Finding Missed Optimizations in LLVM (and other compilers)

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

EuroLLVM, Bristol, April 16, 2018

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

Main takeaways

Does your compiler *always* optimize well?

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:

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

Clang:

GCC:

Example: missing range analysis

Generated source code:

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

Clang:

```
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:

Example: missing range analysis

Generated source code:

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

Clang:

```
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:

```
mov r0, r1
```

Example: missing range analysis

Generated source code:

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

($p \% 6 \in [-5, 5]$,
division truncates to 0)

Clang:

```
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:

```
mov r0, r1
```

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

Example: redundant code

Source code:

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

Clang:

GCC:

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

Example: redundant code

Source code:

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

Clang:

```
vcvt.s32.f64 s2,d0
vstr s2,[r0]
```

GCC:

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

Example: redundant code

Source code:

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

Clang:

```
vcvt.s32.f64 s2,d0
vstr s2,[r0]
vcvt.s32.f64 s2,d0
```

GCC:

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

Example: redundant code

Source code:

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

Clang:

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

GCC:

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

Example: redundant code

Source code:

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

Clang:

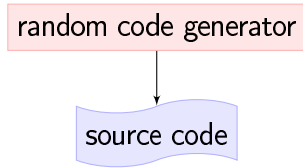
```
vcvt.s32.f64 s2,d0
vstr s2,[r0]
vcvt.s32.f64 s2,d0
vcvt.s32.f64 s0,d0
vmov r0,s0
vstr s2,[r1]
```

GCC:

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

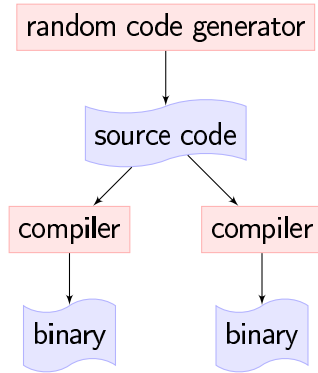
https://bugs.llvm.org/show_bug.cgi?id=33199 (fixed, then reverted)

Randomized differential testing



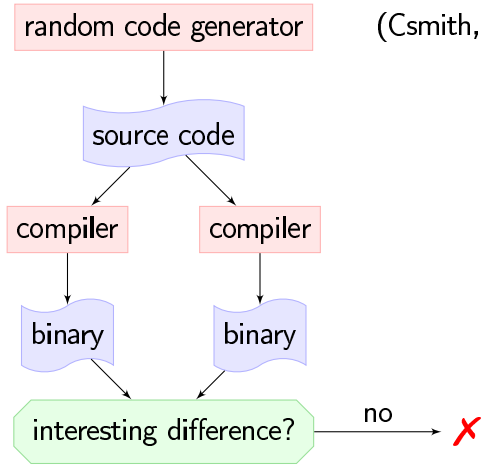
(Csmith, Idrgen, . . .)

Randomized differential testing

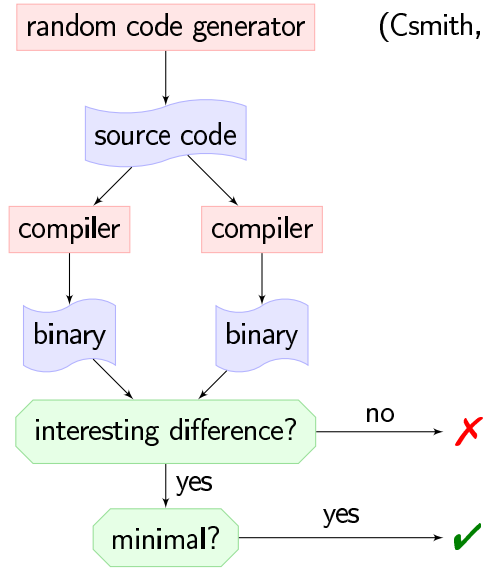


(Csmith, Idrgen, . . .)

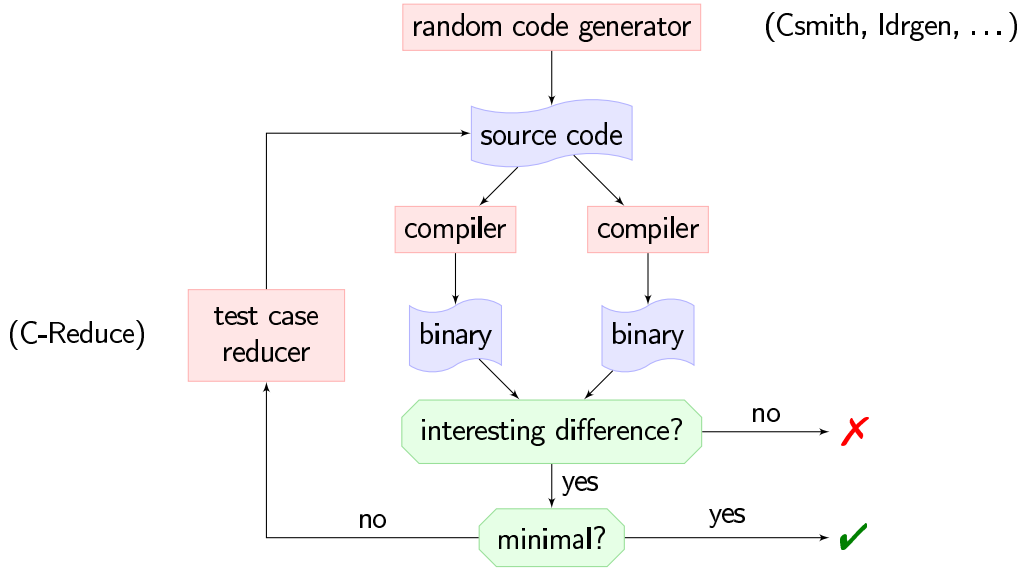
Randomized differential testing



Randomized differential testing



Randomized differential testing



Randomized differential testing for missed optimizations

interesting difference?

custom tool: `optdiff`

- binary analysis to find optimization differences
- assigns scores to binaries, compares
- based on `angr` binary analysis framework
  - multi-platform (x86, x86-64, ARM, PowerPC, ...)
  - Python API

optdiff

Preliminaries: load binary, compute CFG, estimate basic block frequencies w_b

Checkers: local scoring functions $c\colon \mathit{instruction} \to \mathbb{N}$

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

Examples checkers: 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

