Efficient Cycle-Accurate Simulation of the UltraSPARC III CPU

Peter Strazdins*, Bill Clarke and Andrew Over,
sim-devel[at]ccnuma.anu.edu.au,
CC-NUMA Project,
Department of Computer Science,
The Australian National University

seminar at The Thirtieth Australasian Computer Science Conference,
31 January 2007

1 Overview

- context: performance evaluation of memory-intensive scientific applications on cc-NUMA multiprocessors
- the Sparc-Sulima UltraSPARC SMP simulator project
- UltraSPARC III Cu instruction execution characteristics
- lightweight CPU simulation by ‘cycle counting’
  - incorporation in the simulator’s Fetch-Decode-Execute loop
  - instruction grouping and register interlocks
  - store buffer modelling
- validation
- performance
- conclusions
Performance Evaluation of Memory-Intensive Applications

- large-scale symmetric multiprocessors (SMP) with cc-NUMA are important HPC platforms
- many scientific applications are limited by memory performance
- e.g. in (quantum) computational chemistry, linear scaling algorithms, as used in applications such as Gaussian:
  - most activity is at user-level; memory intensive
  - irregular memory accesses, limited temporal locality
  - parallelize (via OpenMP) with special emphasis on data placement
  - thread affinity issues also important
  - realistic analysis requires large workloads!
- the CC-NUMA Project is concerned with such analysis
  - use libcpc (CPC library) to obtain useful statistics
  - use simulation for more detailed information (e.g. E-cache miss hot-spots & their causes), or for analysis on larger/variant architectures
3 Sparc-Sulima: an accurate UltraSPARC SMP simulator

- execution-driven simulator with Fetch/Decode/Execute CPU simulator
  - captures both functional simulation and timing simulation
- emulate Solaris system calls at the trap level (Solemn), including LWP traps for thread support
  - permits simulation of unmodified (dynamically linked) threaded binaries
- the CPU is connected to the memory system (caches and backplane) via a ‘bridge’
  - can have a plain (fixed-latency) or fully pipelined Fireplane-style backplane
  - validated to a high degree of accuracy
- simulator speed: slowdowns in range 500–1000 ×
- source code available from Sparc-Sulima home page (Oct 2006)
4 UltraSPARC III Cu Architecture - Overview

(from Alan Charlesworth, *The Sun Fireplane System Interconnect*, Supercomputing’01)
5 UltraSPARC III Cu Instruction Execution Characteristics

- 4-way super-scalar (up to 4 instructions issued per cycle)
- each instruction is dispatched in-order to a functional unit:
  - 2 integer, f.p. *, f.p. +, branch, load/store/special (MS)
  - MS unit for instructions which may block execution of all future instructions for \( L_b \geq 0 \) cycles
  - f.p. loads which hit the Prefetch Cache may be steered to an integer unit
- integer registers (32 available in the current register window);
  32 double precision f.p. registers
  - destination registers locked for a number of cycles:
  - many subtleties and exceptions!  (and ambiguities, omissions . . .)
6 Lightweight CPU Simulation by ‘Cycle Counting’

- traditionally, cycle-accurate CPU simulation has involved explicit modelling of the microarchitecture (main and sub-pipelines, branch prediction etc)
  - typically results in a $10 \times$ or more loss of simulation speed!
- such accuracy (precision) not required if accurate memory system simulation of chief interest
  - the agent injecting events into the memory system (the CPU) must still have reasonable accuracy for situations of interest
- idea: model only as much of the CPU as is needed to ‘count’ the number of clock cycles of an instruction sequence
  - like a human expert predicting the performance of a ‘tight loop’
  - most often, the real machine seems to take longer!
- strategy: when in doubt, be optimistic!
7 Cycle Counting Example

