INTERPRETIVE AND NON-INTERPRETIVE TECHNIQUES FOR INSTRUCTION SET SIMULATION

Tomás Balderas-Contreras

Facultad de Ciencias de la Computación
Universidad Autónoma de Puebla
Puebla, Pue. MÉXICO

Hugo García-Monroy

Departamento de Aplicación de Microcomputadoras
Instituto de Ciencias
Universidad Autónoma de Puebla
Puebla, Pue. MÉXICO

ABSTRACT

Complete machine simulators have been used for long time to aid in the development process of computer systems, as well as to gather behavioral and performance information. An important component of this kind of programs takes each instruction from a target binary program and executes it on a different host machine. This paper describes the most important techniques that have been developed to implement this component and some of the most powerful simulators.

1 INTRODUCTION

During the last decades computer scientists, engineers and manufacturers have been so interested in the behavior and performance of computer systems components, configurations, system software and application programs. Specifically, there have been performance studies about operating systems, memory hierarchies, multiprocessors, database management systems, computer communications networks, user-level applications, distributed systems and *workloads* (one or more executing applications along with all of the operating system activity that occurs during the execution). Several techniques to estimate performance and investigate behavior have been developed (Heidelberger and Lavenberg 1984, Herrod 1998). All of them can be classified, not strictly, into one of the following categories:

Analytical methods. These are inexpensive approximate mathematical models of complex systems. They are based on queuing networks and probability theory. Their goals are to predict performance and behavior and provide some valuable information. However, in order they to be tractable, it is necessary to make some simplifying assumptions about the whole system, reducing the accuracy of the resulting data and the applicability of the model.

System measurements. These techniques perform experimental studies on an existing system or on a prototype of such system. Obviously, their low-level results are the most accurate and fast to obtain. However, it is expensive to build a prototype, and also, it is difficult to modify a system to estimate the effects of such modifications on its performance and behavior. As a result, it is the least flexible approach.

System simulation. In general these techniques consist of the design and development of a computational model of a system or process and the driving of experiments on it. A *trace-driven simulator* is a model which is driven by a sequence of measurements taken from an existing system. An *stochastic discrete event simulator* is a queuing model driven by a sequence of random numbers and it is mostly an statistical approach. These models can include the broad range of features of the real system that would make an analytical model intractable. It is also less expensive and less time-consuming than building a prototype of a given design.

A well-known hybrid approach between system measurements and simulation is *complete machine simulation*, also called *system level simulation* and *virtual machine* (Herrod 1998). A complete machine simulator is a program which has all of the functionality of an specific computer system but runs, although not strictly, on a completely different computer system. This kind of program reproduces all of the hardware found in a typical computer. It can model, for example, a central processor and a memory hierarchy to execute programs stored in memory, it can simulate a disk to store programs and data and it can reproduce a network interface to communicate with other machines. As a result, the simulator can execute an unmodified operating system and its application programs. We have concluded, as a consequence of our studies, that all of the simulators share the following features.

- A model of the processor's state (general purpose and control registers).
- Simulation of a memory hierarchy (main physical memory at least).
- Simulation of instruction processing (machine cycle).
- Collection of behavioral information (instruction and memory use profiles).

It is possible to add extra features to build a complete and more accurate system.

There are many significant advantages in the use of this technique. It permits both the validation of present designs and a prediction of the behavior of future designs in such an accurate way. It is possible to develop operating systems and application programs for a computer system in development process, as well as it is possible to make use of a software substitute of an unavailable expensive computer system. Since a complete simulator consists of several models of hardware components we can replace one of such models with a modified compatible one to study the effects of design modifications in the performance of the overall system, this means a great flexibility. Finally, the code attached to the hardware models can collect detailed and accurate information regarding the execution of such models. For example, an *execution profiler* can increment a counter each time an specific instruction in main memory is executed, as a consequence the simulator has a great visibility into itself.

Unfortunately, the important benefits described above are shadowed by a significant disadvantage. A virtual machine could be much slower than the corresponding actual hardware implementation. The more detailed a simulation model is, the slower its execution will be. Due to this weakness it is important to design efficient simulation techniques to improve the performance of the simulator. In this paper we describe existing systems and their approaches to implement efficient and relatively fast simulation. We focus on the existing methods to simulate the execution of programs whose binary instructions are defined by an specific instruction set. We are interested in the way a simulator fetches an instruction, decodes it and simulate its effects on the state of the simulated processor, in other words, we are interested in the interpretation methods used by a series of instruction set simulators. The simulators we consider in this paper are some of the most representative, they are g88, SimICS and Embra, additionally, we describe the techniques we employed in the development of SPARCSim, our own SPARC architecture simulator. It is not important, for now, the existence of any other component of the virtual processor (such as data and instruction caches, memory management unit or translation look-aside buffer). We concentrate on the implementation techniques and a little bit on the results of the methods.

