RAIVE: Runtime Assessment of Floating-Point Instability by Vectorization

Wen-Chuan Lee[1], Tao Bao[1], Yunhui Zheng[1], Xiangyu Zhang[1], Keval Vora[2], Rajiv Gupta[2]

[1]Department of Computer Science, Purdue University, US
{lee1938, tbao, zheng16, xyzhang}@cs.purdue.edu
[2]CSE Department, University of California, Riverside, US
{kvora001, gupta}@cs.ucr.edu

Abstract

Floating point representation has limited precision and inputs to floating point programs may also have errors. Consequently, during execution, errors are introduced, propagated, and accumulated, leading to unreliable outputs. We call this the *instability problem*. We propose RAIVE, a technique that identifies *output variations* of a floating point execution in the presence of instability. RAIVE transforms every floating point value to a vector of multiple values – the values added to create the vector are obtained by introducing artificial errors that are upper bounds of actual errors. The propagation of artificial errors models the propagation of actual errors. When values in vectors result in discrete execution differences (e.g., following different paths), the execution is forked to capture the resulting output variations. Our evaluation shows that RAIVE can precisely capture output variations. Its overhead (340%) is 2.43 times lower than the state of the art.

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; instability; cancellation; vectorization

1. Introduction

Data processing using floating point programs is essential in the emerging big data era. During program execution, errors can be introduced, propagated and accumulated, potentially leading to unreliable outputs. We call this the *floating point instability problem*. The errors introduced include those due to precision limitations in physical instruments or human efforts in acquiring the inputs, called the *external errors*, and those from limited representation precision, called the *internal errors*. Handling instability is critical because important decisions may be based on data processing results – results of computer simulations may be used to setup expensive scientific wet bench experiments; commercial decisions may be made based upon results of mining customer data etc. Evidence suggests that widely used data processing programs suffer from instability. We will show in Section 5 that a widely used implementation of the *k-means* data mining algorithm [13] can produce completely different clustering results; an information retrieval program *pagerank* [14] may produce completely different rankings, both due to the instability problem.

Researchers have developed various techniques to address the instability problem. Static techniques such as abstract interpretation and theorem proving [9, 18, 23] were proposed to reason about the existence or the absence of instability. Interval arithmetic [24, 26] and affine arithmetic [10, 12, 17] model errors as ranges or affine formulas to reason about execution stability. Program transformation was proposed to improve precision and stability [1, 5]. Recently in [2], an on-the-fly predictor was proposed to detect instability. Based on the prediction result, the execution may switch to a higher precision.

While mostly focusing on internal errors, existing works use the *ideal execution* with infinite precision as the oracle to reason about instability in *actual execution* and then aim to eliminate the differences between the two executions to produce reliable results. However, we argue that these approaches may be undesirably restricted in their scope. Even if the detected differences between ideal and actual executions are eliminated (e.g., by hoisting the precision) the execution may nonetheless be unstable as the same differences may be easily triggered by minor perturbations of the in-

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...
http://dx.doi.org/10.1145/2814270.2814299

put (due to external errors). Consider the example in Fig. 1 with a predicate (line 3) that is unstable when the input x is close to 0.0045. The curve at the bottom shows how output y varies with input x. Observe that $y = f(x)$ is on the left and $y = g(x)$ on the right, and there is a discontinuity right at $x = 0.0045$. In contrast, the curve on the top is slightly off due to internal errors. Particularly, the discontinuity is at a value close to 0.0045. Assuming the technique in [2] reports that the unstable input range is r_l so that higher precision should be used for values in the range. Unfortunately, even use of infinite precision is insufficient as it does not alert the user that the program output may change substantially in the presence of a small external error when input x is close to 0.0045. In other words, *instability is a property of the computation performed on a given input, which cannot be completely evaded by hoisting representation precision*. Thus we argue that achieving the same results from the ideal and actual executions ignores instability due to external errors which are inevitable in the real world.

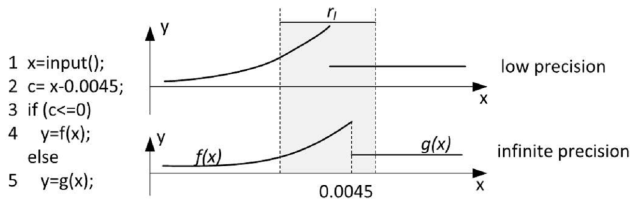


Figure 1. Inevitable External Errors

Moreover, a patch to the problem may not exist, which is different from functional bugs. In our example, the instability is due to the interface between the continuous floating point domain (i.e. variable c at line 3) and the discrete boolean domain (i.e. the branch outcome at line 3) and intrinsic to the algorithm. Changing the implementation, which is a typical method to improve stability for the floating point domain, is unlikely to completely fix the problem due to the involvement of the discrete domain.

In this paper, we observe that the key to handling runtime instabilities is not to achieve an execution close to the ideal one, but rather to *inform the user of the possible (output) effects of the instabilities* so that he/she can adjust the decision accordingly. We propose a novel technique that discloses the possible effects of errors on outputs in an actual execution for a given input i, including both internal and external errors. It does so without explicitly mutating the input i, which usually requires a large number of sample executions to expose output variations if the input is of high dimension and input correlation is complex. Instead, our technique vectorizes the subject program such that each floating point variable is represented by a vector of multiple values. Initially, the values in a vector are identical, representing the value in the actual execution (called the *actual value*). When execution encounters operations that yield non-trivial errors, it explicitly introduces artificial errors to the actual value. The injected errors are the lower and upper bounds of the actual error (note that actual errors cannot be efficiently computed without high precision representations). Vectorization allows the mutated values to go through the same sequence of floating point operations. If they lead to discrete differences (e.g., taking different paths), the execution is forked to follow the different discrete options to capture the output variations caused by errors.

Consider the example in Fig. 1 . Given an input x_0 within r_l, our technique detects that the predicate at line 3 is unstable. It hence forks the execution to take both branches. As such, the output variations within r_l are indicated by the differences between $y = f(x_0)$ (from the true branch) and $y = g(x_0)$ (from the false branch). The essence of our approach is that if internal representation errors can lead to different branch outcomes at a predicate, a small perturbation in external input is very likely to lead to the same difference. Hence we model the possible effects of external errors by modeling only internal errors.

Our key contributions include:

- We propose a novel vectorization based approach that addresses instability that can be triggered by internal or external errors and captures output variations in the presence of instability.
- We address a number of critical technical challenges, including avoiding unnecessary forks that may generate too many processes, and handling error suppressions that can properly stop error propagation.
- We develop the RAIVE (Runtime Assessment of floating point Instability by VEctorization) prototype. It precisely predicts output variations in the presence of errors, with both the precision and the recall close to 100%. The detected output changes are substantial. Its overhead is 2.43 times smaller than the state of the art runtime instability predictor [2] that cannot predict output variations, but rather just the stability of an execution. It correctly classifies executions on over 99.99% of the inputs of the programs we studied as stable.

2. Background

Floating Point Representation. IEEE 754 [19] defines the format of a 64-bit floating point value as shown in Fig. 2. The corresponding decimal value f is given by $f = (1-2s) \times (1 + m \times 2^{-52}) \times 2^{e-1023}$ where variable s is the sign bit, m the significand, which is also called mantissa, and e the exponent. There are 53 mantissa bits, including an implicit

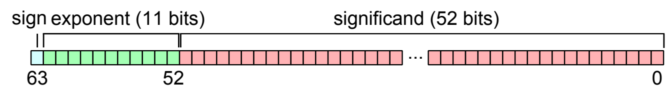


Figure 2. Float Point Representation.

leading bit of “1”. Any values that require more significand bits to represent cannot be precisely represented.

We use x to denote the floating point value of a variable x in the actual world (with limited precision), called the *actual value*. The corresponding value in the ideal world (with infinite precision) is called the *ideal value* and denoted as $\hat{x}$. The difference between the two is called the *absolute error* of x, denoted as $\hat{\Delta}_x$. The *relative error* x, denoted by Δ_x, is defined as $|\hat{\Delta}_x/x|$. It indicates whether the actual value is reliable. Initially errors are introduced in source code constants at compile time, or when reading external inputs at runtime. The initial errors are propagated to the internal of program execution through operations.

Discrete Factor and Instability. A discrete factor is an operation that has floating point values as operands and produces a discrete value [3]. They are the interface between the continuous domain and the discrete domain (e.g., integer and boolean). Typical examples are control flow predicates and type casts. Discrete factors have been used to detect instability caused by representation errors [2]. If an actual floating point value (with error) and the corresponding ideal value lead to different discrete results – one having the true branch outcome and the other the false – the execution is considered unstable. A discrete factor that yields different discrete values in the actual and the ideal worlds is called an *unstable factor* [2]. The intuition is that if we consider the output of an execution as a mathematical function over inputs, the form of the function is determined by the executed path. In the presence of unstable factors, the paths and hence the functions are different in the two worlds, indicating substantial output differences.

Relative Error Inflation and Cancelled Bits. Many existing works on instability detection [2, 22] are built upon detecting relative error inflation, i.e., instances when the relative error of the result of a floating point operation is substantially larger than those of the operands. This is because instabilities are rooted at relative error inflation in most cases. Per IEEE 754, the result of a subtraction/addition is normalized by left-shifting to remove the leading zeros. The relative error inherited from the operands thus gets inflated. If after subtraction $z = x - y$, the significand bits are left-shifted by d bits, the relative error inherited from the operands, $(\hat{\Delta}_x - \hat{\Delta}_y)/z$, may become 2^d times larger than the relative errors of x and y, because z is 2^d times smaller than x or y. The left-shifted bits are also called the *cancelled bits* [22], which can be cost-effectively monitored by comparing the exponents of the result and the larger operand. In particular, $d = \max(\varepsilon_x, \varepsilon_y) - \varepsilon_z$, with ε_x being the exponent of x. In TAG [2], when the number of cancelled bits is larger than some pre-defined threshold τ_c, the relative error is considered inflated. TAG further tracks the propagation of the value with an inflated relative error. An execution is considered unstable if the value can reach a discrete factor, as discrete difference (from the ideal execution), such as path difference, may be induced by the value.

