ACCEPTED FOR PUBLICATION IN *IEEE Parallel and Distributed Technology* - This is not the final version.

Complete Computer System Simulation: The SimOS Approach

Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta
Computer Systems Laboratory
Stanford University
{mendel, herrod, witchel, gupta}@cs.stanford.edu

Abstract:

The complexity of modern computer systems, coupled with the diverse workloads that they must support, presents a challenge to researchers and designers who need to understand a system's behavior. We describe SimOS, a machine simulation environment designed for the efficient and accurate study of both uniprocessor and multiprocessor computer systems. SimOS simulates computer hardware in enough detail to run an entire operating system. By running a commercial operating system, SimOS provides the ability to investigate realistic workloads, which was not possible with previous simulation tools. SimOS also provides substantial flexibility in the trade-off between the speed and detail of a simulation. We employ fast simulation techniques to scan over the less interesting, time-consuming parts of a workload. To focus in on interesting sections of a workload's execution, we employ slower, more detailed levels of simulation. SimOS' ability to change levels of detail on-the-fly enhances its ability to study complex workloads, allowing the investigator to pay the cost of detailed simulation only when the resulting data is desired.

Keywords: Machine Simulation, Performance Evaluation, Multiprocessing Systems, Virtual Machines

1. Introduction

As the complexity of computer systems has increased, software simulation has become the dominant method of testing, evaluating, and prototyping them. Simulation is used at almost every step when building a computer system; from the evaluation of research ideas, to the verification of the hardware design, to performance tuning once the system has been built. One common trade-off faced by all these simulations is between the speed of the simulation and its accuracy. Frequently the accuracy of simulations must be reduced in order to make the simulation run in an acceptable amount of time.

For computer system simulations, one simplification used to reduce simulation time is to model only user-level code and not the privileged operating system code of the machine. 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 when modeling these environments can result in incorrect conclusions. Furthermore, these environments use the operating system services heavily which can make it difficult to study such applications in a simulation environment that does not model an operating system. The lack of ability to run this type of application means that the behavior of operating system intensive applications tend to be less well understood. Finally, operating systems researchers and developers are unable to 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 of SimOS help make this possible. First, SimOS provides an extremely fast simulation of the hardware of modern systems. Workloads[1] running in the SimOS simulation environment can achieve speeds that are less than a factor of 10 slower than native execution. At this simulation speed, we can boot and interactively use the operating system under study.

One feature which 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 between a number of hardware component simulators. These simulators vary in the amount of detail that they model and in 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 parts of the workload that are of less interest. The ability to select the right simulator for the job is very useful. For example, most researchers are interested in the computer system running in “steady state” rather than its behavior while booting and initializing data structures. We typically use SimOS’ 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 provides the ability to repeatedly jump into and out of the more detailed levels of simulation. Collecting statistics during each of the detailed simulator’s “samples” provides a good indication of a workload’s behavior, but with simulation times that are on par with the quicker, less detailed models.

1. In this paper we use the term “workload” to refer to the execution of one or more applications and their associated operating system activity.

SimOS allows a computer system designer to evaluate all of the hardware and software factors that contribute to the performance of the computer in the context of the actual programs that will be run on the machine. SimOS has been used by computer architects to study the effects of new processor and memory system organizations on appropriate workloads. These workloads include large, scientific applications as well as a commercial database system. SimOS has also been used by operating system designers developing, debugging, and performance tuning an operating system for a next-generation multiprocessor.

2. The SimOS Environment

In order 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 (e.g. a page fault). An operating system also assumes the existence of a set of I/O devices that include a periodic interrupt timer that interrupts the CPU on regular intervals, a block storage device such as a magnetic disk, and access to the outside world through devices such as a console or a network connection.

SimOS is a simulation layer that runs on top of general-purpose Unix multiprocessors such as the Silicon Graphics Inc. (SGI) Challenge series (see Figure 3.2). It simulates the hardware of a 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. While the current implementation of SimOS simulates the SGI platform, previous versions have supported other operating systems, and the techniques utilized are applicable to most general-purpose operating systems.

