Talisman: Fast and Accurate Multicomputer Simulation

Robert C. Bedichek

Laboratory for Computer Science, NE43–629
Massachusetts Institute of Technology
Cambridge, MA 02139
robertb@lcs.mit.edu

Abstract

Talisman is a simulator that models the execution semantics and timing of a multicomputer. Talisman is unique in combining high semantic accuracy, high timing accuracy, portability, *and* good performance. This good performance allows users to run significant programs on large simulated multicomputers. The combination of high accuracy and good performance yields an ideal tool for evaluating architectural trade-offs. Talisman models the semantics of virtual memory, a circuit-switched internode interconnect, I/O devices, and instruction execution in both user and supervisor modes. It also models the timing of processor pipelines, caches, local memory buses, and a circuit-switched interconnect. Talisman executes the same program binary images as a hardware prototype at a cost of about 100 host instructions per simulated instruction. On a suite of accuracy benchmarks run on the hardware and the simulator, Talisman and the prototype differ in reported running times by only a few percent.

1 Introduction

This paper describes the structure, performance, accuracy, calibration, and use of Talisman, the Meerkat[1] [2] system simulator We used Talisman to extend performance results from a four node hardware prototype to systems with hundreds of nodes. We also used Talisman to evaluate the performance implications of architectural tradeoffs in the Meerkat design space.

Talisman has a desirable combination of features that, as far as we know, is unmatched by any other simulator. Unlike other simulators used for architectural evaluation, Talisman models the fine detail of hundreds of nodes running significant programs. It is unusually fast for a simulator that models fine detail: it simulates about 700,000 processor cycles per second on a SPARC 10 host. In addition, it models the semantics of virtual address translation and processor supervisor and user modes, and thus executes a full range of operating system and user code.

We achieved simulation efficiency by starting with a fast, threaded-code simulator and adding only those timing models needed to achieve accuracy. This approach resembles that used to gain semantic accuracy in Talisman's predecessor [1]. To measure timing accuracy, we ran a suite of benchmarks on Talisman and the Meerkat prototype. Test results guided our grafting of timing models onto the threaded-code simulator base.

We introduce the roles of architecture simulation in Section 2. Section 3 describes some of the varied approaches to simulation. Section 4 relates the structure of Talisman, the performance of which is analyzed in Section 5. The development of timing models and Talisman timing accuracy are discussed in Section 6. Section 7 describes some of the specific advantages of using Talisman.

[1] Meerkat is a moderately scalable multicomputer architecture that uses a software controlled, circuit-switched network

2 Roles and Benefits of Architecture Simulation

Computer architecture simulators vary widely in their application. They are used by processor architects to evaluate uniprocessor design tradeoffs [8], operating system authors to debug their code [1] and to evaluate operating system performance [6, 25], parallel system architects to assess the performance of large systems [4, 9, 23], and end users to execute programs written for one system on a different host system [19, 26, 27].

Simulators also vary in their performance and the level of detail they can model A common metric is the *slow-down*, or the average number of simulator host instructions executed per simulated instruction (see work by Magnusson and others for a more extensive discussion of slow-down [15, 12]). In general, the more detail that the simulator captures, the greater its slow-down [8, 19]. Slow but accurate simulators have the advantage of capturing subtleties of the target system However, their slow speed limits the size of the system they can model and the number of simulated instructions they can execute. The simulator designer must choose a level of simulation detail that is fine enough to capture important performance artifacts, yet fast enough to model large systems and long-running applications in an acceptable timeframe.

2.1 Benefits of Architecture Simulation

There are several benefits of using a simulator for evaluating multicomputer architectures:

- Simulators can be augmented relatively easily with new measurement and debugging features
- Simulators of large systems are easier to make work and are less expensive than hardware implementations.
- New network interfaces can be added in a few days. It is often impractical to retrofit hardware with new interfaces.

- Simulators can model "ideal" networks that are impossible to build, e.g., an infinitely fast network.
- The determinism of simulator execution makes program bugs repeatable, which is not always the case for hardware implementations.
- Simulators are not subject to prototype failures.
- Simulators can be given to other researchers. While hardware is difficult to transport, a simulator can be sent electronically
- As many simulators can be running as there are hosts and host memory. Thus, multiple experiments can be run concurrently on a simulator. Hardware prototypes are usually few in number.

There are other benefits of simulators, such as the ability to: (1) non-intrusively generate address traces of user and system code, and (2) stress-test operating system software by causing the most serious and complex interrupt and exception conditions.

3 Simulation Strategies

The best simulation method depends on the application of the simulation results. This section outlines several simulation strategies and their applications.

3.1 Microarchitecture Simulation

Microarchitecture simulators are built by logic designers to express and test new designs. They can also execute short sequences of code, enabling designers to evaluate architectural features and debug microcode. Microarchitecture simulators typically have a slow-down around 20,000 [22], making them too slow for debugging all but the shortest code sequences.

3.2 Macroarchitecture Simulation

Macroarchitecture simulators (also called *macro* simulators or *instruction set architecture* simulators) can execute longer-running programs. They are used for studying cache performance and debugging operating system code in advance of hardware availability. Unlike micro simulators, macro simulators often model the chip's timing closely, but not perfectly.

Because they model less detail, they are much smaller and faster than micro simulators. Conventional macro simulators have slow-downs on the order of 10 to 1000. They dispatch instructions by fetching from a simulated memory, isolating the operation code fields, and branching based on the values of these fields. Once dispatched, the instruction's semantics are simulated by reading and manipulating simulation variables that represent the target system's state.

Several techniques can improve the performance of macro simulators. Instead of decoding the operation fields each time an instruction is executed, the instruction is translated once into a form that is faster to execute. This idea has been used in a variety of simulators for a number of applications [8, 10, 17, 19, 26]. It is also used in some processors to translate an instruction set that programmers see into a more RISC-like form that is more efficient to execute [7, 11].

3.3 Direct Execution

The target program can also be executed directly on the simulator host [5, 13, 23] by encasing the program in an environment that makes it execute as though it were on the simulated system. This technique requires that either the host system have the same instruction set as the target or that the program be recompiled. Instructions that cannot execute directly on the host are replaced with procedure calls to simulator code.

Most direct execution simulators inspect and translate all instructions before simulation begins, i.e., statically. This is incompatible with the needs of a simulator that can model operating system code (this is discussed in Section 4.10). To evaluate tradeoffs in multicomputer architecture, we needed to simulate both user and kernel code.

