Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

EFFICIENT INSTRUCTION CACHE SIMULATION AND EXECUTION PROFILING WITH A THREADED-CODE INTERPRETER

Peter S. Magnusson

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, SWEDEN

ABSTRACT

We present an extension to an existing SPARC V8 instruction set simulator, SimICS, to support accurate profiling of branches and instruction cache misses. SimICS had previously supported profiling data cache efficiency and virtual memory performance (TLB misses), and estimated execution profiling using sampling. The new design allows a system-level, threaded-code simulator of a computer system to efficiently support a relatively complete range of instrumentation. Principal applications include computer architecture studies and performance tuning of software. Both application areas require reasonable performance in order to support realistic workloads, and both benefit from the flexibility, generality, and portability of a fast threaded-code simulator. The presented design supports multiprocessor simulation, system-level (operating system) programs, and, in principle, arbitrary user programs including run-time generated code. We evaluate the performance using the SPECint95 benchmark suite, and the result, with full profiled instrumentation enabled, is an execution time 26-108 times slower than native execution.

1 INTRODUCTION

Instruction set simulators are used by computer architects and programmers for a variety of tasks, including architecture design, porting system software, and performance tuning of software.

The basic design of modern instruction set simulators is often a variation of threaded code (Bell 1973). This design can be extended to support full system level simulation (Bedichek 1990) and multiprocessors (Magnusson 1993). An element that has been missing in instruction set simulators is efficient support for accurate execution profiling and instruction cache modeling. Instruction cache behavior is important for computer architects, and both instruction cache and execution profiling are important to support performance tuning of software.

Given their close relationship with the threaded code interpreter itself, both are most suitably addressed by designing a threaded code kernel that directly supports instruction cache and branch profiling. The design must be efficient enough to allow realistic workloads to be studied, as well as supporting the simulation of a wide range of architectures (notably shared memory multiprocessors) and workloads (notably operating systems).

We present such a solution in this paper, and describe an implementation of it in SimICS, a SPARC V8 simulator. The design is in many ways similar to an earlier design for memory simulation (Magnusson and Werner 1994). We use the SPEC95 integer benchmark suite for evaluation. Despite being enhanced with instruction cache modeling and accurate branch profiling, in addition to previous data cache and translation look-aside buffer (TLB) profiling, the resulting slowdown of SimICS is in the range of 26-108.

1.1 Instruction Set Simulation

Instruction set simulators run a program by simulating the effects of each instruction on a target machine, one instruction at a time. Instruction set simulators are attractive for their flexibility: they can, in principle, model any computer, gather any statistic, and run any program that the target architecture would run, including the operating system. They easily serve as back-ends to traditional debuggers as well as architecture design tools such as cache simulators (Bedichek 1990, Darcy et al 1992, Lebeck and Wood 1994).

Naturally, this flexibility comes at a cost—instruction set simulators are often slow, easily over 3 orders of magnitude slower than native execution. Such poor performance severely hampers their practicality, limiting them to toy benchmarks or very patient users. This has prompted several efforts to improve the performance of traditional simulation or to find alternate methods. This work has met with some success: several fast instruction set simulators have been developed over the last several years (Bedichek 1990 and 1995, Cmelik and Keppel 1994, Goldschmidt 1993, Rosenblum et al. 1995, Veenstra and Fowler 1994, Witchel and Rosenblum 1996).

1.2 Instruction Caches

An execution profile showing the frequency of execution of any line of code is a traditional component in performance tuning, and will be meaningful to most experienced programmers. The importance of instruction caches for performance, on the other hand, is less widely known. Relatively recent studies, such as one by the RS/6000 group at IBM (Maynard et al. 1994), has shown that instruction cache behavior is often a significant factor for commercial workloads. In another commercial system study that the author participated in, between a third and a half of potential performance was lost to poor instruction cache behavior (Werner and Magnusson 1997).

The remainder of this paper organized as follows. Section 2 describes SimICS, an instruction set simulator that we will use as our simulator platform. In section 3 we describe the SPECint95 benchmark suite, eight programs which we use as our basis for performance evaluation of the final design. Section 4 is the core of the paper. It describes the algorithms and data structures used to extend the threaded code interpreter core of SimICS. Section 5 presents the performance of SimICS with full instrumentation enabled, and contrasts it with native execution. We conclude in section 6.

