B²Sim: A Fast Micro-Architecture Simulator Based on Basic Block Characterization

October, 23, 2006

Wonbok Lee, Kimish Patel, Massoud Pedram

Dept. of Electrical Engineering
University of Southern California
Contents

- Introduction
- Prior Work
- A key Observation
- Basic Block Characterization Based Micro-architectural Simulator (B²Sim)
  - Cycle Characterization
    - On-Chip CPI Characterization
    - Off-Chip CPI Computation
  - B²Sim Framework
- Simulation Environment and Benchmarks
- Experimental Results
- Conclusions and Future Work
Introduction

- Micro-architectural simulators:
  - A software infrastructure that mimics micro-processor behaviors

- Where to use:
  - Performance projection during the pre-silicon phase of the chip design (HW)
  - Performance evaluation / architectural exploration (SW)
  - Hardware-software co-design/verification (HW + SW)

- Strong points in micro-architectural simulators:
  - Validate hardware before/without the actual implementation (cost)
  - Easy to explore micro-architectural design space (time)

- Weak points in micro-architectural simulators:
  - Slower than the native program execution (order of 4X ~ 5X)
  - Cannot properly handle system call & operating system related parts

- Why micro-architectural simulators are slow?
  - Pipeline simulation in software
  - Pipeline stages which are implemented with inter-stage queue management

- Can we make it faster?
Prior Work for Reducing Simulation Time

- Simulation with Reduced Benchmark

- (Periodic) Sampled Simulation

- Program Behavior (Phase) Based Sampled Simulation

- Statistical Simulation
  - HLS++ (Hybrid Laboratory Simulator) – M. Oskin et al., ISCA, 2000.

- Accelerate the Warm-up

- Accelerate the Fast-forwarding

- Truncated Simulation

- Instruction Set compiled Simulator
A Key Observation

- Once a program is compiled, program code structure does not change
  - The relative dependencies between instructions do not change
  - On-chip behavior of instructions does not change either

- Characterize the behavior of a basic block (BB) in terms of cycles:
  - On-chip cycles remain the same, since dependencies between instructions do not change
  - However, Off-chip cycles due to memory access changes due to the dynamic behavior of caches and TLBs

- On-Chip behavior (CPI) of a BB:
  - ALU operations
  - Register-Register transfer
  - Dependency, stall, etc.

- Off-Chip behavior (CPI) of a BB:
  - I/D-TLB accesses/misses,
  - I/D-Cache accesses/misses,
  - Memory accesses, etc.
Key Observation (Cont’d)

- **On-Chip CPI of a BB is consistent**
  - On-Chip CPI comes from \((\text{On-chip cycles} = \text{Total cycles} - \text{Off-Chip cycles})\)
  - Number of instruction in a BB is always fixed
  - Table shows some of the frequently visited BBs in their first 100 visits

- **How to use this characteristic?**
  - Once a BB is characterized in terms of its On-Chip cycles, we do not need to simulate the BB in the detailed pipeline simulator to get the On-Chip CPI

<table>
<thead>
<tr>
<th>Program</th>
<th>BB Size (inst #)</th>
<th>No. of Visit</th>
<th>Average On-chip CPI</th>
<th>On-chip CPI Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip - graphic</td>
<td>21</td>
<td>41.9M</td>
<td>1.333</td>
<td>2.0e-4</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>12.0M</td>
<td>1.356</td>
<td>5.3E-6</td>
</tr>
<tr>
<td>gcc - expr</td>
<td>12</td>
<td>77.1M</td>
<td>1.667</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>27.2M</td>
<td>1.215</td>
<td>2.9e-5</td>
</tr>
<tr>
<td>bzip - program</td>
<td>44</td>
<td>35.5M</td>
<td>1.408</td>
<td>2.3e-4</td>
</tr>
<tr>
<td></td>
<td>69</td>
<td>20.8M</td>
<td>1.231</td>
<td>1.9e-5</td>
</tr>
<tr>
<td>mcf - inp.in</td>
<td>16</td>
<td>93.2M</td>
<td>1.624</td>
<td>3.9e-5</td>
</tr>
<tr>
<td></td>
<td>137</td>
<td>16.5M</td>
<td>1.400</td>
<td>6.4e-5</td>
</tr>
<tr>
<td>vortex - lendian1</td>
<td>20</td>
<td>44.6M</td>
<td>1.596</td>
<td>7.9e-4</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>26.3M</td>
<td>1.581</td>
<td>1.3e-3</td>
</tr>
</tbody>
</table>

