Implementation Aspects of a SPARC V9 Complete Machine Simulator

Bill Clarke **Adam Czezowski** **Peter Strazdins**

CAP Program
Department of Computer Science
Australian National University
Acton, ACT 0200, Australia
Email: {wpc,adam,peter}@cap.anu.edu.au

Abstract

In this paper we present work in progress in the development of a complete machine simulator for the UltraSPARC, an implementation of the SPARC V9 architecture. The complexity of the UltraSPARC ISA presents many challenges in developing a reliable and yet reasonably efficient implementation of such a simulator. Our implementation includes a heavily object-oriented design for the simulator modules and infrastructure, caching of repeated computations for performance, adding an OS (system call) emulation mode to the simulator and a variety of testing strategies. An ultimate and critical goal in constructing such an artifact is to successfully boot an existing operating system from it; we describe techniques implemented so far, and outline the remaining work and issues, in order to achieve this goal.

Keywords: execution-driven simulation, SMP, complete machine simulator, SPARC V9 ISA, UltraSPARC, object-oriented design

1 Introduction

Architectural performance analysis is an increasingly important technique in modern computer systems design (Bose & Conbte 1998). Its main component is called *simulation*, where a model of the system is made; usually this model can reproduce the functional and, in the case of what is termed *execution-driven simulation*, the intended timing behaviour of the system. With the increasing popularity of medium to large scale *symmetric multiprocessors* (SMPs), the memory system is a potential bottleneck for the performance of many applications (Ranganathan, Gharachorloo, Adve & Barroso October 1996). Hence, simulation that can accurately analyse SMP memory performance is particularly important.

Most simulation tools can be classified as *user-level simulators*: these simulate the execution of a (user) process, emulating any system calls made on the *target* (simulated) computer using the operating system of the *host* computer (running the simulator). *Complete machine simulators* (Rosenblum, Herrod, Witchel & Gupta 1995, Magnusson, Dahlgren, Grahn, Karlsson, Larsson, Lundholm, Moestedt, Nilsson, Stenstrvm & Werner 1998) simulate the target computer from the boot stage, including all code executed by the PROM, the OS that is loaded by the PROM, and any processes subsequently created.

As the OS abstracts a vast amount of detail of the underlying computer system from an executing process, complete machine simulators are greatly more complex and difficult to construct than other performance analysis tools. On the other hand, a complete machine simulator can be used for: (1) analysing workloads, with full visibility into machine behaviour (i.e., collecting statistics on events that are unavailable for collection on real hardware), (2) OS development (including debugging: determinism can be ensured; also for performance tuning: it may be a better alternative to modifying the kernel to add instrumentation code), (3) perform simulation before the hardware exists, and (4) perform architectural studies with varied machine parameters.

Note that user-level simulators can partially satisfy requirements (1), (2) and (4). However, they cannot give any insight into the machine activity resulting from a process' system calls, nor even into the resulting interference of the operating system activity (Herrod 1998) (e.g., cache and TLB pollution). DMA requests to devices, which typically occur within the kernel, cannot be captured by a user-level simulator. Generally, they can only simulate a single user process (or perhaps, by emulating thread creation, multiple threads within a process). Furthermore, user-level simulators cannot capture behaviour involving physical addresses, including accurate analysis of physically indexed caches, TLB misses and page faults. Finally, and most importantly for SMP simulation, the timing of memory operations by different CPUs affects the behaviour of lock acquisitions and barrier synchronisations, often a major source of overhead. These effects can only be accurately captured by an execution-driven complete machine simulator.

This paper introduces *Sparc Sulima* — a complete machine simulator for the UltraSPARC implementation (Sun Microelectronics 1997) of the SPARC V9 processor architecture (Weaver & Germond 2000) — currently under development at the Australian National University. Its aim is to give accurate analysis of memory behaviour for operating systems code and kernel-intensive applications; especially of interest are threads interactions in the SMP context. It is based on the open-source Sulima simulator infrastructure, developed recently at the University of New South Wales (Zadarnowski 2000*b*) for the 64-bit MIPS processor.

There are various challenges in such an endeavour. Firstly, as for most complete machine simulators, it must emulate faithfully all of the functionality of the computer system visible to its operating system. Not only does this add complexity, this may require proprietary information (e.g., details of the PROM and motherboard). The simulator must also successfully boot the operating system (simulating possibly billions of instructions), thus requiring a very high degree of fidelity and accuracy in its implementation. Secondly, the 64-bit UltraSPARC ISA is large and complex, arguably much larger than the corresponding MIPS and even the Alpha ISAs, for which complete machine simulators have been more widely developed. This is due to its requirement for compatibility with 32-bit SPARC V8, its large set of Address Space Identifiers (ASIs) for address translation and external register access, and the UltraSPARC special memory access and graphics instructions (Sun Microelectronics 1997). Thirdly, because of both of these factors, there is a need to improve simulator speed so that long execution sequences can be efficiently analysed. Thus the implementation of such a simulator requires the overcoming of many software engineering challenges; this paper describes the main approaches and techniques employed for doing this.

This paper is organised as follows. §2 presents a brief overview of related work. An overview of Sparc Sulima is given in §3, including its modelling of the CPU. Aspects of memory system modelling are discussed in §4 . The simulator's 'OS Emulation mode', which enables it to boot off a very simple kernel ('nucleus') is presented in §5, and the PROM interface (for booting the Linux kernel) is described in §6. Testing strategies for the simulator are described in §7. Finally, preliminary performance results are given in §8, with conclusions and future work discussed in §9.

2 Related Work

RSIM (Pai, Ranganathan & Adve 1997) is a user-level execution-driven SPARC simulator. It simulates the execution of 'V8plusa' executables (which can be run on an UltraSPARC processor in 32-bit compatibility mode) and has a detailed CPU modelling (including pipelines and branch prediction) and detailed modelling of the memory system. SMP simulation can be used on programs using a restricted threads library, providing its source code is available.

There are few well-known complete machine simulators developed to date. Those suitable for analysing the memory behaviour of large workloads include SimOS (Rosenblum et al. 1995, Herrod 1998) (32-bit MIPS and 64-bit Alpha processors) and SimICS (Magnusson et al. 1998, Werner & Magnusson 1997) (originally developed for the SPARC V8 and Motorola 88110 processors).

The SimOS simulation system contains several simulator *cores* (implementations of the target processor and memory system). These cores use a common state representing the machine and simulated workload; thus it is possible to dynamically switch between them. This enables realistic workloads to be studied at a desired level of detail, with the 'uninteresting' sections of the workload 'fast-forwarded' over by a faster core. The fastest core, called Embra, is based on the *binary translation* of the target computer's code to that of the host computer. The next fastest core, Mipsy, is based on a *fetch-decode-execute* interpreter, and is adequate for accurate memory system analysis. The third core, MXS, simulates at a similar level of detail to RSIM.