2 SIMICS

SIMICS is an instruction set simulator developed at the Swedish Institute of Computer Science (SICS) that simulates multiple SPARC V8 processors and supports multiple physical address spaces, system-level code, and emulation of the SunOS 5.x execution environment (ABI) for direct analysis of user-level programs. SimICS itself is sequential, allowing it to be fully deterministic, a crucial feature for an instrument. SIMICS is publicly available at `http://www.sics.se/simics/`

The main design principle of SimICS is to have a core which is both general and efficient, and thus inevitably very complex, in such a way that even advanced users need only have a rough idea of how it works. This allows SimICS to be reasonably efficient, with a slowdown of 50-200, and yet provide several hooks for end-user extensions that can effectively benefit from this performance.

2.1 Interpreter Core

The core of SimICS is a hand-written threaded-code interpreter. The simplest interpreters execute programs by running a central fetch-decode-execute loop. Threaded code, in contrast, separates the decode and dispatch tasks, thus reducing the cost for decoding and allowing for innovative dispatch techniques (Bell 1973, Klint 1981).

When applied to instruction set simulation, the target program, in object code format, is translated to an intermediate format which is in turn interpreted. Whereas the target instruction set is designed for interpretation by hardware, the SimICS intermediate format is designed to be easy for software. For each intermediate format instruction there is a small segment of code, called a *service routine*, that emulates the effects of that instruction, as well as performing any administrative tasks for the simulation.

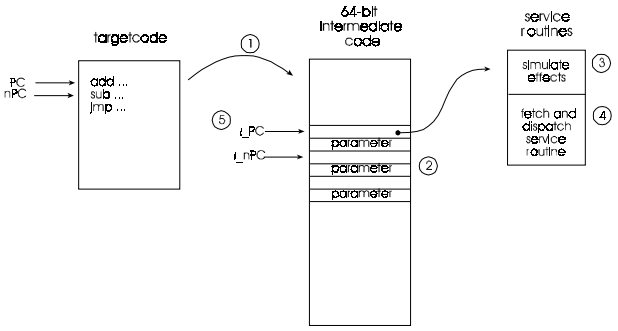

Figure 1: SimICS Interpreter Core

Figure 1 illustrates the main principles. The design is a variation of Bedichek's (Bedichek 1990, Magnusson and Samuelsson 1994). The target code consists of instructions for the target architecture. These are (1) lazily translated to a fixed-sized, 64-bit format (2) consisting of a 32-bit pointer to a service routine (3), and 32 bits of parameter space. These parameters generally contain offsets into a register file (not shown) and/or constants. The choice of a 32-bit parameter field allows for a pointer if more space is required.

The service routine (3) begins execution with its 32 bits of parameter in a specific global register. It does its thing and when finished performs two simple but key operations in its *epilogue* (4). First, it checks for an event by decrementing a time-to-next-event counter, and, if no event is due, fetches a pointer to the next service routine together with its parameter through by using the intermediate code pointer (5), and jumps to the address contained in the first.

Fall-through execution is modeled by simply incrementing the intermediate code pointer. Branch instructions must calculate a new intermediate pointer first. Note that the figure includes two pointers, `PC` and `nPC`, in order to support an architecture with branch delay slots (SPARC 1992).

Most service routines are simple, typically 15-30 host processor instructions, of which 6 constitute the standard epilogue (4). This sets an upper limit on performance for this technique of about 20 times slower than native execution. Achieving significantly better performance than this requires more sophisticated translation, including run-time generation of host code (Magnusson 1993, Witchel and Rosenblum 1996).

