On-the-fly Detection of Instability Problems in Floating-Point Program Execution

Tao Bao Xiangyu Zhang
Department of Computer Science, Purdue University
{tbao, xyzhang}@cs.purdue.edu

Abstract

The machine representation of floating point values has limited precision such that errors may be introduced during execution. These errors may get propagated and magnified by the following operations, leading to instability problems, e.g., control flow path may be undesirably altered and faulty output may be emitted. In this paper, we develop an on-the-fly efficient monitoring technique that can predict if an execution is stable. The technique does not explicitly compute errors as doing so incurs high overhead. Instead, it detects possible places where an error becomes substantially inflated regarding the corresponding value, and then tags the value with one bit to denote that it has an inflated error. It then tracks inflation bit propagation, taking care of operations that may cut off such propagation. It reports instability if any inflation bit reaches a critical execution point, such as a predicate, where the inflated error may induce substantial execution difference, such as different execution paths. Our experiment shows that with appropriate thresholds, the technique can correctly detect that over 99.999996% of the inputs of all the programs we studied are stable while a traditional technique relying solely on inflation detection mistakenly classifies majority of the inputs as unstable for some of the programs. Compared to the state of the art technique that is based on high precision computation and causes several hundred times slowdown, our technique only causes 7.91 times slowdown on average and can report all the true unstable executions with the appropriate thresholds.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.2.5 [*Software Engineering*]: Testing and Debugging; G.1.0 [*Numerical Analysis*]: General

Keywords floating point representation; floating point errors; instability; continuity; discrete factors; sampling

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509526

1. Introduction

The machine representation of floating point values has precision limitations. When a value cannot be precisely represented, an error is implicitly introduced. For instance, when reading an input number of 0.9997 into a 32-bit single precision variable, the best representable value is $0.999700009822\cdots$. We hence have an initial error of $-0.000000009822\cdots$. When we add a very small value to a very large value, the small value may be too small to make a difference in the represented result, leading to an error. Such errors are propagated and accumulated, and eventually may lead to serious problems when they become comparable to the program values.

During the Gulf War in 1991, the patriot missile's failure to intercept an incoming missile, causing 28 casualties and around 100 injuries, was due to the loss of precision in computation. As revealed in the U.S. Government Accountability Office report [32], "*the range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. The conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation.*" Consequently the predicated position drifted away from the target.

Due to its importance, researchers have developed various techniques to address the problem. Traditionally, error analysis is conducted on mathematical models [36]. However, modern data processing uses more complex models and relies on computers and programs, rendering mathematical analysis difficult. Interval arithmetic [29, 31] and affine arithmetic [10, 11, 13, 15] are program analysis that model errors as ranges or affine formulas to reason about stability. Program transformation was proposed to improve precision and stability [1, 4]. Abstract interpretation and theo-

rem proving techniques [9, 16, 27] were developed to reason about stability statically.

Most existing techniques treat instability as bugs and focus on identifying buggy statements so that developers can fix them. We observe that the input range in which a floating point program is unstable is usually a very very small portion of the input domain. In other words, even a naive implementation may work fine in most cases. While this explains why many people are willing to stay with their unstable implementations, it also suggests that using high precision computation [4] or crafting a more stable implementation [35] may not pay off. With a traditional view of debugging, one may argue that we should nonetheless fix an instability bug despite its low probability to occur. However, different from traditional functional bugs, we observe that instability bugs are fundamentally inevitable with the limited precision of the machine representation. Using high precision computation or more stable implementation could *mitigate* them, but unlikely *fixes* them completely (we will further elaborate this observation in Section 2). Fortunately, compared to functional bugs, instability bugs are predictable. Evidence can be collected during a floating point program execution to predict if the execution is stable.

Moreover, many existing techniques are too expensive to be used on-the-fly for real world tasks. For example, the state of the art technique using high precision [4] causes 167–1016 times slowdown (i.e. the time-to-complete increased by 167–1016 times).

Hence, we argue that to tackle instability problems, instead of following the traditional way of finding and fixing bugs, we shall develop efficient prediction technique that runs together with the original program. Upon detecting a potentially unstable execution, the execution should be automatically restarted with a higher precision. This approach avoids paying the substantial overhead of high precision computation for most inputs, and saves the human efforts in developing more stable implementation, which may not be feasible in many cases.

In this paper, we develop a stability predictor that is practically conservative, sufficiently precise, and much more cost-effective compared to using high precision libraries (HPL) [4]. The technique does not explicitly compute errors as doing so incurs high overhead. Instead, it detects possible places where an error becomes substantially inflated regarding the corresponding value, and then tags the value with single bit to denote that it has an inflated error. It then tracks inflation bit propagation, taking care of operations that may cut off such propagation. It reports instability if any inflation bit reaches a critical execution point, such as a predicate, where the inflated error may induce substantial execution difference, such as following different execution paths.

Our contributions are highlighted as follows.

(a) original form

```
1  float x, z;
2  x = input();
3  z = x*x*x*x – 4*x*x*x
       + 6*x*x – 4*x + 1;
4  if (z > 0.5)
5    printf("hit");
6  else
7    printf("miss");
```

(b) alternative type

```
double x, z;
x = input();
z = x*x*x*x – 4*x*x*x
    + 6*x*x – 4*x + 1;
if (z > 0.5)
  printf("hit");
else
  printf("miss");
```

(c) alternative form

```
double x, z;
x = input();
z = (x – 1) * (x – 1)
   * (x – 1) * (x – 1) ;
if (z > 0.5)
  printf("hit");
else
  printf("miss");
```

Figure 1. Various code snippets that compute $z = (x-1)^4$.

- We analyze the characteristics of floating point instability bugs and disclose their differences from functional bugs. Such differences serve as the basis of our work.
- We propose a novel online prediction technique. The technique approximates errors with single bits to allow efficient representation and propagation. Its low cost compared to existing techniques allows us to screen out stable executions, which are the most common cases, while its conservativeness allows us to still capture all the true instability problems in practice.
- We study the completeness and the soundness of the technique.
- We have overcome a number of challenges when applying the technique to real-world programs, including handling specific floating point programming patterns that could introduce a lot of false warnings, e.g. convergence tests in iterative methods.
- Our evaluation shows that the predictor is very effective. With threshold τ_c=48[1], it can correctly determine over 99.999996% of the input space is stable with 691% overhead on average, which is 14 times cheaper than a high-precision-always system such as [4]. Even with a more conservative threshold (τ_c=36), it can still correctly detect over 99.9997% of the input space is stable. With τ_c=48, our technique reports instability for 7.5-93 times more inputs compared to the ground truth, due to its conservativeness, which however only counts as 1.5E-11% - 3.3E-6% of the input domain.

2. Motivation

Consider the example in Fig.1(a). Mathematically, the code snippet computes $z = (x-1)^4$ (in its expanded form) and makes decision according to the result at line 4. Assume that the input value is $x = 1.84089642$. In the ideal world, it produces $z = 0.5000000112886329660976$, and thus prints "`hit`" at the end. However, when we execute it on a 32-bit x86 machine, we get "`miss`" instead as some of the intermediate values cannot be precisely represented. The representation errors get propagated and enlarged, and eventually falsify the branch outcome at line 4. A plausible solution is to

[1] The meaning of the threshold will be defined in Section 5.

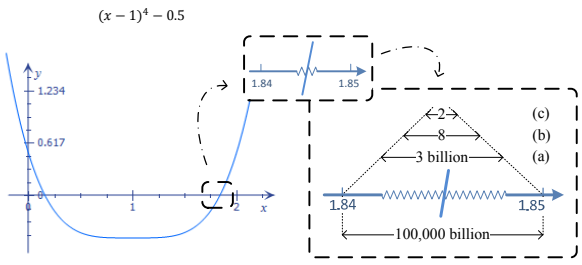

Figure 2. Problematic ranges for versions in Fig.1.

use data types with higher precision as in Fig.1 (b), in which the previous single-precision variables are refined with the double-precision type. The modified program produces the expected result with the original input. However, it produces wrong outputs with two other slightly different inputs $x = 1.8408964152537\mathbf{146}$ and $x = 1.8408964152537\mathbf{148}$. In fact, such unstable cases can always be found for any finite precision.

Another solution is to devise a numerically stable version of the underlying algorithm, as in Fig.1 (c). It produces correct output for some inputs that induce wrong outputs in Fig.1 (a) and (b). For instance, it produces the correct result for $x = 1.8408964152537\mathbf{148}$. However, it remains problematic for some other inputs including $x = 1.8408964152537\mathbf{146}$.

To further study the effect of these improvements on precision, we execute the three versions with a large number of input samples that are evenly distributed within the range from 0.0 to 2.0, with the interval of 1E-16. A sample run is considered problematic if its output differs from the ideal one that is emulated using the *high precision library* (HPL) [4]. We then count the number of problematic cases. These numbers represent the precision of the corresponding versions. Fig.2 shows the results, with wave-lines representing the problematic cases. Observe that the majority of the problematic runs occur around the intersections of the curve $y = (x-1)^4 - 0.5$, which is the mathematical function of the predicate at line 4, and the x-axis. We zoom-in one of the two intersections, namely $[1.84, 1.85]$. Out of the 100,000 billion samples in this range, there are about 3 billion problematic samples for the single-precision version in Fig.1 (a). The number gets down to 8 for the double-precision version in Fig.1 (b). Two problematic cases are still observed with the stable approach in Fig.1(c).

Observations. Based on the previous discussion, we summarize the characteristics of the instability problem as follows.

