating mechanism and policy, *Choices* was designed to encourage experimentation with both. Mechanisms permit reconfiguration of operating system components to support new parallel architectures and applications. For policy experiments, *Choices* supports a set of components that can be combined to support different models of parallel programming. Generic components are customized through object-oriented inheritance and specialization to match the specific concurrency requirements of applications.

2.1 Design Philosophy

Choices has, as its kernel, a dynamic collection of C++ objects. System resources, mechanisms, and policies are represented as objects that belong to a class hierarchy [2]. The object-oriented application interface has a name server that implements inheritance and polymorphism and provides access to system services, local and remote servers, and persistent objects.

In the *Choices* design, a conceptual framework subsumes the conventional organization of an operating system as a group of layers [13]. The framework for the system provides generalized components and constraints to which the specialized subframeworks must conform. The subframeworks introduce additional components and constraints and subclass components of the framework.

2.2 Current Implementations

Choices is most properly viewed as an operating system schema whose instantiations contain varying fractions of the code base. At present, parallel versions of *Choices* are operational on the Intel iPSC/2 hypercube, the shared memory Sun Sparc/660 multiprocessor, and the bus-based Encore Multimax shared memory system. All versions share most of the abstract classes, but an instantiation for a particular parallel system necessarily contains only that subset of the concrete classes appropriate for that hardware platform.

Choices is an evolving system, both because it can be configured in many ways and because development of new software modules continues. As a basis for our experiments, we selected the most stable and widely used variant, an instantiation on the Encore Multimax. Although the Multimax hardware is no longer near the state of the art, it did provide a well-understood hardware platform for study. We believe the majority of our results will translate directly to other hardware configurations.

At the time of our experiments, this version supported:

1. Two native programming models, shared memory and message passing, with an object-oriented interface that supports application access to operating system kernel objects.
2. Unix System V and Berkeley file systems.
3. A compatibility mode that allows Unix application programs to be compiled and executed without change.
4. A message-passing system, with shared memory and copy-based variants.
5. Networking, with telnet, ftp, and other basic applications.
6. A multithreaded kernel with a variety of task schedulers (FIFO, LIFO, round robin, multilevel feedback queue, "standard" Unix, highest-response-ratio-next, and shortest-remaining-time).
7. A general set of performance instrumentation and data capture objects.

The particular software configuration used for our experiments is described in Section 4.

3 OPERATING SYSTEM INSTRUMENTATION

The volume and diversity of performance data obtainable from an operating system are potentially enormous, and one must judiciously balance the volume of data against both its accuracy and the potential utility; the penalty for insufficient data is exceeded only by that for inaccurate or misleading data. Unfortunately, data volume and accuracy are antithetic; most instrumentation and data capture techniques induce some perturbation (e.g., by modifying code or by interrupting a processor to record data) [3, 4].

Operating system performance instrumentation imposes particularly thorny problems because operating systems are, by their nature, reactive, responding to external stimuli. Changing the operating system response time for requests often will also change the pattern of requests. Moreover, recording operating system performance data often require operating system services—one must ensure that use of these services is isolated and not part of the subsystems being measured (e.g., if measuring file system activity is the goal, one should not use the file system to incrementally archive file system performance data). In addition to these constraints, common to all operating system instrumentation, the object-oriented *Choices* operating system design has additional implications for performance data capture mechanisms.

3.1 Implications of Object Orientation

Choices was originally designed to be portable and to operate efficiently on both shared and distributed memory systems. Indeed, one of the major motivations for the *Choices* design was to encourage and permit cross-architecture performance comparisons (e.g., by measuring the performance of the same code on disparate parallel systems). To maximize portability, the abstract classes of the *Choices* design hierarchy include few assumptions about the underlying architecture, and only a subset of the concrete classes embody machine-specific details. Performance instrumentation should not inhibit this portability by unduly relying on particular architectural features. For example, on a shared memory parallel system, it is tempting to allocate buffers for recording performance data that can be shared by all processors. However, an implementation based on this approach cannot be ported to a distributed memory, message-passing system without a major redesign. Hence, the *Choices* performance instrumentation provides a separate performance data buffer for each processor, making it efficient on systems with both shared and distributed memories. This has the ancillary benefit of eliminating synchronization for data buffer access, allowing simultaneous performance data recording by multiple processors.

Reflecting the object-oriented nature of *Choices*, the instrumentation system was designed as a hierarchical set of objects. However, the instrumentation implications of objects are more profound than simply a design style for data capture software. The heavy reliance of *Choices* on inheritance, where classes inherit other C++ classes and member functions from higher-level, base classes, means that instrumentation in any class that is not a leaf of the hierarchy will be inherited by all derived classes. In some cases, this is beneficial because the same performance data are needed from all variants. In other cases, the desired data are either different or a superset of that available to the base class, mandating customized instrumentation of the derived class. To maximize flexibility, the *Choices* instrumentation supports combinations of inherited and customized instrumentation.

Based on these issues, and our instrumentation experiences, both with *Choices* and with other systems [3, 4] we believe that parallel operating system instrumentation must be *general purpose*, supporting instrumentation and data capture from a variety of operating system modules using a common interface, *isolated*, with minimal dependence on operating system services, *dynamic*, with triggers to dynamically enable and disable performance data capture based on data volume and system activity, and *integrated* with application program performance data capture, permitting correlation of operating system performance data with application behavior and resource demands. Drawing on these principles, below we describe the design and object-oriented instrumentation system implementation for the *Choices* system software.

3.2 Choices Instrumentation Overview

A variety of techniques have been proposed for capturing operating system performance data, but all are members of three broad categories: timing, tracing, or counting. Because each strikes a different balance between data volume and potential measurement intrusion, the *Choices* instrumentation system supports all three, via `Counter`, `Timer`, and `Event` objects. Each type of instrumentation object can be used to capture either application or operating system performance data.

Figure 1 shows the major application and system instrumentation classes and their inheritance relationships. Although each is discussed briefly below, space limitations preclude a complete description: see Rahman [5] for details. The classes for event tracing, counting, and timing are all derived from the abstract `Instrument` base class. This base class provides methods to temporarily suspend (and later resume) data recording, as well as to reset the instrumentation object. In turn, instances of the derived `Counter` class can be used

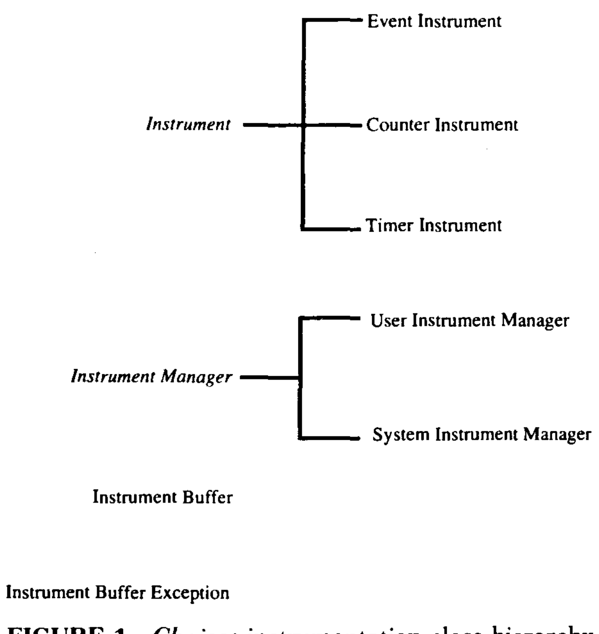


FIGURE 1 *Choices* instrumentation class hierarchy.

to count the number of times an event of interest has occurred, and periodically record the current count in a performance data buffer. Similarly, `Timer` objects can be used to record the time elapsed during the execution of a code fragment. Finally, `Event` objects support generic event tracing, with optional, user-specified data recorded with the default timestamp and event identifier. All three types of instrumentation objects produce performance data that are buffered and periodically written to secondary storage. The following information is common to all three:

1. A timestamp, indicating when the data were generated.
2. An event identifier that uniquely specifies the type of the data.
3. The name of the execution thread from which the data originated.
4. The processor where the event occurred.

As Figure 2 shows, an instrument manager is associated with every instrumentation object. Each of these instrument managers is responsible for certain housekeeping chores associated with the instruments it manages (e.g., temporarily suspending the recording of performance data). Each task of a parallel application program can create one or more `User Instrument Manager` objects to logically group and control related instruments. A single `System Instrument Manager` controls all operating system instruments and coordinates the set of `User Instrument Manager` objects.

A separate instance of an `Instrument Buffer` object for each processor manages a buffer of performance data that has not yet been written to secondary storage. The `Instrument Buffer Exception` object coordinates the dumping of instrumentation buffers by all processors to secondary storage.

Because all application and operating system events on a particular processor are written to the same buffer, they are correctly ordered by the time they occurred, simplifying later correlation of operating system resource requests with system responses. Also, because performance data obtained from each processor are recorded in a buffer specific to that processor, there is no contention for access to a buffer by multiple processors. This approach also obviates migration of performance data buffers between processors when tasks are rescheduled on another processor, and it is easily implemented on both shared and distributed memory parallel systems.

By locking each processor's buffer in nonpageable, kernel memory one avoids page faults during performance data recording. Not only does this minimize the variability of data recording costs, it

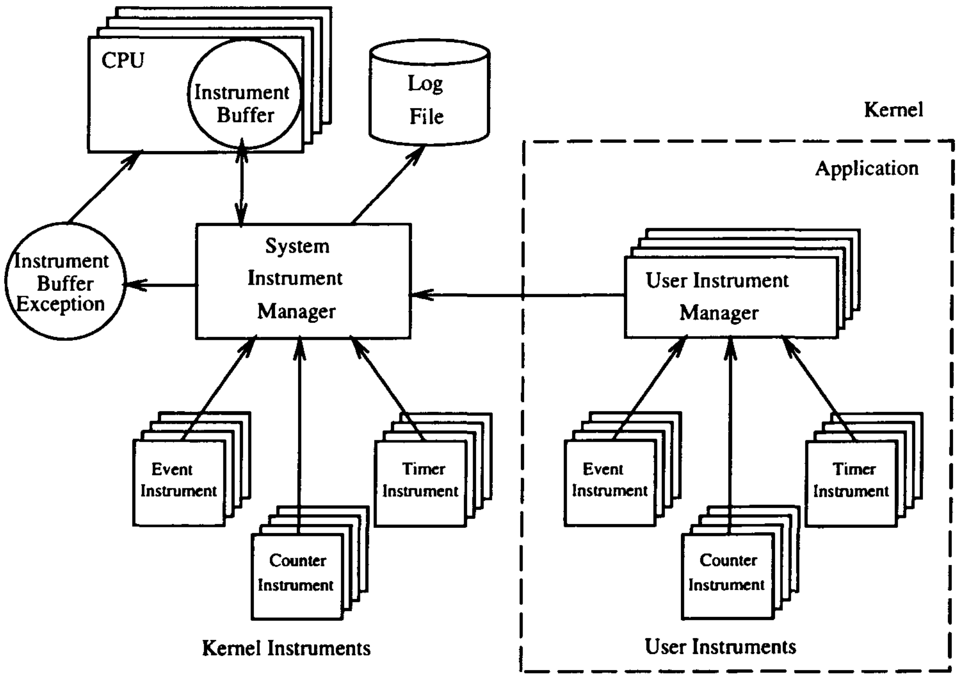


FIGURE 2. *Choices* instrumentation overview.

also makes instrumentation of the virtual memory system possible—the instrumentation system cannot cause additional page faults during tracing of page fault service routines.