Each simulated hardware component in the SimOS layer has multiple implementations that vary in speed and detail. While all implementations are complete enough to run the full workload (OS and application programs), the

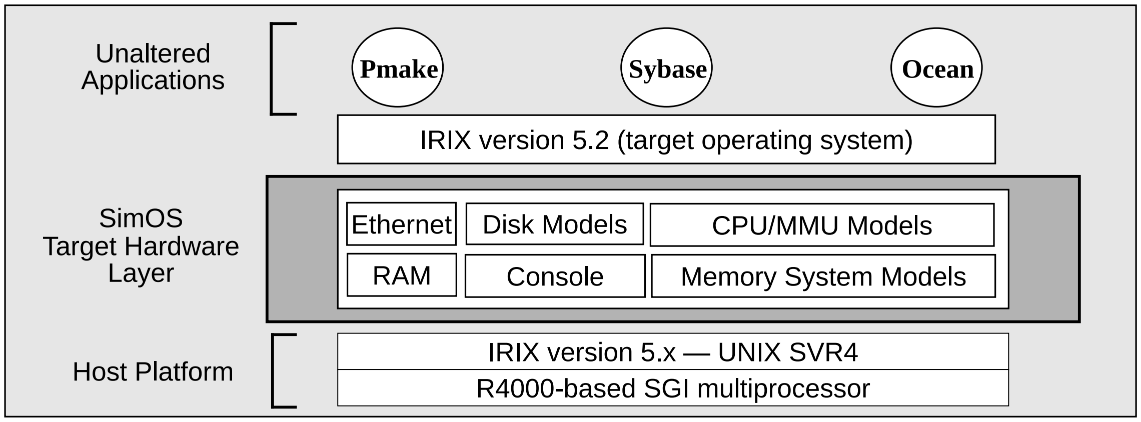
Unaltered Applications
Pmake
Sybase
Ocean
IRIX version 5.2 (target operating system)
SimOS Target Hardware Layer
Ethernet
Disk Models
CPU/MMU Models
RAM
Console
Memory System Models
Host Platform
IRIX version 5.x — UNIX SVR4
R4000-based SGI multiprocessor

FIGURE 2.1. The SimOS Environment. The SimOS environment runs as a layer between the host machine and the target operating system. SimOS provides a variety of CPU and memory system simulators which implement different levels of hardware detail, and therefore run at different speeds. Throughout this paper we use the term *host* to refer to the hardware and software that SimOS runs on, and *target* to refer to the simulated hardware modelled by SimOS and the operating system and applications executing on that architecture.

implementations were chosen to provide speed/detail levels that are of use to computer system researchers and builders. The simulation of 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 simulation of the CPU, MMU, and memory system of an SGI multiprocessor is a SGI multiprocessor. SimOS provides a *direct-execution mode* which 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 that are 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 native on the target hardware. Because of its speed, this mode is frequently used for booting the operating system and for positioning complex workloads. It is also used by operating system developers for testing and debugging new features. Although direct-execution mode is fast, it provides little information about the performance or behavior of the workload, making it unsuitable for studies requiring accurate hardware modelling. Additionally, this mode requires strong similarities between the architecture being simulated and the platform upon which the simulation is performed.

For users of SimOS that only require 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 what instructions were 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. This simulator runs in a loop fetching, decoding, and simulating the effects of instructions on the register set, caches, and the main memory of the machine. This simulator includes a fancy pipeline model which records more detailed performance information at the cost of longer simulation time. Different levels of detail in its memory-system simulation are available. These memory-systems range from simple cache-miss counters to accurate models of multiprocessor cache-coherence hardware. The highly detailed modes of SimOS have been used for computer architecture studies as well as in performance tuning of critical pieces of an operating system.

