Hugh Leather, Edwin Bonilla, Michael O'Boyle

Institute for Computing Systems Architecture
University of Edinburgh, UK
Overview

- Introduction to machine learning in compilers
- Difficulties choosing features
- A feature space for a motivating example
- Searching the feature space
- Features for GCC
- Results
- Further and ongoing work
Introduction to machine learning in compilers

- Problem:
  - Tuning heuristics is hard
  - Architectures and compilers keep changing

- Goal:
  - Replace an heuristic with a Machine Learned one
  - ML performs very well
Introduction to machine learning in compilers

- How it works
  - Summarise data before heuristic (features)
  - Collect examples
  - Learn a model
  - Model predicts heuristic for new program
Introduction to machine learning in compilers

Start with compiler data structures
AST, RTL, SSA, CFG, DDG, etc.
Introduction to machine learning in compilers

Human expert determines a mapping to a feature vector

- number of instructions
- mean dependency depth
- branch count
- loop nest level
- ...
- trip count
Now collect many examples of programs, determining their feature values

Execute the programs with different compilation strategies and find the best for each
Introduction to machine learning in compilers

Now give these examples to a machine learner

It learns a model
This model can then be used to predict the best compiler strategy from the features of a new program.

Our heuristic is replaced.
Overview

- Introduction to machine learning in compilers
- **Difficulties choosing features**
- A feature space for a motivating example
- Searching the feature space
- Features for GCC
- Results
- Further and on going work
Difficulties choosing features

- The expert must do a good job of projecting down to features

- amount of white space
- average identifier length
- age of programmer
- name of program
- ...
- ...
Difficulties choosing features

- Machine learning works well when all examples associated with one feature value have the same type.
Difficulties choosing features

- Machine learning doesn't work if the features don't distinguish the examples
Difficulties choosing features

- Better features might allow classification

![Visual representation of feature classification](image)
Difficulties choosing features

- There are much more subtle interactions between features and ML algorithm
  - Sometimes adding a feature makes things worse
  - A feature might be copies of existing features
- There is an infinite number of possible features
Overview

- Introduction to machine learning in compilers
- Difficulties choosing features
- A feature space for a motivating example
- Searching the feature space
- Features for GCC
- Results
- Further and on going work
A feature space for a motivating example

- Simple language the compiler accepts:
  - Variables, integers, '+', '*', parentheses

- Examples:
  - a = 10
  - b = 20
  - c = a * b + 12
  - d = a * (( b + c * c ) * ( 2 + 3 ))
A feature space for a motivating example

- What type of features might we want?

\[
\text{count-nodes-matching(}
\text{is-times} \&\&
\text{left-child-matches(}
\text{is-plus}
\text{)}\&\&
\text{right-child-matches(}
\text{is-constant}
\text{)}
\]

\[a = ((b+c)*2 + d) * 9 + (b+2)*4\]

Value = 3
Define a simple feature language:

\[
\text{<feature>} ::= \text{"count-nodes-matching(\" <matches> \")"}
\]

\[
\text{<matches>} ::= \text{"is-constant"}
| \text{"is-variable"}
| \text{"is-any-type"}
| ( \text{"is-plus" | "is-times"})
| ( \text{"&\& left-child-matches(\" <matches> \")"})\ ?
| ( \text{"&\& right-child-matches(\" <matches> \")"})\ ?
\]
A feature space for a motivating example

- Now generate sentences from the grammar to give features
- Start with the root non-terminal

Grammar

\[ <A> ::= <A><A><A> | "b" \]

Sentence

\[ A \]
A feature space for a motivating example

- Now generate sentences from the grammar to give features
- Choose randomly among productions and replace

Grammar

\[ <A> ::= <A><A><A> | "b" \]

Sentence

AAA
A feature space for a motivating example

- Now generate sentences from the grammar to give features
- Repeat for each non-terminal still in the sentence

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;A&gt; ::= (&lt;A&gt;&lt;A&gt;&lt;A&gt;) \mid \text{“b”})</td>
<td>(b\text{AAAb})</td>
</tr>
</tbody>
</table>
A feature space for a motivating example

- Now generate sentences from the grammar to give features
- Continue until there are no more non-terminals

Grammar

\[ <A> ::= <A><A><A> | “b” \]

Sentence

\[ bbbbb \]
Overview

- Introduction to machine learning in compilers
- Difficulties choosing features
- A feature space for a motivating example
- **Searching the feature space**
- Features for GCC
- Results
- Further and on going work
Search space is parse trees of features
- Genetic programming searches over feature parse trees
- Features which help machine learning are better
Searching the Feature Space

- Overview of searching
Searching the Feature Space

- Data structures extracted from benchmarks
Searching the Feature Space

- Population of parse trees evolved with genetic programming
Searching the Feature Space

- Parse trees are interpreted over program data giving feature values
ML tool learns model using feature values and target heuristic values.
Searching the Feature Space

- Model predicts heuristic with cross validation
- Quality found (accuracy or speedup)
Searching the Feature Space

- After some generations first feature fixed
- Used when learning model for next feature
Searching the Feature Space

- Build up features, one at a time
- Stop when no improvement
Introduction to machine learning in compilers

Difficulties choosing features

A feature space for a motivating example

Searching the feature space

Features for GCC

Results

Further and on going work
Features for GCC

- Start by dumping data structures to XML

GCC Exports Data Structures to XML
Structure Analysed
Grammar Created
Sentence Generator Compiled
Feature Search
Final Features Produced

Benchmarks
.GCC .c
Program Data
.xml

Grammar Stylesheet
Grammar
Sentence Generator
Sentence Generator Compiled
Feature Interpreter
Final Features