In addition, instrumentation buffer dumping is completely synchronous. When any processor's performance data buffer fills, all processors are interrupted, and no processor is allowed to resume normal execution until all data buffers have been written to secondary storage. Hence, the perturbation induced on each processor is identical, and there is no skewing of the tasks on different processors. By recording the total time needed to dump all data buffers, we can postprocess the performance data and adjust the observed event times to eliminate these costs [3]. Finally, because all processing is suspended during buffer dumping, the instrumentation system does not contend with application processes for access to disks.

4 PERFORMANCE ANALYSIS METHODOLOGY

Earlier *Choices* performance measurements [6] focused on the cost of individual system operations (e.g., system calls) and the costs of virtual function table lookups imposed by a C++ implementation. These studies showed that the performance penalties for an object-oriented design need not be prohibitive, but they did not explore the interactions of operating system components. Hence, two of our major research goals were to explore the overheads for detailed operating system performance instrumentation and to study the dynamic interactions among object-oriented operating system components when supporting a parallel scientific workload.

Our research goals required measurements of *Choices* operating system behavior and its component interactions when subjected to a realistic scientific workload, and a comparison of these measurements to equivalent data obtained from a traditional operating system. The latter was necessary both to validate our performance measurement system, and to assess the system performance of a parallel, object-oriented operating system. Figure 3 illustrates our experimental methodology. Our experimental environment was a two-processor Encore Multimax 320 shared memory multiprocessor, executing a shared memory variant of *Choices*. We obtained comparative performance data from Umax 4.2, Encore's Unix implementation. (The Multimax 320 supports up to 16, 15 MHz, 2 MIP, NS32332 processors on a shared bus. Each processor has a 64K byte write-through cache.) Although the Multimax 320 is no longer state of the art, and most parallel systems now contain far more than two processors, our experience with this system and experimental data from other contexts both suggest that the data obtained are typical of what would be observed on larger or more modern shared memory parallel systems.

As a representative scientific computation workload, we selected programs from the SPLASH benchmark suite [7]. The SPLASH benchmarks are typical engineering and scientific codes of moderate size, written in C and Fortran, and drawn from a variety of application domains. Each is an explicitly parallel, shared memory program, parallelized using the Argonne National Laboratory's Parmacs macro package.

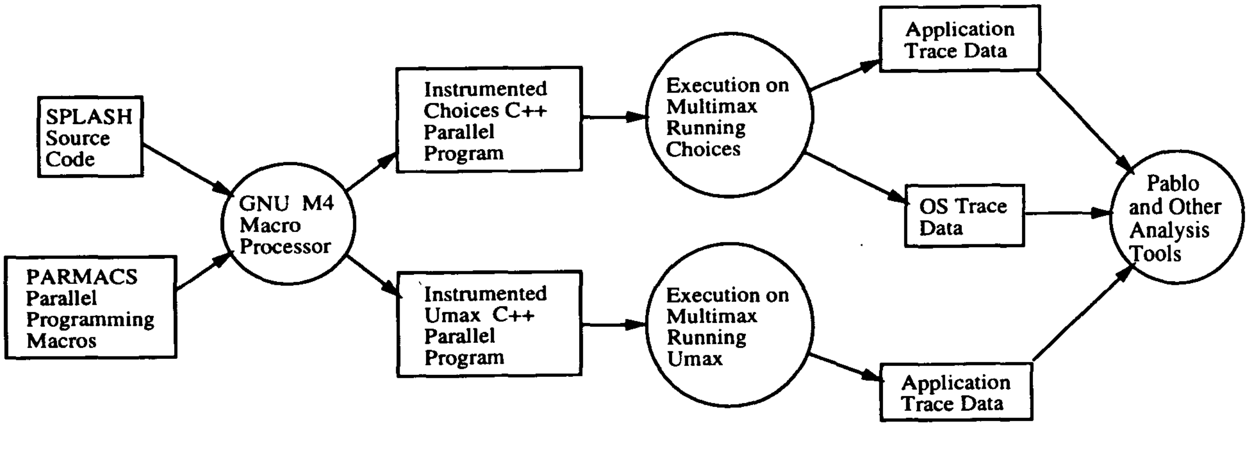

SPLASH Source Code
PARMACS Parallel Programming Macros
GNU M4 Macro Processor
Instrumented Choices C++ Parallel Program
Instrumented Umax C++ Parallel Program
Execution on Multimax Running Choices
Execution on Multimax Running Umax
Application Trace Data
OS Trace Data
Application Trace Data
Pablo and Other Analysis Tools

FIGURE 3 Performance analysis methodology.

4.1 Experimental System Configuration

Several pragmatic issues arose when adapting the SPLASH codes for execution on *Choices*. First, the *Choices* application programming interface does not support system calls in the traditional sense; instead, the system supports requests for operating system services via proxies [2, 8], C++ interfaces to the system software that allow interaction with objects that are not in the same protection domain. More significantly, the model of parallelism used in the Argonne Parmacs package (i.e., multiple, heavyweight, Unix-style processes) differs from the native, shared memory parallel programming model on *Choices* (i.e., lightweight threads that execute in a shared address space).

To execute the SPLASH codes on *Choices*, we converted the C versions of the codes to C++[1] and relied on the *Choices* Unix compatibility mode. Although the compatibility mode fails to capitalize on either the lower overhead, threads model of *Choices*, or the *Choices* system services directly available via proxies, it did permit performance comparison of both *Choices* and Encore's parallel Unix (Umax 4.2).

Our goal was to measure the behavior of *Choices* and Umax 4.2 under conditions typical of real scientific workloads. Hence, during all experiments, the regular operating system services of both *Choices* and Umax were enabled. On *Choices*, networking daemons periodically serviced interrupts resulting from incoming and outgoing packets; ftp and telnet servers were awaiting connections from remote machines; and scheduler time slice timeout interrupts continued to occur. During each test, a single login shell was created to initiate execution of a SPLASH benchmark. Hence, both operating systems experienced a relatively quiescent but "realistic" background workload, in addition to that imposed by the SPLASH code itself. The presence of this background workload was invaluable in identifying and isolating the causes of performance anomalies.

In all our experiments, *Choices* was configured with a task scheduler that managed all user tasks on a single, preemptible (by interrupts and system tasks) round-robin (FIFO) queue with one second quanta. System tasks were on a separate, nonpreemptible FIFO queue whose members had higher priority than the members of the user task queue. All experiments used a Berkeley file system that was Umax compatible (i.e., files written by *Choices* were readable by Umax).

4.2 Instrumentation and Data Collection

Given sufficiently detailed information about the state of each system and application task (i.e., the locus of control, values of key program variables, and assigned processor), it is possible to accurately correlate application requests with system processing, and to identify system processing that has no causal relationship with application requests. However, collection of detailed performance data using software instrumentation is not without price: Instrumentation perturbs the measured system and may result in observed behavior and event orders that would not be feasible in a system without instrumentation [9].

Because our performance experiments were the first intensive tests of the *Choices* instrumentation system, and because we wished to study primarily the effects of the *Choices* scheduler and disk input/output system on the performance of the SPLASH codes, we elected to instrument only a subset of the *Choices* modules.[2] Given constraints on the number of possible operating system instrumentation points, context switch instrumentation provides the most information [10]—it exposes not only the decisions of the task scheduler, but also the interactions of application tasks and the execution patterns of service daemons. Hence, we instrumented *Choices* to trace the time of each context switch, the identity of the currently executing task, and the identity of the newly scheduled task.

Using interval timing, we modified the *Choices* disk input/output class to record the starting and ending time of each input/output operation. We also instrumented the *Choices* disk input/output interrupt service routines; this allowed us to capture physical disk input/output rather than logical input/output to file buffers.

This limited set of instrumentation points strikes a balance between sufficient performance data to understand system dynamics and excessive instrumentation perturbation, and sufficed to determine both which tasks were executing at each point in time and when input/output re-

[1] To avoid potential effects of compiler differences, we also used the C++ versions of the SPLASH codes for our Unix performance experiments.

[2] For the SPLASH codes, task scheduling and input/output processing are the primary points of interaction with the operating system. To maximize portability, the SPLASH codes use few operating system facilities.

quests were being serviced. To correlate application and operating system behavior, we instrumented the SPLASH codes to record the time of occurrence and duration of each procedure call, outermost loop entry/exit, and interprocessor synchronization.

4.3 Comparative Measurements

To provide a reference point against which the performance of *Choices* could be compared, one also needs performance data from an execution of the SPLASH codes on another operating system, in this case Encore's Umax. Unfortunately, Umax provides no native performance instrumentation system, either at the system or application level. Because the lack of access to the Umax system source code precluded instrumenting Umax, we concentrated on application-level performance data as a basis for comparisons. We developed a portable, minimalist instrumentation package for collecting application-level traces. This package, which can be used with either *Choices* or Umax, preallocates large trace buffers that reside in the address space of each instrumented task, avoiding interactions with the file system.

