Using the SimOS Machine Simulator to Study Complex Computer Systems

MENDEL ROSENBLUM, EDOUARD BUGNION, SCOTT DEVINE, and STEPHEN A. HERROD
Computer Systems Laboratory, Stanford University

SimOS is an environment for studying the hardware and software of computer systems. SimOS simulates the hardware of a computer system in enough detail to boot a commercial operating system and run realistic workloads on top of it. This paper identifies two challenges that machine simulators such as SimOS must overcome in order to effectively analyze large complex workloads: handling long workload execution times and collecting data effectively. To study long-running workloads, SimOS includes multiple interchangeable simulation models for each hardware component. By selecting the appropriate combination of simulation models, the user can explicitly control the tradeoff between simulation speed and simulation detail. To handle the large amount of low-level data generated by the hardware simulation models, SimOS contains flexible annotation and event classification mechanisms that map the data back to concepts meaningful to the user. SimOS has been extensively used to study new computer hardware designs, to analyze application performance, and to study operating systems. We include two case studies that demonstrate how a low-level machine simulator such as SimOS can be used to study large and complex workloads.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems; B.3.3 [**Hardware**]: Memory Structures—*performance analysis, and design aids*

Additional Key Words and Phrases: computer architecture, computer simulation, computer system performance analysis, operating systems

1. INTRODUCTION

SimOS is a machine simulation environment designed to study large complex computer systems. SimOS differs from most simulation tools in that it simulates the complete hardware of the computer system. In contrast, most other environments only simulate portions of the hardware. As a result, they must also simulate portions of the system software.

SimOS was developed as part of the Stanford FLASH project, funded by ARPA grant DABT63-94-C-0054. Mendel Rosenblum is partially supported by an NSF Young Investigator Award. E. Bugnion and Steve Herrod are supported in part by NSF Graduate Fellowships Awards. Steve Herrod currently holds an Intel Foundation Graduate Fellowship.
Authors' addresses: Computer Systems Laboratory, Stanford University, Stanford, CA, 94305; http://www-flash.stanford.edu/SimOS; e-mail⟨{mendel,bugnion,devine,herrod}@cs.stanford.edu⟩

SimOS simulates the computer hardware with sufficient speed and detail to run existing system software and application programs. For example, the current version of SimOS simulates the hardware of multiprocessor computer systems in enough detail to boot, run, and study Silicon Graphics' IRIX operating system as well as any application that runs on it, such as parallel compilation and commercial relational database systems.

Simulating machines at the hardware level has allowed SimOS to be used for a wide range of studies. Computer architects can evaluate the impact of new hardware designs on the performance of complex workloads by modifying the configuration and timing model of the simulated hardware components. Operating system programmers can develop their software in an environment that provides the same interface as the target hardware, while taking advantage of the system visibility and repeatability offered by a simulation environment. Application programmers can also use SimOS to gain insight into the dynamic execution behavior of complex workloads. The user can nonintrusively collect detailed performance-analysis metrics such as instruction execution, memory-system stall, and interprocessor communication time.

Although machine simulation is a well-established technique, it has traditionally been limited to small system configurations. SimOS enables the study of complex workloads by addressing some particularly difficult challenges. The first challenge is to achieve the simulation speed needed to execute long-running workloads. Given sufficient speed, machine simulators produce voluminous performance data. The second challenge is to effectively organize these raw data in ways meaningful to the user.

To address the first challenge, SimOS includes both high-speed machine emulation techniques and more accurate machine simulation techniques. Using emulation techniques based on binary translation, SimOS can execute workloads less than 10 times slower than the underlying hardware. This allows the user to position the workload to an interesting execution state before switching to a more detailed model to collect statistics. For example, emulation can be used to boot the operating system and run a database server until it reaches a steady execution state. SimOS can dynamically switch between the emulation and simulation techniques, allowing the user to study portions of long running workloads in detail.

To address the second challenge, SimOS includes novel mechanisms for mapping the data collected by the hardware models back to concepts that are meaningful to the user. Just as the hardware of a computer system has little knowledge of what process, user, or transaction it is executing, the hardware simulation models of SimOS are unable to attribute the execution behavior back to these concepts. SimOS uses a flexible mechanism called annotations to build knowledge about the state of the software being executed. Annotations are user-defined scripts that are executed when hardware events of particular interest occur. The scripts have nonintrusive access to the entire state of the machine, and can control the classification of simulation statistics. For example, an annotation put on the context switching routine of the operating system allows the user to determine the currently scheduled process and to separate the execution behavior of the different processes of the workload.

This article describes our solution to the two challenges. We begin with an overview of SimOS in Section 2. Section 3 describes the use of interchangeable hardware simulation models to simulate complex workloads. Section 4 describes the data collection and classification system. In Section 5, we describe our experiences with SimOS in two case studies. In Section 6, we discuss related techniques used for studying complex systems. We conclude in Section 7.

2. THE SIMOS ENVIRONMENT

The SimOS project started in 1992 as an attempt to build a software simulation environment capable of studying the execution behavior of operating systems. Many of SimOS's features follow directly from this goal. To study the behavior of an operating system, SimOS was designed as a complete machine simulator where the hardware of the machine is simulated in enough detail to run the actual operating system and application workloads. Furthermore, the large and complex nature of operating systems required SimOS to include multiple interchangeable simulation models of each hardware component that can be dynamically selected at any time during the simulation.

In the rest of this section, we present a brief overview of these features of SimOS. Readers interested in the implementation details should refer to the previous papers on SimOS [Rosenblum et al. 1995] and Embra [Witchel and Rosenblum 1996] for a much more thorough discussion of the simulation techniques. The use of interchangeable simulation models for complete machine simulation, as discussed in Section 2.2, is introduced in Rosenblum et al. [1995]. That paper also describes in detail the implementation of SimOS's original approach to high-speed emulation based on direct execution. Embra, SimOS's current approach to high-speed emulation based on binary translation, is described in detail in Witchel and Rosenblum [1996].

2.1 Complete Machine Simulation

Despite its name, SimOS does not model an operating system or any application software, but rather models the hardware components of the target machine. As shown in Figure 1, SimOS contains software simulation of all the hardware components of modern computer systems: processors, memory management units (MMU), caches, memory systems, as well as I/O devices such as SCSI disks, Ethernets, hardware clocks, and consoles. SimOS currently simulates the hardware of MIPS-based multiprocessors in enough detail to boot and run an essentially unmodified version of a commercial operating system, Silicon Graphics' IRIX.

In order to run the operating system and application programs, SimOS must simulate the hardware functionality visible to the software. For example, the simulation model of a CPU must be capable of simulating the

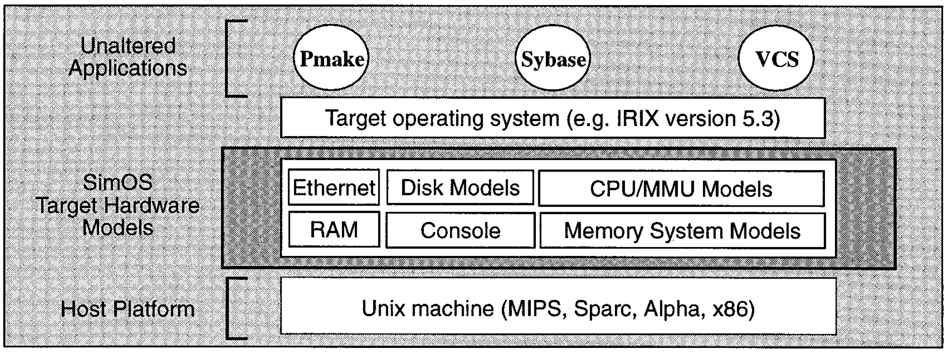

Unaltered Applications
Pmake
Sybase
VCS
Target operating system (e.g. IRIX version 5.3)
SimOS Target Hardware Models
Ethernet
Disk Models
CPU/MMU Models
RAM
Console
Memory System Models
Host Platform
Unix machine (MIPS, Sparc, Alpha, x86)

Fig. 1. The SimOS environment.

execution of all MIPS CPU instructions including the privileged instructions. It must also provide the virtual address to physical address translations done by the memory management unit (MMU). For the MIPS architecture this means implementing the associative lookup of the translation lookaside buffer (TLB), including raising the relevant exceptions if the translation fails.

