SICS Technical Report
T97:02

ISRN: SICS-T—97/02-SE
ISSN: 1100-3154

Performance Debugging and Tuning using an Instruction-Set Simulator

Peter S. Magnusson
Johan Montelius

Friday the 13th, June 1997

{psm, jm}@sics.se

Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden

ABSTRACT

Instruction-set simulators allow programmers a detailed level of insight into, and control over, the execution of a program, including parallel programs and operating systems. In principle, instruction set simulation can model any target computer and gather any statistic. Furthermore, such simulators are usually portable, independent of compiler tools, and deterministic—allowing bugs to be recreated or measurements repeated. Though often viewed as being too slow for use as a general programming tool, in the last several years their performance has improved considerably.

We describe SIMICS*, an instruction set simulator of* SPARC*-based multiprocessors developed at SICS, in its rôle as a general programming tool. We discuss some of the benefits of using a tool such as* SIMICS *to support various tasks in software engineering, including debugging, testing, analysis, and performance tuning. We present in some detail two test cases, where we've used SimICS to support analysis and performance tuning of two applications, Penny and EQNTOTT. This work resulted in improved parallelism in, and understanding of, Penny, as well as a performance improvement for EQNTOTT of over a magnitude. We also present some early work on analyzing SPARC/Linux, demonstrating the ability of tools like SimICS to analyze operating systems.*

KEYWORDS: instruction set simulation, profiling, software engineering, performance debugging, SIMICS

1. Introduction

Achieving good performance in software remains an important component of software engineering. Despite tremendous improvements in the raw computational power of computers, the demand for further improvements appears insatiable.

Achieving best possible or even adequate efficiency for a particular program is complicated by the surprising truism that programmer intuition is a poor guide. This in turn is a result of (a) the ease with which a performance problem can be introduced into a program, (b) the size and complexity of real-world programming tasks, and (c) the ever increasing complexity of the underlying hardware.

Performance *debugging* views an unnecessary performance loss in a software system as a design fault and attempts to detect and remove it. Performance *tuning* is a related process, where the programmer attempts to understand where computation resources are consumed in a program, but where the bottlenecks are expected to be more subtle than simple programming errors or naive algorithms. An example of a performance bug is using an unnecessarily expensive library routine, and an example result of performance tuning is the discovery that a single function accounts for the majority of data cache misses.

There is a vast array of tools and techniques to support performance debugging and tuning. Instruction set simulators, today a rare component in such work, offer some particular advantages. Their primary benefit can be summarized rather succinctly: they can, in principle, analyze any program for any computer. Thus a programmer can study otherwise troublesome programs—such as real-time systems, embedded systems, parallel programs, and operating system software. They also allow the programmer to assume any target computer, including a fictitious or planned system (or both!).

A more subtle but equally important characteristic is that a simulator can offer an all-in-one environment. Whereas other tools or combination of tools can essentially answer any significant question relating to performance, a simulator-based tool can provide the data in a single setting and through a single interface. This significantly improves the efficacy of the tool.

In this paper we describe an instruction set simulator, SimICS, and describe some case studies of performance analysis work we have done on a range of applications: a parallel programming language, a sequential integer benchmark, and an operating system. The benefits of a tool such as SimICS is that not only can it analyze such a disparate set of software, but it can gather a wide variety of performance-related statistics within a single environment.

1.1 Road map

Section 2 describes instruction set simulation, and introduces our prototype, SimICS, including a brief presentation of its internals. The core of the paper is in sections 3, 4, and 5. Section 3 presents a number of tools and features in SimICS, with the intention of giving the reader a feel for what type of information and support such a tool can provide. This includes both ready-made commands for the end-user as well as hooks for building more advanced functions.

In sections 4 and 5 we proceed with two detailed case studies of performance debugging and tuning. The first case is Penny, a parallel implementation of a concurrent constraint programming language. Penny is already heavily optimized, despite which SimICS locates a number of problems, and also provides for a quantitative analysis of Penny’s performance on a shared memory multiprocessor. Our second test case, in section 5, is EQNTOTT, a well-known integer benchmark. Here SimICS is an excellent guide in locating a sequence of problems in the program.

Section 6 presents a brief look at the boot phase of SPARC/Linux, demonstrating the applicability of a tool such as SimICS on full system studies. In section 7 we describe the performance of SimICS. We discuss related work in section 8, including alternatives to the simulation approach *per se* as well as some competing simulator designs. Finally, we offer some conclusions in section 9.

2. 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.

For their flexibility, instruction set simulators have long been popular in computer architecture research. There they help designers understand the trade-offs involved in architectural decisions by simulating the effects on user programs.

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. We discuss some of this work in section 8.

Besides the issue of performance, a full implementation is also complicated by the difficulty of recreating the execution environment. To run a given program, we can either emulate the underlying operating system faithfully, or we can bypass this difficulty entirely by running the operating system directly. Unfortunately, the execution environment of modern computers is large. Running the operating system as an “application” is an alternative, but is challenging since this requires faithful emulation of the system-level architecture.

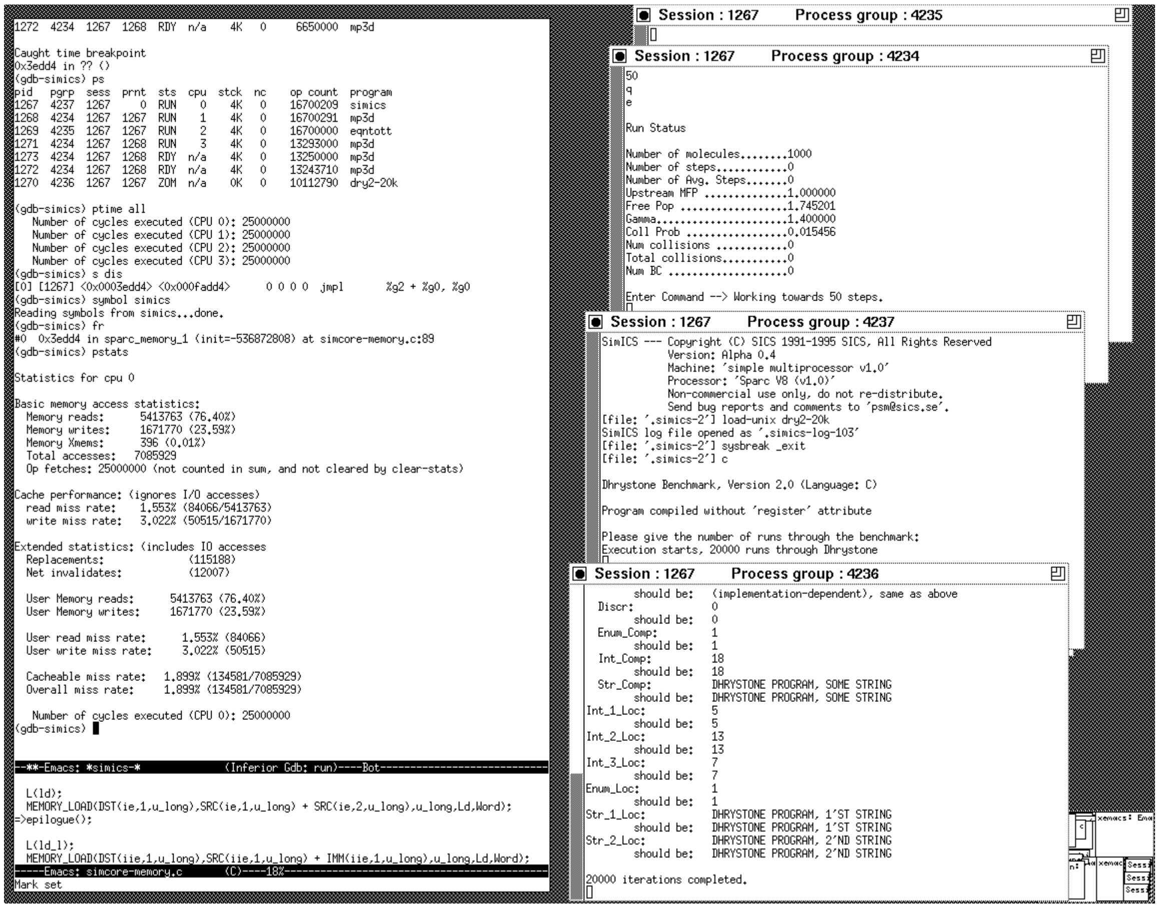
2.1 SimICS

SimICS is an instruction-set simulator developed at the Swedish Institute of Computer Science (SICS). It one or more SPARC V8 processors, and supports multiple physical address spaces, system-level code, and emulation of the SunOS 5.x ABI for direct analysis of user-level programs. The performance of SimICS is fairly acceptable even for large problems, with a slowdown of around 50 per simulated processor. SimICS itself is sequential, allowing it to be fully deterministic, a crucial feature for an instrument.

SimICS allows a program to be studied interactively, both for debugging and for profiling. Of primary interest, SimICS can profile data and instruction cache misses, translation lookaside buffer misses, and instruction counts. These figures can be weighted, sorted, and related to source code lines, allowing the programmer to quickly zoom in on the portions of code that consume resources.

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 [3, 5, 16].