The existence of a portable, application-level instrumentation package allowed us to decouple the effects of possible instrumentation overheads and operating system differences. By measuring application performance on both Umax and *Choices* with the same portable application instrumentation and C++ compiler, we could be sure that any observed differences in performance were directly attributable to operating system differences. In addition, by comparing application performance data captured on *Choices* with both the portable instrumentation and the native *Choices* instrumentation, we could assess both the accuracy of the *Choices* instrumentation and the differences in instrumentation overhead.

5 INSTRUMENTATION ANALYSIS

The primary danger when instrumenting any stimulus-driven software system is that instrumentation may change both the time needed to process stimuli and the temporal order of the generated responses. Typically, perturbations are either direct, resulting from simple increases in stimulus processing times attributable to the insertion of instrumentation code, or indirect, resulting from the reordering of asynchronous stimuli or their responses.

Under constrained conditions, the effects of direct perturbations can be removed by postprocessing the captured performance data to adjust the observed event times [3]. (If the events are totally ordered and cannot be changed by instrumentation, and the cost for each instrumentation point is known, adjusting the event times involves only a simple linear transformation.) More generally, inserted software instrumentation has more subtle effects (e.g., displacing data values from the cache or causing pipeline stalls), the exact cost of each instrumentation point is not known, and exact compensation for instrumentation effects is not possible.

Indirect perturbations are more pernicious, and in the worst case may require a complete system simulation to recover the event order that would have occurred had instrumentation not been present [9]. For example, if the events have differing priorities (e.g., system and user task resource requests), or are time dependent (e.g., scheduler time slice interrupts), software instrumentation may change the event order or even alter the number of events.

On a parallel system, the observed events are partially ordered, and the observed event order may not have been feasible on an uninstrumented system (i.e., if the instrumentation costs were zero, the observed event order would have been impossible under any execution circumstances). In short, system instrumentation is subject to an uncertainty principle: Measurement perturbs the system, and one must balance the volume of desired data against its accuracy.

An important first step in the analysis of performance data captured using software instrumentation is to bound the potential perturbation of the nominal execution time and the event reordering. Below, we discuss the costs of capturing application and operating system performance data, followed by an analysis of possible perturbations induced by the instrumentation of system and application code.

5.1 Application Instrumentation Costs

To estimate the time needed to record performance data using both the *Choices* instrumentation and our portable instrumentation package, we began with a set of in vitro measurements on a synthetic benchmark that contained a single loop. We compared the execution time when the loop body was empty to the execution time of the same loop when a single instrumentation point was inserted, taking care to ensure that compiler optimi-

Table 1. Event Recording Costs (microseconds)

Operating System	Instrumentation	In Vitro Overhead	In Vivo Overhead
Umax	Portable	15.1	13.6
Choices	Portable	144.8	153.3
	Choices	509.6	596.5

zations did not eliminate the loop iterations. From this, we calculated the time to record a single event, as follows.

If N is the number of loop iterations, t_n is the execution time of the empty loop, and t_i is the execution time of the instrumented loop, the estimated cost C_i of an instrumentation point is

$$C_i = \frac{t_i = t_n}{N}.$$

The in vitro data in Table 1 summarizes the result of these measurements on both *Choices* and Umax. Figure 4 shows a portion of the corresponding raw instrumentation event times.

Figure 4 and Table 1 show that the in vitro instrumentation costs for the portable instrumentation and the native *Choices* instrumentation systems differ greatly. The chief reason for these differences is that on both Umax and *Choices*, the portable instrumentation software executes within the context of the user process. (Recall that the performance data buffer resides in the process' address space and no buffer input/output occurs until the process completes execution.) The value of a high-resolution, memory-mapped hardware

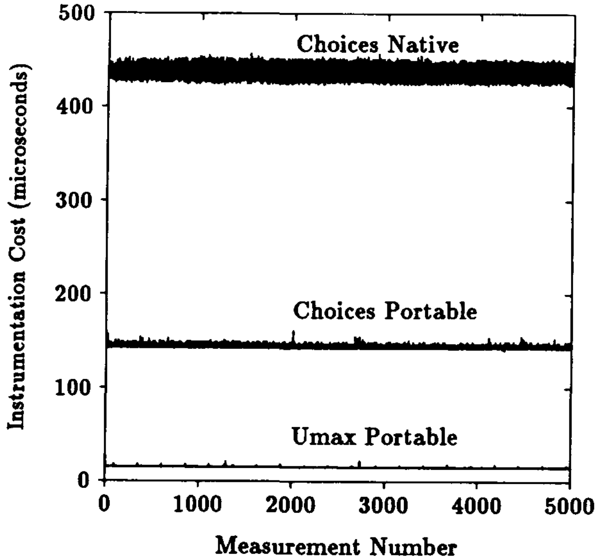


FIGURE 4 In Vitro event recording overhead.

clock is reflected in the instrumentation cost differences for the Umax and *Choices* versions of the portable instrumentation package. On Umax, the Multimax's microsecond hardware timer is memory mapped to the application address space, whereas access to the same timer on *Choices* requires a proxy-based system call; this is the sole cause for the portable instrumentation overhead differences in Figure 4 and Table 1.

In contrast to the portable instrumentation package, the native *Choices* performance data buffers reside in system memory, requiring protection boundary crossings to record data. Moreover, because the *Choices* instrumentation provides greater functionality, and hence is more complex, recording data require the interaction of several objects via C++ virtual function calls (Fig. 2). Conversely, the portable instrumentation system records data using inline code, avoiding the overhead of procedure calls and most protection boundary crossings.

Periodically writing the performance data buffers to secondary storage is an additional, unavoidable source of overhead in the *Choices* instrumentation; any general purpose instrumentation system that must capture arbitrary amounts of performance data requires access to external storage or a data transport medium. As described in Section 3, the *Choices* instrumentation synthesizes performance trace events that specify the time required to write the performance data buffers to secondary storage; these synthesized events are embedded in the performance data. Because all other system and application activity is suspended during buffer dumping, the effect on all user and system tasks is identical and can be easily removed from the performance data by subtracting the cost of buffer dumping from subsequent event occurrence times.

Because a *Choices* proxy call to obtain the time involves several procedure calls and a protection boundary crossing, there is both greater cost to obtain the time and, because the timing code may not be present in the cache, there also is greater variation in the cost of reading the clock. This variation is clear in Figure 4—both the magnitude and the variation in event recording times increase from the Umax portable instrumentation, with a memory mapped clock, to the portable instrumentation on *Choices*, with an operating system call required to read the clock, to the native *Choices* instrumentation, with more complex data recording and multiple proxy calls.

5.2 Observed Perturbations

To establish the veracity of the application instrumentation cost model, we compared the in vitro estimates of Table 1 to in vivo measurements obtained from the measured execution of the SPLASH [7] WATER benchmark, a molecular dynamics simulation. We measured the sequential execution time of the WATER code, with and without the presence of the portable application instrumentation, and divided the difference in execution times by the number of captured events to obtain the mean instrumentation cost. (The WATER code contains no timing dependent code that might generate differing numbers of trace events based on the execution schedule and instrumentation overhead.) The result is the in vivo data of Table 1. In general, the modest difference in the instrumentation costs, less than 20% in the worst case, suggests that the in vitro measurements capture the salient effects of the instrumentation code in the in vivo case.

Comparing the in vitro and in vivo values shows that the in vivo values are lower for Umax and higher for *Choices*. We conjecture, but have been unable to confirm, that the values are lower for Umax because the portable instrumentation on that system consists only of inline code. No procedure calls are needed to record the performance data, and the compiler can more effectively optimize the larger basic blocks that result when instrumentation is inserted. In contrast, on *Choices* both the portable and native instrumentation require system calls to obtain the current time. This fragments the basic blocks and reduces opportunities for compiler optimization. In addition, the greater complexity of the *Choices* system instrumentation is more likely to perturb the cache, increasing the warm start miss ratio for application codes and increasing the in vivo instrumentation costs.

Using a trace of synchronization events from a parallel execution of the WATER code, we also compared the partial event order obtained with the portable and the native *Choices* instrumentation. Because the parallel version of the WATER code has a static work distribution (i.e., work is not dynamically assigned to tasks), differing instrumentation costs cannot cause work to be shifted from one task to another, nor can they change the number of recorded events in each application task. Analysis showed that the portable and *Choices* system instrumentation traces had the same partial event order, despite large differences in the instrumentation costs. However, for more dynamic, timing-dependent codes, larger perturbations are more likely.

5.3 Scalability

Using the native *Choices* operating system instrumentation and the portable application instrumentation, we instrumented members of the SPLASH benchmark suite to assess the performance of both *Choices* and Umax, Encore's parallel Unix, on a two-processor Encore Multimax 320.

Although resource limitations did not allow us to conduct experiments with larger numbers of processors, we are confident that this approach scales to substantial numbers of processors. The belief is based on our implementation of similar instrumentation on systems with tens to hundreds of processors [11] and the use of these techniques on other massively parallel systems [12] that have hundreds of processors.

6 EXPERIMENTAL DATA ANALYSIS

Using the native *Choices* operating system instrumentation and the portable application instrumentation, we instrumented members of the SPLASH benchmark suite to assess the performance of both *Choices* and Umax, Encore's parallel Unix, on a two-processor Encore Multimax 320. Because our primary goals were to understand the costs of dynamic operating system instrumentation, the interactions between operating system and application program resource demands, and the overhead for malleable system software, we did not explore the effects of multiprogramming; all experiments involved only one active application program.

The high dimensionality of the experimental space (i.e., two operating systems, a variety of potential operating system configuration options, and multiple programs from the SPLASH benchmark set), together with the time required to conduct an experiment and the large volume of performance data obtained from each experiment, precluded a complete factorial analysis. Instead, we selected a single member of the SPLASH benchmark suite, the WATER code, as the basis for study; this allowed us to study its behavior in detail, and using this knowledge, to understand the implications for parallel operating system performance and software configuration.

6.1 WATER Application Benchmark

WATER is an N-body molecular dynamics code that simulates the evolution of water molecules in the liquid phase [7]. In its parallel version, the molecules are partitioned and statically assigned to tasks. Each parallel task is responsible for calculating the time–evolutionary state of its assigned molecules. To reduce the number of pairwise force calculations, only interactions between pairs of molecules with distances less than a specified cut-off radius are calculated. At each time step, the molecules move in response to the force calculations. Hence, the spatial distribution of molecules is not uniform, and the task load balance and synchronization costs potentially change at each time step.

