Background

- **Education**
  - **B.Sc.** (2005-2009): Iran University of Science & Tech (IUST)
  - **M.Sc.** (2010-2012): Politecnico di Milano (POLIMI)
  - **Ph.D.** (2013-2016): Politecnico di Milano (POLIMI)
    - PhD Advisors:
      - Cristina Silvano (Polimi, Italy)
      - Gianluca Palermo (Polimi, Italy)
      - John Cavazos (University of Delaware, USA)
  - **Postdoctoral Fellow** (2017- ): University of Toronto, Canada
    - Advisor: Tarek Abdelrahman
    - Industrial Partner: Qualcomm Inc. Canada
High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
1. Organization
A Survey on Compiler Autotuning using Machine Learning

- ACM Computing Surveys (ACM CSUR) – 2018
- arXiv preprint version¹: Will be updated quarterly
- Send us your new autotuning papers to be added

High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
Proebsting Law

- **Proebsting’s law:**
  “Compilers Advances Double Computing Power Every 18 Years”
  - “Even if we assume that the beginning of useful compiler optimization research began in the mid 1960's, the uniform performance improvement on integer intensive codes due to compiler optimization is still 3.6% per year, this lies in stark contrast to the 60% per year performance we CAN expect from hardware due to Moore's law”

- Some initial inferences:
  - A Need more autotuning
  - Adapting new machine learning techniques
  - Leveraging more accelerators (FPGAs, GPUs, DSPs, etc.)

1. Introduction/Motivation

What do compilers do? (1/2)
1. Introduction/Motivation

What do compilers do? (2/2)

- C/C++ Application Programs
- Fortran Application programs

Front End

Middle End

- Loop Optimizer
- Global Optimizer

Back End

- Code Expand
- Resource Binding
- Code Emission
- Assembly Code
- Machine Description

Amir H. Ashouri – Compiler Autotuning Seminar – Milan, Italy - July 2018
1. Introduction/Motivation

What is a compiler optimization? (1/2)

- It transforms and optimizes a piece of code:

  **Loop-unrolling**

<table>
<thead>
<tr>
<th>Normal loop</th>
<th>After loop unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>int x;</td>
</tr>
<tr>
<td>for (x = 0; x &lt; 100; x++)</td>
<td>for (x = 0; x &lt; 100; x += 5)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>delete(x);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>delete(x);</td>
</tr>
<tr>
<td></td>
<td>delete(x + 1);</td>
</tr>
<tr>
<td></td>
<td>delete(x + 2);</td>
</tr>
<tr>
<td></td>
<td>delete(x + 3);</td>
</tr>
<tr>
<td></td>
<td>delete(x + 4);</td>
</tr>
</tbody>
</table>
1. Introduction/Motivation

What is a compiler optimization? (2/2)

- It transforms and optimizes a piece of code:

  **Loop-tiling (i.e. 2*2 block):**

  ```
  int i, j, a[100][100], b[100], c[100];
  int n = 100;
  for (i = 0; i < n; i++) {
    c[i] = 0;
    for (j = 0; j < n; j++) {
      c[i] = c[i] + a[i][j] * b[j];
    }
  }
  ```

  **After tiling**

  ```
  int i, j, x, y, a[100][100], b[100], c[100];
  int n = 100;
  for (i = 0; i < n; i += 2) {
    c[i] = 0;
    c[i + 1] = 0;
    for (j = 0; j < n; j += 2) {
      for (x = i; x < min(i + 2, n); x++) {
        for (y = j; y < min(j + 2, n); y++) {
          c[x] = c[x] + a[x][y] * b[y];
        }
      }
    }
  }
  ```
1. Introduction/Motivation

Why do we need compiler optimizations?

