Proxy Benchmarks for Reproducible Research

Prof. Lizy K. John
Email: ljohn@ece.utexas.edu
www: http://users.ece.utexas.edu/~ljohn
Outline

- Introduction to Proxy Benchmarks
  - Miniaturization, Software Stack Abstraction, Proprietary Codes
- Metrics, Techniques and Results for early Proxies
  - Bell, Joshi, Ganesan, and Panda Proxies; Results
- SimPoint as an alternative (CPU 2017 SimPoints) – Steven Flolid
- Challenges - Improving Accuracy of Proxy Benchmarks
  - Branch and Memory Behavior – Problem and Current State
  - Presentations by Zachary Susskind and Steven Flolid
- Industry Use
  - Intel – Presentation from Emily Shriver
- ISA Independence - LLVM Proxies
- Reproducibility of Research using Proxies
- Other Applications of Proxy Code Generator
Introduction

- Bad Press on Early Synthetic Benchmarks
  - Whetstone, Dhrystone
  - Misuse of Synthetic Benchmarks
  - Not many good memories
  - A totally new look at a new kind of synthetic benchmarks
- A few new scenarios
  - The Pre-Silicon Nightmare
  - Proprietary Applications that cannot be shared
  - Expiring Benchmarks
  - Power and Thermal Stress Benchmarks
The Presilicon Design Nightmare

Modern microprocessors are built from millions of lines of VHDL/Verilog code and billions of transistors.

RTL and Performance Models are 1000X or more slower than hardware.

Industry standard benchmarks run for hours on actual hardware.

Design tradeoffs need to be analyzed quickly to make timely design decisions.

If the wrong decisions are made, the presilicon nightmare will become a serious postsilicon nightmare.
Methodology: Workload Modeling and Synthesis

1. Identify a list of intrinsic properties $\alpha, \beta, \gamma \ldots \omega$ to uniquely describe the behavior of a modern program

2. Develop a methodology to reproduce the same behavior synthetically
Representing a Program with Intrinsic Properties

- Is it possible to find a list of intrinsic properties $\alpha, \beta, \gamma, \ldots, \omega$ to uniquely describe a program?

- What will be the characteristics $\alpha, \beta, \gamma, \ldots, \omega$?

- Platform (Micro-architecture) Independent Characteristics
  - i-mix, locality, address patterns, dependency

- Platform (Micro-architecture) Dependent Characteristics
  - Cache-hit rates, IPC/throughput, coherency traffic
Proxy Workload Generation

Workload Space of Inherent Program Characteristics

Workload Signature = Point in Workload Space

Modeling Workload Attributes into Synthetic Workload

Synthetic Benchmark Clone

Experiment Environment

VHDL/Verilog Models

Performance Simulator

Lizy K. John, LCA, UTAustin

7

11/3/19
Miniaturized Proxy Generation Process

**Real Workload**

1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010

**Detailed Workload Profile**

Instruction Mix, ILP, Data Access Patterns, branch patterns

![Instruction Mix Diagram]

**Synthetic Clone**

ADD R1, R2, R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R6
STORE R3, R10, R20
ADD R1, R2, R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R6
BEQ R3, R6, LOOP
SUB R3, R5, R6
STORE R3, R10, R20
DIV R10, R2, R1
............

**Measure Workload Characteristics at Basic Block Granularity**

**Generate Synthetic Clone with Similar Inherent Characteristics**
Steps in Proxy generation

- Generate Workload profile
  - Command: `./mkJSON_both.sh $input $output $output_name.fixed`

- Generate Proxy
  - Command: `./CodeGen.py --configFile=$file -o $output`

- Run Proxy
  - Command: `c executable`
Statistical Flow Graph (SFG)
## Profiling the Original Workload

<table>
<thead>
<tr>
<th>No.</th>
<th>Metric</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dynamic execution freq. of basic blocks</td>
<td>Control flow predictability</td>
</tr>
<tr>
<td>2</td>
<td>Average basic block size</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Branch taken rate for each branch</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Instruction pattern in a basic block</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Branch transition rate for each branch present in the workload</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>% Integer multiplication</td>
<td>Instruction mix</td>
</tr>
<tr>
<td>7</td>
<td>% Integer division</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>% FP multiplication</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>%FP division</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>% Loads</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>% Stores</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>% Branches</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Dependency distance distribution per instruction type</td>
<td>Instruction level parallelism</td>
</tr>
<tr>
<td>14</td>
<td>Stride value per static load/store</td>
<td>Data locality</td>
</tr>
<tr>
<td>15</td>
<td>Data Footprint of the workload</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Mean and standard deviation of the MLP</td>
<td>Memory level parallelism</td>
</tr>
<tr>
<td>17</td>
<td>Mean number of consecutive dynamic inst. when no outstanding long-latency loads</td>
<td></td>
</tr>
</tbody>
</table>
### Abstract

#### Workload Model

<table>
<thead>
<tr>
<th>No.</th>
<th>Metric</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basic block size</td>
<td>Control flow predictability</td>
</tr>
<tr>
<td>2</td>
<td>Branch taken rate for each branch</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Branch transition rate</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Proportion of INT ALU, INT MUL, INT DIV, FP ADD, FP MUL, FP DIV, FP MOV, FP SQRT, LOAD &amp; STORE</td>
<td>Instruction mix</td>
</tr>
<tr>
<td>5</td>
<td>Dependency distance distribution</td>
<td>Instruction level parallelism</td>
</tr>
<tr>
<td>6</td>
<td>Private stride value per static load/store</td>
<td>Data locality</td>
</tr>
<tr>
<td>7</td>
<td>Data Footprint of the workload</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Mean and standard deviation of the MLP</td>
<td>Memory Level Parallelism (MLP)</td>
</tr>
<tr>
<td>9</td>
<td>MLP frequency</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Number of threads</td>
<td>Thread level parallelism</td>
</tr>
<tr>
<td>11</td>
<td>Thread class and processor assignment</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Percentage loads to private data</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Percentage loads to read-only data</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Percentage migratory loads</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Percentage consumer loads</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Percentage irregular loads</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Percentage stores to private data</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Percentage producer stores</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Percentage irregular stores</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Shared stride value per static load/store</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Data pool distribution based on sharing patterns</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Number of lock/unlock pairs and</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Number of mutex objects</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Number of Instructions between lock and unlock</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Dependency Distance

- Instruction level parallelism:
  - Lower dependency distance corresponds to lower ILP and vice versa.

```
ADD R1, R3, R4
MUL R5, R3, R2
ADD R5, R3, R6
LD R4, (R1)
SUB R8, R2, R1
```