In threaded code, 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.



```
1272  4234  1267  1268  RDY  n/a    4K   0     6650000  mp3d

Caught time breakpoint
0x3edd4 in ?? ()
(gdb-simics) ps
pid   pgrp  sess  prnt  sts  cpu  stck  nc    op count  program
1267  4237  1267     0  RUN    0    4K   0    16700209  simics
1268  4234  1267  1267  RUN    1    4K   0    16700291  mp3d
1269  4235  1267  1267  RUN    2    4K   0    16700000  eqntott
1271  4234  1267  1268  RUN    3    4K   0    13293000  mp3d
1273  4234  1267  1268  RDY  n/a    4K   0    13250000  mp3d
1272  4234  1267  1268  RDY  n/a    4K   0    13243710  mp3d
1270  4236  1267  1267  ZOM  n/a    0K   0    10112790  dry2-20k

(gdb-simics) ptime all
   Number of cycles executed (CPU 0): 25000000
   Number of cycles executed (CPU 1): 25000000
   Number of cycles executed (CPU 2): 25000000
   Number of cycles executed (CPU 3): 25000000
(gdb-simics) s dis
[0] [1267] <0x0003edd4> <0x000fadd4>      0 0 0 0  jmpl       %g2 + %g0, %g0
(gdb-simics) symbol simics
Reading symbols from simics...done.
(gdb-simics) fr
#0  0x3edd4 in sparc_memory_1 (init=-536872808) at simcore-memory.c:89
(gdb-simics) pstats

Statistics for cpu 0

Basic memory access statistics:
  Memory reads:      5413763 (76.40%)
  Memory writes:     1671770 (23.59%)
  Memory Xmems:      396 (0.01%)
  Total accesses:   7085929
  Op fetches: 25000000 (not counted in sum, and not cleared by clear-stats)

Cache performance: (ignores I/O accesses)
  read miss rate:    1.553% (84066/5413763)
  write miss rate:   3.022% (50515/1671770)

Extended statistics: (includes IO accesses
  Replacements:              (115188)
  Net invalidates:           (12007)

  User Memory reads:      5413763 (76.40%)
  User Memory writes:     1671770 (23.59%)

  User read miss rate:      1.553% (84066)
  User write miss rate:     3.022% (50515)

  Cacheable miss rate:   1.899% (134581/7085929)
  Overall miss rate:     1.899% (134581/7085929)

   Number of cycles executed (CPU 0): 25000000
(gdb-simics) █


--**-Emacs: *simics-*            (Inferior Gdb: run)----Bot--------------------------

  L(ld);
  MEMORY_LOAD(DST(ie,1,u_long),SRC(ie,1,u_long) + SRC(ie,2,u_long),u_long,Ld,Word);
=>epilogue();

  L(ld_1);
  MEMORY_LOAD(DST(iie,1,u_long),SRC(iie,1,u_long) + IMM(iie,1,u_long),u_long,Ld,Word);
-----Emacs: simcore-memory.c       (C)----18%-----------------------------------------
Mark set
```

```
Session : 1267       Process group : 4235
```

```
Session : 1267       Process group : 4234
50
q
e

Run Status

Number of molecules........1000
Number of steps............0
Number of Avg. Steps.......0
Upstream MFP ..............1.000000
Free Pop ..................1.745201
Gamma......................1.400000
Coll Prob .................0.015456
Num collisions ............0
Total collisions...........0
Num BC ....................0

Enter Command --> Working towards 50 steps.
```

```
Session : 1267       Process group : 4237
SimICS --- Copyright (C) SICS 1991-1995 SICS, All Rights Reserved
           Version: Alpha 0.4
           Machine: 'simple multiprocessor v1.0'
           Processor: 'Sparc V8 (v1.0)'
           Non-commercial use only, do not re-distribute.
           Send bug reports and comments to 'psm@sics.se'.
[file: '.simics-2'] load-unix dry2-20k
SimICS log file opened as '.simics-log-103'
[file: '.simics-2'] sysbreak _exit
[file: '.simics-2'] c

Dhrystone Benchmark, Version 2.0 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark:
Execution starts, 20000 runs through Dhrystone
```

```
Session : 1267       Process group : 4236
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

20000 iterations completed.
```

Figure 1: Sample session with SimICS

Most service routines are simple, typically 10-30 host processor instructions. 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. We will briefly discuss some of the options in section 6.

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). This Unix emulation mode can be disabled, in which case SimICS will emulate the target machine at the system architecture level (sun4m) allowing operating system code to run unmodified.

The combination of an intermediate format and an emulated operating system interface creates a virtual environment with a great deal of flexibility for adding features of interest to software engineers.

3. Performance debugging using SimICS

The current interface to SimICS is command-line oriented, see Figure 1. The figure shows SimICS simulating a four-processor machine running four programs, one of which has `fork`:ed three times, and one of which is SimICS running another program. We make use of the ability of GDB to support simulator back ends [27] to provide a source code debugging environment. In the figure, the window on the left shows GDB being used to inspect the SimICS process.

(a) instruction cache misses
(b) write cache misses (data)
(c) read cache misses (data)
(d) translation lookaside buffer misses
(e) branches to the instruction
(f) branches from the instruction
(g) count of instruction execution
(h) flag for instruction execution

Figure 2: SimICS profilers

```
(1) profiler_t * unix_tlb_profiling_id;

(2) unix_tlb_profiling_id =
      add_to_profile_services("number of TLB misses",
                              "TLB misses passed on to Unix emulation",
                              "$SIM_TLB_MISS_WEIGHT",
                              sizeof(counter_t),
                              sizeof(uint32),
                              sizeof(uint32),
                              Instruction,
                              NULL, NULL, NULL);

(3) prof_counter_inc(unix_tlb_profiling_id);

(4) int stat_tlb_miss;

(5) exception_type_t
      my_mmu(memory_transaction_t * mmu_reply)
    {
      if (lookup_tlb_entry(mmu_reply) == -1) {
        INC_STAT(stat_tlb_miss);
        if (mmu_reply->data_or_instr == Instruction)
          return code_access_exception;
        else
          return data_access_exception;
      } else {
        mmu_reply->physical_address = tlb_table[i].p_addr +
          T_POFF(mmu_reply->logical_address);
      }
      return No_Exception;
    }

(6) stat_tlb_miss = add_stat_vector("tlb misses",0);

(7) set_mmu(my_mmu, my_mmu_inq, NULL);
```

Figure 3: Examples of gathering statistics

SimICS can be extended at run-time with the "**`load-object`**" command, which uses the Solaris 2.x support for dynamically loadable modules. This is currently used for things such as adding new cache hierarchies, devices, TLB simulator, and command-line interface.

In the rest of this section we describe some of the features currently available in SimICS to support performance tuning. Most of these are made available with simple commands, but it is also relatively straightforward for a user to extend SimICS with new abilities. We will limit the exposition to features that were especially useful in the test cases (sections 4, 5, and 6). The interested reader is referred to the SimICS web pages for more examples.

3.1 Profiling

A profiler gathers and presents statistics that are related to a memory address range. A simple example is an *execution profiler*, which counts how many times an instruction at a particular address has been executed.

Profiler values are shown whenever the user lists source code, see for example Figure 14 at the end of the paper. The profilers currently supported by SimICS includes those listed in Figure 2, as well as profilers for number of reads and writes to an address.

Profilers are the principal tools for program analysis. Therefore, SimICS provides a framework for easily adding new ones. Figure 3 shows some SimICS code that illustrates several features related to program analysis. The top part of the listing (1-3) shows the complete code for implementing profiling of TLB misses that are handled by the Unix emulation code in SimICS. (1) declares a profiler handle, and (2) creates the profiler object. The parameters in (2) both describe and define the profiler, allowing various aspects to be overriden. Since the installed profiler in (2) is simple, only a simple function call is used to increment the profile when the event occurs (3). We'll return to the remainder of the listing later.

```
(gdb-simics) prof-info
Active profilers, from 'left to right':
Column 1:  Instruction cache misses caused by program line
           ($SIM_INSTR_MISS_WEIGHT = 10.000000)
Column 2:  Cache misses (writes) caused by program line
           ($SIM_WRITE_MISS_WEIGHT = 1.000000)
Column 3:  Cache misses (reads) caused by program line
           ($SIM_READ_MISS_WEIGHT = 8.000000)
Column 4:  TLB misses passed on to Unix emulation
           ($SIM_TLB_MISS_WEIGHT = 10.000000)
Column 5:  Number of (taken) branches *to* the code block
           ($SIM_TO_WEIGHT = 0.000000)
Column 6:  Number of (taken) branches *from* the code block
           ($SIM_FROM_WEIGHT = 1.000000)
Column 7:  Count of instruction execution (based on branch arcs)
           ($SIM_PC_WEIGHT = 1.000000)
Column 8:  Number of addresses from which instr have been fetched
           ($SIM_INSTR_WEIGHT = 0.000000)

(gdb-simics) prof-weight 32 20
Weighted profiling results:
  Physical    Virtual    ( source )
  0x00005c20  0x00011c20 (pid 1001) 518199272.00
  0x00005c40  0x00011c40 (pid 1001) 366859495.00
  0x00005c60  0x00011c60 (pid 1001) 335490415.00
  0x00005c00  0x00011c00 (pid 1001)  38342452.00
  0x00005d20  0x00011d20 (pid 1001)  33332216.00
  0x000084a0  0x000144a0 (pid 1001)  21651844.00
  0x00005d40  0x00011d40 (pid 1001)  20545152.00
  0x00005c80  0x00011c80 (pid 1001)   9771702.00
  0x000084c0  0x000144c0 (pid 1001)   7240831.00
  0x00005be0  0x00011be0 (pid 1001)   5890173.00
  0x00006460  0x00012460 (pid 1001)   5768754.00
  0x00005ca0  0x00011ca0 (pid 1001)   4945636.00
  0x00008480  0x00014480 (pid 1001)   4405064.00
  0x000084e0  0x000144e0 (pid 1001)   4155135.00
  0x000064e0  0x000124e0 (pid 1001)   4059607.00
  0x00017b20  0x00023b20 (pid 1001)   3921297.00
  0x00008900  0x00014900 (pid 1001)   3569070.00
  0x00005ba0  0x00011ba0 (pid 1001)   3353840.00
  0x00008c60  0x00014c60 (pid 1001)   3244719.00
  0x00008500  0x00014500 (pid 1001)   3215813.00
Sum:            1397962487.00 (90%)
Not shown:      160930057.00 (10%)
System total:   1558892544.00
```

Figure 4: `prof-weight` listing

```
Counter 5:

  Number of times counter activated: 79949
  Total tick count spent in counter: 1729885

  Detailed counts:

    read operations              254074
    write operations             192874
    xmem (swap) operations       41178
    read cache misses            83
    write cache misses           6296
    xmem (swap) cache misses     4444
    cache replacements           184
    cache net invalidates        2424
    number of TLB misses         467

Counter 6:

  Number of times counter activated: 1
  Total tick count spent in counter: 6

  Detailed counts:

    read operations              2
    cache net invalidates        1
```

Figure 5: Example of a counters

Profilers allow various generic analysis tools to be built on top of them. For example, the **`prof-weight`** command produces the output in Figure 4. Each profiler has an associated weight. The weight name of the instruction count profiler is defined in Figure 3 as **`$SIM_TLB_MISS_WEIGHT`**. Weight values can be interactively changed by the user. In Figure 4, the TLB miss weight has been set to 10 (“Column 4”).