- 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
1. Introduction/Problem description

Choosing the right optimizations (1/2)

- Compilation $\approx$ Translation [O’Boyle2014]

*2mm application from Polybench [Pouchet2012]
1. Introduction/Problem description

Choosing the right optimizations (2/2)

Compilation = Translation + Optimization

Opt.3
Opt.4
Opt.5
Opt.6

Many Billions of different high-level languages can be generated from a simple piece of code.
1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
1. Introduction/Problem description

Compiler Optimization Passes

- There are several pre-built compiler flags:
  - Standard flags, i.e. -O1, -O2, -O3, -Os
  - Many different optimization passes
    - ~ 160 in LLVM
    - ~ 250 in GCC
    - ~ 75 in ICC

- Identifying the right set of compiler passes (flags) will have substantial benefits on:
  - Performance
  - Code-size
  - Power metrics

- The design space is large and choosing the right ordering matters:

- Two Major Problems:
  1. Selecting the right set of compiler passes
  2. Phase-ordering of compiler passes
Autotuning addresses:
- Automatic code generation
- Automatic code optimization
- Enabling many different scenarios
  - Minimizing/Maximizing certain objective functions in the optimization space

Historically Autotuning was only done in the back-end:
- Downsides:
  - Requires exclusive knowledge of compiler construction
  - Higher overhead to modify and tune the back-end internally

State-of-the art autotuning research leverages machine learning techniques
A Sample Autotuning Framework

Amir H. Ashouri – Compiler Autotuning Seminar – Milan, Italy - July 2018
Identifying The Best Compiler Passes
The Selection Problem

- **The Selection Problem:**
  - Several compiler optimizations passes form an optimization sequence. We disregard the ordering of these optimizations and focus on whether to apply an optimization [Agakov et al. 2006].
    - Optimization space $P^n$:
      - $n = \#$ of optimizations, $p = \#$ of available optimization levels
  - Example: (let $n=5$ optimizations, $p=3$ optimization levels)
    - loop-unrolling – tiling – dce – loop-fusion – mem2reg
    - loop-unrolling(3) – tiling(2,2) – loop-fusion
    - loop-unrolling(1)
    - dce – loop-fusion

$\Rightarrow$ Different optimizations to choose: $3^2 \times 2^3 = 72$
State-of-the-art: Iterative Compilation

- Several re-compilation of the a source-code using different optimization flags to choose the best found version\(^1\).

  - **Advantage:**
    - Normally brings good results in a long-run
    - No machine intelligence needed (on its pure version)

  - **Downside:**
    - High overhead task and time-consuming
    - Must be repeated for a new dataset, application and architecture
    - No knowledge transfer involves

- Can we do better ???
  - Employing machine learning based models with Iterative Compilation

\[1\] Bodin et al. 1998. Iterative compilation in a non-linear optimisation space.
The \textbf{Phase-ordering} Problem:

- There is \textit{no ideal ordering of phases}. Optimization pass A transforms the program in ways that obliterate some optimizations that otherwise could have been performed by pass B, which follows it.

- Optimization space: $\sum_{i=0}^{m} n^i$

- Example: (let $n=5$ optimizations, $m=5$ optimization-vector max-length)
  - $\{-\text{loop-unrolling} - \text{tiling} - \text{dce} - \text{loop-fusion} - \text{mem2reg}\}$
  - $\{-\text{dce} - \text{loop-unrolling} - \text{loop-fusion}\}$
  - $\{-\text{loop-fusion} - \text{mem2reg}\}$
  - $\{-\text{dce} - \text{mem2reg} - \text{dce} - \text{mem2reg}\}$

\textgreater\textgreater\ Different optimizations to choose: $5^0 + 5^1 + 5^2 + 5^3 + 5^4 + 5^5 = 3906$
Tackling the Phase-ordering Problem

- **Major issue:**
  - Optimization space is significantly big that you cannot reproduce the previous approaches used for the selection problem:
    - Let $n=20$ optimization
    - The selection space $\sim 10^7$
    - The phase-ordering space $>>$ at least $10^{18}$ (no repetition case)

- **Possible approaches:**
  - Grouping the optimizations in sub-sequences
    - Random clustering
    - Manual clustering
    - Automatic clustering
  - Predicting the performance speedup of an optimization vector rather than the predicting the compiler optimization sequence:
    - Speedup predictor vs. Sequence predictor
High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers

Amir H. Ashouri – Compiler Autotuning Seminar – Milan, Italy - July 2018
ML applications to compiler autotuning requires a representation of the code being optimized.

