Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2019-IC-007

2019-IC-007

Global optimization of numerical programs via prioritized stochastic algebraic transformations

Xie Wang, Huaijin Wang, Zhendong Su, Enyi Tang, Xin Chen, Weijun Shen, Zhenyu Chen,

Linzhang Wang, Xianpei Zhang, Xuandong Li

International Conference on Software Engineering

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations

Xie Wang[1] Huaijin Wang[1] Zhendong Su[2,3] Enyi Tang[1*] Xin Chen[1]
Weijun Shen[1] Zhenyu Chen[1] Linzhang Wang[1] Xianpei Zhang[1] Xuandong Li[1]

[1]State Key Laboratory of Novel Software Technology, Software Institute of Nanjing University, Nanjing, China
[2]Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland
[3]Department of Computer Science, University of California, Davis, USA
*corresponding author: eytang@nju.edu.cn

***Abstract*—Numerical code is often applied in the safety-critical, but resource-limited areas. Hence, it is crucial for it to be correct and efficient, both of which are difficult to ensure. On one hand, accumulated rounding errors in numerical programs can cause system failures. On the other hand, arbitrary/infinite-precision arithmetic, although accurate, is infeasible in practice and especially in resource-limited scenarios because it performs thousands of times slower than floating-point arithmetic. Thus, it has been a significant challenge to obtain *high-precision*, *easy-to-maintain*, and *efficient* numerical code. This paper introduces a novel *global optimization* framework to tackle this challenge.**

Using our framework, a developer simply writes the infinite-precision numerical program directly following the problem's mathematical requirement specification. The resulting code is correct and easy-to-maintain, but inefficient. Our framework then optimizes the program in a global fashion (*i.e.*, considering the whole program, rather than individual expressions or statements as in prior work), the key technical difficulty this work solves. To this end, it analyzes the program's numerical value flows across different statements through a symbolic trace extraction algorithm, and generates optimized traces via stochastic algebraic transformations guided by effective rule selection. We first evaluate our technique on numerical benchmarks from the literature; results show that our global optimization achieves significantly higher worst-case accuracy than the state-of-the-art numerical optimization tool. Second, we show that our framework is also effective on benchmarks having complicated program structures, which are challenging for numerical optimization. Finally, we apply our framework on real-world code to successfully detect numerical bugs that have been confirmed by developers.

I. Introduction

Numerical programs are often key components of safety-critical systems, so it is crucial for them to get correct. As many of such systems are restricted with limited resources, such as a very limited energy budget for an on-board satellite system, it is also important for numerical programs to be efficient. However, both the correctness and the efficiency of numerical programs are difficult to ensure. On one hand, accumulated rounding errors in numerical programs can cause system failures. On the other hand, numerical software with arbitrary-precision arithmetic performs thousands of times slower than the traditional fixed-precision floating point programs, which makes it infeasible in the resource-limited scenario. Hence, experts need to elaborately design stable numerical programs to make sure of their correctness and efficiency.

Unfortunately, the elaborately designed numerical programs are difficult to maintain. Numerical experts often introduce complicated tricks in them. For example, a lot of numerical algorithms are not as intuitive as its original mathematical requirement specification because numerical experts use different ways of calculation to make sure that the representation and rounding errors are accumulated as less as possible. Furthermore, experts also introduce precision-specific operations that are difficult to understand by developers who do not familiar with numerical programming. Our goal is to address such maintenance problems.

In this paper, we propose a global optimization framework (shown in Figure 1) that transforms numerical programs directly following the mathematical requirement specification to the efficient fixed-precision programs with numerically stable algorithms. Using our framework, a developer just needs to simply write a numerical program following the mathematical specification from the software requirements. When the input program is easy to maintain, we call it a *direct numerical program* written in infinite-precision arithmetic. Then our framework changes the calculation of the direct program and generates an efficient equivalent program in fixed-precision floating-point arithmetic with stable numerical algorithms.

The framework decouples the knowledge of numerical calculation from the software development. In other words, developers do not need to focus on the details in numerical programming, but just thinking in terms of the real number, which makes their programs intuitive and easy to maintain. Our optimization framework considers its floating-point approximation on computers by the optimization rules from numerical experts, who focus on the numerical analysis techniques and make our framework incrementally powerful.

Our optimization is *global*. It not only rewrites individual expressions or statements, but optimizes the unstable calculation flows of the whole program. The global optimization framework brings two major challenges: 1) Different from a local numerical expression, a numerical program often contains multiple branches with complicated constraints. *A global numerical optimization needs not only to rewrite the numerical operations with high local error in a program, but also reduce the cumulative floating-point errors with constraints.* 2) A global

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00116

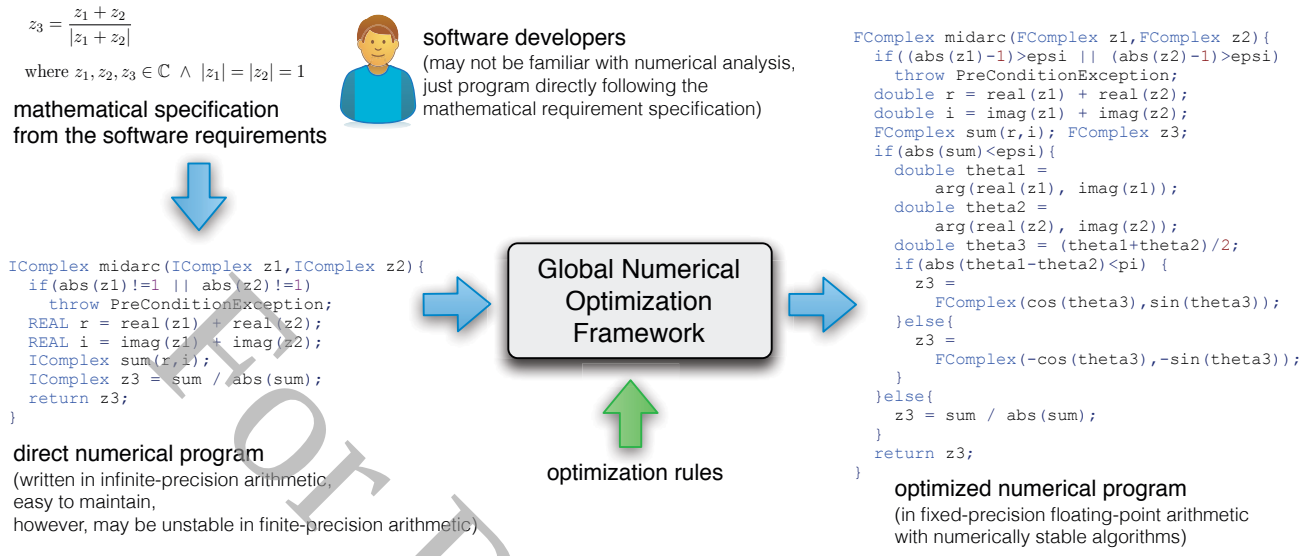


Fig. 1: Illustration of the Optimization Framework for Globally Rewriting Numerical Programs

numerical flow is often much longer than a local numerical expression. That is, more potential numerical rewrites consist in a global flow, which leads to vaster search space than just rewriting local numerical expressions.