Altogether, SimOS provides a wide range of hardware simulators ranging from very approximate to highly accurate models. Similarly, the slowdowns resulting from execution on these simulators ranges from well under ten to well over a factor of 1000.

3. Efficient Simulation Using Direct Execution

The largest challenge faced by the SimOS direct-execution mode is that the environment expected by an operating system is different from the environment experienced by user-level programs. The operating system expects to have access to privileged CPU resources and to have access to an MMU that it can configure to map 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.

3.1 CPU Instruction Execution

To achieve the fastest possible CPU simulation speed, direct-execution mode uses the host processor for the bulk of instruction interpretation. Direct-execution mode simulates a CPU 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 is run within a user level process, and each CPU in the target architecture is modelled 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 speed of the native CPU. On multiprocessor hosts, the target CPU processes can execute in parallel, further increasing the simulation speed.

Because operating systems use CPU features that are not available to user-level processes, it is not possible to simply run the unmodified OS in a user-level process. Two features provided by most CPUs that are inaccessible to a user level process are the trap architecture and the execution of privileged instructions. Fortunately, most workloads use these features relatively infrequently, and so they can be provided using slower simulation techniques.

The trap architecture of a CPU allows an operating system to take control of the machine when an exception occurs. An exception interrupts the processor, and records some information about the cause of the exception in processor accessible registers. The processor then resumes execution at a special address which 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 which represents a SimOS CPU needs to 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 trap architecture of the target machine.

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 the execution of the user process, saves the processor state, and provides some 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' direct-execution mode registers signal handlers for each of the exceptional events that can occur. The SimOS signal handlers that are responsible for trap simulation take the information provided by the host OS and convert that information into input for the target operating system. 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 operating system's trap handlers, and transfers control to the target operating systems' floating point exception handling code.

In addition to the trap architecture, most CPUs provide privileged instructions which the operating system uses to manipulate 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, however the use of these instructions at user-level causes an illegal instruction exception. As explained, the host OS notifies the CPU process of this exception by sending it a signal.

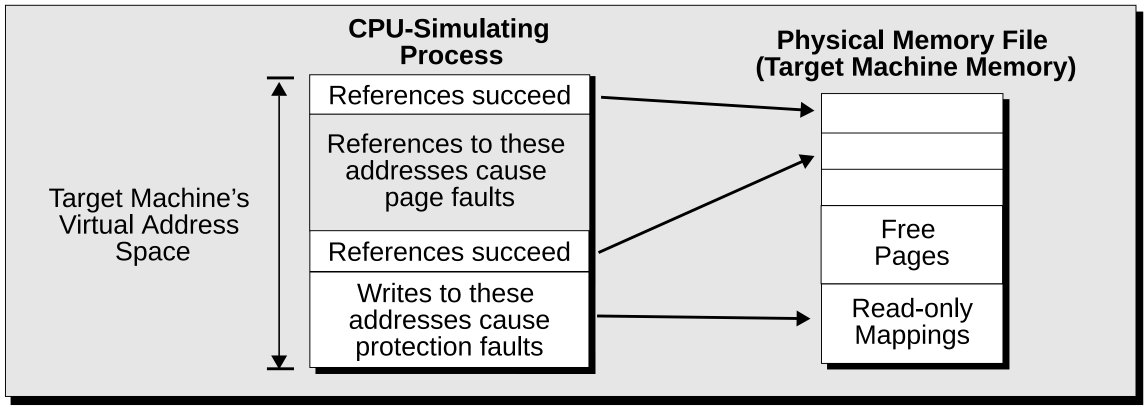
CPU-Simulating Process
Physical Memory File (Target Machine Memory)
Target Machine's Virtual Address Space
References succeed
References to these addresses cause page faults
References succeed
Writes to these addresses cause protection faults
Free Pages
Read-only Mappings

FIGURE 3.1. Memory Management Unit (MMU) Simulation. The simulation of a processor's memory management unit is accomplished by mapping page-sized chunks of a file representing physical memory into a target application's address space. References made to unmapped portions of the user's address space are converted to page faults for the target operating system. Similarly, writing pages that are mapped without write permission results in protection faults for the target operating system.

