Efficient Memory Simulation in SimICS

Peter Magnusson and Bengt Werner
{psm,werner}@sics.se
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

Abstract

We describe novel techniques used for efficient simulation of memory in SimICS, *an instruction level simulator developed at SICS. The design has focused on efficiently supporting the simulation of multiprocessors, analyzing complex memory hierarchies and running large binaries with a mixture of system-level and user-level code.*

A software caching mechanism (the Simulator Translation Cache, STC) improves the performance of interpreted memory operations by reducing the number of calls to complex memory simulation code. Major data structures are allocated lazily to reduce the size of the simulator process. A well-defined internal interface to generic memory simulation simplifies user extensions. Leveraging on a flexible interpreter based on threaded code allows runtime selection of statistics gathering, memory profiling, and cache simulation with low overhead.

The result is a memory simulation scheme that supports a range of features for use in computer architecture research, program profiling, and debugging.

Keywords: interpreter, simulator, multiprocessor, SimICS, memory simulation, memory hierarchy, cache simulation

1 Introduction

A computer program can execute in several environments. Generally, the program was written with a particular compiler, operating system, and hardware in mind. This *target environment* will then typically be the most efficient environment for that program. Most of computer science revolves around improving existing target environments or components thereof.

Most target environments divide the execution of a program into two distinct phases. The first, compilation, generates an *object binary*. In the second phase, the resulting binary is executed directly on an operating system and hardware platform, but is not transformed any further. The intention of this division of labor is to perform the cumbersome task of determining the intention of the program only once, thus allowing the more frequent operation of direct execution to be as efficient as possible. Let us call the platform of the second phase the *execution environment*.

It is common to insert a layer between the binary and the execution environment. Most of these layers involve some form of transformation of the program, which may or may not be visible to the user. Common reasons for doing this are to profile the binary to determine where to make improvements in the program [18], mimicking older execution environments to protect software investments [2, 3, 27, 36], controlling the execution in order to debug the program [39], transforming the program to improve run-time performance [21, 28], and looking for common source code bugs by monitoring the execution [20].

1.1 Execution Analysis

A recurring theme in computer science is the need to understand what real programs do. For example, new computer architectures are developed to allow high-performance implementations. There is a trove of statistics available to guide computer architects when they are deciding what to optimize. Sets of programs such as the SPECint92 [40] and Splash [35] benchmarks are common points of reference in the academic community, and many believe them to be representative of user workloads.

A representative program needs to be analyzed to understand what optimizations in an underlying architecture are globally applicable. On the other hand, if a program is non-representative, this needs to be determined, and furthermore decided whether to modify the program or specialize the hardware.

Another important example is studying the interaction between application, system software, and computer architecture. A growing body of research indicates that the large, complex systems in common use today are poorly understood [12, 19].

In both examples, we are often concerned with the execution of a binary. The reason is simple: in both cases we are interested in studying real programs, and the task becomes too large if we must write realistic applications or implement new compilers. Therefore, the starting point is a "real" user binary whose compilation may or may not have been modified by making small changes to the source code, libraries, compiler, or the binary itself.

This execution analysis has traditionally been done by running the program on top of a simulator.

1.2 Simulator Issues

Traditional use of simulation as an instrument have often suffered from the consequences of poor simulator design. If the simulator is slow or has a large memory overhead, only small programs ("toy benchmarks") can be studied. If the simulator fails to simulate system-level effects, the resulting statistics will be non-representative of real workloads. Among the more important system-level effects are those caused by page faults, interrupt-driven I/O, cache interference, and multiprogramming. The common reason for their omission is that they are difficult to support, especially in fast simulation techniques such as variations of direct execution.

We believe that once these design difficulties are dealt with, the resulting simulator will be both efficient and multi-purpose:

- *computer architecture investigations;* a common domain for simulators, the purpose here is to understand the frequency and character of hardware events triggered by software,
- *program profiling;* traditional techniques of detailed program behavior analysis are too invasive or inflexible for complex systems, such as real-time operating system kernels with extensive interaction among server components,
- *debugging;* simulators allow the debugging of programs that are otherwise difficult to deal with, such as system-level code; furthermore, the control over execution that a simulator can offer allows for new approaches to program debugging.

1.3 The Importance of Memory

Running system-level programs requires MMU simulation, and if the simulation of data caches is added then the simulation is required to perform several functions for every memory operation. For many applications of simulators, this simulation of memory is a significant if not dominant portion of the workload. Furthermore, studying different memory hierarchies requires end-users to add or modify code which is inserted onto the critical path of the simulation.

Previously, implementation of memory simulation algorithms has been done as one of many tasks in the process of developing a simulator. The contribution of this paper is the development of several techniques that together are both more flexible and more efficient than previously published algorithms. Furthermore, they are applicable to almost any simulation or analysis tool where simulation of memory operations is involved.

1.4 Objectives of SimICS

SimICS is an instruction-level simulator developed at SICS. SimICS has several objectives, in particular, SimICS should:

- be *fast* and *memory efficient* to allow for large programs, large working sets, and long execution runs,
- support complex *memory hierarchies* with minimal loss of performance,[1]
- simulate *multiple processors*, including interprocessor interrupts, message passing, and external TLB invalidations,
- run *system-level* code to allow studies of the interaction of operating system, user program, and architecture,
- not restrict programs to any particular *programming language, compiler, or library*,
- gather *statistics*, such as memory usage, frequency of important events, and instruction profiling. In addition, users should be able to develop their own *extensions* with minimal understanding of the core of the simulator,
- be able to run *interactively* and support *debugging primitives* to allow symbolic debuggers as front-ends,
- be *deterministic* to allow repeats of both statistics gathering and debugging,
- have a *low startup cost* and thus a short "edit-compile-simulate" cycle,
- be as *host-independent* as possible.

All of the above goals set constraints on the simulation of memory. To meet them, SimICS contains a combination of techniques. This has allowed us to write a single simulator that:

- supports multiple address spaces,
- simulates caches, including linking with a memory hierarchy simulator written by a user,
- deals correctly with supervisor and multiple user address spaces, handling page faults etc,
- is fast and has low memory overhead.

Thanks greatly to the modularized design, all of the features in the following list can be selected while SimICS is running. The section number where the feature is discussed in more detail is indicate in parenthesis.