The rest of this paper is organized as follows. Section 2 explains the main concepts about interpreters an threaded code. Section 3 contains information about g88. Section 4 summarizes SimICS. We describe Embra in section 5. Our own system, SPARCSim, is discussed in section 6. Some interesting details and a performance small talk is given in section 7. We finally conclude in section 8.

2 INTERPRETERS AND THREADED CODE

An interpreter is a computer program which performs the following tasks (Gries 1971).

- Translates a sequence of sentences written in a source language into a sequence of instructions defined by the internal form specified by the interpreter (there are several internal forms proposed, such as *polish notation* and *quadruples*).
- Takes each of these internal instructions and executes it through the use of the corresponding service routine chosen from a set of routines, or through a substatement of a long *case* statement.

Although the execution of an interpreted program is slower than the execution of a machine language program generated by a compiler, interpreters have a significant advantage. The interpreter can easily provide powerful debugging facilities to the user, they may be useful to find design flaws and for educational purposes. Interpreters have also helped some object-oriented languages (such as Smalltalk, Objective C and Java) provide their key features, such as *dynamic binding* and *dynamic typing*.

A technique that can be used to improve interpreters' performance is known as *threaded code* (Bell 1973). In the rest of this section we use SPARC assembly language (SPARC International, Inc. 1992) to show a piece of code needed to understand this technique. Let us suppose that the integer register `%g1` points to the memory location containing the address of the beginning of the service routine currently executing. That the next memory locations contain the addresses of the beginning of other service routines needed. Furthermore, let us suppose that each service routine has the following four instructions as its last instructions.

```
inc    4, %g1
```

```
ld      [ %g1 ], %l0
jmp     %l0
nop
```

The first of the previous instructions increments the value of the register `%g1`, the instruction counter. The next instruction gets the address of the beginning of the next service routine to execute, such address is stored into the local register `%l0`. Then an unconditional delayed control transfer to the first instruction of the next service routine takes place. The last instruction is the delay instruction.

As we can see, each service routine changes control to the next routine to be executed. As a result, we have interpretive code that does not require an interpreter at all, improving the execution time of a program. Threaded code will play a crucial role in the next sections of this paper.

3 DECODED INSTRUCTIONS AND G88

In this section we describe the first of the four simulators mentioned above, the g88 simulator (Bedichek 1990). This is a complete machine simulator that models a workstation based on the MC88100 RISC processor and the MC88200 cache and memory management unit integrated circuit (Tabak 1990). Its implementation runs on a workstation based on the MC68020 processor. Although it does not simulate some of the features of the MC88100 processor, the whole system can model many components with enough detail to run an unmodified operating system and its applications without any problem. Such components include the MC88200, main memory and I/O devices, for example a timer chip, a Zilog 8530 serial communications controller, interrupt controllers, a DMA controller and a simple disk simulator.

Threaded code plays an important role in the interpretation process of this simulator. All of the binary instructions that make up a program, which we call from now on *raw instructions*, are translated into an internal form, defined by the simulator, before their execution. Each raw instruction is 4 bytes-long while the corresponding *decoded instruction* is 16 bytes-long. A decoded instruction is an structure which consists of a pointer to the sequence of sentences that simulate the execution of a raw instruction, called *instruction handler*, and pointers to the operands, i.e. pointers to memory locations that simulate the 32 general purpose registers or to memory locations that contain immediate values. The simulator has a pointer to the currently executing decoded instruction, called the *decoded instruction pointer*, to keep track of the execution flow. The decoded instructions corresponding to the instructions `addu r4, r5, r6` and `subu r2, r4, 1000` are shown in figure 1. In addition to the code needed to execute a raw instruction, the instruction handlers contain the required sentences to increment the decoded instruction pointer and jump indirectly to the next instruction handler. The decoded instructions that make up an entire program are stored in memory to avoid the need for translating them again later.

