Sigmetrics ‘96

Embra: Fast and Flexible Machine Simulation

Emmett Witchel
Laboratory for Computer Science
Massachusetts Institute of Technology
witchel@lcs.mit.edu
http://www.pdos.lcs.mit.edu/~witchel/

Mendel Rosenblum
Computer Systems Laboratory
Stanford University
mendel@cs.stanford.edu
http://www-flash.stanford.edu/SimOS/

Abstract

This paper describes Embra, a simulator for the processors, caches, and memory systems of uniprocessors and cache-coherent multiprocessors. When running as part of the SimOS simulation environment, Embra models the processors of a MIPS R3000/R4000 machine faithfully enough to run a commercial operating system and arbitrary user applications. To achieve high simulation speed, Embra uses dynamic binary translation to generate code sequences which simulate the workload. It is the first machine simulator to use this technique. Embra can simulate real workloads such as multiprocess compiles and the SPEC92 benchmarks running on Silicon Graphic’s IRIX 5.3 at speeds only 3 to 9 times slower than native execution of the workload, making Embra the fastest reported complete machine simulator. Dynamic binary translation also gives Embra the flexibility to dynamically control both the simulation statistics reported and the simulation model accuracy with low performance overheads. For example, Embra can customize its generated code to include a processor cache model which allows it to compute the cache misses and memory stall time of a workload. Customized code generation allows Embra to simulate a machine with caches at slowdowns of only a factor of 7 to 20. Most of the statistics generated at this speed match those produced by a slower reference simulator to within 1%. This paper describes the techniques used by Embra to achieve high performance, focusing on the requirements unique to machine simulation, including modeling the processor, memory management unit, and caches. In order to study Embra’s memory system performance we use the SimOS simulation system to examine Embra itself. We present a detailed breakdown of Embra’s memory system performance for two cache hierarchies to understand Embra’s current performance and to show that Embra’s implementation techniques benefit significantly from the larger cache hierarchies that are becoming available. Embra has been used for operating system development and testing as well as for studies of computer architecture. In this capacity it has simulated large, commercial workloads including IRIX running a relational database system and a CAD system for billions of simulated machine cycles.

1 Introduction

This paper describes Embra, a high speed simulator of the processors, caches, and memory systems of uniprocessors and cache-coherent multiprocessors. Embra models the hardware of these machines in enough detail to boot and run commercial operating systems with arbitrary application workloads. Embra’s high-speed, detailed simulation has allowed us to construct a sophisticated machine simulation environment capable of supporting research on operating systems and computer architecture. Using Embra, we can run large, complex workloads, such as commercial database management systems, in a simulation environment, enabling us to study the workload’s execution and how it interacts with the operating system and computer architecture.

Embra achieves high speed through the aggressive use of on-the-fly or dynamic binary translation. Rather than simulating CPUs by interpreting a workload’s instructions, Embra translates blocks of instructions into code that, when executed, simulates the execution of the original block. This use of binary translation allows Embra to eliminate most of the overhead of instruction interpretation. The result is that Embra can simulate workloads running at up to one fourth the speed of the unsimulated workload, faster than any other complete machine simulator described in the literature.

Binary translation also allows Embra to support a high degree of simulation flexibility without high performance costs. Embra can customize the translations it generates to model specific machine features or to compute specific information about the simulated execution. The translations only include the code needed to perform the tasks specified by the user, so extra features incur no cost when they are not being used. For example, operating system developers can use Embra to test new algorithms with quick turn-around time. Once the code is known to execute correctly, the developer can instruct Embra to model processor caches, producing more accurate performance estimates. Embra’s cache modeling mode enables it to generate workload statistics, most of which match a much slower reference simulator within 1%.

The dynamic nature of Embra’s translations allows the user to change the level of detail in the middle of a simulation run. This allows the user to employ a high speed mode to skip over uninteresting parts of the workload, and switch to a more detailed mode for the sections of interest. This ability to simulate in detail only the interesting parts of a workload is important when studying complex workloads that have long initialization or setup periods before a steady-state is reached.

In this paper, we describe Embra in enough detail to allow other developers to build similar systems. Section 2 describes the SimOS machine simulation environment, of which Embra is a part. SimOS both provides motivation for high speed simulation and places requirements for features Embra must support. Section 3 presents a detailed description of the basic machine simulation techniques used in Embra. This includes the use of on-the-fly binary translation for fast instruction set interpretation and support for fast modeling of memory management hardware for instruction fetches and data accesses. We also describe a set of optimizations we found were necessary to maintain high speed for large, complex workloads and for modeling multiprocessors. Section 4 presents Embra’s technique of customized translations which provide flexibility in what is modeled and reported. The section focuses on translations customized to include modeling of processor caches.

We measure the simulation speed and accuracy of Embra in Section 5. This section also contains a study of Embra’s memory system behavior for two different cache hierarchies. The study allows us to better understand Embra’s current performance and to predict its performance for future cache hierarchies. Section 6 presents related work and Section 7 concludes.

2 Machine simulation with SimOS

Embra is part of SimOS, a simulation environment developed

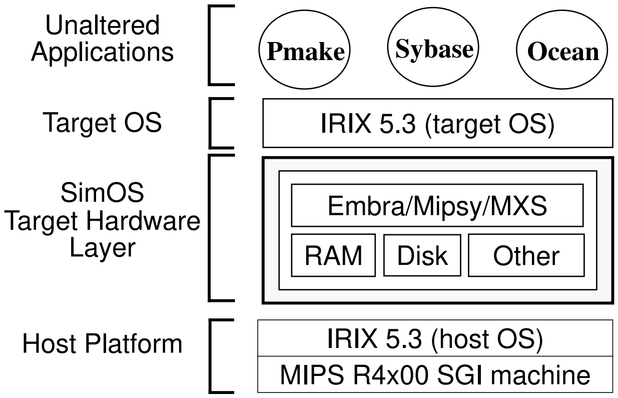

FIGURE 2.1. The SimOS Environment
SimOS runs as a layer on IRIX that can model the hardware of MIPS uniprocessors and shared-memory multiprocessors such that IRIX and any application run on it. Embra is the fastest of several processor models within SimOS.

for the study of operating systems and computer architecture. SimOS, depicted in Figure 2.1, contains simulation models for all of the hardware commonly present on modern computers including processors, memory subsystems, DMA and interrupt controllers, consoles, network interfaces, disks, frame buffers, mice, and keyboards. SimOS models these devices in enough detail to run a commercial Unix operating system complete with all its application programs. A key feature of SimOS is its ability to run large, highly realistic workloads such as commercial database management systems and commercial CAD packages.

SimOS contains a range of compatible simulation models of processors (including Embra), memory systems, and I/O devices that vary greatly in simulation speed and detail. The user of SimOS is able to select the simulator that provides the desired level of simulation detail with the greatest possible speed. For example, a processor pipeline study might use MXS, a SimOS resident CPU simulator capable of accurately modeling next generation microprocessors which feature multiple instruction issue with dynamic scheduling. Like MXS, Embra is a selectable CPU model. Embra models a simple processor pipeline, but it is several orders of magnitude faster than MXS.

Combining Embra's speed with the rest of the SimOS system enables operating system and computer architecture studies that would otherwise be very difficult or impossible. Embra's speed enables it to be used for positioning large, complex workloads for more detailed study. In a recent computer architecture study [Rosenblum95a], Embra was used to boot the operating system and run a commercial database management system until it reached a steady state. This setup took many tens of billions of instructions on the simulated machine. Without a fast simulator such as Embra it would have been extremely time consuming to study such workloads. For example, we used Embra interactively to setup a database system and then used MXS to study how the next generation of processors will run such a large and complex workload. Booting and initializing the operating system and database with MXS would have taken several years.

In addition, Embra has been used in the debugging, development and testing of the Hive multiprocessor operating system [Chapin95]. The Hive developers use Embra to run tests where failures were injected at different points and the behavior of the crash recovery system was observed. The speed of Embra allowed them to run many more tests than would have been possible on the slower simulators in SimOS. Using Embra in a more detailed mode that includes a model of the processor's caches, the Hive developers were also able to study the memory system behavior of their code.

3 Embra implementation

This section describes the techniques used in Embra to achieve its high simulation speed. We start the section with a description of dynamic binary translation, the main technique used by Embra. Dynamic binary translation has previously only been used for user-level application simulation. Adapting this technique for machine simulation required extensions to the basic techniques, as cataloged in Section 3.2. The biggest challenge for this adaptation is modeling the address relocation hardware (i.e. the MMU) of the machine, described in Section 3.3. Section 3.4 describes how Embra handles multiprocessors. Having described the complications of using dynamic binary translation to model a machine, we conclude in Section 3.5 with a discussion of how we maintain the speed found in user-level tools by adapting an important optimization called chaining.

