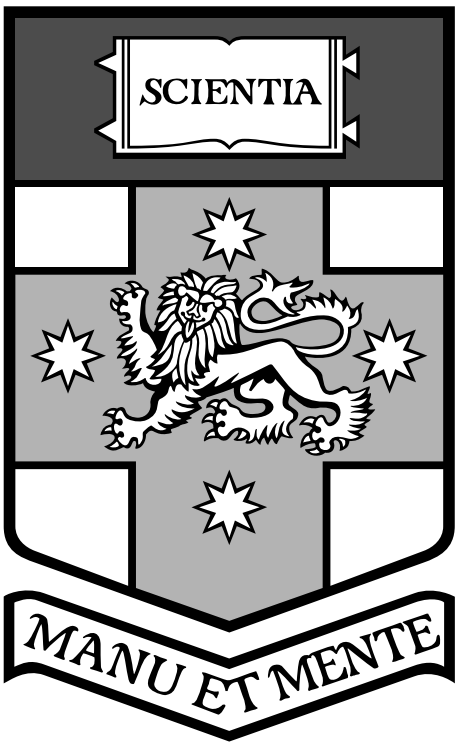
The Design and Implementation of an Extendible Instruction Set Simulator

Patryk Zadarnowski
<patrykz@cse.unsw.edu.au>

Undergraduate Thesis

School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia

November 2000



Thesis Supervisor: **Gernot Heiser**

Abstract

This thesis describes the design of a system-level instruction set simulator, a software tool that emulates enough software-visible aspects of a computer system to allow direct execution of an unmodified operating system in a controlled, transparent environment. Instruction set (ISA) simulation is an established technique used for debugging, profiling and validation of computer systems since the introduction of the first EDSAC ISA in 1949. To date, all such simulators have been designed in an *ad hoc* fashion, targeted for a particular combination of the target architecture, operating system and simulation technique. This approach leaves little room for code reuse, and is unsatisfactory as instruction set simulators represent a substantial code investment, but are typically employed in a research or a development environment as short-lived, project-specific tools. Therefore, there exists a clear need for a robust framework for sharing and reusing simulator components. Sulima is a result of my attempt to design such a framework, in which system-level simulators can be constructed from opaque, autonomous modules, each representing a distinct component of a complete hardware system. For example, one can design modules simulating a particular processor, peripheral device, memory configuration or a floating-point unit for the desired level of accuracy and performance. In this thesis, I also discuss issues associated with design of a simulator based on dynamic translation of instruction sets and a novel technique that uses the Sulima framework to efficiently distribute a simulation of a multiprocessor system across multiple simulator hosts. Finally, I describe the implementation of a fairly sophisticated simulator for a complete MIPS64-based system.

Contents

1 Introduction

The Sulima project begun when I was faced with the task of adopting the well-known instruction set simulator SimOS [1] for use as a teaching and research tool in an Operating Systems course at the University of New South Wales. It soon became obvious that SimOS, while efficient and fairly complete, falls short of its promise of delivering an extendible framework suitable for use in an arbitrary research environment. Based on my experiences with the SimOS code, I have decided to implement a new system-level simulator which would provide levels of extendibility and flexibility not found in any of the existing products.

This thesis describes the design of Sulima, a system-level instruction set simulator. A system-level instruction set simulator is a software tool that emulates enough software-visible aspects of a computer system to allow direct execution of an unmodified operating system (referred to as the "simulator workload") in a controlled, transparent environment. Instruction set (ISA) software simulation is an established technique used for debugging and profiling of computer software, dating back to the introduction of the first EDSAC ISA in 1949 [2]. I include an overview of the relevant work in this field in the *History* section at the end of the *Background* chapter. More recently, software models have been proposed as a specification and validation tool for hardware architectures in projects such as Hawk [3] and sim-nML [25]. However, most of the existing simulators are concerned only with application-level software, and do not simulate such system features as the memory management unit (MMU), direct access to memory by peripheral devices (DMA) or hardware exceptions and interrupts. All of those affect the behaviour of system software and, as such, must be modeled when this behaviour is the subject of an investigation. Extending an arbitrary instruction set simulator to accommodate system software is a fairly difficult task due to the combinatorial explosion of the potential interactions between system components, and the need to simulate multiple value-indexed arrays (such as caches and TLB) which are cumbersome to implement efficiently and prone to programming errors. To date, all such system-level simulators have been designed in an *ad hoc* fashion, targeted for a particular combination of the target architecture, operating system and simulation technique. This approach leaves little room for code reuse, as demonstrated by the attempts to extend SimOS [1] to simulate multiple architectures. SimOS has originally been designed at Stanford University to model behaviour of the Flash multiprocessor [4] based on the MIPS architecture [5]. Subsequently, various groups have attempted to extend the simulator with a support for the Alpha [6], PowerPC [7], x86 [8] and SPARC [9] architectures. Of those, only the Alpha port by Digital (now Western Research Laboratory) and the PowerPC port by Austin Research Lab have been successful, mostly due to the similarity between the MIPS and Alpha architectures and the amount of resources available to the companies sponsoring the projects. The remaining two ports, attempted by the original SimOS team at Stanford, have failed (private communication with Bob Lantz from Stanford.) The URLs for all three SimOS projects have been included in the *WWW Resources* chapter.

Classical uses of an instruction set simulator are described in the

Background chapter. Like most simulator tools, instruction set simulators are typically short-lived tools specialized for a single research project and usually discarded after its completion. However, unlike most other simulators, a system-level simulator is a very sophisticated application that represents a substantial code investment. Many components of a simulator, such as cache models and peripheral devices, can be readily reused in many different scenarios, dramatically decreasing the amount of time and effort required to implement a system to the specific requirements of a particular project. The primary goal of Sulima is to provide a stable framework in which the different components of a simulator for an arbitrary computer system can be modeled as autonomous, reusable software modules. Performance was of a secondary concern, although, naturally, I took care to ensure reasonable efficiently.

This document has been divided into five chapters.

The **Background** chapter introduces the concept of an instruction set simulator by discussing the various simulation techniques and their uses. At the end of the chapter, I include an overview of the historic developments in the field, annotated with the relevant bibliographic references.

In the **Design** chapter, I introduce Sulima by discussing the different components of a simulator and their treatment by the framework. I also describe the design of a simple interpreting processor module. Although interpreter-based simulators are known for their inefficiency [35], they are still an important technique for simulating secondary processors, such as DMA controllers, that tend to be too specialized to benefit from the more sophisticated instruction set translation techniques. Interpreting simulators are also important for executing rarely-used code in a generational instruction set translator. In this chapter, I avoid the use of the actual C++ code fragments in favour of an abstract description of the interfaces, in order to emphasize the design rather than the implementation details, which are discussed later.

The **Distributed Simulator** chapter describes a design of an efficient simulator that utilizes multiple host systems in order to maintain acceptable performance, even when the complexity of the simulator workload significantly exceeds the complexity of the host. In most cases, it is impossible or impractical to extract sufficient amount of parallelism out of the workload to significantly utilize commodity multiprocessing hardware, and consequently there has been little success in the area. I propose to solve the problem by distributing the simulator state, and decoupling the simulated processors by resorting to backtracking whenever a true state dependency is violated.

The **Instruction Set Translation** chapter discusses some issues involved in designing a simulator module that performs a dynamic (or "just in time") translation of the simulator workload into the native instruction set of the host. This popular technique dramatically improves performance of the simulator for frequently-executed portions of the workload such as loops and certain portions of the operating system. The technique has received a reasonable coverage in literature, and accordingly this section simply summarizes the established methods and their suitability for implementation in the Sulima framework.

The final chapter, **Implementation**, describes the actual C++ implementation of the framework and the Koala-R4600 interpreting simulator for the IDT R4600 [10] implementation of the MIPS64 ISA.

2 Background

The benefits of using software tools to simulate the behaviour of a hardware system for both engineering and research purposes are well known. Today, ISA simulators exist for all major instruction set architectures, including Alpha (SimOS [1], SimICS [11]), ARM (ARMulator), IA-32 (SimICS), IA-64 (the HP/Intel IA-64 simulator), MIPS (SimOS, SPIM [12]), PowerPC (SimOS, PSIM]), SPARC (Shade [14], SimICS) and SuperH (e-sim, et al.). Pointers to URLs for most of these projects have been included in the *WWW Resources* chapter. In many cases, these simulators have been developed by the original instruction set design team and provided by the ISA vendors as part of the official software development kits. The four traditional applications of instruction set simulators have been described below.

2.1 Applications of ISA Simulators

2.1.1 System Development

Development of many low-level operating system services on real hardware is a tedious and thankless task, with little debugging information available to the programmer. Further, even when kernel debugging is supported by the architecture, these kernel debuggers usually interfere with the state of the processor executing the investigated code, reducing the quality of the available information. Traditionally, operating system developers have resorted to logic analyzers and hardware debuggers such In-Circuit Emulators (ICE) [14] because software debuggers, being themselves executed in the debugged environment, cannot be used to examine the system state during execution of exception handles and other critical routines on most architectures. The importance and difficulties of operating system debugging are best illustrated by the provisions that most modern architectures make for supporting system debuggers at the hardware level. These provisions range from the watch registers on Alpha [6], IA-32 [8], MIPS [5] and SPARC [9], which trigger an exception every time a specified physical or virtual address is accessed by a software instruction, to hardware support for single-stepping through system code (IA-32), and a complete set of alternate system registers for exclusive use by the debugger on the UltraSPARC implementation of the SPARC architecture [15].

An instruction simulator can potentially eliminate the need for those cumbersome and expensive tools and reduce development time by exposing the complete machine state to the programmer, even during sensitive code segments such as exception handlers and bootstrap loaders. In addition, the simulator, itself running in a complete operating system, may be given access to the complete source code and symbolic information for the debugged kernel, and utilize it to provide some symbolic debugging facilities impossible with hardware tools [16].

2.1.2 Instruction Set Design

The recent trends in post-RISC instruction set architecture design includes

development of highly elaborate software architectures such as HP PA-RISC [17], Transmeta Crusoe [18] and HP/Intel IA-64 [19]. These architectures share the common theme of shifting the burden of performance tuning from the hardware to the compiler technology and other software techniques. In order to achieve good performance, designers of these architectures resort to software simulations for an evaluation of the effectiveness of the design trade-offs and novel architecture features. A good example of this application of software simulation can be found in [20].

An interesting recent application of software simulation is illustrated by the Hawk project [21], which proposes to use a lazy functional language (Haskell) to specify the instruction set architecture [22] and its implementation [23]. Krstic, et al. argue that such a model can be used directly to verify processor implementation [24]. A similar project is conducted at the Indian Institute of Technology, using another functional language ML [25].

2.1.3 System Behaviour Analysis

The most common use of software simulators involves modeling of a computer system in order to analyze its behaviour under certain conditions. Instruction simulators can extract statistical data unavailable to or difficult to collect by a natively-executed software. Data acquisition was the original purpose of the SimOS project [26, 27].

Instruction set simulators are particularly suitable for obtaining memory usage and cache utilization statistics that are imperative to the performance tuning of system software, but difficult to collect without a simulator or some instruction trace data [28, 29].