To overcome the first challenge, we propose a symbolic trace extraction algorithm inspired by the symbolic execution technique [1], [2]. The algorithm connects the numerical operations in different statements together, and transforms the input numerical program to an *intermediate representation* (IR) that records the numerical traces with their corresponding path constraints. To overcome the second challenge, we present a stochastic algebraic transformation guided by an effective rule selection strategy. The strategy characterizes the floating-point domain knowledge as prioritized preconditions of transformation rules, and prioritizes the transformation in consideration of both its precondition and success rate.

State of the Art Numerical verification and analysis are extensively studied topics [3], [4], [5], [6], [7], [8]. Most of these researches provide techniques that compute error bounds, detect numerical instabilities, or elaborately design stable numerical programs manually. Herbie [9] is a state-of-the-art transformation tool that rewrites a local expression to improve its accuracy. But it can not handle a numerical program with path constraints and program structures. To the best of our knowledge, none of the previous work has considered a global optimization framework that automatically rewrites the whole program to a stable one.

The main contributions of this paper are as follows:

- We propose a global optimization framework that transforms infinite precision numerical programs to the stable and efficient fixed-precision implementations. It is the first framework that automatically takes global numerical errors into consideration to our knowledge.
- We carefully design the framework with a symbolic trace extraction algorithm, a numerical intermediate representation (IR), and a stochastic algebraic transformation. With these techniques, our framework overcomes the major challenges in the global numerical optimization.
- We first evaluate our optimization on groups of numerical benchmarks from the literature. Results show that our global optimization achieves significantly higher worst-case accuracy than the state-of-the-art numerical optimization tool, which is also effective on benchmarks having complicated program structures. Furthermore, we apply our framework on real-world projects of an open source graphics library and a driving tool for 3D printers, and successfully detect numerical bugs that have been confirmed by developers.

II. Floating-Point Background & Numerical Errors

According to IEEE 754 [10], each floating-point number represents a numerical value in the form of

$$(-1)^s(1+m)2^e \tag{1}$$

where s is the sign bit (zero or one), m is the significand (also called the mantissa), e is the exponent. The floating-point standard also defines a variety of precisions: double-precision floats are defined with a 52 bit significand and an 11 bit exponent, while single-precision floats are are defined with a 23 bit significand and an 8 bit exponent.

When the floating-point format bounds the number of bits in its representation, the precision limitations are the root cause of the numerical errors [11]. Our framework uses infinite-precision arithmetic [12] to evaluate the errors that are introduced by a fixed-precision floating-point program. The infinite-precision arithmetic expands the precision automatically on an arbitrary-precision floating-point format, such as GNU MPFR [13]. It increases the working precision until the observed bits of the

significand (such as the first 52 bits of the computed answer for evaluating a double-precision program) does not change any more. As the infinite-precision arithmetic achieves a sufficiently large working precision, we treat the output as the exact result in its observed bits of the significand.

Metrics are necessary to evaluate the errors between the floating-point outputs and the exact results. The absolute and relative error functions are natural measures, but they will suffer from ill-suited cases in evaluating floating-point values [14]. Following a few recent practices [15], [9] in floating-point evaluation, we also use the (base-2 logarithm of the) number of floating-point values between the exact result x and approximate result $\hat{x}$ as the error metric $\mathcal{E}$, which we call *error in bits* in this paper:

$$\mathcal{E}(x,\hat{x}) = \log_2 \left| \{n | n \in \mathbb{FP} \wedge \min(x,\hat{x}) \leq n \leq \max(x,\hat{x})\} \right|$$

Error in bits $\mathcal{E}$ represents uniformly the entire range of floating-point values including the `Inf` and `NaN` values. Intuitively, it measures the number of most-significant bits that x and $\hat{x}$ agree on. Note that the significand `0x00FF` and `0x0100` just have 1 bit error, not 9 bits. The metric $\mathcal{E}$ is still available if x and $\hat{x}$ have a different sign bit or exponent bits. So the range of error in bits for double-precision values is [0,64], and [0,32] for single-precision values.

III. Motivating Example

In Figure 1, we have presented a `midarc` example from Guo et al. [16], which calculates the midpoint z_3 of an arc (z_1, z_2) in the complex plane.

As z_1, z_2, z_3 are complex numbers and $|z_1| = |z_2| = 1$, the arc (z_1, z_2) is a segment of the unit circle in the complex plane (shown in Figure 2). z_3 is the midpoint of the arc (z_1, z_2), and can be calculated according to the equation:

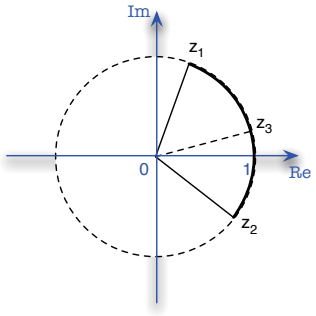


Fig. 2: The Argand Diagram of our Example that Calculates the Midpoint z_3 of the Arc (z_1, z_2)

$$z_3 = \frac{z_1 + z_2}{|z_1 + z_2|} \quad (2)$$

Following Equation 2 and the basic arithmetic rules of complex numbers, software developers can implement the `midarc` program directly as the code snippet in Figure 3. The code in Figure 3 is intuitive and easy to maintain, however it is not an efficient and stable numerical implementation. First, state-of-the-art implementations of infinite-precision arithmetic perform thousands times slower than fixed-precision floating-point programs. So we need the efficient fixed-precision implementation. Second, a fixed-precision floating-point program can be unstable if it follows the direct algorithm in Figure 3. The instability occurs

```
1  IComplex midarc(IComplex z1,IComplex z2){
2    if(abs(z1)!=1 || abs(z2)!=1)
3      throw PreConditionException;
4    REAL r = real(z1) + real(z2);
5    REAL i = imag(z1) + imag(z2);
6    IComplex sum(r,i);              //sum=z1+z2
7    IComplex z3 = sum / abs(sum);
8    return z3;
9  }
```

Fig. 3: The Direct Numerical Program for Calculating the Midpoint `z3` in Infinite Precision Arithmetic

```
1  FComplex midarc(FComplex z1,FComplex z2){
2    if((abs(z1)-1)>epsi || (abs(z2)-1)>epsi)
3      throw PreConditionException;
4    double r = real(z1) + real(z2);
5    double i = imag(z1) + imag(z2);
6    FComplex sum(r,i); FComplex z3;
7    if(abs(sum)<epsi){
8      double theta1 = arg(real(z1), imag(z1));
9      double theta2 = arg(real(z2), imag(z2));
10     double theta3 = (theta1+theta2)/2;
11     if(abs(theta1-theta2)<pi) {
12       z3 = FComplex(cos(theta3),sin(theta3));
13     }else{
14       z3 = FComplex(-cos(theta3),-sin(theta3));
15     }
16   }else{
17     z3 = sum / abs(sum);
18   }
19   return z3;
20 }
```

Fig. 4: The Optimized Fixed-Precision Numerical Program for Calculating the Midpoint `z3`

when $|z_1 + z_2| < \varepsilon$, where ε is a positive real value close to 0.

