Stability-Preserving Unsafe Compiler Optimizations for Numeric Programs

ANONYMOUS AUTHOR(S)

Code optimizations are typically expected to preserve the input/output relationship defined by a program. This requirement is too strong, however, for floating-point data types, where even cautious expression rewriting can cause changes in the output. A relaxed notion of correctness permits aggressive *unsafe* optimizations—supported by many compilers—, which trade in bit-precise I/O equivalence for performance gains. Few studies exist on the questions whether the impact of such optimizations on the numeric output is tolerable, and how to reduce that impact without significantly affecting efficiency.

To address both questions, this paper presents a technique that determines whether, for every input to a floating-point program, the original and the optimized code are equivalent in terms of *numeric stability*, a widely used metric that estimates the susceptibility of programs to both computational rounding errors and input sensitivities. Further, our technique identifies individual expressions in the code whose optimizations are largely to blame for stability changes, if any; the optimizations are disabled *for these expressions*. Our implementation, which handles the unsafe fragment of the LLVM compiler's InstCombine pass, returns to the programmer a stability-equivalent optimized floating-point program that enjoys near-uncompromised efficiency benefits.

ACM Reference Format:
Anonymous Author(s). 2019. Stability-Preserving Unsafe Compiler Optimizations for Numeric Programs. 1, 1 (July 2019), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Compilers optimize code to increase time or memory efficiency. 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, just with reduced resource consumption. This ensures contextual equivalence of the two code versions.

This expectation is unrealistic, however, for data types with fragile I/O semantics, a notorious example of which are floating-point numbers. Intended to simulate real-number calculations, they seem to invite the use of high-school math simplifications to perform compile-time arithmetic on constants, or rewrite expressions into seemingly equivalent forms. Consider the assignment

```
y = 0.5 / x * 0.5 + 2.0 / x        (1)
```

which tempts the compiler to simplify (1) to `y = 0.25 / x + 2.0 / x` and then further to `y = 2.25 / x`, reducing the assignment to a single operation. This can amount to non-trivial efficiency gains when the assignment happens in a loop.

Such simplifications are, however, not value-preserving in floating-point. The second step above, for instance, apparently exploits distributivity of addition over division, but `a/x + b/x` is not I/O-equivalent to `(a+b)/x` in floating-point.[1] For this reason, optimization switches that turn on aggressive expression rewriting using real-algebra identities (valid in $\mathbb{R}$), such as from the fast-math family (available on `gcc`, `LLVM`, and Microsoft Visual `C++`), silently assume that the programmer is aware that doing so may change the computed floating-point value and is thus *unsafe*.

Aware or not, a significant consequence of the loss of I/O-equivalence is that analyses performed at the source code level may be invalidated by the optimization. For example, in a sensible

[1]For instance, the outputs of (1) and the final optimized version differ in double precision for `x = 1,000`.

2019. XXXX-XXXX/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

numeric-software development process, the programmer is first concerned with the "mathematical correctness" of the implementation, followed by precision tuning using testing or tools like PRECIMONIOUS [32] or FPTUNER [6]. Value-changing optimizations sabotage source-level code tuning techniques.

This paper aims to both increase the accountability of aggressive floating-point optimizations, and reduce their negative side effects, by addressing the following questions:

(1) How can we formalize the idea that a numeric optimization alters the output value *too much* to be justified, and how can this be efficiently determined?
(2) How can we reduce this effect in a way that retains some or much of the optimization benefits?

To address **Question (1)**, we propose in this paper to admit an optimization if, for each input, the unoptimized and optimized programs $\mathbb{P}$ and $\mathbb{P}'$ are equivalent in terms of numeric *stability* [15]. A widely used concept, stability of a program $\mathbb{P}$ requires that, for each input, the ratio between the rounding error and the *(relative) condition number* [33] of the mathematical function $\mathbb{P}_0$ approximated by $\mathbb{P}$ is small. Intuitively, this means that rounding errors can be explained via small changes in the input. In addition, the condition number helps "cut some slack" to *ill-conditioned* mathematical problems, where function $\mathbb{P}_0$ is highly sensitive to input changes.

Toward stability equivalence, we quantify the stability difference between $\mathbb{P}$ and $\mathbb{P}'$, and determine whether this quantity is too large, in two steps. First, we estimate the pointwise rounding error difference introduced by the optimization, as a function of the input. This estimate, which we call the *sensitivity difference* between the programs, includes the effects of propagating rounding error changes on their way to the program output. Note that any numeric operation, including those unmodified by optimizations, can amplify rounding error changes, so we must consider propagation.

Second, we declare the original and the optimized programs stability-equivalent if what we call the *stability difference coefficient* is, for all inputs, at most 1. In analogy to individual programs, this coefficient explains sensitivity differences via small changes in the input.

Intuitively, stability equivalence of $\mathbb{P}$ and $\mathbb{P}'$ suggests that, for every input, the two programs show similar rounding-error behavior. In this case, we also say the optimization *preserves* (otherwise, it *corrupts*) stability. We propose this notion as a specification for aggressive floating-point compiler optimizations for cases where bit-precise I/O-equivalence is too rigid. We discuss alternative specifications, and why we believe they are less appropriate in general, in Sec. 2.

Toward **Question (2)**, if the optimization corrupts stability, we use the expression-level stability change information computed by our technique to determine how much each individual optimization is to blame for the overall stability difference. We then employ a greedy strategy and disable unsafe optimizations for the most severely affected expressions, i.e., we "freeze" them, such that the resulting partially optimized program is stability-equivalent to $\mathbb{P}$. The stability difference after partially disabling optimizations can be determined without running the entire analysis again, so that selecting the smallest set of expressions to freeze is inexpensive.

We have incorporated our technique into the LLVM compiler framework; it handles all unsafe optimizations applied by the INSTCOMBINE pass of this compiler (this pass is invoked as part of the fast-math optimizations; all INSTCOMBINE rewritings are valid in real arithmetic; a fact that we exploit). We have modified INSTCOMBINE to a new pass, ICDIFF, such that running ICDIFF over program $\mathbb{P}$ computes an expression quantifying the sensitivity difference introduced by the optimizations. Towards determining whether the optimizations corrupt stability, we offer two implementations to maximize the difference coefficient over all inputs (the maximum is then compared to 1): one based on interval analysis, and one based on stochastic search. The two

implementations feature complementary strengths in terms of efficiency and formal guarantees. We evaluate them on a number of benchmarks widely used in the floating-point community, such as FPBench [10], and various other programs used in prior work. Our implementations permit the following observations and conclusions.

- For roughly half of our benchmark programs, the InstCombine optimizations corrupt stability. Since we approximate the exact rounding error differences by sensitivity differences, we confirm using a sampling technique that programs $\mathbb{P}$ and $\mathbb{P}'$ that are not stability-equivalent indeed suffer from a relatively large maximum pointwise rounding-error difference, while the others do not.
- For the case of corrupted stability, the rounding behavior of only few expressions, often a single one, is significantly affected by the optimization pass. This justifies our strategy to disable the optimization only for these expressions, i.e., to freeze them.
- After freezing the affected expressions, the InstCombine optimizations preserve stability for all programs. We again sample the pointwise rounding-error differences and this time confirm that they are small. Finally, we observe that the speed-up of the stability-preserving optimization is between 75% and 100% of the speed-up of the stability-corrupting optimization.

2 MOTIVATION

We use the program in List. 1, call it $\mathbb{P}$, to illustrate our goal. $\mathbb{P}$ is automatically generated by the transformation technique in [9] and provides ample opportunity for optimizations. We investigate the impact of LLVM's InstCombine pass, which is performed as part of the fast-math optimization flags, on the values computed by the program. The pass uses real-algebra laws to simplify expressions, which affects almost every line in List. 1. Let $\mathbb{P}'$ denote the optimized version of $\mathbb{P}$. "Real-algebra laws" means that, with infinite precision, the two programs compute the same function.

```
1  void position(double sl) {                                          // output produced in x,y
2    double theta = 0.0, y = 0.0, x = 0.0;
3    for (int i = 0; i < 100; i++) {
4      double TMP_6 = 0.1 * (0.5 * (9.691813336318980 - (12.34 * sl)));
5      double TMP_23 = (theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5)) *
6                       (theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5));
7      double TMP_25 = (theta + TMP_6) * (theta + TMP_6) *
8                       (theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5);
9      double TMP_26 = (theta + TMP_6);
10     x = (0.5 * (((1.0 - (TMP_23 * 0.5)) + ((TMP_25 * TMP_26) / 24.0))
11          * ((12.34 * sl) + 9.691813336318980))) + x;
12     double TMP_27 = (TMP_26 * TMP_26) * (theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5));
13     double TMP_29 = ((TMP_26 * TMP_26) * TMP_26) * (theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5));
14     y = ((9.691813336318980 + (12.34 * sl)) *
15          (((TMP_26 - (TMP_27/6.0)) + ((TMP_29 * TMP_26) / 120.0)) * 0.5)) + y;
16     theta = theta + (0.1 * (9.691813336318980 - (12.34 * sl))); }}
```

Listing 1. Computing the position of a 2-wheeled robot [9]

One metric to estimate the impact of numeric code transformations is to compare the maximum rounding errors of $\mathbb{P}$ and $\mathbb{P}'$. To get an idea of these errors, we uniformly selected a set *SL* of 10,000 samples from the input interval $[0.5, 0.6]$ and computed the rounding errors for both programs, reported in row *before*, Columns 1 and 2 of Table 1 (in ulp = *unit in the last place*: the place value of

the least significant digit in the floating-point representation, which varies by number). We observe that these two numbers are nearly identical.[2]

	1 $\max_{\texttt{sl}\in SL} \text{re}^{\mathbb{P}}_{abs}(\texttt{sl})$ (ulp)	2 $\max_{\texttt{sl}\in SL} \text{re}^{\mathbb{P}'}_{abs}(\texttt{sl})$ (ulp)	3 $\max_{\texttt{sl}\in SL} \Delta^{re}_{\mathbb{P},\mathbb{P}'}(\texttt{sl})$ (ulp)	4 Speedup (%)
disabling fast-math: before	84.20	86.39	82.70	22.4
after	(unchanged)	84.20	5.91	22.0

Table 1. Effect of InstCombine on the value of x, before and after disabling fast-math in Line 16

These results might suggest that the rounding errors of $\mathbb{P}$ and $\mathbb{P}'$ are similar. There are, however, many inputs for which they differ significantly. To see this we have computed, for each $\texttt{sl} \in SL$, the difference $\Delta^{re}_{\mathbb{P},\mathbb{P}'}(\texttt{sl}) = |\text{re}^{\mathbb{P}}_{abs}(\texttt{sl}) - \text{re}^{\mathbb{P}'}_{abs}(\texttt{sl})|$ between the rounding errors of $\mathbb{P}$ and $\mathbb{P}'$ in output variable x; the distribution is shown in Fig. 1 (left).

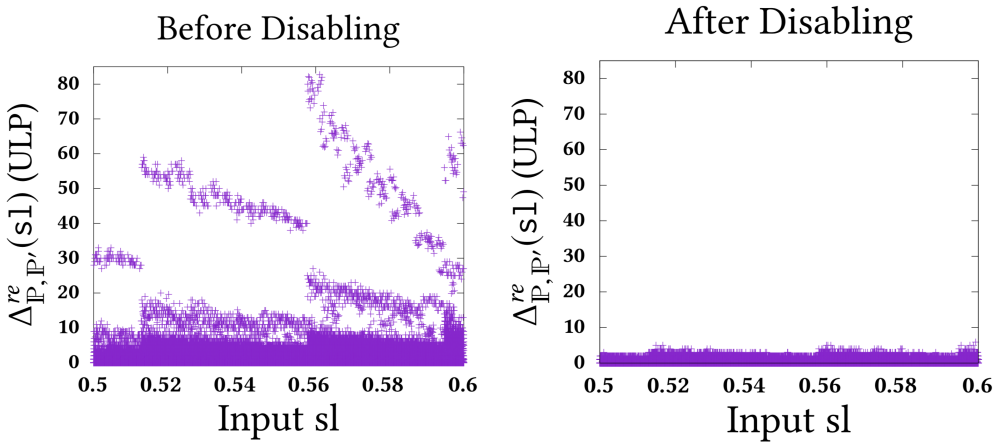

Fig. 1. Rounding error difference comparison $\Delta^{re}_{\mathbb{P},\mathbb{P}'}(\texttt{sl})$, before (left) and after (right) stabilization. While on the right all inputs induce a rounding error difference below 6ulp, on the left about 5.5% of the sampled inputs (i.e. a non-negligible number) feature a difference greater than 6ulp

The maximum difference, shown in Column 3 in Table 1 (row *before*), reaches about 83ulp. Indeed, evaluating $\text{re}^{\mathbb{P}}_{abs}(\texttt{sl}_{\max})$ and $\text{re}^{\mathbb{P}'}_{abs}(\texttt{sl}_{\max})$ at the point $\texttt{sl}_{\max}$ that maximizes $\Delta^{re}_{\mathbb{P},\mathbb{P}'}(\texttt{sl})$ yields 0.15 and 82.85ulp, resp., which means that the outputs x of the two programs are about 83ulp apart. This indicates that the near-identical maximum rounding errors are misleading.

