Continuous Profiling:
(It’s 10:43; Do You Know Where Your Cycles Are?)

Jennifer Anderson  Lance Berc  Jeff Dean
Sanjay Ghemawat  Monika Henzinger  Shun-Tak Leung
Dick Sites  Mitch Lichtenberg  Mark Vandevoorde
Carl Waldspurger  Bill Weihl

What’s the problem?

- Performance
  - 15 of 16 issue slots wasted in some applications, at least 1 of 2 in most
- Complexity
  - superscalar, out-of-order, SMP, SMT, clusters, ...

- How pinpoint performance problems and causes?
- How fix them?
Our solution

• **DIGITAL Continuous Profiling Infrastructure**
  – Transparent
  – Complete
  – Efficient
  – Produces accurate fine-grained information

  – Designed for continuous use on production systems
  – Intended for programmers and optimization tools

Related Work

• Simulation (e.g., SimOS)
  – slow
• pixie *et al.*
  – single app
  – modifies executable
• Samplers (prof, Morph, Vtune, SGI Speedshop)
  – some tied to existing interrupts (timers)
  – overhead often too high
• None give accurate fine-grained information and low overhead
System Overview: Acquiring and analyzing sample data

Analysis tools:
- system-, load-file-, procedure-, and instruction-level profiles
- Load files
- In progress: optimization tools

Load-file-level analysis example

Total samples for event type cycles = 6095201, imiss = 1117002

The counts given below are the number of samples for each listed event type.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Sample Count</th>
<th>% of Total</th>
<th>Cumulative %</th>
<th>Load File</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>2064143</td>
<td>33.87%</td>
<td>33.87%</td>
<td>/usr/lib/X11/lib_dec_fvb_ev5.so</td>
</tr>
<tr>
<td>imiss</td>
<td>517464</td>
<td>8.49%</td>
<td>42.35%</td>
<td>/usr/lib/X11/libos.so</td>
</tr>
<tr>
<td>procedure</td>
<td>305072</td>
<td>5.01%</td>
<td>47.36%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
<tr>
<td>load file</td>
<td>271158</td>
<td>4.45%</td>
<td>51.81%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
<tr>
<td>load file</td>
<td>245450</td>
<td>4.03%</td>
<td>55.84%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
<tr>
<td>load file</td>
<td>209835</td>
<td>3.44%</td>
<td>59.28%</td>
<td>/usr/lib/X11/libdix.so</td>
</tr>
<tr>
<td>load file</td>
<td>186413</td>
<td>3.06%</td>
<td>62.34%</td>
<td>/usr/lib/X11/lib_dec_fvb_ev5.so</td>
</tr>
<tr>
<td>load file</td>
<td>170723</td>
<td>2.80%</td>
<td>65.14%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
<tr>
<td>load file</td>
<td>161326</td>
<td>2.65%</td>
<td>67.78%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
<tr>
<td>load file</td>
<td>133768</td>
<td>2.19%</td>
<td>69.98%</td>
<td>/usr/lib/X11/libmi.so</td>
</tr>
</tbody>
</table>
Instruction-level analysis example

*** Best-case 8/13 = 0.62CPI
*** Actual 140/13 = 10.77CPI

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Samples</th>
<th>CPI</th>
<th>Culprit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(p = branch mispredict)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9810</td>
<td>ldq t4, 0(t1)</td>
<td>3126</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>9814</td>
<td>addq t0, 0x4, t0</td>
<td>1636</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>981c</td>
<td>ldq a0, 24(t1)</td>
<td>390</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>9820</td>
<td>ldq t1, 32(t1)</td>
<td>1482</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>9828</td>
<td>stq t4, 0(t2)</td>
<td>27766</td>
<td>18.0</td>
<td>9810</td>
</tr>
<tr>
<td>9830</td>
<td>stq t5, 8(t2)</td>
<td>1493</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>983c</td>
<td>stq a0, 24(t2)</td>
<td>1548</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>9840</td>
<td>lda t2, 32(t2)</td>
<td>0</td>
<td>(dual issue)</td>
<td></td>
</tr>
</tbody>
</table>

s = (s = slotting hazard)
dwD = (d = D-cache miss)
dwD = (w = write-buffer overflow)
dwD = (w = write-buffer overflow)
dwD = (w = write-buffer overflow)

C source code for assembly code above (unrolled 4 times):

```c
for (i = 0; i < n; i++)
    c[i] = a[i];
```

Procedure-level summary example

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-cache (not ITB)</td>
<td>0.0% to 0.3%</td>
</tr>
<tr>
<td>ITB/I-cache miss</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>D-cache miss</td>
<td>27.9% to 27.9%</td>
</tr>
<tr>
<td>DTB miss</td>
<td>9.2% to 18.3%</td>
</tr>
<tr>
<td>Write buffer</td>
<td>0.0% to 6.3%</td>
</tr>
<tr>
<td>Synchronization</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>0.0% to 2.6%</td>
</tr>
<tr>
<td>IMUL busy</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>FDIV busy</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>Unexplained stall</td>
<td>2.3% to 2.3%</td>
</tr>
<tr>
<td>Unexplained gain</td>
<td>-4.3% to -4.3%</td>
</tr>
</tbody>
</table>

Subtotal dynamic 44.1%

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slotting</td>
<td>1.8%</td>
</tr>
<tr>
<td>Ra dependency</td>
<td>2.0%</td>
</tr>
<tr>
<td>Rb dependency</td>
<td>1.0%</td>
</tr>
<tr>
<td>Rc dependency</td>
<td>0.0%</td>
</tr>
<tr>
<td>FU dependency</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Subtotal static 4.8%

