IMPROVING PROGRAMS THROUGH SOURCE CODE TRANSFORMATIONS

Dissertation Proposal
Jonathan Dorn
April 19, 2016
Beyond Functional Correctness

• Software development involves *tradeoffs*.
  
  • “Fast, good, or cheap. Pick any two.”
  
  • Time vs. memory.
  
  • Maintainability vs. efficiency.
  
  • ...

float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalves = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = *(long*) &y;                        // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );           // what the ?

    y = *(float*) &i;
    y = y * ( threehalves - ( x2 * y * y ) ); // 1st iteration

    // y = y * ( threehalves - ( x2 * y * y ) ); // 2nd iteration, this can be removed

#ifdef Q3_VM
    #ifdef __linux__
        assert( !isnan(y) ); // bk010122 - FPE?
    #endif
#endif

    return y;
}
Non-Functional Properties

• “How good” instead of “what” [Paech 2004].
  • “More” or “less;” “higher” or “lower.”

• Difficult to reason about (e.g., security).

• Characterize implementations by how much of a property they possess.
Non-Functional Tradeoffs
Non-Functional Tradeoffs
Quantifying Non-Functionality

- Different metrics for different properties.
  - **Image quality**: RGB distance (e.g., $L^2$), SSIM, EMD.
  - **Runtime**: seconds, speedup/slowdown.
  - **Energy efficiency**: joules, watts.
  - **Maintainability**: bug fix time, defect density.
  - **Correctness**: % error, accuracy, precision, PSNR.
Local Changes

• **Small** changes can have **large** effects.
  • E.g., `bubble_sort(a) → quick_sort(a)`.

• Option of **fine-grained** control.

• Program lines, statements, AST nodes.
Proposal Thesis

By applying *local software transformations*, we can select better tradeoffs between *non-functional properties* of existing software artifacts.
The rest of this proposal

• Overview of the proposed research thrusts
  • Visual error and runtime performance
  • Energy usage
  • Coding style
• Proposed research timeline
• Conclusion
The rest of this proposal

• Overview of the proposed research thrusts
  • Visual error and runtime performance
  • Energy usage
  • Coding style
• Proposed research timeline
• Conclusion
Computer Generated Imagery

• 11% of all tickets in 2015 went to computer animated movies.*
  • Does not include live movies with CGI.
• Video games topped $90B in 2015.**

* http://www.boxofficemojo.com/
** http://www.gamesindustry.biz/articles/2015-04-22-gaming-will-hit-usd91-5-billion-this-year-newzoo
Visual Error and Runtime Performance
Hypothesis

• We can apply local changes to the abstract syntax tree of a graphics program to produce an approximation that is:
  • Visually faithful to the original and
  • Efficient to compute.

• Evaluate both image quality and runtime.
Simple Example

```
return
(color)
-
floor
+
p.x
floor
```

```
p.x 0.5
```
Simple Example

```
return (floor + p.x) / floor
```

```
return (color) floor
```

```
return (floor + p.x) / floor
```

```
return (floor + p.x) / floor
```
Simple Example

```
return
  (color)
    -
      floor
        +
          p.x
          0.5

floor
```

Before

After
Approach

• Transformation: Replace node N with N’.
• Determine replacements offline (manual).
• Genetic search to select nodes to replace.
  • Use image quality as fitness function.
Experimental Setup

• Benchmarks chosen from previous work.
• Record **runtime** and **image quality**.
• Three data points for each benchmark:
  1. Original program.
  2. Baseline “slower but less error” approach.
  3. Best transformed variant from our search.
Runtime Results

![Runtime Results Chart]

- **Baselines**
  - step
  - ridges
  - pulse
  - noise1
  - checker
  - circles1
  - wood
  - brick
  - noise2
  - circles2
  - perlin

- **Our Approach**

---

23
Image Quality Results

![Image Quality Results](image)

Baseline vs. Our Approach for various patterns and textures.
Summary

• We can apply local changes to produce programs that:
  • Are significantly *faster* than the baseline approach,
  • Have *less error* than the original program, and
  • Often have *less error* than the baseline.
Outline

• Overview of the proposed research thrusts
  • Visual error and runtime performance
  • Energy usage
  • Coding style
• Proposed research timeline
• Conclusion
Data Center Energy Use

Percentages of US electricity use in a given year

Electricity Use (billion kWh/year)

<table>
<thead>
<tr>
<th>Year</th>
<th>Infrastructure</th>
<th>Communications</th>
<th>Storage</th>
<th>High-end servers</th>
<th>Mid-range servers</th>
<th>Volume servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0.8%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>1.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td>2.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Genetic Optimization Algorithm

- Local changes to assembly code.
- Tradeoff between reduced energy and relaxed semantics.
  - Validated with test suite.
Genetic Optimization Algorithm

• Local changes to assembly code.

• Tradeoff between reduced energy and relaxed semantics.
  • Validated with test suite.
Scaling to Larger Programs

![Graph showing energy reduction vs lines of code]

- Energy Reduction (%)
- Lines of Code (Assembly)
Hypothesis

By directing the genetic search more effectively and \textit{reducing} the search space, we can achieve \textit{larger} energy optimizations \textit{faster}.

