✓ Artifact evaluated by FSE

Detecting and Fixing Precision-Specific Operations for Measuring Floating-Point Errors*

Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong†, Lu Zhang, Gang Huang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
{wangrancs,zoudm,hexinrui,xiongyf,zhanglucs,hg}@pku.edu.cn

ABSTRACT

The accuracy of the floating-point calculation is critical to many applications and different methods have been proposed around floating-point accuracies, such as detecting the errors in the program, verifying the accuracy of the program, and optimizing the program to produce more accurate results. These approaches need a specification of the program to understand the ideal calculation performed by the program, which is usually approached by interpreting the program in a precision-unspecific way.

However, many operations programmed in existing code are inherently precision-specific, which cannot be easily interpreted in a precision-unspecific way. In fact, the semantics used in existing approaches usually fail to interpret precision-specific operations correctly.

In this paper, we present a systematic study on precision-specific operations. First, we propose a detection approach to detect precision-specific operations. Second, we propose a fixing approach to enable the tuning of precisions under the presence of precision-specific operations. Third, we studied the precision-specific operations in the GNU C standard math library based on our detection and fixing approaches. Our results show that (1) a significant number of code fragments in the standard C math library are precision-specific operations, and some large inaccuracies reported in existing studies are false positives or potential false positives due to precision-specific operations; (2) our detection approach has high precision and recall; (3) our fixing approach can lead to overall more accurate result.

CCS Concepts

•**Theory of computation** → **Program analysis;**

Keywords

Floating-point accuracy; precision-specific operations.

1. INTRODUCTION

A well-known problem in software development is the inaccuracies from floating-point numbers. Since floating-point numbers can represent only limited number of digits, many real numbers cannot be accurately represented, leading to inaccuracies in computation. In a critical software system, inaccuracies from floating-point numbers may cause serious problems. A well-cited example [4, 26] is that a Patriot missile fails to intercept an incoming missile in the first Persian Gulf War due to floating-point inaccuracies.

Given the importance of floating-point accuracy, many research efforts have been devoted to the accuracy problem of floating-point programs. These approaches detect whether a program can produce a significantly inaccurate output [4, 8, 26, 2], verify whether all errors may be produced by a program are within an upper bound [18, 12, 9], or optimize the program so that the output is more accurate [10, 17]. A key problem in implementing such an approach is to get the ideal output for an execution of a program. In detection and verification approaches, we need to know the ideal output so that we can get the error of the actual output by comparing it with the ideal output. In optimization approaches, we need to know the ideal output so that we can change the program to make the output close to the ideal output.

Typically, these approaches take a *precision-unspecific semantics* to interpret the program to get the ideal output. That is, floating-point variables are interpreted as real numbers, and floating-point operations are interpreted as real arithmetic operations. The outputs produced by the precision-unspecific semantics are treated as the ideal outputs, and methods are used to approximate the ideal outputs in an execution. One commonly-used technique is *precision tuning*[1]. That is, the computation interpreted by the precision-unspecific semantics is executed in a high-precision floating-point format, such as double, long double, or higher, and the output is expected to be close enough to the ideal output. For example, several detection approaches [4, 8, 26] use a precision that is double of the original precision to get an accurate output, and the accurate output is compared with the original output to get the error on the original output.

*This work is supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A203, and the National Natural Science Foundation of China under Grant No.61421091, 61225007, 61529201, 61432001.

†Corresponding author.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950355

[1]In this paper, we use *accuracy* and *precision* differently. Accuracy means how close the output is compared with the ideal output. Precision means how many bits are used in computing the output.

However, a program may contain *precision-specific operations*, which would be difficult to be interpreted in a precision-unspecific way. Precision-specific operations are operations that are designed to work on a specific precision. For example, let us consider the following program, which is simplified from a code piece in `exp` function in GLIBC.

```
1:  double round(double x) {
2:    double n = 6755399441055744.0;
3:    return (x + n) - n;
4:  }
```

The goal of the program is to round `x` to an integer and return the result. The constant `n` is a "magic number" working only for double-precision floating-point numbers. When a double-precision number is added with `n`, the decimal digits of the sum are all rounded off due to the limited precision. When we subtract `n` again, we get the result of rounding `x` to an integer.

Since the operation is precision-specific, precision-unspecific semantics used in existing approaches usually cannot correctly interpret it. In the aforementioned semantics, the variables x and n would be recognized as real numbers, and the whole procedure would be interpreted as computing $x+n-n$, which is mathematically equivalent to x. When existing approaches follow such a semantics, they would produce undesirable results. For example, previously we mentioned that most detection techniques [4, 8, 26] rely on precision tuning to get the error of the program. When we compute this program in a higher-precision, e.g., all variables are long doubles, we would not be able to get the rounding effect as long doubles have enough digits to accommodate the decimal digits. As a result, the higher precision would return x in most cases, which is less accurate than the original program, and these detection approaches would report an incorrect error when they compare the outputs from the high-precision program and the original program.

To fix this problem, a direct method is to extend the existing precision-unspecific semantics, so that such precision-specific operations are correctly interpreted. For instance, the above procedures should be interpreted as "rounding `x` to integer" rather than "adding n to x and then subtracting n away". However, it is difficult to make such an extension. First, it is difficult to capture all precision-specific operations. The example shows only one form of precision-specific operation. There may be other types of precision-specific operations, such as adding a different constant, subtracting a constant, and perform bit operations. It is difficult to enumerate all such forms. Second, even if we can enumerate all such forms, it is difficult to interpret all operations in a precision-unspecific way. We need to understand the intention of the operations and map it to real numbers. In the example program, we need to understand its rounding behavior rather than interpreting it as adding a large constant. Third, as can be seen from the example, the semantics is not syntax-directed. We need to understand at least the value-flow of the program to correctly interpret the program. This makes the semantics difficult to define and to implement.

To deal with this problem, in this paper we propose a lightweight approach to precision-specific operations. Particularly, we focus on detection approaches based on precision tuning [4, 8, 26], aiming to reduce the false positives caused by precision-specific operations. First, we propose a heuristic to detect precision-specific operations. This heuristic is based on the observation when a precision-specific operation is incorrectly interpreted in precision-unspecific semantics, executing the semantics in a high precision and in a low precision usually produce large relative errors, except for the variables storing errors. Second, we propose a fixing approach to enable precision tuning under the presence of precision-specific operations. The basic idea of the fixing approaches is, rather than trying to interpret precision-specific operations in a precision-specific way, we always execute the operations in the original precision. As our evaluation will show later, the fixing approach leads to overall more accurate output than both the original precision and the high precision.

Based on our approach, we performed a systematic study of precision-specific operations in the GNU C standard math library, v2.19. We used our approach to detect precision-specific operations in the library and to fix them for high-precision execution. We also manually evaluated the reported precision-specific operations and summarized them into patterns. Our study leads to several findings: (1) a significant number of code fragments in the standard C math library are precision-specific operations, and some large inaccuracies reported in existing studies [26, 2] are false positives or potential false positives due to precision-specific operations; (2) there are three main patterns of precision-specific operations, namely rounding, multiplication, and bit operations; (3) our detection approach has high precision and recall; (4) our fixing approach can produce overall more accurate result than both the original programs and programs with raised precision, and the automatic fix has a close performance to the manual fix.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes precision-specific operations and discusses their effect on existing tools. Section 4 demonstrates how to detect precision-specific operations under a certain precision-unspecific semantics. Section 5 describes how to fix precision-specific operations. Section 6 shows our experiment on analyzing precision-specific operations and evaluation of our detection and fixing approach. Section 7 concludes the paper.