3. Vectorization and RAIVE Overview

In this section, we overview RAIVE with an example and briefly explain a few important technical challenges.

Given a subject program, RAIVE leverages the compiler to transform it to a vectorized version, in which each floating point variable x is represented by a vector of four 64-bit floating point values $\langle x_1, x_2, x_3, x_4 \rangle$. All floating point operations are transformed to the corresponding vector operations. Initially, the vector values x_1,..., x_4 have the same actual value of x. These values remain identical after floating point operations. A lightweight online relative error inflation detection mechanism is also inserted in the subject program. When an error inflation is detected for a variable x, the corresponding value is considered unreliable. Since we cannot precisely compute the error of x, which requires using higher precision, RAIVE computes an upper bound of the absolute error, denoted as δ, and mutates the first two values in x's vector, called the *left pair*, to $x - \delta$ and $x + \delta$, respectively. The same perturbation is applied to the last two values, called the *right pair*. As such, the right pair is a simple duplication of the left pair. The perturbed vector further goes through floating point operations such that the value differences in x's vector lead to value differences in other vectors, simulating the propagation of the unreliable actual value.

Upon executing a discrete factor (e.g., a predicate on floating point values), RAIVE tests if the values in the left pair lead to different discrete values (e.g., different branch outcomes) – meaning that some previously inflated relative errors have been propagated to the factor and caused discrete difference; the execution may likely become unstable. A naive solution would be to fork the execution right away so that each of the resulting executions proceeds with a unique discrete value generated at the unstable discrete factor. For instance, assuming an unstable predicate, the original execution is forked to two with one executing along the true branch and the other along the false branch. As such, the outputs generated by the forked executions denote the possible output variations in the presence of errors.

However in real-world programs, discrete differences do not necessarily lead to final output variations. For instance, even if a predicate on floating point values has different branch outcomes in the presence of errors, the two branches may have identical or highly similar effect such that the two forked executions have very little or no state differences. To address this problem, we develop a highly sophisticated runtime. Particularly, when encountering an unstable predicate, RAIVE executes both branches in sequence (in the same execution). To avoid interference between the branches (e.g. a write in one branch affects a read in the other branch, which is infeasible according to program semantics), RAIVE executes one branch on the left pairs of the vectors and the other branch on the right pairs. In other words, each variable update in a branch only modifies half of the vector. At the join

point of the two branches, RAIVE compares the left and the right pairs in each vector that was updated in the branch(es). If all of them have negligible differences, the unstable predicate was benign and there is no need to fork the execution. Otherwise, RAIVE forks the execution.

Example. Consider an example program in the first column of Fig. 3. It involves both computation and decisions (i.e. lines 5, 10 and 15). All the variables are of the 64-bit double-precision type. The execution on an x86 platform is presented in the second column. The ideal execution, shown in the third column, uses infinite precision emulated via software with two to three orders magnitude of slow-down.

Initially, a very large value is assigned to a (line 1); the same large value plus one is assigned to b (line 2). In the actual execution, the increment cannot be represented and b has the same value as a. Thus, an error is introduced at line 2 in the actual execution. As such, $c = b - a$ at line 3 has values 0 and 1 in the actual and ideal executions, respectively. Then c is used in subtraction with a large value, the result of which is further used in a conditional statement (lines 4 and 5). The error introduced earlier does not cause any problem as it is too small in comparison with the large operand value 2^{37} at line 4. At line 7, a and b are passed to *max()*, inside which $b-a$ is again performed. However, since the result is directly used in a conditional (line 10), the error manifests itself by yielding different branch outcomes in the two executions, leading to different return values, which are used in the subtractions with a (line 13) and then with a constant value 0.5. The resulting different values cause different branch outcomes at line 15, and eventually different outputs.

Working of RAIVE. Initially, the vector of variable a holds four identical 64-bit value 2^{54}. At line 2, the floating point addition is transformed to a vector addition. Due to the same precision limitation as the non-vectorized actual execution (in the second column), b's vector holds four identical values. Inside box Ⓓ, the subtraction at line 3 causes relative error inflation (Section 2). In particular, the result of the operation is 0 whereas the operands are very large values. RAIVE detects the inflation such that it introduces artificial errors in the vector that denote the bounds of the absolute errors. The first value -64 is a lower bound; the second value 64 denotes an upper bound. The right pair is a simple duplication of the left pair. We will discuss how the bounds are computed in Section 4.1. At line 4, the vector of d becomes four identical values despite the differences in c. In other words, the errors are *suppressed* by the subtraction with the large operand 2^{37} (due to the precision limitation of the operation). Therefore, the four values of d's vector yield the same branch outcome (at line 5), correctly modeling the fact that both the actual (column 2) and the ideal (column 3) executions take the same branch.

In box Ⓔ inside the function `max()`, the same relative error inflation is observed at line 9. However, since the variable

	TAG	RAIVE
	x' the error bit of x	$\langle x_1, x_2, x_3, x_4 \rangle$ denoted as x^v,
1	$a = 2^{54};$	$a^v = \langle 2^{54}, 2^{54}, 2^{54}, 2^{54} \rangle;$
1.1	[$a' = F$];	
	...	...
3	$c = b - a;$	$c^v = b^v - a^v;$
3.1	[**if** $(b' \wedge a')$ $c' = T;$	[**if**$(\max(\varepsilon_b, \varepsilon_a) - \varepsilon_c > \tau_c)$
3.2	**elseif** (b') $c' = \neg(\varepsilon_a - \varepsilon_b > \tau_s);$	$\delta = 2^{\max(\varepsilon_{b_1}, \varepsilon_{a_1}) - \tau_c + 1};$
3.3	**elseif** (a') $c' = \neg(\varepsilon_b - \varepsilon_a > \tau_s);$	$c^v = c^v + \langle -\delta, \delta, -\delta, \delta \rangle;$]
3.4	**else**	
3.5	$c' = (\max(\varepsilon_b, \varepsilon_b) - \varepsilon_c > \tau_c);$]	
	...	...
18	$o = h * 2;$	$o^v = h^v * 2^v;$
18.1	[$o' = h' \vee 2'$]	

Figure 4. Boxed statements correspond to instrumentation. Note that in RAIVE, the original floating point related statements are completely replaced by vector statements. Labels 3.1-3.5 denote instrumentation for line 3.

t with inflated error is directly used in the predicate at line 10, different branch outcomes are observed. The predicate is an unstable discrete factor. Observe that in this case, the actual and ideal executions do differ at line 10. RAIVE does not fork the execution at this point as the instability may be benign. Instead, inside box Ⓕ, it first executes the true branch with a vector mask that enables only the left pairs of vectors. As a result, the left pair of e is assigned the left pair of a, which holds two identical values of 2^{54} (line 11). After that, it further executes the false branch with a mask enabling the right pairs. Hence, the right pair of e is assigned the right pair of b, which also holds the same two values. At the join point of the two branches, RAIVE tests if the left and right pairs of e are identical or have negligible differences. In this case, since they are identical, the instability at line 10 is benign. No forking is needed. Intuitively, although the actual and ideal executions have different control flow inside the `max()` method due to the error, the relative error of the return value is very small (i.e. $1/r$). In other words, the predicate at line 10 only becomes unstable when a and b are very close; however in such a case, returning either a or b does not make much difference and thus the instability is benign. RAIVE continues execution with the full vectors.

In box Ⓖ, RAIVE detects error inflation at line 13 and introduces artificial errors. The errors cannot be suppressed by the operation at line 14. As such, another unstable predicate is detected at line 15. However in this case, the two branches yield different left and right pairs in h. RAIVE forks the execution (box Ⓗ). In one execution, the left pair of h is copied to the right pair so that h holds four identical value of 10 for the continuation. In the other execution, the right pair is copied to the left. As such, both executions can proceed with full vectors. Eventually, the two executions report two possible outputs, -20 and 20, which precisely capture the possible output variations in the presence of any internal error or external error (e.g., error on input variable a).

Key Advances Over the State-of-The-Art. RAIVE has the following advantages over the state-of-the-art runtime insta-

	STATEMENTS	ACTUAL	IDEAL	TAG	RAIVE
1	$a = 2^{54}$;	r †	r	r^F	<r,r,r,r>
2	$b = 2^{54} + 1$;	r	r+1	r^F	<r,r,r,r>
3	c = b - a;	0	1	0^T	<-64,64,-64,64>
4	$d = c - 2^{37}$;	$-r_1$	$-r_1+1$	$-r_1^F$	<$-r_1$,$-r_1$,$-r_1$,$-r_1$>
5	if (d <= 0)	T	T	T^F (A)	<T,T,T,T> (D)
6	...;				
7	e = max(a,b);				
8	max(a,b): {				
9	t=b-a;	0	1	0^T	<-64,64,-64,64>
10	if (t <= 0)	T	F	T^T (B)	<T,F,T,F> (E)
11	e=a;	r		r^F	<r,r,-,->
12	else e=b;		r+1		<-,-,r,r>
	ret e;				e=<r,r,r,r> (F)
	}				
13	f = e – a;	0	1	0^T	<-64,64,-64,64>
14	g = f – 0.5;	-0.5	0.5	-0.5^F	<-64.5,63.5, -64.5, 63.5>
15	if (g > 0)	F	T	F^F (C)	<F,T,F,T> (G)
16	h=10;		10		<10,10,-,->
17	else h=-10	-10		-10^F	<-,-,-10,-10> fork
					(H) h=<10,10,10,10> h=<-10,-10,-10,-10>
18	o=h*2;	-20	20	-20^F	<20,20,20,20> <-20,-20,-20,-20>

† We use r to denote the large number 2^{54}, or 1801439 8509481984, and r_1 to denote 2^{37}. Other large values are represented relative to r and r_1.

Figure 3. (First column) example program with `max()` inlined; (second column) actual execution with each entry denoting actual computed value; (third column) ideal execution; (fourth column) TAG [2] execution; and (fifth column) RAIVE execution. r^F in the TAG approach means value r is tagged with a false error bit. The shaded sub-execution denotes the new execution after the user manually annotates the benign unstable predicate (line 10).

bility detector TAG [2]. TAG detects relative error inflation, i.e., instances when the relative error of the result of a floating point operation is substantially larger than those of the operands. It taints a variable when its relative error is inflated. It further monitors the propagation of the taint bit. The bit may be reset when a tainted operand is used in a binary operation with a much larger untainted operand. If a taint bit reaches a discrete factor, the execution is considered unstable and terminated. Re-execution with a higher precision is needed.