Recently, SimICS 1.0 has been released (Virtutech 2000) covering several processor families, including SPARC V9 (UltraSPARC I–III) systems. However, as this is a commercial product, little is publicly known about its internal design and techniques of construction apart from the information in the earlier SimICS papers.

For a more complete overview of simulation methods, simulators, and their main concepts and techniques, see Strazdins (2000).

3 Overall Structure and Object-Oriented Design

Our basic approach to the design of Sparc Sulima is to model the UltraSPARC CPU and memory system 'as is', using object-oriented techniques implemented in C++. This leads to a modular approach, with modules corresponding closely to the components of the real system, and aids in the readability and understandability of the simulator source. Simplifications (and omission of details) of the system's functional behaviour are avoided where possible, since it is not clear which details will be needed by an OS and which will not.

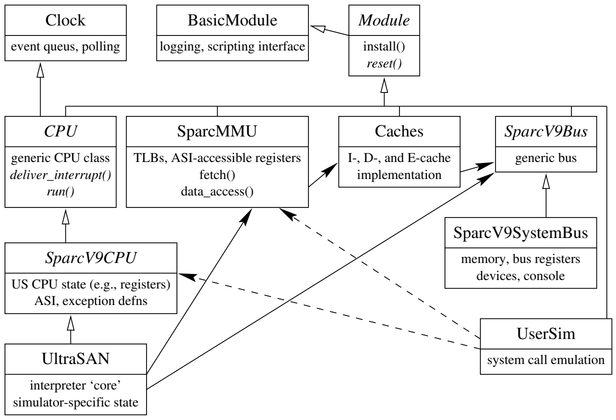

Figure 1: Simplified Sparc Sulima class diagram.

Figure 1 gives a simplified view of the object structure of the simulator. Most major components inherit the `Module` class, thus inheriting properties common to system components, and facilities for installation from a scripting interface. These components also must supply an implementation of the `reset()` method. The components also can inherit common checkpoint capabilities from a `virtual Serializable` class, and debugging facilities from a `Traceable` class. Currently there is only one simulator 'core' (`UltraSAN`: UltraSPARC Simulated Architecture iNterpreter); this design allows for the installation of further cores, for example a faster core based on binary translation.

At present one or more `UltraSAN` and one `SparcV9System` bus module at least must be installed from the scripting interface. The `UserSim` module can then be optionally installed; if present, the CPU implementation will interpret special `illtrap` instructions for emulating of system calls, and break out of the simulation; this is also used for PROM library call emulation. The reset of the root (`Sulima`) object in turns resets all installed modules, with the reset of the `UltraSAN` module also creating a `SparcMMU` and `Caches` modules. The `CPU::run()` method is then invoked from the script to begin a simulation.

At present, for the main part of the simulator source codes (Figure 1), the SPARC V9 (more correctly UltraSPARC) dependent code amounts to $\approx 14,000$ lines of C++ (not including automatically generated code: see §3.1). There was little scope to re-use codes from the existing MIPS64-Sulima codes (Zadarnowski 2000*b*), for although there are many similar features in both architectures (e.g., top-level caches and the 64-bit add instruction) the semantics are usually slightly different, and it is crucial for complete machine simulators to implement the exact semantics of the architecture.

The architecture-independent code (general Sulima infrastructure, which includes also generic type definitions and integer/bit field functions) amounts to $\approx 5,000$ lines of C++ code. Specialised codes for testing currently amounts to $\approx 3,000$ lines of SPARC assembler and C code.

`UltraSAN` interprets each instruction to be executed in a standard fetch-decode-execute cycle. This means that timing effects due to groupings and dependencies cannot be accurately modelled; at present, most 'simple' instructions, excepting memory operations, merely add 1 cycle to the simulated clock. The fetch stage uses an 32-byte instruction buffer, as does the real UltraSPARC; this reduces the number of calls to `SparcMMU::fetch()`, and so improves simulator performance.

The simplest way of running the simulator ("booting from main") is to provide an (ELF) executable to simulate. This executable must be statically linked to a very limited implementation of the C standard library. The simulator loads the executable, using standard ELF libraries to perform this and also locate the address of `main()`. Then execution begins by setting the simulated `pc` register to this address. If exceptions can occur, a minimal nucleus (including at least a trap table) must be linked in with the executable, and `main()` must call a specialised initialisation function so that the trap table base pointer register points to this table (and any other desired initialisation). This is useful for simple testing of the simulator. Alternatively, the full boot process involving a reset exception can be invoked, which requires the nucleus to be installed in ROM at the reset exception's vector address, as is done in the OS emulation mode (§5).

The following sub-sections give further details of `UltraSAN`'s implementation: the decode and execute stages, and the trap architecture. Other aspects of the simulator are treated in more detail later in the paper.

3.1 Instruction decoding

During simulation we need to be able to convert a 32-bit SPARC V9 instruction (which consists of an *opcode* and various *operands*) into a form which the simulator can use in order to evaluate the required effect of the instruction. This *decoding* phase must be correct, complete, and efficient.

Other simulators such as SimOS/mipsy (Rosenblum et al. 1995) and Sulima/koala (Zadarnowski 2000*a*) have taken the approach of jump tables or functions based on partial decodes of the instruction. This approach is difficult for SPARC V9 due to the complexity of the ISA.

This complexity also means that it is difficult to be sure that we decode each instruction correctly.

We use the New Jersey Machine Code Toolkit (Ramsey & Fernández 1998) to generate our decoder. This toolkit uses specifications written in SLED (Specification Language for Encoding and Decoding (Ramsey & Fernández 1997)) and can generate encoders, decoders, and, with the aid of an external assembler, can verify the correctness of the specification[1].

We have written a SPARC V9 syntax specification in SLED (Clarke 2000) and have used the toolkit to verify the specification. The generated decoder is 6000 lines of C code in a header file.

Our approach ensures that the decoding is both correct and efficient. It also has the side-effect of being able to generate an *encoder* for when we wish to write a binary-translating simulator. Completeness is more problematic to prove, but in more than 6 months of testing no opcode and no version of an instruction have been found to be missing.

The simulator uses the decoder to fill in a structure that contains the opcode (a 32-bit enumeration) and a 64-bit operands structure (up to 6 operands). We also have a compile-time optimisation where the I-Cache can also behave as a decode-cache, which reduces the cost of decoding the same instruction multiple times at the expense of storing additional data.

