Instability of Floating-Point Exceptions Against Code Optimizations (Draft of May 5, 2020)

The Impact of Code Optimizations on Floating-Point Exceptions (Draft of May 5, 2020)

Christopher Stadler
Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115

Thomas Wahl
Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115

Yijia Gu
Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115

***Abstract*—Floating-point arithmetic, an approximation of real-valued arithmetic employed on many general-purpose computing platforms, is known to be fragile: apparently harmless expression rewriting often causes the value of the expression to change, on some inputs, even if the rewriting is equivalent under real-algebra laws. This results in the loss of I/O-equivalence of compiler optimizations, such as those permitted by the fast-math flag family included in many modern compilers.**

In this paper we investigate the impact of such optimizations on floating-point *exceptions*. Exceptions are raised, among others, when floating-point calculations over- or underflow, or when a number is divided by zero. In many applications, exceptions indicate an unexpected event in the calculation that, if allowed to continue unchallenged, will likely have an undesired impact on the computation. Both testing and more formal techniques have been developed to flag the possibility of numeric code to raise floating-point exceptions to programmers and end-users. The effectiveness of such techniques is, however, greatly affected when optimizations cause enough changes to the code that new exceptions arise, or existing ones disappear.

We present an approach, and a tool called FOE, that first identifies inputs to a program that are likely to trigger exceptions of interest. This step is based on earlier work on the detection of floating-point exceptions that did not, however, offer an implementation. Our tool then uses a neighborhood search that attempts to find inputs where the exception behavior of the program and a fast-math-optimized version diverge. We demonstrate in this study that the phenomenon of floating-point exceptions impacted by code optimizations is very real, and ... (summarize insights). Ways to stabilize exceptions against optimizations are discussed.

I. Introduction

The goal of code optimizations performed by compilers is to improve the "efficiency" of program execution, which can mean reduced runtime or memory consumption, code size, or other resource requirements. In most software applications, the expectation is that this does not alter the functional semantics of a program: for any given input, the optimized program should return the same output as the original source code. This ensures contextual equivalence of the two code versions.

This expectation is unrealistic, however, for floating-point programs, where compile-time arithmetic on constants, or expression rewriting into seemingly equivalent forms impacts the I/O-semantics. Consider the expression [1]

```
3.0 + 2.0 / (r * r) - 0.125 * (3.0 - 2.0 * v) * w * w
    * r * r / (1.0 - v) - 4.5                                  (1)
```

source of `turbine`? I want to suggest that the example is not concocted which tempts the compiler to distribute the multiplication `0.125 *` over the difference expression (shown in red), folding constants along the way. Such real-algebra simplifications are not, however, value-preserving in floating-point, which is well-known among experts. Optimization switches that turn on aggressive expression rewriting, such fast-math and its equivalents on `gcc`, LLVM, and Microsoft Visual `C++`, assume that the programmer is aware of this *unsafe* optimization behavior, and of the consequences for the program.

A more discrete change in the behavior of a floating-point program caused by code optimizations is the introduction or removal of *exceptions*. These are raised in response to certain numeric events, such as an overflow or a division by zero. Exceptions almost always indicate an unwanted situation in the calculation that will likely have an undesired impact on the computation, unless it is specifically handled by the code, or the computation is aborted.

Consider now a source-code analysis technique, or a test suite run on plainly compiled code, that claims the code to be exception-free over some input range, eliminating the need for installing proper handlers. With the *implicit* expectation of I/O equivalence in mind, if the deployed code is aggressively optimized, we may be surprised to find that a live code run produces unusual computational results that are ultimately attributable to uncaught exceptions. Such analyses are almost inevitably very expensive.

The goal of the approach proposed in this paper is discover inputs in a (loop-free) program $\mathbb{P}$ such that its exception behavior deviates from that of program $\mathbb{P}'$ obtained by subjecting $\mathbb{P}$

to aggressive optimizations triggered by the fast-math flag. We say the exception behaviors of $\mathbb{P}$ and $\mathbb{P}'$ deviate if, for some input, they produce different traces of exceptions, for instance the empty vs. a non-empty trace. As an example, letting $\mathbb{P}$ be a program that implements expression (1), the following input triggers no exceptions whatsoever when given to $\mathbb{P}$, while its optimized version $\mathbb{P}'$ overflows on this input:

```
|       v:   -2.50000000000000000000e-01
I:      w:   +1.34078079299425970996e+154
        r:   -1.25000000000000000000e-01
```

Program $\mathbb{P}$ returns a finite (if large-magnitude) value and therefore passes (on $\mathcal{I}$) the assume/guarantee-style formula, "non-special in $\Rightarrow$ non-special out", while $\mathbb{P}'$ fails it, returning $-\infty$.