When $|z_1 + z_2|$ is very small, both the addition at line 4 and line 5 in Figure 3 cause serious massive cancellations if we calculate it in finite-precision arithmetic. The error is further enlarged by the division at line 7 because the divisor is also very small, which is equal to $|z_1 + z_2|$. Hence, when we run the program with double-precision floating point arithmetic on an Apple MacBook Pro with Intel Core i7 2.9GHz, where the input $z_1 = e^{i\pi/4}$, $z_2 = -z_1 * e^{-i\varepsilon\pi}$ and $\varepsilon = 10^{-16}$, it outputs $z_3 = -0.5547 + 0.8321i$ with about 51 bits error in both the real part and the imaginary part. It is obviously wrong since the correct $z_3 \approx -0.7071 + 0.7071i$.

When every local expression in Figure 3 (such as `imag(z1)+imag(z2)`, `sum/abs(sum)`) does not have a rewriting form to make the program numerically stable, existing local optimization techniques are not effective in this problem. Our framework optimizes the code globally. It extracts a numerical IR that connects the numerical calculation flows together with a symbolic trace extraction algorithm. Inspired by symbolic execution [1], [2], the symbolic trace extraction algorithm propagates symbolic states through every execution path, and records constraints between numerical variables.

TABLE I: Transformation Rules that Match the Path & Variable Constraints in Equation 3

No.	Rule
#1	$A + B \rightsquigarrow B + A$
#2	$(A+B)^2 \rightsquigarrow A^2 + 2AB + B^2$
#3	$R_1{+}R_2 \rightsquigarrow \sqrt{(R_1{+}R_2)^2{+}(I_1{+}I_2)^2}\cos(\theta_1{+}\arg(m_1{+}m_2\cos(\theta_2{-}\theta_1), m_2\sin(\theta_2{-}\theta_1)))$ where $m_1{=}\sqrt{R_1^2{+}I_1^2}, m_2{=}\sqrt{R_2^2{+}I_2^2}, \theta_1{=}\arg(R_1, I_1), \theta_2{=}\arg(R_2, I_2)$
#4	**if** $(R_1^2{+}I_1^2 == R_2^2{+}I_2^2 \wedge \lvert\arg(R_1, I_1){-}\arg(R_2, I_2)\rvert < \pi \boxed{\wedge \lvert R_1{+}R_2\rvert < \varepsilon})$ $R_1{+}R_2 \rightsquigarrow \sqrt{(R_1{+}R_2)^2{+}(I_1{+}I_2)^2}\cos(\frac{\arg(R_1,I_1){+}\arg(R_2,I_2)}{2})$
#5	**if** $(R_1^2{+}I_1^2 == R_2^2{+}I_2^2 \wedge \lvert\arg(R_1, I_1){-}\arg(R_2, I_2)\rvert \geq \pi \boxed{\wedge \lvert R_1{+}R_2\rvert < \varepsilon})$ $R_1{+}R_2 \rightsquigarrow -\sqrt{(R_1{+}R_2)^2{+}(I_1{+}I_2)^2}\cos(\frac{\arg(R_1,I_1){+}\arg(R_2,I_2)}{2})$
...	... ...

When the algorithm unfolds the subfunction such as `abs`, the collected global trace is:

$$\begin{aligned}\textbf{if}\ \ & (\sqrt{\texttt{z1}.Re^2 + \texttt{z1}.Im^2} == 1 \\ & \bigwedge\ \sqrt{\texttt{z2}.Re^2 + \texttt{z2}.Im^2} == 1) \qquad (3)\\ \texttt{z3}.Re &:= \frac{\texttt{z1}.Re + \texttt{z2}.Re}{\sqrt{(\texttt{z1}.Re + \texttt{z2}.Re)^2 + (\texttt{z1}.Im + \texttt{z2}.Im)^2}} \\ \texttt{z3}.Im &:= \frac{\texttt{z1}.Im + \texttt{z2}.Im}{\sqrt{(\texttt{z1}.Re + \texttt{z2}.Re)^2 + (\texttt{z1}.Im + \texttt{z2}.Im)^2}}\end{aligned}$$

where $\texttt{z1}.Re$ refers to the memory object returned by `real(z1)` that stores the real component of the complex value `z1`, whereas $\texttt{z2}.Re$, $\texttt{z3}.Re$, $\texttt{z1}.Im$, $\texttt{z2}.Im$ and $\texttt{z3}.Im$ refer to the corresponding memory objects.

After comparing the outputs between the infinite-precision program and the corresponding fixed-precision program with a heuristic sampling strategy, our framework locates the input regions that trigger the numerical instabilities. The framework focuses the optimization on these unstable input regions, and yields a new optimized branch with the path constraint derived from these regions in the output program. In our example, the path constraint of the new optimized branch is `abs(sum)<epsi`, where `epsi` is a small value close to 0.

The insight of our optimization is to find the numerically stable forms for the calculations under the unstable regions of the input program. Our framework substitutes these optimized forms in the new branch of the output program. Unfortunately, if the framework cannot find them, it will keep the infinite-precision calculations in the new optimized branch to make sure the correctness of the output program. Meanwhile, the output program is still efficient in other stable input regions since the calculations under these regions are transformed to the fixed-precision floating-point arithmetic directly.

Our framework searches the numerically stable forms for optimization with a stochastic algebraic transformation, which rewrites the global calculations in a rule-based manner. Specifically, it rewrites the global constraint in the numerical IR with a database of transformation rules from numerical experts. As a global numerical constraint is often more complicated than a local expression, our framework records the optimization success count for every rule in the database, and applies it as a gauge of the probability for selecting the rule in future. Furthermore, we refine the optimization rules with prioritized preconditions to further introduce the floating-point domain knowledge in the transformation. These transformation strategies significantly increase the chance of finding the effective stable forms in the large search space for the global numerical optimization.

Table I depicts some of the rules that can be applied in our example. We define sub-procedures for complicate rules such as Rule #3 (the polar form conversion rule for the real-part addition). Rule #4 and Rule #5 are special cases of Rule #3 with preconditions. Some of the preconditions such as $R_1^2{+}I_1^2 == R_2^2{+}I_2^2$ guarantee the correctness of the rule. So we call them the *correctness preconditions*. Other preconditions boxed in Table I are *prioritized preconditions* that significantly increase the chance to apply the corresponding rule if any of them is satisfied. The insight of a prioritized precondition is the prior knowledge numerical experts already know that a transformation should be effective under such condition. This information is fundamental in the rule-based transformation and effectively reduces the search space of rule selection in our framework. When the constraints restrict the application of Rule #4 and Rule #5, they have a high success rate when applied in the optimization.

As the derived target path constraint in our example matches the prioritized preconditions in Rule #4 and Rule #5, our stochastic transformation applies them in Equation 3. With the same procedure of the imaginary-part addition and a few post procedures such as simplification and code generation, our framework optimizes the code to a stable fixed-precision program shown in Figure 4. When we run it in double precision with the same input $z_1 = e^{i\pi/4}$, $z_2 = -z_1 * e^{-i\varepsilon\pi}$ and $\varepsilon = 10^{-16}$, we get the correct output $z_3 = -0.7071 + 0.7071i$.

IV. Global Numerical Optimization

This section presents the technical details of our global numerical optimization framework. Figure 5 depicts its main workflow consists of 4 stages: 1) The symbolic trace extraction stage transforms the input program to the numerical IR that globally connects numerical operations in different statements together. 2) The instability analysis stage generates the unstable input regions that need to be optimized. 3) The stochastic algebraic transformation stage generates the optimized traces from the unstable traces in the numerical IR. 4) With a code generation stage, the verified optimized traces are merged together and finally translated to the optimized program. The rest of this section further describes the procedures in every stage separately.