[1]This relies on the correctness of the assembler. Indeed, we found a bug in the Solaris WorkShop Compilers 4.X assembler where a particular variant of the `prefetcha` instruction was incorrectly encoded.

```
void UltraSAN::EvalUDIVcc (InstrOperands *ops) {
  Int32 rs1 = ops->get_rs1(), rd = ops->get_rd();
  UInt64 v1 = (UInt64(Y) << 32)
            | LOWER_32(GPReg(rs1));
  UInt32 v2 = ValSrcOrImm(*ops);
  if (v2 == 0)
    process_exception(EXC_DivideBy0, 0);
  else {
    UInt64 res = v1 / v2;
    int V = 0;
    UInt64 MxRes = UInt64(1) << 32;
    if (res >= MxRes) { res = MxRes-1; V = 1; }
    GPReg(rd) = res;
    UInt32 ru32 = bits(res, 31, 0);
    CCR = (bit(ru32,31) << (ICC_1st+CC_N))
        | ((ru32==0)    << (ICC_1st+CC_Z))
        | (bit(res,63)  << (XCC_1st+CC_N))
        | ((res==0)     << (XCC_1st+CC_Z))
        | (V            << (ICC_1st+CC_V));
  }
}
```

Figure 2: The `UDIVcc` evaluation function. This is an unsigned divide of a 64-bit by 32-bit number, where the 64-bit number is created by concatenating the `Y` register with `rs1`. The result is 32 bits only, and the condition codes are updated.

3.2 Instruction evaluation

After decoding the instruction into an opcode and a set of operands, the simulator evaluates the instruction. It does this by looking up a table of evaluation functions (actually, member functions of the `UltraSAN` class) and calling the function corresponding to the opcode. It passes in the operands as a parameter.

There are 326 evaluation functions in the `UltraSAN` class, although many of them are quite similar and their implementations are parameterised into macros. For example, we have specialised evaluation functions for each comparison type variant for the `BPcc` family of opcodes; the only difference between them is which condition-codes to check. To reduce the proliferation of evaluation functions, `Bicc` (and `FBfcc`) instructions are converted to the equivalent `BPcc` (`FBPfcc`) instruction.

As an example of evaluation functions, the `UDIVcc` evaluation function is shown in Figure 2.

3.3 Inline assembler evaluation functions

The example in Figure 2 illustrates the complexity of some of the instructions involved. Attempting to reproduce the exact behaviour in a high-level language is quite difficult.

Since we are simulating SPARC V9 instructions on a SPARC V9 host we can take advantage of knowing that the result of executing the equivalent instruction *directly* should be correct. All we need to know are the inputs to and outputs from the instruction; hence the other name of "black-box" simulation.

The compiler provides an excellent interface to inline assembler which we utilise in these evaluation functions. As an example of the inline assembler method, the g++-specific evaluation function for `UDIVcc` is shown in Figure 3.

All the integer operations, floating-point operations and UltraSPARC-specific graphics operations have been implemented using this method, and using a similar method for Sun's Forte CC compiler.

```
void UltraSAN::EvalUDIVcc (InstrOperands *ops) {
  UInt64 *rs1_p = &GPReg(ops->get_rs1());
  UInt64 *rd_p  = &GPReg(ops->get_rd());
  UInt32 tmp1, tmp2, tmp3;
  UInt8 *cc_p = &CCR;
  UInt32 *y_p = &Y;
  if (bits(ValSrcOrImm(*ops), 31, 0) == 0)
    return process_exception(EXC_DivideBy0, 0);
  if (ops->get_is_reg()) {
    ...
  } else {
    Int32 imm = ops->get_imm();
    asm volatile
      ("ldx [%3], %0;"     // tmp1 = *rs1_p
       "signx %4, %1;"     // tmp2 = signx(imm)
       "lduw [%7], %2;"    // tmp3 = *y_p
       "wr %2, 0, %%y;"    // Y    = tmp3
       "udivcc %0, %1, %0;"
                  // tmp1,CCR = Y[]tmp1 / tmp2
       "rd %%ccr, %1;"     // tmp2  = CCR
       "stb %1, [%6];"     // *cc_p = tmp2
       "stx %0, [%5]"      // *rd_p = tmp1
/*out*/ : "=r" (tmp1), "=r" (tmp2), "=r" (tmp3)
/*ins*/ : "0" (rs1_p), "1" (imm), "r" (rd_p),
          "r" (cc_p), "2" (y_p)
/*clobbers*/ : "memory", "cc");
  }
}
```

Figure 3: g++-specific inline assembler (black-box) evaluation function for the `UDIVcc` instruction. Addresses of register values and 32-bit temporary variables are used to work around the compiler's 64-bit behaviour.

We have portable (non-inline assembler) versions of most of the integer operations, but the floating-point and graphics operations are probably too complex to attempt to simulate portably and efficiently.

3.4 General purpose registers

There are 32 64-bit general purpose registers (GPRs) available at any given moment in SPARC V9 CPUs. SPARC V9 provides a "windowing" mechanism, where 24 of the registers (ins, locals and outs) are a window into a larger set of registers (UltraSPARC has 128 windowed registers). In addition, UltraSPARC has four sets of globals: normal, alternate, MMU and interrupt.

Two or three GPRs are accessed on virtually every instruction, so ensuring fast access to the registers is imperative. Changes of current-window and current-globals occur infrequently compared to register access so our implementations of GPRs reduce the cost of individual register access at the expense of more complex current-window and current-globals changes.

The implementations we have tested are:

1. calculate the offset into windowed/global registers for each access;
2. cache the offset for each of globals, ins, locals and outs when they change; and
3. have a scratch set of 32 registers; the main register files are kept up-to-date when the current-window or set of globals change[2].

In testing the latter seems to be the most efficient, but it depends on the ratio between the frequency of register-accesses and the frequency of window-changes.

[2]Optimisations can be made by changing the method of copying the register values to and from the scratch set: the UltraSPARC-specific block copy seems to be the fastest.

3.5 Trap Architecture

The trap architecture can be considered the heart of the CPU and indeed the computer system. The UltraSPARC has a relatively complex architecture supporting five trap levels, including 4 types of reset traps; execution at the top level is called the Reset-Error-Debug state (Weaver & Germond 2000). Fortunately, there is a great deal of commonality between the state changes at different levels, allowing a reasonably compact implementation of trap entry and return. Only *precise* and *disrupting traps* need be implemented for the UltraSPARC (Sun Microelectronics 1997).

Precise traps are normally called *exceptions*: these must be detected immediately in the instruction cycle, and the exception of *highest priority* is then raised before any state changes occur. This can be implemented by the evaluation functions (§3.2) checking for exceptions in order of priority, and returning with the corresponding exception code *before* any of the (simulated) state is changed. Thus the traditional method of implementing exceptions in simulators — the host-dependent Unix 'long jump' facility (Rosenblum et al. 1995, Pai et al. 1997, Zadarnowski 2000*b*) — is avoided. This improves the readability of the codes and avoids the potentially large performance penalty of the 'long jump'.