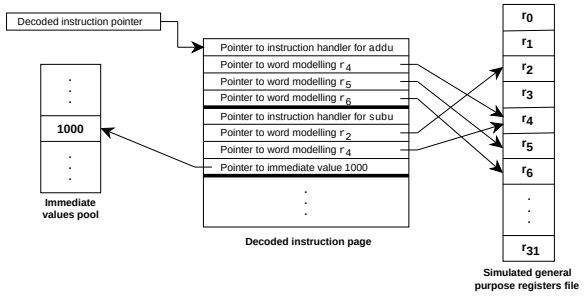


Figure 1: Decoded Instructions

Now it is time to glance at the memory simulation scheme used by g88 because it is closely related with the interpretation approach. The simulated physical memory is organized into a series of 4 KB-long pages. When the simulator starts it allocates an array of null pointers but no pages of memory are available yet. A physical memory page is allocated until an access to one of its addresses is needed, then a pointer to the recently allocated page is placed in the corresponding location of the array of pointers according to the required address. A simulated physical memory page can contain up to 1024 raw instructions and can be associated a page containing the corresponding 1024 decoded instructions. Such *decoded instruction page* is allocated when the simulator attempts to execute the raw instructions. Figure 2 depicts the former comments. When a decoded instruction page is allocated the simulator fills its decoded instruction slots with a *decode pseudo instruction* which, when executed, takes a raw instruction, translates it and replaces itself with the resulting decoded instruction. Once the decoded pseudo instruction has been placed its execution begins. This process continues for every raw instruction that has not been translated and executed.

Note that when all of the raw instructions have been translated, the set of decoded instructions and instructions handlers conform to the definition given above for threaded code.

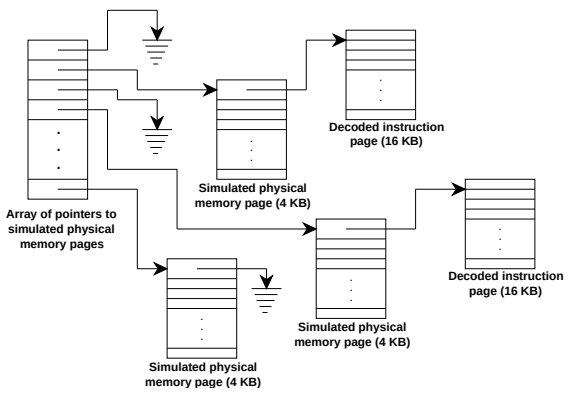


Figure 2: Organization of g88's Simulated Physical Memory

4 SIMICS' INTERMEDIATE FORM

SimICS is one of the most important and advanced simulators that have been developed (Magnusson 1997, Magnusson and Montelius 1997). It simulates one or more SPARC V8-compliant processors (SPARC International, Inc. 1992) and runs, getting several interesting measurements, on a 250 MHz UltraSPARC workstation. This system executes instructions in a sequential way, one instruction at a time. This feature makes SimICS fully deterministic. SimICS provides two alternatives to support realistic workload execution. It can explicitly emulate system calls from an Unix-like operating system such as SunOS or, alternatively, it can run an unmodified operating system. This is possible because SimICS can be extended with faithful-enough device models to build a full virtual computer system. A SimICS-based complete machine simulator commonly referred to as SimICS/sun4m (Magnusson et al. 1998) contains models for an Ethernet interface, support for serial communications through a console and disk storage. It is also possible to evaluate different memory configurations by adding new cache models to the memory hierarchy.

To achieve performance tuning and to understand program behavior SimICS provides a set of advanced profilers. Such set includes the following profilers:

1. Instruction cache misses.
2. Write data cache misses.
3. Read data cache misses.
4. Translation look-aside buffer misses.
5. Branches to an instruction.
6. Branches from an instruction.
7. Count of instruction execution.
8. Flag for instruction execution.

Those profilers could be the foundations for more powerful analysis tools built on top of them. As a singular advantage, it is possible for the user to add a new profiler module at run-time.

SimICS' interpretation process is slightly different than g88's. Each SPARC instruction is translated to an intermediate instruction which is 64 bit-long. Such intermediate instruction has a 32-bit pointer to the corresponding service routine and a 32-bit parameter word which contains register addresses and/or immediate values. Service routines perform their tasks, they use the parameter word to read or write the register file, calculate addresses and access memory and perform arithmetic or logical operations. The last instructions of a service routine correspond to the *epilogue* which performs administrative tasks for the simulation (such as event handling), fetches the pointer to the next service routine along with its parameters and jumps to execute such service routine. Figure 3 depicts the process.