3.1 Dynamic binary translation

The design of Embra was influenced by Shade [Cmelik94], a high speed instruction set simulator that used dynamic binary translation. This subsection presents a brief description of the basic techniques and terminology developed in Shade.

Rather than interpreting program instructions, a dynamic binary translation simulator translates them into code that, when executed, simulates their execution. The original code is translated into code which operates on the simulated state rather than on the real machine state (see Figure 3.1.) A basic translation consists of loading the source registers of an instruction from the simulated register file, simulating the instruction execution, and storing the result (if any) back into the simulated register file. In Embra, basic blocks (code sequences which end with a jump or branch instruction) are the unit of translation.

To avoid having to frequently retranslate blocks of instructions, translated blocks are kept in a *Translation Cache* (TC). The execution of a block of instructions is simulated by locating the block's translation in the TC and jumping to it. A data structure, called the *pc2tc hash table*, maintains the mappings from a program counter to the address of the translated code in the TC.

The main loop of a dynamic binary translation simulator is shown in Figure 3.2. The loop checks to see if the current simulated

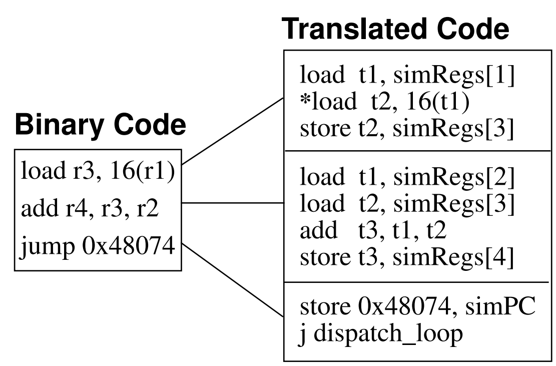

FIGURE 3.1. Instruction set simulation using binary translation
This figure depicts the use of binary translation to simulate instruction execution. Here binary code to be simulated is translated into instruction sequences that perform the equivalent function on the simulated machine state stored in simRegs rather than the machine's registers. The jump instruction is simulated by updating the simulated PC (simPC) and returning to the main dispatch loop. The * indicates where the translation uses a program virtual address. Embra must use an MMU model to relocate these addresses.

program counter address is present in the TC. If it is present in the TC, the translated block is executed. If it is not, the translator is called to add the block to the TC. Each block of translated code ends by loading the new simulated program counter and jumping back to the main loop for dispatching.

Several optimizations of the basic mechanism are possible. Since some blocks always follow each other at run time, they can be *chained*, so the translation for one block simply transfers control to the next rather than returning to the dispatch loop. Chaining improves performance by eliminating many of the lookups in the pc2tc data structure. Better register allocation and usage in the translations is also possible. For example, the reload of simulated register r3 (simReg[3]) in Figure 3.1 is redundant and could be eliminated.

3.2 Embra extensions for full machine simulation

The previous section describes dynamic binary translation as implemented by Shade. Using this technique, Shade was able to simulate the execution of user programs only 2 to 6 times slower than they ran on the machine running Shade. User level simulators such as Shade run a single application program, but they do not simulate the hardware of a machine. Although some of the tasks of user-level application simulation and machine simulation are the same, there are many machine features that are not visible to user programs and hence are not addressed by user level simulators. In order to run full system workloads, Embra needs to model all the features of a machine and not simply those available to user level processes.

Embra features required for complete machine modeling include:

- ***MMU address translation***. Most modern machines contain a memory management unit (MMU) that translates the *virtual addresses* used by the software into *physical addresses* used to access the memory and I/O devices of the machine. The MMU is used by every instruction fetch and every load and store. User applications deal only with virtual addresses and hence user application simulators do not have an MMU model.
- ***Multiple virtual address spaces***. In order to support multi-user operating systems and multi-process applications, modern machines, and hence Embra, support concurrent but disjoint virtual address spaces. In addition to MMU translation, features of the MMU that guarantee protection between processes must also be modeled.
- ***Exceptions and interrupts***. Modern machines contain a trap architecture in which exceptions and externally generated interrupts stop the current flow of execution, and invoke an OS-resident trap handler. For example, Embra detects references to unmapped virtual pages and raises a page fault exception. Embra also promptly detects and simulates the effect of interrupts from I/O devices.
- ***Privileged instructions***. Modern machines contain instructions only usable by the operating system kernel. Embra needs to support instructions for manipulating the MMU, changing the interrupt mask, and other privileged operations.
- ***Miscellaneous operations***. There are several features of the architecture that are either not visible to the user or are rarely used. Since these features are necessary to the functioning of the machine, they must be modelled by a machine simulator. These features include uncached loads and stores to I/O devices, DMA from I/O devices, self-modifying code, and dynamically generated code.

Many machine simulation requirements are straightforward to implement. For example, a privileged instruction, like any other instruction, can be translated into code which simulates its execution. For privileged instructions that have complex semantics, the translation can simply *call out* to a support routine written in C that performs the simulation. This solution is acceptable provided that privileged instructions are rare, so their simulation does not need to be fast. Similarly, instructions that always generate a trap, such as system call and breakpoint instructions, are translated into calls to routines that simulate the proper trap and change the simulated program counter to the correct trap handler.

Unfortunately, some machine features require fundamental modifications to a simulator. Modeling the MMU requires particular attention. User-level simulators can fetch instructions and access data at the virtual address specified by the program. Because the hardware MMU must relocate every instruction fetch and data address generated by the CPU in a real machine, efficient modeling of the MMU is crucial if Embra is to maintain the high speed achieved by user-level dynamic translation simulators. In the following section we present the fast MMU simulation techniques used in Embra.

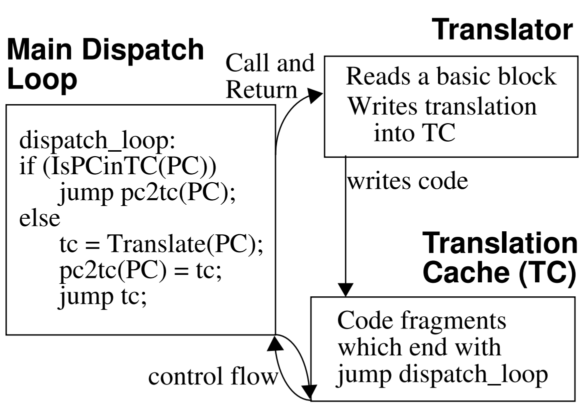

FIGURE 3.2. Main loop of a dynamic translation simulator

The main dispatch loop of a dynamic translation simulator checks to see if the translation for the current simulated PC is present in the translation cache. If the translation is present, it is executed, otherwise the translator is called to generate it. Having translations constantly return to the main dispatch loop is the performance concern addressed by chaining.

3.3 MMU address translation in Embra

Embra models the MMU of the MIPS R3000 microprocessor. The R3000 translates 32 bit virtual addresses to 32 bit physical addresses using a 64-entry, fully associative translation table (TLB). Each TLB entry contains the address of a 4 kilobyte virtual memory page and the corresponding physical memory address of the page along with some protection bits. A lookup must match both the virtual address and the current address space id (ASID) stored in the TLB. The 6-bit ASID allows mappings from up to 64 different virtual address spaces to be present in the TLB without having to flush the TLB on context switches between processes.

Like several other RISC microprocessors, the R3000 has a software reloaded TLB. When the CPU issues a virtual address that is not found in the TLB, the R3000 generates a TLB-miss exception. This exception is normally handled by the operating system which fills in a TLB entry using information from the page table. The software reloaded TLB is visible to the operating system and hence it must be modeled by Embra for the OS to run.

Embra could model the MIPS TLB by inserting a call to a function that models the TLB's fully-associative lookup every time an address translation is needed. However, the cost of function invocation and the cost of the TLB modeling code could easily consume several tens of instructions, making Embra several orders of magnitude slower than the native machine. The frequency of address translations necessitates an efficient solution.

3.3.1 Data access MMU modeling

Every load and store instruction simulated by Embra must first translate the virtual address generated by the instruction into a physical address. This physical address can then be used to index into the array containing the simulated machine's main memory. To make this virtual to physical translation fast and compact enough to be inlined with every translated load and store instruction, Embra maintains an data structure called the *MMU relocation array.* The MMU relocation array is an array indexed by the virtual page number (virtual address divided by page size) of the memory reference. Each entry in the array contains the address of the physical page mapped at that virtual page and the protection bits for the page. Embra keeps this MMU relocation array synchronized with the simulated TLB by applying any changes to the TLB to the MMU relocation array as well. Therefore, only translations with a valid entry in the TLB have a valid entry in the MMU relocation array.