Direct execution mode uses these 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 CPUs instruction set. This software is also responsible for maintaining privileged registers such as the processor's interrupt mask and the MMU registers described in the next section.

3.2 Memory Management Unit (MMU) Simulation

While a process is an obvious way to simulate the instruction interpretation of a CPU, an analog for the memory management unit is not so obvious. This is due to the fact that a user-level process' 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 physical memory of the machine 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. In order 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 memory management unit also presents a special simulation challenge because it is used constantly–by each instruction fetch and each data reference. As depicted in Figure 3.1, 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 OS to map a page-sized 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 file mapping routines of the host system. These calls map or unmap page-sized chunks of the physical memory file into or out of the simulated CPU's address space. If a target application instruction accesses 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 in Section 3.1, 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 a MMU by using the protection capabilities of the file mapping system calls. For example, mapping a page-sized 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 result in signals which are converted into protection faults and sent to the target operating system.

In many architectures the operating system resides outside of 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 circumvent 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. The SimOS code itself is also placed 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 3.2 illustrates the layout of SimOS in an IRIX address space.

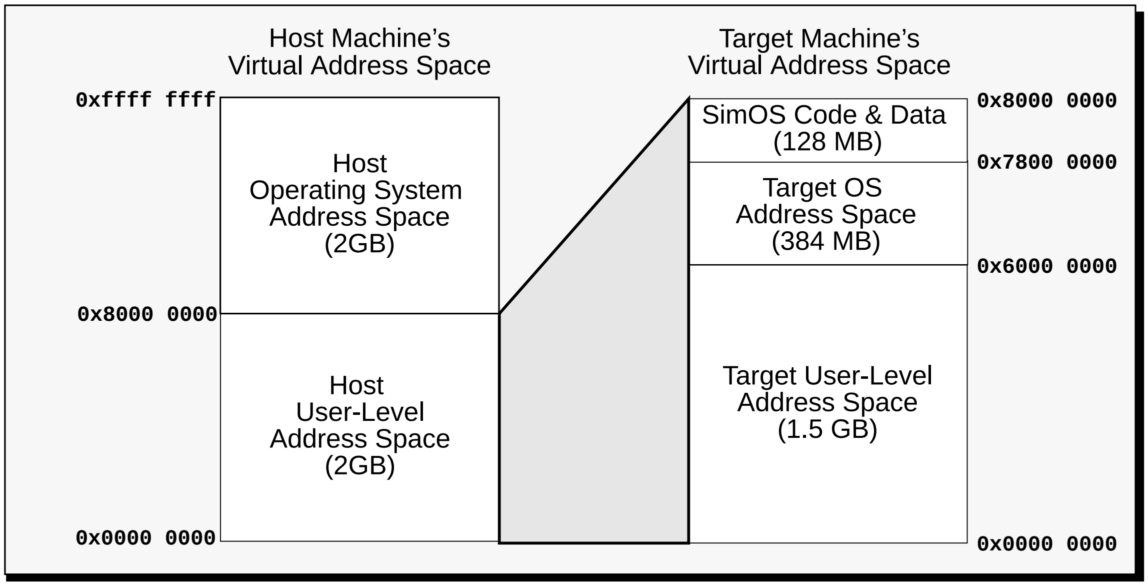
Host Machine's Virtual Address Space
Target Machine's Virtual Address Space
0xffff ffff
0x8000 0000
0x0000 0000
Host Operating System Address Space (2GB)
Host User-Level Address Space (2GB)
SimOS Code & Data (128 MB)
Target OS Address Space (384 MB)
Target User-Level Address Space (1.5 GB)
0x8000 0000
0x7800 0000
0x6000 0000
0x0000 0000

FIGURE 3.2. Address space relocation for direct execution. Running the entire workload within an IRIX user-level process necessitates compressing the 2 gigabytes of host operating system address space into 384 megabytes. We re-link the target operating system to reside with SimOS' code and data in the upper portion of the user-level virtual address space. Both SimOS and the target operating system reside in each process that is simulating a CPU for the target architecture.