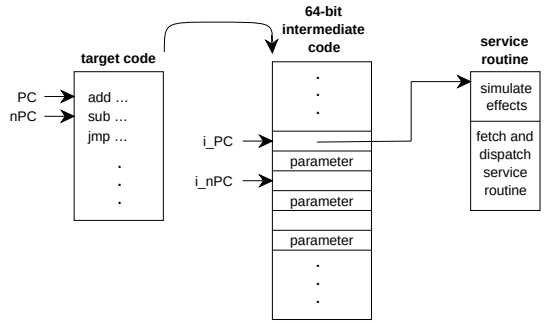


Figure 3: SimICS' Interpretation Process

SimICS's core has been implemented in two different ways. The scheme described above is used by a hand-written version of the interpreter. Later implementations are built by a sophisticated metatool called SimGen which designs the intermediate format and generates the decoders, encoders and service routines needed. To perform its task the metatool must be provided with a high-level specification of the target instruction set to simulate.

5 THE TRANSLATOR: EMBRA

The next simulator to discuss is called Embra (Witchel and Rosenblum 1996). It simulates a MIPS R4000/R4400 microprocessor along with a cache and a memory system. Embra is part of a bigger complete simulation environment known as SimOS (Her-

rod 1998). This system is binary-compatible with most SGI's workstations and can run the IRIX operating system along with its application programs. Later versions model DEC's Alpha architecture. For each hardware component to be simulated SimOS provides a number of different simulation models (it contains three processor simulation models: Embra, Mipsy and MXS). Although all of these models provide the same basic functionality, they are different to each other in the level of detail at which they behave and in their speed rate. It is possible to combine several different simulation models to provide different simulation environment configurations, known as *execution modes*, each with different speed-detail trade-offs. SimOS supports three execution modes: *positioning mode*, *rough characterization mode* and *accurate mode*. The user selects the first mode to speed up a workload's initialization process. When it is done, it is possible to change to a more accurate execution mode to obtain realistic behavioral information of a more interesting stage of the workload's execution, but at an slower speed rate. Figure 4 illustrates the SimOS system.

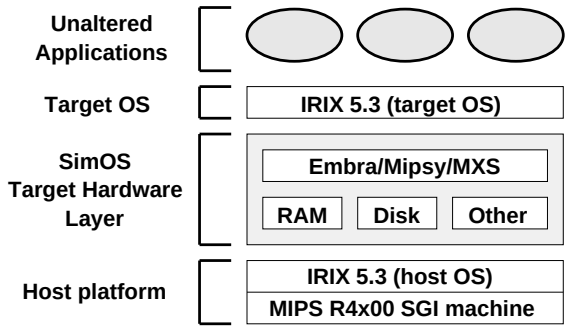


Figure 4: The SimOS Complete Machine Simulator

Embra's simulation scheme differs greatly from the approaches just discussed. Embra simulates the target processor's state by using some data structures and performs *dynamic binary translation* instead of interpretation techniques. Each target instruction is translated by Embra into a sequence of host instructions that simulates the effect of the target instruction over the processor's state. Embra translates each basic block (a sequence of instructions which ends with a control transfer instruction) and caches the resulting translation in an special data structure called the *translation cache*, this avoids the need for later translation. Figure 5 sketches binary translation for an small basic block.

A simple implementation of Embra's translator depends on a *dispatch loop* which examines the current value of the simulated program counter. If the instruction pointed to by this program counter has already been translated, the corresponding translated block is executed. If it has not been translated yet, the entire block the instruction belongs to is translated and cached for future execution. The last instructions of a translated block changes the value of the simulated program counter and transfer control back to the dispatch loop. Note that this approach needs the loop to dispatch the translated blocks, whereas each service routine in a threaded coded program dispatches the next one.

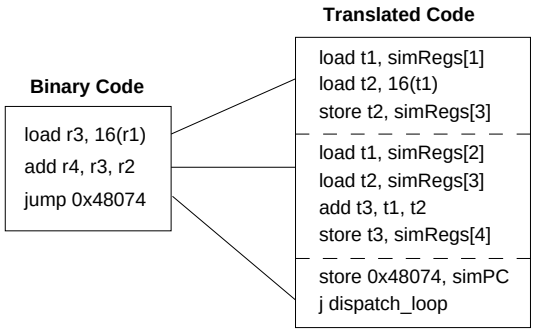


Figure 5: Binary Translation

6 SPARCSIM'S POINTERS

SPARCSim simulates the SPARC V8 instruction set. It is mainly intended to be an educational tool for those people interested in RISC architectures (Balderas and García 1999). Figure 6 sketches the organization of the system which can be divided into the following layers.