Read After Write Dependency Distance = 3
### CFG Information - SPEC CPU2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of B.Blks</th>
<th># B.Blks for 90% of Prog Exec.</th>
<th>Branch Transition Rate</th>
<th>Average B.Blk Size</th>
<th>Average Number of Succ. B.Blks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CINT 2006</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400.perlbench</td>
<td>1620</td>
<td>169</td>
<td>0.07</td>
<td>5.99</td>
<td>1.93</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>93</td>
<td>15</td>
<td>0.06</td>
<td>7.91</td>
<td>1.94</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>617</td>
<td>129</td>
<td>0.15</td>
<td>6.97</td>
<td>2.42</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>142</td>
<td>12</td>
<td>0.10</td>
<td>8.12</td>
<td>1.49</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>281</td>
<td>70</td>
<td>0.17</td>
<td>5.97</td>
<td>2.54</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>29</td>
<td>3</td>
<td>0.03</td>
<td>4.59</td>
<td>1.41</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>1074</td>
<td>55</td>
<td>0.09</td>
<td>15.13</td>
<td>1.69</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>443</td>
<td>112</td>
<td>0.08</td>
<td>5.25</td>
<td>2.06</td>
</tr>
<tr>
<td>473.astar</td>
<td>96</td>
<td>16</td>
<td>0.19</td>
<td>8.77</td>
<td>1.43</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>62</td>
<td>11</td>
<td>0.00</td>
<td>3.54</td>
<td>2.38</td>
</tr>
<tr>
<td>429.mcf</td>
<td>179</td>
<td>96</td>
<td>0.01</td>
<td>10.58</td>
<td>1.21</td>
</tr>
<tr>
<td>403.gcc</td>
<td>698</td>
<td>363</td>
<td>0.08</td>
<td>7.14</td>
<td>1.76</td>
</tr>
<tr>
<td><strong>CFP 2006</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410.bwaves</td>
<td>32</td>
<td>10</td>
<td>0.26</td>
<td>32.61</td>
<td>1.50</td>
</tr>
<tr>
<td>433.milc</td>
<td>42</td>
<td>18</td>
<td>0.44</td>
<td>15.32</td>
<td>1.48</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>48</td>
<td>13</td>
<td>0.02</td>
<td>43.29</td>
<td>1.67</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>6</td>
<td>3</td>
<td>0.07</td>
<td>518.06</td>
<td>1.33</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>22</td>
<td>7</td>
<td>0.18</td>
<td>247.16</td>
<td>1.27</td>
</tr>
<tr>
<td>437.lelie3d</td>
<td>533</td>
<td>13</td>
<td>0.02</td>
<td>20.43</td>
<td>1.74</td>
</tr>
<tr>
<td>444.namd</td>
<td>70</td>
<td>8</td>
<td>0.03</td>
<td>18.95</td>
<td>1.61</td>
</tr>
<tr>
<td>450.soplex</td>
<td>358</td>
<td>14</td>
<td>0.03</td>
<td>7.79</td>
<td>1.45</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>119</td>
<td>10</td>
<td>0.01</td>
<td>48.81</td>
<td>1.40</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>544</td>
<td>20</td>
<td>0.07</td>
<td>12.76</td>
<td>1.70</td>
</tr>
</tbody>
</table>
## CFG Information – Implant Bench

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of B.Blks</th>
<th># B.Blks for 90% of Prog. Exec.</th>
<th>Branch Transition Rate</th>
<th>Average B.Blk Size</th>
<th>Average # of Succ. B.Blks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI_Adaline</td>
<td>342</td>
<td>6</td>
<td>0.05</td>
<td>39.22</td>
<td>1.63</td>
</tr>
<tr>
<td>AI_BPN</td>
<td>410</td>
<td>13</td>
<td>0.06</td>
<td>54.27</td>
<td>1.80</td>
</tr>
<tr>
<td>AI_GA</td>
<td>503</td>
<td>160</td>
<td>0.23</td>
<td>7.7</td>
<td>2.26</td>
</tr>
<tr>
<td>Bioinformatics_ELO</td>
<td>194</td>
<td>15</td>
<td>0.05</td>
<td>9.42</td>
<td>1.72</td>
</tr>
<tr>
<td>Bioinformatics_LMGC</td>
<td>498</td>
<td>111</td>
<td>0.25</td>
<td>7.55</td>
<td>1.91</td>
</tr>
<tr>
<td>Genomics_HMM</td>
<td>463</td>
<td>119</td>
<td>0.28</td>
<td>8.03</td>
<td>1.80</td>
</tr>
<tr>
<td>Genomics_NJ</td>
<td>196</td>
<td>18</td>
<td>0.17</td>
<td>9.41</td>
<td>2.03</td>
</tr>
<tr>
<td>HeartActivity_pNNx</td>
<td>479</td>
<td>97</td>
<td>0.17</td>
<td>6.41</td>
<td>1.68</td>
</tr>
<tr>
<td>Physiology_AFVP</td>
<td>679</td>
<td>105</td>
<td>0.26</td>
<td>7.07</td>
<td>1.77</td>
</tr>
<tr>
<td>Physiology_ECGSYN</td>
<td>647</td>
<td>142</td>
<td>0.18</td>
<td>8.99</td>
<td>1.65</td>
</tr>
<tr>
<td>Reliability_alder32</td>
<td>143</td>
<td>101</td>
<td>0.43</td>
<td>6.81</td>
<td>1.88</td>
</tr>
<tr>
<td>Reliability_crc</td>
<td>172</td>
<td>89</td>
<td>0.36</td>
<td>6.1</td>
<td>2.20</td>
</tr>
<tr>
<td>Reliability_luhn</td>
<td>155</td>
<td>90</td>
<td>0.40</td>
<td>6.46</td>
<td>2.10</td>
</tr>
<tr>
<td>Reliability_reed_solm</td>
<td>255</td>
<td>4</td>
<td>0.02</td>
<td>4.71</td>
<td>1.93</td>
</tr>
<tr>
<td>Security_haval</td>
<td>338</td>
<td>82</td>
<td>0.25</td>
<td>8.1</td>
<td>1.94</td>
</tr>
<tr>
<td>Security_KHAZAD</td>
<td>198</td>
<td>17</td>
<td>0.14</td>
<td>121.65</td>
<td>1.97</td>
</tr>
<tr>
<td>Security_sha2</td>
<td>313</td>
<td>7</td>
<td>0.06</td>
<td>37.32</td>
<td>1.76</td>
</tr>
</tbody>
</table>

- Dynamic number of instructions varies from hundreds million to few thousands
Workload Synthesis Algorithm

1. Generate program spine using basic block information.
2. Populate basic block with instructions satisfying target IMIX.
   - Model inter-instruction dependencies
     - Assign producer-consumer chains
3. Model memory instruction stride/offset assignment
   - Stride-based memory model
4. Model branch predictability
   - Insert test operation before every branch instruction to control branch outcome
5. Code generation
   - GCC inline assembly for x86

C code with embedded assembly instructions
- Outer loop controls the # of dynamic instructions
- Inner loop iterations control the data footprint

Diagram:
- Outer-loop starts
  - Reset data arrays
  - Outer loop iterations
    - Inner loop iterations
      - Memory instructions
      - Branch instructions
      - Code generation
Code Generation

- **Step 1:**
  - Usage of two loops
  - Based on Instruction footprint of original, fix the number of basic blocks in the synthetic

[Diagram showing the process of code generation with loops and instruction footprints]
Code Generation

- **Step 2: For each Basic Block**
  - Choose the instruction pattern from a pool of instruction patterns based on the frequency in the profile in terms of Instruction type
  - If not use BB size and global Instruction mix

```
- 10.3%
  a,fa,m,m,a,ld,st,Br
  TR: 0.15

- 23.1%
  m,m,ld,a,a,Br
  TR: 0.07

- 2.5%
  a,x,ld,ld,ld,s,s,Br
  TR: 0.08

- 50.9%
  a,fm,a,m,m,ld,ld,ld,a,a,a,Br
  TR: 0.44

- 13.2%
  a,fa,a,a,a,ld,st,Br
  TR: 0.01
```
Step 3: Bind the basic blocks together using conditional jumps