- The instability problem cannot be completely evaded by using more precise types or different implementations. Using more precise types implies higher overhead. Developing a more stable implementation requires in-depth understanding of the program and a lot of human efforts. In many cases, such more stable implementation may not be easily achievable.
- A floating point program only suffers from the instability problem for a very small input range. Consider Fig.2. Even the implementation with the lowest precision, i.e. case (a), only causes problems with the likelihood of 0.003% within the small input sub-range $[1.84, 1.85]$. It works properly for most inputs. This partially explains why there are so many unstable programs being used in reality. For example, we observe that four of the SPEC programs we analyzed are unstable. Detailed results are presented in Section 8.2.
- For a deterministic functional bug, a program must fail given the same failure inducing input. In contrast, given a particular input that leads to instability in a floating point program, we can evade the problem *for the particular input* using more precise data types and operations, which can be done automatically.
- Instability problems are predictable. For example, a very straightforward sign of such problems is that some of the internal computations have a value very close to zero, such as the subtraction entailed by the comparison at line 4 in Fig. 1.

Solution Overview. Hence, instead of testing/analyzing a program exhaustively to expose potential instability problems caused by representation errors and fixing them like fixing functional bugs, as advocated in existing work [4, 7, 10, 35], *we propose to develop an efficient runtime detection technique to predict on-the-fly if an execution is stable in the presence of representation errors. If not, the user could choose to restart the program execution with the high precision support.* It avoids using high precision all the time, which is very expensive, leveraging the observation that low precision suffices most of the time.

The basic idea is to monitor program execution and detect instruction executions that lead to substantial growth of *relative error*, i.e. the ratio between an error and its corresponding value. Such detection is performed without explicitly computing errors because doing so is very expensive. Instead, it is approximated by observing the operand and result values of an operation, especially their exponents. Intuitively, if the operands have large exponents but the result has a small one and the differences exceed a threshold, we consider the relative error has encountered an *inflation*. The subtraction at line 4 in Fig. 1 is an example for such operations. In practice, these operations occur quite often while only very few of them really cause problems. In most cases, the inflated relative errors are suppressed/masked during execution such that they cannot cause any problems. For example, if a small value with an inflated relative error is added to a large value, the relative error is suppressed, because the relative error of the result returns to a very low level. Hence,

```
1.  float x,y,z;
2.  x=input ();
3.  y=f(x);
4.  if (y>1.0)
5.     z=x+1.0;
    else
6.     z=x-1.0;
7.  output (z);
```
(a)

```
1.  float x,z;
2.  float A[10]={1.0, 2.0, ...};
3.  int i;
4.  x=input ();
5.  i=(int)f(x);
6.  z=A[i]*x;
7.  output (z);
```
(b)

Figure 3. Discrete difference examples.

our technique not only detects error inflation, but also tracks propagation of inflated relative errors and checks if they can reach critical execution points, e.g. predicates, without being suppressed. Note that when an inflated error reaches a predicate, it may induce a different execution path, leading to undesirable outcome, as illustrated in the example.

If our technique detects that an inflated relative error can reach any critical execution point, the execution is flagged as unstable. The technique provides the capability of automatically switching to executing a high precision version of the program, which is automatically generated at compile time.

3. Problem Definition

Our goal is to develop a cost-effective technique to determine if an execution is stable in the presence of representation errors. According to the previous discussion, representation errors are inevitable due to the limited precision in computation. However, in most cases, they are not substantial enough to cause any problems. Therefore, we first need to define the criterion when such errors become un-acceptable. In existing work [4, 10], this is usually determined by observing final output errors. However, it remains difficult to determine how much output difference should be considered un-acceptable.

In this paper, we define *an execution is unstable if the actual execution (with limited precision) and the ideal execution (with infinite precision) have discrete differences*.

Discrete differences are differences of discrete types, such as int and bool. Sample discrete differences include control flow differences and array index differences. An execution is unstable if its control flow in the actual world is different from that in the ideal world; or an array index generated by a type cast of a floating point value has different values in the two worlds. The intuition is that if we consider the output of an execution as a mathematical function over inputs, the function becomes discontinuous, or has different continuous forms in the input space, due to discrete differences. Consider the examples in Fig. 3. In (a), if y is very close to 1.0, the representation error may change the branch outcome so that the mathematical form of the output could be either $z = x + 1.0$ or $z = x - 1.0$. In (b), if $f(x)$ is very close to 1.0, the representation error may cause $i = 0$ or $i = 1$. Consequently, the mathematical form of the output could be either $z = 1.0 * x$ or $z = 2.0 * x$.

Besides places that may have discrete differences, there are some mathematical functions that are intrinsically discontinuous so that small input changes could also cause arbitrarily large output changes. For instance, $f = tan(x)$ is discontinuous at $x = -\frac{\pi}{2}, \frac{\pi}{2}, \cdots$. When x is around those values, representation errors could cause substantial output differences. In practice, we observe that programmers usually insert predicates to guard these functions against discontinuous inputs, to avoid uncontrollable output variations. As a result, the discontinuity manifests itself as discrete difference of the predicate. Hence, our discussion will focus on discrete differences.

The problem statement, however, implies expensive detection techniques as it requires detecting any changes in the discrete domain of a program execution. Next we introduce the concept of discrete factors so that we can simplify the problem statement to make it more tractable.

Background: Discrete Factors. Discrete factors are introduced in [2] to model output discontinuity caused by external input uncertainty. *A discrete factor is an operation that has floating point values* [2] *as operands and produces a discrete value as result.* Discrete factors are the interface between the continuous and the discrete domains. Since representation errors originate from the continuous domain, they can only cause discrete differences by inducing different discrete values at discrete factors. As a result, we only need to monitor execution of discrete factors instead of all program artifacts with discrete types.

Control flow predicates that are relational operations of floating point values are the dominant type of discrete factor. Other discrete factor examples include *type casts* that cast a floating point value to an integer, and discrete mathematical library functions, e.g. `sign()`. Most discrete factors can be determined by the compiler statically. □

Therefore, we have the following refined problem definition.

We consider an execution unstable if the errors of the floating point operands at any discrete factor are large enough to induce a different discrete value.

As such, we only need to detect discrete differences at discrete factors.

4. Floating Point Representation and Errors

Representation precision limitations are the root cause of the instability problem.

According to the standard of the 32-bit single precision floating point format representation defined in IEEE 754 [20], a decimal value f is represented as follows.

$$f = (1 - 2s) \times (1 + m \times 2^{-23}) \times 2^{e-127}$$

Variable s is the sign bit (zero or one), m is the significand, which is also called mantissa, and e is the exponent. Fig. 4 shows this representation.

[2] To simplify discussion, we assume floating point a continuous type equivalent to real type.

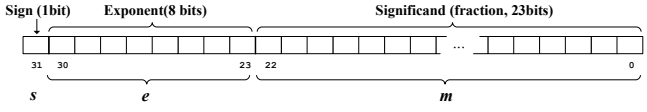

Figure 4. Float Point Representation.

x, y, z:	32-bit single precision floating point program variables so that their precision is limited by the format specification.
$\hat{x}$:	the precise value of x (with infinite precision).
$\hat{\Delta}_x$:	the error of x (with infinite precision).
Δ_x:	the relative error of x.

Table 1. Definitions.

Observe that there are 8 exponent bits denoting the exponent range [-127,128]. There are altogether 24 mantissa bits, including a hidden leading bit of "1", so that any values that require more significand bits to represent cannot be precisely represented.

During execution of a floating point program, errors are generated and propagated as follows. Floating point constants in the source code may not be precisely represented such that initial errors are introduced at compile time. At runtime, when floating point values are loaded from files or the standard input, they are usually converted from strings so that representation errors may also be introduced. As shown in Table 1, we denote the value of a variable x in the ideal world (with infinite precision) as $\hat{x}$. The symbol x also denotes the value in the actual world (with error). Hence, we have $\hat{x} = x + \hat{\Delta}_x$.

The initial errors are further propagated to the internal of program execution through operations. In particular, the error of the result of an operation includes those inherited from the operands, and the representation error of the result value itself.

For example, assume two variables x and y with errors.

$$\hat{z} = \hat{x} + \hat{y} = (x + \hat{\Delta}_x) + (y + \hat{\Delta}_y) = (x + y) + (\hat{\Delta}_x + \hat{\Delta}_y) \quad (1)$$

The sub-expression $x + y$ denotes the addition in the actual world and $\hat{\Delta}_x + \hat{\Delta}_y$ denotes the inheritance of the operand errors. Note that it is possible that the results of the two sub-expressions themselves cannot be precisely represented so that further representation errors are introduced.

Explicitly monitoring errors requires representing and computing errors that are usually in a very small scale, which is expensive. Therefore, we introduce the notion of *relative error*.

Definition 1. *The relative error of a variable x, denoted by* Δ_x*, is computed as* $|\hat{\Delta}_x / x|$.

Given a discrete factor $x > y$, if the relative error of $x - y$ is larger than 1.0, the factor may have different discrete outcomes in the actual world and the ideal world. This is because the error of $x - y$ is larger than $x - y$ itself and may have a different sign. As such, the relational expression $(x - y > 0)$ has a different boolean outcome. For a discrete factor $i = (int)x$, if the relative error of $x - (float)i$ has a larger than 1.0 relative error, the factor may have different discrete outcomes as well.

One can easily infer from the floating point format representation (Fig. 4) that an initial relative error, i.e. the relative error of a constant or an input v, is bounded by $1/2^{(\#\ of\ mantissa\ bits\ of\ v)-1}$ because the error is bounded by the value represented by the least mantissa bit. As we will show later, floating point operations except subtractions (or additions of operands with opposite signs) lead to no growth or very slow growth of relative errors. It is unlikely that they can grow so large (e.g. larger than 1) over time to induce discrete differences. Instead, most unstable executions have involved sudden inflation of relative errors caused by subtractions (or additions of operands with opposite signs).

