Automated Repair of High Inaccuracies in Numerical Programs

Xin Yi, Liqian Chen, Xiaoguang Mao, Tao Ji
College of Computer Science, National University of Defense Technology
Changsha, China
Email: {yixin09, lqchen, xgmao, taoji}@nudt.edu.cn

***Abstract*—Rounding errors are introduced pervasively when using floating-point arithmetic to approximate real arithmetic. The accumulation or catastrophic cancellation of rounding errors in numerical programs may produce high inaccuracy results, which can cause serious software failures once being triggered. High inaccuracies are known hard to debug and fix manually for developers. Hence, the automated techniques are desired for solving the high inaccuracy problem. In this paper, we propose a novel framework for automated repair of high-inaccuracy bugs in numerical programs. The framework includes the phases of detecting high-inaccuracy bugs, localizing the buggy code, generating and validating the patches, and synthesizing the repaired program at last. Based on this framework, we develop a prototype tool for repairing high inaccuracies in numerical programs. Our preliminary experimental results are encouraging.**

***Index Terms*—automated repair, floating-point, numerical code, dynamic analysis**

I. Introduction

Many areas even safety-critical areas (such as avionics, aerospace, automotives, medical devices, etc.) use floating-point arithmetic to implement mathematical calculations, and rounding errors are inevitable due to the approximation. The accumulation or catastrophic cancellation of rounding errors in numerical programs may produce high inaccuracy results and have led to the confusion in stock index[1], the sinking of offshore platform [2] and the failure of missile interception [3].

The high inaccuracies hidden in the program are difficult to detect and debug for programmers [4][5]. The rounding errors that lead to high inaccuracies can be far away from the location where the failure occurs. Furthermore, the developers without deep understanding of floating-point arithmetic can not eliminate the high inaccuracies correctly even when the locations of errors are found.

Therefore, automated repair techniques [6][7][8] are highly desired for solving the high inaccuracy problem in numerical programs. However, to automatically repair high-inaccuracies, there exist several challenges. First, the search space of finding inputs that trigger high inaccuracies in numerical programs is huge, due to the large range of the floating-point representation. Second, rounding errors are propagating and accumulating during the execution of numerical programs, and thus it is hard to identify the main source causing high inaccuracies. Third, automated repair techniques need to generate candidate patches that improve the accuracy of the buggy code but without changing its original semantics.

In this paper, we treat high inaccuracies as so-called high-inaccuracy bugs and propose a novel framework to automatically repair high-inaccuracy bugs in numerical programs. Our framework consists of four phases: detecting high-inaccuracy bugs, localizing the buggy code, generating and validating the patches, and synthesizing the repaired program at last. Based on this framework, we have developed an automatic repair prototype tool called AutoFP. We evaluate AutoFP on programs drawn from FPBench [9], which is a benchmark for floating-point analysis. Our results demonstrate that AutoFP can automatically repair high-inaccuracy bugs in those cases.

The rest of the paper is organized as follows. Section II reviews background. Section III gives an overview of our approach. Section IV introduces the details of our framework. Section V shows and analyzes the experimental results. Section VI discusses related work. Section VII concludes.

II. Background

A. Absolute and Relative Error

The absolute error and relative error are two common indicators to evaluate the error of floating-point program. For a real function $f(x)$, we use $f_p(x)$ to denote its corresponding floating-point program. Formulas (1)(2) respectively show the absolute error and relative error. For the relative error, we add a δ (a small positive value) to prevent division by zero.

$$Absolute_error = |f(x) - f_p(x)| \tag{1}$$

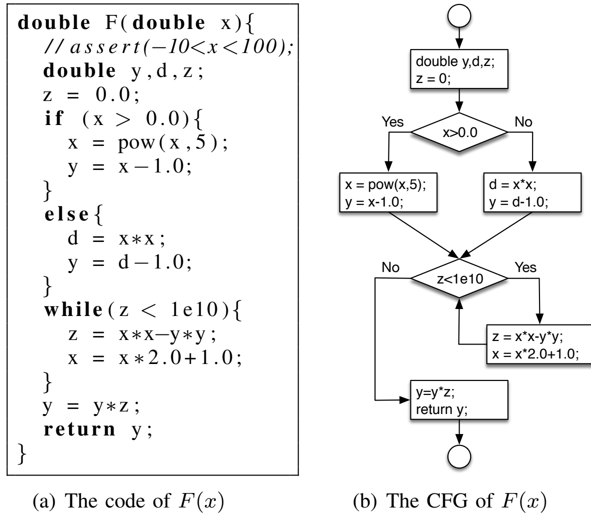
$$Relative_error = \frac{|f(x) - f_p(x)|}{max(|f(x)|, \delta)} \tag{2}$$

