

Some Efficient Techniques for Simulating Memory

Peter Magnusson
psm@sics.se

Bengt Werner
werner@sics.se

September 1994

Parallel Computer Systems
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. A lazy memory allocation scheme reduces 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 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.

Some Efficient Techniques for Simulating Memory

Peter Magnusson and Bengt Werner

Swedish Institute of Computer Science
Box 1263, Kista
Stockholm, Sweden
{psm,werner}@sics.se

September 1994

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. A lazy memory allocation scheme reduces 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 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

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 and Splash are common points of reference, 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. If a program is non-representative, we need to determine this, and furthermore to decide whether to modify the program or specialize the hardware. In either case, there is a need for tools to analyze program behavior and the interaction of programs with the underlying architecture.

Traditional use of simulation as an instrument has often suffered from the consequences of poor simulator design. If the simulator is slow or has a large memory overhead, then 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 that are often omitted 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 code that is otherwise difficult to deal with, such as system-level; furthermore, the unchallenged control over execution that a simulator can deliver offers the opportunity for new approaches to program debugging.

SIMICS is an instruction-level simulator developed at SICS that can play all the above roles. Our goals with SIMICS include:

- *performance*; an efficient simulator, both in execution speed and memory requirements, allows for large programs, large working sets, and long execution runs,
- *language independence*; we do not want to be restricted to any particular target program implementation language, compiler, library, or even require source code for the programs,
- *system level*; the simulator must simulate some form of system binaries to allow studies of the interaction of operating system, user program, and architecture,
- *interaction*; we wanted a simulator that supported debugging primitives so that we could use symbolic debuggers such as GDB¹ or Xray² as a front end,
- *determinism*; the execution of the simulator must be completely deterministic, to allow repeats of both statistics gathering and debugging,
- *memory simulation*; we need ways to study multiprocessor memory hierarchies without excessive simulator performance loss,³
- *portability*; a simulator should be portable to allow quick adaptation to new host platforms,
- *multiprocessor*; we wish to efficiently simulate multiple processors, including pertinent multiprocessor features such as inter-processor interrupts, message passing, or external TLB invalidations,
- *statistics*; we will need to extend the simulator to gather pertinent architecture and/or program statistics, such as memory usage, frequency of important events, or instruction profiling,
- *low startup cost*; the simulator should start quickly to speed the “edit-compile-simulate” cycle—complex initializations should be kept to a minimum,
- *low turnaround*; we wish the simulator to be structured (internally) such that it can quickly be modified or extended without requiring excessive portions of it to be re-compiled,
- *extendibility*; a user of the simulator should be able to develop their own extensions with minimal understanding of the core of the simulator,

All the above goals set constraints on the simulation of memory. To meet them, we have developed a combination of techniques that we mix and match within SIMICS. By designing SIMICS in an object-oriented manner, we can isolate the complexities in well-defined modules.⁴ This has allowed us to write a single simulator that:

- supports memory management unit semantics,
- simulates single or multiprocessor,

¹The Gnu Debugger, from the Free Software Foundation.

²A symbolic debugger from MRI.

³A 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.

⁴SIMICS is written in C. We isolate all data relating to an object in a single structure. Generally, only functions in a single C file can manipulate the object. To avoid offending the purists, we will hereafter refer to this as a *modular* design.

- simulates cache, including linking with memory hierarchy simulator written by user,
- deals correctly with supervisor and user addresses, handling page faults etc.,
- supports shared physical address space (e.g. bus-based multiprocessor), distributed (e.g. message-passing architecture), or hybrids,
- profiles memory usage (working set),
- is fast and with low memory overhead.

Thanks greatly to the modularized design, all the features in this list can be selected interactively. Thus, SIMICS can simulate a four-processor architecture with a shared memory bus and 64K direct-mapped first level cache, or a 16-processor distributed memory architecture with message passing devices mapped into supervisor address space and 8K two-way associative cache, without recompiling. The performance impact of this flexibility is minimal compared to a specialized design.

The design of these features is the topic of this paper.

2. Overview of this Paper

In section 3 we give a brief overview of the internal structure of SIMICS as background information for the remainder of the paper. Section 4 described the generic problem of simulating memory in a system-level simulator.⁵ Section 5 defines some important terms, and gives a short overview of the principal components of memory simulation in SIMICS. Sections 6-16 dig into the details of the implementation. This is followed by some performance figures in section 17 before the summary and conclusions.