Table 1: Characteristics of the SPECint95 Programs

	go	***m88ksim***	***gcc***	***compress***	***li***	***ijpeg***	***perl***	***vortex***
Instructions (1000:s)	512297	131531	1221510	38363	185427	1986703	2361542	2410224
Data cache miss rate	0.12%	0.88%	1.0%	3.4%	1.7%	0.55%	1.4%	0.72%
Instr. cache miss rate	0.11%	0.000%	0.12%	0.000%	0.000%	0.001%	0.000%	0.072%
Unique instructions	12348	2676	33439	722	1485	3118	8140	15339
Size of binary (Kbyte)	750	1138	3352	257	417	1304	906	2920
Instr:s per TLB miss	2464	29326	1268	1444	816860	67951	1868	384
Native execution (sec)	3.2	0.5	8.1	0.1	1.0	8.3	14.5	13.0
Branches (1000:s)	52779	19969	189037	4461	28298	166501	383110	321047
relative on page	75%	68%	73%	79%	65%	67%	68%	65%
relative off page	13%	16%	12%	0.42%	7.6%	20%	13%	16%
absolute on page	1.0%	2.2%	6.2%	21%	20%	1.1%	2.6%	2.2%
absolute off page	11%	14%	9.0%	0.039%	7.5%	11%	16%	16%
annulled instructions	*6688*	*2370*	*29816*	*521*	*3716*	*14954*	*50350*	*22382*

SimICS emulates a SunOS 5.x kernel by explicitly emulating common system calls. This includes support for running multiple programs (multitasking) as well as running programs on several processors (multiprocessing). SimICS can also disable Unix emulation and run system-level code; the SPARC port of Linux can run unmodified on SimICS.

3 BENCHMARKS

The most common use for SimICS that we anticipate is to support advanced performance debugging. A representative workload is therefore the SPECint95 benchmark suite from the Standard Performance Evaluation Corporation. This suite consists of 8 compute-intensive programs that emphasize the performance of the processor and memory system. We've compiled them using GCC version 2.7.2.1, with the "-O2 -g -static" flags. GCC is not the fastest SPARC compiler, but has the advantage of not only being widely available, but old versions tend to be available also, simplifying future repetition of experiments.

The SPECint95 programs can run with one of three input data sets: test, train, and reference. *Test* input is self-explanatory, *train* input is meant for profile-driven compilation, and *reference* input is for computer manufacturers to use for publishable results. We chose to use the *train* input data as these are large enough to be significant yet more manageable for experimentation than *reference* data sets.

Table 1 gives some detailed characteristics on each program. All quantitative data given on the SPECint95 programs in this paper have been gathered using SimICS. The native execution times were measured on an Ultra Enterprise, with four 248MHz UltraSPARC-II processors and 1Gb main memory, using the system *time* facility and taking the median of five runs. The "unique instructions" count is the number of code addresses that were actually fetched. The binary size is of the non-stripped, statically linked program. Instructions per TLB miss indicates the frequency of misses to the address translation cache; the number is the average distance in instructions between misses, which are caused by either instruction or data accesses. All percentages are relative to number of instructions.

For the evaluation in this paper, we've modeled the first-level data and instruction caches of the SuperSPARC processor (16 Kbyte 4-way associative data cache with 32-byte cache lines, and 20 Kbyte 5-way associative instruction cache with 32-byte lines and 64-byte tag allocation). The translation lookaside-buffer we model is fully associative with 64 entries, also corresponding to the SuperSPARC processor.

4 BRANCH SIMULATION

In general when profiling code, we have the choice of counting basic blocks or jumps, or a mixture. An execution diagram consists of a weighted directed graph, where the weights of the nodes correspond to the number of times the corresponding basic block has been executed, and the weights of the arcs to the number of times the corresponding branch was taken. For any given node, the sum of the input arcs equals the weight of the node equals the sum of the output arcs. As shown by Ball and Larus (1992), the node weights can always be deduced by the arc weights, but not vice versa. Also, the minimum cost to gather the necessary information to reconstruct the diagram is generally achieved by a mixture of arc and node counters in the code. However, node counters requires identifying all basic block entry points, which is difficult in the general case. Thus, a practical and reasonably efficient method to generate an execution diagram is to count all taken branches.

Furthermore, to simulate the instruction cache we need to check the validity of all branches, explicitly or implicitly.

Therefore, we choose to perform an operation on every taken branch that:

- increments a counter for that branch,
- checks access rights, and
- checks instruction cache and TLB presence of the target instruction.

We begin by stating some "known" characteristics of mainstream computers:

- instruction cache misses are expensive on the target machine,
- the most common branch type is an on-page, PC-indirect conditional jump, and
- changing page is infrequent.