To detect the problem exhibited in List. 1, we need a more precise analysis, namely one of the programs' *pointwise* rounding error differences. This is computationally a very hard problem (not addressed by existing tools like Rosa [11] and FPTaylor [35]): $|\text{re}^{\mathbb{P}}_{abs}(\texttt{sl}) - \text{re}^{\mathbb{P}'}_{abs}(\texttt{sl})|$ is a highly complex function of sl; Fig. 1 (left) shows its unwieldy behavior, with large magnitude changes across small input variations. In this paper we approximate, reasonably accurately and efficiently, pointwise rounding error differences using pointwise *sensitivity differences*, which in turn are used

[2]Maximum rounding errors can also be estimated using the FPTaylor tool [35]. We did not use it in this section, since it runs out of memory for program $\mathbb{P}$ over the given input interval (it can only handle a small number of iterations of the for loop). It is also not directly applicable to optimized code (which comes in LLVM IR representation).

to determine the maximum *stability difference* between the two programs. For List. 1, our technique reports that the programs are not equivalent in terms of numeric stability. Furthermore, toward fixing this problem, the technique determines that the main culprit for the gap in stability is Line 16, the computation of `theta`. This information is used to eliminate the stability-corrupting effect of the optimizations, namely by disabling fast-math *for Line 16 only*.

Fig. 1 (right) shows the rounding error differences after disabling fast-math for Line 16. The maximum difference has dropped by more than one order of magnitude, from 82.70 to 5.91, as shown in Column 3 (row *after*) in Table 1. Column 4 compares the speed-up effect of InstCombine before and after freezing Line 16: the result is that 98% of the fast-math speed-up is retained.

Discussion: alternative difference measures. We have so far demonstrated that comparing a program $\mathbb{P}$ and its optimization $\mathbb{P}'$ via their maximum rounding errors over a given input interval is not a good specification for an optimizing compiler: the expectation is not that the optimization produce a similar *worst-case* behavior, but similar (ideally, identical) results per input. The same problem persists in various flavors of this definition, for instance if one were to (asymmetrically) require, for an optimization to be admissible, that the maximum rounding error of the optimized program be below some suitable threshold.

A more subtle alternative, which acknowledges the input-dependence of rounding error sensitivities of a program, is to require that, for each input I, the rounding error in $\mathbb{P}'(I)$ should be no more than that of $\mathbb{P}(I)$. This solution (also asymmetric) allows $\mathbb{P}'$ to be more accurate than $\mathbb{P}$. Accuracy, however, is not monotone: *in*creasing it for an intermediate result can of course *de*crease it for the final result, or have other unexpected consequences. This was ultimately the cause for the infamous *patriot missile failure*; more concretely, "the introduction of a subroutine for converting clock-time more accurately" in some places of the code [30]. The following example illustrates the problem, using the familiar assignment (1):

```
double diff(double x) {
  double P; // the assignment from (1), but evaluated incrementally from left to right
  P  = 0.5 / x;
  P *= 0.5;
  P += 2.0 / x;

  double Pp = 0.5 / x * 0.5  + 2.0 / x; // the assignment from (1)

  return (P - Pp); }
```

The example is designed such that `-ffast-math` optimizes the expression for `Pp`, but not the code for `P`. For input $x = 1000$, the two expressions evaluate to the same (0.00225) in infinite precision. In floating-point, using `gcc` version 7.4.0 with `-ffast-math`, the value of intermediate result `Pp` is more precise than without, which satisfies the above "more subtle alternative" specification. The final output, however, which is 0 in unoptimized floating-point (and in math), becomes $\approx$ `4.3E-19` with `-ffast-math`.

3 FLOATING-POINT STABILITY AND UNSAFE OPTIMIZATIONS

We review background concepts in support of this paper.

3.1 Floating-Point Arithmetic

Floating-point arithmetic is the most widely used approximation of real-number arithmetic on computers. Its implementation is loosely regulated by the IEEE 754 floating-point standard [16, "the Standard" in this paper]; we consider here the common *binary* variant. The Standard defines a family of floating-point number formats, parameterized by the *precision p* and the maximum *range*

emax.The Standard also defines five rounding modes; we develop the techniques in this paper for the most common mode *round-to-nearest-ties-to-even* (RN). The relative error $\delta(a)$ introduced by rounding a real-valued quantity a to $rd(a)$ is defined as $\delta(a) := (rd(a) - a)/a$ and satisfies $|\delta(a)| \leq u := 2^{-p}$, for the precision parameter p (e.g. $p = 52$ for double precision). Bound u is called *unit roundoff*. As a result, for the floating-point addition operator $\oplus$ and floating-point values a, b, there exists a value δ with $|\delta| \leq u$ and

$$a \oplus b = (a + b) \cdot (1 + \delta) \; ; \tag{2}$$

similarly for the other binary floating-point operations, denoted $\ominus$, $\otimes$, $\oslash$. Throughout this paper, we denote the corresponding real-arithmetic operators using the same symbol but without the enclosing circle.

Eq. (2) holds for *normal* numbers [16]: those equal to zero or with absolute value at least 2^{1-emax}. We develop our technique for now assuming the normal number range and the absence of special floating-point data ($\pm\infty$, NaN). Extensions are discussed at the end of Sec. 8.

Numeric stability measures how a floating-point computation is susceptible to rounding errors, taking into account the mathematical properties of the computation. Slightly adapting [15], we call a program $\mathbb{P}$ over input vector I *(forward) numerically stable* if its *stability coefficient*

$$c_{\mathbb{P}} \;=\; \min\{c \in \mathbb{Z}^+ : \forall I : \mathbb{P}_0(I) \neq 0 : \; |\mathbb{P}(I) - \mathbb{P}_0(I)| \;\; \leq \;\; c \cdot u \cdot \boldsymbol{\kappa}_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)|\} \tag{3}$$

is "small". Here, $\mathbb{P}_0$ is the mathematical function approximated by $\mathbb{P}$, and $\boldsymbol{\kappa}_{\mathbb{P}_0}$ stands for $\mathbb{P}_0$'s *(relative) condition number*, which measures how a mathematical calculation is affected by changes in the input [33]:

$$\boldsymbol{\kappa}_{\mathbb{P}_0}(I) \;=\; \max\left\{ \sum_i \left| I_i \cdot \frac{\partial \mathbb{P}_0}{\partial I_i}(I) \right| \; / \; |\mathbb{P}_0(I)| \;\; , \;\; 1 \right\} . \tag{4}$$

We define $\boldsymbol{\kappa}_{\mathbb{P}_0}(I)$ to be at least 1, in order to ensure that (3) permits only multiples ≥ 1 of $u \cdot |\mathbb{P}_0(I)|$ as an upper bound of the rounding error. The absolute notion of a "small" stability coefficient is informal; we use stability to compare the behavior of two programs.

3.2 Evaluation Models and Computational Processes

The value computed by a floating-point program depends not only on the program inputs, but also on the way the expressions in the program are evaluated. This includes things like evaluation order of mathematically associative operators, but also the decision whether and where to apply optimizations that rewrite expressions. Floating-point arithmetic lacks invariance under different evaluation schemes.

To formalize this volatility, prior work has introduced the concept of an *evaluation model*, collectively referring to all conditions that affect the evaluation of expressions in programs [14]. The idea is that, after fixing the evaluation model, the value of an expression depends only on its inputs, turning it into a mathematical function. Parsing a source program into an abstract syntax tree, such as in LLVM's intermediate representation (IR), fixes the evaluation model, since now all ambiguity has been resolved. We treat the pre-optimization and post-optimization versions of a program as two evaluation models that determine different source code semantics.

We first present our technique for *straight-line* program segments, i.e. code inside basic blocks; branches, loops, and functions are discussed in Sec. 8. Given an evaluation model, the semantics of a straight-line program can be formalized by converting the program into a real-valued *computational process* [17, 38]. This conversion resolves all floating-point expressions into sequences of unary or binary operations, whose outcome is defined by the FP Standard and thus depends only on

the inputs. It then encodes these operations as real operations with respective relative errors as in (2). Given a (straight-line) floating-point program $\mathbb{P}$ and an evaluation model, the syntax of a computational process τ is

$$\tau: \quad \begin{array}{rcl} v_1 & := & e_1(I, \delta_1); \\ & \vdots & \\ v_j & := & e_j(U_j, \delta_j); \\ & \vdots & \\ r & := & e_k(U_k, \delta_k); \end{array} \tag{5}$$

where I is a tuple of m input variables, each v_j is an intermediate variable, r is the output variable, e_j is either a unary or binary *real-arithmetic* operation with a relative-error correction factor $(1 + \delta_j)$ in the form (2), and U_j is the input tuple of e_j: each component of U_j is a program input, an intermediate variable, or a constant. Computational process τ defines a (homonymous) function $\tau\colon \mathbb{R}^m \times \mathbb{R}^k \to \mathbb{R}$ over the m inputs and the k relative-error variables $D = \{\delta_1, \ldots, \delta_k\}$, called *process function*. The formal relationship between $\mathbb{P}$ under the given evaluation model and τ is as follows: for each m-tuple input I there exists a valuation $V_I \in \mathbb{R}^k$ of the error variables in D such that $\mathbb{P}(I) = \tau(I, V_I)$. The existence of V_I with this property is guaranteed by Eq. (2).

For a computational process τ we denote by $\tau^{\mathbb{R}}$ the computational process obtained by setting $\delta_j = 0$ for all j; its process function is of the simplified form $\tau_0\colon \mathbb{R}^m \to \mathbb{R}$. We call two processes τ, τ' *real-equivalent* if, for all I, $\tau_0(I) = \tau_0'(I)$.

Example 1. *The floating-point program $r = d_1 \otimes d_2 \otimes d_3$ can be converted into two computational processes by disambiguating the product as either $(d_1 \otimes d_2) \otimes d_3$ or $d_1 \otimes (d_2 \otimes d_3)$:*

$$\tau\colon \begin{array}{rcl} v_1 & := & d_1 d_2 \cdot (1 + \delta_1) \\ r & := & v_1 d_3 \cdot (1 + \delta_2) \end{array} \qquad \tau'\colon \begin{array}{rcl} v_1' & := & d_2 d_3 \cdot (1 + \delta_1') \\ r' & := & v_1' d_1 \cdot (1 + \delta_2') \end{array}$$

The semantics of the process on the left is that, for each input d_1, d_2, d_3 there exist values $\delta_1, \delta_2 \le u$ such that the process computes the same value as the given floating-point program; similarly on the right. Processes τ and τ' are real-equivalent. The InstCombine compiler optimizations produce programs whose computational processes are real-equivalent to the process of the unoptimized program, for any evaluation model.

3.3 Rounding Errors and Sensitivity

Let $\mathbb{P}$ be a floating-point program under a given evaluation model, τ the corresponding computational process and, for each I, V_I a valuation of the error variables in D such that $\mathbb{P}(I) = \tau(I, V_I)$. Due to this relationship, the rounding error of $\mathbb{P}$ can be defined in terms of that of τ:

Definition 2. *The **(absolute) rounding error** of τ is the function $re^{\tau}_{abs}\colon \mathbb{R}^m \to \mathbb{R}$ defined by*

$$re^{\tau}_{abs}(I) = |\tau(I, V_I) - \tau_0(I)| \,. \tag{6}$$

Eq. (6) is not directly computable: while for each input I the existence of an appropriate error valuation V_I is guaranteed, it depends on the inputs in a highly complex fashion. Prior work has proposed to address this problem in several steps. The first is to build the Taylor expansion formula

$$\mathrm{re}^{\tau}_{abs}(I) = \left| \sum_{\delta \in D} \frac{\partial \tau}{\partial \delta}(I, V_I) \cdot \delta + O(\delta^2) \right| , \tag{7}$$

where $O(\delta^2)$ represents higher-order rounding error. The second is to smooth formula (7), as done by the FPTaylor analysis tool [35], by abstracting away the dependence of V_I on the input I and instead treating τ as a function evaluated on independent variables I and D. Maximizing the D

variables then results in an overapproximation of $\mathrm{re}^{\tau}_{abs}(I)$. Recall that Eq. (2) specifies a bound of $[-u, u]$ for the rounding error variables, for the constant unit roundoff u. This bound is the domain for the maximization.

A further simplification is to drop the higher-order term $O(\delta^2)$, as done in some prior work [6, 18, 24], This results in what is called the *sensitivity* of τ in [24] and has a linear relationship with I:

$$I \in \mathbb{R}^m \quad \mapsto \quad \sum_{\delta \in D} \left| \frac{\partial \tau}{\partial \delta}(I, V) \right| \cdot u \in \mathbb{R} \,. \tag{8}$$

Note the arguments (I, V) in (8) compared to (I, V_I) in (6), and the factor $u = \max_{\delta \in [-u,u]} \delta$.

3.4 Unsafe InstCombine Optimizations

The fast-math compiler optimizations, available on many C compilers under varying names, aggressively rewrites expressions in a way that may change the value computed by the expression and is thus unsafe. Specifically, the LLVM compiler suite [20] features an optimization pass called InstCombine, applied under the fast-math switches, which performs many optimizations that replace expressions by new ones that are equivalent in infinite precision but not in floating-point. We summarize in this section the optimizations undertaken by InstCombine. Unaware of a formal documentation, we have extracted the unsafe optimization rules from the LLVM source code. Fig. 2 shows the complete set of unsafe rules. The other optimizations in the pass do not change the output values (i.e. they exploit identities that hold even in floating-point).

Example 3. *Fig. 3 shows how InstCombine optimizes the source code expression* $(0.6 \oslash x) \otimes 0.5 \oplus 2 \oslash x$, *employing rules from the groups* **FoldFMulConst** *and* **Factorization**.