2. RELATED WORK

2.1 Inaccuracy Detection

Many approaches treat large inaccuracies as bugs and try to identify such bugs from programs. One of the earliest approaches is FPDebug [4], which estimates the error on the result of an execution. FPDebug dynamically analyzes the program by performing all floating-point computations side by side in higher precision and produces two results, the result in high-precision and that in low-precision. Then the result in high precision is used as ground truth to calculate the error.

Based on the approach of FPDebug, several detection approaches [26, 8, 23] have been proposed. The goal of a detection approach is to locate a test input that could maximize the error on the output. The current approaches all rely on search algorithms, where typical algorithms include LSGA [26] and BGRT [8], while LSGA is reported to have a better performance than BGRT [26].

An approach that works differently is the dynamic analysis proposed by Bao and Zhang [2]. This approach is designed to reduce the high overhead in FPDebug. Instead of executing in high-precision, it relies on a heuristic to detect large errors:

if an operation whose operands are large on exponents, and the result is small on exponents, the result probably contains large errors. In this way, only slight overhead is introduced to the original execution.

All these approaches interpret the program using similar precision-unspecific semantics, which interprets floating-point numbers as reals and floating-point operations as real operations. As a result, these approaches could not correctly interpret precision-specific operations, and may report large errors that do not exist in the program.

Our detection approach could help identify the false large errors reported by these approaches. Our fixing approaches could still report an accurate result under the existence of precision-specific computations, allowing these approaches to continue to work. As our evaluation will show later, several inaccurate program reported in existing papers [26, 2] are false positive or potentially false positive due to precision-specific operations, and our fixing approach could help produce accurate results on these programs.

2.2 Program Optimization

While increasing precision is commonly used for getting more accurate results, reducing precision is commonly used in program optimization to get faster code. Lam et al. [14], Rubio-González [19], and Schkufza et al. [20] propose approaches that automatically reduce the precision of floating-point computation to enhance performance with an acceptable loss of precision. These approaches typically try to locate, typically by search algorithms, variables in the program, which have only limited effect on the accuracy of the result when replaced by variables with lower precisions, and replace these variables by lower-precision ones.

Since these approaches rely on precision-unspecific semantics, they may incorrectly interpret precision-specific operations. When a variable is involved in a precision-specific operation, the variable probably cannot be selected by these approaches, as the change of its precision would lead to significant change in the estimated accuracy of the result. As a result, all such variables would never be selected for optimization, potentially reducing the effectiveness of the optimization. Our fixing approach could potentially be combined with these approaches to enable the selection of these variables: the variables will be computed as low-precision in most of the time, and be computed in the original precision only during the precision-specific operations. In this way, we can gain the performance boost without losing much precision.

Another optimization direction is to increase the accuracy of the computation. Darulova and Kuncak [10] propose an approach that automatically increases precision to guarantee the final result meets the accuracy requirement. Panchekha et al. [17] propose an approach that searches the program fragments that cause significant errors and rewrite them into more accurate versions. However, these approaches still use precision-unspecific semantics, and thus may rewrite precision-specific operations into an undesirable direction. By combining this approach with our detection approach, such cases could potentially be avoided.

2.3 Verification

A lot of efforts are put into verification techniques, trying to verify whether the error produced by a program is always within a threshold. Typical approaches [18, 12, 9] combine dataflow analysis with interval [13] or affine arithmetic [21] to get the upper bound of the errors. However, these approaches also rely on precision-unspecific semantics, thus may erroneously interpret precision-specific operations and report imprecise or unsound upper bound. Our approach could potentially be combined with these approaches to provide more informed result: the precision-specific computations could also be reported to the users to help the users determine where the upper bounds may be imprecise or unsound.

Besides full automatic verification, several approaches have been proposed to facilitate manual verification for floating-point programs. Boldo et al. [5, 6] build support for floating-point C programs in Coq, allowing one to easily build proofs for floating-point C program. Ayad and Marché [1] propose the use of multiple provers to try to automate the proofs of floating-point properties. Whether these approaches are affected by precision-specific operations depends on how the verified properties are specified. If a property is inferred from the program, these approaches may also be affected because the precision-specific operations may be erroneously interpreted.

2.4 External Errors

So far we only discuss internal errors, which is about how much inaccuracy may be introduced during the execution of the program. Some approaches concern about external errors, which is about the stability of the program: how much the errors already carried on the input will be magnified during the computation of the program. Recent work includes static verification of robustness by Chaudhuri et al. [7], dynamic analysis by Tang et al. [22], and dynamic sampling by Bao et al. [3]. These approaches are not affected by precision-specific operations.

3. OVERVIEW

In this section, we demonstrate how the correctness of FPDebug [4], an existing approach we concerned particularly, is affected by precision-specific operations.

3.1 Semantics of FPDebug

Here we informally describe the precision-unspecific semantics of FPDebug. As introduced in Section 2, FPDebug is a dynamic analysis approach based on precision tuning for detecting the error and canceled bits produced in one execution of the program. Other detection approaches are either directly built upon FPDebug [26] or are designed to behave similarly to FPDebug [2, 8].

FPDebug implements precision tuning by instrumenting the binary code. For each address that can be possibly accessed as a floating-point variable in the program, either on stack or on heap, a shadow variable is created for the address. The precision of the shadow variable can be specified by a parameter, which is 120-bit by default. Whenever a floating-point operation $v_o = \diamond v_1$ or $v_o = v_1 \circ v_2$ is performed, where $\diamond$ denotes an unary operator and $\circ$ denotes a binary operator, the system first checks whether the shadow variables of v_1 and v_2 are initialized, and if so, the same operation is performed in high precision, where the result is stored in the shadow variable of v_o. If the shadow variable of v_1 or v_2 is not initialized, the value of the corresponding variable is converted into the high precision to initialize the shadow variable. When the program ends, the difference between

the shadow variable of the output and the original output is taken as the error of the program.

A dedicated point is that the difference between the shadow variables and the original variables may cause a conditional statement to take a different branch. To enable side-by-side comparison of each statement, FPDebug always takes the branch that is chosen in the original precision.

As we can see, the semantics FPDebug uses to interpret floating-point operations are precision-unspecific. The operations are interpreted as operations on real numbers and thus can be adapted to a different precision. Similarly, the floating-point numbers are interpreted as reals such that can be adapted to a different precision.