A "page" here is the largest sequence of instructions guaranteed to be contiguous in both virtual and physical address space. For our target, this is 4kbyte, or 1024 instructions.

Table 1 also shows some statistics from our benchmarks to support our branch-related claims. In the table we classify branches as either relative or absolute, and with targets being either on the same page as the branch instruction or another page (off page). On the SPARC architecture, relative branches are fixed offsets from the program counter, and absolute branches are register indirect. The numbers are all dynamic, i.e. they classify taken branches. For completeness, the last line of the table shows thousands of annulled branches. These correspond to untaken branches where the instruction in the *delay slot* of the branch instruction is ignored (annulled). From Table 1 we can deduce that we should optimize for the case of relative on-page branches, but that none of the branch types are systematically rare enough to be ignored.

Comment: Any system-level simulation involves three address spaces, which we shall call *logical*, *physical*, and *real*. The logical (or *virtual*) address space is the one seen by a traditional, user-level program. For every access, these must be translated to a corresponding *physical* address. The *real* address is the location of simulated data in the virtual address space of our simulator process.

4.1 Branch Simulation Overview

Figure 2 illustrates the major control flow for simulating branches. A service routine is dispatched based on the contents of intermediate code (1). If the branch is taken, the instruction hache table is first checked (2), which, if successful, will immediately provide any information needed. This data structure will be described in Section 4.3 (where we will also explain why *hache* is not a misspelled *hash*).

If this lookup fails, a general branch miss handler is invoked (3). It will confer with several general modules—4, 5, and 6. These all have in common that they may be replaced (dynamically at run-time if so desired) by code written by an end-user of the simulator, using a set of simple programming interfaces.

The first module is the TLB (translation look-aside buffer) which handles translation from virtual to physical addresses (4). In our basic configuration, this simulates a simple, 64-entry TLB with round-robin replacement.

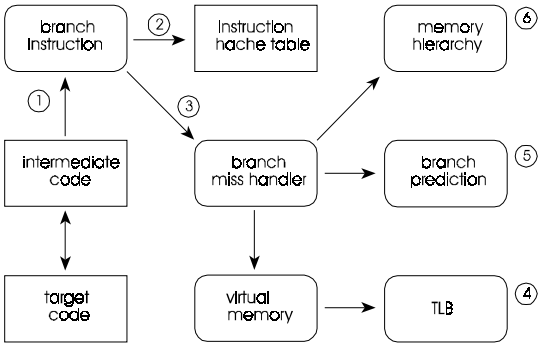

Figure 2: Branch Simulation Overview

The second module is the branch prediction simulator (5), which is optional but gives some support to simulating simple branch prediction schemes.

Finally, the memory hierarchy (6) is passed a memory transaction for an instruction fetch from the target address. This module typically simulates an instruction cache or a shared data and instruction cache, possibly also modeling cache coherency between multiple processors.

The general branch miss handler will update the instruction hache table unless either of modules 4, 5, and 6 have vetoed such insertion. Thus, the hache table serves as a filter, handling the common cases efficiently and invoking complex modules only when something "interesting" occurs.

4.2 Deducing the Execution Profile

We mentioned earlier that a set of branch arc counters are sufficient to deduce the execution graph. We now describe more precisely how this works.

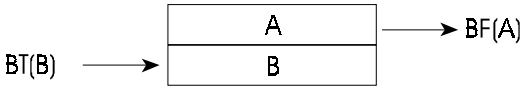

Figure 3: Execution Profile Deduction

An execution profile is a list of instruction addresses and for each instruction a count of how many times it has been executed. Given a set of arc counters, we can deduce a profile for any execution diagram using the following relationship, see Figure 3:

```
E(B) = E(A) + BT(B) - BF(A).
```

A and B are two consecutive instructions, BF(A) is the number of taken branches *from* instruction A, BT(B) is the number of taken branches *to* instruction B, and E(A) is the number of times instruction A was executed.

An execution profile can thus be deduced from a recursive relationship. This recursion starts with BT() at the beginning of a page, because we do not allow "fall through" execution across page boundaries. That is to say, every execution across a page boundary is explicitly represented by a from-to arc.