Direct execution simulators are fast to execute instructions that can be run directly, but are often slow both to handle instructions that can not be run directly and to handle exceptions. Depending on the ratio of directly executed instructions to non-direct instructions and exceptional events, direct execution simulators can be faster *or slower* than other kinds of simulators. For example, a simulator that on average executes two instructions directly and then switches to another simulated processor by executing lengthy context-switching code may be slower than a threaded-code simulator (threaded-code is discussed in Section 3.5) that takes more time to simulate instructions but much less time to switch from one simulated processor to another.

A promising approach is the one taken by SimOS [24]. This simulator handles both user and kernel code by dynamically translating target instructions into short sequences of host-native code. Unlike most direct execution simulators, SimOS keeps only a little of the target machine's state in host registers and so is able to multiplex between target processors quickly.

3.4 Blurred Lines Between Simulation Techniques

Several tools can be considered fast macro simulators that dynamically translate code, or direct execution simulators, or profiling tools. The UNIX utility `prof` is a profiling tool that is not usually called a simulator. But it could be considered a direct execution simulator that wraps the target program in an environment enabling execution measurement. Shade [8] is thought of as a fast macro simulator that uses dynamic compilation. While it is more flexible than `prof` and uses dynamic instead of static compilation, it is also a tracing tool. The line between different tracing and simulation techniques is often blurred despite efforts to neatly categorize them.

3.5 The Talisman Approach: Threaded-Code

Measuring Meerkat's design required a simulator efficient enough to run significant programs on hundreds of simulated processors. In addition, it had to model timing accurately. We could not afford to spend years constructing a complex simulator or waiting for results from a slow one.

For these reasons, we wrote a simulator that translates instructions to threaded code [3, 14], which is then executed. The threaded code is cached, so that the price of translation for most instructions is paid just once, the first time they are encountered in the code stream. The result is a simulator that has a slow-down of about 100 per simulated processor. Its timing is close enough to the prototype's that we can use it to run large programs and make meaningful measurements.

4 Structure of Talisman

Figure 1 shows the structure of Talisman. Users interact with Talisman through a symbolic debugger called gdb [28]. Talisman consists of an instruction translator, a threaded-code interpreter [3], cache models, a TLB model, a physical memory system model, and I/O models. Meerkat uses Motorola MC88100 processors [20] and we refer to MC88100 in the text below as the "target"

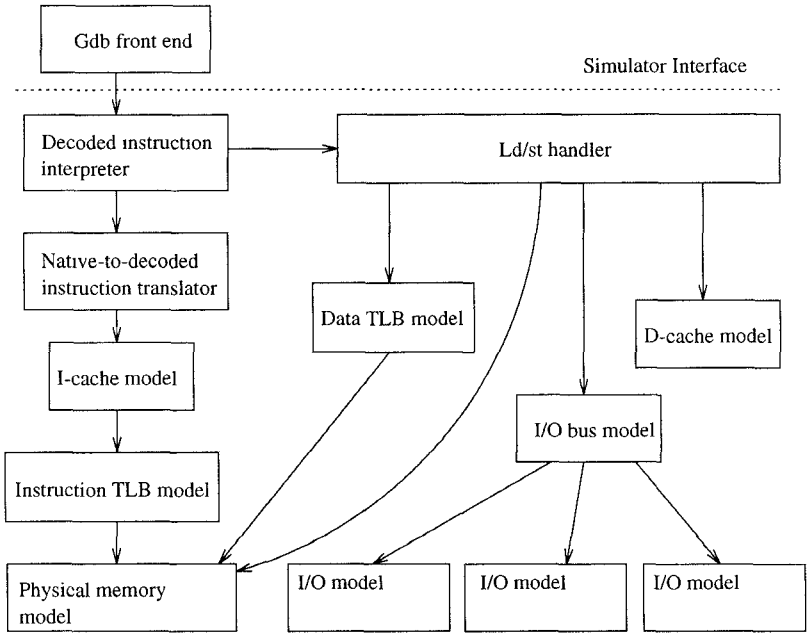

Figure 1: Talisman Structure

Meerkat programs are compiled, assembled, and linked with a set of GNU cross-development tools on a SPARCstation host. The resulting binary image can then be run on either the Meerkat prototype or on Talisman. On invocation, Talisman loads the code, data, and symbol table. It then calls through the simulator interface to (1) place the code and data in simulated memory, (2) set up profiling structures, and (3) initialize registers.

When the user issues the `run` command, the front end calls the threaded code interpreter. Execution returns to the front end upon an exceptional condition, such as a breakpoint or a special trap instruction indicating that the simulated program has finished.

The remainder of this section relates the key techniques that Talisman uses.

4.1 Translation to Threaded Code

Talisman does not interpret target instructions directly. Instead, instructions are first translated to *decoded instructions*, which are cached in structures called *decoded instruction pages*. Only instructions encountered during execution are translated and cached, so unlike Mimic [19], there is little startup overhead.

Decoded instructions contain up to six fields The first field always points to the decoded instruction's *handler*, the code that interprets the instruction. This pointer makes it easy to dispatch decoded instructions. On most simulator hosts, this dispatch consists of two instructions: a load followed by an indirect jump. For triadic target instructions, three of the decoded instruction fields are pointers to host memory that models the target registers. The last two fields hold the length and type of memory access instructions.

Figure 2 shows an unsigned add instruction followed by a load instruction. The add instruction sums the contents of r5 and r6 and stores the result in r4. The load calculates the effective address as the sum of r4 and 1000. It loads a word from the effective address and puts it in r2.

4.2 Instruction Pointers

Talisman maintains two instruction pointers that are updated together and are always kept consistent: the decoded instruction pointer (DECIP) and the modelled instruction pointer (IP). The IP is the value of the target virtual address of the next instruction to execute. After non-branching instructions, the IP and the DECIP are incremented by four and twenty, respectively (target instructions take four bytes, decoded instructions take twenty bytes). Branching instructions are described in Section 4.11.

4.3 Decoded Instruction Pages

Decoded instruction pages contain slots that hold decoded instructions. Each decoded instruction page corresponds to a physical page of a particular node's memory. Decoded instruction pages are allocated when a running program attempts to execute code on a physical page that does not yet have a corresponding decoded page. Both physical page structures and decoded instruction page structures are allocated lazily.

