Floating Point Exceptions and "Fast Math" optimizations

Christopher Stadler

December 20, 2019

Barr et al. aimed to discover inputs that raise floating point exceptions in a given program [1]. As an extension of this work we aim to discover inputs that lead to different exception-raising behavior between two versions of a given program — one optimized with "fast math".

Our approach has two steps:

1. Solve: find candidate inputs using an SMT solver (Z3).
2. Search: Test inputs in the neighborhood of each candidate against $\mathbb{P}$ and $\mathbb{P}'$ until one produces different exception traces.

Our first step is conceptually similar to that of Barr et al., differing only in implementation choices. Our primary contribution is to test the inputs generated by the first step on both $\mathbb{P}$ and $\mathbb{P}'$, comparing their exception-raising behavior.

Our implementation takes a single C source code file, compiles it with and without "fast math" optimizations, and runs both steps.

1 Find candidate inputs

To find an input that raises an exception at an operation in a program we should be able to construct a formula representing the conditions under which an exception would occur. For example, given an operation `a + b` we could construct the formula $|a+b| = \infty$. Solutions to this formula would be guaranteed to raise an overflow exception if given as inputs to the operation. And if the formula is unsatisfiable this would prove that the operation could not raise the exception.

However, these conclusions are only sound if our formula uses a theory of floating point numbers which corresponds to the execution environment. SMT solvers such as Z3 do include floating point theories, but solving these formulas is not generally practical. Adopting the approach of Barr et al. we instead construct formulae over the real numbers. To find overflow-raising inputs for the above example we construct the formula $|a+b| > \Omega$, where Ω is the greatest representable floating point value (e.g. `DBL_MAX` if a and b are double precision

values). Since floating point arithmetic is an approximation of real arithmetic we expect that solutions to this formula are likely to either raise an overflow or be near values which will raise an overflow.

Expression	Overflow	Underflow	Invalid	Divide by Zero
$x \odot y$	$\lvert x \odot y \rvert > \Omega$	$0 < \lvert x \odot y \rvert < \omega$	N/A	N/A
x/y	$\lvert x \rvert > \lvert y \rvert \Omega$	$0 < \lvert x \rvert < \lvert y \rvert \omega$	$x = 0 \wedge y = 0$	$x \neq 0 \wedge y = 0$

Table 1: Formulae for each exception type. Where $\odot \in \{+, -, *\}$, Ω is the largest representable value, and ω is the smallest positive "normal" value (e.g. `DBL_MIN`)

For each instruction in the given program we construct such a formula for each type of exception that could be raised (See 1). We do not make formulae for inexact exceptions as these are too common to be notable.

1.1 Implementation

Ariadne instruments the program source code with a conditional for each exception type and then uses KLEE to find inputs that cause the body of the conditional to be reached [1, pp. 3–4]. However, KLEE does not support floating point values so this approach required modifying KLEE to interpret floating point values as reals [1, p. 2].

We instead have implemented our own technique for building these formulae that does not use KLEE. Our Python program takes two files as inputs: LLVM IR for $\mathbb{P}$ and $\mathbb{P}'$. Each of these is parsed into an AST using *llvmlite*, which provides python bindings for LLVM. For each instruction in each program we build the formulae given by 1 using Z3.

When an operand of an instruction is an identifier we substitute the corresponding expression into the formula. For example, to find an input that causes an overflow to be raised in the second addition below we would solve the formula $|(a + b) + a| > \Omega$, since $x = a + b$.

```
double add2(double a, double b) {
  double x = a + b;
  double y = x + a;
  return y;
}
```

This results in formulae containing only constant terms and the program parameters, which are left free. A satisfying assignment for such a formula therefore can be interpreted as inputs to the program.

The union of the sets of formulae from the two programs is then taken. Since $\mathbb{P}$ and $\mathbb{P}'$ are likely to share many instructions some of the same formulae are generated from both programs. Finally we solve each formula (using Z3) and return the satisfying assignments for each satisfiable formula.

