Intelligent Compilation

John Cavazos

Department of Computer and Information Sciences
University of Delaware
Proposition: Autotuning is a component of an Intelligent Compiler.
Proposition: Autotuning is a component of an Intelligent Compiler.

- Code Analyzer
  - Dense Matrix Optimizer (ATLAS)
  - Sparse Matrix Optimizer (OSKI)
  - Simple Code Generation

Dept. of Computer and Information Sciences : University of Delaware
Proposition: Autotuning is a component of an Intelligent Compiler.

- Code Analyzer
  - Dense Matrix Optimizer (ATLAS)
  - Sparse Matrix Optimizer (OSKI)
  - Another “Berkeley Dwarf” Optimizer

- Simple Code Generation

Dept. of Computer and Information Sciences : University of Delaware
Proposition: Autotuning is a component of an Intelligent Compiler.
Proposition: Autotuning is a component of an Intelligent Compiler.
Proposition: Autotuning is a component of an Intelligent Compiler.
Traditional Compilers

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

- *Intelligent Compilers*
  - Use machine learning
  - Learn to optimize
    - Specialized to each Application/Data/Hardware

```
Applications
Intelligent Compiler (Statistical Machine Learning)
Operating System/Virtualiz’n
Hardware
```

Feedback

Dept. of Computer and Information Sciences : University of Delaware
We want intelligent, robust, adaptive behaviour in compilers.

Often hand programming very difficult

Get the compiler to program itself, by showing it examples of behaviour we want.

This is the machine learning approach!

We write the structure of the compiler and it then tunes many internal parameters.
Intelligence in a compiler

- Individual optimization heuristic
  - Instruction scheduling [NIPS 1997, PLDI 2005]
- Whole-program optimizations [CGO '06 / '07]
- Individual methods [OOPSLA 2006]
- Individual loop bodies [PLDI 2008]

http://www.cis.udel.edu/~cavazos
How to use Machine Learning

- Phrase as machine learning problem
- Determine inputs/outputs of ML model
  - Important characteristics of problem (features)
  - Target function
- Generate training data
- Train and test model
  - Learning algorithms may require “tweaking”
Train and Test Model

Training of model
- Generate training data
- Automatically construct a model
- Can be expensive, but can be done offline

Testing of model
- Extract features
- Model outputs probability distribution
- Generate optimizations from distribution

Offline versus online learning
Case Studies

- Whole Program Optimization
- Individual Method Optimization
Putting Perf Counters to Use

- **Model Input**
  - Aspects of programs captured with perf. counters

- **Model Output**
  - Set of optimizations to apply
  - Automatically construct model (Offline)
    - Map performance counters to good opts
  - Model predicts optimizations to apply
    - Uses performance counter characterization
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 several benchmarks
Characterization of 181.mcf

- Perf cntrs relative to several benchmarks

Problem: Greater number of memory accesses per instruction than average

Dept. of Computer and Information Sciences : University of Delaware
Programs to train model (different from test program).
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
Experimental Methodology

- PathScale industrial-strength compiler
  - Compare to highest optimization level
  - Control 121 compiler flags
- AMD Athlon processor
  - *Real* machine; Not simulation
- 57 benchmarks
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
PCModel/CE (SPEC INT 95/SPEC 2000)

Obtained > 25% on 7 benchmarks and 17% over highest opt.
Case Studies

- Whole Program Optimization
- Individual Method Optimization
Method-Specific Compilation

- Integrate machine learning into 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+
Overall Approach

► Phase 1: Training
  ► Generate training data
  ► Construct a heuristic
  ► Expensive offline process

► Phase 2: Deployment
  ► During Compilation
    ► Extract code features
    ► Heuristic predicts optimizations
Generate Training Data

- For each method
  - Evaluate many opt settings
  - Fine-grained timers
    - Record running time
    - Record compilation time
- For optimization level O2
  - Evaluate 1000 random settings
- One model for the optimization level
Training example for each method

- Inputs - Features of method
- Outputs - Good optimization setting

### Training examples

<table>
<thead>
<tr>
<th>methods</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>108;25;0;0; ... ;08;0;</td>
<td>1;0;1;1; ... 1;1;1;0</td>
</tr>
<tr>
<td>bar</td>
<td>93;21;0;1; ... ;50;0;</td>
<td>1;1;0;0; ... 1;0;0;0</td>
</tr>
<tr>
<td>...</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>...</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>Method Features</td>
<td>Meaning</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Number of <strong>bytecodes</strong></td>
<td></td>
</tr>
<tr>
<td>Locals Space</td>
<td>Words allocated for <strong>locals space</strong></td>
<td></td>
</tr>
<tr>
<td>Characteristics</td>
<td>Is <strong>synchronized</strong>, has <strong>exceptions</strong>, is <strong>leaf</strong> method</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>Is it declared <strong>final, static, private</strong></td>
<td></td>
</tr>
<tr>
<td>Fraction of Bytecodes</td>
<td>Has array <strong>loads</strong> and <strong>stores</strong> primitive and long computations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compares, <strong>branches</strong>, jsrs, switches, put, get, invoke, new, arraylength</td>
<td></td>
</tr>
<tr>
<td></td>
<td>athrow, checkcast, monitor</td>
<td></td>
</tr>
</tbody>
</table>

