FPCC: Detecting Floating-Point Errors via Chain Conditions

XIN YI, National University of Defense Technology, China
HENGBIAO YU*, National University of Defense Technology, China
LIQIAN CHEN, National University of Defense Technology, China
XIAOGUANG MAO, National University of Defense Technology, China
JI WANG, National University of Defense Technology, China

Floating-point arithmetic is notorious for its rounding errors, which can propagate and accumulate, leading to unacceptable results. Detecting inputs that can trigger significant floating-point errors is crucial for enhancing the reliability of numerical programs. Existing methods for generating error-triggering inputs often rely on costly shadow executions that involve high-precision computations or suffer from false positives. This paper introduces chain conditions to capture the propagation and accumulation of floating-point errors, using them to guide the search for error-triggering inputs. We have implemented a tool named FPCC and evaluated it on 88 functions from the GNU Scientific Library, as well as 21 functions with multiple inputs from previous research. The experimental results demonstrate the effectiveness and efficiency of our approach: (1) FPCC achieves 100% accuracy in detecting significant errors for the reported rank-1 inputs, while 72.69% rank-1 inputs from the state-of-the-art tool ATOMU can trigger significant errors. Overall, 99.64% (1049/1053) of the inputs reported by FPCC can trigger significant errors, whereas only 19.45% (141/723) of the inputs reported by ATOMU can trigger significant errors; (2) FPCC exhibits a 2.17x speedup over ATOMU in detecting significant errors; (3) FPCC also excels in supporting functions with multiple inputs, outperforming the state-of-the-art technique. To facilitate further research in the community, we have made FPCC available on GitHub at https://github.com/DataReportRe/FPCC.

CCS Concepts: • **Software and its engineering → General programming languages**; **Software testing and debugging**.

Additional Key Words and Phrases: chain condition, floating-point error, error-triggering input, accuracy

ACM Reference Format:
Xin Yi, Hengbiao Yu, Liqian Chen, Xiaoguang Mao, and Ji Wang. 2024. FPCC: Detecting Floating-Point Errors via Chain Conditions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 324 (October 2024), 28 pages. https://doi.org/10.1145/3689764

1 Introduction

Floating-point numbers have become the *de facto* standard for representing real numbers in modern computers. Since floating-point numbers use a finite number of bits to represent real

*Corresponding author

Authors' Contact Information: Xin Yi, College of Computer, National University of Defense Technology, Changsha, China, yixin09@nudt.edu.cn; Hengbiao Yu, College of Computer, National University of Defense Technology, Changsha, China, hengbiaoyu@nudt.edu.cn; Liqian Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, Changsha, China, lqchen@nudt.edu.cn; Xiaoguang Mao, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, Changsha, China, xgmao@nudt.edu.cn; Ji Wang, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, Changsha, China, wj@nudt.edu.cn.

ACM 2475-1421/2024/10-ART324
https://doi.org/10.1145/3689764

values, inaccuracy is unavoidable in floating-point computations. In fact, the rounding errors that occur during these calculations can propagate and accumulate, leading to final results that may be unacceptable. Particularly concerning are the inaccuracies that arise in safety-critical software systems, which have resulted in catastrophic outcomes, such as the loss of human lives [Skeel 1992], the failure of rocket launches [Wikipedia 2024], and disruptions in the stock market [Quinn 1983].

Detecting inputs that can trigger large floating-point errors is crucial for assessing the accuracy of numerical programs. In addition to testing, these error-triggering inputs can help identify the root causes of precision loss [Guo et al. 2020; Sawaya et al. 2017], enhance accuracy [Panchekha et al. 2015; Zou et al. 2022], and optimize precision configurations [Guo and Rubio-González 2018; Menon et al. 2018]. Unfortunately, considering only a small portion of the huge input space can trigger large floating-point errors [Bao and Zhang 2013] and understanding the propagation of rounding errors is often counterintuitive. Consequently, generating error-triggering inputs that can trigger large floating-point errors remains a significant challenge.

Several approaches [Chiang et al. 2014; Guo and Rubio-González 2020; Yi et al. 2017, 2019; Zou et al. 2015] have been proposed to detect inputs that trigger large floating-point errors. These methods identify error-triggering inputs through heuristic searches, such as binary guided search [Chiang et al. 2014], condition number guided search [Yi et al. 2017, 2019], and genetic algorithm-based search [Zou et al. 2015]. FPGen [Guo and Rubio-González 2020] reduces the problem of generating error-triggering inputs to a code coverage problem and solves it through symbolic execution. However, these approaches heavily rely on oracles generated by expensive high-precision computations, which require expert knowledge to manage *precision-specific operations* and significantly increase computational overhead. ATOMU [Zou et al. 2019] conducts atomic condition-guided searches for error-triggering inputs without relying on high-precision computations (*a.k.a* oracle-free). Its strength lies in efficiently pinpointing inputs that trigger significant floating-point errors. However, ATOMU's focus on individual operations often leads to false positives, limiting its overall effectiveness in identifying true error-triggering inputs. For the given function foo in Fig. 1, ATOMU reports an error-triggering input "x=1.7320508075688770". This is because the input can trigger a large value of atomic condition ($\left|\frac{a}{1-a}\right|$ =*"3.0023997515803315e+15"*) in the unstable subtraction $1.0 - a$. The calculation "a = 3.0/(x*x)" results in "a = 1.0000000000000004" and the subtraction "(1.0 - a) = -4.440892098500626e-16" introduce a absolute error around *7.1819e-17* which is relatively large compared to *-4.440892098500626e-16*. However, the absolute error *7.1819e-17* is less than the machine epsilon (around *2.22e-16*) of 1.0 and is suppressed by the last operation "b + 1.0", resulting in a very small final relative error. Therefore, ATOMU reports a false positive input. Our experimental results (see §5) also demonstrate that only 19.45% of the inputs reported by ATOMU can actually trigger significant floating-point errors.

```
double foo(double x){
  double a  = 3.0/(x*x);
  double b = fabs(1.0-a);
  return b + 1.0;
}
```

Fig. 1. An example of atomic condition induced false positive.

As a step toward addressing these challenges, we propose a *chain condition* guided *oracle-free* search to detect error-triggering inputs. Our key insight is that a numerical program can be regarded as a sequence of arithmetic operations and the interactions of consecutive condition numbers capture the propagation of floating-point errors. Unlike atomic condition which only considers the condition number of an individual floating-point operation, *chain condition* focuses on the transfer flow of condition numbers. Actually, chain condition reflects how the input error is amplified by a sequence of floating-point operations. It's worth pointing out that chain condition deals successfully with the false positives introduced by atomic condition. Let's revisit the function foo. For the same given input x, the chain condition of the return value is 5.0 (according to formulas in §4.2). This indicates a small value, suggesting that the result does not contain significant errors. Consequently, the chain condition helps prevent false positives.

The main contributions of this paper are following:

- We introduce the concept of *chain condition* and demonstrate its utility in capturing the propagation of floating-point errors. (See §4.2).
- We propose a chain condition guided search to detect inputs triggering large floating-point errors and a error localization algorithm to localize the source code responsible for floating-point errors. (See §4.3 and §4.4).
- We implement our approach in a tool named FPCC based on the insight of chain condition and evaluate it on 88 functions from GNU Scientific Library. Compared to the state-of-the-art approach ATOMU, FPCC tool identifies all rank-1 inputs that can cause significant floating-point errors, in contrast to ATOMU which fails to detect errors in 27% of its rank-1 inputs. Moreover, FPCC reports a higher percentage (99.64%) 1049 of 1053 inputs that can trigger significant errors compared to ATOMU (19.45%) 141 of 723 inputs. Additionally, FPCC achieves a 2.17X speedup over ATOMU and FPCC excels in supporting multiple inputs functions, outperforming FPGen[Guo and Rubio-González 2020] (See §5).

To facilitate follow-up research in the community, we make FPCC and all relevant artifacts publicly available at https://github.com/DataReportRe/FPCC. We have also created a Docker image and user-friendly scripts to make it convenient for users to replicate our experiments.

The rest of this paper is organized as follows. Section 2 gives a brief background on floating-point arithmetic. Section 3 illustrates FPCC on a representative example. Section 4 details the methodology of our approach. Section 5 presents the implementation and evaluation of our approach. We discuss related work in Section 7 and conclude in Section 8.

2 Background

This section presents the background on floating-point numbers, including floating-point format, error measurement, and condition number.

2.1 Floating-Point Format

Modern computers use floating-point numbers to represent real numbers in an approximate manner. With the development of new processors and accelerators, many new floating-point representations have been proposed, such as Intel's Bfloat16 and Nvidia's TensorFloat32. Nevertheless, IEEE 754 standard [IEEE-754 2008] still remains as the most popular floating-point representation. In IEEE 754, a floating-point number can be represented as below.

$$(-1)^S \times M \times 2^E$$

A floating-point number consists of three parts: a 1-bit sign S ($S \in \{0, 1\}$), a n-bit significand M ($M = m_0.m_1m_2 \ldots m_n$, m_0 is the hidden bit), and a p-bit biased exponent E ($E = e - (2^{p-1} - 1)$). Notably, when all the exponent bits are one, the floating-point representation denotes special values, *i.e.*, $\pm\infty$ or NaN (not a number).

2.2 Floating-Point Error Measurement

Since floating-point numbers use finite bits to represent real numbers, only a subset of reals can be represented. When a real number cannot be precisely encoded, IEEE 754 standard provides different rounding modes (such as `roundNearestTiesToEven`) to convert a real number to a nearby floating-point number. Let $\mathcal{R}$ and $\mathcal{F}$ represent real numbers and floating-point numbers, respectively. Given a real number $x \in \mathcal{R}$ and let $x_f \in \mathcal{F}$ denote its floating-point representation, the rounding error is $x - x_f$.

Absolute error and relative error are two common indicators to measure floating-point error. Given a floating-point program f and an input x, we use $\hat{f}(x)$ to represent the ideal mathematical result of program f. The absolute error $Err_{abs}(f(x), \hat{f}(x))$ and the relative error $Err_{rel}(f(x), \hat{f}(x))$ are defined below.

$$Err_{abs}(f(x), \hat{f}(x)) = \left|\hat{f}(x) - f(x)\right| \qquad Err_{rel}(f(x), \hat{f}(x)) = \left|\frac{\hat{f}(x) - f(x)}{\hat{f}(x)}\right| \tag{1}$$

Unit in the last Place (*a.k.a* ULP) of a floating-point number x is the distance between the two consecutive floating-point numbers nearest to x. ULP error has been widely used to measure the floating-point error and can be defined as below.

$$Err_{ulp}(f(x), \hat{f}(x)) = \left|\frac{\hat{f}(x) - f(x)}{ulp(\hat{f}(x))}\right| \tag{2}$$

2.3 Condition Number

A function's condition number [Higham 2002] measures its sensitivity to small perturbations of the input. Condition number depends on the function's functionality and input, but is independent of the implementation.

Suppose input x carries a small error Δx, according to the Taylor series, we have

$$\begin{aligned} f(x + \Delta x) &= f(x) + f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 + \dots \\ &\approx f(x) + f'(x)\Delta x \end{aligned} \tag{3}$$

Based on Equation 3, it's obvious that $f(x+\Delta x) - f(x) \approx f'(x)\Delta x$. Then, the relative error between $f(x + \Delta x)$ and $f(x)$ can be defined as

$$\begin{aligned} \left|\frac{f(x + \Delta x) - f(x)}{f(x)}\right| &\approx \left|\frac{f'(x)\Delta x}{f(x)}\right| \\ &\approx \left|\frac{xf'(x)}{f(x)}\right| \cdot \left|\frac{\Delta x}{x}\right| \end{aligned} \tag{4}$$

The definition of condition number is given below. Apparently, condition number reflects how much the input relative error $|\Delta x/x|$ is amplified in the relative error of the output.

$$C_f(x) = \left|\frac{xf'(x)}{f(x)}\right| \tag{5}$$

Given a floating-point program f, it's not easy to obtain $f'(x)$ unless its mathematical functionality is pre-given. ATOMU [Zou et al. 2019] introduces atomic condition, which corresponds to the condition number of an atomic floating-point operation. An atomic operation could be elementary arithmetic ($+, -, \times, \div$) or basic functions (such as `sin` and `cos`) and its atomic condition evaluates its stability. ATOMU utilizes atomic conditions to guide the search for error-triggering inputs.