Our approach to finding such discrepancies consists of two steps. First, we build on an earlier idea by Barr et al. to automatically detect floating-point exceptions in programs [2], in order to determine inputs that trigger them. The earlier work did not come with a tool, so we implemented a similar technique for our own purpose. Our technique must operate at the level of the intermediate representation of a compiler, such as LLVM IR, since it is this level at which code optimizations are applied. This allows us to compute input candidates for raising exceptions from both $\mathbb{P}$ and $\mathbb{P}'$, rendering our approach "symmetric".

Our second step is to supply each candidate input $\mathcal{I}$ to both $\mathbb{P}$ and $\mathbb{P}'$, and to observe the triggered exception traces. This step serves two purposes: (i) to confirm that $\mathcal{I}$, or an input in a small neighborhood of $\mathcal{I}$, does indeed cause exceptions to be triggered in the program from which $\mathcal{I}$ was obtained in the first step (see Section III for details), and (ii) to detect any differences in the entire trace of exceptions between $\mathbb{P}$ and $\mathbb{P}'$.

We demonstrate in this (preliminary) study that the destabilizing effect (in terms of exception behavior) code optimizations have on floating-point code is real. (summarize insights)

II. Background and Problem Definition

Floating-point arithmetic is an approximation of real arithmetic widely available in computers today, and loosely regulated in the IEEE 754 floating-point standard [3]. A floating-point number consists of a sign bit, a mantissa, and an exponent, for which a certain number of bits are reserved, depending on a particular floating-point format; examples in this paper use the `double` format. To make numbers fit into the format restrictions, floating-point arithmetic heavily employs *rounding* of computational results.

Whenever a floating-point operation causes a deviation from a real-arithmetic result, the IEEE Standard requires an *exception* to be raised by the implementation, so that programmers can catch these likely disruptive outcomes. Such deviations include: (i) the mathematical result does not fit into the current representation and must be rounded ("inexact"); (ii) overflows and underflows; (iii) a division by zero; and (iv) an "invalid" result, such as an attempt to compute $0 \cdot \infty$, $\infty - \infty$, or $0/0$. Exceptions of type (i) are very common, since many operations require rounding. Optimizations typically involve compile-time constant folding and therefore almost always change the number of roundings applied, and hence the occurrence of inexact exceptions. We therefore exclude them in this study.

We say an overflowing calculation *raises* an exception, while an input leading to this calculation *triggers* an exception.

Problem definition. Given program $\mathbb{P}$ and an input $\mathcal{I}$, the triggered *exception trace* is the trace of exception types observed when running $\mathbb{P}$ on $\mathcal{I}$. (Note that, for the purposes of this preliminary study, $\mathbb{P}$ is loop-free, so any trace is finite.) The exception behaviors of (any two) programs $\mathbb{P}$ and $\mathbb{P}'$ *deviate* if there exists an input $\mathcal{I}$ such that the exception traces of $\mathbb{P}$ and $\mathbb{P}'$ triggered by $\mathcal{I}$ differ. Two traces are *equal* if they are equal as tuples, i.e. of the same length and point-wise equal. The goal of this paper is to propose, and present a preliminary evaluation of, a technique to determine whether the exception behaviors of a program $\mathbb{P}$ and a fast-math-optimized version $\mathbb{P}'$ deviate.

III. Finding Exception-Raising Input Candidates

Given program $\mathbb{P}$ and an optimized version $\mathbb{P}'$, our overall approach is to proceed in two steps:

1) **(this section)** Find candidates for inputs that cause an exception to be triggered in *either* $\mathbb{P}$ or $\mathbb{P}'$;
2) **(Section IV)** Test the candidate inputs, and inputs in a neighborhood of them, as to whether they witness deviating exception behaviors of $\mathbb{P}$ and $\mathbb{P}'$.

To find an input that triggers an exception at some operation in a program we essentially follow the approach proposed by Barr et al. [2]. Namely, we symbolically execute the *loop-free* program from the inputs along any path to any floating-point operation that has the potential to raise an exception. Then we conjoin the path formula with the conditions that will trigger the exception, e.g. a dividend equal to zero, or a result that exceeds the value of `DBL_MAX` (defined in `<float.h>`), the maximum representable finite floating-point number in the double format. Table I shows most subformulas we have used to encode the exceptions of the various types. "invalid" means more than just 0/0: it also means $\infty - \infty$, etc. What kinds of invalid exceptions did you look for?