The situation is complicated in practice by the simulator being interactive and supporting both multitasking and multiprocessing. Thus, there arise several special cases where the nature of the "branch" becomes unclear. For example, a trap instruction can be viewed as an indirect branch over the contents of the trap table. We resolve such issues, as they occur, into a *compensation table* which is permitted to contain branches where either source or target is undefined.

4.3 Instruction Hache Table

Changing pages requires recalculation of intermediate program pointers, including possibly simulating a TLB miss. Within a page, we take advantage of the colinearity of virtual, physical, and intermediate program addresses. Thus, we set:

```
v_diff = PC - (i_PC >> S),
```

where `S` is the difference between the sizes of a target instruction and the intermediate code format, in base 2 logarithm. In our case, target instructions are 4 bytes and intermediate code is 8 bytes, so `S` is 1. We keep the value `v_diff` in a global register. `PC` is the (virtual address) program counter, and `i_PC` its corresponding intermediate code pointer (see Figure 1).

(A variable-length instruction set would require a different scheme for `v_diff`, but the implementation of `c_diff` should be applicable; the important characteristic for `c_diff` is that an intermediate code location corresponds to a unique physical address.)

Thus, as long as we remain on the same page, the virtual address of our program counter can always be reconstructed as follows:

```
PC = v_diff + (i_PC >> S),
```

when we change page we reevaluate `v_diff`, otherwise we thus allow `PC` and `nPC` to be implicit in our intermediate code pointers. This saves updating them for every interpreted instruction. An alternative would be to maintain a separate pointer to the current virtual code page, but in that case we would need a second pointer for the current intermediate page since the latter is arbitrarily aligned—its size cannot be a simple multiple of a host page because it needs special case intermediate code pointers past its end to handle events including fall-through execution to the next page.

Given that we need to update `v_diff` on every change of page, we can use a similar trick to implement a hash table.

Since we wish to count every taken jump, this means that we need to locate a "from-to" pair. We choose to identify these by their physical, as opposed to their virtual, address since thereby we're independent of how the virtual memory is implemented.

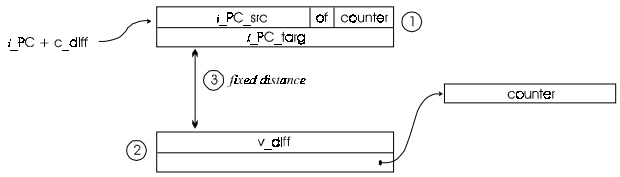

Figure 4: "from" Table in Instruction STC

We locate this pair in a table, see Figure 4. We call this table a *hache* table, since it caches data from a slower, complete data structure using a small hash table. Another similarity with caches is that the presence of an entry can have a semantic implication. The hache table consists of two sets of two words, separated by a fixed distance (3) known at compile time. We make this hache table as large as an intermediate code page. Notice how each entry in the first half (1) is as large as an intermediate code entry in Figure 1. Thus, we can use the same trick as we used with `v_diff`, namely, we form `c_diff`:

```
c_diff = (page(PC) << S) - i_PC +
         code_table_start,
```

which allows us to form the hash table lookup function simply by adding `i_PC` and `c_diff` to get the address of (1). The `page()` function gives the offset on a page, which in our case is the lower 12 bits.

The first set of two 32-bit words contains the source and target addresses of the branch arc (stored pre-translated to intermediate code pointers) and a counter. We can fit a counter since 12 bits of the source address are redundant as the hash table is direct-mapped on 10 bits, and the bottom 2 bits are always zero since instructions are word-aligned. We form a tag comparison as follows:

```
(i_PC_targ ^ target) |
((i_PC_src ^ i_PC) & ~0x7ff),
```

where we use logical operators expressed in C syntax: the "^" operator is bitwise exclusive `or`, "~" is bitwise `not`, and "&" is bitwise `and`. This expression compiles to 4 host instructions.

Essentially, we compare the top 21 bits of the current instruction pointer in parallel with all the bits of the target instruction pointer, and if both match (i.e. the result is zero) then we have a "hit".