For example, given an operation $op(x, y) = x + y$, $C_+(x) = \left|\frac{x}{x+y}\right|$ and $C_+(y) = \left|\frac{y}{x+y}\right|$. It's obvious that the operation's atomic condition could be very large if $x \approx -y$. In other words, operation $x + y$ could introduce large floating-point error when $x \approx -y$, which agrees with the notorious catastrophic cancellation.

3 Example

In this section, we first present a code snippet of a real-world numerical program from the GNU Scientific Library. Through this example, we demonstrate how atomic operations can potentially introduce false positives and highlight the effectiveness of using chain conditions to address this issue.

Program $\mathcal{P}_1(x)$ shown in Fig. 2 is from the GSL function *gsl_sf_airy_Ai* (line 680 in gsl/specfunc/airy.c), where a and b come from previous Chebyshev polynomial evaluation and are regarded as constants. Program $\mathcal{P}_2(x)$ is a mutant of $\mathcal{P}_1(x)$, *i.e.*, replacing the last floating-point addition in $\mathcal{P}_1(x)$ with a multiplication operation. It is worth noting that the specific value of x used in this example is *-7.7274027910331625e-2*. ATOMU has identified this value as the most likely input that triggers a significant floating-point error (i.e. the relative error is larger than 1e-3).

Each program can be regarded as a sequence of five floating-point operations. Fig. 2 shows the operands, chain conditions, atomic conditions, operation results, and relative errors of the five operations. The atomic conditions are calculated by ATOMU [Zou et al. 2019], while chain conditions are calculated using the transition rules defined in Fig. 4.

Example program 1: $\mathcal{P}_1(x) = 0.375 + (a - x * (0.25 + b))$
Example program 2: $\mathcal{P}_2(x) = 0.375 * (a - x * (0.25 + b))$

$P_1(x)$	Operand(s)	Chain condition	Atomic condition	Operation result	Relative error
v1 = 0.25+b	0.25, 0.008809451767620686893 4	$C_+(0.25) = 0.966$, $C_+(b) = 0.034$, $CC(v1) = 1.0$	1.0	0.25880945176762071291	1.01e-16
v2 = x*v1	-0.077274027910331625, 0.25880945176762071291	$C_*(x) = 1.0$, $CC(v1) * C_*(v1) = 1.0$, $CC(v2) = 2.0$	2.0	-0.019999248799348751104	1.86e-16
v3 = a-v2	-0.019999248799348758043, -0.019999248799348751104	$C_-(a) = 2.88e+15$, $CC(v2) * C_-(v2) = 5.76e+15$, $CC(v3) = 8.64e+15$	5.76e+15	-6.9388939039072283776e-18	3.49e-1
v4 = 0.375+v3	0.375, -6.9388939039072283776e-18	$CC_-(0.375) = 1.0$, $CC(v3) * C_-(v3) = 0.16$ $CC(v4) = 1.16$	1.0	0.375	2.84e-17
return v4					
$P_2(x)$					
v5 = 0.375*v3	0.375, -6.9388939039072283776e-18	$CC_*(0.375) = 1.0$, $CC(v3) * C_*(v3) = 8.64e+15$ $CC(v5) = 8.64e+15$	1.0	2.6020852139652106e-18	3.49e-1
return v5					

Fig. 2. An motivation example with error propagation over operations.

- **op1: v1=0.25+b**
  - Chain Condition: $CC(v1) = C_+(2.5) + C_+(b) = \left|\frac{0.25}{0.25+b}\right| + \left|\frac{b}{0.25+b}\right| \approx 1.0$
  - Atomic Condition: $C(v1) = \left|\frac{0.25}{0.25+b}\right| + \left|\frac{b}{0.25+b}\right| \approx 1.0$

  - Relative Error: 1.01×10^{-16}, the error is introduced by the + operation and is less than 0.5 ULP.
- **op2: v2=x*v1**
  - Chain Condition: $CC(v2) = C_*(x) + CC(v1) * C_*(v1) = 1.0 + 1.0 = 2.0$
  - Atomic Condition: $C(v2) \approx 2.0$
  - Relative Error: 1.86×10^{-16}, the error is slightly amplified by $C(v2)$.
- **op3: v3=a-v2**
  - Chain Condition: $CC(v3) = C_-(a) + CC(v2) * C_-(v2) = \left|\frac{a}{a-v2}\right| + \left|2.0 * \frac{v2}{a-v2}\right| \approx 8.64e{+}15$
  - Atomic Condition: $C(v3) = \left|\frac{a}{a-v2}\right| + \left|\frac{v2}{a-v2}\right| \approx 5.76e{+}15$
  - Relative Error: 3.49×10^{-1}, the error is further amplified by $C(v3)$.
- **op4: v4=0.375+v3**
  - Chain Condition: $CC(v4) = C_+(0.375) + CC(v3) * C_+(v3) = \left|\frac{0.375}{0.375+v3}\right| + \left|8.64e{+}15 * \frac{v3}{0.375+v3}\right| \approx 1.16$
  - Atomic Condition: $C(v4) = \left|\frac{0.375}{0.375+v3}\right| + \left|\frac{v3}{0.375+v3}\right| \approx 1.0$
  - Relative Error: 2.84×10^{-17}, the absolute error introduced by v3, approximately 2.42e-18, is significantly smaller than the first operand 0.375 and can therefore be disregarded. As a result, the error in the calculated results $P_1(x)$ is negligible.
- **op5: v5=0.375*v3**
  - Chain Condition: $CC(v5) = C_*(0.375) + CC(v3) * C_*(v3) \approx 8.64e{+}15$
  - Atomic Condition: $C(v5) \approx 2.0$
  - Relative Error: 3.49×10^{-1}, the error from $v3$ is propagated to $v5$, which is consistent with the value of final chain condition value $8.65e + 15$.

This example illustrates that

- A large atomic condition can result in a significant relative error for an atomic operation. However, such an error may not propagate to the final result, as demonstrated by the relative error of $v4$ in **op4**.
- A **false positive** of ATOMU occurs when an input triggers a large atomic condition but fails to cause a significant floating-point error for the final output.
- The chain condition values consistently correspond to the relative errors, indicating the potential effectiveness of the chain condition in avoiding false positives.
- Chain conditions can be used to do backward error tracing. By utilizing chain conditions, we can locate the root causes of relative errors. In the example, we can observe that the error in $v5$ can be traced back to $v3$, where a catastrophic cancellation occurs in operation $a - v2$.

Based on these observations, we propose a new oracle-free approach called FPCC (**F**loating-**P**oint **C**hain **C**ondition). Unlike ATOMU, which relies on atomic conditions, FPCC utilizes chain conditions to identify inputs that can potentially trigger significant floating-point errors. By leveraging the ability of chain conditions to avoid false positives and accurately reflect error propagation, FPCC eliminates the need to search for inputs that activate large atomic conditions for each floating-point operation. Instead, FPCC focuses on identifying inputs that can lead to substantial final chain conditions. Considering that numerical programs often consist of numerous floating-point operations, the adoption of FPCC not only mitigates false positives but also improves the efficiency of detecting significant errors in numerical programs. We detail FPCC in §4.

4 Error Analysis via Chain Conditions

In this section, we first provide the definition of the problem addressed by our approach and the framework of the approach (§4.1). Then, we introduce chain condition and its corresponding calculation method (§4.2). After that, we present a chain condition guided search method for finding

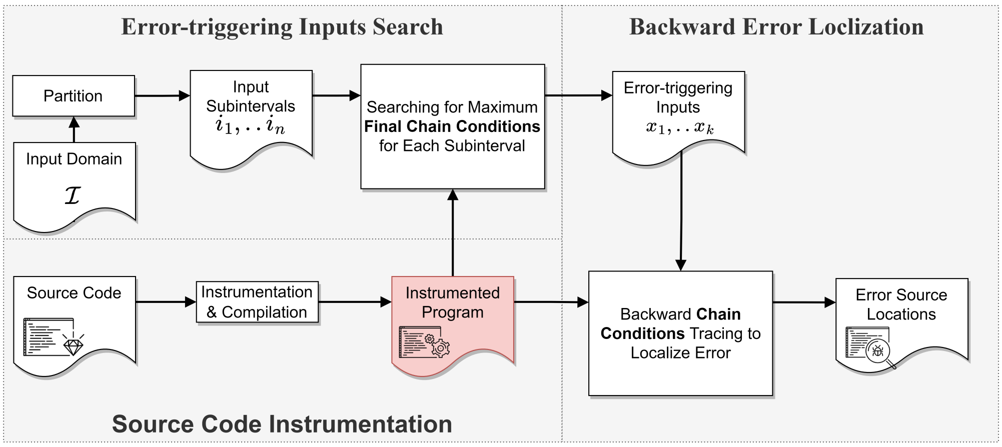

Error-triggering Inputs Search
Backward Error Loclization
Partition
Input Subintervals $i_1, .. i_n$
Searching for Maximum Final Chain Conditions for Each Subinterval
Error-triggering Inputs $x_1, .. x_k$
Input Domain $\mathcal{I}$
Source Code
Instrumentation & Compilation
Instrumented Program
Backward Chain Conditions Tracing to Localize Error
Error Source Locations
Source Code Instrumentation

Fig. 3. The framework of chain condition based error analysis.

inputs triggering significant floating-point errors (§4.3). Finally, we elucidate a chain condition based backward error tracing method for localizing the root causes of errors (§4.4).

4.1 Problem Definition and Approach Overview

Given a numerical program $\mathcal{P}(x)$, we aim to identify a set of error-triggering inputs (denoted as $\mathcal{X} = \{x_0, x_1, \ldots, x_k\}$) from its input domain $\mathcal{I}$. Specifically, suppose $\mathcal{F}(x)$ denotes an ideal implementation of $\mathcal{P}(x)$, for $x \in \mathcal{X}$, $Err(\mathcal{F}(x), \mathcal{P}(x)) > \epsilon$, where ϵ is a predefined error threshold.

Unfortunately, as discussed in §1, obtaining the ideal implementation $\mathcal{F}(x)$ can be computationally expensive and time-consuming due to the necessity of using higher precision to simulate $\mathcal{P}(x)$. We propose chain condition of $\mathcal{P}(x)$ as a criterion to measure the final floating-point error of $\mathcal{P}(x)$ and utilize it to guide the search of $\mathcal{X}$ over $\mathcal{I}$.

Fig. 3 shows the framework of our approach. We instrument the source code of a floating-point program to compute the final chain condition. The input domain $\mathcal{I}$ is divided into many subintervals, *i.e.*, $\{i_1, i_2, ..., i_n\}$, and a simple yet efficient search is employed to identify the input set $\mathcal{X} = \{x_1, x_2, ..., x_k\}$ that can trigger large final chain conditions. Finally, we utilize the runtime information and the calculated chain conditions to locate of the root causes of floating-point errors.

$$\begin{aligned}
\left|\frac{f(x,y)-f(x+\Delta x, y+\Delta y)}{f(x,y)}\right| &= \left|\frac{f(x,y)-f(x,y+\Delta y)+f(x,y+\Delta y)-f(x+\Delta x,y+\Delta y)}{f(x,y)}\right| \\
&= \left|\frac{f(x,y)-f(x,y+\Delta y)}{f(x,y)}\right| + \left|\frac{f(x,y+\Delta y)-f(x+\Delta x,y+\Delta y)}{f(x,y)}\right| \\
&\approx C_{f,y}(x,y)\cdot\left|\frac{\Delta y}{y}\right| + C_{f,x}(x,y+\Delta y)\cdot\left|\frac{\Delta x}{x}\right| \\
&\approx C_{f,y}(x,y)\cdot\left|\frac{\Delta y}{y}\right| + C_{f,x}(x,y)\cdot\left|\frac{\Delta x}{x}\right| \qquad (6)
\end{aligned}$$

4.2 Chain Condition

Atomic condition measures how the input error is amplified by an atomic operation. Specifically, for single-input operation $op(x)$, the relative error of output can be calculated by Formula 4, while for binary-input operation $op(x, y)$, the resulting relative error can be measured by Formula 6.

To capture the propagation of floating-point errors during the execution of an operation sequence, we introduce chain condition.

Definition 4.1. Given an operation sequence $\langle op_0, \ldots, op_i, \ldots, op_n \rangle (0 \leq i \leq n)$, operation op_i's chain condition CC_{op_i} evaluates how the input floating-point errors are amplified by the operation sequence $\langle op_0, \ldots, op_i \rangle$.