- number of processors and number of nodes (2)
- choice of memory hierarchy (5.2)
- Unix emulation (6)
- shared physical address space among nodes, or distributed with separate physical memories for each node (3.5)
- choice of memory management unit (MMU) (3.4)
- profiling memory usage (5.3)
- simulating caches, including varying cache size and number of sets (5.1)

Thus, SimICS can simulate a four-processor architecture with a shared memory bus and 64 kB direct-mapped first level cache, or a 16-processor distributed memory architecture with message passing devices mapped into supervisor address space and 8 kB two-way associative cache, without recompiling. The performance impact of this flexibility is small compared to a specialized design.

[1] A memory hierarchy is a hierarchy of caches, possibly using different coherency schemes. In evaluating multiprocessor memory systems, it is often of interest to look at the frequency of different coherency protocol transactions.

2 Internal Structure of SimICS

In this section we give a brief description of the overall structure of SimICS. Figure 1 shows the principal objects of the simulator. The machine model is that of one or more nodes, each with one or more processors. Devices are unique to a node, and are memory-mapped. Generic data structures exist for nodes, processors, and devices. The processor and node objects are on linked lists. All nodes of the machine are on a single list, while the processors are on multiple lists: node list (processors on a node), machine list (all processors), and a circular scheduling list for round-robin scheduling to simulate concurrency.

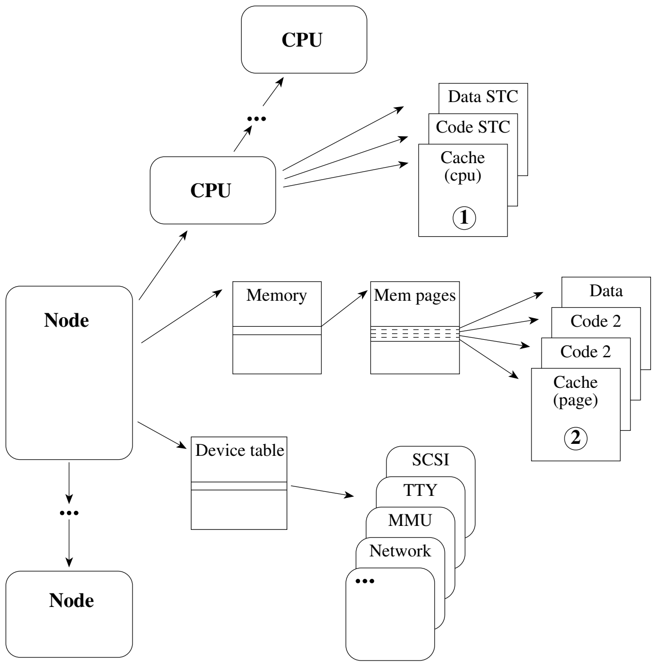

Figure 1: Principal SimICS data structures

The program code for SimICS is structured with separate file pairs (C file and header file) to delineate responsibility. We have strived for a strictly modular design. All data relating to common objects are isolated in a single structure, and generally only functions in a particular file pair can manipulate it:

- machine.[ch] describes general architecture dependencies, such as where the layout of target physical memory, the generic device table (see below), minimum and maximum number of processors and nodes.
- node.[ch] data structures and functions for nodes. A node contains one or more processors, physical memory, and a set of memory-mapped devices.
- processor.[ch] contains information that is common to any processor type, including data structures to support threaded code.
- local-processor.[ch] describes non-generic processor attributes, such as number of registers, functions for manipulating generic processor state such as interrupt enable/disable, and functions for accessing MMU.
- various device files, e.g. SCSI.[ch], SCC.[ch], etc. Each device type needs to define initialization routines. Thus, when a node is created (at runtime), the generic device table is copied and the listed initializers called. Generic device functions and data structures are kept in device.[ch].
- memory.[ch] implements generic memory simulation code.
- memory-hier.[ch] implements specific memory simulation code, i.e. a particular memory hierarchy.
- unix.[ch] support code for emulating a Unix (Solaris 2.x) host for running user binaries.

All objects are allocated dynamically. The user can interactively set number of processors and nodes, and re-initialize. Global functions such as `for_all_processors()`, `for_all_memory_pages()`, and `for_all_nodes()` exist to simplify flexible design.

The only limitation on the number of processors, nodes, size of application binary, or size of simulated memory is the available virtual address space that the host can comfortably support.

3 Memory Simulation in SimICS

To support system-level simulation, SimICS needs to faithfully simulate logical address spaces. Table 1 lists the operations that may be required for every memory access (including instruction fetches). Since memory accesses are common, the operations need to be dealt with efficiently.

1. calculate the logical address,
2. translate from a logical address to a physical,
3. check for TLB misses,
4. simulate hardware table walks,
5. check protection,
6. check for alignment violation,
7. perform the read/write operation, and
8. update processor state.

Table 1: Memory transaction tasks

The logical address is determined by the semantics of the particular instruction. Steps 2 through 4 then translates this to a physical address and locates the access rights. Simulating correct TLB contents are required to correctly interleave user and system code—TLB misses or protection violation generates page faults that cause a page fault handler to execute. On some processors TLB misses will result in an (attempted) hardware table walk, and we want to catch these memory references.

Misalignment refers to the inability of some processors to access data block boundaries. In the simple case, the address is rounded of. For example, the 88110 can be programmed to do this [29]). In the more complex case, the processor generates a trap if the access is misaligned. For example, the Alpha architecture relies on such traps to divide responsibility for non-word accesses between the compiler and the operating system [37]. SimICS needs to support this latter case since the Sparc always traps on misaligned accesses [38]. This case is more "complex" in the sense that for a simulator such checks are expensive.

In addition to the list in table 1, we want to implement watchpoints[2], profile memory usage (section 5.3), and simulate a memory hierarchy (section 5.1). The memory simulation in SimICS should therefore be flexible.

We now need to address the issue of how accurately the memory hierarchy should be simulated. There are at least four levels of increasing accuracy, see table 2.

1. memory contents of program (user-level)
2. address trace (system-level)
3. cache contents (off-chip memory accesses)
4. memory timing (exact performance)

Table 2: Memory simulation accuracy levels

The first level simply maintains a correct memory content from the perspective of the user-level program. The second level also simulates contents of TLBs, thus allowing table walks, page faults and operating system code to be interleaved with the program execution. The third level simulates first (and second) level cache contents, thus correctly modeling the address trace coming off the chip. This level allows us to simulate coherency actions in a shared-memory multiprocessor, which may be performed in hardware or with a combination of hardware and software. The fourth and final level simulates correct latency times for performing memory transactions.