Notice that since 12 bits are redundant, comparing 21 bits is one bit too many. We use the 12^{th} bit as an overflow bit, initially setting it to a valid value for the entry (a zero for the first 512 entries, a one for the next 512). When the counter overflows, we get an impossible tag value and will thus "miss" the hash table. The counter thus utilizes 11 bits, and can count up to 2047.

Upon a hit, we have already loaded the counter into a register, and can simply write back the (incremented) value to memory and proceed with the branch.

```
 1 if (take branch)                                  2
 2   target = i_PC + offset;                         1
 3  (i_PC_src, i_PC_targ) = *(i_PC + c_diff)         1
 4   i_PC_src ++;                                    1
 5   if ((i_PC_targ ^ target) |
        ((i_PC_src ^ i_PC) & ~0x7ff) == 0)           5
 6     *(i_PC + c_diff) = i_PC_src;                  1
 7     i_PC = i_nPC;                                 1
 8     i_nPC = target;                               1
 9     dispatch(i_PC);                               4
10   else
11     do_it_the_slow_way();
12 else
13   i_PC = i_nPC;
14   i_nPC ++;
15   dispatch(i_PC)
```

Figure 5: Pseudo code for on-page, conditional branch

Entries are put into this table if the following is true for the *target* address:

- it is allowed to execute,
- it is resident in the instruction cache,
- the target address would hit in the branch prediction table, if simulated,
- it has previously been executed, such that a counter has been allocated and the proper data structures set up, and
- the target virtual-to-physical address translation is valid.

The last two points brings us to the second set of words, (2) in Figure 4. This contains a pointer to a main counter for that branch arc. When the table entry is kicked out, we simply accumulate the cached counter into this main counter, making invalidation a cheap operation.

The other word in part (2) of the data structure contains the `v_diff` value for the target page. The table thus also caches virtual-to-physical translations. This allows register indirect branches—which are infrequent but for a simulator potentially very expensive—to also use the table.

In summary, this 4-word data structure contains the following information for a specific branch arc:

- intermediate code address of source,
- virtual address of target,
- intermediate code address of target,
- counter, up to 2047 taken branches,
- overflow bit for counter, and
- pointer to a main counter, up to 4 billion taken branches.

The physical addresses of both source and target can be derived from the intermediate code addresses since these are unique in the direction intermediate ⇒ physical (they are not unique in the other direction since the simulator may generate different versions). This translation is done by maintaining a sorted list of intermediate page addresses and using unrolled binary search to locate the entry (Knuth 1973).

Figure 5 shows the pseudo code for simulating an on-page, conditional, program-counter-relative jump instruction. As we noted in the beginning of Section 4, this is the worst common branch case, representing between two-thirds and three-quarters of taken branches.

On line (1) in the listing, some condition triggers the branch, typically simulated condition codes. We first calculate the target instruction, using the parameter *offset* which is part of our intermediate format. We next fetch the 64-bit instruction STC entry (3), increment the embedded counter (4), and perform the parallel tag comparison (5). If we hit in this hash table (5), we first write back the top 32-bits of the instruction STC entry containing the updated counter (6), and finally on lines (7) through (9) complete the branch. The numbers on the right of the listing show the number of host SPARC instructions corresponding to a compiled version of the common path in the listing—a total of 17 instructions. Only 3 of these instructions are memory operations.

4.4 Improving Hit Rate

As with any caching-like structure, collisions are a key concern. The performance of the table as described thus far was poor. We take two steps to improve it, which we describe only briefly for lack of space.

We first introduce a “to” table. It is identical in design to the “from” table in Figure 4, except that source and target addresses change places. This allows us to decide whether the from-to pair is faster to find via the source or target address. This also provides a mechanism to avoid some systematic collisions.

Second, and equally important, we increase the size of the tables significantly, to several thousand entries.

Large ISTC tables become sparse if event frequencies are high—for example when modeling small caches or running programs with high TLB miss rate. We therefore need to handle invalidations efficiently.

Invalidations are primarily one of two types invalidation of all entries for a particular *virtual page;* and the removal of an entry relating to a particular *physical address*.

We currently solve this problem by maintaining completely separate data structures. In the case of virtual invalidations, we are aided by the fact that these can only be relevant for cross-page jumps, since on-page jumps will be implicitly valid by the execution of the source instruction.