Fig. 4 gives the transition rules for chain conditions, *i.e.*, how chain conditions can be calculated via traversing the operation sequence, where $op_i \longrightarrow op_j$ means that the output of op_i serves as an input of op_j and $C_{op}(x)$ denotes the atomic condition of operation op *w.r.t.* x.

Given an operation, when its inputs do not depend on previous operations, the resulting floating-point error is only affected by the carry-in error and the operation itself. Hence, we can utilize the operation's atomic condition to evaluate the error propagation. As illustrated in rules Init-1 and Init-2, the chain condition CC_{op_j} is the summation of the atomic conditions of the inputs. The intuitions behind rules Unary, Binary-1 and Binary-2 are demonstrated by the proof of the following three theorems. For brevity, ε_x denotes the relative error of variable x (*i.e.*, $\frac{\Delta x}{x}$) and $\bar{x}$ represents the mean of x_1 and x_2.

$$\frac{y = op_j(x) \land \nexists op_i \longrightarrow op_j}{CC_{op_j} = C_{op_j}(x)} \tag{Init-1}$$

$$\frac{z = op_j(x, y) \land \nexists op_i \longrightarrow op_j}{CC_{op_j} = C_{op_j}(x) + C_{op_j}(y)} \tag{Init-2}$$

$$\frac{y = op_i(x, \ldots) \land z = op_j(y) \land op_i \longrightarrow op_j}{CC_{op_j} = CC_{op_i} \cdot C_{op_j}(y)} \tag{Unary}$$

$$\frac{y_1 = op_i(x_1, \ldots) \land z = op_k(y_1, y_2) \land op_i \longrightarrow op_k \land \nexists j \neq i, op_j \longrightarrow op_k}{CC_{op_k} = CC_{op_i} \cdot C_{op_k}(y_1) + C_{op_k}(y_2)} \tag{Binary-1}$$

$$\frac{y_1 = op_i(x_1, \ldots) \land y_2 = op_j(x_2, \ldots) \land z = op_k(y_1, y_2) \land op_i \longrightarrow op_k \land op_j \longrightarrow op_k}{CC_{op_k} = CC_{op_i} \cdot C_{op_k}(y_1) + CC_{op_j} \cdot C_{op_k}(y_2)} \tag{Binary-2}$$

Fig. 4. Transition rules of chain condition.

Theorem 3.1 Given an operation sequence $seq = \langle y = op_1(x_1, x_2); z = op_2(y) \rangle$, rule Unary captures how the input floating-point error is amplified by *seq*.

Proof. According to Equation 6, we have

$$\begin{aligned} \varepsilon_y &\approx C_{op_1}(x_1) \cdot \varepsilon_{x_1} + C_{op_1}(x_2) \cdot \varepsilon_{x_2} \\ &\approx \varepsilon_{\bar{x}} \cdot (C_{op_1}(x_1) + C_{op_1}(x_2)) \\ &= \varepsilon_{\bar{x}} \cdot CC_{op_1} \end{aligned}$$

The input error of op_2 (*i.e.*, ε_z) comes from input y. According to Equation 4, we have

$$\begin{aligned} \varepsilon_z &\approx C_{op_2}(y) \cdot \varepsilon_y \\ &\approx \varepsilon_{\bar{x}} \cdot CC_{op_1} \cdot C_{op_2}(y) \end{aligned}$$

Hence, rule Unary holds. □

Theorem 3.2 Given an operation sequence $seq = \langle x = op_1(x_1, x_2); z = op_2(x, y)\rangle$, rule Binary-1 captures how the input floating-point error is amplified by *seq*.

Proof. According to Equation 6, We have

$$\begin{aligned} \varepsilon_x &\approx C_{op_1}(x_1) \cdot \varepsilon_{x_1} + C_{op_1}(x_2) \cdot \varepsilon_{x_2} \\ &\approx \varepsilon_{\bar{x}} \cdot (C_{op_1}(x_1) + C_{op_1}(x_2)) \\ &= \varepsilon_{\bar{x}} \cdot CC_{op_1} \end{aligned}$$

Suppose the carry-in relative error of y is ε_y. The input error of op_2 (*i.e.*, ε_z) comes from the two inputs x and y. According to Equation 6, we have

$$\begin{aligned} \varepsilon_z &\approx C_{op_2}(x) \cdot \varepsilon_x + C_{op_2}(y) \cdot \varepsilon_y \\ &\approx \varepsilon_{\bar{x}} \cdot CC_{op_1} \cdot C_{op_2}(x) + \varepsilon_y \cdot C_{op_2}(y) \\ &\approx \varepsilon_{\bar{input}} \cdot (CC_{op_1} \cdot C_{op_2}(x) + C_{op_2}(y)) \end{aligned}$$

Hence, rule Binary-1 holds. □

Theorem 3.3 Given an operation sequence $seq = \langle m_1 = op_1(x_1, x_2); m_2 = op_2(y_1, y_2); z = op_3(m_1, m_2)\rangle$, rule Binary-2 captures how the input floating-point error is amplified by *seq*.

Proof. According to Equation 6, We have

$$\begin{aligned} \varepsilon_{m_1} &\approx C_{op_1}(x_1) \cdot \varepsilon_{x_1} + C_{op_1}(x_2) \cdot \varepsilon_{x_2} \\ &\approx \varepsilon_{\bar{x}} \cdot (C_{op_1}(x_1) + C_{op_1}(x_2)) \\ &= \varepsilon_{\bar{x}} \cdot CC_{op_1} \end{aligned}$$

Similarly, the relative error of operation op_2 (*i.e.*, ε_{m_2}) is approximately $\varepsilon_{\bar{y}} \cdot CC_{op_2}$. The input error of op_3 (*i.e.*, ε_z) comes from the two inputs m and n. According to Equation 6, we have

$$\begin{aligned} \varepsilon_z &\approx C_{op_3}(m_1) \cdot \varepsilon_{m_1} + C_{op_3}(m_2) \cdot \varepsilon_{m_2} \\ &\approx \varepsilon_{\bar{x}} \cdot CC_{op_1} \cdot C_{op_3}(m_1) + \varepsilon_{\bar{y}} \cdot CC_{op_2} \cdot C_{op_3}(m_2) \\ &\approx \varepsilon_{\bar{input}} \cdot (CC_{op_1} \cdot C_{op_3}(m_1) + CC_{op_2} \cdot C_{op_3}(m_2)) \end{aligned}$$

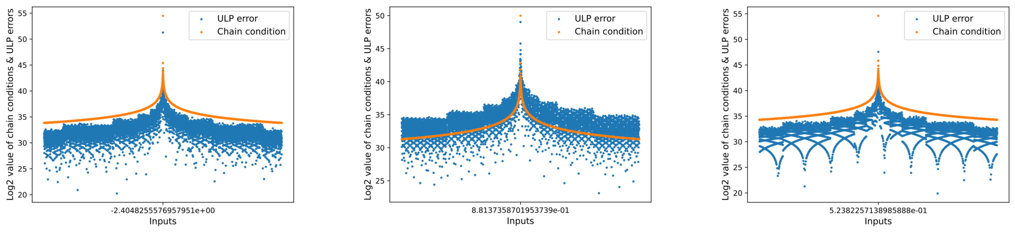
Hence, rule Binary-2 holds. □

The final chain condition of numerical program $\mathcal{P}$: For an operation sequence Seq $= \{st_0, \ldots, st_n\}(n \geq 0)$ derived from an instrumented numerical program $\mathcal{P}$. The main idea is sequentially calculating the chain conditions of Seq$'s$ prefixes. For two operations st_i and st_j, $st_i \longrightarrow st_j(x)$ indicates that the output of st_i serves as st_j's input x. Given an operation $st_i (0 \leq i \leq n)$, through checking the preconditions of the transition rules in Figure 4, we apply the enabled one to calculate chain condition CC_i for sequence $\{st_0, \ldots, st_i\}$. Finally, CC_n is returned as the final chain condition of Seq.

4.3 Chain Condition-Guided Global Search

We aim to develop a global search approach that can effectively detect inputs triggering large chain conditions in floating-point programs. The quest for identifying inputs causing high floating-point errors has long been a challenging problem, with techniques such as binary search [Chiang et al. 2014], differential evolution [Yi et al. 2017, 2019], MCMC [Yi et al. 2019], and genetic algorithm [Zou et al. 2015, 2019] being employed. Our objective is to design a simple yet efficient search algorithm specifically tailored to the characteristics of the chain condition, enabling rapid identification of large chain conditions.

To analyze the characteristics of chain conditions, we randomly sample a large number of inputs from input domains of three GSL functions and identify an input that triggers significant error for each function. We then examine the distributions of floating-point errors and chain conditions around these inputs. As illustrated in Figure 5, There exists a remarkable consistency between the distribution of final chain conditions and the distribution of floating-point errors. Moreover, the distribution of final chain conditions exhibits a clear trend of gradual increase. Based on these observations, we derive two guiding rules: 1) **R1**: Chain condition values can be used to guide the search for the maximum floating-point error; 2) **R2**: Changes in chain conditions can guide the search for the potentially maximum chain condition.



(a) gsl_sf_bessel_J0 (b) gsl_sf_lnsinh (c) gsl_sf_Chi

Fig. 5. Distributions of chain conditions and ULP errors w.r.t. inputs.

In accordance with the two guiding rules, we employ two simple and well-established search algorithms (*i.e.*, *Direct* Search [Gablonsky and Kelley 2001; Jones et al. 1993] and *Line* Search [Moré and Thuente 1994]) to explore error-triggering inputs. Following **R1**, we solely utilize chain conditions to direct the search for error-triggering inputs. In line with **R2**, *Direct* Search is employed to localize input regions that tend to have large chain conditions, and *Line* Search is used to find inputs that could potentially trigger the maximum chain condition within the identified input region based on the changes in chain conditions.

Algorithm 1 shows the workflow of our search approach, including three primary steps: Partition, *Direct* Search, and *Line* Search.

Partition. The algorithm begins by partitioning the input domain $\mathcal{I}$ into a set of sub-intervals (denoted as I_s). The fundamental concept of Partition is to decompose the input domain of a program, thereby effectively reducing the search space and avoiding getting trapped in local optima. We employ a non-uniform partitioning method based on the density distribution of floating-point numbers. We consider the input range $[-2^{88}, 2^{88}]$, which covers almost all physical constants, to be a commonly used input range in scientific computing. Therefore, we increase the numbers of sub-intervals within this range. For example, for programs with one input that ranges from $-fmax$ to $fmax$ (where $fmax \approx 2^{1024}$), the presentation of a positive sub-interval $i \in I_s$ for floating-point numbers is shown in Formula 7, and the negative sub-intervals in I_s is just reversing i and multiplying -1.

$$for\ i \in I_s,\ i = \begin{cases} [0, 2^{-1022}] & \text{if } e \leq -1022 \\ (2^e, 2^{e+30}] & \text{if } -1022 < e \leq -22 \\ [2^e, 2^{e+1}] & \text{if } -22 < e \leq 88 \\ [2^e, 2^{e+60}) & \text{if } 88 < e \leq 1023 \end{cases} \tag{7}$$

Algorithm 1: Chain Condition-Guided Global Search

```
Input   : an instrumented floating-point program $\mathcal{P}$ and an input domain $\mathcal{I}$
Output : a list $X$ for the inputs that trigger large chain conditions
1  $I_s \leftarrow partition(\mathcal{I})$
2  $CC_l \leftarrow \emptyset$
3  $TempCC_l \leftarrow \emptyset$
4  for $i \in I_s$ do
5      $(x_i, cc_i) \leftarrow DirectSearch(i, \mathcal{P})$
6      $TempCC_l.append([x_i, cc_i])$
7  end
8  Sort($TempCC_l$)
9  $k \leftarrow 0$
10 for $tc \in TempCC_l$ do
11     $x_k \leftarrow tc.x_k$
12     $cc_k \leftarrow tc.cc_k$
13     if $k < limit$ then
14         $(x_k, cc_k) \leftarrow LineSearch(x_k, , \mathcal{P})$
15     end
16     if $cc_k > threadhold$ then
17         $CC_l.append([x_k, cc_k])$
18     end
19     $k \leftarrow k + 1$
20 end
21 Sort($CC_l$)
22 $X \leftarrow GetInputs(CC_l)$
23 return $X$
```

***Direct* Search.** To identify inputs that trigger large chain conditions for program $\mathcal{P}$, we apply the Direct search algorithm [Gablonsky and Kelley 2001] to each sub-interval i in I_s, resulting in inputs x_i and their corresponding chain condition values cc_i (lines 4-7). These values are stored in the set $TempCC_l$ (line 8). To refine the results, we sort $TempCC_l$ in decreasing order. The sorted set is then passed to the line search process.
***Line* Search.** Finally, a line search is optionally conducted for a limited number of iterations if the condition $k < limit$ is satisfied (lines 13-15). This line search explores the vicinity of the input x_k obtained from the direct search to find a larger chain condition value cc_k. If cc_k exceeds a specified threshold, both the input x_k and its corresponding chain condition value cc_k are added to the set CC_l (lines 16-18). Set CC_l is also sorted in descending order and the resulting list is assigned to list X (lines 21-22).