SimOS models the behavior of I/O devices by responding to uncached accesses from the CPU, interrupting the CPU when an I/O request has completed, and performing direct memory access (DMA). For some devices, useful emulation requires communication with the nonsimulated host devices. For example, the SCSI disk simulator reads and writes blocks from a file in the host machine's file system, making it possible to transfer large amounts of data into the simulated machine by creating the appropriate disk image in a file. Similarly, the console and network devices can be connected to real terminals or networks to allow the user to interactively configure the workloads that run on the simulator.

The complete machine simulation approach differs from the approach generally used in simulation systems for studying application programs. Because application-level simulators are not designed to run an operating system, they only need to simulate the portion of the hardware interface visible to user-level programs. For example, the MMU is not visible to application programs and is not modeled by application-level simulators. However, complete machine simulators must perform an MMU lookup on every instruction. Although this requires additional work, complete machine simulators have many advantages as described in Section 3.1.

2.2 Interchangeable Simulation Models

Because of the additional work needed for complete machine simulation, SimOS includes a set of interchangeable simulation models for each hardware component of the system. Each of these models is a self-contained software implementation of the device's functional behavior. Although all models implement the behavior of the hardware components in sufficient detail to correctly run the operating system and application programs, the models differ greatly in their timing accuracy, interleaving of multiprocessor execution, statistics collected, and simulation speed.

Furthermore, the user can dynamically select which model of a hardware component is used at any time during the simulation. Each model supports the ability to transfer its state to the other models of the same hardware component. For example, the different MIPS CPU models transfer the contents of the register file and the translation-lookaside buffer. As we show in Section 3, the ability to switch between models with different simulation speed and detail is critical when studying large and complex workloads.

A complete machine simulation system must model all components of a computer system. However, only a few components play a determining factor in the speed of the simulation. The processors, MMUs, and memory hierarchy account for the bulk of the simulation costs. In the rest of this section, we summarize the implementation techniques used by SimOS to model these critical components.

2.2.1 *High-Speed Machine Emulation Models.* To support high-speed emulation of a MIPS processor and memory system, SimOS includes Embra [Witchel and Rosenblum 1996]. Embra uses the dynamic binary translation approach pioneered by the Shade system [Cmelik and Keppel 1994]. Dynamic binary translators translate blocks of instructions into code sequences that implement the effects of the original instructions on the simulated machine state. The translated code is then executed directly on the host hardware. Sophisticated caching of translations and other optimizations results in executing workloads with a slowdown of less than a factor of 10. This is two to three orders of magnitude faster than more conventional simulation techniques.

Embra extends the techniques of Shade to support complete machine simulation. The extensions include modeling the effects of the memory-management unit (MMU), privileged instructions, and the trap architecture of the machine. The approach used in Embra is to handle all of these extensions with additional code incorporated into the translations. For example, Embra generates code that implements the associative lookup done by the MMU on every memory reference. Embra also extends the techniques of Shade to efficiently simulate multiprocessors.

Besides its speed, a second advantage of using dynamic binary translation is the flexibility to customize the translations for more accurate modeling of the machine. For example, Embra can augment the emitted translations to check whether instruction and data accesses hit in the simulated cache. The result is that SimOS can generate good estimates of the instruction execution and memory stall time of a workload at a slowdown of less than a factor of 25.

2.2.2 *Detailed Machine Simulation Models.* Although Embra's use of self-generated and self-modifying code allows it to simulate at high speeds, the techniques cannot be easily extended to build more detailed and accurate models. To build such models, we use more conventional software engineering techniques that value clean well-defined interfaces and ease of programming over simulation speed. SimOS contains interfaces for supporting different processor, memory system, and I/O device models.

SimOS contains accurate models of two different processor pipelines. The first, called Mipsy, is a simple pipeline with blocking caches such as used in the MIPS R4000. The second, called MXS [Bennett and Flynn 1995], is a superscalar, dynamically scheduled pipeline with nonblocking caches such as used in the MIPS R10000. The two models vary greatly in speed and detail. For example, MXS is an order of magnitude slower than Mipsy, because the R10000 has a significantly more complex pipeline.

Mipsy and MXS can both drive arbitrarily accurate memory system models. SimOS currently supports memory system models for a bus-based memory system with uniform memory access time, a simple cache-coherent nonuniform memory architecture (CC-NUMA) memory system, and a cycle accurate simulation of the Stanford FLASH memory system [Kuskin et al. 1994].

For I/O device simulation, SimOS includes detailed timing models for common devices such as SCSI disks and interconnection networks such as Ethernet. For example, SimOS includes a validated timing model of the HP 97560 SCSI disk [Kotz et al. 1994].

3. USING SIMOS TO SIMULATE COMPLEX WORKLOADS

The use of complete machine simulation in SimOS would appear to be at odds with the goal of studying large and complex workloads. After all, simulating at the hardware level means that SimOS must do a significant amount of work to study a complex workload. Our experience is that complete machine simulation can actually simplify the study of complex workloads. Furthermore, by exploiting the speed/detail tradeoff through the use of interchangeable hardware models, SimOS can run these workloads without excessively long simulation time. In the rest of this section we describe how these two features of SimOS support the study of complex workloads.

3.1 Benefits of Complete Machine Simulation

Although complete machine simulation is resource intensive, it has advantages in ease of use, flexibility, and visibility when studying complex workloads.

Because SimOS provides the same interface as the target hardware, we can run an essentially unmodified version of a commercial operating system. This operating system can then in turn run the exact same applications that would run on the target system. Setting up a workload is therefore straightforward. We simply boot the operating system on top of SimOS, copy all the files needed for the workload into the simulated system, and then start the execution of the workload. In contrast, application-level simulators are generally not used to study complex workloads since they would need to emulate a significant fraction of the operating system's functionality to simply run the workload. This emulation task is likely to be more complex than a complete machine simulation approach.

For example, we have used SimOS to study complex multiprogrammed workloads such as parallel compilation and a client-server transaction processing database. We were able to simply copy the necessary programs and data files from an existing machine that runs the workloads. No changes to the application software or the simulator were required.

SimOS is flexible enough that it can be configured to model entire distributed systems. For example, we have studied file servers by simulating at the same time the machine running the file server software, the client machines, and the local area network connecting them. This made it possible to study the entire distributed system under realistic request patterns generated by the clients. Although there are certainly limits as to how far this approach will scale, we have been able to simulate tens of machines and should be able to study hundreds of machines.

Using software simulation has a number of additional advantages for SimOS. First, software simulation models are significantly easier to change than the real hardware of a machine. This makes it possible to study the effects of changes to the hardware. Secondly, simulating the entire machine at a low level provides SimOS excellent visibility into the system behavior. It "sees" all events that occur in the system, including cache misses, exceptions, and system calls, regardless of which part of the system caused the events.

3.2 Exploiting the Speed/Detail Tradeoff

SimOS's use of complete machine simulation tends to consume large amounts of resources. Furthermore, complex workloads tend to run for long periods of time. Fast simulation technology is required to study such workloads. For example, the study of a commercial data processing workload described in Section 5.1 required the execution of many tens of billions of simulated instructions to boot the operating system and start up the database server and client programs. Executing these instructions using the simulator with the level of detail needed for the study would have taken weeks of simulation time to reach an interesting execution state.

