Test-case Reduction and Deduplication Almost for Free with Transformation Based Compiler Testing

Alastair F. Donaldson, Paul Thompson, Vasyl Teliman, Stefano Milizia, Andre Perez Maselco, Antoni Karpinski

Presentation by
Akshay Gopalakrishnan

McGill University

July 7, 2021

Introduction

- Compiler Testing.
- Transformation-based testing.
- Two problems: test-case duplication and reduction.
- Two birds with one stone - Transformation-based bug detection.
- Tool created- SpirvFuzz

What is Compiler testing?

- Testing for compilation bugs (including transformations phase).
- Naive way is to test with random programs keeping optimizations on/off depending on the requirement.
- Testing tools neccessitate the requirement that the semantics of the program does not exhibit Undefined Behavior.

Two main types exist (taken from paper)

- Multiple compiler testing - One program test on many compilers, then compare the end result of each.
- Single compiler testing - One program, apply semantics-preserving transformations randomly, check if the result of the final program is same.

This paper mainly focuses on Single compiler testing.

Two main issues in single compiler testing

Duplication of same bugs

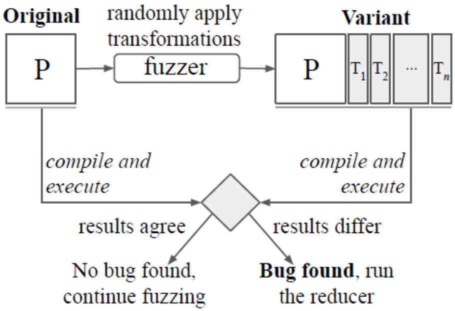
- Yes bug is found ! (Yay)
- But how to ensure that we do not generate the same bug in the next batch of testing?
- If we have a collection of potential bugs, how to ensure each bug is distinctive of one feature in the compiler?

Reduction strategy so that we can address the bugs sanely

- If the bug is detected in a transformed program of 10000 lines, it is not feasible to work resolving the bug in that big of a code.
- So we need to reduce to possibly find the local region of code that induces the bug.
- But manually doing this is impractical.
- How can we then automate this process?

Transformation-based Compiler Testing

- Pick a program.
- Subject it to a series of semantics preserving transformations.
- After each (or set) transformation(s), compile and ensure you get the same output as the original program.
- If not, a bug seems to exist (as transformations are all valid).



Example

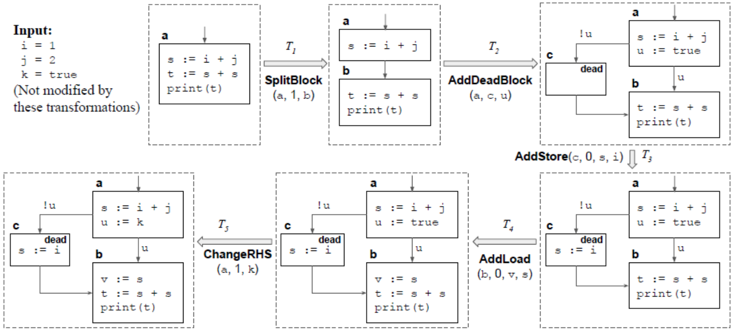


Figure 4. A series of transformations applied to a "basic blocks" program; **dead** denotes a "block is dead" fact

Example - cnt'd

Suppose bug is introduced by adding just a Dead Block and Obfuscating the fact that it is dead. Then the minimal bug (after reduction) would be as below:

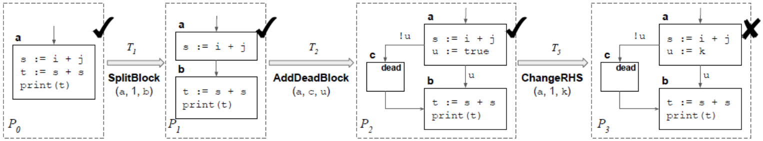
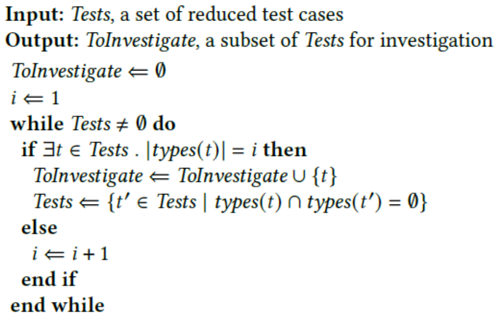


Figure 5. A minimized transformation sequence obtained by applying test-case reduction to the transformations of Figure 4