SimICS supports the first three levels. The fourth level is currently not supported.

3.1 Memory Simulation Modules

We will use three terms for different memory spaces. A *logical address* is an address used by a program, sometimes called a virtual address. On the target architecture, this address would be translated by the memory management unit to the *physical address*, which in turn would be used to actually look up the data. In a simulator, there is a third level, since the simulator itself exists in a virtual-physical environment. The physical address needs to be translated to an address in the simulator's virtual address space, and this address we call the *real address*.

Figure 2 shows the principal components of memory simulation in SimICS. The interpreter executes the individual instructions. If the instruction is a memory operation, the interpreter attempts a translation using the simulator translation cache, STC (1). If successful, the STC will return the corresponding real address

[2]A watchpoint is a breakpoint on an address containing data. Execution stops on the instruction that reads or writes an address that the user has set a watch-point on.

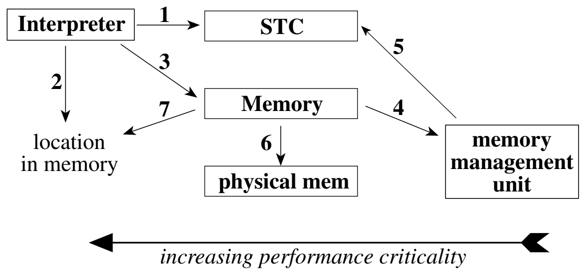

Figure 2: SimICS modules for simulating memory

(2). If STC fails, the interpreter delegates to the memory simulator (3). This module will first do a correct logical to physical translation via the full state of the MMU (4). If the MMU module doesn't want to see a future access to this page, it calls the STC module with the appropriate routine (5).[3] If the MMU detects a page fault, it will change the processor state to reflect this (not shown). Next, the physical memory module will translate the physical address to a real address (6), allocating new space if necessary. The memory module now performs the memory operation (7), and execution in the interpreter resumes. The next time this page is accessed the STC will succeed in step (1).

Note how in figure 2, the frequency of operations increases towards the right. Only every fourth or fifth instruction performs the STC lookup (1), and only misses in the STC cause the MMU to be called (4).

3.2 Memory Transactions

To simplify communication between the memory simulation modules, we have defined a generic memory transaction data type:

```
typedef struct memory_transaction {
   uint32             physical_address;
   uint32             logical_address;
   uint32             real_address;
   processor_t *      processor_ptr;
   read_or_write_t    read_or_write;
   processor_mode_t   mode;
   data_or_instr_t    data_or_instr;
   unsigned           snoop_bit:1;
   unsigned           cache_bit:1;
   unsigned           writethrough_bit:1;
} memory_transaction_t;
```

Any transaction that misses the STC causes a memory transaction object to be allocated by the memory module (in step 3 above). Only a single pointer is then passed between the modules until the operation is resolved. Not all information is valid at any given time—functions read and write to the structure as they see fit. For example, the MMU is expected to fill in the snoop, cache, and write-through bits so that any cache simulation code can determine whether cache is enabled for this particular memory operation. (This is discussed further in section 3.7.) This has

[3]The STC code will call the physical memory module to obtain the real address, but this is not shown.

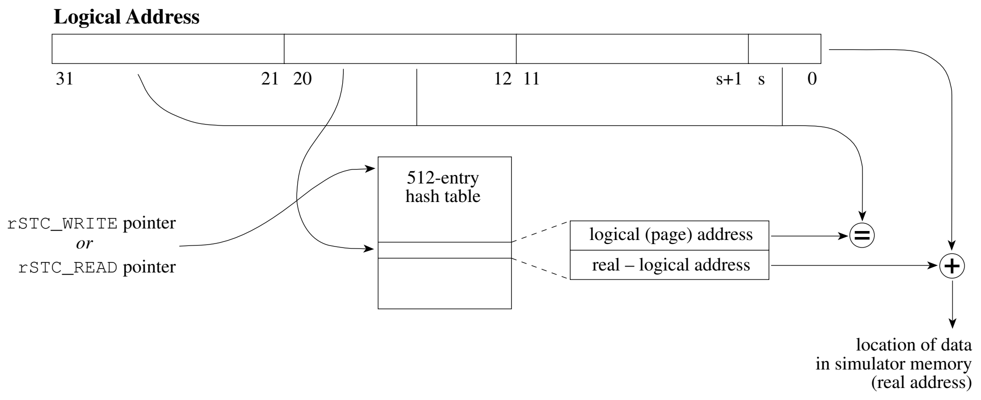

Figure 3: Simulator Translation Cache (STC)

to be programmed carefully, since there is no simple way to assert that data in the structure is valid. The advantage is design simplicity and efficiency.[4]

3.3 Intermediate Code Support

The core of SimICS is based on threaded code techniques [8]. We translate target object code to an internal format. This intermediate format is then interpreted. This translation need not be 1:1, and we use this to allow the memory simulation features described in this paper to be chosen interactively. With regards to memory simulation, there are separate sets of intermediate codes to support four combinations: functional, minimal statistics, memory profiling, and cache simulation.

The functional mode is optimized for speed, thus simulating only the functional correctness of the execution—this includes correct MMU simulation. The minimal statistics mode counts memory accesses according to user or supervisor space, and whether the access is a read or write. Memory profiling and cache simulation are described in more detail further on.

This structure allows a single simulator binary to support all of the features described in this paper. Furthermore, the individual features can be toggled during execution—the necessary internal data structures are allocated as necessary, while the execution state is unaffected.[5]

Details on the internals of SimICS are described elsewhere [23, 24, 25, 26, 34].

3.4 Memory Management Unit

The MMU module is well isolated from the other memory simulation components. The MMU needs to provide a `mmu_logical_to_physical()` routine to report on legal translations. Conversely, the MMU simulation code can call a routine to clear address intervals that are cached by the STC. This design allows new MMU designs to be implemented with only a cursory understanding of the rest of the simulator. The MMU simulation code need not be excessively efficient.

SimICS currently simulates the m88110 MMU and a pseudo-MMU for direct execution of user binaries. The m88110 MMU is programmed using control registers, which are special instructions on the 88k.