- *Capturing Output Variations Caused by Both Internal and External Errors.* TAG cannot detect output variations. Instead, when TAG detects an unstable discrete factor, it simply terminates the execution and switches to a higher precision. Furthermore, as we discussed in Section 1, if external errors are possible, even using the infinite representation precision cannot address the problem that the execution may produce different outputs due to the errors. In contrast, RAIVE does not terminate an unstable execution, but rather forks multiple executions to capture output variations. It handles both internal and external errors.

The third column in Fig. 3 shows the execution of TAG. In box Ⓐ, value 0 in c is tainted due to the error inflation at line 3. But the taint bit is reset at line 4. Later, TAG detects that a tainted value reaches a discrete factor in box Ⓑ and terminates. In contrast, RAIVE reports the possible output variations. Assume the value 2^{54} of a at line 1 is loaded from a file. Even if we used infinite precision in the execution, the same output variation would still occur if a has some small external error.

- *Handling Benign Differences.* TAG cannot automatically determine if an unstable factor is benign or harmful. It simply terminates when an unstable factor is detected. Or, the user can choose to manually annotate some discrete factors beforehand such that instability warnings at those factors are ignored. The predicate in box Ⓑ in Fig. 3 is unstable but benign. TAG cannot handle this. In contrast, RAIVE leverages a novel runtime that evaluates both branches to overcome the problem.

- *Avoiding Undesirable Error Suppression.* TAG uses a single bit to denote the presence of an inflated relative error. However, this may become problematic in error suppression. In box Ⓒ, the subtraction $e - a$ causes a relative error inflation. Because f is 0 at line 14, substantially smaller than 0.5, the error bit is reset. The predicate at line 15 is hence considered stable. This is problematic: as shown in the second and the third columns, the actual and ideal executions have different branch outcomes at line 15, leading to different final outputs.

In normal cases, absolute errors are much smaller than the actual values even when the relative errors are inflated. However at line 14, the absolute error of f is 1.0 (from the actual and ideal values), larger than the value of f itself (i.e. $f = 0$) and even the other operand 0.5. Unfortunately, this information cannot be represented by the single error bit.

In contrast, RAIVE injects artificial errors that denote the bounds of the absolute errors (box Ⓖ). It easily supports error propagation in which operand(s) with inflated error(s) lead to a result with inflated error, since the different values in the operand vectors often lead to different values in the result vector. Furthermore, error suppression can occur implicitly and appropriately during floating point operations. For example, in box Ⓓ, the errors are suppressed by the subtraction whereas in box Ⓖ, the operand 0.5 is not large enough to suppress the errors.

- *Lower Runtime Overhead by Vectorization.* Although TAG features much lower overhead compared to techniques based on high precision libraries [5] and affine analysis [10], it is still very expensive (827% overhead according to Section 5). This is due to the expensive instrumentation and the poor instruction pipeline performance. The left column of Fig. 4 shows part of the TAG instrumentation for the example in Fig. 3. Lines 1.1, 3.1-3.5, 18.1 denote instrumentation for lines 1, 3, and 18, respectively. Line 3.1 means that the result is tagged true if both operands are tagged (true). Lines 3.2-3.3 handle the case when only one operand is tagged. In this case, if the difference between the operand exponents ε_a and ε_b is larger than a threshold τ_s, the relative error is suppressed and the result tag c' is false. Line 3.5 detects relative error inflation. Instrumentation similar to lines 3.1-3.5 is added for each subtraction/addition. For multiplications and divisions, since neither inflation nor suppression could happen, the result tag is simply the union of the operand tags (e.g. line 18.1). The nesting branches in instrumentation lead to poor instruction pipeline performance. The fine-grained interleaving of the boolean type error bit propagation and the floating point type computation also prevents aggressive instruction scheduling, causing performance penalty.

RAIVE leverages the native support for vectors. In particular, each original floating point instruction is rewritten to a vector instruction. Note that the operations on individual vector values are performed simultaneously on separate FPUs such that they do not cost additional cycle(s). Such vectorization is shown in the right column in Fig. 4. At line 3, the original subtraction is replaced with a vector subtraction. In addition, we only need instrumentation for detecting error inflation and checking discrete factors. For example, the instrumentation for line 3 in RAIVE is much simpler than that in TAG. No instrumentation is needed for multiplications or divisions. This not only reduces the number of instructions, but also avoids interleavings of integer/boolean instructions and floating point instructions.

4. Design of RAIVE Runtime

RAIVE is a runtime technique. The given floating point program is transformed using the compiler. This transformed program has a special execution model that is supported via vectorization. The execution may fork multiple processes and produce a set of outputs that denote possible variations in the presence of errors. The execution model can be intuitively described as follows. The program state (for floating point variables) is a set of pairs, each representing an interval for the possible values of the variable (e.g., $\langle x_1, x_2\rangle$ for variable x). Initially, each pair has two identical values (e.g., $\langle x, x\rangle$). Floating point operations are performed on the pairs. RAIVE monitors these operations. When it detects relative error inflation, it introduces artificial errors to the pair so that it becomes $\langle x-\delta, x+\delta\rangle$. The introduced error δ over-approximates the error incurred by the operation and hence the pair denotes the lower and upper bounds of x at this operation. Upon encountering a conditional, if there exists a pair of values $\langle x_1, x_2\rangle$ for x such that one of the values (x_1 or x_2) satisfies the condition and the other does not, RAIVE executes both branch outcomes of the conditional. The states for the branch outcomes are managed separately. At the join point of the branch we get two different sets of (output) pairs – $\langle y_1^t, y_2^t\rangle$ for the true branch and $\langle y_1^f, y_2^f\rangle$ for the false branch. If these pairs agree, RAIVE joins them, knowing that there is no output variation induced by the branch deviation. If they do not agree, RAIVE separates the pairs (from the two branches) permanently via forking.

Observe that the aforementioned execution model requires maintaining a store for pairs and supporting non-interference when evaluating both branches of a conditional. In order to achieve these goals, we develop a vector based semantics that can be implemented using the latest vector instruction support. In particular, we use a vector of four values to denote each floating point variable in the original

program. Normally, the first two values (called the *left pair*) denote the interval of the variable value and the right pair is simply a duplication of the left pair. When both branches of a conditional need to be evaluated, the left and right pairs are used/updated in isolation to achieve non-interference.

4.1 Semantics

$$
\begin{array}{lll}
\textit{Program } P & ::= & s \\
\textit{Stmt } s & ::= & s_1;\ s_2 \mid \textbf{skip} \mid x := e \mid x := \textbf{f2i}(y) \mid \\
& & \textbf{if } x \bowtie y \textbf{ then } s_1 \textbf{ else } s_2 \mid \\
& & \textbf{while } x \bowtie y \textbf{ do } s \\
\textit{Expr } e & ::= & x \mid v \mid e_1\ op\ e_2 \mid \textbf{sin}(e) \\
\textit{BinOp } op & ::= & + \mid - \mid * \mid / \\
\textit{Value } v, w & ::= & n \mid r \mid b
\end{array}
$$

$$Var\ x, y \in Identifier \quad n \in \mathbb{Z} \quad r \in Real \quad b \in Boolean$$

Figure 5. Language.

We use a language in Fig. 5 to facilitate discussion. We model two kinds of discrete factors: **f2i** denoting a type cast from floating point to integer and $\bowtie$ denoting a relational operation on two floating point variables. Mathematical functions are modeled by a representative function $\textbf{sin}(e)$.

The semantics is presented in Fig. 6. The related definitions are presented close to the top. In particular, the store σ is a mapping from a variable to a vector of four values. It is constituted by two disjoint stores, σ_l and σ_r, denoting the mappings from a variable to the first two values (i.e., *left pair*), and to the last two values (i.e., *right pair*), respectively. The execution mode is denoted by ω, which has three possible values: $REGULAR$ denoting *regular execution* in which both the left and right stores are updated, $LEFT$ denoting *left execution* in which only the left store is updated, and $RIGHT$ denoting *right execution* in which only the right store is updated. When an unstable predicate is encountered, that is, the values in the vectors (involved in the predicate) yield non-uniform branch outcomes, RAIVE needs to determine if the instability is benign by executing both branches in sequence. The executions of the two branches need to be isolated so that they do not interfere with each other and their results can be properly compared at the join point. In particular, the first branch execution only operates on the left pairs, called the *left mode*, whereas the second branch execution only operates on the right pairs, called the *right mode*. Execution modes and mode changes are implemented using the *mask* instruction provided by the CPU. The instruction defines which values in a vector are visible and operatable. The variable join set Φ contains the variables defined during the branch executions of an unstable predicate. At the branch join point, Φ is scanned to determine if forking is necessary.

To make presentation easier, we extend the syntax of expression to represent a vector of values, and the syntax of statement to include a few new commands: **mode_switch** to switch the current execution mode; **join** is to commit the updates from the two branches of a predicate and determine if the execution should be forked; **spawn** is to fork an execution.These statements are auxiliary and only present during evaluation.

4.1.1 Expression Rules

Expression rules evaluate a floating point expression to a vector of four values. The evaluation may be moderated by the execution mode. During expression evaluation, relative error inflation is also detected. Rule [CONST] shows that a floating point value v is expanded to a vector of four identical values.