The importance of software simulation for memory usage analysis is not lessened by the fact that most modern architectures, including IA-32, IA-64, Alpha and SPARC include performance monitoring counters that provide hardware usage statistics such as the cache hit and miss rates. The data available from performance monitoring counters is fairly crude and often noisy due to the interference of the system software. Most importantly, however, performance counters cannot be used to evaluate the impact of adjustments to the cache parameters such as size and associativity which are fixed on an particular implementation of an ISA.

2.1.4 Architecture Emulators

Substituting software simulators for the real hardware is a common technique in education (SPIM, et al.) and in development projects that target hardware with a limited availability, such as obsolete or newly-developed architectures (ARMulator, HP/Intel IA-64 simulator.)

More recently, the success of the IA-32 (Intel) architecture has prompted creation of many software emulators (Bochs, VMware, em86, fx!32, SoftWindows and Crusoe). The URLs for most of these projects are listed in *WWW Resources*. Many techniques employed by these tools are applicable to other instruction set simulators, especially as these emulators are faced with the daunting task of accurately modeling multimedia hardware and other complex devices. However, with the notable exception of Bochs and Crusoe, none of these simulators simulate the workload with sufficient fidelity to be of any use as a debugging or data gathering tool. In particular,

in order to achieve reasonable performance, these tools do not preserve the memory access patterns of the real machine, which renders them useless for performance analysis applications. Further, none of the IA-32 architecture emulators except for Bochs provide debugger access to the simulated machine state.

2.2 Types of ISA Simulators

Instruction set simulators may be categorized according to the simulation methodology and precision. Simulators have been constructed for modeling the system at the switch or circuit level (various VHDL simulators, Hawk, et al.) but these are mostly used for hardware design rather than system analysis. Functional or ISA-level simulators simulate the software-visible state of the system. The majority of the simulators mentioned in this document belong to this class. A few simulators model the hardware at an intermediate level, implementing an informal, descriptive model of the machine rather than its format specification, but simulating its operation with a substantial amount of detail, usually providing a mostly-accurate picture of each pipeline stage in a pipelined microprocessor. Therefore, these simulators are typically capable of computing the exact instruction timings with fidelity impossible in the more relaxed functional models. These simulators are usually referred to as "cycle-level". Examples of cycle-level simulators include MXS (part of SimOS), the SPARC v8 simulator Shade [13] and the SimpleScalar family of processor models (URL included in the *WWW Resources* list.)

2.3 History

The first tool that passes for an instruction set simulator is described in the 1951 EDSAC Debug paper by Maurice V. Wilkes, the creator of the first instruction set architecture [3]. The EDSAC Debugger was a tracing simulator that operated by fetching and decoding the simulated instructions, updating the program counter for branches and executing other instructions directly in the simulator loop. The ST-80 (1984) and Mimic (1987) simulators [30, 31] have introduced the idea of dynamic instruction set translation, with Mimic being the first to implement the technique for a real instruction set and demonstrate the importance of optimizing the just-in-time-compiled code. The first reasonably-efficient system simulator was the g88 [32] written by Rober Bedichek (1990), who has reduced the simulator overhead to less than 100 host instructions per simulated instruction by optimizing the address translation code. Some of the techniques from g88 have later found their way into SimICS, a commercial system-level simulator by Peter Magnusson, et al. [11, 33]. SimOS, written at Stanford as part of the Flash project [1] is by far the best-known and most complete instruction set simulator, which has popularized the technique of dynamic code translation. The use of check-pointing and state roll-backs is due to Sathaye [34] who uses them to model the effects of speculative execution. More recently, VMWare has introduced their nested-system product, the first simulator with a performance overhead of less than 100%. In 2000, Transmeta has announced Crusoe, the first product that substitutes simulation technology for hardware instruction decoding in a commercial processor.

3 Design

As mentioned in the introduction, the primary goal of Sulima is to provide a solid, extendible platform for development of specialized simulators. Extendibility of a software system refers to its ability to adapt to a varying set of requirements. Traditionally, extendibility has been achieved by insisting on modularity of the system, which promotes code reuse in components with similar functionality, and a set of well-defined interfaces that facilitate gradual addition of new components without the need for making modifications to the existing portions of the system. This is the approach taken in Sulima.

The problem of simulating a computer system is equivalent to that of computing a sequence of machine states as a function of the initial state (also referred to as *the workload*), the simulation time, and possibly some interactive events such as the console traffic. This third component of the input is specific to system-level simulators and architecture emulators, but fortunately no more difficult to handle provided that we use only the most recent copy of the state in the computation. An implementation of an interactive console is described briefly in the *Implementation* chapter.

From the point of view of an instruction set simulator, the machine state consists of the register file and the address space visible to each simulated processor, possibly extended with the I/O space on architectures such as IA-32. In system-level simulators, the state also includes the output of peripheral devices such as hard disks and consoles, which would otherwise be masqueraded as part of an address space. Note that, in general, a multiprocessing system has a number of register files and address spaces that contribute towards the global state. When the simulator is used to gather system behaviour data, it is usually necessary to include in the state additional, otherwise inaccessible to software, information such as certain event counters and the various hardware buffers and caches. Finally, the simulator clock, representing a measure of the simulation time, forms a part of the state. Because all instruction set simulators model discrete events, the clock is a simple integer counter whose value is directly proportional to the number of hardware clock cycles elapsed since the commencement of simulation.

The machine state is computed by hardware as a sequence of small modifications to the initial state. This behaviour can be modeled in software directly as a state transition function applied iteratively to the supplied workload, once for each increment of the simulator clock. Therefore, the main problem that the design of Sulima must solve is a method of efficiently partitioning the monolithic system state to promote code reuse and modularity.

3.1 Modules

In Sulima, each distinct component of the simulated system is represented, literally, by a *module.* A module is simply an autonomous portion of the simulator state, annotated with a set of interface functions. Modules with similar structure are grouped together into classes in order to allow definition of common interfaces to similar hardware components. Hence, modules correspond directly to objects in the object-oriented terminology, although,

unlike generic objects, Sulima modules represent a concrete, static component of the simulated hardware. Except for the limited amount of book-keeping necessary to provide a convenient user interface, every piece of simulator state must be attributed to a specific module, which greatly simplifies the component interface.

To illustrate the use of Sulima modules, *Figure 1* describes the decomposition of a simulator of a complete MIPS64 R4600-based system designed and built locally at UNSW.

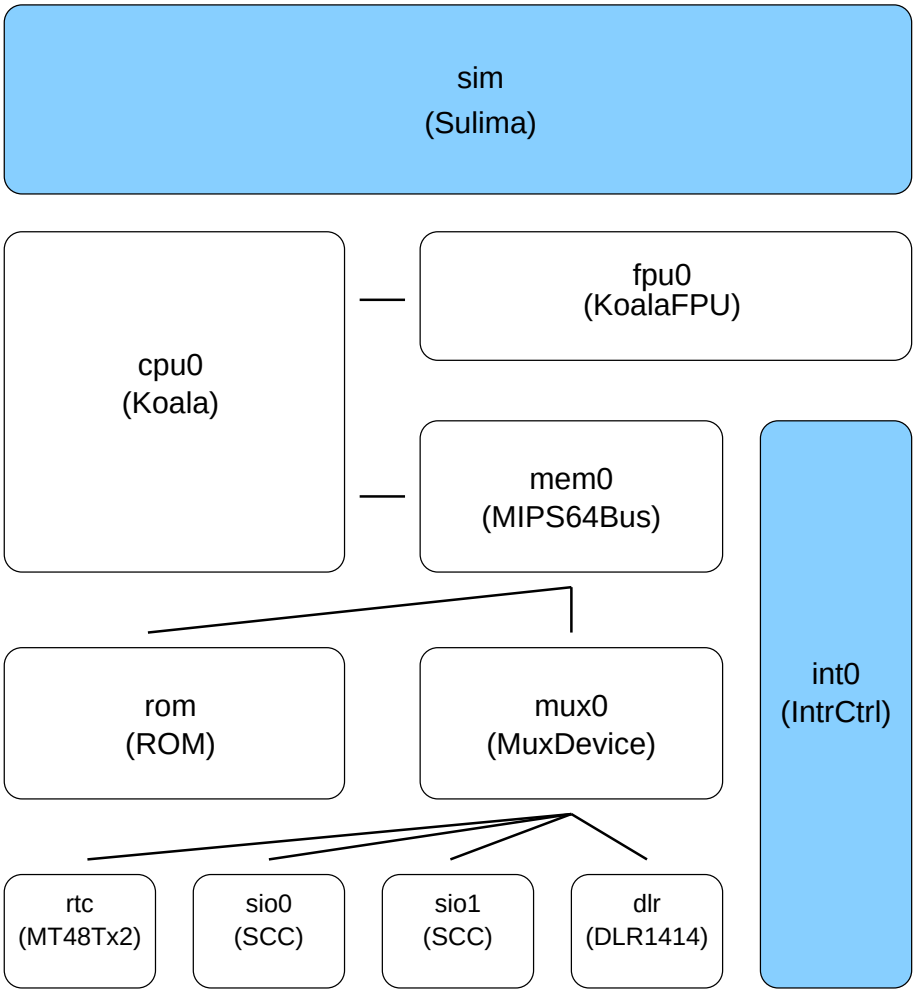


Figure 1 — Modular Decomposition of U4600

In *Figure 1*, lines connecting different modules represent corresponding software interfaces. For example, the peripheral devices (`rom`, `rtc`, `sio0`, `sio1` and `dlr`) are not visible to the main processor (`cpu0`), except through the address space abstraction of the `mem0` module. Both the book-keeping `sim` module and the interrupt controller `int0` are visible to every other component in the system. Each module in a particular system configuration has a unique name and a type representing its implementation. The module types

are specified in parentheses under their names in *Figure 1*. Although not evident in the example decomposition, a particular module may have more than one implementation, chosen dynamically during simulation. In particular, Witchel and Rosenblum [35] have demonstrated that implementation of multiple processor models with a range of performance and accuracy constraints (often spanning several orders of magnitude) is essential for usability of a system-level simulator. A fast, low-accuracy simulator may be used to execute a substantial amount of initialization code leading to the code section under investigation. The simplicity of the module interface in Sulima makes such multiple models trivial to implement and interface.

With the exception of the `mux0`, `mem0` and `fpu0` modules, all modules in *Figure 1* represent distinct hardware components. This is fortunate, as it allows Sulima to model the hardware design directly to provide a robust and flexible interface between the corresponding simulator modules. For example, the complexity of various peripheral devices can be hidden conveniently behind a simple interface consisting of only three functions: `read_mem`, which returns up to one word from a device register stored at a particular physical address, the resulting access latency and a possible bus error indicator, `write_word`, which modifies a particular device register, and returns the access latency, and `deliver_interrupt`, which asserts a particular interrupt line of the interrupt controller, (`int0` in *Figure 1*.)

The `sim` module represents some common infrastructure available to all components of the system. In particular, it provides support for event logging, processor scheduling and module management as described in the *Implementation* chapter.