Using the Parmacs computation model, the WATER code consists of a serial initialization phase (including assignment of work to processes), followed by a fork of the requisite number of participating computation processes, initialization of the processes, and the actual computation. Table 2 summarizes the major procedures of the WATER code that correspond to these phases.

Unless otherwise indicated, all experiments involved 64 water molecules and two time steps; the volume of performance data and the 2 MIP processing rate of the Multimax NS32332 processors made the execution times and data analysis costs of larger inputs prohibitive. Nevertheless, even these small input data sets suffice to capture the salient aspects of program and system behavior.

Although we made every attempt to minimize the differences between the WATER code variants on Umax and *Choices*, the disparity in application programming models and libraries on the two systems necessitated some changes. Of these, the most important change was the use of the same math library on both systems to permit fair performance comparisons.

6.2 Sequential Application Behavior

As a basis for comparing operating systems and for understanding the parallel execution of the WATER code, we captured application procedure entry/exit traces from a sequential execution on both *Choices* and Umax using our portable instrumentation software. On *Choices* we also used the system instrumentation to capture operating system data. Figures 5–8 show the pattern of procedure calls for the sequential Umax execution. In the figures, the WATER code's two time steps are clearly visible in the pattern of procedure calls, with a transition from intermolecular to intramolecular force calculations near times 28.5 and 53.

In contrast to the Umax execution time of ap-

Table 2. WATER Procedures and Event Identifiers

Major Procedure	Event Identifier	Brief Description
main	1	Main routine
CNSTNT	2	Other constants initialization
SYSCNS	3	System constants initialization
INITIA	4	Data file input and initialization
WorkStart	5	Initiate computation of the parent process
MDMAIN	6	Initiate computation of a child process
PREDIC	7	Predict new displacement values
CORREC	8	Correction of predicted values
BNDRY	9	Boundary condition computation
KINETI	10	Kinetic energy calculation
POTENG	11	Potential energy calculation
INTRAF	12	Intramolecular force calculation
INTERF	13	Intermolecular force calculation with global communication
CSHIFT	14	Molecular distance calculation
UPDATE_FORCES	15	Molecular force update

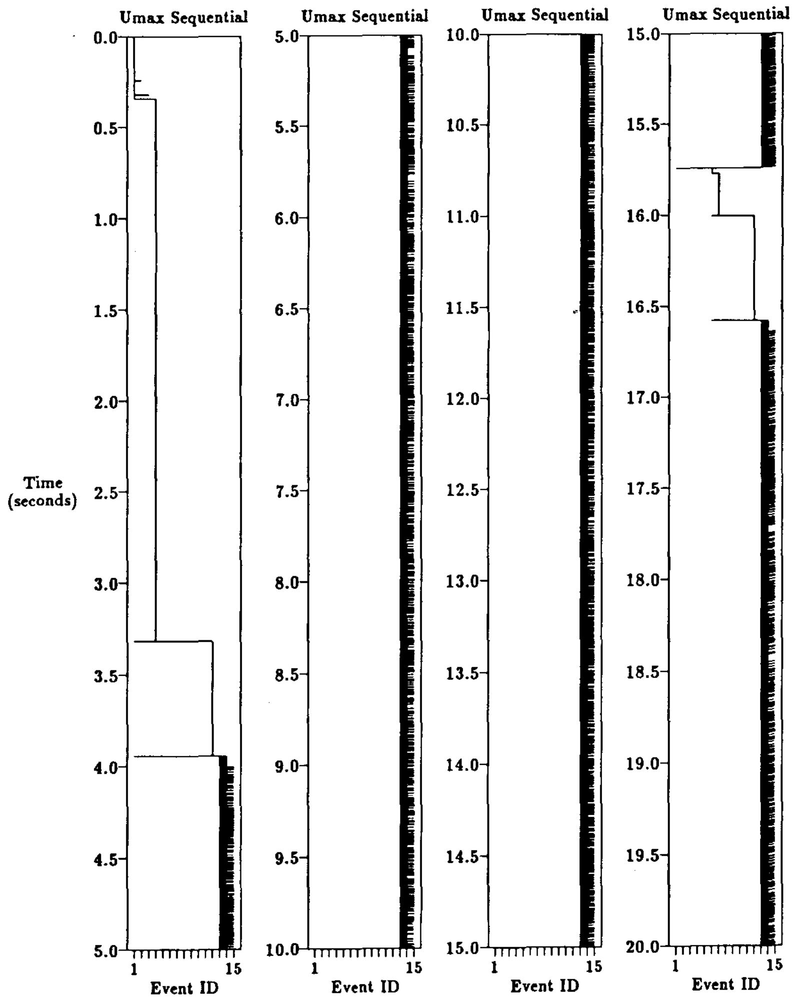


FIGURE 5 Representative sequential procedure activations on Umax.

proximately 65 seconds, a sequential execution of the same code under *Choices* requires almost 84 seconds (see Table 3). A detailed analysis of the trace data showed several reasons for this discrepancy. First, as Figure 4 and Table 1 show, the cost for timestamp acquisition on *Choices* is substantially greater than under Umax. This difference, approximately 150 microseconds in the in vivo case, coupled with roughly 92,000 recorded events, adds nearly 14 seconds to the sequential execution time. The cumulative magnitude of this overhead highlights the critical importance of a memory-mapped clock. Without such a clock, obtaining detailed performance data incurs large overheads.

The second cause for the disparity in the se-

Table 3. WATER Execution Time Distribution (seconds)

	Umax Time			*Choices* Time		
Component	Sequential	Parent	Child	Sequential	Parent	Child
Computation	61.36	44.65	24.28	65.12	50.79	28.94
Input/Output	2.19	2.02	0.00	4.49	4.48	0.00
Application Instrumentation	1.25	0.77	0.48	14.14	8.72	5.43
Instrumentation Initialization	0.00	4.51	0.00	0.00	34.43	0.00
Process Fork	0.00	0.33	0.00	0.00	74.70	0.00
Total	64.80	52.27	24.77	83.74	173.12	34.36

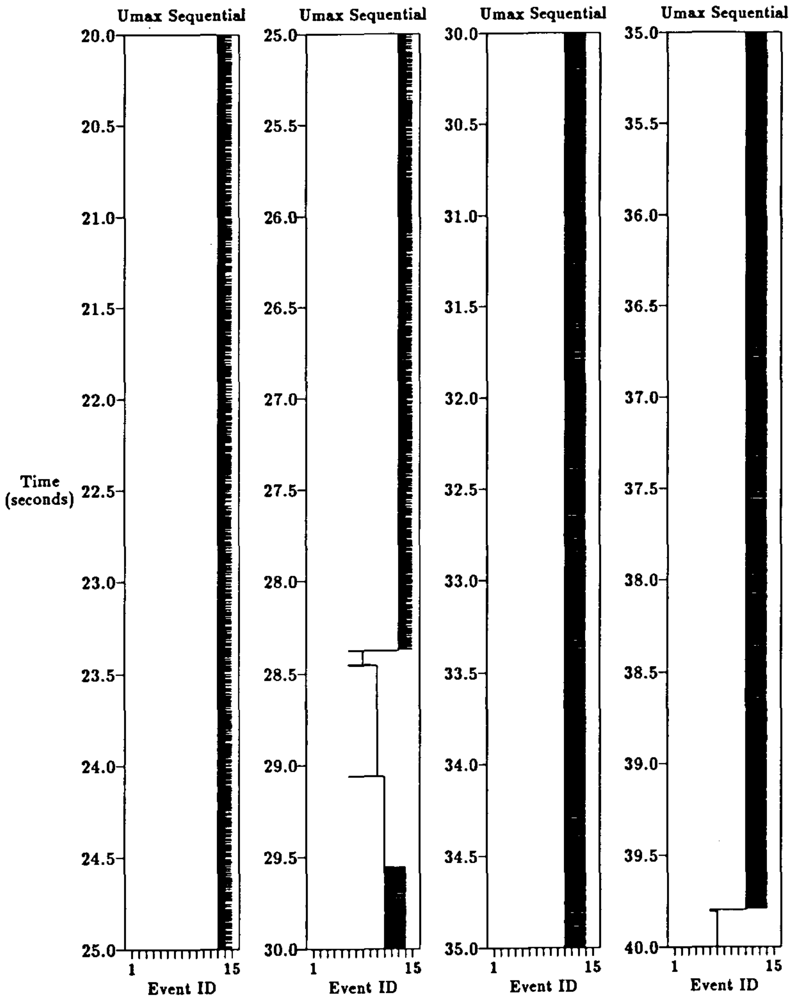


FIGURE 6 Sequential procedure activations on Umax (continued).

quential application execution times across the two operating systems is input/output overhead. At the time of these experiments, the *Choices* disk device drivers were not yet fully optimized, and the disk transfer rate under *Choices* was approximately half that of Umax. At the beginning of its execution, the WATER code reads a 193K byte molecular description file from disk; this adds approximately 2.5 seconds to the *Choices* execution. Finally, during execution under *Choices*, there was a modest amount of extra overhead for system event recording that is not present under Umax. Subtracting these overheads from the sequential *Choices* execution time yields an execution time comparable to that for Umax.

Given an accounting for the disparities in sequential execution times, we turn to an analysis of the dynamics of operating system behavior. Figure 9 shows a small portion of these dynamics—the procedure call pattern on both Umax and *Choices* during comparable periods of execution. The patterns in Figure 9 are strikingly similar, although shifted in time by the greater overhead for event recording on *Choices*. To see this distortion, we extracted the time of procedure call and activation lifetimes for two of the dominant procedures in the WATER code, UPDATE and CSHIFT. Figures 10–11 show the distribution of these lifetimes.

The CSHIFT procedure calls no other application procedures; it simply computes the distance between two molecules using a simple loop that contains a conditional. The horizontal banding in Figures 10–11 reflect the distribution of times when the conditional is true. This banding is much less evident in Figure 10, the *Choices* execution, than in Figure 11, the Umax execution,

Table 4. *Choices* Task Description and Time Distributions

Process	Process Description	Sequential Total Time	Parallel Total Time
WATER parent task	WATER main task	82.83	105.94
WATER child task	WATER child task	0.00	49.16
Idle task	Executes when no other task is ready	0.95	62.85
ARB retransmit daemon	Checks/retransmits TCP packets	83.57	128.14
Waste manager	Recovers when tasks complete	0.00	0.00
Kernel setup	System startup/shutdown task	0.00	0.00
Console interrupt	Manages console inputs	0.00	0.00
Telnet server daemon	Processes new telnet connections	0.00	0.00
Ethernet manager	Handles Ethernet receive	0.09	0.08
Ethernet control	Handles Ethernet transmission	0.00	0.17
Zoot receive daemon	Passes TCP packets to TCP clients	0.00	0.00
Zoot controller	Processes incoming Ethernet packets	0.05	0.06

because the cost of event recording with the portable instrumentation package has much higher variance on *Choices* (see Fig. 4). The extra overhead for event recording is also manifest in the shifting of the procedure duration time scale; the differential shift for the two procedures occurs because they contain a different number of instrumentation points.