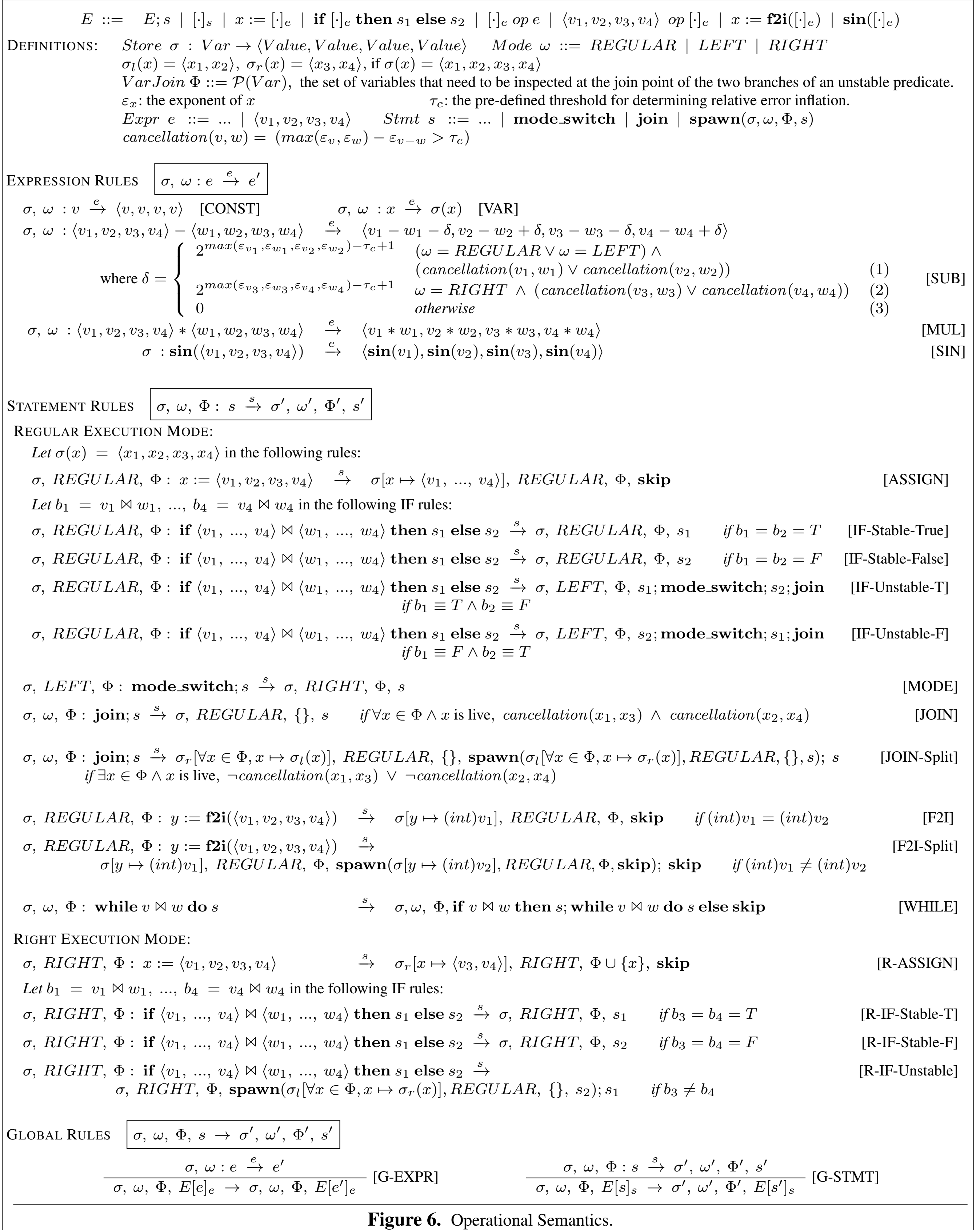
Subtraction of two vectors (Rule [SUB]) is performed by subtracting the corresponding values and adjusting the resulting values with δ, which is computed as follows: (1) if the current execution mode is $REGULAR$ or $LEFT$ (for branch execution using the left store) and there is relative error inflation (or cancellation) in the subtraction of the actual values v_1 and w_1, or v_2 and w_2, δ is computed from the exponents by $\delta = 2^{max(\varepsilon_{v_1}, \varepsilon_{w_1}, \varepsilon_{v_2}, \varepsilon_{w_2}) - \tau_c + 1}$. Intuitively, τ_c is the threshold to detect inflation, meaning that an addition/-subtraction causes relative error inflation if the result is left-shifted by at least τ_c bits, suggesting the first τ_c significand bits of the two operands are identical. Since we consider that the result value cannot be trusted in this case, it is equivalent to that the absolute error of the result can be as large as the value represented by the τ_c-*th* significand bit of the largest operand (i.e. the first bit that differs in the operands), which can be computed by the aforementioned formula. (2) If the current mode is $RIGHT$, the values in the right pairs are used to detect inflation and compute δ. (3) If there is no inflation, $\delta = 0$. Additions are handled in the same way. The first line in box Ⓓ in Fig. 3 shows an example of condition (1), with $\tau_c = 49$.

In Rule [MULT], the values in the operand vectors are multiplied respectively, denoting the propagation of errors, if injected previously. The rule for division is similar. Note that although multiplication or division can enlarge absolute errors, they do not inflate relative errors. Intuitively, when an operand x with an absolute error $\hat{\Delta}_x$ is multiplied with another operand y, both x and $\hat{\Delta}_x$ are enlarged by y so that the resulting relative error (i.e., the ratio between the absolute error of result and the actual result) is unchanged.

For non-library function calls, parameter vectors are directly passed to callees and used there. In contrast, an external library function is generally evaluated on the respective vector values (Rule [SIN]). We cannot vectorize library functions as we do not have their source code. For better efficiency, we re-implement some frequently used library functions to directly support vectorization.

4.1.2 Statement Rules

RAIVE has three execution modes. Statement semantics may be different in these modes.



$$E \ ::= \ E; s \mid [\cdot]_s \mid x := [\cdot]_e \mid \textbf{if } [\cdot]_e \textbf{ then } s_1 \textbf{ else } s_2 \mid [\cdot]_e \ op \ e \mid \langle v_1, v_2, v_3, v_4\rangle \ op \ [\cdot]_e \mid x := \textbf{f2i}([\cdot]_e) \mid \textbf{sin}([\cdot]_e)$$

DEFINITIONS: $Store\ \sigma : Var \rightarrow \langle Value, Value, Value, Value\rangle$ $\quad Mode\ \omega ::= REGULAR \mid LEFT \mid RIGHT$

$\sigma_l(x) = \langle x_1, x_2\rangle,\ \sigma_r(x) = \langle x_3, x_4\rangle$, if $\sigma(x) = \langle x_1, x_2, x_3, x_4\rangle$

$VarJoin\ \Phi ::= \mathcal{P}(Var)$, the set of variables that need to be inspected at the join point of the two branches of an unstable predicate.

ε_x: the exponent of x $\quad \tau_c$: the pre-defined threshold for determining relative error inflation.

$Expr\ e ::= \ldots \mid \langle v_1, v_2, v_3, v_4\rangle$ $\quad Stmt\ s ::= \ldots \mid \textbf{mode_switch} \mid \textbf{join} \mid \textbf{spawn}(\sigma, \omega, \Phi, s)$

$cancellation(v, w) = (max(\varepsilon_v, \varepsilon_w) - \varepsilon_{v-w} > \tau_c)$

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

$\sigma, \omega : v \xrightarrow{e} \langle v, v, v, v\rangle$ [CONST] $\qquad \sigma, \omega : x \xrightarrow{e} \sigma(x)$ [VAR]

$$\sigma, \omega : \langle v_1, v_2, v_3, v_4\rangle - \langle w_1, w_2, w_3, w_4\rangle \xrightarrow{e} \langle v_1 - w_1 - \delta, v_2 - w_2 + \delta, v_3 - w_3 - \delta, v_4 - w_4 + \delta\rangle$$

$$\text{where } \delta = \begin{cases} 2^{max(\varepsilon_{v_1}, \varepsilon_{w_1}, \varepsilon_{v_2}, \varepsilon_{w_2}) - \tau_c + 1} & (\omega = REGULAR \vee \omega = LEFT) \wedge (cancellation(v_1, w_1) \vee cancellation(v_2, w_2)) \quad (1) \\ 2^{max(\varepsilon_{v_3}, \varepsilon_{w_3}, \varepsilon_{v_4}, \varepsilon_{w_4}) - \tau_c + 1} & \omega = RIGHT \wedge (cancellation(v_3, w_3) \vee cancellation(v_4, w_4)) \quad (2) \\ 0 & otherwise \quad (3) \end{cases} \quad \text{[SUB]}$$

$\sigma, \omega : \langle v_1, v_2, v_3, v_4\rangle * \langle w_1, w_2, w_3, w_4\rangle \xrightarrow{e} \langle v_1 * w_1, v_2 * w_2, v_3 * w_3, v_4 * w_4\rangle$ [MUL]

$\sigma : \textbf{sin}(\langle v_1, v_2, v_3, v_4\rangle) \xrightarrow{e} \langle \textbf{sin}(v_1), \textbf{sin}(v_2), \textbf{sin}(v_3), \textbf{sin}(v_4)\rangle$ [SIN]

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

REGULAR EXECUTION MODE:

Let $\sigma(x) = \langle x_1, x_2, x_3, x_4\rangle$ in the following rules:

$\sigma, REGULAR, \Phi : x := \langle v_1, v_2, v_3, v_4\rangle \xrightarrow{s} \sigma[x \mapsto \langle v_1, \ldots, v_4\rangle], REGULAR, \Phi, \textbf{skip}$ [ASSIGN]

Let $b_1 = v_1 \bowtie w_1, \ldots, b_4 = v_4 \bowtie w_4$ in the following IF rules:

$\sigma, REGULAR, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, REGULAR, \Phi, s_1$ $\quad$ *if* $b_1 = b_2 = T$ [IF-Stable-True]

$\sigma, REGULAR, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, REGULAR, \Phi, s_2$ $\quad$ *if* $b_1 = b_2 = F$ [IF-Stable-False]

$\sigma, REGULAR, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, LEFT, \Phi, s_1; \textbf{mode_switch}; s_2; \textbf{join}$ [IF-Unstable-T]
if $b_1 \equiv T \wedge b_2 \equiv F$

$\sigma, REGULAR, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, LEFT, \Phi, s_2; \textbf{mode_switch}; s_1; \textbf{join}$ [IF-Unstable-F]
if $b_1 \equiv F \wedge b_2 \equiv T$