3.2 FPDebug on Precision-Specific Operations

Now we use two examples to show how FPDebug shall report false errors on precision-specific operations. The first example is the example program we have shown in Section 1. As explained in Section 1, the purpose of the program is to round `x` to an integer. However, when executed in FPDebug, two shadow variables would be created for `x` and `n`, respectively, and a high-precision computation for `x+n-n` would be performed. Since the high-precision variables can represent a much more precise scope than the original precision, the high-precision computation would produce a result that is closer to the original `x` for most inputs. On the other hand, the low-precision would produce the accurate result of rounding `x` to an integer for most inputs. As a result, false errors would be produced when `x` is not an integer.

In the second example, let us consider the following program, which is also a typical example of precision-specific operations and is reduced from `log` function in GLIBC.

```
1:  union Double2Int {
2:      int i[2];
3:      double d;
4:  };
5:
6:  double test(Double2Int x) {
7:      x.d = calc();
8:      x.i[HIGH_HALF] = (x.i[HIGH_HALF] & 0xFFFFF) | 0x40000000;
9:     return x.d;
10: }
```

The purpose of the code is to get μ, such that

$$\mu \cdot 2^n = |x.d|, \mu \in [2, 4)$$

x is the input of function `test`, and μ is the output. The union type `Double2Int` allows us to manipulate different bits of a `double` variable directly: `x.i[LOW_HALF]` represents the lower half of `x.d`, `x.i[HIGH_HALF]` represents the higher half of `x.d`, and bit operators can be directly applied to `x.i[LOW_HALF]` and `x.i[HIGH_HALF]`. The values of `HIGH_HALF` and `LOW_HALF` depend on the infrastructure of the machine. The `0xFFFFF` extracts the significand of double, while `0x40000000` sets the exponent to 1 and the sign bit to 0. This operation is obviously precision-specific and is not easy to map to other precisions: we need the knowledge of floating-point format to properly map the constants to other precisions.

When executing this program in FPDebug, FPDebug would create a shadow variable for `x.d`. However, the operation at line 8 is not a floating-point operation, so the high-precision computation on shadow variables would not be performed. As a result, when the function returns, the original value is changed by line 8 but the shadow value is not changed, leading to a false error reported by FPDebug. In other words, the operation at line 8 is precision-specific, and when FPDebug maps the operation to high-precision, it erroneously maps this operation to an empty operation which does nothing.

Note that one may argue that the problem of this example is a bug in FPDebug implementation: the implementation does not recognize the alias relation between `x.d` and `x.i`. However, although we know the shadow value of `x.d` should be changed, we do not know how to change it as the change operation is specific to the original precision. For example, we need to construct constants to replace `0xFFFFF` and `0x40000000` in other precision.

4. DETECTING PRECISION-SPECIFIC OPERATIONS

4.1 Precision Tuning

To describe the detection approach, let us first define precision tuning. Our definition is independent of any particular precision-unspecific semantics, so that our approach can be integrated into different detection tools with different semantics. We assume an instruction set L containing all instructions that can be written in a program. Each instruction l takes the form $v_o = f(v_{i_1}, \ldots, v_{i_n})$, v_o is known as the output variable, denoted by $out(l)$, the set $\{v_{i_1}, \ldots, v_{i_n}\}$ is known as the input variables, denoted by $in(l)$, and f is a function computing the output from the input. We do not consider jump instructions because we follow the side-by-side execution in FPDebug, where the high-precision program is always executed on the same path as the low-precision program. A *precision tuning function*, $\phi : L \to L$, takes an instruction as input and produces an instruction in the high precision as output, where for any $l \in L$, $in(l) = in(\phi(l))$ and $out(l) = out(\phi(l))$.

For example, in the first example program in Section 1, line 3 becomes two instructions at the binary level: `tmp=x+n` and `ret=tmp-n`. The output variable of `tmp=x+n` is `tmp`, while the input variables are `x` and `n`. The precision tuning function of FPDebug maps these two instructions to the same operations in the high precision. In the second example program in Section 3.2, line 8 also becomes two instructions. Since FPDebug fails to correctly capture the two instructions, we consider FPDebug maps the two instruction to void instructions, which do nothing.

Next, we describe how to estimate the error on a variable by comparing values from the original and the high-precision executions. We adopt the same side-by-side execution model as FPDebug: the high-precision execution always follows the same path as the original execution. Let l be an instruction in a program, v_o be the value of $out(l)$ after the execution of l in the original execution, and v_h be the value of $out(l)$ after the execution of $\phi(l)$ in the high-precision execution. The *estimated relative error*, or shortly *error*, of output variable $out(l)$ at l is $\left|\frac{v_h - v_o}{v_h}\right|$. Similarly, let $x \in in(l)$ be an input variable of l, v_o be the value of x before the execution of l in the original execution, and v_h be the value of l before the execution of $\phi(l)$ in the high-precision execution. The estimated relative error of input variable x at l is $\left|\frac{v_h - v_o}{v_h}\right|$.

For example, in the aforementioned two instructions, the error of output variable `tmp` at `tmp=x+n` is calculated by

comparing the values of `tmp` in the two executions after executing the instruction, while the error of the input variable `x` is calculated by comparing the values of `x` in the two executions before executing the instruction.

4.2 The Heuristics

4.2.1 Possibly Precision-Specific Operations

Given two thresholds E and P, we detect precision-specific operations based on a heuristic. The basic heuristic is as follows, which is based on executing the program on a large number of input values.

Definition 1. Let l be an instruction, e_i be the maximum error among the input variables of l at l in an execution, and e_o be the error of the output variable at l in the same execution. We say l is *possibly precision-specific* if $e_o/e_i > E$ on at least P% of the executions.

In other words, if the error inflates significantly after an operation for many inputs, we consider the operation possibly precision-specific.

We explain the intuition of the heuristic from two aspects. **First, precision-specific operations usually meet this criterion.** Recall our goal is to reduce the false errors caused by precision-specific operations. If a detection approach reports a false error due to a precision-specific operation, the precision-specific operation must have been interpreted incorrectly by the precision-unspecific semantics. As shown by our examples, when the precision-specific operation is incorrectly interpreted, the high-precision program is usually largely different from the original program, and thus will cause a significant inflation of the error.

For example, in the first example program in Section 1, the operation "rounding to integer" is interpreted as an identity transformation. As a result, the original program rounds `x` to an integer, while the high-precision program directly returns `x`. Unless `x` is a very large number or `x` is exactly an integer, a significant relative error will be reported. In the second example program in Section 3, the first instruction at line 8,

`tmp=x.i[HIGH_HALF]&0xFFFFF`,

is interpreted as a void operation. As a result, `tmp` is assigned in the original program but remains a random uninitialized value in the high-precision program, which leads to large errors in most cases.

Second, normal operations seldom meet the criterion. As analyzed in many existing papers [4, 2], there are two possibilities of large errors in normal operations: accumulation and cancellation. Accumulation indicated that errors propagate through a sequence of operations and are accumulated on variables in those operations. Though the error from each operation is small, the accumulated error can be large. Cancellation indicates that the subtraction of two close numbers, where many significant digits will be subtracted off, may cause large errors. Obviously, accumulation cannot occur in one operation, and thus the only possibility to cause large error inflation by one operation is cancellation.

However, when a cancellation causes large errors, the input variables need to be close to each other and at least one of the operands contains rounding errors. Among the whole input space, such input pairs only consist of a small portion. Therefore, an operation that is not precision-specific would probably not cause cancellation on many inputs.