Modules may be classified into two groups, according to the mechanisms available for affecting their state. The state of a *passive module* may be modified only by another module. Because of that property, passive modules are very easy to design, interface and maintain. All peripheral devices, busses and memory caches are implemented in that manner. *Active modules*, on the other hand, are truly autonomous citizens in a simulator, and represent components whose state is computed iteratively as a function of the simulator clock. They include all processors in a multiprocessing system as well as an occasional DMA, memory or bus controller whose state is more readily computed in this manner. The two kinds of modules are discussed in detail below.

3.1.1 Passive Modules

An implementation of a passive module consists of a data structure annotated with a set of predefined interface procedures. These procedures are invoked whenever a component of the simulator requires access to the encapsulated portion of the simulator state.

Passive modules are easy to implement, but are best suited for modeling of synchronous devices such as memory banks and graphic hardware that do not perform any significant asynchronous processing. Even though dynamic memory technology such as DRAM involves write buffers and periodic refresh cycles, in most cases their effect on the simulator state can be modeled by adjusting access latencies by a factor computed as a relatively simple function of the simulator clock.

For example, a ROM module may be implemented as an array of integers. The `read_mem`(*paddr*, *size*) interface would extract and return the number of bits specified by the *size* parameter from the word at the array index and bit offset computed from *paddr* and return a fixed preconfigured value for the access latency.

Most input hardware, on the other hand, utilizes asynchronous interrupts to avoid expensive software event polling. In general, there is no efficient method of delivering these to the processor simulator directly from a passive module without some form of periodic polling of the device state. However, in most cases asynchronous interrupts can be predicted from an earlier request issued by some processor in the system. For example, interrupts generated by SCSI and ethernet devices are a delayed response to earlier commands issued by writes to a device register. In Sulima, these devices are most naturally implemented as passive modules interacting with an active interrupt controller module, as described in the *Interrupts* section below.

3.1.2 Active Modules

Active modules represent portions of the simulator state that are best modeled iteratively. This kind of modules is implemented by extending the usual module interface with a `run` procedure that iteratively updates the module state until it exhausts the time quantum allocated to it, or encounters a dependency on the state of another active module. `run` is called directly by the Sulima scheduler.

The reasons for selecting a passive or active implementation of a module vary. There must be at least one active module in the system, as they are the only mechanism available for inducing state change and advancing the simulator clock. Therefore, I expect all central processing units (CPUs) to be implemented as active modules. On the other hand, a module designer has a much greater freedom when modeling an asynchronous input device or a secondary processor such as a DMA controller. In those cases, the choice between the module type is largely subjective, and depends mostly on the style of programming the user is more comfortable with and the available host hardware. If used indiscriminately, active modules may impose a significant performance penalty on uniprocessor simulator hosts due to the increased frequency of interruptions to the performance-critical dispatch loop of the main processor. However, computing the device state by repetitive application of a state transition function is typically easier than designing a passive function that performs the same task given only the simulator clock and a previous snapshot of the module state.

The active module mechanism allows Sulima users to extend the simulator with custom processor simulators in much the same manner as that used to implement peripheral devices in a conventional simulator framework such as SimICS [36]. To the best of my knowledge, Sulima is the first instruction set simulator that distinguishes between active and passive module and modularizes the interpreter loop at the heart of every instruction set simulator. Both SimICS and SimOS provide capabilities for statically replacing the simulated architecture, but these involve major modifications to the simulator core. In the case of SimOS, significant modifications to existing code are necessary to extend the model even with a simple peripheral device,

while SimICS provides a simple public interface to custom devices but not processor models. SimOS compensates for the inflexibility of its core with support for generic "event queues." An event queue, polled at the end of each iteration of the main simulator loop, allows users to schedule arbitrary code for execution at the specified simulator time. Because the SimOS events must be scheduled at predefined times, the mechanism, although sufficient for a periodic collection of statistics and simple implementations of polling input devices, is too tightly interlocked with the main execution loop to be useful as a generic implementation mechanism for arbitrary asynchronous components. In Sulima, the functionality of SimOS event queues is available through an active interrupt controller module.

3.2 Simulator clock

Every instruction set simulator, including Sulima, is a discrete event simulator. Because the simulator is largely concerned with computing the software-visible state of the system, there is a simple relationship between the sampling interval of the simulator clock and the frequencies of the hardware clocks driving the simulated processors. Specifically, the optimal sampling granuality can always be computed as the least common multiple of the frequencies of all hardware clocks in the system. In Sulima, in order to assign unique times to the otherwise simultaneous events and to disambiguate software race conditions in a predetermined fashion, this frequency is also multiplied by the number of active modules in the system, rounded to the nearest power of two for efficiency. Events attributed to a particular active module in a simulator with n active modules are offset from the scheduler clock by the module index in the range [0, n-1].

The simulator clock, calculated in this way, can be used to distinguish simulator events and to synchronize modules as required. However, there is no need to maintain its value explicitly, and indeed it is, in general, impossible to do so, because active modules, executed in parallel according to the simulator clock, are usually scheduled sequentially in a round-robin fashion. Therefore, each active module maintains its own idea of the simulator clock, usually scaled down to its local hardware clock frequency.

3.3 Scheduler

A conventional instruction set simulator such as SimOS or SimICS simulates a multiprocessing system using a double nested loop as follows:

```
do
        for each processor
                fetch and execute n instructions on that processor
        end for
repeat
```

where n is either 1 for an interpreting simulator, or a small, fixed number, usually representing a small multiple of the number of instructions fitting in one instruction cache line for translating simulators such as Embra [35].

The simplest way of achieving the same effect in Sulima is to call the `run` function for each active module in much the same manner as above:

```
do
        for each active module i
                call run_i(τ_i)
        end for
repeat
```

Active modules are effectively implemented as cooperative user-level threads. The parameter τ_i is the time slice hint that specifies the average number of simulator cycles that the $\texttt{run}_i$ function should simulate. In order to preserve relative speeds of the different hardware clocks and prevent distortion of the simulator results, the parameters τ_i are chosen so that

$$\tau_i \times \mathrm{f}(i) = F$$

where f(i) represents the declared frequency of the hardware clock driving the corresponding system component, and F is the least common multiple of all clock frequencies in the system.

It is important to note that τ_i is unrelated to the amount of host processor time spent computing the state of the corresponding module. In particular, secondary processors such as interrupt controllers may receive a very large time slice to ensure prompt delivery of interrupts (see the *Interrupts* section below), but, in reality, its state at the end of the time slice may be usually computed using a single comparison testing for pending interrupt events as described in the *Interrupts* section.

Because the overhead of calling $\texttt{run}_i$ may be high compared to the time spent computing τ_i cycles-worth of state, all time slices may be multiplied by a small constant parameter α, effectively "unrolling" the inner simulator loop.

α represents a compromise between the simulator accuracy and the scheduling overhead the user is prepared to pay for it. Most active modules can also be accessed using a passive interface such as interrupt delivery or shared memory. Therefore, by partitioning the simulator state, we introduce inter-modular state dependencies. Because every state access operation is tagged with a specific simulator time, the framework should, ideally, ensure that the state of each module depends only on the past states of the system. As discussed in the *Interrupts* section below, the accuracy of the dependencies between processors quickly degrades with an increase of the magnitude of α, but for most applications the round-robin scheduler is still an acceptable solution.

3.3.1 Preserving State Consistency

At first sight, the simple round-robin scheduler is incapable of preserving dependencies between module states, such as those arising from delivery of interrupts or accesses to the memory shared between different processors.

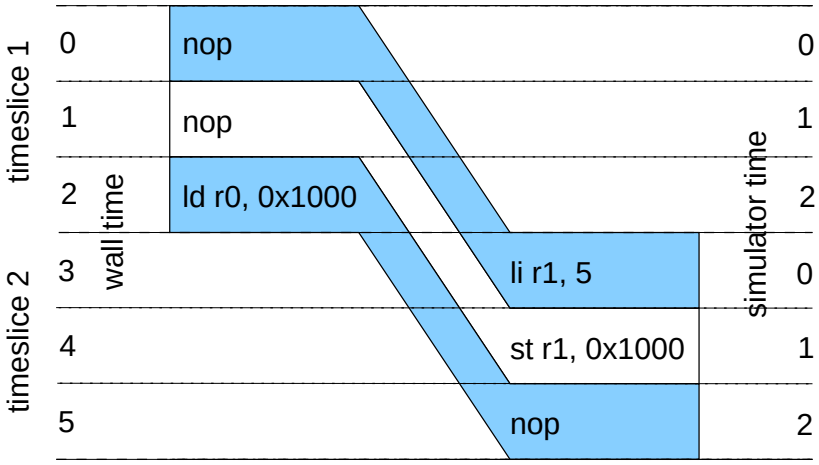


Figure 2 — Violation of Instruction Dependencies

In *Figure 2*, two processors are scheduled sequentially, each receiving a three-cycle time slice. Because the two time slices represent overlapped time intervals (ie. a total period of only three simulator clock cycles), the `st` instruction, scheduled at simulator time 1, should be executed before the `ld` in the first time slice, which is scheduled to occur at time 2. However, because the two processors are actually simulated sequentially rather than in parallel, all three instructions in the first time slice will be simulated before the instructions on the second processor, and the `ld` will not see the value stored at the address 0x1000 by the `st`.

Fortunately, this situation is acceptable, provided that the length of the time slices is comparable to the actual timing accuracy of the events that introduce the state dependency. Although simulated with uniform resolution, in most cases the hardware involved in those events operates at frequencies many orders of magnitude lower than those of the system processors. For example, an access to shared memory costs between 20 and 100 processor cycles on most architectures, and mechanical input devices such as hard disks operate at a millisecond resolution compared to the nanosecond accuracy of the simulator clock. Therefore, small inconsistencies in event timings will be comparable to the behaviour observed on physical hardware. Of course, any such inconsistency should not affect a properly-written system software that correctly implements locks around critical code sections.

The round-robin scheduler is appropriate for simple uniprocessor simulators. On a distributed host or a systems with a large number or simulated processors, it is beneficial to implement a more elaborate scheduler that maintains module dependencies explicitly by examining event timestamps, and rolls inconsistent portions of the state back as required. In *Figure 2*, the state of the first processor would be rolled back to the simulation time 0 when the second processor discovers that it is just about to update a memory location already accessed at a future simulation time. Such a scheduler may significantly improve the performance of a complex system simulator and can even be used to execute multiple active modules in parallel on a multiprocessing host, but requires a significant amount of cooperation between modules and tracking of changes to the simulator state. A design of such

scheduler is described in the *Distributed Simulator* chapter.

3.4 Asynchronous Interrupts

Up to now, I have assumed that all asynchronous input devices are implemented as active modules. This is clearly a suboptimal solution, as increasing the number of active modules in the system may result in an excessive scheduling overhead. Further, the clock frequencies of the input peripheral modules are going to be low compared to those of the main processors, resulting in long processor time slices and consequently a reduced accuracy of the state dependencies.