$\sigma, LEFT, \Phi : \textbf{mode_switch}; s \xrightarrow{s} \sigma, RIGHT, \Phi, s$ [MODE]

$\sigma, \omega, \Phi : \textbf{join}; s \xrightarrow{s} \sigma, REGULAR, \{\}, s$ $\quad$ *if* $\forall x \in \Phi \wedge x$ is live, $cancellation(x_1, x_3) \wedge cancellation(x_2, x_4)$ [JOIN]

$\sigma, \omega, \Phi : \textbf{join}; s \xrightarrow{s} \sigma_r[\forall x \in \Phi, x \mapsto \sigma_l(x)], REGULAR, \{\}, \textbf{spawn}(\sigma_l[\forall x \in \Phi, x \mapsto \sigma_r(x)], REGULAR, \{\}, s); s$ [JOIN-Split]
if $\exists x \in \Phi \wedge x$ is live, $\neg cancellation(x_1, x_3) \vee \neg cancellation(x_2, x_4)$

$\sigma, REGULAR, \Phi : y := \textbf{f2i}(\langle v_1, v_2, v_3, v_4\rangle) \xrightarrow{s} \sigma[y \mapsto (int)v_1], REGULAR, \Phi, \textbf{skip}$ $\quad$ *if* $(int)v_1 = (int)v_2$ [F2I]

$\sigma, REGULAR, \Phi : y := \textbf{f2i}(\langle v_1, v_2, v_3, v_4\rangle) \xrightarrow{s}$ [F2I-Split]
$\sigma[y \mapsto (int)v_1], REGULAR, \Phi, \textbf{spawn}(\sigma[y \mapsto (int)v_2], REGULAR, \Phi, \textbf{skip}); \textbf{skip}$ $\quad$ *if* $(int)v_1 \neq (int)v_2$

$\sigma, \omega, \Phi : \textbf{while } v \bowtie w \textbf{ do } s \xrightarrow{s} \sigma, \omega, \Phi, \textbf{if } v \bowtie w \textbf{ then } s; \textbf{while } v \bowtie w \textbf{ do } s \textbf{ else skip}$ [WHILE]

RIGHT EXECUTION MODE:

$\sigma, RIGHT, \Phi : x := \langle v_1, v_2, v_3, v_4\rangle \xrightarrow{s} \sigma_r[x \mapsto \langle v_3, v_4\rangle], RIGHT, \Phi \cup \{x\}, \textbf{skip}$ [R-ASSIGN]

Let $b_1 = v_1 \bowtie w_1, \ldots, b_4 = v_4 \bowtie w_4$ in the following IF rules:

$\sigma, RIGHT, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, RIGHT, \Phi, s_1$ $\quad$ *if* $b_3 = b_4 = T$ [R-IF-Stable-T]

$\sigma, RIGHT, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \sigma, RIGHT, \Phi, s_2$ $\quad$ *if* $b_3 = b_4 = F$ [R-IF-Stable-F]

$\sigma, RIGHT, \Phi : \textbf{if } \langle v_1, \ldots, v_4\rangle \bowtie \langle w_1, \ldots, w_4\rangle \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s}$ [R-IF-Unstable]
$\sigma, RIGHT, \Phi, \textbf{spawn}(\sigma_l[\forall x \in \Phi, x \mapsto \sigma_r(x)], REGULAR, \{\}, s_2); s_1$ $\quad$ *if* $b_3 \neq b_4$

GLOBAL RULES $\boxed{\sigma, \omega, \Phi, s \rightarrow \sigma', \omega', \Phi', s'}$

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

Figure 6. Operational Semantics.

Regular Mode. The first set of statement rules is for the regular execution mode. Rule [ASSIGN] describes that all the four fields of the left hand side variable are updated.

Most of the complexity of RAIVE lies in the handling of conditional statements. It first determines if a predicate is stable. If so, the execution proceeds with the uniform branch outcome. Otherwise, it executes the two branches in sequence to detect if the instability is benign. If not, RAIVE forks the execution to capture the different effects of the instability.

Rules [IF-Stable-True] and [IF-Stable-False] describe that during regular execution, if the left pair has the identical true/false value, the execution proceeds to the true/false branch. Note that only the left pair needs to be inspected as the right pair is a duplication during regular execution.

Rules [IF-Unstable-T] and [IF-Unstable-F] specify the semantics when the left pair does not concur. If the first value b_1 is false (Rule [IF-Unstable-F]), it first executes the false branch in the left mode, and then executes the $\mathbf{mode_switch}$ statement to change to the right mode for true branch execution (Rule [MODE]). After executing the two branches, the updates in them are inspected by the $\mathbf{join}$ statement.

Rules [JOIN] and [JOIN-Split] specify the semantics of the $\mathbf{join}$ statement. In Rule [JOIN], the left and right pairs of all live updated variables are identical or have trivial differences, suggesting benign instability. In particular, RAIVE inspects each *live* variable in Φ, by comparing its left and right pairs. If the comparison of two values incurs cancellation (i.e. the number of cancelled bits is larger than the threshold τ_c), we consider the two values under comparison have trivial difference. If the differences are always trivial for all live variables, the instability is benign, the execution is not forked. During inspection, only variables that are live at the join point of the branches are considered (i.e. those that may be used beyond the join point). We use a standard static live variable analysis. After joining, Φ is reset. In Rule [JOIN-Split], if the differences are not trivial, the execution is split. In the parent process, the updates during right execution are discarded, by overwriting the right store values with the left store values. In the child process, the updates during left execution are discarded, by overwriting the left store values with the right store values.

Rules [F2I] and [F2I-Split] specify the semantics of type casts from floating point to integer in the regular mode. If the left pair yields different integer values, the execution forks based on the different discrete integer values.

The evaluation of while loops (Rule [WHILE]) is standard, which unrolls the loop once each time. Since loops are essentially unrolled during evaluation, unstable loop predicates are handled like normal predicates. To prevent potential infinite forking, we limit the number of forks allowed for a loop predicate (to 10). In practice, such a limit is never reached. But if the limit is reached simply continue with one of the executions.

Example. Box Ⓗ in Fig. 3 shows an example of Rule [JOIN-Split]. At the join point, $\Phi = \{h\}$ and the differences between the left and right pairs of h are substantial and the execution is forked. In the continuation of the original execution, $h^v = \langle 10, 10, 10, 10\rangle$ after copying the left pair to the right, whereas in the spawned execution, $h^v = \langle -10, -10, -10, -10\rangle$ after copying the right to the left. □

Right Mode. The next set of rules is for the right execution mode. According to Rule [ASSIGN-Right], in right execution, only the right pairs are updated, while the left pairs retain their values. Moreover, the left-hand-side variable x is inserted to the variable set Φ for inspection at the join point.

Rules [R-IF-Stable-T], [R-IF-Stable-F], and [R-IF-Unstable] evaluate conditional statements in the right mode. Rule [R-IF-Unstable] specifies the case in which another unstable predicate is encountered, which suggests nesting unstable predicates. Since we use only the right pairs in the right mode, we cannot afford evaluating the two branches of the inner unstable predicate in sequence. Therefore, we fork the execution right away. Particularly, the original execution proceeds with the true branch (of the inner predicate) with the same right mode. The spawned execution proceeds with the false branch in the regular mode, discarding all the updates during the former branch evaluation (in the left mode). As such, the $\mathbf{join}$ operation at the join point of the outer unstable predicate has no effect. The left mode rules are similar and hence omitted.

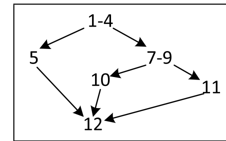


Figure 8. The control flow graph for the example in Fig. 7.

Example. Fig. 7 shows an example for nesting unstable predicates. The first column shows the program. The next three columns show its original execution. The last three columns show the spawned execution. Initially, a has a large value r and the input to b is $r + 1$. However due to the limited precision, the represented value in b is r. At line 3, the actual value of c is 0, and artificial errors are introduced due to the relative error inflation. Since line 4 is unstable, line 5 is executed in the left mode and Φ contains t. The execution is switched to the right mode at line 6. Another inflation is detected at line 8 so that errors are introduced to the third and fourth values, leading to non-uniform branch outcome. The execution is forked according to Rule [R-IF-Unstable]. Note that the updates on the left pairs of all the variables in Φ are discarded and replaced with the right pairs in the

Program	Exec. I			Exec. II		
	σ	ω	Φ	σ	ω	Φ
1 $a = r;$	$\sigma[a] = \langle r, r, r, r \rangle$	REGULAR	$\{\}$			
2 $b = input();$	$\sigma[b] = \langle r, r, r, r \rangle$	REGULAR	$\{\}$			
3 $c = a - b;$	$\sigma[c] = \langle -\delta, \delta, -\delta, \delta \rangle$	REGULAR	$\{\}$			
4 **if** $(c < 0)$	$\langle T, F, T, F \rangle$	LEFT	$\{\}$			
5 $t = a + 6;$	$\sigma[t] = \langle r + 6, r + 6, -, - \rangle$	LEFT	$\{t\}$			
6 **else** {		RIGHT	$\{t\}$			
7 $c = b + 2;$	$\sigma[c] = \langle -, -, r + 2, r + 2 \rangle$	RIGHT	$\{t, c\}$			
8 $d = c - a;$	$\sigma[d] = \langle -, -, 2 - \delta, 2 + \delta \rangle$	RIGHT	$\{t, c, d\}$			
9 **if** $(d < 0)$	$\langle -, -, T, F \rangle$	RIGHT	$\{t, c, d\}$	$\sigma[c] = \langle r + 2, ..., r + 2 \rangle$ $\sigma[d] = \langle 2 - \delta, 2 + \delta, 2 - \delta, 2 + \delta \rangle$ $\sigma[t] = \langle 0, 0, 0, 0 \rangle$	REGULAR	$\{\}$
10 $t = c + 6;$	$\sigma[t] = \langle -, -, r + 8, r + 8 \rangle$	RIGHT	$\{t, c, d\}$			
11 **else** $t = a * 2;$				$\sigma[t] = \langle 2r, 2r, 2r, 2r \rangle$	REGULAR	$\{\}$
}	$\sigma[t] = \langle r + 6, r + 6, r + 8, r + 8 \rangle$	REGULAR	$\{\}$			
12 $o = t + a;$	$\sigma[o] = \langle 2r + 6, 2r + 6, 2r + 8, 2r + 8 \rangle$	REGULAR	$\{\}$	$\sigma[o] = \langle 3r, 3r, 3r, 3r \rangle$	REGULAR	$\{\}$

Figure 7. An example for nesting unstable predicates. Symbol r represents a large floating point value. Assume the input value is $r + 1$ which cannot be precisely represented and hence the represented value is r at line 2. Other large values can be precisely represented.

spawned execution, which proceeds with the regular mode and an empty Φ. The original execution continues with the right mode and the join operation is performed before line 12. Since variables c and d are not live beyond line 12, only t's vector values are compared. Since the differences are trivial, the execution is not forked.