SimOS addresses this problem by exploiting the inherent tradeoff between simulation speed and simulation detail. Each of SimOS's interchangeable simulation models chooses a different tradeoff between simulation speed and detail. We have found the combination of three models to be of particular use: emulation mode, rough characterization mode, and accurate mode.

3.2.1 *Emulation Mode.* As indicated, positioning a complex workload usually requires simulating large amounts of uninteresting execution such as booting the operating system, reading data from a disk, and initializing the workload. Furthermore, issues such as memory fragmentation and file system buffer caches can have a large effect on the workload's execution. Many of these effects are not present in a freshly booted operating system;

they only appear after prolonged use of the system. Realistic studies require executing past these “cold start” effects and into a steady-state representative of the real system.

To position a workload, SimOS can be configured to run the workload as fast as possible. We refer to this configuration as emulation mode because its implementation shares more in common with emulation techniques than with simulation techniques. The only requirement is to correctly execute the workload; no statistics on workload execution are required.

Emulation mode uses Embra configured to model only the hardware components of the system that are necessary to correctly execute the workload. No attempt is made to keep accurate timing or to model hardware features invisible to the software, such as the cache hierarchy and processor pipelines. I/O devices such as disks are configured to instantaneously satisfy all requests, avoiding the time that would be required to simulate the execution of the operating system’s idle loop.

To enable the high speed emulation of multiprocessors, Embra can run as multiple parallel processes where each process simulates a disjoint set of processors. Embra can make highly effective use of the additional processors, achieving linear and sometimes superlinear speedups for the simulation [Witchel and Rosenblum 1996]. Embra is able to make such an optimization because it is only used in emulation mode, and that does not require the separate simulated processors to have their notions of time closely synchronized.

In parallel Embra, the scheduler of the host multiprocessor has an impact on the interleaving of the simulated processors. The final machine state is guaranteed to be one of a set of possibilities that are feasible if no timing assumptions are made about code execution. Note, however, that the simulation is not deterministic and that different simulation executions will result in different final machine states. In practice, the operating system and application software execute correctly independently of the actual interleaving of the instructions executed. As a result, all reached machine states, although temporally inaccurate, are functionally plausible, and can be used as the starting point for more accurate simulation models.

Early versions of SimOS contained an emulation mode based on direct execution of the operating system and the applications [Rosenblum et al. 1995]. Direct execution was frequently used to position the workloads, but was removed in 1996 in favor of the binary translation approach. Specifically, binary translation is more amenable to cross-platform support than direct execution mode.

3.2.2 *Rough Characterization Mode.* The speed of emulation mode is useful for positioning and setup of workloads, but the lack of a timing model makes it unsuitable for many uses. To gain more insight into the workload’s behavior, SimOS supports a rough characterization mode that maintains high simulation speed yet provides timing estimates that approximate the behavior of the machine. For example, rough characterization mode includes timing models that can track instruction execution, memory stall, and I/O device behavior, yet it is only two or three times slower than emulation mode.

Rough characterization mode is commonly used in the following ways. First, it is used to perform a high-level characterization of the workload in order to determine first-order bottlenecks. For example, it can determine if the workload is paging, I/O bound on a disk, or suffering large amounts of memory system stall. Since the simulation speed of rough characterization speed is similar to that in emulation mode, it can be used to examine the workload over relatively long periods of simulated time. The second common use of rough characterization is to determine the interesting parts of the workload that should be further studied in greater detail with the accurate simulators. The rough characterization provides enough information to determine the interesting points to focus on in the more detailed modes.

An example of the use of the rough characterization mode can be found in Bugnion et al. [1996]. The benchmark programs used in that study required far too much execution time to run the entire program in the accurate simulation modes. Instead we used the rough characterization mode to run the program to completion. We observed that all the benchmarks in this study had a regular execution behavior. This allowed us to study limited representative execution windows. Having the rough characterization of the entire benchmark gave us confidence that the selected window of the workload, studied in the accurate mode, would be representative of the benchmark as a whole.

The rough characterization mode uses Embra configured to model a simple CPU pipeline and a large unified instruction and data cache much like the second-level cache of the MIPS R4000. The memory system models a fixed delay for each cache miss. I/O devices use a realistic timing model. For example, the SCSI disk models seek, rotation, and transfer times. This mode gives estimates of the instruction, memory system behavior, and I/O behavior of the workload.

3.2.3 *Accurate Mode.* The accurate simulation mode tries to model a given hardware configuration as accurately as possible. Because of their detail, these configurations lead to simulation speeds that are too slow to use for workload positioning. The goal of accurate mode is to collect detailed statistics on the workload's execution behavior. Essentially all results reported in studies that use SimOS were generated in the accurate mode.

In this mode, we use either the Mipsy or MXS processor models to study the performance of a simple processor pipeline or of a dynamically scheduled processor. Because of the complexity of the dynamically scheduled processor it models, MXS can only simulate on the order of 20,000 instructions per second when run on current machines. Because of this slow simulation speed, it takes a long time for the simulation to warm up the state of the cache hierarchy. We therefore generally use Mipsy, which is an order of magnitude faster, to warm up the caches before switching into MXS.

SimOS includes even more detailed gate-level models of some hardware components of the FLASH machine. Unfortunately, these models are so detailed that the simulation speed is limited to a few simulated cycles per second. With such slowdowns, simulating even a single transaction of a database workload is infeasible.

To study such workloads, we use random sampling techniques that switch between different levels of detail. This allows us to use statistical analysis to estimate the behavior of the most detailed models during the execution of the workload. Sampling is also used to switch between the Mipsy and MXS processor models. For example, two architectural studies sampled 10% of the executed cycles using MXS, running the remainder in the faster Mipsy [Nayfeh et al. 1996; Wilson et al. 1996].

4. DATA COLLECTION MECHANISMS

Low-level machine simulators such as SimOS have a great advantage in that they see all events that happen on the simulated system. These events include the execution of instructions, MMU exceptions, cache misses, CPU pipeline stalls, and so on. The accurate simulation models of SimOS are heavily instrumented to collect both event counts and timing information describing the simulated machine's execution behavior. Unfortunately, when studying complex systems, collecting these data presents two problems. The data are generated at a low hardware level and at a high rate.

The low hardware level of data collected is problematic because the user needs to assign costs to higher-level concepts such as processes or transactions that are not known to the hardware. For example, tracking memory stalls by physical memory address is not useful if the mapping from the physical address back to the virtual address of the process being studied is not known. Even if the memory stalls are recorded by virtual address, it is often difficult to determine to which process they correspond in a multiprogrammed workload.

Mapping events to higher-level concepts is also important when studying the behavior of the hardware. For example, the cycles per instruction (CPI) of a processor is a useful metric to computer architects only if the instructions are factored out that are executed while in the operating system's idle loop.

The classification of events in SimOS is further complicated by the high rate at which the data are generated. Unless the classification and recording mechanism is efficient, the overall performance of the simulation will suffer.

