Checking Correctness of Code Generator Architecture Specifications

Niranjan Hasabnis, Rui Qiao and R. Sekar
Stony Brook University, NY
International Symposium on Code Generation and Optimization (CGO) 10 February, 2015

Introduction
Problem

Problem: code generator bugs

1. Modern instruction sets are complex
   - Huge instruction set references!
2. Code generators model instruction set semantics **manually**.

A typical compiler backend

We target semantic modelling bugs.

Code generator semantic modelling bugs

- Semantic modelling bug = semantic inconsistency
  - **Semantic inconsistency**: soundness of \( \langle I, A \rangle \)
  - Soundness of \( \langle I, A \rangle \): \( \text{Semantics}(I) \supseteq \text{Semantics}(A) \).
- Bug statistics
  - GCC: > 40 architectures.
  - All architecture specs: **400K** lines.
  - x86: **30K** lines.
  - In 1 year, **25 bugs** resulted in updates to GCC’s x86 code generator
- What are compilers doing?
  - End-to-end testing

Existing practices to detect modelling bugs

- **Compiler verification**
  - CompCert\(^1\) promising work (modern compilers?)
  - Applied to optimizers (e.g., llvm, gcc).. for code generators?
  - **Equivalence checking**: needs modelling of assembly semantics
- **Translation validation**
  - GCC’s optimization passes (Necula et al.\(^2\), Tristan et al.\(^3\))
  - Needs modelling of assembly instructions (we want to avoid)
- **Compiler testing**
  - CSmith\(^4\) randomly generate valid C programs
  - **End-to-end test suites**

---


\(^3\) Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating Value-graph Translation Validation for LLVM. In PLDI, 2011.

Our approach

Targeted testing of code generators for semantic modelling bugs

1. Start state generation: how many states? and how to generate?
2. Obtain \((I, A)\) pairs
3. Develop compiler- and architecture-neutral approach

Contributions

1. Novel approach to check correctness of code generators
2. Architecture-neutral, compiler-neutral approach
3. Practical: easy integration with compiler development cycle
4. Evaluation
   - GCC’s x86 code generator for basic and SSE instructions (140 in total)
   - Found 7 soundness violations
   - 1 was confirmed as a bug in latest GCC. (GCC accepted and fixed it).
   - Verified against 15 known bugs

Approach Details

ArCheck (Architecture Checking) design

Code generators = architecture specifications

Obtaining \((I, A)\) pairs

- Compiler-specific approach
- Our approach: treat compiler as blackbox

Compiler-neutral approach = compiler as a blackbox
Start state generation: objectives and approach

- Generating all possible start states is practically infeasible
- Instead of verification, we perform testing

**Objective:** generate start states such that

1. **Increase** the possibility of finding soundness violations
2. **Reduce** the number of states such that testing is practically feasible
3. Architecture-neutral

**Approach**

1. Test all of the “components” of the instruction semantics (“interesting outcomes”)
2. Select sufficient number of test cases for each component (“interesting input”)
3. White-box analysis of IR

---

"interesting outcomes" and "interesting inputs" satisfy objective 1 and 2.

---

Obtaining “interesting inputs”

- We define “interesting outcomes” for all IR operators and types combination.

- Type (1-byte vs 4-byte, etc), Usage (value used in memory reference or immediate)

Problem of obtaining “interesting inputs” = constraint satisfaction problem
Test execution and result comparison

- “CPU” for IR: **IR interpreter**
- Assembly execution system: **user-level process monitoring framework**
- Memory semantics
  - Compare pre- and post-execution snapshots
- **Optimization**: compare-on-write
- Result comparison
  1. **Strict-equivalence** checking: $\text{Semantics}(I) = \text{Semantics}(A)$
  2. **Loose-equivalence** checking: $\text{Semantics}(I) \supseteq \text{Semantics}(A)$

Evaluation

- Tested GCC-4.5.1 x86 code generator for basic (80) and SSE (60) instructions
- **Strict-equivalence** checking: 39 soundness violations
- **Loose-equivalence** checking: 7 soundness violations
- found a bug in latest GCC. Bug was accepted by GCC and fixed.
- Verified against 15 known bugs in older GCC

<table>
<thead>
<tr>
<th>Description</th>
<th>Mnemonic</th>
<th>Template</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Mapping Rules</td>
<td>140</td>
<td>1132</td>
<td>150K</td>
</tr>
<tr>
<td># of Test Cases</td>
<td>1056</td>
<td>5762</td>
<td>421,090</td>
</tr>
<tr>
<td>% of Useful Test Cases</td>
<td>92%</td>
<td>59%</td>
<td>52%</td>
</tr>
<tr>
<td>Time to Run Test Cases</td>
<td>5 mins 7 sec</td>
<td>4 mins 10 sec</td>
<td>1 day 1 hrs</td>
</tr>
</tbody>
</table>