```
@checker
def instructions(arch, instr):
    """Number of instructions."""
    return 1
```

Checker implementation: loads

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

Example: instruction selection

Source code:

```
double fn1(double p1, double p2,
    double p3)
{
  double a = -p1 - p2 * p3;
  return a;
}
```

Clang:

GCC:

```
vnmla.f64 d0, d1, d2
```

Example: instruction selection

Source code:

```
double fn1(double p1, double p2,
    double p3)
{
  double a = -p1 - p2 * p3;
  return a;
}
```

Clang:

```
vneg.f64 d0, d0
vmls.f64 d0, d1, d2
```

GCC:

```
vnmla.f64 d0, d1, d2
```

Clang selected `vnmla` for `-(a * b) - c` but not for `-c - (a * b)`

`https://reviews.llvm.org/D35911` (merged)

Example: register allocation

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, v, e;
  a = p4 - 5 + 9 * p1 * 7;
  c = 2 - p6 + 1;
  b = 3 * p6 * 4 - (p2 + 10) * a;
  d = (8 + p3) * p5 * 5 * p7 - p4;
  v = p1 - p8 - 2 -
    (p8 + p2 - b) * ((p3 + 6) * c);
  e = v * p5 + d;
  return e;
}
```

Example: register allocation

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, v, e;
  a = p4 - 5 + 9 * p1 * 7;
  c = 2 - p6 + 1;
  b = 3 * p6 * 4 - (p2 + 10) * a;
  d = (8 + p3) * p5 * 5 * p7 - p4;
  v = p1 - p8 - 2 -
    (p8 + p2 - b) * ((p3 + 6) * c);
  e = v * p5 + d;
  return e;
}
```

```
...
vmov.f64 d13, #4.000000e+00
vmov.f64 d14, #1.000000e+01
vmul.f64 d10, d10, d13
vadd.f64 d11, d1, d14
vmov.f64 d12, #2.000000e+00
vmov.f64 d15, #8.000000e+00
vsub.f64 d5, d12, d5
vadd.f64 d12, d2, d15
vmls.f64 d10, d11, d9
vstr d6, [sp]
vmov.f64 d6, #6.000000e+00
...
```

Example: register allocation

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, v, e;
  a = p4 - 5 + 9 * p1 * 7;
  c = 2 - p6 + 1;
  b = 3 * p6 * 4 - (p2 + 10) * a;
  d = (8 + p3) * p5 * 5 * p7 - p4;
  v = p1 - p8 - 2 -
    (p8 + p2 - b) * ((p3 + 6) * c);
  e = v * p5 + d;
  return e;
}
```

```
...
vmov.f64 d13, #4.000000e+00
vmov.f64 d14, #1.000000e+01
vmul.f64 d10, d10, d13
vadd.f64 d11, d1, d14
vmov.f64 d12, #2.000000e+00
vmov.f64 d15, #8.000000e+00
vsub.f64 d5, d12, d5
vadd.f64 d12, d2, d15
vmls.f64 d10, d11, d9
vstr d6, [sp]      ⟵ spill d6
vmov.f64 d6, #6.000000e+00
...
```

Example: register allocation

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, v, e;
  a = p4 - 5 + 9 * p1 * 7;
  c = 2 - p6 + 1;
  b = 3 * p6 * 4 - (p2 + 10) * a;
  d = (8 + p3) * p5 * 5 * p7 - p4;
  v = p1 - p8 - 2 -
    (p8 + p2 - b) * ((p3 + 6) * c);
  e = v * p5 + d;
  return e;
}
```