Kernel layer. This layer is made up of a set of modules written in C. It provides an assembler, a disassembler, the instruction interpreter, a set of service routines and a simple execution profiler. It also contains the data structures needed to simulate the processor's state, some useful tables and a set of support routines. It can be ported to several hardware and software platforms but it is not functional by itself.

GUI layer. This layer invokes the kernel modules needed to perform a user's request. A GUI can be built for every operating environment the kernel is ported to. A first release of SPARCSim was developed as an application for the NeXTSTEP object-oriented operating environment running on a Pentium II-based personal computer. The kernel along with a command line-oriented interface were also implemented in a SunUltra-10 workstation running the Solaris operating environment.

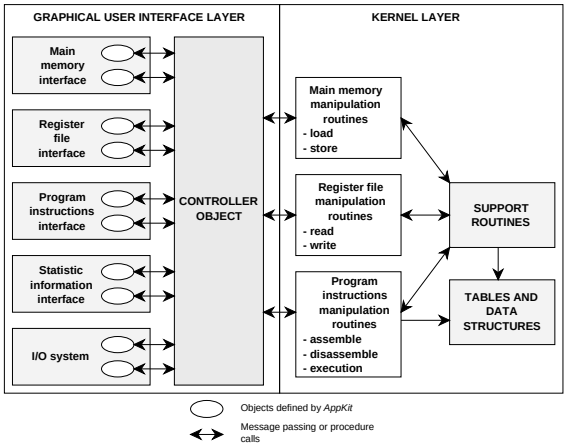


Figure 6: Layer Organization of SPARCSim

SPARCSim is not a complete machine simulator yet. It does not provide simulation models for TLB, cache memories, virtual memory management or input/output devices. As a consequence SPARCSim can run neither an operating system nor an operating system-intensive workload, but it can run a lot of interesting user programs. We plan to extend the simulator, in a near future, to support complete machine simulation.

Since SPARCSim does not support virtual memory management all of the addresses referred to by programs are effective addresses. Main physical memory is simulated simply by allocating a long array of characters. Register file is simulated by a circular array of 32 bit-long integers, special care is taken when a read or write operation is performed because the register file is organized as a set of overlapped register windows.

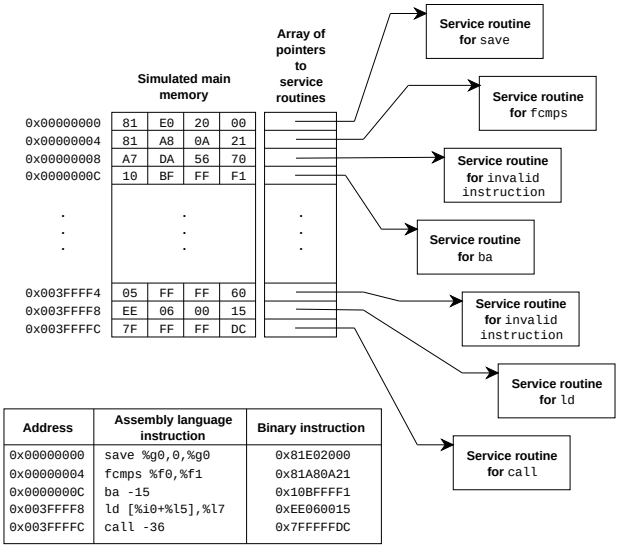
SPARCSim's current interpretation scheme is based on a fetch-decode-execute loop which reads an instruction from the simulated memory, checks the kind and the opcode of the instruction by using a case statement, looks for a pointer to the appropriate service routine in an internal table and then calls this service routine to execute the instruction. Each service routine simulates execution, modifies program counters and then returns to the main loop. The main loop along with the case statement, the search loop and the call/return overhead slowdown the overall execution of a program. This situation motivates us to modify the interpretation process to improve simulation performance.