- ‘instruction trace’ from a compiler-optimized matrix-vector multiply code

<table>
<thead>
<tr>
<th>clock</th>
<th>pc</th>
<th>opcode</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>297621</td>
<td>0x19298</td>
<td>ldd</td>
<td>[%g1], %f2</td>
</tr>
<tr>
<td>297621</td>
<td>0x1929c</td>
<td>fmuld</td>
<td>%f4, %f18, %f6</td>
</tr>
<tr>
<td>297621</td>
<td>0x192a0</td>
<td>faddd</td>
<td>%f14, %f8, %f14</td>
</tr>
<tr>
<td>297621</td>
<td>0x192a4</td>
<td>add</td>
<td>%g1, %o4, %g1</td>
</tr>
<tr>
<td>297622</td>
<td>0x192a8</td>
<td>ldd</td>
<td>[%g2+%i1], %f18</td>
</tr>
<tr>
<td>297622</td>
<td>0x192ac</td>
<td>fmuld</td>
<td>%f4, %f20, %f8</td>
</tr>
<tr>
<td>297622</td>
<td>0x192b0</td>
<td>faddd</td>
<td>%f16, %f10, %f16</td>
</tr>
<tr>
<td>297623</td>
<td>0x192b4</td>
<td>ldd</td>
<td>[%g2+%i3], %f20</td>
</tr>
<tr>
<td>297623</td>
<td>0x192b8</td>
<td>fmuld</td>
<td>%f4, %f24, %f10</td>
</tr>
<tr>
<td>297624</td>
<td>0x192bc</td>
<td>faddd</td>
<td>%f2, %f12, %f2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># %f2 locked till t=297621+3</td>
</tr>
<tr>
<td>297625</td>
<td>0x192c0</td>
<td>ldd</td>
<td>[%g2+%i0], %f24</td>
</tr>
<tr>
<td>297625</td>
<td>0x192c4</td>
<td>fmuld</td>
<td>%f4, %f26, %f12</td>
</tr>
<tr>
<td>297625</td>
<td>0x192c8</td>
<td>faddd</td>
<td>%f4, %f6, %f4</td>
</tr>
<tr>
<td>297626</td>
<td>0x192cc</td>
<td>ldd</td>
<td>[%i2],%f26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
8 Design: installable module

- incorporation into simulator Fetch-Decode-Execute loop
  
  $\text{CI} = \text{FetchInstr}(pc); // \text{pc} = \text{value of Program Counter}$
  
  $\text{di} = \text{DecodeInstr}(\text{CI}); // \text{di} : \text{decoded instrn. structure}$
  
  if (cycleCount)  // cycle counter module installed
    clock += cycleCount->InstrCountCycles(
        di.opcode, &di.operands, clock);
  else
    clock += 1;

  if (itracer)  // instrn. trace module installed
    itracer->TraceInstr(clock, pc, CI, di);
  clock += ExecuteInstr(di.opcode, &di.operands);

- CPU's clock is simulated by the variable $\text{clock}$

- $\text{InstrCountCycles()}$ returns latency $L \geq 0$ where:
  
  - if CI is first in a 'group', $L = L_b + 1$ ($L_b$ was from previous group)
  - o.w. $L > 0$ means $L$ cycles required to 'unlock' any source registers
9 Implementation: Main Features

- information required per instruction:
  - functional unit(s) required, blocking latency \((L_b)\), group breaking characteristics; register operands: types, # cycles destn. locked
- two-level table: arrange \((\approx 250)\) instructions into \((\approx 50)\) categories to reduce space and facilitate easy change
- maintain bitmask of all functional units used so far in the group
  - form a new group if required functional unit(s) are already in bitmask, or if group breaking characteristics say so
  - typically requires only 11 load/stores, 4 compares and 10 integer ops.
- register locking implemented by arrays of (64-bit!) timestamps
  - indicates time \(t'\) when the source register becomes available
  - if current time \(t < t'\), add a latency of \(L = t' - t\)
  - worst-case overhead involves \(\approx 10\) loads, 10 compares and 2 stores
  - note: may be up to 12 f.p. registers may be locked at any time!
10 Design: Store Buffer and Event Counting

- grouping rules allow 1 store per cycle, but no (L1-cache resident) benchmark sustained more than 1 store per 3 cycles!
- need to (efficiently) model an 8-entry store buffer:
  - maintain $t_S$, time of last store emitted, and $l_S$, # of pending stores
  - if $t_S < t$, drain $n_S = \left\lfloor \frac{t-t_S}{3} \right\rfloor$ stores from buffer ($l_S-=n_S$), and add $3n_S$ to $t_S$
  - if buffer is full, update $t_S$ to next emission time ($t_S+3$), and add a latency of $L = t_S - t$
- can be disabled in order to use Sparc-Sulima’s detailed store buffer and write-cache model
- f.p. load steering modelling added some complexity
- need also count CPU-specific hardware events, notably the number of cycles lost due to integer and f.p. register locking and a full store buffer
11 Validation – Methodology

- verifying simulator accuracy is critical for useful performance analysis
  - essential in any kind of performance modelling! (but often not done . . .)
  - validation is an ongoing issue in field of simulation
- 1st stage: manually examine instruction traces and check timestamps are consistent with (documented) rules
- 2nd stage: compare (mini-)benchmark performance with real hardware
  - compare real vs simulated hardware event counts, and/or instruction execution histograms to gain insight into any discrepancies
  - construct microbenchmarks to verify the suspected cause
- several ambiguities/errors in the ‘bible’ took a long time to discover:
  - “no instruction can bypass the result to another in the same group”
  - when a register is locked ”the offending instruction and all younger instructions are recirculated”
  - $L_b$ is the number of cycles “the dispatch unit waits before issuing another group” (Jan’07!)
12 Validation: Mini-benchmarks