4.2.2 Probably Precision-Specific Operations

While the heuristic works for most of the time, it would incorrectly identify a normal case as precision-specific: the calculation of errors. Many algorithms that aim to produce an accurate result use an extra variable to store the error. Given an operation $z = f(x, y)$, the floating-point computation result z' is usually different from the ideal result z due to the rounding errors. To make the final result more accurate, a typical accurate algorithm uses an additional operation g to get the error e_z on z', where $e_z = g(x, y, z')$. Though we cannot store the accurate value of z as a floating-point number, we can usually store the error e_z accurately as a floating-point number. So this error is stored as a separate floating-point number and will be used to correct the final result or estimate the error of the final result later. For example, Dekker proposes an algorithm [11] to get the error (called *correction term*) in the addition operation. Suppose we are going to calculate $z = x + y$. The error can be calculated as $e_z = x - (x + y) + y$ in floating-point operations. This kind of correction is widely used in real world programs such as the C standard math library.

However, given an operation and its high-precision form, the errors produced by the two operations are usually largely different. The estimated error on the output is likely to be much larger than the errors on the input. As a result, if we rely on Definition 1, we may incorrectly identify such operations as precision-specific operations.

To overcome this problem, we further introduce a negative condition to filter out these false detections. When a variable is used to store errors, the errors are usually very close to zero, and we can use this feature to filter out these false detections. Given three thresholds V_o, V_h, and Q, we have the following definition.

Definition 2. A possibly precision-specific instruction l is *probably precision-specific* if on at least Q% of the executions, the output value of the instruction is smaller than V_o at the original precision and smaller than V_h at the high precision.

4.3 Detection Algorithm

Based on the heuristic, it is easy to derive our detection algorithm to detect precision-specific operations. The algorithm consists of the following steps.

- Sampling. We first sample a large number of input values to execute the program. We sample the space of each parameter within a fixed interval. The purpose of sampling within a fixed interval rather than random sampling is to evenly cover the input space.
- Execution. We execute the program with the input values. The high-precision program is executed side-by-side with the original program. At the execution of each floating-point operation, we record the estimated relative errors of the input and the output variables.
- Data Filtering. We calculate the error inflation by each instruction execution l in an execution, and once we identified an inflation larger than E, we filter out all data collected from the rest of this execution. This is because l may be a precision-specific operation, and thus the later operations may be affected by the error propagated from that operations. The relative errors of later operations may be much different from a normal execution.

- Detection. We analyze the recorded data and determine the probably precision-specific operations based on Definition 2. Note that since we filter out all instruction executions after a large error inflation, we may only detect the precision-specific operations that are executed early on the sampled inputs.
- Fixing and Repeating. We use the method introduced in the next section to fix the precision-specific operations and restart the process to detect more precision-specific operation.

5. FIXING PRECISION TUNING

In this section, we describe how to fix precision tuning for precision-specific operations. Our idea is to reduce the precision of the variables right before the precision-specific operation, and raise the precision right after the precision-specific operation. In this way, the precision-specific operation is still performed in the original precision, while most operations are performed in the high-precision, so the overall result would be more precise than both the original precision and the high precision without the fix.

Please note that a precision-specific operation may be detected by our approach at one instruction, but the operation itself may not contain only this instruction. In out first example, the operation contains two instructions: `tmp=x+n` and `ret=tmp-n`, but our detection approach would report only at the latter instruction.

To mark the boundaries of the operations, we introduce two fixing statements, `ps_begin(`v_1`,...,`v_n`)` and `ps_end (`v_1, ..., v_n`)`. The programmers could use the two statements to precisely mark the boundary of a precision-specific operation. The variables v_1,...,v_n are variables that would be used in the precision-specific operation, whose precision will be reduced at `ps_begin` and be resumed at `ps_end`.

It is also desirable to make the insertion of the fixing statements automatically. We note that, for many precision-specific operations, reducing the precision at the correct boundaries is the same as reducing the precision at the last instruction. For example, in the first example, if we perform `tmp=x+n` in the high precision and reduce the precision of `tmp` after the statement, the decimal part of `x+tmp` will still be rounded off. Since our detection approach can correctly detect the last instruction in a precision-specific operation, automatically inserting `ps_begin` and `ps_end` around the detected instruction would also produce the desirable result in many cases. Therefore, we also implement an approach that automatically inserts the two statements. As will be shown later in the evaluation, with a proper E in the detection approach, the automatic fix can achieve a close performance to the manually inserted ones.

6. EVALUATION

Our experiments are designed to answer the following research questions.

- (RQ1) How many precision-specific operations are detected and what are the main patterns?
- (RQ2) How precise and complete is our detection approach?
- (RQ3) How different are possibly and probably precision-specific operations?
- (RQ4) How effective is our fixing approach?
- (RQ5) Are the errors reported in existing studies affected by precision-specific operation?

6.1 Implementation

We have implemented our approach by modifying the FPDebug code. Basically, we reused the same instrumentation mechanism of FPDebug to implement the precision tuning and added additional statements for fixing precision tuning, which include manipulations of different precision values. We also instrumented code to track the non-floating-point operation of a union at the high precision, which was not tracked by FPDebug.

To precisely analyze our results, we used MPFR, a C library for multiple-precision floating-point computations with correct rounding. In this way, we can ensure the relative errors are accurately calculated without significant floating-point errors. Note that FPDebug also uses MPFR for precision tuning and relative error calculation.

Our implementation and all experimental data are available at the project web site [2].

6.2 Experimental Setup

6.2.1 Subjects

We evaluated our approaches on the functions of the scientific mathematical library of the GNU C Library (GLIBC), version 2.19. Most math functions in the GNU C Library contain three versions for float, double, and long double. A version for a particular precision takes input values at that precision, performs the calculation at that precision, and produces the result at that precision in most cases. Since many implementations for different versions are similar, we take only the double version, which includes 48 functions. The subjects cover calculation such as trigonometric functions, inverse trigonometric functions, hyperbolic functions, logarithmic functions, exponentiation functions, etc. Each function takes 1-3 double-precision floating-point or integer input parameters and produces a double-precision floating-point or integer output[3]. A more detailed description of the subjects is available on the project web site.

6.2.2 Oracles

To understand the performance of our fixing approach, we need to know the ideal output values of the functions at a given input. To get such ideal output values, we re-implemented the subject functions using a calculator with controlled error [25][24]. With a specified accuracy requirement, e.g., the digits should be accurate at least for two decimal places, the calculator guarantees that the result meets the accuracy, by dynamically raising the precision when necessary.

Since different operations require different calculation methods and some are yet unknown, the calculator supports only a limited set of operations. Using these operations, we implemented in total 21 GLIBC mathematical functions, which are listed in Table 1. The second column shows how we encode the function using the operations provided by the calculator. As we can see from the table, the 21 functions are the most commonly used math functions of GLIBC.

[2]https://github.com/floatfeather/PSO

[3]Function `s_fmaf` takes float as input and output, but the computation is done with double, so we still include it in our subjects

Table 1: Formulas for scientific functions in GLIBC.

