Complete Computer System Simulation: The SimOS Approach

Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta
Stanford University

Designed for efficient and accurate study of both uniprocessor and multiprocessor systems, SimOS simulates computer hardware in enough detail to run an entire operating system and provides substantial flexibility in the tradeoff between simulation speed and detail.

The complexity of modern computer systems and the diverse workloads they must support challenge researchers and designers who must understand a system's behavior. As computer system complexity has increased, software simulation has become the dominant method of system testing, evaluating, and prototyping. Simulation is used at almost every step of building a computer system: from evaluation of research ideas, to verification of the hardware design, to performance tuning once the system has been built. In all these simulations, designers face trade-offs between speed and accuracy. Frequently, they must reduce accuracy to make the simulation run in an acceptable amount of time.

One simplification that reduces simulation time is to model only user-level code and not the machine's privileged operating system code. Omitting the operating system substantially reduces the work required for simulation. Unfortunately, removing the operating system from the simulation model reduces both the accuracy and the applicability of the simulation environment. Important computing environments, such as database management systems and multiprogrammed, time-shared systems, spend as much as a third of their execution time in the operating system.[1,2] Ignoring the operating system in modeling these environments can result in incorrect conclusions. Furthermore, these environments use the operating system services heavily, so it is difficult to study such applications in a simulation environment that does not model an operating system. The inability to run OS-intensive applications means that their

1063-6552/95/$4.00 © 1995 IEEE

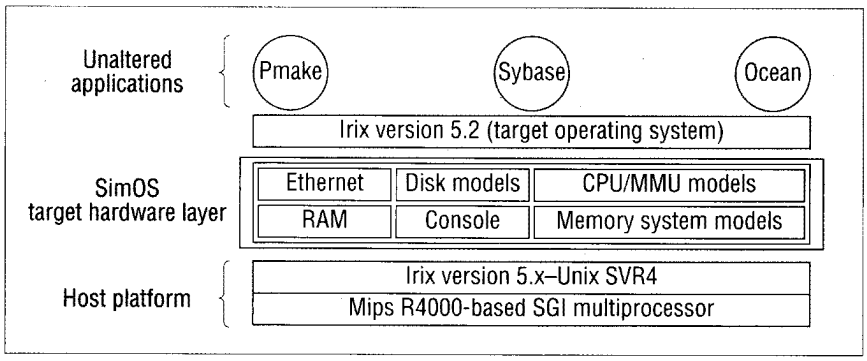

Figure 1. The SimOS environment. SimOS runs as a layer between the host machine and the target operating system. (We use *host* to refer to the hardware and software on which SimOS runs and *target* to refer to the simulated hardware modeled by SimOS and to the operating system and applications executing on that architecture.)

behavior tends to be poorly understood. Finally, operating system researchers and developers cannot use these simulation environments to study and evaluate their handiwork.

SimOS is a simulation environment capable of modeling complete computer systems, including a full operating system and all application programs that run on top of it. Two features help make this possible. First, SimOS provides an extremely fast simulation of system hardware. Workloads running in the SimOS simulation environment can achieve speeds less than a factor of 10 slower than native execution. At this simulation speed, researchers can boot and interactively use the operating system under study. (We use the term *workload* to refer to the execution of one or more applications and their associated OS activity.)

The other feature that enables SimOS to model the full operating system is its ability to control the level of simulation detail. During the execution of a workload, SimOS can switch among a number of hardware component simulators. These simulators vary in the amount of detail they model and the speed at which they run. Using multiple levels of simulation detail, a researcher can focus on the important parts of a workload while skipping over the less interesting parts. The ability to select the right simulator for the job is very useful. For example, most researchers are interested in the computer system's running in steady state rather than its behavior while booting and initializing data structures. We typically use SimOS's high-speed simulators to boot and position the workload and then switch to more detailed levels of simulation. The process is analogous to using the fast forward button on a VCR to position the tape at an interesting section and then examining that section at normal speed or even in slow motion. Additionally, SimOS lets us repeatedly jump into and out of the more detailed levels of simulation. Statistics collected during each of the detailed simulation samples provide a good indication of a workload's behavior, but with simulation times on a par with the quicker, less detailed models.

SimOS allows a system designer to evaluate all the hardware and software performance factors in the context of the actual programs that will run on the machine. Computer architects have used SimOS to study the effects of new processor and memory system organizations on workloads such as large, scientific applications and a commercial database system. OS designers have used SimOS for developing, debugging, and performance-tuning an operating system for a next-generation multiprocessor.