3.5 Physical Memory

The physical memory is simulated using a separate module that allocates space on a page-size basis upon the first memory access to that page. It supports sparse memory usage anywhere in the physical address space.[6] A single pointer associates a node with the (hierarchical) memory data structure—see figure 1. Multiple processors on a single node always share the same physical address space. When simulating a shared memory address space, the same data structures are used by all nodes. When simulating a distributed memory architecture, the memory is allocated separately. Thus, the decision by the user to simulate a shared memory or a distributed memory architecture can be decided interactively.

Physical address space on a real machine is seldom a continuous 4 GB. Therefore, the machine description file can define a macro `FIX_ADDRESS()` that is applied to all physical addresses.

3.6 Simulator Translation Cache (STC)

The STC caches legitimate address translations for quick access. Thus it contains a subset of the TLB entries. The STC translates directly from logical address to real address. Whenever there is a miss in the STC, it calls the `mmu_logical_to_physical()` routine described earlier. The MMU simulation code can, in turn, call the `mem_add_to_STC()` routine to tell the STC module to enable a particular translation. The intent is that any future accesses to this logical page should hit the STC. The STC code is complex and intimately tied to the simulator core, but this is hidden from the MMU simulation code. STC entries can likewise be invalidated by the MMU module.

For each processor, there are six separate STCs for each combination of read, write, or execute with supervisor or user.

[4] Procedure calls generally become faster since we reduce parameter copying.

[5] If this is done for cache simulation, the caches will be cold-started and will not yield correct statistics until they have been warmed up. Also, deallocation is not guaranteed so multiple mode switches will leak virtual memory.

[6] SimICS supports at most 32-bit physical address spaces.

The principal STC data structure and translation scheme is illustrated in figure 3. The input is a logical address provided by the instruction interpreter. The bottom bits of the page number (bits 12-20) is the index into a hash table. The tag is formed by the whole page number, in this case the top 20 bits. The bottom 12 bits of the tag are zero. For a 2^s byte memory object, the bottom (s+1) bits of the address are not cleared. A tag mismatch indicates either a translation miss or a misaligned address. Finally, the value stored in the hash table is the difference between the real and logical address. The real address can be formed by adding the logical address to this value.[7]

Note the rather large 512-entry hash table. Since the simulated TLB is generally fully associative or has a high associativity, we need a large direct-mapped STC to give comparable performance. We do not need to handle misses, so there is no alternate linked-list structure. This design handles items 2-7 in table 1.

The two pointers, `rSTC_WRITE` and `rSTC_READ`, point to the current STC table for memory accesses. They change value whenever the processor switches between user and supervisor mode, or when multiprocessor time slicing occurs (see section 7). Having separate tables for read and write operations allows us to support a range of features in MMU and/or cache designs, such as valid bit, dirty bit, and write protection. In SimICS, the two tables lie adjacent in memory (read table first) and its location is cached in a register during interpreter execution.

In the source code of the interpreter, the following C macro is used to perform memory operations: `MEMORY_OPERATION(DST, SRC, TYPECAST, MEM_OP_TYPE, SIZE)`. `DST` evaluates to an Lvalue to store the result in the case of loads, and the value to be written in the case of stores. `SRC` evaluates to the effective address. `TYPECAST` is the type conversion to be applied (signed byte, for instance). `MEM_OP_TYPE` identifies the type of operation: load, store, swap, etc. `SIZE` is the number of bytes of the operation.

The same macro is used by all memory instructions in both the Sparc and m88110 simulation code, and should be directly usable to simulate any similar RISC instruction set.

3.7 Simulating a Data Cache

We want to simulate the cache contents to gather performance statistics, and to support the simulation of memory hierarchies. Naturally, as soon as caches are simulated they need to be involved in every memory operation. More specifically, it adds a step between 6 and 6 in table 1.

Fortunately, cache line look-up and TLB look-up are often done in parallel on real hardware. The result is that the lower bits (those typically used for cache look-up) remain the same after the logical address has been translated to a physical one. We can therefore extend the STC to support cache lines, thus performing a cache and TLB look-up in one operation.

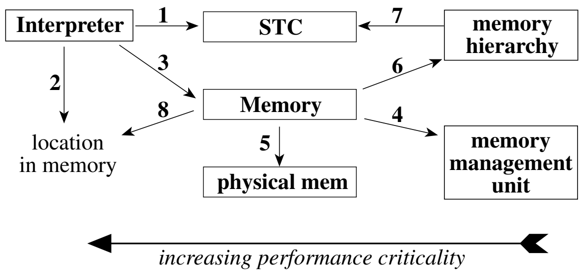

Figure 4: SimICS modules for cache simulation

Figure 4 illustrates memory simulation when cache simulation is enabled. The algorithm is analogous to figure 2, except that the memory module does not update the STC directly. Instead, a failed STC look-up is ultimately passed on to the memory hierarchy module (6). The memory hierarchy module simulates the cache in whatever manner it sees fit, gathers statistics, etc. It is up to the memory hierarchy module to update the STC (7). We return to the memory hierarchy in the next section.

The enhanced STC scheme for cache simulation is very similar to figure 3. We do the look-up as if the cache was a direct-mapped, 4 kB cache with 16-byte cache lines.[8] The hash table is 256 entries, rather than 512, and the the hash index is formed by bits 4-11 of the logical address, rather than bits 12-20.

We need to check the logical address and the alignment, as in the previous section. This does not restrict the memory hierarchy to a particular first-level cache size or organization, since cache lines are only added to the STC if told to by the memory hierarchy code.[9] Note also that cache lines should only be placed in this STC if accesses to them do not affect any cache simulation state. For example, if the replacement scheme of the simulated cache is LRU, then only MRU lines can be put in the STC.[10] If the replacement is random, then any line can be in the STC.

SimICS will not track STC hits other than to count them in broad categories: supervisor read hits, supervisor write hits, user read hits, user write hits.

Note that the this design could support either simulating full cache contents or just the cache logic (i.e. tags, state bits, and memory references generated by misses). The implementation in SimICS only supports the latter.

3.8 STC Efficiency

To illustrate the efficiency of the STC design, consider the following Sparc assembler. This is the code that an optimizing C compiler should generate for an STC look-up for a load-word instruction (Sparc assembler):

[7]This works due to the following observation: let P_l be the starting address of the simulated logical page containing the logical address Q_l to be translated. Let P_r be the starting address in the host address space (real address) of the corresponding page. Then $(P_r - P_l) + Q_l = Q_r$, where Q_r is the real address of Q_l. We store $(P_r - P_l)$ in the hash table. The concept is similar to that of the *virtual origin* of an array [17].

