Rapidly Selecting Good Compiler Optimizations Using Performance Counters

John Cavazos\textsuperscript{1} Grigori Fursin\textsuperscript{2}
Felix Agakov\textsuperscript{1} Edwin Bonilla\textsuperscript{1}
Michael O'Boyle\textsuperscript{1} Olivier Temam\textsuperscript{2}

\textsuperscript{1}University of Edinburgh, UK
\textsuperscript{2}INRIA Futurs, France
Members of HiPEAC
Traditional Compilers

- “One size fits all” approach
- Tuned for average performance
- Aggressive opts often turned off
- Need to “understand” all layers below
  - Hard to model analytically
Solution

- Use performance counter characterization
  - Train model off-line
  - Counter values are “features” of program
  - Out-performs highest optimization setting in production quality compiler
  - 2 orders of magnitude faster than pure search
Performance Counters

- 60 counters available
- 5 categories
  - Floating point, Branch, L1 cache, L2 cache, TLB, Others
- Examples:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Avg Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU_IDL</td>
<td>(Floating Unit Idle)</td>
<td>0.473</td>
</tr>
<tr>
<td>VEC_INS</td>
<td>(Vector Instructions)</td>
<td>0.017</td>
</tr>
<tr>
<td>BR_INS</td>
<td>(Branch Instructions)</td>
<td>0.047</td>
</tr>
<tr>
<td>L1_ICH</td>
<td>(L1 Icache Hits)</td>
<td>0.0006</td>
</tr>
</tbody>
</table>
Characterization of SPEC FP
Characterization of SPEC FP

Larger number of L1 icache misses, L1 store misses and L2 D-cache writes
Characterization of 181.mcf

Problem: Greater number of memory accesses per instruction than average
Characterization of 181.mcf

Problem: BUT also Branch Instructions
Characterization of 181.mcf

Use LNO (loop nest optimizations)
Reduce total/branch instructions and L1 I-cache/D-cache accesses.
Characterization of 181.mcf

Model applies -m32 (32 bit pointers)
Reduces L1 cache misses which reduces L2 cache accesses.
Putting Perf Counters to Use

- Important aspects of programs captured with performance counters
- Automatically construct model (PC Model)
  - Map performance counters to good opts
- Model predicts optimizations to apply
  - Uses performance counter characterization
Training PC Model

- Programs (training set) → best set of transformations (option sequences)
  \[ (t_1^1, t_2^1, \ldots, t_M^1), (t_1^2, t_2^2, \ldots, t_M^2), (t_1^N, t_2^N, \ldots, t_M^N) \]

- \( T_B \) (baseline option)

- Compiler and Architecture

- Performance counter features for the baseline \( X \)

- Speedups:
  - \( s_1 \)
  - \( s_2 \)
  - \( s_N \)

- PCMModel
Training PC Model

Programs to train model (different from test program).
Baseline runs to capture performance counter values.
Obtain performance counter values for a benchmark.
Training PC Model

Best optimizations runs to get speedup values.
Training PC Model

Best optimizations runs to get speedup values.
New program interested in obtaining good performance.
Using PC Model

Baseline run to capture performance counter values.
Using PC Model

Feed performance counter values to model.
Model outputs a distribution which we generate sequences from.
Using PC Model

Optimization sequences drawn from distribution.
PC Model

- Trained on data from Random Search
  - 500 evaluations for each benchmark
- Leave-one-out cross validation
  - Training on N-1 benchmarks
  - Test on Nth benchmark
- Logistic Regression
Logistic Regression

- Variation of ordinary regression
- Inputs
  - Continuous, discrete, or a mix
  - 60 performance counters
    - All normalized to cycles executed
- Outputs
  - Restricted to two values \((0,1)\)
  - Probability an optimization is beneficial
Experimental Methodology

- PathScale compiler
  - Compare to highest optimization level
  - 121 compiler flags
- AMD Athlon processor
  - *Real* machine; Not simulation
- 57 benchmarks
  - SPEC (INT 95, INT/FP 2000), MiBench, Polyhedral
Results

► Combined Elimination and PC Model
► Performance versus Evaluations
► Most Informative Performance Counters
Evaluate Search Strategies

- PC Model
- RAND
  - Randomly select 500 optimization seqs
- Combined Elimination [CGO 2006]
  - Pure search technique
    - Evaluate optimizations one at a time
    - Eliminate negative optimizations in one go
  - Out-performed other pure search techniques
Combined Elimination (CE) versus PC Model

Relative to ofast

ac  air  bitcount  susan_c  susan_e  susan_s  channel  jpeg_c  jpeg_d  lame  dodec  drag  fatigue  gas_dyn  linpk  mdbx  dijkstra  patricia  stringsearch  pix  protein  mflow  rijndael  sha  adpcm_c  cRC32  gsm  test_fpu  tiff  average
1. 9 benchmarks over 20% improvement and 17% on average!
2. CE uses 607 iterations (240-1550) and PC Model 25 iterations.
Combined Elimination (CE) and PC Model

Relative to ofast

1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0

average
301.apsi
300.twolf
256.bzip2
197.parser
191.fma3d
189.lucas
188.ammp
186.crafty
183.equate
179.art
178.galgel
175.vpr
173.appiu
172.mgrid
171.swim
168.wupwise
164.gzip
146.wave5
141.apsi
125.turb3d
107.mgrid
104.hydro2d
103.su2cor
102.swim
101.tomcatv

1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9

PC Model
CE
1. Obtain over 25% improvement on 7 benchmarks!
2. On average, CE obtains 9% and PC Model 17% over -ofast!
Performance vs Evaluations

Performance versus Number of Evaluations (PC Model, CE, RAND)

- Speedup
- Number of Evaluations

Graph comparing the speedup of different evaluation models (PC Model, CE, RAND) with the number of evaluations.
Performance vs Evaluations

- Random (17%)
- Combined Elim (12%)
- PC Model (17%)
Why is CE worse than RAND?

► Combined Elimination
  ► Dependent on dimensions of space
  ► Easily stuck in local minima

► RAND
  ► Probabilistic technique
  ► Depends on distribution of good points
  ► Not susceptible to local minima

Note: CE would perform better where many opts degrade performance.
## Most Informative Features

<table>
<thead>
<tr>
<th>Most Informative Performance Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. L1 Cache Accesses</td>
</tr>
<tr>
<td>2. L1 Dcache Hits</td>
</tr>
<tr>
<td>3. TLB Data Misses</td>
</tr>
<tr>
<td>4. Branch Instructions</td>
</tr>
<tr>
<td>5. Resource Stalls</td>
</tr>
<tr>
<td>6. Total Cycles</td>
</tr>
<tr>
<td>7. L2 Icache Hits</td>
</tr>
<tr>
<td>8. Vector Instructions</td>
</tr>
<tr>
<td>9. L2 Dcache Hits</td>
</tr>
<tr>
<td>10. L2 Cache Accesses</td>
</tr>
<tr>
<td>11. L1 Dcache Accesses</td>
</tr>
<tr>
<td>12. Hardware Interrupts</td>
</tr>
<tr>
<td>13. L2 Cache Hits</td>
</tr>
<tr>
<td>14. L1 Cache Hits</td>
</tr>
<tr>
<td>15. Branch Misses</td>
</tr>
</tbody>
</table>
Conclusions

► Use performance counters to find good optimization settings
► Out-performs production compiler in few evaluations (+ 3 for counters)
► 2 orders of magnitude faster than best known pure search technique