The SimOS environment

To boot and run an operating system, a simulator must provide the hardware services expected by the operating system. A modern multiprocessor operating system, such as Unix SVR4, assumes that its underlying hardware contains one or more CPUs for executing instructions. Each CPU has a memory management unit (MMU) that relocates every virtual address generated by the CPU to a location in physical memory or generates an exception if the reference is not permitted (for example, a page fault). An operating system also assumes the existence of a set of I/O devices including a periodic interrupt timer that interrupts the CPU at regular intervals, a block storage device such as a magnetic disk, and devices such as a console or a network connection for access to the outside world.

SimOS, diagrammed in Figure 1, is a simulation layer that runs on top of general-purpose Unix multiprocessors such as the SGI (Silicon Graphics Inc.) Challenge series. It simulates the hardware of an SGI machine in enough detail to support Irix version 5.2, the standard SGI version of Unix SVR4. Application workloads developed on SGI machines run without modification on the simulated system. SimOS can therefore run the large and complex commercial applications available on the SGI platform. Although the current SimOS implementation simulates the SGI platform, previous versions have supported other operating systems, and the techniques SimOS utilizes are applicable to most general-purpose operating systems.

Each simulated hardware component in the SimOS layer has multiple implementations that vary in speed

Table 1. SimOS's direct-execution mode mapping.

Target machine feature	Implementation on host machine
CPU	Unix process
Physical memory	File #1
Disk storage	File #2
Exceptions	Unix signals
Resident page	Mapping of File #1 into process
I/O device interrupts	Unix signals

and detail. While all implementations are complete enough to run the full workload (operating system and application programs), we designed the implementations to provide various speed/detail levels useful to computer system researchers and builders. The CPUs, MMUs, and memory system are the most critical components in simulation time, so it is these components that have the most varied implementations.

By far the fastest simulator of the CPU, MMU, and memory system of an SGI multiprocessor is an SGI multiprocessor. SimOS provides a direct-execution mode that can be used when the host and target architectures are similar. SimOS exploits the similarity between host and target by directly using the underlying machine's hardware to support the operating system and applications under investigation. Configuring a standard Unix process environment to support operating system execution is tricky but results in extremely fast simulations.

The direct-execution mode often executes an operating system and target applications only two times slower than they would run on the host hardware. Because of its speed, researchers frequently use this mode to boot the operating system and to position complex workloads. Operating system developers also use it for testing and debugging new features. Although the direct-execution mode is fast, it provides little information about the workload's performance or behavior, and thus it is unsuitable for studies requiring accurate hardware modeling. Moreover, this mode requires strong similarities between the simulated architecture and the simulation platform.

For users who require only the simulation accuracy of a simple model of a computer's CPUs and memory system, SimOS provides a binary-translation mode. This mode uses on-the-fly object code translation to dynamically convert target application and operating system code into new code that simulates the original code running on a particular hardware configuration. It provides a notion of simulated time and a breakdown of instructions executed. It operates at a slowdown of under 12 times. It can further provide a simple cache model capable of tracking information about cache contents and hit and miss rates, at a slowdown of less than 35 times. Binary translation is useful for operating system studies as well as simple computer architecture studies.

SimOS also includes a detailed simulator implemented with standard simulation techniques. The simulator runs in a loop, fetching, decoding, and simulating the effects of instructions on the machine's register set, caches, and main memory. The simulator includes a pipeline model that records more detailed performance information at the cost of longer simulation time. Different levels of detail in memory system simulation are available, ranging from simple cache-miss counters to accurate models of multiprocessor cache coherence hardware. SimOS's highly detailed modes have been used for computer architecture studies as well as for performance tuning of critical pieces of an operating system.

Altogether, SimOS provides hardware simulators ranging from very approximate to highly accurate models. Similarly, the slowdowns resulting from execution on these simulators ranges from well under a factor of 10 to well over a factor of 1,000.

Efficient simulation by direct execution

The greatest challenge faced by the SimOS direct-execution mode is that the environment expected by an operating system is different from that experienced by user-level programs. The operating system expects to have access to privileged CPU resources and to an MMU it can configure for mapping virtual addresses to physical addresses. The SimOS direct-execution mode creates a user-level environment that looks enough like "raw" hardware that an operating system can execute on top of it. Table 1 summarizes the mapping of features in SimOS's direct-execution mode.

CPU instruction execution

To achieve the fastest possible CPU simulation speed, the direct-execution mode uses the host processor for the bulk of instruction interpretation. It simulates a CPU by using the process abstraction provided by the host operating system. This strategy constrains the target instruction to be binary-compatible with the host instruction set. The operating system runs within a user-level process, and each CPU in the target architecture is modeled by a different host process. Host operating system activity, such as scheduler preemptions and page faults, is transparent to the process and will not perturb the execution of the simulated hardware. CPU simulation using the process abstraction is fast because most of

the workload will run at the native CPU's speed. On multiprocessor hosts, the target CPU processes can execute in parallel, further increasing simulation speed.