For every load and store instruction translated, Embra adds a sequence of instructions that does the following:

1. Retrieves the TLB information from the MMU relocation array using the virtual page number of the memory address as an index.
2. Performs the TLB permission checks by using the protection bits in the MMU relocation entry. For loads the page must be valid and for stores the page must be valid and writable.
3. Checks for exceptions and if one occurs calls out to a support routine that will simulate the appropriate exception.
4. Combines the physical page address from the MMU relocation array with the page offset bits of the memory address to form the physical address to load or store.

Embra's 8 instruction sequence to implement these four steps takes 8 cycles (assuming cache and TLB hits on the host) on an R4400 pipeline in the common case of a simulated TLB hit. Adding even 8 cycles to load and store instructions is a large performance overhead. Since roughly every third instruction is a load or store, this implies Embra's slowdown is at least a factor of 3 or 4. Unfortunately, this overhead is necessary for full machine simulation.

To keep the MMU simulation instruction sequence compact and fast, Embra does not check the TLB ASID. Embra only puts entries in the MMU relocation array that are valid for the currently executing ASID. This necessitates updating the MMU relocation array on MMU context switches (changes of the ASID). The entries for the old ASID must be removed and the entries for the new ASID must be added. Fortunately, the small size of the MIPS R3000 64-entry TLB puts a low bound on the amount of work this switch requires. Even if all TLB entries have to be replaced only 128 entries in the array need to be modified. Additionally, changes in ASID are infrequent relative to instruction interpretation.

Using the MMU relocation array in the generated code allows it to be independent of the details of the TLB size or organization, so Embra can modeling different size TLBs without changing the code generator. The array does occupy a sizable amount of virtual memory in the simulator. A 32-bit architecture with 4 kilobyte pages requires the MMU relocation array to be 4 megabytes. We discuss the implication of this in Section 5.2.

Modeling different TLB sizes allows an important performance optimization for Embra. By modeling a TLB which is much larger than the R3000's, Embra can reduce the number of TLB exceptions. Since these exceptions are handled in C code, they are computationally expensive, and avoiding them increases the simulation speed. Of course, the TLB miss rates reported with large TLBs do not correlate with those reported by smaller TLB sizes.

3.3.2 Instruction fetch MMU modeling

Instruction fetches are also translated by the MMU. The use of dynamic binary translation simplifies the simulation of the instruction MMU lookups. Unlike the data MMU modeling, the actual computation of the physical address and access to main memory need only be performed when a block of code is translated into the translation cache. Once placed in the translation cache, the actual instructions in the simulated main memory no longer need to be accessed, eliminating the need to translate addresses on every instruction fetch.

However, Embra must still detect attempts to execute from unmapped pages. When such an attempt is made Embra must simulate the proper MMU exception. To detect this condition, Embra starts each translated basic block with a highly optimized sequence of instructions which checks the TLB state of the code. This is done by querying the MMU relocation array with the virtual address of the code block being simulated. If the check determines that the page is not in the TLB, a TLB miss exception is raised.

The instruction sequence used for instruction TLB lookups is similar to the sequence preceding load and store instructions except it need only perform the TLB residence check (no relocation is needed). When Embra is running on the MIPS R4400, this check (assuming hits in the host cache and TLB) takes 3 instructions (5 cycles including pipeline stalls) for uniprocessor modeling and 5 instructions (8 cycles including pipeline stalls) in the common case of a simulated TLB hit.

3.3.3 Support for kernel address translation

In order to run real workloads, Embra needs to handle the special address translation features expected by the operating system kernel. On the MIPS platform these include uncached load and store instructions used to access I/O devices and untranslated access to physical memory using the KSEG0 region. Since uncached operations in MIPS processors differ from cached operations only by bits returned from the TLB lookup, the Embra translator can not tell uncached loads and stores from normal loads and stores. Embra handles these instructions at run time by setting the MMU relocation array entries for uncached pages to an invalid entry. This causes the translated code to attempt to raise a TLB exception. The routine that implements the TLB exception first checks to see if the address is really mapped uncached, and if it is it forwards the uncached access to the appropriate I/O device simulator for handling.

When running in kernel mode, the operating system on the MIPS architecture has access to all of physical memory using a part of the address space called KSEG0. Since KSEG0 is not mapped using the TLB, operating systems frequently put kernel text and data in KSEG0 to avoid TLB misses. Embra models KSEG0 by filling in the MMU relocation array for the KSEG0 virtual address range with the addresses of the corresponding physical memory pages. This allows KSEG0 to be handled without having to special case it in the translated code.

3.3.4 Support for self-modifying code

Self-modifying code is a problem for Embra because once a block of code is cached in the translation cache the original instructions are not re-read when the code is executed. If the code has changed, Embra is in danger of executing the old code from a stale translation. To avoid this, Embra keeps track of the pages that have translated code in the translation cache. When the contents of any of these pages is overwritten, Embra detects the write and flushes the entire translation cache. It would be possible to track which translations should be flushed, but the rarity of this kind of event has not warranted the additional bookkeeping.

3.4 Multiple processors

Embra can model the multiple processors of shared memory multiprocessors in two ways. The first way is to replicate the CPU state and the MMU relocation array for each simulated CPU. These simulated processors are then multiplexed within a single Embra process so they share memory, disks and other devices. The simulated processors are set up in a ring, and each is run for a user configurable timeslice in a round-robin fashion. Long timeslices result in low processor switch overheads but unrealistic execution interleavings. Because Embra keeps the simulated integer register state in memory, it must only save 6 simulator registers on a processor switch. Embra also advances the processor clock, and saves the floating point registers if they are being used. Because this is a small amount of work, short timeslices can be efficient.

3.4.1 Parallel Execution

A second way of simulating multiple processors is to duplicate the entire Embra CPU/MMU mechanism in multiple processes that share the same simulated main memory. This mode of execution, called Parallel Embra, is attractive when the machine running the simulator has at least as many processors as are being simulated. Using Parallel Embra, CPU simulations proceed in parallel so multiprocessor systems can be simulated on multiprocessor hosts without the linear slowdowns typical of multiprocessor machine simulation.

The primary disadvantage of parallel execution is that the random interleaving (due to load, scheduling, etc.) among the Embra processes makes the simulation non-deterministic. Running the simulator twice with the same initial conditions can produce different results. While non-determinism is inherent in parallel execution, the unrealistic interleaving introduced by simulated processors running at different speeds can be controlled by synchronizing the CPU processes using barriers. The frequency of sychronization can be varied to trade performance for accuracy. Parallel Embra is useful for testing and positioning tasks which do not require repeatability or high degrees of accuracy.

3.5 Embra Chaining

Our experience with Embra confirms that chaining, the patching of translated blocks to bypass the translation cache lookup for the current PC, is an important optimization for high performance simulation. The average basic block is small (5.7 instructions for pmake and 6.1 for Sybase), making the main dispatch loop a major overhead. By default, all Embra translations end with a jump back to the dispatch loop, as depicted in Figure 3.1. Embra implements chaining by overwriting this jump with a jump to the next translated block in program execution order. Embra chains both the taken and not taken side of conditional branches.

Having the MMU translate addresses presents some interesting design choices for the translation cache (TC). Although using virtual addresses to index into the translation cache would have allowed the lookup to proceed without having to translate the PC into a physical address, Embra could not use this approach because different processes may have different code mapped at the same virtual address. Instead, Embra uses physical addresses in the tags of the translation cache.

Using physical address tags means that re-translation is only necessary when the physical memory containing code is changed. Since most operating systems cache code pages in physical memory between program invocations, they change infrequently. Because operating systems such as Unix make heavy use of shared code segments and shared libraries, the benefits from physical address tags include lower translation rates for multiprogrammed workloads, lower space requirements for the translation cache, and lower startup time for simulated applications.

Physical address tags have two performance disadvantages. The first is that translation cache lookups require translating the virtual address of the program counter to a physical address. This results in a slower main dispatch loop. Switching to physical addresses also breaks the user level chaining optimization because it is possible that a jump or branch instruction transfers control to different physical memory pages for different processes. The effect of slowing down the dispatch loop and increasing its execution frequency causes significant performance degradation. To maintain performance we redesigned the Embra chaining system to handle physical addresses.

3.5.1 Chaining using physical addresses

Using physical addresses for translation cache tags means that Embra can not implement chaining as easily as user-level simulators. It is possible for two processes to map a common code page at one virtual address and have different code pages mapped at another virtual address. Control transfer instructions which jump between these two virtual addresses will generate chains that are not valid for all of the processes sharing the page.

