The Journal of Supercomputing
https://doi.org/10.1007/s11227-023-05523-6

Hierarchical search algorithm for error detection in floating-point arithmetic expressions

Zuoyan Zhang[1] · Jinchen Xu[1] · Jiangwei Hao[1] · Yang Qu[1] · Haotian He[1] · Bei Zhou[1]

Accepted: 25 June 2023

Abstract
Scientific and engineering applications rely on floating-point arithmetic to approximate real numbers. Due to the inherent rounding errors in floating-point numbers, error propagation during calculations can accumulate and lead to serious errors that may compromise the safety and reliability of the program. In theory, the most accurate method of error detection is to exhaustively search all possible floating-point inputs, but this is not feasible in practice due to the huge search space involved. Effectively and efficiently detecting maximum floating-point errors has been a challenge. To address this challenge, we design and implement an error detection tool for floating-point arithmetic expressions called HSED. It leverages modified mantissas under double precision floating-point types to simulate hierarchical searches from either half or single precision to double precision. Experimental results show that for 32 single-parameter arithmetic expressions in the FPBench benchmark test set, the error detection effects and performance of HSED are significantly better than the state-of-the-art error detection tools Herbie, S3FP and ATOMU. HSED outperforms Herbie, Herbie+, S3FP and ATOMU in 24, 19, 27 and 25 cases, respectively. The average time taken by Herbie, Herbie+, and S3FP is 1.82, 11.20, and 129.15 times longer than HSED, respectively.

Keywords Floating-point arithmetic · Error detection · Dynamic analysis · Hierarchical search

1 Introduction

The floating-point type is an indispensable data type in computers that is widely used in numerical and floating-point programs. These programs are widely used in fields as diverse as human spaceflight, weather forecasting, gene sequencing, and

Zuoyan Zhang and Jinchen Xu these authors contributed equally to this work.

Extended author information available on the last page of the article nuclear explosion simulations. Ensuring security and reliability is a critical issue for floating-point programs. Due to the limited number of bits available in computers, it is impossible to accurately represent infinite real numbers, resulting in inherent floating-point errors [7]. In addition, the gradual propagation and accumulation of floating-point errors in computations can have serious or even catastrophic consequences, such as disrupting stock market indices [1], failing rocket launches [2], and missile trajectories resulting in the death of soldiers [3]. In addition, deep learning systems and probabilistic programming language systems are also susceptible to the effects of floating-point errors. Therefore, the detection of floating-point errors is crucial.

Direct analysis of floating-point programs is difficult and inefficient. Since floating-point programs contain a large number of floating-point operations, and a series of floating-point operations can be abstracted into a series of equivalent floating-point arithmetic expressions [4], error detection for floating-point arithmetic expressions is often much more effective, so this paper focuses on the error detection for floating-point arithmetic expressions.

Methods of error detection include static analysis and dynamic analysis. Static analysis uses the abstract semantics of numerical programs to approximate error bounds and often results in higher than actual errors. Dynamic analysis calculates the actual value under each input and the oracle (correct rounded value) under that input using a high precision library, and compares to obtain the true error. This paper investigates dynamic analysis methods. Many heuristic search algorithms have been developed by domestic and international scholars for dynamic analysis, such as random sampling, atomic-condition-based search (ACES) [5], binary guided random testing (BGRT) [6], Markov Chain Monte Carlo (DEMC) [9] and local sensitive genetic algorithm (LSGA) [11]. Existing heuristic search algorithms have some drawbacks, such as random sampling may miss the input that triggers the maximum error; ACES may generate false positives, i.e. the reported input does not trigger a large error; BGRT requires a predefined search time to guide the search, and the length of the search time directly affects the quality of the results, and so on. Therefore, it is a challenge to investigate an efficient and effective search algorithm.

To address this challenge, this paper designs and implements a hierarchical search algorithm for floating-point arithmetic expressions to search for the input that triggers the maximum error and report the maximum error, and implements the error detection tool HSED (Hierarchical search error detection) based on this algorithm, which enables error detection for floating-point arithmetic expressions at user-specified intervals.

In summary, the main contributions of this paper are as follows:

- We introduce the concept of significant error and the starting search layer is selected based on the presence or absence of significant error to balance detection effectiveness and performance.
- We design and propose a hierarchical search algorithm for half precision to double precision three-layer and single precision to double precision two-layer simulations with modified mantissa length in double-precision floating-point type.
- We design and implement an error detection tool, HSED.

- We extensively evaluate HSED on a selection of 32 single reference arithmetic expressions from the FPBench and the results show significant advantages over Herbie , S3FP and ATOMU.

Section 2 of this paper presents work related to floating-point error detection. Section 3 introduces the basics knowledge of this paper, including floating-point errors, error measures and significant errors. Section 4 introduces the HSED framework. Section 5 details the core HSED algorithm and implementation. Section 6 analyses the experiments performed in this paper and their results. Finally, the paper concludes.

2 Related work

In recent years, a great deal of research has been done on floating-point error detection, and this section mainly discusses the research relevant to our approach.

Many error detection methods explore heuristic search algorithms to find maximum errors, for example ATOMU [5] uses atomic-condition-based search (ACES) to find the inputs that trigger the significant errors, the tool converts the search for larger errors into a search for problems that trigger larger atomic state functions; S3FP [6] employs binary guided random testing (BGRT) to search for maximum errors; AutoRNP [9] uses the Markov Chain Monte Carlo (DEMC) algorithm to detect accuracy flaws, where the DEMC algorithm is based on the Differential Evolutionary (DE) algorithm and Markov Chain Monte Carlo (MCMC) with the aim of searching for the input that triggers the maximum error. The ultimate purpose of AutoRNP is for error repair, and recently Oracle-free error repair methods have also attracted attention [20].

Based on the binary staking tool valgrind [10] can detect errors in floating-point operations, error detection tools based on this tool include FPDebug [13], Verrou [15] and Herbgrind [16]. FPDebug [13] is the first error detection tool to use valgrind and it uses random tests to detect catastrophic errors and rounding errors in floating-point programs; Verrou [15] uses the Monte Carlo algorithm (MCA) to detect floating-point errors; Herbgrind [16] is used to locate the source of errors in large programs by randomly sampling and calculating high precision values using the MPFR high precision library, and helps users to find suspicious floating-point code in large floating-point programs by dynamically tracking the dependencies between program operations and outputs. Both Herbie [19] and Herbgrind [16] are randomly tested for maximum error. Herbie is an error detection and precision optimization tool that first locates error-prone regions by random testing and then improves precision by rewriting expressions in error-prone regions. Herbgrind can be combined with Herbie to improve suspect floating-point code in large floating-point programs by using the expressions generated by Herbgrind as input to Herbie.

Another important branch of error detection is static analysis. It does not require program execution, but instead approximates error bounds through theoretical concepts such as abstract semantic analysis [22], interval operations [23], affine operations [24], and symbolic execution [26], the error results often exceed the actual errors, so static analysis tools [12, 14, 17, 18, 21, 25, 27, 28, 32] attempt to tighten error bounds.

In summary, we find that existing heuristic search algorithms have the following shortcomings, (1) random testing has a high probability of missing the input that triggers the maximum error, (2) binary guided random testing requires a predefined long search time, the length of which has a large impact on the results, and (3) atomic-condition-based search can generate error false alarms, reported inputs do not trigger large errors, etc. Compared to static analysis tools, HSED is dynamic and the maximum error found is not the approximate error. Compared to existing search algorithms, HSED (1) for the first time uses low precision inputs below the precision of the original data type to guide the search, (2) the low precision layer locates the error hotspot interval and the high precision layer densely samples this interval greatly improving effects, and (3) there are no error false alarms.

3 Background knowledge

The main task of this paper is to detect floating-point errors, the following will introduce related concepts and basic knowledge.