Because operating systems use CPU features unavailable to user-level processes, it is not possible simply to run the unmodified operating system in a user-level process. Two such features provided by most CPUs are the trap architecture and the execution of privileged instructions. Fortunately, most workloads use these features relatively infrequently, and so SimOS can provide them by means of slower simulation techniques.

A CPU's trap architecture allows an operating system to take control of the machine when an exception occurs. An exception interrupts the processor and records information about the cause of the exception in processor-accessible registers. The processor then resumes execution at a special address that contains the code needed to respond to the exception. Common exceptions include page faults, arithmetic overflow, address errors, and device interrupts. To simulate a trap architecture, the process representing a SimOS CPU must be notified when an exceptional event occurs. Fortunately, most modern operating systems have some mechanism for notifying user-level processes that an exceptional event occurred during execution. SimOS uses this process notification mechanism to simulate the target machine's trap architecture.

Altogether, SimOS provides hardware simulators ranging from very approximate to highly accurate models.

In Unix, user-level processes are notified of exceptional events via the signal mechanism. Unix signals are similar to hardware exceptions in that the host OS interrupts execution of the user process, saves the processor state, and provides information about the cause of the signal. If the user-level process registers a function (known as a signal handler) with the host OS, the host OS will restart the process at the signal handler function, passing it the saved processor state. SimOS's direct-execution mode registers signal handlers for each exceptional event that can occur. The SimOS signal handlers responsible for trap simulation convert the information provided by the host OS into input for the target OS. For example, upon receiving a floating-point exception signal, the invoked SimOS signal handler converts the signal information into the form expected by the target OS's trap handlers and transfers control to the target OS's floating-point-exception-handling code.

In addition to the trap architecture, most CPUs provide privileged instructions that the operating system uses to manipulate a special state in the machine. This special state includes the currently enabled interrupt level and the state of virtual-memory mappings. The privileged instructions that manipulate this state cannot be simulated by directly executing them in a user-level process; at user-level these instructions cause an illegal instruction exception. The host OS notifies the CPU process of this exception by sending it a signal. The direct-execution mode uses such signals to detect privileged instructions, which it then interprets in software. SimOS contains a simple software CPU simulator capable of simulating all the privileged instructions of the CPU's instruction set. This software is also responsible for maintaining privileged registers such as the processor's interrupt mask and the MMU registers.

MMU SIMULATION

A process is an obvious way to simulate a CPU's instruction interpretation, but an analog for the memory management unit is not so obvious because a user-level process's view of memory is very different from the view assumed by an operating system. An operating system believes that it is in complete control of the machine's physical memory and that it can establish arbitrary mappings of virtual address ranges to physical memory pages. In contrast, user-level processes deal only in virtual addresses. To execute correctly, the target operating system must be able to control the virtual-to-physical address mappings for itself and for the private address spaces of the target user processes.

The MMU also presents a special simulation challenge because it is used constantly—by each instruction fetch and each data reference. As Figure 2 (next page) shows, we use a single file to represent the physical memory of the target machine. For each valid translation between virtual and physical memory, we use the host operating system to map a page-size chunk of this file into the address space of the CPU-simulating process. The target operating system's requests for virtual memory mappings appear as privileged instructions, which the direct-execution mode detects and simulates by calling the host system's file-mapping routines. These calls map or unmap page-size chunks of the physical memory file into or out of the simulated CPU's address space. If a target application instruction accesses

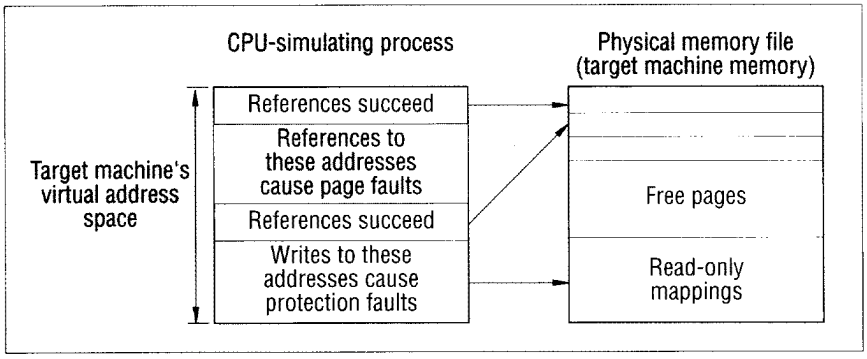

Figure 2. MMU simulation. SimOS simulates a processor's MMU by mapping page-size chunks of a file representing physical memory into a target application's address space. References to unmapped portions of the user's address space are converted to page faults for the target OS. Similarly, writing to mapped pages without write permission results in protection faults for the target OS.