- speed in MOPs for vector (length 4000) and in MFLOPs for vector-matrix ($A \times 64 \times 64$) computations (CC = cycle counter, SB = store buffer, SB'/WC = detailed store buffer and write-cache)

<table>
<thead>
<tr>
<th>computation</th>
<th>simulated</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no CC</td>
<td>CC-SB</td>
</tr>
<tr>
<td>$y \leftarrow x$ ($8 \times 1$ dd; $8 \times$ std)</td>
<td>309</td>
<td>445</td>
</tr>
<tr>
<td>$y \leftarrow x$ ($1 \times 1$ dd; $1 \times$ std)</td>
<td>150</td>
<td>299</td>
</tr>
<tr>
<td>$y \leftarrow 2x$</td>
<td>123</td>
<td>216</td>
</tr>
<tr>
<td>$y \leftarrow y + x$</td>
<td>170</td>
<td>288</td>
</tr>
<tr>
<td>$y \leftarrow x$</td>
<td>169</td>
<td>221</td>
</tr>
<tr>
<td>$a \leftarrow \sum_{i=0}^{n-1}</td>
<td>x_i</td>
<td>$</td>
</tr>
<tr>
<td>$x \leftarrow 2x$</td>
<td>210</td>
<td>446</td>
</tr>
<tr>
<td>$x \leftarrow 0$</td>
<td>497</td>
<td>866</td>
</tr>
<tr>
<td>$y \leftarrow Ax$</td>
<td>476</td>
<td>1376</td>
</tr>
<tr>
<td>$y \leftarrow A^{-1}x$</td>
<td>442</td>
<td>1141</td>
</tr>
</tbody>
</table>
13 Application-level Validation and Performance

- NAS Benchmark (S-class) validation, with detailed memory simulation:

<table>
<thead>
<tr>
<th>workload</th>
<th>bt</th>
<th>cg</th>
<th>ft</th>
<th>is</th>
<th>lu</th>
<th>lu-hp</th>
<th>mg</th>
<th>sp</th>
<th>ua</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio host:sim.</td>
<td>1.03</td>
<td>1.01</td>
<td>0.92</td>
<td>0.99</td>
<td>0.98</td>
<td>1.01</td>
<td>0.93</td>
<td>0.98</td>
<td>0.68</td>
</tr>
</tbody>
</table>

- generally better, due to lower overall intensity of store operations

- slowdowns, indicating overall speed of simulator and added overheads (CC = cycle counter, RIL = register interlocks, LS = f.p. load steering)

<table>
<thead>
<tr>
<th>computation</th>
<th>host: no CC</th>
<th>no CC: CC-RIL-LS</th>
<th>no CC: CC-LS</th>
<th>no CC: CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \leftarrow x$</td>
<td>966</td>
<td>1.07</td>
<td>1.21</td>
<td>1.35</td>
</tr>
<tr>
<td>$y = y + x$</td>
<td>1435</td>
<td>1.03</td>
<td>1.22</td>
<td>1.42</td>
</tr>
<tr>
<td>$a \leftarrow \sum_{i=0}^{n-1}</td>
<td>x_i</td>
<td>$</td>
<td>590</td>
<td>1.19</td>
</tr>
<tr>
<td>$y \leftarrow Ax$</td>
<td>1757</td>
<td>1.17</td>
<td>1.27</td>
<td>1.41</td>
</tr>
</tbody>
</table>

- groupings adds 10–20% overhead, register locking adds 10–20% more
f.p. load steering adds a variable overhead, total overhead ≈ 40%
Conclusions and Future Work

- relatively simple approach to modelling CPU aspects of an UltraSPARC
  - its modular design permits activation only when needed
  - most architecture-specific information is recorded in tables
- added modest simulation overhead while achieving high accuracy in most situations of interest
  (all codes available!)
- approach can be adapted to other post-RISC architectures
  - modelling out-of-order execution is somewhat problematic
  - multi-core era may see a return of simpler CPU microarchitectures
- validation against real H/W resulted in several important ‘corrections’
  - methodology based on hardware event counters etc. important
  - main discrepancies on store-intensive benchmarks
    - real machine behaviour is between simple store buffer and sophisticated memory system models
  - the real limiting factor is lack of complete and accurate documentation
15 Questions?

Acknowledgements: this work was sponsored by the Australian Research Council, Sun Microsystems Inc. and Gaussian Inc under the ARC Linkage Grant LP0347178.