Function	Formula
acos	$arccos(a)$
acosh	$ln(a+(a^2-1)^{\frac{1}{2}})$
asin	$arcsin(a)$
asinh	$ln(a+(a^2+1)^{\frac{1}{2}})$
atan	$arctan(a)$
atan2	$arctan(\frac{a}{c})$
atanh	$\frac{1}{2} \times ln(\frac{(1+a)}{(1-a)})$
cos	$cos(a)$
cosh	$\frac{(e^a+e^{-a})}{2}$
exp	e^a
exp2	2^a
exp10	10^a
log	$ln(a)$
log2	$\frac{ln(a)}{ln(2)}$
log10	$log(a)$
log1p	$ln(a+1)$
pow	a^c
sin	$sin(a)$
sinh	$\frac{(e^a-e^{-a})}{2}$
tan	$tan(a)$
tanh	$\frac{(e^a-e^{-a})}{(e^a+e^{-a})}$

6.2.3 Parameters

We need to set the thresholds in our detection approach, E, $P\%$, V_o, V_h and $Q\%$. To set the thresholds, we perform a preliminary study on function *exp*. The result suggests that changing E has some effects on the result while changing the other thresholds have little effect. As a result, we set five values to E: 10^4, 10^6, 10^7, 10^8 and 10^{10}. For the other thresholds, we set them with fixed values: $P\%$ with 70%, V_o with 10^{-9}, V_h with 10^{-15}, and $Q\%$ with 10%.

We also need to set the accuracy requirement of the calculator with controlled error. We require the result to be accurate for at least 400 decimal places, which are guaranteed to cover the dynamic range of double-precision floating-point numbers.

We also need to set the interval for sampling the input space. Since each function has a different input space, we determine the interval based on the size of the input space, where each function is sampled 1000 to 5000 times.

6.2.4 Procedures

We ran our tool to detect precision-specific operations on all subjects. Note that in the detection approach we need to fix all detected operation to discover more operations. We used the automatic fix in the detection process. So, in the end, we already have all operations discovered and fixed. We repeated the process five times setting the threshold E with five different values.

Next, we reviewed all precision-specific operations discovered in all experiments and determined the false positives. Then we manually inserted fixing statements for all true positives and ran the fixed functions again for all inputs to get the result of the manual fix.

Finally, we review the inaccuracies reported by two existing papers [26, 2], and check whether the result may be affected by the precision-specific operations we detected.

Our detection approach used 9819s to analyze all 48 functions, in average 205s for one function.

Table 2: Precision-specific operations per function

Function	Number	Function	Number	Function	Number
acos	1	acosh	1	asin	3
asinh	1	atan	2	atan2	4
cos	11	cosh	2	erf	2
erfc	2	exp10	2	exp2	1
exp	2	gamma	1	j0	17
j1	17	lgamma	1	log	1
pow	7	sin	15	sincos	22
sinh	2	tan	2	y0	17
y1	17				

Table 3: Precision-specific operations per files

Function	Number	Function	Number
e_asin.c	3	e_atan2.c	4
e_exp2.c	1	e_exp.c	4
e_pow.c	5	e_log.c	1
s_atan.c	2	s_sin.c	24
s_tan.c	2	dosincos.c	2

6.3 Results

6.3.1 RQ1: How many precision-specific operations are detected and what are the main patterns?

In the 48 tested functions, there are 25 functions detected to contain precision-specific operations. The functions are listed in Table 2, where the number column shows the number of precision-specific operations detected when testing the function. In total, 153 precision-specific operations are detected. Note that different functions may call the same internal functions, so there are duplicated operations among different functions. After excluding duplicated operations, we have 48 unique probably precision-specific operations, which spread over 10 files as shown in Table 3.

This result indicates that precision-specific operations are widely-spread in the C standard math library. Note that functions in the C standard math library are basic building blocks of other applications performing floating-point computation, so any applications calling these functions are affected by precision-specific operations.

We also take a look at C math library functions with the two different precisions, float and long double. We found that, when the implementation algorithms for different versions are similar, similar precision-specific operations can also be found in the other versions.

By manually reviewing all probably precision-specific operations, we identify three main patterns, namely, rounding, multiplication, and bit operation.

Rounding. The rounding pattern takes the form of $x = (x+n)-n$, where x is a variable and n is a constant. We have already seen an example of the rounding pattern in the example in Section 1. When applying the pattern, different n can be used to round x to different levels. Generally, the larger the n is, the more bits are shifted off. A list of constants we found in the C math library is listed in Table 4.

Multiplication. The multiplication pattern takes the following form, where n is a constant and a is a variable.

$$t = n * a$$
$$a_1 = t - (t - a)$$
$$a_2 = a - a_1$$

Table 4: Values of constants.

Constant name	Value	Type
big	1.5×2^{45}	Rounding
big	1.5×2^{36}	Rounding
bigu	$1.5 \times 2^{42} - 724 \times 2^{-10}$	Rounding
bigv	$1.5 \times 2^{35} - 1 + 362 \times 2^{19}$	Rounding
THREEp42	1.5×2^{41}	Rounding
three33	1.5×2^{34}	Rounding
three51	1.5×2^{52}	Rounding
toint	1.5×2^{52}	Rounding
TWO52	2^{52}	Rounding
t22	1.5×2^{22}	Rounding
t24	2^{24}	Rounding
CN	$2^{27} + 1$	Multiplication

Table 5: Detecting probably precision-specific operations (without duplicate).

Threshold E	10^4	10^6	10^7	10^8	10^{10}
Rounding	44	44	43	42	39
Multiplication	2	2	2	1	0
Bit operation	2	2	2	2	2
False positive	30	16	14	9	7
Total	78	64	61	54	48
Precision	61.54%	75.00%	77.05%	83.33%	85.41%
Recall	100%	100%	97.92%	93.75%	85.42%

This block of code splits a into two components a_1 and a_2 using the constant n. Later, the two components can be used to calculate an accurate multiplication of a using an existing algorithm [11, 16].

This operation is precision-specific because n needs to be determined based on the precision of a. The value of n for double precision is also listed in Table 4, which is identified from the detected precision-specific operations in our experiments.

Bit Operation. The bit operation pattern treats a floating-point number as an integer and apply bit operators to the integer. We have seen an example of bit operation in the example program in Section 3.2. Since the manipulation of bits is related to the format of the specific precision, the operations are naturally precision-specific.

The identification of the three patterns from the 48 operations show that there may be a limited number of patterns for precision-specific operations, showing the hope of improving the precision-unspecific semantics to handle these cases. However, the three patterns are identified only from the C math library, and we do not know how complete the pattern set is. Furthermore, we also do not see easy ways to interpret these patterns precision-unspecifically.

6.3.2 RQ2: How precise and complete is our detection approach?

The result of the detected probably precision-specific operations is shown in Table 5. We show the numbers of each pattern, false positives, the total numbers, and the precisions for each threshold E without duplicates.

We make the following observations from the table.