B. High-inaccuracy Bug

Definition 1. *In numerical programs, given a positive threshold of an input x, if the relative error between real arithmetic output $O_r(x)$ (i.e., mathematical output) and floating-point output $O_f(x)$ is greater than or equal to the threshold ε, we say that the input x triggers a* high-inaccuracy bug*, i.e.,*

$$\frac{|O_r(x) - O_f(x)|}{max(|O_r(x)|, \delta)} \geq \varepsilon \ (\varepsilon > 0.0) \tag{3}$$

Note that ε is a threshold for determining the high-inaccuracy bug. The value of ε can come from the accuracy requirements of customers or the specifications of numerical programs.



(a) The code of $F(x)$ (b) The CFG of $F(x)$

Fig. 1. The code and CFG of numerical program $F(x)$

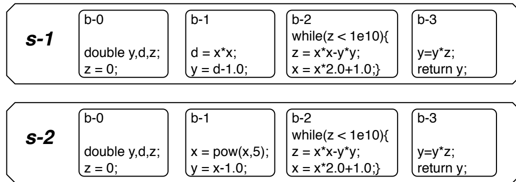


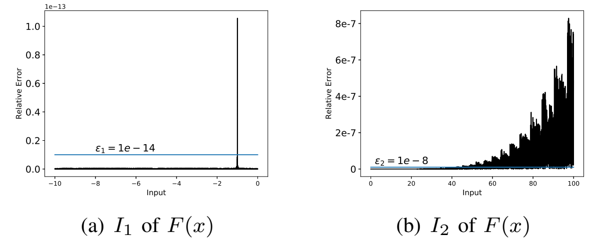
Fig. 2. The slices and blocks of $F(x)$

III. Overview

In this section, we use a numerical program $F(x)$, which includes catastrophic cancellation and accumulation of rounding errors, to illustrate our approach for repairing high-inaccuracy bugs. As shown in Fig. 1(a), the numerical program $F(x)$ contains loops and conditional statements, and it is difficult for programmers to inspect where high-inaccuracy bugs are hidden. To repair high-inaccuracy bugs in $F(x)$, our framework consists of four phases:

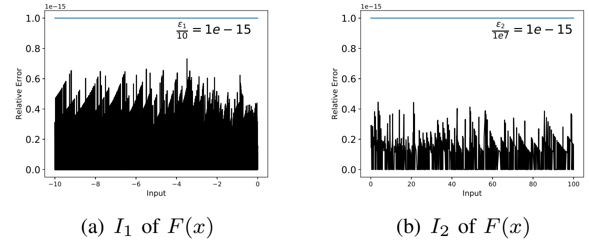
1) ***Detecting high-inaccuracy bugs****:* According to the condition statement and the input domain of $F(x)$, we divide the input domain into two parts: $I_1 = \{-10.0 < x <= 0.0\}$ and $I_2 = \{0.0 < x < 100.0\}$. Then, we generate sample points inside each part. For each sample point, we use shadow value execution (described in Section IV) to obtain real output, and calculate the relative error (according to formula (2)). After that, we can get the distribution of relative errors over those sample points as shown in Fig. 3. We set the threshold $\varepsilon_1 = 1e-14$ for Fig. 3(a) and $\varepsilon_2 = 1e-8$ for Fig. 3(b). Then, we get two small input intervals that trigger high-inaccuracy bugs: $x \in [-1.0042, -0.9982]$ for I_1 and $x \in [39.5303, 100.0000]$ for I_2.

2) ***Localizing buggy code****:* We localize the buggy code by extracting program slices that correspond to those input intervals. Furthermore, as shown in Fig.2, we divide those program slices into blocks based on the control-flow graph of the original program. We use the input that triggers the highest relative error among sampling inputs to calculate the relative errors that are introduced by each block, and we rank those blocks according to the relative errors they introduce (from large to small).