When a decoded instruction page is first allocated, and when it is flushed, it is filled with *decode-me* pseudo instructions. When a decode-me pseudo instruction executes, a target instruction is translated to a decoded instruction. The target instruction is fetched from the address on the physical page that corresponds to the position of the decode-me pseudo instruction in the decoded instruction page. For example, if the decode-me pseudo instruction in the tenth slot is executed, the tenth word in the corresponding physical page is translated. The new decoded instruction replaces the pseudo instruction, and this new instruction is executed.

There are 1025 decoded instruction slots in each decoded instruction page. The first 1024 of these hold decoded instructions that can be in a physical page (a physical page is four kilobytes, and each MC88100 instruction takes four bytes). The 1025th slot holds a sentinel called the *requalify-decoded-ip* pseudo instruction. When a non-branching decoded instruction in the 1024th slot is finished executing, the DECIP is incremented and points to the

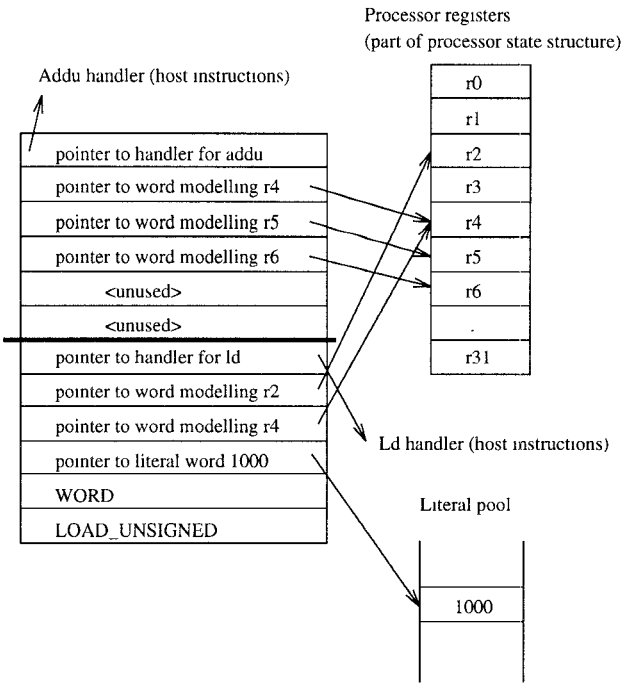

Figure 2: Decoded Instructions for `addu r4,r5,r6` and `ld r2,r4,1000`

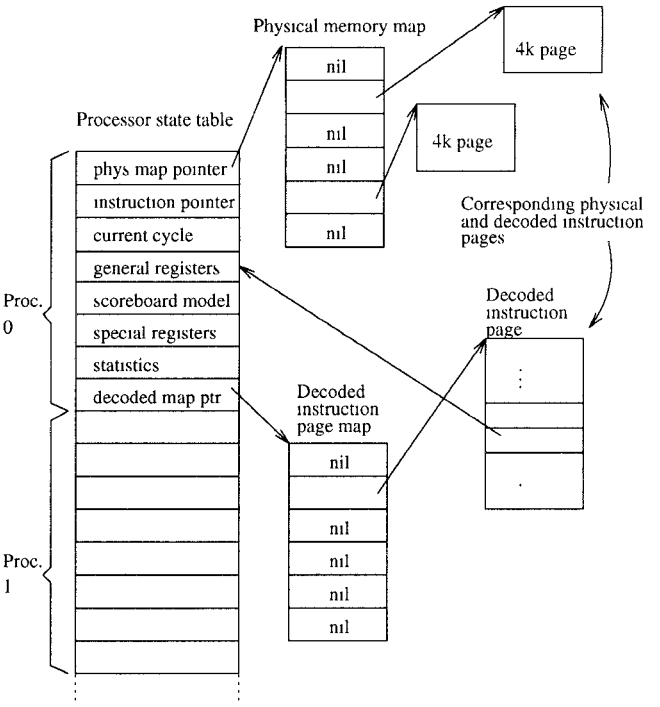

Figure 3: Talisman Data Structures

requalify-decoded-ip pseudo instruction. Because the flow of control has moved off of the page, the DECIP must be recomputed to point to the first decoded instruction of the new page. The use of requalify-decoded-ip pseudo-instructions as sentinels speeds instruction handlers, because they never need to explicitly check for the end-of-page condition.

Dividing the physical and decoded instruction spaces into pages allows incremental, demand-driven allocation of memory. This lazy memory allocation conserves host virtual memory (a point discussed in the next section).

Talisman does not perform garbage collection. Avoiding garbage collection kept Talisman simpler at the expense of higher virtual memory consumption. Since the units of allocation are large (4k for a physical page structure, 20k for a decoded instruction page), garbage collection would probably be redundant with the host's virtual memory system. That is, garbage collection would reclaim memory contained on pages which would otherwise be paged out. Thus, garbage collection would conserve host virtual memory, not physical memory, at the expense of higher simulator complexity.

4.4 Processor State and Memory Use

Figure 3 shows the relationship among processor state, simulated physical memory, and the decoded instruction pages. Upon simulation start-up, the user specifies the number of nodes, number of processors per node, and size of node memory. Talisman allocates an array of processor state structures to match the user's request. The processor state holds pointers to the physical memory map and the decoded instruction map for the node. All of a given node's processor state structures point to the same maps: a node's memory is shared by all of its processors. The state structure also contains all of the processor's registers and counters to maintain execution statistics, such as the cache hit rate.

In Figure 3, note that the second page of physical and decoded instruction memory is allocated (i.e., the second pointer in both the physical and decoded instruction page maps are filled in). In this example, either the front end or a running program accessed the second page, causing a physical memory page to be allocated. Also, an attempt has been made to execute an instruction on the second page, causing a decoded instruction page to be allocated.

The memory maps will gradually fill in as the running program executes memory access instructions and branches to new pages. Typically, however, large pieces of both maps are vacant. The decoded instruction map is especially likely to be sparse; vacancy represents conserved host virtual address space. Because these maps are replicated for each node, large simulations would require far more virtual memory than is available on the largest hosts if Talisman did not use a sparse representation. The startup time to initialize simulation memory would also be enormous.

For example, an FFT of 32,768 points on 256 nodes, each with 1 MB of simulated physical memory, requires about 50 MB of decoded instruction memory out of a total process requirement of 260 MB. (In this example the working set is approximately 100 MB.) If Talisman did not allocate memory on demand, the FFT would consume 1342 MB for decoded instruction memory alone. We estimate that the steady-state penalty for the sparse representation is well less than one percent in both execution time and memory consumption.