Disrupting traps include the invocation of interrupts and reset events, and bus errors/time-out traps. These need not be delivered precisely in a simulator (Zadarnowski 2000*a*). Interrupts and reset events are invoked in the simulator by the virtual `reset()` and `deliver_interrupt()` methods, which set a bit in an 'events vector'. This vector is checked at the beginning of the instruction cycle, where also timer interrupts invocation are set. On the UltraSPARC, the module invoking the interrupt must also use the UDB interrupt delivery protocol (Sun Microelectronics 1997) by manipulating the corresponding UDB registers, which are implemented in the `SparcMMU` module.

In a simulator context, bus errors and timeouts can be treated as other exceptions detected by the MMU: they are detected before the corresponding `fetch()` or `data_access()` call returns. While this is unrealistic in the sense that they would be detected later on a real system, this should not affect the functional behaviour of the simulator.

4 Memory System Modelling

The modelling of the UltraSPARC memory system, and in particular the Memory Management Unit (MMU), adds considerable complexity to Sparc Sulima. It accounts for approximately 1/4 of the SPARC-dependent source code for the entire system.

Firstly, there are about 60 ASI-addressable registers (including pseudo-registers used to access TLB and cache entries), most of which have several bit fields. These registers are accessed via SPARC V9 'load/store alternate' instructions, which pass the ASI and a virtual address to the MMU (call to the `data_access()` method). Writes to the registers have to take into account any read-only and 'sticky-bit' bit fields in each case.

Secondly, for non register-access calls, complexity is exacerbated by the need to check for 14 (generally complex) exception conditions; correct setting of the MMU's fault (error) registers requires these to occur in a block after translation is attempted, which in turn requires determining the context corresponding to the ASI followed by a TLB lookup.

A successful translation returns four attributes: the physical address, whether to bypass the 1st and

2nd levels of cache, and whether the byte ordering is little-endian. The corresponding request is then passed to the appropriate cache or bus module.

Performance is enhanced considerably by *caching* previously successful translations; in this case a tag is made up from the following fields: the virtual address page number, the ASI, the relevant bits from the CPU's 'processor state' register. A successful lookup would return the 4 attributes mentioned above plus whether the access was writable, and the expensive TLB lookup and exception checks (excluding an alignment check) may be skipped. Invalidation of this *translation cache* must occur when the TLB or any other register affecting translation is modified. To permit selective invalidation, TLB indices and context identifiers are added to each entries. Two sets of 16 entries were found to be sufficient.

Similar schemes have been used in other simulators (Herrod 1998, Magnusson et al. 1998, Zadarnowski 2000*a*), but for SPARC V9 the translation caching is necessarily more complex.

To ensure translation caching introduces no errors, a checking mode has been provided. When activated, all initial lookups are regarded as a 'miss'; this forces the translation to be re-evaluated. Before the translation is inserted into the cache, it is checked if the tag was in fact present; if so, any discrepancy in the new and cached attributes results in the simulation aborting. The other element of this mode is to check that when an exception occurred (which implies an unsuccessful translation), the initial lookup would have failed.

This mode added very little extra code and such a scheme can be similarly incorporated in any kind of computational caching. A subtle error involving the reset of the TLB's "used bit" upon replacement in a full TLB was detected in this way (an invalidation of the translation cache at that point was missing).

We have chosen to model the `SparcMMU::data_access()` interface as handling transfers of up to 64-bits wide, with the ASIs corresponding to special UltraSPARC memory transfers being handled by the memory access evaluation functions (logically part of the CPU)[3]. Thus, for example, a normal block load (64-byte) operation (ASI = `ASI_BLK`) is implemented by 8 calls to `data_access()` with an ASI value of `ASI_PRIMARY`.

The two levels of cache in UltraSPARC are implemented in a standard way, similar to SimOS/Mipsy and MIPS-64 Sulima, except that as this is a complete machine simulator, the proper UltraSPARC ASI-addressable register accesses (which access the cache entries directly) are explicitly implemented in these modules. Any special flags for cache access (e.g., non-allocation upon miss for a block load operation) are passed from the MMU to the caches.

The bus module implements the UltraSPARC System Bus (Sun Microelectronics 1997). It decodes the physical address, discerning between memory and devices accesses, and, for the case of the latter, invoking the `read()` or `write()` method for the corresponding device module. The address ranges for the latter must match those returned by the simulated PROM library (§6). It is not clear at this stage whether the lower-level buses (SBUS and PCI bus) need be explicitly modelled in a complete machine simulator.

Latencies due to cache and bus accesses are passed back up to the invoking CPU's memory operation evaluation function, and there the CPU clock is updated accordingly.

Aside from complexity, a serious problem with implementing the UltraSPARC MMU and System Bus is that the publicly available documents are inconsistent and incomplete in several details which might be important to complete machine simulation under an existing OS. The main strategy left to us for dealing with this problem is to study the SPARC-64 Linux or the recently released Solaris kernel sources, to see if these details are likely to matter to the OS, and if so to find what assumption the kernel developer has made.

[3]The UltraSPARC documentation does not make it clear what the actual MMU interface is, in this respect.

5 OS Emulation Mode

To aid in the evaluation and debugging of the simulator, an "OS emulation mode" has been developed. This mode can simulate specially linked 32-bit executables that behave as though they are running on the host machine (with respect to files in particular), albeit at a much slower rate (usually between 200 to 300 times slowdown).

The main advantage of this mode is that we can compare the output from the simulated program with that of an almost identical program that is directly executed. We also do not have the complexity of running a full operating system, but we can still test some of the MMU features such as the TLBs and caches.

The main features of the OS emulation mode that set it apart from normal simulation are that a special nucleus is provided that handles most basic exceptions and memory management, and most basic system calls are passed through to the native OS.

System calls are handled completely separately from memory management. The system calls are handled by hooks into the `illtrap` evaluation function (i.e., external to the simulated machine). The memory management is handled completely within the simulated program by nucleus code that sits in the PROM address space of the simulated System Bus.

5.1 Memory management in OS emulation mode

Since the reset and normal trap tables are included in the simulated code (as a separate file loaded at run-time) we can also test some of the MMU and caching systems. These trap tables are mapped in the I-TLB to a locked, privileged, 64K page where virtual address equals physical address. Nucleus data is given the bottom 8K (VA = PA) of physical memory. The nucleus is not given direct access to any other pages of memory, although it can access physical addresses directly through the ASI_PHYS_* alternate address space identifiers (ASIs) and user-level pages through the ASI_AS_IF_USER_* ASIs.

The user-level program is given a fixed amount of physical memory it can use (set at run-time before reset). The relationship between user-level virtual and physical addresses is VA = PA + offset, where offset is the base address of the executable minus the size of the nucleus data page (currently 8K). User level pages are set to 8K.