The **`prof-weight`** command thus calculates a linear sum over the entire memory, sorting and displaying the largest values. In the figure, we have asked SimICS to calculate the weights in address intervals of 32 bytes and to display details for the top 20.

3.2 Statistics vectors, counters and other magic

Although not restricted to any particular target program, SimICS allows programs that are being analyzed to pass various types of commands on to SimICS using *magic services*. The interface to all magic services is the same, and passes parameters to SimICS by using a SPARC instruction that is interpreted as a no-op by the real SPARC processor.[1]

Three magic services are especially useful: breakpoints, clear statistics, and *counters*. Counters are accumulators that can be turned on or off. Figure 5 shows example output from using counters. The data listed is from a *statistics vector*. There is one global statistics vector per processor, which is updated when the listed events occur. Other tools (such as counters) can sample or otherwise copy and manipulate statistics vectors.

[1] This works since several different instructions map to no-op, and SimICS can choose to discern among them. Also, a header file makes the use of this feature simple for the programmer.

Counters are useful when we want to study events in particular segments of code, or exclude such events from global aggregates, but where it is unpractical to do so manually. Typically, such portions of code are well isolated in the program source, in macros or procedures, and are thus easily instrumented.

Sometimes we want events reflected both in statistics vectors and as profiles. The second half of Figure 3 (5-7) extends the statistics vector with an entry for TLB misses: (4) declares a stat handle, and (6) creates it. The statistic is used by a user-defined TLB simulator (5), which in turn is installed using one of several hooks (7) to modify or extend SimICS.[2] The example shows the complete glue coded needed to model a new TLB. The macro **`INC_STAT()`** increments the new statistic, propagating the effect to active counters etc, perhaps contributing to the value 467 in Figure 5.

3.3 Branch profiles

The execution profile produced by SimICS includes an exact count of all taken branches. These are listed for any code of interest, and can also be studied by simple queries, as in the example in Figure 6.

The example is for the **`cmppt()`** function in EQNTOTT, discussed in section 5, and is a superset of the branch information presented in Figure 14 at the end of the paper. For instance, in Figure 6 we can see that virtual addresses **`0x144a0`** and **`0x144d0`** are the most common reasons for the function **`cmppt()`** being called.

```
(gdb-simics) prof-address 0x5bf8 (0x11c78 - 0x11bf8)
Jumps to 0x00005bf8 - 0x00005c77:
         source address                    target address
  virtual  (  pid   )   physical       virtual  (  pid   )   physical       count
0x00014278 (pid 1001)  0x00008278    0x00011bf8 (pid 1001)  0x00005bf8          3
0x00014314 (pid 1001)  0x00008314    0x00011bf8 (pid 1001)  0x00005bf8       4114
0x000143fc (pid 1001)  0x000083fc    0x00011bf8 (pid 1001)  0x00005bf8       2730
0x00014420 (pid 1001)  0x00008420    0x00011bf8 (pid 1001)  0x00005bf8       2730
0x00014448 (pid 1001)  0x00008448    0x00011bf8 (pid 1001)  0x00005bf8       1383
0x000144a0 (pid 1001)  0x000084a0    0x00011bf8 (pid 1001)  0x00005bf8    1967847
0x000144d0 (pid 1001)  0x000084d0    0x00011bf8 (pid 1001)  0x00005bf8     862814
Jumps within 0x00005bf8 - 0x00005c77:
         source address                    target address
  virtual  (  pid   )   physical       virtual  (  pid   )   physical       count
0x00011c04 (pid 1001)  0x00005c04    0x00011c0c (pid 1001)  0x00005c0c    5683242
0x00011c30 (pid 1001)  0x00005c30    0x00011c38 (pid 1001)  0x00005c38   39419660
0x00011c3c (pid 1001)  0x00005c3c    0x00011c44 (pid 1001)  0x00005c44   41959940
0x00011c40 (pid 1001)  0x00005c40    0x00011c44 (pid 1001)  0x00005c44  111108894
0x00011c48 (pid 1001)  0x00005c48    0x00011c50 (pid 1001)  0x00005c50    2562766
0x00011c4c (pid 1001)  0x00005c4c    0x00011c60 (pid 1001)  0x00005c60  150506068
0x00011c54 (pid 1001)  0x00005c54    0x00011c70 (pid 1001)  0x00005c70      14462
0x00011c5c (pid 1001)  0x00005c5c    0x00011c70 (pid 1001)  0x00005c70    2548304
0x00011c64 (pid 1001)  0x00005c64    0x00011c6c (pid 1001)  0x00005c6c    3120476
0x00011c68 (pid 1001)  0x00005c68    0x00011c28 (pid 1001)  0x00005c28  147385592
Jumps from 0x00005bf8 - 0x00005c77:
         source address                    target address
  virtual  (  pid   )   physical       virtual  (  pid   )   physical       count
0x00011c74 (pid 1001)  0x00005c74    0x0001427c (pid 1001)  0x0000827c          3
0x00011c74 (pid 1001)  0x00005c74    0x00014318 (pid 1001)  0x00008318       4114
0x00011c74 (pid 1001)  0x00005c74    0x00014400 (pid 1001)  0x00008400       2730
0x00011c74 (pid 1001)  0x00005c74    0x00014424 (pid 1001)  0x00008424       2730
0x00011c74 (pid 1001)  0x00005c74    0x0001444c (pid 1001)  0x0000844c       1383
0x00011c74 (pid 1001)  0x00005c74    0x000144a4 (pid 1001)  0x000084a4    1967847
0x00011c74 (pid 1001)  0x00005c74    0x000144d4 (pid 1001)  0x000084d4     862814
```

Figure 6: Branch profile example

3.4 Parallel programming

Writing parallel programs is generally more difficult than writing sequential ones. Debugging parallel applications implemented in traditional languages is notoriously difficult, partly because of the added complexity of visualizing multiple "workers" co-ordinating their efforts, but also because of recurring problems in tracking down bugs. In a real environment, no two executions of a parallel program are the same. Classic tricks of the trade such as debuggers, assertions, or logging, may change the timing sufficiently to hide a bug, a phenomenon sometimes called *heisenbugs*.

[2] See the SimICS Extensions Manual on the SimICS web site for more examples.

Shared memory also introduces new dimensions to performance. These effects, including alignment problems and communication patterns, are difficult to predict. Unfortunately, they can also have significant effects on performance.

SimICS can simulate a multiprocessor with shared or distributed memory. Concurrency of execution is simulated by executing a window of instructions on each processor in a round-robin fashion. This window can be made relatively small, with a size of 10 instructions incurring a performance loss of about a factor 3-4.

The determinism of SimICS allows a parallel execution to be exactly recreated. The scheduling of Unix processes as well as round-robin simulation of simultaneous execution are both carefully designed to be unaffected by any passive commands from the user, including reading state and setting breakpoints. Thus, tricky parts of parallel programs, such as optimized synchronization schemes, can be stepped through in detail without perturbing the execution. The same applies to the gathering of statistics.

4. Parallel Test Case: The Penny System

In this section, we apply some of the features of SimICS to the problem of analyzing and tuning a parallel program. The section is structured as follows: first, we described the program Penny, followed by a brief description of the simulated target architecture.

In subsection 4.3 we describe four examples of performance problems identified or insights gained using SimICS. In subsection 4.4, we attempt a global quantitative view of Penny, concluding that a small number of event types explain a large portion of the execution time. Finally, in subsection 4.5, we discuss the test case, including considering how traditional programming tools would have fared with the issues raised in subsection 4.3.

4.1 Penny

The Penny system is a parallel implementation of the Agents Kernel Language (AKL) [15, 24], a language for Concurrent Constraint Programming (CCP). CCP views an execution as a set of concurrently executing *agents* that communicate through a shared *constraint store*. The language is similar to a data-flow language, the difference being that the agents wait for constraints to be fullfilled rather than data to arrive.

The outcome of a computation does not depend on the order of execution. This allows Penny to successfully find parallelism in AKL programs without the programmer having to write hints into the source code. Called *implicit parallelism*, this greatly simplifies the expression of many algorithms.