Though the lazy allocation conserves memory, the space taken for decoded instructions can limit the simulator's use. We were not so limited in our work, but it is easy to imagine programs that, when run on hundreds of processors, would take a gigabyte of decoded instruction memory. There are several straightforward solutions to this problem: (1) allocate smaller decoded instruction pages to reduce internal fragmentation, (2) reclaim decoded instruction pages that have not been recently used, (3) share decoded instruction pages between processors. The first solution is straightforward and will only increase the page-crossing cost. The second solution could be done in conjunction with full modelling of the i-cache and will probably result in only a small decrease in performance

The third solution is possible, and is used by Magnusson [17], but has a number of problems with instruction cache modelling, debugger breakpoints, register access, and support for multiple program workloads

4.5 Modelling Basic Instruction Execution Time

To model the number of cycles an instruction takes, each simulated processor has an associated current cycle count. This variable is incremented by every handler to reflect the number of cycles it takes to issue the instruction.

The MC88100 has a register scoreboard; it contains one bit for each general purpose register. When multicycle instructions issue, they set the scoreboard bit corresponding to the instruction's destination registers. When multicycle instructions finish, their destination register scoreboard bits are reset. Issued instructions stall the processor until the operand register's scoreboard bits are clear. The scoreboard mechanism ensures correct operation of instructions that use values produced by multicycle instructions.

Talisman models the scoreboard with an array of *time-available* values that correspond to general registers (see Figure 4). Every multicycle instruction handler sets the time-available slot corresponding to its destination register to the cycle count value when the register in the hardware would be available. The handler then produces the result and stores it in the destination register. Instructions that read registers advance the processor cycle count to the maximum of the time-available slots corresponding to each operand register and the current cycle count.

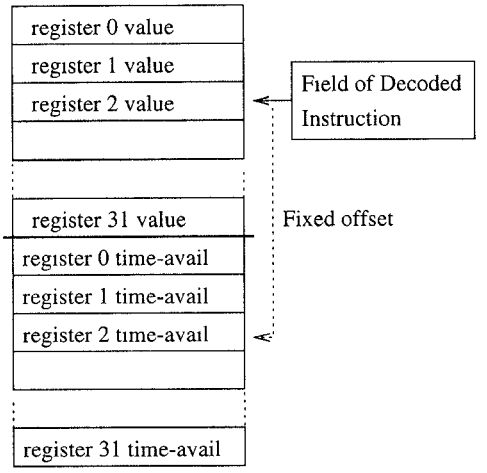

Figure 4: Relationship between General Registers and Time-Available Array

For example, if an `addu` instruction reads registers three and four and takes one cycle to issue, and if registers three and four have time-available counts of 105 and 107 respectively, and if the cycle count before the `addu` issue is 100, then the processor cycle count will be set to 107 by the `addu` handler. In this case, the one-cycle issue time and the time to wait for register three are hidden by the time to wait for register four, as in the real processor.

The scoreboard model depends on being able to calculate the time-available for all multicycle instructions at the time the instructions issue. This was possible in Talisman, and we believe that it will be so for most similar systems.

Instruction handlers access the time-available slots using the same pointers they use to access the register values. Handlers access the time-available value by adding a constant to the register pointer[2]

The timing of the data cache is modelled by keeping track of what the tag state of a real MC88200 cache [21] would be. Talisman models the MC88200's Least Recently Used (LRU) behavior by keeping the cycle count of the most recent access to each cache line. While this takes more storage than the LRU bit scheme that the hardware uses, it is simpler to understand and faster to execute. The data cache model calculates the time-available value for load instructions and puts this value in the destination register's corresponding time-available slot.

The MC88100 processor has a three-slot pipeline in the Data Memory Unit (DMU) through which all memory access instructions must flow after they are issued (see Figure 5). Each load or store of a word or less uses one DMU slot; double-word loads and stores use two slots. When the pipeline is full, a memory instruction attempting to issue will stall until the slots it requires become available. When the request in slot zero is satisfied, the pipeline shifts the contents of slot one to slot zero and slot two to slot one. After the shift, slot two is free to accept a new request.

Talisman keeps a three-element array of time-available values to model when the reference in each slot of a real MC88100 DMU pipeline would be available. Each DMU slot used advances the current processor cycle to the maximum of the current cycle count and the time-available value in the oldest slot (slot 0). The array is shifted down to eliminate the oldest slot and to make available the slot at the other end of the array (slot two). Slot two is then filled with the time at which the reference in a real MC88100 would vacate the DMU.

4.6 Processor Switching

Talisman simulates a multiple-processor system on a single-processor host by executing a few cycles of each simulated processor before switching to the next processor. The default number of cycles per switch is 10. The user can change this number. Each instruction handler checks to see if the processor cycle quantum has expired. If it has, the handler branches to code that finds another processor to simulate and then dispatches the next instruction for the new processor. Because each instruction handler runs to completion, processor switching is done between instructions only A processor can thus take more time than the quantum provides.

Talisman chooses processors so as to minimize skew between any pair of processors. It examines a circular list of running processors in a round-robin order and picks the first one whose cycle count is below the current system cycle-count threshold. It increases the threshold only when all processors have executed beyond it.

To keep the cost of switching low, only four key processor-dependent interpreter variables are kept in host registers: DECIP, IP, a pointer to the current processor's state structure, and the processor current cycle[3].

There is one instruction that can take a long time to execute: a store of a **cache-copyback** command to the control register of a data cache that is full of dirty data will take about 10,000 cycles. The processor executing this instruction will jump far ahead in simulated time and will not execute its next instruction until the other processors have caught up. Instructions are not interruptible on the simulator or the hardware, and the **cache-copyback** command writes no data that is visible to the program. This means that

[2]A subtle implication of this method is that the literal pools must have dummy time-available slots that are in the same relationship to the literal values as the real time-available slots are to the register values instruction handlers dereference operand pointers the same way for both register and immediate operands.