According to the IEEE 754 standard, the result of an subtraction/addition needs to be normalized by left-shifting, to remove the leading zeros after the operation. The relative error inherited from the operands thus get inflated during this process. Assume a subtraction operation $x - y$. After the operation, the significand bits are left-shifted by d bits, the relative error inherited from the operands, $(\hat{\Delta}_x - \hat{\Delta}_y)/(x - y)$ may very likely become 2^d times larger than the relative errors of x and y, because $x - y$ is 2^d times smaller than x or y, suggested by the left shifting. The d left-shifted bits are also called the *cancelled bits* [25]. Note that cancelled bits can be cost-effectively monitored by comparing the exponents of the computation result and the larger operand, particularly, $d = max(e_x, e_y) - e_{x-y}$, with e_x the exponent of x. An example can be found in Section 5.

Therefore, our technique is based on detecting inflation of relative errors by monitoring cancelled bits of each operation.

5. Propagation of Relative Errors

Detecting relative error inflation alone is not sufficient. A simple approach that detects occurrences of cancellation [25] reports instability for any execution of a few SPEC CFP programs according to our study. This is because cancellation is common even in stable executions. We observe that a lot of inflated errors are not problematic because they are suppressed by other operations in the following execution. For example, in `183.equake`, a lot of cancellations originate from an expression *c+=a-b* inside a loop when a and b are very close. However, the errors from *a-b* are always suppressed by the large value of c, i.e. the inflated relative error by *a-b* becomes trivially small after the addition with the large c. Hence, a key step in our technique is to monitor propagation of inflated errors to determine if they can be propagated to a discrete factor and cause discrete differences, without being suppressed.

Due to the efficiency concern, we cannot afford computing the true relative errors and monitoring their changes during execution. Instead, we develop a cost-effective algorithm to abstract the process. In particular, we tag a value with an inflation bit when we observe the relative error of the value is inflated by cancellation. We develop a set of rules to propagate the inflation bits. These rules respect the semantics regarding relative errors such that during an operation, a result value inherits the bit from its operand(s) if and only if the operation does not suppress the error. In this case, the relative error of the result is comparable to that of the operand.

$$
\begin{array}{lll}
\textit{Program } P & ::= & s \\
\textit{Stmt } s & ::= & s_1; s_2 \mid \textbf{skip} \mid x := e \mid x = \textbf{f2i}(y) \mid x = y \bowtie 0 \mid \\
 & & \textbf{while } x \textbf{ do } s \mid \textbf{if } x \textbf{ then } s_1 \textbf{ else } s_2 \mid \textbf{fail} \\
\textit{Expr } e & ::= & x \mid v \mid e_1 \; op \; e_2 \mid \textbf{sin}(e) \\
\textit{BinOp } op & ::= & + \mid - \mid * \mid / \\
\textit{Value } v & ::= & n \mid r \mid b \\
\multicolumn{3}{c}{\textit{Var } x \in \textit{Identifier} \quad n \in Z \quad r \in \textit{Real} \quad b \in \textit{Boolean}}
\end{array}
$$

Figure 5. Language

Language. To facilitate formal discussion, we introduce a kernel language. The syntax is presented in Fig. 5. We model three kinds of discrete factors: the type cast from a floating point value to an integer **f2i**, relational operations that are normalized to $y \bowtie 0$, with $\bowtie$ denoting a relational operator, and discrete mathematical library functions. We normalize relational operations to study the real values (not the boolean values) involved in these expressions. Such values are explicitly denoted by y after normalization. For instance, a conditional statement "**if** $t < 1.0$ ..." is normalized to "$y := t - 1.0$; $x = y < 0$; **if** x". It allows us to explicitly reason about the value of y and its error.

We model three kinds of values, i.e. integer, real, and boolean, and the commonly used binary operators. We model one mathematical function **sin**() to demonstrate how we handle library functions.

Operational Semantics. The semantics is presented in Fig. 6. The expression rules have the form of $\boxed{\sigma, \Gamma : e \xrightarrow{e} e'}$, with store σ and error store Γ. The error store indicates if a variable is holding a value with an inflated relative error, that is, the ratio between the actual error and the value is large. The resulting value of an evaluation is tagged with the inflation bit. The evaluation of a variable yields the value from the store, tagged with the bit from the error store. The evaluation of a value yields the value itself tagged with F, meaning its relative error is small.

The result of the addition of two values tagged with T (i.e., with inflated errors) is also tagged with T. Intuitively, if both operands have significant errors, the resulting error of the addition is also significant (Rule [**ADD-TT**]). According to Rule [**ADD-TF**], if one operand v_1 is tagged with T and the other v_2 tagged with F, the exponents of the two operands are compared. If the exponent of v_1, denoted as e_{v_1} is much smaller than e_{v_2}, particularly when their difference

$$
\begin{array}{l}
E ::= E; s \mid [\cdot]_s \mid x := [\cdot]_e \mid \textbf{if } [\cdot]_e \textbf{ then } s_1 \textbf{ else } s_2 \mid [\cdot]_e \; op \; e \mid \\
\quad v \; op \; [\cdot]_e \mid x := \textbf{f2i}([\cdot]_e) \mid x := [\cdot]_e \bowtie 0 \mid \textbf{sin}([\cdot]_e)
\end{array}
$$

DEFINITION:
Store $\sigma : Var \to Value$ *ErrStore* $\Gamma : Var \to Boolean$
e_x, s_x: the exponent, and the sign of x respectively.
v^b: b indicates if a value v carries an inflated relative error.

EXPRESSION RULES $\boxed{\sigma, \Gamma : e \xrightarrow{e} e'}$

$$
\begin{array}{lll}
\sigma, \Gamma : x \xrightarrow{e} \sigma(x)^{\Gamma(x)} \qquad \sigma, \Gamma : v \xrightarrow{e} v^F & & \\
\sigma, \Gamma : v_1^T + v_2^T \xrightarrow{e} (v_1 + v_2)^T & & [\textbf{ADD} - \textbf{TT}] \\
\sigma, \Gamma : v_1^T + v_2^F \xrightarrow{e} (v_1 + v_2)^b & \text{where } b = \neg(e_{v_2} - e_{v_1} > \tau_s) & [\textbf{ADD} - \textbf{TF}] \\
\sigma, \Gamma : v_1^F + v_2^T \xrightarrow{e} (v_1 + v_2)^b & \text{where } b = \neg(e_{v_1} - e_{v_2} > \tau_s) & [\textbf{ADD} - \textbf{FT}] \\
\sigma, \Gamma : v_1^F + v_2^F \xrightarrow{e} (v_1 + v_2)^b & \text{where } b = (max(e_{v_1}, e_{v_2}) - e_{v_1+v_2} > \tau_c) & \\
 & & [\textbf{ADD} - \textbf{FF}] \\
\sigma, \Gamma : v_1^{b_1} * v_2^{b_2} \xrightarrow{e} (v_1 * v_2)^{b_3} & \text{where } b_3 = b_1 | b_2 & [\textbf{MULTI}] \\
\sigma, \Gamma : \textbf{sin}(v^b) \xrightarrow{e} v_1^{b_1} & \text{where } v_1 = \textbf{sin}(v), \text{ and } & \\
 & b_1 = \begin{cases} T & |\textbf{cos}(v) \cdot v \,/ \textbf{sin}(v)| > 2^{\tau_c} \\ F & |\textbf{cos}(v) \cdot v \,/ \textbf{sin}(v)| < 2^{-\tau_s} \\ b & \textit{otherwise} \end{cases} & [\textbf{SIN}]
\end{array}
$$

STATEMENT RULES $\boxed{\sigma, \Gamma : s \xrightarrow{s} \sigma', \Gamma', s'}$

$$
\begin{array}{lll}
\sigma, \Gamma : x := v^b \xrightarrow{s} \sigma[x \mapsto v], \Gamma[x \mapsto b], \textbf{skip} & \sigma, \Gamma : \textbf{skip}; s \xrightarrow{s} \sigma, \Gamma, s & \\
\sigma, \Gamma : x := \textbf{f2i}(v^T) \xrightarrow{s} \sigma, \Gamma, \textbf{fail} & & \\
\sigma, \Gamma : x := \textbf{f2i}(v^F) \xrightarrow{s} \sigma, \Gamma, \textbf{fail} & \text{if } (e_v - e_{v-(int)v} > \tau_c) & \\
\sigma, \Gamma : x := \textbf{f2i}(v^F) \xrightarrow{s} \sigma[x \mapsto (int)v], \Gamma[x \mapsto F], \textbf{skip} & \text{otherwise} & \\
\sigma, \Gamma : x = v^T \bowtie 0 \xrightarrow{s} \sigma, \Gamma, \textbf{fail} & & \\
\sigma, \Gamma : x = v^F \bowtie 0 \xrightarrow{s} \sigma[x \mapsto v \bowtie 0], \Gamma[x \mapsto F], \textbf{skip} & & \\
\sigma, \Gamma : \textbf{if } b^T \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, \Gamma, \textbf{fail} & & \\
\sigma, \Gamma : \textbf{if } T^F \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, \Gamma, s_1 & & \\
\sigma, \Gamma : \textbf{if } F^F \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, \Gamma, s_2 & & \\
\sigma, \Gamma : \textbf{while } x \textbf{ do } s \xrightarrow{s} \sigma, \Gamma, \textbf{if } x \textbf{ then } s; \textbf{while } x \textbf{ do } s \textbf{ else skip} & &
\end{array}
$$

GLOBAL RULES $\boxed{\sigma, \Gamma, s \to \sigma', \Gamma', s'}$

$$
\frac{\sigma, \Gamma : e \xrightarrow{e} e'}{\sigma, \Gamma, E[e]_e \to \sigma, \Gamma, E[e']_e} \text{ [G-EXPR]} \qquad \frac{\sigma, \Gamma : s \xrightarrow{s} \sigma', \Gamma', s'}{\sigma, \Gamma \, E[s]_s \to \sigma', \Gamma', E[s']_s} \text{ [G-STMT]}
$$

Figure 6. Operational Semantics

is larger than a pre-defined threshold τ_s, the tag of the result value is set to F. Intuitively, this corresponds to when a value with an inflated error is added to a much larger value with a trivial error, the inflated error is suppressed. Rule [**ADD-FT**] is similar.