[8]We have experimented with different sizes, but the performance improvement is not dramatic for larger STCs.

[9]Cache line size must be a multiple of the STC line size.

[10]LRU = Least Recently Used, MRU = Most Recently Used.

```
srl   %rLA, 9, %rA
and   0xff8, %rA, %rA
ldd   [ %rA + %rSTC_READ ], %rA
andn  %rLA, 0xffc, %rT
cmp   %rA, %rT
bne   _do_full_lookup
```

The logical address is presumed to be in register **rEA**. We first extract the index into the STC (this takes two instructions because the Sparc does not have a bit-extract instruction). Next we load the double-word STC entry into the register pair (**rA, rB**). We now extract bits 0-1 and 12-31 from the logical address (refer to figure 3) and compare with the STC tag. If they are equal, then we are finished and the resulting real address is in **[rEA + rB]**. (The Sparc supports register+register addressing so the required load can be done in one instruction.)

Thus, six instructions calculate the hash index, fetch the hash table entry, check the tag value, check for misalignment, and return the real address. The memory overhead is approx. 24 kB per processor.

The above code describes an STC look-up without data cache. For the cache simulation described in the previous section, the shift distance in the first instruction and the bits extracted in the 2nd and 4th instruction would be different.

The small amount of code required makes the STC approach suitable for inclusion in run-time generated code, such as is becoming increasingly popular in simulator designs [14, 43, 33]. SimICS has been designed with this future extension in mind [25].

4 Code Memory

SimICS obviously needs to deal with instruction memory as well. It supports separate MMUs for instructions and data, such as used by the 88110. TLB look-ups for instruction accesses are less crucial, however, since they can largely be done implicitly. We use the technique used in g88—decoded instructions lie consecutively in blocks of logical pages [5]. This means that instruction TLB look-up is only required when branching between pages, which in our measurements are several orders of magnitude less common then branches on pages.

Furthermore, when instructions are decoded we distinguish between on-page branches and off-page branches (whenever we can do so statically). On-page branches require no TLB look-up, and thus execute faster.

Currently, SimICS does not simulate instruction caches.

5 Memory Hierarchies

A particular memory hierarchy module is used to implement whatever functionality an investigator needs in order to analyze a program's behavior with respect to memory operations. We have defined an internal interface for writing problem-specific memory hierarchies well isolated from the internals of SimICS. In this section, we describe this interface.

The user code needs to implement four routines, listed in table 3. `user_mem_possible_cache_miss()` is the most important routine. It is called whenever

- `user_mem_possible_cache_miss()`
- `user_mem_flush_cache()`
- `user_mem_alloc_pp()`
- `user_mem_alloc_cpu()`

Table 3: Memory hierarchy interface

the STC misses. It is passed a memory transaction which already contains all relevant information, including logical, physical, and real addresses, and information from the MMU including cache valid bit.[11] It is up to the memory hierarchy module to update cache state and keep track of relevant statistics. In order to reduce the number of unnecessary calls to this routine, the memory hierarchy module can filter out cache line accesses by calling `mem_add_to_STC()`. This will cause the STC to try to handle future accesses to the specified cache line directly. STC contents can be invalidated by calling `mem_flush_STC()`.

`user_mem_flush_cache()` is called to tell the memory hierarchy that a particular processor has asked to flush its cache (or portions thereof).

One of the difficulties with memory hierarchy simulation is the allocation of suitable data structures for different machine configurations. Therefore, we have written much of this code into the simulator core and it does not have to be re-implemented with every new memory hierarchy. Whenever a physical memory page or processor is allocated, the user memory hierarchy code is called to obtain a pointer to a suitable data structure. `user_mem_alloc_pp()` and `user_mem_alloc_cpu()` are used to dynamically allocate a suitable amount of data during simulation. The memory hierarchy simulator has to decide what structures are related to the number of processors, and what structures are related to the amount of memory that is being used. This is exemplified in the next section.

5.1 Memory Hierarchy Example

In this section we describe an example implementation of a memory hierarchy which we are using to evaluate the cache performance of a real-time kernel. The module simulates a first level cache attached to a common memory bus. The first-level cache is direct-mapped, with the number of sets and associativity selectable at run time. The common memory bus uses a simple Illinois protocol to deal with cache coherency [31].

Since the STC has dealt with most of the performance issue, our focus here is to keep a low memory overhead and a flexible design. Specifically, we wanted the memory overhead to be order $(M + P)$, where M is the amount of physical memory being used by the application (working set) and P is the number of processors.[12]

[11] The cache valid bit indicates whether the particular page should be cached or not.

[12] This precludes simple gathering of several classes of statis-

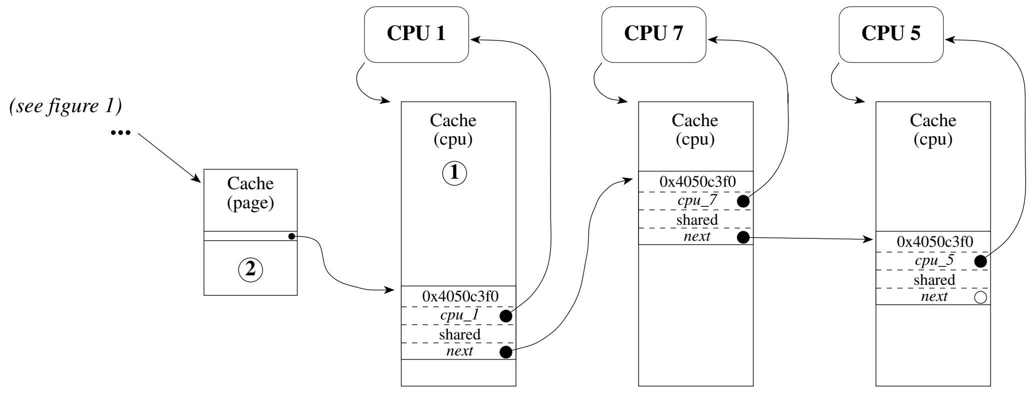

Figure 5: Example Memory Hierarchy Data Structure

The current contents of the caches are represented in an array of cache line info data structures:

```
typedef struct cache_line_info {
  uint32                   physical_address;
  processor_t *            my_cpu;
  cache_state_t            state;
  struct cache_line_info * next;
} cache_line_info_t;
```

This array is labeled "1" in figures 1 and 5. Whenever there is a possible cache miss on a processor, this array is searched and updated as required. Also, for every physical page of memory we allocate an array of pointers. This array is labeled "2" in figures 1 and 5. Thus, `user_mem_alloc_cpu()` returns an initialized array of `cache_line_info_t` structures, and `user_mem_alloc_pp()` returns a zeroed array of pointers (see previous section). In this manner, when the number of processors changes, such as when the user redefines the architecture, or the number of physical pages grows, as when the working set of the program grows, we incrementally allocate just enough space to keep track of cache state. Also, this run-time behavior allows cache parameters to be set at run-time.