We have not found a correspondingly elegant insight for physical invalidations, so currently a binary tree is maintained that is consulted for every addition or removal of a valid instruction cache line. This tree currently absorbs over 10% of execution.

Table 2: SimICS Performance

	caches	TLB	***go***	***m88ksim***	***gcc***	***li***	***ijpeg***	***perl***	***vortex***
Native execution (sec)	N/A	N/A	3.2	0.5	8.1	1.0	8.3	14.5	13.0
Native MIPS			160	260	150	190	240	160	190
Sim 1 (sec)	infinite	1024	84.5	19.2	267.7	33.0	216.8	574.5	491.8
MIPS			6.0	6.9	4.6	5.6	9.2	4.1	4.9
Slowdown			x26	x38	x33	x32	x26	x40	x38
Sim 2 (sec)	16k/20k	64	137.2	23.9	584.4	52.9	257.6	810.1	1401.8
MIPS			3.7	5.5	2.1	3.5	7.7	2.9	1.7
Slowdown			x43	x48	x72	x52	x31	x56	x108

4.5 Instruction Cache Modeling

To support instruction cache modeling, we need a second element, in addition to branch target control and profiling. This regards handling fall-through execution across instruction cache line borders.

Again, unfortunately, this is a common operation. In our implementation, we support a granularity of 32 bytes, thus every 8th sequential instruction needs to check that it is permitted to execute.

To implement this, we extend the semantics of our "to" table such that an entry is one of three types: invalid, valid branch arc, and valid instruction cache line crossing. We then insert an artificial instruction in our intermediate code that asserts that the "to" table entry is either a valid branch arc with the instruction cache line as target (in which case it must be valid to fall into), *or* a special cache line crossing entry. The latter will be treated as invalid by the branch handlers. This is done by putting a magic value in `v_diff` in the "to" table, corresponding to (2) in Figure 4.

In uniprocessor mode, we simplify this by using the intermediate code to book-keep which instruction cache lines can be crossed (as suggested by Bedichek 1990). This does not work for multiprocessors, however, unless we wish to replicate all intermediate code. This in turn is probably not a good idea since we will rapidly worsen the data cache performance of our host.

4.6 Multiple Processors and System Level Code

Since the design presented in the previous section supports virtual memory and arbitrary execution flow, it therefore supports running system-level software such as operating systems.

It's ability to support multiple processors is a little more subtle. SimICS uses the same intermediate code for all processors within the same physical address space in order to reduce pressure on the host data cache when simulating large multiprocessors. Notice that only a single (global register) value characterizes the instruction cache and branch arc status of a page of intermediate code, namely `v_diff`. Therefore, we only need to change `v_diff` upon switching simulated processor. This is fortunate since we wish to have a very low overhead for processor switching so as to allow for fine-grained interleaving of events. Currently the cost of switching processor on SimICS is equivalent to 2 or 3 simulated instructions, which allows simulation runs to have an interleaving on the order of 10-50 "cycles" and remain reasonable efficient. Our design will not significantly worsen this performance.

5 EVALUATION

Table 2 shows the relative performance of SimICS over native execution. The timings were performed on an Ultra Enterprise, with the median of 5 *time* measurements shown (we've omitted *compress* since its native execution time was too small). The table shows a range of performance of 26-108 for different configurations of SimICS. This resonates fairly well with our goal of maintaining the historical performance of SimICS of 50-200 while at the same time adding significant new instrumentation.

We show two different runs of SimICS. The first, Sim 1, is with infinite data and instruction caches, and a very large TLB (1024 entries). Thus in Sim 1 the tables are used to maximum effect. In the second configuration, we simulate small on-chip caches (see Section 3).

All simulation runs generated full profiling of data cache read and write misses, instruction cache misses, TLB misses, and branch arc counts. Execution profiling is included in all runs.

As the caches and the TLB gets smaller, the frequency of expensive events increases and worsens our slowdown. The baseline performance with minimum activity is close to the expected peak performance of the interpreter technique that we use, which is in the vicinity of 20. The performance loss for more realistic resource restrictions remains reasonable. The only benchmark with a slowdown worse than 72 is *vortex*, and this is caused by simulating a realistic TLB. Referring back to Table 1, *vortex* encounters TLB misses every 384 instructions! The poor performance for *vortex* could be improved—running *vortex* on SimICS on SimICS (recursively) shows that the data structures being used to handle TLB invalidations behave poorly.