If both operands are tagged with F, we test if the addition causes any cancelled bits. If the number of cancelled bits is larger than a pre-defined threshold τ_c, we consider there is an inflation of the relative error associated with the result value and tag the result with T. In later sections, we will study the soundness and completeness of the semantic rules and the effect of threshold selection. Subtractions are handled in a way similar to additions and thus their rules are omitted.

For multiplications, the tag of the result is the disjunction of the tags of the operands (Rule **[MULTI]**). Intuitively, multiplying a value v_1 with a significant error with another value v_2 enlarges both the value and its error with the same

	statements	Execution 1							Execution 2						
		op1(x, Δ_x)		op2(x, Δ_x)		result(x, Δ_x)		Rule	op1(x, Δ_x)		op2(x, Δ_x)		result(x, Δ_x)		Rule
	inv_J (a): {														
	...														
1	det=a[0]*c[0];	0.00	inf	−1.90	3E-17	−0.00	inf	**MULTI**	2.92	0	7.58	4E-17	22.11	6E-17	**MULTI**
2	t1=a[1]*c[1];	0.00	inf	0.00	inf	0.00	inf	**MULTI**	2.92	0	−7.58	4E-17	−22.11	5E-17	**MULTI**
3	det=det+t1;	−0.00	inf	0.00	inf	0.00	inf	**ADD-TT**	22.11	6E-17	−22.11	5E-17	-1.4E-14	0.013	**ADD-FF**
4	t2=a[2]*c[2];	1.46	0	1.90	2E-17	2.76	7E-17	**MULTI**	2.92	0	0.00	inf	0.00	inf	**MULTI**
5	det=det+t2;	0.00	inf	2.76	7E-17	2.76	7E-17	**ADD-TF**	-1.4E-14	0.013	0.00	inf	-1.4E-14	0.013	**ADD-TT**
6	...														
7	vol=det/6;	2.76	7E-17	6.00	0	0.46	7E-17	**MULTI**	-1.4E-14	0.013	6.00	0	-2.4E-15	0.013	**MULTI**
8	b1=(vol⩽0);	0.46	7E-17	0.00		*F*			-2.4E-15	0.013	0.00		*F*		
9	if(b1) report();	*F*							*F*				**fail**		
	}														

Table 2. Two partial execution traces from `183.equake`. Relative errors are detected and propagated according to the rules in Fig. 6. Shaded cell indicates that the variable carries an inflated relative error. In Execution 1, relative errors get suppressed at line 5; while in Execution 2 inflated errors are detected at line 3 then flow to the discrete factor at line 9.

factor and hence the relative error remains the same. Divisions are similarly handled.

For the library function **sin**(), since the function behaves differently with different inputs, i.e. sometimes small input errors induce large output changes whereas in other cases, large input errors induce trivial output differences, we have to tag the result differently based on both the operand tag and the operand value.

$$\Delta_{\mathbf{sin}(x)} = \frac{\hat{\Delta}_{\mathbf{sin}(x)}}{\mathbf{sin}(x)} = \frac{\frac{d\ \mathbf{sin}(x)}{d\ x} \times \hat{\Delta}_x}{\mathbf{sin}(x)} = \frac{\mathbf{cos}(x) \times (\Delta_x \times x)}{\mathbf{sin}(x)}$$

From the above equation, we can observe that the inflation of the relative error is $\Delta_{\mathbf{sin}(x)}/\Delta_x = \mathbf{cos}(x) \cdot x/\mathbf{sin}(x)$. According to Rule [**SIN**], if the inflation factor is larger than 2^{τ_c}, which corresponds to having more than τ_c cancelled bits, the tag is set to T. Observe that in some cases, the inflation factor can be smaller than 1 and close to 0, in such a case, it suppresses the operand error instead. Therefore, if the factor is smaller than $2^{-\tau_s}$, with τ_s the threshold for suppression in Rules [**ADD-TF/FT**], the tag is reset to F. For other cases, the operand tag is propagated to the result. We handle other mathematical library functions in a similar fashion. Note that the input ranges that cause the different behavior can be pre-computed based on the thresholds so that we simply determine if the operand value falls in these ranges at runtime, without performing the expensive trigonometric function evaluations.

For an assignment statement, the value is saved in the store and the corresponding tag is saved in the error store. If the assignment is a type cast, depending on the tag and the value, the statement is evaluated differently. More specifically, if the floating point operand is tagged with T, meaning that the relative error is significant, discrete differences may be induced. Hence the execution is considered unstable. If the operand is tagged with F, but the value is very close to the boundary of a discrete value, which can be detected by observing the cancelled bits of (v-(int)v), discrete differences may be induced and the execution is considered unstable. If the right-hand-side (RHS) of an assignment is a relational operation $v \bowtie 0$, the execution is considered unstable if v is tagged with T. The predicate in a conditional statement is similarly handled.

The global rules are standard.

Example. Table 2 shows the propagation of relative errors in part of `183.equake`. The code snippet computes the determinant *det* for a given Jacobian matrix in a. Two separate executions are presented. In Execution 1, some of the values in array a have been tagged with T upon entering the function, denoted as shaded cells. For example, $a[0]$ and $a[1]$ at lines 1 and 2, respectively, hold inflated relative errors from previous computation. Since their values are very small, the relative errors $\Delta_{a[i]}$ are very large, inf in both cases (with double precision). As a result, the computation results at lines 1, 2, and 3 also carry inflated errors, and thus are tagged with T as well according to Rules [**MULTI**] and [**ADD-TT**]. The errors get suppressed at line 5 because of Rule [**ADD-TF**], where the operand *det* tagged with T is added to a large value t_2 tagged with F.

In Execution 2, initially values in array a are all tagged with F. Cancelled bits are detected during the computation at line 3 by Rule [**ADD-FF**]. The two operands and their relative errors are as follows[3].

$det = 22.1090526719999829...$, with $\hat{\Delta}_{det} = 1.3E-15$.
$t_1 = -22.109052671999997...$, with $\hat{\Delta}_{t_1} = -1.1E-15$.

When performing $det + t_1$, there are 50 cancelled bits. The result is $det = -0.0000000000000142...$, with $\hat{\Delta}_{det} = 1.8E-16$ and $\Delta_{det} = \hat{\Delta}_{det}/det \approx 0.013$, which is roughly 2^{50} times of the relative errors of the operands. Note that $\Delta_{det} * 2^{50} \approx 0.068$ and $\Delta_{t_1} * 2^{50} \approx 0.056$.

The inflated error is propagated afterwards, which eventually flows into the discrete factor at line 9. The execution is hence considered unstable.

6. Soundness and Completeness

In Section 3, we define the problem as detecting if errors can induce discrete differences at discrete factors. In this section, we discuss the soundness and completeness of our

[3] Note that our technique does not compute actual errors or relative errors. Here we present error values just for the illustration purpose.

method regarding the problem definition. Recall that we use inflation bits to approximate significant relative errors. As a result of the approximation, the proposed technique is neither sound nor complete. Next, we discuss the conditions that affect completeness and soundness. Note that we will show in Section 8 that with appropriate threshold settings, the technique does not miss detecting any unstable executions in practice, and the input subranges in which instability is detected are trivial compared to the whole input ranges.

The discussion consists of two parts. The first one discusses the essence of cancelled bits and the second part discusses the propagation rules.

6.1 The Essence of Cancelled Bits

Cancelled bits have been used as an indicator for instability in existing works [4, 25], in which executions are considered unstable if there is any operation causing a large number of cancelled bits. However, this causes a lot of false warnings because a lot of inflated errors are suppressed and thus do not cause any problems. Hence in our approach, we only use them to detect inflation of relative errors, and we further track the propagation and suppression of inflated errors. In the following, we discuss a few issues critical to our method.

I1: Cancelled Bits May not Mean Relative Error Inflation. Consider a subtraction $x-y$. The relative error of the result is $|(\hat{\Delta}_x-\hat{\Delta}_y)/(x-y)|$. The occurrence of d cancelled bits during the subtraction means that $x-y$ is around 2^d times smaller than the maximum of x and y. If $|\hat{\Delta}_x-\hat{\Delta}_y|$ has an exponent close to that of $\hat{\Delta}_x$ or $\hat{\Delta}_y$, the result's relative error is around 2^d times larger than that of the operand. However, observe that if $\hat{\Delta}_x-\hat{\Delta}_y$ is close to 2^d times smaller than $\hat{\Delta}_x$ and $\hat{\Delta}_y$, which is similar to having close to d cancelled bits if $\hat{\Delta}_x-\hat{\Delta}_y$ were performed with finite precision, the inflation does not happen. In practice, the event of $\hat{\Delta}_x-\hat{\Delta}_y$ being 2^d times smaller is independent of the event of $x-y$ having d cancelled bits, as $\hat{\Delta}_x$ and $\hat{\Delta}_y$ are mainly caused by precision limitations. It is hence unlikely these two events happen simultaneously.

However, we do observe this becomes problematic in some rare cases, particularly when x and y are semantically equivalent. It means x and y are computed from the same inputs, through the same/equivalent sequences of operations, even though they may be computed separately. In this case, $x=y$ and $\hat{\Delta}_x=\hat{\Delta}_y$. Our technique will detect a large number of cancelled bits caused by $x-y$. However, there is no relative error inflation.

