IMPROVING COMPILER OPTIMIZATIONS
BY EMPLOYING MACHINE LEARNING

Raphael Mosaner – Johannes Kepler University Linz
Compiler Heuristics

*Metrics to decide which transformation / optimization to apply in what way*
Compiler Heuristics

Metrics to decide which transformation / optimization to apply in what way

if(\Delta \text{codeSize} \times \text{weight}_{\text{size}} < \Delta \text{performance} \times \text{weight}_{\text{perf}}) \rightarrow \text{doTransformation}()
Compiler Heuristics

Metrics to decide which transformation / optimization to apply in what way

heuristic

if(\(\Delta \text{codeSize} \times \text{weight}_{\text{size}}\) < \(\Delta \text{performance} \times \text{weight}_{\text{perf}}\)) \rightarrow \text{doTransformation}()
Compiler Heuristics

Metrics to decide which transformation / optimization to apply in what way

if ($\Delta \text{codeSize} \cdot \text{weight}_{\text{size}} < \Delta \text{performance} \cdot \text{weight}_{\text{perf}}$) \rightarrow \text{doTransformation}()
COMPILER HEURISTICS STATE-OF-THE-ART

Compiler Expert
COMPILER HEURISTICS STATE-OF-THE-ART

Heuristics creates Compiler Expert
COMPILER HEURISTICS  STATE-OF-THE-ART

- Heuristics implemented in Compiler
- Compiler creates Compiler Expert
COMPILER HEURISTICS STATE-OF-THE-ART

Heuristics implemented in Compiler evaluated on Benchmark Programs

creates

Compiler Expert
COMPILER HEURISTICS STATE-OF-THE-ART

Heuristics → implemented in → Compiler → evaluated on → Benchmark Programs
Heuristics creates Compiler Expert
Compiler Expert analyzed by
COMPILER HEURISTICS STATE-OF-THE-ART

Heuristics implemented in Compiler

evaluated on Benchmark Programs

creates / updates

analyzed by Compiler Expert
HUMAN-CRAFTED HEURISTICS
HUMAN-CRAFTED HEURISTICS

- Require domain expertise
HUMAN-CRAFTED HEURISTICS

- Require domain expertise

- Abstracts „real“ world to few benchmark programs
HUMAN-CRAFTED HEURISTICS

- Require domain expertise

- Abstracts „real“ world to few benchmark programs

- Often static / one-size-fits-all
HUMAN-CRAFTED HEURISTICS

- Require domain expertise
- Abstracts „real“ world to few benchmark programs
- Often static / one-size-fits-all
- Manual maintainance and updates
HUMAN-CRAFTED HEURISTICS

- Require domain expertise
- Abstracts „real“ world to few benchmark programs
- Often static / one-size-fits-all
- Manual maintenance and updates
- Error prone
HUMAN-CRAFTED HEURISTICS

- Require domain expertise
- Abstracts „real“ world to few benchmark programs
- Often static / one-size-fits-all
- Manual maintainance and updates
- Error prone

Compiler

Heuristics

Benchmark Programs

experience driven

JLU

27.02.2021
MACHINE LEARNED HEURISTICS  STATE-OF-THE-ART

Features → ML Blackbox → Heuristics

**Feature**

A feature is a measurable property of an object of interest.
(e.g., #branches, #memoryOperations)

**Target**

The target is the feature to be predicted.
(e.g., best optimization decision)
ML MODELS IN COMPILERS STATE-OF-THE-ART

Compiler

Benchmark Programs

User Programs

... Programs
ML MODELS IN COMPILERS STATE-OF-THE-ART
ML MODELS IN COMPILERS STATE-OF-THE-ART

- Compiler
- Benchmark Programs
- User Programs
- Features
- ML Model

Compiles -> produce -> used to train
ML MODELS IN COMPILERS STATE-OF-THE-ART

Compiler

Benchmark Programs

compiles

produce

User Programs

...

Programs

Features

used to train

ML Model

makes decisions in

ML Compiler
ML MODELS IN COMPILERS

Compiler

Benchmark Programs

User Programs

Programs

ML Model

compiles

makes decisions in

ML Compiler

Feature
ML IN COMPILERS

Compiler

Benchmark Programs

User Programs

… Programs

Features

ML Model

ML Blackbox

Features Target

Heuristics

ML Compiler
ML IN COMPILERS

- Captures large amount of „real“ world
ML IN COMPILERS

- Captures large amount of „real“ world
- (Semi-)automated
  - Waterfall
ML IN COMPILERS

- Captures large amount of „real“ world

- (Semi-)automated
  - Waterfall

- Black boxes
  - Lack maintainability & understandability
  - Hard to infer compiler knowledge
ML IN COMPILERS

- Captures large amount of "real" world

- (Semi-)automated
  - Waterfall

- Black boxes
  - Lack maintainability & understandability
  - Hard to infer compiler knowledge

