Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization

Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, Ari Freund

Presented by: Ziyang Wang, Ruoyao Wang, Likai Sheng, Junjie Xing

Overview

Introduction and Related Work

Terminology Definition

Feature Extraction

Methodology

Evaluation and Conclusion

Introduction and Related Work

Previous work:

- Iterative Compilation: Try different sets and orders of optimization
  - Optimization-Space Exploration: Identify the most relevant optimization
  - Expert System for Tuning Optimization(ESTO)

Timing Consuming!
Restrictive in Practice
(excessive recomplies and run)

General ML compiler phases

- Training Phase
- Prediction Phase

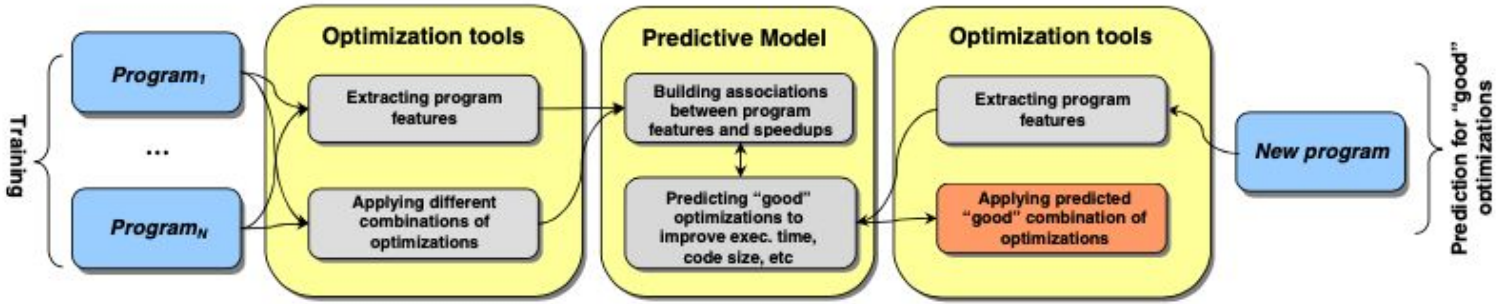

Figure 1: Typical machine learning scenario to predict "good" optimizations for programs. During a training phase (from left to right) a predictive model is built to correlate complex dependencies between program structure and candidate optimizations. In the prediction phase (from right to left), features of a new program are passed to the learned model and used to predict combinations of optimizations.

Related Terminologies

- Entities:
  - functions
  - instructions and operands
  - variables
  - types
  - constants
  - basic blocks
  - loops
  - compiler-generated temporaries

Related Terminologies

Relations:

A relation over one or more sets of entities is a subset of their Cartesian Product

Ex. $in \subseteq I \times B$, $in = \{(i\_k, b\_l) \mid$ instruction i_k is in basic block $b\_l\}$

call graph	dominator tree	anticipatibility information
CFG	data dependence graph	alias information
Loop Hierarchy	liveness information	
Control dependence graph	availability information	

Related Terminologies

Datalog:

- a Prolog-like language with more restricted semantics, suitable for expressing relations and operations
- Elements of Datalog are called atoms of form *p(X_1, ..., X_n)* P: predicates, X: variable or constants
- Datalog database consist of a list of rules: *H: - B_1, B_2, …, B_n* Facts: bodyless rule
- Datalog query: *- B_1, …, B_n*
- Datalog is capable of performing operations on relations to generate new relation
    - Ex. *st_in(I, B): - store(I), in(I, B)*

Automatic Inference of New Relations

- Complex relations can be inferred from basic relations
- Two relations can be joined on their shared variables

Example:

$r(E_1, E_2, E_3)$, $p(F_1, F_2, F_3)$, $E_2 = F_1$, $E_3 = F_2$

$rel1(E_1, E_2, E_3, F_2, F_3)$

$rel2(E_1, E_2, E_3, F_1, F_3)$

$rel3(E_1, E_2, E_3, F_3)$

Extracting Features from Relations

- Features: quantitative measurement of a program
- Convert relations to features
  - Relations with numerical value entities
  - Relations with categorical value entities
  - Binary relations
  - K-arity relations

Relations with Numerical Value Entities

- Values can be aggregated into their sum, average, variance, max, min, etc.
- Example:

$$count = \{(b,n) \mid b \text{ is a basic block whose estimated number of executions is } n\}$$

Relations with Categorical Value Entities

- Categorical values: Usually symbols like instructions, basic blocks
- Apply *num* to each entity:

  Example:

  st_in_block={(*i*, *b*) | *i* is a store instruction in basic block *b*}

  num(*st_in_block*[0]) : # of store instructions in all basic blocks

  num(*st_in_block*[1]) : # of basic blocks that contains store instructions

Binary Relations & K-arity Relations

- $r \subseteq E_1 \times E_2$, $e \in E_i$, $r(e)$ denotes the set of pairs in r that contains e.
- Features are extracted by aggregating the numerical values of (e, $num(r(e))$)
- I.e. Consider *st_in_block* again. Aggregate (b, $num(st\_in\_blocks(b))$).
- K-arity relations can be converted to binary relations

Structural Code Patterns

Problem:

Properties like # of edges, avg # of neighbors for a vertex present poorly for graph structures with small number of labels for vertices and nodes (e.g. CFG, DDG, dominator tree)

How to solve?

Characterize such graphs by a number of subgraph patterns

Structural Code Patterns

An example

Control Flow Graph can be considered as a relation over *B x B*, where *B* is the set of basic blocks.

```
bb_ifthen(B1,B3) :-
  bb_edge(B1,B3), bb_edge(B1,B2), bb_edge(B2,B3).

bb_ifthen_else(B1,B4) :-
  bb_edge(B1,B2), bb_edge(B1,B3),
  bb_edge(B2,B4), bb_edge(B3,B4).
```

Structural Code Patterns

Use these code patterns to iteratively induce new patterns

```
bb_ifthen_n(B1,B4) :-
  bb_edge(B1,B4), bb_edge(B1,B2),
  bb_ifthen(B2,B3), bb_edge(B3,B4).
```

This approach can be applied to other patterns in graphs such as cycles (loop structure) and bipartite graph (*def-use* relation).

Exploring the Structural Pattern Space

Problem:

Arbitrary complex patterns can be derived from the initial patterns.

Need to limit the pattern space for efficiency with constraints (by compiler expert).

Exploring the Structural Pattern Space

An example with CFG. Constraint (m,n):

m = maximal number of occurrences of the variable as the first argument

n = maximal number of occurrences of the variable as the second argument

```
constraint(B1) = (2,2)
constrains(B2) = (1,1)
constrains(B3) = (1,2)

Before extension
:- bb_edge(B1,B2), bb_edge(B1,B3).

After extension
:- bb_edge(B1,B2), bb_edge(B1,B3), bb_edge(B2,B3).
```

Evaluation

1. Use state-of-the-art predictive model
    - Compared with all similar features of other programs using a nearest classifier
2. Compile a new program & extract features
    - 56 features extracted after performing principal component analysis(PCA)
3. Recompiled the program with the combination of optimizations for the most similar program encountered so ar

Evaluation

Feature #	Description:
ft1	Number of basic blocks in the method
ft2	Number of basic blocks with a single successor
ft3	Number of basic blocks with two successors
ft4	Number of basic blocks with more than two successors
ft5	Number of basic blocks with a single predecessor
ft6	Number of basic blocks with two predecessors
ft7	Number of basic blocks with more than two predecessors
ft8	Number of basic blocks with a single predecessor and a single successor
ft9	Number of basic blocks with a single predecessor and two successors
ft10	Number of basic blocks with a two predecessors and one successor
ft11	Number of basic blocks with two successors and two predecessors
ft12	Number of basic blocks with more than two successors and more than two predecessors
ft13	Number of basic blocks with number of instructions less than 15
ft14	Number of basic blocks with number of instructions in the interval [15, 500]
ft15	Number of basic blocks with number of instructions greater than 500
ft16	Number of edges in the control flow graph
ft17	Number of critical edges in the control flow graph
ft18	Number of abnormal edges in the control flow graph
ft19	Number of direct calls in the method
ft20	Number of conditional branches in the method
ft21	Number of assignment instructions in the method
ft22	Number of unconditional branches in the method
ft23	Number of binary integer operations in the method
ft24	Number of binary floating point operations in the method
ft25	Number of instructions in the method
ft26	Average of number of instructions in basic blocks
ft27	Average of number of phi-nodes at the beginning of a basic block
ft28	Average of arguments for a phi-node
ft29	Number of basic blocks with no phi nodes
ft30	Number of basic blocks with phi nodes in the interval [0, 3]
ft31	Number of basic blocks with more than 3 phi nodes
ft32	Number of basic block where total number of arguments for all phi-nodes is in greater than 5
ft33	Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
ft34	Number of switch instructions in the method
ft35	Number of unary operations in the method
ft36	Number of instruction that do pointer arithmetic in the method
ft37	Number of indirect references via pointers ("*" in C)
ft38	Number of times the address of a variables is taken ("&" in C)
ft39	Number of times the address of a function is taken ("&" in C)
ft40	Number of indirect calls (i.e. done via pointers) in the method
ft41	Number of assignment instructions with the left operand an integer constant in the method
ft42	Number of binary operations with one of the operands an integer constant in the method
ft43	Number of calls with pointers as arguments
ft44	Number of calls with the number of arguments is greater than 4
ft45	Number of calls that return a pointer
ft46	Number of calls that return an integer
ft47	Number of occurrences of integer constant zero
ft48	Number of occurrences of 32-bit integer constants
ft49	Number of occurrences of integer constant one
ft50	Number of occurrences of 64-bit integer constants
ft51	Number of references of local variables in the method
ft52	Number of references (def/use) of static/extern variables in the method
ft53	Number of local variables referred in the method
ft54	Number of static/extern variables referred in the method
ft55	Number of local variables that are pointers in the method
ft56	Number of static/extern variables that are pointers in the method