- Low Transition Rate -> always taken/not taken
- Group into pools & use modulo operations

**Diagram:**

- Inner Loop 1:
  - 10.3%: a,fa,m,m,a,ld,st,Br
  - TR: 0.15

- Outer Loop:
  - 23.1%: m,m,ld,a,a,Br
  - TR: 0.07

- Inner Loop 2:
  - 50.9%: a,fa,m,m,ld,ld,ld,a,a,Br
  - TR: 0.44

- 2.5%: a,x,ld,ld,ld,s,s,Br
  - TR: 0.08

- 13.2%: a,fa,a,a,a,ld,st,Br
  - TR: 0.01
Step 4: For each instruction

- Find a producer instn to assign a register dependency
- Not compatible? Move up/down

10.3%
\[ a,fa,m,m,a,ld,st,Br \]
TR: 0.15

23.1%
\[ m,m,ld,a,a,Br \]
TR: 0.07

2.5%
\[ a,x,ld,ld,ld,s,s,Br \]
TR: 0.08

50.9%
\[ a,fa,m,m,m,ld,ld,a,a,Br \]
TR: 0.44

13.2%
\[ a,fa,a,a,ld,st,Br \]
TR: 0.01

11/3/19

Lizy K. John, LCA, UT Austin
# Code Generation

## Step 5: Assign registers

- **Destination registers** – RoundRobin
- **Source registers** – based on dependency

![Flowchart with register assignments](chart.png)

**Step 5: Assign registers**

- Destination registers – RoundRobin
- Source registers – based on dependency

**Flowchart Description**

- **10.3%** a,fa,m,m,a,ld,st,Br TR: 0.15
- **23.1%** m,m,ld,a,a,Br TR: 0.07
- **50.9%** a,fa,m,m,ld,ld,a,a,a,Br TR: 0.44
- **13.2%** a,fa,a,a,ld,st,Br TR: 0.01
- **2.5%** a,fa,a,m,m,ld,ld,ld,a,a,a,Br
- **13.2%** a,fa,m,m,ld,ld,ld,a,a,a,Br
- **23.1%** a,x,ld,ld,ld,s,s,Br
- **50.9%** a,fa,m,m,ld,ld,a,a,a,Br
- **10.3%** a,fa,m,m,a,ld,st,Br
- **50.9%** a,fa,m,m,ld,ld,a,a,a,Br
- **23.1%** a,x,ld,ld,ld,s,s,Br
- **13.2%** a,fa,a,a,ld,st,Br
- **13.2%** a,fa,a,a,ld,ld,a,a,a,Br
- **23.1%** m,m,ld,a,a,Br
- **50.9%** a,fa,m,m,ld,ld,ld,a,a,a,Br
- **10.3%** a,fa,m,m,a,ld,st,Br
- **50.9%** a,fa,m,m,ld,ld,a,a,a,Br
- **23.1%** a,x,ld,ld,ld,s,s,Br
- **13.2%** a,fa,a,a,ld,st,Br
- **13.2%** a,fa,a,a,ld,ld,a,a,a,Br
- **23.1%** m,m,ld,a,a,Br
- **50.9%** a,fa,m,m,ld,ld,ld,a,a,a,Br
- **10.3%** a,fa,m,m,a,ld,st,Br
- **50.9%** a,fa,m,m,ld,ld,a,a,a,Br
- **23.1%** a,x,ld,ld,ld,s,s,Br
- **13.2%** a,fa,a,a,ld,st,Br
- **13.2%** a,fa,a,a,ld,ld,a,a,a,Br
- **23.1%** m,m,ld,a,a,Br
Code Generation

- **Step 6: Memory access model**
  - Ld/st access a set of 1-D arrays in a strided fashion
  - Ld/St - group into pools, assign array, 1 address calc instruction
  - Pointers - top of array at end of inner loop when required data footprint is touched

- **Inner Loop 1**
  - 10.3% a,fa,m,m,a,ld,st,Br
  - TR: 0.15

- **Inner Loop 2**
  - 23.1% m,m,ld,a,a,Br
  - TR: 0.07
  - 2.5% a,x,ld,ld,ld,s,s,Br
  - TR: 0.08
  - 50.9% a,fa,a,m,m,ld,ld,a,a,a,Br
  - TR: 0.44

- **Outer Loop**
  - 13.2% a,fa,a,a,a,ld,st,Br
  - TR: 0.01
  - 10.3% a,fa,m,m,a,ld,st,Br
  - TR: 0.07

- **Array 1**
  - a,fa,m,m,a,ld,st,Br

- **Array 2**
  - a,fa,m,m,ld,ld,ld,a,a,a,Br

- **Array 3**
  - m,m,ld,a,a,Br

- **Array 4**
  - a,fa,a,a,a,ld,st,Br
## Code Generation

### Step 6: MLP model

- Load-Load dependencies
- Placement of highly strided loads
- For very infrequent highly bursty long latency loads, only 1 loop has all the long latency loads
Performance Results from Early Proxies

Bell: Some microarch dependent metrics; Strides computed based on measured cache miss rates

Average Error of 6.3% & Maximum Error of 9.9% (mcf) [Bell05,06]
Overall Power Estimation

Power Results from Bell Proxies

Average Error of 7.3% & Maximum Error of 10.6% (mcf)
Five Orders of Magnitude Speedup!

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Speedup from Synthetic Benchmark Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applu</td>
<td>22,300</td>
</tr>
<tr>
<td>apsi</td>
<td>34,700</td>
</tr>
<tr>
<td>art</td>
<td>4,500</td>
</tr>
<tr>
<td>bzip2</td>
<td>12,800</td>
</tr>
<tr>
<td>crafty</td>
<td>19,100</td>
</tr>
<tr>
<td>eon</td>
<td>8,000</td>
</tr>
<tr>
<td>equake</td>
<td>13,100</td>
</tr>
<tr>
<td>gcc</td>
<td>4,600</td>
</tr>
<tr>
<td>gzip</td>
<td>10,300</td>
</tr>
<tr>
<td>mcf</td>
<td>6,100</td>
</tr>
<tr>
<td>mesa</td>
<td>14,100</td>
</tr>
<tr>
<td>mgrid</td>
<td>41,900</td>
</tr>
<tr>
<td>swim</td>
<td>22,500</td>
</tr>
<tr>
<td>twolf</td>
<td>34,600</td>
</tr>
<tr>
<td>vortex</td>
<td>11,800</td>
</tr>
<tr>
<td>vpr</td>
<td>8,400</td>
</tr>
<tr>
<td>wupwise</td>
<td>34,900</td>
</tr>
</tbody>
</table>
Improvements from Joshi and Ganesan

- Joshi added the following metrics:
  - Branch Transition Rate
  - Data Reuse Distance

- Ganesan added the following metrics:
  - MLP (Memory Level Parallelism)
  - Parallel Code Data sharing and Synchronization Metrics
Previous approaches and Memory Level Parallelism