To address these challenges, SimOS's data classification mechanisms need to be customized for the structure of the workload being studied as well as the exact classification desired. A Tcl scripting language interpreter [Ousterhout 1994] embedded in SimOS accomplishes this in a simple and flexible way. Users of SimOS can write Tcl scripts that interact closely with

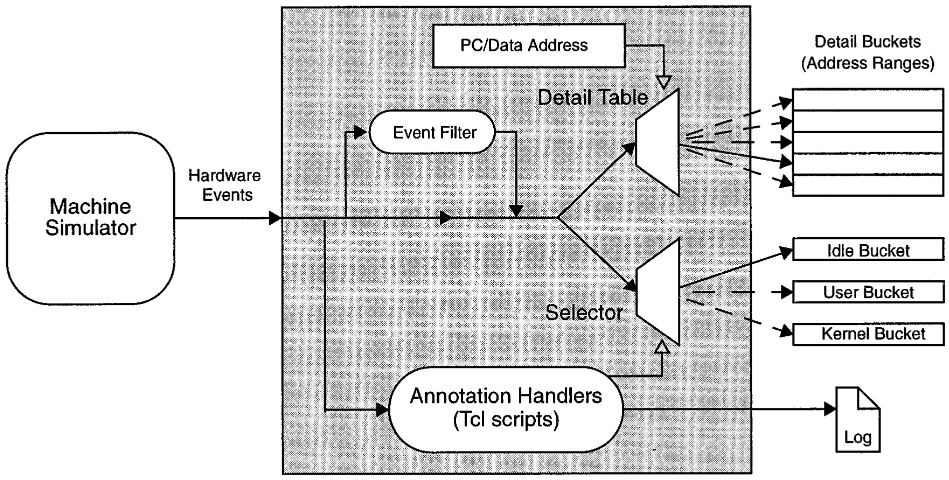


Fig. 2. Data collection in SimOS.

the hardware simulation models to control data recording and classification. These scripts can nonintrusively probe the state of the simulated machine and therefore can make classification decisions based on the state of the workload. The use of a scripting language allows users to customize the data collection without having to modify the simulator.

In the rest of this section we describe the data recording and classification mechanism of SimOS. Figure 2 illustrates the architecture of the mechanisms. Hardware events are recorded in different buckets. They also trigger the annotation scripts used to control to which bucket a selector points. In this example, events can be recorded according to whether they occurred in user, kernel, or idle mode. The detail tables classify events according to the instruction and data addresses that cause them. Finally, the event filter generates an additional set of events used to characterize the memory system behavior. The Tcl language is extended with the concept of annotations, which are Tcl scripts that allow the simulator to efficiently track the state of the workloads. Annotations are described in Section 4.1. To allow for highly efficient recording of these data, SimOS supports selectors and detail tables as described in Section 4.2. Finally, Section 4.3 presents the event filter mechanism used in SimOS.

4.1 Annotations

Annotations are the key to mapping low-level events to higher-level concepts. Annotations are Tcl scripts that the user can attach to specific hardware events. Whenever an event occurs that has an annotation attached to it, the associated Tcl code is executed. Annotations can run without any effect on the execution of the simulated system. Annotations have access to the entire state of the simulated machine, including registers, TLBs, devices, caches, and memories. Furthermore, annotations can access the symbol table of the kernel and applications running in the simulator, allowing symbolic references to procedures and data.

By using annotations, a user of SimOS can build higher-level knowledge about the workload's execution into the simulator. For example, by reading from data structures in the simulated machine's operating system kernel, an annotation can discover the process ID of the currently running process. This information can then be used by the annotation to classify data by process ID.

Examples of simulated hardware events on which annotations can be set include:

—Execution reaching a particular program counter address. This type of annotation is invoked whenever a processor executes an instruction at a particular virtual address. This address can be either in the kernel's address space or within the context of a particular user application. Since SimOS can access the symbol information of any executable, program counter values can be specified symbolically.

—Referencing a particular data address. These annotations are similar to debugger watchpoints and are set on virtual addresses. They are invoked whenever a memory location in a specified range is read or written. As with program counter annotations, data structure annotations can be specified using symbols from the application being studied. Data address annotations are useful for debugging data corruption problems.

—Occurrence of an exception or interrupt. Annotations can be set on any or all types of traps or interrupts. For example, we use annotations on the system call exception in order to classify the behavior of the operating system by the system services it provides.

—Execution of a particular opcode. Annotations may be set for particular opcodes or classes of opcodes. For example, in the MIPS architecture, an rfe (return from exception) or eret (exception return) instruction is executed to return from an exception. Annotations set on these instructions allow SimOS to efficiently track whether a processor is in kernel or user mode.

—Reaching a particular cycle count. These annotations are triggered after a specified number of simulation cycles and are typically used to periodically output statistics.

Annotations may also trigger other events that correspond to higher-level concepts. Using this mechanism we can define annotation types that are triggered by software events. For example, an annotation on the operating system's context switching code triggers a "context switch" event which may be of interest to a user studying an application program. The user can determine when the process is active without having to directly instrument the operating system to gather this information.

Annotations can be efficiently implemented even in emulation mode. In Embra, program counter annotations are implemented by generating custom translations for annotated program counter addresses that invoke the

```
# Define a new annotation type for process events
annotation type process enum {switchOut switchIn}

# Program Counter annotation at the end of the exec system call
annotation set pc kernel:exece:END {
        # On an exec, only the name of the process changes, not the pid
        set PROCESS($CPU) [symbol read kernel:u.u_comm]
}

# Program Counter annotation at the end of the context-switching code
annotation set pc kernel:resume:END {
        # Execute higher-level annotation
        annotation exec process switchOut
        # Update executable name and pid
        set PID($CPU) [symbol read kernel:u.u_procp->pid]
        set PROCESS($CPU) [symbol read kernel:u.u_ucomm]
        annotation exec process switchIn
}

# Program Counter annotation at the beginning of the idle loop
annotation set pc kernel:idle:START {
        annotation exec process switchOut
        set PID($CPU) -1
        set PROCESS($CPU) "Idle"
        annotation exec process switchIn
}
```

Fig. 3. Process-tracking Tcl script.

Tcl interpreter. Program counter addresses for which there are no annotations set do not call into the Tcl interpreter so there is no overhead when annotations are not being used.

Figure 3 shows an example of how annotations are used to build higher-level knowledge of the operating system into the simulator. The state machine is driven by three program counter annotations set on addresses in the kernel's text. The script raises the level of abstraction of system events to include operating system events. Client modules can install annotations on *process switchIn* and *process switchOut. symbol* is a Tcl command added to SimOS that symbolically accesses data structures in the simulated machine. "u" is a variable in the operating system that gives access to the process table entry of the current process. We use a group of program counter annotations on the kernel context-switching code to track the currently scheduled process on each CPU. These annotations are set on the process management system calls, in the context-switching code, and at the beginning of the kernel idle loop. A Tcl array maintains the current process ID (PID) and process name for each processor. Although this process-tracking module does not generate performance data directly, it is used by other modules to attribute events to specific processes.

We have developed a large set of annotations that build knowledge about the state of IRIX. This includes scheduling, synchronization, virtual memory, I/O, and system call behavior. These annotations are clearly dependent on the target operating system. However, SimOS itself remains free of any operating system dependency. Porting another operating system to the SimOS MIPS architecture only requires rewriting portions of the Tcl scripts.

4.2 Event Classification

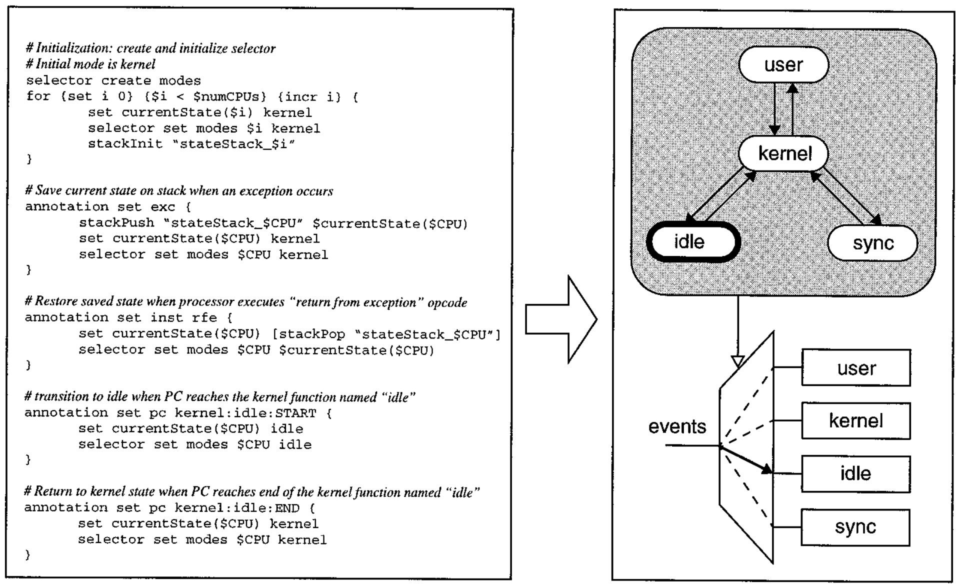
Although we could place annotations on each hardware event and count them in Tcl, the frequency at which events can occur in the faster simulators of SimOS makes this too slow. To avoid spending too much time in the Tcl interpreter, annotations are used to control how the frequent events are recorded rather than having the events themselves invoke annotations. As shown in Figure 2, hardware events are processed through an event classifier (either a selector or a detail table) and recorded into data collection buckets.