A pass(ess) over the application is needed to collect its representative features.

These vectors of features will be used as the representation of an application.

Conforms to these two conditions:
- It must be representative enough of its application
- Different applications must not have same feature vectors
3- Cont’d Examples

- **Dynamic** (Micro-architectural Independent Characterization of Applications- MICA [1])
  - Pros
    - Fine-grain characterization, higher accuracy
  - Cons
    - (May be) Architecture depended, slower to collect

- **Static analysis on the source-code** (Milepost EU-Project [2])
  - Pros
    - Faster to fetch, independent of the architecture
  - Cons
    - Transparent to input datasets, lower accuracy [2,3]

- **Hybrid characterization** (COBAYN [4])

3- Cont’d Examples (MICA)

MICA (99 features – 6 categories)

<table>
<thead>
<tr>
<th>i</th>
<th>types</th>
<th>instruction mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppm</td>
<td>taken rate, transition rate, Markov-chain based branch prediction</td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>distribution of register dependency distances, avg. number of input registers, degree of use</td>
<td></td>
</tr>
<tr>
<td>stride</td>
<td>distribution of memory access address distances</td>
<td></td>
</tr>
<tr>
<td>memfootprint</td>
<td>memory footprint (# blocks/pages touched)</td>
<td></td>
</tr>
<tr>
<td>ilp</td>
<td>amount of available inherent ILP</td>
<td></td>
</tr>
</tbody>
</table>
1. **Introduction**
2. **Compiler Optimization Concept**
3. **Characterization Techniques**
4. **Machine Learning Models**
5. **Prediction Types**
6. **Optimization Space Exploration**
7. **Target Domain**
8. **Influential Papers**
4- Machine Learning Models

- It investigates the study and construction of techniques/Algorithms that are able to learn certain attributes from data.

- Three Broad categories:
  - Supervised Learning
  - Unsupervised Learning
  - Reinforcement Learning
Addressing this question:
Given the characteristics of my application under analysis, what optimizations should I use to maximize its performance?

- **Input**: Characteristics ($\alpha$) of an application under analysis
- **Output**: Best sequence of optimization vector ($\bar{o}$) to use to maximize the performance of an application

What is a BN?

- It is a probabilistic graphical model expressed by graphs.
- It represents a set of random variables and their conditional dependences via a directed acyclic graph (DAG):
  - $\alpha_i$ represents an application characteristic.
  - $O_i$ represents a set of optimizations under analysis.

- DAGs are based on probability distribution.
  - It is a statistical machine learning model.
How does a BN work?

- Based on a joint probability function:

\[
\Pr(G, S, R) = \Pr(G|S, R) \Pr(S|R) \Pr(R)
\]

- i.e. Three Boolean variables (Grass wet, Raining, Sprinkler):

<table>
<thead>
<tr>
<th>RAIN</th>
<th>SPRINKLER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>0.4</td>
</tr>
<tr>
<td>T</td>
<td>0.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RAIN</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.2</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPRINKLER</th>
<th>RAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>F</td>
<td>0.8</td>
</tr>
<tr>
<td>T</td>
<td>0.9</td>
</tr>
<tr>
<td>T</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Advantage of using BN

- There is no way to identify an analytic model to fit the vector function $\bar{\alpha}$:
  - Therefore, the method should infer its probability distribution of sequence ($\bar{\alpha}$)

- Why using a BN?
  - Suitable to represent cause-effect phenomena
  - Allows us to use as predictor for a vector of optimization vs. a single predictor
  - Intrinsically can be paired with iterative compilation method
Goal is to identify the best compiler optimizations to be applied to a target application.

