Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters - An Architecture Overview

Linz 13.09.2018

Advisors: Prof. Dr. Dr. h.c. Hanspeter Mössenböck, Prof. (FH) Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller
How to not use Lists (Java)

Listing 1: Running Example - for loop calculating the sum of all values in a list of positive integers

```java
int x = 0;
List<Integer> a = listOfPositiveInt();
for (int i = 1; i < a.size(); i++) {
    x += a.get(i);
}
```
Outline

Introduction

Background

Architecture Overview

Future Work

Conclusion
Outline

Introduction

Background

Architecture Overview

Future Work

Conclusion
Introduction I

Code

Figure: Code Interpretation and Compilation with GP (Truffle and Graal images from [1])
Introduction II

Optimizing Code for the Compiler

- Use Abstract Syntax Tree (AST)
- Keep semantic (in - out)
- Alter syntax (nodes)
- Language specific
Introduction III

Possible Applications

- Runtime optimization of (unforeseen) bottlenecks
- Write generalized code - specialize automatically
  - Increase runtime performance
  - Decrease memory usage
  - Reduce energy footprint
Outline

Introduction

Background

Architecture Overview

Future Work

Conclusion
Background I

Graal [2, 3, 4]

- Just-in-time (JIT) compiler
- Aggressive optimizations
- Part of OpenJDK
Background II

Truffle [5, 1]

- Interpreter based on AST
- Generalized and Specialized AST - Nodes
- Self optimizing by AST-Rewriting
- Can run any Guest Language on the JVM
- Python, Ruby, JavaScript, ...
Background III

Genetic Improvement

- Search Based Machine Learning - Primarily Genetic Programming
- Breeding Solutions with small changes towards a goal
- ≠ Superoptimization (usually Brute Force or Random Search)
Outline

Introduction

Background

Architecture Overview

Future Work

Conclusion
Optimization Process

Figure: Offline Pattern Discovery and Online Optimization
Figure: The architecture of the optimization framework in combination with Truffle and Graal
Language Analysis

Truffle Language Information (TLI)

- Truffle Language Analyzer (TLA) loads all nodes from Classpath -> Generates TLI
- Terminal / Non-Terminal Nodes
- Mappings from *abstract* class to implementations
- Generic initializer for all nodes
Stochastic Optimization

Experiments

Let's developer set goals (performance, memory, ...)

Defines test cases for correctness (1)

\[
\text{correctness}_{(\text{AST}, \text{tests})} = \frac{\sum_{n=1}^{\text{tests}} \text{succeeded}_{(\text{AST}, \text{test})}}{\sum \text{tests}}
\]  

Optimizer

- Solves Experiment Using Truffle Language Information (TLI)
- Performance Measurement averages over 200,000 runs (first 100,000 discarded for warmup)
Stochastic Optimization Example

for to foreach optimization (running example)

Truffle AST

```
size a = i
++ i
{} +=
get a x
```

Optimized AST

```
each obj a +=
```

Page 16 | 25
Pattern Discovery

Knowledge Base

- information on language (TLI)
- experiments and results
- found patterns for language

Truffle Pattern Detector

- Pattern mining of experiment results
- IN - OUT Replacement patterns
- Including Wildcards (..)
- Patterns can be hierarchical
Pattern Discovery Example

for to foreach pattern (running example)

AST IN

```
for
{}
...
[]
a
for (..){
a[i]; ..
```

replace

```
AST OUT

for
each
{}
obj
a
for (Double obj : a){
obj; ..
```
Pattern Application

Truffle Pattern Injector

- Applies patterns to code
- Application after parser
- Application before interpretation by Truffle
- Application by dispatch interception (ex. often called functions, bottlenecks)
Outline

Introduction

Background

Architecture Overview

Future Work

Conclusion
Much to do

- Extending optimization framework with automated test generation
- HeuristicLab as external optimizer
- Pattern Framework only conceptual
Evaluation

- PolyBench/C [6]
  - numerical computations
  - testing outcome preservation

- CortexSuite [7]
  - machine learning / computer vision
  - optimization potential
  - runtime <-> exactness

- DaCapo [8]
  - Java
  - higher level language
  - context preservation

- Case Studies - Testing with open source / research projects
Conclusion

- Architecture for stochastic code optimizations
- Language specific patterns
- KnowledgeBase enables Optimiation an Patterns
- Supplementary to already existing optimizations
Contact

