Automated Test Program Generation for an Industrial Optimizing Compiler

Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, Zhaohui Wang

Institute of Software, Chinese Academy of Sciences

June 2009

The 2nd ISCAS/USC/UMass Joint Workshop on Trustworthy Software Systems
Beijing, China
Outline

• Motivation

• Brief Introduction to Compiler Optimizations

• JTT: Automated Tool for Test Program Generation

• Formal Model-based Generation: From Formula to Program

• Evaluation in Testing UniPhier Compiler

• Contributions
Motivation

UniPhier: A popular, high-performance, embedded chip from Panasonic

All optimizations must be TRUSTABLE!
What is compiler optimization?

“Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable computer program.” ——Wikipedia

Well-Known Optimizations:

Dead Code Elimination, Common Subexpression Elimination, Loop Interchange, Loop Unrolling…

\[
\begin{align*}
a &= b \times c + g; \\
d &= b \times c \times d;
\end{align*}
\]

\[
\begin{align*}
tmp &= b \times c; \\
a &= tmp + g; \\
d &= tmp \times d;
\end{align*}
\]
Compiler Optimizations (Cont’d)

• **General Form**

\[ I \rightarrow I' \text{ if } \{\text{Side Conditions}\} \]

• **Features**
  – Semantic-preserving
  – Many possible prerequisites
  – Dependent to data types
How to Test An Optimization

• Basic approach
  Find a program to active the optimization

Find this program

P₁  P₁  P₁  P₁  P₂  P₂  P₂

OPT₁  OPT₂  OPT₃  OPT₄  ......  OPTᵣ

Is P₂ semantic equivalent to P₁?

• Requirements for test programs
  – Definitely active specific optimization
  – Cover as many prerequisites as possible
  – Use various data types
  – Interleave optimizations and data types
Extra Difficulties in Testing UniPhier Compiler

Architecture-dependent optimizations
→ not common optimizations
→ special program to active such optimizations

Architecture-dependent data types
→ not common data types
→ special program to use such data types

Especially when such optimizations and data types are interleaving with each other!
# Possible Solutions

<table>
<thead>
<tr>
<th>Solution</th>
<th>What’s it?</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>SPEC Benchmarks, EEMBC, etc</td>
<td>Popular</td>
<td>Performance-oriented, not optimization-oriented; without UniPhier-related features</td>
</tr>
<tr>
<td>Public Test Suite</td>
<td>GCC Test Suite and so on</td>
<td>Public</td>
<td>Language-oriented, not optimization-oriented; without UniPhier-related features</td>
</tr>
<tr>
<td>Internal Test Suite by Panasonic</td>
<td>Legacy of past testing in Panasonic</td>
<td>Optimization-oriented, Concerning both common and UniPhier-dependent optimizations</td>
<td>Not available for new architecture features, new optimizations and new data types</td>
</tr>
<tr>
<td>Manual Programming</td>
<td>Test program written by test engineers</td>
<td>Optimization-oriented, concerning all optimizations, be aware for evolution of UniPhier and its compiler</td>
<td>Labor-intensive, test efficiency and quality are difficult to achieve</td>
</tr>
</tbody>
</table>
JTT: An Automated Tool for Test Program Generation

HLS (High Level Script): specify test requirements, including optimizations, data types used, statement types, operators and so on. Readable for test engineers.

MLS (Middle Level Script): specify test program template, including program statement template, data types, operators and so on

H2M: transform HLS to MLS  M2P: transform MLS to test program
H2M is based on formal model of compiler optimizations
HLS Script & MLS Script

An Example HLS Script

A Possible MLS Script

- **Advantage of Two-Level Script**
  - Separate test program requirements and test program specification
  - Hide details of test program and architecture in MLS
  - Prerequisites and data types can be interleaved easily
  - Thousands of test programs with respect to HLS can be generated automatically and easily
Model-based Generation: from Formula to Program

CTL Formulas for Optimizations: Computation Tree Logic (CTL) Formulas (only name listed in HLS)

NCG: Node Control Graph, a special CFG

Construction of CFG: enrich complexity of resulting code

Expansion of Temporal Quantifiers: make all nodes in CFG only labeled with first-class formulas
CTL Formulas for Compiler Optimizations

A conditional rewriting rule containing CTL formulas for Dead Code Elimination

LHS specifies some statements in test program
Side conditions specify why an optimization is semantic-preserving – a relationship between LHS and other statements in test program

LHS + Side Conditions → Test Programs
Node Control Graph

- A special CFG
  - Constructed according to CTL Formula
    * X operator leads to a first-successor node and a solid edge to current node
    * U operator leads to a successor node and a dashed edge to current node
  - Each dashed edge can be replaced by a basic block
  - Each edge is labeled with a formula for all or some paths in the basic block
  - Each node is labeled with a formula for statement at the node
• Replace each dashed edge with a basic block

The basic blocks are generated according to constraints specified in HLS
Expand temporal quantifiers to all involved nodes

Make all nodes only labeled with first-class formulas for convenience of generation

Different expansion for A/E path quantifiers and G/F state quantifiers
Example Generation

\[
\begin{align*}
&n : (x := e) \Rightarrow \text{remove statement } n \\
&\vdash \neg \text{EX}(f(x) \cup \text{use}(x) \land \text{mode}(n))
\end{align*}
\]

(a) DCE Formula
Data Dependence Generation for Loop Optimizations

- Assign flow-, anti-, and output-dependence to statements within loop nests
  
  - **Steps:**
    1. Assign valid types of data dependence to statements
    2. Generate proper subscription function according to data dependence (Linear Integer Programming)
Evaluation in UniPhier Compiler

- **JTT v.s. Benchmark & Public Test Suite**

<table>
<thead>
<tr>
<th>Module</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark &amp; Public Test Suite</td>
<td>85%</td>
<td>18%</td>
<td>79%</td>
<td>0%</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>With JTT</td>
<td>88%</td>
<td>54%</td>
<td>80%</td>
<td>63%</td>
<td>49%</td>
<td>68%</td>
<td>56%</td>
</tr>
</tbody>
</table>

  Statement Coverage of 7 Optimizing Modules in UniPhier Compiler

- **JTT v.s. Manual Programming**
  
  - 6,000+ test programs v.s. 1,000+ test programs

- **Bugs Found**
  
  - 6 new serious bugs found by JTT
  
  - Found 21% bugs by JTT during testing
Contributions

- Model-based test program generation for optimizing modules
- Automated script-driven test case generation for a large system with complex inputs
- Application of formal methods in a large complex system
Thanks for Your Attention!
Any Question?

Yunzhi Xue

yunzhi@itechs.iscas.ac.cn

Institute of Software, Chinese Academy of Sciences