The next sub-step is to pass a number of formulas constructed in this manner—essentially one for each pair of floating-point operation in the program and exception type—to an SMT solver. In the current version of our tool, and like the approach in [2], we translate these formulas into the SMT theory of Real Arithmetic (RA), rather than Floating-Point Arithmetic (FPA). This has the advantage of greater flexibility and efficiency, but the disadvantage of the semantic disagreements between the two theories: a satisfying assignment returned by the solver, once converted into a floating-point assignment to the program variables, is not guaranteed to lead to an exception, despite the specification; nor does an "unsatisfiable" result guarantee the absence of an input triggering the targeted exception. However, as argued in several earlier works, a satisfying RA assignment suggests that a satisfying floating-point assignment is "likely" to be found nearby [2], [4], [5]. We keep this in mind for Step 2 (Section IV).

Implementation. Since the earlier work by Barr et al. did not provide a tool, we built our own symbolic execution engine, implemented in Python. Our goal is to compare (the exception behavior of) programs compiled with and without optimizations applied to them—these optimizations are typically applied not at the source-code level but to the code in an intermediate compiler language such as LLVM's IR. Our Python program therefore takes as input programs $\mathbb{P}$ and $\mathbb{P}'$ in LLVM IR, parses them into an AST using LLVMLITE, and constructs the required SMT formulas, augmented with subformulas encoding exceptions, according to Table I.

We use Z3 as the back-end SMT solver [6]. The result returned by Step 1 is the set of satisfying assignments (in RA) obtained from both programs $\mathbb{P}$ and $\mathbb{P}'$. Since the two programs differ only in the optimizations applied in $\mathbb{P}'$, the formulas generated from them typically overlap. We use simple syntactic checks to avoid passing formulas easily seen to be equivalent to the solver. We also eliminate duplicates from the final set of satisfying assignments obtained. Note that, for Step 2, it is irrelevant which program the satisfying assignment was generated from—our approach is "symmetric".

TABLE I

SUBFORMULAS DEFINING THE PRESENCE OF EXCEPTIONS OF THE VARIOUS TYPES; $\odot \in \{\oplus, \ominus, \otimes\}$. DBL_MAX IS THE LARGEST FINITE REPRESENTABLE DOUBLE VALUE, AND DBL_MIN IS THE SMALLEST POSITIVE REPRESENTABLE VALUE (A NORMAL FLOATING-POINT NUMBER).

Expression	**Overflow**	**Underflow**	**Invalid**	**Divide by Zero**
$x \odot y$	$\lvert x \odot y \rvert >$ DBL_MAX	$0 < \lvert x \odot y \rvert <$ DBL_MIN	*false*	*false*
$x \oslash y$	$\lvert x \rvert > \lvert y \rvert \cdot$ DBL_MAX	$0 < \lvert x \rvert < \lvert y \rvert \cdot$ DBL_MIN	$x = y = 0$	$x \neq 0 \wedge y = 0$

IV. Inputs Triggering Distinct Exception Traces

The output of Step 1 is a set of real-number tuples that, when rounded to their nearest floating-point tuples, are "likely" [2] to trigger an exception as inputs in $\mathbb{P}$ or $\mathbb{P}'$ or both. The goal now is to search for inputs that trigger deviating exception behavior in the two programs. This first of all means they must trigger an exception in at least one program and, according to Section II, they must trigger different exception traces.

Search space. The uncertainty associated with the capability of Step-1 inputs to trigger exceptions stems from the rounding of the real numbers to floating-point inputs, and of course from the differences in the semantics of RA and FPA. We borrow here the proposal by Barr et al. that, if the RA assignment satisfies the exception condition, then a floating-point number is likely to be found close to it that triggers the exception [2]. We therefore include in the search inputs in some small neighborhood of each rounded input returned by Step 1. For each such input we now execute both $\mathbb{P}$ and $\mathbb{P}'$. If neither run raises any exception, we discard this input. If at least one program does, we move on to the next sub-step.

Distinguishing exception traces. Defining precisely what it means for two exception traces to differ is not obvious since "trace equality" should mean that the *same* exceptions occur at the *same* locations in the programs. But what are "same locations"? Programs $\mathbb{P}$ and $\mathbb{P}'$ can be structurally quite different—we are at the mercy of the optimizer as to how much rewriting it performs. Consider the subexpression highlighted in red in Equation (1), and an optimized version that distributes the outer multiplication and then folds constants:

```
original:    0.125 * (3.0 - 2.0 * v)
optimized:   0.375 - 0.25 * v
```

For values of `v` near `DBL_MAX`, both expression evaluations will raise an overflow exception. It is now for the implementation to decide whether this happened at the "same location", or whether the two exceptions should be considered different because the arguments to the overflowing multiplication are not the same. In summary, due to the code changes incurred by the optimization, there is no obvious definition of "same" or "corresponding" program locations.