3.3 Device Simulation

SimOS simulates a large collection of devices which support the target operating system. These devices include a console, magnetic disks, ethernet interfaces, periodic interrupt timers, and an inter-processor 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 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 in Section 3.1, SimOS-installed signal handlers take information from these signals and convert it into input for the target operating system. The timer, inter-processor, and disk interrupts are all implemented using 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 the transfers that occur during DMA.

We simulate a disk with a file whose contents correspond to that of a real disk. The standard file-system build program takes standard files and converts them into raw disk format. We use this program to generate disks for SimOS which contain 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 (LAN) that allows simulated machines to communicate with each other and with the outside world. The implementation of the network interface hardware in SimOS sends messages to an ethernet simulator process. Communication with the outside world uses the ethernet simulator process as a gateway onto the local ethernet. With network connectivity, SimOS users can remotely login to the simulated machines and transfer files using services such as *ftp* or a *NFS*. For ease of use, we established an internet subnet for our simulated machines, and entered a set of host names was entered into the local name server.

SUGGESTED SIDEBAR

TARGET MACHINE ARCHITECTURAL 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

4. 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 a more detailed level of performance evaluation, SimOS provides other CPU and memory system models. These more detailed models simulate the CPU and MMU in software so that they can more accurately model the CPU and the timing of the target machine. Additionally, software-simulated architectures remove the requirement that the host and target processors be compatible.

4.1 CPU Simulation via Binary Translation

The first level in the hierarchy of more detailed simulators in SimOS is a CPU model that uses binary translation to simulate the execution of the target operating system and applications. This software technique allows for greater control of the execution then is possible in direct-execution mode. Rather than executing unaltered workload code as in direct-execution mode, the host processor executes a translation of that code that is produced at run-time. The binary translator takes a block of application or operating system code and creates a translation which applies the