2 Search for exception raising inputs

The output of the previous step is a set of inputs, each of which we *expect* to cause an exception in either $\mathbb{P}$ or $\mathbb{P}'$. The goal now is to find inputs that (1) do cause an exception in $\mathbb{P}$ or $\mathbb{P}'$ and (2) have different exception-raising behavior in the two programs.

Because our formulae are formulated over the reals but the concrete execution is done in floating point the candidate inputs may not cause the expected exception to be raised. For example, suppose for some program that any input x greater than or equal to a constant α will raise an exception. The solver may return the assignment $x = \alpha$, but this real value for x may not be exactly representable in floating point. When given as input to the concrete program x will therefore be rounded to some x'. If rounded down then $x' < \alpha$, and an exception will not be raised.

An insight of Barr et al. was that even if the satisfying assignment does not raise an exception there is likely to be a floating point number close to it that does [1, p. 2]. We therefore search the neighborhood of the satisfying assignment for an exception-raising input. Furthermore, we search for an input that has different exception-raising behavior in $\mathbb{P}$ and $\mathbb{P}'$. If the optimizations of $\mathbb{P}'$ have eliminated the possibility of a certain exception then we just need to find an input which raises it in $\mathbb{P}$.

Defining "different exception-raising behavior" is difficult because $\mathbb{P}$ and $\mathbb{P}'$ represent different computations, even if they are intended to be approximately the same. For example, suppose for some input both programs raise an overflow exception in the LLVM instruction `%2 = fadd double %0, %1`. Despite having the same representation these instructions could represent different computations if optimizations have changed the expressions assigned to the operands. I.e. it is not possible to define a correspondence between instructions in the two programs.

We have chosen to define "different exception-raising behavior" in terms of the sequence of exception types produced by each program. If these sequences are different then we say the programs have different behavior. If they are the same, regardless of where these exceptions occur in the program, we say they are not different. This definition is quite strict in what it considers "different". For example, if both programs produce a single overflow but at very different points in the program this is not considered a difference. The advantage of this definition is that there are no "false positives".

2.1 Implementation

Since the above definition is in terms of sequences of exceptions we need to collect these sequences for $\mathbb{P}$ and $\mathbb{P}'$. To accomplish this we have implemented an LLVM pass which adds a call to a `check_for_exception` function after every floating point instruction. `check_for_exception` queries the floating point status word [3] to determine if an exception has occurred, and it's type. If one has occurred it records this in a global variable. After the execution of the program

this variable therefore stores the sequence of exceptions that occurred.

The given program is compiled into $\mathbb{P}$ and $\mathbb{P}'$, each of these is instrumented using the pass described above, and these are both linked to a driver program (written in C++). Candidate inputs are read from a file (produced by the first step). For each of these inputs we first test it on $\mathbb{P}$ and $\mathbb{P}'$, and if a difference is not found we search in the neighborhood of the input. This search is done using the `nextafter` function [2] to find the next higher and lower representable floating point values. This search is done until a difference is found or a constant bound is reached. This results in an exhaustive search centered around the given input.

Because an input to the program may consist of multiple discrete arguments this search is done for each argument, so that all combinations are tested.

3 Results

We have tested our implementation on several benchmark programs (See 3). These are all straight line programs, except for `odometer` which has a loop. We eliminated this by fully unrolling it at the LLVM level. The `identity` program (shown below) was crafted so that all operations are eliminated by the "fast math" optimizations.

```
double identity(double x) {
  double a = 2 * x;
  double b = a * 0.5;
  return b;
}
```

Program	Source lines	Arguments	LLVM instructions ($\mathbb{P}$, $\mathbb{P}'$)
identity	2	1	2, 0
turbine1	1	3	14, 11
turbine3	1	3	14, 11
jetengine	6	2	27, 27
carbongas	7	2	7, 7
odometer	17	1	55, 39 (unrolled)

The results are shown in 3. The search range was set to 3 (in each direction, resulting in a total range of 7) because larger ranges did not yield different results.

