Intelligent Compilation

John Cavazos

Department of Computer and Information Sciences
University of Delaware
Proposition: “The focus on specialized tuning systems is too narrow …”

Specialized tuning systems are compiler component

- Code Analyzer
  - Dense Matrix Optimizer
  - Simple Code Generation
Proposition: “The focus on specialized tuning systems is too narrow …”

Specialized tuning systems are compiler component

- Code Analyzer
  - Dense Matrix Matrix Optimizer
  - Sparse Matrix Matrix Optimizer
  - Simple Code Generation

Dept. of Computer and Information Sciences : University of Delaware
Proposition: “The focus on specialized tuning systems is too narrow …”

Specialized tuning systems are compiler component
Proposition: “The focus on specialized tuning systems is too narrow …”

Specialized tuning systems are compiler component
Proposition: “The focus on specialized tuning systems is too narrow …”

Specialized tuning systems are compiler component
Traditional Compilers

► “One size fits all” approach
► Tuned for average performance
► Aggressive opts often turned off
► Target hard to model analytically

Applications

Compilers

Operating System/Virtualiz’n

Hardware
Proposed Solution

- Intelligent Compilers
  - Use Machine Learning
  - Learn to optimize
    - Specialized to each Application/Data/Hardware

Diagram:

- Applications
- Intelligent Compilers (Neural Networks, Decision Trees, Genetic Algorithms, etc.)
- Operating System/Virtualiz’n
- Hardware

Feedback
Intelligence in a compiler

- Global
  - Controlling compiler flags [CGO 2007]

- Local
  - Individual methods [OOPSLA 2006]
  - Individual loop bodies [PLDI 2008]

- Individual Optimization Heuristic
  - How and When to Perform Instruction Scheduling [NIPS 1997, PLDI 2005]

http://www.cis.udel.edu/~cavazos
Overall Approach

► Training of Model
  ▶ Generate training data
  ▶ Automatically construct a heuristic
  ▶ Can be expensive, but can be done offline

► Testing of Model
  ▶ During Compilation
    ▶ Extract features
      ▶ Model outputs probability distribution
      ▶ Generate optimizations from distribution

► Offline versus online learning
Outline

► Using Performance Counters
► Intelligent Polyhedral Search
► Method-Specific Compilation
Putting Perf Counters to Use

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

- Many performance counters available

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 181.mcf

Perf cntrs relative to 4 benchmark suites

Dept. of Computer and Information Sciences : University of Delaware
Characterization of 181.mcf

- Perf cntrs relative to 4 benchmark suites

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

- Using -Ofast and search with a Model.
Characterization of 181.mcf

- Using -Ofast and search with a Model.

Both reduce total instructions and branch instructions and L1 I-cache and D-cache accesses.
Characterization of 181.mcf

- Using -Ofast and search with a Model.

Model also reduces L1 total cache misses which reduces L2 cache access by same amount.
Training PC Model

Dept. of Computer and Information Sciences : University of Delaware
Traning PC Model

Programs to train model (different from test program).

Dept. of Computer and Information Sciences : University of Delaware
Baseline runs to capture performance counter values.
Obtain performance counter values for a benchmark.
Best optimizations runs to get speedup values.
Best optimizations runs to get speedup values.
New program interested in obtaining good performance.
Baseline run to capture performance counter values.
Feed performance counter values to model.
Model outputs a distribution that is used to generate sequences.
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
PathScale compiler
- Compare to highest optimization level
- 121 compiler flags

AMD Athlon processor
- *Real* machine; Not simulation

57 benchmarks
- SPEC (INT 95, 2000), MiBench, Polyhedral
Evaluated Search Strategies

- 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
- PC Model
Obtained > 25% on 7 benchmarks and 17% over highest opt.
Two Additional Approaches

- Intelligent Polyhedral Search [PLDI 2008]
- Method-Specific Compilation [OOPSLA 2006]
Intelligent Polyhedral Results

Performance improvement for AMD Athlon64

Relative to gcc -O3 -ftree-vectorize -msse2
Method-Specific Compilation

- Integrate Machine Learning into a Java JIT compiler
- Use simple code properties
  - Extracted from one linear pass of bytecodes
- Model controls up to 20 optimizations
- Outperforms hand-tuned heuristic
  - Up to 29% SPEC JVM98
  - Up to 33% DaCapo+
Conclusions

- Using **performance counters**
  - Out-performs production compiler in few evaluations

- **Intelligently traverses** Polyhedral Space

- Using **code characteristics**
  - Can outperform hand-tuned heuristic
  - Opts applied only when beneficial
Static vs Dynamic Features

- Static vs Dynamic Features

Relative to ofast

1.05
1.1
1.15
1.2
1.25
1.3
1.35

164 gzip
175 vpr
181 mcf
186 crafty
197 parser
256 bzip2
300 twolf
sieve
**Most Informative Performance Counters**

<table>
<thead>
<tr>
<th>#</th>
<th>Most Informative Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1 Cache Accesses</td>
</tr>
<tr>
<td>2</td>
<td>L1 Dcache Hits</td>
</tr>
<tr>
<td>3</td>
<td>TLB Data Misses</td>
</tr>
<tr>
<td>4</td>
<td>Branch Instructions</td>
</tr>
<tr>
<td>5</td>
<td>Resource Stalls</td>
</tr>
<tr>
<td>6</td>
<td>Total Cycles</td>
</tr>
<tr>
<td>7</td>
<td>L2 Icache Hits</td>
</tr>
<tr>
<td>8</td>
<td>Vector Instructions</td>
</tr>
<tr>
<td>9</td>
<td>L2 Dcache Hits</td>
</tr>
<tr>
<td>10</td>
<td>L2 Cache Accesses</td>
</tr>
<tr>
<td>11</td>
<td>L1 Dcache Accesses</td>
</tr>
<tr>
<td>12</td>
<td>Hardware Interrupts</td>
</tr>
<tr>
<td>13</td>
<td>L2 Cache Hits</td>
</tr>
<tr>
<td>14</td>
<td>L1 Cache Hits</td>
</tr>
<tr>
<td>15</td>
<td>Branch Misses</td>
</tr>
</tbody>
</table>
**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 may improve in space with many bad opts.
Characterizing large programs hard

Performance counters effectively summarize program's dynamic behavior

Previously* used static features [CGO 2006]

Does not work for whole program characterization