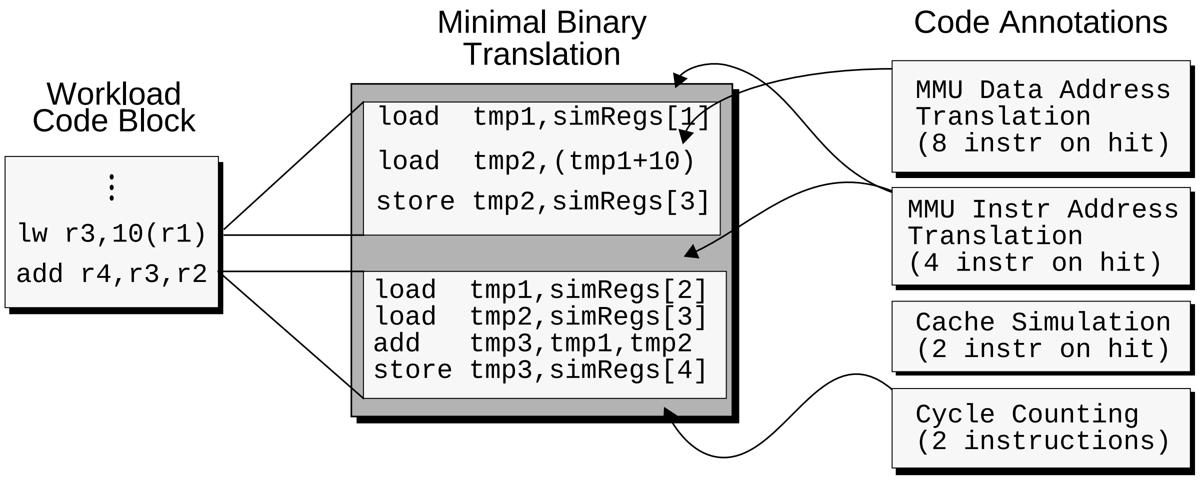

FIGURE 4.1. The mechanics of binary translation. Binary translation converts a sequence of instructions from a block of code in the target machine's instruction set architecture into another sequence of code that runs on the host machine. `simRegs` is an array that holds the state of the target CPU's registers. The translated code uses `simRegs` to perform operations on the target machine's registers and memory. The code annotations for MMU address translation and for cycle counting are needed to support the operating system. Adding a cache simulation entails adding 2 instructions to the MMU translation. All instruction counts are for the hit case. Misses necessitate calling into support functions.

operations specified by the original code to the state of the simulated architecture. An example of this translation and the flexibility it provides is presented in Figure 4.1.

Many interesting workloads execute large volumes of code, so the translator needs to be fast. Time spent on the run-time code generation is amortized by storing the translations of code blocks in a large *translation cache*. Upon entering each basic block, SimOS searches the translation cache to see if the translation of this code already exists. Basic blocks that are present in the translation cache are re-executed without incurring translation costs.

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

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

4.2 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 provides complete control over the instruction interleaving of multiple processors.

This first detailed CPU simulator 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 inability to exploit parallelism on a multiprocessor host result in slowdowns that are more than an order of magnitude larger than the binary translation CPU.

As its second detailed CPU model, SimOS includes a dynamically-scheduled CPU similar to many upcoming next-generation processors such as the Intel P6, MIPS R10000, and AMD K5. This model incorporates highly-aggressive processor design techniques including multiple instruction issue, out-of-order execution, and hardware branch prediction. The dynamically-scheduled CPU simulator is completely paramaterizable and accurately models the pipeline behavior of these advanced processors. This accuracy, coupled 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 that it supports.

4.3 Switching Between Simulators and Sampling

The hierarchy of CPU simulators in SimOS makes accurate studies of complex workloads feasible. The slowdowns of the most detailed level are far too expensive to run entire complex workloads for any reasonable amount of time, so we exploit SimOS’ ability to switch between different CPU modes. We control this switching by specifying that a given program be run in a more detailed mode or by specifying that we want to sample a workload. Sampling a workload consists of executing it in one simulator for a given number of simulated cycles, and then switching execution to another CPU simulator. By toggling simulators, it is possible to obtain most of the information of the more detailed mode at 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 between levels of detail is negligible.

Sampling is useful for understanding workload execution because application programs usually have phases of execution which 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, where the initialization phase may be single threaded while the parallel, computation phase may alternate between heavy computation and extensive communication. Because of the different phases of application behavior, capturing an accurate picture of the workload by examining only a single 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 large slowdowns of the detailed simulator.

Switching modes is not only useful for sampling between a pair of modes. 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 system’s 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. If the detailed pipeline behavior available in our next-generation CPU model is desired, we sample the execution with it to get more information with only a small performance penalty. Specific performance numbers of these simulation levels are presented in Section 5.

4.4 Memory System Simulation

The growing gap between the speed of processors and the speed of memory means that the memory system has become a large factor in the performance of modern computer systems. Recent studies have found that as much as 30% to 50% of some multiprogrammed workloads' execution time is spent waiting for the memory system rather than executing instructions. Clearly an accurate simulation of a computer system must contain 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 that are used to hold recently accessed blocks of memory. 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 as part of the CPU model. This allows the different CPU simulator implementations to model caches at a level of accuracy that makes sense for that simulator.