Table 1: List of program features produced using our technique to be able to predict good optimizations

Evaluation

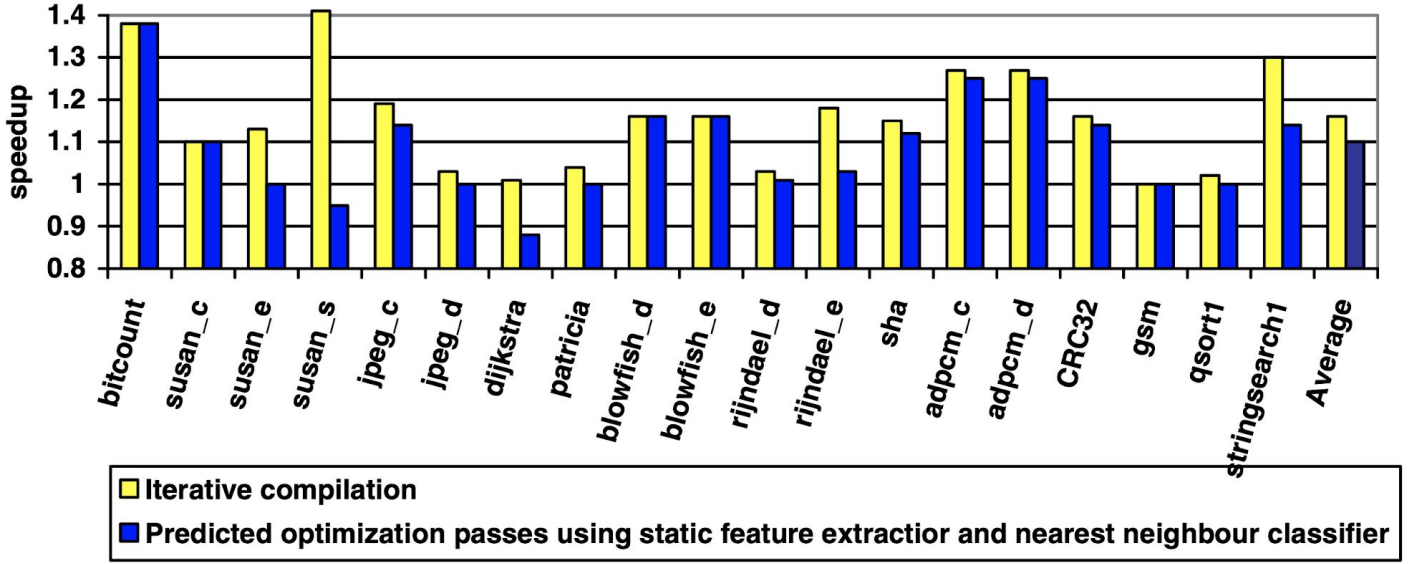

Figure 3: Speedups when predicting best optimizations based on program features in comparison with the achievable speedups after iterative compilation based on 500 runs per benchmark (ARC processor)

Strengths & Weakness

+ Systematically generating numerical features from a program for ML purposes

+ Using ML model to select optimizations takes less time than iterative compilation

- Limited amount of features from auto-generation

- The speedups obtained from extracted features are still far from optimal

Conclusion

- Though obtaining strong speedups, the iterative compilation process is very time-consuming and impractical in production
- Building a systematic method to construct features is an important step towards generalizing machine learning techniques to tackle the complexity of present and future computing system

Questions?