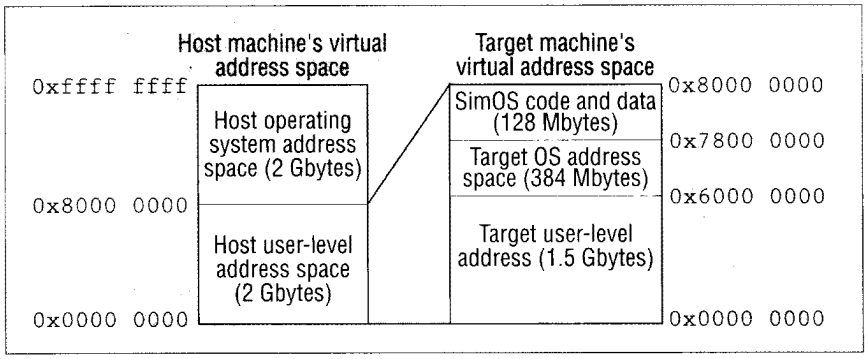

Figure 3. Address space relocation for direct execution. Running the entire workload within an Irix user-level process necessitates compressing the 2 Gbytes of host OS address space into 384 Mbytes. We relink the target OS to reside with SimOS's code and data in the upper portion of the user-level virtual address space. Both SimOS and the target OS reside in each process simulating a CPU for the target architecture.

a page of the application's address space that has no translation entry in the simulated MMU, the instruction will access a page of the CPU simulation process that is unmapped in the host MMU. As discussed earlier, the simulated trap architecture catches the signal generated by this event and converts the access into a page fault for the target operating system.

We simulate the protection provided by an MMU by using the protection capabilities of the file-mapping system calls. For example, mapping a page-size section of the physical memory file without write permission has the effect of installing a read-only translation entry in the MMU. Any target application write attempts to these regions produce signals that are converted into protection faults and sent to the target operating system.

In many architectures the operating system resides outside the user's virtual address space. This causes a problem for the SimOS MMU simulation because the virtual addresses used by the operating system are not normally accessible to the user. We circumvented this problem by relinking the kernel to run at the high end of the user's virtual address space in a range of addresses accessible from user mode. We also placed the SimOS code itself in this address range. Although this mechanism leaves less space available for the target machine's user-level address space, most applications are insensitive to this change. Figure 3 illustrates the layout of SimOS in an Irix address space.

Device simulation

SimOS simulates a large collection of devices supporting the target operating system. These devices include a console, magnetic disks, Ethernet interfaces, periodic interrupt timers, and an interprocessor interrupt controller. SimOS supports interrupts and direct memory access (DMA) from devices, as well as memory-mapped I/O (a method of communicating with devices by using loads and stores to special addresses).

In direct-execution mode, the simulated devices raise interrupts by sending Unix signals to the target CPU processes. As described earlier, SimOS-installed signal handlers convert information from these signals into input for the target operating system. The timer, interprocessor, and disk interrupts are all implemented by this method. We implement DMA by giving the devices access to the physical memory file. By accessing this file, I/O devices can read or write memory to simulate transfers that occur during DMA.

To simulate a disk, SimOS uses a file with content corresponding to that of a real disk. The standard file-system build program converts standard files into raw disk format. SimOS uses this program to generate disks containing files copied from the host system. Building disks from host files gives the target OS access to the large volume of programs and data necessary to boot and run large, complex workloads.

SimOS contains a simulator of an Ethernet local-area network, which allows simulated machines to communicate with each other and the outside world. The implementation of the network interface hardware in

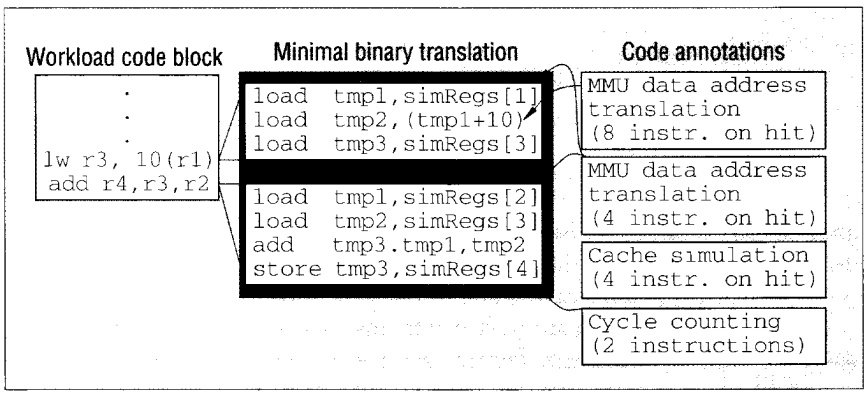

Figure 4. Binary translation mechanics. Binary translation converts an instruction sequence from a block of code in the target machine's instruction set into a code sequence that runs on the host. Above, `simRegs` is an array that holds the state of the target CPU's registers. The translated code uses `simRegs` to perform operations on the target machine's registers and memory. The MMU address translation and cycle-counting annotations support the operating system. Adding a cache simulation entails adding two instructions to the MMU translation. All instruction counts are for the hit case. Misses necessitate calling support functions.