(a) I_1 of $F(x)$ (b) I_2 of $F(x)$

Fig. 3. Before repairing: error distribution of $F(x)$



(a) I_1 of $F(x)$ (b) I_2 of $F(x)$

Fig. 4. After repair: error distribution of $F(x)$

3) ***Generating and validating patches****:* First of all, we extract and compose floating-point expressions symbolically from the block ranked first in the block list. For example, for the block *b-1* in slice *s-1*, we get the floating-point expression: $x * x - 1.0$ after replacing d with $x * x$ in $d - 1.0$. Then, we use the mathematically equivalent transformation to change $x * x - 1.0$ to $(x - 1.0) * (x + 1.0)$, and then validate the expression by directly testing it on the input interval $[-1.0042, -0.9982]$. After testing, we find that the accuracy of expression $x * x - 1.0$ is improved after transformation over the given input interval. Next, we insert the transformed expression back to its block and re-evaluate the whole program slice over the input interval. After the evaluation, we find that the accuracy of the whole program slice is also improved over the input interval, and thus the previous high inaccuracies are reduced. At last, we get a valid patch. Similarly, a valid patch for program slice *s-2* is also found, where the expression $x * x - y * y$ in *b-2* of *s-2* is transformed to $(x - y) * (x + y)$.

4) ***Program synthesis and simplification****:* After obtaining the valid patches, we insert those patches back into the original program by program synthesis. The synthesized program is shown in Fig. 5(a). Furthermore, if we retest the patches for this example over the whole input domain $[-10, 100]$ rather than over the small input intervals triggering high-inaccuracy bugs, we find that no high-inaccuracy bugs are introduced, and thus we can simplify the repaired program to the version shown in Fig 5(b). The distributions of relative error after repair are depicted in Fig 4, where the maximum relative error is below the $1e-15$.

```
double F(double x){
  //assert(-10<x<100);
  double y,d,z;
  z = 0.0;
  if (x > 0.0){
    x = pow(x,5);
    y = x-1.0;}
  else{
  if ((x >= -1.0042)
      &&(x <= -0.9982))
         {
     d = x*x;
     y=(x-1.0)*(x+1.0);
    }else{
      d = x*x;
      y = d-1;}}
  if ((x>=35.5303)
    &&(x<=100)){
    while(z < 1e10){
      z = (x-y)*(x+y);
      x = x*2.0+1.0;}
  }else{
    while(z < 1e10){
      z = x*x-y*y;
      x = x*2.0+1.0;}}
  y = y*z;
  return y;
}
```

(a) Before simplification

```
double F(double x){
  //assert(-10<x<100);
  double y,d,z;
  z = 0.0;
  if (x > 0.0){
    x = pow(x,5);
    y = x-1.0;
  }
  else{
    d = x*x;
    y=(x-1.0)*(x+1.0);
  }
  while(z < 1e10){
    z = (x-y)*(x+y);
    x = x*2.0+1.0;
  }
  y = y*z;
  return y;
}
```

(b) After simplification

Fig. 5. The synthesized program for repairing $F(x)$

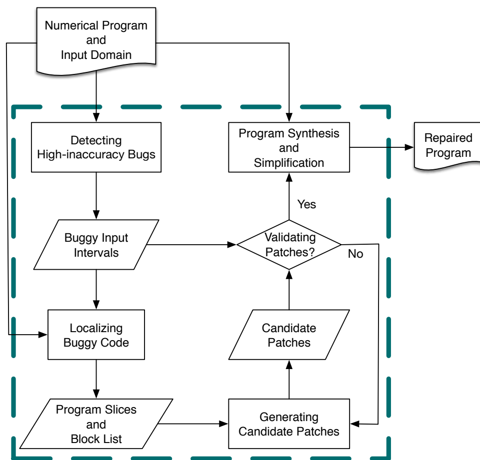


Fig. 6. The workflow of our framework for automated repair of high-inaccuracy bugs

IV. A Framework for Automated Repair of High Inaccuracies

In this section, we describe our framework for automated repair of high inaccuracies. Fig. 6 depicts the workflow of our framework. There are four main phases: detecting high inaccuracies, localizing buggy code, generating and validating the patches, synthesizing the repaired program. In the following subsections, we detail these four phases.