According to the new scheme SPARCSim allocates an array of pointers to an instruction simulation service routine. The first pointer of this array is associated to the first four bytes of main memory, the second pointer is associated to the next four bytes and so on. SPARCSim, and the other simulators discussed, take advantage of the fact that several RISC processors handle 4 bytes-long instructions (when all of the instructions have the same length they can be fetched by a single read, they can be easily decoded by hardware or simulators and program counters are always incremented by a fixed value), thus for each instruction in main memory there is a service routine pointer. The simulator allows the user to write a byte, a halfword (2 bytes) or a word (4 bytes) on main memory, so does an store instruction. When this happens the simulator calls the `write_memory` procedure which calculates the effective address of the word the modified characters belong to and fetches it. If this word contains a valid binary instruction then the `write_memory` procedure looks for, in an internal table, the pointer to the appropriate service routine and places it on the pointer array element corresponding to the word just fetched. When the fetched word does not correspond to a valid instruction the `write_memory` procedure places a pointer to an *invalid instruction service routine*. This service routine modifies the program counters so that the next instruction to be executed, i.e. this service routine simulates a no-operation instruction. Additionally, the invalid instruction service routine can increment an invalid instruction counter and store the address of the invalid word for future references. Figure 7 depicts some binary instructions stored in a 4 MB-long simulated main memory and their associated pointers to the corresponding services routines. As we can see, there is a one to one mapping between valid instructions and pointers to simulation service routines and between invalid instructions and invalid instruction service routines.

This approach transfers the fetch and decode stages from the interpretation loop to the procedure devoted to write data on main memory. Thus, to run a program from an initial to a final address the main interpretation loop simply fetches each pointer from the array, starting at the position given by the initial address divided by four and ending at the position given by the final address divided by four. A service routine is indirectly called when the corresponding pointer has been fetched and it transfers control back to the main loop when its execution has finished. The new approach resembles the scheme used by g88 to cache decoded instructions in a memory space attached to each simulated physical memory page.

7 DETAILS AND PERFORMANCE COMMENTS

In this section we mention some final interesting issues about the systems just discussed. They concern imple-



Address	Assembly language instruction	Binary instruction
0x00000000	save %g0,0,%g0	0x81E02000
0x00000004	fcmps %f0,%f1	0x81A80A21
0x0000000C	ba -15	0x10BFFFF1
0x003FFFF8	ld [%i0+%l5],%l7	0xEE060015
0x003FFFFC	call -36	0x7FFFFFDC

Figure 7: Binary Instructions and Their Pointers to Service Routines

mentation details and performance measurements.

Developers report that when run on a 2.5-MIPS MC68020-based workstation g88 simulator executes approximately 130000 MC88000 instructions per second (Bedichek 1990), this tells us that the average number of host instructions per service routine is 19 or 20. It is also reported the way threaded code was implemented on a C compiler for MC68020 instruction set and the modifications to the assembly code required to produce a correct program.

A comparison between native execution and simulated execution of SPECint95 suite component programs shows a lost of performance ranging from 39 to 75. This performance lost can slow down considerably a program execution, as an example SimICS executes the *vortex* program of SPECint95 in 16.35 minutes while the native hardware executes it in 13 seconds (Magnusson et al. 1998). The number of host instructions per service routine ranges from 10 to 30, this because SPARC architecture has its own features that make it a little bit more time-consuming to simulate.

It is possible to improve Embra's simulation performance with a technique called *chaining* (Witchel and Rosenblum 1996). When two or more translated blocks are always executed one after another at run time they can be linked together by replacing each jump to the dispatch loop instruction with a jump to the next block. This reduces the overhead caused by the execution of the dispatch loop.

Kernel portability was a major goal in the design process of SPARCSim, its interpretation scheme differs from threaded code due to the fact that implementing threaded code is not easy for most of the C compilers. We could have taken advantage of the ability of GCC compiler to get the addresses of the labels defined by a C program, store those addresses into a table and use them along with the `goto` statement as in the following piece of code.

```
main(void)
{
    void   *labels[3] = { && a, && b, && c };
    int    A, B, C;

    A = 100;
    B = 30;
    C = 20;

    goto *labels[2];

 a: A = 3;
 b: B = A + 10;
 c: C = A * B + 100;

/* At this point A = 100, B = 30 and C = 3100 */
   printf("A = %d\tB = \%d\tC = %d\n", A, B, C);
}
```

The reader should realize the great benefits this scheme provides to the implementation of threaded code. A threaded-coded implementation of SPARCSim's interpreter can be developed taking into account that GCC is available for many platforms, unfortunately its ability is not present in all of the C language compilers and thus portability is not guaranteed at all. On the other hand we could patch assembly code generated by the C compiler to implement threaded code, however this approach has two main disadvantages, it is both time consuming and not portable. SPARCSim simulates SPARC architecture properly but at a cost, there are special procedures to perform register file operations, main memory accesses and arithmetic processing. This special procedures are called by the service routines that need them, thus introducing a certain overhead to the simulation process. Despite this situation, current SPARCSim performance is acceptable for its educational purposes (Balderas and García 1999).