Fig. 7 shows one example from `187.facerec`. It scans a large image to locate the part that is the most similar to a small image. The best fit position with the highest similarity is recorded in *pos*. The algorithm has two rounds. In the first round (lines 1-6) it iterates through each position at a given stride and identifies the best fit position. Then it performs the second scan (lines 9-14) around the best position found in the first round, but at a smaller stride.

```
    /* first scan with grid spacing given by Step */
    pos = (0, 0); similarity = 0.0;
1   for (LLY = StartY; LLY <= EndY; LLY += Step) {
2       for (LLX = StartX; LLX <= EndX; LLX += Step) {
3           current = GraphSimFct (LLX, LLY, ...);
4           if (current > similarity) {
5               similarity = current; pos = (LLX, LLY);
6   }   }   }

    /* second scan around best position found previously */
7   StartY = pos(2)-((Step+1)/2); EndY = ...;
8   StartX = pos(1)-((Step+1)/2); EndX = ...;

9   for (LLY = StartY; LLY <= EndY; LLY += Scale) {
10      for (LLX = StartX; LLX <= EndX; LLX += Scale) {
11          current = GraphSimFct (LLX, LLY, ...);
12          if (current > similarity) {
13              similarity = current; pos = (LLX, LLY);
14  }   }   }
```

Figure 7. Pseudocode snippet from `187.facerec`.

During this process, it is likely that variables *current* and *similarity* have exactly the same value (and the same error) in the predicate at line 12, when *pos* identified in the first round coincides with *LLX* and *LLY* at line 11. In this case, *current* and *similarity* are semantically equivalent, as they are from the same inputs and go through the same sequence of operations. The cancellation does not lead to any relative error inflation.

Note that we cannot simply preclude all addition/subtraction operations that yield 0 because if x and y are not semantically equivalent, $x-y=0$ means large inflation. In fact, this is the common case.

I2: Only Addition and Subtraction Can Cause Inflation. According to the operational semantics in Fig. 6, we only detect cancelled bits in addition and subtraction operations. It is hence important to show that other binary operations cannot cause inflation. In the following discussion, we assume two variables x and y with the same sign, and $\Delta_x=\Delta_y=\tau$ for simplicity. Note that $\Delta_x=|\hat{\Delta}_x/x|$.

For multiplication $x*y$, we have the following.

$$\hat{x}\cdot\hat{y}=(x+\hat{\Delta}_x)\cdot(y+\hat{\Delta}_y)=x\cdot y+x\cdot\hat{\Delta}_y+y\cdot\hat{\Delta}_x+\hat{\Delta}_x\cdot\hat{\Delta}_y$$

$$\Delta_{x\cdot y}=|\tfrac{\hat{x}\cdot\hat{y}-x\cdot y}{x\cdot y}|=|\tfrac{x\cdot\hat{\Delta}_y+y\cdot\hat{\Delta}_x+\hat{\Delta}_x\cdot\hat{\Delta}_y}{x\cdot y}|$$
$$\leqslant|\tfrac{x\cdot\hat{\Delta}_y}{x\cdot y}|+|\tfrac{y\cdot\hat{\Delta}_x}{x\cdot y}|+|\tfrac{\hat{\Delta}_x\cdot\hat{\Delta}_y}{x\cdot y}|=|\tfrac{\hat{\Delta}_y}{y}|+|\tfrac{\hat{\Delta}_x}{x}|+\boxed{|\tfrac{\hat{\Delta}_y}{y}\cdot\tfrac{\hat{\Delta}_x}{x}|}^{[1]}$$

Observe that normally the sub-expression [1] in the above formula is orders of magnitude smaller than the other two sub-expressions and hence can be ignored. Moreover, since $|\tfrac{\hat{\Delta}_y}{y}|=\Delta_y=\tau$ and $|\tfrac{\hat{\Delta}_x}{x}|=\Delta_y=\tau$, we have

$$\Delta_{x\cdot y}\leqslant\Delta_x+\Delta_y=2\tau$$

It shows that the growth of the relative error in multiplication is normally bounded by a factor of 2. In practice, the growth is usually much smaller than 2.

Consider division y/x, which can be rewritten as $y*(1/x)$.

$$\Delta_{1/x}=\frac{|\frac{1}{x+\hat{\Delta}_x}-\frac{1}{x}|}{|\frac{1}{x}|}=|\frac{x}{(x+\hat{\Delta}_x)}-1|=|\frac{\hat{\Delta}_x}{(x+\hat{\Delta}_x)}|\leqslant\frac{\tau}{1-\tau}$$

Note that since τ is a very small number, usually slightly larger than 0, the growth factor of the relative error is a number slightly larger than 1.

Consider addition (of two operands with the same sign).

$$\Delta_{x+y} = \frac{|(x+\hat{\Delta}_x+y+\hat{\Delta}_y)-(x+y)|}{|x+y|} = \frac{|\hat{\Delta}_x+\hat{\Delta}_y|}{|x+y|} \leq \tau$$

The relative error does not grow.

Therefore normally relative errors can only get inflated by subtractions (or additions of operands with different signs). Note that even in subtractions, if the two operands are not very close, inflation does not happen either.

I3: Unstable Executions Always Involve Inflation In Practice. An important question is whether inflation (i.e. evidenced by a large number of cancelled bits) is a necessary condition of unstable execution. In theory, it is not true as errors can gradually aggregate through operations and eventually become comparable to the corresponding values, leading to discrete differences. Such examples can be constructed. However, in the programs we have studied, we haven't encountered any case in which an unstable execution is caused by incremental growth of errors (i.e. without the presence of inflation)[4]. The reasons are as follows. The growth of errors is usually very slow without inflation based on our above discussion in **I2**. Moreover, error aggregation may enlarge or diminish errors.

6.2 Soundness and Completeness of Propagation Rules

The semantic rules in Fig. 6 are unsound due to the approximation. For instance, upon the addition of two operands tagged with T, the result value is always tagged with T (Rule [**ADD-TT**]). If the two operands are semantically equivalent except that they have different signs, their errors will be exactly the same but with different signs. Consequently, they cancel each other out. However, this cannot be detected by our approximation. Also, according to the rules for type cast and relational operations, we consider an execution unstable if a T tag can reach a discrete factor. However, a significant relative error (as indicated by T) does not necessarily induce any discrete difference. It also depends on the signs of the actual error and the value. For instance, assume a value v in the actual world is slightly larger than 0.0. A type cast to the integer domain gets 0. If $\hat{\Delta}_v$ has a positive sign, even though it is comparable to the value, it does not cause discrete difference between the actual world and the ideal world. Unfortunately, it is infeasible to track signs without computing explicit error values.

In practice, the completeness of our technique is largely affected by the conservativeness of the two thresholds τ_c and τ_s. Recall that τ_c decides the occurrence of inflation and τ_s determines if an inflated error is suppressed when a smaller operand with T tag is added to a much larger operand. The two thresholds are currently configured based on our experience. In Section 8, we study the effect of different threshold configurations empirically. Our experiment shows that when appropriate thresholds are set, our technique does not miss any unstable executions.

[4] We used the high precision library (HPL) to precisely compute the error for each value encountered during execution. For each discrete difference observed by the HPL method, we could always backtrack to an inducing inflation (with τ_c=48).

1	a=((1+2E54)-2E54)-1	1.1	$1^F + 2E\ 54^F = 2E\ 54^F$
2	x=a+0.5;	1.2	$2E\ 54^F - 2E\ 54^F = 0^T$
3	y=x>0;	1.3	$0^T - 1^F = -1^F$
4	...	2	$x = -1^F + 0.5^F = -0.5^F$
		3	$y = -0.5^F > 0 = F$ (stable)
	(a) program		**(b) execution in actual world**

Figure 8. An example for missing unstable execution. In the ideal world, $a = 0$, $x = 0.5$ and hence $y = T$, which is different from the actual world result in (b). However, our approximation cannot catch the instability problem. Symbol 1.2 means the second sub-step in statement 1.

However, since our technique only uses one bit to approximate error related information for a value, there are cases in which even conservative thresholds fail to catch unstable executions. Consider the example in Figure 8. According to Rule [**ADD-TF**], the propagation is cut off at step 1.3 when the smaller value 0 with inflation is added to the large value 1 without inflation. As a result, the execution is considered stable although y has a different boolean value from that in the ideal world. The root cause is that the single inflation bit is insufficient to model that the actual error is much larger than the value 0 at step 1.3. A scheme that tags a value with more bits and has more sophisticated propagation rules may mitigate this problem. We will leave it to our future work. In our empirical study, we haven't encountered such cases.

7. Handling Practical Challenges

A fundamental assumption of our technique is that an execution is unstable if errors can cause discrete differences. The intuition is that discrete differences leads to different forms of the output mathematical function such that output values have discontinuous differences. However, due to the flexibility of modern programming languages and the nature of certain computation, discrete differences may not always lead to discontinuous output differences.

7.1 Continuous Cores

Continuous core was defined in [2] as a program region (usually a conditional statement and its branches) in which control flow variations do not cause output discontinuity. In our context, we should not consider an execution unstable if errors cause (discrete) control flow differences inside continuous cores.

Fig. 9 (a) shows a continuous core code snippet from [2]. It returns the maximum value of an array. Note that while [2] focuses on studying the effect of continuous cores on external input uncertainty, we focus on their effect on execution

```
1  o = A[0]
2  for i := 1 to c
3    if o < A[i]
4      o:=A[i];
```
(a)

```
10 if (x < y)
11   o:=x/y
12 else
13   o:=y/x
```
(b)

Figure 9. Continuous core examples.

```
1  x=...;
2  repeat {
3    old=x;
4    x=F(x); /* F() is the iterative function*/
5    i++;
6    if (i>bound) exit(non-convergent);
7  } until (|x-old|< t);
```

Figure 10. Iterative algorithm example.

stability in the presence of internal representation errors. In the former case, input errors are usually large enough to be representable, sometimes comparable to the input values, whereas representation errors are introduced because they are not representable. In the example, assume an array $A[0-2] = \{1.0,\ 2.00000000001,\ 2.0\}$. Observe the comparison $A[1] < A[2]$ at line 3 (in the second iteration of the loop with $o \equiv A[1]$) in the actual world. The subtraction $A[1] - A[2]$ causes a large number of cancelled bits. The relative error is hence considered inflated. It is propagated immediately to the predicate at line 3, which is a discrete factor. Hence, according to our semantics, the execution is considered unstable because a different branch outcome may be taken in the ideal world.