3.1 Floating-point representation

Computers use finite-precision floating-point numbers to approximate infinite real numbers. Floating-point operations are also an approximate implementation of real number operations. Many standards for floating-point numbers have been published in the past. The most widely used is the IEEE 754 standard. The IEEE 754 standard specifies that a floating-point number consists of sign, exponent and mantissa bits, the normalized representation is: $f = (-1)^S \times M \times 2^E$, where S is the sign bit and $S \in \{0, 1\}$ occupies one bit to represent the positive or negative of the floating-point number. If $S = 0$, it represents a positive number and if $S = 1$, it represents a negative number, $M = d_0.d_1d_2...d_{p-1}$ represents the mantissa bits, which is the effective number of f, p is the length of the effective number, which represents the precision of the floating-point number. $E = e - bias$ represents the exponent, where e is an unsigned integer of e bits, and $bias = 2^{e-1} - 1$. The most commonly used floating-point types are double and float, and the sign, exponent and mantissa occupy 1 (1), 11 (8), and 52 (23) bits, respectively. Figure 1 shows the float and double formats.

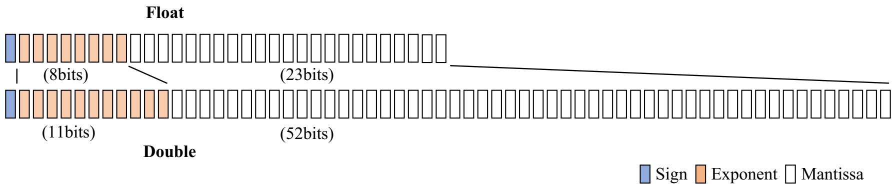


Fig. 1 IEEE 754 common floating-point formats

3.2 Error measurement standard

Floating-point error detection requires the use of error measures. The most commonly used error measures are absolute and relative error, and for floating-point errors there are also ULP and Bits error. Given an arithmetic expression $f(x)$, and assuming that the oracle obtained by the computer at $x = x_0$ is represented by $f(x_0)$, with the computed value represented by $\tilde{f}(x_0)$, the absolute error ($error_{abs}$) and the relative error ($error_{rlt}$) of the expression are defined as shown in Eq. 1 and 2.

$$error_{abs} = \left|\tilde{f}(x_0) - f(x_0)\right|. \tag{1}$$

$$error_{rlt} = \left|\frac{\tilde{f}(x_0) - f(x_0)}{f(x_0)}\right|. \tag{2}$$

Floating-point numbers are non-uniformly distributed, with more numbers concentrated near 0. This non-uniform distribution leads to inconsistency in the measurement of absolute error, and can lead to division by zero errors when measuring relative error. If the result is NaN or $\pm\infty$, neither absolute nor relative error can be effectively calculated. Therefore, the error of floating-point numbers is generally measured by ULP(unit in the last place). In this paper, ULP is defined consistently with Goldberg's definition [8], and for double-precision floating-point numbers, ULP is defined as in Eq. 3.

$$For\ x \in \mathbb{R}, ULP(x) = \begin{cases} 2^{k-52}, \text{if } |x| \in \left[2^k, 2^{k+1}\right), k \in [-1022, 1023] \cap \mathbb{Z}. \\ 2^{-1074}, \text{if } |x| \in \left[0, 2^{-1022}\right). \end{cases} \tag{3}$$

For a real number x, $ULP(x)$ represents the distance between the two closest floating-point numbers to x. The ULP error($error_{ulp}$) of the expression $f(x)$ at $x = x_0$ can be defined as shown in Eq. 4.

$$error_{ulp}(f(x_0)) = \left|\frac{\tilde{f}(x_0) - f(x_0)}{ULP(f(x_0))}\right|. \tag{4}$$

The number of floating-point numbers between the oracle and the computed value can be used to obtain the error in Bits ($error_{bits}$), which is defined as shown in Eqs. 5 and 6.

$$FP_{number}\{f(x_0),\tilde{f}(x_0)\} = \left|\left\{a_i \in \mathbb{F} \mid \min(f(x_0),\tilde{f}(x_0)) \le a_i \le max(f(x_0),\tilde{f}(x_0))\right\}\right|. \tag{5}$$

$$error_{bits}\{f(x_0),\tilde{f}(x_0)\} = log_2\left(FP_{number}\{f(x_0),\tilde{f}(x_0)\}\right). \tag{6}$$

FP_{number} represents the number of floating-point numbers between the oracle $f(x_0)$ and the computed value $\tilde{f}(x_0)$, while $error_{bits}$ represents the difference in Bits between the oracle and the computed value. For example, there are 2251799813685249 floating-point numbers between the double-precision floating-point numbers 1.0 and 2.0, and taking the log_2 of 2251799813685249 gives 51, indicating that there is a difference of 51 bits between 1.0 and 2.0, with an $error_{bits}$ of 51. Since double-precision floating-point numbers have only 64 bits, the $error_{bits}$ between any two floating-point number will be greater than or equal to 0 and less than 64. Compared to absolute error, Bits error can maintain measurement consistency and avoid by zero error caused by relative error, and also support *NaN* and $\pm\infty$

This paper uses the standard ULP error, with the oracle computed using the MPFR [31] high-precision library at a precision of 128 bits.

3.3 Significant error

In this paper, the significant error is defined as shown in Eq. 7, where ϵ is the error threshold, indicating the maximum allowed error, and $error_{ulp}(f(x_0))$ is the ULP error of the function $f(x)$ under the input of x_0. If $error_{ulp}(f(x_0))$ exceeds the set threshold ϵ, it is considered that there is a significant error, otherwise, there is no significant error. The concept of significant error is used to determine whether an error hotspot or error hotspot interval is located in the low precision layer of the hierarchical search. If no significant error is detected, it means that there is no input or small interval that triggers a larger error is found in the low precision layer and the input precision must be increased to improve the detection effect.

$$significant\ error = \begin{cases} 1, \text{if } error_{ulp}(f(x_0)) \geq \epsilon. \\ 0, \text{if } error_{ulp}(f(x_0)) < \epsilon. \end{cases} \tag{7}$$

4 HSED framework

The most accurate detection method for double precision data type is to exhaust all 2^{64} floating-point numbers, which is not feasible in reality due to the large search space, while half precision is only 16 bits and it is feasible to exhaust all half precision floating-point numbers for a total of 2^{16} numbers. The error distribution of the floating-point program is unchanged as the input precision increases, as shown in Fig. 2a–c, while the floating-point error becomes more and more accurate as the input precision increases, so we can use low-precision inputs to locate the error hotspots, and then use high-precision inputs to increase the sampling of the error hotspot interval to obtain more accurate error results, through this kind of hierarchical search can be a good balance between detection effect and performance.

In order to achieve error detection using a hierarchical search below the original input precision for floating-point arithmetic expressions, this paper designs the error detection tool HSED for floating-point arithmetic expressions as shown in Fig. 3. It uses a floating-point arithmetic expression $f(x)$ as input and a user-specified detection interval as a parameter, and detects the maximum error within the user-specified interval and reports the corresponding input x through two modules: automatic generation

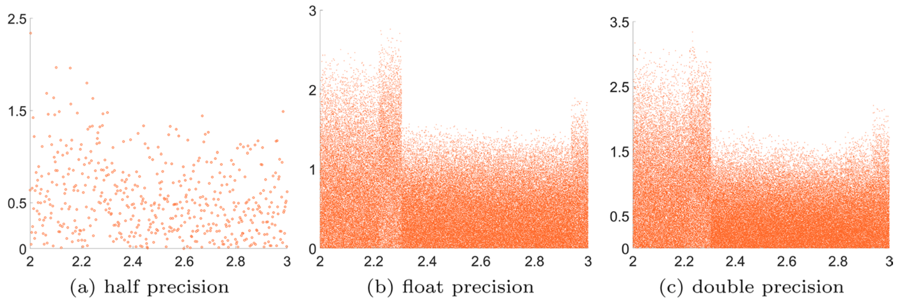


Fig. 2 The error distribution graphs of the predatorPrey benchmark in three data types are shown, where the detection interval is [2,3], the *x* axis represents the input range, and the *y* axis represents the ULP error. **a** Is the exhaustive under half precision, **b** is uniform sampling with 100,000 samples under float type, and **c** is uniform sampling with 100,000 samples under double type