The solution used in Sulima has been inspired by the SimOS event queues mentioned above. Interrupt events are queued by passive devices for delivery to the indicated processor at the specified type. As mentioned previously, in most cases an event can be predicted by a passive device from a processor-initiated access to its control registers (for an example, see *A Timer Device Module* below.) The device computes the time of the future event and enqueues it in a global interrupt controller module (`int0` in *Figure 1*) using the `schedule_interrupt` interface. The interrupt controller stores the scheduled events in a list sorted by the delivery time. The `run` function of the interrupt controller is a simple test:

```
while next event ≤ end of this time slice
        deliver the interrupt
end while
```

The `deliver_interrupt` interface delivers the corresponding interrupt (represented by an interrupt number) to the indicated processor. The processor module typically implements the interrupt interface by setting a bit in an internal register, and polling the value of that bit after each instruction fetch. For an example, see the *Implementation* chapter.

The interrupt controller module may also be used to periodically collect statistical data, and for internal communication between active modules. For example, in a distributed simulator it is necessary to synchronize the simulator before examining its state with a debugger. This may be implemented using a barrier synchronization signal delivered using the interrupt controller.

3.5 A Timer Device Module

To illustrate the techniques described in this chapter, I will provide a pseudocode for a passive module implementing a simple count-down timer. The device state consists of a single register implemented as an integer `counter`, specifying the simulation time at which the timer is scheduled to expire. When read from, the timer register contains the current count-down value in seconds. When written with a non-zero value, the device resets the counter to the specified value. An interrupt is scheduled when the timer expires, and the timer is disabled by setting the counter to zero.

Like all memory-mapped devices, the timer provides two interfaces, `write_mem` and `read_mem`. The *paddr* (physical address) and *size* parameters

to both interfaces, usually employed to select the accessed register, are ignored for simplicity. In reality, the module should ensure that the parameters select a valid register by verifying its alignment and size requirements, and set the "bus error" bit in the returned result if an illegal access attempt is discovered.

The `read_mem` interface is implemented as follows:

```
read_mem (paddr, size)
        if counter < current simulation time then
                return 0
        else
                return (counter - current time) / clock frequency
        end if
```

The `write_mem` is implemented as follows:

```
write_mem (paddr, size, data)
        counter = current time + (data × clock frequency)
        schedule_interrupt(timer_interrupt, counter)
```

Finally, the `timer_interrupt` routine called by the interrupt controller may be implemented as:

```
timer_interrupt ()
        if counter ≤ current time
                deliver_interrupt
        end if
```

where the current time seen by `timer_interrupt` is set to the end of the interrupt controller's time slice, and the interrupt number and destination is preset in the simulator's configuration script.

Note that, by carefully planning the implementation of the timer state, we were able to predict all relevant events and model a distinctly-asynchronous device using a passive module, without any need for periodic updates of the device registers. Interestingly, a similar technique may be employed in an accurate model of a pipelined processor, by capturing only the final "write-back" stage of the pipeline in the processor state and carefully predicting pipeline disturbances such as jumps, slips and exceptions. Consult the *Implementation* chapter for more details.

The simple technique illustrated above is powerful enough to simulate virtually all hardware devices that do not require cooperation between multiple simulator modules. Controllers and other "connecting" devices may be easier to implement as active modules in order to reduce the amount of public state and the complexity of the interface functions.

4 Distributed Simulation

In this chapter, I describe a design of a distributed, deterministic system simulator based on the Sulima framework. When the simulator workload is significantly more complex than the host on which it is executing, as is the case when simulating a distributed, networked or otherwise multiprocessing system, it becomes desirable to spread the computation across multiple hosts in order to maintain a reasonable performance. Unfortunately, extracting parallelism out of a typical workload is surprisingly difficult, and the existing instruction set simulators have had little success in the area. I propose to solve the problem by distributing the simulator state between hosts, and decoupling active modules in the simulated system by executing instructions speculatively in absence of dependency information, and rolling the simulator state back whenever a state inconsistency is discovered.

4.1 A Case for Determinism

A deterministic system is one in which the output is completely determined by a well-defined function of the input. For example, a multiprocessing hardware system is non-deterministic because of the randomizing effect of the interrupt hardware and the memory access latencies, combined with software race conditions. On the other hand, a single-threaded software simulator is completely deterministic as all execution, including detection of interrupt signals, occurs in a well-defined order in a single dispatch loop nest as described in the *Design* chapter above.

Absence of randomness in a simulator system implies that all results can be reproduced faithfully, simply by executing the simulator on the same workload, and replacing any interactive input with the data played back from the log files created during the original simulation. Because the amount of interactive (usually console) data is relatively small, it is reasonable to store it in a simple text file, annotating each input event with the corresponding value of the simulator clock.

An ability to reproduce all results of a simulation is essential to the usefulness of the simulator as a debugging tool, and beneficial when the simulator is employed to collect performance figures. Without determinism, it would be impossible to retrospectively analyses events leading to a particular condition of interest without maintaining full instruction traces and checkpoints of each execution, reducing the simulator to an elaborate instruction set tracer. Therefore, I consider determinism to be an essential property of any instruction set simulator.

4.2 Extracting Parallelism

Parallelism refers to the ability to perform portions of a single computation simultaneously, decreasing the total computation time. The simulated hardware typically exhibits multiple levels of parallelism. Most processors are implemented using a bit-slicing design, introducing parallelism between the operations on the individual bits of a register. This, however, is closely matched by the similar parallelism of the host. For example, all bits of the

result of a bitwise AND operation can usually be computed using one or two AND instructions on the host system, completely exhausting this level of parallelism. Similarly, the parallelism in a super-scalar and super-pipelined processor is best utilized by translating the simulated instructions into the native instruction set of the host and relying on the instruction-level parallelism of the host system. As demonstrated by ST-80 [30] and Embra [35], dynamic instruction translation can easily improve the performance of an interpreting simulator by an order of magnitude, while threading a model of a quad-issue processor is unlikely to achieve the ideal performance improvement of 400%, at the cost of a much higher code complexity.

Further parallelism exists between distinct components of a hardware single device, such as memory caches and the address translation hardware (TLBs) is exhausted when these components are implemented as passive functional modules illustrated in the *Design* chapter above. Therefore, a distributed simulator should target parallelism between multiple active modules, such as the distinct processors in a multiprocessing system. The task is made realistic by the costs associated with accessing shared data in a multiprocessing system [37]. Because of those costs, sharing of data is typically minimized in a well-designed system, a fact that a simulator can use to its advantage by selectively improving the performance of the common case of independent instruction streams.

4.3 Scheduler

In order to spread a multiprocessing workload across multiple simulator hosts, we must modify the round-robin scheduler described in the *Design* chapter, to allow parallel execution of distinct active modules on separate host processors.

In most cases, the number of simulation cycles and the amount of host CPU time spent in each module is directly proportional to its declared clock frequency. Even though some instructions are more expensive to simulate than others (for example, memory loads is an order of magnitude more expensive than simple register operations), the heavy-weight instructions owe their cost mostly to the amount of state that they process, and therefore migrating such process to a less-utilized host is not a feasible option. For that reason, it is reasonable to assign active modules to host processors statically, based only on their declared clock frequencies and the known relative computational power of the hosts. Once the active modules have been distributed in that manner, a simple round-robin scheduler can be utilized on each host processor without further modifications to the framework. If the number of host processors is comparable to the number of active modules in the simulated system, we can further simplify the system by executing each active module in a separate simulator process, and therefore eliminate the need for a scheduler entirely.

4.4 Event Dependencies

As mentioned previously, every simulator event is associated with a unique simulation time. Two events (for example, two instructions executed on different processors) are dependent if they access the same portion of the

simulator state, and at least one of them modifies the state, as illustrated earlier in *Figure 2*. In that case, the simulator must ensure that the event scheduled first according to the simulator clock is simulated before the later event, and the later event sees the state updated by the earlier one. It is important to note that each processor's program counter, modified by every instruction executed on that processor, contributes to the shared simulator state, and it is also modified by the interrupts potentially issued by remote modules. Further, every instruction is fetched from potentially shared memory. Therefore, sharing opportunities are common and naive synchronization algorithms are unsuitable for use in a serious distributed simulator.

The scaling of the simulator clock described in the *Design* chapter introduces a strong ordering between simulation events. Therefore, no two distinct events in Sulima will be marked with the same time stamp, which simplifies the design of the distribution algorithm presented below.

4.5 Distributed State

The techniques for reducing sharing in distributed memory systems have been covered exhaustively in literature, with further references listed in [38]. Most of those techniques apply to a simulator system if we substitute the global simulator state for the the distributed shared memory.

The granularity of sharing is somewhat arbitrary and depends mostly on the available networking hardware. Li and Hudak [38] argue, based on their informal experiences, for sharing granularity between 256B and 1KB in size. In the remainder of this chapter, I optimistically refer to a unit of sharing as "an object". Typically, every interrupt on every system will be treated as a separate object, and the address space will be partitioned into fixed 1KB pages to maximize utilization of the ethernet medium with its 1500 byte maximum packet size (MTU). In particular, there is no need for the objects of the address space to correspond to the architecture page size, although operating systems are probably likely to partition the working set of each processor on page boundaries to maximize utilization of the address translation hardware.

A distributed version of the memory hierarchy module distributes the physical address space rather than a particular level of the caching hierarchy. Therefore, every cache line with a copy of the data from a particular memory location is included in the corresponding shared object. This approach avoids the overhead of distributing data at the granularity of the individual cache lines (which are typically 32 bytes in size and therefore very inefficient to transfer over the network) and the implementation of "real" cache coherency algorithms in a distributed environment. It is also more efficient because shared access to a particular level of the cache hierarchy usually affects the remaining levels, even if only by modifying the coherency and line ownership information. As described in the above-mentioned IVY paper [38], maintaining memory cache coherency in a loosely coupled distributed environment is prohibitively expensive.

In order to minimize sharing, we attempt to distribute objects between hosts based on their usage. The IVY algorithm maintain multiple read-only copies of a shared object, one on each host which has recently requested its value, or a single writable copy on a host that has updated the

object. Initially, a single writable copy of every object is stored on a predetermined host. Below, I present a modification of the fully distributed IVY algorithm to further decouple the simulated processors, by optimistically executing instructions in absence of certain sharing, and rolling the state back when a state inconsistency is discovered.

4.5.1 Read Operations

When a module requires read access to an object stored locally, it updates its local "read time stamp" for the object, and proceeds with the simulation without any communication with other modules. Therefore, such accesses are almost as efficient as the same operation in a monolithic system. The time stamp does not have to be updated accurately if the overhead of doing so is found to be prohibitive. It is used only to limit the extend of a potential roll-back to the absolutely necessary instructions, and roll-backs of a consistent state, although redundant, are technically correct.

When the object is stored on a remote node, the module requests read access to the object using the standard distributed owner-search protocol, as described in *Figure 3*. The message sent to the owner includes the simulation time of the sender. At this point, no local state has been modified by the current operation, and the module is prepared to receive remote object and roll-back messages, as well as the expected reply.

When the read request is received by a remote node (the temporary "server" for that object) holding a copy of the object, the server must verify consistency of the state before returning a copy of the object to the requesting node. The simulator state is too inconsistent to proceed with speculative execution if and only if the read request is younger than the current value of the object, as indicated by the object's write time stamp. Every reader knows the accurate value of this time stamp which it receives together with its copy of the object.

