

# Experiences in Developing the Sparc-Sulima Complete Machine Simulator

Peter Strazdins,  
(with Bill Clarke and Adam Czezowski)

Department of Computer Science,  
Australian National University,  
and  
The ANU-Fujitsu CAP Program

<http://cs.anu.edu.au/~Peter.Strazdins/seminars/SulimaDCS>

12 June 2002

# 1 Talk Outline

- performance analysis of computer systems
- motivations for simulation (of computer systems via software)
- simulation: basic concepts
- challenges for complete machine simulation
- challenges for SMPs
- the state-of-the-art in complete machine simulation
- the UltraSPARC ISA
- the Sparc-Sulima project
- the structure of Sparc Sulima
- implementation: instruction decoding & evaluation, memory system, OS-emulation mode, scripting interface, and PROM emulation
- performance and current status
- conclusions and future work

## 2 Performance Analysis of Computer Systems

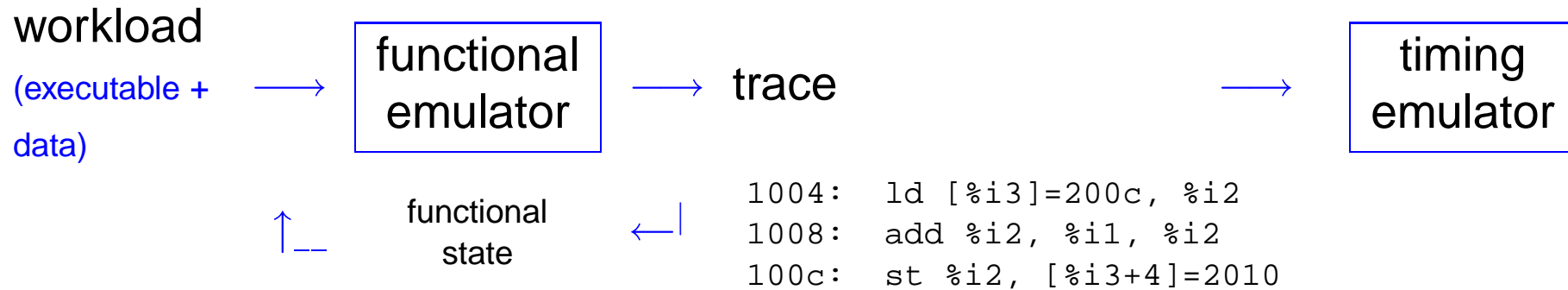
- One cannot understand the design tradeoffs or performance of multi-processors without understanding the interaction of algorithms and architecture – John L. Hennessey, 1999
- systematic performance analysis techniques now drives design
  - driven by key applications, and benchmarks (simplified programs derived from such applications)
    - traditionally based on scientific/engineering applications, eg. SPEC
    - increasingly based on commercial applications, eg. databases (TPC)
  - includes the use of performance instrumentation libraries:
    - here, we insert calls to manipulate hardware event counter registers in the source code
    - events affecting performance include branch mispredictions, TLB misses, cache misses *etc* (usually, memory-related events are the most important!)
  - and computer simulation:
    - use software to interpret programs and predict their performance
- events are categorized into user-level and (operating) system-level
  - *ie.* those resulting from executing traps (via system calls)

### 3 Motivation: why use computer simulation?

- or, why not just run and time the application on the computer itself?
- advantages of **simulating** applications:
  - can have full visibility: actual H/W may not count all events of interest
  - the simulated computer may not even exist! (essential for design!)
    - can use any suitable host computer to run the simulator
  - can vary parameters for architectural studies (eg. cache size)
- user-level computer simulation: run executable program; when a trap instruction is simulated, use OS on the host computer to perform the corresponding action
- complete computer simulation: boot the simulated computer from its OS's kernel image; can then run applications from the (simulated) shell
  - ✓ many commercial applications spend 30% time in system-level code
  - ✓ can accurately capture TLB miss / page fault events
  - ✓ for operating systems:
    - development / debugging (can make the simulation *deterministic*)
    - performance analysis