4 STABILITY-EQUIVALENCE ANALYSIS FOR INSTCOMBINE

We present a high-level overview of our technique to address the change in numeric stability caused by InstCombine compiler optimizations. The key component is a new LLVM pass called ICDiff. It shares with the standard InstCombine pass, from which it is derived, the floating-point optimization pattern matching, but differs in the rewriting: our pass generates code that estimates, as a function of the input, the difference between the rounding errors incurred in the expression before and after applying any of the optimization rules in Fig. 2.

Given an input program $\mathbb{P}$ in LLVM IR, our stability-preserving optimization itself consists of the steps shown in Fig. 4. Step 1 is symbolic: it runs the new pass on $\mathbb{P}$, resulting in code that computes *local*, expression-level error sensitivity differences introduced by each optimization rule. Step 2 is quantitative: it takes an input interval and propagates the local differences to the final program result. This is necessary since value changes incurred by each unsafe optimization rule may be amplified during the data flow.

To decide whether the changes affect $\mathbb{P}$'s numeric stability property, Step 3 computes a bound to the error sensitivities, based on condition numbers. If the change is smaller than this bound, we declare the stability is preserved under InstCombine. Otherwise, Step 4 disables fast-math on a minimal set of expressions such that optimizing only the remaining expressions preserves the stability. The final output is produced by passing the program to InstCombine (with the disablements in effect), resulting in a stability-equivalent optimized program $\mathbb{P}'_{\text{match}}$.

We have implemented the above roadmap in a fully automatic, LLVM IR-based optimization pass, intended to replace the unsafe fragment of InstCombine. In the technical part of this paper we describe the steps in detail. After laying some foundations in Sec. 5, Sec. 6 defines our ICDiff pass, and how the gathered error differences between the original and optimized programs are propagated to the program output (Steps 1 and 2). Sec. 7 employs the condition number analysis to determine a

$$(X \odot c_0) \odot c_1 \rightarrow X \odot (c_0 \odot c_1)$$
$$c_0 \odot (c_1 \odot X) \rightarrow (c_0 \odot c_1) \odot X$$
$$(c_0 \odot X) \odot c_1 \rightarrow (c_1 \odot c_0) \odot X$$
$$c_0 \odot (X \odot c_1) \rightarrow X \odot (c_1 \odot c_0)$$
$$(X \odot c_0) \odot (Y \odot c_1) \rightarrow (X \odot Y) \odot (c_0 \odot c_1)$$

(a) Simplification rules Assoc/Commut ($\odot \in \{\oplus, \otimes\}$)

Factorization:

$$(X \otimes Y) \frac{\oplus}{\ominus} (X \otimes Z) \rightarrow X \otimes (Y \frac{\oplus}{\ominus} Z)$$
$$(Y \oslash X) \frac{\oplus}{\ominus} (Z \oslash X) \rightarrow (Y \frac{\oplus}{\ominus} Z) \oslash X$$

SimplifyFAdd:

Input: vector $A = (\langle a_1, x\rangle, \langle b_1, y\rangle, \ldots)$ of FAddends
Output: FAdd instruction, $a' \otimes x \oplus b' \otimes y \oplus \ldots$

- partition the FAddends in A according to equal symbolic (non-constant) part
- collapse each partition into a single FAddend, by folding the constant coefficients
- create FAdd instruction *Result* by accumulating all partition expressions using $\oplus$

return *Result*;

(b) Optimization rules *FAdd*

FoldFDivConst:

$$(X \otimes c_1) \oslash c_2 \rightarrow X \otimes (c_1 \oslash c_2)$$
$$(X \oslash c_1) \oslash c_2 \rightarrow X \oslash (c_1 \otimes c_2)$$
$$c_1 \oslash (X \otimes c_2) \rightarrow (c_1 \oslash c_2) \oslash X$$
$$c_1 \oslash (X \oslash c_2) \rightarrow (c_1 \otimes c_2) \oslash X$$
$$c_1 \oslash (c_2 \oslash X) \rightarrow (c_1 \oslash c_2) \otimes X$$

ConvertFDivToFMul:

$$(X \oslash Y) \oslash Z \rightarrow X \oslash (Y \otimes Z)$$
$$Z \oslash (X \oslash Y) \rightarrow (Z \otimes Y) \oslash X$$

(c) Optimization rules *FDiv*

FoldFMulConst:

$$(X \otimes c_0) \otimes c \rightarrow X \otimes (c_0 \otimes c)$$
$$(c_0 \oslash X) \otimes c \rightarrow (c_0 \otimes c) \oslash X$$
$$(X \oslash c_1) \otimes c \rightarrow \text{isNormal}(c \oslash c_1)\ ?\ X \otimes (c \oslash c_1)\ :\ X \oslash (c_1 \oslash c)$$

MulDivideConst:

$$(\text{MDC} \frac{\oplus}{\ominus} c_1) \otimes c \rightarrow (\text{MDC} \otimes c) \frac{\oplus}{\ominus} (c_1 \otimes c)$$

Log2OfHalf:

$$X \otimes log(0.5 \otimes Y) \rightarrow X \otimes log(Y) \ominus X$$

PowerX:

$$(X \otimes Y) \otimes X \rightarrow (X \otimes X) \otimes Y$$

(d) Optimization rules *FMul*. MDC denotes an expression of the form $X \otimes c$, $c \oslash X$, or $X \oslash c$ (c constant)

Fig. 2. Unsafe optimizations of InstCombine. Capitals X, Y, Z denote floating-point variables; lowercase letters a, b, c (possibly with subscripts) are floating-point constants. A notation of the form $\frac{\oplus}{\ominus}$ indicates that the rule comes in two instances: one for $\oplus$ and one for $\ominus$

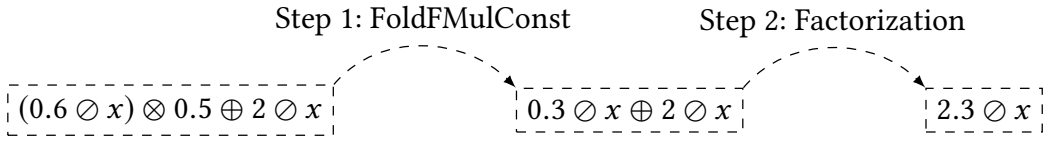

Fig. 3. Optimizing the expression used in Ex. 3. The source expression and the final optimization are not equivalent in floating-point (e.g. they return different results for $x = 4.025$).

threshold against which to compare the error differences, and how to rectify the optimization if the comparison indicates the stability may be disturbed (Steps 3 and 4). Our implementation and experiments are described in Sec. 8 and 9, resp.

Throughout the paper we use the two simple functions in Fig. 5 to illustrate the progress of our approach. Function h (right) represents the result of the InstCombine pass on function g (left), except that the constants are not folded, to keep the process traceable for the reader. For now, the only difference between the two functions is the expression to compute v1 (shown in red).

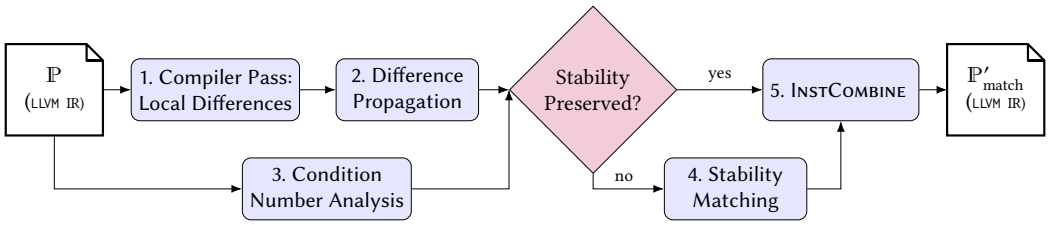

Fig. 4. Roadmap: Stability-preserving unsafe compiler optimizations

```
double g(double x, double y) {
  double v1 = (0.6 / x) * 0.5 + 2 / x;
  double v2 = v1 * y;
  double v3 = v2 + y;
  return v3; }
```

```
double h(double x, double y) {
  double v1 = (0.6 * 0.5 + 2) / x;
  double v2 = v1 * y;
  double v3 = v2 + y;
  return v3; }
```

Fig. 5. A demo function (left); after InstCombine expression rearrangement (right)

5 SENSITIVITY DIFFERENCE ANALYSIS

In this section we define a notion of sensitivity difference that estimates rounding error differences incurred by unsafe optimizations. Consider real-equivalent computational processes τ and τ' modeling two floating-point programs $\mathbb{P}$ and $\mathbb{P}'$ over a a fixed evaluation model and a common set of inputs I (such as the unoptimized and optimized versions of the same code). The rounding error difference between τ and τ' is defined by $\Delta^{re}_{\tau,\tau'}(I) = |\mathrm{re}^{\tau}_{abs}(I) - \mathrm{re}^{\tau'}_{abs}(I)|$ and is a highly complex function. To simplify it, we use a process similar to the simplification of the rounding error function $\mathrm{re}^{\tau}_{abs}(I)$ shown in Sec. 3.3, as follows.

We start from the Taylor expansion formula for $\mathrm{re}^{\tau}_{abs}(I)$ (Eq. (7)) and drop the higher-order terms. Note that some tools, including FPTaylor [35], do not perform this simplification, instead maintain an overapproximation of $\mathrm{re}^{\tau}_{abs}(I)$, and can therefore produce sound rounding error estimates. In contrast, the goal of our analysis is to compute *differences*. If we replace $\mathrm{re}^{\tau}_{abs}(I)$ and $\mathrm{re}^{\tau'}_{abs}(I)$ in $|\mathrm{re}^{\tau}_{abs}(I) - \mathrm{re}^{\tau'}_{abs}(I)|$ by overapproximations, the result of course does not overapproximate the difference. Retaining higher-order terms would therefore not preserve soundness in our approach, which is why we drop them. To accommodate this uncertainty, part of our experiments demonstrate that our difference analysis is consistent with actual rounding error differences.

The second step is similar to the smoothing step in Sec. 3.3, which disentangles the dependence of V_I on I and maximizes the D variables. In our context, however, we can slightly improve this step: we do not need to maximize all rounding error variables. Instead, we partition D into $D = D_d \cup D_c$ according to whether the variables are *input-**d**ependent* (D_d) or not (D_c, ***c**onstant*, input-independent). Constant rounding errors are accrued when performing constant folding, such as with v_1' in (9) below: value δ_1' equals $(0.6 \otimes 0.5)/(0.6 \cdot 0.5) - 1$ and hence does not vary as we maximize over inputs I.

$$\tau: \quad \begin{array}{lcl} v_1 & := & 0.6/x \cdot (1+\delta_1) \\ v_2 & := & v_1 \cdot 0.5 \cdot (1+\delta_2) \end{array} \qquad \tau': \quad \begin{array}{lcl} v_1' & := & 0.6 \cdot 0.5 \cdot (1+\delta_1') \\ v_2' & := & v_1'/x \cdot (1+\delta_2') \end{array} \tag{9}$$

In contrast, input-dependent rounding errors, such as δ_1, δ_2 and δ_2' above, are part of the parameter space over which we maximize the rounding error differences. The classification into input-[in-]dependent error variables is done statically and results in a more precise approximation of

rounding error differences than the original sensitivity definition in [24]: for many constant-folding rules in Fig. 2, the sensitivities of the left- and right-side processes are equal under that definition (an example are τ and τ' in Eq. (9)). Our input dependence analysis enables us to accurately model the error differences for constant-folding rules. Following this improvement, the sensitivity formula becomes:

Definition 4. *The **rounding error sensitivity** of τ is the function $S_\tau : \mathbb{R}^m \to \mathbb{R}$ defined by*

$$S_\tau(I) = \left| \sum_{\delta \in D_c} \frac{\partial \tau}{\partial \delta}(I, V) \cdot \delta \right| + \sum_{\delta \in D_d} \left| \frac{\partial \tau}{\partial \delta}(I, V) \right| \cdot u \ . \tag{10}$$

The first summation term represents (the first-order contributions of) input-independent rounding errors. In contrast, $\sum_{\delta \in D_d} |\partial \tau(I, V)/\partial \delta| \cdot u$ represents input-dependent rounding errors. We can now model the rounding error difference between τ and τ' with its error sensitivity difference:

Definition 5. *The **sensitivity difference** between two real-equivalent computational processes τ, τ' is the function $\Delta^S_{\tau,\tau'} : \mathbb{R}^m \to \mathbb{R}$ defined by*

$$\Delta^S_{\tau,\tau'}(I) = |S_\tau(I) - S_{\tau'}(I)| \ . \tag{11}$$

The case split (non-negative/negative) inherent in working with absolute values as in (11) can often be avoided when τ and τ' satisfy the following ordering property (as we will see, this is often the case when τ and τ' form the two sides of an InstCombine optimization rule):

Definition 6. *Given τ and τ' as in Def. 5, $\leq$ denotes a relation defined as follows:*

$$\tau \leq \tau' \ = \ \forall I \ \ S_\tau(I) \leq S_{\tau'}(I) \ . \tag{12}$$

Relation $\leq$ is a preorder, but not a partial order. Therefore, $\tau \leq \tau' \wedge \tau' \leq \tau$ defines a (non-trivial) equivalence relation; we say τ and τ' are *sensitivity-equivalent* in this case, denoted $\tau \equiv_S \tau'$. Sensitivity difference cannot distinguish between such processes. We write $\tau < \tau'$ to denote $\tau \leq \tau' \wedge \tau \not\equiv_S \tau'$. We also use the inverse relations $\geq$ and $>$ in the obvious way, and we denote imcomparability by $\tau \parallel \tau'$.

6 SENSITIVITY ANALYSIS FOR INSTCOMBINE

In this section we illustrate how sensitivity differences accrued during InstCombine optimizations can be effectively and efficiently measured. We split this process into two stages. We first describe how to collect differences introduced by the application of individual optimization rules (Sec. 6.1). We then explain how such differences are propagated from the point of initial accrual to the program output (Sec. 6.2).