SimOS sends messages to an Ethernet simulator process. Communication with the outside world uses the Ethernet simulator process as a gateway to the local Ethernet. With network connectivity, SimOS users can remotely log in to the simulated machines and transfer files using services such as FTP (file transfer protocol) or NFS (Network File System). For ease of use, we established an Internet subnet for our simulated machines and entered a set of host names into the local name server.

Detailed CPU simulation

Although the SimOS direct-execution mode runs the target operating system and applications quickly, it does not model any aspect of the simulated system's timing and may be inappropriate for many studies. Furthermore, it requires compatibility between the host platform and the architecture under investigation. To support more detailed performance evaluation, SimOS provides a hierarchy of models that simulate the CPU and MMU in software for more accurate modeling of the target machine's CPU and timing. Software-simulated architectures also remove the requirement that the host and target processors be compatible.

CPU simulation via binary translation

The first in SimOS's hierarchy of more detailed simulators is a CPU model that uses binary translation[3] to simulate execution of the target operating system and applications. This software technique allows greater execution control than is possible in direct-execution mode. Rather than executing unaltered workload code as in direct execution, the host processor executes a translation of that code, which is produced at runtime. From a block of application or OS code, the binary translator creates a translation that applies the operations specified by the original code to the state of the simulated architecture. Figure 4 presents an example of this translation and the flexibility it provides.

Many interesting workloads execute large volumes of code, so the translator must be fast. We amortize the time spent on runtime code generation by storing the code block translations in a large translation cache. Upon entering each basic block, SimOS searches the cache to see if a translation of this code already exists. Basic block translations present in the cache are reexecuted without incurring translation costs.

As in the direct-execution mode, each CPU in the target machine is simulated by a separate user process. Using separate processes reduces the accuracy of instruction and memory reference interleaving, but it allows the simulated processors to run concurrently on a multiprocessor host. Since the binary-translation mode's emphasis is execution speed, the efficiency obtained by parallel simulation outweighs the lost accuracy.

The binary-translation CPU's ability to dynamically generate code supports on-the-fly changes of the simulator's level of detail. A faster binary-translation mode instruments its translations only to count the number of instructions executed. A slower binary-translation mode counts memory system events such as cache hits and misses. A highly optimized instruction sequence emitted by the translator performs cache hit checks, quickly determining whether a reference hits in the cache; thus, the full cache simulator is invoked only for cache misses (which are infrequent). Adding cache simulation yields a more accurate picture of a workload's performance on the simulated architecture.

CPU simulation via detailed software interpretation

Although the binary-translation CPU is extremely fast and provides enough detail to derive some memory system behavior, it is not sufficient for detailed multiprocessor studies. Because each simulated CPU executes as a separate Unix process on the host machine, there is no fine-grained control over the interleaving of memory references from the multiple simulated CPUs. To

address this deficiency, SimOS includes two more-detailed CPU simulators that provide complete control over the instruction interleaving of multiple processors.

The first of these simulators interprets instructions using a straightforward fetch, decode, and execute loop. Because we want precise, cycle-by-cycle interleavings of all CPUs, we simulate them all in a single Unix process. Precise cycle interleavings allow device interrupts to occur with precise timing and allow more accurate cache simulation. The additional accuracy of this mode and its current inability to exploit parallelism on a multiprocessor host result in slowdowns more than an order of magnitude larger than the binary-translation CPU.

As its second more-detailed CPU model, SimOS includes a dynamically scheduled CPU similar to many next-generation processors such as the Intel P6, the Mips R10000, and the AMD K5. This model incorporates highly aggressive processor design techniques including multiple instruction issue, out-of-order execution, and hardware branch prediction. The CPU simulator is completely parameterizable and accurately models the pipeline behavior of the advanced processors. This accuracy, combined with the model's single-process structure, results in extremely time-consuming simulation. We use this CPU model to study the effect of aggressive processor designs on the performance of both the operating system and the applications it supports.

Switching simulators and sampling

The hierarchy of CPU simulators in SimOS makes accurate studies of complex workloads feasible. The slowdowns at the most detailed level make running entire complex workloads for a reasonable amount of time far too expensive, so we exploit SimOS's ability to switch modes. We control switching by specifying that a given program be run in a more detailed mode or that we want to sample a workload. Sampling a workload consists of executing it in one CPU simulator for a given number of simulated cycles and then switching execution to another simulator. By toggling simulators, we can obtain most of the information of the more detailed mode at a performance near that of the less detailed mode. Because the different modes share much of the simulated machine's state, the time required to switch levels of detail is negligible.