- Existing synthetic workload generation methodologies Joshi et al [IISWC 06], Bell et al [ISPASS ’06, ICS ’05]
  - Capture control flow, cache access and ILP behavior
  - Do not capture the burstiness of long-latency loads or the Memory Level Parallelism (MLP)
  - May still match the missrate of the original, but not the performance
Memory Level Parallelism
IPC Comparison – SPEC CPU2006

- MLP Aware - Avg = 2.8%, Max = 7.7% for 464.h264ref
- MLP Unaware – Avg = 15.3%
IPC Comparison – ImplantBench

- Avg = 2.9%, Max = 7.2%
Power Comparison

- **CPU2006**
  - **Avg = 14%**

- **ImplantBench**
  - **Avg = 2.5%**

- **Graphs show Power per cycle (W) for various benchmarks**
  - **Orig**, **MLP aware synth**, **MLP unaware synth**

Lizy K. John, LCA, UT Austin
11/3/19
Relative IPC Accuracy for CINT2006 Benchmarks

462.libquantum

Machine configurations

471.omnetpp

Machine configurations
Relative Power Accuracy for CPU2006

**462.libquantum**

![Graph showing power per cycle for 462.libquantum with machine configurations and power data for Orig and Synth.]  

**433.milc**

![Graph showing power per cycle for 433.milc with machine configurations and power data for Orig and Synth.]
Simulation Time Reduction
Performance Proxies for Big Data Workloads

- Web Browsers
- Mobile Devices
- Personal Computers
- Device Specific Renderers
- HTML, JavaScript, etc

Presentation Layer

Business Logic Layer

Data Access Layer, Load Balancing

- SQL / NoSQL Server Drivers
- Database Drivers
- XML Query
- File System Query

- Database Server
- Database
- XML Storage
- File System

Operating System
Instruction-level Profiling Impractical; Estimate METRICS from Perf Counters

- **Instruction-mix**
  - Measured using hardware performance counters
  - Fraction of integer (INT) ALU, INT MUL, INT DIV, floating-point (FP) ADD, FP DIV, FP MUL, loads, stores, control-flow instructions.

- **Instruction footprint**
  - Derived based on the instruction cache miss rate of the original application on a default instruction cache size (64KB, 64B line-size, 2-way set-associative).

- **Average basic-block size**
  - Estimated based on the instruction count and control instruction mix metrics.
Instruction-level METRIC Estimation (Contd)

- **Instruction level parallelism:**
  - Modeled by controlling the dependency distance (DD) between instructions.
  - Lower dependency distance corresponds to lower ILP and vice versa.

  ```
  ADD R1, R3, R4
  MUL R5, R3, R2
  ADD R5, R3, R6
  LD R4, (R1)
  SUB R8, R2, R1
  ```

  Read After Write Dependency Distance = 3

- Avg DD estimated based on the fraction of dependency-related stall events (measured using hardware performance counters) of an application.
Memory-access characteristics

• Original approach – PIN-based memory locality characterization
  
  • Local Stride - Per ld/st stride distribution
  
  • Global Stride – Stride distribution of the global memory access streams
  
• Perf Counter Approach: Strides estimated based on the data cache miss rates of the original application.
  
  • E.g., 50% hit-rates can be modeled using a stride of 8 (assuming 64B cache-line size)
CONTROL-FLOW BEHAVIOR

- Branch predictability (BP) of the original application is measured using the branch prediction rate.

- A branch instruction’s predictability can be controlled using its transition frequency.

\[
\text{Branch Transition Rate} = \frac{\# \text{ of Taken} - \# \text{ of Not Taken transitions}}{\# \text{ of times executed}}
\]

- PerfProx estimates the fraction of control instructions in the proxy benchmark that will have a particular predictability behavior.

Find the set of \( f_i \) such that

\[
BP = f_1 \times 1 + f_2 \times 0.5 + f_3 \times 0.33 + \ldots
\]

- Assuming a 2-bit saturating counter based branch predictor, \( f_1, f_2, f_3 \ldots \) are the fraction of branches with 100% (TTTTTTTT or NNNNNNNN), 50% (TNTNTNTN), 33% (TTNTTNTTN) predictability respectively.
System CALL Activity

• Many emerging big-data applications spend a significant fraction of their execution time (upto 30%) executing operating system (OS) code.

• System activity monitored using STRACE tool and the fraction of user-mode and kernel-mode instructions using hardware performance counters.

• Modeled by inserting target fraction of system calls into the basic blocks in the proxy benchmark.
DATABASE PROXY EVALUATION

• Databases - Cassandra, MongoDB and MySQL

• Benchmarks
  • Data analytics (TPC-H Benchmarks) - 10GB, Full runs of Q1, Q3, Q6, Q14 and Q19.
  • Data serving (Yahoo! Cloud Serving Benchmark, YCSB) – 12GB, 100 Million operations.

• Performance monitoring performed on real systems using Linux Perf tool.

• Average instruction-count of the generated proxy benchmarks is ~2 billion (~520 times smaller than original database applications).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>System-A</th>
<th>System-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Architecture</td>
<td>64-bit processor, Core micro-architecture</td>
<td>64-bit processor, Ivy-bridge micro-architecture</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>2 GHz</td>
<td>2.50 GHz</td>
</tr>
<tr>
<td>Cache Configuration</td>
<td>Private L1 caches (64 KB I and D caches), 12 MB L2 cache</td>
<td>Three levels of caches, 1.5MB L2, 15MB L3 cache</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB DRAM</td>
<td>64 GB DRAM</td>
</tr>
</tbody>
</table>
The proxy benchmarks mimic the IPC of the original applications with 94.9% (avg) accuracy for data-serving applications and 93.5% (avg) accuracy for data-analytics applications.
Performance VALIDATION ON system-A

- Branch Prediction Rate
- Normalized L1 miss rate
- Error between Actual and Proxy

ρ = 0.99
ρ = 0.937
ρ = 0.967

Proxy Kernel, Proxy User, Actual Kernel, Actual User

%OS, %user IMIX

Lizy K. John, LCA, UT Austin
11/3/19
Correlation between the average power consumption of the proxy and actual applications is high (0.97).
PERFORMANCE CROSS-VALIDATION ON system-B

Cassandra YCSB

MySQL TPC-H

MongoDB YCSB
SUMMARY OF ORIGINAL TO PROXY COMPARISON

- Cassandra-ORIG
- Cassandra-Proxy
- MongoDB-ORIG
- MongoDB-Proxy
- MySQL-ORIG
- MySQL-Proxy
Where to find proxies

• Github repository with most up to date proxies: https://github.com/UT-LCA/Proxy-Benchmarks

• Proxies are all c programs that can be compiled with GCC
  • Example: gcc Deepsjeng.c –o Deepsjeng.o

• Performance counters can be gathered with perf or other tools
  • Example: perf Deepsjeng.o
Cannot retrieve the latest commit at this time.

..

- Big_Data_Benchmarks
- CPU_2017
Cannot retrieve the latest commit at this time.

..