The direct execution mode does not model a memory system so it does not include any cache model. The binary translation mode is capable of modeling a single level of cache for its memory system. This modeling includes keeping the multiple caches coherent when simulating a multiprocessor and ensuring that DMA requests for I/O devices interact properly with the caches. Finally, the detailed CPU simulators include a multi-level cache model that can be parameterized to model caches with different organization and timings. It can be configured to provide an accurate model of the caches of most modern computer systems. Like the CPU models, each level of detail in the cache increases the accuracy of the simulation as well as its 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 where due to memory banks which can service only one request at a time. Using this model, it is 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 the Stanford DASH multiprocessor, have non-uniform memory access (NUMA) times due to the distribution of memory. Each processor can access local memory more quickly than remote memory. The simulator also models the increased latency that occurs on memory requests that require coherency-maintaining activity. This simulator has been useful for examining the effects of the NUMA architecture of modern operating systems.

While 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 non-blocking 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. Similarly, simpler CPU simulation models are best coupled with simpler, faster memory system models.

5. SimOS Performance

There are two primary criteria for evaluating a simulation environment: what information it can obtain and how long it takes to obtain this information. Table 5.1 compares the simulation speed of several of the SimOS simulation modes with each other and with execution on the native machine. The workloads used in the comparison are:

- *SPEC benchmarks* — To evaluate uniprocessor speed we run three programs selected from the SPEC92 benchmark suite[7]. The performance of these applications has been widely studied and they provide a convenient reference point for performance comparisons to other simulation systems.
- *Multiprogrammed Mix* — This workload is intended to be typical of a multiprocessor used as a compute server. It consists of two copies of a parallel program (raytrace) from the SPLASH benchmark suite[8] and one of the SPEC benchmarks (eqntott). The mix of parallel and sequential applications is typical of current multiprocessor use. The operating system is used to start and stop the programs as well as timeshare the machine between the multiple programs.
- *Pmake* — This workload is intended to be representative of a multiprocessor being used in a program development environment. The workload is comprised of two independent program compilations taken from the compile stage of the Modified Andrew Benchmark[6]. Each program consists of multiple files that are 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 operating system services.
- *Database* — This workload represents the usage of a multiprocessor as a database server, like one that might be found in a bank. We run a Sybase database server supporting a transaction processing workload, modelled after TPC-B.[4] The workload contains four processes that make up 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 points out a strength of the SimOS environment: the ability to run large, commercial workloads.

We execute the simulations on a Silicon Graphics' Challenge multiprocessor equipped with four 150 MHz R4400 CPUs. The native execution numbers present the wall-clock time it takes to execute each workload directly on the Challenge, while the simulation numbers present the wall-clock time required to execute the same workloads at each level of simulation. We divide the simulation wall-clock time by the native execution time to compute the slowdown shown in the table.

The trade-off between CPU speed and the level of detail being modelled is readily apparent in Table 5.1. For the uniprocessor workloads the highly detailed simulations are over a factor of 100 times slower than the less detailed direct-execution mode and around 200 times slower than the native machine. The moderate accuracy level (instruction counting and simple cache simulation) of the binary translation mode 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 a multiprogramming mix, the accurate simulations take 500 times longer than the native machine. Since the detailed CPU model does not exploit the parallelism of the underlying machine 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.

Workload	Native Execution	Direct Execution	Binary Translation without Memory Contention Modelling		Detailed CPU with Memory Contention Modelling	
	wall-clock time		w/ no Caches	w/ L2 Cache	L1 and L2 Caches	Dynamically Scheduled
Uniprocessor (1 processor, 16MB RAM)						
023.eqntott	20 sec	1.9x	5.2x	9.2x	229x	~5,700
052.alvinn	97 sec	1.1x	5.4x	9.8x	180x	~3,900x
008.espresso	36 sec	1.6x	8.8x	11.2x	232x	~6,400x
Multiprocessor (4 processors, 128MB RAM)						
Multiprog Mix	36 sec	4.1x	4.5x	10.1x	502x	~25,000
Pmake	15 sec	36x	12.1x	26.5x	1134x	~52,000x
Database	13 sec	145x	10.5x	36.2x	849x	~27,000x

TABLE 5.1. SimOS Performance Numbers. This table presents performance numbers for the different levels of the SimOS environment. Native execution is the time required to run the workload directly on our host machine. We then configure the different simulators to behave like this host platform and run the same workloads on top of them. The native execution numbers indicate the wall-clock time required to complete each workload. The other numbers indicate how much slower the workload ran under simulation. The detailed simulations do the multiprocessor simulations in a single process.