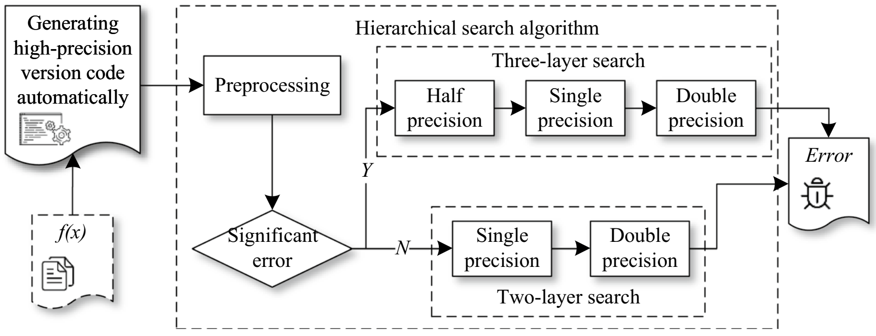


Fig. 3 HSED framework

of a high-precision version of the code and detection by a hierarchical search algorithm.The detection of the hierarchical search algorithm is the core work of this paper.

- The module for automatic generation of high-precision version code is the preliminary work for the computation of errors by performing syntactic, lexical and semantic analysis of floating-point arithmetic expressions, parsing them into an Abstract Syntax Tree (AST), using MPFR high-precision version equivalent substitution for the nodes of the tree including quadratic operations, function operations and variables, and finally recursively generating the high-precision version code of the expressions.
- The hierarchical search algorithm module is the core of the error detection work, first through the preprocessing stage to determine whether there is a significant error, if there is a significant error to carry out a two-layer search, otherwise carry out a three-layer search, and finally output the maximum error of the arithmetic expression in the detection interval and the corresponding input.

It should be noted that the half precision and single precision data types used in this paper are achieved by modifying the mantissa bits of the double precision floating-point type, manipulating the first 10 and 23 mantissa bits, respectively, and are not really the half precision and single precision data types used, as the detection interval is limited by its exponent bit size when used directly, and the term is quoted in this paper for distinction.

5 HSED implementation

This section details the implementation of HSED, including the two modules of automatic code generation for high-precision versions of floating-point arithmetic expressions and the hierarchical search algorithm.

5.1 Generating high-precision version code automatically

HSED supports arbitrary floating-point arithmetic expressions, including quadratic operations and basic function operations (e.g. exponential, logarithmic, trigonometric, square, hyperbolic functions). HSED decomposes expressions into the four basic elements of variables, numbers, functions and binary operators for processing. HSED parses the floating-point arithmetic expression into an AST. The root and middle nodes of the AST are binary operators (e.g. addition, subtraction, multiplication, division), and the leaf nodes can be numbers, variables, or functions. When a leaf node is a function, it extends a number of links to other trees corresponding to the individual parameters of the function being represented. The root and middle nodes of the AST tree are replaced using equivalent MPFR high-precision operations, and the leaf nodes are replaced using the variable type of the MPFR library, with the precision set to 128 bits. The final recursion can generate equivalent MPFR high-precision versions of the code, which are used to obtain the oracle of the expression and to compute the error using the oracle. For example, for the expression $x^2 + \sin x + 1.0$, its AST and the AST after replacement by MPFR high-precision operations are shown in Fig. 4.

5.2 Preprocessing

The preprocessing stage only operates on the first 10 mantissa bits to simulate "half precision", with a small time overhead to exhaust all floating-point detection at this

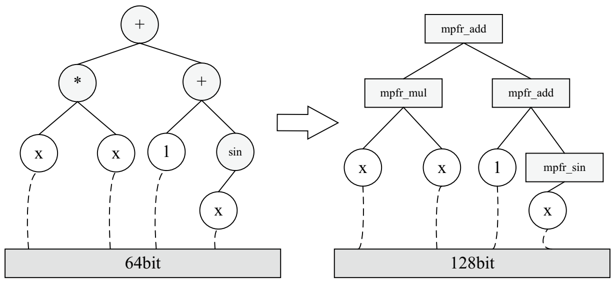


Fig. 4 Replace expression node with MPFR

layer. For example, there are 1,054,273 floating point numbers in the interval [0,100] at the “half precision" layer, and the time overhead for HSED to exhaust all floating point numbers if this size to detect errors is about 1.8s, while there are only 13,603 floating point numbers in the interval [0.01,100], and HSED takes 0.8s. The floating-point number in the interval [0,100] is two orders of magnitude larger than the floating-point number in the interval [0.01,100], which is in accordance with the law of floating-point distribution, the closer the floating-point number is to 0, the denser the distribution, the detection interval is far from 0 even though the interval is large, but the number of floating-point numbers is also very small, for example, in the [0.01,1,000,000] there are only 23,805 floating-point numbers. So HSED performs exhaustive detection of the “half precision" layer in the preprocessing stage.

During the preprocessing stage, HSED exhaustively detects all floating-point numbers within the test interval to obtain a maximum $error_{ulp}$ and an input x that triggers the error, if $error_{ulp} < \epsilon$, it means that there is no significant error in the expression within that interval, otherwise, there is a significant error. The concept of significant error is used to solve the problem that it is difficult to locate the error hotspots or error hotspot regions even if the “half precision” layer is exhaustive. For example, in the NMSEproblem341 benchmark in the FPBench, the expression $f(x) = \frac{1-\cos x}{x^2}$ has a larger error near $x = 0$, this is because when x is close to 0, the difference between $\cos x$ and 1 is very small, and the subtraction operation will introduce a large amount of rounding error, x^2 is very close to 0 and is more likely to introduce truncation error, causing a very large error in the calculation result of the whole expression, and may lead to numerical instability (such as overflow or underflow). If the user detects the maximum error of this expression in the interval [0.001,2], the error near 0 will definitely be larger, and the error hotspot must be near the position near the left endpoint $x = 0.001$ of the interval, where a significant error will appear. From Fig. 5a and b, it can be seen that there is a significant error in the expression within the interval [0.001,2], and although only 11,241 floating-point numbers were exhaustively searched in the “half precision” layer, the region of the error hotspot(indicated by the green box) can still be effectively located due to the existence of significant errors.

However, for other types of expression, there may be no significant error within the detection interval, and the overall error may be small. For example, in the predatorPrey benchmark in FPBench, the arithmetic expression is $f(x) = \frac{4x^2}{1+(\frac{x}{1.11})^2}$, assuming that the detection interval is still [0.001,2], Fig. 6a shows that there is no significant error in the expression within the interval [0.001,2], and the overall error is small. However, due to the exhaustive search of only 11,241 floating-point numbers in the “half precision" layer, the input of floating-point numbers is sparse and cannot locate the error hotspot, the error hotspot should be near the pink circle in Fig. 6a, but the “half precision” layer locates it near the green circle in Fig. 6b. For expressions without significant errors, increasing the number of sample points can effectively locate the error hotspot, and setting the starting search layer to the “single precision" layer can solve this problem.