Niranjan Hasabnis, Rui Qiao and R. Sekar

shrdl bug found in GCC

Bug: [https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61503](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61503)

```
shrdl $16, %ebx, %eax
  (set (reg:SI eax)
   (ior:SI
    (ashift:SI (reg:SI eax) (const_int 16))
    (ashift:SI (reg:SI ebx)
     (minus:QI (const:QI esk) (const_int 16))))
   (clobber (reg EFLAGS)))
```

- Arithmetic vs logical right shift
- Start state: eax = 0xb72f60d0, ebx = 0xbfcbd2c8,
- Assembly semantics: eax = 0xd2c8b72f,
- RTL semantics: eax = 0xffffb72f.

GCC accepted the bug and fixed promptly.

movzwl soundness violation

```
movzwl 8(%esp), %eax
  (set (reg:HI ax)
   (mem:HI
    (plus:SI (reg:SI esp)
     (const_int 8)))

movzbl 8(%esp), %eax
  (set (reg:QI ax)
   (mem:QI
    (plus:SI (reg:SI esp)
     (const_int 8)))
```

Niranjan Hasabnis, Rui Qiao and R. Sekar
**Evaluation**

**Verified against 15 Known GCC bugs**

```
sbbl %ebx, %ebx
(set (reg:SI ebx)(neg:SI
 (ltu:SI (reg:CC EFLAGS) (const_int 0)))))
```

**movsd bug**

```
movsd %xmm1, %xmm0
(set (reg:V2DF xmm0)
 (vec_merge:V2DF (reg:V2DF %xmm0) (reg:V2DF %xmm1) (const_int 1)))
```

**Results discussion**

- 7 of 39 semantic differences are soundness violations.
- Not all lead to crashes in GCC ⇒
  - Bugs guarded by assumptions: not a good practice
- We believe: improve compiler quality = make no assumptions

**Conclusion**

- **Novel approach** to test correctness of code generators
- **Easy integration** with compiler test suites
- Evaluation for GCC’s x86 code generator
  - Strict-equivalence checking: 39 violations
  - Loose-equivalence checking: 7 violations
  - At least 1 was confirmed a bug in latest GCC.
  - GCC accepted and fixed the bug.
Conclusion

Thank you. Question?

nhasabni@cs.stonybrook.edu
http://seclab.cs.stonybrook.edu

Implementation: GCC’s code generator for x86

1. **Rule extraction**: GCC plugin
2. **Start state generation**: 1K lines of C code + 500 lines of Prolog code as constraint solver
3. **RTL interpreter**: 3K lines of architecture-neutral C++ code, 50 lines of architecture-specific code
4. **Assembly execution system**: process monitoring framework
   - fork() + ptrace() + signal handling + segmentation
5. **Test execution and result comparison**
   - Executes multiple tests in parallel
   - 700 lines of C code

Evaluation: soundness violations

<table>
<thead>
<tr>
<th>D1: Imprecise modeling of EFLAGS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>shrdl example</td>
<td></td>
</tr>
<tr>
<td>sbbl %ebx, %ebx</td>
<td></td>
</tr>
<tr>
<td>{set (reg:SI ebx)(neg:SI ltu:SI (reg:CC EFLAGS) (const_int 0)))) (clobber EFLAGS)}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D2: Incorrect value in destination</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>movzwl B(%esp), %eax</td>
<td></td>
</tr>
<tr>
<td>{set (reg:HI ax) (mem:HI (plus:SI (reg:SI 7) (const_int 8)))}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D3: Incorrect operation in IR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>shrdl example</td>
<td></td>
</tr>
<tr>
<td>{set (reg:SI dx) truncate:SI lshiftrt:DI (mult:DI zero_extend:DI (reg:SI ax)) (zero_extend:DI (reg:SI bx)) (const_int 32 )))) (clobber (reg:SI ax)) (clobber (reg EFLAGS))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D4: Update to destination missing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mull %ebx</td>
<td></td>
</tr>
<tr>
<td>{set (reg:SI dx) truncate:SI lshiftrt:DI (mult:DI zero_extend:DI (reg:SI ax)) (zero_extend:DI (reg:SI bx)) (const_int 32 )))) (clobber (reg:SI ax)) (clobber (reg EFLAGS))</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation: soundness violation results

Total of 39 semantic differences found.

<table>
<thead>
<tr>
<th>Description</th>
<th>ArCheck</th>
<th>Random Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>D2</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>D3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>31</strong></td>
<td><strong>14</strong></td>
</tr>
<tr>
<td># of New Bugs Found</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># of Existing Bugs Found</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure: Evaluation of mapping rules obtained from GCCs x86 code generator
Backup slides

shrdl bug found in GCC

- Bug manifest for negative input values of eax