“Predictive Modeling Methodology for Compiler Phase-ordering”

Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo and Cristina Silvano
Outline

• Problem Description
  ○ Introduction And Motivation
  ○ Phase-ordering of compiler optimizations

• Proposed Methodology
  ○ Using Machine Learning to focus on iterative compilation
  ○ Application-Specific compiler tuning
  ○ Experimental results
  ○ Conclusion

• Ongoing and future research directions
Outline

• Problem Description
  ○ Introduction And Motivation
  ○ Phase-ordering of compiler optimizations

• Proposed Methodology
  ○ Using Machine Learning to focus on iterative compilation
  ○ Application-Specific compiler tuning
  ○ Experimental results
  ○ Conclusion

• Ongoing and future research directions
Introduction

Why Compiler Optimizations?! (1/2)

- Wide range of parallel architectures
  - ARM, INTEL X86_64, Qualcomm, AMD/ATI, GPUs

- Different goals in different domains:
  - Mobile and embedded systems
    - Power and area issues
  - Desktops and HPC domain
    - Performance

- Application (dataset/input) specific

- Different levels of optimizations
  - Os → code-size
  - Ox → code-size/exec_time
  - Ofast → exec_time, approximated_calc

Motivation
Why Compiler Optimizations?! (2/2)

- There are several pre-built compiler flags:
  - Standard flags, i.e. (O1, O2, O3, Os, etc)
  - Many different optimization passes
    - around 100 in LLVM
    - around 250 in GCC
    - around 75 in ICC, etc

- Applying the right set of compiler flags will give substantial benefits on:
  - Performance, code-size and power metrics

- On Embedded domain:
  - App compiled once and then deploys on million of market devices

Problem Description
Optimization Challenges

• Big exploration space \( \mathcal{O} \) considering \( n \) options and \( A \) apps:

  Problem 1): **Selection problem:**
  \[-2^n \text{ Optimizations to enable/disable: } O_{\text{selection}} = \{0, 1\}^n\]

  Problem 2): **Phase-ordering problem:**
  \[-\text{without repetition: } |O_{\text{phases\_noRep}}| = n!\]
  \[-\text{With repetition and max length } m: |O_{\text{phases\_repetition}}| = \sum_{i=0}^{m} n^i\]
  \[-\text{With repetition and no upper bound on length: Infinite answer}\]

• Multi-objective nonlinear optimization in NP-hard space
• Application/Platform/Compiler specific
Outline

• Problem Description
  ○ Introduction And Motivation
  ○ Phase-ordering of compiler optimizations

• Proposed Methodology
  ○ Using Machine Learning to focus on iterative compilation
  ○ Application-Specific compiler tuning
  ○ Experimental results
  ○ Conclusion

• Ongoing and future research directions
Compiler Auto-tuning Background (I)

[John Cavazos, Mike O’boyle et al, CGO’06 “Using machine learning to focus on iterative compilation”]

- State-of-the-art approaches tackle the problem using auto-tuning:
  - **Iterative compilation** (several re-compilation of the kernel using different optimization flags to choose the best found option)

**Downside:**
- High overhead and time-consuming task

*Can’t we do better?*
- Using heuristics, algorithms and statistical methods
State-of-the-art approaches tackle the problem using auto-tuning:

- **Non-iterative Approach** (Building predictive modeling to come up with the right prediction of the speedup)
  
  **Downside:**
  - Error-rate on prediction and big exploration time
  
  *Can’t we do better?*
  - Using heuristics and algorithms to build more accurate speedup predictors
Proposed Methodology

The Challenges

- Conventional approaches for Phase-ordering problem can’t be useful since:
  
  a. *Design space is huge*
  
  b. *Classic predictive modeling in M.L needs fixed-length feature-vector*
  
  c. *Inter correlation between the compiler optimization on every level*

Goal is to identify the best compiler optimizations to be applied to a target application.
Proposed Methodology
The Workflow

- **Two** level approach:
  
  a. *Kernel Characterization*:
     
     - Each application passes through a characterization phase that generates the performance counter dynamic feature vectors
  
  b. *Predictive Modeling*:
     
     - Using predictive modeling to iteratively predict the immediate speedup (next-best optimization to be applied)
Proposed Methodology
High-level View

I) Collecting Instances
- Program A
  - Int main(...)
    - ...
  - Program source code
- Backend Compiler
- Features Collection
- Speedup Value
- Optimization Sequences
- Collected Instances

II) Training Phase:
- Generating Immediate Speedup Dataset
- Speedup over previous program version
  - Next Optimization
  - Program Features
- Collected Instances
- Immediate Speedup Computation

3- Exploitation Phase:
- Predictions Using the Trained Model
- Current Program Features
- Next Optimization
- Trained Predictive Model
- Predicted Immediate Speedup
- Machine Learning Algorithm
1. DFS Immediate-speedup Predictor

A. Start from empty optimization set $O_0$
B. For each $O_i$ possible opt:
   - we predict speedup $\Phi_i$ derived from applying $O_i$ after $O_i$
   - computing immediate speedup $e_i = \text{Exec}(O_i)/\text{Exec}(O_j)$
   - sorting the values of speedup $\Phi_i$
   - If no other associate speedup >1 found
     - we choose $O_i$ to start exploring for the next level
   - otherwise repeat the same exploration policy