In summary, in the preprocessing stage, HSED simulates “half precision" by operating on the first 10 bits of the mantissa of double precision floating-point types, and it exhaustively searches all floating-point numbers in the “half precision" layer

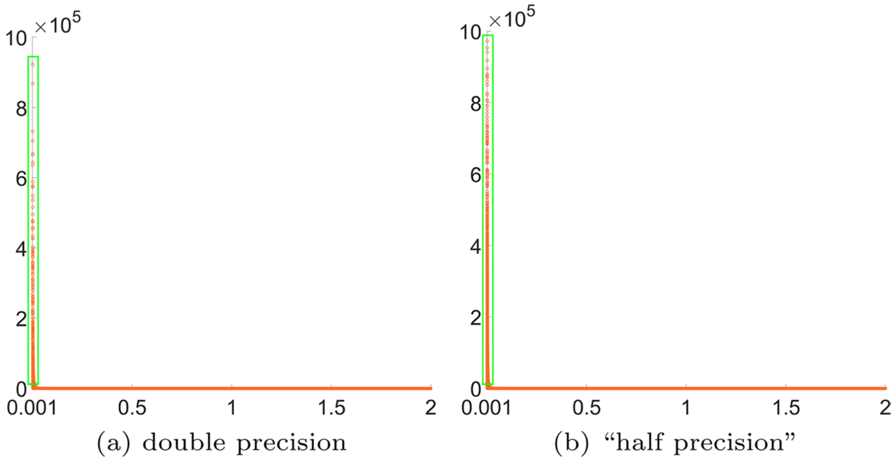


Fig. 5 The error distribution diagram of $f(x) = \frac{1-\cos x}{x^2}$, x axis represents the input range and y axis represents the ULP error. Figure 5a shows the error distribution with 300,000 uniformly sampled inputs in double precision, Fig. 5b shows the error distribution with all floating-point numbers exhaustively searched in the “half precision" layer

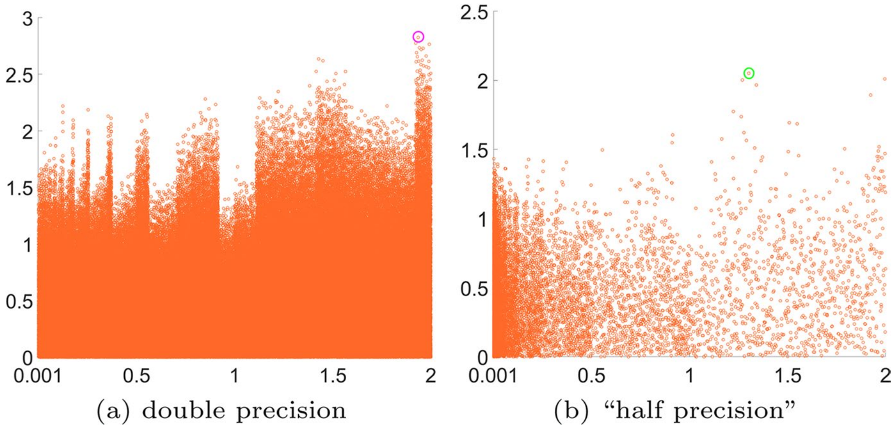


Fig. 6 The error distribution diagram of $f(x) = \frac{4x^2}{1+(\frac{x}{1.11})^2}$, x axis represents the input range and y axis represents the ULP error. **a** Shows the error distribution with 300,000 uniformly sampled inputs in double precision, **b** shows the error distribution with all floating-point numbers exhaustively searched in the “half precision" layer

to detect if there is a significant error. HSED sets the significant error to 100ULP, if there is a significant error, it performs a three-layer search from “half precision" to double precision, otherwise it performs a two-layer search from “single precision" to double precision. It should be noted that since the preprocessing stage already exhaustively detects the “half precision" layer, the results can be reused in the “half precision" layer of the three-layer search, saving the search time for the “half precision" layer. HSED sets the interface for users to change the significant error size, too large a significant error setting is more likely to enter the two-layer search, sampling more points than the three-layer search, the detection effect will not be worse but the performance will be affected. Significant error sets too small are more likely to enter the three-layer search, which is limited by the fact that the “half precision" layer in the three-layer search may not locate the actual error hotspots and detection may be poor, but performance is faster than in the two-layer search.

5.3 Hierarchical search algorithm

The hierarchical search algorithm of HSED includes a three-layer search algorithm from “half precision" to double precision and a two-layer search algorithm from “single precision" to double precision.

Algorithm 1 Three-layer search algorithm

```
Require: $x_l, x_r$(endpoints of the input interval)
Ensure: $error_{max}, x$
1: $\{error_{max}, x\} \Leftarrow 0$
2: $\{x_{hl}, x_{hr}\} \Leftarrow$ convert_to_half $\{x_l, x_r\}$
3: for $x_i$ in $[x_{hl}, x_{hr}]$ do
4:     $error_i \Leftarrow$ compute_error$(x_i)$
5:     if $error_i \geq error_{max}$ then
6:         $error_{max} \Leftarrow error_i$
7:         $x_h \Leftarrow x_i$
8:     end if
9:     $x_{i+1} \Leftarrow x_i +$ 0x40000000000
10: end for
11: $\{x_h, x_{fr}\} \Leftarrow$ convert_to_float$(x_h)$
12: for $x_j$ in $[x_h, x_{fr}]$ do
13:     $error_j \Leftarrow$ compute_error$(x_j)$
14:     if $error_j \geq error_{max}$ then
15:         $error_{max} \Leftarrow error_j$
16:         $x_d \Leftarrow x_j$
17:     end if
18:     $x_{j+1} \Leftarrow x_j +$ 0x20000000
19: end for
20: $\{x_d, x_{dr}\} \Leftarrow$ convert_to_double$(x_d)$
21: for $x_k$ in random_search$(x_d, x_{dr})$ do
22:     $error_k \Leftarrow$ compute_error$(x_k)$
23:     if $error_k \geq error_{max}$ then
24:         $error_{max} \Leftarrow error_k$
25:         $x \Leftarrow x_k$
26:     end if
27: end for
28: return $\{error_{max}, x_h \text{ or } x_d \text{ or } x\}$
```

If there is a significant error in the preprocessing stage, the three-layer search algorithm described in Algorithm 1 is executed. First, the function convert_to_half should be executed to set the last 42 bits of the two floating-point numbers at the interval endpoints to 0, converting them to “half precision". The for loop in lines 3-10 exhaustively detects all float numbers in the “half precision" layer, note that in line 9, we only exhaust the first 10 bits of the mantissa bits, so the offset value is 0 x 40000000000 each time. By exhaustively detecting all floating-point numbers in the “half precision" layer, we can determine the maximum error and the corresponding input that triggers the error. Then, the

Algorithm 2 Two-layer search algorithm

```
Require: $x_l, x_r$(endpoints of the input interval)
Ensure: $error_{max}, x$
 1: $\{error_{max}, x\} \Leftarrow 0$
 2: $\{x_{fl}, x_{fr}\} \Leftarrow$ convert_to_float $\{x_l, x_r\}$
 3: for $x_i$ in random_search$(x_{fl}, x_{fr})$ do
 4:     $error_i \Leftarrow$ compute_error$(x_i)$
 5:     if $error_i \geq error_{max}$ then
 6:         $error_{max} \Leftarrow error_i$
 7:         $x_d \Leftarrow x_i$
 8:     end if
 9: end for
10: $\{x_d, x_{dr}\} \Leftarrow$ convert_to_double$(x_d)$
11: for $x_k$ in random_search$(x_d, x_{dr})$ do
12:     $error_k \Leftarrow$ compute_error$(x_k)$
13:     if $error_k \geq error_{max}$ then
14:         $error_{max} \Leftarrow error_k$
15:         $x \Leftarrow x_k$
16:     end if
17: end for
18: return $\{error_{max}, x_d \text{ or } x\}$
```

convert_to_float function in line 11 is executed to operate on the input and set the extra 13 mantissa bits to get a tiny interval in the “single precision" layer. The for loop in lines 12-19 exhaustively detects all float numbers in the tiny interval in the “single precision" layer. Note that in line 18, we operate on the first 23 bits of the mantissa, so the offset value is 0x20000000 each time. By exhaustively detecting all floating-point numbers in the tiny interval in the “single precision" layer, we can update the maximum error and the corresponding input. Similarly, the last 29 mantissa bits of the input obtained in the previous layer are operated on in line 20 to obtain a tiny interval in the double precision layer. The for loop in lines 21-27 detects in the double precision layer. Unlike the previous two layers, exhaustive search is not used in the double precision layer because it operates on

29 bits of the mantissa, and the time cost of exhaustive search is too high, so we choose to generate 100,000 random numbers to detecting.