With the selector mechanism, the Tcl scripts control which of a set of predefined buckets is used to record events. Once an annotation has set the selector to point to a particular bucket, all subsequent events will be funneled into that bucket. With this model, buckets correspond to different phases of the workload, and annotations are set on events that signal the start of a new phase.

Detail tables are like selectors except the target bucket is computed based on the address of the event. The address can be either the current value of the program counter or the address of a memory access. Detail tables allow events to be classified based on the code or data that caused them.

Figure 4 shows how the selector mechanism can be used to separate the execution of the workload into four basic modes that correspond to execution at user level, in the kernel, in the kernel synchronization routines, and in the kernel idle loop. The left side of the figure shows the Tcl source used to implement the processor tracking functionality. The script implements a state machine for each processor. The state machine is shown in the top right of the figure. The state machine also controls the setting of a selector, as shown in the bottom right of the figure. The source code has been simplified and does not include the annotations that define synchronization. Annotations placed on exceptions and the return-from-exception opcode, at the start and end of the idle function, and the kernel synchronization routines are used to direct statistics into the bucket corresponding to the current mode.

The event classification mechanisms of SimOS lend themselves to an efficient implementation. Accessing the counter associated with the event requires a simple pointer dereference. In the common case, the Tcl interpreter is not invoked. Annotations only need to be placed on events that change the selector to point to a new bucket. In practice, this occurs infrequently. Detail tables are implemented similarly except that the bucket is determined automatically as a function of the address of the event.



```
# Initialization: create and initialize selector
# Initial mode is kernel
selector create modes
for {set i 0} {$i < $numCPUs} {incr i} {
        set currentState($i) kernel
        selector set modes $i kernel
        stackInit "stateStack_$i"
}

# Save current state on stack when an exception occurs
annotation set exc {
        stackPush "stateStack_$CPU" $currentState($CPU)
        set currentState($CPU) kernel
        selector set modes $CPU kernel
}

# Restore saved state when processor executes "return from exception" opcode
annotation set inst rfe {
        set currentState($CPU) [stackPop "stateStack_$CPU"]
        selector set modes $CPU $currentState($CPU)
}

# transition to idle when PC reaches the kernel function named "idle"
annotation set pc kernel:idle:START {
        set currentState($CPU) idle
        selector set modes $CPU idle
}

# Return to kernel state when PC reaches end of the kernel function named "idle"
annotation set pc kernel:idle:END {
        set currentState($CPU) kernel
        selector set modes $CPU kernel
}
```

Fig. 4. Processor mode tracking.

4.3 Event Filters

Although the event classification mechanisms in SimOS can control the recording of hardware events, frequently it is not sufficient to know that an event occurred; the cause of the event is needed. An example of this is memory stall caused by cache misses on a shared-memory multiprocessor. Knowing that some code or a data structure suffered cache misses does not necessarily tell the programmer if these cache misses are necessary or how they can be avoided.

To address this problem, SimOS provides a mechanism for installing filters on the hardware event stream that provide additional information about events. Figure 5 illustrates the state machine of an example filter, used for classifying the cache misses suffered by each memory location. SimOS instantiates one such state machine per cache line and processor. At any point in time, the line is either in a processor's cache or not. The bold lines represent cache misses. The dotted lines represent transitions that occur either when the line is removed from the processor's cache or when the cache line is invalidated. Misses are classified as either cold, replacement, or invalidation misses. The first reference to a particular memory block by a particular processor is classified as a cold miss. Replacement misses occur as a result of the limited capacity or associativity of the cache. Replacement misses consist of both conflict and capacity misses. Invalidation misses are a result of cache-coherency protocols and are an indication of interprocessor communication.

The filter can be used in conjunction with detail tables to associate memory stalls with particular pieces of code or data structures, which can be valuable to parallel application developers. Cold misses are typically

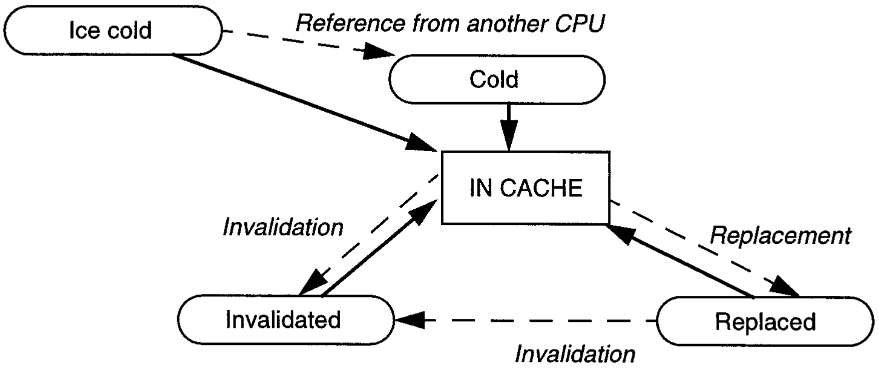


Fig. 5. Cache miss classification.

unavoidable, whereas replacement misses can sometimes be eliminated by restructuring data structures to eliminate conflict misses due to the limited associativity of the cache. Excess invalidation misses may indicate that better control of the interprocessor communication is needed.

To further help programmers, SimOS supports a second filter that classifies cache misses according to the true or false sharing types defined by Dubois et al. [1993]. Misses are classified as true sharing when data are actually being communicated between two processors. False sharing typically occurs when different processors read and write different portions of a particular cache line. False sharing is a byproduct of the cache coherence protocol and occurs when a cache line is invalidated from a cache even though no actual data communication takes place in the application. When this state machine is in use, SimOS tracks not only cache misses but also all data references.

5. EXPERIENCE USING SIMOS

SimOS has been used in several contexts, including the investigation of new architectural designs [Nayfeh et al. 1996; Wilson et al. 1996; Olokotun et al. 1996], the development of the Hive operating system [Chapin et al. 1995; Rosenblum et al. 1995], and for various performance studies of operating systems [Rosenblum et al. 1995; Verghese et al. 1996] and applications [Bugnion et al. 1996]. This section presents two case studies to illustrate how the different features of SimOS can be utilized.