Deduplication process

Two test exhibiting bugs are the same bug if they are produced using the same set of program transformations (heuristic approach). The algorithm adopted is as follows:



Input: *Tests*, a set of reduced test cases
Output: *ToInvestigate*, a subset of *Tests* for investigation

```
$ToInvestigate \Leftarrow \emptyset$
$i \Leftarrow 1$
while $Tests \neq \emptyset$ do
  if $\exists t \in Tests \,.\, |types(t)| = i$ then
    $ToInvestigate \Leftarrow ToInvestigate \cup \{t\}$
    $Tests \Leftarrow \{t' \in Tests \mid types(t) \cap types(t') = \emptyset\}$
  else
    $i \Leftarrow i + 1$
  end if
end while
```

Figure 6. Deduplicating reduced test cases; $types(t)$ denotes the set of transformation types associated with test case t

Reduction process

1. Take the sequence of transformations used to induce bugs.
2. Test out every subsequence of the transformation.
3. Can leverage the extra information of facts (preconditions for each transformation).
4. Will get transformation dependance chains and our search space for reduction would reduce greatly.

SpirvFuzz

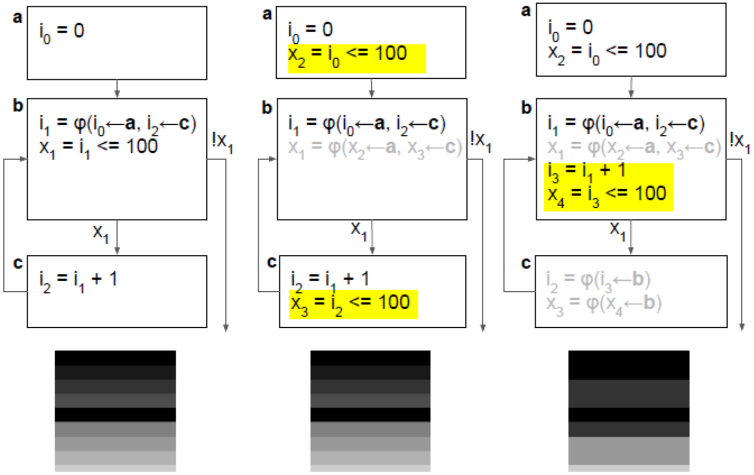
Takes three arguments -

1. SPIR-V module.
2. Input on which the module will execute.
3. Set of types of transformations.

Tool features upto 85 transformations. Some of them are:

1. *CompositeConstruct*
2. *ReplaceConstWithUniform*
3. *AddFunction*
4. *MoveBlockDown*

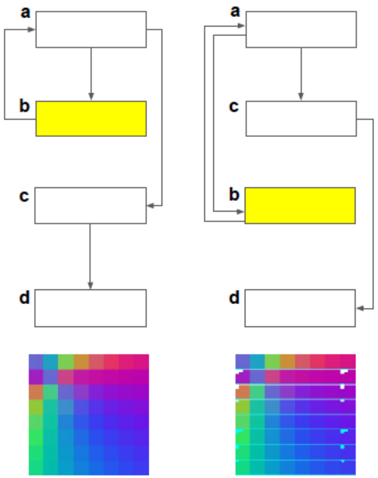
Miscompilation Bug Example 1



a
$i_0 = 0$
b
$i_1 = \varphi(i_0 \leftarrow a, i_2 \leftarrow c)$
$x_1 = i_1 <= 100$
$!x_1$
x_1
c
$i_2 = i_1 + 1$
a
$i_0 = 0$
$x_2 = i_0 <= 100$
b
$i_1 = \varphi(i_0 \leftarrow a, i_2 \leftarrow c)$
$x_1 = \varphi(x_2 \leftarrow a, x_3 \leftarrow c)$
$!x_1$
x_1
c
$i_2 = i_1 + 1$
$x_3 = i_2 <= 100$
a
$i_0 = 0$
$x_2 = i_0 <= 100$
b
$i_1 = \varphi(i_0 \leftarrow a, i_2 \leftarrow c)$
$x_1 = \varphi(x_2 \leftarrow a, x_3 \leftarrow c)$
$i_3 = i_1 + 1$
$x_4 = i_3 <= 100$
$!x_1$
x_1
c
$i_2 = \varphi(i_3 \leftarrow b)$
$x_3 = \varphi(x_4 \leftarrow b)$

(a) Mesa compiler bug

Miscompilation Bug Example 2



(b) Pixel 5 compiler bug

Comparison with glsl-fuzz

1. Bug Finding Ability.
2. Quality of Test-Case reduction.
3. Effectiveness of Test-Case deduplication.

Thank you

Questions?