The Impact of Code Optimizations on Floating-Point Exceptions (Short Regular Paper)

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, a widely available approximation of real-valued arithmetic, 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, such as those permitted by the fast-math family of optimizations.**

In this paper we investigate the impact of optimizations on floating-point *exceptions*. Exceptions are raised, for instance, when floating-point calculations over- or underflow. 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. The effectiveness of such techniques is, however, greatly affected when optimizations cause enough disruptions 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. Our tool then uses a neighborhood search that attempts to find inputs where the exception behavior of the program and an optimized version diverge. We demonstrate experimentally that the problem of floating-point exceptions impacted by code optimizations is real, and wide-spread in some programs.

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 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

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

(a simplified version of the `turbine1` example [1]), 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 value-preserving in floating-point. 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 future 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. 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 root-cause analyses are almost inevitably very expensive.

The goal of the approach proposed in this paper is to 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. 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:   -2.5000000000000000000e-01
   I:   w:   +1.3407807929942597099e+154
        r:   -1.2500000000000000000e-01
```

Program $\mathbb{P}$ returns a finite (if large-magnitude) value and thus passes (on $\mathcal{I}$) the assume/guarantee 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 offer a tool, so we implemented a similar technique for our purposes. Our technique operates at the level of the intermediate representation of a compiler, such as LLVM IR, since code optimizations are applied at this level. This allows us to compute candidate inputs for raising exceptions from either $\mathbb{P}$ or $\mathbb{P}'$, 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 $\mathcal{I}$, indeed triggers exceptions in the program from which it was obtained in the first step (see Section III), 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 of code optimizations on the exception behavior of floating-point code is real, and in fact occurs frequently in some programs. We briefly suggest ways to stabilize exceptions against optimizations.

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. A certain number of bits is reserved for each of these, depending on the 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 possibly disruptive outcomes. The Standard defines the following exception *types*: [**inexact**] a finite mathematical result that must be rounded to fit into the format; [**overflow**, **underflow**] a finite mathematical result that is too large or too small to fit into the format; [**divideByZero**] an infinite mathematical result, such as $1/0$; and [**invalid**] a non-mathematical result, such as $0/0$. Inexact exceptions are very common, since many operations require rounding. To denoise our report, we exclude them in this study.

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 $\mathcal{I}$. 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. (For the purposes of this study, programs are loop-free, so their traces are finite.) 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 Candidate Inputs

Given program $\mathbb{P}$ and an optimized version $\mathbb{P}'$, our overall approach consists of two steps:

Step 1 (this section) Find candidates for inputs that cause an exception to be triggered in either $\mathbb{P}$ or $\mathbb{P}'$;

Step 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 every path to every *elementary* floating-point operation ($\oplus$, $\ominus$, $\otimes$, $\oslash$). Then we conjoin the path formula with the conditions that will trigger the various exceptions, most of them shown in Table I. Note that in Step 1 we do not handle "secondary" exceptions: those caused by operations on arguments that are already exceptional, such as an invalid exception raised by computing $0 \cdot \infty$. The reason is the use of real arithmetic as the back-end theory; see below. However, Step 2 still observes such exceptions if they occur.

We now 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 as done 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 more solver support and greater efficiency, but suffers from semantic differences between the two theories: a satisfying RA assignment returned by the solver, once converted into a floating-point assignment to the program variables, does not guarantee 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 behaviors of programs compiled with and without optimizations. 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.

We use Z3 as the back-end SMT solver [6]. The result returned by Step 1 is the set of satisfying RA assignments 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. Note that for Step 2 it is irrelevant which program the satisfying assignment was generated from—our approach is "symmetric".

IV. Inputs Triggering Different 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 in $\mathbb{P}$ or $\mathbb{P}'$ or both. The goal now is to search for inputs that trigger deviating exception behavior

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 *normal* FLOATING-POINT VALUE (DEFINED IN <FLOAT.H>).

Expression	**overflow**	**underflow**	**invalid**	**divideByZero**
$x \odot y$	$\lvert x \odot y\rvert >$ 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	$x = y = 0$	$x \neq 0 \wedge y = 0$

in the two programs. This first of all means they must trigger an exception in at least one program.

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 input 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 amount of rewriting the optimizer 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 different. 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 liberal in what it considers "same". For example, if both programs produce a single overflow but at different locations, we still consider the traces identical.

Another reason for this design choice is the testing flavor of this work, where we do not expect false positives. Indeed, any trace difference reported by our technique definitely witnesses optimizations messing with exception behavior. With our definition of "same trace", we found differences in many of our test programs.

Implementation. We implemented a function that queries the floating-point status word [7] to determine if an exception has occurred. If so, the function appends it and its type to a global list variable L. We built an LLVM pass that calls the function after every floating-point instruction. At the end of the program execution, L stores the trace of encountered exceptions.

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. The neighborhood search is performed for each of the n individual program arguments independently, so that the n-dimensional search space is exhaustively explored.

Algorithm 1 summarizes the search routine. Inputs are the compiled programs $\mathbb{P}$ and $\mathbb{P}'$, and the list of candidates inputs obtained by 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 (we use $r = 3$ in all $2n$ directions in our implementation). Lines 3 and 4 execute $\mathbb{P}$ and $\mathbb{P}'$ on X and project the execution traces to the traces of observed exception types. If these traces differ, we report them, along with the offending input X.

Algorithm 1 $\mathit{Search}(P, P'; \mathcal{I}s)$