Sampling is useful for understanding workload execution because application programs' execution phases usually display different behavior. For instance, program initialization typically involves file access and data movement, while the main part of the program computation phase may stress the CPU. These phases may be even more dissimilar for multiprocessor workloads, in which the initialization phase may be single-threaded, while the parallel, computation phase may alternate between heavy computation and extensive communication. Thus, capturing an accurate picture of the workload by examining only one portion is not possible. Sampling enables a detailed simulator to examine evenly distributed time slices of an entire workload, allowing accurate workload measurement without the detailed simulator's large slowdowns.

Switching modes is not only useful for sampling between a pair of modes. It is also useful for positioning workloads for study. For example, we usually boot the operating system under the direct-execution mode and then switch into the binary-translation mode to build the state of the system's caches. Once the caches have been "warmed up" with referenced code and data, we can begin more detailed examination of the workload. We switch to the more detailed CPU model for an accurate examination of the workload's cache behavior. We present specific performance numbers for the various simulation levels later in the article.

Memory system simulation

The growing gap between processor speed and memory speed means that the memory system has become a large performance factor in modern computer systems. Recent studies have found that 30 to 50 percent of some multiprogrammed workloads' execution time is spent waiting for the memory system rather than executing instructions. Clearly, an accurate computer system simulation must include a model of these significant delays.

To hide the long latency of memory from the CPU, modern computer systems incorporate one or more levels of high-speed cache memories to hold recently accessed memory blocks. Modeling memory system stall time requires simulating these caches to determine which memory references hit in the cache and which require additional latency to access main memory. Since processor caches are frequently controlled by the same hardware module that implements the CPU, SimOS incorporates the caches in the CPU model. This allows the different CPU simulator implementations to model caches at an appropriate level of accuracy.

The direct-execution mode does not model a memory system, so it does not include a cache model. The binary-translation mode can model a single level of cache for its memory system. This modeling includes

keeping multiple caches coherent in multiprocessor simulation and ensuring that DMA requests for I/O devices interact properly with caches. Finally, the detailed CPU simulators include a multilevel cache model that we can parameterize to model caches with different organization and timing. It can provide an accurate model of most of today's computer system caches. As in the CPU models, each level of cache detail increases simulation accuracy as well as execution time.

SimOS also provides multiple levels of detail in modeling the latency of memory references that miss in the caches. In the fastest and simplest of these models, all cache misses experience the same delay. More complex models include modeling contention due to memory banks that can service only one request at a time. This model makes it possible to accurately model the latencies of most modern, bus-based multiprocessors.

Finally, SimOS contains a memory system simulator that models the directory-based cache-coherence system used in multiprocessors with distributed shared memory. These machines, such as Stanford's DASH multiprocessor, have nonuniform memory access (NUMA) times due to the memory distribution. Each processor can access local memory more quickly than remote memory. The simulator also models the increased latency caused by memory requests that require coherency-maintaining activity. This simulator has been useful for examining the effects of the NUMA architecture in modern operating systems.

Although we have presented our CPU simulator hierarchy independently from our memory system hierarchy, there are correlations. Certain CPU simulations require certain memory system simulation support. For example, the dynamically scheduled CPU simulator requires a nonblocking cache model to exploit its out-of-order execution. Furthermore, without a memory system that models contention, the timings reported by the processor will be overly optimistic. Likewise, simpler CPU simulation models are best coupled with simpler, faster memory system models.

SimOS performance

The two primary criteria for evaluating a simulation environment are what information it can obtain and how long it takes to obtain this information. Table 2 (next page) compares the simulation speeds of several of the SimOS simulation modes with each other and with execution on the native machine. We used the following workloads in the comparison:

- *SPEC benchmarks*: To evaluate uniprocessor speed, we ran three programs selected from the SPEC92 benchmark suite.[4] The performance of these applications has been widely studied, providing a convenient reference point for comparisons to other simulation systems.
- *Multiprogram mix*: This workload is typical of a multiprocessor used as a compute server. It consists of two copies of a parallel program (raytrace) from the SPLASH benchmark suite[5] and one of the SPEC benchmarks (eqntott). The mix of parallel and sequential applications is typical of current multiprocessor use. The operating system starts and stops the programs as well as time-sharing the machine among the multiple programs.
- *Pmake*: This workload represents a multiprocessor used in a program development environment. It consists of two independent program compilations taken from the compile stage of the Modified Andrew Benchmark.[6] Each program consists of multiple files compiled in parallel on the four processors of the machine, using the SGI pmake utility. This type of workload contains many small, short-lived processes that make heavy use of OS services.
- *Database*: This workload represents the use of a multiprocessor as a database server, such as might be found in a bank. We ran a Sybase database server supporting a transaction-processing workload, modeled after TPC-B.[7] The workload contains four processes that comprise the parallel database server plus 20 client programs that submit requests to the database. This workload is particularly stressful on the operating system's virtual memory subsystem and on its interprocess communication code. It also demonstrates a strength of the SimOS environment: the ability to run large, commercial workloads.