- **How about Off-Chip CPI of a BB?**
  - Cache simulator is needed all the time
    - Cache miss counts
    - Memory latency

- **Key Idea of B^2Sim**
  - If a new BB is executed in the detailed pipeline simulator, characterize its On-Chip cycles
  - For subsequent visits of the BB, avoid detailed pipeline simulation
BB’s On-Chip CPI Characterization

- Is it really true that \((\text{On-Chip cycles} = \text{Total cycles} – \text{Off-Chip cycles})\) ?
  - Given l-cache miss latency of 32 cycles
  - Assume data dependency between 4\(^{th}\) (add r1, r3, r5) & 5\(^{th}\) (sub r2, r3, r1) instructions

- Consider that an l-cache miss occurs in the 4\(^{th}\) instruction
  - On-Chip cycle of BB \#k is 39 - 32 = 7 (shown on the left)

- Consider that an l-cache miss does not occur in the 4\(^{th}\) instruction
  - On-Chip cycle of BB \#k is 8 (shown on the right)
BB’s On-Chip CPI Characterization (Cont’d)

- What happens? Some I-cache misses may hide the data dependencies
  - By the time 5\textsuperscript{th} instruction is fetched (in a miss), 4\textsuperscript{th} instruction is finished, which depends on 5\textsuperscript{th} instruction
  - With an I-cache miss, no wait cycle for 5\textsuperscript{th} instruction once it is fetched
  - Without an I-cache miss, dependency introduces a wait cycle to 5\textsuperscript{th} instruction

- Solutions to characterize On-Chip cycle?
  - Do not characterize On-Chip cycle of a BB till there is no I-/D-cache misses
  - Allow a trial of threshold value to the characterization of On-Chip cycle of a BB

With an instruction cache miss
On-chip cycle = 39 − 32 = 7

Without an instruction cache miss
On-chip cycle = 8
BB’s Off-Chip CPI Calculation

Is Off-Chip cycle always equal to the memory latency?
- Given D-cache miss latency of 32 cycles
- Assume data dependency between 3rd (ld r1 0(r5)) and 5th (sub r2, r3, r1) instructions

Consider that a D-cache miss occurs in the 3rd instruction
- D-Cache miss latency is 32 - 1 = 31 (shown in the left)

Consider that a D-cache miss does not occur in the 3rd instruction
- D-Cache miss latency is 32 (shown in the right)

With a data cache miss
Actual D-cache latency = 32 - 1 = 31

Without a data cache miss
BB’s Off-Chip CPI Calculation (Cont’d)

- **What happens?**
  - Pipeline does not stall immediately on D-cache miss
  - Instead, it stalls after one more (4th) instruction is executed

- **The compiler puts independent instructions btw. dependent instructions**
  - Some memory latency might be hidden since independent instructions are scheduled in between
  - Hidden latency is hard to characterize and we do not know the exact distance

- **Solutions?**
  - For a BB that has D-cache miss, use the average distance value

With a data cache miss

Actual D-cache latency = 32 - 1 = 31

Without a data cache miss
Simulator Framework

- Basic Block characterization (in terms of On-Chip & Off-Chip CPIs) based Simulator ($B^2$Sim)
- For each of BBs,
  - Execute sim-outorder for/to the first run/visit
  - Execute sim-cache for its subsequent run
  - BB’s total cycle: On-Chip cycle + Off-Chip cycle