Penny is a suitable case study for evaluating the usefulness of tools like SimICS, for several reasons. First, Penny is a platform rather than an end-user program. We expect tools like SimICS to remain difficult to use for performance tuning in the near future, partly because they implicitly expect the user to understand a great deal about computer architecture. Platform applications, such as programming language environments or operating systems, is software that is used to run other applications. They are generally developed by expert programmers, who are not only concerned with performance, but are capable of using complex tools and willing to invest the time.

Second, Penny is a difficult program to analyze. The parallelism in Penny is fine-grained and heterogeneous, and dynamically adapts to the problem. The location of performance bottlenecks can vary greatly depending on the program that Penny is executing.

Finally, Penny is already a carefully optimized system.

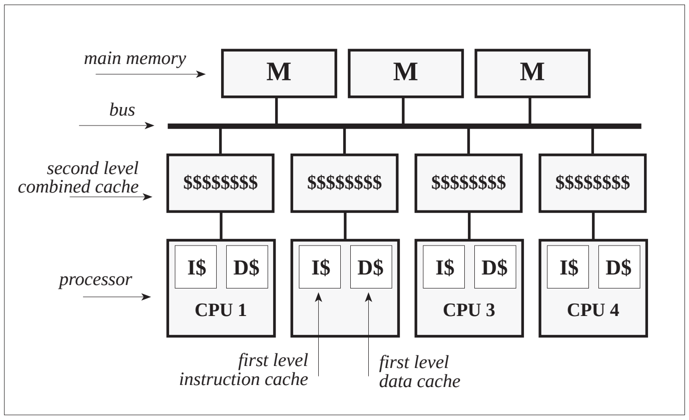


Figure 7: Generic shared-memory multiprocessor

4.2 Shared Memory Multiprocessors

Our workhorses for Penny timings have been two SPARCCenter 2000 (SC2000) multiprocessors with 8 and 20 processors, respectively. The SC2000 is a bus-based shared-memory multiprocessor from Sun Microsystems. Figure 7 is a sketch of a generic shared-memory multiprocessor—though the SC2000 is considerably more complex, the figure serves to highlight some principal features.

Each processor in the figure has two on-chip caches, one for instructions and one for data. On the SC2000's processors, 50MHz Super SPARCs, the data cache is 16KB, four-way associative with 32 byte long cache lines. The instruction cache is 20KB, 5-way associative with 64 byte cache lines.

The processors connect to an off-chip second-level cache of 2Mbytes, direct-mapped with 64-byte cache lines. These caches are connected to a bus with a peak sustainable read/write throughput of 500Mbytes per second.

The first level cache, being on-chip, can react with new data in one cycle. The second level takes 5-10 cycles, whereas accessing the main memory takes 20-60 cycles. A high cache hit rate is therefore crucial to good performance.

SimICS emulates the cache hierarchy in Figure 7, which is close enough to real life to give a good prediction of the performance of an application.[3]

4.3 Analyzing penny

As input to Penny we've selected a small number of AKL program. The programs are used to exercise Penny with a variety of problems. Though running on the same emulator, these programs trigger remarkably different behavior.

Example 1: Detecting sequential components

The first benchmark was a program that solved the towers of Hanoi puzzle. This benchmark is an effective test of how well recursive definitions are executed. When Penny ran Hanoi with four workers under

[3] In fact, during our profiling we initially had significant discrepancies between predicted performance and measured values, until we discovered that both the SC2000 machines we used for timing measurements had faulty SuperSPARC processors with only 4KB caches, not 16KB. The faulty processors had gone unnoticed for several years, despite the machine being used extensively for benchmarking of parallel programs. Therefore, all simulated values for the Penny study assume a 4KB first-level cache with 32-byte cache lines, and a 2Mbyte second-level cache with 64-byte cache lines, both direct-mapped.

SimICS, simulating the first-level cache, we first looked for read cache and TLB misses. A TLB miss generally causes a cache miss on our target machine. These misses stall our CPU.

It turned out that a single assembler instruction caused almost 14% of all read misses. The instruction profiling furthermore revealed that roughly half of all instructions executed were spent in the one enclosing line of C code (specifically, 4 lines of assembler).

The implicated code was not part of the main Penny machinery but was a clean-up procedure that runs after an execution of an AKL program has completed. The code traversed a linked list that was created during runtime. If more than one processor was used the list would be spread across several caches. This traversal then becomes very expensive. Since the result has already been delivered, and the internally calculated execution time reported, the performance bug had never shown up as increased execution time.

The problem occurred with other AKL programs as well, though slightly less dramatic. When running a benchmark that implements the Smith–Waterman algorithm for DNA sequence matching the bug was a limiting factor for the garbage collector. Not only was it unnecessarily slow but it introduced a sequential component in the otherwise parallel garbage collector.

An implementation technique that avoids building the list had been sketched out a year earlier, but the benchmarking techniques that had been used had not seen the traversal as a potential problem. Making this correction to Penny improved performance significantly, as seen for example in Figure 8.

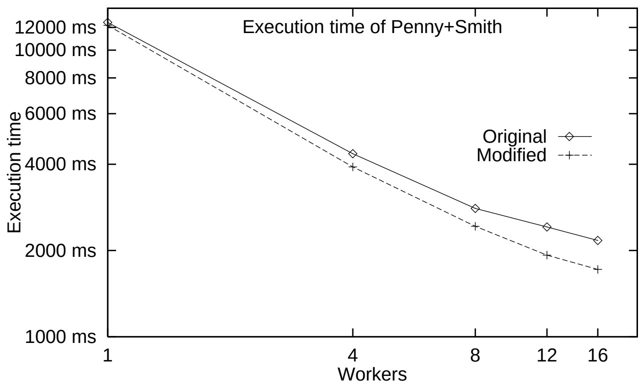


Figure 8: Smith-Waterman before and after a sequential bottleneck was removed (note: log-log scale)

Example 2: Deciding on prefetching

Another issue that drew our attention was a high cache read miss rate for some instructions in the emulator that were used to access AKL data structures. The problem is that the structures are often constructed by one processor and then accessed by another processor.

The remedy to this problem was to add prefetch instructions (coded in C) in all instructions in the emulator that accessed AKL terms. The prefetch would read the next field so that the following instruction would find the value in the cache. The time to read the field would hopefully overlap with useful work.

The fix did not work as expected. The read misses in the instructions did decrease but we had added a large number of read instructions, most of which contributed nothing since the data was already in the first-level cache.

The reason for this was in retrospective obvious. When a larger structure was accessed the following fields would in seven cases out of eight be in the same cache line. The only place where a prefetch instruction would make any sense was in the first instruction that accesses a structure. Removing all but the one effective prefetch left us with an overall performance improvement of 3-4% for several of the workloads.

	Number of workers				
	1	**2**	**4**	**8**	**16**
runtime (ms)	13822	7238	3996	2487	1789
incremental speedup	n/a	1.84	1.84	1.70	1.44
read operations (x 10^6)	124	59.5	29.8	15.2	7.74
read misses (x 10^3)	8.66	121	71.7	43.6	29.0
miss rate	0.007%	0.204%	0.240%	0.287%	0.375%
write operations (x 10^6)	47.5	22.3	11.1	5.60	2.81
write misses (x 10^3)	138	143	80.9	52.8	34.0
miss rate	0.292%	0.643%	0.725%	0.942%	1.211%
Instr. count (x 10^6)	610	3008	156	81	42
Instr. footprint	4840	5584	5789	5991	6104
Bus load indication Mbyte/sec	0.682	4.89	10.0	21.2	35.3

Table 1: Detailed profiling and timing on Penny-1 running Smith-Waterman, with 2Mbyte cache

Example 3: Read and write operations

Though the performance fix we found in example Example 1 improved both performance and speedup (parallelism), there remains a significant sequential element in Figure 8.

Table 1 shows more detail on this particular execution. In the table, we have profiled the improved Penny (called Penny-1) running Smith-Waterman with a growing number of workers. The cache statistics are for a 2 Mbyte cache, i.e. our target machines' second level cache.

The number of read and write operations per processor is a good measure of the amount of work performed, and the figures in the table indicate an uncanny linearity in the load per processor, i.e. no significant overhead is added. Yet speedup (proportional improvement in runtime for every doubling of number of workers) quickly worsens. The data cache statistics provide an explanation.

As we increase parallelism, the miss rate of reads and writes both increase, which is natural since we are spreading work and communicating more. The increasing miss rate explains why the incremental speedup is only 1.84.

A second effect comes into play in larger configurations. The miss rates increase and, at the same time, execution time is decreasing, thus further increasing the intensity of bus transactions. If we estimate bus load by assuming one cache line is communicated for every read or write miss, then the last line shows number of bytes per second the bus needs to carry.[4] It should continue doubling from two processors and up, but comes to a limit. As the load increases, the frequency of bus saturation increases, and incremental speedup worsens to 1.70 and then 1.44. The reduced speedup was thus a fundamental design problem, with no simple fix.[5]

Example 4: How dangerous is a lock?

Penny uses locks in two different situations. The first is when workers move between different parts of the execution state or steal tasks from each other. The second situation is when AKL variables are locked in order to add a new binding or suspension.

[4] Though write misses generally do not cause data to be moved on the bus, they do cause a bus transaction.

[5] A faster memory bus would obviously remove the bottleneck, and measurements over a year later on a next-generation SMP from Sun Microsystems confirmed this.

Both of these situations could cause a hotspot in the implementation. Since all locks are spin locks, a worker stalls if a lock is taken, so it is interesting to know how often the locks are actually missed and how long it takes for a worker to acquire a missed lock.

To get an idea how often locks are missed, *counters* (see subsection 3.2) were placed around the lock primitives.