However, the execution is stable. Because even if the predicate has different branch outcomes in the two worlds, $A[1]$ and $A[2]$ are so close to each other that selecting either one has little effect on the rest of the execution. Particularly, The relative error of the output value o is essentially $|(\hat{A}[2] - A[1])/A[1]|$ (assuming $A[1]$ was selected in the actual world), which is a very small value. Therefore, we shall suppress warnings inside the continuous core.

Fig. 9 (b) shows another example we have found in practice. Assume $x = 2.0$ and $y = 2.00000000000001$ and their correspondences in the ideal world are $\hat{x} = 2.0000000000001$ and $\hat{y} = 2.0$. In the actual world, the subtraction in line 10 has a large number of cancelled bits and the result is immediately used in the predicate. Hence, the execution is flagged unstable. In fact, the cancelled bits do correctly indicate the control flow differences between the actual world and the ideal world. However, the control flow differences are not important for the rest of the execution. More particularly, in both worlds, the output o has a value slightly smaller than 1.

We use a profiling technique similar to that in [2] to identify potential continuous core candidates. We manually inspect these candidates, which are in a small number (less than 20 in the largest program and only a few on average) and have fixed coding patterns such as selecting the maximum/minimum value from a set. We annotate the true cores. During execution, our runtime takes the annotations and suppresses warnings inside the annotated regions.

7.2 Predicates in Convergence Tests

In practice, a lot of computation tasks are iterative. An iterative method is supposed to generate a sequence of improving approximate solutions that eventually converge. Ideally, an iterative algorithm must be supported by a mathematically rigorous convergence analysis; however, heuristic-based iterative methods are also common. In the latter case, the methods may not converge. The implementation of an iterative algorithm must have a termination predicate that usually compares a value representing the solution generated by the current iteration and the value by the previous iteration. If their difference is small enough, the procedure is considered converged. If the algorithm is not provably convergent, a constant iteration bound is usually provided such that an execution is considered not converging if the solution difference is still large when the iteration bound is reached. The code snippet in Fig. 10 abstracts such a template.

Since the termination threshold is usually a very small floating point value, errors may induce different branch outcomes at the convergence test (e.g. line 7 in Fig. 10). If the algorithm is provably convergent, which can be inferred from the absence of an iteration bound from the program, we suppress any warnings generated at the convergence predicate. Intuitively, assume the procedure terminates at the ith iteration in the actual world but at the $(i+c)$th iteration in the ideal world, with c a positive or negative integer. The solutions in the two worlds do not differ much due to the iterative nature of the computation.

If the algorithm is not provably convergent, we have to take special care. We handle the following two sub-cases differently. Our discussion is based on the template in Fig. 10.

- If the $(bound-1)$th instance of the convergence test is flagged unstable, i.e. the result of $|x - old| - t$ is tagged with T, we consider the execution unstable. Assume the predicate at line 7 takes the true branch in the ideal world (and hence the procedure terminates), the significant error may induce a different branch outcome in the actual world (and hence the procedure gets to the $(bound)$th iteration and then fails to terminate at line 6). In other words, the output has discrete difference (i.e. having a convergent solution vs. no solution). It is similar when line 7 takes the else branch in the ideal world.
- If the convergence test is detected to be unstable at the ith instance with $i < bound - 1$, we consider the execution stable and suppress the warning. This is because although the significant error may cause different branch outcomes at the particular instance of line 7, the difference only leads to different iterations of computation, which only cause continuous output differences.

While these are the prominent challenges we have encountered when applying the technique to a set of real world programs, there may be others given the expressiveness of modern programming languages.

8. Evaluation

We implement the predictor by performing source code instrumentation on subject programs. We modify GCC-4.7.2 to instrument programs on GIMPLE IR. C/C++ and Fortran languages are naturally supported through the GCC frontend. Instrumented binaries execute at their original precision, with the add-on runtime system that monitors relative error inflation and propagation. Upon detecting an instability issue, the tool allows automatically switching to high precision. Particularly, programs are restarted with the high precision version which is statically generated by program transformation in the compiler.

We perform experiments to evaluate the efficiency and the effectiveness of our technique. We compare our technique with the state-of-the-art technique that relies on HPL [4], in which high precision values are stored in addition to original values, and side-by-side comparisons are performed to ensure execution stability. When implementing the HPL technique, we choose 128-bit quad double types as the HPL, which are slightly faster than using the GNU MPFR library as in [4].

In addition, we also implement another instability predictor that dynamically tracks the propagation of actual errors (not relative error). It records the corresponding error for each floating point variable, in the same precision as the variable. Note that separating an error from its corresponding value allows us to precisely represent the error as their exponents are allowed to be substantially different. For each operation, we compute the result error from the operand errors and the representation error of the result value itself. Such computation is much cheaper than high precision operations. Consider equation (1) in Section 4. Upon the addition operation, the result error is the aggregation of $\hat{\Delta}_x + \hat{\Delta}_y$ and the representation error of $x+y$. However, the latter cannot be precisely computed as $x+y$ is performed at normal precision. It is hence over-approximated by the value denoted by the least significand bit of the result value for safety. This approach is therefore more efficient but less precise than the HPL technique. We call it the *approximate* solution. The reason we implemented this method and compared it with others is that it is the most straightforward optimization of the expensive HPL solution.

Our experiments are performed on an Intel Core-i7 2.80GHz machine with 8GB RAM.

8.1 Performance

We evaluate the runtime overhead for the aforementioned three approaches, HPL, approximate and the proposed predictor. We use SPEC CFP 2000, a biochemical data processing program *deisotope* from [2], and *poly* from the motivating example in Fig. 2 (c), as the benchmark. We use the test input data set for SPEC programs except `191.fma3d`. The execution time for `191.fma3d` was too short with the test input, leading to difficulties in overhead measurement. Hence we used the larger training input for this program. The results are shown in Table 3. Column 2 shows the native execution time. Columns 3-4 present the results of HPL. Observe that the method is expensive, causing 109 times slow down on average. Such overhead is lower than what was reported in [4].

Columns 5-6 show the results of the approximate approach that monitors errors. The average slowdown is 34 times, which is about 3 times faster than the HPL approach.

The last two columns present the result by the proposed predictor. We collect the data with $\tau_c = 48$. It incurs a slowdown between 3.18 and 22.80, with an average of 7.91. It is 14 times faster than HPL. One outlier is `172.mgrid`, which degrades the performance by over 20 times. Note that `172.mgrid` also incurs large overhead in the HPL approach. The reason is that it is floating point computation intensive. Any instrumentation to the program introduces large performance penalty. Excluding this program, the average slowdown for the predictor is 6.84x. This offers more practical use than the HPL approach for on-the-fly detection of instability.

program	native time	HPL		approximate		ours	
		time	s/d	time	s/d	time	s/d
168.wupwise	1.46	106.69	73.08	41.69	28.55	9.50	6.51
171.swim	0.17	11.27	66.29	4.81	28.29	1.02	6.00
172.mgrid	1.96	855.14	436.30	88.38	45.09	44.69	22.80
173.applu	0.06	8.32	138.67	2.33	38.83	0.52	8.67
177.mesa	0.36	16.25	31.86	10.44	29.00	1.40	3.89
178.galgel	0.56	46.73	83.45	18.92	33.79	4.63	8.27
179.art	0.37	15.97	43.16	5.77	15.59	2.97	8.03
183.equake	0.14	10.11	72.21	3.74	26.71	0.82	5.86
187.facerec	1.09	109.58	100.53	29.06	26.66	6.18	5.67
188.ammp	1.90	108.77	57.25	40.07	21.09	6.05	3.18
189.lucas	0.72	81.05	112.57	23.63	32.82	6.47	8.99
191.fma3d	11.97	1187.68	99.22	510.35	42.64	85.08	7.11
200.sixtrack	1.41	249.56	176.99	73.52	52.14	11.05	7.84
301.apsi	1.17	121.58	104.09	65.20	55.82	8.35	7.15
deisotope	0.02	0.50	33.00	0.39	19.50	0.13	8.67
AVERAGE			109.46		34.07		7.91

†time is in seconds; s/d stands for slowdown.

Table 3. Performance.

Breakdown of Execution Time We further investigate the distribution of execution time for our predictor to understand the overhead of individual pieces. The breakdown is measured by computing the differences between enabling and disabling each component in the predictor. The results are shown in Fig. 11. Tag propagation, colored in red, dominates the execution time for the majority of the programs. It contributes to 35% to 70% of the whole execution. For each floating point operation, we have to compute the addresses of the operand tags and the result tag, load the tags, and perform tag operation. The purple portion in each bar represents the time spent on detecting error inflation. The green portions include costs for error suppressions and others.

8.2 Effectiveness

In this experiment, we compare the effectiveness of the different approaches. We only report results for the programs

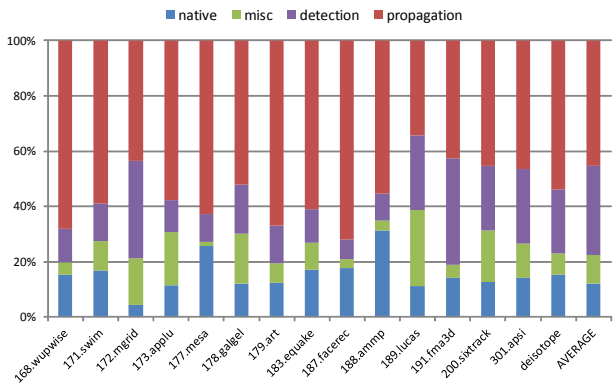

Figure 11. Breakdown of the Overhead.

that have instability problems. For the other programs, all the three approaches correctly detect that they are stable. The experiment is similar to that in Fig. 2. We take a large number of samples in a small input sub-domain of each program. For each sample, we run the three techniques to detect if it is stable. In HPL, if any discrete differences are observed at discrete factors between the regular precision version and the high precision version, the execution is considered unstable. In the approximate approach, an execution is unstable if the floating point values at discrete factors could cause discrete differences when they are aggregated with the computed errors.