For `identity`, `turbine1`, `turbine3`, and `odometer` we were able to find inputs for which $\mathbb{P}$ and $\mathbb{P}'$ produced different sequences of exceptions. From 3 we can see that `jetengine` and `carbongas` both had the same number of instructions in $\mathbb{P}$ and $\mathbb{P}'$, suggesting that they were not significantly changed by the optimizations.

Program	Formulae ($\mathbb{P}$, $\mathbb{P}'$)	Satisfiable	Unique inputs	Diff producing
identity	4 (4, 0)	3	3	1
turbine1	48 (32, 26)	44	29	14
turbine3	48 (32, 26)	44	30	11
jetengine	106 (58, 58)	96	41	0
carbongas	26 (16, 16)	23	4	0
odometer (2 iterations)	190 (122, 76)	167	70	15

4 Limitations

The primary theoretical limitation of our approach is that our first step — finding candidate inputs — takes into account only one program ($\mathbb{P}$ or $\mathbb{P}'$) at a time. We thus have no guarantee that a candidate input will produce a difference between $\mathbb{P}$ and $\mathbb{P}'$. In practice we do find such differences, but only for some inputs.

Our implementation is limited in that it can currently only be applied to straight line programs consisting purely of floating point values and operations. Programs with loops can be analysed only if the loop can be fully unrolled.

5 Algorithm

This gives a pseudo-code overview of our implementation.

```
fun main(source):
  P = compile source to LLVM without optimizations
  P' = compile source to LLVM with optimizations

  inputs = find_inputs(P, P')
  search(inputs, P, P')

fun find_inputs(P, P')
  p_formulae = make_formulae(P)
  p'_formulae = make_formulae(P')

  formulae = p_formulae ∪ p'_formulae

  inputs = collect_inputs(formulae)
  return map ()

fun collect_inputs(formulae):
  solutions = []

  for formula in formulae:
    solution = solve(formula)
```

```
    if solution is sat:
      solutions << solution.inputs

  return solutions

fun make_formulae(llvm):
  env = {} # Map of identifiers to symbolic values

  # Require that each input is in the representable range.
  inputs_constraint = True
  for f in llvm.formals:
    env[f] = f
    inputs_constraint = (inputs_constraint and (abs(f) < DBL_MAX))

  formulae = []

  for inst in llvm.instructions:
    result = symbolically_execute(inst, env)
    env[inst.destination] = result

    if inst.op == fdiv:
      # Invalid
      formulae << (inst.numerator == 0 and inst.denominator == 0)
      # DivByZero
      formulae << (inst.numerator != 0 and inst.denominator == 0)
      # Overflow
      formulae << (abs(inst.numerator) > (abs(inst.denominator * DBL_MAX)))
    else:
      # Overflow
      formulae << (abs(result) > DBL_MAX)
      # Underflow
      formulae << (abs(result) > 0 and abs(result) < DBL_MIN)

  formulae = map (f => f and inputs_constraint) formulae
  return formulae

fun search(inputs, P, P'):
  P = instrument P to check for FP exceptions
  P' = instrument P' to check for FP exceptions

  for input in inputs:
    for input in nearby_fp_numbers(input):
      p_trace = exec(P, input)
      p'_trace = exec(P', input)

      if p_trace != p'_trace:
```

```
report_diff(input, p_trace, p'_trace)
```

References

[1] Earl T Barr et al. "Automatic Detection of Floating-Point Exceptions". In: *POPL* (2013), p. 12.

[2] *nextafter(3) - Linux manual page*. URL: http://man7.org/linux/man-pages/man3/nextafter.3.html (visited on 12/19/2019).

[3] *Status bit operations (The GNU C Library)*. URL: https://www.gnu.org/software/libc/manual/html_node/Status-bit-operations.html (visited on 12/19/2019).