## 4 Simulation: basic concepts

- trace-driven simulation:



- ✓ has separation of concerns

- ✓ seen as a good performance–accuracy tradeoff

- especially fast if functional emulator is an ‘instrumented’ executable

- ✓ can ‘skip’ or sample traces passed to timing emulator

- execution-driven simulation: combined functional & timing emulators

- ✓ higher accuracy possible (also proper correctness for SMPs)

- **basic idea:**

- simulator runs as a single user-level process on the host machine:
- interpret each instruction; update (functional & timing) state; collect desired events

## 5 Challenges for complete computer simulators

- functional state:
 

<ul style="list-style-type: none"> <li>● <u>user-level only</u>:</li> <li>● (user-level) registers</li> <li>● a single VM space</li> <li>● emulate system calls</li> </ul>	<ul style="list-style-type: none"> <li>● <u>complete</u>:</li> <li>● all registers</li> <li>● multiple VM address spaces</li> <li>● emulate devices (boot PROM, DMA, disk, network, ...)</li> <li>● MMU/TLB and physical memory, bus</li> </ul>
--	---
- info. for bus and device behaviour is scant, conflicting or unavailable!
- must *efficiently* perform address translation (VM→PM, ie. model the MMU)
- high performance is required for realistic applications
  - eg. simulate the booting process ( $\approx 10^9$  instructions!)

and also high reliability! (debugging is very hard!)
- need to be able to simulate arbitrary, (preferably) un-modified executables (+ OS boot images)
  - eg. 'badly-behaved' processes, dynamically loaded libraries, self-modifying code, ...

## 6 Challenges: shared memory issues

- kernel-level memory behaviour becomes more significant for SMPs
- cannot (accurately) use trace-driven simulation:
  - timing issues can affect *which* instructions get executed, values of load instructions (eg. acquiring a spinlock in SMP execution)x
- cache coherency protocols:
  - must be modelled faithfully, and yet efficiently
- to simulate shared memory parallelism, can either:
  1. a single process simulates  $s$  cycles of each CPU in a ‘round-robin’ fashion
    - switching between CPUs must be made efficient!
    - $s$  must be small for accurate CPU interleavings
  2. have a separate simulation thread per CPU
    - SMP parallelism on host machine possible
    - frequent synchronizations required for accurate CPU interleavings
      - upon every possible simulated access, ie. every cycle! requires  $\approx 10^3$  host cycles!

## 7 The State-of-the-art in complete SMP machine simulation

- many user-level simulators, eg. MINT (MIPS, 1994), RSIM (SPARC V8+, 1998)
- complete machine simulators are complex and **HUGE!** (and very few!)
- SimOS (MIPS/Alpha, 1996–1998)
  - ✓ used to develop Hive OS; can have `gdb` ‘back-end’
  - ✓ source code available; has 3 compatible simulator ‘cores’
    - Embra: based on dynamic binary translation 5–10× slowdown
    - Mipsy: interpreted, 2-levels of caches  $\approx 100\times$  slowdown
    - MXS: cycle accurate CPU & memory modelling;  $\approx 1000\times$  slowdown
  - ✗ not designed to port! code very complex; little/out-of-date doc.
- SimICS (M8810/SPARC V8, 1998; SPARC V9/others, 2000–1)
  - similar functionality to SimOS, detail as for Mipsy  $\approx 100\times$  slowdown
  - ✓ cluster support; modular, object-oriented, relatively portable
  - ✗ source code unavailable, licenses were expensive!
- Halsim (SPARC V9, 2001) and other proprietary simulators