In the current stage of this work, we chose to drop the location requirement but enforce order. That is, we define an exception trace to be the trace of exception types observed when running a program, irrespective of the exact location where it occurs, or the exact operation that triggers it, but respecting the order in which they occur in the program, if several (which is common). 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, we do not consider the traces different.

The reason for this design choice is, next to its simplicity, is that it is compatible with the testing flavor of this work, where we do not expect false positives: any traces differences reported by our technique are definitely a witness of optimizations impacting exception behavior. Despite this strict definition, we found differences in many of our test programs.

Implementation. We have implemented an LLVM pass that calls a `check_for_exception` function is this built in to LLVM, or did you write this function? after every floating point instruction. Function `check_for_exception` queries the floating point status word [7] to determine if an exception has occurred, and its type. If one has occurred it records this in a global variable. is this global variable a list, or a scalar? if scalar, it is not clear to me how it can store a trace After the execution of the program, this variable therefore stores the trace of exceptions that occurred.

The given source program is compiled into $\mathbb{P}$ and $\mathbb{P}'$, both are instrumented using the pass described above, and both are linked to a driver program (written in `C++`). Candidate inputs are read from a file (produced by Step 1). We run both $\mathbb{P}$ and $\mathbb{P}'$ on each input. If no difference is found, we search in a neighborhood of the input. This search uses the `nextafter` function [8] to determine the next neighboring lower and higher floating-point values. The search continues until a trace difference is found, or the search radius r is reached. If the input to the program consists of $n > 1$ individual arguments, the neighborhood search is performed for each argument independently, so that the n-dimensional search space is exhaustively explored.

Algorithm 1 summarizes the search routine in the form of pseudo code. Inputs are the compiled programs $\mathbb{P}$ and $\mathbb{P}'$, and the list of input candidates obtained during Step 1. The routine iterates through n-tuples of floating-point values $X \in FP^n$ within a cube (not ball) of radius r around any input $\mathcal{I}$ found in Step 1. Lines 3 and 4 execute $\mathbb{P}$ and $\mathbb{P}'$ on X and project the execution traces to the traces of observed exceptions. If these traces differ, we report input X, along with the point of divergence between the two traces, abbreviated as $div(t, t')$ in Algorithm 1. what exactly do we report: the first point of divergence? all of them?

Algorithm 1 $Search(P, P'; \mathcal{II})$

Input: P, P': LLVM IR of $\mathbb{P}$ and $\mathbb{P}'$;
$\mathcal{II}$: list of inputs (n-tuples) obtained in Step 1
Output: offending input, point of trace divergence

1: **instrument** P and P' to report raised exceptions
2: **for** $X : X \in FP^n \wedge \exists \mathcal{I} \in \mathcal{II} : |\mathcal{I} - X|_\infty \leq r$ **do**
3: $t = P(X).trace()$
4: $t' = P'(X).trace()$
5: **if** $t \neq t'$ **then**
6: **output** $X, div(t, t')$

V. Preliminary Results: Impact of Code Optimizations on Floating-Point Exceptions

We have evaluated our implementation on several benchmark programs, shown in Table II. These programs all use `double` floating-point numbers and are straight-line programs except for `odometer`, which has a simple loop, which we fully unrolled at the LLVM IR level. The `identity` program, shown here, was hand-crafted to be reduced to a no-op by fast-math optimizations. for jetengine, carbongas, the # of LLVM instructions does not change during the optimization. Can you check whether there was any reduction at all? It seems if any constant folding is happening, there should be a reduction. If not, we can include one such example here, but perhaps not two

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

TABLE II
BENCHMARKS

Program	#args	#LOC	#LLVM instr. ($\mathbb{P}$, $\mathbb{P}'$)
`identity`	1	2	(2, 0)
`turbine1`	3	1	(14, 11)
`turbine3`	3	1	(14, 11)
`jetengine`	2	6	(27, 27)
`carbongas`	2	7	(7, 7)
`odometer`	1	17	(55, 39)

The results are shown in Table III. The search radius r was set to 3 in each direction; larger radii did not yield different results. please define the column headers in Table III (very important in any piece of scientific work). Especially define "diff producing" carefully. We need to add an interpretation of these results—currently there is none

For `identity`, `turbine1`, `turbine3`, and `odometer` we were able to find inputs for which $\mathbb{P}$ and $\mathbb{P}'$ produced different traces of exceptions. From Table II 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. to be revised

TABLE III
RESULTS

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 iter's)	190 (122, 76)	167	70	15

VI. Related Work

We consider the analysis of differences in the exception behavior of programs due to code optimizations as an extension of research investigating the computational equivalence of programs and their optimizations. For floating-point programs, such equivalence can in principle be verified using SMT solvers for the FPA theory, such as the one proposed in [9]. This theory has support for special values like infinities, but does not account for "operational aspects" [9] like exceptions. By their very nature, exceptions are difficult to cleanly integrate into logical theories, suggesting alternative approaches to analyze their occurrence in programs.