A. Detecting High-inaccuracy Bugs

The goal of this phase is to find the input intervals that cause high-inaccuracy bugs. This phase takes the numerical program and its input domain as input. The input domain is sampled and the relative error for each sample point is calculated. When the input triggering high inaccuracy is found, we continue to find inputs that can trigger high inaccuracy near this input, and finally use a sufficiently small interval to include those inputs in order to get an input interval that triggers high inaccuracies.

To achieve the goal, two critical issues need to be addressed. One is how to calculate the real arithmetic result to calculate the relative error. A common solution is to use shadow value execution, a technique often used in dynamic floating-point analysis [10][11]. The main idea is to execute a program P under the given precision and also maintain a higher precision execution on the program. After the execution, two outputs are produced and can be used to calculate the relative error (see Formula (2)). However, the higher precision execution may introduce extra errors for some precision-specific operations, e.g., the implementation of many functions in C math library is implemented specifically for 64 bit floating point number, while using the 128 or higher bit may break the original semantic and introduce extra error [12].

Another is how to efficiently find the input intervals that trigger high-inaccuracy bugs. In Section III, we use random samples for the small input domain $[-10, 100]$ and find high-inaccuracy bugs. However, when the input domain of a numerical program is large, the method of random search with limited number of sampled points may not be able to find high-inaccuracy bugs. In this situation, we need a good search algorithm to find the buggy input intervals. There are two recent choices to trigger the high inaccuracies for the numerical program: Locality-Sensitive Genetic Algorithm (LSGA) [13] and Binary Guided Random Testing (BGRT) [11].

B. Localizing Buggy Code

In this phase, we extract the program slice from the whole execution path that produces the high-inaccuracy bug and divide the slice into blocks based on the control flow graph of the original program.

The reason why we localize the high-inaccuracy bug inside a whole program slice lies in that floating-point errors are non-local, which means that the source causing the high-inaccuracy bug may be far away from the statement that outputs the high-inaccuracy results. For example, in Section III, with respect to the slice *s-1*, the source of the high-inaccuracy is due to the statement $y = d - 1$, while the statement $y = d - 1$ and the output statement $y = y * z$ are separated by a loop.

The blocks in the program slice are ranked according to the values of relative errors introduced in each block (from large to small). We only consider the relative error of each block that are related to those variables directly connected to the expression that produces output of the program. For example, in Section III, for the expression $y * z$ that produces the final output of y, the relative error of the block *b-1* in slice *s-1* is measured according to the relative error of variable y whose value is propagated to $y * z$.

C. Generating and Validating Patches

We first try to generate patches inside a single block. If those patches can not repair the program, we then try to generate patches crossing multiple blocks. The patches are validated through regression testing over the input interval found in the detecting phase.

In detail, for the case of a single block, we first choose a block from the block list and compose the floating-point expressions that may influence the output of the numerical program. For example, in Section III, we symbolically calculate the expression $d - 1$, which result in expression $x * x - 1$ in block *b-1* of *s-1*. The expression $x * x - 1$ provides more information for producing patches. Then, we try to transform the floating-point expressions by rewriting rules, including the commutativity, associativity, and other laws of basic real arithmetic. For example, we transform $x*x-1$ into $(x - 1) * (x + 1)$ by factorization. After the transformation, we insert the expression back to the program slice and retest it on the input interval to validate the patch. A patch is valid if the relative errors during retesting become less than ε in the input interval.

For the case of acrossing multiple blocks, we connect adjacent blocks as a bigger block to generate patches. It is worth noting that for the block includes a loop, currently, we only optimize the body of a loop, and more general methods could consider acrossing different loop iterations by loop unrolling.

D. Program Synthesis and Simplification

In this phase, we apply the patch into the original program. To be more clear, we insert a new branch for the patch to make sure that the program executes the code of patch only in the input interval identified by the detecting phase. For example, in Fig.5(a), the patch is inserted in the program under the condition $(x \geq -1.0042)\&\&(x \leq -0.9982)$. We could further simplify the synthesized program by trying to remove the branch condition restriction. In other words, we try to replace the original program slice with the new synthesized patch without creating a new branch. E.g., in Fig. 5(b), we remove the branch condition $(x \geq -1.0042)\&\&(x \leq -0.9982)$ which appears in Fig. 5(a). However, to validate this simplification, we need to retest over the whole input domain and make sure no high-inaccuracy are introduced by this simplification. Note that retesting the patched program on the whole input domain is time-consuming.