A. Numerical IR & Symbolic Trace Extraction

Our numerical IR provides a global view of the input numerical program. It denotes every execution path with

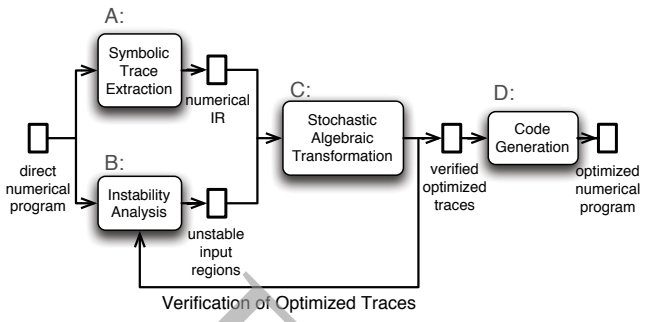


Fig. 5: Main Workflow of the Global Numerical Optimization Framework

numerical calculations in the input program by a numerical trace t, so syntactically the numerical IR is a set of traces. Every trace consists of a path constraint $t.c$, and a set of variable constraints $t.V$. A variable constraint is in the form of $v := e$, where e is often a long expression that records all the numerical updates of the output numerical variable v along the corresponding execution path.

For example, the numerical IR of Figure 3 should have two numerical traces when the code contains two execution paths. But for one of the paths (with the path constraint `abs(z1)!=1 || abs(z2)!=1`) does not contain any numerical calculation, our numerical IR just records the trace of the other path. Equation 3 depicts the information in the trace with the path constraint $\sqrt{\texttt{z1}.Re^2+\texttt{z1}.Im^2} == 1 \bigwedge \sqrt{\texttt{z2}.Re^2+\texttt{z2}.Im^2} == 1$ and two variable constraints that updates $\texttt{z3}.Re$ and $\texttt{z3}.Im$ separately.

Inspired by the symbolic execution technique [1], [2], [17], [18], [19], we present a symbolic trace extraction algorithm to generate the numerical IR. It replaces each input numerical value of the program with a symbol that initially represents "anything" (denoted by $\top$). Then it executes the program step by step, collects operations on the symbols, and generates constraints for the numerical IR.

Algorithm 1 depicts the symbolic trace extraction technique. At the entry of the program $\mathcal{P}$, our algorithm initializes an execution state es that holds all the input symbols (line 1). For the code in Figure 3, $\texttt{z1}.Re$, $\texttt{z1}.Im$, $\texttt{z2}.Re$, and $\texttt{z2}.Im$ are initialized as symbols when the code accepts complex numbers. For further propagating the execution state, every state es records 4 fields (ins, sym, out, c), where 1) $es.ins$ records the instruction that es is currently propagated in the input program, 2) $es.sym$ refers to a map that collects the corresponding variable constraints of all the symbols at the current step (such as $\{\texttt{z1}.Re \to \top, \texttt{z1}.Im \to \top, ...\}$ when es is just initialized), 3) $es.out$ records the output symbols (such as $\texttt{z3}.Re$, $\texttt{z3}.Im$), and 4) $es.c$ holds the current path constraint (such as `abs(z1)!=1 || abs(z2)!=1` when es is propagated to line 3 of Figure 3).

Algorithm 1 maintains a set of execution states ES (line 2), and processes it with a loop (lines 3-25). In each iteration, it propagates a step forward of a randomly selected execution state es (line 4) by processing its current instruction $es.ins$

Algorithm 1 Symbolic Trace Extraction

```
Input: P                                          //input numerical program
Output: IR                                        //syntactically a set of numerical traces
1:  es ← initialState(P); IR ← ∅;
2:  ES ← {es};                                    //ES is a state set
3:  while ES ≠ ∅ do
4:    es ← selectRandom(ES);                      //es is the current state
5:    es.ins.forward();
6:    switch es.ins.Type
7:    case v := exp ∧ v ∈ FP:                     //an update of v
8:      e ← symSubstitute(exp, es.sym);
9:      es.sym.update(v → e);
10:   case fork:                                  //such as the if statement
11:     {es1, es2}← forkExecution(es);
12:     ES ← ES - {es} ∪ {es1, es2};
13:   case OUTPUT(v) ∧ v ∈ FP:                    //such as printf(v)
14:     es.out ← es.out ∪ {v};
15:   case P.EXIT:
16:     new trace t (es.c, ∅);                    //initialize a new trace
17:     for all v ∈ es.out do
18:       t.V ← t.V ∪ {v := es.sym(v)};
19:     end for
20:     IR ← IR ∪ {t};
21:     ES ← ES - {es};
22:   default:                                    //same as default symbolic execution
23:     es.execInstruction(es.ins);
24:   end switch
25: end while
```

(lines 5-24). When the current instruction updates a numerical variable v with an expression exp (line 7), it substitutes the variable constraint in $es.sym$ for every symbol in the exp (line 8), and writes the substituted expression back to $es.sym$ as a new variable constraint of v (line 9). This operation connects multiple numerical updates in different statements together. For example, when the algorithm propagates an execution state es to line 7 of Figure 3 and unfolds the assignment of the complex number, it encounters a real-part update of `z3`. In this case, $es.ins$ is $\texttt{z3}.Re := \texttt{sum}.Re/\texttt{abs}(\texttt{sum}.Re, \texttt{sum}.Im)$. When es has already been propagated through lines 4-6 of Figure 3, $es.sym(\texttt{sum}.Re)$ is $\texttt{z1}.Re + \texttt{z2}.Re$, and $es.sym(\texttt{sum}.Im)$ is $\texttt{z1}.Im + \texttt{z2}.Im$. So the algorithm substitutes the variable constraints in $es.sym$ for the expression in $es.ins$, and further maps $\texttt{z3}.Re$ to $(\texttt{z1}.Re + \texttt{z2}.Re)/\texttt{abs}(\texttt{z1}.Re + \texttt{z2}.Re, \texttt{z1}.Im + \texttt{z2}.Im)$. After unfolding the subfunction `abs`, the expression will further be transformed to the form in Equation 3.

We fork the execution states (line 11 of Algorithm 1) to collect variable constraints in multiple execution paths, because every state just holds the constraints for one path. When es encounters a branch condition b (or a `fork` instruction), it generates new execution states es_1 and es_2 following both the `true` and `false` directions separately. es_1 and es_2 are the same as es except their path constraints, where $es_1.c$ is updated to $es.c \wedge b$, and $es_2.c$ is updated to $es.c \wedge \neg b$.

Lines 13-14 of Algorithm 1 collect the output variables, and lines 15-21 extract information from the execution state to build a trace in IR when the current execution state is propagated to an exit of the program. By default, the trace extraction algorithm executes the current instruction as the same way of symbolic execution [2], [17], which unfolds subprocedures in the program automatically (lines 22-23).

B. Instability Analysis in Heuristic

It is not necessary to optimize the program at the inputs that already make the program stable in fixed-precision arithmetic. So our framework integrates an instability analysis to distinguish the unstable input regions from its input space. Meanwhile, our framework also uses it to verify if an optimized transformation improves the accuracy of the program.

