Simulation Techniques for NUMA Architectures: Performance Evaluation via Simulation and Hardware Counters

Peter Strazdins,
(with Alistair Rendell, Bill Clarke and Andrew Over)

The CC-NUMA Project,
Department of Computer Science,
Australian National University,

http://cs.anu.edu.au/~Peter.Strazdins/seminars/SimTechNUMA

18 August 2003
1 Talk Outline

- performance analysis of CC applications on NUMA architectures
- the Sparc-Sulima UltraSPARC SMP simulator
  - history & approach (as a complete machine simulator)
  - object-oriented structure
  - OS-emulation modes
  - implementation issue
  - performance & portability
  - control and debugging infrastructure
- status and future challenges
2 Performance Analysis of CC on NUMA

- isolate linear scaling kernels in Gaussian
  - primarily user-level, and (SMP) memory effects of most interest
  - parallelize with special emphasis on data placement
  - thread affinity issues also important
  - use libcpc to instrument kernels; obtain useful statistics such as

- limitations of using instrumented benchmarks:
  - some useful information may not be available
    - eg. E-cache misses by type (conflict, invalidation) bus contentions, success of prefetches
    - counts of such events do not necessarily translate to cycles lost
  - results on #’s of CPU limited to available hardware
    - larger systems may have qualitatively different NUMA effects! (eg. extra level of interconnect, contention)
  - results are also limited to actual cache / interconnect configurations
    - interesting to explore variants, even completely different models

In principle, a simulator can supply all such information . . .
3 The Sparc-Sulima Project

- the Sulima project from the DiSy Group at UNSW (1998-2000)
  - (prototype) 64-bit MIPS simulator, grew out of a frustrated SimOS ‘port’
  - heavily OO approach designed for portability and modularity
- a brief history of the Sparc-Sulima project:
  - 10/99: ANU-Fujitsu CAP Program (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
  - 10/01: Ultra I/II functionality; boot (limited) executables; fully optimized; achieved goal of $200 \times$ slowdown
  - 12/01: first code release (under GPL / BSD)
  - 10/02: 2nd release (SWIG/Python scripting interface, annotations & tracing infrastructure, SMP)
  - 01/03: development continued under CC-NUMA Project
  - 08/03: Solemn emulation mode (dynamically linked executables)
4 The Sparc-Sulima Approach

- UltraSPARC is an an implementation of the 64-bit SPARC V9 architecture
  - RISC, but not so \textit{Reduced} any more!
    - eg. \( \approx 320 \) instruction types; complex operations; sliding GPR windows, ‘alternate’ global register sets SPARC V8 (32-bit) backward compatibility, 5 trap levels, . . .
  This has an impact on a (robust and fast) simulator design!
- model all aspects of (US-I/II) architecture explicitly using a heavily OO approach
  - eg. D-cache data as well as tags explicitly modelled; E-cache module only accessed upon (simulated) miss
  - however, bus coherency transactions not explicitly modelled
- CPU modelled by a fetch-execute-decode core (UltraSAN)
  - generally deemed suitable for analysing memory performance (speed-accuracy tradeoff)
  - nominally 1 cycle per instruction, with (possible) added latency from memory system
5 Structure of Sparc Sulima

- **Clock**
  - event queues, polling

- **BasicModule**
  - logging, scripting interface

- **Module**
  - install()
  - reset()

- **CPU**
  - generic CPU class
  - `deliver_interrupt()`
  - `run()`

- **SparcMMU**
  - TLBs, ASI-accessible registers
  - `fetch()`
  - `data_access()`

- **Caches**
  - I-, D-, and E-cache implementation

- **SparcV9Bus**
  - generic bus

- **SparcV9SystemBus**
  - memory, bus registers, devices, console

- **UltraSAN**
  - interpreter ‘core’
  - simulator-specific state

- **UserSim**
  - system call emulation
6 Emulation Modes in Sparc-Sulima

- but isn’t it a complete machine simulator?!? Has emulation modes:
  - ‘boot from `main()`’: executable uses `stdin`/`stdout` only; can optionally install a (tiny) nucleus:
    - enable MMUs, trap handlers; \( PA = VA \)
    - runs in privileged mode! can run SPMD-style SMP programs
  - User-sim: emulate at system call level, as for RSIM (but no SMP)
    - boot from pseudo-PROM; nucleus is non-trivial; \( PA = VA + \text{offset} \)
    - limited \( v8plusa \) executable must be specially (statically) linked
    - CMS issue: memory-accessing calls must be restartable! (TLB miss)
  - Solemn: emulate Solaris at system trap level
    - implements `mmap`, dynamic linking; \( PA = f(VA) \)
    - can simulate most 32/64-bit executables unmodified; (will be) sufficient for CC-NUMA
  - there is a continuum between the user level and CMS approaches
    - to extend Solemn for `lwp` traps: expand nucleus to simulate as much as possible: captures memory overheads (of a minimal OS)
7 Implementation Issues