If there is no significant error in the preprocessing stage, the two-layer search algorithm described in Algorithm 2 is executed, where lines 3-9 and 11-17 correspond to the “single precision" and double precision layers, respectively. Unlike the three-layer search, the two-layer search starts from the “single precision" layer. As the “single precision" layer requires 23 mantissa bits to be processed, and exhaustive search is too-consuming, so random search is used for this layer instead of an exhaustive search.

In summary, whether it is the three-layer search or the two-layer search, the lowest layer is used to quickly locate the error hotspot with lower precision, as shown in Fig. 7aand 8a, while the higher layers are used to obtain a very small interval of the error hotspot and sampling within that interval to obtain the most accurate error results possible, as shown in Figs. 7b, 8b. The sampling within the green boxes in Figs. 7d and 8c is dense, which is exactly the effect achieved by hierarchical search. By quickly locating the error hotspot with lower precision, and intensively sampling and detecting the error hotspot region in the higher precision layer, the maximum error can be effectively detected quickly and efficiently.

6 Experimental results

HSED is completely implemented in the C++ language, using over 1100 lines of code for automatically generate high-precision versions of floating-point arithmetic expressions, and over 700 lines of code to implement a hierarchical search algorithm for error detection. The experimental environment is based on the Ubuntu 20.04.4 LTS operating system with the Linux 5.14.0-1051-oem kernel, running on an Intel Xeon E5-6230 v4 CPU. HSED generates a cpp program that calls MPFR for error measurement, which is compiled using GCC 9.4.0 with the options "-lm -lmpfr". To verify the effectiveness of HSED in detecting errors in floating-point arithmetic expressions, this paper evaluates its effects and performance on 32 single-parameter arithmetic expressions from the FPBench benchmark suite, and compares the results with those achieved by the state-of-the-art error detection tools Herbie, Herbie+, S3FP and ATOMU.

6.1 Test case

The benchmark test information in this paper is shown in Table 1. The FPBench benchmark test set contains a total of 46 single-parameter arithmetic expressions, 32 of which are selected as test objects for this study. Fourteen benchmarks are excluded because they contain loops, conditionals or repeated expressions. The 32 benchmark expressions are divided into two categories: general expressions (12) and function operation expressions (20). The detection interval is set according to the default range provided by FPBench. If FPBench does not provide a default interval, the interval [0.01,100] is used, which is the usual interval.

6.2 Precision results and analysis

The HSED is compared with the state-of-the-art error detection tools, including Herbie, Herbie+, S3FP and ATOMU.

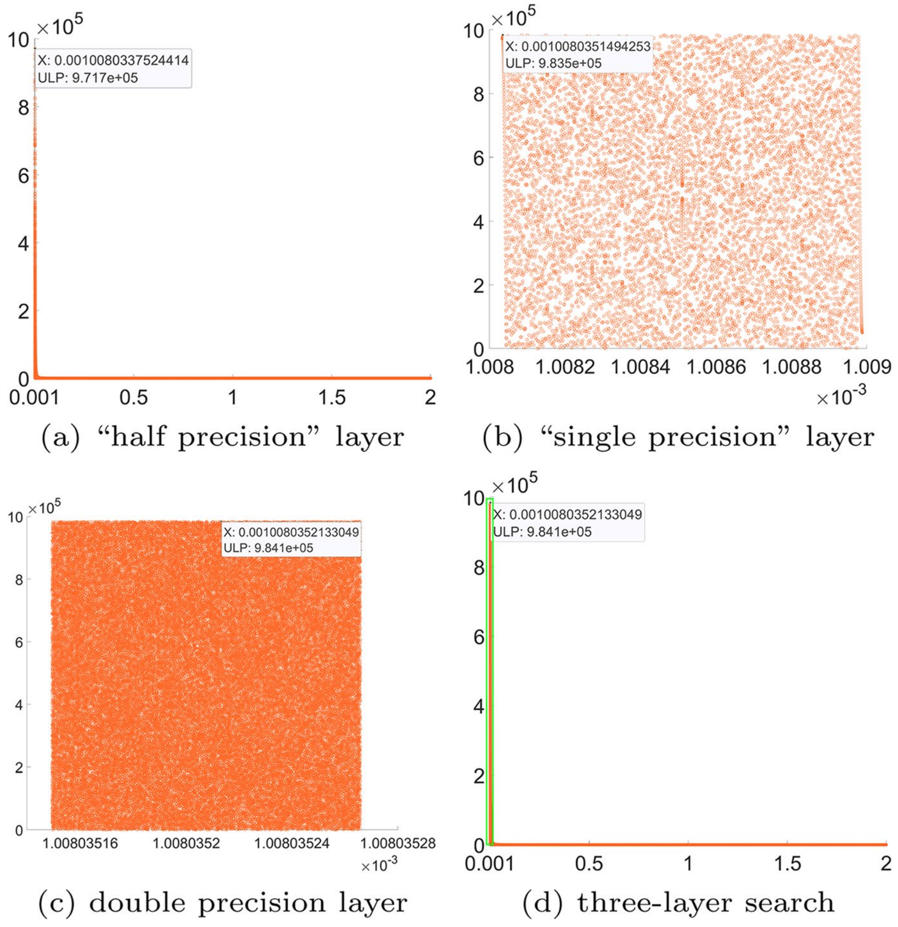


Fig. 7 The distribution of three-layer search errors on the interval [0.001,2] for the function $f(x) = \frac{1-\cos x}{x^2}$, **a**–**c** represent error distribution for different layers, **d** represents error distribution for the overall three-layer search

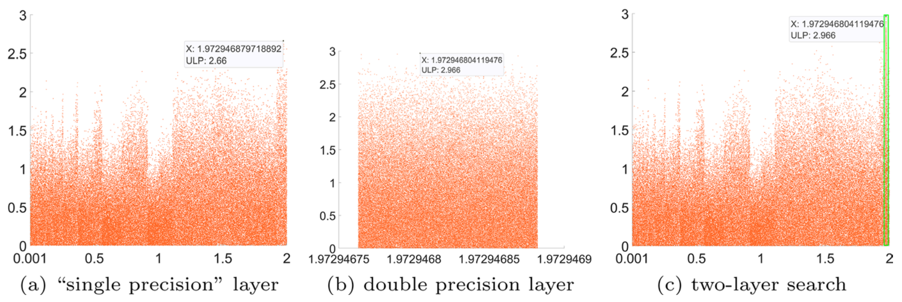


Fig. 8 The distribution of two-layer search errors on the interval [0.001,2] for the function $f(x) = \frac{4x^2}{1+(\frac{x}{1.11})^2}$, **a**, **b** represent error distribution for different layers, **c** represents error distribution for the overall two-layer search

6.2.1 Comparison with Herbie and Herbie+

Herbie is an error detection and precision optimization tool that finds the maximum error by randomly sampling 8000 input points in the error detection module. To ensure fairness in the comparison, the number of input points for Herbie's random sampling is increased to 100,000, referred to as Herbie+. Figure 9 shows the maximum error test results of HSED compared to Herbie and Herbie+. Since the output of Herbie and Herbie+ is in Bits error, the comparison with these two tools is in Bits error.