4.4 Chain Condition Based Error Localization

To localize the source code of an error, we perform a backward tracing of the chain conditions to identify floating-point operations that introduce large chain conditions and propagate them to the output of the program. As shown in Algorithm 2, the algorithm extracts runtime information about all instructions in $\mathcal{P}$ for the given error-triggering input x, resulting in an instructions sequence Seq = $\{st_0, \ldots, st_n\}$. Then, it uses set $SubSeq$ (initialized to be empty) to store the localized source

Algorithm 2: Chain Condition Based Error Localization Algorithm

```
Input: $\mathcal{P}$: an instrumented floating-point program; $x$: an error-triggering input
Output: $SubSeq$: the sequence of localized source code.
1  Seq = $\{st_0, \ldots, st_n\} \leftarrow \mathcal{P}(x)$
2  $SubSeq \leftarrow \emptyset$
3  BackwardErrorTrace($st_n, Seq, SubSeq$)
4  return $SubSeq$
5  Function BackwardErrorTrace($st_n, Seq, SubSeq$):
6      $st_i, st_j \leftarrow$ GetPredecessors($Seq, st_n$)
7      $SubSeq \leftarrow st_n$
8      if $CC(st_i) > CC(st_n)/\omega$ then
9          BackwardErrorTrace($Seq, st_i, SubSeq$)
10     end
11     if $CC(st_j) > CC(st_n)/\omega$ then
12         BackwardErrorTrace($Seq, st_j, SubSeq$)
13     end
14     return 0
15 End Function
```

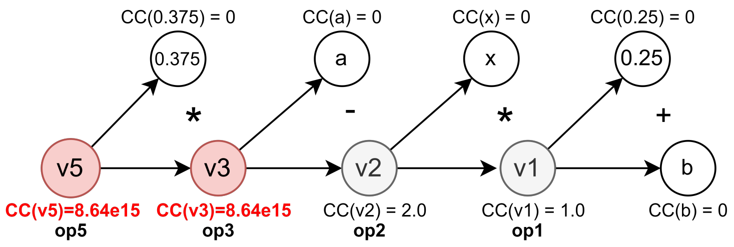


Fig. 6. Error trace for example code $\mathcal{P}_2$ in §3.

code. Function *GetPredecessors* obtains the two precedent statements st_i and st_j of st_n in Seq. If the chain condition value of a precedent statement exceeds a threshold ($CC(st_n)/\omega$), where ω is a user-defined constant greater than 1, then the function recursively calls itself with the respective statement and updates *SubSeq*. Finally, the algorithm returns the localized source code stored in *SubSeq*. For the example code $\mathcal{P}_2$ in §3, as shown in Fig. 6, the algorithm traces the error from ***op5*** to ***op3*** based on the values of chain conditions and forms the resulting $SubSeq = \{op5, op3\}$.

5 Evaluation

In this section, we first talk about the implementation of FPCC. Then, we present our experimental settings and evaluate FPCC on real-world numerical programs. Finally, we provide insights on utilizing FPCC for tracing and localizing floating-point errors in numerical programs.

5.1 Implementation

Our implementation of FPCC follows the high-level structure depicted in Fig. 3 using C++ and Python. FPCC has the capability to instrument a provided numerical program, identify inputs that trigger significant chain conditions, and trace and localize the source code of the potential errors. Its implementation consists primarily of three key components:

(1) A **Transformer** that lifts a numerical program $\mathcal{P}$ to an instrumented program $\hat{\mathcal{P}}$ to calculate the chain conditions and collect the information of instructions in $\mathcal{P}$. The transformer operates on the LLVM IR code of $\mathcal{P}$. It mainly consists of an LLVM PASS module that instruments the IR code of $\mathcal{P}$ and includes handler functions to calculate chain conditions and store instruction information. For each floating-point instruction, the *Transformer* injects a function call to an external handler to calculate the chain condition within the instruction and store relevant information (such as instruction ID for backward error trace).
(2) A **Searcher** that implements Algorithm 1 described in §4.3 to detect error-triggering inputs. The *Searcher* consists of a direct search engine and a line search engine. Notably, we implement the line search ourselves and adopt an open-source implementation of direct search [1].
(3) A **Tracer** that implements Algorithm 2 described in §4.4 to trace and localize the source code of errors. Given an instrumented numerical program $\hat{\mathcal{P}}$ and an error-triggering input x, the *Tracer* collects runtime information of $\hat{\mathcal{P}}$ and provides the possible locations of source code that leads to significant errors.

5.2 Experimental Setup

5.2.1 Subjects. A series of experiments were conducted to assess FPCC on a selection of subjects acquired from the GNU Scientific Library (GSL version 2.5)[2]. GSL is a publicly available numerical library that offers an extensive collection of mathematical algorithms, including random number generators, special functions, and least-squares fitting methods. GSL has been widely used to evaluate various floating-point research methodologies [Guo and Rubio-González 2020; Yi et al. 2017, 2019; Zou et al. 2015, 2019].

GSL includes a total of 154 special functions, each utilizing floating-point parameters and return values. ATOMU [Zou et al. 2019] supports only single inputs and selects 88 univariate functions from the aforementioned 154 special functions. In order to directly compare with the state-of-the-art approach ATOMU [Zou et al. 2019], we also concentrate on these 88 univariate functions.

To compare with FPGen [Guo and Rubio-González 2020], the state-of-the-art tool for detecting high floating-point errors in functions with multiple inputs, we utilize the benchmarks provided by FPGen [Guo and Rubio-González 2020], consisting of 21 benchmarks: 3 summation algorithms, 9 matrix computation routines from the Meschach library [Stewart and Leyk 1994], and 9 statistical routines from the GNU Scientific Library (GSL).

The code for the 88 GSL functions spans between 14 and 329 lines, while the 21 multiple-input programs from FPGen consist of 8 to 132 lines, predominantly featuring nested loops like matrix multiplication.

5.2.2 Oracles. Although FPCC is an oracle-free tool for detecting significant errors, it is still necessary to have knowledge of the correct results of programs in order to validate the effectiveness of FPCC. For the 88 special functions from GSL, in line with prior research studies [Yi et al. 2017; Zou et al. 2019], to mitigate precision-related operations that may introduce errors in high precisions, we employ *mpmath* (a library that facilitates floating-point arithmetic with arbitrary precision) to compute the oracles of a numerical program $\mathcal{P}$. As for the 21 functions with multiple inputs, where precision-specific operations were not identified, following the approach of FPGen [Guo and Rubio-González 2020], we opt to utilize higher precision (long double) exclusively for the oracles.

[1]https://github.com/rlnx/DiRect
[2]https://www.gnu.org/software/gsl/

5.2.3 Parameters Setup. To ensure a fair comparison between FPCC and ATOMU, both of which rely on random seed in their search process, we conduct 100 repeated experiments using FPCC and ATOMU and use the average results. This was done to reduce the influence of randomness on the experimental results. In Algorithm 1, we set the *limit* parameter (line 13) to be 10, meaning that the top 10 sub-intervals will undergo a fine line search. And we set the *threshold* parameter (line 16) to be 1e14. We set ω in Algorithm 2 to 100 in order to trace floating-point operations that magnify chain conditions by a factor of 100.

For k-ary functions, the number of input ranges to be searched increases exponentially, making it difficult for both FPCC and FPGen to search all input ranges within a limited time. Consequently, a time limit must be imposed. In the corresponding paper of FPGen [Guo and Rubio-González 2020], the experimental time was set to 7200 seconds (2 hours) to yield its outcomes. Hence, we have adhered to FPGen's established setting. Given that FPCC does not rely on high-precision test oracles and executes at a notably rapid pace, we have set a time limit of 100 seconds for FPCC.

5.2.4 Evaluation Metrics. In our experiments, we employ the following metrics to assess the effectiveness and efficiency of the methods.

Significant floating-point error: We measure the error using the relative error Err_{rel} specified in Formula 1. Following previous research [Zou et al. 2019], we consider a relative error greater than 10^{-3} as significant.

Error-triggering inputs: For each project, both FPCC and ATOMU produce a list of inputs that are suspected of causing significant floating-point errors. For example, ATOMU provided a list of 26 inputs for the function airy_Ai, and 7 of these inputs were confirmed by *mpmath* to indeed trigger significant floating-point errors. We denote the count of these error-triggering inputs as 26, and this result is recorded as "7/26" in the "Error-triggering Inputs" column of Table 1.

Rank-1 input: In the context of FPCC and ATOMU, rank-1 input refers to the top-ranked suspicious input identified by these methods for potential significant floating-point error. These methods, being oracle-free and based on condition numbers, do not provide actual error values, so the rank-1 input may not necessarily trigger a significant floating-point error. After conducting 100 repeated experiments, we obtain 100 rank-1 inputs, and we assess the proportion of these inputs that actually trigger a significant floating-point error. The results of this evaluation are presented in the "% of Rank-1 Inputs" column of Tables 1 and 2,

We conduct experiments on a desktop running Ubuntu 18.04 LTS with an Intel Core i9-13900 @ 5.20 GHz CPU and 32GB RAM.

5.3 Evaluation Results

To better evaluate FPCC, we compare it with ATOMU [Zou et al. 2019], the state-of-the-art tool for detecting floating-point errors while not relying on oracles. Our experimental evaluation focuses on investigating the following research questions (RQs):

- **RQ1:** How effective is FPCC in detecting functions with significant errors?
- **RQ2:** How efficient is FPCC in detecting functions with significant errors?
- **RQ3:** How scalable is FPCC?
- **RQ4:** How stable is FPCC?

5.3.1 RQ1: How effective is FPCC in detecting functions with significant errors? As depicted in Tables 1 and 2, the "% of Rank-1 Inputs" column presents the percent of error-triggering inputs among the rank-1 inputs reported by both FPCC and ATOMU across 100 repeated experiments. The "Error-triggering Inputs" column displays the total number of inputs that effectively trigger significant errors among all the reported inputs. The "Max RelErr" column illustrates the maximum relative floating-point errors triggered by FPCC and ATOMU across all reported inputs. The “Max ULPErr” column shows the maximum *log2* value of ulp errors triggered by all reported inputs. In Table 1, the values that have better results are bold for each column[3].

Table 1. Results for the 42 functions with significant errors.