Fortunately, the condition causing this problem is fairly rare. Most of the time in IRIX a code segment's pages are mapped at the same virtual address in all processes using the code segment. To detect if code segments are mapped to different locations, Embra includes code at the start of all translations which checks that the physical address of the current program counter is the same as the physical address of the translation being executed. This lookup is done by indexing into the MMU relocation array using the program counter and comparing the physical address there with the physical address of the translation—a constant known when the code was translated. If these addresses are on the same physical memory page, the rest of the translation is executed. If the addresses are not on the same physical memory page, the code jumps to the main dispatch loop to go though the full lookup. In this case the old chaining value is overwritten with a new chaining value so that future chains will work for this process.

3.5.2 Additional chaining performance

Even with the above optimizations, we still measured a significant dispatch lookup overhead in Embra. We investigated the problem and found it was due to the heavy use of register indirect jumps in many workloads. Register indirect jumps cause a problem for chaining because the register could take on different values on each invocation and hence can not be chained. These register indirect jumps were mostly due to the procedure calling sequence used by the MIPS compilers, which requires that most procedures are called with a register indirect jump. At runtime these registers usually have the same value each time a given jump instruction is executed.

To reduce the overhead of these register indirect jumps we implemented *speculative chaining* which enables Embra to chain any jump. Speculative chaining works by chaining indirect jumps to a code fragment that checks to see if the destination code is at the correct virtual and physical address. This check is performed by comparing the current program counter virtual address with the virtual address used to generate the translation. If the addresses match then the chaining is valid and the rest of the translation can be executed. If the addresses do not match, control is transferred to the slow path of the main dispatch loop. Speculative chaining improved Embra performance on some workloads by over 20%.

To avoid unnecessary checking during chaining, the prelude of a basic block translation first checks to see if the virtual PC is correct, it then checks if the physical PC is correct, and finally it performs the MMU residency check. Different kinds of chaining use different entry points. The first entry point allows incorrect speculative chains to be detected; the second entry point allows incorrect non-speculative chains to be detected (where two unrelated processes might be sharing a single physical code page). The MMU residency check can be bypassed for chains between code that resides on the same physical page, allowing one translation to transfer control directly to the next. This optimization reduces MMU modeling overheads for application loops that are contained in a single code page.

3.6 SimOS support in Embra

The previous section presented the features required of Embra for efficient machine simulation. Being a part of SimOS places additional requirements on Embra that are explained here.

3.6.1 Event callback queue

In the SimOS environment, CPU simulators must support the event callback mechanism used by the I/O devices of the simulator. The event callback mechanism permits device simulators to request that a function be called at some point in the simulated future. For example, a periodic interrupt timer might request a callback for 10 milliseconds in the future so it can post a timer interrupt. It is the responsibility of Embra to call this function at the specified time.

To support this functionality without a large overhead, Embra emits code in its translations to track simulated time. This is done by incrementing the simulated time based on an estimate of how long the instructions of the translation take to execute. The generated code also checks to see if the time has reached a particular value and calls out to a support function when this occurs. When callbacks are inserted, Embra simply insures that it will call out of the translation cache when the first callback is due to be activated. This allows Embra to execute for as long as possible in the translation cache.

When modeling a multiprocessor, any processor can cause events to be inserted in the callback queue. To avoid expensive per processor checks, Embra only polls the callback queue after each processor has executed a timeslice. This check only adds a few instructions to be the processor switching code of the multiprocessor simulation described in Section 3.4.1.

3.6.2 Annotations

SimOS supports a mechanism called annotations which allow user-provided routines to be called when simulated execution reaches a particular address. Annotations allow SimOS to perform non-intrusive execution monitoring. For example, by annotating the context switch code of the kernel, annotations can track the currently running process. The study in [Rosenblum95a] demonstrates how annotations can be used to track execution time and latencies for various system services.

Embra implements annotations by noting when an annotated code address is being translated. Embra emits code in the translation to call the annotation function. For annotations set at memory accesses, Embra keeps the target page invalid in the MMU relocation array. All memory accesses to this page call out of translated code so all references can be detected.

3.6.3 Debugging

One of Embra's primary functions is for operating system development. It is therefore useful to be able to access the state of the operating system with a debugger. We have modified the serial interface to gdb and added support to Embra so the two can work together. Embra listens on a port which a user can connect to and have full gdb functionality including the examination of memory and the setting of breakpoint and watchpoints (even in OS code where exceptions usually can not be tolerated). Breakpoints and watchpoints are implemented in the same manner as the annotation mechanism described above.

4 Customized translations

This section describes how Embra uses run time code generation to vary the simulation accuracy and output statistics while maintaining high performance. Embra customizes its translations to include code that collects desired statistics or adds additional precision to the machine model. The generation of customized translations can be selected at any time simply by informing the Embra translator and flushing the translation cache. Each newly generated translation will include the extra features.

The performance of customized translations depends on the efficiency of the additional code. For example, if information about instruction opcode frequency or register usage is desired, it is easy to have the translator generate a few extra instructions to increment counters based on the make up of the translated instructions. Since the counter increments will be a small overhead compared to the execution of the translation, this information can be obtained with little additional slowdown. Embra can be configured to quickly gather a number of different statistics including the dynamic counts of floating point operations, taken and not taken branches, and basic block sizes.

It is also possible to customize translations to include more accurate modeling of the machine. For example, customized translations can include instructions that model the pipeline stalls of a processor. Even if it is not reasonable to generate the code to fully model a machine feature, customized translations can model the common cases inline and call out to support routines to handle the complex parts of the simulation. If the common case is fast, and the complex cases infrequent, large slowdowns can be avoided.

4.1 Cache simulation

Currently the largest and most complex example of customized translations in Embra models the memory system stall time of a workload. Memory system stall has been shown to be a large component of many important workloads, particularly for shared memory multiprocessors because they use memory for inter-processor communication. Since most modern machines employ a CPU cache to hide memory stall from the processor, the cache must be modeled to measure the memory stall.

Cache modeling is suited to customized translations because the common case, a cache hit, is simple enough to be processed in the translated code while a cache miss can handled by support routines. These routines model the more complex behavior of the memory system. The challenge for Embra is to make the cache hit detection small and fast enough that it can be included in the translated code. The performance of this check is critical because, like the MMU, the processor cache is accessed on every instruction fetch and data access instruction.

4.1.1 Data access cache check

Embra detects loads and stores to memory addresses not present in the cache by using a scheme similar to the data MMU strategy discussed in Section 3.3.1. To perform this check quickly and compactly, Embra uses an array called the *virtual quick check* or *vQC* that is indexed by virtual cache line number and contains the access status of the cache line. The vQC is similar to the MMU relocation array, except that there is one entry for each cache line in the virtual address space rather than for each page.

The similarity of MMU lookup and cache residency checks allows Embra to further optimize performance by combining the checks in the translated code. All the places that Embra would have to check for MMU relocation are places that it needs to check for cache misses as well. By folding these two checks into one data structure, the customized translation can simulate caches with less slowdown than by adding a cache model on top of the existing MMU relocation.

Each entry in the vQC is only a single byte, the smallest memory unit that can be accessed in a single instruction on the MIPS architecture. This byte is used to encode both the TLB information for the cache line as well as its residence status in the cache (shared or exclusive). If the vQC lookup for an address succeeds, the relocation using the MMU relocation array described in Section 3.3.1 is allowed to proceed. The instruction sequence used by Embra on a data load or store can perform the cache check, the TLB check, and address relocation in 10 instructions, taking 10 cycles (assuming cache and TLB hits in the host system) on an R4400 pipeline.

When an access “misses” in the vQC, the translation calls out to a support routine which determines if the miss was in the TLB or the cache. If it is a TLB miss the appropriate exception is raised and the program counter is set to the OS exception vector. If it is a cache miss, the support routine calls into the memory system simulator. The memory system simulator models the memory stall caused by the cache miss and updates the vQC to reflect the memory line being brought into the cache. Once the memory stall is over, the support routine returns, allowing the simulated instruction simulation to continue.

Because the vQC encodes the state of both the TLB and the cache, it must be updated both when the contents of either the cache or the TLB changes. When a cache line is replaced, the corresponding entry in the vQC must be marked as invalid. When a TLB mapping is removed, all cache lines on that virtual page must have their vQC entry marked as invalid. When a TLB mapping is established, all resident cache lines on the page have their vQC entry filled in. For unmapped memory, Embra reloads the vQC lazily. The cache miss support routine first checks to see if it is really a cache miss before calling into the memory system simulator. If the address was in the cache, the vQC entry is restored and no miss is recorded or modelled.