[3]Talisman's predecessor kept just the DECIP in a host register and calculated IP when its value was needed [1] The predecessor was written for a host with few registers, and it therefore made sense to calculate the IP from the DECIP Doing so in the current version would complicate the threaded-code interpreter, and Talisman's host has enough registers for both DECIP and IP variables

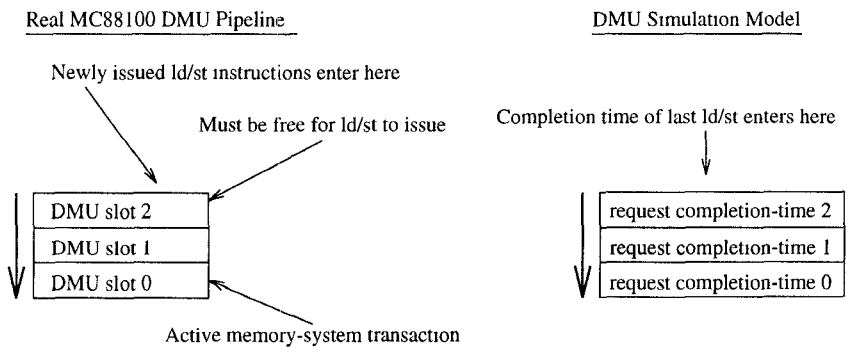

Figure 5: Real MC88100 DMU and Simulation Model

the simulator's semantic and timing behavior is the same as the hardware's. Long-running instructions execute frequently in the benchmarks discussed in Section 6.

4.7 Modelling Data Memory Access

While most instruction handlers are in a single large function, memory access instruction handlers are complex and so call a separate function for most of their semantic effect. The memory access function first translates the data's virtual address by calling the TLB model. The TLB model simulates the Motorola MC88200's 56-entry TLB. TLB misses delay the memory access, as they do in the hardware.

The physical address computed by the TLB model is checked to see if it is an I/O address or a memory address. If it is the former, the I/O module is called. Otherwise, the physical memory map is accessed to find the host memory that models the addressed simulated memory. Host memory is allocated if it is not yet in the map. The memory operation is performed, and the data cache state is updated using the physical address returned by the TLB model.

Unlike the real cache, the data cache model does not contain the data itself. It contains tags, LRU information, and state bits. The lack of data storage in the cache means that memory coherence errors can be latent on Talisman that are manifest on the hardware, and vice versa. In other words, an artifact of the data cache model is that memory is always coherent. If the operating system does not flush the data cache when it should, the error will not be seen on Talisman, though it may be seen on the hardware. Or, a bug may occur running on Talisman that is the result of cached memory being overwritten. Such a bug may be latent on the hardware, because the hardware processor may read the correct value from cache and not see the erroneous value in memory.

For our purposes, this difference in memory coherency was a small price to pay for a simpler cache model and smaller cache state. In addition, it causes no timing inaccuracies. Magnusson and Werner describe an alternative approach to modelling memory systems in their paper on SimICS [16].

4.8 Modelling Instruction Memory Access

Talisman models instruction cache cold misses, but not capacity misses. It models cold misses by distinguishing between translated and untranslated instructions. When a decode-me pseudo-instruction is encountered:

1. The processor's cycle count is advanced to reflect the time to fetch an instruction cache line.
2. All four MC88100 instructions in the cache-line that corresponds to the executed decode-me pseudo-instruction are translated to decoded instructions.

This method for modelling instruction cache cold misses requires no extra work in the case of a decoded instruction cache hit and little extra work in the case of a miss. However, it simulates only cold misses: while the real instruction cache is finite (4096 instructions), the decoded instruction cache is unlimited. A straightforward extension would model capacity misses by invalidating a cache-line of decoded instructions when a real instruction cache would replace a valid cache line. We considered implementing this extension, but felt it was unnecessary because the benchmarks fit in the instruction cache and experience few capacity misses.

4.9 Modelling I/O

When the load/store function encounters an address outside the range of physical memory, it calls the I/O module with this address and a pointer to the decoded load/store instruction. The I/O module searches a table that relates address ranges to particular I/O model functions. The table lookup corresponds to the address decoders found in hardware between the processor address bus and the I/O device select signals [4].

The I/O models include Meerkat's 32-bit cycle counter, interrupt controller status and control registers, internode status and control registers, internode DMA controller,
MC88200 [21] (cache and memory management unit) control pages, and some pseudo-devices. Pseudo-devices models are accessed by the simulated program just as are other device models, but pseudo-devices do not correspond to any real devices. Pseudo-device models allow the simulated program control over execution statistics collection. For example, one such model allows a program to turn execution profiling on and off.

4.10 Modelling Supervisor Mode

Many simulators, especially fast simulators, model only user mode. This limitation greatly reduces the value of such simulators in evaluating architectural tradeoffs because operating system performance and operating system interactions are significant factors in determining overall system performance.

Talisman supports both user and supervisor modes by allowing instructions to be discovered at run-time, and by modelling the semantics of virtual-to-physical address translation, cache and TLB manipulation operations, synchronous and asynchronous exceptions, the trap-time registers that support exception processing, various I/O devices, and supervisor-only instructions. Operating systems typically reveal code at run-time and so cannot run on

[4] Talisman's predecessor used a hashing scheme to speed the lookup [1] However, Talisman has a small table of devices, and the lookup time is not significant

simulators that depend on processing instructions prior to simulation [19]. For example, a simulator that has to preprocess all instructions before the simulation is started cannot deal with an operating system that loads program text from a simulated disk. In addition, some operating systems generate code on the fly in response to user requests [18]

The execution cost of supervisor modelling is mostly seen in the memory access instructions, which must consult the TLB. To mitigate this cost, we carefully coded the TLB model to be efficient. We estimate that overall execution performance is reduced by less than ten percent due to supervisor-mode modelling.

Supervisor mode modelling adds significant complexity to the simulator, especially in the area of exceptions and device models. However, this complex code was both tractable and necessary. All of the programs we ran made extensive use of supervisor mode.

4.11 Modelling Control Transfer Instructions

The handlers for control transfer instructions update both the IP and the DECIP. Updating the IP is easy: the new value is computed by adding a branch displacement to the IP's value or by taking a target register value. Updating the DECIP is a little more complicated.

A branch instruction to a target that is on the same page as the branch itself is called *on-page* and is translated to a decoded form that contains a pointer to the decoded form of the target of the branch. On-page branches are fast: they need only to dereference a pointer. Off-page branches are not as efficient, as the decoded target pointer must be resolved during execution.

The execution-time translation of an instruction address into a decoded instruction pointer is necessary because the instruction translator makes no assumptions about the contents or allocation of other decoded instruction pages. That is, the only pointers to the decoded instructions within a given page are contained in that same page. Talisman does not allow cross-page pointers because decoded instruction pages correspond to physical pages while branch semantics are defined in terms of virtual addresses. The correspondence between virtual target instruction addresses and decoded instruction slots is dynamic, i e., it can change after threaded code is generated, and there can be multiple virtual mappings of a single physical page. By resolving cross-page branches dynamically, these semantics are preserved.