6.1 Expression-Level Sensitivity Analysis

In this section we introduce our new LLVM pass ICDiff. The pass augments existing InstCombine rules such that they (tightly) approximate, via sensitivity differences, the rounding error difference accrued in each application of the rewriting rules in Fig. 2. To this end, we view each side of each rule as a computational process. For example, the left and right computational processes τ and τ' for rule $(c_0 \oslash X) \otimes c \to (c_0 \otimes c) \oslash X$ applied to expression $(0.6 \oslash x) \otimes 0.5$ are shown in Eq. (9) earlier.

In the following we employ the concept of *perturbation factors* to capture the sensitivity difference between τ and τ'. This concept achieves two purposes: (i) it renders the sensitivity difference (11) efficiently computable, and (ii) it makes it *compositional*, in the sense that the combined effect of multiple rules applied to one expression can be obtained easily from the net effects caused by

each individual rule. This is essential, since—as seen in Fig. 3—the InstCombine pass often applies several rules to rewrite the same expression.

Applying the preorder definition (12) to the rules $\tau \longrightarrow \tau'$ in Fig. 2, we find that the optimized form τ' typically features less rounding error sensitivity. This is due to constant folding after rewriting: the rounding error of the constant part of the expression is input-independent, so we do not overapproximate it by the unit roundoff (see Eq. (10)). In the unoptimized form, it may be the case that no single operation involves *only* constants, so its rounding errors are input-dependent, resulting in higher sensitivity. Precisely: (i) *none* of the rules satisfy $\tau < \tau'$; (ii) most of them satisfy $\tau \geq \tau'$; (iii) sometimes, however, τ and τ' are incomparable: $\tau \parallel \tau'$.

We will compute the sensitivity difference as the sensitivity value $S_{\dot{P}}(I)$ for a suitable real-valued function $\dot{P}$ called a *perturbation factor*. What "suitable" means depends on the rule $\tau \longrightarrow \tau'$: in the comparable case (ii) we determine $\dot{P}$ such that $S_{\dot{P}}(I)$ equals the sensitivity difference $\Delta^S_{\tau,\tau'}(I)$. In the incomparable case (iii) we determine $\dot{P}$ such that $S_{\dot{P}}(I)$ overapproximates the difference:

$$\textbf{comparable:} \quad \tau \geq \tau' \quad \Rightarrow \quad \forall I \;\; S_{\dot{P}}(I) \;=\; \Delta^S_{\tau,\tau'}(I) \tag{13}$$

$$\textbf{incomparable:} \quad \tau \parallel \tau' \quad \Rightarrow \quad \forall I \;\; S_{\dot{P}}(I) \;\geq\; \Delta^S_{\tau,\tau'}(I)\,. \tag{14}$$

In the following we determine, rule by rule, which of the two cases applies, and a perturbation factor $\dot{P}$ with the corresponding property (13) or (14). Along with these per-rule definitions, we explain how our tool processes these rules during the ICDiff rewriting pass.

6.1.1 Union Operations for Atomic Optimizations. Consider a rule of the form $\tau \longrightarrow \tau'$. The InstCombine pass uses pattern matching against τ to check rule applicability and, if applicable, replaces the instance of the τ subexpression by an appropriate instantiation of τ'. Our tool uses the same patterns, but changes the replacement action. Since our analysis needs to estimate the sensitivity difference between τ and τ', the replacement needs to be an expression involving both τ and τ'. We denote this expression by $\tau \sqcup \tau'$, pronounced *union*.

The replacement expression for all rules has the form $\tau \sqcup \tau' = \tau' + \dot{P}$. This form reflects the fact that the terms produced by ICDiff are supposed to track both the original optimization (the τ' part), as well as the sensitivity difference (the $\dot{P}$ part). In this section, for each InstCombine rule we define $\dot{P}$ and show the complete modified rule $\tau \longrightarrow \tau \sqcup \tau'$ used by ICDiff.

Fig. 6 and 7 shows $\dot{P}$ and the union definitions for some of the rewriting rules in Fig. 2. To keep the presentation succinct, we show the union operations for all unsafe InstCombine rules in Fig. 9 in the Appendix. Again purely for succinctness, in Fig. 6, 7 and 9 we write the left and right sides of each rule unambiguously using fully parenthesized expressions, instead of computational processes. The expressions mix floating-point with real-arithmetic operations. It is straightforward to translate such expressions to computational processes. For example:

Example 7. *The computational process for expression $X \otimes (c_0 \otimes c_1) + \dot{P}$ (first rule of Fig. 6) is*

$$\begin{aligned} v_1 &:= (c_0 \cdot c_1) \cdot (1 + \delta_1); \\ v_2 &:= (X \cdot v_1) \cdot (1 + \delta_2); \\ r &:= v_2 + \dot{P}; \end{aligned}$$

To handle the rules of *FAdd* in Fig. 2 (b), we first show how to define union operations for atomic associative, commutative, and distributive rewriting rules. Complex rules like *SimplifyFAdd* are obtained as compositions of atomic rules. In Sec. 6.1.2, we show that our union definitions are indeed composable. In the rest of Sec. 6.1.1 we distinguish between "comparable rules" (where the two sides of the rule satisfy $\tau \geq \tau'$), and "incomparable rules" ($\tau \parallel \tau'$); recall that the case $\tau < \tau'$ does not occur.

Simplify Assoc/Commut :

$$(X \otimes c_0) \otimes c_1 \sqcup X \otimes (c_0 \otimes c_1) := X \otimes (c_0 \otimes c_1) + \dot{P}$$
$$c_0 \otimes (c_1 \otimes X) \sqcup (c_0 \otimes c_1) \otimes X := (c_0 \otimes c_1) \otimes X + \dot{P}$$
$$(c_0 \otimes X) \otimes c_1 \sqcup (c_1 \otimes c_0) \otimes X := (c_1 \otimes c_0) \otimes X + \dot{P}$$
$$c_0 \otimes (X \otimes c_1) \sqcup X \otimes (c_1 \otimes c_0) := X \otimes (c_1 \otimes c_0) + \dot{P}$$
$$(X \otimes c_0) \otimes (Y \otimes c_1) \sqcup (X \otimes Y) \otimes (c_0 \otimes c_1) := (X \otimes Y) \otimes (c_0 \otimes c_1) + \dot{Q}$$
$$\dot{P} := c_0 c_1 X \cdot (1 - \epsilon/u) \cdot \delta, \quad \dot{Q} := c_0 c_1 XY \cdot (1 - \epsilon/u) \cdot \delta, \quad \epsilon := rel(c_0 c)$$

Fig. 6. Union operations for some unsafe optimizations of InstCombine: *FMul*. Function $rel(x)$ returns the relative error of x.

Union definitions for comparable rules. The comparable rules include *Simplify Assoc/Commut for FMul*, *FoldFMulConst*, and *FoldFDivConst*, in which the rewriting rules fold constants during optimization. For these rules we have $\tau \geq \tau'$. It is easy to see that the sensitivity difference between τ and τ' is due to input-independent rounding error in τ' (Sec. 5). Since the sensitivity difference function (11) is simple in this case, we directly compute the perturbation factor $\dot{P}$ using this equation. The following theorem states, for the multiplicative constant folding rule (top rule of Fig. 6), the equivalence of the left-hand side process τ and the union $\tau \sqcup \tau'$, and why this implies Eq. (13).

Theorem 8. *Let* $\tau = (X \otimes c_0) \otimes c_1$, $\tau' = X \otimes (c_0 \otimes c_1)$, *hence* $\tau \sqcup \tau' = \tau' + \dot{P} = X \otimes (c_0 \otimes c_1) + \dot{P}$ *with* $\dot{P} = (X c_0 c_1) \cdot (1 - rel(c_0 c_1)/u) \cdot \delta$, *as in Fig. 6. Then, for all* I, $S_{\dot{P}}(I) = \Delta^S_{\tau,\tau'}(I)$.

Proof. The computational processes for τ and $\tau \sqcup \tau'$ are:

$$\tau: \quad \begin{array}{lcl} v_1 & := & X \cdot c_0 \cdot (1 + \delta_1) \\ v_2 & := & v_1 \cdot c_1 \cdot (1 + \delta_2) \end{array}$$

$$\tau \sqcup \tau': \quad \begin{array}{lcl} v_1' & := & c_0 \cdot c_1 \cdot (1 + rel(c_0 c_1)) \\ v_2' & := & X \cdot v_1' \cdot (1 + \delta_2') + \dot{P} \end{array}$$

We first show that τ and $\tau \sqcup \tau'$ are sensitivity-equivalent: $\tau \equiv_S \tau \sqcup \tau'$. Using Eq. (10), for any X:

$$\begin{aligned} S_{\tau \sqcup \tau'}(X) &= \left| \sum_{\delta \in D_c} \frac{\partial v_2'}{\partial \delta}(X) \cdot \delta \right| + \sum_{\delta \in D_d} \left| \frac{\partial v_2'}{\partial \delta}(X) \right| \cdot u \\ &= |X c_0 c_1 \cdot rel(c_0 c_1)| + |X c_0 c_1| \cdot u \\ &\quad + |X c_0 c_1 \cdot (1 - rel(c_0 c_1)/u)| \cdot u \\ &= 2\,|X c_0 c_1| \cdot u \\ &= S_\tau(X)\,. \end{aligned}$$

From this result, using the linearity of Eq. (10), we obtain, for any input X:

$$S_{\dot{P}}(X) = |S_{\tau \sqcup \tau'}(X) - S_{\tau'}(X)| = |S_\tau(X) - S_{\tau'}(X)| = \Delta^S_{\tau,\tau'}\,.$$

□

Union definitions for incomparable rules. The remaining rules are mainly used by InstCombine to collect like terms in the expression, such as the Assoc rule for FAdd in Fig. 2 (a). For these rules, the left and right computational processes are incomparable. The goal here is to find an easy-to-maximize perturbation factor $\dot{P}$ whose sensitivity overapproximates the sensitivity difference between τ and τ'. Thm. 9 shows the union definition for the Assoc/Commut rule of *FAdd*.

Const-Assoc/Commut :

$$(c_1 \oplus X) \oplus c_2 \sqcup (c_1 \oplus c_2) \oplus X := (c_1 \oplus c_2) \oplus X + \dot{P}$$
$$(X \oplus c_1) \oplus c_2 \sqcup X \oplus (c_1 \oplus c_2) := X \oplus (c_1 \oplus c_2) + \dot{P}$$
$$\dot{P} := (X - c_2) \cdot \delta_1 + (c_1 + c_2) \cdot (1 - \epsilon/u) \cdot \delta_2,$$
$$\epsilon := rel(c_1 + c_2)$$

Assoc/Commut :

$$(X \oplus Y) \oplus Z \sqcup (X \oplus Z) \oplus Y := (X \oplus Z) \oplus Y + \dot{P},$$
$$(X \oplus Y) \oplus Z \sqcup X \oplus (Y \oplus Z) := X \oplus (Y \oplus Z) + \dot{Q},$$
$$\dot{P} := (Y - Z) \cdot \delta, \dot{Q} := (X - Z) \cdot \delta$$

Fig. 7. Union operations for some unsafe optimizations of InstCombine: *FAdd*. Function $rel(x)$ returns the relative error of x.

Theorem 9. *Let $\tau = (X \oplus Y) \oplus Z$ and $\tau' = (X \oplus Z) \oplus Y$. The perturbation factor $\dot{P} = (Y - Z) \cdot \delta$, used in $\tau \sqcup \tau' := (X \oplus Z) \oplus Y + \dot{P}$, satisfies Eq. (14): $\forall I\ S_{\dot{P}}(I) \geq \Delta^S_{\tau,\tau'}(I)$.*

Proof. The computational processes for τ and τ' are:

$$\tau: \begin{array}{lcl} v_1 & := & (X + Y) \cdot (1 + \delta_1) \\ v_2 & := & (v_1 + Z) \cdot (1 + \delta_2) \end{array}$$
$$\tau': \begin{array}{lcl} v'_1 & := & (X + Z) \cdot (1 + \delta'_1) \\ v'_2 & := & (v'_1 + Y) \cdot (1 + \delta'_2) \end{array}$$

Based on Eq. (10), the sensitivity of τ and τ' for a concrete input $I := (X, Y, Z)$ is:

$$\begin{array}{rcl} S_\tau(I) & = & \left|\frac{\partial v_2}{\partial \delta_1}(X,Y,Z)\right| \cdot u + \left|\frac{\partial v_2}{\partial \delta_2}(X,Y,Z)\right| \cdot u \\ & = & (|X + Y| + |X + Y + Z|) \cdot u \\ S_{\tau'}(I) & = & \left|\frac{\partial v'_2}{\partial \delta'_1}(X,Y,Z)\right| \cdot u + \left|\frac{\partial v'_2}{\partial \delta'_2}(X,Y,Z)\right| \cdot u \\ & = & (|X + Z| + |X + Y + Z|) \cdot u \end{array}$$

With the triangle inequality, it is easy to verify that

$$\Delta^S_{\tau,\tau'}(I) = \big|\ |X + Z| - |X + Y|\ \big| \cdot u \leq |Y - Z| \cdot u = S_{\dot{P}}(I)\ .$$

Notice that the overapproximation introduced by $\dot{P}$ is tight in the sense that when τ and τ' are the same expression ($Z = Y$), then $S_{\dot{P}}$ equals 0. □

6.1.2 Composition Union Definitions. We want our union definitions to be compositional, for two reasons: (i) complex rules like *SimplifyFAdd* can then be handled by splitting them up into atomic rule constituents; (ii) for complex expressions, the InstCombine pass repeatedly performs pattern matching to apply rewriting rules, such as in Ex. 3. Compositionality here means that the combined sensitivity difference of multiple rules can be calculated from the sensitivity difference in each individual rule. Specifically, the composition union should have the same properties as the individual one, namely: (a) the pattern matching process is still the same as in the original rules, and (b) the sensitivity of $\dot{P}$ in the composition union should overapproximate the actual sensitivity difference.