```
Input:  $P, P'$: LLVM IR of $\mathbb{P}$ and $\mathbb{P}'$;
        $\mathcal{I}s$: list of inputs ($n$-tuples) obtained in Step 1
Output: diverging traces, offending input
1: instrument $P$ and $P'$ to report raised exceptions
2: for $X : X \in FP^n \wedge \exists \mathcal{I} \in \mathcal{I}s : |\mathcal{I} - X|_\infty \leq r$ do
3:     $t = P(X).\mathit{trace}()$
4:     $t' = P'(X).\mathit{trace}()$
5:     if $t \neq t'$ then
6:         output $t$, $t'$, $X$
```

V. PRELIMINARY RESULTS

We have evaluated our implementation on several benchmark programs, shown in Table III. These programs all use double floating-point numbers and are straight-line except for odometer, which has a simple loop, which we fully

TABLE II
RESULTS. Entry x (y, z) in column **Formulae ($\mathbb{P}$, $\mathbb{P}'$)**: y formulas were generated for $\mathbb{P}$, one per applicable instruction and exception type; similar for z and $\mathbb{P}'$. Quantity x is the size of the union. Remaining columns: number of satisfiable formulas (out of x); number of non-duplicate inputs generated from the satisfying assignments; number of such inputs that produced a difference in the exception trace

Benchmark name	**#formulas ($\mathbb{P}$, $\mathbb{P}'$)**	**#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

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.

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

TABLE III
BENCHMARKS

Benchmark name	**#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 II (top of this page). We see a rough correlation between the impact of the optimization (measured in terms of instruction count difference between $\mathbb{P}$ and $\mathbb{P}'$) and the number of "diff producing" inputs we found. For example, for `turbine1`, nearly half the candidate inputs generated different exception traces for $\mathbb{P}$ and $\mathbb{P}'$. At the other end of the spectrum, no differences were found for the `jetengine` and `carbongas` programs. By Table III, in both cases $\mathbb{P}$ and $\mathbb{P}'$ feature equal LLVM instruction counts. While the optimization did rewrite these programs ($\mathbb{P}$ and $\mathbb{P}'$ are not equal), the numeric impact of the optimization is small.

VI. RELATED WORK

The equivalence of a (loop-free) floating-point program and an optimized version 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 SMT theories, suggesting alternative approaches to analyzing 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 in optimized code[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 on the impact of aggressive floating-point optimizations on the occurrence of arithmetic exceptions. If caught, exceptions provide an important signal to the programmer that something unusual happened in the calculation, which should normally not be ignored. Optimizations suppressing or adding exceptions to a program therefore are a reason for concern. We have sketched in this paper our current approach to detecting instances of this problem, and demonstrated that it is real: it occurs on many of a selection of small programs, and is widespread in some.

There are many exciting directions to continue this work:

- use floating-point, not real-arithmetic, solvers to determine 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 (this requires some kind of [stuttering] bisimulation relation between $\mathbb{P}$ and $\mathbb{P}'$);
- extend the implementation to programs with loops or non-floating-point data types, and to larger programs;
- consider *proofs of optimization stability*: formal equality of exception traces. Currently our approach has the flavor of ad-hoc difference detection;
- consider ways to *stabilize* exceptions against optimizations. LLVM permits applying fast-math rules on a per-program-location basis. This allows us to omit those that cause deviations in exception behavior.

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.