If the state is consistent, the object is returned to the reader and the object tables updated as described in *Figure 3*. Otherwise, the state of the simulator must be rolled back past the write time stamp of the object. This is achieved by sending a roll-back request to all modules that have read the object since the most recent write operation, as indicated by the copy set stored in the table. The roll-back is essentially the IVY ıinvalidate operation that restores a past module state in addition to adjusting the object table. For more information, see *State Roll-Back* below.

4.5.2 Write Operations

Local writes and the client side of a remote write operation are analogous to the read operations described in *Read Operations* above. Unlike read time stamp, write time stamps must be maintained accurately at all times.

The server for ownership and write access requests must verify the consistency of the state before returning a copy of the object. If the requested write is older than the current value of the object, we consider the state consistent and proceed with an update to the object tables, flushing the copy set and resetting the server's access rights to read-only as described in *Figure 3*. Otherwise, the reply is delayed until the state of the server advances past that of the caller. Deadlocks are impossible as no module may depend on

the future state of another. The interrupt controller from the *Design* chapter may be conveniently reused to enque the write requests until they would no longer result in an inconsistent state.

4.5.3 Invalidation and Roll-Back Server

The distributed page invalidation server in IVY propagates the request to all nodes found in the copy set of the page, and resets the access permissions and the copy set. The modified Sulima version of the algorithm verifies the consistency of the state in addition to the above. Any module that has already accessed the object after the invalidation request has been issued, as indicated by the read time stamp, must roll its own state back to preserve its consistency, possibly generating cascaded roll-back events. If maintaining an accurate read time stamp proves to be inefficient, a pessimistic guess of its value can be made using the simulator clock.

4.5.4 The Algorithm

The shared simulator state includes an object table (probably implemented using one of the well-known page table data structures), which associates a *probable owner*, *access* (read or write) and a *a copy set* (a set of modules holding a local copy of the object) with each shared object in the simulator. These three fields are present in the original design of the IVY algorithm. In order to implement the above modifications, we extend the object table with a read and a write time stamp.

```
Read fault handler:
        ask ptable[p].prob_owner for read access to p
        ptable[p].prob_owner = reply node
        ptable[p].access = read
        ptable[p].read_time = now
        ptable[p].write_time = clock form the reply node
Read server:
        if ptable[p].access ≠ nil then
                if ptable[p].write_time < requesting clock then
                        ptable[p].copyset =
                                ptable[p].copyset ∪ {request node}
                        ptable[p].access = read
                        send p
                else
                        roll_back(p, ptable[p].copyset,
                                ptable[p].write_time)
                        tell the client to retry the request
                end if
        else
                forward request to ptable[p].prob_owner
                ptable[p].prob_owner = request_node
        end if
```

Figure 3 — Algorithm for distributing simulator state (cont'd)

```
Write fault handler:
        ask ptable[p].prob_owner for write access to p
        invalidate(p, ptable[p].copyset)
        ptable[p].prob_owner = self
        ptable[p].access = write
        ptable[p].write_time = now
        ptable[p].copyset = ∅

Write server:
        if the owner then
                if ptable[p].write_time < request time then
                        ptable[p].access = nil
                        send p and ptable[p].copyset
                        ptable[p].prob_owner = request_node
                else
                        enque the request for later
                end if
        else
                forward request to ptable[p].prob_owner
                ptable[p].prob_owner = request_node
        end if

Invalidate and roll-back server:
        if ptable[p].access ≠ nil then
                invalidate(p, ptable[p].copyset)
                if request time > ptable[p].read_time then
                        ptable[p].access = nil
                        ptable[p].copyset = ∅
                else
                        roll_back(p, ptable[p].copyset,
                                request time)
                end if
        end if
```

Figure 3 — Algorithm for distributing simulator state

I believe the resulting algorithm, presented in *Figure 3*, to be correct. However, proving its correctness remains to be done as a future work that must be completed before the algorithm can be used with confidence in a simulator.

4.6 State Roll-Back

The state roll-back operations mentioned above may be implemented easily by interactively restoring any state updated by the rolled-back instructions. Because the state change contributed by each instruction is small, this is a reasonable implementation. For example, a typical three-register instruction modifies a single register and the program counter. Position of branches in the input stream must be recorded in a separate vector, as, in general, it is impossible to predict whether a given instruction has been reached sequentially or as a branch destination. Other operations such as memory accesses

are handled similarly although the amount of state stored for those events is considerably larger.

If we store the branch positions in a bit queue and other data on a simple data queue, the roll-back operation may be designed as follows:

```
Roll-back:
        while the state is inconsistent
                if the top of the bit queue indicates a jump
                        program counter =
                                top word of the data queue
                        discard the top word from the data queue
                else
                        program counter =
                                program counter - instruction size
                end if
                read the instruction at the program counter
                read the corresponding state from the data queue
                restore the simulator state using that data
        end while
```

In the process of restoring the simulator state from the queued data, we may have to initiate roll-backs of remote active modules if undoing the effect of a load or store operation or a reception of an interrupt.

The size of the data and branch queues may be bounded by a reasonable value, as the queues may be always emptied simply by unconditionally synchronizing all active modules in the system to the same clock value.

4.7 Distributing Device Modules

The device modules such as disk drives or SCSI controllers represent a substantial amount of code, but typically benefit very little from distribution. Because these passive modules are usually bound statically to a particular processor, it is reasonable to expect little or no sharing of their state. Therefore, the single-threaded implementation of these devices may be reused in a distributed simulator by unconditionally synchronizing all active modules before every access to a device. This approach is essential for devices that maintain a record of their output (such as disks and consoles) as, in general, it is impossible to roll their stack back once it has been updated.

5 Instruction Set Translation

In this chapter, I discuss issues involved in designing a processor simulator module that performs dynamic (or "just in time") translation of the executed instructions into the native instruction set of the host. This technique, discussed in [31] and [35], can dramatically improve performance of a simulator for the frequently-executed portions of the workload such as the inner loops and most of the operating system. The Embra [35] simulator in SimOS demonstrates that the technique may reduce the simulator overhead to as little as 500% (five host instructions per simulated instruction.) The fastest interpreting simulators claim a slowdown of 50 to 100 times compared to the native execution.

5.1 Simple Instruction Translation

Most instructions are translated by loading the relevant processor state into registers on the host machine, modifying the register and updating the simulator state with the result. On CISC host architectures that support load-and-operate and store-and-operate instruction forms, most instructions can be translated into only two native instructions. For example, *Figure 4* describes the translation of a simple MIPS code fragment into the IA-32 instruction set. For the purpose of the illustration, I assume that the host `edi` register contains the address of the simulated register file. For frequently executed blocks, redundant register accesses can be eliminated using standard copy propagation techniques as mentioned in the *Optimizations* section below.

In this scheme, the main fetch execute loop, marked by the `main_loop` label, is used to dispatch the basic blocks of simulated instructions (ie. the smallest blocks of code ending with a branch instruction) as described by the following pseudo-code:

```
do
        if the program counter is in the translation cache then
                if the cache does not contains a valid translation
                        translate the code
                else if required (*)
                        optimize the existing code
                end if
                jump to the translated code
        else
                insert an entry in the translation cache
                interpret the instruction
        endif
repeat
```

The heuristics for choosing to optimize the code (on the line marked with (∗) above) can be based on a counter stored in the translation cache and incremented on each use of the corresponding code fragment. The possible optimizations are discussed briefly in the *Optimizations* chapter below.

The translation cache consists of a large hash table that maps values of the program counter (ie. jump destinations) to fragments of pre-translated

code.

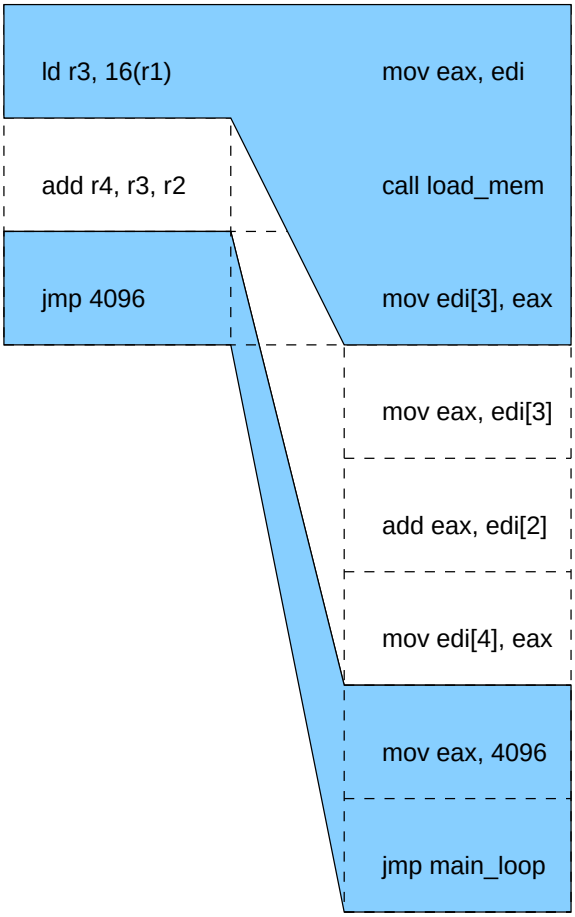


Figure 4 — Instruction Set Translation

5.2 Memory Hierarchy Issues

Maintaining memory cache information in a dynamic instruction translator is more problematic than in a simple interpreter, as demonstrated by Mimic [31], Embra [35] and SimICS [33]. Because loads and stores involve a complicated physical address computation and are unlikely to benefit from instruction set translation, we can simply call the interpreted implementation of those instructions from the translated code. However, instruction fetches must still be handled reasonably by the translator, even though the fetched instructions are not used after they have been translated into the native instruction set. Updating the simulated instruction cache state before every simulated instruction would dramatically reduce the benefits of instruction translation. Fortunately, the interpreter interacts with the instruction cache only during instruction fetches, and therefore the additional checks can usually be removed from the translated code and performed once on each entry into a

basic block. Embra and SimICS further improves the performance of simulated memory accesses by utilizing the host's address translation hardware. Embra replaces the simulated TLB with a linear page table (virtual array) that can be accessed with as little as eight native instructions [35].

5.3 Memory Management

The translator has to manage a large volume of small, variable-size code fragments. Traditional dynamic memory allocators are a particularly poor fit for this usage pattern. A better data structure for an instruction set translator is a simplified generational garbage collector, described, amongst others, by Jacob Seligmann and Steffen Grarup [39].