Supervised Learning Flow:

a. Application Characterization:
   - Each application is passed through a characterization phase that generate the performance counter dynamic feature vectors

b. Model Induction:
   - Bayesian Networks correlates those features to compiler optimization and build a model.

c. Inference (Prediction):
   - Inferring the trained Bayesian Networks using the evidence from its characterizations to predict the best sets of application-specific compiler flags

COBAYN’s Methodology

Training Phase

- Application characterization
- Dimension reduction
- Bayesian net. training
- Top N% Solutions

Design Space Exploration Engine

Proposed framework

Target application

Prediction Phase

- Application characterization
- Dimension reduction
- Application-specific exploration
- Design Space Exploration Engine
- Optimal Solution

Evidence

Output binary
COBAYN’s Inference

Principal Components of Application Features

- PC.1
- PC.2
- PC.3
- PC.4
- PC.5
- PC.6
- PC.7
- PC.8
- PC.9
- PC.10

Math-opt

fn-tree-opt

fn-inline

funroll-lo

fn-gss-br

fn-ivopt

O2

Loop unrolling impacts on branch probability

Tree-based optimizations are interdependent

Compiler Transformations
High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
5- Prediction Types

- Compiler autotuning seeks to provide several prediction output classes:
  1. Identifying the right set of compiler sequences (pass)
  2. Predicting the speedup of a compiler sequence
    - Intermediate-speedup (tournament) prediction
    - Full-sequence speedup prediction
  3. Clustering (grouping) the effective compiler passes
  4. Identifying the right set of application features to use
5. Cont’d

Speedup Prediction Models

- **Intermediate speedup prediction** [Kulkarni & Cavazos 2012; Ashouri et al. 2016]
  - Easier/more intuitive to build a model
  - Needs continuous application characterization at each state
  - Higher error rate due to the multiple use of the prediction model

- **Full-sequence prediction** [no literature as of 2017]
  - Harder to build as a model
  - Needs an encoding scheme and an exploration policy to generate fixed feature-vector length
  - At best, needs only one round of feature selection
Intermediate Speedup Prediction Model

[Kulkarni & Cavazos 2012; Ashouri et al. 2016]

Prog. Features

Opt. Passes

Prog. Features

Opt. Passes

A

B

Final Opt={A, B, B}
**Full-sequence Speedup Prediction Model**

[Ashouri et al., MiCOMP, ACM TACO 2017]

Program Features at a Baseline

Opt. Passes

\[ \text{Final Opt}=\{A, B, B, C\} \]
LLVM opt \(-O3\) has:
- \(\sim 157\) optimization passes (\(\sim 60\) unique passes)

We represent \(G=(V,E)\) where
- \(V\) is the set of opts (nodes)
- \(E\) is the set of dependence between the opts (edges)
  - \(E(v_x,v_y)=1\) if that pair of optimizations consecutively appears in \(-O3\)
  - An edge can be annotated with the number of times that each pair of optimizations consecutively appears in the sequence
  - \(++E(v_x,v_y)\);
  - This will result in an Optimization Dependency Graph for \(-O3\).

MiCOMP Clustering (2/4)
Visualizing -O3’s Optimization Dependency Graph

Amir H. Ashouri – Compiler Autotuning Seminar – Milan, Italy - July 2018
Agglomerative clustering:
- Iterative hierarchical clustering
- Bottom-up approach that merges smaller clusters

Algorithm:
- Receive as input the matrix of graph (Optimization Dependency Graph) and desired number of clusters (n)
- Calculate the weighted matrix
- Build K-nearest neighbors
- Build transition probability and forms sample clusters ($c_1, \ldots, c_n$)
- Iteratively try to add more subclusters to the cluster list provided:
  - Conditional sum of the all path integrals within the new sub-clusters maximizes the objective function