- Our approach has an overall high precision. At the threshold 10^{10}, our approach has a precision of 85.41%. A possible reason for the false positives, as we suspect, is the insufficient sampling at some instructions. There are instructions at which only a few inputs can reach, and we have to determine whether they are precision-specific based on the few executions, which are not statistically significant.
- Our approach has an overall high recall. At the thresholds 10^4 and 10^6, our approach achieves a recall of 100%.
- The threshold E has a significant effect on the precision and recall. The higher the threshold, the higher the precision, but lower the recall. Threshold 10^6 is superior to 10^4 in both recall and precision, and no other threshold can be completely superior to another threshold on both precision and recall.

Table 6: Detecting possibly precision-specific operations (without duplicate).

Threshold E	10^4	10^6	10^7	10^8	10^{10}
Rounding	44	44	43	42	39
Multiplication	2	2	2	1	0
Bit operation	2	2	2	2	2
False positive	94	82	82	76	72
Total	142	130	129	121	113
Precision	33.80%	36.92%	36.43%	37.19%	36.28%
Recall	100%	100%	97.92%	93.75%	85.42%

Note that when calculating the recall, we treat the detected precision-specific operations on all threshold as the complete set. Theoretically, there can be precision-specific operations that are not detected at any threshold in our approach. However, as will be shown later in Table 7, our approach achieved very small relative errors compared to the oracles after we fixed all identified precision-specific operations, so it is likely that we have identified all precision-specific operations that cause large false errors.

6.3.3 RQ3: How different are possibly and probably precision-specific operations?

This question concerns about the usefulness of the negative condition we used to determine probably precision-specific operations. Table 6 shows the result of detecting possibly precision-specific operations, which does not use the negative condition to filter the operations calculating errors. By comparing Table 5 and Table 6, we can see that (1) the negative condition is very effective in filtering out false positives, where the precision has an increase up to 49%, (2) the negative condition did not filter out any true positives, as the recalls between the two tables are the same for all thresholds. The result indicates the negative condition is effective in improving the performance of our approach.

6.3.4 RQ4: How effective is our fixing approach?

Table 7 shows the average relative errors of each function after fixing precision tuning for the detected precision-specific operations. Here we show the result of both automatic fix and manual fix, where the automatic fix is performed based on the detection at four E thresholds: 10^6, 10^7, 10^8, 10^{10}. We do not include 10^4 because it is inferior to 10^6 on both precision and recall.

Note that there are three factors that affect the precision of the automatic fix. (1) Precision: the lower the precision, the more false positives are identified, leading to more instructions unnecessarily executed in low precision. (2) Recall: when we missed a precision-specific operation, failing to fix

Table 7: Average relative error to standard value. Functions with * have precision-specific operations. OP: original precision. HP: high precision. FOD: fix only the detected instruction on true positives.

Subject	Automatic Fixing				Manual	FOD	OP	HP
	$E = 10^6$	$E = 10^7$	$E = 10^8$	$E = 10^{10}$	–	–	–	–
acos*	$1.6106E-19$	$1.6106E-19$	$5.8999E-11$	$9.7265E-20$	$9.7265E-20$	$9.7265E-20$	$4.1048E-17$	$9.7265E-20$
acosh*	$1.0333E-17$	$1.0333E-17$	$1.0333E-17$	$1.0333E-17$	$1.0329E-17$	$1.0333E-17$	$4.3585E-17$	$5.5122E+04$
asin*	$2.0898E-19$	$2.0898E-19$	$2.0898E-19$	$2.0898E-19$	$2.0898E-19$	$2.0898E-19$	$3.9899E-17$	$2.0898E-19$
asinh*	$1.3004E-17$	$1.3004E-17$	$1.3004E-17$	$1.3004E-17$	$1.3000E-17$	$1.3004E-17$	$4.3561E-17$	$7.5351E+03$
atan2*	$1.3947E-18$	$1.3947E-18$	$1.3947E-18$	$1.3947E-18$	$1.3947E-18$	$1.3947E-18$	$4.0843E-17$	$1.3947E-18$
atan*	$4.0231E-19$	$4.0231E-19$	$4.0231E-19$	$4.0231E-19$	$4.0231E-19$	$4.0231E-19$	$3.5873E-17$	$4.0231E-19$
atanh	$3.3868E-17$	$3.3868E-17$	$3.3868E-17$	$3.3868E-17$	$3.3868E-17$	$3.3868E-17$	$4.9363E-17$	$3.3868E-17$
cos*	$6.0255E-19$	$2.3145E-21$	$2.3145E-21$	$2.3145E-21$	$2.3145E-21$	$2.3145E-21$	$3.7273E-17$	$2.5843E-03$
cosh*	$4.2901E-22$	$4.2901E-22$	$4.2901E-22$	$4.2901E-22$	$4.2901E-22$	$4.2901E-22$	$4.4157E-17$	$9.5417E-07$
exp10*	$8.4842E-23$	$8.4842E-23$	$8.4842E-23$	$8.4842E-23$	$8.4842E-23$	$8.4842E-23$	$6.9779E-17$	$1.2630E-06$
exp2*	$6.0300E-20$	$6.0300E-20$	$6.0300E-20$	$6.0300E-20$	$6.0300E-20$	$6.0300E-20$	$4.1163E-17$	$3.2491E-04$
exp*	$1.9083E-25$	$1.9083E-25$	$1.9083E-25$	$1.9083E-25$	$1.9083E-25$	$1.9083E-25$	$4.1266E-17$	$9.5402E-07$
log10	$8.1172E-18$	$8.1172E-18$	$8.1172E-18$	$8.1172E-18$	$8.1172E-18$	$8.1172E-18$	$4.2657E-17$	$8.1172E-18$
log1p	$1.2697E-17$	$1.2697E-17$	$1.2697E-17$	$1.2697E-17$	$1.2697E-17$	$1.2697E-17$	$4.2729E-17$	$1.2697E-17$
log2	$6.3510E-18$	$6.3510E-18$	$6.3510E-18$	$6.3510E-18$	$6.3510E-18$	$6.3510E-18$	$4.3646E-17$	$6.3510E-18$
log*	$8.6195E-21$	$8.6195E-21$	$8.6195E-21$	$8.6195E-21$	$8.6195E-21$	$8.6195E-21$	$4.0822E-17$	$1.0916E+01$
pow*	$2.5147E-16$	$2.5147E-16$	$1.6663E-08$	$1.6663E-08$	$5.5188E-24$	$5.5188E-24$	$3.7539E-17$	$8.8444E-07$
sin*	$7.7354E-21$	$7.7354E-21$	$7.7354E-21$	$7.7354E-21$	$7.7354E-21$	$7.7354E-21$	$3.5656E-17$	$4.2733E-02$
sinh*	$1.7909E-17$	$1.7909E-17$	$1.7672E-17$	$1.7672E-17$	$1.7672E-17$	$1.7672E-17$	$4.6532E-17$	$5.3316E-07$
tan*	$1.7135E-17$	$1.7135E-17$	$1.7135E-17$	$1.7135E-17$	$1.7135E-17$	$1.7135E-17$	$3.8932E-17$	$1.4335E+13$
tanh	$2.5412E-19$	$2.5412E-19$	$1.6890E-19$	$1.6890E-19$	$1.6890E-19$	$1.6890E-19$	$5.9692E-18$	$1.6890E-19$

Table 8: Sign test for $E = 10^7$ and manual fixing. Functions with * have precision-specific operations. N: how many times the function is executed. O: original precision. H: high precision. >O(>H): the number of executions that are more accurate than the original precision (high precision).