With a few trials on different instability analysis approaches, we integrate a sampling-based heuristic technique in our framework for the practicality. Because the framework keeps the infinite-precision calculations at the unknown input regions, our optimization is still sound even if the sampling-based technique misses some unstable inputs.

Our sampling-based instability analysis starts from 128 random samples uniformly from the floating-point bit patterns. It separately samples the sign bit, exponent bits, significand bits, and then combines them together. This sampling strategy generates very small values, very large values, and values of normal size with both positive and negative signs. When some inputs are empirically easy to trigger instabilities in numerical software, we also integrate an additional group of sample points with the following guided strategy: Let $\varepsilon = 5 \times 10^{-10}$ and $I = \{-e, -\pi, -2, -1, -\pi/2, 0, \pi/2, 1, 2, \pi, e\}$, $\forall i \in I$, we uniformly select 10 sample points in the range of $(i-\varepsilon, i+\varepsilon)$, including i. These sample points will be further refined with the feedback of program evaluation, which strongly improves the robustness of the instability analysis.

For every sample input, we evaluate the accuracy by executing it in both the fixed-precision and infinite-precision arithmetic (described in Section II), and mark it as a stable input if the accuracy is acceptable. Otherwise it is marked as a unstable input. Then our framework tries to build input regions by refining the sampling process. It resamples more points around the marked inputs to seek the boundary between the stable and unstable regions, and derives the constraint of regions heuristically in a template-matching way. In this paper, we represent the constraint of unstable input regions by c_u, the constraint of stable input regions by c_s, and other unknown input regions by c_{uk}. For example, when our framework performs instability analysis on Figure 3, it obtains a series of unstable inputs that matches a linear template $-\varepsilon < AX_1 + BX_2 + C < \varepsilon$, where $A = B = 1, C = 0$, $X_1 = \texttt{z1}.Re$, $X_2 = \texttt{z2}.Re$ as well as $X_1 = \texttt{z1}.Im$, $X_2 = \texttt{z2}.Im$. Our framework follows the template to generate $c_u = |\texttt{z1}.Re + \texttt{z2}.Re| < \varepsilon \wedge |\texttt{z1}.Im + \texttt{z2}.Im| < \varepsilon$, which is equivalent to the path constraint of the new optimized branch in Section III. If the inputs does not match any template, the input regions $i \in (i_1, i_2) \cup (i'_1, i'_2)$ can also build a constraint by themselves such as $(i_1 < i < i_2) \vee (i'_1 < i < i'_2)$.

Algorithm 2 Main Optimization Procedure

Input: IR_{in} //input IR
Output: IR_{opt} //optimized IR
1: $\mathit{IR}_{opt} \leftarrow \{t_s\}$; //initialized as the stable input trace
2: **for every** $t \in \mathit{IR}_{in} \bigwedge(t.c \wedge c_u \neq \text{false})$ **do**
3: **for every** $v := e \in t.V$ **do**
4: $E \leftarrow \{e\}$; //build an equivalent set
5: **do**
6: $e \leftarrow$ selectRandom(E);
7: $\langle e', c', rule \rangle \leftarrow$ stochasticTransform(e, $t.c \wedge c_u$);
8: **if** stableVerification(e',c') **then**
9: **new trace** t' (c', $\{v := e'\}$); //optimized trace
10: $rule$.succ++; //increase the success count
11: **else**
12: $E \leftarrow E \cup \{e'\}$;
13: **end if**
14: **while** $t' =$ NIL $\wedge$!TIME_OUT
15: $\mathit{IR}_{opt} \leftarrow \mathit{IR}_{opt} \cup \{t'\}$; //update optimized traces
16: $\mathit{IR}_{in} \leftarrow \mathit{IR}_{in} - \{t\} \cup (t.c \wedge \neg c', t.V)$;
17: **end for**
18: **end for**

Algorithm 3 Stochastic Algebraic Transformation

Input: e, c
Output: $\langle e', c', rule \rangle$
1: $R \leftarrow \{rule | (\exists e_0 \in \text{SubExp}(e), e_0.\text{match}(rule.i)) \bigwedge(rule.cp \wedge c \neq \text{false})\}$ //a rule candidate set
2: **for every** $r \in R$ **do**
3: **if** $r.pp \wedge c \neq$ false **then** //update the priority
4: prior(r) $\leftarrow r$.succ $\times$ pfactor;
5: **else**
6: prior(r) $\leftarrow r$.succ;
7: **end if**
8: **end for**
9: $rule \leftarrow$ prioritizedRandom(R, prior); //select the rule
10: **return** $\langle$simplify($rule$.apply(e)), $rule.cp \wedge c$, $rule\rangle$;

The stable input regions create a optimized trace t_s natively, where $t_s.c$ is c_s, $t_s.V$ is the original updates in the input program with the fixed-precision arithmetic.

C. Stochastic Algebraic Transformation

Our framework transforms the global variable constraints in the numerical IR with a rule-based manner. A $rule$ in our framework is specified as a 5-tuple $(cp, pp, i, o, \text{succ})$, where cp represents a correctness precondition, pp represents a prioritized precondition, i denotes its input pattern, o denotes its output pattern, and `succ` is the success count of finding stable optimizations when applying the current rule, which is initialized by 1.

If a subexpression e_0 of the global constraint expression e matches the input pattern of a rule $rule.i$ (line 1 of Algorithm 3), our framework applies the rule on e (line 10 of Algorithm 3) by substituting the corresponding output pattern $rule.o$ for e_0 in the constraint expression. For example, the

rule $A + B \rightsquigarrow B + A$ performs commutative law of addition when A and B matches any subexpressions.

Some complicated transformations are only valid under conditions, which is defined by the correctness preconditions. For example, Rule #4 in Table I is only valid when $R_1^2+I_1^2 == R_2^2+I_2^2$ is `true`. So we disjunct it $rule.cp$ with the input path condition c (line 10 of Algorithm 3) to generate the optimized path constraint when applying the corresponding rule.

The insight of a prioritized precondition pp in a rule is a floating-point domain knowledge that the rule should be especially effective under such condition. For example, $(A + B) + C \rightsquigarrow A + (B + C)$ is always correct in real arithmetic, but it will improve the accuracy of floating-point arithmetic only when $A > B \wedge A > C$. Hence, we put it as a prioritized precondition of the rule, and significantly increase the chance of choosing the rule by multiply its priority with a factor `pfactor` (line 4 of Algorithm 3) when the precondition is satisfied. The default value of `pfactor` is 100.

The default priority of selecting a $rule$ in our framework is its success count $rule.\texttt{succ}$. When it is initialized as 1, every rule has a chance to be selected in the transformation. Higher success count means the rule is empirically more useful in the transformation, which leads to more chances to be selected. When a number of rules $r_1, r_2, r_3...r_n$ are prepared for being applied on the current global constraint expression, the probability of selecting a rule r_i (line 9 of Algorithm 3) is: $\text{prior}(r_i)/\sum_{x=1}^{n}\text{prior}(r_x)$.

Our framework integrates an extensible rule database, which contains 258 rules in the current version. The database includes both the simple transformations such as the commutative, associative, distributive laws, and several complicated rules such as the complex number transformations and the facts of trigonometry, exponents, logarithms and gamma functions. The database can be extended with more rules to support other types of floating-point transformation. With the help of prioritized preconditions, numerical experts are easy to express the domain knowledge of floating-point arithmetic in the database.