Intermediate instructions for g88 and SimICS require memory to be allocated to hold both a pointer and parameters. This is advantageous because service routines would not need to process binary instructions to get operands. On the other hand, every SPARCSim service routine does need to extract operands from binary instructions, this approach saves memory but introduces an small overhead to instruction simulation. As a consequence a memory space-execution speed trade-off arises.

8 CONCLUSIONS

We have discussed interpretive and non-interpretive simulation techniques in enough detail to motivate the reader to research by himself about this such an interesting topic. Our main focus was on the most powerful simulators ever developed, their advantages and their weaknesses. An interesting point is worth considering here, some simulator systems, such as SimICS and SimOS, have been built on top of the same platform they model and some other, such as g88 and SPARCSim, have not. This situation allows the user realize the great benefits any simulation environment may provide him with, it does not matter his particular condition. We mentioned the great influence threaded code has had in the development of those systems. We also talked a little bit about performance. As a conclusion of those comments we can say that an efficient dispatching technique, such as threaded code, along with the shortest service routines possible could produce better and faster interpreters for instruction set simulation.

Computer architecture simulation, at any of its levels, has received great interest from many people for many years. The list of projects is very large and includes simulators for RISC and CISC workstations, such as the simulators described above, and even simulators for video games consoles are nowadays available. The goal of this document was to introduce the reader to some of the methods employed to build these kind of tools.

ACKNOWLEDGEMENTS

Thanks to the referees for their comments, they were useful to locate the points in the document where our ideas were not clearly expressed, these people played a central role to improve this document. Thanks, also, to María Auxilio Osorio Lama from FCC-UAP for letting us to work, play, e-mail and develop an SPARCSim prototype in OPTIMA, her SunUltra-10 workstation.

REFERENCES

Balderas, T., and H. García. 1999. Desarrollo de un sistema simulador de la arquitectura SPARC sobre el sistema operativo Mach y el ambiente NeXTSTEP. In *Memorias del Segundo Encuentro Nacional de Computación, ENC'99*.

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

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

Gries, D. 1971. *Compiler construction for digital computers*. New York: John Wiley & Sons Inc.

Heidelberger, P., and S. S. Lavenberg. 1984. Computer performance evaluation methodology. In *Performance evaluation for computers architects*, ed. C.M. Krishna, 20–45. California: IEEE Computer Society Press.

Herrod, S. A. 1998. Using complete machine simulation to understand computer systems behavior. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California.

Magnusson, P. 1997. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 1997 Winter Simulation Conference*.

Magnusson, P., and J. Montelius. 1997. Performance debugging and tuning using an instruction-set simulator. Technical Report T97:02, Swedish Institute of Computer Science, Kista, SWEDEN.

Magnusson, P., F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. 1998. SimICS/sun4m: a virtual workstation. In *USENIX Annual Technical Conference*.

SPARC International, Inc. 1992. *The SPARC architecture manual, version 8*. New Jersey: Prentice-Hall, Inc.

Tabak, D. 1990. *RISC systems*. New York: John Wiley & Sons Inc.

Witchel, E., and M. Rosenblum. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference*.

AUTHOR BIOGRAPHIES

TOMÁS BALDERAS CONTRERAS received his B.S. degree in Computer Science from Universidad Autónoma de Puebla. His current research interests include computer architectures, operating systems, programming languages and instruction set simulation. His e-mail address is `balderas@optima.cs.buap.mx`.

HUGO GARCÍA MONROY is a researcher in the Instituto de Ciencias and a professor in the Facultad de Ciencias de la Computación at Universidad Autónoma de Puebla. He received his B.S. degree in Computer Science from the same institution. His current research interests include operating systems, instruction set and operating systems simulation, distributed computing and computer architectures. His e-mail address is `gmonroy@servidor.unam.mx`.