For our predictor, we also present the results with different settings of threshold τ_c, which is the threshold for deciding inflation (Rule [**ADD-FF**]). The other threshold τ_s that detects suppressions (Rule [**ADD-FT/TF**]) is always set to 4. The results are not sensitive to τ_s and thus we omit the data regarding the different settings of τ_s.

The results are shown in Table 4. The second column shows the different methods/configurations. The *overall* rows indicate the total samples collected and the sampling domain. These samples are selected in a very small range around the original input. The third column shows the number of unstable samples. The fourth column shows the percentage of unstable samples over the total number of samples. The last column shows the unstable sample ranges. Some are not continuous, meaning the range may have stable and unstable sub-ranges.

We have the following observations. (1) Even SPEC programs are unstable. Consider `galgel`. In many samples, the high precision version produces results but the corresponding regular precision version fails to produce any results, or vice-versa. Note that the unstable samples reported by HPL are essentially true positives. (2) Our detector reports 7.5-93 times more problematic cases with the setting of $\tau_c = 48$, when compared with the ground truth (i.e. the results by HPL). However, the percentage of the reported cases over the entire input range is very very small. This implies in most cases (over 99.999996%[5]), our technique can correctly predict that an execution is stable, with low overhead. (3) Our detector has comparable effectiveness as the approximate approach with $\tau_c = 48$. Note that the proposed predictor is 5 times faster. (4) The number of false positives of our technique grows when τ_c decreases because it admits more inflations. However, even with the most conservative setting, the percentage of the unstable cases is still very small. Note that the maximum possible τ_c is 53 because there are 53 bits for the significand in the representation of a double precision variable. (5) There are false negatives when τ_c gets close to the maximum. When having $\tau_c = 52$ it misses some unstable cases for `187.facerec` and `poly`. It also fails to capture some unstable cases for `178.galgel` with τ_c greater than 48. Our detector does not miss any true positives with $\tau_c = 48$. For programs other than `178.galgel`, a more aggressive threshold $\tau_c = 50$ does not cause any false negatives. Observe that `178.galgel` has a lot of unstable executions, which may suggest that one should use a more conservative setting when the number of warnings is high.

Continuous core and convergence test annotations are important for our technique. For the first four programs in Table 4, it always reports unstable for every execution, if there is no annotation of continuous cores. Convergence tests are commonly used in `178.galgel`. Without the convergence test annotations, we receive a lot of false warnings for this program. In both cases, all the false warnings are issued within some continuous cores or right at some convergence test predicates.

We have also evaluated a traditional technique that is solely based on cancelled bit detection [25]. The technique issues a warning whenever a larger number of cancelled bits are detected. Our experiment shows that it always reports warnings for all the executions (including the stable ones) of the first four programs in Table 4, even after we suppress the warnings in continuous cores and convergence tests. This is because bit cancellation is very common in programs. It may not be harmful unless the imprecise values are further used in critical places.

8.3 Tracing the Cause of Instability Issues

We extend our predictor to trace the root cause of instability issues. The extension is straightforward. In addition to the inflation bit, we also record and propagate the location that an error inflation originates. For a binary operation with both operands tagged with T, we propagate the location record of the larger operand. In the future, we plan to develop a set based implementation to propagate the records from both operands. When an instability issue arises, the cause can be quickly identified from its location record. We discuss one real-world example below.

[5] This lower bound is from `galgel` with $\tau_c = 48$. The problematic range is only $3.28E-6\%$ of the input range such that 99.999996% of the inputs are correctly predicated as stable.

program	approach	# of cases	%	detected range
equake	HPL	3	3.00E-12%	[0.8690799016130847, 0.8690799016130848]
	approx	158	1.58E-10%	[0.869079901613077**9**, 0.869079901613093**6**]
	ours(τ_c=36)	1155972	1.16E-6%	[0.86907990149748**77**, 0.8690799016130848]
	ours(τ_c=40)	72247	7.22E-8%	[0.8690799016**058602**, 0.8690799016130848]
	ours(τ_c=44)	4516	4.52E-9%	[0.8690799016**026333**, 0.8690799016130848]
	ours(τ_c=48)	279	2.79E-10%	[0.869079901613057**0**, 0.8690799016130848]
	ours(τ_c=50)	71	7.10E-11%	[0.869079901613077**8**, 0.8690799016130848]
	ours(τ_c=52)	20	2.00E-11%	[0.86907990161308**29**, 0.8690799016130848]
	overall	1E+14		[0.8650, 0.8750]
facerec	HPL	849	8.49E-10%	[0.596265750063108, 0.596265750064926]
	approx	4217	4.22E-9%	[0.59626575006**1204**, 0.59626575006**6312**]
	ours(τ_c=36)	59457611	5.95E-5%	[0.5962657**20335802**, 0.5962657**79793411**]
	ours(τ_c=40)	2716295	2.72E-6%	[0.5962657**48204800**, 0.596265751**922657**]
	ours(τ_c=44)	232165	2.32E-7%	[0.5962657**49946110**, 0.596265750**181511**]
	ours(τ_c=48)	12613	1.26E-8%	[0.59626575005**6901**, 0.59626575007**1028**]
	ours(τ_c=50)	2643	2.64E-9%	[0.5962657500**62749**, 0.5962657500**66600**]
	ours(τ_c=52)	296	2.96E-10%	[0.596265750063**257**, 0.596265750065**373**] †
	overall	1E+14		[0.5900, 0.6000]
galgel	HPL	57695	5.77E-8%	[0.8184459012000007, 0.8184459012253359]
	approx	4797213	4.80E-6%	[0.818445902**02708479**, 0.818445902**2880854**]
	ours(τ_c=36)	255406459	2.55E-4%	[0.8184458**871521804**, 0.8184459**127084799**]
	ours(τ_c=40)	126738078	1.27E-4%	[0.8184458**934897399**, 0.818445902**61871598**]
	ours(τ_c=44)	37972131	3.80E-5%	[0.8184458**998299998**, 0.818445902**39575860**]
	ours(τ_c=48)	3278089	3.28E-6%	[0.818445902**02196792**, 0.818445902**0723309**]
	ours(τ_c=50)	1801215	1.90E-6%	[0.818445901**3178548**, 0.818445902**0723309**] †
	ours(τ_c=52)	1233455	1.23E-6%	[0.818445901**9084903**, 0.818445902**0723309**] †
	overall	1E+14		[0.8184, 0.8185]
deisotope	HPL	2	2.0E-12%	[1.1156381266106556, 1.1156381266106557]
	approx	18	1.8E-11%	[1.1156381266106556, 1.115638126610657**3**]
	ours(τ_c=36)	167157	1.7E-7%	[1.115638126**5939401**, 1.1156381266106557]
	ours(τ_c=40)	10447	1.0E-8%	[1.1156381266**096111**, 1.1156381266106557]
	ours(τ_c=44)	653	6.5E-10%	[1.115638126610**5905**, 1.1156381266106557]
	ours(τ_c=48)	40	4.0E-11%	[1.11563812661065**18**, 1.1156381266106557]
	ours(τ_c=50)	7	7.0E-12%	[1.11563812661065**51**, 1.1156381266106557]
	ours(τ_c=52)	5	5.0E-12%	[1.11563812661065**53**, 1.1156381266106557]
	overall	1E+14		[1.1100, 1.1200]
poly	HPL	2	2.0E-12%	[1.8408964152537146, 1.8408964152537147]
	approx	211	2.1E-10%	[1.840896415253704**1**, 1.840896415253726**0**]
	ours(τ_c=36)	61177	6.1E-8%	[1.84089641525**06552**, 1.84089641525**67738**]
	ours(τ_c=40)	3821	3.8E-9%	[1.8408964152535**234**, 1.8408964152539**065**]
	ours(τ_c=44)	235	2.4E-10%	[1.840896415253702**8**, 1.840896415253726**9**]
	ours(τ_c=48)	15	1.5E-11%	[1.84089641525371**32**, 1.84089641525371**62**]
	ours(τ_c=50)	2	2.0E-12%	[1.8408964152537146, 1.8408964152537147]
	ours(τ_c=52)	0	0.0%	†
	overall	1E+14		[1.8400, 1.8500]

Table 4. Comparisons of the detected problematic ranges with different predictors. † following a detected range indicates that there are false negatives for the current setting of τ_c.

```
    void syshtn(Y, ...) {
        double *H;
        ...
62      H(i) = ...;
63      for (...)
            Y(i) = Y(i) - H(i);
        ...
    }
```

```
    void nwtn() {
        while (...) {
110         call Fname(Y, ...);
            ...
          /* d=DOT_PRODUCT(Y,Y); */
122         for (...) {
123             d += Y(i) * Y(i); }
124         if (d <eps2) { break; }
    } }
```

Figure 12. Pseudocode snippet from `178.galgel`.

Fig. 12 shows a pseudocode snippet from `178.galgel`. In one of the executions, the program produces a different output from the one by HPL. The observed difference lies in the predicate outcome at line 124 in function nwtn(). The HPL version takes the true branch to jump out of the loop and continues, whereas the regular version takes the false branch.

Our predictor successfully reports an unstable run due to the predicate at line 124. It shows variable d carries an inflated error when comparing against $eps2$. In addition, it further indicates that the cause of the inflation is due to the cancellation at line 63 in function syshtn() which was called earlier from line 110.