- proxy_cas_wc.c
- proxy_cas_wc_sys.c
- proxy_cas_wd.c
- proxy_mongowa.c
- proxy_mongowb.c
- proxy_tpcq1.c
- proxy_tpcq14.c
- proxy_tpcq19.c
- proxy_tpcq3.c
- proxy_tpcq6.c
Cannot retrieve the latest commit at this time.

```
..
deepsjeng.c
exchange2.c
gcc.c
leela.c
mcf.c
omnetpp.c
perl.c
x264.c
xz.c
```
SPEC’17 BENCHMARK OVERVIEW

• CPU2017 Benchmark Overview
  • 43 Benchmarks: Speed & Rate versions of INT and FP programs

• CPU2017 vs CPU2006 Runtime Comparison

<table>
<thead>
<tr>
<th>Runtime Chars.</th>
<th>SPEC CPU2017</th>
<th>SPEC CPU2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Inst. Count</td>
<td>7,876,399,266,409</td>
<td>1,896,452,771,834</td>
</tr>
<tr>
<td>Total run time (all benchmarks)</td>
<td>72, 299 seconds</td>
<td>9860 seconds</td>
</tr>
</tbody>
</table>
# Benchmark Characteristics

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Language</th>
<th>KLoC</th>
<th>Application area</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench_s</td>
<td>C</td>
<td>362</td>
<td>Perl interpreter</td>
</tr>
<tr>
<td>Deepsjeng_s</td>
<td>C++</td>
<td>10</td>
<td>Artificial intelligence: alpha beta tree search (chess)</td>
</tr>
<tr>
<td>Omnetpp_s</td>
<td>C++</td>
<td>134</td>
<td>Discrete event simulation – computer network</td>
</tr>
<tr>
<td>Gcc_s</td>
<td>C</td>
<td>1304</td>
<td>GNU C compiler</td>
</tr>
<tr>
<td>Leela_s</td>
<td>C++</td>
<td>21</td>
<td>Artificial intelligence: Monte carlo tree search (Go)</td>
</tr>
</tbody>
</table>
## Benchmark Characteristics

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Language</th>
<th>KLoC</th>
<th>Application area</th>
</tr>
</thead>
<tbody>
<tr>
<td>xalanxbmk_s</td>
<td>C++</td>
<td>520</td>
<td>XML to HTML conversion via XLST</td>
</tr>
<tr>
<td>mcf_s</td>
<td>C</td>
<td>3</td>
<td>Route Planning</td>
</tr>
<tr>
<td>x264_s</td>
<td>C</td>
<td>96</td>
<td>Video compression</td>
</tr>
<tr>
<td>exchange2_s</td>
<td>Fortran</td>
<td>1</td>
<td>Artificial Intelligence: recursive solution generator (Sudoku)</td>
</tr>
<tr>
<td>xz_s</td>
<td>C</td>
<td>33</td>
<td>General Data Compression</td>
</tr>
</tbody>
</table>
Proxy Performance (IPC)

Instructions Per Cycle (IPC)

orig    proxy    error

Perlbench  Deepsjeng  Omnetpp  Gcc  Leela  X264  Exchange2  Xz  Xalanxbmk  Mcf  Average

0.00%  5.00%  10.00%  15.00%
Proxy Performance (Power)

Running Average Power Limit (RAPL)

- orig
- proxy
- error

Error values for different benchmarks:
- peribench
- deepsjeng
- omnetpp
- gcc
- leela
- x264
- exchange2
- xz
- xalanxbrmk
- mcf
- average

Lizy K. John, LCA, UTAustin
## Degree of Miniaturization

<table>
<thead>
<tr>
<th></th>
<th>perlbench</th>
<th>deepsjeng</th>
<th>omnetpp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>Proxy</td>
<td>Orig</td>
</tr>
<tr>
<td>#Dyn Instructions</td>
<td>1,283,859,685,081</td>
<td>630,103,279</td>
<td>2,275,616,966,086</td>
</tr>
<tr>
<td>Orig/Proxy # Instructions</td>
<td>2038</td>
<td></td>
<td>3388</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>leela</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>Proxy</td>
</tr>
<tr>
<td>#Dyn Instructions</td>
<td>4,577,230,000,000</td>
<td>982,597,009</td>
</tr>
<tr>
<td>Orig/Proxy # Instructions</td>
<td>4658</td>
<td></td>
</tr>
</tbody>
</table>

- all proxies have less than 1 billion dynamic instruction count
# Degree of Miniaturization

<table>
<thead>
<tr>
<th>Orig/Proxy</th>
<th>xalanxbmk</th>
<th>Orig</th>
<th>Proxy</th>
<th>mcf</th>
<th>Orig</th>
<th>Proxy</th>
<th>x264</th>
<th>Orig</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>1,315,970,803,200</td>
<td>1,782,190,000,000</td>
<td>789,554,535</td>
<td>5030,000,000,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td>510,531,529</td>
<td>789,554,535</td>
<td>5030,000,000,000</td>
<td>809,326,203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Dyn Instructions</td>
<td>2578</td>
<td>2257</td>
<td>6219</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Orig/Proxy</th>
<th>exchange2</th>
<th>Orig</th>
<th>Proxy</th>
<th>xz</th>
<th>Orig</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>6,643,720,000,000</td>
<td>5,030,530,000,000</td>
<td>818,316,812</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td>895,456,832</td>
<td>5,030,530,000,000</td>
<td>818,316,812</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orig/Proxy # Instructions</td>
<td>7419</td>
<td>6147</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- all proxies have less than 1 billion dynamic instruction count
Control-flow Behavior

Comparing branch misprediction rate of ORIG and PROXY

Branch Misprediction Rate (%)
## Cache & TLB Performance (1/2)

<table>
<thead>
<tr>
<th></th>
<th>perlbench</th>
<th>deepsjeng</th>
<th>omnetpp</th>
<th>gcc</th>
<th>leela</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ORIG</td>
<td>PROXY</td>
<td>ORIG</td>
<td>PROXY</td>
<td>ORIG</td>
</tr>
<tr>
<td>DTLB MPKI</td>
<td>0.10</td>
<td>0.01</td>
<td>0.22</td>
<td>0.01</td>
<td>5.00</td>
</tr>
<tr>
<td>Dcache MPKI</td>
<td>3.51</td>
<td>4.77</td>
<td>4.55</td>
<td>7.30</td>
<td>36.64</td>
</tr>
<tr>
<td>L2 MPKI</td>
<td>1.64</td>
<td>0.00</td>
<td>1.41</td>
<td>0.02</td>
<td>17.79</td>
</tr>
<tr>
<td>L3 MPKI</td>
<td>0.01</td>
<td>0.00</td>
<td>1.10</td>
<td>0.00</td>
<td>6.31</td>
</tr>
</tbody>
</table>
## Cache & TLB Performance (2/2)

<table>
<thead>
<tr>
<th></th>
<th>X264</th>
<th>Exchange2</th>
<th>xz</th>
<th>xalanxbmk</th>
<th>mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ORIG</td>
<td>PROXY</td>
<td>ORIG</td>
<td>PROXY</td>
<td>ORIG</td>
</tr>
<tr>
<td>DTLB MPKI</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td>0.02</td>
<td>2.04</td>
</tr>
<tr>
<td>Dcache MPKI</td>
<td>1.44</td>
<td>0.58</td>
<td>0.02</td>
<td>1.38</td>
<td>16.06</td>
</tr>
<tr>
<td>L2 MPKI</td>
<td>0.34</td>
<td>0.33</td>
<td>0.00</td>
<td>0.74</td>
<td>5.00</td>
</tr>
<tr>
<td>L3 MPKI</td>
<td>0.10</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.87</td>
</tr>
</tbody>
</table>
SimPoint as an Alternative – UT LCA SimPoints

- SIMPOINTS are popularly used with simulators to reduce simulation time.