As in Fig. 8, RAIVE captures the effects of the leftmost path (1-5→12) and the middle path (1-4→7-10→12) through the original execution, and the effect of the rightmost path (1-4→7-9→11-12) through the spawned execution. □

4.2 Understanding the Essence of RAIVE

Next, we informally discuss a few properties of RAIVE.

First, the goal of RAIVE is to expose output variations caused by discrete differences. Such differences are caused by discrete factors (i.e. operations that have floating point operands and discrete type result such as integer or boolean) instead of floating point operations. Hence, RAIVE is most suitable for programs with both floating point and discrete operations (e.g., the example in Section 1), and less effective for mathematical cores composed of floating point operations only (e.g., a code snippet that computes $y = x^2 + 2x + 3$). The real world floating point programs we study contain non-trivial number of discrete operations. As such, output variations are mainly due to discrete differences (caused by errors). Our experiments will illustrate this later. Intuitively, errors through floating point operations cause continuous output changes, whereas errors through discrete operations may cause discontinuous differences that are usually much more substantial.

Second, RAIVE introduces artificial errors that are bounds of the absolute error when relative error inflation occurs. The resulting value differences in the vector are further propagated via the following vectorized floating point operations. However, the values in the vector of a variable are not guaranteed to stay as the bounds of the variable value, as during binary operations errors from multiple operands may interfere with each other. In other words, at the moment when the artificial errors are introduced, the values in the vector denote the lower and upper bounds of the actual value. However, when these values (with errors) are propagated to other variables through operations, especially binary operations, the values in the vectors of result variables may not represent the bounds. A very important point, however, is that RAIVE *does not need to guarantee these values to be the bounds*. Instead, the (different) values in a vector are essentially samples in the error range. *These samples are sufficiently distant so that they expose discrete differences*. This is because the artificial errors RAIVE introduces are very conservative. In contrast, affine analysis [10] focuses on modeling *continuous* changes caused by errors through affine formulas. Hence, they need to compute the bounds of values, which is very expensive (i.e. 3-4 magnitude of slowdown according to [10]).

Note that an approach that tries to use multiple sample runs to expose output variations is inferior to RAIVE. This is because RAIVE essentially not only packs multiple sample executions into a vectorized execution, but also avoids unnecessary samples by detecting benign unstable predicates. According to our experiment (Section 5), while an execution may encounter many unstable predicates, most of them are found to be benign.

Third, similar to existing work [2, 22], RAIVE uses a threshold to detect relative error inflation, which is key to detecting unstable discrete factors. In theory, the detection is neither sound nor complete due to the use of threshold. However, RAIVE does not aim to detect unstable discrete factors, but rather expose output variations in the presence of errors. It has a sophisticated runtime mechanism to determine if an unstable predicate is benign. Therefore, we use a conservative threshold for relative error inflation detection. The resulting false positives of unstable predicates are effectively suppressed by the runtime mechanism. Furthermore,

as shown in Section 5, all inputs falling into the unstable range tend to cause the same discrete differences and hence the same (or highly similar) output variations. The net effect of having false positives in detecting unstable factors is to report the same output variations for a larger set of inputs. We also show in Section 5 that the unstable ranges are very tiny such that even though RAIVE reports output variations for a larger range, such output variations are well possible because they can be easily induced by external errors.

5. Evaluation

We implement RAIVE using GCC-4.7.2. to support C/C++ and Fortran. We leverage the *Advanced Vector Extensions* (AVX) on x86_64 architecture to support vectors. It features instructions operating on 256-bit vector registers, called `ymm` registers, which can store up to four double precision floating point values. The execution mask in our semantics is also natively supported by the CPU. We modify the lexical analysis of GCC. Each floating point variable is replaced with a 256-bit type supported by GCC. Floating point operations are also replaced accordingly. After lexical analysis, we further instrument the GIMPLE IR code to support functionalities such as error inflation detection and the execution modes.

Since the Fortran frontend of GCC does not support AVX natively, some language features are difficult to vectorize. One example is the primitive complex type in Fortran. Since a double precision complex value consists of two 64-bit values (the real and the imag parts), a complex value cannot be directly transformed to a vector. Hence we transform a complex value to a C struct that consists of two 256-bit vectors. This transformation is more challenging than vectorizing scalar floating point variables because we need to further replace the operations on complex values with operations on the C struct values. RAIVE handles all the language features that we have encountered in the benchmark set.

We evaluate efficiency and effectiveness of RAIVE and compare it with HPL [2, 5] and TAG [2]. HPL uses 128-bit quadruple precision. For fair comparison, we re-implemented HPL using GCC. Our implementation is faster than [5]. We use the programs in [2] for comparison, including SPEC CFP2000 and a biochemical data processing program `deisotope`. In addition, we include two widely used data-mining programs `k-means` and `pagerank`. Note that these programs are much more complex than those used in studies that focus on numerical programs (e.g., [10]). They contain a lot of discrete operations. Their LOCs are shown in Table 1 column 2. All experiments were run on a machine with Intel i7-2640M 2.80GHz processor and 8GB RAM.

5.1 Performance

In the first experiment, we evaluate the runtime overhead of RAIVE. We use the reference inputs from SPEC. For `deisotope`, `k-means` and `pagerank`, we use the inputs that come with the programs. The results are shown in Table 1. Column 3 shows the native execution time. Columns 4 and 5 present the overhead for HPL and TAG. Observe that the average overhead of HPL exceeds 98x. The average overhead for TAG is 827%.

Program	LOC	Native time(s)	HPL o/h	TAG o/h	Vec-only		RAIVE	
					time(s)	o/h	time(s)	o/h
168.wupwise	2.1k	79.70	6043%	292%	141.6	178%	179.5	225%
171.swim	0.4k	122.5	7356%	359%	163.7	133%	178.5	146%
172.mgrid	0.4k	39.27	35031%	1639%	89.9	228%	355.7	905%
173.applu	4k	39.91	21112%	1458%	108.6	272%	163.9	410%
177.mesa	63k	8.60	3364%	538%	19.97	232%	27.63	321%
178.galgel	15.3k	28.26	23867%	1592%	124.5	441%	209.2	740%
179.art	1.2k	10.08	15786%	735%	21.13	210%	28.00	277%
183.equake	1.3k	12.55	21876%	1525%	52.37	417%	65.28	520%
187.facerec	2.4k	35.72	10784%	1492%	145.8	408%	180.8	506%
188.ammp	13.4k	53.02	16263%	822%	78.76	148%	160.6	303%
189.lucas	3k	24.24	23536%	1333%	74.52	307%	87.56	361%
191.fma3d	60k	38.33	13169%	1110%	142.9	373%	155.9	406%
200.sixtrack	47.2k	59.50	47540%	1056%	95.34	160%	214.6	360%
301.apsi	7.5k	51.47	13220%	719%	104.6	203%	166.1	322%
deisotope	2.2k	11.82	469%	205%	15.01	127%	18.69	158%
k-means	7k	12.69	925%	329%	14.12	111%	16.23	127%
pagerank	0.25k	13.29	5491%	1653%	41.17	309%	66.83	502%
AVERAGE			9826%	827%		229%		340%

Table 1. Performance (o/h stands for overhead). AVERAGE is geometric mean.

The last two columns present the time and overhead of RAIVE. We collect the data with $\tau_c = 48$, which is the threshold used in detecting relative error inflation (Section 2). The average overhead is 340%, which is 2.43 times smaller than that in the TAG approach. The higher overhead in some of the programs (e.g., `178.mgrid`) is due to the exceptionally large number of additions and subtractions in the hot loops. These operations have to be instrumented for error inflation detection.

We further study the breakdown of overhead for RAIVE. We run the programs with vector instructions but without detecting instability. It means that a program is transformed to its vector version, where floating point values are stored in 256-bit vectors and operated with AVX instructions. The four values in a vector are always identical. This is to study the overhead of vectorization. The results are shown in the `vec-only` columns. The average overhead is 229%. While theoretically AVX instructions should not cost additional cycles, there are a few possible reasons for the slow-down. First, the processor we use (CPUID: 06_2AH) is an early version supporting AVX. According to the Intel Manual, the latencies for AVX in our processor are higher than later versions [20]. Second, we suspect the compiler is not able to perform aggressive optimizations for AVX instructions because they are relatively new. We anticipate RAIVE will have lower overhead in the future with new hardware and better compiler support. We argue that the overhead is acceptable given the capability of reporting output variations. Note that without our technique, achieving the same capability may entail a large number of sample executions, especially when the input dimension is high.

5.2 Effectiveness

Instability Detection. The experiment is setup as follows. For each program, we collect 1E+14 samples within an input range. We execute the program on these samples with HPL, TAG, and RAIVE. We extend HPL to also compute the actual floating point values with 64-bit precision such that they can be compared with their high precision version to collect the ground truth (of instability). For TAG and RAIVE, we collect data for three configurations with different τ_c values. The results are shown in Table 2. We only focus on the programs with instability reported. For each program, the input sample range and the total number of samples are shown in the first row labeled OVERALL. The six following rows present the detection results for the configurations. The second column shows the configuration. The third column shows the number of samples in which instability is reported (i.e. at least one unstable discrete factor has been encountered), and its percentage over the total sample number is presented in the fourth column. The last column shows the detected problematic range that contains the unstable samples.