Delayed branches increment both the DECIP and the IP, just as non-branching instructions do. In addition, they set a flag if the delayed branch is taken (i.e., the branch condition is true). Non-branching instruction handlers check this flag, which, when true, indicates that the previous instruction was a taken delayed branch. In this case control is passed to the target of the delayed branch after the non-branching handler finishes its semantic actions. Branching instructions need not verify that they are not in a branch delay slot, because the MC88100 specification prohibits control-transfer instructions from being in delay slots.

Jump instructions transfer control to locations whose targets come from a register and are thus not known until execution For this reason, a jump is treated like an off-page branch. Its execution requires translating a virtual target code address into a pointer to a decoded instruction pointer.

A small cache of translations from target virtual instruction addresses to decoded instruction pointers speeds the interpretation of jump and off-page branch instructions. This cache is flushed whenever the virtual-to-physical address mapping changes.

5 Talisman's Performance

Talisman's performance is a function of workload and the processor switch time All tests reported here use a processor switch time of 10 cycles. We found the difference in simulator accuracy between switching every cycle and switching every 10 cycles to be insignificant (our notion of "accuracy" is discussed in the next section). At this setting, the performance ranged from 500,000 to 750,000 simulated processor cycles per second on a SPARC-10/30 host[5]. Thus, on average, Talisman simulates one Meerkat processor cycle in 54 to 72 SPARC-10 cycles. While some workloads could cause much lower or higher performance, all of the tests we used were in this range. The programs we used to measure simulator performance are the same as those used to test accuracy (see Section 6).

The SPARC-10 host can do more per cycle than the processor we model. Therefore, the figure of 54 to 72 SPARC-10 cycles per Meerkat processor cycle must be adjusted to make a fair estimate of Talisman's slow-down. We estimate that the SPARC-10 has half the Clocks Per Instruction (CPI) of the Meerkat processors. This means that there is roughly a slow-down of 100 to 150 per simulated processor.

The system slow-down is the ratio of the number of host cycles it takes to simulate one cycle of a whole Meerkat. Simulating one cycle of a multi-node Meerkat requires simulating one cycle of each processor and the interconnect. To calculate this figure, we multiply the per-processor slow-down by the number of simulated processors. Thus, a simulated 256-node Meerkat has a slow-down of 27,000 - 37,000 to one. Because our host is about four times faster than the Meerkat processor, however, the ratio between wall-clock time and simulated time is not this large For example, a 32k FFT simulation runs for 850,000 cycles, or 42 milliseconds This takes six minutes of SPARC-10 time, which means that the ratio of SPARC-10 time to Meerkat time is 8,600 to 1.

6 Timing Models and Simulator Accuracy

We used a suite of tests both to guide development of timing models and to evaluate Talisman's overall accuracy. This section addresses these two issues.

6.1 Timing Model Development

Our method of timing model development was to iteratively:

1. Measure the difference between execution times of a moderate or complex benchmark program on Talisman and the prototype.
2. Identify which aspect of the prototype's timing was most responsible for the difference.
3. Write a low-level test that is more sensitive than the benchmark to the aspect identified in Step 2.
4. Verify that the low-level test shows a significant performance difference between Talisman and the prototype.
5. Add a timing model to Talisman that captures behavior identified in Step 2. The new timing model often has parameters that the user can adjust. Pick default values for these parameters.
6. Rerun the low-level test to verify that the new timing model makes Talisman accurate on the low-level test This may require adjusting the model's parameters. If not, examine the test and Talisman on a cycle-by-cycle level. Fix the model.
7. Rerun the higher-level benchmark to see if the aspect identified in Step 2 was correct. If not, repeat. If so, check the accuracy of Talisman on another benchmark test.
8. Stop when Talisman is accurate for all the tests.

[5]The host workstation uses a Super-SPARC processor clocked at 36 MHz, and has a SPECint92 rating of 45 2.

Test name	Hardware (μ-seconds)	Talisman (μ-seconds)	Difference	
			μ-seconds	Percent
loop with no mem activ	0.301	0.301	0.000	0.0
FP add	**0.704**	**0.754**	**0.050**	**6.6**
FP multiply	0.857	0.855	-0.002	-0.2
FP divide	3.404	3.404	0.000	0.0
FP mem add/multiply	1.711	1.710	-0.001	-0.1
cache read hit	0.501	0.501	0.000	0.0
double cache read hit	0.551	0.551	0.000	0.0
cache write hit	0.350	0.351	0.001	0.3
double cache write hit	0.401	0.401	0.000	0.0
uncached read	0.867	0.862	-0.005	-0.6
double uncached read	1.336	1.323	-0.013	-1.0
uncached write	0.360	0.357	-0.003	-0.8
double uncached write	0.718	0.714	-0.004	-0.6
I/O read	0.851	0.851	0.000	0.0
I/O write	0.452	0.449	-0.003	-0.7
CMMU control page read	0.701	0.701	0.000	0.0
CMMU control page write	0.301	0.301	0.000	0.0

Table 1: Low-Level Talisman Accuracy Test Results

Test name	Hardware (μ-seconds)	Talisman (μ-seconds)	Difference	
			μ-seconds	Percent
cache read misses	1.074	1.079	0.005	0.5
double cache read misses	1.132	1.125	-0.007	-0.6
cache write misses	0.926	0.927	0.001	0.1
double cache write misses	0.975	0.977	0.002	0.2
full write pipeline	0.923	0.931	0.008	0.9
instruction cache line fill	0.782	0.781	-0.001	-0.1
copy uncached to cached	0.938	0.936	-0.002	-0.2
exchange memory instruction	0.971	0.974	0.003	0.3
bus contention 3 reads	1.381	1.391	0.010	0.7
write back on write miss	1.286	1.291	0.005	0.4
write back on read miss	1.427	1.430	0.003	0.2
invalidate empty cache line	0.502	0.505	0.003	0.6
copyback full cache line	**0.925**	**0.860**	**-0.065**	**-7.6**
invalidate empty page	15.868	15.784	-0.084	-0.5
copyback half full page	75.060	75.008	-0.052	-0.1
copyback full page	134.224	134.252	0.028	0.0
copyback empty data cache	54.800	54.640	-0.160	-0.3
copyback full data cache	528.800	530.360	1.560	0.3
copyback+invalid whole cache	529.800	530.720	0.920	0.2

Table 2: Medium-Level Talisman Accuracy Test Results (Uniprocessor)

6.2 Timing Accuracy Tests

Table 1 shows the results of low-level tests of individual floating point and memory access instructions. The times reported are in microseconds and are an average of the time to execute a single iteration of the loop containing the measured instruction. The largest error is in the floating point add instruction test. Talisman overstates the cost of this instruction by one cycle, or 50 nanoseconds. Talisman does not model contention for the single write-port of the MC88100's register file. The MC88100 is capable of writing one word per cycle to its register file. Talisman makes a pessimistic guess as to whether contention will occur. In the case of the floating point add below, this guess is incorrect. We considered adding a timing model to correct this, but decided that the increased accuracy would not compensate for the effort, simulator performance degradation, and added simulator complexity.