The level of detail provided by the instrumentation during these runs was sufficient to support performance tuning of both parallel and sequential programs (Magnusson and Montelius 1997).

6 CONCLUSIONS

We have presented a significant redesign of the threaded code interpreter core of an existing instruction set simulator, SimICS. The design does not significantly change the overall performance, which has historically been in the range of a slowdown of 50-200 for realistic workloads and significant instrumentation. Despite adding detailed profiling of instruction flow and instruction cache performance, the simulator runs the SPECint95 benchmark suite with slowdowns in the range 26-108 while generating a detailed profile on data cache events, TLB misses, instruction cache misses, and taken branches. The result is an increased range of instrumentation at a similar performance penalty, with direct application in computer architecture studies and performance debugging tasks.

ACKNOWLEDGMENT

The author would like to thank Bengt Werner for valuable comments on earlier drafts of this paper.

REFERENCES

Ball, T., and J. R. Larus. 1992. Optimally profiling and tracing programs. In *Conference Records of the Nineteenth ACM Symposium on Principles of Programming Language,* 59-70.

Bedichek, R. C. 1990. Some efficient architecture simulation techniques. In *Proceedings of Winter '90 USENIX*, 53-63.

Bedichek, R. C. 1995. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the '95 SIGMETRICS Conference,* 14-24.

Bell, J. R. 1973. Threaded code. *Communications of the ACM*, 16(6):370-372.

Cmelik, R. F., and D. Keppel. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the '94 SIGMETRICS Conference,* 128-137.

Darcy, G. A., R. F. Brender, S. J. Morris, and M. V. Iles. 1992. Using simulation to develop and port software. *Digital Technical Journal,* 4(4), 181-92.

Goldschmidt, S. R. 1993. *Simulation of multiprocessors: Accuracy and performance.* Ph.D. thesis, Stanford University, Dept of Electrical Engineering.

Klint, P. 1981. Interpretation techniques. *Software - Practice and Experience,* 11(9):963-973.

Knuth, D. E. 1973. Art of Computer Programming. Vol. 3, *Sorting and Searching*. Addison-Wesley, Reading.

Lebeck, A. R., and D. A. Wood. 1994. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer,* 27(10):15-26.

Magnusson, P. S. 1993. A design for efficient simulation of a multiprocessor. In *Proceedings of MASCOTS'93,* 69-78.

Magnusson, P. S. 1993. Partial translation. SICS Technical Report T93:05, ISSN 1100-3154.

Magnusson, P. S., and D. Samuelsson. 1994. A compact intermediate format for SimICS. SICS Research Report, R94:17, ISSN 0283-3638.

Magnusson, P. S., and B. Werner. 1995. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 62-73.

Magnusson, P. S., and J. Montelius. 1997. Performance debugging and tuning using an instruction-set simulator. SICS Technical Report T97:02.

Maynard, A. M. G., C. M. Donnelly, and B. R. Olszewski. 1994. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of ASPLOS VI*, 145-155.

Rosenblum, M., S. Herrod, E. Witchell, and A. Gupta. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 34-43.

SPARC International, Inc. 1992. *The SPARC Architecture Manual, Version 8.*

Veenstra, J. E., and R. J. Fowler. 1994 MINT: A front end for efficient simulation of shared memory multiprocessors. In *Proceedings of MASCOTS '94*, 201-207.

Werner, B., and P. S. Magnusson. 1997. A hybrid simulation approach enabling performance characterization of large software systems. In *Proceedings of MASCOTS 97,* 73-80.

Witchel, E., and M. Rosenblum. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the '96 SIGMETRICS Conference,* 68-79.

AUTHOR BIOGRAPHY

PETER S. MAGNUSSON is a researcher at the Swedish Institute of Computer Science. He received his MBA from the Stockholm School of Economics in 1991, and his M.Sc. from the Royal Institute of Technology in 1992. His research focus is instruction set simulation, including software engineering issues, efficient instrumentation, and performance modeling.