Valid user-level virtual addresses range from the base address up to the top of the stack, which is initialised to point to the top of physical memory. Nucleus code maintains a reference to the "brk" point, and any attempt to access a page between the brk point and the stack pointer fails (the simulation finishes with an error indication). Any attempt to grow the brk point into the stack fails, but no check is made for excessive stack growth.

Pages from the base address up to the bottom of the data are not writable, and any attempt to write to those pages will halt the simulation. All valid pages are able to be executed (i.e., can be mapped in the I-TLB).

This basic behaviour — a known fixed relationship between VA and PA, and a fixed set of checks to make to ensure validity — means that it is not difficult to hand code the assembler for the I-MMU and D-MMU miss handlers.

5.2 System call handling

The OS emulation mode uses a slightly modified version of the RSIM applications library (RSIM 1997) which provides a front-end to the standard basic system calls (`open`, `close`, `read`, `write`, `sbrk`, ...). It requires a special final (static) link to produce the executable but no other special (re)compilation is required.

All these system calls are reduced to a particular `illtrap` call, which, when evaluated, is intercepted by the OS emulation code to determine what system call (if any) to emulate. The parameters to the system call are in registers `%o0`, `%o1`, ... and any return value goes in register `%o0`.

5.2.1 Strings and buffers

Dealing with strings and buffers as input and output is reasonably difficult, because it is possible for the page being accessed to not be in the D-TLB. Due to the structure of the simulator we cannot generate the exception and continue the simulation such that the miss handler adds the entry to the TLB and returns to the point where it failed.

We must instead make the system call *restartable*, so that, up to some point in the emulation of the system call, no user-visible state is changed and the system call can be restarted if necessary.

If an exception occurs when reading from or writing to a buffer, the exception is generated and — as per our normal exception handling — the evaluation completes normally. The simulator will then call the miss handler, which will (presumably) add the page entry to the TLB and perform a `retry` instruction. The trapping instruction (in this case the `illtrap` system call) will be re-evaluated, this time with that address existing in the D-TLB[4].

The only problem with this behaviour is that all of the pages of the buffer must be able to be in the D-TLB at once. This limits the size of buffers used in system calls to 504KB — assuming one locked page and 63×8K pages. The only calls this affects are the read and write system calls, which can be worked around by writing a wrapper around the system call which repeatedly calls the system call one page at a time.

The `open` and `write` system calls provide read-only buffers. Reading from user-level simulated memory has no side-effect (other than a possible exception, which is handled as described above) so we simply copy the buffer into local memory before performing the equivalent system call on the host. The `write` system call handler is shown in Figure 4.

The `read` system call is more difficult since it is a write-only buffer. Since we perform the host read into a local buffer before attempting to copy the buffer into simulated memory, if the copy fails then the side-effect of doing the host read will still have occurred and data will be lost. We work around this problem by requiring that the entire buffer exists in the D-TLB before performing the host read; this is done within the `read` system call wrapper which does a dummy read from the buffer to ensure that it is in the D-TLB before calling the actual `read` system call.

[4]An alternative is to keep user-visible state up to date with what the system call handler is up to so it can continue where it left off. This would require changing the parameters to the system call (e.g., incrementing the buffer address register in the read and write system calls) before generating the exception.

```
int UserSim::handle_sys_write() {
  UInt64& o0 = get_o0();
  int fd = o0;
  VA buf = get_o1();
  UInt32 nbyte = get_o2();
  UInt8 localbuf[nbyte];
  int exc = 0;
  memcpy_from_sim(localbuf, buf, nbyte, &exc);
  if (exc) return exc;  // will retry on page miss
  UInt32 nwritten = files->write(fd, localbuf, nbyte);
  o0 = nwritten;
  return 0;
}
```

Figure 4: An example of a system call handler: the `write` system call. The actual implementation uses an optimisation where it creates `localbuf` with the same alignment as `buf`, so 64-bit copies can be used instead of 8-bit copies.

5.2.2 Files

Simulated file descriptors are mapped to host file descriptors via an array: the index is the simulated file descriptor and the value is the host file descriptor. This is the `files` member object used in the write system call handler in Figure 4.

5.2.3 Sbrk

The “sbrk” system call is special because the nucleus has control of the brk point, not the system-call handler. To satisfy this, the handler simply generates an implementation-specific exception (0x70) that is handled by the nucleus trap table. All the exception handler needs to do is check whether the requested change is legal (the parameter is given in register `%o0`), perform the change, store the old “brk” point in register `%o0`, and return successfully with the `done` instruction.

6 OpenBoot^{TM} PROM Library

OpenBoot is a Sun Microsystems trademark for the firmware that controls a computer before the operating system has begun execution. Open Firmware is the non-proprietary name of firmware complying with IEEE Std 1275-1994 (IEEE 1994*a*).

Open Firmware[5] is stored in a ROM (or more often a PROM) and is executed immediately after the computer is turned on. The main tasks of firmware are to: test and initialise the system hardware (POST: power-on self-test), determine the hardware configuration, provide a device interface to boot *plug-in* cards, initialise booting of the operating system and provide interactive debugging facilities for testing hardware and software.

The main advantage of Open Firmware is that new plug-in devices may be added to the system, including ones used for booting, without modification of the main Open Firmware system PROM. Each such device has its own driver[6] usually located in a PROM on the device itself. Thus, the set of I/O devices supported by a particular system is independent of the system PROM.

[5]In this paper we often refer to the Open Firmware as PROM.

[6]The device drivers are written in a byte-coded machine-independent interpreted language called FCode. FCode is based on Forth.

Often, as it is in the case of SPARC versions of Solaris and Linux, OS kernels use the PROM services to learn about the hardware configuration of the host and perform other, system related tasks. In our specific case we are interested in booting the Linux SPARC kernel[7] on our simulator.

Sun's OpenBoot 3.x source code was unavailable to us and therefore we had to find other ways to make Linux kernel calls to the firmware (PROM) satisfied. We have chosen to emulate PROM calls by using one of the OpenBoot interfaces, namely - the *client interface.*

6.1 OpenBoot Interfaces

The Open Firmware has three interfaces (Sun Microsystems 1997): user, device and client interface. The *user interface* provides a system administrator with a set of Open Firmware services or commands (FirmWorks 1998) for low level configuration management and debugging of hardware, software and firmware. The *device interface* enables Open Firmware to identify and use plug-in devices. The *client interface* (FirmWorks 1997) allows *client programs*, such as the kernel, to make use of services provided by the Open Firmware.

The central data structure that holds all the information about the particular hardware is called the *device tree.* It is a hierarchical data structure which mimics the organisation of the system hardware, viewed as a hierarchy of interconnected busses and attached to them devices. An individual *device node* represents either a hardware bus, a hardware device, or a set of interrelated software procedures or methods. The root node of a device tree is a node representing the computer's main physical address bus.