HSED performs significantly better than Herbie. Among the 32 benchmark tests, HSED detects a maximum error higher than that of Herbie in 24 cases, while only in a few cases the results are lower of equal to Herbie (less than 3 and equal to 5). HSED performs worse than Herbie in three benchmarks, sqroot, NMSEsection311 and predatorPrey, For these benchmarks, increasing the number of random search sampling input points can improve the detection effect. For sqroot, when improving the sampling points of the random search to 150,000, the same effect as Herbie can be achieved. Therefore, for benchmarks where HSED does not outperform Herbie, optimization can be achieved by increasing the number of sampling points.

In the precision comparison test with Herbie+, HSED also performs significantly better than Herbie+. The number of cases where it performs better, equal to, and worse than Herbie+ is 19, 8, and 5, respectively. This confirms that increasing the number of sampling points can indeed improve detection effect, but the downside is reduced performance.

6.2.2 Comparison with S3FP

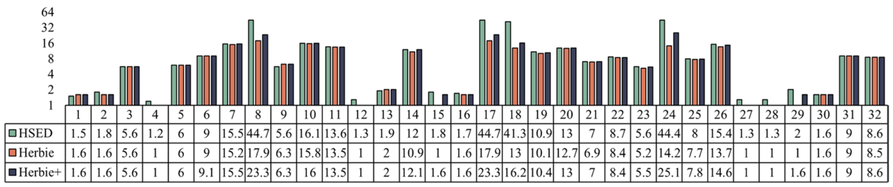
S3FP is an error detection tool that uses binary-guided random testing (BGRT) and we define its TIMEOUT parameters as 1000, 10,000 and 50,000, respectively. Table 2 shows the maximum error test results of HSED compared to S3FP . The output of S3FP is in maximum relative error, so the comparison with the S3FP tool is in terms of relative error.

Table 1 Test set information

No	FPBench	D	No	FPBench	D
1	sqroot	[0,1]	17	NMSEproblem341	[0.01,100]
2	sqrt_add	[1,1000]	18	NMSEexample38	[0.01,100]
3	exp1x	[0.01,0.5]	19	NMSEproblem334	[0.01,100]
4	exp1x_log	[0.01,0.5]	20	NMSEproblem333	[0.01,100]
5	NMSEexample37	[0.01,100]	21	NMSEproblem331	[0.01,100]
6	NMSEproblem336	[0.01,100]	22	NMSEexample36	[0.01,100]
7	NMSEexample39	[0.01,100]	23	NMSEexample35	[0.01,100]
8	NMSEproblem341	[0.01,100]	24	NMSEexample34	[0.01,100]
9	NMSEsection311	[0.01,100]	25	NMSEexample31	[0,100]
10	NMSEproblem345	[0.01,100]	26	test05_nonlin1_r4	[1.00001,2]
11	NMSEproblem337	[0.01,100]	27	test05_nonlin1_test2	[1.00001,2]
12	verhulst	[0.1,0.3]	28	intro-example-mixed	[1,999]
13	predatorPrey	[0.1,0.3]	29	sineOrder3	[-2,2]
14	logexp	[0.01,8]	30	bsplines3	[0,1]
15	sine	$[-\frac{\pi}{2},\frac{\pi}{2}]$	31	NMSEexample310	[0.001,1]
16	carbonGas	[0.1,0.5]	32	NMSEproblem343	[0.001,1]

Additionally, S3FP could not complete 14 benchmarks due to time constraints and we mark these results as NA. The results show that both detection effects and efficiency are significantly better than the S3FP tool. With the S3FP TIMEOUT set to 1000, HSED has 29 benchmarks (including NA) that outperform S3FP, and still has 27 benchmarks that outperform the S3FP as the TIMEOUT increases. For benchmarks where S3FP cannot complete within the given time budget, we also try increasing the TIMEOUT parameter, but in most cases there are still no effective results. The experimental data in Table 2 that as the search time of S3FP gets longer the results get better, but that

increasing the search time on top of a larger search time has a weak improvement on the results, for example changing TIMEOUT from 10,000 to 50,000 results in a smaller improvement.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
🟩 HSED	1.5	1.8	5.6	1.2	6	9	15.5	44.7	5.6	16.1	13.6	1.3	1.9	12	1.8	1.7
🟥 Herbie	1.6	1.6	5.6	1	6	9	15.2	17.9	6.3	15.8	13.5	1	2	10.9	1	1.6
■ Herbie+	1.6	1.6	5.6	1	6	9.1	15.5	23.3	6.3	16	13.5	1	2	12.1	1.6	1.6

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
🟩 HSED	44.7	41.3	10.9	13	7	8.7	5.6	44.4	8	15.4	1.3	1.3	2	1.6	9	8.6
🟥 Herbie	17.9	13	10.1	12.7	6.9	8.4	5.2	14.2	7.7	13.7	1	1	1	1.6	9	8.5
■ Herbie+	23.3	16.2	10.4	13	7	8.4	5.5	25.1	7.8	14.6	1	1	1.6	1.6	9	8.6

Fig. 9 Comparison of the maximum error (higher is better). x axis is the benchmark numbers; y axis is log-2 scaled

6.2.3 Comparison with ATOMU

We compare HSED with ATOMU and the experimental data are shown in Table 3. We set the test interval to [− 100, 100] by default, which is a common interval and ATOMU for most of the benchmark reported test cases all fall into this interval, only the bench 6, 18, 19, 21–23 report a large test case, for these 6 we set the test interval to $[0, +\infty]$ and $[-\infty, +\infty]$, the purpose is also to cover the test cases reported by ATOMU. HSED has 25 benchmarks that detect higher errors than ATOMU, which are bolded in Table 3. Average maximum relative error HSED is 5.11E+14 higher than ATOMU, with an average improvement of 5.50E+14. But the performance of ATOMU is far better than HSED, with performance 158x better than HSED. As ATOMU relies on condition numbers to localize input values of significant errors and does not need to compute oracles, its overhead is very small.

6.3 Performance results and analysis

For performance testing, to avoid the influence of software and hardware, we execute the program ten times, removing the longest and shortest execution times, and finally calculate the average execution time of eight runs as the time cost of a single test. When comparing performance with the S3FP, we compare it with its TIMEOUT of 10,000, as the S3FP performs better in terms of accuracy at this magnitude. Figure 10 shows the time cost in seconds for HSED and three other tools. Equation 8 is used in this paper to calculate the average time cost of all benchmarks, where n is the number of benchmarks, and Eq. 9 is used to calculate the time radio between different tools. As there are 14 benchmarks with no results in S3FP, we set n to 18 when calculating the average time cost of S3FP. Using Eq. 8 and 9, the time ratio of HSED to Herbie, Herbie+ and S3FP is calculated to be 1.82, 11.20, and 129.15, respectively. The larger the time ratio, the better the performance of HSED.

$$average\_time = \frac{\sum_{i=1}^{n} time_i}{n}. \tag{8}$$

$$radio\_time = \frac{average\_time_{other\_tools}}{average\_time_{HSED}}. \tag{9}$$

The benchmark 15 and 29 time overheads are greater, because in section 5.2, it is mentioned that there are a large number of floating-point numbers near 0, if the detection range includes 0, the preprocessing phase will take longer . If there is no significant error, a two-layer search is performed, and each layer requires random search to generate sample points, the number of sample points is more than that of the three-layer search, which reduces the performance. Overall, HSED has a fast performance. For example, 17 benchmarks have a time cost of less than 1000

Table 2 The comparative accuracy between HSED and S3FP. The size of the S3FP TIMEOUT parameter is given in parentheses