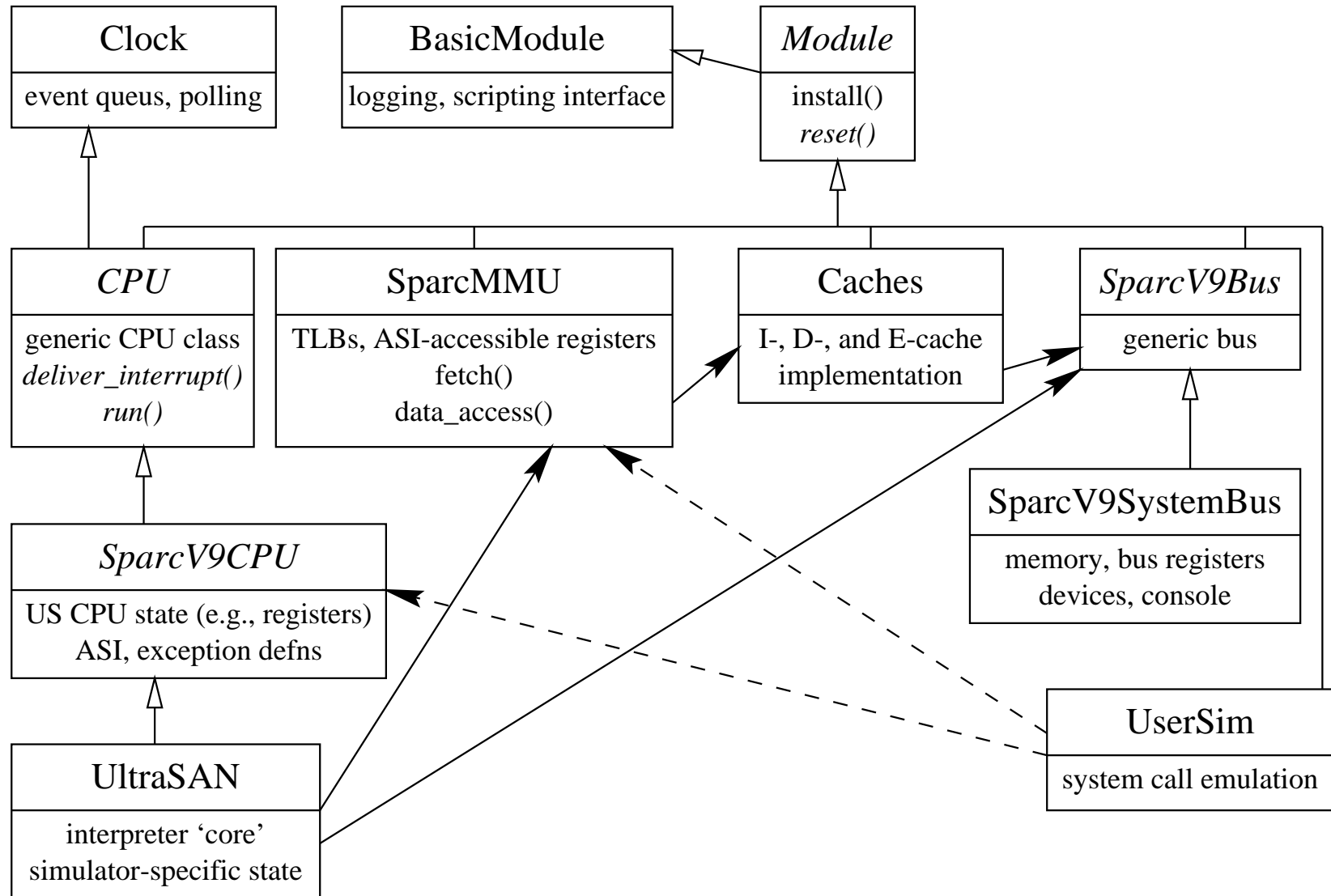
## 8 The UltraSPARC ISA

- RISC architecture (but not so *reduced* any more)
- an implementation of the 64-bit SPARC V9 architecture
  - $\approx$  320 instruction types; complex (sub-) formats
  - complex operations
    - 32-bit & 64-bit integer and floating point, graphics, block memory accesses, . . .
  - 32 64-bit GPRs (general purpose registers) with sliding windows and 'alternate' global register sets
  - 32 single, 32 double, 16 quad overlapping floating-point registers
  - fast trap handling with 5 levels of traps (exceptions)
  - 64-bit address space (and 32-bit backward-compatible)
  - highly complex Memory Management Unit
    - large range of Address space identifiers; little-endian support
- arguably, much more complex than other RISC processors
  - thus, more difficult to construct as robust *and* fast simulator for!
  - software emulation must serialize the complex operations that can be parallelized by the hardware!