- highly reliable instruction decoding using SLED / njmctk tools to specify the syntax of the UltraSPARC ISA

- optimizations: use caching of expensive calculations:
  - instruction decoding, GPR indexing and address translations
  - lookup on latter must fail if an exception is raised

- use of inline assembler to capture precise semantics of f.p., graphics & some integer instructions
  - optimisation for SPARC V9 hosts!

- caches (data, instruction and external) are implemented directly, each with their own copies of the data
  - 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)
8 Performance and Portability Issues

- performance: main technique was to ‘cache’ expensive calculations repeatedly carried out by simulator:
  - dc: recent instruction decodings
  - gprs: register window-related calculations
  - mmu: virtual $\rightarrow$ physical addresses, incl. TLB lookup
- on a US-I host:

<table>
<thead>
<tr>
<th>compiler</th>
<th>exe</th>
<th>cc, 64-bit</th>
<th>cc 32-bit</th>
<th>g++ 2.95.2</th>
<th>g++ 3.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimisation</td>
<td>none</td>
<td>gprs</td>
<td>dc</td>
<td>mmu</td>
<td>all</td>
</tr>
<tr>
<td>time (s)</td>
<td>2.12</td>
<td>1934</td>
<td>1671</td>
<td>1269</td>
<td>894</td>
</tr>
<tr>
<td>slowdown</td>
<td>1</td>
<td>912</td>
<td>788</td>
<td>599</td>
<td>422</td>
</tr>
</tbody>
</table>

- slowdowns of $\approx 200$ achieved for the Linpack benchmark
- code bloat problems (l-cache misses on the host):
  - above figures from 10/01; seems slightly little slower now . . .
- have recently made portable, using autoconf; runs on a Power PC
  - endianness is still an issue
Control & Debugging Infrastructure

- scripting interface via Python (has 64-bit/object/array support)
- use 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 debugging, use tracing of exceptions, calls and instructions (the latter two are highly customizable and reasonably fast loadable modules)
  - also recently added gdb support
- for (workload-specific simulator control), use scipt-accessible annotation classes, with hooks from UltraSAN run loop, MMU etc
  - main effect: manipulate simulator control variables
    - controlling start/stop, tracing, event collection
  - use external state of corresponding module, plus own internal state
10 Scripting Interface: Example

- `runsim-swig.sh MatFact.sim -w 64 -C 100`
- `runsim-swig.sh contains:

  ```
  ...
  import SparcSulima
  sim = SparcSulima
  bus = sim.SPARCV9SystemBus("bus")
  cpu = sim.UltraSAN("cpu", 200L)
  cpu.bus = bus

  user = sim.UserSim("user")  # load user-sim module
  user.exe_filename = "$1"
  user.set_args(['printf "%s", "$@"; printf \"\n\"'])
  prom = sim.ROM("prom")  # load corresponding prom

  cpu.symtab = sim.SymbolTablePtr.create(user.exe_filename)
  a = sim.UltraAnnoteState();  # use annotations to break at dgemm
  a.pc_watch = symtab.ELF_StringLookup("dgemm");

  itracer_impl = sim.BinaryITracer.create()
  cpu.itracer = sim.ITracer(itracer_impl)

  ...
  sim.reset()  # reset all modules
  sim.run()    # go!
  ```
11 Reflections and Future Work

- Sulima’s OO approach is appropriate, but is no silver bullet!
- 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
- future work:
  - UltraSPARC III version (major changes to memory system)
  - adding _lwp to Solemn, possibly to extent of many threads per CPU
  - checkpointing and event collection infrastrucutre
  - improving NUMA simulation accuracy/speed
    - ‘nearly’ cycle-accurate CPU and interconnect modelling
    - threaded implementation for SMP host
    - full OS boot
  - comparing/calibrating simulation results with real results
  - undoubtedly will turn out challenging that we realize . . .