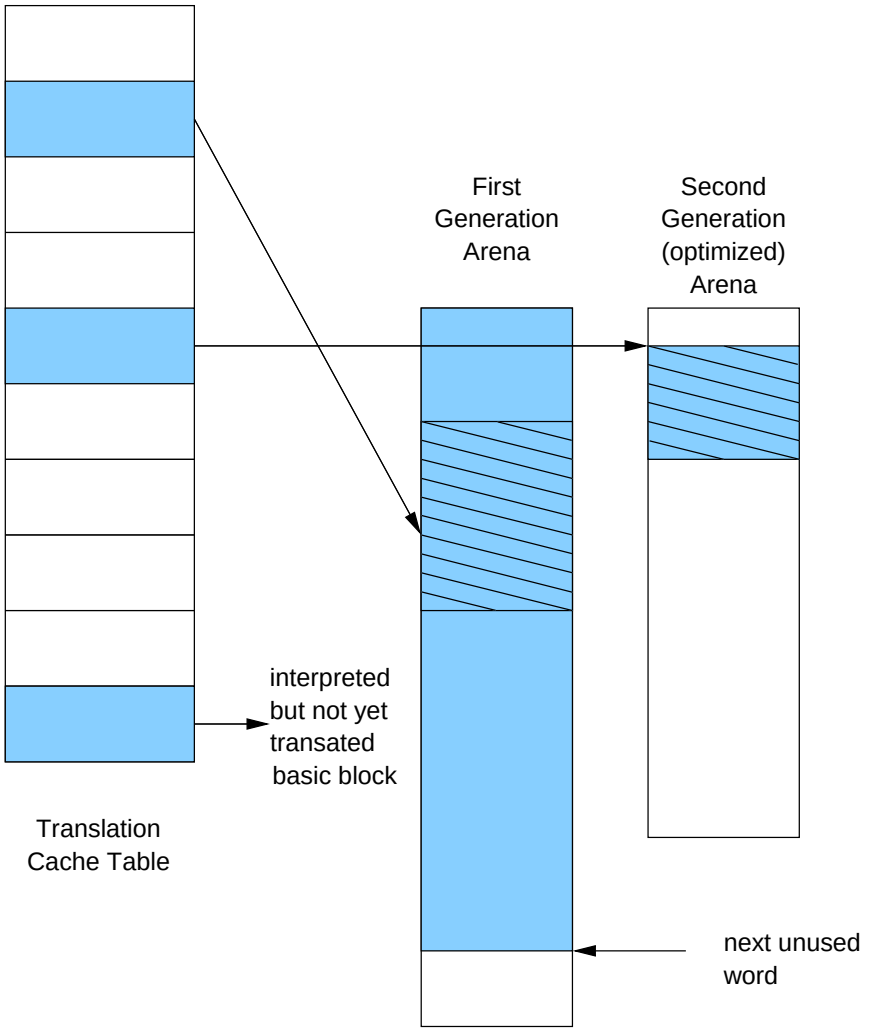


Figure 5 — Memory Management in a Translator

The first-generation arena contains the freshly translated code, while subsequent generations contain only optimized code. This style of garbage

collection is particularly well-suited to our needs as each code fragment is built incrementally without any assumptions about its final size, by storing each translated instructions at the end of the main arena maintained by a single "free store" pointer as illustrated in *Figure 5*. This eliminating the need for expensive reallocations of large fragments when they grow to exceed the initial size estimate. When the preallocated space is exhausted, the garbage collector is called, which makes a single pass over the first-generation arena, moving any optimized code into the second-generation arena and discarding any unoptimized (and hence rarely used) fragments. Because all pointers to the translated code represent jump destinations and as such are stored only in the translation cache table, and all unoptimized jumps are simulated using the table, heap compaction is trivial. The hash function that translates jump destinations into translation cache indexes should be designed carefully to allow quick invalidation of cache entries: in particular, it must be possible to compute the range of translation cache entries for a particular page of physical memory in order to invalidate those entries when the page is modified. To improve the performance of store instructions, each TLB entry should be extended with an "executable" bit to indicate presence of translation cache entries referring to the page. On architectures that support "execute" permissions for a page (IA-64), this information, maintained by the simulated operating system, may be used instead.

5.4 Optimizations

Optimization refers to transformation of a code fragment into an equivalent, but more efficient code. After a code fragment has been executed frequently enough, it may be beneficial to optimize it in order to improve subsequent performance of the simulator [31]. The most significant optimization that can be performed on the generated code is merging of basic blocks by predicting certain branches, therefore reducing the frequency of jumps to the main dispatch loop. As demonstrated in [40] and other publications, backward branches can be predicted as taken with a success rate of over 90%. This is particularly significant as many inner loops are likely to consist of a single basic block, and as such may be simulated completely using translated code without any intervention of the main dispatch loop and the translation cache, provided that one integrates the usual interrupt and instruction cache tests into the code in the block.

Other useful optimizations include copy propagation, constant folding, code scheduling and register reallocation [41]. More sophisticated optimization techniques are not feasible due to the excessive, almost always non-linear, costs involved. One may reasonably assume that the simulated code has already been optimized whenever appropriate, and therefore any optimizations performed by the simulator should utilize the differences between the simulated and native instruction sets.

Of the optimizations listed above, copy propagation is the most important as a naive translation will result in a contention of only a few registers used to fetch operands from the register file. Further, the loads and stored of simulated registers introduce many data movement instructions into the translated code that may be optimized away with little difficulty. This optimization should be followed by register allocation to accommodate the

available register file sizes between the host and the simulator. Constant folding is another simple optimization which may be used whenever the source instruction supports immediate (constant) operands larger than those on the host architecture. For example, MIPS supports 16 bit operands while Alpha has only 8 bit operands, so a naive translator will generate two Alpha instructions for a single MIPS instruction with an immediate operand.

6 Implementation

In this chapter, I describe the current implementation of the framework. The prototype includes a complete implementation of the runtime module manager, the configuration interface and logging, and a partial implementation of state check-pointing. I have also implemented two memory-mapped devices: an accurate model of the MT48T02 real-time clock chip and a simple file-backed ROM module needed for boot-strapping of an operating system. The implementation also includes the state interface for the MIPS64 ISA and a reasonably accurate interpreting model of the IDT R4600 microprocessor. Finally, the `MIPS64SimpleBus` module provides a simple model of a data bus and the main memory.

The framework is implemented in ISO C++, the only polymorphic language sufficiently popular to be used in a project claiming to be extendible. Data polymorphism is essential for a clean implementation of module interfaces. C++ is far from the ideal language for our purposes, but the alternatives such as Ada95 and Modula3 are much less popular and could reduce the potential user base of the framework. Java offers no significant software engineering advantages over C++ (portability of bytecode is clearly immaterial for customized, open-source applications such as Sulima) while introducing additional virtual machine overhead. The deficiencies of C++ are most visible with relation to linking. Ideally, modules should be compiled as individual static or dynamic libraries, but C++ linker bugs make it impossible to do so portably. As a work-around, I link Sulima statically using a single invocation of a linker, and a very complicated makefile. The current implementation of Sulima consists of approximately 10,000 lines of commented code.

6.1 Architecture

As described in the *Modules* section in the *Design* chapter, Sulima is structured as a collection of independent modules. The module types are represented by C++ classes. The arrangement and configuration of simulation components in a particular system is specified statically using a Tcl configuration script. I have chosen Tcl as the configuration language for Sulima, because it comes with a very convenient C and C++ programming interface and is already known to the SimOS users.

6.1.1 Module Hierarchy

The best way of introducing an object-oriented framework is by describing its class hierarchy. In Sulima, I use the C++ inheritance mechanism to group modules according to common functionality, and employ virtual functions to implement the public interfaces.

- BasicModule
  - Module
    - CPU
    - MIPS64Cpu
      - Koala_R4600
    - MIPS64Bus
      - MIPS64SimpleBus
      - GT_64010A
    - ROM
    - MT48Tx2
  - Sulima
- Device
  - ROM
  - MT48Tx2

6.1.1.1 BasicModule

`BasicModule` provides the infrastructure common to the `Sulima` class and all runtime modules. Specifically, it creates the Tcl namespace with the same name as the module, and implements the logging system, by providing the following functions:

```
const char* BasicModule::name() const
```

`name` provides public access to the module name.

```
void BasicModule::log(char c) const
void BasicModule::log(const char* templ, ...)  const
void BasicModule::msg(char c) const
void BasicModule::msg(const char* templ, ...)  const
```

This group of functions provides the log file management and services in the framework. Because logging is essential if the framework is to be applied as a debugger or a data gathering tool, these functions have been implemented with considerable care. Every character output to the log file (`log`) or a console (`msg`) is attributed to a particular module. Each line in the log file is prefixed with the name of the module that has generated the corresponding output, resulting in very clear record of the simulation, that may be processed both manually and with scripts.

```
void BasicModule::define(
       const char* nm,
       T& var,
       T val = T(),
       bool ro = false)
```

Every module has a distinct configuration namespace. This namespace is populated using the `define` function, which creates a configuration variable `nm` bound to the C++ variable `var` with the initial value of `val`. `ro` is used to create read-only constants. The type *T* can be one of `int`, `long` or `const char*`.

```
void BasicModule::define(
       const char* nm,
       SimArg (*fn)(const SimArgs&))
void BasicModule::define(
       const char* nm,
       BasicModule* obj
       SimArg (BasicModule::*fn)(const SimArgs&))
```

Besides variables, modules have to define commands such as a debugger interface to their state. These are introduced into the module namespace using further two `define` methods, which interface C++ functions and class members respectively.

6.1.1.2 Module

The `Module` class extends `BasicModule` with dynamic module management facilities. It provides two constructors for creating a module object from a list of Tcl arguments and from checkpoint data.

```
virtual void Module::reset(bool warm)
```

The `reset` interface is called to initialize or reinitialize the module state from the configuration script. the `warm` parameter may be used to request a partial initialization such as that performed by a "warm" reset on some microprocessors.

```
virtual void Module::checkpoint(Checkpoint& cp) const
```

The `checkpoint` interface is called while the simulator state is being saved to a checkpoint file, and should be implemented for each module that contributes to the global state. `cp` is a C++ output stream.

```
BasicModuleType* find_type(const char* name)
Module* find_module(const char* name)
```

All module types are transparently maintained in a linked list using the C++ static constructor magic, and all modules are linked into yet another list. Both lists are used to identify the installed modules and their types by name in the configuration scripts.

6.1.1.3 CPU

The `CPU` class implements the functionality common to all active modules, in particular the round-robin scheduler, an interrupt interface and the asynchronous event queues.

`ClockValue CPU::`**now**`()`

`now` returns the current value of the simulator clock.

`void CPU::`**poll_events**`()`

The interrupt controller module described in the *Design* chapter is currently merged with the active module interface in the `CPU` class. `poll_events` is called on each iteration of the dispatch loop. It is implemented as:

```
void CPU::poll_events() {
       if (now > earliest) {
             Event* e = queue;
             try {
                   do {
                         Event* f = e->next;
                         e->invoke();
                         delete e;
                         e = f;
                   } while (now > e->when);
                   queue = e;
                   earliest = e->when;
             }
             catch (...)  {
                   queue = e->next;
                   earliest = queue->when;
                   delete e;
                   throw;
             }
       }
}
```

Note that events may use C++ exceptions to leave the polling loop early, for example when preempting an active module. Once enqueued, an event object remains in the queue until scheduled, at which time its `invoke` interface is called. Events are destroyed immediately after invocation, as once-off events tend to be more useful than periodic ones. The functionality of a periodic event may be achieved in Sulima by scheduling a new copy of the event from within its `invoke` method.

`void CPU::`**deliver_interrupt**`(int n)`

This function implements the interrupt delivery mechanism discussed in the *Design* chapter.