GSLfunctions	Pct. of Rank-1 Inputs		Error-triggering Inputs		Max RelErr		Max ULPErr		Time		Speedup
	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	
airy_Ai	**100%**	0%	**58/58**	7/26	**1.38E+298**	2.91E+04	**63.54**	62.72	**0.217**	0.562	2.59
airy_Bi	**100%**	5%	**58/58**	8/27	**1.35E+302**	1.12E+10	**63.55**	62.99	**0.221**	0.538	2.43
airy_Ai_scaled	**100%**	0%	**58/58**	7/26	**1.38E+298**	1.53E+04	**63.54**	62.79	**0.239**	0.567	2.37
airy_Bi_scaled	**100%**	1%	**58/58**	8/27	**1.35E+302**	5.81E+09	**63.55**	62.99	**0.245**	0.561	2.29
airy_Ai_deriv	**100%**	**100%**	**10/10**	1/16	**4.34E+00**	3.46E-02	**63.22**	46.88	**0.067**	0.216	3.23
airy_Bi_deriv	**100%**	**100%**	**10/10**	2/16	**2.58E+00**	8.03E-01	**63.20**	50.03	**0.068**	0.222	3.27
airy_Ai_deriv_scaled	**100%**	**100%**	**10/10**	1/15	**4.34E+00**	4.02E-02	**63.22**	47.05	**0.082**	0.222	2.72
airy_Bi_deriv_scaled	**100%**	**100%**	**10/10**	2/15	**2.58E+00**	3.23E-01	**63.20**	49.59	**0.092**	0.228	2.49
bessel_J0	**100%**	**100%**	**6/6**	2/14	**3.90E-01**	1.65E-01	**51.26**	49.04	**0.402**	0.419	1.04
bessel_J1	**100%**	**100%**	**8/8**	2/15	**1.87E-01**	1.04E-01	**50.52**	48.68	**0.417**	0.422	1.01
bessel_Y0	**100%**	**100%**	**5/5**	2/24	**2.86E+00**	1.84E-01	**61.92**	49.46	**0.176**	0.392	2.23
bessel_Y1	**100%**	89%	**5/5**	2/24	**1.17E-01**	8.68E-02	**49.43**	48.56	**0.172**	0.401	2.34
bessel_j1	**100%**	81%	**4/4**	1/3	3.10E-02	**3.38E-02**	**47.80**	42.55	**0.012**	0.039	3.23
bessel_j2	**100%**	91%	**4/6**	1/4	**3.61E-01**	7.74E-02	**51.03**	45.94	**0.017**	0.042	2.53
bessel_y0	**100%**	**100%**	**46/47**	7/14	**8.32E+119**	1.36E+04	**63.13**	62.72	**0.077**	0.132	1.72
bessel_y1	**100%**	**100%**	**48/48**	13/23	**9.29E+114**	2.51E+09	**63.14**	62.92	**0.170**	0.334	1.96
bessel_y2	**100%**	0%	**46/47**	13/25	**8.32E+119**	8.44E+10	**63.13**	62.92	**0.170**	0.360	2.12
clausen	**100%**	**100%**	**18/18**	3/11	6.60E-01	**1.01E+00**	52.66	**57.31**	**0.098**	0.143	1.46
dilog	**100%**	36%	**1/1**	0/10	**5.52E-01**	3.24E-01	**52.16**	20.16	**0.112**	0.165	1.48
expint_E1	**100%**	**100%**	**1/1**	**1/16**	**4.58E-01**	1.81E-01	**51.76**	49.23	**0.101**	0.261	2.58
expint_E2	**100%**	**100%**	**1/1**	**1/17**	**2.65E+02**	6.17E+01	**62.92**	56.18	**0.102**	0.290	2.85
expint_E1_scaled	**100%**	99%	**1/1**	1/16	**4.58E-01**	1.57E-01	**51.84**	48.69	**0.263**	0.378	1.44
expint_E2_scaled	**100%**	**100%**	**58/58**	2/17	**7.76E+291**	3.40E+288	**62.92**	62.54	**0.295**	0.383	1.30
expint_Ei	**100%**	**100%**	**1/1**	**1/16**	**4.58E-01**	1.71E-01	**51.76**	49.07	**0.092**	0.268	2.91
expint_Ei_scaled	**100%**	**100%**	**1/1**	**1/16**	**4.58E-01**	1.49E-01	**51.84**	49.07	**0.261**	0.386	1.48
Chi	**100%**	**100%**	**2/2**	1/17	4.40E-02	**8.57E-02**	47.56	**48.39**	**0.216**	0.504	2.34
Ci	**100%**	**100%**	**49/49**	13/36	**9.29E+114**	2.55E+08	**63.14**	62.58	**0.245**	0.931	3.80
lngamma	**100%**	0%	**2/2**	2/21	1.35E+00	**3.98E+00**	**62.93**	53.02	**0.054**	0.170	3.14
lambert_W0	**100%**	50%	**62/62**	1/8	**1.00E+00**	3.51E-01	**61.76**	50.49	**0.033**	0.063	1.91
lambert_Wm1	**100%**	99%	**31/31**	2/9	**1.00E+00**	8.94E-01	**61.76**	58.78	**0.035**	0.069	1.95
legendre_P2	**100%**	**100%**	**2/2**	1/1	4.19E-02	**7.55E-02**	47.48	**47.95**	**0.007**	0.011	1.51
legendre_P3	**100%**	**100%**	**2/2**	1/1	**1.00E+00**	1.14E-01	**61.92**	48.64	**0.010**	0.013	1.34
legendre_Q1	**100%**	**100%**	**2/2**	1/5	**5.03E-01**	1.32E-01	**52.01**	48.38	**0.011**	0.025	2.20
psi	**100%**	**100%**	**12/12**	3/19	3.49E-01	**5.06E+00**	**51.48**	51.08	**0.184**	0.355	1.93
psi_1	**100%**	0%	**11/11**	1/7	**5.76E-01**	1.07E-01	**52.38**	47.99	**0.084**	0.179	2.14
sin	**100%**	**100%**	**80/80**	7/14	**9.29E+114**	2.53E+10	**63.25**	62.89	**0.114**	0.160	1.41
cos	**100%**	**100%**	**78/78**	7/14	**8.32E+119**	1.43E+04	**63.24**	62.87	**0.113**	0.157	1.40
sinc	**100%**	**100%**	**174/174**	8/16	1.00E+00	**1.00E+00**	**62.36**	62.22	**0.126**	0.227	1.81
lnsinh	**100%**	**100%**	**1/1**	**1/2**	1.13E-01	**1.65E-01**	49.03	**49.36**	**0.007**	0.016	2.25
zeta	**100%**	0%	**6/6**	2/38	**6.24E-01**	2.39E-02	**51.35**	45.98	**0.144**	0.589	4.10
zetam1	**100%**	2%	**3/3**	1/42	**5.65E-02**	6.27E-03	**48.15**	43.09	**0.124**	0.646	5.22
eta	**100%**	0%	**6/6**	4/43	**6.24E-01**	1.84E-02	**51.33**	46.23	**0.154**	0.624	4.06
Summary	**100%**	72.69%	1049/1053 99.62%	141/723 19.45%					0.139	0.302	2.17

Percent of error-triggering inputs over rank-1 inputs. As shown in Table 1, both FPCC and ATOMU have successfully identified inputs that can lead to significant errors on 42 GSL functions. Since both methods are oracle-free, the rate of false positives plays a crucial role in evaluating their effectiveness in detecting significant floating-point errors. Table 1 provides insights into this evaluation. The “% of Rank-1 Inputs” column reveals that all of the rank-1 inputs reported by FPCC triggered significant floating-point errors, achieving a 100% (42/42) detection rate. In contrast, there only exist 25 functions (25/42, 59.5%) that ATOMU’s rank-1 inputs can 100% trigger significant errors over 100 repeated experiments. On average, 100% of rank-1 inputs that reported by FPCC can

[3]Due to the limited number of significant digits, some values in the table may appear identical.

trigger significant errors while 72.69% rank-1 inputs reported by ATOMU can trigger significant errors.

Number of error-triggering inputs. The "Error-triggering Inputs" column highlights FPCC's remarkable consistency in reporting inputs that are capable of triggering significant floating-point errors across all functions, thereby demonstrating its robustness. On the contrary, ATOMU exhibits limited ability to identify such inputs for most functions. In total, FPCC reports 1053 inputs, with an impressive 99.62% (1049 inputs) capable of triggering significant floating-point errors.

In comparison, ATOMU reports 723 inputs, of which only 19.45% (141 inputs) can trigger significant floating-point errors. These statistics further emphasize the superior performance of FPCC in accurately identifying inputs that lead to significant errors, showcasing its effectiveness and reliability.

There exist only four inputs reported by FPCC fail to trigger significant errors. We discovered that these inputs, specifically in the functions *bessel_j*2(2), *bessel_y*0(1), *bessel_y*1(1), can actually induce a relative error larger than *1e-4*, which is very close to *1e-3*. Additionally, FPCC accurately identified the source code responsible for the error based on these four inputs. It is reasonable to argue that chain condition utilized in FPCC is an approximation of relative error and the 4 inputs mentioned above are still valuable inputs for debugging numerical programs.

More error-triggering inputs may help developers better understand the nature of the error. For instance, in the case of the *airy_Ai* function, FPCC identified 58 error-triggering inputs. These inputs reveal that significant errors occur when the inputs are large, which aligns with the characteristics of the inaccuracy reduction bug described in §6.

Bug duplication. We analyzed the duplication of all reported bugs by tracing the triggering locations within functions. We found that all reported bugs can be traced back to 43 bug locations across 42 functions with significant errors. We offer detailed insights into these bugs and their respective locations in §6. Notably, all 1053 inputs reported by FPCC can trigger all these bug locations. Additionally, ATOMU can trigger all these bug locations within 141 inputs. ATOMU iteratively detects all unstable floating-point instructions, meaning it traverses all bug locations. Therefore, ATOMU's detection should be comprehensive but not necessarily reliable. For example, for the GSL function airy_ai, ATOMU reports 26 inputs pointing to more than 10 locations. While these locations cover the bug locations, users cannot determine which one is the actual bug location. In contrast, all inputs from FPCC point to the actual bug locations. Thus, FPCC primarily addresses the false positive issue of ATOMU, aiming to enhance the detection reliability.

Maximum errors. In terms of the detected maximum relative errors, FPCC performs better in 34 functions, while ATOMU detects higher relative errors in 8 functions. Similarly, FPCC detects higher ULP errors in 38 functions, while only 4 functions for ATOMU.

False positives and false negatives in the 46 functions without significant errors. Regarding the 46 functions without significant errors, as shown in Table 2, FPCC does not report any inputs. In contrast, ATOMU reports a total of 303 inputs for these functions. However, none of these inputs have the capability to trigger significant errors. We re-ran FPCC with 100 times more sampled inputs across 46 functions, yet we did not encounter any new functions with significant errors. The empirical results demonstrate that FPCC does not produce false negatives.

RQ1: *Compared to ATOMU, FPCC excels with 100% accuracy in detecting major errors for its top-ranked inputs, versus ATOMU's 72.69%. Across all reported inputs, FPCC identifies significant floating-point errors in 99.62%, while ATOMU only detects such errors in 19.45% (141/723). These figures underscore FPCC 's reliability and effectiveness in error detection.*

Table 2. Results for the 46 functions without significant errors.