We ran the Smith-Waterman benchmark on Penny-1 with sixteen workers. The AKL variable lock counters are listed in Figure 5. Counter 5 was for entries into the lock, and counter 6 was entered if there was a collision and one worker had to spin. As we can see, on this cpu there is only one (!) collision out of almost 80 thousand.

These figures indicate that the locks in the Penny machinery are not a performance bottleneck. In fact, it might be worthwhile redesigning some of these locks to be more aggressive in assuming low contention, since over 20 instructions are executed each time the lock is successfully taken.

4.4 Explaining overall Penny performance

The brief examples so far in this section were intended to underline SimICS' ability to "zoom in" on and study performance problems, and to explore design alternatives. The criteria we used for deciding what was "good" were a small number of characteristics, essentially the type of data listed in the *counters* example in Figure 5. That these relatively simple statistics are good guidelines can be shown by correlating them against a large number of performance measurements from the real target machine.

We ran various combinations of Penny—using eight versions of Penny itself (with different improvements and modifications), the four benchmark inputs, and three levels of parallelism (4, 8, and 16 workers). For each of the resulting 128 timing points, we report the median of 31 runs. For each point, we used SimICS to generate 12 aggregate values, covering the different cache and TLB miss types in the target system.

We next selected three variables that we presumed to be important for explaining performance, namely the number of memory reads, number of read misses to the first level cache, and the number of misses to the second level cache. A multiple regression of these variables against the database, and fudging, results in a rather naive prediction of execution time:

$$t_{\text{explained}} = 0.1 * \mathit{Reads} + 0.33 * \mathit{Read\ Misses}_{\text{L1}} + 10 * \mathit{Read\ Misses}_{\text{L2}}$$

In Figure 9 we have plotted the "explained" time against the real time. The correlation is 0.96, not exceptional but clearly a good indicator. Observing the figure, we note that the performance of the large configuration (16 processors) is overestimated and, conversely, the small configuration (4 processors) is underestimated. We suspect the reason for this is that we are lacking a fourth coefficient to measure bus

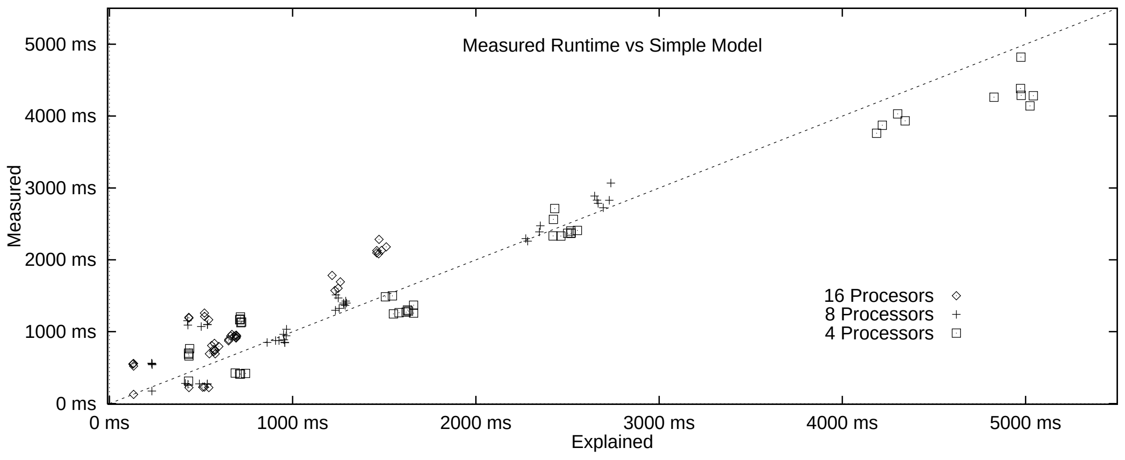


Figure 9: Results of predicting performance using a simple best-fit

contention, an issue that, as discussed in Example 3, becomes significant as the number of communicating processes increases.[6]

4.5 Discussion: parallel test case

We have profiled Penny on SimICS running various AKL programs, and found SimICS to be useful in understanding Penny's behavior under load, and in uncovering performance problems that would have been difficult to locate with more traditional tools. We have also used a large number of runs to demonstrate that a few simple statistics go a long way towards explaining the total run time of a parallel program such as Penny.

Many profilers would have identified the problem described in example Example 1 as a potential performance problem, but would not have explained why—namely that one line of assembler traversing a list misses the second-level cache over 80% of the time, and misses the TLB almost 4% of the time. This in turn was caused by the creation of the list being spread across multiple processors, and would not have been a performance problem in a sequential version of Penny. Without knowing *why* the traversal was expensive, a programmer could easily have concluded that the work involved was a constant—i.e. that traversing the list in batch and maintaining the list as compact as possible was equally expensive.

In Example 2, we were able to use SimICS both to follow where the cache misses moved to, and to quickly quantify the overhead induced by the fix. Note that an optimizing compiler could not have identified this prefetch, since it wouldn't know about the restrictions placed on the sequences of abstract instructions—this required the intervention of the Penny designer, using SimICS to explore trade-offs. Also, without a detailed simulation, the programmer would not be able to see which prefetches were effective without arduous trial-and-error programming of various combinations.

In Example 3, SimICS reports sufficient detail to help us reconstruct what is causing speedup to begin trickling off. We now suspect that for large configuration, we should focus on second-level cache misses. Many of the data structures in Penny have been optimized to be cache-line aligned, etc. The total size of data had not been seen as a significant problem. This analysis shows that it is, and SimICS' source-line profiling of second-level cache misses could be used to evaluate different design changes aimed at reducing the amount of data communicated.

The use of SimICS counters in the fourth and final example greatly simplified instrumentation of the locks. We added half a dozen different types of counters, to a few dozen different procedures and macros. Though it required modification in the source code and recompilation, the binary can run on the real machine unchanged with an insignificant effect on performance.[7] Also, since the time spent within the locks obviously affects the statistics, a non-intrusive method of instrumentation is preferable.

5. Sequential Test Case: EQNTOTT

Our second case study is of a uniprocessor benchmark from the SPECint92 suite. EQNTOTT is an integer-intensive benchmark that translates a logical representation of a Boolean equation to a truth table, and was written at the University of California at Berkeley.

EQNTOTT became the focus of some controversy, as it spends most of its time, around 80%, in sorting routines—primarily in the **`cmppt()`** function. This function compares product terms, returning -1, 0, or 1. Some compilers introduced EQNTOTT-specific optimizations, including vectorized code. Unfortunately, these improvements were not always useful for any other programs, and were sometimes semantically incorrect [11].

[6] This also explains the abnormally high coefficient for read misses to the second level cache, which of course is the one of the three closest correlated with bus contention and thus has to "carry" the bus contention load.

[7] In fact, in the real execution this instrumentation adds less than 4 no-op instructions for every 10000 "real" instructions.

For these reasons, EQNTOTT was dropped in the transition from SPECint92 to SPECint95 [29]. For historical reasons, we've used EQNTOTT to profile the efficiency of SimICS itself. As the profiling abilities of SimICS increased, it became interesting to see how easily SimICS could explain EQNTOTT behavior.

In this section we will use the profile weight method, illustrated earlier in Figure 4. The weights used have varied but are similar to the values in that figure.

	Count			**Misses**			**Time**	
Improvement	**Instructions**	**Memory**	**Branches**	**Reads**	**Write**	**TLB**	**sec**	**rel**
Baseline version	1230695766	232619677	249276233	9166606	94227	548989	19.52	1.00
vectorize **`cmppt()`**	653226389	195891538	99696757	8035639	2224009	623232	12.24	1.59
change data format to bit vectors	366985995	98674847	77231772	2579131	101614	40909	5.53	3.53
use **`putchar()`** instead of **`print()`**	241269891	57448956	46799460	2603297	101092	40823	4.64	4.20
static recalc of **`max_iter`**	225488657	58081624	42380867	2602598	101095	40831	4.58	4.26
specialized vector comparison	205318461	58361433	40819767	2594348	101715	38681	4.28	4.56
defragment prior to print-out	194335772	47176207	40869427	561872	95313	14994	3.49	5.59
defragment prior to sort by **`duple()`**	182232026	48566743	33734596	392081	111672	8258	3.13	6.24
shuffle prior to sort	145809235	38445886	26229718	404240	112686	15186	2.29	8.52
BIT type char	141933950	38205365	25194582	263884	70650	9207	2.18	8.95
various minor touches	120426421	29567471	20881962	265993	70278	8961	1.86	10.49

Table 2: Improvements to EQNTOTT

5.1 Analyzing EQNTOTT

Not surprisingly, SimICS easily located the offending function. Figure 14 at the end of the paper shows the renegade function **`cmppt()`** in great detail. This case study was done over a year after the Penny study, and SimICS could now do instruction cache and exact execution profiling. In the figure, the C code is interspersed with the corresponding optimized assembly code, with detailed profiling data for each line. The numbers in the columns correspond to the profilers listed in Figure 2. To facilitate for the reader, we've added column headings "a" through "h" to the listing.

There are very few instruction cache misses, of course, since this is a small benchmark. Also, this function causes no write cache misses. The cache modeled is the same as the one in section 4.

We can use the details from branch profiling, discussed in section 3, to perform a careful optimization of the function. Firstly, we can deduce a profile of the return values, namely:

Value	**Frequency**
0	1560238
1	7231
-1	1274152

The return values of "equal" and "less than" completely dominate. (We will return to this observation later.)

The central loop is 13 instructions per iteration. In total, we spend close to 1 billion instructions in this routine.