- UT Laboratory for Computer Architecture has generated SimPoints for SPEC CPU 2017

SimPoints

This folder holds the results of SimPoint study. Pinballs for SimPoints are available for download [here](#).

**Usage**

Pinballs can be run natively using the Software Develop Emulator from Intel ([https://software.intel.com/en-us/articles/intel-software-development-emulator](https://software.intel.com/en-us/articles/intel-software-development-emulator)). Gem5 ([http://gem5.org/Simpoints](http://gem5.org/Simpoints)) and Sniper ([http://snipersim.org/w/Pinballs](http://snipersim.org/w/Pinballs)) also support the replaying of pinballs.

Interpreting the meaning of the 10 to 30 Pinballs for each workload in SPEC CPU 2017 is fairly straightforward. Each pinball base name specifies the key details for the pinball. An example is included below:

depsjeng.test_38307_t0r1_warmup1501_prolog0_region100000038_epilog0_001_0-05045.0.address

The file has many parts to its name, each with a specific meaning.

"program name","run name"_"run number"_t"thread number"r"region number"_warmup"num warm up instructions"_prolog"num prolog instructions"region"num instructions in region of interest"epilog"num epilog instructions""region number again""thread number again"-"weight of the region".0."file type"
SimPoint as an Alternative

- **Simpoint** [1] breaks a workload into equal sized regions (100 Million Instructions)
- Regions are profiled based on micro-architecture independent Basic Block Vectors (BBV)
- Similar regions are clustered together based on the BBV using K-means
- A single region is simulated to represent each cluster

SimPoint Limitations

- Dominant region can vary significantly from average behavior

<table>
<thead>
<tr>
<th>Cluster ID</th>
<th>Cluster Weight</th>
<th>Representative Perf. Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.87%</td>
<td>1.60</td>
</tr>
<tr>
<td>1</td>
<td>23.62%</td>
<td>1.21</td>
</tr>
<tr>
<td>2</td>
<td>1.36%</td>
<td>1.91</td>
</tr>
<tr>
<td>3</td>
<td>12.00%</td>
<td>1.47</td>
</tr>
<tr>
<td>4</td>
<td>0.09%</td>
<td>1.04</td>
</tr>
<tr>
<td>5</td>
<td>0.44%</td>
<td>1.47</td>
</tr>
<tr>
<td>6</td>
<td>0.69%</td>
<td>1.61</td>
</tr>
<tr>
<td>7</td>
<td>0.36%</td>
<td>1.46</td>
</tr>
<tr>
<td>8</td>
<td>0.84%</td>
<td>1.54</td>
</tr>
<tr>
<td>9</td>
<td>0.17%</td>
<td>1.33</td>
</tr>
<tr>
<td>10</td>
<td>3.60%</td>
<td>1.16</td>
</tr>
<tr>
<td>11</td>
<td>0.82%</td>
<td>1.74</td>
</tr>
<tr>
<td>12</td>
<td>5.28%</td>
<td>1.77</td>
</tr>
<tr>
<td>13</td>
<td>1.21%</td>
<td>1.41</td>
</tr>
<tr>
<td>14</td>
<td>48.65%</td>
<td>1.85</td>
</tr>
</tbody>
</table>

Weighted Sum: 1.61
SimPoints for CPU 2017

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Simulation Points</th>
<th>90 percentile Points</th>
<th>Instructions (billions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPECs</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>600.perlbench_s</td>
<td>25</td>
<td>13</td>
<td>2696</td>
</tr>
<tr>
<td>602.gcc_s</td>
<td>15</td>
<td>5</td>
<td>7226</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>21</td>
<td>11</td>
<td>1775</td>
</tr>
<tr>
<td>620.omnetpp_s</td>
<td>8</td>
<td>5</td>
<td>1102</td>
</tr>
<tr>
<td>625.x264_s</td>
<td>14</td>
<td>9</td>
<td>12546</td>
</tr>
<tr>
<td>631.deepsjeng_s</td>
<td>20</td>
<td>16</td>
<td>2250</td>
</tr>
<tr>
<td>641.leela_s</td>
<td>29</td>
<td>21</td>
<td>2245</td>
</tr>
<tr>
<td>648.exchange2_s</td>
<td>19</td>
<td>15</td>
<td>6643</td>
</tr>
<tr>
<td>657.xz_s</td>
<td>21</td>
<td>15</td>
<td>8264</td>
</tr>
<tr>
<td><strong>Int Average</strong></td>
<td><strong>19.1</strong></td>
<td><strong>12.22</strong></td>
<td><strong>4607</strong></td>
</tr>
</tbody>
</table>

|                  | SPECs              | Floating-point       |                         |
|                  | Float             | Integer              |                         |
| 603.bwaves_s     | 9                 | 3                    | 66395                   |
| 619.ibm_s        | 8                 | 4                    | 4416                    |
| 638.imagick_s    | 15                | 6                    | 66788                   |
| 644.nab_s        | 12                | 5                    | 13489                   |
| 649.fotonik3d_s  | 17                | 7                    | 4280                    |
| **FP Average**   | **12.2**          | **5**                | **24204**               |
### Multiple SimPoints

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Start Instructions(100 million)/Weight(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.peribench_s</td>
<td></td>
</tr>
<tr>
<td>602.gcc_s</td>
<td></td>
</tr>
<tr>
<td>605.mcf_s</td>
<td></td>
</tr>
<tr>
<td>620.omnetpp_s</td>
<td></td>
</tr>
<tr>
<td>625.x264_s</td>
<td></td>
</tr>
<tr>
<td>631.deepsjeng_s</td>
<td></td>
</tr>
<tr>
<td>641.leela_s</td>
<td></td>
</tr>
<tr>
<td>648.exchange2_s</td>
<td></td>
</tr>
<tr>
<td>657.xz_s</td>
<td></td>
</tr>
<tr>
<td>603.bwaves_s</td>
<td></td>
</tr>
<tr>
<td>619.lbm_s</td>
<td></td>
</tr>
<tr>
<td>638.imagick_s</td>
<td></td>
</tr>
<tr>
<td>644.nab_s</td>
<td></td>
</tr>
<tr>
<td>649.fotonik3d_s</td>
<td></td>
</tr>
</tbody>
</table>
How to read the Table of Multiple SimPoints

<table>
<thead>
<tr>
<th>SimPoint</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.perlbench_s</td>
<td>(214681/28.85), (924424/12.60),</td>
</tr>
<tr>
<td></td>
<td>(789265/2.19), (1258340/2.06),</td>
</tr>
<tr>
<td></td>
<td>(236842/0.54), (1339730/0.53),</td>
</tr>
<tr>
<td>602.gcc_s</td>
<td>(23077/48.65), (35817/23.62),</td>
</tr>
<tr>
<td></td>
<td>(36933/0.44), (27933/0.36),</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>(5943/21.24), (16201/12.54),</td>
</tr>
<tr>
<td></td>
<td>(9255/2.25), (7606/2.12),</td>
</tr>
</tbody>
</table>
In many cases, the dominant SimPoint has less than 10% weight