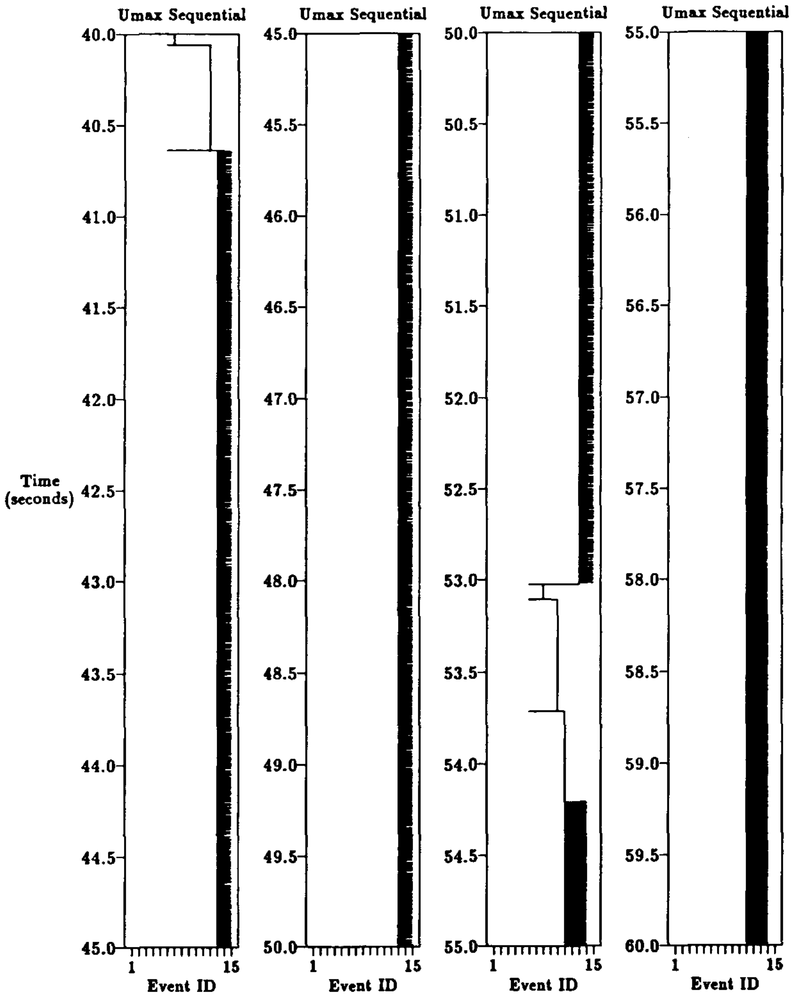


FIGURE 7 Sequential procedure activations on Umax (continued).

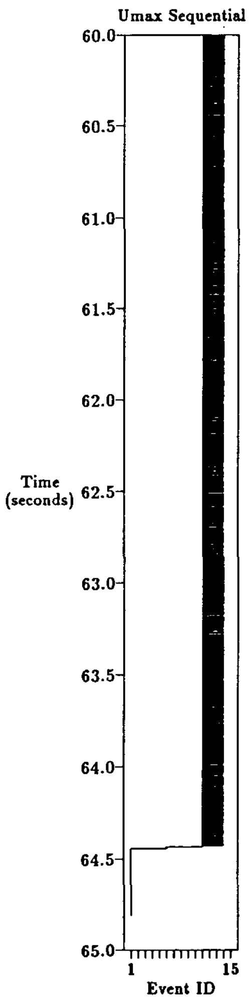


FIGURE 8 Sequential procedure activations on Umax (continued).

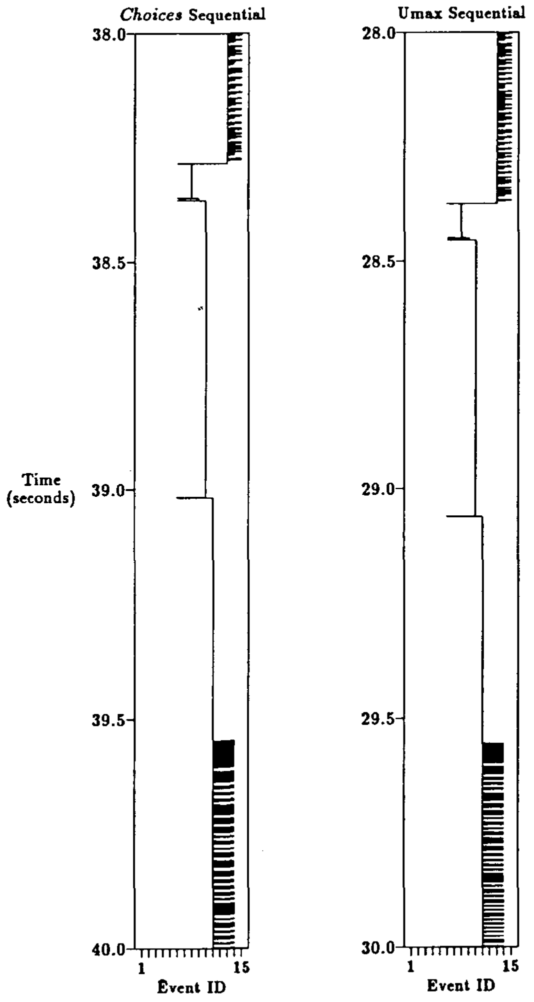
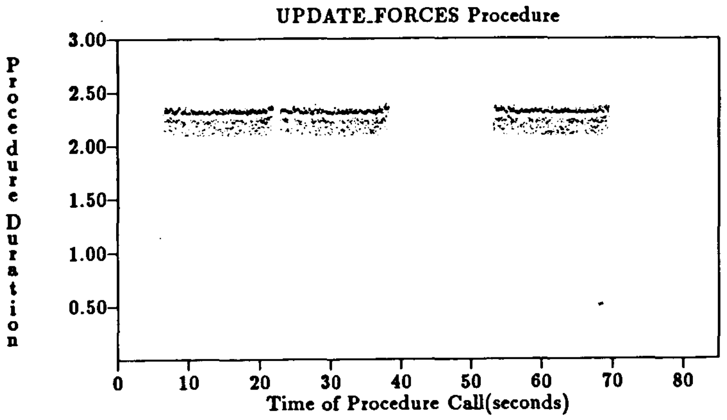


FIGURE 9 Sequential procedure activations on *Choices* and Umax.

Finally, Table 4 summarizes the function of each active *Choices* service daemon and its total processor time during the sequential execution of the WATER code. Figure 8 shows the temporal pattern of task scheduling activity and context switches that lead to these times. During program initialization, the number of context switches is high because there are user interactions with the

Table 5. WATER Parallel Execution Phase Durations (seconds)

Computation Phase	Beginning	End	Duration
Choices serial initialization	0.00	21.80	21.80
Choices fork	21.80	96.50	74.70
Choices child initialization	96.50	130.93	34.43
Choices parallel computation	130.93	173.12	42.19
Umax serial initialization	0.00	15.58	15.58
Umax fork	15.58	15.91	0.33
Umax child initialization	15.91	20.42	4.51
Umax parallel computation	20.42	52.28	31.86



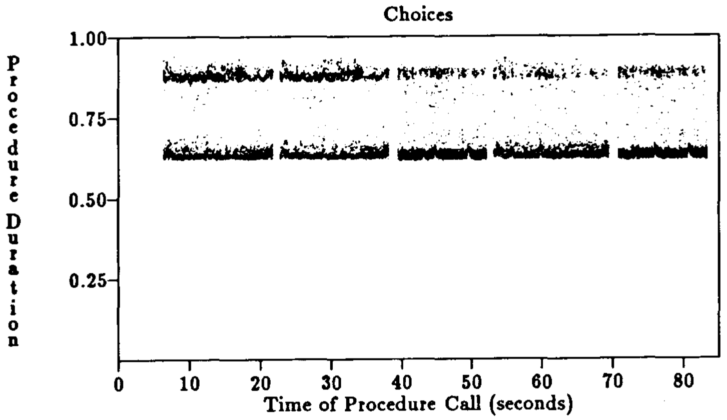


FIGURE 10 *Choices* procedure durations in milliseconds (sequential execution on two processors).

command interpreter to specify program parameters, following this the context switch pattern quickly stabilizes.