- Online prediction has overhead
  - Crucial for dynamic compilation
ML IN COMPILERS

- Captures large amount of „real“ world
- (Semi-)automated  
  - Waterfall
- Black boxes  
  - Lack maintainability & understandability  
  - Hard to infer compiler knowledge
- Online prediction has overhead  
  - Crucial for dynamic compilation
UTILIZE MACHINE LEARNING ASSISTIVELY

- Human-crafted heuristics
- Maintainable
- One-size-fits-all
- Iterative
- Domain expertise

- Machine learned heuristics
- Compile time slowdown
- Black box
- Data-driven
- (semi-) automated
UTILIZE MACHINE LEARNING ASSISTIVELY

- Maintainable
- One-size-fits-all
- Domain expertise
- Iterative

- Compile time slowdown
- Black box
- Data-driven (semi-)automated

- Human-crafted heuristics
- Machine learned heuristics

fit all

UTILIZE MACHINE LEARNING ASSISTIVELY

- Maintainable
- One-size-fits-all
- Iterative
- Domain expertise
- Human-crafted heuristics

- Compile time slowdown
- Black box
- (Semi-) automated
- Data-driven
- Machine learned heuristics
UTILIZE MACHINE LEARNING ASSISTIVELY

- Maintainable
- Iterative
- One-size-fits-all
- Domain expertise

- Human-crafted heuristics

- Compile time slowdown
- Black box
- Data-driven
- (semi-) automated

- Machine learned heuristics

- Data-oriented

- (semi-) automated
COMBINED APPROACH

- Maintainable
- Iterative
- Data-driven
- (semi-) automated
COMBINED APPROACH

implemented in

Compiler

Heuristics

Compiler Expert

data-driven

maintainable

iterative

(semi-) automated
COMBINED APPROACH

- Heuristics implemented in Compiler
- Compiler compiles Programs
- Programs produce Features
- Features used to train/update ML Model
- Compiler Expert
- Maintainable, Data-driven, Iterative (semi-) automated
COMBINED APPROACH

- Heuristics implemented in Compiler
- Programs compiled
- Features produced
- ML Model assists Compiler
- Compiler Expert

Features used to train/update

- Data-driven
- Maintainable
- Iterative (semi-automated)
COMBINED APPROACH

Heuristics implemented in Compiler

Compiler compiles

ML Model assists

Features produce Programs

verify / fix

Compiler Expert

ML Model used to train/update

data-driven

iterative (semi-) automated

maintainable
COMBINED APPROACH
COMBINED APPROACH

- Avoid black boxes in parts crucial for understandability
COMBINED APPROACH

- Avoid black boxes in parts crucial for understandability

- Automated feedback based on environmental changes
  - Do current heuristics fit data?
COMBINED APPROACH

- Avoid black boxes in parts crucial for understandability

- Automated feedback based on environmental changes
  - Do current heuristics fit data?

- Infer knowledge for compiler experts
  - Helpful for problem analysis
  - Compensate lack in experience
COMBINED APPROACH

- Avoid black boxes in parts crucial for understandability

- Automated feedback based on environmental changes
  - Do current heuristics fit data?

- Infer knowledge for compiler experts
  - Helpful for problem analysis
  - Compensate lack in experience
CASE STUDY:
DUPLICATION IN THE GRAAL COMPILER
if (x > 0) {
    phi = x;
} else {
    phi = 0;
}
return phi + 2;
if (x > 0) {
    phi = x;
} else {
    phi = 0;
}
return phi + 2;

Copy code after control flow merges…
if \( x > 0 \) {
    \( \phi = x \);
} else {
    \( \phi = 0 \);
}
return \( \phi + 2 \);

DUPLICATION

Copy code after control flow merges into predecessor blocks …
if \( x > 0 \) {
\[
\phi = x;
\]
} else {
\[
\phi = 0;
\]
}

return \( \phi + 2 \);

Copy code after control flow merges… … into predecessor blocks … … to enable further optimizations.
CASE STUDY:
DUPLICATION IN THE GRAAL COMPILER
CASE STUDY: DUPLICATION IN THE GRAAL COMPILER

- **Heuristic** to trigger duplication:  \( \text{codeSize} \uparrow < \text{performance} \uparrow \) → duplicate
CASE STUDY:
DUPLICATION IN THE GRAAL COMPILER

- **Heuristic** to trigger duplication: \(\text{codeSize} \uparrow < \text{performance} \uparrow \) ? duplicate
  - \(\Delta\text{codeSize} = \sum_{\text{node}} \text{size}(\text{node}) \times \#\text{node}\)
  - Nodes are manually annotated with their size (= cost)
CASE STUDY: DUPLICATION IN THE GRAAL COMPILER