		$E = 10^7$						Manual					
Function	N	$> O$	$< O$	p-value	$> H$	$< H$	p-value	$> O$	$< O$	p-value	$> H$	$< H$	p-value
acos*	1000	999	0	$1.87E-301$	17	15	$4.30E-01$	999	0	$1.87E-301$	2	0	–
acosh*	1001	869	131	$2.63E-135$	991	0	$4.78E-299$	869	131	$2.63E-135$	991	0	$4.78E-299$
asin*	1000	994	4	$1.54E-290$	18	13	$2.37E-01$	994	4	$1.54E-290$	18	13	$2.37E-01$
asinh*	1000	836	163	$8.19E-110$	959	0	$2.05E-289$	836	163	$8.19E-110$	959	0	$2.05E-289$
atan2*	3600	3518	82	$0.00E+00$	0	0	–	3518	82	$0.00E+00$	0	0	–
atan*	1000	994	5	$1.54E-288$	0	0	–	994	5	$1.54E-288$	0	0	–
atanh	1997	1380	616	$2.46E-67$	0	0	–	1380	616	$2.46E-67$	0	0	–
cos*	1257	1257	0	$0.00E+00$	1156	0	$0.00E+00$	1257	0	$0.00E+00$	1156	0	0
cosh*	1001	1001	0	$4.67E-302$	1001	0	$4.67E-302$	1001	0	$4.67E-302$	1001	0	$4.67E-302$
exp10*	1000	1000	0	$9.33E-302$	1000	0	$9.33E-302$	1000	0	$9.33E-302$	1000	0	$9.33E-302$
exp2*	1000	999	0	$1.87E-301$	999	0	$1.87E-301$	999	0	$1.87E-301$	999	0	$1.87E-301$
exp*	1000	1000	0	$9.33E-302$	1000	0	$9.33E-302$	1000	0	$9.33E-302$	1000	0	$9.33E-302$
log10	1000	889	111	$1.06E-151$	0	0	–	889	111	$1.06E-151$	0	0	–
log1p	1000	862	138	$7.91E-129$	0	0	–	862	138	$7.91E-129$	0	0	–
log2	1000	949	51	$1.74E-215$	0	0	–	949	51	$1.74E-215$	0	0	–
log*	4999	4998	1	$0.00E+00$	4484	0	$0.00E+00$	4998	1	$0.00E+00$	4484	0	0
pow*	1681	221	1379	$2.25E-205$	1600	0	$0.00E+00$	1600	0	$0.00E+00$	1600	0	0
sin*	1257	1257	0	$0.00E+00$	1157	0	$0.00E+00$	1257	0	$0.00E+00$	1157	0	0
sinh*	1000	834	165	$2.12E-108$	560	14	$6.83E-146$	835	164	$4.18E-109$	560	0	$2.65E-169$
tan*	1257	865	392	$1.07E-41$	942	0	$5.38E-284$	865	392	$1.07E-41$	942	0	$2.69E-284$
tanh	1001	213	6	$9.52E-57$	0	4	$1.25E-01$	214	5	$2.21E-58$	0	0	–

this operation may lead to significant errors. (3) Boundaries: the automatic fix cannot correctly detect the boundary but only one instruction, which may lead to instructions that should be executed in the original precision still executed in the high precision. The effects of the first two factors can be seen by comparing the results from different thresholds, but it is not easy to distinguish the last factor. To understand the effect of the last factor, we further perform a fix which fixes only the detected instruction in all true positives (FOD column).

Furthermore, as the baselines for comparison, we also list the errors produced by the original precision (OP column) and those by the high precision (HP column).

From the table we can make the following observations:

- The results from the manual fix are much more accurate than both the original precision and the high precision. On all functions, the average errors from the manual fix are the smallest, which are usually smaller than the original precision several orders of magnitude, and are greatly smaller than the high precision on many functions with precision-specific operations.

- The results from automatic fix are also more accurate than the original precision and the high precision in the vast majority of cases. At a proper threshold, the automatic fix can have a close performance to the manual fix. For example, For threshold 10^{10}, on all of the functions except `pow`, the error is close to the

Table 9: Inaccuracies reported in previous work.

Function	Reported error	Actual error	Paper
exprel_2	$2.85E+00$	$5.25E-12$	[26]
synchrotron_1	$5.35E-03$	$2.24E-13$	[26]
synchrotron_2	$3.67E-03$	$5.97E-15$	[26]
equake	around $5.00E-01$	–	[2]

manual fix. The error of `pow` is mainly because of relatively low recall under the high threshold 10^{10}.

- When the detection precision decreases, the error of automatic fix may increase because of excessive fixes. For example, the error of `tanh` increases when the threshold moves from 10^8 to 10^7.
- When the detection recall decreases, the errors of automatic fix may increase noticeably because of missing fixes. For `pow`, a precision-specific multiplication is ignored when thresholds are 10^8 and 10^{10}, which shows a significant increase in error.
- Fixing only detected instruction and fixing all instructions have almost identical performance. Only slight difference is observed from the function `asinh` and `acosh`.
- Precision-specific operations may not always have a significant effect on the result. For example, `acos` contains precision-specific operations, but the error from the high-precision is the same as the manual fix. This is because although large errors may be produced in the precision-specific operations, these errors may not have a large effect on the final result. For example, the result of precision-specific operation is converted to an integer. In the semantics, the shadow value of the integer is not tracked. The error of precision-specific operation is lost, which dramatically produces a right result.

The average errors cannot show on how many executions one approach is superior to another. To further get this information, we perform a sign test between the errors from the fixed executions and the errors from the original executions and high-precision executions. We chose the threshold at 10^7 for automatic fix, because 10^7 is a balance between precision and recall. The result is shown in Table 8.

We can see that the manual fix outperforms both the original execution and the high-precision execution, and all results are significant because p is much smaller than 0.05. Note some p-value cannot be calculated because there are too few positive and negative instances [15]. We can also see that the automatic fix outperforms the high-precision execution in all functions, and outperforms the original execution in all but the `pow` function. A further investigation reveals that the detection approach detects a false positive, where the detected instruction corrects the result with a stored estimated error. When we reduce the precision on this instruction, we will use a high-precision error to correct a low-precision result, leading to the inaccurate result.

6.3.5 *RQ5: Are the errors reported in existing studies affected by precision-specific operations?*

We reviewed the inaccuracies reported in two existing publications [26, 2] and checked whether they are possibly false positives caused by precision-specific operations. For each program reported to be inaccurate in the publications [26, 2], we investigate the source code of the program and collect the C math functions called by the source code. If a function contains precision-specific operations detected in our experiments, we determine the program as a potential false positive. Then we try to run the program using the input specified in the publication, but with the precision-specific operations fixed, and check whether the new error is significantly smaller than the reported error.