`void CPU::run(ClockValue timeslice)`

The `run` interface represents the main dispatch loop of an active module. In the current implementation of the framework, every `run` function must simulate exactly `time slice` simulator clock cycles before returning. Preemption is implemented individually by each active module, which, at the beginning of every time slice, should schedule a preemption event at the indicated time. The `run` methods of each processor are invoked iteratively by the `run_all` function, as described in the *Scheduler* section of the *Design* chapter.

6.1.1.4 MIPS64Cpu

`MIPS64Cpu` implements the processor state defined by the MIPS III architecture [5], including all registers, caches and the TLB. It therefore facilitates dynamic switching between modules that implement different models of the MIPS64 ISA in order to adjust the performance/accuracy trade-off for a particular section of the simulated code:

```
VA pc;
UInt64 gpr[32];
UInt64 fpr[32];
UInt64 cp0[32];
UInt64 cp1[32];
UInt64 hi, lo;
bool ll_bit;
```

`void MIPS64Cpu::sync_state()`

Derived modules are not required to use the state information of `MIPS64Cpu` directly. Instead, they must implement the `sync_state` interface that synchronizes the module state with the state information in `MIPS64Cpu`. For example, caches and TLBs are better maintained directly by each CPU model, but must be preserved across a model switch.

6.1.1.5 MIPS64Bus

A bus module interfaces general memory devices to a processor, by providing the following two interfaces:

```
virtual ClockValue read(
       MIPS64Cpu& cpu,
       PA paddr,
       UInt64* buf,
       int size)
virtual ClockValue write(
       MIPS64Cpu& cpu,
       PA paddr,
       const UInt64* data,
       int size)
```

The interfaces include the requesting processor to provide for interrupt delivery. `paddr` specifies the requested region of memory, while `size` is the number of bytes requested. Valid sizes include all integers between 1 and 8 (MIPS64 supports odd memory access sizes) and multiples of 8 (for block accesses). Results smaller than eight bytes are returned in the following bit range:

$$[(\text{paddr} \,\%\, 8) \times 8, (\text{paddr} \,\%\, 8 + \text{size}) \times 8)$$

This, somewhat surprising, design models the behaviour of the hardware, which maintains memory data on the bus lines selected by the low-order bits of the address. It allows devices to implement memory directly as an array of 64 bit integers, simplifying the interface and eliminating endianess issues.

The value returned by both interfaces represents the pin-to-pin access latency of the memory operation.

6.1.1.6 Device

Similarly, the interface between the bus and the individual devices is provided by the `Device` class.

```
virtual ClockValue read(
       MIPS64Cpu& cpu,
       PA paddr,
       UInt64* buf,
       int size)
virtual ClockValue write(
       MIPS64Cpu& cpu,
       PA paddr,
       const UInt64* data,
       int size)
```

These interfaces differ from the processor equivalents in the restrictions imposed on the `size` parameter. To simplify device implementation, block accesses are handled completely in the bus simulator and therefore the individual devices do not have to handle requests above 8 bytes.
Both interfaces return the access latency. Unlike the processor interfaces however, these latencies exclude the address bus cycles, which are computed by the data bus simulator.

6.2 Modules

In the remainder of this chapter, I discuss the implementation of the actual simulator components currently implemented in the framework.

6.2.1 A Simple Data Bus

The `MIPS64SimpleBus` module provides a minimal implementation of the main memory, data bus and console devices. The module namespace contains the following configuration parameters:

memory configures the amount of RAM in megabytes. The RAM is mapped

at the addresses [0, memory * 2^{20} - 1]. The maximum supported value is 500 to leave room for the device banks.

dev0 specifies the name of the boot device module

dev1 specifies the name of the first device module

dev2 specifies the name of the second device module

dev3 specifies the name of the third device module

The programming interface is just as simple. The interrupt map is:

0 bus interrupt (write error)

1 system console

The status register, stored in the register bank at the physical address 0x1F400000, has the following layout:

bits 0..1 interrupt enable mask

8..9 interrupt cause bits

All of these are read/write. Except for bits 8..9, these bits are never modified by the module. Bits 8..9 are never cleared by the module (they are treated as "sticky" bits.) Setting bits 8..9 in software does not raise the corresponding interrupt.

Each of the five device banks has a preallocated address region which the device is free to use for its own registers. This reduces the problem of decoding the physical address supplied by the processor to only five conditionals. Because the vast majority of memory instructions access the main memory, we can isolate this common case and obtain a reasonable performance out of the simulator. The memory itself is a simple dynamically-allocated array of `UInt64` integers.

The other interesting aspect of `MIPS64SimpleBus` is the implementation of the interactive console module. The console consists of a single one-byte register storing the most recent byte of data read from the terminal. For simplicity, I follow the SimOS technique of polling the host terminal using the `select` UNIX system call, inserted into the event loop at regular, long intervals.

6.2.2 A ROM Device

The ROM device is an illustrative example of a memory-mapped device. Its simple `read_mem` interface is implemented as follows:

```
Time ROM::read_mem(
        CPU& cpu,
        MemAddr addr,
        MemData* datap,
        int size)
{
        assert(size >= 1 and size <= MemData::size);
        assert(addr % size == 0 or
```

```
                !is_power_of_two(size));

        if (freq != cpu.freq)
                update_latencies(cpu.freq);

        // Decode the address.  Note that,
        // as with real hardware, we simply ignore
        // irrelevant address lines (the hardware
        // has no way of knowing where in
        // the address space it is currently mapped.)
        addr &= image.size - 1;

        // Fetch eight bytes around the given address.
        *datap = image.data[addr / MemAddr::size];

        // Assume an 8-bit ROM:
        return latency.read * size;
}
```

6.2.3 The Koala_R4600 Processor Simulator

The R4600 processor is a single-issue MIPS64 processor with a five-stage pipeline. Its simple design makes it possible to model it with high fidelity using a simple interpreting module.

The `Koala_R4600` class implements a simple interpreting simulator of the single-issue R4600/R4700 microprocessor. It is possible to obtain a cycle-accurate simulator by simulating only the final (retire) stage of the pipeline, and adjusting operation latencies by the number of pipeline "bubbles" (wasted cycles) associated with them. For example, an exception taken in the first state of a five-stage pipeline has a latency of five cycles. The same technique is unfortunately inapplicable to super-scalar and out-of-order processors.

The remainder of this section discusses the details of the main dispatch loop at the heart of every interpreting simulator, which provides an illustrative example of a typical implementation of such tool.

```
void Koala_R4600::run(
        ClockValue timeslice)
{
```

First, we register an event that will wake us up at the end of the current timeslice and perform a long jump out of the dispatch loop into the scheduler.

```
        // Register the preemption event
        register_preemptor(now + timeslice);
```

Next, we must prepare for handling exceptions. All synchronous exceptions are simulated using a long jump back to the start of the simulator loop. By taking care to check for the exceptional conditions in the proper order, the exception priorities are implemented without any further overhead.

```
// Prepare for exceptions
setjmp(env);
for (;;) {
```

On MIPS, the first register is hard-wired to 0. Rather than testing each register update, I reset `%r0` at the beginning of each cycle.

```
// Reset gpr[0].
gpr[0] = 0;
```

Advance the simulator clock measuring the progress of the simulation. We start by assuming that every instruction has a latency of one cycle.

```
// First of all, increment the PClock.
// After this, all we need
// to handle is any slips and stalls.
++now;
```

As argued earlier, we must poll for interrupts before each instruction fetch. The instrrupts, and certain other conditions such as hardware resets, are represented by bits in the `events` byte. The details of the calculation are specified in the *R4600 User's Manual.*

```
// Check for interrupts.  In real hardware,
// these have a priority lower
// than all exceptions, but simulating
// this effect is too hard to be
// worth the effort (interrupts and
// resets are not meant to be
// delivered accurately anyway.)

 if (events) {
        if (bits(events, 7, 0))
               process_reset();
        else if (bit(cp0[SR], SR_IE)
                  && (events & cp0[SR]))
               process_interrupt();
 }
```

Address translation is expensive and must be avoided at all cost. Therefore, we maintain a two-entry instruction TLB that caches the two most recent instruction address translations. Because a page of code contains a large number of instructions, the hit rate of this cache is very good. The `asid_match` function performs a comparison of the address space identifiers, encoded for efficiency.

```
// Look up the ITLB. It's not clear
// from the manuals whether the ITLB
// stores the ASIDs or not.  I assume
// it does.  ITLB has the same size
// as in the real hardware, mapping
// two 4KB pages (again, the documentation
// is incomplete, but the size of ITLB
// does not affect anything other than
// the instruction timings.)  Because
// decoding a MIPS64 virtual address
// is far from trivial, ITLB and DTLB
// actually improve the simulator's
// performance:  something I cannot say
// about caches and JTLB.

PA pa;

VA vpn = pc / 4096;
if (vpn == itlb[0].vpn &&
       asid_match(asid, itlb[0].asid))
{
       pa = itlb[0].pa + (pc % 4096);
       lru_itlb = 1;
}
else if (vpn == itlb[1].vpn &&
       asid_match(asid, itlb[1].asid))
{
       pa = itlb[1].pa + (pc % 4096);
       lru_itlb = 0;
}
else {
```

Full address translation involves decoding the most-significant bits of the address and traversing the simulated TLB (implemented as a hash table) to compute the physical address of the instruction. Notice the adjustment made to the processor clock to account for a bubble in the pipeline.

```
       // Do a full address translation.
       // This introduces a slip in the I
       // pipeline stage.  The slip costs
       // 1 cycle for branch, jump and
       // ERET instructions, and 2 cycles
       // otherwise.

       ++now;
       pa = translate_vaddr(pc,
              instr_fetch);
       itlb[lru_itlb].vpn = vpn;
       itlb[lru_itlb].asid = asid;
       itlb[lru_itlb].pa = pa / 4096;
```

```
        lru_itlb = !lru_itlb;
}
```

Similarly, it is advantegous to cache the most recent instruction cache lookups, although the benefits are much smaller as each cache line contains only eight instructions. If we have no luck with the lookup buffers, a full instruction fetch is simulated by calling `fetch`. It will access the cache state (an array of words indexed by the tag obtained from the virtual address), perform the LRU replacement on the cache lines in the set, and update the lookup buffers for future reference.

```
// Access the instruction cache.
// Because the simulated caches are
// slow, we maintain a two-entry
// buffer to cut down full fetches by
// the factor of (up to) sixteen.

Instr instr;

if (ibuf_match(pc, ibuf[0].tag)) {
        instr = swizzle(ibuf[0][pc], pc);
        lru_ibuf = 1;
}
else if (ibuf_match(pc, ibuf[1].tag)) {
        instr = swizzle(ibuf[1][pc], pc);
        lru_ibuf = 0;
}
else {
        // No such luck:
        // fetch the data from the cache.
        instr = fetch(pc, pa);
}
```

Finally, we can decode the simulated instruction. `decode` is implemented as a large switch statement that examines the various fields of the instruction word and performs necessary updates of the state. The MIPS64 architecture has delayed branches (ie. the instruction immediately following a branch is executed unconditionally), which requires some special treatment in the simulator. The `pipeline` and `next_state` variables form a two-position stack of pipeline states that differentiate between normal instructions, branches (which require to update the program counter from the branch target rather than with an implicit increment) and the special case of the instruction address errors, possible only after indirect jumps. In all cases, the stack arrangement of `pipeline` and `next_state` ensure that the result of `decode` is processed with a delay of one loop iteration.

```
// Now, decode and run the instruction.
int next_state = decode(instr);
// Dump the registers if required.
if (trace_level >= dump_gprs)
```

```
                        dump_gpr_registers();

                // Advance the PC.
                switch (pipeline) {
                case nothing_special:
                        pc += 4;
                        break;
                case branch_delay:
                        pc = branch_target;
                        break;
                case instr_addr_error:
                        process_address_error(
                                instr_fetch, branch_target);
                }
                pipeline = next_state;
        }
}
```

6.3 Building and Using Sulima

To install Sulima, download and extract the tar file into your home directory. The tar file will extract itself into a directory `sulima.`

There is no makefile in the source distribution. Instead, the makefile is created dynamically using the `install` shell script.