According to the original rewriting rules in InstCombine, we only need to define composition union for floating-point addition and multiplication:

Definition 10. *Given two unions $\tau \sqcup \tau' := \tau' + \dot{P}$ and $v \sqcup v' := v' + \dot{Q}$, the* ***composition unions*** *for floating-point addition and multiplication are:*

$$(\tau \oplus v) \sqcup (\tau' \oplus v') := (\tau' \oplus v') + (\dot{P} + \dot{Q}) \tag{15}$$

$$(\tau \otimes v) \sqcup (\tau' \otimes v') := (\tau' \otimes v') + (v_0 \cdot \dot{P} + \tau_0 \cdot \dot{Q}) \tag{16}$$

(Recall that τ_0 and v_0 denote the rounding-error free versions of τ and v, respectively.) With these definitions, property (a) from above holds because we do modify the pattern matching process (only the replacement value). Property (b) from above follows from the following theorem.

Theorem 11. *The sensitivities of the perturbation factors in the composition unions (15) and (16) each overapproximate the respective sensitivity difference:*

$$\begin{aligned} \forall I\ S_{\dot{P}+\dot{Q}}(I) &\geq |S_{\tau' \oplus v'}(I) - S_{\tau \oplus v}(I)| \\ \forall I\ S_{v_0 \cdot \dot{P} + \tau_0 \cdot \dot{Q}}(I) &\geq |S_{\tau' \otimes v'}(I) - S_{\tau \otimes v}(I)| \ . \end{aligned}$$

Proof. We only show the correctness of $S_{\dot{P}+\dot{Q}}$. The case of $S_{v_0 \cdot \dot{P} + \tau_0 \cdot \dot{Q}}$ can be proved similarly.

Since $\tau \oplus v = (\tau + v) \cdot (1 + \delta) = \tau + v + (\tau + v) \cdot \delta$, based on Eq. (10), the sensitivity of $\tau \oplus v$ for arbitrary input I is:

$$S_{\tau \oplus v}(I) = S_\tau(I) + S_v(I) + (\tau_0(I) + v_0(I)) \cdot u$$

Similarly, for $\tau' \oplus v'$ we have:

$$S_{\tau' \oplus v'}(I) = S_{\tau'}(I) + S_{v'}(I) + (\tau_0(I) + v_0(I)) \cdot u$$

With the triangle inequality, we have

$$\begin{aligned} |S_{\tau' \oplus v'}(I) - S_{\tau \oplus v}(I)| &= |S_{\tau'}(I) - S_\tau(I) + S_{v'}(I) - S_v(I)| \\ &\leq S_{\dot{P}}(I) + S_{\dot{Q}}(I) \\ &= S_{\dot{P}+\dot{Q}}(I) \ . \end{aligned}$$

□

6.2 Program-Level Sensitivity Analysis

The given program $\mathbb{P}$ will generally contain many numeric operations outside the scope of InstCombine, such as trigonometric functions. While the optimizations do not alter such operations, the sensitivity differences accrued in inputs to them may nonetheless be affected and must be propagated.

The definition of *sensitivity difference* extends immediately from computational processes to programs, since our (for now straight-line) programs are instances of the former. Our goal in this section is therefore to compute, reasonably precisely, the quantity $\Delta^S_{\mathbb{P},\mathbb{P}'}(I)$ (note $\mathbb{P}, \mathbb{P}'$ in the subscript). The following theorem accomplishes the crucial step. Let $\mathbb{P} \sqcup \mathbb{P}'$ denote the program obtained after running our ICDiff pass over $\mathbb{P}$ (Step 1 in Fig. 4). Then, $\Delta^S_{\mathbb{P},\mathbb{P}'}(I)$ can be approximated from above by the sensitivity difference $\Delta^S_{\mathbb{P}',\mathbb{P}\sqcup\mathbb{P}'}(I)$ between $\mathbb{P}'$ and $\mathbb{P} \sqcup \mathbb{P}'$, i.e. by the contribution of the perturbation factors inserted by our union rules to $\mathbb{P}'$.

Theorem 12. *The sensitivity difference $\Delta^S_{\mathbb{P},\mathbb{P}'}(I)$ between $\mathbb{P}$ and $\mathbb{P}'$ satisfies*

$$\Delta^S_{\mathbb{P},\mathbb{P}'}(I) \leq \Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I) \ .$$

We prove this relationship using the following equality:

Theorem 13. *Let r be the result computed by program $\mathbb{P} \sqcup \mathbb{P}'$, and $D_{\dot{P}}$ denote the set of all rounding error variables in the perturbation factors in $\mathbb{P} \sqcup \mathbb{P}'$. Then*

$$\Delta^{S}_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I) = \sum_{\delta \in D_{\dot{P}}} \left| \frac{\partial r}{\partial \delta}(I) \right| \cdot u \ . \tag{17}$$

Proof. According to our sensitivity definition (Eq. (10)), for a given input I, we have:

$$S_{\mathbb{P}\sqcup\mathbb{P}'}(I) = \left| \sum_{\delta \in D_c} \frac{\partial r}{\partial \delta}(I) \cdot \delta \right| + \sum_{\delta \in D_d} \left| \frac{\partial r}{\partial \delta}(I) \right| \cdot u + \sum_{\delta \in D_{\dot{P}}} \left| \frac{\partial r}{\partial \delta}(I) \right| \cdot u \tag{18}$$

$$= S_{\mathbb{P}'}(I) + \sum_{\delta \in D_{\dot{P}}} \left| \frac{\partial r}{\partial \delta}(I) \right| \cdot u \tag{19}$$

Eq. (18) is due to the fact that all the perturbation factors in our union definitions (both *atomic* and *composition*) have the form $\sum_i E_i \cdot \delta_i$, where E_i is some real expression; and we ignore higher-order error terms in our computation of sensitivity (10). Eq. (19) is true because the InstCombine-optimized program $\mathbb{P}'$ and the program $\mathbb{P} \sqcup \mathbb{P}'$ optimized with our union rules are the same except for the perturbation factors. As a result, we have:

$$\Delta^{S}_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I) = S_{\mathbb{P}\sqcup\mathbb{P}'}(I) - S_{\mathbb{P}'}(I) = \sum_{\delta \in D_{\dot{P}}} \left| \frac{\partial r}{\partial \delta}(I) \right| \cdot u \ .$$

□

With Thm. 13, we can complete the results in this section:

Proof. Let L be the set of locations in $\mathbb{P}$ modified by InstCombine. For $l \in L$, let v_l denote the variable assigned in l, and τ_l and τ'_l the computational processes before and after rewriting in l. For a given input I,

$$\begin{array}{lll}
\Delta^{S}_{\mathbb{P},\mathbb{P}'}(I) & = & |S_{\mathbb{P}}(I) - S_{\mathbb{P}'}(I)| \qquad \{ \text{ def. } \Delta^{S}_{\mathbb{P},\mathbb{P}'} \ \} \\
& = & \sum_{l \in L} \left| \frac{\partial r}{\partial v_l}(I) \right| \cdot \left| S_{\tau_l}(I) - S_{\tau'_l}(I) \right| \qquad \{ \ (10) \ \} \\
& \leq & \sum_{l \in L} \left| \frac{\partial r}{\partial v_l}(I) \right| \cdot S_{\dot{P}_l}(I) \qquad \{ \text{ soundness of } S_{\dot{P}_l} \ \} \\
& = & \Delta^{S}_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I) \qquad \{ \ (10) + \text{Thm. } 13 \ \}
\end{array}$$

□

The relatively simple form of the summation term for $\Delta^{S}_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}$ (Thm. 13) enables us to efficiently find its maximum value across the input constraints. In Sec. 7 we use this maximum as our estimate of the sensitivity difference between programs $\mathbb{P}$ and $\mathbb{P}'$.

Example 14. *Fig. 8 (left) shows the result of applying our ICDiff pass to the demo function in Fig. 5, namely the perturbation factors inserted by the union rules. Note that the computations in the perturbation factors are in real. The first factor δ_1 is due to the union rule of* FoldFMulConst*; the remaining two come from the union rule of* Const-Factorization*. On the right is the final sensitivity difference formula.*

```
double g(double x, double y) {
  double v1 = 2.3/x + 0.3/x · δ1 + 0.6/x · δ2 + 2.3/x · δ3;
  double v2 = v1 * y;
  double v3 = v2 + y;
  return v3; }
```

Sensitivity difference estimate:

$$\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'} = \sum_{\delta\in D_{\dot{P}}} \left|\frac{\partial v_3}{\partial \delta}\right| \cdot u = |y| \cdot \left(\left|\frac{0.3}{x}\right| + \left|\frac{0.6}{x}\right| + \left|\frac{2.3}{x}\right|\right) \cdot u$$

Fig. 8. Perturbation factors (in green) inserted by our union rules (left); sensitivity difference estimate (right)

7 STABILITY-PRESERVING FLOATING-POINT OPTIMIZATIONS

The goal of this paper is to determine whether, for each input I, the programs $\mathbb{P}$ and $\mathbb{P}'$ before and after applying InstCombine are either both numerically stable, or both are not. The intuition is as follows. Recall the notion of stability coefficient (Eq. (3)), and consider the stability *difference* coefficient

$$\Delta^c_{\mathbb{P},\mathbb{P}'} = \min\{c \in \mathbb{Z}^+ : \forall I : \ \big|\, |\mathbb{P}(I) - \mathbb{P}_0(I)| \ - \ |\mathbb{P}'(I) - \mathbb{P}_0(I)| \,\big| \leq c \cdot u \cdot \kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)| \ \} \ . \tag{20}$$

The relationship between these coefficients is that the stability coefficients $c_{\mathbb{P}}$ and $c_{\mathbb{P}'}$ of $\mathbb{P}$ and $\mathbb{P}'$ are within $\Delta^c_{\mathbb{P},\mathbb{P}'}$ of each other: let X abbreviate $u \cdot \kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)|$, $\Delta_{\mathbb{P}}$ abbreviate $|\mathbb{P}(I) - \mathbb{P}_0(I)|$, and $\Delta_{\mathbb{P}'}$ abbreviate $|\mathbb{P}'(I) - \mathbb{P}_0(I)|$. Then, for every I,

$$\begin{aligned} \Delta_{\mathbb{P}'} = \Delta_{\mathbb{P}} + \Delta_{\mathbb{P}'} - \Delta_{\mathbb{P}} &\leq \Delta_{\mathbb{P}} + |\Delta_{\mathbb{P}'} - \Delta_{\mathbb{P}}| \\ &\leq c_{\mathbb{P}} \cdot X + \Delta^c_{\mathbb{P},\mathbb{P}'} \cdot X = (c_{\mathbb{P}} + \Delta^c_{\mathbb{P},\mathbb{P}'}) \cdot X \ , \end{aligned}$$

so $c_{\mathbb{P}'} \leq c_{\mathbb{P}} + \Delta^c_{\mathbb{P},\mathbb{P}'}$, symmetrically $c_{\mathbb{P}} \leq c_{\mathbb{P}'} + \Delta^c_{\mathbb{P},\mathbb{P}'}$, and thus $|c_{\mathbb{P}'} - c_{\mathbb{P}}| \leq \Delta^c_{\mathbb{P},\mathbb{P}'}$. Therefore, if $\Delta^c_{\mathbb{P},\mathbb{P}'}$ is small, $c_{\mathbb{P}'}$ and $c_{\mathbb{P}}$ are close, suggesting that $\mathbb{P}$ and $\mathbb{P}'$ are stability-equivalent.

To formalize this idea, we turn to the sensitivity difference $\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}$ developed earlier, which allows us to efficiently estimate pointwise rounding error differences. We define stability equivalence as follows. Let $\mathbb{P} \sqcup \mathbb{P}'$ the program optimized with our union rules $\tau \longrightarrow \tau' + \dot{P}$ from Sec. 6.1.

Definition 15. *Given some input constraints $\mathbb{I}$, program $\mathbb{P}$ and the InstCombine-optimized program $\mathbb{P}'$ are* ***stability-equivalent*** *if, for all $I \in \mathbb{I}$,*

$$f(I) := \frac{\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I)}{\kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)|} \leq u \ . \tag{21}$$

Eq. (21) resembles the condition for the stability difference coefficient (20), except that we approximate $\big||\mathbb{P}(I) - \mathbb{P}_0(I)| - |\mathbb{P}'(I) - \mathbb{P}_0(I)|\big|$ by $\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I)$, and we conservatively set $\Delta^c_{\mathbb{P},\mathbb{P}'}$ to 1, the *strictest* possible coefficient, permitting the smallest stability changes. Increasing the right-hand side of $\leq$ in (21) amounts to permitting larger stability discrepancies.