We have the following observations. (1) RAIVE has comparable or better effectiveness than TAG in instability detection. The detected ranges by RAIVE are similar to or smaller than those by TAG for the same configuration. Both of them can correctly determine that over 99.99% of the inputs lead to stable executions. However, the overhead of RAIVE is 2.43x smaller than TAG. (2) Threshold $\tau_c = 52$ is the best configuration for most benchmarks considered. It reports the smallest number of unstable samples without any false negatives except for `pagerank`. The maximum possible threshold value is 53. Threshold $\tau_c = 44$ is safe (for the programs we considered) as it does not cause any false negatives. Note that using a larger τ_c means that we have a stricter condition in determining relative error inflation. (3) With $\tau_c = 44$, although the number of detected unstable samples is 3.75-2258 times larger than the ground truth, these samples only denote a trivial part of the input range. This implies it is unlikely for RAIVE to have false warnings in instability detection. In contrast, according to [2], interval analysis and techniques based on solely detecting error inflations report a lot more false positives. More importantly, as we will show later, the false positives in detecting instability have little effects on the main results, output variations.

Handling Benign Unstable Predicates and Forking. An important advantage of RAIVE is the capability of handling benign unstable predicates. The results are in Columns 2-4 in Table 3. Column 2 lists the average number of unstable predicates encountered in *a single sample execution*. Note that column 3 in Table 2 shows *the number of sample runs* in which unstable factors were detected. They have different meanings. Column 3 in Table 3 shows how often RAIVE can proceed without forking after executing the two branches separately. Column 4 shows the number of forks. Observe that 2 of the 6 programs (with instability detected) encounter benign unstable predicates. If these predicates were not properly handled, there would be a lot of unnecessary forks. Also observe that since most unstable predicates are benign, the number of forks is very small. Intuitively, it is unlikely for a program to have multiple sources of instability for *a given input*.

Output Variations. The experiment is set up as follows. We first identify the input range that is reported as unstable by HPL. Then we collect two samples at the boundary of the range, denoted as lb and ub, and use the output variations between the two executions as the ground truth as they denote the two boundary *stable* executions. Observe that the range is mostly very small such that external errors can easily cause input variations in the range. Then we execute the program on RAIVE for all the unstable samples (with $\tau_c = 52$) and collect the output variations for each sample. We then compute the output coverage for a sample i as follows. Let $O_1, ..., O_n$ be the n output variables and $O_t(ub)$ the value of a t'th variable in the ub sample. Since an execution in RAIVE may fork, we use $range(O_t(i))$ to denote the range of O_t for the ith sample.

$$recall = \left(\sum_{t=1}^{n} \frac{range(O_t(i)) \cap [O_t(lb),\, O_t(ub)]}{|O_t(ub) - O_t(lb)|}\right) / \, n$$

$$precision = \left(\sum_{t=1}^{n} \frac{range(O_t(i)) \cap [O_t(lb),\, O_t(ub)]}{range(O_t(i))}\right) / \, n$$

$Recall$ denotes how much ground truth output variation is covered by RAIVE; $precision$ represents how much output variation reported by RAIVE denotes true variation.

The results for $\tau_c = 52$ are in columns 5-7 in Table 3. Column 5 shows the number of output variables. The last two columns show the average precision and recall over all samples. Observe that RAIVE has close to 100% precision and recall for most cases, meaning that *any sample* within the range can precisely predict the same output variations in the presence of (both internal and external) errors. This is because the output variations caused by discrete differences are much more substantial than those by continuous numerical operations and RAIVE can precisely simulate discrete differences caused by errors. Since the discrete differences are stable within the range, the output variations are mostly stable across sample runs too. The precision and recall are not 100% in some cases because of the continuous differences. `Galgel` has the lowest recall as its continuous differences are non-trivial compared to the discrete differences.

The last column shows the maximum relative standard deviation (i.e., standard deviation divided by mean) of the output values for the same output variable. We only present the maximum as there are outputs whose values are almost identical across all the forked runs as they are not affected by the path differences. Observe that there are substantial output variations. The RSD for `galgel` cannot be computed as many of its forked executions do not produce any output. Note that existing techniques focusing on numerical behav-

	approach	# of cases	%	detected range
equake	OVERALL	1E+14		[0.8650, 0.8750]
	HPL	2	2.00E-12%	[0.8690799016130847, 0.8690799016130848]
	TAG(τ_c=44)	4516	4.52E-09%	[0.869079901**6026333**, 0.8690799016130848]
	TAG(τ_c=48)	279	2.79E-10%	[0.869079901613**0570**, 0.8690799016130848]
	TAG(τ_c=50)	20	2.00E-11%	[0.869079901613**0829**, 0.8690799016130848]
	RAIVE(τ_c=44)	4516	4.52E-09%	[0.86907990161**28591**, 0.86907990161**33106**]
	RAIVE(τ_c=48)	280	2.80E-10%	[0.869079901613**0709**, 0.869079901613**0988**]
	RAIVE(τ_c=52)	20	2.00E-11%	[0.869079901613**0829**, 0.8690799016130848]
facerec	OVERALL	1E+14		[0.6694, 0.6695]
	HPL	30700	3.07E-08%	[0.6694295316764218, 0.6694295316764524]
	TAG(τ_c=44)	37267458	3.73E-05%	[0.669429531**6577891**, 0.669429531**6950752**]
	TAG(τ_c=48)	2314523	2.31E-06%	[0.66942953167**51556**, 0.66942953167**77159**]
	TAG(τ_c=52)	162943	1.63E-07%	[0.669429531676**2312**, 0.669429531676**6320**]
	RAIVE(τ_c=44)	115423	1.15E-07%	[0.669429531676**3000**, 0.669429531676**6643**]
	RAIVE(τ_c=48)	70847	7.08E-08 %	[0.669429531676**2782**, 0.669429531676**6333**]
	RAIVE(τ_c=52)	55712	5.57E-08%	[0.669429531676**2792**, 0.669429531676**5709**]
galgel	OVERALL	1E+14		[0.8184, 0.8185]
	HPL	57695	5.77E-08%	[0.8184459012000007, 0.8184459012253359]
	TAG(τ_c=44)	37972131	3.80E-05%	[0.818445**8998299998**, 0.81844590**39575860**]
	TAG(τ_c=48)	3728089	3.28E-06%	[0.81844590**02196792**, 0.81844590**20723309**]
	TAG(τ_c=52)	1233455	1.23E-06%	[0.818445901**9084903**, 0.81844590**20723309**]
	RAIVE(τ_c=44)	43930919	4.39E-05%	[0.81844**37893753897**, 0.81844**81724703180**]
	RAIVE(τ_c=48)	3258832	3.26E-06%	[0.818445901**1753019** , 0.8184459**399554056**]
	RAIVE(τ_c=52)	229573	2.30E-07%	[0.818445901**1900151**, 0.81844590**14121039**]
deisotope	OVERALL	1E+14		[1.11, 1.12]
	HPL	2	2.00E-12%	[1.1156381266106556, 1.1156381266106557]
	TAG(τ_c=44)	653	6.53E-10%	[1.115638126610**5905**, 1.1156381266106557]
	TAG(τ_c=48)	40	4.00E-11%	[1.11563812661065**18**, 1.1156381266106557]
	TAG(τ_c=52)	5	5.00E-12%	[1.115638126610655**3**, 1.1156381266106557]
	RAIVE(τ_c=44)	315	3.15E-10%	[1.1156381266106**398**, 1.1156381266106**712**]
	RAIVE(τ_c=48)	22	2.20E-11%	[1.11563812661065**45** , 1.11563812661065**66**]
	RAIVE(τ_c=52)	2	2.00E-12%	[1.1156381266106556, 1.1156381266106557]
k-means	OVERALL	1E+14		[0.5640, 0.5650]
	HPL	233	2.33E-10%	[0.56446068002405417 0.56446068002405649]
	TAG(τ_c=44)	100819	1.01E-07%	[0.5644606800**2355170**, 0.5644606800245**5988**]
	TAG(τ_c=48)	6064	6.06E-09%	[0.5644606800240**2601**, 0.5644606800240**8664**]
	TAG(τ_c=52)	325	3.25E-10%	[0.56446068002405417, 0.56446068002405**741**]
	RAIVE(τ_c=44)	50572	5.06E-08%	[0.5644606800**2380185**, 0.56446068002**430756**]
	RAIVE(τ_c=48)	3140	3.14E-09%	[0.5644606800240**3901**, 0.5644606800240**7040**]
	RAIVE(τ_c=52)	325	3.25E-10%	[0.56446068002405417, 0.56446068002405**741**]
pagerank	OVERALL	1E+14		[1.10, 1.11]
	HPL	122	1.22E-10%	[1.1026503685992210, 1.1026503685992331]
	TAG(τ_c=44)	593	5.93E-10%	[1.102650368599**1619**, 1.1026503685992**211**]
	TAG(τ_c=48)	38	3.80E-01%	[1.1026503685992**174** , 1.1026503685992**211**]
	TAG(τ_c=52)	2	2.00E-12%	[1.1026503685992210, 1.1026503685992**211**]
	RAIVE(τ_c=44)	593	5.93E-10%	[1.102650368599**1912**, 1.1026503685992**504**]
	RAIVE(τ_c=48)	38	3.80E-11%	[1.1026503685992**190**, 1.1026503685992**227**]
	RAIVE(τ_c=52)	2	2.00E-12%	[1.1026503685992210, 1.1026503685992**211**]

Table 2. Instability detection.

program	# of unstable preds.	# preds (%) that merge	# fork	# output var.	precision	recall	%RSD
178.galgel	253385	253382 (99%)	3	7	100%	71%	-
183.equake	1	0 (0%)	1	12	99%	99%	69%
187.facerec	14	9 (64%)	5	10	100%	100%	6.4%
deisotope	1	0 (0%)	1	30	97%	97%	43%
k-means	1	0 (0%)	1	92	100%	100%	55%
pagerank	1	0 (0%)	1	10	100%	100%	16%

Table 3. Average number of unstable predicates, and forks for an execution, and output variations across samples. RSD stands for relative standard deviation.

ior [10, 16] cannot report output variations caused by discrete differences.