5.1 Case Study: Operating System Characterization

We used SimOS to characterize the IRIX operating system and to predict the impact of architectural trends on its performance [Rosenblum et al. 1995]. SimOS was a critical component of this study for several reasons. First, we required realistic workloads that stressed the operating system in significant ways; toy applications and microbenchmarks do not drive the operating system realistically, and thus cannot provide an accurate picture of overall operating system performance. Second, the IRIX operating sys-

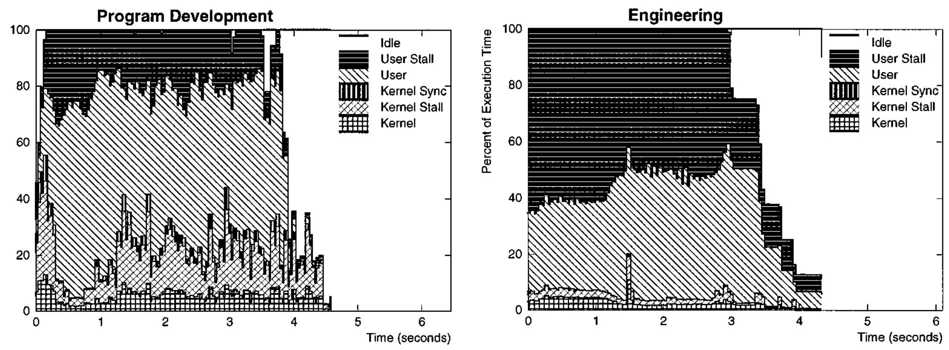


Fig. 6. Processor mode breakdown.

tem is large, complex, and multithreaded. We needed a flexible data characterization mechanism to help make sense of the wealth of information generated. In fact, many of the data collection mechanisms of SimOS were developed during this study. Third, the goal of the study was to analyze the behavior of the operating system on machines that are likely to appear several years in the future. The flexibility of complete machine simulation allowed us to model hardware platforms before they were available.

To stress the operating system in realistic ways, we picked workloads that traditionally run on workstations and shared-memory multiprocessors: a commercial database workload, a program development workload, and an engineering simulation workload. We configured SimOS to simulate a machine with 128 MB of RAM and up to eight processors.

The first step in preparing the workloads was to position them using the fast emulation mode. This included booting the operating system and running through the initialization phases of the applications. We ensured that the system had run long enough to get past the cold start effects due to the operating system boot.

Once the workloads were initialized to an interesting point, we used SimOS's checkpointing facility to transfer the entire machine state from the simulation models into a set of files. The checkpoint contained all software-visible machine state including the contents of physical memory, all registers, and the state of I/O devices. A single checkpoint can be restored in multiple different hardware configurations and characterization modes.

We restored the checkpoints and ran them to completion using the rough characterization mode while employing the annotations shown in Figure 4. This gave us a temporal characterization of each workload, and allowed us to pinpoint the interesting sections. Figure 6 shows examples of the rough characterization results for two of the workloads. This figure shows the execution profile of a program development workload (left) and an engineering workload (right) running on an 8-CPU multiprocessor. The execution time of each processor is broken down between the kernel, synchronization, user, and idle modes. For user and Kernel modes, the graph further

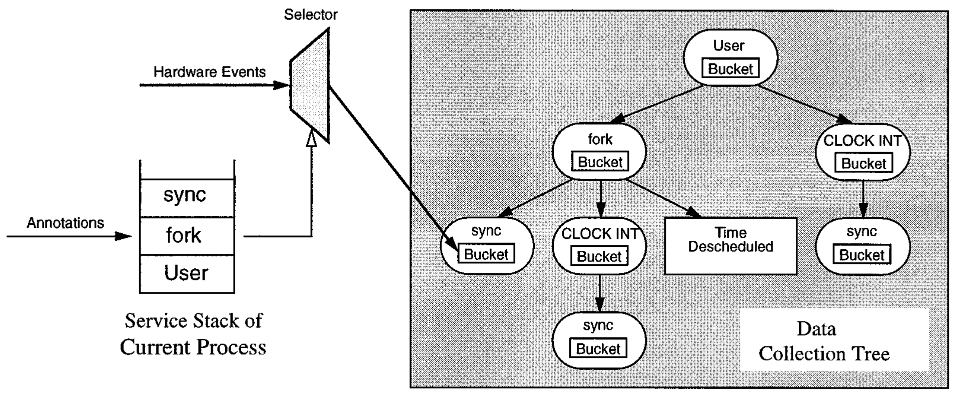


Fig. 7. Timing decomposition.

separates the time spent executing instructions and the time stalled in the memory system. The breakdown, averaged over 8 processors, is determined for each 20-millisecond time step.

We then used the accurate simulation modes, configured to model future generation hardware, to collect relevant statistics. Checkpoints were crucial during this process because we were able to replay the exact same workload under three different hardware models. This allowed us to make a fair comparison between the operating system performance on machines typical in 1994 and those likely to appear in 1996 and 1998.

The annotation mechanism of SimOS allowed us to analyze the execution of the kernel in many different ways. One way we found particularly interesting was to classify the kernel execution time by the service the kernel was providing to user level processes. These services include the various system calls, virtual-memory fault processing, exceptions, and interrupt processing. A key advantage of this decomposition is that it allowed us to compare the performance of the same service across workloads and architectures. Many of the services share the same subroutines and have common execution paths, so it would be impossible to do this decomposition based solely on program counter addresses.

Generating this decomposition required extensive use of annotations and selectors. It was insufficient to simply put annotations at the entry and exit from the kernel. With interrupt processing, services can be nested within other services. Furthermore, certain services can be descheduled, for example, waiting for an I/O operation to complete. The solution was to set annotations throughout the operating system code to track the service currently executing. We set annotations on exceptions, synchronization routines, the scheduler, and the idle loop. Among other functions, these annotations tracked interrupts and when processes were descheduled for I/O.

We used the flexibility of Tcl to arrange the buckets into a tree structure as shown in Figure 7. A complete breakdown of a workload's execution can be easily obtained using the timing mechanism provided by SimOS. In this example, the system is currently performing a synchronization action within the fork system call. The selector will thus attribute all of the current events to this particular node in the tree, differentiating the execution of synchronization routines acting on behalf of the fork from those acting on behalf of the clock interrupt. The "time descheduled" box is also a useful feature. This extra state in the tree allows us to track the overall latency of a service, including the effects of any context switches that occur during its execution. This tree structure enabled us to discriminate among the various kernel services and to decompose each service into its components such as synchronization or time spent descheduled. Collecting data in this tree form made it possible to defer much of the analysis of the data to postprocessing phases. For example, from the tree in Figure 7 it would be possible to compute the fraction of time spent in the fork system call waiting on kernel synchronization. It would also be possible to compute the total kernel synchronization time by summing all the "sync" buckets.

In order to point the selector at the right bucket in the tree, annotations were set that maintain the notion of the current service for each processor. To handle the case of nested services, the current service is kept on a stack. When a service starts, the annotation pushes the new service on the stack which then becomes the current service. When the service ends, the annotation removes it from the stack. Since a process might be descheduled at any point during the execution of a service, the annotations actually maintain a separate stack for each process. As a result, the current state of the system is represented as an array of stacks, and the execution behavior of the workload is represented as a tree of buckets.

This breakdown of kernel services was important in our study as it allowed us to compare the performance of the same services across different workloads, number of processors, and generations of hardware. For example, we observed differences in the performance improvements of different services across architectural trends. We also observed that most operating system services in the multiprocessor workloads consumed 30 to 70% more time than their uniprocessor counterparts.

This type of decomposition can be easily extended to include any aspect of kernel performance. In fact, we found this type of decomposition to be so useful that we developed it into a general mechanism that can separate the execution of a workload, including application programs, into different phases. Our experiences have shown this form of user-defined decomposition to be much more useful than the simple procedural breakdown offered by conventional profilers.

5.2 Case Study: The SUIF Parallelizing Compiler

We used SimOS to study automatically parallelized applications generated by SUIF, a research parallelizing compiler [Wilson et al. 1994]. We were approached by the SUIF group because they were not achieving the performance they expected for some applications, and were unable to identify the source of the performance problems using the tools available to them. Using SimOS, we were able to discover the sources of the problems and suggest several performance improvements. After implementing all the performance improvements suggested, we ran the SUIF-generated applications on a high-end multiprocessor and measured the highest SPEC95fp ratio reported to date [Bugnion et al. 1996].