Another interpretation of Eq. (21) is to relate it to the concept of *Wilkinson number*, developed by Miller [24] to compare different floating-point implementations for the same numeric problem. Consider the case of a single output variable:

Definition 16 ([24]). *The* ***Wilkinson number*** *of program $\mathbb{P}$ is the function $\omega_{\mathbb{P}} : \mathbb{R}^m \longrightarrow \mathbb{R}$ defined by*

$$\omega_{\mathbb{P}}(I) = \frac{S_{\mathbb{P}}(I)}{\kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)| \cdot u} \ . \tag{22}$$

The only difference between the Wilkinson number and the stability coefficient $c_{\mathbb{P}}$ (Eq. (3)) is that $\omega_{\mathbb{P}}$ uses the sensitivity $S_{\mathbb{P}}$ instead of the rounding error. Stability equivalence implies the closeness of the Wilkinson numbers of $\mathbb{P}$ and $\mathbb{P}'$: if $\mathbb{P}$ and $\mathbb{P}'$ are stability-equivalent, then

$$|\omega_{\mathbb{P}}(I) - \omega_{\mathbb{P}'}(I)| = \frac{|S_{\mathbb{P}}(I) - S_{\mathbb{P}'}(I)|}{\kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)| \cdot u} \leq \frac{\Delta^S_{\mathbb{P} \sqcup \mathbb{P}', \mathbb{P}'}(I)}{\kappa_{\mathbb{P}_0}(I) \cdot |\mathbb{P}_0(I)| \cdot u} \leq 1. \tag{23}$$

Stability equivalence is therefore *stronger* than Wilkinson equivalence.

Given input constraints in the form of an interval $\mathbb{I}$, Eq. (21) can be efficiently checked using various techniques with different strengths, such as interval analysis and stochastic search. We discuss this in detail in our implementation section (8). If (21) cannot be proved across $\mathbb{I}$, the *Matching* Step 4 in Fig. 4 selectively disables optimizations in some expressions of the program, the goal being the preservation of stability while minimizing the efficiency impact of the "de-optimization".

Our strategy here is to disable the smallest number of optimizations, selected in order of decreasing contribution to the stability change, that is sufficient to preserve stability. This selection can be computed directly using Thm. 13, without re-running the analysis: The terms in the sum in Eq. (17) can be partitioned into groups of terms whose variables come from the same perturbation factor. The terms in each such group add up to the contribution of the one expression whose optimization is modeled by the perturbation factor.

In the worst case, disabling *all* optimizations results in a trivially stability-equivalent program. In our experiments, this was required only in the *invplaneN* case (Table 4).

Example 17. *Continuing Ex. 14, given $x \in [4, 5]$ and $y \in [1, 2]$, an upper bound of $f(I)$, estimated using interval analysis, is $0.35u$. Since $0.35u \leq u$, the aggressive fast-math optimizations preserve numeric stability over the input interval; the program can be returned to the user fully optimized.*

8 IMPLEMENTATION

We have implemented our approach in the form of three LLVM optimization passes: ICDiff, *Analysis*, and *Matching*. They have been integrated into the LLVM 5.0 optimization pass chain. To use our tool, the user invokes the `clang` front-end to LLVM, passing to it the `-ffast` flag, in order to obtain an intermediate representation file (`.bc`) of the analyzed program. Our tool is then invoked directly as the LLVM optimizer `opt`, which now fully automatically produces stability-equivalent code. There is no further user interaction, except for some diagnostic output.

ICDiff. This pass equals the original InstCombine, except that the optimization rules have been replaced by those based on the union expressions annotated with perturbation factors (Fig. 9). The pattern matching process that determines rule applicability is unchanged, since the union expressions do not modify the left side (τ) of each rule.

Most perturbation factors $\dot{P}$ in our union rules require the computation of the relative rounding error $rel\colon \mathbb{R} \to \mathbb{R}$ of a real number, which relies on infinite-precision arithmetic. We rewrite $\dot{P}$ in terms of the *absolute* rounding error, which can be computed directly in floating-point. For example, the perturbation factor $\dot{P} = (Xc_0c_1) \cdot (1 - rel(c_0c_1)/u) \cdot \delta$ in the union rule $\tau \sqcup \tau' = X \otimes (c_0 \otimes c_1) + \dot{P}$ (Thm. 8) equals $X \cdot (c_0c_1 - c_0c_1 rel(c_0c_1)/u) \cdot \delta$; here, $c_0c_1 rel(c_0c_1)$ is the absolute error of $c_0 \otimes c_1$. Under the normal-range assumption, $c_0c_1 rel(c_0c_1)$ can be computed as

$$c_0c_1 rel(c_0c_1) = fma(c_0, c_1, -(c_0 \otimes c_1)) \,,$$

which is purely based on floating-point arithmetic [27]. Here, *fma* denotes the fused multiply-add operation. Similar laws exist for extracting the rounding errors in addition $rel(c_0 + c_1)$ and division $rel(c_0/c_1)$ (see [27] for more details). When applying our rewriting rules, for each $\delta \in D_{\dot{P}}$ we also

record the expression affected by δ. This information is used to determine which expressions to freeze.

Analysis. This pass corresponds to Steps 2 and 3 in Fig. 4 and computes the expression in Eq. (21) across the input constraints $\mathbb{I}$ to decide stability-equivalence. We assume that the input constraints are given as floating-point intervals. Our computation of (21) is based on the *logarithmic differential* [3]. Since this differential satisfies the *chain rule*, we can compute it via *automatic differentiation* (AD) [13], which is a numerical method to accurately evaluate the derivative of a math function specified by a program based on the chain rule. We have implemented the forward mode of AD for LLVM IR code. To check the validity of Eq. (21) across the input intervals $\mathbb{I}$, we provide two approaches: *interval analysis* and *stochastic search.*

Interval analysis computes the function $f(I) := \frac{\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I)}{\boldsymbol{\kappa}_{\mathbb{P}_0}(I)\cdot|\mathbb{P}_0(I)|}$ over the abstract domain of intervals. The result interval $[\downarrow f, \uparrow f]$ overapproximates the set of values of f across $\mathbb{I}$. Clearly, $\uparrow f \leq u$ ($\downarrow f > u$, resp.) implies that optimizations do (do not, resp.) preserve stability. In the case of $\downarrow f \leq u < \uparrow f$, we use a *branch-and-bound* algorithm [19], to split the input intervals until we can decide stability-equivalence of optimizations conclusively. We have implemented this approach on top of the *Boost Interval Library* [5]. Our implementation supports math functions commonly found in numeric programs, such as trigonometric, logarithmic, and exponential functions.

For the stochastic search module, we first transform Eq. (21) to the target function Eq. (24):

$$Q(I) := \sum_{\delta\in D_{\hat{P}}} \left|\frac{\partial r}{\partial \delta}(I)\right| - \max\left\{\sum_{I_i}\left|I_i\cdot\frac{\partial r}{\partial I_i}(I)\right|, |r|\right\} \quad \leq \quad 0 \tag{24}$$

where r is the computed result. The maximum of $Q(I)$ can be found by applying the *stochastic subgradient ascent* (see [37] for more details). The key step is computing the iterative search function (25), which informs us of a new input that may have a provide value of Q:

$$I := I + \eta\nabla Q(I)\,, \tag{25}$$

where η is the step size, $\nabla Q(I)$ computes the subgradient of Q with respect to I. Using Eq. (25) we can check whether the maximum value of $Q(I)$ is greater than zero or not, which implies that the optimizations do (do not, resp.) preserve stability.

Matching. If the optimizations do not preserve stability, we derive from the terms in $\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I)$ the individual contribution of each rewriting step to the final sensitivity difference. We now remove the terms in the summation of $\Delta^S_{\mathbb{P}\sqcup\mathbb{P}',\mathbb{P}'}(I)$, in decreasing order of their numeric value, until $f(I) \leq u$ holds. Each term corresponds to an expression we wish to freeze, i.e., to be ignored by the InstCombine optimizer. This is conveniently supported by LLVM, via instruction modifiers in the IR representation. After applying these modifiers, our tool runs the InstCombine pass on $\mathbb{P}$ to obtain a stability-equivalent program $\mathbb{P}'_{\text{match}}$. Our tool reports diagnostic information about the frozen expressions to the user.

Extensions: Control Flow; Special Floating-Point Data

Control flow. We treat branches and loops as common in symbolic execution: we apply path-wise analysis and unroll the loops. For example, for the *Odometer* benchmark, presented in Sec. 9, which has a loop with an adjustable iteration count, we unrolled the loop body up to a thousand times, resulting in a rather large straight-line program. If the loop iteration number is not a constant, the user can bound that number and apply stability analysis up to that number of iterations, after partial unwinding. The remaining iterations are kept in a loop in which, however, InstCombine is conservatively disabled entirely. This way, no unaccounted-for aggressive optimizations are

applied. Finally, we inline (non-recursive) function calls where possible; this is directly supported by LLVM.

Special floating-point data. Sub-normal numbers (very close but not equal to zero) are treated specially by IEEE-compliant floating-point hardware and therefore have different rounding semantics. This semantics can be modeled but requires a new set of union rules.

Special floating-point data ($\pm\infty$, NaN) lend themselves less to quantitative comparison, as inevitably necessary for any numeric difference analysis, than real-valued floating-point data. It is also less meaningful to define a notion of propagation. Therefore, handling such data is not just a matter of extending the implementation, but requires some theory of its own. One version of stability that accommodates such data, call it *stability invariance*, is to require that, for each intermediate program variable v, if the value of v in one of $\mathbb{P}$ or $\mathbb{P}'$ is special, then the two must be the same; in particular, the other must then be special, too. (Recall that, according to our notion of [numeric] stability equivalence, intermediate results of $\mathbb{P}$ and $\mathbb{P}'$ need not be stability-equivalent—what counts is the comparison of the final output values.)

The above notion of stability invariance can principally be implemented using tools like Ariadne [2], which detects floating-point *exceptions*. The paper executes the input program in real arithmetic and then checks for numeric conditions that trigger exceptions, indicating conditions like sub-normal numbers, infinities, and NaNs. A very rough sketch of such an implementation is as follows. If, using Ariadne, we can prove programs $\mathbb{P}$ and $\mathbb{P}'$ exception-free[3], we can continue with our current numeric stability analysis. Otherwise, we can check if the occurring exceptions indicate stability invariance violations. If there are no such violations, we apply our analysis restricted to the exception-free fragment of the program.

The above sketch does not even cover all problems that special values incur. For example, it is not clear how stability matching should be performed, i,e., how the invariance violation should be removed. Further, and perhaps more serious, even given stability invariance, there are other kinds of optimization effects: $-$NaN compared with anything returns *true* under `-ffast-math` compilation with `gcc`, *false* otherwise [29], with possibly far-reaching consequences for the program.

While these extensions are not currently implemented, our technique warns users of the potential presence of sub-normals or special data in the calculation. For instance, in the interval-analysis implementation, if some of the encountered intervals include 0, the calculation may involve sub-normals. For the stochastic-search implementation, which is based on executing the program on samples, we use exception triggering mechanisms to serve as warnings.

9 EMPIRICAL EVALUATION

The goal of our evaluation is to determine if (i) sensitivity differences are consistent with rounding error differences, (ii) an analysis based on maximum rounding errors would have come to the same conclusions, and if (iii) sensitivity information can be used to remove stability-corrupting optimizations at acceptable cost. All experiments are performed under Mac OS 10.12.6 with 2.9GHz Intel Core i7 and 16GB memory.

Benchmarks. Table 2 shows the programs along with their input constraints. *Turbine1*, *turbine3*, *pendulum*, *carbongas* are from FPBench [10], a benchmark library used in the floating-point research community (other programs in FPBench are unaffected by InstCombine). *Odometer* is a function from [9] to compute the position of a robot. *COSHestonEngine*, used for pricing options, is from the quantitative finance library *QuantLib* [36]. Finally, *precess*, *invarplane*, *triton*, and *ssats* are from a computational astronomy library [31].

[3]For $\mathbb{P}'$, we have to apply the optimizations at the source-code level, to be able to pass the resulting program to Ariadne.