The first of the detailed sections, section 6, describes the concept of a *memory transaction*, a data structure for communicating memory operations within SIMICS. Section 7 mentions some relevant consequences of the underlying interpretation technique on the memory simulation. Sections 8 and 9 mention some relevant issues about virtual memory and physical memory, respectively. Section 10 is one of the key sections, describing the concept of the simulator translation cache (STC). Section 11 describes an extension of the STC (the enhanced STC) that supports data cache simulation.

The extendibility of SIMICS by the user is the topic of sections 12 and 13, the first describing what a *memory hierarchy interface* is, and the latter detailing a sample implementation. Sections 14, 15, and 16, finally, briefly discuss some peripheral aspects and alternatives.

If you wish a casual reading of the most important ideas in this paper, you should focus on sections 4, 5, 10 and 12.

3. 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. Generic data structures exist for nodes, processors, and devices. 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. 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 (null-terminated list of processors on a node), machine list (all processors), and a scheduling list (for round-robin scheduling to simulate concurrency).

⁵We use the term *system-level simulator* to mean a simulator that deals with system-level aspects of architectures and multiprogramming systems, such as protection domains, virtual memory, memory-mapped devices, interrupts, exception handling, paging, inter-processor communication, etc.

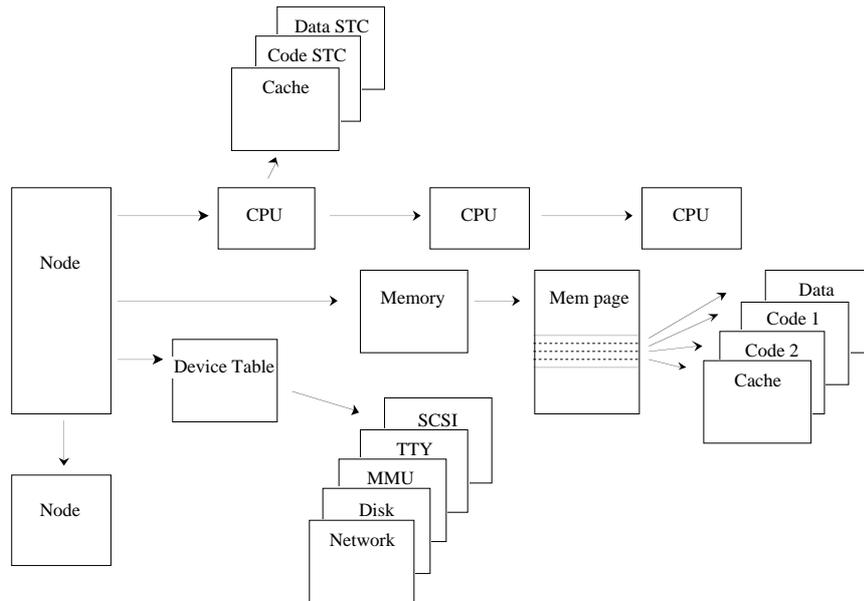


Figure 1: Principal SIMICS data structures

The program code for SIMICS is structured with separate file pairs (c file and header file) that cleanly delineate responsibility:

- machine.[ch] describes general architecture dependencies, such as where RAM is located, the generic device table (will return to this), minimum and maximum number of processors and nodes.
- 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 run-time), the generic device table is copied and the listed initializers called. Generic device functions and data structures are kept in device.[ch].
- 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.
- memory.[ch] implements generic memory simulation code.
- memory-hier.[ch] implements specific memory simulation code, i.e. a particular memory hierarchy.

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()* are used to apply a function to multiple objects.

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.

4. Simulating Memory

To support system-level simulation, we need to faithfully simulate virtual memory. This includes virtual to physical translation, checking access rights, and simulating TLB⁶. Correct TLB contents are required to simulate the interleaving of user and system code due to page faults in the case of software-loaded TLBs, and to generate correct memory accesses for table walks in the case of

⁶Translation look-aside buffer, sometimes called the address translation cache.

hardware-loaded TLBs. The memory simulation must be efficient, since on the order of every fourth instruction in typical RISC code is a memory access.⁷

Specifically, for every memory access, we need to do the following:

- 1 calculate the effective address,
- 2 translate from a virtual address to a physical,
- 3 check for TLB misses,
- 4 check protection,
- 5 check for alignment violation,
- 6 perform the read/write operation, and
- 7 update processor state.