No	HSED	S3FP(1000)	S3FP(10000)	S3FP(50000)
1	3.09E−16	2.74E-16	3.00E-16	3.13E-16
2	2.72E−16	NA	NA	NA
3	1.09E−14	1.09E−16	1.10E−16	1.10E−16
4	2.52E−16	NA	NA	NA
5	8.62E−15	NA	NA	NA
6	6.73E−14	NA	NA	NA
7	5.59E−12	2.93E−13	1.84E−12	2.07E−12
8	3.58E−03	1.09E−16	1.10E−16	1.10E−16
9	8.59E−15	NA	NA	NA
10	7.79E−12	2.87E−16	2.97E−16	3.05E−16
11	1.49E−12	NA	NA	NA
12	1.66E−16	1.91E−16	2.09E−16	2.10E−16
13	3.15E−16	2.07E−16	2.10E−16	2.15E−16
14	4.99E−13	NA	NA	NA
15	2.74E−16	2.23E−16	2.99E−16	3.03E−16
16	2.89E−16	2.98E−16	3.05E−16	3.41E−16
17	3.58E−03	1.09E−16	1.10E−16	1.10E−16
18	5.58E−04	1.27E−14	1.30E−14	6.59E−14
19	2.13E−13	NA	NA	NA
20	1.54E−12	2.60E−14	2.88E−14	2.88E−14
21	1.68E−14	6.22E−15	6.28E−15	6.49E−15
22	4.76E−14	NA	NA	NA
23	9.57E−15	NA	NA	NA
24	3.58E−03	NA	NA	NA
25	3.27E−14	NA	NA	NA
26	4.80E−12	8.20E−17	8.30E−17	8.30E−17
27	1.66E−16	8.20E−17	8.30E−17	8.30E−17
28	1.67E−16	5.73E−17	1.07E−16	1.09E−17
29	3.73E−16	2.19E−16	2.80E−16	2.80E−16
30	2.19E−16	1.64E−16	1.64E−16	1.66E−16
31	1.11E−13	NA	NA	NA
32	4.57E−14	NA	NA	NA

milliseconds. In the performance comparison test with Herbie+, after increasing the sampling points of Herbie+ to 100,000, HSED outperforms Herbie+ for all benchmarks. S3FP itself has to define a longer time cost to guide the search, so the performance of HSED is much better than that of S3FP.

Since the comparison interval with ATOMU is different from Herbie and S3FP, and ATOMU itself does not compute oracles, the performance comparison is not shown here, and the specific data can be seen in section 6.2.3.

Table 3 HSED vs ATOMU: Error and time overhead data

No.	D	RelErr			Time(s)	
		HSED	ATOMU	Improvement	HSED	ATOMU
1	[− 100,100]	3.07E−13	**2.11E−01**	− 2.11E−01	4.124	0.054
2	[− 100,100]	**2.71E−16**	NA	2.71E−16	3.232	0.03
3	[− 100,100]	**1.00E+00**	2.52E−02	9.75E−01	6.507	0.024
4	[− 100,100]	**2.75E−16**	5.41E−17	2.21E−16	19.056	0.055
5	[− 100,100]	**1.00E+00**	2.52E−02	9.75E−01	5.662	0.032
6	$[0,+\infty)$	6.10E+01	**9.30E+01**	− 3.20E+01	22.222	0.048
7	[− 100,100]	**2.30E+00**	3.10E−01	1.99E+00	1.859	0.047
8	[− 100,100]	**1.00E+00**	1.33E−01	8.67E−01	1.559	0.049
9	[− 100,100]	**5.00E−01**	9.93E−02	4.00E−01	5.491	0.032
10	[− 100,100]	**1.00E+00**	**1.00E+00**	0.00E+00	4.087	0.067
11	[− 100,100]	**3.60E+16**	1.84E+16	1.76E+16	5.261	0.031
12	[− 100,100]	3.48E−13	**9.91E−02**	− 9.91E−02	2.258	0.02
13	[− 100,100]	**3.48E−13**	1.23E−16	3.47E−13	2.36	0.04
14	[− 100,100]	**1.88E−02**	2.15E−16	1.88E−02	5.69	0.038
15	[− 100,100]	2.09E−12	**1.39E+00**	− 1.39E+00	5.09	0.037
16	[− 100,100]	1.20E−13	**2.07E−02**	− 2.07E−02	3.823	0.029
17	[− 100,100]	**1.00E+00**	1.33E−01	8.67E−01	1.504	0.053
18	$[0,+\infty)$	**2.47E+00**	9.93E−01	1.47E+00	24.915	0.046
19	$(-\infty,+\infty)$	**1.17E+01**	2.50E−01	1.14E+01	6.911	0.038
20	[− 100,100]	1.55E−12	**2.62E+05**	− 2.62E+05	2.803	0.05
21	$(-\infty,+\infty)$	**1.00E+00**	5.89E−02	9.41E−01	3.707	0.037
22	$[0,+\infty)$	**4.43E+00**	1.29E−02	4.41E+00	2.61	0.031
23	$(-\infty,+\infty)$	**1.00E+00**	7.95E−17	1.00E+00	27.192	0.07
24	[− 100,100]	**1.00E+00**	1.33E−01	8.67E−01	1.812	0.068
25	[− 100,100]	**3.27E−14**	5.00E−16	3.22E−14	2.934	0.034
26	[− 100,100]	**1.60E−12**	2.22E−16	1.60E−12	2.328	0.037
27	[− 100,100]	**6.65E−17**	0.00E+00	6.65E−17	1.998	0.03
28	[− 100,100]	**8.59E−17**	0.00E+00	8.59E−17	2.317	0.026
29	[− 100,100]	9.99E−11	**3.89E−02**	− 3.89E−02	2.73	0.035
30	[− 100,100]	**2.17E−16**	NA	2.17E−16	1.931	0.032
31	[− 100,100]	**1.00E+00**	**1.00E+00**	0.00E+00	15.736	0.061
32	[− 100,100]	**1.00E+00**	3.27E−01	6.73E−01	8.958	0.042
Average		1.12E+15	6.12E+14	5.50E+14	6.5208	0.04134

7 Conclusion and future work

This paper proposes and implements an algorithm that uses hierarchical search to detect the maximum error of floating-point arithmetic expressions, and implements the error detection tool HSED. The core idea of HSED is to use the lower precision below the original input precision to guide the search, quickly locate the error

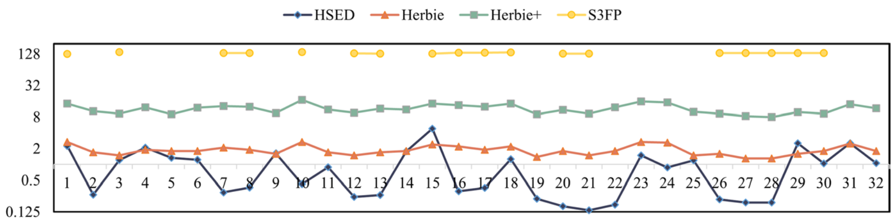


Fig. 10 The performance comparison among HSED and the other three tools (lower is better). *x* axis is benchmark numbers; *y* axis is log-2 scaled, the unit is second

hotspots, and use the high precision layer to increase the sampling of the extremely small intervals that cause the error hotspots to obtain more accurate error results. Future work includes mainly expanding HSED's support for multi-parameter floating-point arithmetic expressions and parallelizing HSED to further improve performance. Given the slow performance of the MPFR library, some other scalable precision library may be used in the future to replace MPFR [29, 30].

Author contributions ZZ and JX were responsible for the methodology, implementation and writing; YQ and HH were responsible for some of the experiments; JH was involved in the review and supervision; and BZ was involved in the editing, review and supervision.

Funding Supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing. Fund No. 2023B02.

Data availability The datasets used for the experiments are benchmark datasets. The experimental data can be reproduced in the github open source address.

Code availability The github open source address will be included here when the paper is accepted.

Declarations

Conflict of interest The authors declare no competing interests.

Ethics approval Not applicable.