Proxies allow us to make a single proxy for the whole benchmark

Or

One proxy per SimPoint, giving more accuracy than the dominant SimPoint
Industry Use

IBM – 2006

AMD – 2010

Intel 2017-2019
Miniature Benchmarks for RTL Model Validations

Improving Accuracy of Proxies

- Current Challenges
  - Major Inaccuracies
  - Branch Behavior Capture
  - Memory Behavior Capture
Improving Branch Behavior

- Branch Profiling has become more precise in order to increase accuracy
  - Initially Global Transition Rate [1]
  - Added Global Taken Rate
  - Added Per Branch (local) Transition and Taken Rate
  - Added inter Branch Correlation [2]
  - Adding Variable inter Branch Correlation

Proxy Branch Generation – Overview

- Branch mispredictions reduce overall IPC
  - Speculative work must be discarded
- Proxies should have branch predictor performance similar to original
- Metric of Interest: MPKI
  - Misses per thousand branch instructions
Profiling Branch Behavior

- Profile local behavior, i.e. per-static-branch
- Maintain a data structure for each static branch in the workload
- Every time a branch is encountered:
  - Increment the “encountered” count for the branch
  - If taken: increment the “taken” count for the branch
  - Compare taken behavior with the last time it was encountered
    - If different: increment the “transition” count for the branch
- Write raw results to file
- Discard data from extremely rare branches
  - 10% of static branches can make up 99.99% of dynamic branches
Proxy Branch Generation

- Use original static branches as “templates” for proxy branches
  - Each proxy branch matches the taken and transition rates of its template

- How many branches should use each template?
  - Each proxy branch is dynamically executed the same number of times
  - Particularly common original branches should be used for multiple proxy branches
  - Scaling step automatically performed
Limitations of Existing Branch Model

- **High degree of randomness in branches**
  - Makes prediction harder, resulting in too-high MPKI for some workloads

- **Disabling randomization has the opposite problem**
  - MPKI too low for some workloads

- **What’s causing this?**
  - Modern branch predictors examine interbranch relations
    - Branch history registers
    - Proxy branch generator only looks at branches individually
Improving Memory Access Patterns

- The proxies in the repository use a stride based memory model that captures a per instruction memory pattern
  - Example: a load instruction accesses x3000, then x3008, x3010, x3018, ....

- Recent research (HALO) has found better accuracy through modeling[1]:
  - Intra-region stride locality
  - Inter-region reuse locality

- Currently developing proxies that implements these two techniques into the proxy

HALO Memory Behavior

• For intra-region locality, a Markov chain is created for each region of memory to track patterns within.

• The probability of switching between regions is calculated for the inter-region reuse locality.

• HALO combines the intra-region locality and inter-region reuse distance to create similar memory traces.
HALO Memory Behavior

- Focuses on two key components

Figure 4: Intra-region locality profiling using cascaded stride tables (CSTs).
Reproducibility

- Reproducibility of Research using Proxies
  - Miniaturization Standpoint
  - Software Stack Abstraction Standpoint
  - Proprietary Codes Standpoint
  - Simulator Validation Standpoint

- Would like the research community to use the proxies along with other benchmarks
  - SPEC CPU 2017 Proxies
  - Data base (Cassandra, MongoDB, MySQL) Proxies
  - Github
ISA Independent Synthetic Benchmark Generation

- **Step 1: Generate synthetic code in LLVM IR**
  - Use Static Single Assigned (SSA) form
  - Not bounded to specific ISA
  - Targeting total number of dynamic instructions to desired value

- **Step 2: Create assembly for target ISAs**
  - Use modified LLVM backend
  - Disable optimization paths to preserve characteristics

- **Step 3: Compile binary for target ISAs**
  - Use general compiler for each ISAs
  - Does not require specialized tools
ISA Independent Synthetic Benchmarks

- **LLVM compiler infrastructure**
  - A virtual instruction set language
  - Various back-end support – x86, PowerPC, ARM, Alpha, SPARC, etc.

- **Use virtual ISA in SSA form**
  - Micro architecture independent
  - Avoid machine-specific optimizations in the front-end
  - Back-end takes care of machine-dependent optimizations

```c
#include <stdio.h>

int main()
{
    int i;
    int q,r,t, x[100000], y[100010], z[100011];
    for (i=0;i<100000;i=i+1)
    {
        x[i]= q + y[i] * (z[i+10] + t * z[i+11]);
        printf("%d\n",x[i]);
    }
    return 0;
}
```
ISA Independent Synthetic Benchmarks

### LLVM ISA

- **x86**
  - Inst. Size: 45 (18)
  - Cycle Count: 80.7M
  - IPC: 1.18

- **Sparc**
  - Inst. Size: 93 (55)
  - Cycle Count: 50.8M
  - IPC: 1.08

- **Alpha**
  - Inst. Size: 100(44)
  - Cycle Count: 107.4M
  - IPC: 1.48

---

### Code Examples

**LLVM ISA**

```c
; movl 12(%esp), %eax
; movl $00012 (%esp,%eax,4), %ecx
; imull 24(%esp), %ecx
; movl $000108 (%esp,%eax,4), %edx
; imull 20(%esp), %edx
; addl %ecx, %edx
```

**Alpha**

```c
 /* addq $0,$3,$3 */
mulq $1,$4,$1
mulq $2,$5,$2
lda $4,20($30)
ldl $3,0($0)
addq $2,$1,$1
ldl $2,0($4)
mulq $3,$1,$1
lda $3,32($30)
addq $1,$2,$1
s4addq $0,$3,$0
stl $1,0($0)
ldl $0,0($11)
s4addq $0,$3,$0
```

**Sparc**

```c
/* add $1,$2,$1 */
sml $11, $14, $11
sml $12, $13, $12
ld [%15+10], $13
add $12, $11, $11
sethi 4193522, $g1
add $g1, $16, $g1
add $g1, 680, $15
smul $11, $14, $11
smul $12, $13, $12
ld [%g1+28], $12
smul $13, $11, $11
sethi 4193132, $g1
add $g1, $16, $g1
add $g1, 40, $13
add $11, $12, $11
st $11, [%13+10]
sethi 4193132, $g1
add $g1, $16, $g1
```

---

Lizy K. John, LCA, UT Austin
Other Applications

- Generate Clones by setting knobs to appropriate values
- Adaptable
- Scalable
- Futuristic

![Diagram showing the Proxy Benchmark Generation Framework with application behavior space, workload characteristics, 'knobs' for changing program characteristics, workload synthesis algorithm, benchmark synthesizer, synthetic benchmark, and compile and execute steps.](image)
Synthetic Benchmarks

- Performance and Power Clones for RTL
- Stress Benchmarks
- Clones for Proprietary Applications
- Benchmarks with Knobs Adjustable Benchmarks
- Scalable Benchmarks
- Futuristic Benchmarks
Sharing Proprietary Applications

- Military Applications cannot be shared with Vendors
- Many commercial applications cannot be shared freely between software and hardware developers

Proprietary Applications

Performance/Power Clones
Early Performance Testing

- **Usage Scenario**
  - Derive proxy applications from a set of workload characterizations
  - Proxies convey no proprietary information, but capture the execution behavior of XX’s applications
  - Proxy applications can be available very early in project
  - Proxies focus on the computationally significant portions of code, not the small parts that are hardware-dependent (e.g., DMA, etc.)