Our first attempt at improvement is a manual vectorization—a rather complex coding task. The performance improvement is significant, however, almost 60 percent, from 19.52 seconds to 12.24 seconds. This essentially corresponds to the EQNTOTT-specific compiler-directed vectorizations mentioned earlier.

This result is shown in Table 2, as the first of a series of improvements over the baseline version. The table contains three groups of columns, the first two being SimICS summary statistics for interesting events, the last set of columns showing the real execution time on our target machine for each version—execution time in seconds and improvement over baseline.[8]

Interestingly, the vectorization produced as a side effect that write cache miss rates skyrocketed. Again, profiling tells us that this was due to the implementation of the inner loop of the new **`cmppt()`**, which used sentinels. These would cause cache write misses—with a vector size of 48, sentinels were a particularly bad idea since they added a fourth cache line to the structure.

We also still have a very high data cache read miss rate, almost 5 percent. Profiling tells us that most of these misses are in **`cmppt()`**. In fact, querying SimICS tells us that 93 percent of read misses are in this routine. This problem was no surprise either, since Lebek and Wood had spotted the same problem using CProf [18].

As pointed out by Lebek and Wood, the data structure could easily be represented using bytes rather than half-words. (The vectors being compared only contain values of 0, 1, or 2, corresponding to zero, one, and dash.) However, we can take this a step further. The comparison routine equates the value 2 with 0. In other words, **`cmppt()`** recognizes only two values, which we might as well represent with binary numbers. This would both improve the vectorization code and reduce the memory footprint.

Thus, before each sort, we first convert and compact all the vectors, and then call a new sorting routine. This cuts execution time so significantly that we are 3.53 times faster than when we started, and with rather simple code. (Much simpler than the manual vectorizations.)

Suddenly, a new routine pops up on top of the profile-weighted cost stack, namely **`memchr()`**. Using our branch profiler, we can trace **`memchr()`** back to **`printf()`**, which in turn is called mostly by **`putpt()`**. This whole chain is caused by **`printf()`** being called over 300000 times with the simple format string "**`%c`**", which prints a single character. We can replace this with the much faster macro **`putchar().`** This cuts execution time a further 25 percent.

This moves our new and improved **`cmppt()`**, called **`cmppt2()`**, back on top. Of the remaining 241 million instructions, approximately 85 million remain in it. Some of the work is recalculation of **`max_iter`**. We can calculate that value statically, doing so removes approximately 15 million instructions.

The next step is to polish this solution slightly. Since vector lengths are easily handled in blocks of 32, we can specialize functions for lengths 32, 64, 96, 128, and a general version. This amounts to simple unrolling, but again of a variety that compilers will generally fail to detect. The various specialized routines are also very simple, only a few lines of code, so this does not overly complicate the program. This reduces execution time by another 5 percent, to 4.28 seconds.

[8] All measurements are the median of 13 runs, running on a SC2000, the target architecture described in section 4.

```
(gdb-simics) list-detail 535,544
535                                             int
536                                               cmppt2_2(PTERM *a[], PTERM *b[])
537                                             {
538                                               register u_long *ap, *bp;
539                                               register u_long aa, bb;
540           (READ) (TLB)
541      1 0 1249938 15166 2841621  0  5683242 2     ap = a[0]->ptand_v;

0x120d0  1 0  157289   468 2841621  0  2841621 1 ld  [ %o0 ], %g2
0x120d4  0 0 1092649 14698       0  0  2841621 1 ld  [ %g2 + 0x14 ], %o2

542      0 0   40047   633       0  0  2841621 1     bp = b[0]->ptand_v;

0x120d8  0 0   40047   633       0  0  2841621 1 ld  [ %o1 ], %g2

543
544      0 0  708213  5553  0 1281383 17049726 6     if ((aa = *ap) == (bb = *bp))

0x120dc  0 0  384887  2380       0  0  2841621 1 ld  [ %o2 ], %g3
0x120e0  0 0  213291  2529       0  0  2841621 1 ld  [ %g2 + 0x14 ], %o0
0x120e4  0 0  110035   644       0  0  2841621 1 ld  [ %o0 ], %g2
```

Figure 10: cmppt2_2() listing

`cmppt2_2()`, which handles vector lengths of 32 to 63, is now completely dominated by data read misses and TLB misses, see the third and fourth columns in Figure 10.

These misses are in turn caused by multiple levels of indirection which fragments the data structure. We rewrite the print-out routine to first copy to and from a better structure. We are now 5.59 times faster than when we started. We do a similar improvement to another comparison routine, **`cmppth()`**, which is also used as a comparison for a sort (in **`duple()`**). This brings us to 6.24 times faster than baseline.

Let us now return to the frequency table at the beginning of this section. The emphasis on equal or less-than return values would indicate that the sorting routine doesn't work particularly well. Indeed, a closer analysis shows that the "optimized" quicksort routine in EQNTOTT is forced to deal with data that is largely reverse-sorted, triggering worst-case quadratic behavior. Interestingly enough, nobody appears to have seen this problem before. There are many approaches to enhancing quicksort to tolerate various input. We choose the rather simplistic approach of randomly shuffling any input.[9] Doing so cuts the total number of comparisons by almost two thirds. Unfortunately, we are now digressing from the reference output of EQNTOTT. Since the principal sort equated zero and dash, the result of a sort is dependent on the order of comparisons.

Next we introduce the more compact data format, using a byte rather than halfword for each logical variable, as suggested by Lebek and Wood. This cuts another 5 percent.

Finally, we polish slightly: we unroll another routine, and enhance quicksort to better handle word-aligned objects.

At this point we stop adding improvements, not because there are no further to be easily found (there are), but because we've passed the psychologically satisfying limit of one magnitude performance improvement.

5.2 Discussion: sequential test case

As Table 2 shows fairly clearly, a sequence of performance enhancements not only move bottlenecks between different parts of a program, but also changes their nature. For example, the first vectorization caused an increase in write cache misses; defragmentation reduces TLB and read cache misses; static calculation of a value reduces instruction and branch count. For a tool to support a long sequence of program changes, it will need to model a disparate set of events, and to allocate these to particular sections of a program in detail.

Several of the improvements we've made to EQNTOTT are non-trivial, but the detailed profiling of SimICS made their implementation and verification much simpler.

[9] It's difficult to implement a smarter solution without being prejudiced by the particular data distributions in EQNTOTT.

6. System Test Case: Linux

Our third and final case study is a brief analysis of the boot phase of SPARC/Linux. Operating systems are particularly difficult to analyze, since apart from being complex, they consist of code from a mixture of sources—such as C, C++, and hand-written assembler—and they expect a machine-level view of their host.

Since instruction set simulators are a software-only solution to target architecture modelling, there is no difficulty in principle to emulate the target faithfully enough to “fool” an operating system—the difficulty lies in the details [3,19,26]. A *sun4m* architecture model for SimICS has been implemented that is sufficiently accurate to boot Linux (see Acknowledgements). It uses runtime loadable modules to SimICS for each of the devices and MMU (SPARC Reference MMU).

The benefit of using a simulator to study an OS is that it cuts through all the layers of abstractions and can show a simple, flat model of events on the target machine. For example, in Figure 12 the **`devs`** command shows a count of accesses to the memory-mapped devices during the boot phase of Linux. “dma” and “esp” are closely related and are used for disk I/O. “tty” is of course the text output. Further detail can be given by the **`io`** command, which lists information on the last accesses from a history buffer showing time of access, processor, program counter, etc.[10]

We’ve looked superficially at the first 55 million instructions of the boot. To get an overview of what code

```
(gdb-simics) devs
[ 1] [ 19714] from 0x08400000 to 0x08400ffe is the dma
[ 2] [ 12484] from 0x08800000 to 0x08800ffe is the esp
[ 3] [     7] from 0x10000000 to 0x10002ffe is the iommu
[ 4] [     0] from 0x50200000 to 0x50200ffe is the display
[ 5] [     0] from 0x50300000 to 0x50300ffe is the display
[ 6] [     0] from 0x50700000 to 0x50700ffe is the display
[ 7] [     0] from 0x50800000 to 0x508fdffe is the display
[ 8] [    79] from 0xd0000000 to 0xd0000ffe is the zs
[ 9] [   389] from 0xd0001000 to 0xd0001ffe is the zs
[10] [    21] from 0xf1200000 to 0xf1201ffe is the eeprom
[11] [     1] from 0xf1400000 to 0xf1403ffe is the interrupt
[12] [   368] from 0xf1410000 to 0xf1410ffe is the interrupt
[13] [     2] from 0xf1900000 to 0xf1900ffe is the auxio
[14] [     1] from 0xf1d00000 to 0xf1d03ffe is the counter
[15] [   890] from 0xf1d10000 to 0xf1d10ffe is the counter
[16] [  3807] from 0xfec01000 to 0xfec01ffe is the tty
[17] [     1] from 0xfec02000 to 0xfec02ffe is the info
[18] [    24] from 0xfec03000 to 0xfec03ffe is the elf_loader

(gdb-simics) io 1
 19714: cpu0 ip=0xf0084e54 st    data=0x00000210 addr=0x08400000 dma+0x0
 19713: cpu0 ip=0xf0084e4c ld    data=0x00000200 addr=0x08400000 dma+0x0
 19712: cpu0 ip=0xf0084e38 st    data=0xfff00000 addr=0x08400004 dma+0x4
 19711: cpu0 ip=0xf0084e0c st    data=0x00000200 addr=0x08400000 dma+0x0
 19710: cpu0 ip=0xf0084e04 ld    data=0x00000000 addr=0x08400000 dma+0x0
 ...
```

Figure 12: Device accesses in Linux boot

```
(gdb-simics) prof-page-map
Profile stats per page (zeroes not listed):
Physical
0x00004000    56829
0x0000f000  1477856
0x00010000    86855
0x00011000 15805107
0x00012000   270667
...

(gdb-simics) prof-page-map 0x11000
Profile stats for page 0x00011000 (zeroes not listed):
Physical
0x00011000        0
0x00011100     2006
0x00011200    24100
0x00011300      355
0x00011400    27070
0x00011500    16685
0x00011600      160
0x00011700 15239915
...
```

Figure 11: Linux boot statistics

has been executed, we can use the **`prof-page-map`** and **`prof-page-details`** commands, see Figure 11. Here we’re benefiting from SimICS’ focus on physical addresses. The only profile shown is the number of instructions executed. The first command summarizes over physical pages, and the second command allows a zoom into one page, aggregating in groups of 256 bytes (64 instructions).