Table 1: Memory Operation Checklist

Misalignment refers to the inability of some processors to access data block boundaries. For instance, a word (32 bits) can often only be read from a word-boundary address, i.e., from an address where the bottom two bits are zero. Some architectures will round (clear) the affected bits if so desired (m88110), others will always trap (Sparc).⁸

In addition to the list above, we want to use the simulation of memory to implement watchpoints⁹, profile memory usage, and simulate a memory hierarchy. It should therefore be flexible.

Another issue is the accuracy with which we should simulate the memory hierarchy. There are at least four levels of increasing accuracy, listed in table 2.

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

Table 2: Memory Simulation Accuracy

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

SIMICS supports the first three levels. We will discuss cache simulation in the next section. The fourth level is not supported.¹⁰ We will discuss cache simulation in the next section.

5. Memory Simulation in SIMICS

We define 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 simulated physical address needs to be translated to an address in the simulator's virtual address space, and this address we call the *real* address.

⁷Obviously, every instruction fetch is also a memory access.

⁸Some architectures expect a division of labor between processor and operating system. As an example, the Alpha architecture has efficient byte manipulation instructions. If the compiler determines that a memory access is likely to be misaligned, it can output explicit code. If the compiler thinks that misalignment is unlikely, it can attempt word (64-bit) access, and if it is misaligned the operating system trap handler will perform the operation instead. (Smith, Weiss 94)

⁹A 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 watchpoint on.

¹⁰It is not clear to us how useful such an extension would be. Memory access delays is only one factor that affects performance of a processor, examples of others being: lock-up free caches, multiple dispatch, out-of-order execution, write buffers, speculative branch prediction, and register file bottlenecks. Simulating one without the others would yield a meaningless statistic.

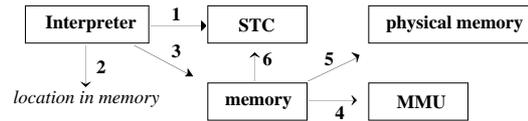


Figure 2: Memory Simulation in SIMICS

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). 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). Next, the physical memory module will translate the physical address to a real address, allocating new space if necessary. With the logical, physical, and real addresses of an operation the memory module updates the STC (6). Thus, the next time this page is accessed the STC will succeed in step (1).

6. 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. 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 so that the cache simulation code can determine whether cache is enabled for this particular memory operation. This has 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.¹¹

7. Intermediate Code Support

The core of SIMICS is based on threaded code techniques (Bell 73). We translate target object code to an internal format, the details of which are beyond the scope of this paper. 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 cases: normal, minimal statistics, memory profiling, and cache simulation.

The normal mode is optimized for speed, thus simulating only the functional correctness of the execution.¹² The minimal statistics mode counts memory accesses according to user or supervisor space, and read or write. Memory profiling and cache simulation are described in more detail further on.

¹¹Procedure calls are generally faster since we reduce the number of unnecessary push/pop operations on the stack. It also simplifies debugging.

¹²This includes correct MMU simulation.

This structure allows a single simulator binary to support all the features described in this paper. Furthermore, the individual features can be toggled during execution—the necessary internal data structure is allocated/deallocated as necessary, the execution state is unaffected.¹³

8. Memory Management Unit

The MMU module is well isolated from the other memory simulation components. The MMU needs to provide a routine—*mmu_logical_to_physical()*—to report on legal translations. Conversely, the MMU simulation code can call a routine to clear address intervals that are cached elsewhere. Depending on the circumstances, a call to *mmu_logical_to_physical()* may or may not be an actual TLB miss. 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 simulates the m88110 MMU and a 1:1 translation for direct execution of user binaries. The m88110 MMU is programmed using control registers, which are special instructions on the 88k. New MMU modules are easy to add.

9. 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.¹⁴ Figure 1 shows that it is associated with a node. Multiple processors on a single node always share the same 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. A single pointer associates a node with the (hierarchical) memory data structure. Thus, the decision by the user to simulate a shared memory or a distributed memory architecture can be decided interactively.

Physical address space is seldom 32-bit. Therefore, the machine description file can define a macro *FIX_ADDRESS()* that is applied to all physical addresses. For example, the top bits are often ignored.

10. Simulator Translation Cache (STC)

The STC caches legitimate 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.

The principal STC data structure and translation scheme are 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) are 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. The bottom $(s+1)$ bits of the logical address being translated are not cleared, where s is the log base 2 of the size of the memory operation. Thus, a tag comparison failure means either a translation miss or a misalignment. Finally, the value actually stored is the difference between the real and logical address. The real address can be formed by adding the logical address to this value.¹⁵

¹³If this is done for cache simulation, the caches will be cold-started and will not yield correct statistics until they have been warmed up.