Note: **26** features used to describe method

Dept. of Computer and Information Sciences : University of Delaware
<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Optimizations Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt Level O0</td>
<td>Branch Opts Low</td>
</tr>
<tr>
<td></td>
<td><strong>Constant Prop</strong> / Local CSE</td>
</tr>
<tr>
<td></td>
<td>Reorder Code</td>
</tr>
<tr>
<td>Opt Level O1</td>
<td><strong>Copy Prop</strong> / Tail Recursion</td>
</tr>
<tr>
<td></td>
<td>Static Splitting / Branch Opt Med</td>
</tr>
<tr>
<td></td>
<td><strong>Simple Opts Low</strong></td>
</tr>
<tr>
<td>Opt Level O2</td>
<td>While into Untils / <strong>Loop Unroll</strong></td>
</tr>
<tr>
<td></td>
<td>Branch Opt High / Redundant BR</td>
</tr>
<tr>
<td></td>
<td><strong>Load Elim</strong> / Load Elim</td>
</tr>
<tr>
<td></td>
<td>Simple Opts Med / <strong>Load Elim</strong></td>
</tr>
<tr>
<td></td>
<td>Expression Fold / Coalesce</td>
</tr>
<tr>
<td></td>
<td>Global Copy Prop / Global CSE</td>
</tr>
<tr>
<td></td>
<td>SSA</td>
</tr>
</tbody>
</table>
Method bytecodes → Jikes RVM

- Compiler Heuristic
- Optimizer

Optimized method →

Feature extractor → Logistic regression model

Feature Vector:
{108;25;0;0;0;1;0;0;2;0;0;0;0;0;0;0;0;0;0;12;0;0;0;0;08;0;0;0;0;0;0;0;2;0;32;0;08;0;0}
The image depicts a diagram of a compiler heuristic optimizer named Jikes RVM. The process involves method bytecodes as input. These bytecodes are processed by a feature extractor, which generates a feature vector as output. The feature vector is then fed into a logistic regression model. The output of the optimizer is an optimized method. The diagram includes a feature vector represented as `{108;25;0;0;0;0;1;0;0:2;0:0;0;0:0;0;0;0;0;0:12;0:0;0:08;0:0;0:0;0:0;0:2;0:32;0:08;0:0}`.
Compiler Heuristic (online)

Jikes RVM

- Compiler Heuristic
- Optimizer

Method bytecodes

Feature extractor

Feature Vector

Logistic regression model

Opt Flags

Optimized method

{108;25;0;0;0;1;0;0;2;0;0;0;0;0;0;0;0;0;0;12;0;0;0;0;0;0;0;0;0;0;0;2;0;32;0;0;6;0;0}

{1;0;1;1;0;0;0;1;1;1;1;1;1;1;1;1;0;1;1;1;0;1;1;1;0}
DaCapo+ (Highest Opt Level)

Opt Level O2

Running  Total

fop  jython  pmd  ps  antlr  pseudojbb  ipsixql  geo-mean

Dept. of Computer and Information Sciences : University of Delaware
Challenges Remaining

- Single-core optimizations still important
  - Optimization phase-ordering
  - Optimization for program phases
  - Speculative optimizations

- Parallel optimizations
  - Task partitioning
  - Communication/computation overlap
  - Task scheduling/migration
  - Data placement/migration/replication
Conclusions

- Using **machine learning successful**
  - Out-performs production compiler in few evaluations
- Using **perf. counters/code characteristics**
  - Determines automatically what characteristics are important
  - Optimizations applied only when beneficial
3rd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART '09)

January 25, 2009, Paphos, Cyprus
(co-located with HIPEAC 2009 Conference)

Sponsored by:

MILEPOST

Program Chair:
David Padua
University of Illinois at Urbana-Champaign, USA

Organizers:
Grigori Fursin
INRIA Saclay, France
John Cavazos
University of Delaware, USA

The rapid rate of architectural change and the large diversity of architecture features has made it increasingly difficult for compiler writers to keep pace with microprocessor evolution. This problem has been compounded by the introduction of multicores. Thus, compiler writers have an intractably complex problem to solve. A similar situation arises in processor design where new approaches are needed to help computer architects make the best use of new underlying technologies and to design systems well adapted to future application domains.

Recent studies have shown the great potential of statistical machine learning and search strategies for compilation and machine design. The purpose of this workshop is to help consolidate and advance the state of the art in this emerging area of research. The workshop is a forum for the presentation of recent developments in compiler techniques and machine design methodologies based on space exploration and statistical machine learning approaches with the objective of improving performance, parallelism, scalability, and adaptability.

Topics of interest include (but are not limited to):
Machine Learning, Statistical Approaches, or Search applied to

Dept. of Computer and Information Sciences : University of Delaware