- Early testing confirms performance, or discovers shortfalls in time to address them before final integration
Proxy Applications can Transform Procurement Process

- Proxy applications can turn vendors into partners in achieving performance goals
  - Provides them with means to directly measure performance
  - Places the iterative test/analyze process in vendor’s labs
  - Reveals no proprietary information

- Turn hardware performance tuning into a supplier task
  - Periodically capture application performance in a new set of proxies
  - Task supplier to re-tune hardware for best performance as applications mature
POWER AMD THERMAL STRESSMARKS

- Hand crafting code snippets for power viruses
  - Very tedious process, complex interactions inside the processor
  - Cannot be sure if it is the maximum case
  - Heavily architecture dependent; heavy domain knowledge

- We automatically generate power viruses

<table>
<thead>
<tr>
<th>Power Virus</th>
<th>Optimized to stress</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPrime</td>
<td>All CPUs, all ISAs</td>
<td>C</td>
</tr>
<tr>
<td>CPUburn-in</td>
<td>X86 machines</td>
<td>x86 assembly</td>
</tr>
<tr>
<td>BurnP5</td>
<td>Intel Pentium w&amp;w/o MMX processors</td>
<td>x86 assembly</td>
</tr>
<tr>
<td>BurnP6</td>
<td>Intel PentiumPro, Pentium II, Pentium III and Celeron CPUs</td>
<td>x86 assembly</td>
</tr>
<tr>
<td>BurnK6</td>
<td>AMD K6 processors</td>
<td>x86 assembly</td>
</tr>
<tr>
<td>BurnK7</td>
<td>AMD Athlon/Duron processors</td>
<td>x86 assembly</td>
</tr>
<tr>
<td>BurnMMX</td>
<td>cache/memory interfaces on all CPUs with MMX</td>
<td>x86 assembly</td>
</tr>
</tbody>
</table>
POWER and THERMAL STRESSMARKS

- Automatically search for power viruses using an abstract workload model and machine learning

- GA: search heuristic to solve optimization problems
  - Choose a random population, evaluate fitness, apply GA operators to generate next population
  - Evolve until required fitness achieved
Power measurement of Viruses on Hardware

- **BurnK7** – 72.1 Watts
- **SPEC CPU2006**: 416.gamess and 453.povray consume highest power of 63.1 and 59.6 Watts
SYMPO Framework – Genetic Algorithm

Individuals -> synthetic workloads, Fitness function -> power on the design under study

Mutation rate, reproduction rate, crossover rate

Single-point Crossover

Single-point Mutation
Our code generator was not equipped to generate code for x86 at that time

- Generated power viruses on SPARC ISA and translated to x86 using LLVM infrastructure
A Solution to The Expiring Benchmark Problem

- SPEC89, SPEC92, SPEC95, SPEC CPU2000, CPU 2006
- TPC A, TPC B, TPC D
- Benchmarks become obsolete very quickly
- Benchmark makers create inflated data sets to stay alive at least 4-5 years
- Generate futuristic benchmarks using the proxy code generator
Interactive Workloads??

High-level feature categories need to be modified for interactive properties

1. Instruction-level Characteristics
2. Control-flow Characteristics
3. Memory-access Characteristics
4. Interactive Workload Characteristics
Weaknesses – The Vitamin Tablet Problem

Vitamin A, B, C, D, E, K

What about undiscovered vitamins? Vitamin P, Q, R?
Conclusion

• We have developed a proxy generation methodology to generate miniature proxy benchmarks.

• PerfProx proxies can enable fast and efficient performance evaluation of emerging workloads without needing back-end database or complex software stack support.

• Significant amounts of tuning required to get accurate proxies now.

• Automating the process will be helpful for deployment of our proxy generation tools (Intel project objective).
References


References


References


References

- Reena Panda, Ph. Dissertation, UT Austin 2017, Accurate Modeling of Core and Memory Locality for Proxy Generation Targeting Emerging Applications and Architectures
BACK UP
Branch bit vectors

- Need a lightweight way to specify taken/transition rates of a branch
  - Minimize computational overhead to avoid skewing results
- Solution: Compare a 1-hot shift register against a static 32 bit vector
  - High transition rate branches use a register that shifts every iteration
  - Low transition rate branches use a register that shifts every 32 iterations
  - Branch if zero flag not set (both vectors had a 1 in the same position)

Examples (using short vectors):
- 11001100: 50% taken rate, 50% transition rate (1.6% with slow shift register)
- 11101110: 75% taken rate, 50% transition rate
- 10101010: 50% taken rate, 100% transition rate
- 10000000: 12.5% taken rate, 25% transition rate
Generation Process

- Generate vector with target number of transitions
  - Roughly equal-sized groups of 0s and 1s, roughly 50% taken rate

- Flip bits to match target taken rate, without impacting transition rate
  - $V[n]$ can be flipped if $V[n-1] \neq V[n+1]$
  - Example: 11110000 and 11100000 have the same transition rate

- Randomize the vector
  - Perform a random rotation
  - Shuffle groups of bits without impacting transition rate
    - Example: 11011010 and 11010110 have the same transition rate
COMPARISON WITH STANDARD BENCHMARKS

- Cassandra-ORIG
- Cassandra-Proxy
- SPEC-CPU2006
- SPECJBB2013
- MySQL-ORIG
- MySQL-Proxy
- Linpack
To add:

Which 5 are good from Intel

Ajay’s reuse distance definition slides

SPARC MAMPO results

MAMPO cite

IISWC 2006 – proprietary reference

SYMPO reference

Find other efforts

Incorporate Steven’s results – Intel results?

3 steps – Intel confidential??
## Branch Errors

<table>
<thead>
<tr>
<th>Workload</th>
<th>Global Transition and Taken Rate (v3)</th>
<th>Local Transitions and Taken Rate (v4)</th>
<th>Local info with complete correlation (bv4)</th>
<th>Local info with half correlation (bv4b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>4518%</td>
<td>5242%</td>
<td>40%</td>
<td>1203%</td>
</tr>
<tr>
<td>perlbench</td>
<td>805%</td>
<td>-49%</td>
<td>-75%</td>
<td>256%</td>
</tr>
<tr>
<td>mcf</td>
<td>-100%</td>
<td>-100%</td>
<td>-99%</td>
<td>-99%</td>
</tr>
<tr>
<td>cactus</td>
<td>364%</td>
<td>49%</td>
<td>-98%</td>
<td>-95%</td>
</tr>
<tr>
<td>povray</td>
<td>1675%</td>
<td>1330%</td>
<td>-16%</td>
<td>49%</td>
</tr>
<tr>
<td>absolute average</td>
<td>(no gcc)</td>
<td>(no gcc)</td>
<td>66%</td>
<td>263%</td>
</tr>
</tbody>
</table>


Industry Use

IBM – 2006

AMD – 2010

Motivating example where proxies were tuned for one platform then transferred to another platform

On Platform 1 10/10 proxies <13% IPC error versus complete benchmark

On Platform 2 5/10 proxies <13% IPC error, 5/10 proxies >37% IPC error