¹⁴Currently SIMICS supports at most 32-bit physical address spaces.

¹⁵This works due to the following observation: let P be the starting address of the simulated logical page containing the logical address Q to be translated. Let RP be the starting address in the host address space (real

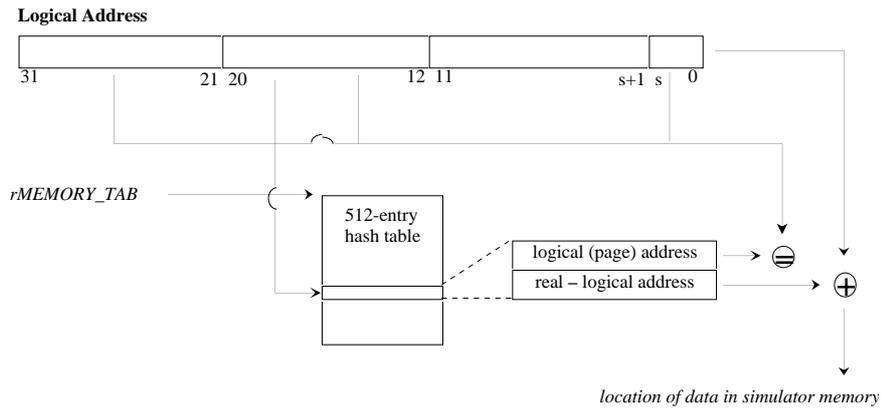


Figure 3.: Simulator Translation Cache (STC)

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. The STC does not need to handle misses, so there is no alternate linked-list structure.

This design handles items 2-6 in Table 1.

The compiled code generates seven Sparc assembler instructions. These seven instructions calculate the hash index, fetch the hash table entry, check the tag value, check for misalignment, calculate the real address, and perform the load.

The C code used for implementing this design uses a single macro: `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 conversion to be applied (read 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.

11. 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.

Fortunately, cache line lookup and TLB lookups are often done in parallel on real hardware. To allow this, the lower bits (those typically used for cache lookup) 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 lookup in one operation.

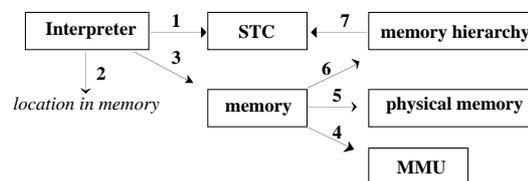


Figure 4: Cache Simulation in SIMICS

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. Finally, the memory hierarchy module can update the STC. We return to the memory hierarchy in the next section.

address) of the corresponding page. Then $(RP - P) + Q = RQ$, where RQ is the real address of Q . We

The enhanced STC scheme for cache simulation is almost identical to figure 3. We do the lookup as if the cache was a direct-mapped, 4k cache with 16-byte cache lines.¹⁶ The hash table is 256 entries, and the hash index is formed by bits 4-11 of the logical address.

We need to check the logical address and the alignment, as in the previous section. Note how this does not restrict the memory hierarchy to a particular first-level cache size or organization, because cache lines are only added to this STC if told to by the memory hierarchy code.¹⁷ 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.¹⁸ If the replacement is random, then any line can be in the STC.

SIMICS will not track STC hits other than to count them (supervisor read hits, supervisor write hits, user read hits, user write hits).

12. Memory Hierarchy Interface

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 (API) for writing problem-specific memory hierarchies to isolate the internals of SIMICS from the implementation of new memory hierarchies. In this section, we describe this interface.

The user code needs to implement four routines; *user_mem_possible_cache_miss()*, *user_mem_flush_cache()*, *user_mem_alloc_pp()*, and *user_mem_alloc_cpu()*.

user_mem_possible_cache_miss() is the most important routine. It is called whenever the STC misses. It is passed a memory transaction that already contains all relevant information, including logical, physical, and real addresses, and information from the MMU including cache valid bit. 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 by the MMU simulator 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.

13. A Simple Memory Hierarchy Module

In this section we describe an example implementation of a memory hierarchy that we are using to evaluate the cache performance of a proprietary 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 MESI protocol to deal with cache coherency.

Since the STC has dealt with most of the performance issue, our focus here is to keep a low memory overhead. Specifically, we want the memory overhead to be $o(M+P)$, where M is the

calculate $(RP - P)$. This eliminates (at least) a mask instruction during execution.