Unlike more traditional operating systems, *Choices* does not preemptively timeslice processes unless the number of processes demanding a processor exceeds the number of available processors. Instead, processes execute until they must relinquish the processor, either due to delays waiting for requested services or competing demands

Table 6. WATER Synchronization Summary (times in milliseconds)

Synchronization Construct	Activation Count	Mean Duration	Duration Variance
Choices sequential execution			
Lock	7916	0.16	4.9×10^{-3}
Barrier	12	2.58	7.89
Umax sequential execution			
Lock	7916	0.02	2.13×10^{-3}
Barrier	12	0.06	4.57×10^{-3}
Choices parallel execution			
Lock	7928	0.20	1.04
Barrier	23	1660	7.00×10^{-3}
Umax parallel execution			
Lock	7928	0.03	1.09×10^{-2}
Barrier	23	223.8	9.15×10^{2}

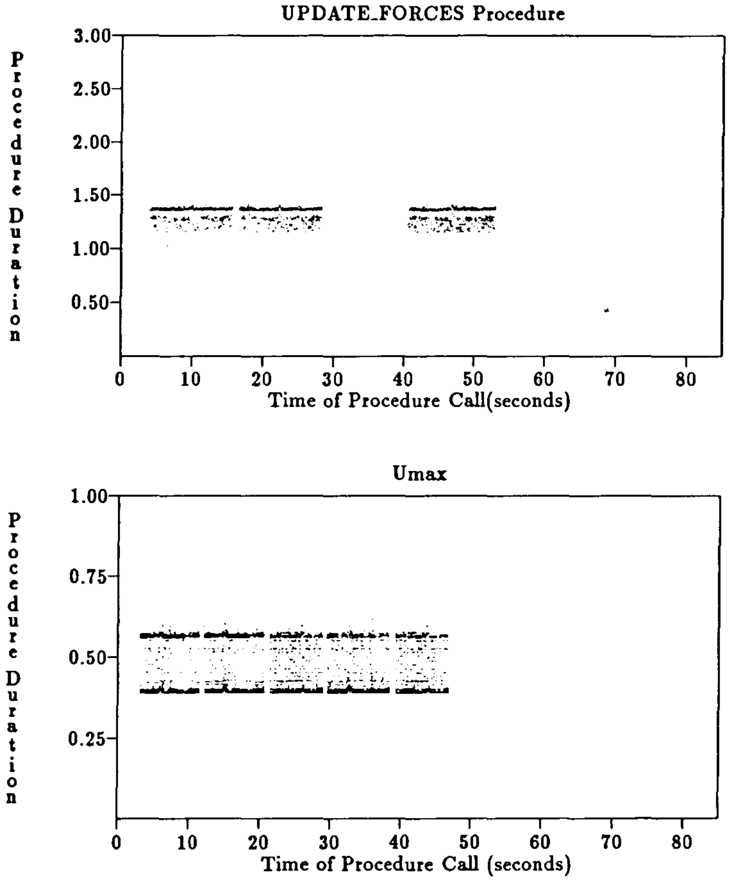


FIGURE 11 Umax procedure durations in milliseconds (sequential execution on two processors).

for processor services. During program execution, the single application process repeatedly migrates between the two processors in response to activation of network daemons. Figure 17a shows that most of these context switches involve network software, and Table 4 shows that most of the system software overhead involves a single daemon, the TCP packet retransmit daemon. (On seeing this behavior, the *Choices* system developers immediately recognized a software design error; the intended goal of the ARB retransmit daemon was to check and retransmit TCP packets only at 1-second intervals. This error was corrected in a later version of the *Choices* software.)

In summary, the single processor performance of the WATER code on *Choices* is similar to that on Umax, albeit with three major differences in the behavior under *Choices*:

1. The absence of a memory-mapped clock makes performance event recording costly, increasing the total execution time.
2. Unoptimized input/output system increases program initialization time.
3. Different software daemons, coupled with a different task scheduling algorithm, change the pattern of application time slices.

Using these observations as a base, we turn to an analysis of the WATER code's parallel execution behavior.

6.3 Parallel Application Behavior

Table 3 shows the distributions of overhead for computation, instrumentation, and input/output for a parallel execution of the WATER code on both Umax and *Choices*. Similarly, Table 5 summarizes the durations of each parallel execution phase on *Choices* and Umax. As with the sequential executions, a portion of the differences are directly attributable to differing instrumentation overheads. In particular, the differences in the parallel computation phases are largely due to dif-

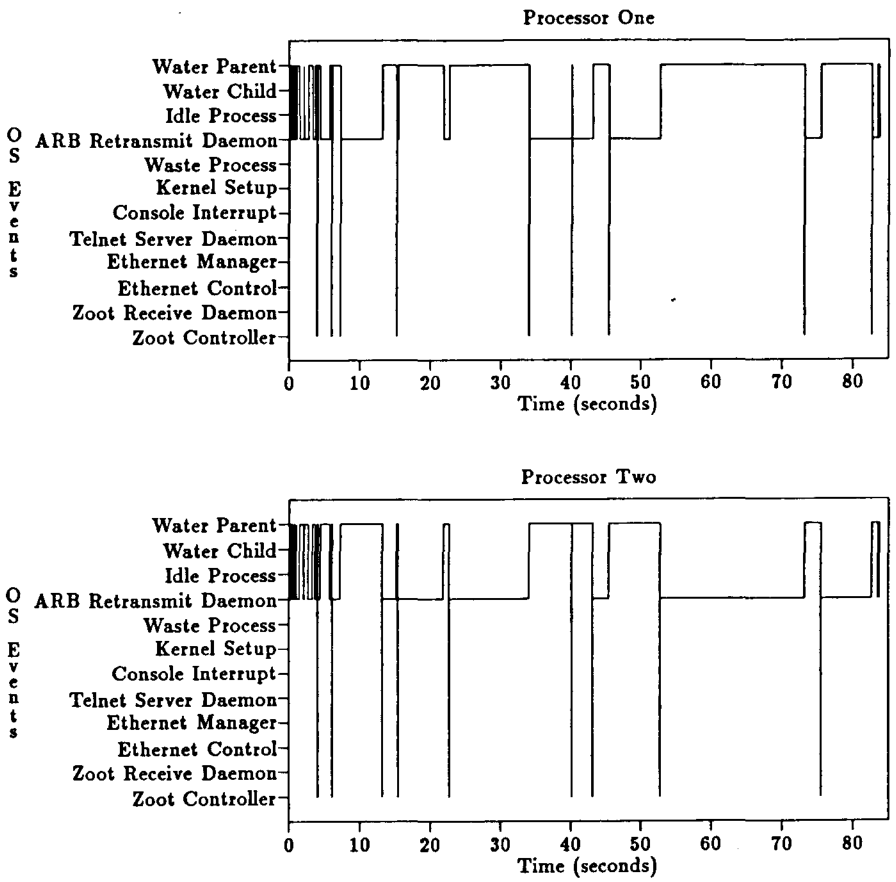


FIGURE 12 Context switch pattern on *Choices* (sequential execution on two processors).

ferences in performance data recording costs. However, the most striking data in the two tables are the enormous increase in the execution time of the parallel *Choices* code: the parallel execution time is roughly double that of the sequential code. The reasons for this increase are rooted in the use of the *Choices* Unix compatibility library.

Unlike Umax, the *Choices* Unix compatibility library does not implement a copy-on-write strategy for replicating the address space of a parent process for a forked child. Hence, *Choices* must copy all data in all pages of the parent address space before the fork system call completes. Moreover, the *Choices* virtual memory system requires all newly copied pages to be mirrored on the backing store; this creates extensive secondary storage activity during a process fork.

In Figure 14, the input/output pattern in the interval 20–100 seconds is the process fork: this contains two distinct behaviors. The pattern in the interval 20–60 seconds (exclusively write requests) reflects the mirroring of the address space to secondary storage, and the pattern in the interval 60–100 seconds (a mixture of read and write requests) is the replication of the parent process's address space from secondary storage onto the child process's address space. In addition, as described in Section 3, the portable instrumentation package used a large, memory resident performance data buffer to minimize secondary storage activity during performance data capture. To fur-

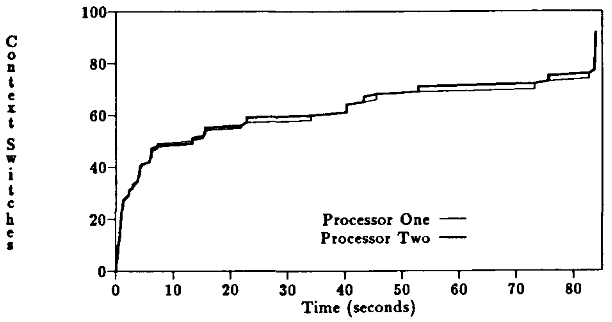


FIGURE 13 Cumulative context switches on *Choices* (sequential execution on two processors).

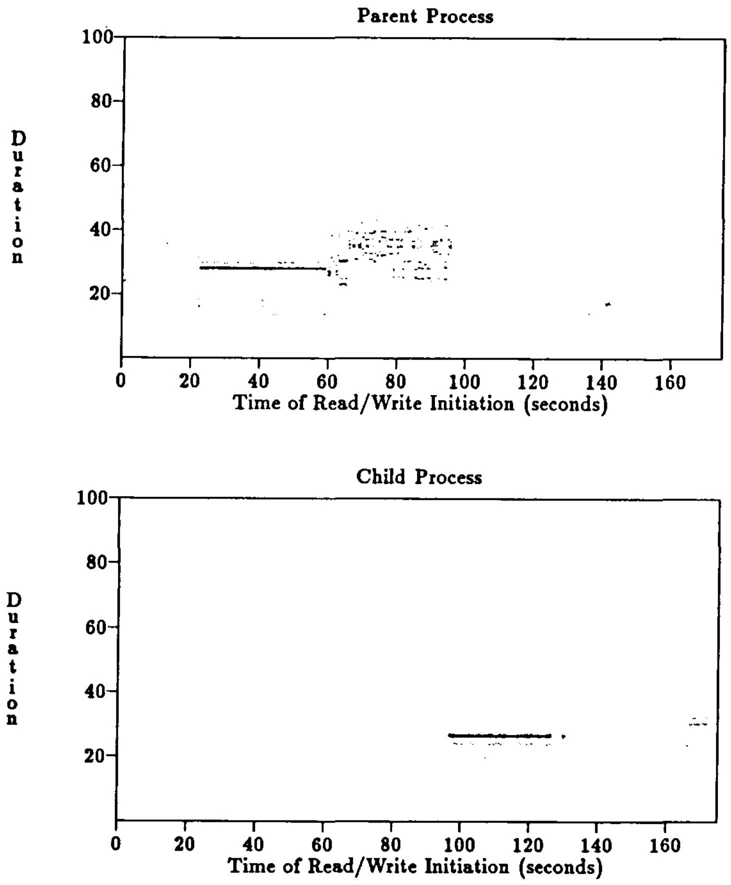


FIGURE 14 *Choices* input/output durations (parallel execution on two processors).

ther minimize interactions with the virtual memory system, the buffer was preemptively faulted into memory during the instrumentation software's initialization by accessing each page of the buffer. At the time of these experiments, the *Choices* page fault handling routines had not yet been tuned, creating substantial overheads for page fault service during initialization of the WATER code's child process. This is the child process input/output activity shown in Figure 14 for the interval 100–130 seconds.

Figure 15 shows comparable fragments of the WATER code's parallel execution traces for both Umax and *Choices*. Although the behavior of the parent processes are similar for the two operating systems, the child process on *Choices* clearly spends a much longer time computing intermolecular forces in the procedure INTERF than the child process on Umax. An examination of the trace of synchronization events reveals that the child spends about 0.75 seconds waiting at a barrier at the end of INTERF, despite the highly effective static load balancing in this application. (This barrier is denoted by the arrows in Fig. 15.)