- Build an On-Chip CPI table that stores each BB’s On-Chip CPI
  - On-Chip CPI table has;
    - # of visit
    - # of cycle
    - # of instruction in BB

- Program’s CPI can be derived by the accumulation of On-Chip & Off-Chip CPIs of BBs
Simulation & Benchmark Programs

- Benchmark programs in the simulation
  - SPEC2000 INT
    - Reference/Train input files
    - MediaBench with custom input files
- Platforms for the simulation: 3 Linux machines
  - Athlon 2500+, Pentium 4 2.5GHz, Pentium 4 1.8GHz
- Table shows the architectural parameters used in B²Sim

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory Latency</td>
<td>32 cycles</td>
</tr>
<tr>
<td>L1 I/D Cache</td>
<td>32KB 32-way 32Byte block, 1 cycle hit latency</td>
</tr>
<tr>
<td>L2 I/D Cache</td>
<td>None</td>
</tr>
<tr>
<td>I/D-TLB</td>
<td>4-way 1024 entries, 32 cycles miss latency</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Bimodal 128 Table</td>
</tr>
<tr>
<td>Functional Units</td>
<td>1 Integer ALU, 1 Integer MULT/DIV, 1 FP ALU, 1 FP MULT/DIV</td>
</tr>
<tr>
<td>RUU/LSQ size</td>
<td>8/8</td>
</tr>
<tr>
<td>Instruction Fetch Queue</td>
<td>8</td>
</tr>
<tr>
<td>In order Issue</td>
<td>True</td>
</tr>
<tr>
<td>Wrong Path Execution</td>
<td>True</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Native exec. time</th>
<th>Number of instruction</th>
<th>Simulation Time (minutes)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>sim-cache</td>
<td>sim-outorder</td>
</tr>
<tr>
<td>gzip-log</td>
<td>2.9</td>
<td>4.4</td>
<td>28</td>
<td>160</td>
</tr>
<tr>
<td>gzip-source</td>
<td>5.3</td>
<td>8.8</td>
<td>55</td>
<td>334</td>
</tr>
<tr>
<td>gzip-random</td>
<td>5.8</td>
<td>7.8</td>
<td>51</td>
<td>295</td>
</tr>
<tr>
<td>gzip-graphic</td>
<td>6.3</td>
<td>9.4</td>
<td>60</td>
<td>365</td>
</tr>
<tr>
<td>gzip-program</td>
<td>8.9</td>
<td>16.8</td>
<td>105</td>
<td>646</td>
</tr>
<tr>
<td>gcc-expr</td>
<td>11.9</td>
<td>15.1</td>
<td>62</td>
<td>417</td>
</tr>
<tr>
<td>gcc-integrate</td>
<td>14.2</td>
<td>16.5</td>
<td>66</td>
<td>444</td>
</tr>
<tr>
<td>gcc-166</td>
<td>62.3</td>
<td>57.0</td>
<td>233</td>
<td>1505</td>
</tr>
<tr>
<td>bzip-graphic</td>
<td>15.7</td>
<td>24.5</td>
<td>129</td>
<td>696</td>
</tr>
<tr>
<td>bzip-program</td>
<td>12.4</td>
<td>20.1</td>
<td>107</td>
<td>590</td>
</tr>
<tr>
<td>bzip-source</td>
<td>10.1</td>
<td>16.7</td>
<td>90</td>
<td>489</td>
</tr>
<tr>
<td>mcf-inp.in</td>
<td>55.8</td>
<td>20.1</td>
<td>116</td>
<td>739</td>
</tr>
<tr>
<td>vortex-lendian1</td>
<td>16.6</td>
<td>13.0</td>
<td>60</td>
<td>404</td>
</tr>
<tr>
<td>djpeg-custom</td>
<td>1.7</td>
<td>2.9</td>
<td>17</td>
<td>94</td>
</tr>
<tr>
<td>cjpeg-custom</td>
<td>3.2</td>
<td>5.3</td>
<td>32</td>
<td>187</td>
</tr>
</tbody>
</table>
Experimental Results (Cont’d)