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total stall</td>
<td>48.9%</td>
</tr>
<tr>
<td>Execution</td>
<td>51.2%</td>
</tr>
<tr>
<td>Net sampling error</td>
<td>-0.1%</td>
</tr>
</tbody>
</table>

Total tallied 100.0%

(35171, 93.1% of all samples)
Generating samples in hardware

- 2 or 3 hardware event counters
- Overflow → high-priority interrupt
- Problem: inaccurate pc’s
  - 6-cycle delay
  - handler sees pc of oldest instruction in issue queue
- So… can’t use counters to attribute most events to instructions
  - (NB: all existing event counters have this problem)

Problems in acquiring samples in OS

- Interrupt rate is very high
  - e.g., one sample every 62K cycles at 400 MHz:
    ~6,100 samples/sec
- Primary issue: performance!
  - Cache misses are expensive (e.g., ~100 cycles/miss to memory)
  - If we took 10 cache misses at 100 cycles each, we’d incur ~1.5% overhead for the interrupt handler alone -- too much.
Making OS software efficient

• Aggregate samples in hash table
  – (pid, pc, event) → count
• Minimize cache misses and maximize benefit from each
  – 4-way associative tables
  – careful packing of data structures
• Eliminate expensive synchronization operations
  – interprocessor interrupts for synchronization with handler

Storing samples in a database

• User-mode daemon: \textit{dcpid}
  – extracts raw samples from driver
  – associates samples with load-files
  – updates disk-based profiles for load-files
• Finding load-files from <PID, PC>
  – \textit{dcpiloader} replaces default dynamic loader
  – exec hook for statically linked load-files
• Profiles
  – text header + compact binary samples
  – organized by \textit{epoch} and \textit{platform}
  – can be shared among machines
Performance of data collection

- Time
  - 1-3% total overhead for most workloads
  - less than variation from run to run
- Space
  - 512 KB kernel memory
  - 2-10 MB resident for daemon
  - 12 MB disk after one week of profiling on heavily used timeshared 4-processor server
- Non-intrusive enough to be run for many hours on massive database machines

Kinds of analysis provided

- Aggregate info:
  - breakdown by load-file or function
  - compare raw profiles by load-file or function
- Detailed info:
  - augmented control flow graph for a procedure
    - execution frequencies, CPI, reason(s) for stalls
    - source code (if available)
  - annotate source or asm w/ results of analysis
  - highlight differences in multiple profiles
Converting cycle samples to CPI and frequency

- Cycle samples are proportional to total time at head of issue queue (where most interesting stalls occur)
- Frequency indicates frequent paths
- CPI indicates stalls

Flow Graph
- DC
- P
- I
- C
- L

Samples
- D
- C
- P
- I
- C
- A
- L

Reasons for stalls
- Frequency
- Cycles per instruction

Estimating frequency from samples

- Problem
  - given cycle samples, compute frequency and CPI
- Approach
  - Let \( F = \) Frequency / Sampling Period
  - \( E(\text{Cycle Samples}) = F \times \text{CPI} \)
  - So ... \( F = E(\text{Cycle Samples}) / \text{CPI} \)
- Idea
  - If no dynamic stall, then know CPI, so can estimate \( F \)
  - Better accuracy: average sample counts from several instructions
Finding instructions w/o dynamic stalls

- Consider a group of instructions with the same frequency (e.g., basic block)
- Assume some instructions execute without dynamic stalls
- Use several heuristics to identify them; then average their sample counts

- Key insight:
  - instructions without stalls have smaller sample counts

Instructions w/o dynamic stalls (cont)

- But ... some small counts are anomalous (e.g., 981c)
- Avoid anomalies: Identify issue points (IP)
- Choose some IPs to average (A)
- Average obtained: 1527 (actual value: 1575)
- Does badly when:
  - few issue points
  - all issue points stall
Improving frequency estimates

- Average over more instructions
  - normalize sample count by static minimum number of cycles
  - compute “frequency equivalence” classes
- Local propagation using flow equations
  - edge frequencies too
- Global propagation using flow equations
  - complete consistent estimates
- Label estimates with confidence levels

How accurate are frequency estimates?

- Compare frequency estimates for blocks to measured values obtained with pixie-like tool

- Similar results for edge frequencies
Identifying reasons (culprits) for stalls

- Explain static stalls by scheduling instructions in each basic block optimistically using a detailed pipeline model for the processor
- Explain dynamic stalls by eliminating suspects
  - The usual suspects:
    - I-cache or ITB miss
    - D-cache or DTB miss
    - Branch misprediction
    - Etc.
  - Eliminate suspects heuristically, and list the remaining possibilities as culprits

Ruling out I-cache misses as culprits

- Is the previously executed instruction in another cache line?

  - Depends
    - Yes
    - No

  - How many imiss samples occurred at this instruction? What is the maximum impact?
Ruling out D-cache misses as culprits

- Is the previous occurrence of an operand register the destination of a load instruction?

```
ldq   t0, 0(s1)  addq   t3, t4, t0
```

- Search backward across basic block boundaries
- Prune by block and arc execution frequencies

```
subq   t0, t1, t2  subq   t0, t1, t2
```

How accurate is culprit analysis?

- Compare with measured event counts for procedures
- E.g., imiss data:

```
Correlation ~ .9
```
Future work

- Optimization
  - code layout and scheduling
  - data structure layout
  - prefetching, inlining, hot-cold optimization
- Enhanced profiling
  - edge samples
  - load/store/jump addresses
- Instruction-level profiling for other processors
  - out-of-order execution
  - speculative execution
  - ...

Summary

- Low-overhead transparent profiling
- Profiles complete system continuously
- Accurate fine-grained analysis
  - CPI
  - execution frequencies for blocks and edges
  - reasons for stalls
- Stay tuned...

http://www.research.digital.com/SRC/dcpi