4.1.2 Instruction fetch cache checks

Just as translated blocks must check for TLB residency on the simulated instructions, as described in Section 3.3.2, each block must also check for cache residency to insure that the code being simulated is present in the cache. Instruction references use the vQC for the TLB and cache residency check in the same way as data references. In the translated code, checks for instruction cache misses occur more frequently than those for TLB misses because there is a much higher probability that an instruction block spans a cache line than that it spans a page. The use of the vQC allows the common case of an instruction cache hit to be processed just as fast as the MMU residency check as described in Section 3.3.2.

4.1.3 Multiprocessor cache coherence

When modeling the caches of a shared-memory multiprocessors there is an additional problem of keeping the caches of different processors coherent. The Embra memory system models the directory structure used in directory-based coherence schemes like the one used in the Stanford DASH multiprocessor [Lenoski92]. As in DASH, the memory system maintains a bitmap for each cache line in physical memory. This bitmap tracks which processors are currently caching the line. If one processor needs exclusive access to a line (e.g. to store to that line), the memory system model invalidates the cached copies of the other processors in the bitmap. Similarly, if the line is exclusive in a cache and another processor accesses it, the first processor has its exclusive copy changed to a shared copy.

The use of the directory bitmap in Embra provides a relatively fast way to model a multiprocessor memory system. Rather than having to check each of the processor’s caches, the bitmap allows the Embra memory system to quickly determine which caches (if any) contain a specific line.

4.1.4 Issues for cache simulation

The use of the virtual quick check in the translated code has several benefits. Embra can perform the quick hit test using a single load instruction, keeping the emitted code fast and compact. Like the MMU relocation array, having the vQC handle the cache hits means the generated code is independent of the cache parameters used. The same hit test works regardless of the size or organization of the cache.

Although the vQC can simulate the behavior of an arbitrary cache organization, it does not necessarily collect the information needed to simulate replacement policies such as LRU. To simulate caches with (non-trivial) associativity, either the translated code can be customized to maintain the LRU bits for a cache line, or the direct mapped data structures can be used with only one line in a set active at any time. In the latter case, the miss handler is responsible for distinguishing real cache misses from LRU maintenance misses. Customization can also support tracking of cache hits as well as misses so that miss rates can be computed.

The chief disadvantage of the vQC is its size, which given in bytes is

$$vQCsize = \frac{VirtualAddrSpaceSize}{CacheLineSize}$$

When using Embra to model the large second level caches found many uniprocessor and multiprocessor systems, we have used a 32 bit address space and 128 byte cache line resulting in a vQC size of 32MB per processor. While it is natural to expect locality in accessing this array, this strategy is a bigger risk for cache simulation than for MMU relocation because the bound on active entries is larger than 64. This bound is dependent on the cache configuration, but for a 1MB direct mapped cache the bound is 8192 entries. The danger is that irregular, sparse access to the vQC will stress the host machine’s cache and TLB. This effect is studied in Section 5.2.

We use Embra with second-level cache configurations, because the virtual quick check will not work as well for smaller caches like the on-chip caches of modern microprocessors. These small caches tend to have small cache line sizes making the vQC quite large. For example, the primary caches of the MIPS R4000 have a 16 byte line size, resulting in a vQC size of 256 megabytes. Although only a very small fraction of these entries are in use at one time, this is still a significant amount of virtual memory; particularly when modeling multiple processors.

Smaller caches also have the problem that they tend to have a higher miss rate than larger caches. Embra’s performance depends on a cache hit rate. If a large percentage of the cache references miss, the vQC optimization does not help performance and can actually hurt performance because of its size (see Section 5.2).

5 Experience and Performance

Embra first booted a multiprocessor operating system in July of 1994 and since then it has been used extensively as both an operating system development platform for the Hive operating system [Chapin95] and as a tool for operating system and computer architecture studies [Rosenblum95a]. Embra has been used extensively to run SGI’s IRIX version 5, a Unix SVR4-based operating system and a large variety of applications running on IRIX. Workloads we have run include large commercial software packages such as the Sybase relational database system and the VCS verilog simulator. Embra has successfully run every IRIX application we have tried. Since the workloads all produce correct results, and many of them contain internal consistency checks, we have a high degree of confidence that Embra executed these workloads correctly.

5.1 Simulation speed

To examine the speed of Embra, we measured the execution time of a workload running “native” on an SGI machine and then measured the speed at which Embra (running on the same SGI machine) could simulate the execution of the workload. By comparing these two workload execution times, we compute the Embra simulation slowdown.

Table 5.1 shows slowdown numbers for various uniprocessor

	Native	Embra			Embra w/caches			Parallel Embra
	Time	Perf	MIPS	%TC	Perf	MIPS	%TC	Perf
Uniprocessor workloads								n/a
052.alvinn	94 sec	3.5x	20.0	97%	7.9x	9.1	60%	
056.ear	163 sec	5.2x	12.4	94%	6.6x	9.9	94%	
023.eqntott	14.4 sec	5.9x	13.4	90%	18.9x	5.3	54%	
008.espresso	30.7 sec	8.7x	11.1	92%	12.1x	8.0	86%	
MAB	19.2 sec	8.9x	5.6	81%	20.5x	3.1	66%	
Multiprocessor (4 CPU) workloads								
ocean -r5000 -t40800	12.0 sec	12.8x	8.1	83%	94.3x	4.4	42%	7.3x
raytrace teapot.env	8.1 sec	13.1x	7.4	94%	81.4x	6.7	72%	4.6x
radix -n2621440 -m5242880	10.9 sec	13.2x	10.8	71%	121.9x	4.7	46%	6.2x
MAB pmake -J4	6.7 sec	38.1x	5.8	83%	221.0x	3.6	63%	9.8x

Table 5.1. Embra simulation speed

For each application we report the native execution time (wall clock times obtained using the shell's timer, best of five trials), the Embra slowdown (Perf), millions of simulated instructions per second (MIPS), and fraction of the Embra execution time spent in the translation cache (%TC). We report these numbers for base Embra, Embra with cache modeling, and Parallel Embra. An SGI Challenge machine with four MIPS-R4400 processors running at 150 MHz was used for the tests. IRIX 5.3 was used as the operating system for both the Challenge and for all the simulated workloads. The workload names starting with a number are from the SPEC92 suite[Dixit92]. The MAB workload is the Modified Andrew Benchmark [Ousterhout90], while the MAB pmake is a slightly modified form of the MAB (described in [Rosenblum95a]). The other multiprocessor workloads are taken from the Splash benchmark suite[Woo95]. Those applications run with settings different from the default have their arguments shown.

and multiprocessor workloads running under three configurations of Embra. The base Embra simulator (Embra) is the fastest possible configuration with the extended TLB support and a course multiprocessor interleaving using a 6000 cycle timeslice. This configuration is intended to be representative of the use of Embra as a workload positioning tool or operating system development platform. In these environments, speed is of primary importance. The **Embra w/caches** configuration is an accurate modeling of a machine using Embra. It uses customized translations to model caches, a standard, 64 entry TLB, and a tight 80 cycle multiprocessing interleaving timeslice. This configuration represents Embra being used as a workload characterization tool. It is the same configuration used in the validation presented in the following section. The third configuration is Parallel Embra configured with one processor per simulated processor, representing the fastest possible multiprocessor simulation using Embra.

The slowdown is presented as the ratio of the time to complete the workload execution on Embra to the execution time on the native machine. Table 5.1 also presents Embra's simulation speed as millions of workload instructions simulated per second (MIPS) and the percentage of the simulator execution time spent in the translation cache (%TC). %TC is correlated with performance because Embra is fastest when executing its translations.

Table 5.1 shows that Embra in its fastest configuration can simulate uniprocessor machines at a rate of over 20 million instructions per second, a slowdown of less than a factor of four from the native execution. This makes it the fastest reported machine simulator capable of running a commercial operating system and applications. In fact, Embra running on start-of-the-art machines in our lab can execute workloads as fast or faster than previous generation machines still in use in our lab.

A more detailed look at Table 5.1 shows that the uniprocessor workload slowdown and instruction execution speed of the base Embra simulator is dependent on the behavior of the workload. 052.alvinn and 056.ear, taken from the SPEC92 benchmark suite, are floating point intensive and see less slowdown than the other, chiefly integer, benchmarks. Embra makes minimal use of floating point registers so the registers of the underlying machine can be dedicated for use by the translated code, reducing traffic to the simulated register file. Dedicated registers and the low exception rates of these workloads, reflected by the large percentage of time spent in the translation cache, accounts for their speed.