Algorithm 2 depicts the main optimization process that transfers the input numerical IR to the optimized IR, while Algorithm 3 defines the function `stochasticTransform` that is called at line 7 of Algorithm 2 to perform a stochastic transformation on a global constraint expression. Algorithm 2 initializes the optimized IR with the native stable trace t_s (line 1), and tries to optimize every trace in the input IR. For every global variable constraint in the trace, it builds an set of expressions E that are equivalent to the original expression e in real arithmetic (line 4), and seeks the stable optimization of e by stochastically transforming a form in E (line 7). When the transformed expression e' is verified to be stable under its optimized path constraint c' with infinite-precision arithmetic (line 9), it creates an verified optimized trace t' and adds it into the optimized IR (line 15). It also removes the corresponding optimized trace from the input IR for further optimizing other traces (line 16).

D. Code Generation & Post Analysis

The final stage of our framework generates the output program from the optimized IR. It translates every trace t in the optimized IR as a branch in the output program with fixed-precision arithmetic, which sets the branch condition as $t.c$ and updates every target variable with the optimized procedure in the trace. Furthermore, our framework also keeps the unoptimized traces in $\mathbb{IR}_{in}$ at the end of the output program with infinite-precision arithmetic, which ensures the soundness of the optimized program.

Our framework also integrates a post analysis to reduce code clones in the output program. It merges traces with the same path constraint or variable constraints. If parts of variable constraints are the same with another trace, the post analysis pushes down the branch condition to make the output program short and easy to read. With the post analysis, our framework generates clean outputs such as the code in Figure 4. Since the post analysis does not affect the soundness and efficiency of the output program, it is an optional module in the optimization framework.

V. Evaluation and Results

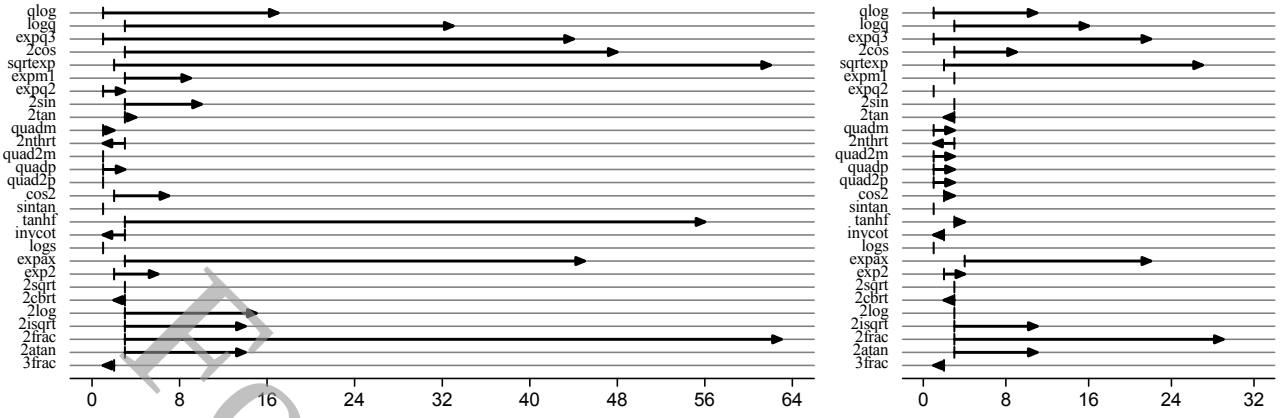
We implement our optimization framework[1] in a loosely coupled manner with a front-end that converts the direct numerical program to the numerical intermediate representation (IR), and a back-end that performs the stochastic algebraic transformation and generates the optimized program. In our front-end, we analyze the structures of the input program with the ROSE compiler [20], and implement our symbolic trace extraction module based on KLEE [2]. We implement our back-end with the `python` language based on the SymPy library. We parse the syntax of every rule with ANTLR4 and evaluate the infinite-precision program with the latest version of iRRAM [12] library.

We conduct several experiments to evaluate our optimization framework, and intend to answer the following key research questions:

- **RQ1:** Does our optimization achieve higher floating-point accuracy when compared with the state-of-the-art numerical optimization tool?
- **RQ2:** Can our framework globally optimize numerical software with complicated program structures (such as loops)?
- **RQ3:** Is our optimization framework helpful for real-world software?

We get several observations in our evaluation: For **RQ1**, our optimization significantly improves the worst case floating-point accuracy of numerical expressions when compared with the state-of-the-art numerical optimization tool. For **RQ2**, our framework is effective in optimizing numerical software with complicated program structures, which are challenging for numerical optimization. For **RQ3**, our framework detects and provides fixing advices of numerical bugs in real-world

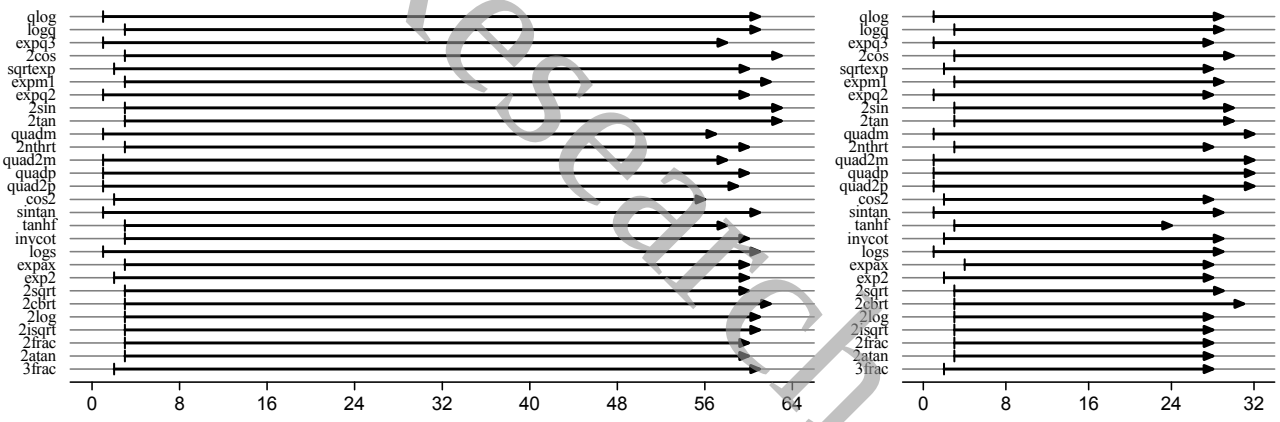
[1]The framework is available at http://seg.nju.edu.cn/eytang/numopt.



(a) Double Precision

(b) Single Precision

Fig. 6: Observed Worst-Case Bits Corrected by Herbie (Longer Arrow to the Right is Better)
Every row shows the improvement in bit-error achieved by Herbie on the worst input of 4096 random input points for a single benchmark. The thick arrow points from the accuracy of the fixed-precision input program to the accuracy of Herbie's output. Accuracy is measured by the number of correct bits when comparing the value to infinite-precision arithmetic.