GSLfunctions	% of Rank1-Inputs		Error-triggering Inputs		Max RelErr		Max ULPErr		Time		Speedup
	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	FPCC	ATOMU	
bessel_I0	0%	0%	0/0	0/9	0	2.11E-16	0	1.18	0.045	0.104	2.32
bessel_I1	0%	0%	0/0	0/9	0	2.19E-16	0	1.18	0.043	0.105	2.42
bessel_I0_scaled	0%	0%	0/0	0/9	0	1.92E-16	0	1.03	0.096	0.177	1.85
bessel_I1_scaled	0%	0%	0/0	0/9	0	1.91E-16	0	1.04	0.100	0.177	1.76
bessel_K0	0%	0%	0/0	0/10	0	2.37E-16	0	1.28	0.027	0.112	4.11
bessel_K1	0%	0%	0/0	0/10	0	2.09E-16	0	1.10	0.029	0.119	4.18
bessel_K0_scaled	0%	0%	0/0	0/8	0	1.82E-16	0	1.02	0.029	0.108	3.79
bessel_K1_scaled	0%	0%	0/0	0/8	0	1.92E-16	0	1.04	0.038	0.116	3.03
bessel_j0	0%	0%	0/0	0/1	0	4.58E-17	0	0.33	0.008	0.023	2.74
bessel_i0_scaled	0%	0%	0/0	0/0	0	0	0	0.00	0.008	0.011	1.37
bessel_i1_scaled	0%	0%	0/0	0/1	0	4.29E-15	0	4.11	0.011	0.017	1.52
bessel_i2_scaled	0%	0%	0/0	0/1	0	5.04E-12	0	15.21	0.012	0.015	1.30
bessel_k0_scaled	0%	0%	0/0	0/0	0	0	0	0.00	0.005	0.006	1.20
bessel_k1_scaled	0%	0%	0/0	0/0	0	0	0	0.00	0.006	0.007	1.17
bessel_k2_scaled	0%	0%	0/0	0/0	0	0	0	0.00	0.007	0.007	1.05
ellint_Kcomp	0%	0%	0/0	0/4	0	1.79E-10	0	20.00	0.073	0.197	2.70
ellint_Ecomp	0%	0%	0/0	0/10	0	1.81E-15	0	2.94	0.206	0.464	2.25
erfc	0%	0%	0/0	0/8	0	9.84E-16	0	2.32	0.056	0.181	3.20
log_erfc	0%	0%	0/0	0/9	0	2.77E-16	0	1.48	0.042	0.123	2.91
erf	0%	0%	0/0	0/6	0	1.02E-16	0	0.92	0.065	0.162	2.51
erf_Z	0%	0%	0/0	0/0	0	0	0	0.00	0.008	0.009	1.26
erf_Q	0%	0%	0/0	0/8	0	4.8E-15	0	3.98	0.069	0.185	2.68
hazard	0%	0%	0/0	0/10	0	2.94E-14	0	5.90	0.040	0.143	3.54
exp	0%	0%	0/0	0/0	0	0	0	0.00	0.005	0.006	1.30
expm1	0%	0%	0/0	0/2	0	2.73E-14	0	6.84	0.007	0.013	1.87
exprel	0%	0%	0/0	0/2	0	2.45E-14	0	7.38	0.007	0.013	1.87
exprel_2	0%	0%	0/0	0/4	0	5.52E-11	0	17.93	0.008	0.018	2.37
Shi	0%	0%	0/0	0/19	0	3.56E-16	0	1.64	0.120	0.367	3.06
Si	0%	0%	0/0	0/13	0	1.59E-16	0	1.07	0.138	0.348	2.53
fermi_dirac_m1	0%	0%	0/0	0/0	0	0	0	0.00	0.006	0.010	1.57
fermi_dirac_0	0%	0%	0/0	0/1	0	1.15E-14	0	6.21	0.007	0.014	1.97
fermi_dirac_1	0%	0%	0/0	0/11	0	4.01E-16	0	1.87	0.075	0.188	2.51
fermi_dirac_2	0%	0%	0/0	0/11	0	6.43E-16	0	2.02	0.068	0.185	2.73
fermi_dirac_mhalf	0%	0%	0/0	0/13	0	8.89E-15	0	5.33	0.140	0.255	1.82
fermi_dirac_half	0%	0%	0/0	0/13	0	2.74E-14	0	6.93	0.142	0.268	1.89
fermi_dirac_3half	0%	0%	0/0	0/13	0	1.91E-14	0	5.96	0.132	0.250	1.90
gamma	0%	0%	0/0	0/20	0	1.36E-13	0	9.45	0.021	0.158	7.40
gammainv	0%	0%	0/0	0/26	0	6.37E-14	0	8.53	0.053	0.278	5.24
legendre_P1	0%	0%	0/0	0/0	0	0	0	0.00	0.005	0.006	1.20
legendre_Q0	0%	0%	0/0	0/3	0	0	0	0.00	0.010	0.020	1.95
log	0%	0%	0/0	0/1	0	0	0	0.00	0.005	0.010	1.97
log_abs	0%	0%	0/0	0/1	0	0	0	0.00	0.005	0.011	2.09
log_1plusx	0%	0%	0/0	0/6	0	1.17E-16	0	0.71	0.026	0.070	2.66
log_1plusx_mx	0%	0%	0/0	0/6	0	2.03E-16	0	1.07	0.026	0.069	2.64
synchrotron_2	0%	0%	0/0	0/9	0	7.08E-13	0	12.17	0.043	0.104	2.43
lncosh	0%	0%	0/0	0/0	0	0	0	0.00	0.028	0.025	0.88
Total number of error-triggering inputs			0/0	0/303				Average time	0.046	0.114	2.50

5.3.2 *RQ2: How efficient is FPCC in detecting functions with significant errors?*

In Table 1, the smaller numbers in column Time are highlighted in bold. In terms of processing functions with significant errors, FPCC is much faster, taking approximately 0.14 seconds, while ATOMU takes 0.30 seconds. The speedup values shown in the SPEEDUP column of Table 1 range from 1.01x to 5.22x, with an average speedup of 2.17x. To assess the statistical significance of the time difference between the two sets, we conducted a Mann-Whitney-Wilcoxon test [Wilcoxon 1992] which yielded a p-value of $1.65 * 10^{-4} (< 0.05)$. This p-value indicates a significant difference between the two sets of time.

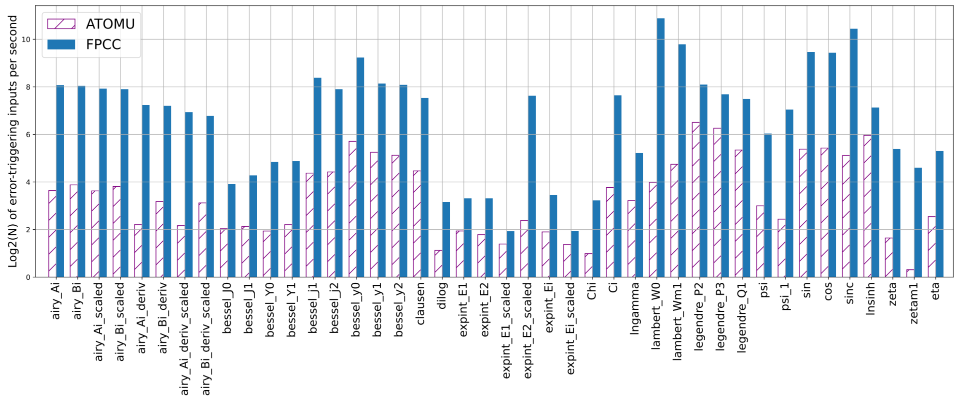


Fig. 7. Comparison of the Log2(N) number of error-triggering inputs per second detected by ATOMU and FPCC.

Among the 46 functions without significant errors, the Time column in Table 2 illustrates that FPCC takes less time than ATOMU for most of these functions. The speedups range from 0.87x to 7.40x, with an average speedup of 2.50x. Additionally, the p-value of 0.0026 (< 0.05) from the Mann-Whitney-Wilcoxon test indicates a significant difference between the two sets of time.

Moreover, Fig. 7 visually represents the logarithm (base 2) of the number of inputs per second that can cause significant errors, detected by both FPCC and ATOMU. As shown in Fig. 7, FPCC surpasses ATOMU in identifying a higher number of error-trigging inputs per second for all 42 functions. On average, FPCC achieves a rate of 253.76 error-triggering inputs per second, while ATOMU manages to detect 18.83 error-triggering inputs per second. In terms of error-triggering inputs, FPCC achieves a 13.47x speedup compared to ATOMU.

RQ2: *In comparison to the state-of-the-art technique, FPCC exhibits an average 2.17x speedup over ATOMU in effectively detecting significant floating-point errors. Moreover, FPCC achieves a 13.47x speedup over ATOMU in terms of the number of error-triggering inputs per second.*

5.3.3 RQ3: How scalable is FPCC in detecting functions with significant errors? High-precision execution results in considerable performance overhead. In a study comparing ATOMU with other high-precision error detection methods across the 88 GSL functions utilized in our benchmark, ATOMU was observed to be 140 times faster than LSGA [Zou et al. 2015] and 1362 times faster than DEMC [Yi et al. 2019] (an enhanced version of EGAT [Yi et al. 2017]). Additionally, BGRT [Chiang et al. 2014] was evaluated against EGAT in a study [Yi et al. 2017], using a subset of 12 GSL functions from our benchmark. However, BGRT failed to detect higher errors and exhibited slower performance compared to EGAT. Considering that FPCC is 2.17 times faster than ATOMU, it also demonstrates superior speed compared to LSGA, EGAT, BGRT, and DEMC across GSL benchmarks based on previous research findings.

FPGen [Guo and Rubio-González 2020] combines high precision with symbolic execution to identify floating-point errors. However, the computational overhead and path explosion associated with symbolic execution can significantly impact the analysis of complex real-world programs, especially numerical library programs with transcendental functions. While FPGen can detect errors across multiple inputs, its focus is limited to precision issues in floating-point addition and subtraction operations. Hence, we conducted a comparison between FPCC and FPGen using 21

Table 3. FPCC vs FPGen over 21 functions with multiple inputs.

Fid	Benchmarks	LOC	FP Params	Relative Error		Time(s)	
				FPCC	FPGen	FPCC	FPGen
0	recursive_summation	10	32	**9.00E+00**	1.00E+00	100	7200
1	pairwise_summation	30	32	**3.00E+00**	1.32E-16	100	7200
2	compensated_summation	13	32	**1.00E+00**	**1.00E+00**	100	7200
3	sum	8	4	**1.00E+00**	**1.00E+00**	100	7200
4	2norm	9	4	**1.76E-16**	0.00E+00	100	7200
5	1norm	8	4	1.57E-16	2.21E-16	100	7200
6	dot	8	8	**1.10E+00**	1.92E-04	100	7200
7	conv	20	8	**3.07E+00**	2.04E-04	100	7200
8	mv	15	20	**1.16E+00**	8.94E-04	100	7200
9	mm	16	32	**2.30E-05**	2.58E-14	100	7200
10	LU	98	16	1.04E+00	**2.73E+00**	100	7200
11	QR	62	16	**1.00E+00**	2.59E-14	100	7200
12	wmean	23	8	**8.52E+01**	1.00E+00	100	7200
13	wvariance_m	49	8	**6.81E-01**	7.63E-02	100	7200
14	wvariance_w	49	8	**8.71E-01**	2.85E-12	100	7200
15	wsd_m	78	8	**3.22E-01**	3.74E-02	100	7200
16	wsd_w	51	8	**6.49E-07**	1.14E-12	100	7200
17	wtss_m	49	8	**9.24E-01**	4.45E-16	100	7200
18	wabsdev_m	21	8	4.94E-01	**1.00E+00**	100	7200
19	wkurtosis_m	132	8	8.19E+00	**2.57E+01**	100	7200
20	wkew_m	128	8	**7.05E+00**	1.77E-12	100	7200

benchmarks from FPGen [Guo and Rubio-González 2020]. As shown in Table 3, FPCC was able to identify significant floating-point errors in 17 of the 21 programs, which is a greater number than the 9 programs where FPGen detected errors. It is worth noting that in 17 of the benchmarks, FPCC reported higher error magnitudes. Furthermore, FPCC 's experimental runtime was 100 seconds, compared to FPGen's 7200 seconds, suggesting that FPCC operates at a speed that is approximately 72 times faster than FPGen.

RQ3: *In comparison to the state-of-the-art technique, FPCC 's comparison revealed its superior detection capability across multiple-input benchmarks, identifying more significant errors than FPGen in the majority of cases.*

5.3.4 RQ4: How stable is FPCC in detecting functions with significant errors? We conducted 100 repeated experiments to measure the stability of FPCC. Fig. 8 displays box plots representing log2 value of the number of error-triggering inputs detected by FPCC and ATOMU for each function across the 100 trials. Fig. 9 displays box plots representing the ULP errors of rank-1 inputs detected by FPCC and ATOMU for each function across the 100 runs. For the majority of functions, the number of error-triggering inputs and the ULP errors of the rank-1 inputs reported by FPCC remain consistent and stable across the repeated trials. The box plots in Fig. 8 and Fig. 9 provide valuable insights into the consistency and stability of FPCC.

To further assess the stability, we calculated the coefficient of variation (CV) for the 100 results. For the number of error-triggering inputs, the average CV value of FPCC is 1.59e-16, while ATOMU is 0.064. For the ULP errors of rank-1 inputs, the average CV values of FPCC and ATOMU are 2.03e-16 and 0.257, respectively. Both results indicate that FPCC has good stability.

RQ4: *FPCC demonstrates good stability in detecting error-triggering inputs, as evidenced by an average coefficient of variation (CV) value of 1.59e-16 for the number of error-triggering inputs and 2.03e-16 for the ULP errors of rank-1 inputs.*

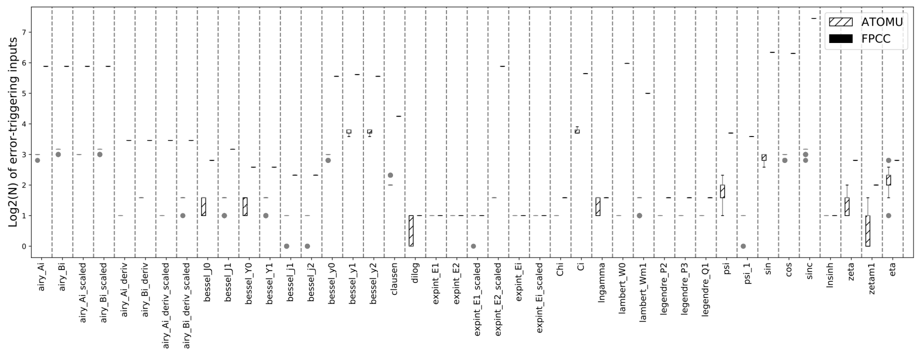


Fig. 8. Box plots representing the log2 value of the number of error-triggering inputs detected by FPCC and ATOMU for each function across the 100 runs.