- Overall average CPI error on B²Sim: 0.57%
- Feasible reasons of CPI error:
  - Inter-BB data dependency
  - Sporadic I-cache misses which are not captured in the middle of On-Chip CPI characterization
- Overall average speedup of B²Sim: 3.31X
- Speedup vs. IPB (Instruction Per Branch)
  - Size and occurrence of each BB determines the speedup
  - IPB has the combined nature of both

![CPI error and Speedup vs. IPB chart]

<table>
<thead>
<tr>
<th>CPI error (%)</th>
<th>gzip-l</th>
<th>gzip-s</th>
<th>gzip-r</th>
<th>gzip-p</th>
<th>gcc-e</th>
<th>gcc-i</th>
<th>gcc-166</th>
<th>gzip-q</th>
<th>bzip-q</th>
<th>bzip-p</th>
<th>bzip-s</th>
<th>mcf</th>
<th>vortex</th>
<th>dpjpeg</th>
<th>cpjpeg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Bar chart comparing CPI error and speedup]
Conclusion & Future Work

- **B²Sim** differs from previous simulation acceleration approaches in:
  - Minimally use detailed pipelined simulator if it is really needed
  - Do not need off-line profiling, fast-forwarding (FF) and warm-up
  - Deterministic in that it generates consistent performance over multiple runs
  - High level of granularity to capture the cycle-accurate information

- **B²Sim** analyzes and utilizes BB level program behavior and information
  - On-Chip CPI of each BB is consistent
  - Once a BB is characterized with its On-Chip CPI in the pipelined simulator:
    - Functional simulator is used for those BBs afterwards
  - Off-Chip CPI of each BB temporally changes
    - I/D-cache level simulator is used for BBs all the time

- Experimental results and analysis:
  - Speed comparison (sim-cache): (sim-outorder): (B²Sim) = 1:6:2
  - Ideal speedup of B²Sim can reach to the speed of sim-cache. However:
    - Every BB need to be executed in the pipelined simulator at least once
    - Branch prediction and wrong/speculative execution are need over BBs

- Work to speedup B²Sim is ongoing!
Backup: Results in Enhanced B²Sim

- Speed-up comparison (over the original sim-outorder)
Backup: Results in Enhanced B²Sim (Cont’d)

- CPI Error comparison (over the original sim-outorder)
Backup: Some details in Implementation

Anatomy of B²Sim (Source Level)

while (inst. for the speculative & wrong path execution) {
  If (branch miss prediction occurs) {
    take branch miss prediction penalty cycle into account in switching
    // B²Sim does know on-chip/off-chip CPI of BB. However,
    // B²Sim does not know branch miss prediction penalty;
  }
}

If (BB is visited)
  switch to modified sim-cache;
else {
  while (fetch inst. until fetch-> dispatch Q fills) {
    access I-$ & I-TLB to fetch inst.;
    update I-$/I-TLB misses of BB;
  }
}
Update branch prediction();

for (number of inst. in a BB) {
  If (last inst. in BB)
    If (BB was visited)
      Modified sim-cache
  If (last inst. in BB)
    B²Sim
  If (last inst. in BB)
    sim-outorder
    If (last inst. in BB)
      IF ID EX WB CT
      If (last inst. in BB)
        if (last inst. in BB)
          access D-$ & D-TLB to update status;
      }
    if (last inst. in BB)
      access I-$ & I-TLB to update status;
  }
Update cycles in a BB;
Update instruction count in a BB;

If (inst. from miss predicted branch) {
  take branch miss prediction penalty cycle into account;
  // Compared to ID stage, prediction penalty cycle differs
  // Simply, one cycle difference;
}