We have also repeated the same experiment on $\tau_c = 44$. In this case, we have much more unstable sample runs. However, the results are very similar to those in Table 3. The precision and recall are still close to 100%. The only difference is that RSD values are smaller due to the larger sample size. The observation is hence that the output variations are not sensitive to the threshold. The reason is that all these unstable sample runs lead to the same path differences and hence the same output variations. In other words, using a more conservative (i.e., smaller) threshold only means that the same output variations are exposed by a larger set of inputs (falling in the unstable range). On the other hand, as long as an input falls into the unstable range, any errors, including internal and external errors, cause the same output variations.

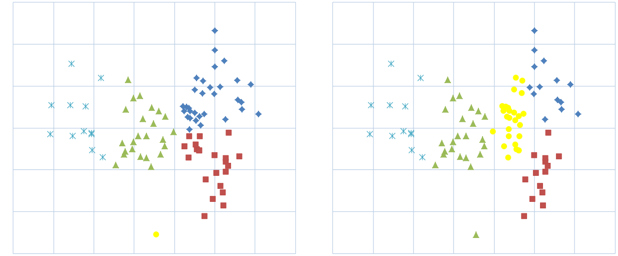
5.3 Case Studies

K-means [13] implements a widely used clustering algorithm, which partitions inputs into k clusters by the given distance metrics. The core algorithm is shown in Fig. 9. In each iteration, it first computes the centroids for the current k clusters (line 2) and then the sum of the distances from each element to its centroid (line 3). It then traverses all elements to check if the current partition can be further improved (lines 5-14). This is done by checking for an element e, if there is a cluster J whose centroid is closer than e's current cluster K. If so, e is moved to J. The algorithm repeats until the overall distance stabilizes (line 15).

```
1  do {
2    getclustermeans(cluster, data);   /* Find the centroids */
3    total_dist = ...;                 /* Compute current total dist */
4    total_dist_new = 0;
5    foreach element e {
6      K = cluster[e];      /* Element e belongs to cluster K */
7      dist = distance[e];
8      foreach cluster J {
9        dist_new = euclid(e, J);
10       if (dist_new < dist) {
          /* Move element e from cluster K to J. */
11        cluster[e] = J;
12        distance[e] = dist_new;
13        dist = dist_new;
       } }
14     total_dist_new+ = dist;
     }
15 } while (total_dist_new < total_dist);
```

Figure 9. Pseudocode snippet for `k-means`.

Since there is an element e with a very similar distance to the centroids of J and K, leading to an unstable predicate $dist_new < dist$ at line 10, RAIVE detects the instability and evaluates both branches. But it cannot merge the two branches and hence forks, yielding two different clustering results as shown in Fig. 10. While we used simple input data in the case study, real world data set could be very complex and can hardly be manually inspected. RAIVE can automatically identify the possible clustering outputs.



(a) Result from the original exec. (b) Result from the forked exec.

Figure 10. Clustering result variations of an unstable execution for `k-means`; 92 genes are grouped into five clusters; each cluster has a unique color.

```
1  while (cont) {
2    cont = 0;
3    foreach page p{
4      foreach neighbor n
5        neighbor_sum+ = score(n)/num_out_links;
6      new_score = ... + (... * neighbor_sum);
7      diff = abs(new_score − score(p));
8      if (diff > threshold)/* unstable */
9        cont = 1
10     }
11     score(p) = new_score;
```

Figure 11. Pseudocode snippet for `pagerank`.

Pagerank [14] is one of the most widely used algorithms in information retrieval. It was invented by Google and used to rank result pages for a search request. It computes a score for each page according to the number and quality of other pages linked to it. The score represents the probability that a random surfer will visit a page. A page with many high-score pages linked to it also has a high score. Part of the algorithm is shown in Fig. 11. In each iteration, a new score is computed for each page according to its incoming neighbors' previous scores. The algorithm repeats until the absolute difference between the new score and the old score is smaller than a threshold for all pages.

In this experiment, we use a set of similar pages. RAIVE reports instability at line 8 as `diff` and `theshold` are very close to each other. Hence, the execution is split to two and one of them iterates more. The iteration differences substantially change the final rankings of the pages. While the program printed the top 10 pages, we showed the top 3 in the following table.

Orig. (35 iterations)		Forked (34 iterations)	
page id	score	page id	score
722	0.999997000...	968	0.99999499...
723	0.999997000...	969	0.99999499...
724	0.999997000...	970	0.99999499...

Observe that the results are completely different. Also, there may not be a patch to the code that can fix the instability problem, which does not lie in the numerical core of the algorithm, but rather in the interface between the floating

point computation and the discrete logic. It is a property of the algorithm and the provided input. Furthermore, generating inputs that expose such instabilities may not be as useful as in exposing functional bugs due to the infeasibility of fixing them. Therefore, we argue that showing the possible outputs to the user provides a reasonable solution.

```
    /*coarse-grained search*/
142 Position = (LLX, LLY)
    ...
    /*fine-grained search*/
168 CurSimilarity = GraphSimFct(LLX, LLY, ...)
169 If (CurSimilarity > Similarity) Then
170   Similarity = CurSimilarity
171   Position = (LLX, LLY)
172 EndIf
```

Figure 12. Benign unstable predicate in `187.facerec`.

Benign Unstable Predicate in 187.facerec. `Facerec` is a Fortran program for face recognition. It consists of two phases of search. The first one is coarse-grained and the second one is fine-grained. It is possible that both phases identify the same object. The related code snippet is shown in Fig. 12. The object identified in the first phase is saved in `Position` at line 142. Lines 168-172 are for the second phase. In particular, the newly computed `CurSimilarity` is compared with the current `Similarity` (line 169). If the new similarity is larger, the current similarity and position are updated. If both phases identify the same object, the difference between the two similarity metrics is very small, leading to instability at line 169. RAIVE executes both branches: lines 170-171 and the fall-through. At the join point 172, the vector for `Similarity` has trivial differences inside. Intuitively, since the two similarity metrics are very close, the update in the true branch has little effect. Moreover, the integer variable `Position` has the same values in the two branches as the same object was identified in the two phases. As such, the instability is benign.

6. Related Work

RAIVE is related to dynamic instability detection techniques such as interval analysis [21, 26], high precision computation [5], and error tagging [2]. Compared to these techniques, RAIVE is much more efficient and can reason about output variations.

A dynamic technique was proposed in [22] to detect bit cancellations. It does not distinguish benign and problematic cancellations and thus reports many false alarms. Researchers have proposed techniques to generate tighter bounds for interval arithmetic [15, 16] and affine arithmetic tools [10, 17]. Affine arithmetic handles variable correlations using affine forms. These techniques are very expensive (3-4 orders of magnitude slowdown [10]) and may have difficulty scaling to complex programs. They mostly focus on numerical cores and can hardly reason about discrete differences caused by errors, which are common in real world programs and usually induce substantial output variations.

There are also a large body of work on abstract interpretation, SMT solving, model checking and code perturbation to tackle the internal error problem [9, 23, 25, 29]. 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 programs are difficult to address by the technique. Moreover, as instability problems are input dependent and rarely happen, dynamic analysis may be more preferable when completely fixing instability is difficult.

RAIVE is also related to uncertain data processing. In [28], a static analysis is proposed to analyze probabilistic programs that operate on uncertain data. In [6], an abstraction was proposed to help developers operate on and reason about uncertain data. A sampling technique was proposed in [3] to expose discontinuity in output functions, in the presence of input uncertainty. Different from RAIVE, they explicitly model and sample external errors.

In [8], a technique is proposed to search for error-causing inputs that can maximize result errors due to internal errors. In [30], researchers propose to reason about the portability of numerical programs by using symbolic analysis to find inputs that cause different branch decisions, when the program is executed with the same input on different platforms. Recently, [11, 27] propose techniques to reason about the required precision to compile a given program with given output requirements.

7. Conclusion

We propose RAIVE, a technique to detect output variations caused by errors (both internal and external) in floating point computation. It transforms a floating point value to a vector of four values and encodes the presence of an error by injecting value differences into the vector. Error propagation and suppression are performed implicitly by vectorized floating point operations. Instability is detected by checking if all vector elements lead to the same discrete result at discrete factors. Evaluation shows that RAIVE can precisely identify output variations. Compared to the state-of-the-art, RAIVE's overhead is 2.43 times lower, averaging 340%, and it has the new capability of reporting output variations.

Acknowledgements

This research is supported, in part, by the National Science Foundation (NSF) under grants 0845870 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] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst. 2008.

[2] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In OOPSLA '13.

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

[4] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In POPL '13

[5] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In PLDI '12

[6] J. Bornholt, T. Mytkowicz, and K. S. McKinley. A first-order type for uncertain data. In ASPLOS '14.

[7] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In ESEC/FSE '11.

[8] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In PPoPP '14.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyzer. In ESOP '05.

[10] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In OOPSLA '11.

[11] E. Darulova and V. Kuncak. Sound compilation of reals. In POPL '14.

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

[13] M. de Hoon. Cluster 3.0. `http://bonsai.hgc.jp/~mdehoon/software/cluster/software.htm`.

[14] L. Pages, S. Brin, R. Motwani, and T .Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[15] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In FMICS '09.

[16] F. D. Dinechin and L. P. Arnaire. Assisted verification of elementary functions using gappa. In SAC '06.

[17] C. F. Fang, T. Chen, and R. 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.

[18] E. Goubault and S. Putot. Static analysis of finite precision computations. In VMCAI'11.

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

[20] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-028. July 2013.

[21] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag New York Incorporated, 2012.

[22] M. O. Lam, J. K. Hollingsworth, and G. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39 (3):146 – 155, 2013.

[23] M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In ESOP '02.

[24] G. Melquiond and C. Munoz. Guaranteed proofs using interval arithmetic. In ARITH '05.

[25] D. Monniaux. The pitfalls of verifying floating-point computations. ACM TOPLAS, 30(3):12:1–12:41, May 2008.

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

[27] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In SC '13.

[28] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In PLDI '13.

[29] E. Tang, E. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In ISSTA '10.

[30] Y. Gu, T. Wahl, M. Bayati and M. Leeser. Behavioral Non-portability in Scientific Numeric Computing. In Euro-Par '15.