Because SimOS supports IRIX, workload preparation was simple. We compiled the SUIF applications on a Silicon Graphics workstation and copied the executables and their input files onto the simulated multiprocessor's disk. No modifications to the applications were required. We employed SimOS's emulation mode to quickly boot the operating system and position the workload. This process, including the boot of IRIX, took less than 10 minutes of simulation time. To avoid excessive simulation time due to the long execution times of the SPEC95fp benchmarks—on the order of a couple of minutes on today's fastest machines—we used the rough characterization mode to pinpoint portions of the execution that would be representative of the overall workload. We then ran only those portions in the accurate models.

Using SimOS, we were able to identify several performance problems and suggest solutions. The regular structure of SUIF-generated applications made them easy to annotate. SUIF generates sections of parallel code interleaved with sections of sequential and other setup code. Annotations were set at the beginning and end of the parallel sections, around the sequential sections, and in the synchronization routines. Coupled with selectors, these annotations separated the "useful" execution from compiler-added administration and synchronization overheads. The annotations also allowed us to quantify the different sources of overhead such as barrier synchronization and load imbalance. We were able to determine that the fine granularity of parallelism exploited by the SUIF compiler was resulting in large overheads due to barrier synchronization at the end of each parallel section. We then optimized the barrier code in the compiler's run-time system, significantly improving the performance of some applications.

We also used the true sharing/false sharing cache miss classification filter described in Section 4.3 to study each application's communication patterns. Using this classification we found a striking behavior in cfft2, one of the NASA7 benchmarks. SimOS reported that 84% of all cache misses on the primary matrix used in the computation were due to false sharing. By modifying the compiler to align all shared data structures on cache line boundaries, we were able to completely eliminate this false sharing.

Other SimOS cache statistics generated in rough characterization mode indicated that conflict misses were a problem for several applications in the SPEC95fp benchmark. Using annotations placed on the cache misses, we were able to collect information about the physical addresses of the frequently used lines. From this information we determined that the page-mapping policy of the operating system led to an inefficient utilization of the cache in parallel execution. This led us to develop a new page-mapping algorithm that significantly improve the performance of compiler-parallelized applications. The ability of SimOS to precisely locate and quantify cache misses was instrumental in the development of the algorithm. For complete information regarding this study, see Bugnion et al. [1996].

6. RELATED WORK

In this section, we compare SimOS to other tools used for studying computer systems. Specifically, we focus on the most relevant features of SimOS including the use of complete machine simulation to study complex workloads, the exploitation of high-speed simulation technology, and the flexible event classification mechanisms.

Although simulation has been widely used to study computer systems, most simulators do not include the effects of operating system execution.[1] The operating system is usually ignored and treated as having no important impact on the conclusions of a particular study. For example, most tools only deal with virtual addresses, avoiding the difficulties of address translation. They also charge only a fixed cost for system calls and include none of their timing and cache pollution effects.

6.1 Techniques for Studying Complete Machines

SimOS's approach of using a complete machine simulator to study complex multiprogrammed workloads differs from most other tools. These tools typically use instrumentation of the system to collect a trace of the behavior that can be processed to extract useful information. This involves running the workload of interest on a system modified to record the trace. The trace can be generated either using software systems such as ATOM [Eustace and Srivastava 1995], Epoxie [Borg et al. 1989], FastCache [Lebeck and Wood 1995], Maxpar [Chen 1985], PatchWrk [Perl and Sites 1996], pixie [Smith 1991], Proteus [Brewer et al. 1991], Paradyn [Miller et al. 1995], or TangoLite [Goldschmidt 1993], or by using a hardware monitor as done in DASH [Chapin et al. 1995; Torrellas et al. 1992] and BACH [Grimsrud et al. 1993].

The popularity of instrumentation approaches attests to their wide applicability and their relative ease of implementation. Some of the challenges facing SimOS are nonissues for instrumentation approaches. For example, instrumentation is generally fast, because the workload can be positioned with all instrumentation turned off. Instrumentation also works well over long periods of execution when only limited events need to be collected into the trace.

SimOS has a number of advantages over the instrumentation approach, including complete event coverage and nonintrusiveness. Care has to be taken when using instrumentation to avoid missing important events. For example, binaries that are not fully instrumented execute correctly but do

[1]See Cmelik and Keppel [1994], Brewer et al. [1991]; Goldschmidt [1993], Irvin and Miller [1996], and Veenstra [1993].

not generate the complete event stream. Furthermore, for bus monitors, only the events collected by the hardware monitor are visible.

Similarly, intrusiveness is often a problem in instrumentation-based systems. Software instrumentation requires rewriting existing applications, resulting in an application that is both larger and longer running than in its original form. As a result, the events generated may not match what actually occurs during application execution. Hardware instrumentation is typically less intrusive, but still can be a factor if it generates excessive data that must be manipulated during workload execution.

Other interesting techniques used to study systems include using the ECC bits of main memory to track cache misses as was done by the Wisconsin Wind Tunnel [Reinhardt et al. 1993], interrupt-based profiling such as prof [Silicon Graphics], and on-chip counters [Chen et al. 1995]. These techniques provide a way to study some details of a workload running on the system but lack the flexibility of complete system simulation.

6.2 High-Speed Simulation

The SimOS approach of using high-speed instruction set simulation to study a system's behavior has been used by a number of different projects. For example, Talisman [Bedicheck 1995] statically generates "threaded-code" from an application to examine its behavior on a multicomputer. Shade [Cmelik and Keppel 1994] and SimICS [Magnusson and Werner 1995] use special compilation and code caching techniques to run an optimized instruction set simulation directly on the host platform. These systems share the advantages of software simulation, such as full event coverage and nonintrusiveness, but do not model the complete hardware of a machine and thus cannot be used to study operating systems or multiprogrammed workloads.

In addition to simulation studies, this technology is being applied to the domain of cross-platform software emulation [Insignia Software 1996; Sun Microsystems 1996].

6.3 Flexible Event Classification

Mapping the low-level machine events back to higher-level concepts is a problem faced by practically all tools for studying systems. Instrumentation-based systems must not only capture the events of interest, but also include information about the workload in order to map these events to their cause. For example, in Chapin et al. [1995] the monitor could only capture references that reached the memory bus. Therefore, the operating system had to be altered to output uncached references that recorded necessary information about the workload.

Memspy [Martonosi et al. 1992] and Flashpoint [Martonosi et al. 1996] support classification of cache misses that occur during the execution of an application and can map misses back to the data structures that caused them. Event classification can occur concurrently with event generation.

For example, Tango Lite [Goldschmidt 1993] and MINT [Veenstra 1993] include cache models that categorize memory reference events as they arrive and even provide timing feedback to the event generator.

A particularly interesting example of the interaction between event generation and processing is found in Paradyn [Miller et al. 1995]. During the execution of an application, Paradyn recognizes troublesome sections of code and directs the event generation mechanism (a code annotator) to produce more detailed events for processing. As a result, the simulator only generates those events that are needed, reducing overall simulation time. SimICS also recognizes the differing levels of detail that may be implemented with run-time code generation, gathering only the amount of detail desired by the user.

SimOS's event classification mechanisms borrow from all these approaches. We incorporate the cache miss classification used in Memspy and exploit the speed/detail tradeoff in a way similar to Paradyn. SimOS also employs both dynamic and post-run event classification. Almost any event of interest can be sent in real-time to a separate event processing component. A user can observe workload behavior as it occurs, dynamically adjusting event collection as desired. In addition, several core event counts are periodically saved to a log file to allow examining aspects of workload execution without requiring the entire simulation to be re-executed.