Run the `install` script:

```
$ ARCH="xxx"
export ARCH
$ sulima/install -g
```

This will also create an "install.log" file with the output of the script, and hopefully build the simulator system.

The `ARCH` environment variable specifies some string used to identify the host architecture. It is used by the install script to maintain separate build trees for each architecture.

The install script should build the executable

```
sulima/work-$ARCH/bin/sulima
```

which may be executed as:

```
$ sulima <TCL configuration script>
```

For example, the tar file includes a sample script that runs a bare L4:

```
$ sulima/work-$ARCH/bin/sulima
        sulima/runtime/mipsL4/mipsL4.tcl
```

Each time the simulator is executed, it will create a log file `sulima.log`. The simulator has been configured to maintain four rotated logs.

7 Conclusions

7.1 Extendibility

The Sulima framework shows how an extensible instruction set simulation may be implemented by partitioning the simulator state into independent modules with a well-defined interface. By modularizing the main simulator loop, customized simulators can be constructed by selectively replacing components of the system model. The active module abstraction gives module implementators unparalleled freedom of implementation when dealing with complex devices such as DMA controllers, while retaining the functionality of traditional event queue implementations.

7.2 Distribution

The modularized design of Sulima lends itself well to distribution across loosely-coupled hosts. The traditional IVY algorithm can be used to maintain internal consistency of the simulator state in a distributed environment, by maintaining a history of state changes. State dependencies can be easily identified and the affected modules rolled back to a consistent state if required. In addition, the state history may be utilized by debuggers and other tools that examine the simulator state.

7.3 Future Directions

The implementation of the framework is still very young. To utilize the potential benefits of the design, models of many more hardware components should be added to the system. I plan to implement models of Alpha microprocessors in a near future, and a complete simulator of the SPARCv9-based Fujitsu multiprocessor is currently being implemented at the ANU/Fujitsu CAP project at Australian National University. To simplify design of active modules, a tool for generating Sulima modules from formal specification would be a welcome addition to the framework. Finally, in order to use Sulima on large workloads, a translating processor simulator is necessary.

8 Bibliography

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

[2] S. Gill, **The Diagnosis Of Mistakes In Programmes on the EDSAC** Proceedings of the Royal Society Series A Mathematical and Physical Sciences, 22 May 1951, (206)1087, pp. 538-554, Cambridge University Press London and New York.

[3] Byron Cook, John Launchbury, and John Matthews, **Specifying superscalar microprocessors in Hawk**, 1998 Workshop on Formal Techniques for Hardware (Marstrand Sweden)

[4] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, **The Stanford Flash Multiprocessor**, Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313, Chicago, IL, April 1994

[5] Gerry Kane and Joseph Heinrich, **MIPS RISC Architecture** Prentice-Hall, September 1991

[6] **Alpha Architecture Handbook**, Digital Equipment Corp., Maynard, MA, 1992

[7] **PowerPC Architecture**, first edition, IBM Corp., Austin, TX, May 1993

[8] **Intel Architecture Software Developer's Manual**, Intel Corp., 1996

[9] David L. Weaver and Tom Germond, **The SPARC Architecture Manual**, Version 9, Prentince-Hall, 1994

[10] **IDT 79RC4600 Microprocessor User's Manual**, Integrated Device

[11] Peter S. Magnusson, **A Design for Efficient Simulation of a Multiprocessor**, International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), San Diego, January 1993

[12] John Hennessy and David Patterson, **Computer Organization and Design: The Hardware-Software Interface** (Appendix A, by James R. Larus), Morgan Kaufman, 1993. Technology, Inc.

[13] Robert F. Cmelik, and David Keppel, **Shade: A Fast Instruction-Set Simulator for Execution Profiling**, Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems May 1994, pp.128-137

[14] James Gateley, Miriam Blatt, Dennis Chen, Scott Cooke, Piyush Desai, Manjunath Doreswamy, Mark Elgood, Gary Feierbach, Tim Goldsbury, Dale Greenley, Raju Joshi, Mike Khosraviani, Robert Kwong, Manish Motwani, Chitresh Narasimhaiah, Sam J. Nicolino Jr., Tooru Ozeki, Gary Peterson, Chris Salzmann, Nasser Shayesteh, Jeffrey Whitman and Pak Wong, **UltraSPARC TM -I Emulation**, 32nd ACM/IEEE Design Automation Conference

[15] **UltraSPARC TM User's Manual**, Sun Microsystems, Palo Alto, 1997

[16] Lars Albertsson and Peter S. Magnusson, **Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads**, Proceedings of MASCOTS 2000.

[17] **PA-RISC 1.1 instruction set architecture**, third edition, Hewlett-Packard, 1994

[18] **The Technology Behind Crusoe Processors**, white paper avaliable from Transmeta, 2000

[19] **The IA-64 Architecture User's Manual**, Intel Corp. 1999

[20] Marc Tremblay, Guillermo Maturana, Atsushi Inoue and Les Kohn, **A Fast and Flexible Performance Simulator for Micro-Architecture Trade-off Analysis on UltraSPARC TM -I**, 32nd ACM/IEEE Design Automation Conference, 1995

[21] John Matthews, John Launchbury, and Byron Cook, **Microprocessor Specification in Hawk**, 1998 International Conference on Computer Languages (Chicago)

[22] John Launchbury, Jeff Lewis, and Byron Cook, **On embedding a microarchitectural design language within Haskell**

[23] Nancy A. Day, Jeffrey R. Lewis, and Byron Cook, **Symbolic Simulation of Microprocessor Models using Type Classes in Haskell**, CHARME'99 poster session, September, 1999 (Bad Herranald, Germany)

[24] Sava Krstic, Byron Cook, John Launchbury, and John Matthews, **Top-level Refinement in Processor Verification**

[25] V. Rajesh and R. Moona, **Processor modelling for hardware software co-design**, Proceedings of International Conference on VLSI Design, Goa, India, January 1999

[26] Stephen A. Herrod, **Using Complete Machine Simulation to Understand Computer System Behavior**, Ph.D. Thesis, Stanford University, February 1998

[27] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Steve Herrod, **Using the SimOS Machine Simulator to Study Complex Computer Systems**, ACM TOMACS Special Issue on Computer Simulation, 1997

[28] Peter S. Magnusson, **Efficient Instruction Cache Simulation and Execution Profiling with a Threaded-Code Interpreter**, Proceedings of Winter Simulation Conference, 1997

[29] Luiz Andre Barroso, Kourosh Gharachorloo and Edouard Bugnion, **Memory System Characterization of Commercial Workloads**, Proceedings of the 25th International Symposium on Computer Architecture, June 1998

[30] Peter Deutsch and Alan M. Schiffman, **Efficient Implementation of the Smalltalk-80 System**, 11th Annual Symposium on Principles of Programming Languages (POPL-11), January 1984, pp. 297-302

[31] Cathy May, **Mimic: A Fast S/370 Simulator**, Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices 22(7), June 1987, pp. 1-13

[32] Robert Bedichek, **Some Efficient Architecture Simulation Techniques**, Winter 1990 USENIX Conference, January 1990

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

[34] Sumedh W. Sathaye, **Mime: A Tool for Random Emulation and Feedback Trace Collection**, Masters thesis, Department of Electrical and Computer Engineering, University of South Carolina, Columbia, South Carolina, 1994.

[35] Emmett Witchel and Mendel Rosenblum, **Embra: Fast and Flexible Machine Simulation**, Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA, 1996

[36] **SimICS User Guide**, avaliable from Virtutech AB and at `http://www.simics.com/support/user-guide.shtml`

[37] Zhichen Xu, James R. Larus and Barton P. Miller, **Shared-Memory Performance Profiling**, 6th ACM SIGPLAN Symposium on P rinciples and Practice of Parallel Programming, Las Vegas, Nevada, June 1997

[38] Kai Li and Paul Hudak, **Memory Coherence in Shared Virtual Memory Systems**, ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989

[39] Jacob Seligmann and Steffen Grarup, **Incremental Mature Garbage Collection Using the Train Algorithm**, Proceedings of ECOOP'95, Ninth European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 952, pp. 235-252, Springer Verlag, 1995

[40] Jason R. and C. Patterson, **Accurate static branch prediction by value range propagation**, Proceedings of the conference on Programming language design and implementation, 1995, Pages 67 - 78

[41] Steven S. Muchnick, **Advanced compiler design and implementation**, Morgan Kaufmann Publishers, 1997.

9 WWW Resources

The SimOS Home Page
Stanford University
`http://simos.stanford.edu/`

SimOS-Alpha
Western Research Laboratory
`http://www.research.compaq.com/wrl/projects/SimOS/`

SimOS-PPC
Austin Research Lab's Full System Simulation Project
`http://www.cs.utexas.edu/users/cart/simOS/`

The SimICS Home Page
Virtutech AB
`http://www.simics.com/`

ARMulator
ARM Ltd.
`http://www.arm.com/`

IA-64 simulator
Intel Corp.
`http://developer.intel.com/ia64/`

PSIM — Model of the PowerPC(tm) Architecture
Andrew Cagne
`http://sources.redhat.com/psim/`

e-sim
(SuperH simulator) The Virtual Product Company
`http://www.e-sim.com/`

Hawk
Oregon Graduate Institute
`http://www.cse.ogi.edu/PacSoft/projects/Hawk/`

SimpleScalar
Simulation Tools for Microprocessor and System Evaluation
`http://www.simplescalar.org/`

MIPS Architecture
MIPS Technologies, Inc.
`http://www.mips.com/`

SPARC Documents
SPARC International, Inc.
`http://www.sparc.org/`

IA-32 Documents
Intel Corp.
`http://developer.intel.com/`

The Crusoe Processor
Transmeta
`http://www.transmeta.com/`

PA-RISC Documents
Hewlett Packard
`http://devresource.hp.com/devresource/Docs/Refs/PA1_1/`

IDT Processors
Integrated Device Technology, Inc.
`http://www.idt.com/`

Bochs
Bochs x86 PC Emulation Software
`http://www.bochs.com/`

VMWare
VMWare, Inc.
`http://www.vmware.com/`

SoftWindows
FWB Software, LLC
`http://www.fwb.com/`