Disassembling the offendor in the block at 0x11700 reveals **`udelay()`** as the culprit. Optimizing a delay loop is obviously not very intelligent, but using branch data quickly reveals that of the 25472 times that **`udelay()`** is called (for a total execution of 15 million instructions), 25000 calls are from function **`keyboard_zsinit()`**. This latter function is trying to detect a keyboard, but times out after occupying the machine for 2.5 seconds. Perhaps this can be done in a better manner.

The second large block of execution is schedule(), absorbing another 13 million instructions. This in turn is mostly called from **`idle()`**, so thus does not offer any good opportunity for improvements.

[10] These commands were inspired by similar ones in g88.

	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

Table 3 - SimICS Performance

The third and final block is interesting. It occurs in the loaded user binary that is the first process to run, a shell. SimICS doesn't distinguish between data and code any more than hardware does, so it easily profiles code copied into memory by a simulated SCSI device. We can set a watchpoint on the write and SimICS will stop execution after the SCSI operation that loaded the page. Next we can make the watchpoint more precise by requesting it to break only on instruction fetches, upon which we can switch symbol table within GDB and debug the user process (assuming we have a separate copy of that binary). This works because SimICS always presents the front-end with the "current" view of the execution. In this case, there is a byte-oriented loop that would benefit from some careful unrolling.

Naturally, what is interesting with OS-level profiling is to study the kernel under heavy load. The analysis in this section provides some minor improvements, but above all demonstrates the applicability of performance debugging using simulators to operating system code.

7. SimICS Performance

Table 3 shows the performance of SimICS running the SPECint95 benchmark suite, a set of integer intensive programs.[11] The table shows only data for uniprocessor modelling. Two sets of running times are given, the first (Sim 1) is a baseline run modelling infinite caches and a very large TLB, thus minimizing the number of events that need to be logged. The second set of times (Sim 2) correspond to modelling the SuperSPARC processor, which is our target model (discussed in subsection 4.2). It gathers all the statistics and profiles used in the preceeding sections.

As the table shows, SimICS runs 30-100 times slower than native execution when gathering the type of information needed for performance tuning.

8. Related work

Using a simulator to support programming is not a new concept. Indeed, it was invented at the very dawn of programming [10]. Among its modern uses, supporting operating system development [3, 9, 30] and studying memory hierarchy behavior [6, 23] bear special mention.

Nor is simulation the only alternative.

Figure 13 shows the traditional sequence for a compiled imperative language, the dominant programming technology today. A program begins as an algorithm and moves through various intermediate formats and tools before ending up as part of the execution environment in the form of a process image. The terms encapsulated by the arrow represent different well-defined formats, proceeding from a general level of

[11] The measurements were done using the system *time* facility, taking the median of 5 runs on an Ultra-Enterprise with four 248MHz UltraSPARC processors. The benchmark uses the *train* input data. The *compress* benchmark was omitted since it was too short.

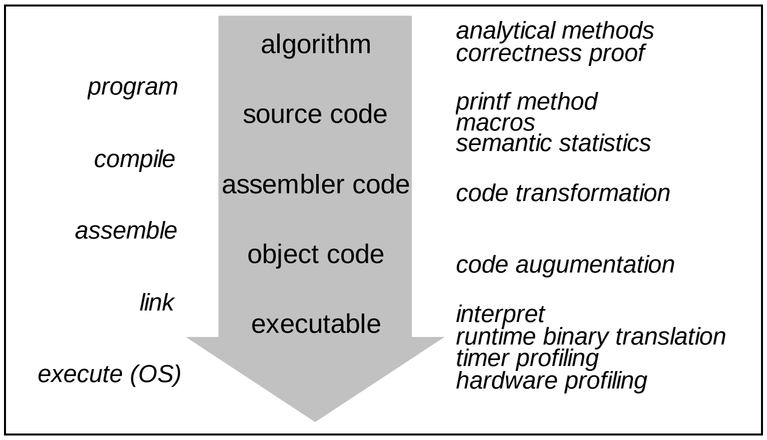


Figure 13 - Traditional programming using a compiled imperative language

abstraction to increasing specificity with respect to the target computer. The left side of the figure lists the tools and processes used to transform each format to the next.

Each level of the sequence offers an opportunity to analyze program behavior with respect to performance. The right side of the figure lists some examples of analysis techniques that may be applicable. For instance, the *printf method* at the source code level refers to the technique of inserting print statements in a program to trace events of interest. An example of *code transformation* at the assembler level is replacing all memory accesses with procedure calls that simulate the effects on a cache. An example of *code augmentation* at the object code level is the insertion of counters in each basic block to produce an accurate execution profile. The bottom level, the executable, is of particular interest.

8.1 The Executable Phase

The executable is the final format of a program before it becomes a process image. An executable is a set of binary instructions for a target computer system. The instructions are from a particular instruction set, and assume a particular execution environment. There are some unique benefits to analyzing a program at this level.

First of all, the format and semantics of executables are defined by the target computer system architecture and therefore independent of the tools and languages that went into their making. Program analysis at this level is therefore also independent of which tools have been used—i.e. language, preprocessor, compiler, assembler, or linker. This includes allowance for a mixture of code sources, which is important since real workloads are often of that nature.

Furthermore, the operating system itself is a form of executable. If a technique for analyzing a user-level executable can be extended to also analyze the operating system, then any target workload can be studied. Also, analyzing the interaction of a program with the operating system, or simply the operating system itself under a realistic load, are important activities. This is especially true given that the steadily growing set of features offered by operating systems lead to their interaction with an application taking on a growing proportion of the execution time.

Given the benefits of analysis at this level, it is natural that several techniques have been developed to this end. It is perhaps also not surprising that no single technique has proved completely satisfactory.

8.2 Analyzing the executable

When analyzing the behavior of a software system, we need to do three things:

1. execute the actual instructions specified by the executable binary
2. perform the system services required by the program, if any
3. generate information about the execution over and above the actual program result

At the level of executable, there are essentially three strategies to solving the first problem:

- host-supported simulation
- execution-driven simulation
- instruction set simulation

Orthogonal to choice of simulation strategy is the choice of whether to generate traces for post-processing, called *trace-driven simulation*, and choice of methods for visualizing or otherwise analyzing the resulting data.

8.3 Host-supported simulation

When available, hardware monitors or other special host hardware features can be used to study programs. Called host-supported simulation, this is generally the fastest approach. Examples include WWT [25], early SimOS designs, Mannequin [9], and VTune [1].

Restrictions inherent in host-supported simulation, other than sheer availability, is selection of what statistics to gather; the hardware support will dictate what studies can be done. (This can result in bizarre effects—for example, WWT cannot model stack references.)

Another fundamental restriction is that the platform must be available. This precludes usage for future designs. It also precludes design-space oriented studies, i.e. studying a program in terms of a range of architecture parameters such as cache sizes.

Finally, detailed profiling is generally not possible, since most host-supported techniques are based on sampling.

When applicable and sufficient, host-supported simulation is the fastest, most accurate, and often simplest to implement.

8.4 Execution-driven simulation

Execution-driven simulation involves running a modified program binary. The modifications can be induced at any stage during generation of the binary, either by modifying intermediate program formats (source code, assembly code, object file, or executable binary) or any of the compiler tools (preprocessor, compiler, assembler, or linker). Examples include gprof [13], Purify [14], Tango [12], and QPT [17].

Execution-driven simulation is generally faster than instruction set simulation, but with more restrictions. Restrictions in common for all execution-driven approaches that the authors have found includes lack of support for one or more of: run-time generated code, multiple processes (several different programs executing simultaneously), multiple address spaces (such as clusters), system-level (operating system) code, determinism, and interactive control of the execution. Approaches that relax these constraints tend to impose new restrictions on the type of programs that can be studied, including programming paradigms and/or source code modifications.

This strategy remains the most common, however, since it is generally the simplest to implement and is often sufficient.

8.5 Instruction set simulation

Also called instruction-level or program-driven simulation, instruction set simulation is the brute-force approach, whereby each instruction in the program is simulated one at a time. This provides an accessible and in some sense correct target machine model for instrumentation, and places minimum restrictions on the architectural relationship between the host and target. Examples include g88 [3], CacheMire [7], Mint [28], Shade [8], SimOS [26, 30], Talisman [4], and SimICS.

All these simulators translate from a target code to an intermediate format. This format can then either be interpreted [3, 4, 19] or directly executed [2, 20, 26, 28, 30].

Instruction set simulation is generally the slowest but most flexible approach.

8.6 SimICS

SimICS is an instruction set simulator that has borrowed many design principles from g88 [3]. SimICS takes the brute force approach to all three problems mentioned earlier in this section. For this, SimICS takes two penalties—first, we accept a lower performance than specialized approaches. This impact is today on the order of 5-10, or a slow-down of approximately 30-100 per simulated processor. Second, we need to deal with a significantly more complex software engineering problem in building the simulator. This effect is, of course, difficult to quantify, but it is significant.