The result is shown in Table 9. In total, we confirmed three inaccuracies reported by Zou et al. [26] are false positives. We also identify a potentially false positive in the errors reported by Bao et al. [2]. The `equake` function is reported to be inaccurate, but this function calls `sin` and `cos`, both containing precision-specific operations, so the inaccuracy might be caused by precision-specific operations. However, we cannot confirm this because Bao et al. did not report what input values are used to trigger the large errors in their experiments. Though a range of a single number is reported, but the input to the `equake` function is a matrix and it is not clear for which number in the matrix the range is specified, nor the size of the matrix.

This result implies that precision-specific operations may have a non-trivial impact on existing detection approaches, and need to be treated properly to get more precise results.

6.4 Threats to Validity

The main threat to internal validity lies within our identification of the true and false positives in the detected operations. We have to manually read the code to identify whether an operation is precision-specific or not. To reduce this threat, we performed an additional process to validate the true positives. For each identified true positive, we removed the fixing statements for this operation, and ran the functions again for the inputs to see whether there is a significant increase in errors for most inputs. The result indicates our manual identification is correct.

The main threat to external validity is the representativeness of our subjects. The subjects are all from GLIBC, and it is not clear how much the results can be generalized to other subjects. Nevertheless, GLIBC is one of most widely-used open source floating-point library and has been contributed by a lot of developers. Thus, the precision-specific operations in GLIBC should cover a large number of patterns in writing precision-specific operations.

7. CONCLUSION

In this paper, we have presented a study around precision-specific operations. The study shows that precision-specific operations are widely spread, and, if not handled properly, may cause existing approaches to act incorrectly. As we analyzed, a lot of existing approaches may be affected by precision-specific operations and some inaccuracies reported in existing papers are false positives or potentially false positives due to precision-specific operations. Nevertheless, we show that precision-specific operations can be detected based on a simple heuristic. Also, though in general, it is difficult to interpret precision-specific operation correctly, we can have a lightweight solution that always execute the precision-specific operations in the original precision. As our results show, this lightweight solution successfully enables precision tuning under the presence of precision-specific operations.

8. ARTIFACT DESCRIPTION

8.1 Materials

The artifact is a replication package for our experiment, consisting of the following materials.

1. A tutorial on subject selection, experiment configuration and how to run experimental scripts.
2. A virtual machine image with all tools and subjects installed.
3. Open source tools the experiment depends on, such as FPDebug, MPFR.
4. Experimental scripts that instrument subjects, detect and fix precision-specific operations in subjects and analyze the results.
5. Experimental data like instrumented subjects, discovered precision-specific operations.

The artifact is published under MIT license (the open source tools and subjects are published under their original licenses) and can be downloaded[4].

8.2 Tutorial

The tutorial provides step by step guidance for reproducing our experiments. The tutorial covers two scenario: the virtual machine and manual installation on a new machine. The former is much simpler and is recommended. The later involves the installation and configuration of several tools, and the change of scripts, and is only recommended if you plan to modify our code for new tools and experiments.

8.3 Virtual Machine Image

The virtual machine has tools installed (FPDebug, GMP, MPFR) and configured. The two versions of instrumented GLIBC are also installed. The scripts and data are also placed in the virtual machine so that rerunning the experiment only needs a few commands, which are described in the tutorial.

8.4 Tools

Our experiments depend on several open source tools. We use FPDebug [4] to tune precision. FPDebug further relies on a modified MPFR, a C library for multiple-precision floating-point computations with correct rounding, for more accurate computations. LLVM and CLANG are used to instrument the subjects. All these tools are installed on the virtual machine except LLVM and CLANG.

8.5 Scripts

We provide scripts to run the experiment, including instrumentation, the detection and fixing approach, and analysis for the results. Here we briefly introduce how to replicate our experiment in the virtual machine.

The experiments are implemented as two steps. The first is to detect and fix precision-specific operations. The second is to analyze the results. Invoking the following command would perform the first step.

[4]https://github.com/floatfeather/PSO/tree/master/artifact

```
cd /home/artifact/work/exe-art
./run_2.sh
```

The second step is implemented as four different scripts. The first script analyzes the precision and recall of the detection approach, and can be invoked using the following commands.

```
cd /home/artifact/work/exe-art
g++ ErrorProcess.cpp -o ErrorProcess
./ErrorProcess
```

The rest three scripts analyze respectively for the three types of fixing in our experiment, and can be invoked by the following commands.

```
cd /home/artifact/work/base-art
./run_auto.sh
./run_manual.sh
./run_fixlast.sh
```

The results will be stored in `result_auto_E.csv`, `result_manual.csv` and `result_fixlast.csv`. `E` stands for the parameter `E` used in our detection approach.

8.6 Data

The data include subject functions and other data needed for the experiments, such as accurate outputs for each subject function. The data also include a list of precision-specific operations found in the double version of the GNU C math library. The tutotial gives a more detailed description of the data.

9. REFERENCES

[1] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *Proc. IJCAR*, pages 127–141, 2010.

[2] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In *Proc. OOPSLA*, pages 817–832, 2013.

[3] T. Bao, Y. Zheng, and X. Zhang. White box sampling in uncertain data processing enabled by program analysis. In *OOPSLA '12*, pages 897–914, 2012.

[4] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proc. PLDI*, pages 453–462, 2012.

[5] S. Boldo and J.-C. Filliâtre. Formal verification of floating-point programs. In *Proc. ARITH*, pages 187–194, 2007.

[6] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *Proc. ARITH*, pages 243–252, 2011.

[7] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *ESEC/FSE '11*, pages 102–112, 2011.

[8] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *PPoPP*, pages 43–52, 2014.

[9] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In *Proc. OOPSLA*, pages 325–344, 2011.

[10] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL*, pages 235–248, 2014.

[11] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[12] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proc. SAS*, pages 18–34, 2006.

[13] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[14] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *ICS*, pages 369–378, 2013.

[15] M. Neuhauser. *Nonparametric statistical tests: A computational approach*. CRC Press, 2011.

[16] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[17] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11, 2015.

[18] S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical software with result verification*, pages 306–313. 2004.

[19] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC*, pages 27:1–27:12, 2013.

[20] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, pages 53–64, 2014.

[21] J. Stolfi and L. De Figueiredo. An introduction to affine arithmetic. *TEMA Tend. Mat. Apl. Comput.*, 4(3):297–312, 2003.

[22] E. Tang, E. T. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proc. ISSTA*, pages 131–142, 2010.

[23] X. Zhang, D. Zhang, Y. Le Traon, Q. Wang, and L. Zhang. Roundtable: Research opportunities and challenges for emerging software systems. *Journal of Computer Science and Technology*, 30(5):935–941, 2015.

[24] S. Zhao. A calculator with controlled error, example section (in Chinese). http://www.zhaoshizhong.org/download.htm, 2015. [Accessed 12-February-2015].

[25] S.-Z. Zhao. Trusted "arithmetic expressions calculator". In *Proc. ControlTech (in Chinese)*, pages 7–15, 2014.

[26] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In *ICSE*, pages 529–539, 2015.