Once the events have been appropriately classified, the data must be presented effectively to the user. Although SimOS's event classification mechanisms are quite mature, its visualization tools are less refined than several performance analysis tools [Reed et al. 1995; Bunde et al. 1996]. To address this deficiency, the next major area of SimOS research will concentrate on interfaces and visual metaphors for effective presentation of processed event information.

7. CONCLUSIONS

By combining the increasing power of today's computers with advances in high-speed machine simulation, we have shown that entire computer systems can be effectively studied using low-level machine simulators. SimOS has been successfully used by a wide range of researchers, from computer architects studying processor pipeline micro-architecture to programmers building transaction processing database systems. Because of its flexibility and system visibility, SimOS has been particularly useful to operating system researchers.

We have found some key features of SimOS critical for studying complex systems. The first key feature is the explicit speed/detail tradeoff offered by the interchangeable simulation models. This has enabled SimOS to have both the speed to position complex workloads and the accuracy to study them in detail. We found three points on the speed/detail tradeoff curve to be of particular interest. Emulation mode focuses on simulation speed and is used to position workloads for detailed study. Rough characterization mode provides estimates of workload performance at relatively high speeds.

Finally, the accurate mode is made up of a collection of detailed simulation models used to generate the final performance data.

The second key feature of SimOS is its ability to flexibly and nonintrusively map low-level machine events generated by the simulator to abstractions meaningful to the user. To accomplish this, SimOS uses the Tcl scripting language extended with annotations, selectors, and detail tables. The user can download Tcl scripts into SimOS to control the simulator, the data that are collected, and how the data are classified.

Ultimately, the value of a tool such as SimOS is measured by the studies it enables. By this measure SimOS has already been successful. SimOS is freely available in source form to the research community and is being used by a number of academic and industrial research efforts.

ACKNOWLEDGMENTS

The authors would like to thank the rest of the Stanford FLASH OS group for their contributions to SimOS, both as users and as developers. We also thank Emmett Witchel, Embra's original developer, whose contribution to SimOS lasted longer than his stay in our group.

REFERENCES

BEDICHECK, R. 1995. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 14–24.

BENNETT, J. AND FLYNN, M. 1995. Performance factors for superscalar processors. Tech. Rep. CSL-TR-95-661, Stanford University.

BORG, A., KESSLER, R., LAZANA, G., AND WALL, D. 1989. Long address traces from RISC machines: Generation and analysis. Tech. Rep. 89/14, DEC Western Research Laboratory.

BREWER, E., DELLAROCAS, C., COLBROOK, A., AND WEIL, W. 1991. Proteus: A high-performance parallel-architecture simulator. Tech. Rep. MIT/LCS/TR-516, MIT.

BUGNION, E., ANDERSON, J., MOWRY, T., ROSENBLUM, M., AND LAM, M. 1996. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct.), 244–257.

BUNDE, M. K., METCALFE, D., AND NOTTINGHAM, K. Visual tools unlock peak performance. Web site, http://www.cray.com/PUBLIC/product-info/sw/PE/vistools.html.

CHAPIN, J., HERROD, S. A., ROSENBLUM, M., AND GUPTA, A. 1995. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of the 1995 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95/PERFORMANCE '95)* (May), 1–13.

CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Dec.), 12–25.

CHEN, D.-K. 1995. MaxPar: An execution driven simulator for studying parallel systems. Ph.D. Thesis, University of Illinois at Urbana-Champaign.

CHEN, J. B., ENDO, Y., CHAN, K., MAZIERES, D., DIAS, A., SELZER, M., AND SMITH, M. D. 1995. The measured performance of personal computer operating systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Dec.), 299–313.

CMELIK, R. F. AND KEPPEL, D. 1994. Shade: A fast instruction set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 128–137.

DUBOIS, M., SKEPPSTEDT, J., RICCIULLI, L., RAMAMURTHY, K., AND STENSTROM, P. 1993. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the Twentieth International Symposium on Computer Architecture* (May), 88–97.

EUSTACE, A. AND SRIVASTAVA, A. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems* (Jan.).

GOLDSCHMIDT, S. 1993. Simulation of multiprocessors: Accuracy and performance. Ph.D. Thesis, Stanford University.

GRIMSRUD, K., ARCHIBALD, J., RIPLEY, M., FLANAGAN, K., AND NELSON, B. 1993. BACH: A hardware monitor for tracing microprocessor-based systems. *Microprocess. Microsyst. 17,* 443–459.

INSIGNIA SOFTWARE. SoftPC product information. Web Site, http://www.insignia.com.

IRVIN, R. B. AND MILLER, B. P. 1996. Mapping performance data for high-level and data views of parallel program performance. In *Proceedings of the International Conference on Supercomputing* (May).

KOTZ, D., TOH, S. B., AND RADHAKRISHNAN, S. 1994. A detailed simulation of the HP 97560 disk drive. Tech. Rep. PCS-TR94-20, Dartmouth College.

KUSKIN, J., OFELT, D., HEINRICH, M., HEINLEIN, J., SIMONI, R., GHARACHORLOO, K., CHAPIN, J., NAKAHIRA, D., BAXTER, J., HOROWITZ, M., GUPTA, A., ROSENBLUM, M., AND HENNESSY, J. 1994. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture* (April), 302–313.

LEBECK, A. R. AND WOOD, D. A. 1995. Active memory: A new abstraction for memory-system simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 220–230.

MAGNUSSON, P. AND WERNER, B. 1995. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium* (April).

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. E. 1992. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June), 1–12.

MARTONOSI, M., OFELT, D., AND HEINRICH, M. 1996. Integrating performance monitoring and communication in parallel computers. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 138–147.

MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. 1995. The Paradyn parallel performance measurement tools. *IEEE Computer* (Nov.), 37–46.

NAYFEH, B., HAMMOND, L., AND OLOKOTUN, K. 1996. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture* (May).

OLOKOTUN, K., NAYFEH, B., AND HAMMOND, L. 1996. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct.), 2–11.

OUSTERHOUT, J. K. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA.

PERL, S. E. AND SITES, R. L. 1996. Studies of Windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation,* 169–184.

REED, D. A., AYDT, R. A., MADHYASTHA, T. M., NOE, R. J., SHIELDS, K. A., AND SCHWATZ, B. W. 1995. An overview of the Pablo performance analysis environment. Tech. Rep., University of Illinois at Urbana-Champaign.

REINHARDT, S., HILL, M., LARUS, J., LEBECK, A., LEWIS, J., AND WOOD, D. 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 48–60.

ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995a. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles,* 285–298.

ROSENBLUM, M., CHAPIN, J., DEVINE, S., TEODOSIU, D., LAHIRI, T., AND GUPTA, A. 1995b. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Dec.), 12–25.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer simulation: The SimOS approach. In *IEEE Parallel Distrib. Technol.,* (Winter), 34–43.

SILICON GRAPHICS. gprof. IRIX 5.2 man page.

SMITH, M. D. 1991. Tracing with pixie. Tech. Rep. CSL-TR-497, Stanford University.

SUN MICROSYSTEMS. Wabi 2.2 product overview. Web Site, http://www.sun.com/solaris/products/wabi.

TORRELLAS, J., GUPTA, A., AND HENNESSY, J. 1992. Characterizing the cache performance and synchronization behavior of a multiprocessor operating system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct.), 162–174.

VEENSTRA, J. 1993. Mint tutorial and user manual. Tech. Rep. 452 (May), University of Rochester.

VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSENBLUM, M. 1996. Operating system support for improving data locality on cc-numa computer servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct.), 279–289.

WILSON, K. M., OLUKOTUN, K., AND ROSENBLUM, M. 1996. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture* (June), 147–157.

WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S.-W., TSENG, C.-W., HALL, M., LAM, M., AND HENNESSY, J. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Not. 29,* 12 (Dec.).

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 68–79.

Received June 1996; revised September 1996 and October 1996; accepted November 1996