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, which are present in the database workload, can not be handled directly on the host system, and constantly invoke the slower software layers of direct-execution mode,

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

6. Experiences with SimOS

The development of SimOS began in the Spring of 1992 with the simulation of the Sprite network operating system running on SPARC-based machines. The MIPS-based SimOS described in this report was started in the fall of 1993 and has been in use since early 1994. A simulation environment is only as good as the results that it can produce, and SimOS has proven to be extremely useful in our research. Recent studies in three areas illustrate the effectiveness of the SimOS environment.

- **Architectural evaluation**
  SimOS is being heavily used in the design of the Stanford FLASH multiprocessor; a large-scale NUMA multiprocessor.[5] Using the detailed CPU modes of SimOS, researchers have examined the performance impacts of several design decisions. The ability of SimOS to boot and run realistic workloads such as the Sybase database and switch to highly accurate machine simulation has been of great value.
- **System software development**

SimOS is being used in the design of an operating system for FLASH. SimOS direct-execution and binary-translation modes have been used to provide a development and debugging environment for the operating system work. SimOS supports full source level debugging, making it a significantly better debugging environment than the "raw" hardware. The detailed CPU models are used to examine and measure time critical parts of the software. Finally, SimOS provides the operating system development group with FLASH "hardware" long before the machine will be complete.

- **Workload characterization**

  SimOS's ability to run complex, realistic workloads including commercial applications enables SimOS to be used to characterize workloads that have not been widely studied before. Examples of this include the Sybase database.

7. Concluding Remarks

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 their execution time in the operating system. Any evaluation of these workloads or the architectures upon which they run must include all operating system effects.

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

8. Acknowledgments

We thank John Chapin, Edouard Bugnion, and the journal referees for useful feedback on early drafts of this paper. We also thank Ben Verghese for his help with the Sybase workload and Jim Bennett for creating the SimOS dynamic processor model. Stephen Herrod is supported by a National Science Foundation Graduate Research Fellowship. Mendel Rosenblum and Anoop Gupta are partially supported by National Science Foundation Young Investigator Awards. This work was supported in part by DARPA grant DABT63-94-C-0054.

9. References

[1] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta, "UNIX Performance on CC-NUMA Multiprocessors", In 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, May 1995, pp. 1-13.

[2] J. Bradley Chen and Brian Bershad, "The Impact of Operating System Structure on Memory System Performance", *Operating Systems Review*, vol. 27, no. 5, Dec. 1993, pp 120-133.

[3] Robert Cmelik and David Keppel, "Shade: A Fast Instruction Set Simulator for Execution Profiling", *Performance Evaluation Review*, May 1994, vol. 22, no. 1, pp. 128-137.

[4] Jim Gray, Ed. "The Benchmark Handbook for Database and Transaction Processing Systems", Morgan Kaufmann Publishers, 1991.

[5] Mark Heinrich, Jeff Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Kourosh Gharachorloo, Dave Nakahira, Mark Horowitz, Anoop Gupta, and Mendel Rosenblum., "The Performance Impact of

Flexibility in the Stanford FLASH Multiprocessor”, In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 274-284.

[6] John Ousterhout. “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?” In *Proceedings of the Summer 1990 USENIX Conference*, June 1990, pp. 247-256.

[7] SPEC Newsletter, vol. 3, No 4, Dec 1991, pp. 18-21.

[8] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta, “SPLASH: Stanford Parallel Applications for Shared Memory”, *Computer Architecture News*, vol. 20, num 1, March 1992, pp. 5-44.