- **Heuristic** to trigger duplication: \( \text{codeSize} \uparrow < \text{performance} \uparrow \) ? duplicate
  - \( \Delta \text{codeSize} = \sum_{\text{node}} \text{size}(\text{node}) \times \#\text{node} \)
  - Nodes are manually annotated with their size (= cost)

![Diagram of code duplication]

\[ \text{size} = 5 \]
CASE STUDY: DUPLICATION IN THE GRAAL COMPILER

- **Heuristic** to trigger duplication: \( \text{codeSize} \uparrow < \text{performance} \uparrow \Rightarrow \text{duplicate} \)
  - \( \Delta \text{codeSize} = \sum_{\text{node}} \text{size}(\text{node}) \times \#\text{node} \)
  - Nodes are manually annotated with their size (= cost)

- **Hand crafted cost model** for over 450 different IR nodes
  - Code size
  - Execution cycles
**CASE STUDY:**
**DUPICATION IN THE GRAAL COMPILER**

- **Heuristic** to trigger duplication: \( \text{codeSize} \uparrow < \text{performance} \uparrow \) ? duplicate
  - \( \Delta \text{codeSize} = \sum_{\text{node}} \text{size}(\text{node}) \times \#\text{node} \)
  - Nodes are manually annotated with their size (\( = \) cost)

- Hand crafted **cost model** for over 450 different IR nodes
  - Code size
  - Execution cycles

- Node costs are only estimations made from experience
GOAL:
VERIFY OR FIX NODE COST MODEL
GOAL:
VERIFY OR FIX NODE COST MODEL

- Learn "actual" code size impact per IR node based on data
GOAL:
VERIFY OR FIX NODE COST MODEL

- Learn “actual” code size impact per IR node based on data

- Features: IR node counts
  - [#AddNode, #SubNode, #IfNode, …]
GOAL: VERIFY OR FIX NODE COST MODEL

- Learn „actual“ code size impact per IR node based on data

- Features: IR node counts
  - [#AddNode, #SubNode, #IfNode, …]

- Target: code size in bytes
GOAL: VERIFY OR FIX NODE COST MODEL

- Learn "actual" code size impact per IR node based on data

- Features: IR node counts
  - [#AddNode, #SubNode, #IfNode, …]

- Target: code size in bytes

<table>
<thead>
<tr>
<th>ID</th>
<th>InstalledCodeSize</th>
<th>#ConstantNode</th>
<th>#AddNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10004</td>
<td>552</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10077</td>
<td>480</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10170</td>
<td>608</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10243</td>
<td>552</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10251</td>
<td>512</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>bigfib.cpp_1_HotSpotCompilation-10411</td>
<td>752</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
PROBLEM: SUBSEQUENT COMPILER PHASES

Compiler PhaseY Duplication PhaseZ

Predictor

influencing compiler phase

Machine Code
PROBLEM: SUBSEQUENT COMPILER PHASES
PROBLEM: SUBSEQUENT COMPILER PHASES

- Transformations on IR level wreck linear relation between nodes and final code size
PROBLEM: SUBSEQUENT COMPILER PHASES

- Transformations on IR level wreck linear relation between nodes and final code size
- Linear regression model mispredicts code size impact
PROBLEM: SUBSEQUENT COMPILER PHASES

- Transformations on IR level wreck linear relation between nodes and final code size

- Linear regression model mispredicts code size impact

- Requires non-linear predictor to account for intermediate compiler phases
ANN PREDICTOR

- Trained a simple ANN on benchmarks
  - dacapo, scala-dacapo, octane,
    jetstream, renaissance
ANN PREDICTOR

- Trained a simple ANN on benchmarks
  - dacapo, scala-dacapo, octane, jetstream, renaissance

Prediction Accuracy

# evaluated methods (relative)

relative error
ANN PREDICTOR

- Trained a simple ANN on benchmarks
  - dacapo, scala-dacapo, octane, jetstream, renaissance

- Accurately predicts code size impact
  - 3 out of 4 predictions have errors <10%
ANN PREDICTOR

- Trained a simple ANN on benchmarks
  - dacapo, scala-dacapo, octane, jetstream, renaissance

- Accurately predicts code size impact
  - 3 out of 4 predictions have errors <10%

- Implemented a prototype predictor in the Graal compiler
PRODUCING HELPFUL OUTPUT

- Analysis mode in Graal
  - Prints differences in duplication decisions based on human model vs. learned model
  - Results provided to compiler expert

Duplication decisions differ
- Function
- Nodes
- Code size (human heuristic)
- Code size (ANN)

Compiler Expert
BENCHMARK PERFORMANCE (SELECTION)

Octane

Jetstream

- Default
- Fixed
- ML
- NoDup

Richards, Gameboy, TypeScript, zlib

container, towers
IMPROVING COMPILER OPTIMIZATIONS
BY EMPLOYING MACHINE LEARNING

QUESTIONS?
raphael.mosaner@jku.at
www.ssw.jku.at/General/Staff/Mosaner