¹⁶We have experimented with different sizes, but the performance improvement is not dramatic for larger STCs.

¹⁷Cache line size must be a multiple of the line size in the STC; 16 in the example.

¹⁸LRU = Least Recently Used, MRU = Most Recently Used.

amount of physical memory being used by the applicatino (working set) and P is the number of processors.¹⁹

For each processor, we allocate an array of cache line info data structures:

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

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. Thus, `user_mem_alloc_cpu()` returns an initialized array of `cache_line_info_t` structures, and `user_mem_alloc_pp()` return a zeroed array of pointers. This way, as the number of processors changes (when the user re-defines the architecture) or the number of physical pages grow (as the working set of the program grows), we incrementally allocate just enough space to keep track of cache state.

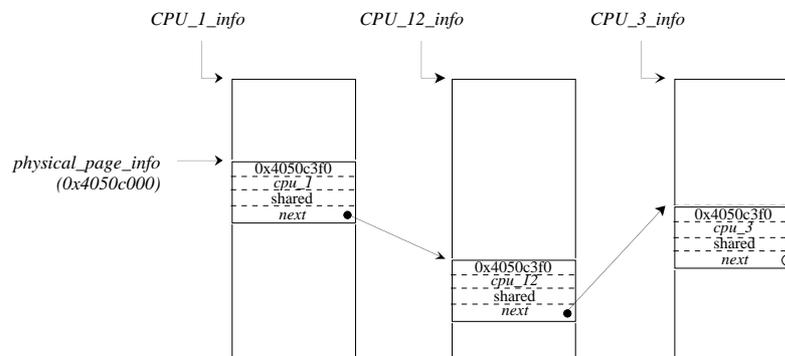


Figure 5: Example Memory Hierarchy Data Structure

The reason we need to allocate an array of pointers for every physical page is to locate the presence of data in other caches. Consider figure 5. The same data is cached in three processors (in a 16-processor machine). If another processor gets a cache miss on this line, it can look at the `physical_page_info` pointer for the particular page. If it is zero, then no other processors cache 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. If, for example, the processor is attempting a write, it will proceed to invalidate the entries in the rest of the list, update its statistics to this effect, and point the `physical_page_info` pointer for this line to it's own cache array.

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 elementary execution statistics.

14. Memory Profiling

The techniques described so 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.²⁰ When memory profiling is enabled, old translations of instructions that operate on memory are discarded. The new ones 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 cleared, it

¹⁹This precludes simple gathering of several classes of statistics in the current implementation, specifically those that depend on knowing the history of a cache line.

²⁰Actually, this particular function could easily be implemented as a memory hierarchy.

allows sections of program execution to be profiled to study the fragmentation of memory accesses in portions of the program.

Also, the implemented memory profiling can optionally break on uninitialized memory reads (UMR).²¹ 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.

15. 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 lookups for instruction accesses are less crucial, however, since they can largely be done implicitly. We use the same technique as g88 in that decoded instructions lie consecutively in blocks of logical pages (Bedichek 90). This means that instruction TLB lookup is only required when branching between pages, which in our measurements are several orders of magnitude less common than 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 lookup, and thus execute faster.

We do not currently simulate instruction caches, though the same techniques used in mg88 could be applied; instructions are only decoded when they are in the cache (Bedichek 94).

16. 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. The STC implementation described above requires only two pointers to be allocated in registers during interpretation (since they are used on every memory access). They need to be re-read from memory upon every processor switch. This is the only overhead that the memory simulation directly contributes to multiprocessor simulation.

17. Performance Figures

The actual performance of SIMICS with the memory simulation features described in this paper is difficult to quantify in any systematic manner. The simplest measurement of simulator performance in general is the number of instructions interpreted per second (measured in thousands, or kips). However, this number will vary greatly depending on the application and what features are enabled.

Table 3 lists three examples that illustrate the performance.²² 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.²³ We also indicate the combined effects.

The first example runs a simple Sparc SunOS 4.1 user program, the infamous Dhrystone 2.1 benchmark (Weicker 84). 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, which requires 1.25 billion instructions to complete. The data cache behavior is worse (a realistic working set), and

²¹We have borrowed the term UMR from the commercial product Purify which has a similar function.

²²The measurements were done on a Sun SC2000.