// Basically, no new operations on this stage
// Instructions which arrive at this stage are supposed to be executed;

// Basically, no new operations on this stage
// Instructions arrive at this stage are supposed to be committed;

// How to measure pipeline stall count?
// On each cycle level, if none of the pipeline stages
// or functions are called, we think it a pipeline stall cycle

// In Simplescalar
// ID = Dispatch, EX = Issue
SMARTS (Sampling Micro-ARchiTectural Simulator)

- Theory of sampling:
  - Select minimal but representative samples that catch the whole program behavior by executing a periodical sampling
  - Simply, combine sim-outorder, sim-cache and sim-bpred
  - Periodically switches among three modes of operations:
    - Sampling mode, i.e., full detailed simulation and gather data
      - All micro-architectural details are accounted for
    - Detailed warm-up mode
      - Detailed simulation mode but do not gather data
    - Functional warm-up mode
      - Only cache and branch predictor are updated
Some details about the sampling

- **N**: Benchmark length (No. of instruction), e.g., N=10 Million
- **U**: Sampling unit size (No. of instruction), e.g., 1000
- **W**: Detailed warm-up size (No. of instruction), e.g., 2000
- **k**: Fixed sampling interval (No. of instruction), e.g., k = 1000
- **n = N/k**: Sampling is executed in this many time, e.g., n=10,000

How to determine **U**? (What amount do we sample at a time?)

- See CPI coefficient of variance w.r.t. increasing U size
- 1000 < U is enough to saturate the variance (see the backup slide)

How to determine **n/k**? (How often we sample in overall execution?)

- n > 10,000 yields 99.7% of confidence interval which is 4% CPI err
Backup: SMARTS (Cont’d)

- **Performance of SMARTS**
  - Speedup: 35X ~ 60X (over sim-outorder) but 1X-1.5X (over sim-fast)
    - N=10,000, U=1,000, W=2,000
  - Average CPI error: 0.64%

- **Observations in SMARTS:**
  - Combine wattch makes sim-outorder to be slow
    - Reference simulation time: (outorder) : (cache) : (fast) is 18:6:1
  - Aggressive code optimization
    - Simulation time acceleration by data structure change?

<table>
<thead>
<tr>
<th>Runtime</th>
<th>parser</th>
<th>migrid</th>
<th>twolf</th>
<th>mesa</th>
<th>ammp</th>
<th>gap</th>
<th>swim</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-outorder</td>
<td>541</td>
<td>414</td>
<td>343</td>
<td>279</td>
<td>323</td>
<td>266</td>
<td>223</td>
<td>98</td>
</tr>
<tr>
<td>sim-fast</td>
<td>9.2</td>
<td>7.0</td>
<td>5.8</td>
<td>4.7</td>
<td>5.5</td>
<td>4.5</td>
<td>3.8</td>
<td>1.7</td>
</tr>
<tr>
<td>SMARTS</td>
<td>15.8</td>
<td>12.1</td>
<td>10.0</td>
<td>8.1</td>
<td>9.6</td>
<td>7.8</td>
<td>6.5</td>
<td>2.9</td>
</tr>
</tbody>
</table>
Backup: SimPoint

- SimPoint 3.0
  - Program behavior (phase) based sampling simulation

- How and why it works?
  - Program behaviors change over time. However, they are not random but repeat themselves in some cyclic patterns
  - Sample some periods/points that reflect the whole program behavior

- Problem
  - Phase characterization in a program may not be easy
Backup: SimPoint (Cont’d)

- How to determine the similarity between any two intervals
  - Each predefined interval size is ranged from 10M 100M cycles
  - Compare basic block vector (BBV) distance of each interval
  - Draw a similarity matrix