The contribution of SimICS has been to achieve a competitive performance point while at the same time avoiding all the restrictions listed above for host-supported or execution-driven simulation. This work has included novel techniques for memory simulation [21] and instruction cache profiling [22].

9. Conclusion

An instruction set simulator provides a completely maleable, artificial execution environment with particular benefits to software engineering. We have described SimICS, a research prototype simulator with several features useful for program analysis, in particular performance tuning, as well as traditional debugging facilities. We’ve selected two programs of interest, Penny and EQNTOTT, and demonstrated the efficacy of a tool such as SimICS to support the process of understanding and improving a program. We’ve also briefly shown that the techniques are applicable to full system studies.

10. Acknowledgments

Bengt Werner co-designed much of the front-end semantics to SimICS, promoted the idea of simple statistics to explain performance, patiently isolated many bugs, and has in general been an invaluable discussion partner and senior SimICS user. Anders Landin has been an enthusiastic supporter of SimICS and has contributed much to the discussion of what a user needs to know about program/architecture interaction. David Samuelsson wrote much of the SPARC V8 interpreter. Henrik Forsberg wrote much of the Unix emulation. Thanks to Peter Fritzson at Linköping University for access to a 20-processor SC2000 for the Penny timings. Thanks to David Keppel for the reference to [10]. Special thanks to Robert Bedichek, author of g88 and thereby providor of a starting point for SimICS' design and for many ideas and discussions. Also, both David and Robert provided detailed comments on an earlier version of this paper.

The Linux port to SimICS was done with Jim Nilsson, Magnus Karlsson, Fredrik Lundholm, and Fredrik Dahlgren at the Department of Computer Engineering, Chalmers, and Håkan Grahn at the Department of Computer Science and Business Administration, University of Karlskrona/Ronneby.

Various parts of this work have been sponsored by Ellemtel and Ericsson in the Enterprise, Hubble, and SimZOY projects, Sun Microsystems in the SOS project, the European Commission in the GPMIMD and SODA projects, and SICS. SICS is sponsored by CelsiusTech, Ellemtel, Ericsson, IBM, Sun Microsystems, the Swedish Defense Material Administration (FMV), Telia, and the Swedish National Board for Industrial and Technical Development (Nutek).

```
(gdb-simics) list-detail 34,59
34                                                              int cmppt (a, b)
35                                                              PTERM *a[], *b[];
36                                                              /*
37                                                                 * compare product terms indirectly pointed to
by a and b.
38                                                               */
39                                                              {
40                                                                      register int i, aa, bb;
41       a b       c      d        e        f         g  h
42       1 0 2384615 150991  5683242  2841621  28416210 10              for (i = 0; i < ninputs; i++) {

0x11bf8  0 0       0    121  2841621        0   2841621  1 sethi  %hi(0x3b400), %g2
0x11bfc  0 0    3996   8207        0        0   2841621  1 ld  [ %g2 + 0x224 ], %g3     ! 0x3b624 <ninputs>
0x11c00  1 0       0      0        0        0   2841621  1 cmp  %g3, 0
0x11c04  0 0       0      0        0  2841621   2841621  1 ble,a   0x11c70 <cmppt+120>
0x11c08  0 0       0      0        0        0         0  0 clr  %o0
0x11c0c  0 0  208116   8109  2841621        0   2841621  1 ld  [ %o0 ], %g2
0x11c10  0 0 1543507  85754        0        0   2841621  1 ld  [ %g2 ], %o3
0x11c14  0 0   75485   5398        0        0   2841621  1 ld  [ %o1 ], %g2
0x11c18  0 0       0      0        0        0   2841621  1 clr  %o2
0x11c1c  0 0       0      0        0        0   2841621  1 sll  %g3, 1, %o1
0x11c20  0 0  553511  43402        0        0   2841621  1 ld  [ %g2 ], %g2

43       0 0 1567937 229925        0        0   2841621  1                      aa = a[0]->ptand[i];

0x11c24  0 0 1567937 229925        0        0   2841621  1 ldsh  [ %o2 + %o3 ], %o0

44                                                                              bb = b[0]->ptand[i];
45       0 0 1436933 110481 73692796 19709830 229603251  3                      if (aa == 2)

0x11c28  0 0       0      0 73692796        0  76534417  1 cmp  %o0, 2
0x11c2c  0 0       0      0        0        0  76534417  1 bne  0x11c38 <cmppt+64>
0x11c30  0 0 1436933 110481        0 19709830  76534417  1 ldsh  [ %o2 + %g2 ], %g3

46       0 0       0      0        0        0  56824587  1                              aa = 0;

0x11c34  0 0       0      0        0        0  56824587  1 clr  %o0

47       0 0       0   2443 19709830 20979970 208623281  3                      if (bb == 2)

0x11c38  0 0       0   2443 19709830        0  76534417  1 cmp  %g3, 2
0x11c3c  0 0       0      0        0 20979970  76534417  1 be,a   0x11c44 <cmppt+76>
0x11c40  0 0       0      0        0        0  55554447  1 clr  %g3

48                                                                                      bb = 0;
49       0 0       0   5631 20979970 76534417 228321868  3                      if (aa != bb) {

0x11c44  0 0       0   5631 20979970        0  76534417  1 cmp  %o0, %g3
0x11c48  0 0       0      0        0  1281383  76534417  1 be,a   0x11c60 <cmppt+104>
0x11c4c  0 0       0      0        0 75253034  75253034  1 add  %o2, 2, %o2

50       0 0       0      0  1281383     7231   2562766  2                              if (aa < bb) {

0x11c50  0 0       0      0  1281383        0   1281383  1 bge  0x11c70 <cmppt+120>
0x11c54  0 0       0      0        0     7231   1281383  1 mov  1, %o0

51                                                                                              return (-1);
52                                                                                      }
53                                                                                      else    {
54       0 0 3210112  14722 75253034 76527186 226747168  5                                      return (1);

0x11c58  0 0       0      0        0        0   1274152  1 b  0x11c70 <cmppt+120>
0x11c5c  0 0       0      0        0  1274152   1274152  1 mov  -1, %o0
0x11c60  0 0       0      0 75253034        0  75253034  1 cmp  %o2, %o1
0x11c64  0 0       0      0        0  1560238  75253034  1 bl,a   0x11c28 <cmppt+48>
0x11c68  0 0 3210112  14722        0 73692796  73692796  1 ldsh  [ %o2 + %o3 ], %o0

55                                                                                      }
56                                                                              }
57                                                                      }
58       0 0       0      0  1560238        0   1560238  1              return (0);

0x11c6c  0 0       0      0  1560238        0   1560238  1 clr  %o0

59       0 0       0      0  1281383  2841621   5683242  2      }

0x11c70  0 0       0      0  1281383        0   2841621  1 retl
0x11c74  0 0       0      0        0  2841621   2841621  1 nop
```

Figure 14: Detailed listing of the `cmppt()` function in EQNTOTT

References

[1] M. Atkins and R. Subramaniam. Pc software performance tuning. *IEEE Computer*, 29(8):47-54, August 1996. http://developer.intel.com.

[2] R. C. Bedichek. Talisman 2 - A Fugu System Simulator. Personal Communication and web documentation, 1995.

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

[4] R. C. Bedichek. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the '95 SIGMETRICS Conference,* 1995.

[5] J. R. Bell. Threaded Code. *Communications of the ACM,* 16(6):370-372, June 1973.

[6] M. Brorsson. Sm-prof: A tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *Proceedings of the 1995 ACM SIGMETRICS Conference*, pages 178-187, May 1995.

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

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

[9] G. A. Darcy, R. F. Brender, S. J. Morris, and M. V. Iles. Using Simulation to Develop and Port Software. *Digital Technical Journal,* 4(4), 1992.

[10] S. Gill. The Diagnosis of Mistakes in Programmes on the EDSAC. In *Proceedings of the Royal Society Series A, Mathematical and Physical Sciences,* Volume 206, pages 538-554. Cambridge University Press, May 1951.

[11] C. Glaeser. EQNTOTT-Specific Optimizations. Technical report, Nullstone Corporation, 1996.

[12] S. R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance.* Ph.D. thesis, Stanford University, June 1993.

[13] 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.

[14] 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.

[15] S. Janson. *AKL A Multiparadigm Programming Language*. PhD thesis, Uppsala Thesis in Computing Science 19/ SICS Dissertation Series 14, 1994.

[16] P. Klint. Interpretation Techniques. *Software - Practice and Experience*, 11(9):963-973, September 1981.

[17] J. R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *Software–Practice and Experience*, 24(2):197-218, February 1994.

[18] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer,* 27(10):15-26, October 1994.

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

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

[21] P. S. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[22] P. S. Magnusson. Efficient Instruction Cache Simulation and Execution Profiling with a Threaded-Code Interpreter. In *Proceedings of WSC'97*. (To appear.)

[23] M. Martonosi, A. Gupta, and T. Anderson. Mem-Spy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference*, 1992.

[24] J. Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala Thesis in Computing Science 28 / SICS Dissertation Series 25.

[25] 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 the 1993 ACM SIGMETRICS Conference,* May 1993.

[26] M. Rosenblum, S. Herrod, E. Witchell, and A. Gupta. Complete Computer System Simulation: the SimOS Approach. *IEEE Parallel and Distributed Technology,* 1995.

[27] 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.

[28] 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.

[29] R. Weicker. An Example of Benchmark Obsolescence: 023.eqntott. Technical report, Standard Performance Evaluation Corporation, December 1995.

[30] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference.* ACM Press, 1996.