## 9 The Sparc-Sulima Project

- the Sulima project from the DiSy Group at UNSW (1998-2000)
  - 64-bit MIPS simulator, grew out of a frustrated SimOS port
  - heavily OO approach designed for portability and modularity
  - C++ source available under BSD (07/00)
  - more or less a prototype; still to boot the L4 microkernel successfully
- a brief history of the Sparc-Sulima project
  - 10/99: CAP Phase III began; required a tool to investigate kernel-level memory behaviour on SPARC V9 SMPs
  - 01/00: initial investigation: an execution-driven CMS needed
  - 08/00: SimOS and other alts. rejected; design of Sparc-Sulima began
  - 12/00: ran 'Hello World!'
  - 07/01: most CPU / memory system functionality added; PROM & system call emulation libraries established
  - 10/01: fully optimized; achieved goal of  $200\times$  slowdown
  - 12/01: first code release (under GPL / BSD)
  - 03/02: SMP bits completed; 1st attempts to boot Sparc-Linux

# 10 Structure of Sparc Sulima



## 11 Instruction decoding

- need to efficiently decode 32-bit instruction → opcode and its operands
- usual method is via jump tables or functions
  - × prone to error, hard to debug, hard for a complex ISA
- we used SLED (the Specification Language for Encoding and Decoding) to specify the syntax of the UltraSPARC ISA
  - ✓ njmctk (New Jersey Machine-Code Toolkit) can automatically generate decoders and encoders from a SLED specification
    - can also verify the specification with aid of an external assembler
- the Sparc-Sulima main instruction decoder distinguishes 252 instructions
  - is  $\approx$  6000 lines of C code
- an 8-word instruction buffer is also used to reduce overhead of simulated fetches (as on real machine)

## 12 Instruction evaluation and Black-box simulation

- decoder produces an opcode and an operands structure
- 326 instruction evaluation functions in a function table (indexed by opcode)
- the decoding is cached in the simulated I-Cache to save repeated decoding (like the UltraSPARC's pre-decode fields)
  
- 43 integer instructions:  $rs1 \odot rs2\text{-or-imm} \rightarrow rd$  (with side-effects)
- some of these are quite difficult to (reliably) specify in C++ (esp. condition-code setting)
- also do floating-point and (complex!) graphics instructions
- potential optimisation for simulating SPARC V9 on SPARC V9: use the exact same instruction!
- have implemented in g++ inline assembler and Forte CC external inline templates (implementation is rather esoteric)

## 13 Memory system modelling

- is quite complex: contributes to 1/4 of SPARC-specific code
- interface is basically load from and store to an address
- ASIs (address space identifiers) change the behaviour of the load/store
  - for UltraSPARC, many ASIs are privileged and are used for manipulation of the memory system
- normal loads and stores need to be translated from the virtual address to the physical address via a lookup in the TLB (translation lookaside buffer)
  - we implement this with both a translation cache and a naive direct implementation as fallback
  - translation cache lookup must fail if an exception is possible
- caches (data, instruction and external) are implemented directly, each with their own copies of the data C++ templates express common parts)
  - MOESI copyback-invalidate coherency protocol implemented without modelling individual bus transactions
  - round-robin CPU scheduling is used
- the bus is currently fairly basic but has RAM and slots for some simple devices (like a PROM or a console)

## 14 OS-emulation mode

- originally for debugging and evaluating the performance of the simulator
- can simulate statically linked `v8plusa` executables
  - linked with a special C library (from RSIM)
  - C library replaces standard system calls with an `illtrap` instruction
    - `read`, `write`, `open`, `sbrk`, ...
  - pseudo-PROM device intercepts the `illtrap` and interprets the results
- the MMU is fully enabled and traps are handled by a special nucleus (in a pseudo-PROM device)
  - here,  $VA = PA + \text{fixed offset}$  ( $\rightarrow$  simple code for TLB replacement)
  - buffer management from the nucleus is tricky
    - what happens when a buffer points to memory that is not in the TLB?
    - system calls need to be restartable!
- for SMP support, most flexible approach is to emulate lowest-level threads  
library (unlike RSIM; degree of difficulty unknown)