Embra shows larger slowdowns for applications that make heavy use of machine features simulated in support routines rather than translated code. The uniprocessor Modified Andrew Benchmark (MAB in Table 5.1) is the most complicated workload presented. It contains over ninety processes being created and destroyed, and frequent traps into the operating system for system calls, TLB misses, and interrupt handling. The simulation of these traps by C language support routines results in the instruction simulation speed dropping to around 5 MIPS and the overall slowdown increasing to a factor of 9. The cache and TLB activity of the host machine, as examined in the next section, also contributes to the larger slowdown for MAB. Nevertheless, being within factor of ten of the real machine allows Embra's user to interact with the simulated workload as they would on the real machine. This feature is of particular value for operating system development and testing.

Table 5.1 also shows the effectiveness of Embra's customized translations at efficiently modeling the details of workload behavior. For the uniprocessor workloads, Embra can add accurate TLB and cache simulation for less than a factor of three over the base configuration. For workloads that have high cache hit rates and low exception rates (like the 056.ear benchmark), the fast cache hit processing of Embra allows cache modeling with a 25% slowdown.

For workloads with a high cache miss rate like the MAB benchmark, the more accurate modeling increases the slowdown by a factor of 2.3. The reduced performance, and reduced %TC, is due to the overhead of handling cache and TLB misses in C support routines.

When modeling a four CPU shared-memory multiprocessor using a single Embra process, Table 5.1 shows slowdowns ranging from less than a factor of 15 for the parallel Splash applications to a factor of 38 for the multiprogrammed workload of the parallel MAB. Intuitively, we expect the simulation of a 4-CPU system using a single native CPU to experience a slowdown of at least 4 times relative to the uniprocessor case. However, when Embra simulates multiple processes in a single process, significant translation and chaining work is shared among the processors, so Embra can do better than linear slowdown. Thus we see parallel applications with better per CPU performance than any uniprocessor application.

MAB pmake provides an interesting comparison between the uniprocessor and multiprocessor versions because the two versions are similar. Embra adds only 7% overhead above the obvious 4x slowdown for simulating four processors using a single real processor. With a large 6000 cycle processor interleaving timeslice, Embra is close to linear slowdown in the number of processors.

By going to Parallel Embra, with each CPU simulated by a real CPU, the slowdowns for multiprocessor simulation almost match those of the uniprocessor workloads. Even for the complex multiprocessor MAB, Parallel Embra slowdown is less than a factor of 10. The cost of this speed increase is a lost of accuracy and determinism making Parallel Embra more useful for positioning workloads and operating system development than for detailed workload characterization.

Accurately modeling multiprocessors requires caches and a tight, 80 cycle processor interleaving. These requirements greatly increase the simulation overheads when compared to fast multiprocessor simulation of base Embra and Parallel Embra. The short timeslices result in more realistic processor interleavings but they increase the overheads due to processor switching. In addition, the modeling of a multiprocessor memory system requires that Embra maintain cache coherence between the processors and provide a timing model for cache misses. These overheads can be seen in the reduction of the percentage time spent in the translation cache.

These factors still do not fully explain the multiprocessor Embra with caches slowdowns. Embra's large data structures are causing memory and TLB stalls. This overhead is described and studied in the following section.

5.2 Self-hosting studies

In implementing Embra we made several design decisions that lower instruction counts at the expense of using large regions of virtual memory. These decisions put pressure on the memory system of the machine running Embra. For example, the loads used to access the vQC array may take close to 1 cycle if the machine's cache and TLB hit rate is high. Otherwise these loads could stall for tens of cycles, greatly reducing the speed of the check.

To better understand Embra's speed on current generation machines and to see how it responds to larger caches, we used Embra and the other CPU simulators of SimOS to study Embra itself. We use Embra to boot the operating system and start an "inner Embra" running on it. Once positioned, we used the more detailed simulators in SimOS to model two different machine configurations running the inner Embra. The first configuration represents a machine available today, and it is close in specifications to the machine used to measure Embra's slowdown in Section 5.1. The second configuration has the same processor with a more aggressive cache hierarchy which will be available in the near future. Figure 5.2 presents the machine characteristics in detail.

SimOS allows us to examine the execution time breakdown of Embra, including the memory system behavior. Figure 5.3 shows this breakdown for Embra simulating a make (part of the MAB benchmark from Section 5.1) on a uniprocessor and the parallel make (the multiprocessor version of the MAB) on a 4-CPU multiprocessor using the two machine models specified in Table 5.2. Results are presented for Embra configurations both with and without cache modeling.

On current machines (1995 model), Figure 5.3 shows Embra spending only about around 40% of its execution time actually executing instructions. The rest of the time is spent stalled on cache misses or TLB misses (these appear as kernel execution and stall time because the MIPS TLB reload is done by an IRIX trap handler). Most of the waiting time is due to primary instruction cache misses, which account for almost 40% of the overall execution. Except for modeling multiple processors with caches, these stall breakdowns are consistent across the different configurations with

	1995	1996
L1 Cache (I)	16 KB, 2-way, 16 byte lines	32 KB, 2-way, 64 byte lines
L1 Cache (D)	16KB, 2-way, 16 byte lines	32 KB, 2-way, 64 byte lines
L2 Cache (U)	1 MB, 1-way, 128 byte lines	4 MB, 1-way, 128 byte lines
L1 Miss	50 ns	
L2 Miss	500 ns	

Table 5.2. 1995 and 1996 machine model cache parameters

Two machine models are used to study Embra's execution behavior. The 1995 machine is modelled after the MIPS R4400-based machines like the SGI Indy workstation running at 200 MHz. It has a split primary (L1) instruction and data cache each 16K and a large 1MB unified secondary cache. The 1996 machine has the same clock speed and pipeline as the 1995 model except that it uses the cache hierarchy of the next generation MIPS processors, the MIPS R10000. The primary caches increase to 32KB and the L2 cache increases to 4MB.

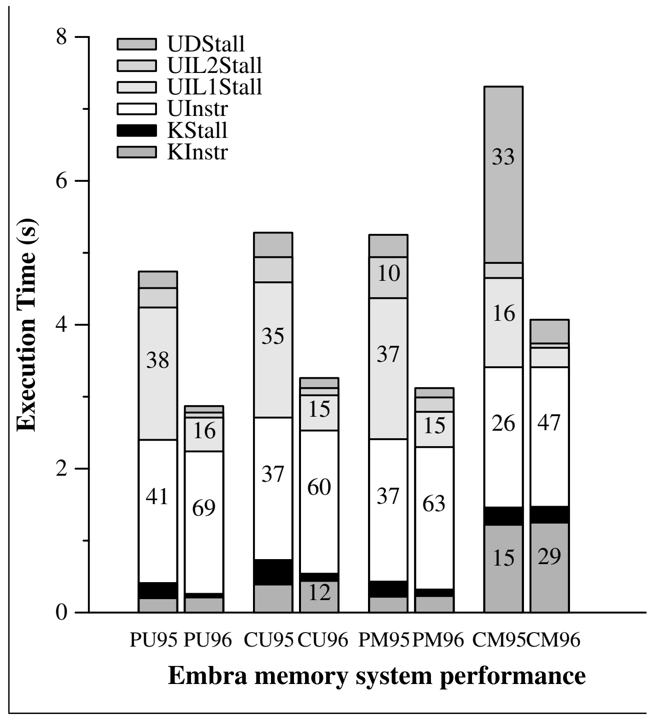

FIGURE 5.1. Execution time breakdown of Embra running on the two machine models.

This graphs shows the execution time breakdown of Embra simulating the MAB workload running on the two machine models. Each bar reaks down the execution time of a run in which Embra executed 400 million instructions (so times are comparable). Bars labeled 95 were run on the 1995 model while those labelled 96 were run on the 1996 model. Experiments include Embra running uniprocessor without caches (PU), Embra uniprocessor with caches (CU), Embra 4-CPU multiprocessor without caches (PM) and Embra 4-CPU multiprocessor with caches (CM). The execution time is further divided into kernel mode instruction execution (KInstr), kernel mode memory stall (Kstall), user mode instruction execution (UInstr), and user mode memory stall. The latter is broken down into stall from primary instruction cache (UIL1Stall), stall from secondary unified cache (UIL2Stall), and user data stall (UDStall). The numbers in the bars are the portion's percentage of the total execution time.