The reason for this anomaly is clear from an examination of the context switch and parallel input/output patterns. Figure 16 shows that during this interval both the parent and child processes are relinquishing their processors to the TCP packet retransmission daemon. Table 4 indicates that the total execution time of the packet retransmission daemon is much higher during parallel execution, and Figure 17 confirms that the number of context switches experienced by the application processes rises dramatically when the WATER code executes in parallel. This, together with the page faults experienced by the parent process, shown near time 140 in Figure 14, and the interactions of process scheduling and contention for lock access are the underlying causes for the increase in execution time. As an illustration of the latter, Table 6 shows that the mean time for executing a barrier increases to over 1.6 seconds for the parallel execution.

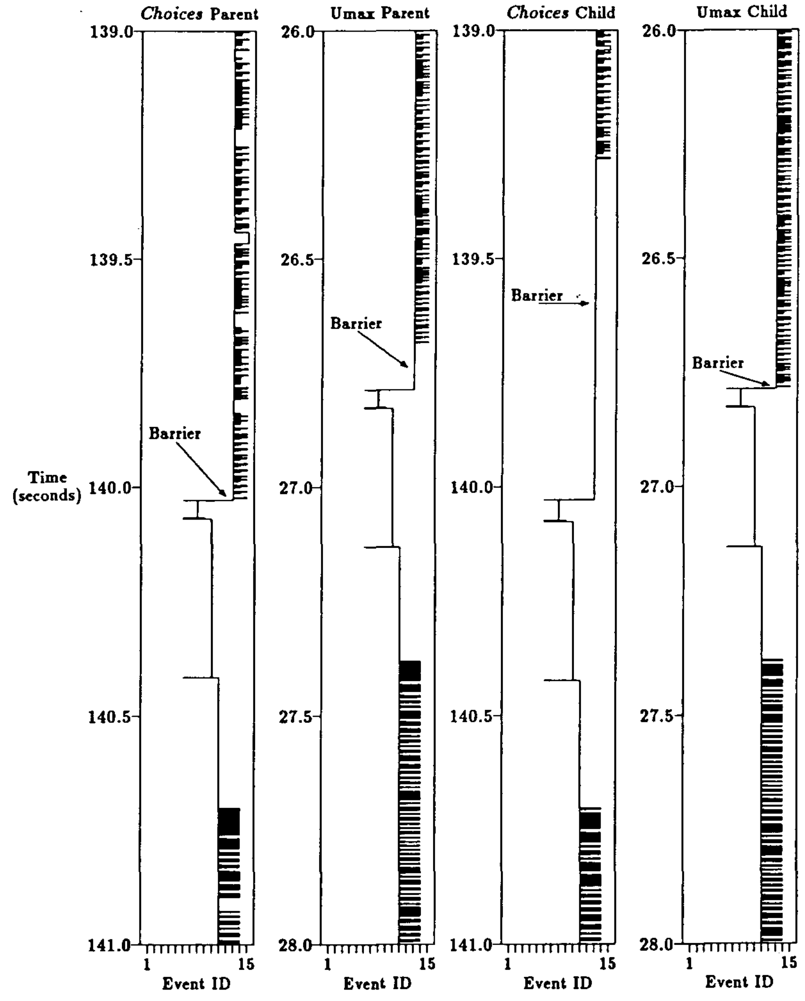


FIGURE 15 Process procedure activations on *Choices* and Umax.

In summary, the parallel execution of the WATER code on *Choices* differs from the Umax execution is two important ways: (1) The *Choices* Unix compatibility library is poorly optimized for parallel execution, creating large overheads for process creation (in fairness, the *Choices* Unix compatibility library was created to ease code porting, not to provide a parallel programming model), and (2) differing process scheduling policies change the pattern of process execution, which affects access times for locks and barriers.

6.4 Performance Observations

As we noted at the outset, two of our major research goals were to explore the overheads for detailed operating system performance instrumentation and to study the dynamic interactions among object-oriented operating system components when supporting a parallel scientific workload. Based on the analysis of Sections 6.2 and 6.3, several lessons are clear.

First, comparing two operating systems that were designed to support different programming models is extraordinarily difficult because one must support a nonnative execution model on one of the two systems. As Section 6.3 shows, this is a recipe for poor performance—*Choices* was not designed to support heavyweight processes, nor was Umax designed to support threads. An implementation of processes atop the *Choices* lightweight thread model, together with emulation of process fork semantics, is not sufficient to obtain good performance.

Second, seemingly small variations in system services can have profound performance implications. The lack of access to a memory-mapped clock on *Choices* made capturing fine-grained

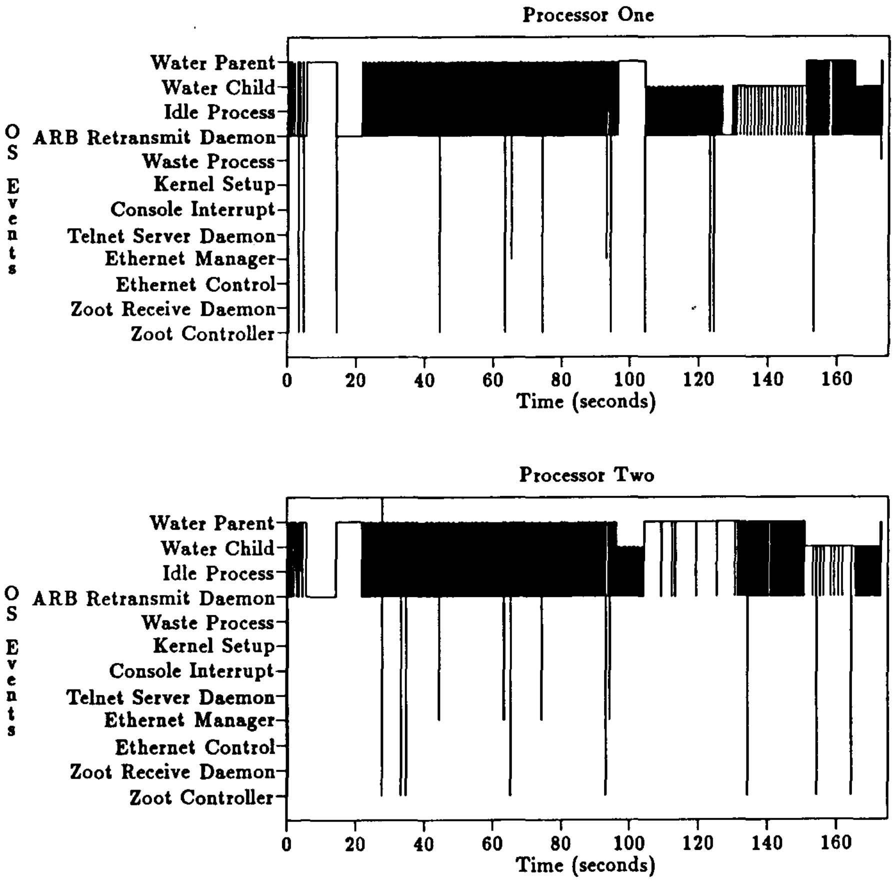


FIGURE 16 Context switch pattern on *Choices* (parallel execution on two processors).

performance data expensive and unnecessarily increased the instrumented execution time of locks and barriers. This deficiency is easily remedied, however. Even with this added overhead, the penalty for detailed operating system and application instrumentation was modest, allowing us to understand the interactions of input/output requests, task scheduling, and application task synchronization.

Third, and more positively, the performance of an object-oriented operating system can be competitive with traditional operating system designs. Except for performance penalties attributable to process emulation or untuned system services (e.g., disk input/output), the performance of applications on *Choices* is competitive with Unix. This is a system-level confirmation of the microscale measurements reported earlier [2].

7 OPERATING SYSTEM MALLEABILITY

The last of our research goals was to assess the feasibility of application performance tuning by adapting the operating system resource management policies to better match application resource demands. In this we were unsuccessful. Below, we summarize our experiences with *Choices* and suggest some guidelines for future implementations of object-oriented operating systems.

An operating system with a well-chosen, object-oriented design potentially provides the requisite infrastructure for efficient, easy replacement or specialization of operating system modules. Inheritance encourages the implementation of module families (e.g., schedulers or memory managers) that share standard interfaces and features. Similarly, the protection and data encapsulation provided by classes in an object-oriented language like C++ isolate the implementation details of specific services.

In *Choices*, a framework for each major software subsystem is defined by a set of abstract C++ classes that are then specialized and instantiated with concrete classes to form a specific implementation. *Choices* also supports a dynamic loading mechanism that allows applications and system programs to add new system services to the

From Process	To Process											
WATER Parent Task			18						8	1		
WATER Child Task												
Idle Task				18								
ARB Retransmit Daemon	27											
Waste Manager												
Kernel Setup												
Console Interrupt												
Telnet Server Daemon												
Ethernet Manager												8
Ethernet Control												1
Zoot Receive Daemon												
Zoot Controller				9								

(a) Processor Zero

From Process	To Process											
WATER Parent Task			18						9			
WATER Child Task												
Idle Task				18								
ARB Retransmit Daemon	27											
Waste Manager												
Kernel Setup												
Console Interrupt												
Telnet Server Daemon												
Ethernet Manager												9
Ethernet Control												
Zoot Receive Daemon												
Zoot Controller				9								

(b) Processor One

Sequential Execution

From Process	To Process											
WATER Parent Task			1086	59					4			
WATER Child Task			42	791					1			
Idle Task	6			1123					2	1		
ARB Retransmit Daemon	1143	833			1				2			5
Waste Manager				1								
Kernel Setup												
Console Interrupt												
Telnet Server Daemon												
Ethernet Manager			2									7
Ethernet Control			1									
Zoot Receive Daemon												
Zoot Controller		1	1	10								

(c) Processor Zero

From Process	To Process											
WATER Parent Task		1	1103	93					3			
WATER Child Task			43	262					2			
Idle Task	25	1		1122					5			
ARB Retransmit Daemon	1175	305										3
Waste Manager												
Kernel Setup												
Console Interrupt												
Telnet Server Daemon												
Ethernet Manager			5									5
Ethernet Control												
Zoot Receive Daemon												
Zoot Controller			2	6								

(d) Processor One

Parallel Execution

FIGURE 17 Context switch transition matrices.