Program	#LoC	#Rewritings
turbine1	3	2
turbine3	3	2
pendulum	3	1
carbongas	15	1
odometerX	22	9
odometerY	22	8
HestENG	52	6
precess	16	4
invplaneN	20	2
invplaneW	20	6
triton	332	9
ssats	556	25

Table 2. Benchmarks characteristics

0	1	2	3	4	5	6	7	8	9
Program	**Input interval**	$\downarrow f$ (u)	$\uparrow f$ (u)	**Stability Preserved?**	**CompTime** (sec)	$\max \Delta^{c}_{\mathbb{P},\mathbb{P}'}$	$\max \mathrm{re}^{\mathbb{P}}_{abs}$ (ulp)	$\max \mathrm{re}^{\mathbb{P}'}_{abs}$ (ulp)	$\max \Delta^{re}_{\mathbb{P},\mathbb{P}'}$ (ulp)
turbine1	$[-4,-1] \wedge [1.5,2] \wedge [4,8]$	–	0.34	✓	1.69	2	3.34	3.56	2.50
turbine3	$[-4,-1] \wedge [1.5,2] \wedge [4,8]$	–	0.40	✓	1.70	2	3.80	3.63	3.00
pendulum	$[1,2] \wedge [1,5]$	–	< 0.01	✓	6.46	1	1.63	1.63	0.86
carbongas	$[0.1,0.5]$	–	0.94	✓	0.01	2	2.18	1.98	2.00
precess	$[20,21] \wedge [1.5,2] \wedge [2,2.5]$	–	0.44	✓	52.96	1	1884.26	1884.26	1140.51
hestENG	$[16,17]$	–	< 0.01	✓	0.87	1	17.83	17.83	3.00
odometerX	$[0.1,0.2]$	5.23	–	X	3.01	36	71.58	73.16	35.68
odometerY	$[0.1,0.2]$	4.96	–	X	3.14	42	106.56	106.84	55.00
invplaneN	$[100,101]$	19,296.69	–	X	0.02	61,297	11,487,483.46	6,581,674.31	10,749,624.00
invplaneW	$[100,101]$	18,856.31	–	X	0.03	61,299	148,527.38	86,029.83	140,373.00
triton	$[5,5.1]$	16,276.01	–	X	0.23	980,762	14,644,943.59	9,707,296.23	14,414,369.00
ssats	$[10,10.1]$	6.44	–	X	0.09	68	118.39	115.60	68.00

Table 3. **stability-equivalence analysis with interval analysis.** u = unit roundoff, ✓ (X) = optimization preserves (does not preserve) stability

0	1	2	3	4	5	6
Program	**Stability Preserved?**	$\max \Delta^{c}_{\mathbb{P},\mathbb{P}'_{\text{match}}}$	$\max \Delta^{re}_{\mathbb{P},\mathbb{P}'_{\text{match}}}$ (ulp)	$\mathcal{T}(\mathbb{P}')$	**#opt. disabled**	$\mathcal{T}(\mathbb{P}'_{\text{match}})$
odometerX	✓	5	5.00	0.78	1/ 9	0.78 (100%)
odometerY	✓	4	4.00	0.71	1/ 8	0.73 (93%)
invplaneN	✓	1	0.00	0.91	2/ 2	1.00 (0%)
invplaneW	✓	2	4.00	0.81	2/ 6	0.85 (79%)
triton	✓	4	48.00	0.92	5/ 9	0.94 (75%)
ssats	✓	2	2.00	0.84	3/25	0.85 (94%)

Table 4. Effect of InstCombine **after** Matching. Columns 3–5 show the running time comparison. $\mathcal{T}(\mathbb{P}')$: running time after unrestricted fast-math, as a factor of the unoptimized running time. **#opt. disabled**: entry $x/y : x$ optimizations disabled due to Matching; y optimizations applied in original InstCombine . $\mathcal{T}(\mathbb{P}'_{\text{match}})$: running time after matching (speed-up percentage retained)

0	1	2	3	4	5	6	7	8	9	10
#Iters	**CompTime** (sec)	**Stability Preserved?**	$\max \Delta^{c}_{\mathbb{P},\mathbb{P}'}$	$\max \mathrm{re}^{\mathbb{P}}_{abs}$ (ulp)	$\max \mathrm{re}^{\mathbb{P}'}_{abs}$ (ulp)	$\max \Delta^{re}_{\mathbb{P},\mathbb{P}'}$ (ulp)	**Stability Preserved?**	$\max \Delta^{c}_{\mathbb{P},\mathbb{P}'}$	$\max \mathrm{re}^{\mathbb{P}'}_{abs}$ (ulp)	$\max \Delta^{re}_{\mathbb{P},\mathbb{P}'}$ (ulp)
200	14.51	X	69	144.20	144.20	68.74	✓	6	144.20	5.81
400	61.40	X	136	283.45	283.45	135.20	✓	4	283.45	4.00
600	127.67	X	309	449.92	451.92	392.41	✓	4	449.92	4.00
800	109.11	X	254	553.61	554.38	254.00	✓	4	553.61	4.00
1000	136.37	X	414	779.09	777.07	547.19	✓	4	779.09	4.00

Table 5. odometerX with different number of iterations. The input interval is [0.1, 0.2]. Columns 3–6 show the sampling results **before** matching. Column 8–10 show the results **after** matching.

0	1	2	3	4	5	6	7	8
Program	**Input interval**	$\max Q$	**Stability Preserved?**	**CompTime** (sec)	$\max \Delta^{c}_{\mathbb{F},\mathbb{F}'}$	$\max \mathrm{re}^{\mathbb{F}}_{abs}$ (ulp)	$\max \mathrm{re}^{\mathbb{F}'}_{abs}$ (ulp)	$\max \Delta^{re}_{\mathbb{F},\mathbb{F}'}$ (ulp)
turbine1	$[-10,-1] \wedge [1,10] \wedge [1,10]$	−4.05	✓	1.78	2	16.60	16.60	2.83
turbine3	$[-10,-1] \wedge [1,10] \wedge [1,10]$	−3.57	✓	1.76	2	3.78	3.43	3.00
pendulum	$[1,10] \wedge [1,10]$	−1.02	✓	0.82	1	1.56	1.56	0.00
carbongas	$[0.1,5]$	−1.21E+6	✓	1.62	2	2.14	2.14	2.00
precess	$[20,30] \wedge [1,10] \wedge [1,10]$	−1.08	✓	6.31	1	250.32	250.32	64.00
hestENG	$[10,20]$	−2.51E+6	✓	47.16	1	17.78	17.78	3.00
odometerX	$[0.1,10]$	6.51E+8	X	120.00	35.16	197.42	273.60	168.26
odometerY	$[0.1,10]$	4.15E+2	X	120.00	34.89	9829.31	9558.69	2418.00
invplaneN	$[100,110]$	7.56E+5	X	1.03	61,297	294,240,929,883.42	99,860,458,696.58	194,380,471,186.85
invplaneW	$[100,110]$	2.26E+6	X	2.15	61,299	147,265.71	85,494.40	140,376.00
triton	$[5,10]$	3.79E+3	X	119.23	978,313	246,604,117,190.51	57,792,698,629.74	239,979,893,197.02
ssats	$[5,15]$	1.24E+2	X	37.44	68.00	120.49	114.30	68.00

Table 6. **stability-equivalence analysis with stochastic search.** ✓ (X) = optimization preserves (does not preserve) stability

Experimental Results. We first show results of our stability-equivalence analysis with the interval-analysis sub-module. Column 1 of Table 3 lists the input constraints for each program. We chose these intervals arbitrarily but subject to the condition that they do not invoke sub-normals or exceptions during execution. In practice, the choice of interval will depend on the user applications. For example, if the input models temperatures, a suitable interval might correspond to room temperature. Columns 2–4 show the raw data from the analysis. $[\downarrow f, \uparrow f]$ represents the output from the *branch-and-bound* analysis of Eq. (21). For stability-equivalent programs, we report the largest $\uparrow f$ among the intervals computed along all branches; this value is below u. Otherwise, we report the $\downarrow f$ value of a sub-interval that exceeds u and thus violates equivalence.

To confirm that our results are consistent with a stability analysis based on actual rounding error differences (not sensitivity differences), we uniformly sample, for each benchmark, 100, 000 inputs within the input constraints (Column 1). With Eq. (20), we compute for the sampling inputs the maximum stability difference coefficient $\max \Delta^{c}_{\mathbb{P},\mathbb{P}'}$. Column 6 shows the results. The expectation is that for **(non-)**stability-equivalent programs, $\max \Delta^{c}_{\mathbb{P},\mathbb{P}'}$ should be small **(large)**. The results confirm our expectation. A case that stands out is `precess`: although its maximum error difference is large, the optimization is classified as stability-equivalent, and rightly so: the problem is ill-conditioned, leading to a high sensitivity to input errors, which dominate the arithmetic error. An "inequivalent" label would falsely blame the optimization.

To compare our analysis with one based on maximum rounding errors, for the sampling inputs we recorded the maximum rounding errors for both $\mathbb{P}$ and $\mathbb{P}'$, and the largest rounding error differences. For the computation of the exact $\mathbb{P}_0$ we use the `MPFR` library with 256-bit precision

arithmetic [12]. Columns 7–9 show the results: for most programs the maximum rounding errors of $\mathbb{P}$ and $\mathbb{P}'$ are similar, including for programs not stability-equivalent, where the rounding error difference for certain inputs is much larger, as evident from Column 9. This demonstrates how our approach more precisely analyzes optimization effects on individual inputs.

To demonstrate the effects of our *Matching* Step 4 in Fig. 4, Table 4 shows, for the same input samples, the results after stability-matching the affected benchmarks. We see that $\max \Delta^c_{\mathbb{P},\mathbb{P}'}$ is small, and that the maximum rounding error difference has decreased, in some cases by orders of magnitude (Column 3).

To evaluate the potential performance degradation of our stability-preserving InstCombine pass, Columns 4–6 of Table 4 compare the running time $\mathcal{T}(\mathbb{P}'_{\text{match}})$ using our pass to that using the original InstCombine pass $\mathcal{T}(\mathbb{P}')$, and using the unoptimized program $\mathcal{T}(\mathbb{P})$, measured as the average over 100,000 runs. We can see that the slow-down due to partially disabling fast-math is negligible for our benchmarks. This is explained by the small number of disabled optimizations. For invplaneN there were very few applicable optimizations to begin with, which all had to be disabled to achieve stability, leading to a speed-up retention of 0%. In general, however, the stability-equivalent optimized program should offer a good trade-off between running time and reproducibility.

The time taken for our various passes themselves, i.e. the compilation time (**CompTime**), is below 1sec for most of our benchmarks (see Column 5 of Table 3). One noticeable exception is the *precess* example, which takes about 53 seconds. The bottleneck lies in the branch-and-bound algorithm, which has to repeatedly split the input intervals to compute more precise ranges of f. In general, as the size of the user-provided input constraints increases, the cost of running interval analysis will become more expensive.

We also evaluate the scalability of our approach. Since many existing floating-point benchmarks in this domain are fairly small, for scalability checking purposes we use the *odometer* example (List. 1) with an increasing number of loop iterations. Following the discussion in Sec. 8, we fully unroll the loops for each test case. The results are shown in Table 5. We learn that, before matching, the $\max \Delta^c_{\mathbb{P},\mathbb{P}'}$ increases quite strongly (from 69 to 414) with the number of iterations. In contrast, it is drastically reduced to around 4 after matching. As for the case of 100 iterations, the culprit for the stability difference before matching is Line 16 of List. 1, for all iteration counts. As a result, the performance impact of our stability-preserving pass is similar as in the 100 iterations case (we do not repeat the numbers here). This again shows that the stability-equivalent optimized program offers a good trade-off between running time and reproducibility. Column 1 shows the compilation time for the various iteration counts. The case of 1000 iterations takes little more than 2mins. These numbers suggest that, for small interval sizes, our analysis scales well with program size.

Finally, to be able to handle larger input intervals, we can replace the interval analysis with the stochastic search module. We tested this module on significantly larger input intervals for all our benchmarks. We set the resource limit to 10,000 stochastic samples or 2mins time, whichever is reached first. Table 6 shows the results. The results are consistent with those obtained by the interval analysis technique. Stochastic search can quickly detect stability-equivalence violations and provide evidence via concrete inputs. A downside is that an insufficient sample size may underapproximate the input space and lead to unreliable results. A strategy to employ both interval analysis and stochastic search in practice may be to run them in parallel, as they enjoy complementary strengths. Designing such a combined strategy is left for for future work.

10 RELATED WORK

In early work, Miller developed a general mechanism, called the *Wilkinson number* (ω) [24], to compare the rounding errors of two numeric programs. We have shown that our stability-equivalent concept implies closeness of Wilkinson number of the unoptimized and optimized program. Also we only consider the intermediate rounding errors where the codes of $\mathbb{P}$ and $\mathbb{P}'$ differ. This is in contrast with [24], which requires to consider all rounding errors occurring in $\mathbb{P}$ and $\mathbb{P}'$. The computing cost of our approach is thus relatively cheaper.

Another approach for comparison is to use static analysis [8, 11, 26, 35], theorem proving [4], or stochastic search [1, 7, 34] to compare *the difference between maximum rounding errors* for the optimized and unoptimized programs. In contrast, our work compares their stability property. Besides the rounding errors in the computed results, our concept also captures the inherited property—the condition number—in the numerical problem itself. Thus, it gives users a more complete picture about the relationship between optimized and unoptimized programs. Moreover, we also provide diagnostic information that informs programmers about the root-causes of the differences so they can be addressed, e.g. by taming the optimizations so as to preserve the program's stability. This has not been considered in previous work.

The authors of [23, 28] use SMT solvers to show bit-level equivalence for peep-hole floating-point optimizations. Such work is part of the general goal of strict program equivalence checking [22]. Unlike our work, equivalence checkers can afford to focus solely on the optimization rule level and can ignore propagation, since local bit-precise equivalence implies global bit-precise equivalence. Such I/O equivalence, however, is too strict for many user applications. For these cases, a more relaxed notion is to allow the computed results of optimized code to differ *slightly* from the original ones. This idea is employed in tools that simply ask users for assistance in finding a suitable threshold [21]. This need is eliminated by our approach: we automatically derive the threshold from the stability-equivalence requirement.

Our work can be used as a follow-up step to efficiency-optimize code generated by precision tuning tools like STOKE [34], FPTuner [6] and Precimonious [32]. Value-changing optimizations like performed by InstCombine sabotage source-level code tuning techniques. Similarly, precision tuning performed after runtime optimization will likely impact any speed-up obtained. The key to achieving the best of the worlds of efficiency and precision is to make the analyses *aware* of each other, or at least the later aware of the earlier. In this spirit, we have presented work that increases the precision accountability of aggressive floating-point optimizations, and is thus able to curb their negative side effects.

11 CONCLUSION

In this paper we have developed the concept of *stability-equivalence analysis* to help programmers decide whether a program's numeric stability is robust against unsafe compiler optimizations. Stability-equivalence implies that these optimizations can be used with increased confidence that their impact on the rounding error is marginal. Our results support the following conclusions: (i) the sensitivity difference, an approximation of rounding error difference, can be determined efficiently and is consistent with the actual stability difference (based on rounding errors) between $\mathbb{P}$ and $\mathbb{P}'$; and (ii) for stability-*in*equivalent optimized programs, one can identify the expressions whose rounding error is affected most by the optimizations. This information can be used to enforce stability-equivalence, while keeping the efficiency benefits mostly intact.