	Program Development			Database		
	Mipsy	**Embra**	**Diff**	**Mipsy**	**Embra**	**Diff**
Workload Time (sec)	6.91	6.92	0.14%	6.53	6.56	0.05%
User Instructions	867032832	867777926	0.09%	163096576	166349727	1.99%
Kernel Instructions	85265664	85444791	0.21%	125053184	123438636	-1.29%
Idle Instructions	251906304	252412708	0.20%	763804672	770374261	0.86%
Data Cache Misses	924844	918413	-0.70%	1302177	1298654	-0.27%
Instruction Cache Misses	862080	861897	-0.02%	1234248	1227044	-0.58%
Data Misses User/Kern	47.56%/52.44%	46.87% 53.13%	±0.69%	63.18%/36.82%	63.19%/36.81%	±0.01%
Instr Misses User/Kern	76.29%/23.71%	76.40% 23.59%	±0.11%	70.47%/29.49%	70.53%/29.44%	±0.05%
TLB Read Miss Exception	699625	703282	0.52%	3072758	3083989	0.37%
TLB Write Miss Exception	54371	54666	0.54%	246928	245175	-0.71%
Avg Disk Latency (ms)	7.864	7.740	1.58%	2.950	2.945	0.02%
Simulation Time (secs)	6159	393	15.7x	12022	637	18.8x

Table 6.1. Embra uniprocessor simulation validation for program development and database workloads
Embra is compared against Mipsy, a machine simulator implemented as a conventional C program. The table shows that Embra can generate the same information, and the same breakdown of that information, an order of magnitude faster than traditional simulators.

the modeling of uniprocessor caches and multiprocessors without caches having similar breakdowns.

The high instruction cache stall means that the large amount of code generated by Embra is causing performance problems. Although the locality of original workload's code may be good, expansion due translation makes Embra's version unable to fit in the 16K instruction cache. The relatively small contribution to the execution time of second level (L2) stall implies that although the translations do not fit in the primary cache they do fit in the 1MB secondary cache.

A striking trend visible in Figure 5.3 is Embra's performance improvement on machines with larger caches. Moving from the 16KB caches of the R4000 to the 32KB caches which will be available on the next-generation MIPS R10000 reduces the instruction stall to less than 20% of the execution time. With these caches the user instruction execution time jumps to between 60% and 70% of the overall execution time. This translates into a nearly 50% performance gain for Embra when running with larger caches. Although Embra's performance is hurt by cache miss stall time on current machines, the larger caches of next generation machines should help alleviate the problem for the studied workloads.

Figure 5.3 also helps explain some of the larger slowdown numbers presented in the previous section. Unlike the other runs, the 4-CPU simulation with cache modeling has a significant amount of data cache stall. Added on top of the instruction cache stall, this results in only a quarter of the execution time being spent actually executing Embra instructions. The source of this data stall is spread fairly evenly across the major data structures of Embra indicating that the data working set size has simply grown too large for the 1MB second level cache.

The data working set of the MP cache run does fit in the larger secondary cache of the 1996 model as evidenced by the dramatic reduction in data stall time. The MP cache run on the 1996 model shows that the percentage of time spent executing user instruction doubles from the 1995 model. Unfortunately, the processor is still only executing Embra instructions half the time. The chief bottleneck here is actually TLB misses which appear as kernel execution and stall because the MIPS TLB reload is done by an IRIX trap handler. Unless TLBs are able to map more virtual address space in the future, Embra techniques, such as the vQC, which use large regions of virtual memory may have their performance suffer from TLB miss processing.

6 Validation and Accuracy

In this section we present the results of studying the accuracy of Embra. Although it would have been desirable to present a comparison of Embra and some real machine as was done in [Bedicheck95], Embra does not attempt to precisely model any particular machine. Furthermore, the information Embra can generate such as cache miss and instruction counts is not normally available on a real machine. Instead we compare Embra with a simulator that has a similar machine model but uses a more traditional implementation method. Our experience with Embra has indicated that accuracy features that can be incorporated into a traditional simulator can also be incorporated into Embra without extreme cost.

We compare Embra with a more detailed simulator in SimOS called Mipsy. Mipsy is structured as a more conventional C program that simulates instruction execution using a simple fetch, decode, and execution loop. We configured Mipsy to model the same single cycle pipeline and single level cache hierarchy used in Embra and compared the statistics reported by the two simulators when running the same workload. We compared counts of instructions, cache misses, OS traps, and the average disk latency. We use the SimOS annotation mechanism to attribute this information to different workload states (user mode, kernel mode, or idle).

The comparison shows that Embra generated statistics that are very close to those generated by much slower, traditional machine simulator. Since the Mipsy statistics have been validated against a real machine [Rosenblum95a], the comparison builds confidence that statistics measured using Embra are correct.

6.1 Uniprocessor validation

The validation workloads are chosen from [Rosenblum95a] because their memory system and operating system behavior is more complex than either the SPEC92 or the Splash benchmarks. The workloads are described in detail in [Rosenblum95a]. The program development workload consists of a parallel make of the modified Andrew benchmark. The database workload consists of Sybase running a modified form of TPC-B. Table 6.1 shows that most of Embra's statistics are within 1% of Mipsy's with the worst case being within 2%. We also looked at more detailed statistics, such as the average length in cycles of each of the different IRIX system calls used in the workloads, and found them to also match. For machine configurations that it can model, Embra can generate the same results as a more conventional implementation an order of magnitude faster.

6.2 Multiprocessor validation

Multiprocessor validation is more subtle than the uniprocessor case. The basic problem is that even simple statistics, such as the number of executed instructions, or the number of cache misses, can be greatly effected by the precise timing of the system. The use of spin locks, other busy waiting methods, and MIPS wait free synchronization primitives means that a slight change in the arrival time of an event could change the execution of hundreds or thousands of instructions as well as affecting the number and distribution of both data and instruction misses.

Table 6.2 presents the comparison of Mipsy and Embra running a 4 CPU parallel make workload. Most of the counts generated by the two simulators agree within 1%, with the disk and idle time within 6% due to a difference in the way the simulators deliver device interrupts. Embra generated its statistics nearly an order of magnitude faster.

	pmake mab		
	Mipsy	**Embra**	**Diff**
Time (s)	13.2	13.5	1.95%
User Instrs	867165952	867814536	0.07%
Kernel Instrs	107628032	106895065	-0.68%
Idle Instrs	1028142080	1085801589	5.61%
Data $ Miss	4757166	4700457	-1.19%
Instr $ Miss	1697353	1696029	-0.08%
D$Miss U/K	36.48%/63.51%	37.32%/62.67%	±0.84%
I$Miss U/K	86.89%/13.09%	86.46%/13.52%	±0.43%
TLB RMiss	700349	699854	-0.07%
TLB WMiss	54920	54584	-0.61%
Av. Disk(ms)	10.279	10.908	6.12%
Sim Time	10432	1481	7.1x

Table 6.2. Embra multiprocessor simulation validation for 4 processor MAB pmake
Embra is compared with Mipsy for a multiprocessor simulation. While the vagarities of multiprocessor simulation are many, most of the final statistics match to within 1%, and Embra generated them much faster. Time is in seconds; U stands for user, K for kernel and $ for cache.

7 Related Work

Embra draws on two areas of current computer systems research—fast simulators and binary translation. Binary translation is a technique used to construct fast simulators, but it is also useful for other types of problems such as software fault isolation[Wahbe93].

7.1 Fast Simulators

The Shade simulator from Sun is a fast, cross architectural, instruction set simulator which uses dynamic translation of binary code. Its influence on Embra has been discussed. Shade's main optimizations are driven by its cross-architectural nature, for instance it tries to balance register usage when the host and target architectures have different numbers of registers. Its basic assumption is that user-provided analysis functions will dominate execution time, so the Shade designers did not extensively optimize the translations or the simulator itself. Embra has a single function—to provide a hardware model detailed enough to run a workload and measure its properties. As such it can be very carefully performance tuned for instruction set interpretation and memory system modeling.

Bedichek [Bedicheck90] and Magnusson [Magnusson93] present complete machine simulators that have focused on execution speed. These simulations have sufficient detail to run arbitrary workloads and can be used for studying system behavior. The fastest of them is still a factor of 20 times slower than real time. Embra can be nearly seven times faster than these simulators while matching their level of detail.

Talisman [Bedicheck95] is a fast simulator that uses threaded code to do fast system simulation. Talisman is an impressive system because it models supervisor mode, and it achieves timing accuracy relative to a hardware prototype. The validation runs rely on several micro-benchmarks that help isolate system components, but does not necessarily give an accurate picture of the system under load. While Talisman models supervisor mode, it does not execute a full OS kernel. Instead, it runs a subset of Intel's NX message passing library. Finally, Talisman models a multicomputer which significantly simplifies the timing model of the simulator (e.g. register interlocks can be modeled by incrementing the cycle count). On shared memory multiprocessors, every memory access acts as an implicit time stamp, making timing accuracy far more complicated.