Each device node may have *children* (other device nodes directly subordinate to it), *properties* (externally visible data structures describing the node and its associated device), *methods* (software procedures that may be used to access the device) and *data* (initial values of *private data* used by the methods). A node with children usually represents a bus and its associated controlling hardware. Each node has a unique *node name.*

Properties describe characteristics of hardware devices, software and user choices. Both firmware procedures and client programs may inspect and modify properties. Each property consists of pairs, the *property name* and its associated *property value.*

The example of the Sun's "Happy Meal" Ethernet card node with its properties is shown on Figure 5. The other empty nodes are to highlight the hierarchical structure of the PROM's device tree.

6.2 Implementation

The simulation of the whole PROM is rather difficult since the details of the PROM code for a particular hardware is proprietary and certainly not open source. A closer look at the Linux kernel reveals that only small part of the whole PROM is used by the kernel. This made it possible to implement only the part of the whole PROM which is used by the Linux kernel. Other PROM-like functionality is directly implemented in Sparc Sulima itself.

We have chosen to emulate PROM functions by implementing our own device tree structure and the client interface. This solution requires only understanding of the PROM's client interface which is well described in (FirmWorks 1997, IEEE 1994*b*) and the source code for the client side of this interface.

[7] We have worked mostly with Linux kernel version 2.2.18.

```
/f0029828
---/f0064360 SUNW,UltraSPARC@0,0
   /f0059434 sbus@1f,0
   ---/f0072c10 SUNW,bpp@e,c800000
      /f006b1c0 SUNW,hme@e,8c00000
      | hm-rev          00 00 00 22
      | device_type     network
      | intr            00000021 00000000
      | interrupts      00000021
      | address-bits    00 00 00 30
      | max-frame-size  00 00 40 00
      | reg             0e 08c00000 00000108
      |                 0e 08c02000 00002000
      |                 0e 08c04000 00002000
      |                 0e 08c06000 00002000
      |                 0e 08c07000 00000020
      | name            SUNW,hme
      /f00647c4 SUNW,fas@e,8800000
      ---/f006a4d4 st
         /f0069c18 sd
...
```

Figure 5: Small fragment of Sun's UltraSPARC I OpenBoot PROM device tree. The root node has the address `f0029828` in the PROM's physical address space. The first child of the root node is the CPU's node itself (we have concealed the CPU's properties). The next node is the main SBUS node. From the many children of this node we have shown just the `hme` Ethernet adaptor with its *property names* and associated *property values.*

Contrary to the Open Firmware the Linux source is widely available.

The layout of the device tree structure is not so critical, yet following a real PROM's device tree structure will make the implementation easier. We have stored device tree nodes and properties in two separate arrays.

Each real PROM function call in the Linux kernel follows a well defined mechanism. The PROM is entered via the `p1275_cmd()` function. In this function, the particular PROM service request and its input and output parameters are decoded and placed in the `p1275buf` structure. Then `prom_world(1)` function is called which reinstates PROM TLB entries. Then the `prom_cif_interface()` function saves global registers and turns off interrupts before it hands control over to the PROM. After the PROM finishes all saved registers are restored and we exit PROM service via `prom_world(0)` call where Linux kernel's TLBs are restored.

In the first instance we have implemented the kernel's own PROM library, i.e., the emulated PROM function calls were handled by the kernel itself. Each of the hundreds of the Linux kernel PROM function calls were eliminated and replaced, one by one, by the kernel's own version of the function. The fact that the real PROM and the emulated PROM could coexist made testing and debugging the code much easier. Although such an approach will suffice it is not very elegant and requires a number of changes to the Linux kernel source.

In the second instance we have used the earlier described `illtrap` mechanism to service PROM calls from the simulator itself. This requires only one change to the kernel, namely to install `illtrap #` instruction instead of `p1275_cmd()` function call. Every time this particular `illtrap` is detected by the simulator, the pointer to the PROM's argument structure is passed and the corresponding emulated PROM function performed.

```
void UltraSAN::EvalUDIVcc (InstrOperands *ops) {
  ...
  trace(trace_instr,
        "%#llx:\tudivcc\t%s,%s,%s\t: "
          "gpr[%d]=%llu (CCR=%#lx)\t: %llu,%lu\n",
        pc, printGPR(rs1), printRegOrImm("%u",*ops),
        printGPR(rd), GPR_ix(rd), GPReg(rd),
        (UInt32) CCR, v1, v2);
}
```

Figure 6: Trace code for the UDIVcc instruction. This adds to code shown in Figure 2.

```
...
0x15e58: subcc  o0,-1,g0        : gpr[0]=1 (CCR=0x11) : 0,-1
0x15e5c: bne,a  icc,0x2         : npc=0x15e60 (=0x15e64)
         cache:fetch:  va=15e60  pa=7e60
0x15e60: or     g0,0,l0         : gpr[16]=0            : 0,0
0x15e64: or     i3,l0,i3        : gpr[27]=0            : 0,0
         cache:load:   va=520cc  pa=440cc  size=4  *data=0
0x15e68: lduw   [i0+68],i0      : gpr[24]=0(0)   : M[0x520cc]
0x15e6c: subcc  i0,0,g0         : gpr[0]=0 (CCR=0x44) : 0,0
0x15e70: bne    icc,0xffffffe2 : npc=0x15e74 (=0x15e78)
...
```

Figure 7: Example tracing output showing both instruction-level and cache-level traces.

6.3 Testing of the PROM library

The kernel-PROM traffic can be monitored in two ways: either by using facilities built in the PROM's user interface or on the side of the Linux kernel by tracing each PROM call. The simpler way is to use the PROM's user interface. By setting the PROM's `cif-verbose` flag to true and providing a way to capture large kernel-PROM traffic all the PROM function calls details can be obtained. We have used the serial communication port to interact with the PROM in a Sun Ultra-1. A small Forth program was written and implanted to the PROM to make the kernel-PROM traffic output more human readable. A comparison of data from real PROM call and emulated PROM call was used as a test for correct PROM function implementation.

The ultimate test was to use the emulated PROM along with the kernel-PROM traffic monitoring system in place during Linux boot and see no real PROM functions requests at all. We have tested a number of Linux kernel configurations in such way. The next goal is to intercept PROM calls completely at the simulator level and have no changes to the Linux kernel source at all.

7 Testing

One of the main issues with a large system like a complete machine simulator is testing and validating, particularly since it has very well defined, and complex, requirements. We discuss some methods of testing that we have used, including tracing and component-wise testing.

7.1 Tracing

A run-time switch enables tracing output with varying amounts of detail. For example, the tracing levels in the `UltraSAN` class from least to most verbose are:

1. no tracing;
2. report entry and exit of exceptions (one line each);
3. print instructions, including input and output values (one line);
4. print all current (general purpose) registers after every instruction.