<table>
<thead>
<tr>
<th>BB</th>
<th>Assembly Code of bzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>srl a2, 0x8, t4</td>
</tr>
<tr>
<td></td>
<td>and a2, 0xff, t12</td>
</tr>
<tr>
<td></td>
<td>addl zero, t12, s6</td>
</tr>
<tr>
<td></td>
<td>subl t7, 0x1, t7</td>
</tr>
<tr>
<td></td>
<td>cmpeq s6, 0x25, v0</td>
</tr>
<tr>
<td></td>
<td>cmpeq s6, 0, t0</td>
</tr>
<tr>
<td></td>
<td>bis v0, t0, v0</td>
</tr>
<tr>
<td></td>
<td>bne v0, 0x120018c48</td>
</tr>
<tr>
<td>2</td>
<td>subl t7, 0x1, t7</td>
</tr>
<tr>
<td></td>
<td>cmple t7, 0x3, t2</td>
</tr>
<tr>
<td></td>
<td>beq t2, 0x120018b04</td>
</tr>
<tr>
<td>3</td>
<td>ble t7, 0x120018bb4</td>
</tr>
<tr>
<td>4</td>
<td>and t4, 0xff, t5</td>
</tr>
<tr>
<td></td>
<td>srl t4, 0x8, t4</td>
</tr>
<tr>
<td></td>
<td>addl zero, t5, s6</td>
</tr>
<tr>
<td></td>
<td>cmpeq s6, 0x25, s0</td>
</tr>
<tr>
<td></td>
<td>cmpeq s6, 0, a0</td>
</tr>
<tr>
<td></td>
<td>bis s0, a0, s0</td>
</tr>
<tr>
<td></td>
<td>bne s0, 0x120018c48</td>
</tr>
<tr>
<td>5</td>
<td>subl t7, 0x1, t7</td>
</tr>
<tr>
<td></td>
<td>gt t7, 0x120018b90</td>
</tr>
</tbody>
</table>

< Example of BBs >

<table>
<thead>
<tr>
<th>ID: 1 2 3 4 5 ..</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB Exec Count: &lt;1, 20, 0, 5, 0, ..&gt;</td>
</tr>
<tr>
<td>Weigh by Block Size: &lt;8, 3, 1, 7, 2, ..&gt;</td>
</tr>
<tr>
<td>= &lt;8, 60, 0, 35, 0, ..&gt;</td>
</tr>
<tr>
<td>Normalize to 1 = &lt;8%, 58%, 0%, 34%, 0%, ..&gt;</td>
</tr>
</tbody>
</table>

- In a similarity matrix
  - Darker gray area means how similar they are
  - Lighter gray area means how different they are
  - Horizontal line: Similarity in future sample
  - Vertical line: Similarity in previous sample
Backup: SimPoint (Cont’d)

- No. of simulation points (K) for interval=100K (djjpeg) and interval=10M (gzip)