The reason we need to allocate an array of pointers for every physical page is to locate the presence of data in other caches. For example, let us assume that a particular line of data beginning at address 0x4050c3f0 is cached in three processors in a 16-processor machine. See figure 5. If another processor gets a cache miss on this line, it can look up the associated pointer in the "cache (page)" table (2). If it is zero, then no other processor caches the data. If it is set, then it will be the header of a linked list of cache lines, one for each processor that has cached the data. The cache protocol can then proceed to do the right thing. If, for example, the processor is attempting a write, then the Illinois protocol proceeds to invalidate the entries in the rest of the list, update statistics, and direct the pointer for this line to its own cache array. Similarly, upon a cache miss, the cache contents can be searched in the "cache (cpu)" table (1). The head of any linked list that is encountered can be obtained via the physical address.

tics in the current implementation, in particular those that depend on knowing the history of a cache line.

Note that the `cache_line_info_t` entries in figure 5 are not at identical offsets within each table. This allows us to simulate multi-set cache designs and even associative caches. For the same reason, each entry needs a pointer to its processor in order to update per-cache statistics since these are kept in the processor structure.

The code is less than 500 lines of C (including comments and declarations). It will adapt dynamically to any machine configuration, maintain correct state of cache lines, and gather basic execution statistics.

The memory overhead of this design is $(16*P*C+4M)/L$ where P is the number of processor, C is the size of each processors' data cache, M is the size of physical memory (working set), and L is the size of a cache line. In this expression we have eliminated trivial overheads, such as pointers to per-processor data cache tables.

5.2 Multiple Memory Hierarchies

Since only a small number of functions are required to attach a memory hierarchy to SimICS (see table 3), and since these are not on the critical path, SimICS supports multiple memory hierarchies in a single binary. Which memory hierarchy module to use can be selected interactively.

5.3 Memory Profiling

The techniques described thus far limit analysis of memory behavior to implementing a memory hierarchy. For some classes of studies, we may wish to look at every memory access, and then the overhead of calling through the modules would be high. As described in the section on intermediate code support, since SimICS uses an intermediate code for interpretation then the set of intermediate pseudo-instructions can include statistics-specific versions.

To demonstrate this, we implemented memory profiling using a specialized set of memory access instructions. In the simplest case, memory profiling involves maintaining a bit map (possibly compressed) of all bytes in memory. The bitmap is initialized to zero, and each individual bit is set by a **write** to that memory address. A more sophisticated version would keep counters for all memory locations.

When memory profiling is enabled, instructions that operate on memory keep track of which bytes in memory have been written to. This allows an exact measurement of working set size. Also, since these maps are easily zeroed, it allows sections of program execution to be profiled interactively to study the fragmentation of memory accesses in portions of a program.

Also, the implemented memory profiling can optionally break on uninitialized memory reads [20]. This allows us to locate the locations in the boot phase of an operating system that might incorrectly depend on the state of undefined memory. Note that traditional memory profiling techniques are not suitable in this instance, since the boot phase of an operating system involves hand-crafted assembler code and run-time modifications of code or run-time linking.

6 Unix Emulation

The Sparc version of SimICS supports emulation of a subset of the Solaris 2.x application binary interface. The support duplicates key data structures and algorithms of the System V kernel, including common Unix system calls such as *execve()*, *fork()*, and *mmap()*. The code (currently around 4000 lines of C) actually emulates aspects of the Unix execution environment such as pre-emptive multitasking, copy-on-write *fork()* semantics, and sharing of file table entries. Running in "Unix mode" can be enabled interactively.

The STC in SimICS allowed quick implementation of TLB simulation and page fault handling. An STC miss triggers a look-up in a fully associative TLB, which if it fails is followed by a search through the current process' memory regions. Thanks to the filtering function of the STC mechanism, these algorithms could be implemented with little concern for efficiency. Indeed, much of the core code of the Unix emulation support was written in a week.

7 Multiprocessor Considerations

SimICS simulates the concurrency of multiprocessors by round-robin scheduling the processors. Each processor is simulated for a fixed time-slice (determined by the user) before switching. This switch must be efficient, or the user will be limited to using long switching intervals [24]. The STC implementation described above requires one pointer to be allocated in a global register during interpretation. It needs to be reloaded from memory upon every processor switch. This is the only overhead that the memory simulation directly contributes to multiprocessor simulation.

8 Performance

The actual performance of SimICS with the memory simulation features described in this paper is difficult to quantify. The simplest measurement of simulator performance in general is the number of instructions interpreted per second. However, this number will vary greatly depending on the application and what features are enabled.

Table 4 lists three examples that illustrate the performance.[13] For each, we compare the performance loss of activating data cache simulation with another useful feature in SimICS, instruction profiling. Instruction profiling counts exactly how many times an instruction in a particular memory location was successfully executed.[14] We also indicate the combined effects.

(a) Dhrystone 2.1	No Data Cache	Data Cache
No Instruction Profiling	2160 (0%)	2117 (-2%)
Instruction Profiling	1824 (-16%)	1827 (-15%)

(b) 023.eqntott	No Data Cache	Data Cache
No Instruction Profiling	1717 (0%)	1864 (+9%)
Instruction Profiling	1564 (-9%)	1640 (-4%)

(c) rt kernel	No Data Cache	Data Cache
No Instruction Profiling	496 (0%)	478 (-4%)
Instruction Profiling	438 (-12%)	439 (-11%)

Table 4: Performance figures in thousands of instructions per second. Figure in parenthesis is the penalty for the feature.

The first example runs a simple Sparc SunOS 4.1 user program, the infamous Dhrystone 2.1 benchmark [44]. In the measurement, it runs 100 000 iterations, which requires approximately 50 million instructions. The cache performance is here excellent (0.001% miss rate), so the STC performs admirably. Accurate data cache contents are maintained at a 2% performance loss.