## 15 Scripting Interface

- objective: to conveniently access (selected) Sparc-Sulima methods and members from a scripting interface
  - eg. top-level `reset()` and `run()` methods, register files
- the original Sulima Tcl interface not satisfactory:
  - no 64-bit integer support
  - any script-accessible objects could have limited type signatures; linkages had to be explicitly created inside Sulima code
  - could not test any top-level modules without it
- needed to replace original Tcl interface with Python (also has object/array support)
- direct Python interface had to be abandoned (Oct '01) (proved not sufficiently reliable)

- **solution: SWIG (Simplified Wrapper and Interface Generator)**  
a tool to connect C/C++ programs to Perl, Tcl, Python, etc
  - create `.swg` files shadowing the `.h` files
    - semi-automated (eg. must remove any function bodies)
  - for Python version, top-level Makefile creates `SparcSulima-module.so`  
and `SparcSulima.py` (contains all Python-C++ linkages)
  - eg. `runsim-swig.sh MatFact.sim -w 64 -C 100`  
`runsim-swig.sh` contains:

```
...
import SparcSulima
im = SparcSulima
bus = sim.SPARCV9SystemBus("bus")
cpu = sim.UltraSAN("cpu", 200L)
cpu.bus = bus
user = sim.UserSim("user")
user.exe_filename = "$1"
user.set_args(['printf "%s", ' "$@";printf "\n"'])
prom = sim.ROM("prom")
...
sim.reset()
sim.run()
```

## 16 PROM and Console Modelling

- functional emulation is sufficient for our objectives
  - challenges lie in their complexity and the lack of documentation
- PROM emulation: (cannot find enough info. on PROM to be able simulate it!)
  - the Sparc Linux kernel accesses the PROM (mainly for low-level system info.) via calls to `p1275cmd( )`
  - replaced these calls with calls to emulated function (bypassing the real PROM)
    - has image of the PROM device tree; returns same information
    - was tested and debugged on a real Ultra-I!
  - for the Sparc-Sulima version of the kernel
    - need only replace body of `p1275cmd( )` with an `illtrap 999 instr'n`
    - the `p1257buf` structure is accessed as per OS emulation mode
    - sample test program (from code fragments in Linux kernel) is provided

## 17 Performance and Current Status

- performance: main technique was to ‘cache’ expensive calculations repeatedly carried out by simulator:
  - dc: recent instruction decodings
  - gprs: register window-related calculations
  - mmu: virtual → physical addresses, incl. TLB lookup

bzip2test

compiler	exe	cc, 64-bit						cc 32-bit	g++ 2.95.2	g++ 3.0.2
optimisation	-	none	gprs	dc	mmu	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

- now, for the complete computer simulator part ...
  - booting Sparc Linux:
    - ‘crashes’ at about 56K instructions into the boot sequence ...
      - due to a memory access that only the PROM can translate
      - on a real U-S, the PROM’s trap handlers are still being used and deal silently with this
    - solving such problems is very tough!
  - console and disk implementation to be completed

## 18 Conclusions and Future Work

- Sulima's OO approach is appropriate, but is no silver bullet!
  - proportion of original UNSW codes is now insignificant. . .
  - an inherently complex task; the devil is in the details!
- performance target was achieved but at expense of greater complexity
- debugging can be very hard; use traces but too low-level and voluminous
  - large number and range of (C / assembler) test programs
  - some defensive programming methods used (eg. self-checking trans. cache)
  - an open problem to develop better techniques
- booting a full OS is very tough for anyone except the vendors . . .
  - wish to achieve this, and add code for event collection
- future possibilities:
  - cluster version of Sparc-Sulima (Hons/MSc)
  - improving the portability of Sparc-Sulima (Hons/BSEng)
  - quantum chemistry applics. on UltraSPARC III SMPs (ARC Linkage proposal with Sun) further extensions to Sparc-Sulima envisaged!