XSL + JavaC
Target Heuristic Values
ML
GP Search
Final Features
Features for GCC

- Start by dumping data structures to XML

GCC Exports Data Structures to XML

Structure Analysed

Grammar Created

Sentence Generator Compiled

Feature Search

Final Features Produced

Benchmarks

Program Data

Grammar

Stylesheet

Sentence Generator

Grammar

Sentence Generator

Feature Interpreter
Features for GCC

- Find out the structure found in the benchmarks
- Allows system to know data format without hard coding

GCC Exports Data Structures to XML

Structure Analysed

Grammar Created

Sentence Generator Compiled

Feature Search

Final Features Produced

Benchmark .c

Structure .xml

Grammar .xsl

Sentence Generator .bnf

Feature Search .class

Final Features .xml

XSL Engine

Grammar Stylesheet

XSL + JavaC

Sentence Generator

Feature Interpreter

ML

Target Heuristic Values

GP Search

Final Features

Find out the structure found in the benchmarks

Allows system to know data format without hard coding
Features for GCC

- Grammar is constructed from structure

 GCC Exports 
Data Structures to XML

 GCC Exports 
Structure Analysed

 Grammar Created

 Sentence Generator Compiled

 Feature Search

 Final Features Produced

 Grammar Stylesheet

 .xml

 .xsl

 .c

 .bnf

 .class

 XSL + JavaC

 XSL Engine

 XSL Engine

 Sentence Generator

 Feature Interpreter

 Final Features

 ML

 Target Heuristic Values

 GP Search

 Final Features

 Program Data

 .xml

 .xml
Features for GCC

- Grammar is constructed from structure
  - Huge grammar > 160kb
  - Ensures minimal useless features
  - Update easy if GCC changes
- Features are in interpreted language
Features for GCC

- Grammar compiled down to Java

---

### GCC Exports
- Data Structures to XML

### Structure Analysed
- Grammar

### Grammar Created
- Sentence Generator

### Sentence Generator Compiled
- Feature Search

### Final Features Produced

---

**Diagram:**
- GCC Exports to XML
- Structure Analysed
- Grammar
- Sentence Generator Compiled
- Feature Search
- Final Features Produced

**Steps:**
1. GCC Exports to XML
2. Structure Analysed
3. Grammar Created
4. Sentence Generator Compiled
5. Feature Search
6. Final Features Produced
Features for GCC

- Feature search
- Features are evaluated over benchmark data

 GCC Exports Data Structures to XML
 Structure Analysed
 Grammar Created
 Sentence Generator Compiled
 Feature Search
 Final Features Produced

Benchmark
 .c

GCC
 .xml

Grammar
 .sl

Grammar Stylesheet
 .bnf

XSL Engine
 .xml

Sentence Generator
 .class

XSL + JavaC

Feature Interpreter

Final Features

ML

Target Heuristic Values

GP Search

Final Features Produced
Features for GCC

- Final features outputted

GCC Exports Data Structures to XML

Structure Analysed

Grammar Created

Sentence Generator Compiled

Feature Search

Final Features Produced

- Benchmarks
- .c
- .xml

- GCC

- XSL Engine
- .xml
- .xsl
- Grammar
- .bnf
- XSL + JavaC
- .class

- GP Search
- Target Heuristic Values

- Feature Interpreter
- ML

- Final Features Produced
Overview

- Introduction to machine learning in compilers
- Difficulties choosing features
- A feature space for a motivating example
- Searching the feature space
- Features for GCC

Results

Further and on going work
Results

- Set up
  - Modified GCC 4.3.1
  - 57 benchmarks from MiBench, MediaBench and UTDSP
  - Pentium 6; 2.8GHz; 512Mb RAM
  - Benchmarks run in RamDisk to reduce IO variability
  - Found best unroll factor for each loop in [0-16]
Results

- Search set up
  - 100 parse trees per generation
  - Stop at 200 generations or 15 without change
  - Double Cross Validation

- Machine learning
  - Decision Trees (C4.5)
Results

- GCC default heuristic vs. oracle
- GCC gets 3% of maximum (1.05 speedup)
- On average mostly not worth unrolling
Results

- State of the art technique – Stephenson
- Hand designed features
- Uses support vector machine
Results

- GCC vs. state of the art vs. ours
- GCC 3%  Stephenson 59%  Ours 75%
- Automated features outperform human ones
Results

- Compare all with same machine learning
  - All with C4.5 decision trees
  - Level playing field
  - GCC 'features' are data used to compute heuristic
Results

- Decision Trees
- GCC 48%  Stephenson 53%  Ours 75%
- Automated features outperform human ones
Results

- Top Features Found
  - `get-attr(@num-iter)`
  - `count(filter(/*, !(is-type(wide-int) || (is-type(float extend) && [(is-type(reg)]/
    count(filter(/*,is-type(int))))) || is-type(union type))))`
  - `count(filter(/*, (is-type(basic-block) && 
      !@loop-depth==2 ||
      (0.0 > ( 
        (count(filter(/*, is-type(var decl))) -
        (count(filter(/*, (is-type(xor) && @mode==HI))) +
        sum(
          filter(/*, (is-type(call insn) && has-attr(@unchanging))),
          count(filter(/*, is-type(real type)))))))) /
        count(filter(/*, is-type(code label))))))))`
Overview

- Introduction to machine learning in compilers
- Difficulties choosing features
- A feature space for a motivating example
- Searching the feature space
- Features for GCC
- Results
- Further and on going work
Further and on going work

- Make all GCC's internals available
- Integrate to Milepost GCC plug-in system and open source it
- Features for multi-core optimisation

http://www.milepost.eu
Conclusion

- Shown a system which automatically finds good features
  - Searches a huge set of features
  - Allows greater experimentation in 'feature ideas'
- More flexible
  - A few feature grammars should service many heuristics
  - Retune features as well as heuristic
- Out performs expert features