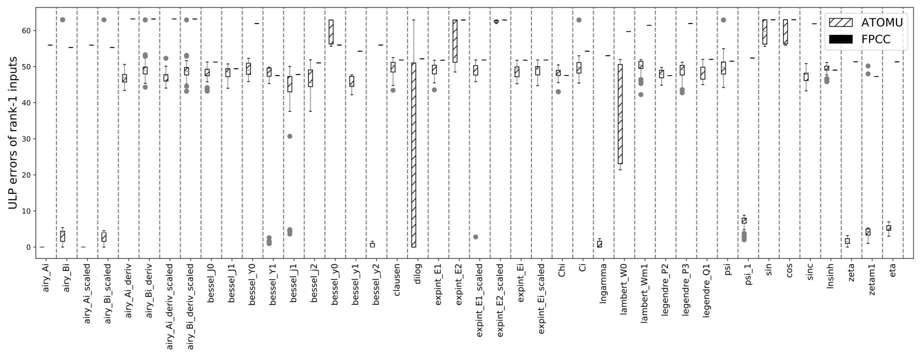


Fig. 9. Box plots representing the ULP errors of rank-1 inputs detected by FPCC and ATOMU for each function across the 100 runs.

5.4 Interesting Bugs Found

We have utilized FPCC to trace and localize the source code of errors in the 42 GSL functions that exhibit significant errors. Table 4 presents the bugs, example codes, number of functions[4], and representative function names associated with these errors. We have identified four main classes of bugs contributing to these significant errors in GSL functions. We provide a localized example code

[4]the sum number is 43 due to *sinc* includes two types bugs that introduced by different branches

Table 4. Results on discovered bugs.

Classes of bugs	Example code	# func	Representative GSL function
Inaccuracy reduction	((x - y * P1) - y * P2) - y * P3; (trig.c 275)	12	airy_Ai, airy_Ai_scaled, bessel_j0, bessel_y1, bessel_y2, sinc, airy_Bi
Danger zone	log(eps);(trig.c 450)	5	airy_Ai_deriv, lnsinh
Inaccuracy sin(M_PI*x)	sin(M_PI * ax); (psi.c 757)	6	psi, psi_1, sinc, zeta, zetam1, eta
Bad cancellation	0.5*x*(5.0*x*x - 3.0); (legendre_poly.c 95)	20	bessel_J0, bessel_Y1, bessel_J1, bessel_Y0

for each bug, and the complete source codes for all functions can be found in the appendix. In the following, we will elaborate on the detected bugs.

Inaccuracy reduction. In Table 4, Column 2 provides the example code for inaccuracy reduction. This code is used to reduce a positive input x to the interval $[0, \pi/4]$. The value of y is obtained by $floor(x/(0.25 * M\_PI))$, while $P1$, $P2$, and $P3$ are precomputed splits of $\pi/4$ to ensure a precision of approximately 75 bits. It is important to note that this reduction method, derived from *fdlibm* in *netlib*[5], only works for inputs less than 823549.6645. To address the inaccuracy bug, larger inputs require higher precision π and using the Payne-Hanek reduction technique to reduce them into the range $[0, \pi/4]$. Interestingly, this bug was initially reported in the bessel_j0 function on April 11, 2012, as *bug#36152*[6] in GSL's bug list. Subsequently, it was discovered that other functions also exhibit the same bug, and the source code of the bug was finally localized on August 13, 2015, as *bug#45746*[7].

Danger zone. The occurrence of danger zone bug is difficult to avoid as it is inherent to mathematical functions. For instance, the use of $log(eps)$ can introduce a significant error when eps is very close to 1. This is problematic because 1 is a well-known danger zone for the log function where the condition number of $log(x)$ is given by $|1/log(x)|$, and as x approaches 1, $|1/log(x)| \to \infty$. This means that even if the error introduced by x is very small, the relative error will be significantly amplified by the condition number, as shown in Formula 4. To address this bug, there are possible suggestions such as using high precision or employing piece-wise polynomial approximation methods [Yi et al. 2019].

Inaccuracy sin(M_PI*x). The bug occurs when $sin(M\_PI * x)$ fails to accurately represent $sinpi(x)$. For instance, when $x = 100$, $sin(M\_PI * 100) = 1.964386723728472e - 15$, whereas $sinpi(100) = 0$. It is important to note that this bug not only affects the accuracy of the results but also causes functional issues. For example, according to the definition of function psi, a domain error will occur if x is positive or negative. This is implemented in the code snippet of gsl_sf_psi_e as below:

```
double s = sin(M_PI*x);
if(fabs(s)<2.0*1.4916681462400413e-154){
    DOMAIN_ERROR( resutls );
}
...
```

[5]http://www.netlib.org/fdlibm/e_rem_pio2.c
[6]https://savannah.gnu.org/bugs/?36152
[7]https://savannah.gnu.org/bugs/?45746

However, for $x = 100$, the condition inside the if statement is not met due to the inaccuracy of $sin(M\_PI * 100)$, resulting in a functional problem with the function.

Bad cancellation. The bad cancellation bug is a well-known accuracy bug in numerical programs [Benz et al. 2012]. Table 4 demonstrates that half of the GSL functions listed in the table encounter errors due to bad cancellations. To mitigate this problem, one can employ high-precision calculations or optimize the implementation used in the numerical program. For the example code in Table 4, the bad cancellation occurs in the subtraction operation of $5.0 * x * x - 3.0$ when the input value of x is *7.745966692414834e-01*. This leads to $5.0 * x * x = 3.0$ with a small error *2.11e-16*. Due to this small error, the result of $5.0 * x * x - 3.0$ becomes 0.0, resulting in a relative error of 1.0. To fix this bug, we can replace the expression with $fma(5.0 * x, x, -3.0)$, utilizing the fused multiply-add (fma) instruction to avoid the small error *2.11e-16*. Alternatively, other methods such as error-free transformation [Muller et al. 2009], high precision calculations [Fousse et al. 2007], or piece-wise polynomial approximation [Yi et al. 2019] can be employed to address the problem.

6 Discussion

Error free transformations. Error-free transformations (*a.k.a* efts) utilize a sequence of standard floating-point operations to evaluate the rounding error of primitive floating-point operation (*i.e.*, +, −, ∗, /). EFTSantizer [Chowdhary and Nagarakatte 2022] employs error-free transformations as shadow execution to detect numerical errors. Both FPCC and EFTSantizer do not rely on expensive high-precision calculations. However, for non-primitive operations (such as the mathematical functions sine and logarithm), depends on n high-precision like as for provide support, while FPCC can leverage exploit the condition numbers of non-primitive operations analyze reason the propagation of rounding errors.

EFTSantizer aims to approximate the final rounding error of an input, while our tool, FPCC , focuses on identifying inputs that trigger significant rounding errors. As illustrated in Table 1, all reported rank-1 inputs of achieve are accuracy. Specifically, *i.e.*, when evaluating the outputs of the reported inputs against the oracles generated by the arbitrary precision library *mpmath*, all detected inputs of FPCC trigger significant rounding errors. To evaluate the effectiveness of EFTSantizer, we input the reported error-triggering data from FPCC into EFTSantizer and assess whether the estimated error produced by EFTSantizer aligns closely with the error calculated by *mpmath*.

Table 5 presents a comparison of the relative errors between *mpmath* and EFTSantizer. The results that show significant differences between *mpmath* and EFTSantizer are highlighted with a grey background. Out of the 42 functions analyzed, there are 7 functions (7/42, or 16.7%) for which EFTSantizer fails to provide accurate rounding errors, exhibiting wildly different magnitudes compared to those generated by *mpmath*.

In addition, chain conditions are more effective at capturing the trends of error variations and identifying significant errors than EFTS-based detection approaches. As illustrated in Figure 5, the distribution of floating-point errors in small regions is discrete, which makes error-based detection susceptible to local optima. In contrast, chain conditions, grounded in the mathematical concept of condition numbers, display characteristics akin to continuous, smooth mathematical curves with distinct gradients. This allows for faster convergence to optimal solutions.

Precision-specific operations. FPCC mitigates potential oracle issues arising from precision-specific operations in two ways: 1) For basic mathematical functions that involve a significant number of precision-specific operations (such as sine, cosine, exponential functions, etc.), FPCC processes them using mathematical formulas based on condition numbers. For instance, the condition number formula for the sine function, i.e. $x * \cot(x)$, can be directly derived through mathematical semantics. 2) In the case of precision-specific operations [Wang et al. 2016] within GSL functions,

Table 5. Relative error comparison of MPMATH and EFTSantizer.

Index	Function Name	Relative Error		Index	Function Name	Relative Error	
		EFTSantier	MPMATH			EFTSantier	MPMATH
0	legendre_P2	3.81E-02	3.81E-02	21	bessel_J1	1.88E-02	1.79E-01
1	legendre_P3	3.72E-02	3.72E-02	22	bessel_Y0	1.56E-02	7.93E-02
2	legendre_Q1	-2.58E-03	1.28E-02	23	bessel_Y1	-2.97E-02	1.04E-01
3	psi	-5.25E-03	9.89E-01	24	bessel_j1	7.09E-03	2.17E-03
4	psi_1	1.66E-16	1.40E-01	25	bessel_j2	2.04E-02	4.99E-03
5	sin	1.38E+00	2.90E+09	26	bessel_y0	8.68E-01	1.72E+04
6	cos	1.38E+00	7.96E+00	27	bessel_y1	1.28E+00	9.58E+03
7	sinc	3.78E-02	1.00E+00	28	bessel_y2	8.49E-01	1.46E+04
8	lnsinh	0.00E+00	2.64E-01	29	clausen	-1.05E-01	9.36E-01
9	zeta	3.90E-16	1.29E-02	30	dilog	1.92E-01	5.52E-01
10	zetam1	5.61E-17	2.26E-03	31	expint_E1	-1.48E-02	2.92E-02
11	eta	7.82E-17	1.53E-02	32	expint_E2	-1.49E-01	2.40E+00
12	airy_Ai	8.78E-01	6.64E+03	33	expint_E1_scaled	-1.48E-02	2.92E-02
13	airy_Bi	1.00E+00	2.89E+09	34	expint_E2_scaled	-5.34E-01	3.01E+212
14	airy_Ai_scaled	9.78E-01	1.40E+04	35	expint_Ei	-3.10E-02	1.11E-02
15	airy_Bi_scaled	1.01E+00	4.91E+09	36	expint_Ei_scaled	4.56E-02	1.41E-01
16	airy_Ai_deriv	-1.26E-16	3.70E-03	37	Chi	5.09E-02	1.28E-01
17	airy_Bi_deriv	-5.53E-01	2.20E-01	38	Ci	1.10E+00	5.74E+02
18	airy_Ai_deriv_scaled	3.67E-18	1.09E-02	39	lngamma	1.85E-01	3.06E-01
19	airy_Bi_deriv_scaled	2.35E-02	1.26E-02	40	lambert_W0	1.00E+00	3.11E-01
20	bessel_J0	-1.91E-03	5.97E-02	41	lambert_Wm1	1.00E+00	1.00E+00

FPCC directly passes the chain condition to the return value for bitwise operations without increasing precision during computation, thereby preserving the semantics of these operations. For rounding operations, such as $(x + n) - n$, which are implemented in GSL functions by invoking the rounding function FPCC directly transmits the chain condition to the return value and does not enter the rounding function. Reduction operations represent another common category of precision-specific operations in GSL, exemplified by the expression $((x - y * P1) - y * P2) - y * P3$. As described in §6 that $P1$, $P2$, and $P3$ are 75-bit precision values of $\pi/4$. This reduction operation is effective within the range of ([-823549.6645, 823549.6645]) using 64-bit precision. However, as the input values increase, the accuracy of the reduction result diminishes, and for this operation, chain conditions escalate with larger input values, aligning with the distribution of errors. Utilizing high precision to directly compute the reduction operation could compromise the semantics of the original operation and may fail to identify inputs that genuinely trigger floating-point errors.