The second example runs a much larger Sparc program from the SPECint92 suite [40], which requires 1.25 billion instructions to complete. The data cache behavior is worse (a realistic working set), and the memory hierarchy simulation code is thus called more often. Despite this, SimICS actually runs faster with cache simulation enabled.[15]

Our third example is considerably different. It is a measurement of the boot process of a industry proto-

[13]The measurements were done on a Sun SC2000, with 50MHz SuperSparc processors.

[14]In system-level simulation this is more complex than just measuring entries into basic blocks, for several reasons; a basic block may be interrupted by an exception and not re-entered, the program may generate code at runtime (such as trap vectors), etc.

[15]Performance for SimICS will easily vary by 10-15%, due to effects such as internal hash table collisions, etc. As we described previously, the STC uses a finer granularity in its hash tables when cache simulation is enabled, which could account for this irregularity.

type real-time kernel running on an 88110-based architecture [29]. The absolute performance is much lower because the core interpreter is older and the boot process is intensive in page faults, interrupts, and device programming. The cache performance is poor since this version of the kernel did not cache portions of the address space. Again, simulating the data cache only diminishes performance slightly.

We used a snapshot version of SimICS for the above measurements. By contrast, carefully tuned versions running 023.eqntott have peaked at 4.2 Mips with all features turned off. We would expect the penalties indicated in table 4 to be higher on faster versions.

9 Related Work

Traditionally, the approach used to gather complete address traces of multiprogram and OS workloads involved microcode or different forms of hardware monitoring or modifications [1, 13, 30, 41]. Recently, more flexible techniques have been developed that rely "only" on the manipulation of ECC bits in the host memory and clever modifications to the host operating system [32, 42]. In general, these techniques are unwieldy and inflexible.

Instrumenting the program binary is a common software solution to simulation. However, this approach tends to sharply constrain the class of programs that can be studied, in particular the programming language and/or compiler being used. For example, PROTEUS [10] requires the C compiler to support specific extensions to the C language, RPPT [15] requires the program to be written in Concurrent C, Tango [16] requires the program to be written using ANL macros [9], and Larus' Abstract Execution [22] requires a modified GNU C compiler. Generally, instrumenting the code is restricted to user-level programs, although recent work demonstrates that it can be extended to simultaneously instrument parts of the host operating system [12].

Analyzing the memory accesses of a program can also be done using straight-forward interpretation, passing all memory references to a memory simulator. An example is the CacheMire testbench [11]. This approach is sometimes called *program-driven simulation*. Mint [43] has a similar approach, though it can also generate run-time code in order to speed up execution.

Program-driven simulation can be extended to perform a full logical-to-physical translation on every memory access and simulating memory-mapped devices, allowing the interpreter to support arbitrary programs including multitasking operating systems running on parallel hardware. An example is Talisman [7]. Talisman performs a detailed simulation of the Meerkat [6] memory system, doing a function call on each memory access. This scheme slows down the simulation, but permits accurate timing. A speedier predecessor to Talisman, g88 [5], simulated an "infinite" TLB using a double indirection that was filled with a simulated hardware table walk on misses. It did not simulate data caches. (g88 is in many senses also a predecessor of SimICS.) Another example is Halsim [4].

In a way, a lesson learned from hardware monitoring is that the most efficient technique for memory simulation is to avoid it entirely. Several different software-based approaches use variations on this theme. A common example is traditional debuggers, which rely on the operating system and native hardware to run the program while the debugger maintains control using OS support (e.g. the Unix *ptrace()* system call).

In Shade [14], the simulator and the program share the same Unix process address space. A fixed offset is added to every simulated memory access to yield the real address. Shade generates native code at run-time to simulate the program code. This scheme allows Shade to vary the amount and type of trace instrumentation dynamically, under the control of an *analyzer* program written by a user. However, the current memory simulation in Shade is restricted to user-level, well-behaved, Unix programs with continuous address space. The continuous address space restriction disallows simulating a shared multiprocessor, for instance, where portions of the address spaces overlap (stack).

The SimOS [33] environment supports a *direct execution mode* wherein the operating system and applications run on a virtual machine implemented within a Unix process. Careful use of *mmap()* maps a subset of the target TLB into the host TLB. By modifying the IRIX Unix kernel to be able to execute in a user process, the execution of a full multi-tasking operating system with applications can be "fast forwarded". While being fast forwarded, user programs must be well-behaved: since the subset of target TLB:s cannot be split into "target user" and "target system" entries, they have full access to the entire simulated physical memory.

Upon reaching a portion of interest, the execution can continue in a more detailed *fast mode* that uses run-time binary translation to emulate the CPU and MMU in software. In fast mode, SimOS performs address translation, page protection check, and cache residency check in an optimized sequence of 12 instructions. This density is achieved by using large tables: a 4 MB array for address translation and an array of around 32 MB for cache and TLB protection check. By contrast, SimICS, using the techniques in this paper, does the same work in 6 instructions and an overhead of 24 kB per processor for the STC tables. Note also that these large tables makes context switches expensive, which could be a bottleneck in environments with very many light-weight processes.

Both Shade and SimOS rely on a close mapping between the target and host hardware and operating system. Neither is particularly portable.

A common characteristic of the above approaches is that little is done to help the user implement efficient memory hierarchies. Typically, the simulator explicitly passes information on *all* memory accesses to the user simulation code. This has led to a situation where simulator execution time has been dominated by simulation of caches, interconnection network, etc. In turn, this has led to the (incorrect) conclusion that it is not worthwhile laboring on very fast software simulation techniques. The STC design in SimICS, by contrast, can radically reduce the performance impact of slow end-user code.

Summary and Conclusions

We presented several techniques that are useful when designing an efficient system-level instruction set simulator. We have addressed the problem of efficiently simulating memory, including cache, so that the performance is not too heavily penalized when simulating a memory hierarchy and gathering statistics. These techniques are more efficient and flexible than previous designs, and should be useful to several existing simulation environments.

Despite having a heavily optimized simulator core, we have written SimICS in a sufficiently modular fashion to support simple addition of new memory hierarchies and interactive specification of number of nodes and processors. Physical memory, data structures for cache simulation, and intermediate code pages are allocated lazily.