<table>
<thead>
<tr>
<th>Program (input)</th>
<th>Total IPC</th>
<th>Total Time</th>
<th>K=1</th>
<th></th>
<th>K=2</th>
<th></th>
<th>K=3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total IPC</td>
<td>Total Time</td>
<td>IPC</td>
<td>Weight</td>
<td>Time</td>
<td>IPC</td>
<td>Weight</td>
<td>Time</td>
</tr>
<tr>
<td>djjpeg (pic.jpg)</td>
<td>0.654</td>
<td>103.1</td>
<td>0.654</td>
<td>1.0</td>
<td>103.1</td>
<td>0.654</td>
<td>1.0</td>
<td>103.1</td>
</tr>
<tr>
<td></td>
<td>0.714</td>
<td>249.6</td>
<td>0.714</td>
<td>0.542</td>
<td>91.2</td>
<td>0.712</td>
<td>0.458</td>
<td>158.4</td>
</tr>
<tr>
<td></td>
<td>0.716</td>
<td>218.1</td>
<td>0.747</td>
<td>0.198</td>
<td>36.9</td>
<td>0.684</td>
<td>0.262</td>
<td>31.0</td>
</tr>
<tr>
<td></td>
<td>0.715</td>
<td>292.4</td>
<td>0.756</td>
<td>0.139</td>
<td>22.6</td>
<td>0.718</td>
<td>0.540</td>
<td>150.2</td>
</tr>
<tr>
<td>gzip (log)</td>
<td>0.6895</td>
<td>379.5</td>
<td>0.689</td>
<td>1.0</td>
<td>379.5</td>
<td>0.689</td>
<td>0.845</td>
<td>379.5</td>
</tr>
<tr>
<td></td>
<td>0.6973</td>
<td>723.2</td>
<td>0.740</td>
<td>0.155</td>
<td>343.7</td>
<td>0.689</td>
<td>0.845</td>
<td>379.5</td>
</tr>
<tr>
<td></td>
<td>0.6989</td>
<td>1032.8</td>
<td>0.689</td>
<td>0.435</td>
<td>379.9</td>
<td>0.694</td>
<td>0.410</td>
<td>309.2</td>
</tr>
<tr>
<td></td>
<td>0.7136</td>
<td>1356.3</td>
<td>0.689</td>
<td>0.437</td>
<td>379.9</td>
<td>0.729</td>
<td>0.178</td>
<td>331.5</td>
</tr>
</tbody>
</table>

- **djjpeg-pic.jpg**
  - Number of instruction: 1.014B, Simulation cycles: 1.325B, Total IPC: 0.7648
  - Total simulation time in original sim-outorder: 2034 seconds

- **gzip-log**
  - No. of instruction: 4.444B, Simulation cycles: 6.070B Total IPC: 0.7321
  - Total simulation time in original sim-outorder: 1356 seconds

- Conclusions: The performance and the accuracy of SimPoint largely varies over the determination of K. Also, the interval size matter.
Backup: Other Prior Works

- Simulation with Reduced Benchmark
  - **MinneSPEC**: Modify the reference input dataset such that it still retains the characteristics of the reference input dataset yet reduces the simulation time.
  - **SPEClite**: Alternating UA (Microarchitectural mode) and FM (Functional mode) and sample the representative 20-50 periods of one million instructions.

- Statistical Simulation (**HLS++**)
  - **Step 1**: Profiling the benchmark program to measure a collection of its execution characteristics to create statistical profile.
  - **Step 2**: Generate a synthetic trace using the statistical profile
    - Synthetic trace consists of instructions contained in basic blocks that are linked together into a control flow graph (CFG).
    - Instead of actual arguments and op-codes, each instruction is composed of a set of statistical parameters such as instruction type, I/D-cache hit probability, I-/D-TLB hit probability, branch miss probability, dynamic data dependency distance (to determine how far a consumer instruction is away from its producer), etc.
  - **Step 3**: Simulating the instructions in the synthetic trace on a trace driven simulator to obtain the performance estimate (Statistical simulation).
Backup: Other Prior Works (Cont’d)

- **Accelerate the Warm-up**
  - To provide the correct cache & branch predictor states and handle the cold start
  - **Checkpoint:** Store the hardware state at the beginning of each sample and impose this state when simulating the sampled trace
  - **Prime:** an empty cache at the beginning of each sample and uses a certain percent of each sample to warm-up the cache
  - **MRRL (Memory Reference Reuse Latency):** A histogram of the number of memory references between two references to the same memory location is used to determine when to start the warm-up

- **Accelerate the Fast-forwarding (SimSnap)**
  - Fast-forwarding takes time and it usually employs functional simulation within the cycle-accurate simulator model, e.g., (native): (functional): (cycle-accurate): (1):(32):(32X42)
  - Substitute functional simulation with native (real-time) execution using checkpoint to transfer application state to a simulator at the desired simulation points (a.k.a. Native FF)
  - Specifically, application level check-pointing (ALC) insert code to save function call sequence along with run-time information of data size, layout, etc.
  - Modify the program to capture its own internal, dynamic state such that program can restart