Evaluate both magnitude of optimization and search time.
Intuition

• Optimizations on different paths through the program are likely to be independent.
  • *Combine* optimizations from separate searches.

• Optimizations on frequently executed paths are likely to have larger impact.
  • *Profile* the program to target hot paths.
Algorithm Overview

**INPUT**

**EVALUATE FITNESS**

- Evaluate fitness
- Eject
- Report

**MUTATE**

**OUTPUT**

- Mutate
- Eject
- Report
Optimizing Two Workloads
Option 1: Share Variants During Search
Option 2: Combine Best After Search
Experimental Setup

Benchmarks
- Collect HPC and data center benchmarks.
- Collect multiple workloads for each benchmark.

Baseline: GOA search
1. Only one workload.
2. All workloads in single fitness function.
Metrics for Energy Optimization

• **Energy** measured at the wall.

• **Wall time** before best variant.
  • Latest best variant if combining after search.

• **Fitness evaluations** before best variant.

• Success if searching separately produces larger energy reduction across all workloads.
Preliminary Results

Energy Improvement

- set 0
- set 1
- combined
- GOA

0% 10% 20% 30% 40% 50%
Outline

• Overview of the proposed research thrusts
  • Visual error and runtime performance
  • Energy usage
  • Coding style
• Proposed research timeline
• Conclusion
Programmer Time

• Programmer salaries in the U.S. exceed $100B.

• Programmers spend much more time reading code than writing it.

Stylish Code

• *Broad consensus* for standardized coding style.

• *Persistent disagreement* on specifics.
  
  • E.g., tabs vs. spaces.

• “Every major open source project has its own style guide.” – Google’s style guide.
Beacons

• Indicate likely structure or functionality.
• Semantic or syntactic.
• May vary
  • Between programmers,
  • And over time.

```c
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    ...
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
    ...
  }
}
```
Beacons

• Indicate likely structure or functionality.

• Semantic or syntactic.

• May vary
  • Between programmers
  • And over time.

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        ...  
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        ...
    }
}```

Possible sort Implementation?
Beacons

• Indicate likely structure or functionality.

• Semantic or syntactic.

• May vary
  • Between programmers,
  • And over time.

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        ...
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        ...
    }
}
```

End of scope.
Classification of Coding Style

• Typographic and Structural [Oman 1988].
  
  • Typographic: whitespace, line length, identifier length, layout.
  
  • Structural: modularity, level of nesting, control and information flow.
Classification of Coding Style

• Typographic and Structural [Oman 1988].
  
  • Typographic: whitespace, line length, identifier length, layout.
  
  • Structural: modularity, level of nesting, control and information flow.
Hypothesis

• We can apply local changes to the typographic elements of source code to
  • Match a programmer’s expected style and
  • *Improve* their understanding of the code.

• Evaluate time and accuracy on tests of understanding.
Modeling Typographic Style

• *N*-gram language model.
  • Uses previous *n*-1 tokens to predict next token.
  • Learn probabilities from existing code.
  • **NATURALIZE** framework [Allamanis 2014].
  • Can predict or suggest whitespace.
Similarity of Typographic Style

• Measure similarity of $N$-gram models.
  • $N$-gram models are probability distributions.

• Measure similarity of style-checker rules.
  • Allamanis et al. generate rules from $n$-gram models.
Experimental Setup

Benchmarks
1. Reformat the same code in different ways.
2. Collect similar code from different authors (e.g., textbook examples).

Participants
- Undergraduate student volunteers from upper level electives.
- Amazon Mechanical Turk workers who pass a screening test.
Human Study

1. Identify written style.
   • Participants write code to accomplish simple tasks.
   • E.g., check that a list is sorted.

2. Perform maintenance tasks.
   • Participants answer questions about code examples.
   • E.g., what is the value of x on line 5?
Metrics for Program Understanding

• Collect *similarity* between code participants wrote and the code samples.

• Collect *time* and *accuracy* in answering questions.

• Measure correlation between similarity and time and between similarity and accuracy.
Outline

• Overview of the proposed research thrusts
  • Visual error and runtime performance
  • Energy usage
  • Coding style

• Proposed research timeline

• Conclusion
Research Timeline

- Energy Optimization (RT2)
- Coding Style of Tests (RT3)
- Coding Style in Guava (RT3)
- Shader Acceptability (RT1)
- Energy Optimization at Scale (RT2)
- Coding Style Acceptability (RT3)

Legend:
- Research Period
- Publication Lag

Timeline:
- Jan '13
- Jan '14
- Jan '15
- Jan '16
- Jan '17
- Jan '18

Today
Graduation
Conclusion

Enable better tradeoffs between non-functional properties through local software transformations.

2. Energy usage.
3. Coding style.
BACKUP
Does Coding Style Matter?

IF $A > B$ THEN
  $S := 1$
ELSE IF $A = B$ THEN
  IF $C > D$ THEN
    $S := 2$
  ELSE
    $S := 3$
  END IF
ELSE IF $C > D$ THEN
  $S := 4$
ELSE IF $C = D$ THEN
  $S := 5$
ELSE
  $S := 6$
END IF

Reproduced from P. W. Oman and C. R. Cook.
Does Coding Style Matter?

Reproduced from P. W. Oman and C. R. Cook.