Uncached read and write tests exercise the DRAM system. Talisman models interference with DRAM refresh, which consumes about two percent of the memory system's bandwidth. DRAM refresh is rarely modelled in system simulations, because it is considered such a small factor. Note, however, that if Talisman did not model refresh, the errors would be several times larger. Many of the errors are below one percent. These errors may be due to slight timing differences between the hardware and Talisman. These differences can cause Talisman to model one more, or one less, refresh cycle than occurs on the hardware. In fact, the hardware measurements show variation from run to run of about one percent.

Table 2 shows the results of more complex operations. These include various cache operations, synchronization instructions, and sequences that load the local memory bus. The largest error is shown in the "copyback full line" test, where Talisman understates the time for a copyback of a cache line by a little over one clock cycle.

We ran several parallel applications on four-node real and simulated Meerkat's. Table 3 shows the correspondence of Talisman and hardware results for a global combine, SOR, and FFT (see [2] for a description of these codes). The largest error is 7.8% on a 32-byte global combine.

Errors on the complex tests are larger than on the simple tests. This is a product of our method for achieving timing accuracy. The simple tests were created to test modeling features of the simulator, so naturally the simulator performs well on them. The complex tests, on the hand, were chosen to gauge the overall accuracy of the simulator. If we kept going with our methodology, we would identify the root of the difference on the 32-byte global combine, write a simple test to exercise this difference, add a model to bring the simulator closer to the hardware, etc.

6.3 Measurement Experience

In some cases we changed our run-time system to make the simulated and prototype execution times closer. For example, we initially used processor 1 on every node to process internode interrupts, while processor 0 did everything else[6]. On a message-exchange test, the simulator reported half the execution time of the hardware: after processing interrupts, processor 1 had dirty cache state that had to be flushed to memory and then reloaded by processor 0. The simulator did not model the MC88200's snooping and associated cache operations, because we were not interested in the performance of programs that used multiple processors per node. We changed the run-time system to use processor 0 for everything. This improved the performance of the prototype on small messages and brought the simulator and prototype execution times in line.

[6]The Meerkat hardware has four processors per node All of the tests used in this paper ran with one processor per node enabled

Some programs have different execution results on the simulator and the prototype: there are aspects of the prototype's timing that we could not - or did not want to - model. We did not model the timing of operations that we believed would not affect the outcome of our measurements and were difficult to model. For example, the prototype runs a debugging monitor that mediates between the cross debugger running on the Sparcstation host and the running Meerkat program. This monitor: (1) lets the cross debugger control the running Meerkat program, and (2) fields requests from the Meerkat program for operating system services performed on the host. The semantic effect of the monitor is modelled in the simulator, but not its timing or its effect on instruction and data caches. The time it takes the prototype to perform operating system services is a function of the load on the Sparcstation host, the relationship between the time of the request and the host's process interval timer, and other factors.

To enable accurate measurements, we were careful to measure only the execution time of sections of code that do no I/O through the monitor and whose initial cache state is not dependent on previous calls to the monitor. In practice, we found this easy to do, and it usually meant avoiding **printf**'s until after a measurement was taken.

6.4 Validity of Large-System Simulations

The hardware prototype has four nodes, but we use Talisman to evaluate systems with up to 256 nodes. We cannot directly verify the simulator's accuracy with systems with more than four nodes, as we do with systems with four or fewer nodes. However, we have good reason to believe that the large-system simulations are correct. First, we know that Talisman accurately models the nodes of these large systems. Second, the interconnect model is fairly detailed and keeps track of each node's use of the interconnect on a cycle basis. The combination of accurate node models and a detailed interconnect model should yield an accurate system model. Third, even the four-node system running several of the benchmarks listed in Table 3 saturate the interconnect at times. If there were errors in the model, one would expect to see a significant difference between the benchmark running times reported by the hardware and Talisman. The lack of significant differences, the low-level of detail with which we model the interconnect, and the high accuracy on small systems gives us good confidence that Talisman is accurate for large systems.

7 Debugging Features

There are a number of benefits of using simulators over hardware, e.g., adaptability, deterministic execution, low cost. In addition to the benefits that most simulators provide, Talisman has a number of features that gave insight into Meerkat behavior and aided in program development:

- Unlike the hardware, Talisman allows single-stepping and breakpointing of exception handlers. Large portions of the message-passing system execute as an exception handler, making it much easier to debug this code in the simulation environment.
- Talisman has a precise memory breakpoint feature that stops execution of all processors the instant the watched location is accessed. This precise stopping of execution aids certain debugging problems tremendously. Building as precise a breakpoint into the prototype would require redesigning the processor.
- Talisman compiles conditional breakpoint expressions into target machine code (see [2] for details) to speed conditional

Test name	Hardware (μ-seconds)	Talisman (μ-seconds)	Difference	
			μ-seconds	Percent
Global Sync Average	94.558	90.142	-4.416	-4.9
Global Combine 8 bytes	130.200	127.400	-2.800	-2.2
Global Combine 16 bytes	94.400	95.800	1.400	1.5
Global Combine 32 bytes	**97.400**	**105.600**	**8.200**	**7.8**
Global Combine 64 bytes	128.200	124.000	-4.200	-3.4
Global Combine 128 bytes	160.000	150.200	-9.800	-6.5
Global Combine 256 bytes	226.200	224.200	-2.000	-0.9
Global Combine 512 bytes	381.600	368.800	-12.800	-3.5
Global Combine 1024 bytes	635.800	614.400	-21.400	-3.5
Global Combine 2048 bytes	887.000	885.400	-1.600	-0.2
Global Combine 4096 bytes	1697.800	1698.000	0.200	0.0
Global Combine 8192 bytes	3323.600	3498.000	174.400	5.0
Global Combine 16384 bytes	8292.200	8234.400	-57.800	-0.7
Global Combine 32768 bytes	15671.200	15223.200	-448.000	-2.9
R/B SOR 32x32	30252.000	29379.000	-873.000	-3.0
R/B SOR 32x32	59248.800	58650.600	-598.200	-1.0
FFT 16 points	2266.200	2300.200	34.000	1.5
FFT 512 points	5152.000	4883.800	-268.200	-5.5
FFT 1024 points	10136.000	9658.400	-477.600	-4.9
FFT 2048 points	21088.600	19987.600	-1101.000	-5.5
FFT 4096 points	46496.400	43895.000	-2601.400	-5.9

Table 3: High-Level Talisman Accuracy Test Results (4-Node)

breakpoint evaluation by several orders of magnitude. The user can thus make liberal use of conditional breakpoints without having to wait long periods for frequently-false conditions to be evaluated.