FPCC's speedup over ATOMU. FPCC and ATOMU were executed on the same single-core machine without any concurrent operations, ensuring a fair evaluation of FPCC's speedup compared to ATOMU. We believe FPCC is faster than ATOMU for three reasons: (1) **Search Granularity:** ATOMU must search for an input that triggers a large condition number for each unstable floating-point operation (e.g., +, -, sin) sequentially. In contrast, FPCC focuses solely on the condition number of the final output, significantly reducing the number of search iterations. (2) **Algorithm Differences:** ATOMU employs a genetic algorithm that requires multiple samplings and executions per iteration. FPCC, on the other hand, combines two algorithms: a direct algorithm for the coarse-grained identification of input ranges that may trigger high floating-point errors, followed by a line search algorithm for fine-grained pinpointing without the need for multiple samplings. Consequently, FPCC requires fewer samples overall. (3) **Guiding Functions:** The consistency in chain conditions and function error distribution, along with inherent properties of function variation, enables FPCC to effectively utilize the gradient of chain conditions as a guide to identify inputs that trigger significant errors.

Correlation between chain conditions and floating-point errors. According to Formula 4, the floating-point relative error is approximately the product of the condition number and the backward error. Chain conditions serve as an approximation of the function's condition number. While the condition number ($\left|\frac{xf'(x)}{f(x)}\right|$) is theoretically invariant to program execution, the backward error depends on it. Since the backward error varies across numerical programs, there is, in theory, no consistent correlation coefficient that links chain conditions to floating-point errors.

During our experimental investigations, we observed a correlation between condition numbers and floating-point errors, which led us to establish a threshold. For example, with a function output of 1, a relative error of 1e-3 corresponds to a condition number of approximately 5e12. To mitigate the impact of backward error, we empirically adjusted this value by a factor of 20, resulting in a threshold of 1e14. Setting the threshold too high or too low could result in undetected errors. Theoretically, we cannot provide a definitive correlation coefficient for this threshold. However, based on empirical data, we have established a threshold that has been validated as effective for multi-input programs. Users are advised to initially use the current threshold and may consider lowering it if a smaller relative error is necessary.

7 Related Work

In this section, we review related work across several areas, including static error-bound analysis, floating-point error detection, and the generation of error-triggering inputs.

Static error-bound analysis. Many approaches have been proposed to analyze the upper bounds of floating-point errors [Blanchet et al. 2003; Chen et al. 2021, 2020; Constantinides et al. 2021; Darulova and Kuncak 2014; Das et al. 2020; Daumas and Melquiond 2010; de Dinechin et al. 2011; Haller et al. 2012; Solovyev et al. 2018; Titolo et al. 2018]. These approaches often abstract the program and employ standard program analysis techniques, such as affine arithmetic, interval arithmetic, symbolic reasoning, and abstract interpretation, to infer upper bound errors. LEE *et al.* [Lee et al. 2016, 2017] reduce the computation of error-bound to an optimization problem, which they solve using off-the-shelf computer algebra systems. Their tool fully supports bit-manipulation operations and has been applied to verify the correctness of transcendental functions. In contrast to these static analyses, our approach is dynamic and focuses on generating error-triggering inputs rather than providing formal guarantees regarding the correctness of floating-point results.

Floating-point error detection. FpDebug [Benz et al. 2012], Herbgrind [Sanchez-Stern et al. 2018], FPSanitizer [Chowdhary et al. 2020], and PFPSanitizer [Chowdhary and Nagarakatte 2021] execute programs concurrently using high-precision shadow execution to evaluate rounding errors in operations by comparing the original results with those produced by high-precision computations. However, high precision computations significantly degrade the performance and do not hold on precision-specific operations [Wang et al. 2016]. EFTSanitizer [Chowdhary and Nagarakatte 2022] employs error-free transformations to accelerate shadow execution while utilizing the MPFR library for high-precision elementary functions and LLVM's intrinsics. Verrou [Févotte and Lathuilière 2016] assesses inaccuracies by perturbing rounding and conducting statistical analyses. Bao and Zhang [Bao and Zhang 2013] propose a runtime monitoring method that tracks the propagation of cancellation errors, which can alter the values of integers or booleans. These approaches focus on the error generated during the execution of a specific input, while our approach aims to generate inputs that can trigger significant floating-point errors.

Generation of floating-point error-triggering input . Many black-box search heuristics have been proposed to identify error-triggering inputs. BGRT [Chiang et al. 2014] performs binary guided random search that focuses on tighter input configurations that trigger more significant errors. LGSA [Zou et al. 2015] is a genetic algorithm that derives insights from empirical analysis. EAGT [Yi et al. 2017] utilizes a condition number-guided search. DEMC [Yi et al. 2019] is based on differential evolution and Markov Chain Monte Carlo (MCMC) sampling.

FPGen [Guo and Rubio-González 2020] transforms the challenge of generating high-error-triggering inputs into a code coverage problem, which it addresses through symbolic execution. Unlike the aforementioned approaches that rely heavily on oracles produced by high-precision computations, our method is *oracle-free* and utilizes *chain conditions* to direct the search.

The closest tool to FPCC is ATOMU [Zou et al. 2019], which is also *oracle-free* and employs atomic conditions to effectively identify inputs that trigger significant floating-point errors. However, ATOMU only considers the condition number of individual operations and does not analyze the transitions of condition numbers, which leads to an increased occurrence of false positives. Additionally, ATOMU searches for inputs for every unstable operation, resulting in decreased search efficiency. In contrast, FPCC introduces the concept of chain conditions to capture the propagation and accumulation of floating-point errors, utilizing this approach to guide the search for error-triggering inputs. Compared to ATOMU, FPCC effectively addresses the false positives introduced by atomic conditions and achieves higher search efficiency through the use of chain conditions that capture the interactions among consecutive atomic conditions.

8 Conclusion and Future Work

This paper introduces chain conditions to capture the propagation of floating-point errors and utilizes them to guide the search for inputs that trigger significant floating-point errors. We have implemented our approach in a prototype tool named FPCC and evaluated it on 88 functions from the GNU Scientific Library (GSL). The experimental results are promising. All reported rank-1 inputs from FPCC can trigger significant floating-point errors, whereas only 27% of ATOMU's rank-1 inputs fail to do so. In total, 99.64% (1,049 out of 1,053) of the inputs reported by FPCC can trigger significant errors, while 19.45% (141 out of 723) of the inputs reported by ATOMU can trigger significant errors. Additionally, FPCC achieves a 2.17x speedup compared to ATOMU. Notably, FPCC also outperforms FPGen across benchmarks with multiple inputs. The future of work encompasses two key aspects: 1) Enhancing the tool to support complex inputs, such as matrices and vectors; 2) Enhancing tools to support parallel programs, such as Message Passing Interface (MPI) programs.

Data-Availability Statement

The artifact is publicly available [Yi et al. 2024]. It includes source code files, scripts, and a Docker image designed to reproduce the experimental results presented in Section 5.

Acknowledgments

We would like to express our gratitude to the anonymous reviewers for their insightful comments. This work is supported by the National Key R&D Program of China (No. 2022YFA1005101), and the National Natural Science Foundation of China (Nos. 62202488, 62272471, 62032024, 62474196).

References

Tao Bao and Xiangyu Zhang. 2013. 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 &amp;*

Applications (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 817–832. https://doi.org/10.1145/2509136.2509526

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 453–462. https://doi.org/10.1145/2254064.2254118

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

Jing Chen, Jianbin Fang, Weifeng Liu, and Canqun Yang. 2021. BALS: Blocked Alternating Least Squares for Parallel Sparse Matrix Factorization on GPUs. *IEEE Trans. Parallel Distributed Syst.* 32, 9 (2021), 2291–2302. https://doi.org/10.1109/TPDS.2021.3064942

Zhaoyun Chen, Wei Quan, Mei Wen, Jianbin Fang, Jie Yu, Chunyuan Zhang, and Lei Luo. 2020. Deep Learning Research and Development Platform: Characterizing and Scheduling with QoS Guarantees on GPU Clusters. *IEEE Trans. Parallel Distributed Syst.* 31, 1 (2020), 34–50. https://doi.org/10.1109/TPDS.2019.2931558

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2555243.2555265

Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 731–746. https://doi.org/10.1145/3385412.3386004

Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel shadow execution to accelerate the debugging of numerical errors. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 615–626. https://doi.org/10.1145/3468264.3468585

Sangeeta Chowdhary and Santosh Nagarakatte. 2022. Fast shadow execution for debugging numerical errors using error free transformations. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1845–1872. https://doi.org/10.1145/3563353

George A. Constantinides, Fredrik Dahlqvist, Zvonimir Rakamaric, and Rocco Salvia. 2021. Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 626–650. https://doi.org/10.1007/978-3-030-81688-9_29

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet rigorous floating-point error analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 51. https://doi.org/10.1109/SC41405.2020.00055

Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37, 1, Article 2 (jan 2010), 20 pages. https://doi.org/10.1145/1644001.1644003

Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. 2011. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.* 60, 2 (2011), 242–253. https://doi.org/10.1109/TC.2010.128

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, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

François Févotte and Bruno Lathuilière. 2016. VERROU: a CESTAC evaluation without recompilation. In *International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*. Uppsala, Sweden.

J. M. Gablonsky and C. T. Kelley. 2001. A Locally-Biased Form of the DIRECT Algorithm. *J. of Global Optimization* 21, 1 (sep 2001), 27–37. https://doi.org/10.1023/A:1017930332101

Hui Guo, Ignacio Laguna, and Cindy Rubio-González. 2020. PLINER: Isolating Lines of Floating-Point Code for Compiler-Induced Variability. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00053

Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA*

2018). Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/3213846.3213862

Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1261–1272.

Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1261–1272. https://doi.org/10.1145/3377811.3380359

Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. IEEE, 131–140.

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027

IEEE-754. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

D. R. Jones, C. D. Perttunen, and B. E. Stuckman. 1993. Lipschitzian Optimization without the Lipschitz Constant. *J. Optim. Theory Appl.* 79, 1 (oct 1993), 157–181.

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* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 70–84. https://doi.org/10.1145/2908080.2908107

Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On automatically proving the correctness of math.h implementations. *Proc. ACM Program. Lang.* 2, POPL, Article 47 (dec 2017), 32 pages. https://doi.org/10.1145/3158135

Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 48:1–48:13.

Jorge J. Moré and David J. Thuente. 1994. Line Search Algorithms with Guaranteed Sufficient Decrease. *ACM Trans. Math. Softw.* 20, 3 (sep 1994), 286–307. https://doi.org/10.1145/192115.192132

Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2009. *Handbook of Floating-Point Arithmetic* (1st ed.). Birkhäuser Basel.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Kevin Quinn. 1983. Ever had problems rounding off figures. This stock exchange has. The Wall Street Journal.

Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 256–269.

Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017*. IEEE Computer Society, 229–238.

Robert Skeel. 1992. Roundoff error and the Patriot missile. In *SIAM News (1992)*.

Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (dec 2018), 39 pages. https://doi.org/10.1145/3230733

David E. Stewart and Zbigniew Leyk. 1994. Meschach Library. Retrieved March 30, 2024 from https://www.netlib.org/c/meschach/

Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. 2018. An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and fixing precision-specific operations for measuring floating-point errors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2950290.2950355

Wikipedia. 2024. Ariane 5 flight 501. http://en.wikipedia.org/wiki/Ariane_5_Flight_501.

Frank Wilcoxon. 1992. *Individual Comparisons by Ranking Methods.* Springer New York, New York, NY, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 11–20. https://doi.org/10.1109/APSEC.2017.7

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proc. ACM Program. Lang.* 3, POPL, Article 56 (jan 2019), 29 pages. https://doi.org/10.1145/3290369

Xin Yi, Hengbiao Yu, Liqian Chen, Xiaoguang Mao, and Ji Wang. 2024. *FPCC: Detecting Floating-Point Errors via Chain Conditions (Paper Artifact)*. https://doi.org/10.5281/zenodo.13618683

Daming Zou, Yuchen Gu, Yuanfeng Shi, MingZhe Wang, Yingfei Xiong, and Zhendong Su. 2022. Oracle-free repair synthesis for floating-point programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 159 (oct 2022), 29 pages. https://doi.org/10.1145/3563322

Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A genetic algorithm for detecting significant floating-point inaccuracies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 529–539.

Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.* 4, POPL, Article 60 (dec 2019), 27 pages. https://doi.org/10.1145/3371128

Received 2024-04-06; accepted 2024-08-18