2. Exhaustive Immediate-speedup Predictor

We simply generate all predicted speedups on every-level and multiply the value of the predicted speedup to find the best immediate to be applied:

$$\sigma_o : \prod_{o_i \in o} \delta(o_i)$$
Experimental Setup

- Architecture: Quad-core Intel-Xeon E1607
- Ubuntu 14.04
- LLVM v3.8
- Benchmarks:
  - cBench [Fursin et al., 2010] -> using 6 different applications
- Characterization techniques:
  - Dynamic -> using MICA [Hoste et al., 2007]
- 4 promising compiler gene each containing several relatively fixed flags
Proposed Methodology
Experimental Results (1/2)

- Performance improvement enabling repetitions in the framework
- Utilized compiler optimizations in 4 different genes

<table>
<thead>
<tr>
<th>Application</th>
<th>Best Opt Found W.Rep</th>
<th>Best Opt Found WOut.Rep</th>
<th>Speedup %</th>
</tr>
</thead>
<tbody>
<tr>
<td>automotive-susan-c</td>
<td>CDD</td>
<td>CD</td>
<td>9.34</td>
</tr>
<tr>
<td>network-patricia</td>
<td>CDCD</td>
<td>AD</td>
<td>60.37</td>
</tr>
<tr>
<td>automotive-qsort1</td>
<td>CBA</td>
<td>CBA</td>
<td>-</td>
</tr>
<tr>
<td>automotive-bitcount</td>
<td>CCBB</td>
<td>CDB</td>
<td>2.41</td>
</tr>
<tr>
<td>network-dijkstra</td>
<td>ACAB</td>
<td>AD</td>
<td>36.59</td>
</tr>
<tr>
<td>automotive-susan-e</td>
<td>CD</td>
<td>CD</td>
<td>-</td>
</tr>
<tr>
<td>Harmonic Mean</td>
<td>-</td>
<td>-</td>
<td>7.68</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gene</th>
<th>Abbreviation</th>
<th>Relative Positioning of the Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>domtreeRULE</td>
<td>-domtree -memdep -dse -adce -instcombine -simplifycfg -domtree -loops -loop-simplify -lcssa -branch-prob</td>
</tr>
<tr>
<td>B</td>
<td>simplifycfgRULE</td>
<td>-simplifycfg -reassociate -domtree -loops -loop-simplify</td>
</tr>
<tr>
<td>C</td>
<td>memdepRULE</td>
<td>-memdep -domtree -memdep -gvn -memdep -memcpyopt -sccp</td>
</tr>
<tr>
<td>D</td>
<td>loopsRule</td>
<td>-loops -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -loop-vectorize</td>
</tr>
</tbody>
</table>

Total Number of Optimizations Under Analysis: 30
Unique Number of Optimizations Under Analysis: 13
Proposed Methodology
Experimental Results (2/2)

- Average Speedup of different search policies w.r.t the actual speedup
  - Average Prediction error-rate:
    - Greedy DFS: 12.4%
    - Iterative exhaustive: 12.9%

- Performance improvement per application

<table>
<thead>
<tr>
<th>Application</th>
<th>Greedy DFS</th>
<th>Exhaustive Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>automotive-susan-c</td>
<td>0.9808</td>
<td>0.9658</td>
</tr>
<tr>
<td>network-patricia</td>
<td>1.0069</td>
<td>1.0002</td>
</tr>
<tr>
<td>automotive-qsort1</td>
<td>1.1255</td>
<td>0.9670</td>
</tr>
<tr>
<td>automotive-bitcount</td>
<td>1.0848</td>
<td>1.1506</td>
</tr>
<tr>
<td>network-dijkstra</td>
<td>0.9988</td>
<td>0.9988</td>
</tr>
<tr>
<td>automotive-susan-e</td>
<td>1.0617</td>
<td>1.1015</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1.0431</strong></td>
<td><strong>1.0242</strong></td>
</tr>
</tbody>
</table>
Proposed Methodology

Wrap-up

a. Introducing a predictive modeling capable of predicting the immediate next-best compiler optimization to be applied given the current status of the code

b. Utilizing *Independent application characterization* so that the features are collected independent from the underlying architecture

c. Acquiring some *pre-defined* good compiler options to be fixed relatively and treat them as one (Derived from LLVM Opt)

d. Applying search heuristics to explore the more interesting regions of the design space to further optimize exploration speedup
Outline

• Problem Description
  ○ Introduction And Motivation
  ○ Phase-ordering of compiler optimizations

• Proposed Methodology
  ○ Using Machine Learning to focus on iterative compilation
  ○ Application-Specific compiler tuning
  ○ Experimental results
  ○ Conclusion

• Ongoing and future research directions
Future Research Directions

1. Using *graph-based representation of compiler IR* for kernel feature extraction and auto-tuning

2. *Multi-core homogeneous compiler optimization* tackling:
   a. Power consumption vs. performance speedup
   b. Utilizing LLVM supporting OpenMP
   c. Focusing on Polyhedral compilation model for loop-nest kernels for HPC domain
Questions?

Thanks for your time!