Finding Missed Compiler Optimizations by Differential Testing

Gergö Barany
Inria Paris, France
gergo.barany@inria.fr

CC 2018
Main takeaways

Does your compiler always optimize well?

- compare compilers’ outputs to find missed optimizations
- automated toolchain finds minimal test cases
- issues found in GCC, Clang, CompCert:
  - peephole optimizations, dead stores, useless spills, missed instruction selection patterns, missed copy propagation, ...
Example: missing range analysis

Generated source code:

```c
int f(int p, int q) {
    return q + (p % 6) / 9;
}
```

(p % 6 ∈ [−5, 5],
division truncates to 0)

Clang:

```assembly
movw r2, #43691
movt r2, #10922
smmul r2, r0, r2
add r2, r2, r2, lsr #31
add r2, r2, r2, lsl #1
sub r0, r0, r2, lsl #1
movw r2, #36409
movt r2, #14563
smmul r0, r0, r2
asr r2, r0, #1
add r0, r2, r0, lsr #31
add r0, r0, r1
```

GCC:

```assembly
mov r0, r1
```

https://bugs.llvm.org/show_bug.cgi?id=34517 (fixed)
Example: redundant code

Source code:

```c
int fn3(
    double c,
    int *p, int *q)
{
    int i = (int)c;
    *p = i;
    *q = i;
    return i;
}
```

Clang:

```c
vcvt.s32.f64 s2, d0
vstr s2, [r0]
vcvt.s32.f64 s2, d0
vcvt.s32.f64 s0, d0
vmov r0, s0
vstr s2, [r1]
```

GCC:

```c
vcvt.s32.f64 s15, d0
vstr.32 s15, [r0]
vmov r0, s15
vstr.32 s15, [r1]
```

https://bugs.llvm.org/show_bug.cgi?id=33199 (fixed)
Randomized differential testing

- Random code generator
  - Source code
    - Compiler
      - Binary
        - Interesting difference?
          - No
            - Yes
              - Minimal?
                - No
                  - Yes
                    -
                - Yes
                  - Yes
                    -
Randomized differential testing for missed optimizations

**random code generator**

off-the-shelf tools: Csmith, ldrgen (or many others)

**test case reducer**

off-the-shelf tool: C-Reduce

**interesting difference?**

custom tool: optdiff

- binary analysis to find optimization differences
- assigns scores to binaries, compares
**optdiff**

- based on angr binary analysis framework
  - multi-platform (x86, x86-64, ARM, PowerPC, ...)
  - Python API
- load binary, compute CFG, estimate basic block frequencies $w_b$

**Checkers:** local scoring functions $c : instruction \rightarrow \mathbb{N}$

**Total score:**

$$s = \sum_{b \in f} w_b \cdot \sum_{i \in b} c(i)$$

**Examples:** number of instructions, general memory loads/stores, stack loads/stores, function calls, floating-point arithmetic instructions, vector instructions, ...
Checker implementation: instructions

Checkers

- Python functions with `@checker` decorator
- inspect one instruction at a time

```python
@checker
def instructions(arch, instr):
    """Number of instructions."""
    return 1
```
def loads(arch, instr):
    """Number of memory loads."""
    op = instrinsn.mnemonic
    if is_arm(arch):
        if op == 'ldrd':
            return 2  # load doubleword
        elif re.match('ldm.*', op):  # load multiple
            return len(instrinsn.operands) - 1
        return bool(re.match('v?ldr.*', op))  # load one word
    ...  # other architectures
Example: useless spill

<table>
<thead>
<tr>
<th>Source code:</th>
<th>Clang:</th>
<th>GCC:</th>
</tr>
</thead>
</table>
| char fn2(  
  float p)  
{  
  char c=(char)p;  
  return c;  
} | vcvt.u32.f32 s0,s0  
   vmov r0,s0 | vcvt.u32.f32 s15,s0  
   sub sp,sp,#8  
   vstr.32 s15,[sp,#4]  
   ldrb r0,[sp,#4]  
   add sp,sp,#8 |

instruction score: 2  
stack load score: 0

instruction score: 5  
stack load score: 1

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80861 (confirmed, diagnosed)
CompCert: an example

Source code:

```c
int fn10(int p1) {
    int a, b, c, d, e, v, f;
    a = 0;
    b = c = 0;
    d = e = p1;
    v = 4;
    f = e * d | a * p1 + b;
    return f;
}
```

CompCert:

```assembly
str r4, [sp, #8]
mov r4, #0
mov r12, #0
mov r1, r0
mov r2, r1
mul r3, r2, r1
mla r2, r4, r0, r12
orr r0, r3, r2
ldr r4, [sp, #8]
```

- dead code \( v = 4 \); causes spilling
- missed copy propagation of \( d = e = p1 \);
- missed constant propagation and folding: \( a * p1 + b = 0 \)
Undefined behavior: the good

Undefined behavior may be compiled arbitrarily
— do we have to be careful?

Unproblematic cases:

- Clang and GCC treat many cases identically, comparisons OK
- `char f(float p) { return (char) p; }`
  (assume never called with bad values of `p`)
- `x < x + 1 → true`
  (undefined for `x` signed integer)
Undefined behavior: the bad

Problematic cases:

- **unconditional undefined behavior**, e.g.,
  ```c
  int fn(int a) { int x = 0; return a / x; }
  ```
- **infinite loops**:
  ```c
  while (x) { y = ...; }
  ```
- C-Reduce likes to produce such cases
- no compiler warnings but different ‘optimized’ code

Workarounds

- static analysis to find UB/nontermination? **ineffective 😞**
- accept some cases
- don’t let random generators produce loops/problematic constructs
Why *randomized* differential testing?

Arguments against random input programs

- examples look artificial
- may not correspond to real-world performance problems

Advantages of random input programs

+ unlimited amount of code available
+ controlled sublanguage (loop-free, only types/constructs of interest)

Also:

$\sim$ reducer output looks artificial even for real-world input
Results

https://github.com/gergo-/missed-optimizations

Some missed optimizations found

<table>
<thead>
<tr>
<th>Compiler</th>
<th>total</th>
<th>reported</th>
<th>fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>13</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Clang</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CompCert</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- generally treated as low priority by developers
- many duplicates

Causes: missing/wrong rules or costs; phase ordering; weak heuristics; ?
Summary

- compare compilers’ outputs to find missed optimizations
- automated toolchain finds minimal test cases
- issues found in GCC, Clang, CompCert

https://github.com/gergo-/missed-optimizations

Thank you for your attention

This research was partially supported by ITEA 3 project no. 14014, ASSUME.