- Talisman optionally animates internode bus activity in one X window and displays an internode bus contention histogram in another. Both of these displays gave valuable insight into system behavior, were easy to add to Talisman, and would be difficult to implement in a hardware prototype.

8 Summary

Simulators vary widely in their application, structure, accuracy, and performance. We outlined several simulator applications and the simulators typically used. Talisman is unique in its combination of speed, efficient use of memory, fine modelling detail, portability, user/supervisor modelling, and modelling of address translation. With this combination of features we were able to model large systems at a fine level of detail and make accurate predictions about the performance of large systems.

We described Talisman after placing it in the context of a range of simulation strategies. Talisman can execute instructions quickly, yet models timing accurately, and can efficiently multiplex amongst many simulated processors.

Talisman's construction began with a fast threaded-code interpreter and a translator to generate threaded-code from machine instructions. To make Talisman usable, we chose a powerful symbolic debugger for the front end. We achieved timing accuracy by carefully adding functional models to the behavioral simulator base. These functional models slowed Talisman, but because we added only those functional models that substantially affected accuracy, the degradation was less than an order of magnitude.

Talisman executes about 100 host instructions per simulated instruction. This is much faster than other timing-accurate simulators. The high performance allows users to model multicomputers with hundreds of processors running substantial programs.

A number of performance monitors and animation features give the user a comprehensive view of the simulated system. These features were easy to add to Talisman, but would be difficult or impossible to add to the prototype. They are nonintrusive, i.e., their presence does not affect the execution behavior of the simulated system.

We showed the results of running a suite of tests on both the Meerkat prototype and Talisman. These tests show that Talisman is a faithful model of the prototype, usually differing from the prototype by only a few percent.

9 Acknowledgements

I owe thanks to many people for their advice on earlier drafts of this paper and for their encouragement, including David Keppel, Ed Lazowska, Hank Levy, Peter Magnusson, Neil McKenzie, and Dylan McNamee. Sandy Kaplan corrected the English in the doctoral dissertation from which this paper is derived; remaining errors are mine. Several referees gave useful advice. This work was done at the University of Washington and was supported in part by the National Science Foundation (grants no CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer.

References

[1] Robert C. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Conference*, pages 53–63, January 1990.

[2] Robert C. Bedichek. *The Meerkat Multicomputer: Trade-offs in Multicomputer Design*. PhD thesis, University of Washington, August 1994. Department of Computer Science technical report 94-06-06

[3] James R. Bell. Threaded code. *Communications of the ACM (CACM)*, 16(2):370–372, June 1973.

[4] Eric A. Brewer, Chrysanthos N. Dellacrocas, Adrian Colbrook, and William E Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[5] Eric A Brewer and William E Weihl Developing parallel applications using high-performance simulation. *ACM/ONR Workshop on Parallel and Dist Debugging ACM SIGPLAN Notices*, 28(12):158–168, December 1993.

[6] J. Bradley Chen and Brian N Bershad. The impact of operating system performance on memory system performance *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 120–133, December 1993

[7] D. W. Clark. Pipelining and performance in the VAX-8800 processor. *Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[8] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Modeling and Measurement of Computer Systems*, May 1994.

[9] R C Covington, S Madala, V Mehta, J.R. Jump, and J.B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM SIGMETRICS and PERFORMANCE Conference*, pages 4–11, May 1988.

[10] Peter Deutsch and Alan M Schiffman. Efficient implementation of the Smalltalk-80 system *11th Annual Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[11] David R. Ditzel, Hubert R. MeLellan, and Alan D. Berenbaum The hardware architecture of the CRISP microprocessor In *Proceedings of the 14th Annual International Symposium on Computer Architecture; Computer Architecture News*, pages 309–319, June 1987

[12] R. M. Fujimoto and W. B. Campbell. Efficient Instruction Level Simulation of Computers. *Transactions of the Society for Computer Simulation*, 5(2):109–124, Apr. 1988.

[13] Stephen R. Goldschmidt. *Simualtion of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, June 1993.

[14] T G Lang, J.T. O'Quin, and R O Simpson. Threaded code interpreter for object code *IBM Technical Disclosure Bulletin*, pages 4238–4241, March 1986

[15] Peter Magnusson Efficient simulation of parallel hardware Masters thesis. Royal Institute of Technology (KTH), Stockholm, Sweden, 1992

[16] Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[17] Peter S. Magnusson. A design for efficient simulation of a multiprocessor *MASCOTS '93 – Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, January 1993.

[18] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the twelth ACM Symposium on operating system principles*, pages 191–201, December 1989.

[19] Cathy May Mimic A fast S/370 simulator. In *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices*, volume 22, pages 1–13, St. Paul, Minnesota, June 1987

[20] *MC88100 RISC Microprocessor User's Manual.* Motorola Corporation, 2900 South Diablo Way, Tempe, Arizona.

[21] *MC88200 Cache/Memory Management User's Manual.* Motorola Corporation, 2900 South Diablo Way, Tempe, Arizona.

[22] Carl Ponder Personal communication, February 1994

[23] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel Virtual prototyping of parallel computers. *Performance Evaluation Review*, 21(1):48–60, May 1993.

[24] Mendel Rosenblum and Emmett Witchel. SimOS. A Platform for Complete Workload Studies. Personal Communication (to be published), 1995

[25] Margo Selzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, January 1990.

[26] Richard L. Sites, Anton Chernoff, Mathew B. Kerk, Maurice P Marks, and Scott G. Robinson Binary translation. *Communications of the ACM*, pages 69–81, February 1993.

[27] Insignia Solutions. *SoftPC Product Information*, 1991.

[28] Richard M. Stallman and Roland H. Pesch *Using GDB: The GNU Source-Level Debugger.* Free Software Foundation, 545 Tech Square, Cambridge, Ma. 02139, March 1992