The higher levels of tracing produce huge volumes of output. The code for the instruction-level trace for the `UDIVcc` instruction is shown in Figure 6, and some example tracing output is shown in Figure 7.

The caches, bus and MMU also have a unified trace mechanism, which provides for varying levels of output, triggering on none, very rare, rare, fairly common, and very common events. They can also produce binary output, which means the output can be generated much faster than text output; post-processing is needed in order to view the results.

When testing, we use tracing to assist us in isolating the point of erroneous execution, and then to find the state change that was incorrect. We can also relate it to the source program, using a disassembled binary.

Tracing has also helped with testing particularly complex instructions, such as memory operations. We have written programs in assembler to test special memory operations that are unlikely to be generated from high-level languages; the traces are used to verify the resulting simulation is correct.

7.2 Component-wise testing

One systematic method of development we have used is component-wise testing, where a component is systematically tested in isolation. This has been used in particular for the inline assembler evaluation functions (see §3.3).

For example, the floating-point operations were completely implemented and tested in isolation before integrating with the rest of `UltraSAN`. To do this, the core components of UltraSAN that were required by the floating-point instructions (in particular, the floating-point registers, and the floating-point status registers) were stripped out into a separate program.

As each floating-point operation was implemented it was tested individually with several sets of inputs. When the resulting output had been manually passed it was saved so future changes or overall optimisations could be automatically tested.

This framework enabled the floating-point operations to be completely implemented and thoroughly tested in about a week. Integration with `UltraSAN` took about a day.

7.3 Booting Linux

The ideal complete machine simulator will boot an unmodified operating system. As was described in §6 we have developed a PROM device to assist us in booting Linux.

We have prepared a minimal Linux kernel which supports a very small number of devices, and does not have an `init.d`, but we have not yet attempted to boot it as the device support is incomplete.

8 Performance

The performance of complete machine simulators is generally specified in terms of a *slowdown*: the ratio of the time it takes to simulate a workload and the time it takes to execute that workload on an actual system.

The performance of other simulators varies considerably, mostly depending on their timing accuracy to the real system. For example, Embra, the binary translating core of SimOS, has a slowdown of only 3 to 9 times, but the level of detail and accuracy provided by Embra means it is best used for *positioning* the workload before using a more detailed, accurate core

compiler	exe	CC 64-bit						CC 32-bit	g++ 2.95.2	g++ 3.0.2
optimisation	-	none	gprs	dc	mmutc	all	all+ipo	all	all	all
time (s)	2.12	1934	1671	1269	894	626	590	744	895	911
slowdown	1	912	788	599	422	296	278	351	422	430

Table 1: Performance comparison of various optimisations of Sparc Sulima OS emulation mode with the `bzip2test` program. The input is a 213 KB PNG image that does not compress well. "exe" is the native executable. "CC 64-bit" is Forte CC 5.3 (WS6U2), options: `-x05 -xtarget=ultra -xarch=v9a` (64-bit); "CC 32-bit" uses `-xarch=v8plusa`; the g++ versions use `-O2 -mcpu=ultrasparc`. "Optimisations" are: "gprs": scratch set of gprs (§3.4); "dc": decode caching (§3.1); "mmutc": MMU translation caching (§4); "all": gprs+dc+mmutc; "+ipo": use Forte CC's *interprocedural optimisations* (this increases link-time considerably). The host is a Sun Ultra-1 Creator, 167 MHz UltraSPARC, 512 KB E-cache and 256 MB of RAM, running Solaris 8.

(Mipsy or MXS). Mipsy typically has a slowdown of 100 to 200 times, and MXS several thousand times. SimICS claims a slowdown of around 100 times.

Performance is a critical issue in complete machine simulation: with a real workload containing billions or trillions of instructions, we do not want to have to wait days for a result. If a typical workload is only 10 minutes on the real machine, then a simulator slowdown of 400 times would correspond to just over a day of simulation time. As an aside, if we had instruction tracing on for the whole time that would generate around 2.5 TB of data.

We have already discussed performance issues with respect to Sparc Sulima earlier in this paper: instruction decoding and caching (§3.1), inline assembler evaluation functions (§3.3), general purpose register access (§3.4), avoiding "long jumps" in handling traps (§3.5) and caching translations in the MMU (§4).

The performance evaluation of Sparc Sulima to date has used some basic programs being simulated in OS emulation mode (§5). Our main benchmark is a compression and decompression program. It reads in a large block of the input, and then uses the bzip2 library (Seward 2000) to compress and decompress that block, and then compares the original block with the decompressed data. This avoids many system calls (only `read` is used, and then not many times) which would distort comparisons with the directly executed program, and also uses a lot of memory.

Table 1 shows the integer and memory performance of the simulator in OS emulation mode. Note that the slowdown is not a true slowdown, since it uses host system calls to emulate the target system calls. Hence the actual slowdown for a complete machine simulator would be higher.

Clearly the optimisations we have used have made a significant impact on the performance of the simulator: the optimised simulator is 3 times faster than the unoptimised simulator. The most improvement comes from the MMU translation caching. For this benchmark, the MMU translation caching and the decode caching have a hit-rate of between 98–99.9%.

The choice of compiler and architecture (32 or 64-bit) also has an impact: the g++-compiled simulators (currently 32-bit only) are quite a bit slower than the CC 32-bit compiled simulator, which is slower than the 64-bit simulator. The latter difference is not surprising since the architecture we are simulating has many 64-bit operations and the 64-bit ABI is more suited to passing around 64-bit values.

Table 2 shows the floating-point performance of Sparc Sulima's OS emulation mode with the Linpack benchmark.

The floating-point performance of the simulator is similar to the `bzip2test` results. Notably, the +ipo-enabled (interprocedural optimisation) simulator was slower than the normal version with a matrix size of 1000: this is probably due to host I-Cache thrashing (the +ipo executable is half again as large as the normal version). With a blocking factor of 1 (which uses a less efficient algorithm and thrashes the caches) the simulator performs much better: this suggests the simulator's cache replacement is relatively more efficient than the hardware. With a blocking factor of 64 the native Linpack is highly optimised so the simulator's slowdown is not likely to be as good as with normal, less-optimised programs.

optimisation	exe	all	+ipo	exe	all
500×500	bf=64			bf=1	
time (s)	1.14	304	294	3.11	737
slowdown	1	267	258	1	237
1000×1000					
time (s)	6.78	2226	2546	23.2	4139
slowdown	1	328	375	1	178

Table 2: Floating-point performance of Sparc Sulima OS emulation mode using the Linpack benchmark. Compiler was CC 64-bit; "bf" is the blocking factor; for other details see Table 1.