(a) Double Precision

(b) Single Precision

Fig. 7: Observed Worst-Case Bits Corrected by Our Optimization Framework (Longer Arrow to the Right is Better)
Every row shows the improvement in bit-error achieved by Our Optimization on the worst input of 4096 random input points for a single benchmark. The thick arrow begins at the same accuracy of the input program as Figure 6, and ends at the accuracy of the output of our optimization.

software, which has been confirmed by developers of open source projects. The rest of this section depicts more details in every experiment.

A. RQ1: Accuracy Improvement on Local Optimization

Herbie[2] is the state-of-the-art numerical transformation tool. Since it mainly aims to optimize local numerical expressions, authors of Herbie have evaluated it [9] on 28 classic benchmarks from Hamming's *Numerical Methods for Scientists and Engineers* [21]. The results show that Herbie is good at improving average accuracy for various numerical expressions. To avoid a bias in our experiment, we also conduct our evaluation on these benchmarks[3] with the same experiment parameters except the following setups:

Instead of using average accuracy, we make our evaluation more rigorous. That is, we evaluate our optimization with the worst accuracy of all the output values for a numerical program. If a system has a high worst case accuracy, it must also have a high average accuracy, but the reverse is not true. As we can only get a random subset of the numerical outputs, we call our measurement the *observed worst case accuracy*. In our point of view, the worst case accuracy is important because it

[2]https://github.com/uwplse/herbie

[3]When our optimization framework accepts input programs instead of numerical expressions, we capsulate every numerical expression in the benchmark with a `main` function.

TABLE II: Specification of the Infinite-precision Benchmark Programs for our Global Optimization

Program	Specification
analytic	evaluate $1/2^{20}$ with a simple iterative algorithm
e_example	calculate Euler's number e≈2.718... with $\sum_{i=0}^{n} 1/i!$
float_ext	evaluate $2+\sum_{i=1}^{100,000} 1/\sqrt{i}$
gamma	generate the Euler-Mascheroni constant γ≈0.577... with Stirling's approximation
harmonic	evaluate the sum of the first 5E+07 terms of the harmonic series
itsyst	iteratively evaluate $x_{i+1} = 3.75x_i(1-x_i)$ with different inputs and calculation orders.
jmmuler	iteratively evaluate $x_{i+2}=3000/(1130-x_i(111-x_{i+1}))$ with different inputs and calculation orders.
lambov	calculate the remainder of taylor series

is very dangerous for a safety-critical numerical software to have only a small number of inaccurate outputs, which will still cause system failures and serious consequences.

Since both Herbie's and our optimization are based on probabilistic mechanism, we repeat 10 times of both the tools for every benchmark and show the result of the worst optimization in Figure 6 and Figure 7. Hence, the results in these figures describe the worst situation for users when they use the optimizations. From the results, our optimization framework has a significant advantage in improving the worst-case accuracy. Such observation is for several reasons: 1) Our transformation strategies for global optimization also fits in finding better transformation rules for the local optimization. 2) Different from Herbie that often infers inaccurate path constraints with a regime algorithm on optimized expressions, our framework derives the constraints directly from the input program and further refines the constraints by the rich context in our optimization rule.

B. RQ2: Effectiveness on Global Optimization

The iRRAM library attaches a group of infinite-precision test programs. After omitting the trivial example programs such as the one just transferring string inputs to the infinite-precision values, we collect 8 of these infinite-precision programs that contain complicated program structures as our benchmark to evaluate our global optimization. All these programs contain at least a loop to calculate the numerical results, some of them introduce complicated iterative refinement algorithms such as Stirling's approximation. Table II specifies these benchmarks that involve various aspects of numerical calculations.

Our framework optimizes the benchmarks to double-precision programs. Figure 8 depicts the observed worst case accuracy improvement from the direct double-precision programs with the original algorithms to our optimized programs. The experiment setup is the same as Figure 7. When `analytic` and `e_example` are stable in their original algorithms, our framework cannot further improve their accuracies. In

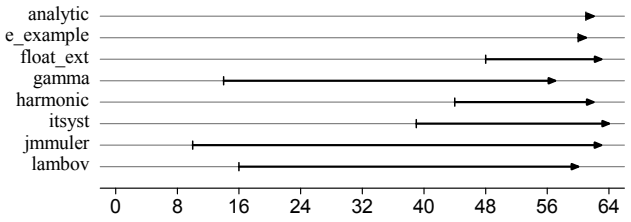


Fig. 8: Observed Worst-Case Bits Corrected by our Global Optimization (Longer Arrow to the Right is Better)

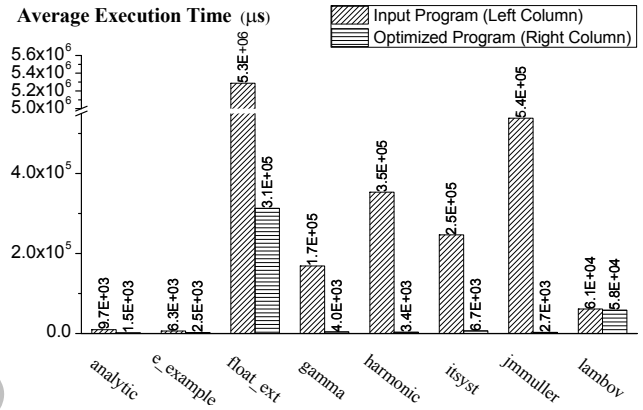


Fig. 9: Average Execution Time of the Input Benchmark Programs and the Corresponding Optimized Programs

fact, our framework rewrites `e_example` from $\sum_{i=0}^{n} 1/i!$ to $\sum_{i=n}^{0} 1/i!$ in order to alleviate the annihilation of adding a big number to small numbers. But the accuracy is not improved in the optimized program because the factorial result $i!$ grows too fast that make $\sum_{i=n}^{i'} 1/i!$ still very small. For other cases, our framework yields useful transformations that obviously improve the accuracies of the optimized programs.

Figure 9 shows the average execution time of the input programs and their corresponding optimized programs. When we optimize the benchmark from the infinite-precision arithmetic to the double-precision arithmetic, most cases have a significant speed-up. The benchmark `lambov` just has a small speed-up from 61 milliseconds to 58 milliseconds because the optimized program still relies on some calculations in the infinite-precision arithmetic of this case.

C. RQ3: Case Study on Real-World Programs

We make a prospect that numerical developers just need to write easy-maintained infinite-precision programs, and implement the framework to generate the corresponding fixed-precision optimizations. However, many current real-world programs are still written directly in fixed-precision floating-point arithmetic with numerical instabilities. Our framework can also help these programs to detect and fix the numerical bugs. This section describes two cases that our framework

improves the accuracy for an open source graphics library `Clipper`, and a driving tool `Slic3r` for three-dimensional (3D) printers.

For every real-world project, we replace the `float` and `double` types in its source code with the `REAL` type in the iRRAM library, and introduce a pre-defined header to make the project look like an infinite-precision program. Then we input the project into our framework at the function level and check if the optimization fixes numerical bugs.

Our framework detects and provides fixing advices of two numerical bugs in the `Clipper` library. One of them is caused by the cumulative errors when the library calculates the area of a polygon, and the other bug is caused by a massive cancellation when the library computes the distance from a point to a line. We report both of the bugs[4] and get the confirmation email from its developer. Our framework also detects a numerical instability in `Slic3r` when it tries to lift the extruder of a 3D printer, and the bug is also reported[5] and get confirmed.