V. Experiments

A. Tool and Experimental Setup

Based on the framework described in Section IV, we have developed a prototype tool, called AutoFP, for repairing high inaccuracies in numerical programs. The tool employs FPDebug [10] to perform the shadow value execution and Herbie [14] to transform the floating point expressions. For detecting the high-inaccuracy bugs, we choose the BGRT [11] algorithm to search inputs that will trigger high-inaccuracy bugs. We have conducted experiments on the programs from FPBech [9], which is a benchmark for floating point analysis.

TABLE I
Experimetal results of evaluating AutoFP on the programs from FPbench

Program	Domain	T1(s)	T2(s)	T3(s)	Max. Relative Error	
					BR	AR
frac2	(0,1e5]	120.22	5.06	125.29	1.38E-11	9.33E-17
frac3	(1,200]	75.54	14.87	90.41	4.80E-12	1.46E-16
sqrt2	(0,1e7]	123.71	5.04	128.76	1.43E-09	1.53E-16
sqr2	(0,1e10]	217.94	3.11	221.05	7.87E-07	0.00E+00
rsqrt	(0,700]	93.76	9.58	103.35	2.33E-13	2.64E-16

B. Experimental Results

Table I shows the experimental results. The "Domain" column indicates the input domain of each program. The "T1" column shows the time consumption of detecting high-inaccuracy bugs, "T2" shows the time consumption of generating and validating patches, and "T3" shows the total time. The "Max. Relative Error" includes two sub-columns that respectively show the detected maximum relative error in the numerical program before repair (BR) and after repair (AR).

The results show that the maximum relative error is significantly reduced after repair. The maximum relative error of program $sqr2$ becomes zero after repair is due to the fact that the outputs of repaired program have 64-bit significant digits, which makes the value of relative error (also 64-bit floating point represent) become zero. We also note that the process of detecting high-inaccuracy bugs is time-consuming as shown in "T1" column. This is because the shadow value execution which uses high-precision computation is slow. And a more efficient search algorithm can improve a lot the efficiency of detecting high-inaccuracy bugs. The time of detecting also increases when the size of input domain increases. In the future, we expect a more efficient algorithm to reduce largely the time of detecting high-inaccuracy bugs especially for programs with large input domain. In addition, the time cost on generating and validating patches (i.e., the "T2" column) is not closely related to the input domain and we find that it is closely related to the complexity of the block to repair. More complex blocks enable more kinds of transformation, which means generating more candidate patches and making more validations. Hence, more efficient transformations are desired to quickly generate effective patches, and thereby shorten the repair time.

C. Threats to Validity

Threats to internal validity are related to pervasive rounding errors in our implementation and experiments. The FPDebug tool that AutoFP used may introduce extra errors for some precision-specific operations [12]. We have tried our best to avoid executing those operations. Threats to external validity are related to the generality of our findings. We have conducted our experiments on the programs from FPBench. In the future,

we plan to extend our tool and evaluate our tool on real-world numerical programs. Threats to construct validity correspond to the suitability of our approach to measure the high-inaccuracy bug. We currently use the relative error to measure the high-inaccuracy bug. We would consider measuring the high-inaccuracy bug in other ways, e.g., measure the high-inaccuracy bug based on the unit in the last place (ULP).

VI. Related work

Martel [15] builds an abstract interpretation framework to support the source-to-source transformation for numerical programs. N. Damouche et al. [16][17] expanded his work and developed a tool called Salsa [18], which can improve the accuracy of numerical programs by automatic transformation. Their work is based on static analysis and focuses more on reducing the worst-case bounds of errors, while this paper is based more on dynamic analysis and targets at automated repair of only high-inaccuracies in numerical programs.

FPDebug [10], which is used in AutoFP, is a dynamic analysis tool built on Valgrind for detecting floating-point accurate problem. As a very recent work, Herbgrind [19] extends the approach of FPDebug and can help developers find the root causes of floating point error in numerical programs. Herbgrind would be rather useful to help localizing the buggy code for our approach, and hence in the future we will exploit Herbgrind to enhance AutoFP.