We executed the simulations on a SGI Challenge multiprocessor equipped with four 150-MHz R4400 CPUs. We configured the different simulators to behave like this host platform and ran the workloads on top of them. The native execution numbers represent the wall-clock time necessary to execute each workload directly on the Challenge machine. The other numbers indicate how much slower the workload ran under simulation. We computed the slowdowns by dividing the simulation wall-clock times by the native execution times. The detailed simulations execute the multiprocessor simulations in a single process.

The trade-off between CPU speed and the level of

Table 2. SimOS performance.

Workload	Native execution wall-clock time (sec.)	Direct execution	Binary translation without memory contention modeling: Without caches	Binary translation without memory contention modeling: With L2 cache	Detailed CPU with memory contention modeling: L1 and L2 caches	Detailed CPU with memory contention modeling: Dynamically scheduled
Uniprocessor (1 processor, 16-Mbyte RAM)						
023.eqntott	20	1.9×	5.2×	9.2×	229×	~5,700×
052.alvinn	97	1.1×	5.4×	9.8×	180×	~3,900×
008.espresso	36	1.6×	8.8×	11.2×	232×	~6,400×
Multiprocessor (4 processors, 128-Mbyte RAM)						
Multiprogram mix	36	4.1×	4.5×	10.1×	502×	~25,000×
Pmake	15	36×	12.1×	26.5×	1134×	~52,000×
Database	13	145×	10.5×	36.2×	849×	~27,000×

modeling detail is readily apparent in Table 2. For the uniprocessor workloads, the highly detailed simulations are more than 100 times slower than the less detailed direct-execution mode and around 200 times slower than the native machine. The binary-translation mode's moderate accuracy level (instruction counting and simple cache simulation) results in moderate slowdowns of around 5 to 10 times the native machine.

The trade-off between accuracy and speed becomes even more pronounced for multiprocessor runs. For the relatively simple case of running several applications in the multiprogram mix, the accurate simulations take 500 times longer than the native machine. Since the detailed CPU model does not exploit the underlying machine's parallelism to speed the simulation, its slowdown scales linearly with the number of CPUs being simulated. This causes a slowdown factor in the thousands when simulating machines with 16 or 32 processors.

The complex nature of the other two multiprocessor workloads, along with heavy use of the operating system, causes all the simulators to run slower. Frequent transitions between kernel and user space, frequent context switches, and poor MMU characteristics degrade the direct-execution mode's performance until it is worse than the binary-translation model. These troublesome characteristics cannot be handled directly on the host system and constantly invoke the slower software layers of the direct-execution mode.

The pmake and database workloads cause large slowdowns on the simulators that model caches because the workloads have a much higher cache miss rate than is present in the uniprocessor SPEC benchmarks. Cache simulation for the binary-translation CPU is slower in the multiprocessor case than in the uniprocessor case due to the communication overhead of keeping the multiprocessor caches coherent. Similar complexities in multiprocessor cache simulation add to the execution time of the detailed CPU modes.

Experiences with SimOS

SimOS development began in spring 1992 with the simulation of the Sprite network operating system, running on Sparc-based machines. We started development of the Mips-based SimOS described in this article in fall 1993 and have been using it since early 1994. A simulation environment is only as good as the results it produces, and SimOS has proven to be extremely useful in our research. Recent studies in three areas illustrate the SimOS environment's effectiveness:

- *Architectural evaluation*: SimOS is playing a large part in the design of Stanford's Flash, a large-scale NUMA multiprocessor.[8] Researchers have used SimOS's detailed CPU modes to examine the performance impact of several design decisions. The ability to boot and run realistic workloads such as the Sybase database and to switch to highly accurate machine simulation have been of great value.
- *System software development*: In the design of an operating system for Flash, SimOS's direct-execution and binary-translation modes provide a development and debugging environment. Because SimOS supports full source-level debugging, it is a significantly better debugging environment than the raw hardware. Developers use the detailed CPU models to examine and measure time-critical parts of the software. In addition, SimOS provides the OS development group with Flash "hardware" long before the machine is to be complete.
- *Workload characterization*: SimOS's ability to run complex, realistic workloads including commercial applications enables researchers to characterize workloads that have not been widely studied before. Examples include parallel compilations and the Sybase database.[9]

Based on our experience with SimOS, we believe that two of its features will become requirements for future simulation environments. These key features are the ability to model complex workloads, including all operating system activity, and the ability to dynamically adjust the level of simulation detail. Modern computer applications, such as database-transaction-processing systems, spend a significant amount of execution time in the operating system. Any evaluation of these workloads or the architectures on which they run must include all operating system effects.