VI. Related Work

Improving the correctness and efficiency for numerical software is a popular topic with a long research history. Numerical experts have proposed a great number of theoretical approaches and technical tricks to help developers write stable programs [3], [22], [23], [4], [5]. In this section, we mainly survey some closely related recent work on numerical optimization and analysis.

Numerical Accuracy Improvement A few recent researches focus on improving the accuracy of floating-point expressions by rewriting them to the forms with smaller rounding errors. Martel presents an abstract semantics [24], [25], which defines a numerical transformation on a set of abstract operations. It does not support large database of rules in numerical rewriting because the technique is bound to a brute-force search, which limits the program transformations that can be found. Tang et al. propose expression perturbation [26], which rewrites numerical expressions in a program with commutative, associative, and distributive laws for finding a form with higher accuracy. Panchekha et al. present Herbie [9], a state-of-the-art numerical transformation tool that rewrites input expressions with a database of transformation rules and a heuristic estimation that localizes rounding errors in the expression with dynamic sampling. Sanchez-Stern et al. further combines Herbie with Valgrind (named Herbgrind) [27], which dynamically detects and reduces numerical error at a significant part (which is called the root cause part in their paper) in the program.

Speedup of Floating-point Programs Schkufza et al. implement an aggressive optimization of floating-point computations as an extension to STOKE [15], which generates reduced precision implementations of numerical binaries with a Markov chain Monte Carlo(MCMC) sampling. Lam et al. [28] present a framework that builds mixed-precision configurations of existing double-precision binaries with identification of code regions that can use lower precision. Rubio-González et al. present Precimonious [29], a dynamic program analysis tool that decreases the precision of intermediate numerical values (precision tuning) to speed up floating-point calculations. They further improve the efficiency of precision tuning with Blame Analysis [30] and a scalable hierarchical search that exploits the community structure of floating-point variables [31]. None of these techniques concern about transforming the numerical program to a form with higher accuracy.

Numerical Verification and Error Detection Verification of a numerical program is difficult [32] when it does not always adhere to the IEEE 754 standard [10]. Darulova and Kuncak [33] tracks the guaranteed range of floating point values in a program. Franco et. al conduct an empirical study of numerical bugs from 5 widely-used numerical libraries [34]. Goubault et al. [35], [36], [6] track the error of floating-point operations, algorithms, and computations with abstract interpretation. Barr et al. [37] detect floating-point overflows and underflows with an SMT solver. Then Darulova et al. [7] also use an SMT solver to prove error bounds in numerical computation. Benz et al. present FPDebug [8], which find numerical accuracy problems with a dynamic testing in higher precision. Chiang et al. [38] develop a heuristic search algorithm to generate test inputs that cause significant floating-point inaccuracies. Zou et al. [39] further propose a genetic algorithm to find the inaccurate inputs. Bao et al. [11] propose a technique to detect the floating-point error inflation that causes different execution paths. Later they integrate the technique in RAIVE [40], a vectorized executor. Tang et al. [41] present a framework to detect and diagnose numerical instabilities automatically in software. None of the above techniques optimize floating point computations in a program.

VII. Conclusion

This paper presents a global optimization framework that helps numerical developers to obtain high-precision, easy-to-maintain, and efficient numerical software. Using our framework, a developer simply writes the infinite-precision numerical program directly following the problem's mathematical requirement specification. Our framework then optimizes the input program in a global fashion, which analyzes the program's numerical value flows across different statements through a symbolic trace extraction algorithm, and generates optimized traces via stochastic algebraic transformations guided by effective rule selection.

Acknowledgment

This research is supported by National Key R&D Program of China (Grant No. 2017YFB1001801) and National Natural Science Foundation of China (Grant No. 61772260, 61632015, 61561146394, and 61402222). Zhendong Su was supported in part by United States National Science Foundation Grants 1528133 and 1618158.

[4]Bugs #180 and #184 at https://sourceforge.net/p/polyclipping/bugs/

[5]https://github.com/slic3r/Slic3r/issues/4497

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[2] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[3] N. Higham, *Accuracy and stability of numerical algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, 2002.

[4] W. Miller and D. Spooner, "Software for roundoff analysis, ii," *ACM Transactions on Mathematical Software*, vol. 4, no. 4, pp. 369–387, 1978.

[5] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Dover, 1994.

[6] E. Goubault and S. Putot, "Static analysis of finite precision computations," in *12th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011, pp. 232–247.

[7] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014, pp. 235–248.

[8] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 453–462.

[9] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 1–11.

[10] American National Standards Institute, *IEEE standard for binary floating-point arithmetic*, Std., 1985.

[11] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013, pp. 817–832.

[12] N. Müller, "The iRRAM: exact arithmetic in C++," in *Computability and Complexity in Analysis*, ser. Lecture Notes in Computer Science, vol. 2064, 2001, pp. 222–252.

[13] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, Jun. 2007.

[14] N. Toronto and J. McCarthy, "Practically accurate floating-point math," *Computing in Science Engineering*, vol. 16, no. 4, pp. 80–95, 2014.

[15] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 53–64.

[16] C.-H. Guo, N. J. Higham, and F. Tisseur, "An improved arc algorithm for detecting definite Hermitian pairs," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 1131–1151, Sep. 2009.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006, pp. 322–335.

[18] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[19] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 19–32.

[20] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, pp. 215–226, 2000.

[21] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed. New York: Dover, 1987.

[22] W. Miller, "Toward mechanical verification of properties of roundoff error propagation," in *Proceedings of the ACM Symposium on Theory of Computing*, 1973, pp. 50–58.

[23] W. Miller, "Software for roundoff analysis," *ACM Transactions on Mathematical Software*, vol. 1, no. 2, pp. 108–128, 1975.

[24] M. Martel, "Semantics-Based transformation of arithmetic expressions," in *Static Analysis Symposium*, 2007, pp. 298–314.

[25] M. Martel, "Program transformation for numerical precision," in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2009, pp. 101–110.

[26] E. Tang, E. Barr, X. Li, and Z. Su, "Perturbing numerical calculations for statistical analysis of floating-point program (in)stability," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 131–142.

[27] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 256–269.

[28] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013, pp. 369–378.

[29] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 27:1–27:12.

[30] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1074–1085.

[31] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 333–343.

[32] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 12:1–12:41, May 2008.

[33] E. Darulova and V. Kuncak, "Trustworthy numerical computation in Scala," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2011, pp. 325–344.

[34] A. Di Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 509–519.

[35] E. Goubault, "Static analyses of the precision of floating-point operations," in *Proceedings of the 8th International Static Analysis Symposium*, 2001, pp. 234–259.

[36] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *Proceedings of the 13th International Static Analysis Symposium*, 2006, pp. 18–34.

[37] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013, pp. 549–560.

[38] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 43–52.

[39] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *Proceedings of 37th International Conference on Software Engineering*, 2015, pp. 20–22.

[40] W.-C. Lee, T. Bao, Y. Zheng, X. Zhang, K. Vora, and R. Gupta, "RAIVE: Runtime assessment of floating-point instability by vectorization," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 623–638.

[41] E. Tang, X. Zhang, N. T. Müller, Z. Chen, and X. Li, "Software numerical instability detection and diagnosis by combining stochastic and infinite-precision testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 975–994, Oct. 2017.