Wang et al. [12] point out an interesting phenomenon that when using shadow value execution to measure floating-point errors, there are a sort of so-called precision-specific operations, for which utilizing higher precision may not result in more precise results. The authors propose an approach to fix this problem by keeping using the original precision (without lifting to higher precision) for those precision-specific operations during the shadow value execution. In this paper, we use their method to handle precision-specific operations during the process of identifying high inaccuracies. Their method focuses on fix the limitation problem of shadow value execution for precision-specific operations, while our paper focuses on automatically repairing programs to reduce high-inaccuracies.

VII. Conclusion and future work

The floating-point error in a numerical program is propagating and accumulating during the execution and may lead to high-inaccuracy results. It is tedious and also difficult for developers to manually debug and fix high inaccuracies in numerical programs. In this paper, we propose a novel framework for automatically detecting, localizing, and repairing high-inaccuracy bugs in numerical programs. Based on this framework, we make use of the existing tools of dynamic floating-point analysis and construct an automated repair prototype tool called AutoFP. We evaluate AutoFP on several benchmark programs and achieve promising results.

For future work, first, we plan to design a more efficient algorithm, such as genetic algorithm [20], to detect high-inaccuracies in numerical programs, especially for those with large input domain. Also, we plan to improve AutoFP and evaluate it on real-world scientific library functions, e.g., the GNU Scientific Library (GSL).

Acknowledgment

This research is supported by the NSFC under Grant Nos.61379054, 61502015, 61672529, and the Open Project of Shanghai Key Laboratory of Trustworthy Computing under Grant No. 07dz22304201504

References

[1] K. Quinn, "Ever had problems rounding off figures," *This stock exchange has. The Wall Street Journal*, p. 37, 1983.
[2] B. Jakobsen and F. Rosendahl, "The sleipner platform accident," *Structural Engineering International*, vol. 4, no. 3, pp. 190–193, 1994.
[3] M. Blair, S. Obenski, and P. Bridickas, "Patriot missile defense: Software problem led to system failure at dhahran," *Report GAO/IMTEC-92-26*, 1992.
[4] W. Kahan, J. D. Darcy, and E. Eng, "How javas floating-point hurts everyone everywhere," in *ACM 1998 workshop on java for high-performance network computing*, p. 81, Stanford University, 1998.
[5] N. Toronto and J. McCarthy, "Practically accurate floating-point math," *Computing in Science & Engineering*, vol. 16, no. 4, pp. 80–95, 2014.
[6] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER, 2016 IEEE 23rd International Conference on*, vol. 1, pp. 213–224, IEEE, 2016.
[7] X.-B. D. Le, D. Lo, and C. Le Goues, "Empirical study on synthesis engines for semantics-based program repair," in *ICSME, 2016 IEEE International Conference on*, pp. 423–427, IEEE, 2016.
[8] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE 2016*, pp. 691–701, ACM, 2016.
[9] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *International Workshop on Numerical Software Verification*, pp. 63–77, Springer, 2016.
[10] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 453–462, 2012.
[11] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors.," in *PPOPP 2014*, pp. 43–52, ACM, 2014.
[12] R. Wang, D. Zou, X. He, Y. Xiong, L. Zhang, and G. Huang, "Detecting and fixing precision-specific operations for measuring floating-point errors," in *FSE 2016*, pp. 619–630, ACM, 2016.
[13] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies.," in *ICSE 2015*, pp. 529–539, IEEE Computer Society, 2015.
[14] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.
[15] M. Martel, "Program transformation for numerical precision," in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 101–110, ACM, 2009.
[16] N. Damouche, M. Martel, and A. Chapoutot, "Numerical accuracy improvement by interprocedural program transformation," in *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pp. 1–10, ACM, 2017.
[17] N. Damouche, M. Martel, and A. Chapoutot, "Improving the numerical accuracy of programs by automatic transformation," *International Journal on Software Tools for Technology Transfer*, pp. 1–22, 2016.
[18] N. DAMOUCHE and M. MARTEL, "Salsa: An automatic tool to improve the numerical accuracy of programs," *6th International Workshop on the Automated Formal Methods*, 2017.
[19] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error with herbgrind," *arXiv preprint arXiv:1705.10416*, May 29,2017.
[20] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *ICSE 2017*, pp. 209–220, IEEE Press, 2017.