Choices kernel during execution. As an example, *Choices* supports several file systems, including system V and BSD; file system code for each file type can be loaded on demand. Finally, *Choices* supports an interface that allows users to query the system about its current state; one can determine the active class hierarchy, the members of a particular class, and the instances of a specified class. Together, the object-oriented design, dynamic object loading, and the query interface potentially provide a backdrop for performance tuning via replacement of classes that implement specific services.

Despite the elegance of the *Choices* design and the protection provided by C++, we encountered several difficulties when attempting to conduct parametric performance experiments on *Choices*. The first of these plagues all large software projects, namely the diversion of implementation from design. We found it difficult to replace specific resource management policies because their implementations often relied on artifacts of classes outside the inheritance hierarchy. Rather than removing a single plant with an isolated root system, we found several plants with intertwined root systems.

To circumvent module entanglement, we need richer, more robust mechanisms that rigidly enforce the design philosophy and that provide system configuration management. By the latter, we mean a software interface that displays not just the class hierarchies and their interactions, but also the valid instantiations of those classes to create specialized operating systems with the desired features. Hardware vendors have developed configuration management systems that prevent sales staff or customers from ordering incomplete or invalid configurations; we need enforced use of similar facilities for operating system software if performance tuning via operating system specialization is to be accessible to large audiences.

The second major problem we encountered was the lack of performance guidance. Although we knew that replacing some subset of the resource management policies would improve performance, it was rarely clear which policies should be replaced or what the potential effects would be. For example, in Section 6.3 we observed that during a fork the *Choices* virtual memory system requires all newly copied pages to be mirrored on the backing store: this creates extensive secondary storage activity during a process fork, and in turn leads to a large number of processor context switches. Although replacing the virtual memory manager or the file system might improve performance, the best solution is implementation of copy-on-write process creation semantics.

To guide software specialization, we need tools that identify not just the proximate performance bottleneck but also the interactions of system components that are the root cause. Detailed performance data are necessary but not sufficient. Understanding system component interactions is but a precursor to informed decision making. In a full-featured operating system. the number of possible configurations is enormous. We need tools that allow the configurer to ask hypothetical questions (e.g., what might happen if one replaced this scheduler with another) and that predict, within bounds, the expected performance.

Despite the problems we encountered, we believe that an object-oriented operating system design is the key to effective performance tuning by operating system module replacement. However, an object-oriented design alone is insufficient. one also needs a rigidly enforced, object-oriented implementation that adheres exactly to a hierarchical design, interactive configuration management tools that allow the user to browse and construct specialized operating systems with specific features, and performance guidance tools that can identify software component interactions and suggest possible module alternatives.

8 CONCLUSIONS

We conjectured that detailed operating system and application performance data, together with a flexible, object-oriented operating system design, are the future cornerstones of systematic application and operating system performance tuning on parallel systems. Detailed performance data reveal the dynamic pattern of application and operating system component interactions, and object-oriented operating systems provide the separation of resource management mechanism .and policy needed to replace operating system modules with those more suited to observed application resource demands.

Our experiments showed that the performance of an object-oriented operating system can be competitive with traditional operating system designs, making the configuration of specialized operating systems easy and their potential performance high. Except for performance penalties attributable to process emulation or untuned system services (e.g., disk input/output), the performance of applications on *Choices* is competitive with Unix. This is a system-level confirmation of the micro-scale measurements reported earlier [6].

We also observed that detailed operating system performance data could be obtained at modest cost via a flexible, general purpose instrumentation infrastructure based on object-oriented design principles. Central to efficient performance data capture is a high-resolution, low-access latency, memory-mapped hardware clock.

Finally, we observed that operating system instrumentation and object-oriented design are not sufficient to support rapid operating system software reconfiguration. One also needs tools that can help the user select feasible module configurations and that can guide operating system performance tuning by module substitution.

ACKNOWLEDGMENTS

Our thanks to Roy Campbell and the members of the University of Illinois Systems Research Group, especially Lee Lup Yuen and Nayeem Islam, for their help with *Choices*, both with code modifications and with explanations of the *Choices* design style. Without their help, this work would not have been possible. Finally, our thanks to Ted Nelson, who developed an early version of the *Choices* instrumentation system. All authors were supported in part by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, and by an equipment grant from the Digital Equipment Corporation External Research Program.

REFERENCES

[1] R. Campbell and N. Islam, "A Parallel Object-Oriented Operating System," in *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MA: MIT Press, 1992.

[2] V. F. Russo, "An object-oriented operating system," PhD thesis, University of Illinois at Urbana-

Champaign, Department of Computer Science, October 1990.

[3] A. D. Malony, D. A. Reed, and H. Wijshoff, "Performance measurement intrusion and perturbation analysis, *IEEE Trans. Parallel Distrib. Systems,* vol. 3, pp. 433–450, 1992.

[4] D. A. Reed and D. C. Rudolph, "Experiences with hypercube operating system instrumentation," *Int. J. High-Speed Comput.,* pp. 517–542, 1989.

[5] M. Rahman, "Choices instrumentation support," Technical Report, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1992.

[6] V. F. Russo, P. W. Madany, and R. H. Campbell, "C++ and Operating Systems Performance: A Case Study," in *Proceedings of the 1990 USENIX C++ Conference.* San Francisco, 1990, pp. 103–114.

[7] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared memory," Technical Report, Stanford University, Department of Computer Science, 1991.

[8] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," in *Proceedings of the Sixth International Conference on Distributed Systems.* 1986.

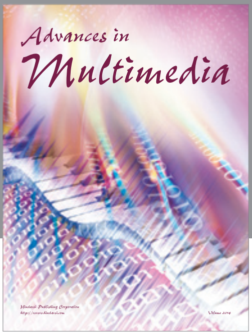
[9] A. D. Malony, "Performance observability," PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1990.

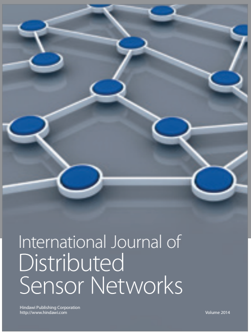
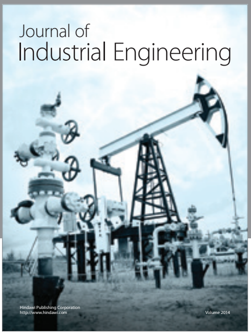
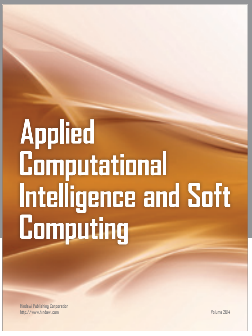
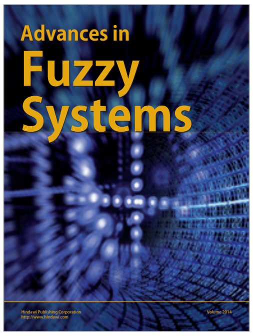
[10] T. Lehr, D. Black, Z. Segall, and D. Vrsalovic, *Proceedings of the 1990 International Conference on Parallel Processing.* 1990, pp. 298–299.

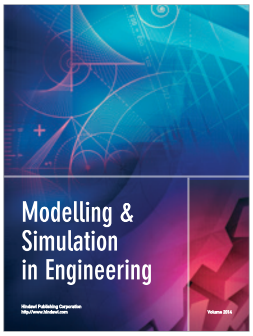
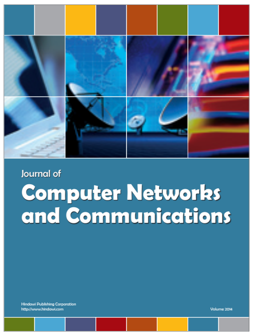
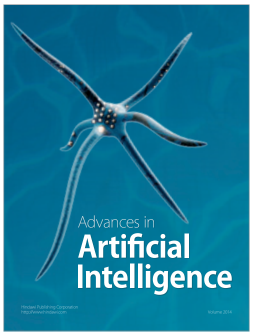
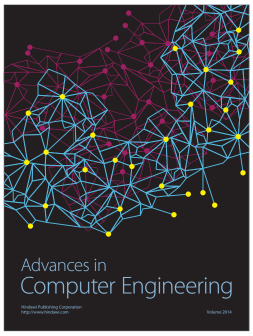
[11] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, *Proceedings of the Scalable Parallel Libraries Conference.* IEEE Computer Society, 1993.

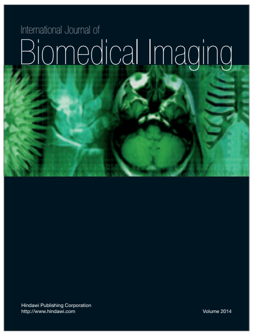
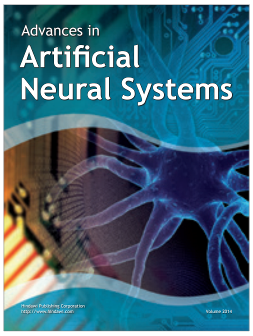
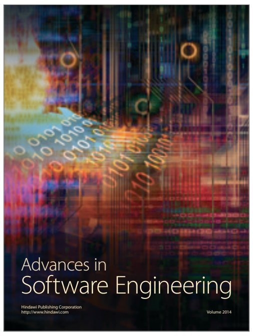
[12] Intel, *Application Tool User's Guide.* Beaverton, OR: Intel Supercomputer Systems Division, 1993.

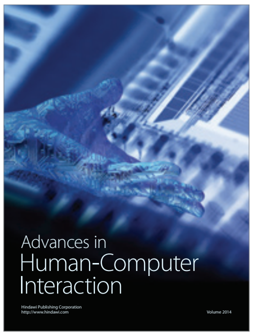
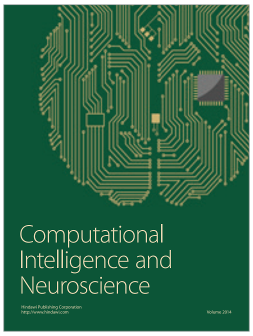
[13] R. H. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany, "*Choices,* Frameworks and Refinement," in *Object-Orientation in Operating Systems,* Luis-Felipe Cabrera, Vincent Russo, and Marc Shapiro, Eds. Palo Alto: IEEE Computer Society Press, 1991, pp. 9–15.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering