Using Graph-Based Characterization for Predictive Modeling

Eunjung (EJ) Park, John Cavazos, Marco A. Alvarez

University of Delaware
Department of Computer and Information Sciences

April 3, 2012
IEEE/ACM 10th International Symposium on Code Generation and Optimization
San Jose, California, USA
Motivation

State-of-the-art Characterizations

Graph-Based Characterization

Building and Using Models

Evaluation
Finding the Best Optimization is Difficult

Applied Optimization Combinations Sorted by Increasing Actual Speedup

- Degrade
- Improvement > 1.0x
- Significant Improvement >= 6.0X
- The best Improvement

Actual Speedup

Normalized Speedup to fast in Intel ICC Compiler

PoCC
- Fusion
- Unrolling
- Tiling
- Parallelization
- Vectorization

2MM (Matrix Multiply)

PoCC
- 600 configs
- PolyBench V2.1

Wednesday, April 11, 12
Finding the Best Optimization is Difficult

Observations

- **Bundled optimization** (-fast) is not always the best.
- **Very low** density of "very" good points.
- There is no single configuration **beneficial to all benchmarks**.

Applied Optimization Combinations Sorted by Increasing Actual Speedup

PoCC

- Fusion
- Unrolling
- Tiling
- Parallelization
- Vectorization

Normalized Speedup to fast in Intel ICC Compiler

Applied Optimization Combinations Sorted by Increasing Actual Speedup

2MM (Matrix Multiply)

Wednesday, April 11, 12
Motivation for Using a Model

Model follows the trend of actual speedup

Applied Optimization Combinations Sorted by Increasing Actual Speedup
Motivation for Using a Model

Applied Optimization Combinations Sorted by Increasing Actual Speedup

- Top 1 - 5 Predicted Combinations
- Top 6 - 10 Predicted Combinations

2MM (Matrix Multiply)

PoCC
- Fusion
- Unrolling
- Tiling
- Parallelization
- Vectorization

PolyBench V2.1

600 configs
Contributions

- Propose a novel characterization technique - **graph-based**
- Evaluation of **graph-based** and state-of-the-art techniques
- Significant perf. improvement using the predictor from **graph-based** technique
State-of-the-art Characterizations

Dynamic

- Performance Counters
- K-Reaction

Static

- Source Code
# State-of-the-art Characterizations

## Performance Counters

<table>
<thead>
<tr>
<th>Category</th>
<th>List of PCs</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Line Access</td>
<td>CA_CLN, CA-ITV, ..</td>
<td>3</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>L1_DCA, L1_DCH, ...</td>
<td>10</td>
</tr>
<tr>
<td>L2/L3 Cache</td>
<td>L2_DCA, L2_DCM, ...</td>
<td>16</td>
</tr>
<tr>
<td>Branch Related</td>
<td>BR_TKN, BR_NTK, ...</td>
<td>7</td>
</tr>
<tr>
<td>Floating Point</td>
<td>FDV_INS, FP_OPS, ...</td>
<td>6</td>
</tr>
<tr>
<td>Interrupt/Stall</td>
<td>HW_INT, RES_STL</td>
<td>2</td>
</tr>
<tr>
<td>TLB</td>
<td>TLB_DM, TLM_IM, ...</td>
<td>4</td>
</tr>
<tr>
<td>Total Cycle/Inst.</td>
<td>TOT_CYC, TOT_IIS, ...</td>
<td>3</td>
</tr>
<tr>
<td>Load/Store Inst.</td>
<td>LD_INS, SR_INS</td>
<td>2</td>
</tr>
<tr>
<td>SIMD Inst</td>
<td>VEC_INS, VEC_DP, ...</td>
<td>3</td>
</tr>
</tbody>
</table>
State-of-the-art Characterizations

Dynamic

Performance Counters

K-Reaction

Static

Source Code

Performance Counters

Underlying Architecture

Complied with -O0

Performance Counters for N-1 Programs
State-of-the-art Characterizations

- **Dynamic**
  - K-Reaction
  - Performance Counters

- **Static**
  - Source Code

- Reaction = Performance impact after applying a given combination of optimizations

- K-Reaction = Set of performance improvement/degradation from applying $K$ different optimizations
State-of-the-art Characterizations

Dynamic

K-Reaction

Static

Performance Counters

Source Code

K Reactions

K Opt. Seqs

1

... 

K

N-1 programs

1

... 

K

Backend Compiler

Underlying Architecture

K Reactions for N-1 programs
State-of-the-art Characterizations

Dynamic

- Performance Counters
- K-Reaction

Static

- Source Code

- Collected from source code or compiler intermediate representations
- Number of different type of statements in a function
- Statistics on control flow graph in a function
State-of-the-art Characterizations

Dynamic

K-Reaction

Performance
Counters

Static

Source Code

Source Code

N-1 programs

Milepost
GCC

R Source code features for N-1 programs

1
...
N-1

R

N-1

...
State-of-the-art Characterizations

**Dynamic**
- Performance Counters
- K-Reaction

**Static**
- Source Code

**Advantages**
- Easy to collect and works with standard ML algorithms

**Disadvantages**
- Fixed-length vector loses information about program (e.g., shape of loop nest)
Proposed Technique: Graph-Based

- Graph-based program characterizations (e.g., control flow graphs, data dependence graphs)
- Main difference from state-of-the-art
  - Using flexible data instead of fixed length of vector
Proposed Technique: Graph-Based

Graph-based - CFG

N-1 programs

MinIR

CFG topology and feature of each node for N-1 programs

Feature vector of each bb

bb1 → bb2, bb3
bb2 → bb5
...
bb5 → bb6

CFG Topology
Graph-based Characterization using CFG

2MM - Matrix Multiplication

Control flow graphs from compiler

Features of Each Node
- Number of Instructions
- Number of Add Instruction
- Number of Sub Instruction
- Number of Mult Instruction
- Number of Div Instruction
- Number of Load Instruction
- Number of Store Instruction
- Number of Comparisons
- Number of Conditional Branches
- Number of Unconditional Branches

Graph-based Characterization using CFG

1: <1,0,0,0,0,0,0,0,1,0,0>
2: <4,0,0,0,0,1,0,1,0,2>
3: <16,2,0,1,0,3,1,0,1,0,8>
...
16: <2,1,0,0,0,0,0,1,0,0,0>
17: <3,0,0,0,0,0,1,1,0,1,0>
18: <1,0,0,0,0,0,0,0,1,0,0>

Feature vector for each node

Topology of CFG

1->6
2->4
3->4
...
16->17
17->10
17->18

Number of Instructions
Number of Add Instruction
Number of Sub Instruction
Number of Mult Instruction
Number of Div Instruction
Number of Load Instruction
Number of Store Instruction
Number of Comparisons
Number of Conditional Branches
Number of Unconditional Branches
Graph-based Characterization using CFG

2MM - Matrix Multiplication

Feature vector for each node
1: <1,0,0,0,0,0,1,0,0,1,0,0>
2: <4,0,0,0,0,1,0,0,1,0,2>
3: <16,2,0,1,0,3,1,0,0,1,0,8>
...
16: <2,1,0,0,0,0,0,0,1,0,0>
17: <3,0,0,0,0,0,1,1,0,1,0>
18: <1,0,0,0,0,0,0,0,1,0,0>

Topology of CFG
1->6
2->4
3->4
...
16->17
17->10
17->18

Features of Each Node

Number of Instructions
Number of Add Instruction
Number of Sub Instruction
Number of Mult Instruction
Number of Div Instruction
Number of Load Instruction
Number of Store Instruction
Number of Comparisons
Number of Conditional Branches
Number of Unconditional Branches
Prediction Model: Speedup Predictor

Control flow graph data

Optimization Sequence $O$

Predicted speedup of $O$ over the Baseline
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

Kernel Definition:

\[
K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)
\]

\[
k_{walk}(e_1, e_2) = k_{node}(v_1, v_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(w_1, w_2)
\]

*\(K_{node}(v_1, v_2)\) indicates the **comparison of feature vectors** of \(v_1\) and \(v_2\).*

*\(K_{edge}(e_1, e_2)\) indicates the **comparison of weights** of \(e_1\) and \(e_2\).*
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

1. Shortest Path Graph

\[ SPG1 = \langle V_1, E_1 \rangle \]

2. Calculate \( k_{sp} \) of input graphs

\[
K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)
\]

2-a. Calculate \( k_{walk} \) of two edges

Floyd-Warshall
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

**Input Graphs**

CFG1

1. **Shortest Path Graph**

SPG1 = \(<V_1, E_1>\)

2. **Calculate** $k_{sp}$ **of input graphs**

$$K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)$$

2-a. **Calculate** $k_{walk}$ **of two edge**

$\text{knode} (v_1, v_2)$

---

Floyd-Warshall

---

**State-of-the-art**

**Motivation**

**Evaluation**

**Build & Use**

**Graph-based**
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

Input Graphs

1. Shortest Path Graph

2. Calculate $k_{sp}$ of input graphs

$$K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)$$

2-a. Calculate $k_{walk}$ of two edges

State-of-the-art

Motivation

Evaluation

Build & Use

Graph-based
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

Input Graphs

<table>
<thead>
<tr>
<th>CFG1</th>
<th>CFG2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="CFG1" /></td>
<td><img src="image2" alt="CFG2" /></td>
</tr>
</tbody>
</table>

1. Shortest Path Graph

SPG1 = \( <V_1, E_1> \)

SPG2 = \( <V_2, E_2> \)

2. Calculate \( k_{sp} \) of input graphs

\[
K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)
\]

2-a. Calculate \( k_{walk} \) of two edge

\( k_{node}(v_1, v_2) \)

\( k_{node}(w_1, w_2) \)

\( k_{edge}(e_1, e_2) \)
Shortest Path Graph Kernel [Borgwardts and Kriegel, 2005]

Input Graphs

CFG1

- 1
- 2
- 3
- 4
- 5

CFG2

- 1
- 2
- 3

1. Shortest Path Graph

SPG1 = \( <V_1, E_1> \)

SPG2 = \( <V_2, E_2> \)

Floyd-Warshall

2. Calculate \( k_{sp} \) of input graphs

\[
K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)
\]

2-a. Calculate \( k_{walk} \) of two edges

\( k_{node} (v_1, v_2) \)

\( k_{node} (w_1, w_2) \)

\( k_{edge} (e_1, e_2) \)

\[
k_{node}(v_1, v_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(w_1, w_2)
\]
Building the Prediction Model

- Leave-one-out cross validation
- Building the prediction model using N-1 programs

Diagram:
- CFG data for N-1 Programs
- Opt. Combinations and their speedup over baseline for N-1 programs
- Machine Learning Algorithm
- Generated model for a given machine
Using the Prediction Model

- Leave-one-out cross validation
- Using the predictor with one left out program

![Diagram showing the process of using the prediction model]

Nth Program (one left out) → Extract Program Characteristics → Program Characteristics → Optimization Sequences → Speedup Predictor → Predicted Target Value
Using the Prediction Model

- Leave-one-out cross validation
- Using the predictor with one left out program

![Diagram showing the process of using the prediction model]

Nth Program (one left out) → Extract Program Characteristics → Program Characteristics → Speedup Predictor

Sort by predicted Speedup

Non-iterative: 1-Shot
Using the Prediction Model

- Leave-one-out cross validation
- Using the predictor with one left out program

Nth Program (one left out) → Extract Program Characteristics → Program Characteristics

Speedup Predictor

Non-iterative: 1-Shot
Iterative (e.g., 5-shot)

Sort by predicted Speedup

top5
Evaluation

- PoCC (Source-to-source Polyhedral Compiler)
- Since source-to-source, requires a backend compiler

All Optimizations Considered
Unrolling Factor = 0, 2, 4, and 8
Loop Fusion = nofuse, maxfuse, smartfuse
Loop tiling = 1, 32
Parallelization = on/off
Vectorization = on/off
Experimental Configuration

- Hardware Configuration
  - (Nehalem) Intel Xeon E5620 2.4GHz, L3 12MB
  - (Quad) Intel Core2 Quad Q9650 3.0GHz, L2 12MB

- Software Configuration
  - Backend compiler: ICC (Baseline: -fast) GCC (Baseline -O3)
  - SVM
  - Benchmark: PolyBench V2.1 (30 scientific kernels)
    - 26 FP, 4 INT C kernels with Average LOC 337
## Experiment: CFG vs. State-of-the-art

- **Single-Shot SVM Model**

<table>
<thead>
<tr>
<th></th>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random</td>
<td>PC</td>
</tr>
<tr>
<td>Nehalem-GCC</td>
<td>2.0x (26%)</td>
<td>4.6x (51%)</td>
</tr>
<tr>
<td>Nehalem-ICC</td>
<td>1.1x (20%)</td>
<td>2.5x (43%)</td>
</tr>
<tr>
<td>Quad-GCC</td>
<td>1.2x (34%)</td>
<td>2.0x (54%)</td>
</tr>
<tr>
<td>Quad-ICC</td>
<td>1.4x (31%)</td>
<td>1.7x (37%)</td>
</tr>
</tbody>
</table>

1. All ML models outperformed Random model.
2. CFG achieved significant performance improvements by using single-shot CFG model (static technique)!
Experiment: CFG vs. State-of-the-art

☐ 5-Shot SVM Model

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>PC</th>
<th>5R</th>
<th>SRC</th>
<th>CFG</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem-GCC</td>
<td>4.7x (58%)</td>
<td>5.2x (57%)</td>
<td>5.8x (63%)</td>
<td>5.8x (63%)</td>
<td>7.2x (79%)</td>
<td>9.06x</td>
</tr>
<tr>
<td>Nehalem-ICC</td>
<td>2.7x (52%)</td>
<td>3.3x (58%)</td>
<td>3.3x (59%)</td>
<td>2.9x (52%)</td>
<td>4.1x (73%)</td>
<td>5.60x</td>
</tr>
<tr>
<td>Quad-GCC</td>
<td>2.3x (61%)</td>
<td>2.4x (66%)</td>
<td>2.4x (66%)</td>
<td>2.4x (65%)</td>
<td>3.2x (88%)</td>
<td>3.67x</td>
</tr>
<tr>
<td>Quad-ICC</td>
<td>2.4x (55%)</td>
<td>2.6x (57%)</td>
<td>2.5x (53%)</td>
<td>2.5x (55%)</td>
<td>3.5x (76%)</td>
<td>4.57x</td>
</tr>
</tbody>
</table>

We achieved up to 88% of OPT just in 5 iterations!
The Best Sequence for Nehalem-ICC
both: fusion=off, tiling=1x1x32, auto-par

The Best Sequence for Q9650-ICC
both: fusion=off, tiling=1x1x32, pluto-par, auto-par

The Best Sequence for Nehalem-GCC
atax: no-fusion, tiling size=32, auto-par
bicg: no-fusion, tiling size=32, pluto-par

The Best Sequence for Q9650-GCC
atax: unroll factor=8
bicg: unroll factor=4
Conclusion and Future Work

- Comparison of different characterizations for predictive modeling
- Graph-based characterization outperformed three state-of-the-art techniques
  - Non-iterative scenario: up to 74% of OPT
  - Iterative scenario: up to 88% of OPT
- Possibility of using different type of graphs