Peter Magnusson branched off from Bedicheck's work and has been developing a fast SPARC system level simulator. His simulation system also deals with the complexities of MMU address translation and physical memory. However, recently Magnusson [Magnusson95] avoids running an operating system by emulating system calls.

Some fast simulation techniques rely on some form of direct execution of the workload. These are not simulators in the same class as Embra because they do not simulate all parts of the execution of the workload, and therefore always lack the ability to collect some specific data. The Wisconsin Wind Tunnel [Reinhardt93] uses techniques to directly execute shared memory programs on a CM-5. While this technique achieves great speed, it is not as flexible as binary translation, requires an expensive host machine, and can not obtain information about cache hits. Similarly trap driven simulation [Uhlig93] uses direct execution to speed simulation, and it also suffers the same problems of inflexibility and inability to gather some information. Binary translation allows a gradual transition where simulation time increases as more data is gathered.

Embra is fast enough to use as a positioning tool for large workloads, obviating the need for hardware positioning. Embra's vQC is similar to the table lookup used by the Fast-Cache simulator [Lebeck95]. Embra's performance is close to Fast-Cache's, even though Fast-Cache simulates a uniprocessor cache without TLB information for user level programs only.

7.2 Binary Translation

Dynamic code generation and binary to binary translation is a growing area of computer science research. Software fault isolation showed how rewriting binaries statically can provide efficient protection domains. It uses some techniques similar to those in Embra to insure that jumps stay within fault domains, and that code does not loop forever. Some operating systems like Synthesis [Massalin92] have used dynamic code generation to efficiently implement system functionality. More recent experimental operating systems, like the Exokernel [Engler95], are also using code generation to implement system services. [Engler96] describes extensions to the C language to support dynamic code generation.

While the details of the nature and extent of code generation used for the above tasks differ, all tasks share concerns about the quality of code that must be generated quickly and that code's instruction cache performance. Our study has shown that as caches get larger, code generation and binary instrumentation will become an even more attractive technique.

7.3 Static vs. Dynamic Translation

Our decision to use dynamic rather than static translation for simulating the CPU and instrumenting the workload code differs from the decision made by many of the user-level simulation platforms such as ATOM [Srivastava94]. In fact, a static translator could generate code that runs faster than the code generated by our dynamic translator because its translation cost is not part of the application running time.

The dynamic scheme provides us with a number of advantages when simulating entire workloads. Dynamic production of code translations allows the degree of instrumentation to be changed at run time. This enables us to totally remove the overhead of the instrumentation during uninteresting parts of the workload and add it back when it is needed. Additionally, dynamic production of these code translations are essential to an OS simulation environment where new code can be generated by compilers, or read in from disk or from the network. Static instrumentation becomes burdensome when studying the behavior of large, complex systems. We currently boot our simulation system from a replica of the distribution disk of a system containing many hundreds of megabytes of binaries. Finally, static instrumentation schemes have trouble with events that are common in full system workloads, such as dynamically-linked libraries and self-modifying or self-generated code (like Embra itself).

8 Conclusion

We have presented the design and measured the performance and accuracy of Embra, a uniprocessor and cache-coherent multiprocessor machine simulator. From our experience with building, performance tuning, and using Embra over the last year, we have concluded the following:

- A high speed machine simulator, such as Embra, is an invaluable tool for developing and studying complex systems, such as general-purpose machines and modern operating systems. It has enabled us to perform studies on these systems that we would not have attempted without Embra's speed.
- Dynamic binary translation is a useful technique for building a machine simulator that is both fast and accurate. Embra can run arbitrary workloads (including large, complex systems such as a commercial relational database system running on Unix) faster than any reported machine simulator. Embra's uniprocessor cache simulation statistics match a more traditionally implemented reference simulator to within 1%. Many statistics for the more complicated case of multiprocessors also match within about 1%.
- Dynamic binary translation allows Embra to give the user control over the speed versus modeling-detail trade-off—a trade-off traditionally built into the simulator. Rather than having only one speed and a fixed level of simulation detail, Embra provides the flexibility, through customized translations, to change this trade-off even in the middle of a simulation run.
- Sparse data structures can be used in dynamically generated code to implement fast simulation of MMU and cache checks. This trade-off between the code size and virtual address space size is a performance win for current machines but it does put pressure on the memory system and the TLB. The larger caches of future machines appear to help the memory system signifigantly. The TLB may continue to be a problem.

Embra started as a project to explore the use of dynamic binary translation for fast machine simulation. We found it to be well suited for this task. We believe that dynamic code generation has a bright future in high performance computing.

9 Acknowledgments

Primary kudos to Edouard Bugnion whose keen eye for data and general volubility proved instrumental to this paper's creation. Greg Ganger, John Chapin, Matt Frank, and Peter Magnusson all made detailed and interesting comments on various drafts. Thanks to big Ben Verghese in the network department, Steve Herrod, for his Las Vegas legacy, Dan Teodosiu, Scott Devine, Roy Goldman, Robert Bosch, Debby Wallach, Eddie Kohler, Helga Beck, and Dan Yaverbaum. Special thanks to Frans Kaashoek and the MIT Lab for Computer Science for supporting Emmett while he finished this work.

Funding for this research was provided by NSF grant CCR-9257104-03 and by ARPA grant DABT63-94-C-0054.

10 References

[Bedicheck90] Robert Bedichek. Some Efficient Architecture Simulation Techniques, Winter 1990 Usenix Technical Conference, Jan, 1990.

[Bedicheck95] Robert C. Bedicheck. *Talisman: Fast and Accurate Multicomputer Simulation*, In SIGMETRICS, Ottawa, Ontario, Canada, May, 1995.

[Chapin95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. *Hive: Fault Containment for Shared-Memory Multiprocessors*. SOSP, Colorado, 1995.

[Cmelik94] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction Set Simulator for Execution Profiling*, SIGMETRICS, Nashville, TN, 1994.

[Dixit92] Kaivalya M. Dixit. *New CPU Benchmark Suites from SPEC*, 37th Annual IEEE International Computer Conference — COMPCON Spring '92, San Francisco, CA, Feb. 1992.

[Engler95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr., *Exokernel: An Operating System Architecture for Application-Level Resource Management*, SOSP, Colorado, 1995.

[Engler96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaahsoek. *`C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation*. POPL, St. Petersburg, FL, 1996.

[Hastings91] R. Hastings, B. Joyce. *Purify: fast detection of memory leaks and access errors*, Proceedings of the Winter 1992 USENIX Conference, Berkeley, CA, 1991, pages 125-36.

[Lenoski92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. *The Stanford DASH Multiprocessor*. IEEE Computer 25(3):63-79, March 1992.

[Magnusson93] Peter Magnusson. A Design For Efficient Simulation of a Multiprocessor, MASCOTS `93 -Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, La Jolla, California, January 1993.

[Magnusson95] Peter Magnusson and Bengt Werner. *Efficient Memory Simulation in SimICS,* 28th Annual Simulation Symposium, Phoenix, April 1995.

[Massalin92] Henry Massalin. *Synthesis: An Efficeint Implementation of Fundamental Operating System Services*, Ph.D. Thesis, Columbia University 1992.

[Ousterhout90] John Ousterhout. *Why Aren't Operating Systems Getting Faster as Fast as Hardware?*, In Proceedings of the Summer 1990 USENIX Conference, pp. 247-256, June 1990.

[Lebeck95] Alvin R. Lebeck, David A Wood. *Active Memory: A New Abstraction for Memory-System Simulation,* SIGMETRICS, Ottawa, Ontario, Canada, 1995.

[Reinhardt93] Steven K. Reinhardt, Mark D. Hill, James R. LarPrototypingus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. *"The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,"* SIGMETRICS, Santa Clara, CA, 1993.

[Rosenblum95a] Mendel Rosenblum, Edouard Bugnion, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. *The Impact of Architectural Trends on Operating System Performance*. SOSP, Colorado, 1995.

[Rosenblum95b] Mendel Rosenblum, Steven A. Herrod, Emmett Witchel, and Anoop Gupta. *Complete Computer System Simulation: The SimOS Approach. IEEE Parallel and Distributed Technology*, Fall 1995.

[Srivastava94] Amitabh Srivastava and Alan Eustace. *ATOM: a system for building customized program analysis tools*, SIGPLAN Notices , June 1994, vol.29, no.6, pages 196-205.

[Uhlig94] Richard Uhlig, David Nagle, Trevor Mudge and Stuart Sechrest. *Trap-driven Simulation with Tapeworm II*, ASPLOS, San Jose, 1994.

[Wahbe93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. *Efficient Software-Based Fault Isolation*." SOSP, December 1993.

[Woo95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*. Proceedings of the 22nd ISCA, Santa Margherita Ligure, Italy, June 1995.