<table>
<thead>
<tr>
<th>sub-seq</th>
<th>Compiler Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-ipsccp -globalopt -deadargelim -simplyfycfg -functionattrs</td>
</tr>
<tr>
<td></td>
<td>-argpromotion -sroa -jump-threading -reassociate -indvars</td>
</tr>
<tr>
<td></td>
<td>-mldst-motion -lcssa -rpo-functionattrs -bdce -dse -inferattrs</td>
</tr>
<tr>
<td></td>
<td>-prune-eh -alignment-from-assumptions -barrier -block-freq</td>
</tr>
<tr>
<td></td>
<td>-loop-unswitch -branch-prob -demanded-bits -float2int</td>
</tr>
<tr>
<td></td>
<td>-forceattrs -loop-idiom -globals-aa -gvn -loop-accesses</td>
</tr>
<tr>
<td></td>
<td>-loop-deletion -loop-unroll -loop-vectorize -sccp</td>
</tr>
<tr>
<td></td>
<td>-strip-dead-prototypes -inline -globaldce -constmerge</td>
</tr>
<tr>
<td>B</td>
<td>-licm -mem2reg</td>
</tr>
<tr>
<td>C</td>
<td>-loop-rotate -instcombine -loop-simplify</td>
</tr>
<tr>
<td>D</td>
<td>-memcpyopt</td>
</tr>
<tr>
<td>E</td>
<td>-loop-unswitch -adce -slp-vectorize -tailcalleeлим</td>
</tr>
</tbody>
</table>
MiCOMP’s Result

![Graph showing normalized speedup to -O3 for MiCOMP proposed sub-seq, baseline -O1 + MiCOMP proposed sub-seq, baseline -O2 + MiCOMP proposed sub-seq, and baseline -O3 + MiCOMP proposed sub-seq. The graph is sorted by increasing actual speedup and highlights a region of interest.](image)
High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
Traversing the large compiler optimization space requires an exploration strategy:

- Adaptive Compilation
  - Profile-guided optimization [1]
- Non-iterative Compilation
  - Presenting a global optimization approach [2]
- Iterative Compilation
  - Approximation procedures, iteratively applied [3]

Several re-compilation of the source code using different optimization flags to choose the best found version\textsuperscript{1}.

- **Advantage:**
  - Normally brings good results in a long-run
  - No machine intelligence needed (on its pure version)

- **Downside:**
  - High overhead task and time-consuming
  - Must be repeated for a new dataset, application and architecture
    - No knowledge transfer involves

- **Can we do better ??**
  - Employing machine learning based models with Iterative Compilation

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
Compiler autotuning is heavily correlated with the type of compiler, target architecture, and platform to be tuned.

- Example of State-of-the-art compilers:
  - GCC, Intel-ICC, LLVM, Polyhedral model, JIT, NVCC, etc.

- Target Architecture:
  - Embedded Domain
  - Desktop
  - HPC Domain
High-level Outline

1. Introduction
2. Compiler Optimization Concept
3. Characterization Techniques
4. Machine Learning Models
5. Prediction Types
6. Optimization Space Exploration
7. Target Domain
8. Influential Papers
8- Influential Papers

- Introducing learning methods

- Genetic algorithm
  - [cooper et al. 1999, 2002; Triantafyllis et al. 2003; Knijnenburg et al. 2003]

- Iterative compilation
  - [Bodin et al. 1998; Stephenson et al. 2005; Agakov et al. 2006, Ashouri et al. 2014, 2016], many other work

- Dynamic & hybrid features
  - [Cavazos et al. 2007; Park et al. 2012; Ashouri et al. 2016]

- Optimization groups and Bayesian learners
  - [Ashouri et al. 2014, 2016]

- Phase-ordering
  - [Kulkarni et al. 2004; Kulkarni and Cavazos 2012, Ashouri et al. (2017)]
Discussion

- Heterogeneous Autotuning
  - GPUs, DSPs, FPGAs

- Faster, more accurate models are needed:
  - Supercomputers
  - Deep learning models
    - LSTM$^{1,2}$ Model
  - Accurate application characterization techniques

Thank you!
Questions & Comments

□ Contact:

- Amir H. Ashouri
- Postdoc @ University of Toronto, Canada
- http://www.eecg.toronto.edu/~aashouri/
- aashouri@ece.utoronto.ca