The design ideas have been used to implement efficient multiprocessor simulators based on Motorola's 88110 processor and Sun's Sparc v8 architecture. The resulting simulator runs programs, including system-level binaries, at around 2 million instructions per second on a high-end Sun workstation.

Traditional simulators typically delegate all work in handling memory references to "trace consumers" written by end users. By using the STC module in SimICS, an analyzer program can get help from a carefully programmed simulator core to filter out superfluous information, thus sharply reducing the overhead of simulating caches.

We conclude that the techniques can be applied to simulate any similar RISC-like architecture and that the performance of a simulator is not dominated by statistics instrumentation if designed carefully. Furthermore, the performance benefits yielded by these techniques is an argument in favor of program-driven simulation in general.

Acknowledgments

We owe many thanks to Robert Bedichek, David Keppel, and Anders Landin for many useful insights relating to memory simulation, including extensive comments on earlier drafts of this paper. We would also like to thank Robert Cmelik, Torbjörn Granlund, Erik Hagersten, Gordon Irlam, and Ashley Saulsbury for many interesting discussions over the years on memory simulation.

This work has been sponsored by Ellemtel, the European Commission through the GPMIMD project, and the Swedish Institute of Computer Science sponsors: CelsiusTech, Ellemtel, Ericsson, IBM, Sun Microsystems, Swedish Defense Material Administration (FMV), Telia, and the Swedish National Board for Industrial and Technical Development (Nutek).

References

[1] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of 13'th ISCA*, pages 119–127, June 1986.

[2] K. Andrews and D. Sand. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of ASPLOS–V*, pages 213–222, October 1992.

[3] Apple Corp. PowerMacintosh Product Information, 1994.

[4] D. Barach, J. Kolhi, J. Slice, M. Spaulding, R. Bharadhwaj, D. Hudson, C. Neighbors, N. Saxena, and R. Crunk. HALSIM - A Very Fast SPARC V9 Behavioral Model. In *Proceedings of MASCOTS*, January 1995.

[5] R. C. Bedichek. Some Efficient Architecture Simulation Techniques. In *Proceedings of Winter '90 USENIX Conference*, pages 53–63, January 1990.

[6] R. C. Bedichek. *The Meerkat Multicomputer: Tradeoffs in Multicomputer Architecture.* PhD thesis, University of Washington, 1994.

[7] R. C. Bedichek. Talisman: Fast and Accurate Multicomputer Simulator. In *Proceedings of the '95 ACM SIGMETRICS Conference*, 1995. Personal Communication (to be published).

[8] J. R. Bell. Threaded Code. *Communication of the ACM*, 16(6):370–372, June 1973.

[9] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[10] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[11] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström. The Cachemire Testbench – A Flexible and Effective Approach for Simulation of Multiprocessors. In *Proceedings of the 26'th Annual Simulation Symposium*, pages 41–49, 1993.

[12] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14'th ACM SOSP*, pages 120–133, December 1993.

[13] D. Clark. Cache Performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1:24–37, 1983.

[14] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-set Simulator for Execution Profiling. In *Proceedings of the '94 ACM SIGMETRICS Conference*, pages 128–137, May 1994.

[15] R. Covington, S. Madala, V. Metha, J. Jump, and J. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the '88 ACM SIGMETRICS Conference*, pages 4–11, 1988.

[16] H. Davis, S. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Tech. Report No CSL-TR-90-439, Stanford University, 1990.

[17] C. N. Fischer and R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[18] S. Graham, P. Kessler, and M. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[19] A. M. Grizzaffi Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting Characteristics and Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of ASPLOS–VI*, pages 145–155, October 1994.

[20] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of Winter '92 USENIX Conference*, pages 125–136, January 1992.

[21] R. Heisch. Trace–directed program Restructuring of AIX Executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[22] J. Larus. Abstract Execution: A Technique for Efficient Tracing Programs. Technical report, Computer Science Department, University of Wisconsin at Madison, 1990.

[23] P. Magnusson. Efficient Simulation of Parallel Hardware. Masters thesis. Royal Instiute of Technology (KTH), Stockholm, Sweden, 1992.

[24] P. Magnusson. A Design for Efficient Simulation of a Multiprocessor. In *Proceedings of MASCOTS*, pages 69–78, January 1993.

[25] P. Magnusson. Partial Translation. Technical Report T93:05, Swedish Institute of Computer Science, October 1993.

[26] P. Magnusson and D. Samuelsson. A Compact Intermediate Format for SIMICS. Technical Report T94:17, Swedish Institute of Computer Science, September 1994.

[27] C. May. Mimic: A Fast System/370 Simulator. In *In Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 1–13, June 1987.

[28] S. McFarling. Program optimization for instruction caches. In *Proceedings of ASPLOS-III*, pages 183–191, April 1989.

[29] Motorola. *MC88110–Second Generation RISC Microprocessor, User's Manual*. Motorola Literature Distribution, Arizona, 1991.

[30] D. Nagle, R. Uhlig, and T. Mudge. Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures. Technical Report TR-147-92, The University of Michigan, 1992.

[31] M. Papamarcos and J. Patel. A Low–Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceeedings of 11th ISCA*, pages 348–354, 1984.

[32] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of '93 ACM SIGMETRICS Conference*, pages 48–60, 1993.

[33] M. Rosenblum and E. Witchell. SimOS: A Platform for Complete Workload Studies. Personal Communication (to be published), 1995.

[34] D. Samuelsson. System Level Interpretation of the SPARC V8 Instruction Set Architecture. Technical Report R94:23, Swedish Institute of Computer Science, August 1994.

[35] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[36] R. L. Sites, A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[37] J. Smith and S. Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *IEEE Computer*, 27(6):46–58, June 1994.

[38] SPARC International, Inc. *The SPARC Architecture Manual, version 8*, 1992.

[39] R. M. Stallman and R. H. Pesch. Using GDB, Edition 4.04 for GDB version 4.5, March 1992. `ftp://prep.ai.mit.edu/pub/gnu` GDB distribution.

[40] Standard Performace Evaluation Corporation (SPEC). Spec Benchmark Suites, 1992. c/o NCGA 2722 Merrilee Drive, Suite 200, Fairfax, Virginia 22031.

[41] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Proceedings of ASPLOS–V*, pages 162–174, October 1992.

[42] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap–driven Simulation with Tapeworm II. In *Proceedings of ASPLOS–VI*, pages 132–144, October 1994.

[43] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of MASCOTS*, pages 201–207, January 1994.

[44] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.