In profiling the simulator, the critical phases in the simulator are: (a) the main run/dispatch loop, (b) MMU data access, (c) instruction decoding, and (d) d-cache load and store. (a) and (c) are difficult to optimise any more. (b) is complicated because of the number of checks required before allowing each data access, but is still an area where optimisations can be made; specialising by data size (using templates) may help here (and with the caches). (d) can probably be optimised somewhat by unifying the d-cache and e-cache, since the d-cache is write-through.

With a slowdown of around 200 to 300 times, the simulator is comparable to other state-of-the-art simulators. This is despite the complexity of the instruction-set and MMU architecture: this complexity means that the regular instruction path is quite long, and that it is likely that we are frequently missing in the I-cache.

9 Conclusions and Future Work

The implementation of the Sulima UltraSPARC complete machine simulator has so far presented several major challenges, due to the complexity of the UltraSPARC ISA and the lack of information on its lower-level sub-systems. We have incorporated several new strategies in order to improve the readability and/or reliability of the system, including heavily object-oriented implementation to improve modularity and re-use, machine-assisted decoding, the use of inline assembler for computational instruction evaluation, a streamlined trap architecture implementation and PROM library emulation. A major new feature in a complete machine simulator is the capability to dynamically load in an OS emulation mode, which works together with a minimal nucleus loaded from ROM at simulated boot time. For performance improvement, we have implemented caching of expensive computations performed by the simulator: instruction decodes, windowed register access, and MMU translations. With these optimisations we have achieved a slowdown of between 200 to 300 times. Even so, performance will remain a problem for UltraSAN due to the complexity of the 64-bit ISA being simulated, and the emphasis on reliability and modularity we have taken in our approach to modelling the ISA.

Because debugging of such a simulator while running a long instruction sequence (e.g., an OS boot) is potentially very difficult, we have implemented several worthwhile strategies for component-wise testing: separately-callable instruction evaluation functions, tests involving synthetic assembler sequences, and the testing of the emulated PROM library on a real system. The restricted boot modes (boot from `main()` and the OS emulation mode) also help testing and debugging by permitting the simulation of simpler and shorter test programs than is possible by a full OS boot. A multi-level tracing facility which can be independently activated across the main simulator modules is also important. Where a high-level of detail is required, we have found it essential to include the results of the state change in question to be included.

However, there is still a significant amount of work top be done before the most critical stage of this project can be achieved: in this case successfully booting SPARC Linux, and then analysing a workload. In terms of functionality, minimally console and disk devices must be implemented (possibly requiring modified device drivers to be included in the kernel), SMP functionality must be implemented in the E-cache and System Bus modules, statistics collection facilities must still be added, and improved debugging infrastructure (possibly including a `gdb` interface) developed. Debugging the boot of a complex existing OS is analogous to porting the OS to a similar but potentially erroneous real computer system: it requires also detailed knowledge of the OS kernel as well. There is surprisingly little described on this critical process in the SimOS and SimICS literature.

Other forms of future work includes implementing a binary translation 'core' for improved performance.

Acknowledgements

We would like to thank Fujitsu Laboratories for their sponsoring of the ANU-Fujitsu CAP Program under whose umbrella the work on Sparc Sulima has been undertaken. The authors would also like to thank Patrick Fagan, former member of the Sparc Sulima team, for his contributions.

References

Bose, P. & Conbte, T. M. (1998), 'Performance Analysis and its Impact on Design', *IEEE Computer* pp. 41–49.

Clarke, B. (2000), SPARC V9 instruction set specification, Technical Report TR-CS-00-03, Department of Computer Science, Australian National University.

FirmWorks (1997), *Open Firmware Client Interface Developer's Guide*. Revision E.

FirmWorks (1998), *Open Firmware Command Reference*. Revision H.

Herrod, S. A. (1998), Using Complete Machine Simulation to Understand Computer System Behavior, PhD thesis, Stanford University. viii+120p.

IEEE (1994*a*), *IEEE Std 1275: IEEE Standard for Boot - Initialisation Configuration - Firmware: Core Requirements and Practices*.

IEEE (1994*b*), *IEEE Std 1275.1: Sparc ISA Supplement*.

IEEE (1994*c*), *IEEE Std 1496: IEEE 1496 (SBus) Bus Supplement*.

Magnusson, P. S., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Lundholm, F., Moestedt, A., Nilsson, J., Stenstrvm, P. & Werner, B. (1998), SimICS/sun4m: A Virtual Workstation, *in* 'Usenix Annual Technical Conference'.

Pai, V. S., Ranganathan, P. & Adve, S. V. (1997), RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors, *in* 'Proceedings of the Third Workshop on Computer Architecture Education'. Also appears in IEEE TCCA Newsletter, October 1997.

Ramsey, N. & Fernández, M. (1998), 'New Jersey Machine-Code Toolkit', `<http://www.eecs.harvard.edu/~nr/toolkit/>`.

Ramsey, N. & Fernández, M. F. (1997), 'Specifying representations of machine instructions', *ACM Transactions on Programming Languages and Systems* **19**(3), 492–524.

Ranganathan, P., Gharachorloo, K., Adve, S. V. & Barroso, L. A. (October 1996), Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors, *in* 'Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems', pp. 12–23.

Rosenblum, M., Herrod, S. A., Witchel, E. & Gupta, A. (1995), 'Complete Computer System Simulation: The SimOS Approach', *IEEE Parallel and Distributed Technology* .

RSIM (1997), 'RSIM home page', `<http://rsim.cs.uiuc.edu/rsim/>`.

Seward, J. (2000), 'bzip2 and libbzip2', `<http://sourceware.cygnus.com/bzip2/>`.

Strazdins, P. (2000), A survey of simulation tools for CAP project phase III, Technical Report TR-CS-00-02, Department of Computer Science, Australian National University.

Sun Microelectronics (1997), *UltraSPARC User's Manual: UltraSPARC-I and UltraSPARC-II*, Palo Alto, California.

Sun Microsystems (1997), *OpenBoot 3.x Command Reference*. Revision A.

van Rossum, G. et al. (n.d.), 'Python 2.1', `<http://www.python.org>`.

Virtutech (2000), 'Simics 1.0', `<http://www.simics.com/>`.

Weaver, D. L. & Germond, T., eds (2000), *The SPARC Architecture Manual, Version 9*, PTR Prentice Hall, SPARC International, Inc., Santa Clara, California.

Werner, B. & Magnusson, P. S. (1997), A Hybrid Simulation Technique for Enabling Performance Characterization of Large Software Systems, *in* 'Mascots 97', pp. 73–80.

Zadarnowski, P. (2000*a*), The design and implementation of an extendible instruction-set simulator, BE thesis, School of Computer Science and Engineering, University of New South Wales, Australia.

Zadarnowski, P. (2000*b*), 'Sulima ISA simulator', `<http://www.cse.unsw.edu.au/~disy/Sulima/>`.

Zadarnowski, P. (2000*c*), 'The Sulima simulator', Private communication.