References

1. Kevin Q (1983) Ever had problems rounding off figures. This stock exchange has. The Wall Street J 1983:37
2. Jacques-Louis L, Lennart L, Jean-Luc F et al (1996) Ariane 5 flight 501 failure report by the inquiry board. European space agency Paris
3. Robert S (1992) Roundoff error and the Patriot missile. SIAM News 25(4):11
4. Xia Y, Guo S, Hao J, Liu D, Xu J (2020) Error detection of arithmetic expressions. J Supercomput 77:5492–5509. https://doi.org/10.1007/s11227-020-03469-7
5. Daming Z, MuHan Z, Yingfei X, Zhoulai F, Lu Z, Zhendong S (2019) Detecting floating-point errors via atomic conditions. In: Proceeding of the 47th ACM on Programming Languages (POPL 2020). Association for Computing Machinery, New York, NY, USA, vol 4, pp 1–27. https://doi.org/10.1145/3371128

6. Wei-fan C, Ganesh G, Zvonimir R, Alexey S (2014) Efficient search for inputs causing high floating-point errors. In: Proceeding of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2014). Association for Computing Machinery, New York, NY, USA, vol 49, pp 43–52. https://doi.org/10.1145/2692916.2555265
7. Qi H, Xu J, Guo S (2018) Detection of the maximum error of mathematical functions. J Supercomput 74:6275–6290. https://doi.org/10.1007/s11227-018-2552-x
8. What GD, Arithmetic ECSSKAF (1991) ACM Comput Surv 23:5–48. https://doi.org/10.1145/103162.103163
9. Xin Y, Liqian C, Xiaoguang M, Tao J (2019). In: Proceeding of the 46th ACM on Programming Languages (POPL 2019). Association for Computing Machinery, New York, NY, USA, vol 3, pp 1–29. https://doi.org/10.1145/3290369
10. Nicholas N, Julian S (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. Sigplan Not SIGPLAN 42:89–100. https://doi.org/10.1145/1273442.1250746
11. Daming Z, Ran W, Yingfei X, Lu Z, Zhendong S, Hong M (2015) A genetic algorithm for detecting significant floating-point inaccuracies. In: Proceeding of the 37th International Conference on Software Engineering (ICSE 2015). IEEE Press, vol 1, pp 529–539
12. Earl T. B, Thanh V, Vu L, Zhendong S (2013) Automatic detection of floating-point exceptions. In: Proceeding of the 40th ACM on programming languages (POPL 2013). Association for Computing Machinery, New York, NY, USA, vol 48, pp 549–560. https://doi.org/10.1145/2429069.2429133
13. Florian B, Andreas H, Sebastian H (2012) A dynamic program analysis to find floating-point accuracy problems. In: Proceeding of the 23th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012). Association for Computing Machinery, New York, NY, USA, vol 47, pp 453–462. https://doi.org/10.1145/2254064.2254118
14. Al B, Ken B, Ben C, Andy C, Bryan F, Seth H, Charles H, Asya K, Scott M, Dawson E (2010) A few billion lines of code later: using static analysis to find bugs in the real world. In: Communications of the ACM. Association for Computing Machinery, New York, NY, USA, vol 53, pp 66–75. https://doi.org/10.1145/1646353.1646374
15. Francois F, Bruno L (2016) VERROU: a CESTAC evaluation without recompilation. In: 17th International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)
16. Alex S, Pavel P, Sorin L, Zachary T (2018) Finding root causes of floating point error. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). Association for Computing Machinery, New York, NY, USA, vol 53, pp 256–269. https://doi.org/10.1145/3192366.3192411
17. Eric G, Sylvie P (2011) Static Analysis of Finite Precision Computations. In: Proceeding of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011). Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, vol 6538, pp 232–247. https://doi.org/10.1007/978-3-642-18275-4_17
18. Hui G, Cindy R (2020) Efficient generation of error-inducing floating-point inputs via symbolic execution. In: Proceedings of the ACM/IEEE 42th International Conference on Software Engineering (ICSE 2020). Association for Computing Machinery, New York, NY, USA. pp 1261–1272. https://doi.org/10.1145/3377811.3380359
19. Pavel P, Alex S, James R, Zachary T (2015) Automatically improving accuracy for floating point expressions. In: Proceeding of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015). Association for Computing Machinery, New York, NY, USA, vol 50, pp 1–11. https://doi.org/10.1145/2737924.2737959
20. Daming Z, Yuchen G, Yuanfeng S, Mingzhe W, Yingfei X, Zhendong S (2023) Oracle-free repair synthesis for floating-point programs. In: Proceeding of the ACM on Programming Languages, vol 6, Issue OOPSLA2, Article No.: 159, pp 957–985. https://doi.org/10.1145/3563322
21. Anastasiia I, Eva D (2017) On sound relative error bounds for floating-point arithmetic. In: Proceeding of the 17th Conference on Formal Methods in computer-Aided Design (FMCAD 2017). FMCAD Inc, Austin, Texas. pp 15–22. https://doi.org/10.23919/FMCAD.2017.8102236
22. Patrick C, Radhia C (2014) Abstract interpretation: past, present and future. In: Proceeding of the Joint Meeting of the Twenty-third EACSL Annual Conference on Computer Science Logic and the Twenty-ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014). Association for Computing Machinery, New York, NY, USA. pp 1–10. https://doi.org/10.1145/2603088.2603165

23. Hickey T, Ju Q, Van EMH (2001) Interval arithmetic: from principles to implementation. J ACM 48:1038–1068. https://doi.org/10.1145/502102.502106
24. Luiz H, Jorge S (2004) Affine arithmetic: concepts and applications. Numer Algorithms 37:147–158. https://doi.org/10.1023/B:NUMA.0000049462.70970.b6
25. Wonyeol L, Rahul S, Alex A (2016) Verifying bit-manipulations of floating-point. In: Proceeding of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016). Association for Computing Machinery, New York, NY, USA, vol 51, pp 70–84. https://doi.org/10.1145/2908080.2908107
26. Roberto B, Emilio C, Daniele C, Camil D, Irene F (2018) A survey of symbolic execution techniques. ACM Comput Surv 51:1–39. https://doi.org/10.1145/3182657
27. Stella S, Alberto B, Omar I, Mirco T (2022) Tight error analysis in fixed-point arithmetic. Form Asp Comput 34:1–32. https://doi.org/10.1145/3524051
28. Alexey S, Marek S, Ian B, Charles J, Zvonimir R, Ganesh G (2018) Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: ACM Transactions on Programming Languages and Systems. Association for Computing Machinery, New York, NY, USA, vol 41, pp 1–39. https://doi.org/10.1145/3230733
29. Hao J, Xu J, Guo S (2022) Design of variable precision transcendental function automatic generator. J Supercomput 78:2196–2218. https://doi.org/10.1007/s11227-021-03937-8
30. Zhou B, YongZhong H, Jinchen X, ShaoZhong G, Hongyuan Q (2019) Memory latency optimizations for the elementary functions on the Sunway architecture. J Supercomput 75:3917–3944. https://doi.org/10.1007/s11227-018-02741-1
31. Laurent F, Guillaume H, Vincent L, Patrick P, Paul Z (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans Math Softw 33:13. https://doi.org/10.1145/1236463.1236468
32. Yuhao Z, Luyao R, Liqian C, Yingfei X, Shing-chi C, Tan X (2020) Detecting numerical bugs in neural network architectures. In: Proceeding of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA. pp 826–837. https://doi.org/10.1145/3368089.3409720

Authors and Affiliations

Zuoyan Zhang[1] · Jinchen Xu[1] · Jiangwei Hao[1] · Yang Qu[1] · Haotian He[1] · Bei Zhou[1]

✉ Bei Zhou
13653970052@163.com

Zuoyan Zhang
zhangzuoyan523@163.com

Jinchen Xu
atao728208@126.com

Jiangwei Hao
haojiangweitimo@foxmail.com

Yang Qu
qy19971130@163.com

Haotian He
m18503880251@163.com

1 Information Engineering University, No. 62 Science Anenue, High-Tech Zone, Zhengzhou 450001, Henan, China