²³In 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.

the memory hierarchy simulation code is thus called more often. Despite this, SIMICS actually runs faster with cache simulation enabled.²⁴

<i>Dhrystone 2.1</i>	No Data Cache	Data Cache
No Instruction Profiling	2160 (0%)	2117 (-2%)
Instruction Profiling	1824 (-16%)	1827 (-15%)
<i>023.eqntott</i>	No Data Cache	Data Cache
No Instruction Profiling	1717 (0%)	1864 (+9%)
Instruction Profiling	1564 (-9%)	1640 (-4%)
<i>rt kernel (88k)</i>	No Data Cache	Data Cache
No Instruction Profiling	496 (0%)	478 (-4%)
Instruction Profiling	438 (-12%)	439 (-11%)

Table 3: Examples of Memory Hierarchy Performance

Our third example is considerably different from the other two. It is a measurement of the boot process of a commercial real-time kernel running on an 88110-based architecture. 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 truly poor (>30% misses), since this version of the kernel did not cache portions of the address space. Again, simulating the data cache only diminishes performance slightly.

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.

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 the Motorola 88110 processor and Sun's Sparc v8 architecture. The resulting simulator runs programs at around 2 million instructions per second on a high-end Sun workstation, including operating system binaries.

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.

Acknowledgements

Robert Bedichek's work has been the standard with which to compare, and he has always been helpful. This work has profited from discussions with several people. In no particular order, we would like to thank Dave Keppel, Gordon Irlam, Bob Cmelik, Per Berg, Staffan Skogby, Anders Landin, Seif Haridi, and Erik Hagersten.

²⁴Performance 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.

Bibliography

- Bedichek, R. 1990. "Some Efficient Architecture Simulation Techniques." In *USENIX - Winter '90*, 53-63.
- Bell, J. R. 1973. "Threaded Code." *Communications of the ACM* 16, no. 6 (June): 370-372.
- Cmelik, R. and Keppel, D. "Shade: A Fast Instruction-Set Simulator for Execution Profiling", in proceedings of *SIGMETRICS 1994*.
- Deware, R. B. K. 1975. "Indirect Threaded Code." *Communications of the ACM* 18, no. 6 (June): 330-331.
- Knuth, D. E. 1973. *The Art Of Computer Programming Vol 1, Fundamental Algorithms*. Addison-Wesley, Reading, Mass.
- Lang, T. G.; J. T. O'Quine; and R. O. Simpson. 1986. "Threaded Code Interpreter for Object Code." *IBM Technical Disclosure Bulletin* 28, no. 10 (March): 4238-4241.
- Magnusson, P. 1992. "Efficient Simulation of Parallel Hardware." Masters thesis. Royal Institute of Technology (KTH), Stockholm, Sweden.
- May, C. 1987. "Mimic: a Fast System/370 Simulator." In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota). June 25-26, 1987, pp 1-13.
- Motorola 1990, *MC88200 Cache/Memory Management Unit User's Manual*, Second Edition, Prentice Hall, 1990.
- Motorola 1990, *MC88100 Risc Microprocessor User's Manual*, Second Edition, Prentice Hall, 1990.
- Patterson, D. A. *Reduced Instruction Set Computers*, Communications of the ACM, Vol 28, No 1 (January 1985), pp 8-21.
- Samuelsson, D. 1994. *System Level Simulation of the SPARC V8 Instruction Set*, SICS Technical Report, August 1994.
- Sites, L.R., Chernoff A., Kirk M.B., Marks, P.M., and Robinson, S.G. *Binary Translation*, Communications of the ACM, February 1993, Vol. 36 No. 2, pp 69-81,
- Stallman, R. M. 1992. *Using and Porting GNU CC, version 2.0 (15 February 1992)*, Free Software Foundation, Mass., USA.
- Smith, J.E. and Weiss, S. *PowerPC 601 and Alpha 21064: A Tale of Two RISCs*, IEEE Computer, June 1994, Vol 27, No 6, pp 46-58.
- Sun 1990, *The Sparc Architecture Manual, Version 8*, Sun Microsystems, USA, December 1990.
- Veenstra, J.E. and Fowler, R.J. "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors", proceedings of *MASCOTS 1994*.
- Weicker, R. P. 1984. "Dhrystone: A Synthetic Systems Programming Benchmark." *Communications of the ACM* 27, no. 10, (Oct.): 1013-1030.
- Weicker, R. P. Dhrystone benchmark, C, version 2.1, Siemens AG, Postfach 3240, 8520 Erlangen, Germany.