With a somewhat different objective in mind, the Alive-FP tool can be used to confirm the bit-precise correctness of floating-point peep-hole optimizations that are supposed to preserve accuracy [10]. In this paper we look at a broader class of optimizations (including those known to modify computational results). In addition, our analysis is intended to be applied to programs, not rewrite rules.

We rely in this work on the ideas for detecting floating-point exceptions in programs by Barr et al. [2]. That work proposed the use of symbolic execution in connection with real-arithmetic solvers (a proposal that is worth reconsidering at this time; see Section VII), and of a neighborhood search, to bridge the semantic gap between real- and floating-point arithmetic. Since we were unable to get hold of their implementation, we built our own symbolic execution engine.

More broadly, our work is an instance of research investigating the impact of code optimizations on side effects of computations other than (time, space) efficiency. Other examples of such effects include the compromise of security properties [11]. Incidentally, even if computational results are not affected, the presence or absence of exceptions can cause timing channels in floating-point arithmetic, which have been exploited in other work [12].

VII. Summary and Future Work

This paper reports on ongoing work in the area of instability caused by aggressive floating-point exceptions. "Instability" here means that the occurrence of arithmetic exceptions is impacted by the optimizations. Exceptions, if caught, provide an important signal to the programmer that something unusual happened in the calculation and should normally not be ignored. Optimizations suppressing or adding exceptions to a program are therefore a reason for concern.

We have sketched in this paper our current approach to detecting such instabilities, and demonstrated that this problem is real: it occurs on many of a selection of small programs, and is widespread in some. There are many obvious directions to continue this work:

- use floating-point, not real-arithmetic, solvers to determine candidate inputs (triggering exceptions) in Step 1;
- come up with a tighter definition of exception trace equality, one that takes the location where the exceptions occur into account. We are looking for some kind of (stuttering) (bi-)simulation relation between $\mathbb{P}$ and $\mathbb{P}'$;
- extend the implementation to programs with loops, programs with non-floating-point data types, and generally to larger programs;
- consider *proofs of optimization stability*: formal equality of exception traces. Currently our approach has the flavor of difference detection and is at times ad-hoc (for instance, the neighborhood search).
- consider ways to *stabilize* exceptions against optimizations. LLVM permits applying fast-math rules on a per-program-location basis. Those that cause deviations in exception behavior can then be omitted.

REFERENCES

[1] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *Numerical Software Verification - 9th International Workshop, NSV 2016, Toronto, ON, Canada, July 17-18, 2016, [collocated with CAV 2016], Revised Selected Papers*. Toronto, ON, Canada: IEEE, 2016, pp. 63–77.

[2] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Symposium on Principles of Programming Languages (POPL)*, 2013, pp. 549–560.

[3] IEEE Standard Association, "IEEE standard for floating-point arithmetic," 2008, http://grouper.ieee.org/groups/754/.

[4] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl, "Make it real: Effective floating-point reasoning via exact arithmetic," in *Design Automation and Test in Europe (DATE)*, 2014, pp. 1–4.

[5] J. Ramachandran and T. Wahl, "Integrating proxy theories and numeric model lifting for floating-point arithmetic," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 153–160.

[6] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[7] Status bit operations (the GNU C library). [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Status-bit-operations.html

[8] nextafter(3) – Linux manual page. [Online]. Available: http://man7.org/linux/man-pages/man3/nextafter.3.html

[9] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl, "An automatable formal semantics for IEEE-754 floating-point arithmetic," in *Symposium on Computer Arithmetic (ARITH)*, 2015, pp. 160–167.

[10] D. Menendez, S. Nagarakatte, and A. Gupta, "Alive-fp: Automated verification of floating point based peephole optimizations in LLVM," in *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, 2016, pp. 317–337.

[11] K. Papagiannopoulos and N. Veshchikov, "Mind the gap: Towards secure 1st-order masking in software," in *Constructive Side-Channel Analysis and Secure Design*, 2017.

[12] C. Gongye, Y. Fei, and T. Wahl, "Reverse engineering deep neural networks using floating-point timing side-channel (to appear)," in *Design Automation Conference (CAV)*, 2020.