Simulation of multiple levels of detail that can be adjusted on the fly allows rapid exploration of long-running workloads. A fast simulation mode that allows positioning of long-running workloads is essential for performance studies. Furthermore, as system complexity increases, accurate simulators will be too slow to run entire workloads. Sampling between fast simulators and detailed simulators will be the best way to understand complex workload behavior.

Acknowledgments

We thank John Chapin, Edouard Bugnion, and the magazine referees for useful feedback on early drafts of this article. We also thank Ben Verghese for his help with the Sybase workload and Jim Bennett for creating the SimOS dynamic processor model. Stephen Herrod receives the support of a National Science Foundation graduate research fellowship. Mendel Rosenblum and Anoop Gupta receive partial support from NSF Young Investigator awards. This work was supported in part by DARPA grant DABT63-94-C-0054.

References

1. J. Chapin et al., "UNIX Performance on CC-NUMA Multiprocessors," *Proc. 1995 ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1995, pp. 1-13.

2. J.B. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Operating Systems Review*, Vol. 27, No. 5, Dec. 1993, pp. 120-133.

3. R. Cmelik and D. Keppel, "Shade: A Fast Instruction Set Simulator for Execution Profiling," *Performance Evaluation Review*, Vol. 22, No. 1, May 1994, pp. 128-137.

4. *SPEC Newsletter*, Vol. 3, No. 4, Dec. 1991, pp. 18-21.

5. J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Computer Architecture News*, Vol. 20, No. 1, Mar. 1992, pp. 5-44.

6. J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. Summer 1990 Usenix Conf.*, Usenix Assn., Berkeley, Calif., June 1990, pp. 247-256.

7. J. Gray, ed. *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann, San Mateo, Calif., 1991.

8. M. Heinrich et al., "The Performance Impact of Flexibility in the Stanford Flash Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 274-284.

9. M. Rosenblum et al., "The Impact of Architectural Trends on Operating System Performance," *Proc. 15th ACM Symp. Operating System Principles*, ACM Press, New York,

Mendel Rosenblum is an assistant professor in the Computer Science Department at Stanford University. His research interests include system software, computer architecture, and simulation environments. He received a BA in math from the University of Virginia (1984) and a MS (1989) and PhD (1992) in computer science from the University of California at Berkeley. He is a 1992 recipient of the National Science Foundation's National Young Investigator award and a 1994 recipient of a Alfred P. Sloan Foundation Research Fellowship. He was a co-winner of the 1992 ACM Doctoral Dissertation Award. He is a member of ACM and IEEE Computer Society. He can be reached at the Computer Science Department, Stanford University, Stanford, CA 94305; mendel@cs.Stanford.edu.

Stephen Herrod is a PhD candidate in computer science at Stanford University. His research interests include simulation technology, multiprocessor architectures, and multiprocessor application performance debugging. He received a BA in computer science from the University of Texas at Austin in 1992 and an MS in computer science from Stanford University in 1995. He is currently supported by a National Science Foundation graduate fellowship and is continually improving the SimOS environment. He can be reached at the Computer Science Department, Stanford University, Stanford, CA 94305; herrod@cs.stanford.edu.

Emmett Witchel is currently a PhD student at the Massachusetts Institute of Technology where he is feeding an addiction to systems programming.He obtained his BS in computer systems engineering and BA in philosophy from Stanford University in 1992, and his MS in computer science from Stanford University in 1994. He then spent a year at the Computer Systems Laboratory at Stanford developing fast machine simulation technology using dynamic code generation. His research interests include operating systems, dynamic code generation, parallel architectures and Yoga. He can be reached at 545 Technology Square Room 521b, Cambridge MA, 02139; witchel@cs.stanford.edu

Anoop Gupta is an associate professor of computer science and electrical engineering at Stanford University. Prior to joining Stanford, he was on the research faculty of Carnegie Mellon University, where he received his PhD in 1986. His research interests include hardware, systems software, and applications for scalable parallel computer systems. Along with John Hennessy, he co-led the design and construction of the Stanford DASH multiprocessor, and is currently working on the next generation Flash machine. He is on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems* and *Journal of Parallel and Distributed Computing*. He can be reached at the Computer Science Department, Stanford University, Stanford, CA 94305; gupta@cs.stanford.edu.

Additional information about SimOS is available at http://www.flash.stanford.edu/SimOS/