We verify the cause of the issue by investigating the computations at line 63, where the elements of array Y get updated. Before performing the subtraction of $Y(i)-H(i)$, neither of them is tagged with T. Taking $Y(7)$ and $H(7)$ for example, $Y(7)$=-15.84869578667144, $H(7)$=-15.8486957866 7145 and the relative errors are both around 2.65E-9. $Y(7)$ is tagged with T after the subtraction, since 50 bits get cancelled. As a result, it has a value of $Y(7)$=2.88E-11, but the relative error of $Y(7)$ grows up to 0.003. Most other elements in Y are also tagged with T similarly. From this point on, Y carries the inflated errors in its following execution. When getting back to the computation of d at line 123, the inflated errors eventually propagate from $Y(i)$ to d. During this process, none of the product $Y(i)*Y(i)$ is large enough and tagged with F to suppress the inflated errors. Upon finishing the loop, Δ_d =0.015, which is correctly captured by the T tag. Hence it triggers a warning at the predicate. With the help of the predictor, the cause of the instability is easily located. However, the implementation around the cause is so complex that a more stable version may be difficult to construct, which supports our argument that online prediction plus on-demand precision hoisting is a more suitable solution.

9. Limitations

We propose to address the floating point instability problem in a way different from traditional techniques. We use runtime prediction to determine if an execution is stable. For the very few unstable executions, we allow the user to switch to running high precision versions on demand. However, as an initial step along this direction, our technique has a number of limitations. First of all, its requires setting two thresholds. While we were able to find a uniform configuration for the programs we have considered, it is unknown if the configuration is equally effective for other programs. Second, it requires the user to annotate continuous cores and convergence test predicates. We argue that such annotations should become an integral part of the programming language support for applications that have high demand for reliability. As such, it will allow programmers to annotate their own code during development. Advanced static analysis may also be developed to determine continuous cores and convergence test predicates. Third, the current implementation has nontrivial overhead even though it is much faster than the HPL solution and the approximate solution (Section 8). We notice that similar dynamic analyses that propagate bits, such as dynamic information flow tracking, may be much more efficient than our technique (e.g. only 6.2% overhead for server applications and 3.6 times overhead for SPEC programs were reported in [33]). However, when we try to apply similar optimizations, we encounter a number of unique challenges for floating point programs. For example, the instruction pipeline behaves differently for integer and floating point programs in the presence of instrumentation. We are working on addressing these problems.

10. Related Work

The instability problem has been studied for a long time. The most prominent difference between our work and many existing works is that existing techniques treat it as a debugging problem. They try to locate the unstable statements using various analysis, report them to the users, and hope they will get fixed. However, our study shows that instability will not happen for most parts of the input domain and they can be suppressed by higher precision. Hence, we aim to develop an efficient on-the-fly predictor to assure stability for most executions, and only switch to using high precision computation when necessary. Also, a key advance of our work is to leverage the concept of discrete factors to have a more precise error reporting criterion that is critical for a low false positive rate.

Interval arithmetic [29, 31] uses an error range to represent a value. It updates ranges in a conservative way, leading to over-sized ranges. Hence, people further propose affine arithmetic [11, 13, 15] to represent errors as affine forms to allow better precision. However, they are very expensive and their goal is in-house testing and debugging. The state of the art [10] has 3-5 orders of magnitude slow down and only supports small programs. Researchers have proposed using high precision library to precisely compute values and errors during execution [1, 4]. In particular, the state-of-the-art [4] is a dynamic analysis that makes use of a binary instrumentation engine to enhance a floating point program on the fly to perform high precision computation. That is, upon executing a single precision operation, the instrumentation executes the corresponding high precision operation. The results of the two precision levels are compared to detect possible in-

stability issues. However, high precision operations are very expensive (e.g. [4] has 2 orders of magnitude slow down). In contrast, our predictor has only 7.91 times slowdown. And our precision loss is small in the picture of the whole input domain. Monte Carlo Arithmetic (MCA) exploits randomness in floating point arithmetic [**?**]. It executes a program multiple times by randomizing floating point arithmetic operators and operands. It studies roundoff errors statistically from the results. The CADNA library [**?**] performs stochastic estimations of errors also by randomly selecting rounding modes at each operation. With randomization approach, it can miss instability problems in certain situations [23]. It is also expensive (90x slowdown for SPEC with CADNA library according to our experiment). In [25], a dynamic technique detects instability by monitoring bit cancellations was proposed. It is very expensive and has a large volume of false positives.

There are also a large body of work on abstract interpretation, SMT solving, model checking, and code perturbation to tackle the representation error problem [5, 9, 12, 14, 16, 21, 27, 28, 30, 35]. Particularly, robustness analysis [7] tries to statically prove that a floating point program is free from instability problems. While it is quite successful in handling simple programs, the mathematical complexity and the iterative nature of many real world floating point programs are difficult to address by the technique. Moreover, as instability problems are input dependent and rarely happen, dynamic analysis like ours have the advantage of allow running the low precision program in most cases for efficiency and switching to high precision on demand. It is especially prefereable when completely fixing instability problems is difficult.

There are some existing works focusing on external errors. The error bounds for external errors are much larger compared to internal representation errors. Program behavior may vary a lot within external input error bounds. Monte Carlo (MC) approaches are widely used [19] in handling external errors. White-box sampling [2] was recently proposed to reduce the number of needed samples. It hashes discrete values at discrete factors and uses the resulting hash values to guide the sampling process. However, sampling cannot be applied in our context as representation error bounds are too small to be representable without using higher precision. Static analysis has also been proposed to prove a program is continuous [6, 17, 26] or robust [7, 34] in the presence of input errors.

There are static analysis, symbolic execution, theorem proving, and model checking techniques that focus on detecting or proving the absence of logical floating point bugs (e.g. divided-by-zero, overflow, and underflow) [3, 8, 18, 24]. Particularly, [3] features solving floating point constraints. They are usually heavy-weight analysis and they focus on bug finding in programs. These bugs are different from instability problems by nature.

11. Conclusion

We develop an online technique to monitor and predict floating point program execution instability problems. We observe that in practice, only a very small portion of inputs can lead to unstable execution and hence the expensive high precision computation based approaches do not pay-off for most inputs. We formally define an unstable execution as execution in which errors cause discrete differences. The definition allows us avoid explicitly computing errors, which is very expensive. Instead, we abstract inflation of relative errors as one bit, monitor the propagation of such bits, and check if they can reach discrete factors. If so, the inflated errors may induce discrete differences at these factors and hence the execution is flagged unstable. The key challenge is to detect places where the inflation bit propagation is cut off, indicating the error is no longer inflated compared to its value. We discuss the soundness and completeness of the technique and the practical challenges that we have overcome. Our experiments show that the technique can correctly classify over 99.999996% inputs as stable with a specific threshold setting (τ_c=48) while a traditional technique that solely detects error inflation mis-classifies majority of inputs as unstable for some of the programs we studied. Compared to the state of the art high precision computation based approach, our technique is much more efficient (with an average overhead reduction of 14 times) and can report all the unstable executions. Due to approximation, our technique classifies 7.5-93 times more inputs as unstable (τ_c=48), when compared to the ground truth. However, these inputs only count for 1.5E-11% - 3.3E-6% of the input domain. In other words, in majority cases, our technique has the same effect as the high precision approach. Therefore, users can use our technique to make prediction with a lower cost. For the very rare cases that are considered unstable by our predictor, our technique provides the capability of automatically switching to high precision.

Acknowledgments

This research is supported, in part, by the National Science Foundation (NSF) under grants 0845870, 0916874 and 1320444. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] David An, Ryan Blue, Michael Lam, Scott Piper, and Geoff Stoker. Fpinst: Floating point error analysis using dyninst. 2008.

[2] Tao Bao, Yunhui Zheng, and Xiangyu Zhang. White box sampling in uncertain data processing enabled by program analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 897–914, New York, NY, USA, 2012. ACM.

[3] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, 2013. ACM.

[4] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM.

[5] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[6] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 57–70, New York, NY, USA, 2010. ACM.

[7] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 102–112, New York, NY, USA, 2011. ACM.

[8] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 7–9, New York, NY, USA, 2007. ACM.

[9] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyzer. In *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[10] Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in scala. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 325–344, New York, NY, USA, 2011. ACM.

[11] L. H. de Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 37:147–158, 2004.

[12] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Claire Fang Fang, Tsuhan Chen, and Rob A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 561–564, 2003.

[14] Eric Goubault. Static analyses of the precision of floating-point operations. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 234–259, London, UK, UK, 2001. Springer-Verlag.

[15] Eric Goubault and Sylvie Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In *Proceedings of the 14th international conference on Static Analysis*, SAS'07, pages 137–152, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 232–247, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Dick Hamlet. Continuity in software systems. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 196–200, New York, NY, USA, 2002. ACM.

[18] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174, March 1996.

[19] J.C. Helton, J.D. Johnson, C.J. Sallaberry, and C.B. Storlie. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Engineering & System Safety*, 91(1011):1175 – 1209, 2006.

[20] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[21] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 661–667, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Fabienne Jzquel and Jean-Marie Chesneaux. Cadna: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933 – 955, 2008.

[23] William Kahan. How futile are mindless assessments of roundoff in floating-point computation? 2006.

[24] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, and Xavier Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Software and Systems - ERTSS 2010*, 2010.

[25] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, (0):–, 2012.

[26] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 355–363, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Matthieu Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 194–208, London, UK, UK, 2002. Springer-Verlag.

[28] Matthieu Martel. An overview of semantics for the validation of numerical programs. In *Proceedings of the 6th international conference on Verification, Model Checking, and Ab-*

stract Interpretation, VMCAI'05, pages 59–77, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] Guillaume Melquiond and Cesar Munoz. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH '05, pages 188–195, Washington, DC, USA, 2005. IEEE Computer Society.

[30] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008.

[31] R.E. Moore. *Interval analysis*. Prentice-Hall series in automatic computation. Prentice-Hall, 1966.

[32] U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. GAO/IMTEC-92-26, Feb 1992.

[33] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[35] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 131–142, New York, NY, USA, 2010. ACM.

[36] B. A. Worley. Deterministic uncertainty analysis. Technical Report ORNL-6428, Oak Ridge National Lab. TN (USA), 1987.