Oliver Krauss
oliver.krauss@fh-hagenberg.at

+43 (0)50804-27195
Bibliography I


Bibliography II


Bibliography III


Appendix 1a - Genetic Programming

Figure: The basic GP Algorithm as defined in [9]
Appendix 1b - Genetic Programming in depth I

Note: Example taken from [10]

Configure Execution

Terminal Set: Var x; Const -5 .. +5
Function Set: + - * /
Fitness Function: \( F = |(x^2 + x + 1) - \text{RESULT}| \)
Run Parameters: Crossover; Mutator; Max-Tree-Depth; ...
Termination Criteria: F <= 0.1; Max-Generations = 100; ...
Appendix 1b - Genetic Programming in depth II

Generate Initial Population (randomly)

1. \( x + 1 \)
2. \( x^2 + 1 \)
3. \( 2 + 0 \)
4. \( x - 1 \)
Appendix 1b - Genetic Programming in depth III

Generate child by Elitism

Parent Individual

\[
\begin{align*}
x & \rightarrow + \rightarrow 0 \\
& \rightarrow - \\
x + 1 & 
\end{align*}
\]

Child Individual

\[
\begin{align*}
x & \rightarrow + \rightarrow 0 \\
& \rightarrow - \\
x + 1 & 
\end{align*}
\]
Appendix 1b - Genetic Programming in depth IV

Generate child by Crossover

**Parent Individuals**
- \( x + 1 \)
- \( x^2 + 1 \)

**Child Individuals**
- \( x \)
- \( x^2 + x + 1 \)
Appendix 1b - Genetic Programming in depth V

Generate child by Mutation

Parent Individual

Child Individual

mutate
Appendix 1b - Genetic Programming in depth VI

GP - The Next Generation

\[
\begin{align*}
\frac{x}{x+1} & \rightarrow x + 1 \\
\frac{x}{x} & \rightarrow x \\
\frac{x}{x^2 + x + 1} & \rightarrow x^2 + x + 1 \\
\frac{x}{0} & \rightarrow 1
\end{align*}
\]
Appendix 2 - JIT Compilation I

Figure: Code Interpretation and Compilation (Truffle and Graal images from [1])
Appendix 2 - JIT Compilation II

Truffle offers a variety of optimizations [1]

- Type Specialization
- Local Variable Access Optimization
- Branch Probability Optimization
- Runtime Call Inlining
- ...

[1] Reference link
Appendix 2 - JIT Compilation III

Figure: Node rewriting in Truffle [1]
for replaced with foreach

```
for (i = 0; i < a.size; i++){
    x += a[i]; ..
```

```
for (Double obj : a){
    x += obj; ..
```
Appendix 3 - Additional Examples II

recursion to iteration

int fib(int n) {
    if (((n == 1) || (n == 0))) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

int fib(int n) {
    int x = 0, y = 1, z = 1;
    for (int i = 0; i < n; i++) {
        x = y; y = z; z = x + y;
    }
    return x;
}
Appendix 3 - Additional Examples III

improved approximation (newton raphson)

double sqrt ( double c) {
    double eps = 0.001;
    double t = 0;
    while (t * t < c) {
        t = t + eps;
    }
    return t;
}

AST from Coco/R

AST produced by GP

double sqrt ( double c) {
    double eps = 0.001;
    double t = c;
    while (abs(t - c/t) > eps*t) {
        t = (c/t + t) / 2.0
    }
    return t;
}
for pattern error and update

for (i = 0; i < 1; i += 2){
    x += a[i]; ..

for (..){
    a[i]; ..
Appendix 4 - Patterns contd. II

for to foreach with collection access

AST from Coco/R

for (i = /zero.pnum; i < a.size; i++){
  x += a[i]; a.del[i]; ..
}

AST produced by GP

{ List<Double> cpy = a.cpy;
  for (Double obj : cpy){
    x += obj; a.del(obj); ..
  }
}
Appendix 4 - Patterns contd. III

for to foreach with collection access discovery

AST from Coco/R

for (i = 0; i < a.size; i++){
    x += a[i]; a.del[i]; ..
}

AST produced by GP

{ List<Double> cpy = a.cpy;
for (Double obj : cpy){
x += obj; a.del(obj); .. }
}
for to foreach collection access pattern

AST IN

```
for (..){
a[i]; a.del ..
```

AST OUT

```
for (Double obj : a){
obj; a.del(obj)..}
```
Appendix 4 - Patterns contd. V

Pattern hierarchy

AST IN

```
for
{}
...

a
```

for (..){
a[i]; a.del ..
}

supercede

AST IN

```
for
{}
...

a
```

for (..){
a[i]; ..
}