```
...
vmov.f64 d13, #4.000000e+00
vmov.f64 d14, #1.000000e+01
vmul.f64 d10, d10, d13
vadd.f64 d11, d1, d14
vmov.f64 d12, #2.000000e+00
vmov.f64 d15, #8.000000e+00
vsub.f64 d5, d12, d5
vadd.f64 d12, d2, d15
vmls.f64 d10, d11, d9
vstr d6, [sp]      ⟵ spill d6
vmov.f64 d6, #6.000000e+00
...
```

d13

d14

d15

spill of register d6 instead of using a free register (d13, d14, d15)

Example: register allocation

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, v, e;
  a = p4 - 5 + 9 * p1 * 7;
  c = 2 - p6 + 1;
  b = 3 * p6 * 4 - (p2 + 10) * a;
  d = (8 + p3) * p5 * 5 * p7 - p4;
  v = p1 - p8 - 2 -
    (p8 + p2 - b) * ((p3 + 6) * c);
  e = v * p5 + d;
  return e;
}
```

```
...
vmov.f64 d13, #4.000000e+00
vmov.f64 d14, #1.000000e+01
vmul.f64 d10, d10, d13
vadd.f64 d11, d1, d14
vmov.f64 d12, #2.000000e+00
vmov.f64 d15, #8.000000e+00
vsub.f64 d5, d12, d5
vadd.f64 d12, d2, d15
vmls.f64 d10, d11, d9
vstr d6, [sp]
vmov.f64 d6, #6.000000e+00
...
```

d13 d14 d15

spill d6

spill of register d6 instead of using a free register (d13, d14, d15)

https://bugs.llvm.org/show_bug.cgi?id=37073 (reported, caused by scheduling)

Example: excessive copying

Clang: GCC:

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, e, f;
  a = d = b = p1 + p3;
  e = p3 + p2 * 8.7 +
    ((p4 - p4) * (a * p7) - 1.7);
  c = b - d - (p2 - 7.4) * 8.1;
  f = e - p2 * p3 -
    d * (4.6 + c) * p4 +
    p5 * p6 - p7 - p8 + p1;
  return f;
}
```

Example: excessive copying

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, e, f;
  a = d = b = p1 + p3;
  e = p3 + p2 * 8.7 +
    ((p4 - p4) * (a * p7) - 1.7);
  c = b - d - (p2 - 7.4) * 8.1;
  f = e - p2 * p3 -
    d * (4.6 + c) * p4 +
    p5 * p6 - p7 - p8 + p1;
  return f;
}
```

Clang:

GCC:

```
vadd.f64 d8, d0, d2
vmul.f64 d9, d8, d6
vsub.f64 d10, d3, d3
...
```

Example: excessive copying

```
double fn1(double p1, double p2,
    double p3, double p4, double p5,
    double p6, double p7, double p8)
{
  double a, b, c, d, e, f;
  a = d = b = p1 + p3;
  e = p3 + p2 * 8.7 +
    ((p4 - p4) * (a * p7) - 1.7);
  c = b - d - (p2 - 7.4) * 8.1;
  f = e - p2 * p3 -
    d * (4.6 + c) * p4 +
    p5 * p6 - p7 - p8 + p1;
  return f;
}
```

caused by scheduling?

Clang:

```
vadd.f64 d8, d0, d2
vmov.f64 d10, d7
vmov.f64 d7, d5
vmov.f64 d5, d4
vmov.f64 d4, d3
vmul.f64 d11, d8, d6
vsub.f64 d12, d4, d4
...
```

5 copies in 30 instrs

GCC:

```
vadd.f64 d8, d0, d2
vmul.f64 d9, d8, d6
vsub.f64 d10, d3, d3
...
```

1 copy in 26 instrs

A GCC example

```
int N;
long fn1(void) {
  short i;
  long a;
  i = a = 0;
  while (i < N)
    a -= i++;
  return a;
}
```

GCC:

Loop vectorized...

A GCC example

```
int N;
long fn1(void) {
  short i;
  long a;
  i = a = 0;
  while (i < N)
    a -= i++;
  return a;
}
```

GCC:

Loop vectorized...

late Feb 2018 no longer vectorized, 1.8× slowdown

A GCC example

```
int N;
long fn1(void) {
  short i;
  long a;
  i = a = 0;
  while (i < N)
    a -= i++;
  return a;
}
```

GCC:

Loop vectorized...

late Feb 2018 no longer vectorized, 1.8 × slowdown

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84986 (fixed)

A CompCert example

```
int f(int a) {
  if (a + 42)
    ;
  else
    ;
  return 1;
}
```

Clang:

```
mov r0, #1
```

A CompCert example

```
int f(int a) {
  if (a + 42)
    ;
  else
    ;
  return 1;
}
```

Clang:

```
mov r0, #1
```

CompCert:

```
add r0, r0, #42
mov r0, #1
```

(fixed)

Results

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

Some missed optimizations found

	total	reported	fixed
GCC	18	7	2
Clang	9	4	2
CompCert	9	3	3

- generally treated as low priority by developers
- many duplicates
- main limitation: loops (C-Reduce generates infinite loops)

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`

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.