In addition to closing the gaps of the tool regarding special floating-point values and rich control structure, future work includes applying our stability-equivalence analysis to other unsafe optimizations such as vectorization and the FMA operation.

APPENDIX

Complete Union Operations for Unsafe Instcombine Optimizations

The complete set of union operations for unsafe InstCombine optimizations is given in Fig. 9.

Simplify Assoc/Commut :

$(X \otimes c_0) \otimes c_1 \sqcup X \otimes (c_0 \otimes c_1) := X \otimes (c_0 \otimes c_1) + \dot{P}$

$c_0 \otimes (c_1 \otimes X) \sqcup (c_0 \otimes c_1) \otimes X := (c_0 \otimes c_1) \otimes X + \dot{P}$

$(c_0 \otimes X) \otimes c_1 \sqcup (c_1 \otimes c_0) \otimes X := (c_1 \otimes c_0) \otimes X + \dot{P}$

$c_0 \otimes (X \otimes c_1) \sqcup X \otimes (c_1 \otimes c_0) := X \otimes (c_1 \otimes c_0) + \dot{P}$

$(X \otimes c_0) \otimes (Y \otimes c_1) \sqcup (X \otimes Y) \otimes (c_0 \otimes c_1)$
$:= (X \otimes Y) \otimes (c_0 \otimes c_1) + \dot{Q}$

$\dot{P} := c_0 c_1 X \cdot (1 - \epsilon/u) \cdot \delta, \dot{Q} := c_0 c_1 XY \cdot (1 - \epsilon/u) \cdot \delta$

$\epsilon := rel(c_0 c)$

PowerX:

$(X \otimes Y) \otimes X \equiv_S (X \otimes X) \otimes Y$

FoldFMulConst:

$(c_0 \oslash X) \otimes c \sqcup (c_0 \otimes c) \oslash X := ((c_0 \otimes c) \oslash X) + \dot{P},$

$\dot{P} := (c_0 c/X) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_0 c)$

$(X \oslash c_1) \otimes c \sqcup X \otimes (c \oslash c_1) := (X \otimes (c \oslash c_1)) + \dot{P},$

$\dot{P} := (Xc/c_1) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c/c_1)$

$(X \oslash c_1) \otimes c \sqcup X \oslash (c_1 \oslash c) := (X \oslash (c_1 \oslash c)) + \dot{P},$

$\dot{P} := (X/(c_1/c)) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1/c)$

Log2OfHalf:

$X \otimes log(0.5 \otimes Y) \sqcup X \otimes log(Y) \ominus X := X \otimes log(Y) \ominus X + \dot{P},$

$\dot{P} := (X \cdot log(Y) + X) \cdot \delta$

MulDivideConst:

$(\text{MDC} \oplus /\ominus c_1) \otimes c \sqcup (\text{MDC} \otimes c) \oplus /\ominus(c_1 \otimes c)$
$:= (\text{MDC} \otimes c) \oplus /\ominus(c_1 \otimes c) + \dot{P}$

$\dot{P} := c_1 c \cdot (1 - \epsilon/u) \cdot \delta, \epsilon = rel(c_1 c)$

(a) Union operations for unsafe optimizations of *FMul*.

FoldFDivConst:

$(X \otimes c_1) \oslash c_2 \sqcup X \otimes (c_1 \oslash c_2) := X \otimes (c_1 \oslash c_2) + \dot{P},$

$\dot{P} := (Xc_1/c_2) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1/c_2)$

$(X \oslash c_1) \oslash c_2 \sqcup X \oslash (c_1 \otimes c_2) := X \oslash (c_1 \otimes c_2) + \dot{P},$

$\dot{P} := (X/(c_1 c_2)) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1 c_2)$

$c_1 \oslash (X \otimes c_2) \sqcup (c_1 \oslash c_2) \oslash X := (c_1 \oslash c_2) \oslash X + \dot{P},$

$\dot{P} := ((c_1/c_2)/X) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1/c_2)$

$c_1 \oslash (X \oslash c_2) \sqcup (c_1 \otimes c_2) \oslash X := (c_1 \otimes c_2) \oslash X + \dot{P},$

$\dot{P} := (c_1 c_2/X) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1 c_2)$

$c_1 \oslash (c_2 \oslash X) \sqcup (c_1 \oslash c_2) \otimes X := (c_1 \oslash c_2) \otimes X + \dot{P},$

$\dot{P} := (Xc_1/c_2) \cdot (1 - \epsilon/u) \cdot \delta, \epsilon := rel(c_1/c_2)$

(b) Union operations for unsafe optimizations of *FDiv*.

Assoc/Commut:

$(X \oplus Y) \oplus Z \sqcup (X \oplus Z) \oplus Y := (X \oplus Z) \oplus Y + \dot{P}$

$\dot{P} := (Y - Z) \cdot \delta$

$(X \oplus Y) \oplus Z \sqcup X \oplus (Y \oplus Z) := X \oplus (Y \oplus Z) + \dot{P}$

$\dot{P} := (X - Z) \cdot \delta$

Factorization:

$(X \otimes Y) \oplus (X \otimes Z) \sqcup X \otimes (Y \oplus Z) := X \otimes (Y \oplus Z) + \dot{P}, \dot{P} := 2XY \cdot \delta$

$(X \otimes Y) \ominus (X \otimes Z) \sqcup X \otimes (Y \ominus Z) := X \otimes (Y \ominus Z) + \dot{P}, \dot{P} := 2XZ \cdot \delta$

$(Y \oslash X) \oplus (Z \oslash X) \sqcup (Y \oplus Z) \oslash X := (Y \oplus Z) \oslash X + \dot{P}, \dot{P} := 2Y/X \cdot \delta$

$(Y \oslash X) \ominus (Z \oslash X) \sqcup (Y \ominus Z) \oslash X := (Y \ominus Z) \oslash X + \dot{P}, \dot{P} := 2Z/X \cdot \delta$

(c) Union Operations for Optimizations of *FAdd*.

Const-Assoc/Commut:

$(c_1 \oplus X) \oplus c_2 \sqcup (c_1 \oplus c_2) \oplus X := (c_1 \oplus c_2) \oplus X + \dot{P}$

$(X \oplus c_1) \oplus c_2 \sqcup X \oplus (c_1 \oplus c_2) := X \oplus (c_1 \oplus c_2) + \dot{P}$

$\dot{P} := (X - c_2) \cdot \delta_1 + (c_1 + c_2) \cdot (1 - \epsilon/u) \cdot \delta_2$

$\epsilon := rel(c_1 + c_2)$

Const-Factorization:

$(X \otimes c_1) \oplus (X \otimes c_2) \sqcup X \otimes (c_1 \oplus c_2) := X \otimes (c_1 \oplus c_2) + \dot{P}$

$\dot{P} := 2Xc_1 \cdot \delta_1 + X \cdot (c_1 + c_2) \cdot (1 - \epsilon/u) \cdot \delta_2, \epsilon := rel(c_1 + c_2)$

$(X \otimes c_1) \ominus (X \otimes c_2) \sqcup X \otimes (c_1 \ominus c_2) := X \otimes (c_1 \ominus c_2) + \dot{P}$

$\dot{P} := 2Xc_2 \cdot \delta_1 + X \cdot (c_1 - c_2) \cdot (1 - \epsilon/u) \cdot \delta_2, \epsilon := rel(c_1 - c2);$

$(c_1 \oslash X) \oplus (c_2 \oslash X) \sqcup (c_1 \oplus c_2) \oslash X := (c_1 \oplus c_2) \oslash X + \dot{P}$

$\dot{P} := 2c_1/X \cdot \delta_1 + (c_1 + c_2)/X \cdot (1 - \epsilon/u) \cdot \delta_2, \epsilon := rel(c_1 + c2)$

$(c_1 \oslash X) \ominus (c_2 \oslash X) \sqcup (c_1 \ominus c_2) \oslash X := (c_1 \ominus c_2) \oslash X + \dot{P}$

$\dot{P} := 2c_2/X \cdot \delta_1 + (c_1 - c_2)/X \cdot (1 - \epsilon/u) \cdot \delta_2, \epsilon := rel(c_1 - c_2)$

(d) Union Operations for Const-Optimizations of *FAdd*.

Fig. 9. Union operations for unsafe optimizations of InstCombine. $rel: \mathbb{R} \rightarrow \mathbb{R}$ denotes the relative error of the input. In the union definition for *Log2OfHalf* we assume that the log function is correctly rounded, which is not required but recommended by the IEEE standard

REFERENCES

[1] Tao Bao, Yunhui Zheng, and Xiangyu Zhang. 2012. White box sampling in uncertain data processing enabled by program analysis. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. SIGPLAN, Tucson, AZ, USA, 897–914. https://doi.org/10.1145/2384616.2384681

[2] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 549–560. https://doi.org/10.1145/2429069.2429133

[3] F. L. Bauer. 1974. Computational Graphs and Rounding Error. *SIAM J. Numer. Anal.* 11, 1 (1974), 87–96. http://www.jstor.org/stable/2156433

[4] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*. 243–252. https://doi.org/10.1109/ARITH.2011.40

[5] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. 2006. The design of the Boost interval arithmetic library. *Theor. Comput. Sci.* 351, 1 (2006), 111–118. https://doi.org/10.1016/j.tcs.2005.09.062

[6] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, Paris, France, 300–315. http://dl.acm.org/citation.cfm?id=3009846

[7] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. 43–52. https://doi.org/10.1145/2555243.2555265

[8] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*. 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

[9] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. 2017. Improving the numerical accuracy of programs by automatic transformation. *STTT* 19, 4 (2017), 427–448. https://doi.org/10.1007/s10009-016-0435-0

[10] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2016. 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*. IEEE, Toronto, ON, Canada, 63–77. https://doi.org/10.1007/978-3-319-54292-8_6

[11] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 8:1–8:28. https://doi.org/10.1145/3014426

[12] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (2007), 13. https://doi.org/10.1145/1236463.1236468

[13] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives - principles and techniques of algorithmic differentiation (2. ed.)*. SIAM, Philadelphia, PA, USA. https://doi.org/10.1137/1.9780898717761

[14] Yijia Gu and Thomas Wahl. 2017. Stabilizing Floating-Point Programs Using Provenance Analysis. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and David Monniaux (Eds.), Vol. 10145. Springer, Paris, France, 228–245. https://doi.org/10.1007/978-3-319-52234-0_13

[15] Nicholas J. Higham. 2002. *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM, Philadelphia, PA, USA.

[16] IEEE Standard Association. 2008. IEEE Standard for Floating-Point Arithmetic. (2008), 58 pages. http://grouper.ieee.org/groups/754/.

[17] Masao Iri, Takashi Tsuchiya, and Mamoru Hoshi. 1988. Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations. *J. Comput. Appl. Math.* 24, 3 (1988), 365 – 392. https://doi.org/10.1016/0377-0427(88)90298-1

[18] Anastasiia Izycheva and Eva Darulova. 2017. On sound relative error bounds for floating-point arithmetic. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, Vienna, Austria, 15–22. https://doi.org/10.23919/FMCAD.2017.8102236

[19] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. 2001. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer London Ltd, Longdon. 398 pages. https://hal.archives-ouvertes.fr/hal-00845131

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and*

Runtime Optimization (CGO '04). IEEE Computer Society, Washington, DC, USA, 75–88. http://dl.acm.org/citation.cfm?id=977395.977673

[21] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 70–84. https://doi.org/10.1145/2908080.2908107

[22] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[23] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. 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*. Springer, Edinburgh, UK, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

[24] Webb Miller. 1975. Software for Roundoff Analysis. *ACM Trans. Math. Softw.* 1, 2 (1975), 108–128. https://doi.org/10.1145/355637.355639

[25] Webb Miller. 1976. Roundoff Analysis by Direct Comparison of Two Algorithms. *SIAM J. Numer. Anal.* 13, 3 (1976), 382–392. http://www.jstor.org/stable/2156306

[26] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

[27] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, US. https://doi.org/10.1007/978-0-8176-4705-6

[28] Andres Nötzli and Fraser Brown. 2016. LifeJacket: verifying precise floating-point optimizations in LLVM. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*. ACM, Santa Barbara, CA, USA, 24–29. https://doi.org/10.1145/2931021.2931024

[29] (omitted). June 2019. Personal communication. (June 2019).

[30] July 1992. The Patriot missile failure. *SIAM News* 25, 4 (July 1992), 11.

[31] Project Pluto. 2016. Project Pluto. http://www.projectpluto.com. (2016). [Online; accessed 03-April-2018].

[32] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 27:1–27:12. https://doi.org/10.1145/2503210.2503296

[33] Timothy Sauer. 2011. *Numerical Analysis* (2nd ed.). Addison-Wesley Publishing Company, USA.

[34] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, Edinburgh, United Kingdom, 53–64. https://doi.org/10.1145/2594291.2594302

[35] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 532–550. https://doi.org/10.1007/978-3-319-19249-9_33

[36] StatPro. 2018. QuantLib. http://www.quantlib.org/. (2018). [Online; accessed 03-April-2018].

[37] Boyd Stephen, Lin Xiao, and Almir Mutapcic. Autumn Quarter 2004. *Subgradient methods*. Stanford University, California, USA.

[38] F. Stummel. 1981. Perturbation Theory for Evaluation Algorithms of Arithmetic Expressions. *Math. Comp.* 37, 156 (1981), 435–473. http://www.jstor.org/stable/2007438