Fast Shadow Execution for Debugging Numerical Errors using Error Free Transformations

SANGEETA CHOWDHARY, Rutgers University, United States
SANTOSH NAGARAKATTE, Rutgers University, United States

This paper proposes, EFTSanitizer, a fast shadow execution framework for detecting and debugging numerical errors during late stages of testing especially for long-running applications. Any shadow execution framework needs an oracle to compare against the floating point (FP) execution. This paper makes a case for using error free transformations, which is a sequence of operations to compute the error of a primitive operation with existing hardware supported FP operations, as an oracle for shadow execution. Although the error of a single correctly rounded FP operation is bounded, the accumulation of errors across operations can result in exceptions, slow convergences, and even crashes. To ease the job of debugging such errors, EFTSanitizer provides a directed acyclic graph (DAG) that highlights the propagation of errors, which results in exceptions or crashes. Unlike prior work, DAGs produced by EFTSanitizer include operations that span various function calls while keeping the memory usage bounded. To enable the use of such shadow execution tools with long-running applications, EFTSanitizer also supports starting the shadow execution at an arbitrary point in the dynamic execution, which we call selective shadow execution. EFTSanitizer is an order of magnitude faster than prior state-of-art shadow execution tools such as FPSanitizer and Herbgrind. We have discovered new numerical errors and debugged them using EFTSanitizer.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Mathematics of computing** → *Mathematical software.*

Additional Key Words and Phrases: rounding errors, floating point, shadow execution, EFTSanitizer

ACM Reference Format:
Sangeeta Chowdhary and Santosh Nagarakatte. 2022. Fast Shadow Execution for Debugging Numerical Errors using Error Free Transformations. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 190 (October 2022), 28 pages. https://doi.org/10.1145/3563353

1 INTRODUCTION

Almost all endeavors in science and engineering need real numbers. A floating point number is an approximation of a real number using a finite number of bits [Goldberg 1991]. In the past, numerous computer vendors had their own convention for floating point resulting in non-portable applications. The IEEE-754 standard significantly advanced the ecosystem by standardizing the floating point (FP) representation, which improved portability and reproducibility of applications.

Rounding errors. Almost all real values cannot be exactly represented in any finite FP representation. When a real value is not exactly representable in the FP representation, it has to be rounded to the nearest value according to the rounding mode specified by the standard. The rounding operation introduces some error with every operation, which is bounded by the gap between two FP values adjacent to the real value. Although the rounding error of an individual

Authors' addresses: Sangeeta Chowdhary, Department of Computer Science, Rutgers University, United States, sangeeta.chowdhary@rutgers.edu; Santosh Nagarakatte, Department of Computer Science, Rutgers University, United States, santosh.nagarakatte@cs.rutgers.edu.

2475-1421/2022/10-ART190
https://doi.org/10.1145/3563353

operation is bounded, this rounding error can accumulate over a sequence of operations and one may encounter a scenario where all the bits in the number are influenced by rounding error (*e.g.*, catastrophic cancellation [Goldberg 1991]). Similarly, there are numerous exceptional values such as Not-a-Number (NaNs) and infinities in the FP representation (see Section 2.1). This accumulation of rounding errors and/or the concomitant exceptions have resulted in various catastrophic incidents (*e.g.*, the Patriot missile failure [United States General Accounting Office 1992]).

Performance vs precision. One way to reduce the amount of rounding error is to increase the precision of the representation, which impacts performance and throughput. However, good floating point performance is paramount in many application domains like scientific computing and machine learning. Hence, many accelerators for machine learning use non-standardized custom FP representations to improve performance. For example, Nvidia's tensorfloat32 (a 19-bit representation), Google's bfloat16 (a 16-bit representation), and Microsoft's MSFP are some recent FP variants that make trade-offs between the range of values that can be represented and the precision of each value.

Static analyses for numerical errors. There is a large body of work to estimate rounding errors without executing the program through static analysis [Chiang et al. 2017; Darulova et al. 2018; Darulova and Kuncak 2014; Feliú et al. 2018; Ghorbal et al. 2012; Goubault 2001; Goubault et al. 2007; Higham 2002; Solovyev et al. 2018]. Such techniques tend to work well for loop-free fragments of code [Chiang et al. 2017; Das et al. 2020] and when there are well-defined error specifications [Darulova and Kuncak 2014]. How to scale these techniques to large code bases is still an open problem.

Shadow execution for detecting numerical errors. Alternatively, dynamic analyses to detect exceptions [Barr et al. 2013; Dinda et al. 2020] and/or cancellations [Bao and Zhang 2013] are useful to detect specific kinds of errors. Beyond such specific errors, an attractive technique to detect a wide classes of numerical errors is shadow execution with high-precision computation [Benz et al. 2012; Chowdhary et al. 2020a; Chowdhary and Nagarakatte 2021; Sanchez-Stern et al. 2018]. Every FP value is shadowed with a corresponding high-precision value (*e.g.*, using the MPFR library [Fousse et al. 2007]). Similarly for every FP operation, the corresponding operation is performed with high-precision computation. The value produced by the FP program is compared with the value in the shadow execution at various points of interest. The instrumentation for shadow execution can be injected either with compile-time instrumentation [Chowdhary et al. 2020a; Chowdhary and Nagarakatte 2021] or with binary instrumentation [Benz et al. 2012; Sanchez-Stern et al. 2018]. Prior research has demonstrated that such shadow execution is effective in detecting numerical errors.

Challenges with existing shadow execution approaches. Shadow execution is not widely used with long-running applications because of the associated performance overhead. Prior binary instrumentation based shadow execution tools such as FPDebug [Benz et al. 2012] and Herbgrind [Sanchez-Stern et al. 2018] have performance overheads of $200\times - 600\times$ or more for small programs. Our prior work, FPSanitizer [Chowdhary et al. 2020a], is a compiler-based shadow execution tool that significantly reduces the overhead compared to Herbgrind and FPDebug. However, it still has performance overheads of 100× or more. Such large performance overheads limit the use of these tools with long-running applications. Software emulation of high-precision FP values using the MPFR library is the primary cause of performance overheads with these shadow execution tools.

Beyond detecting numerical errors, debugging a numerical error often requires reasoning about the propagation of errors. FPSanitizer and Herbgrind provide a limited directed acyclic graph (DAG) of instructions to highlight the propagation of errors. Herbgrind can generate such DAGs only for small unit tests as the extra information needed per-memory location in their approach is proportional to the number of dynamic instructions. In contrast, FPSanitizer addresses this problem by providing DAGs for long-running programs but makes two simplifying assumptions. First, it provides DAGs only when the instructions in the DAG belong to functions in the set of active stack frames when the error is detected. When a function completes execution, the DAG information about operations in that function will be lost. Second, it provides DAGs only for the last iteration in the presence of loops. While debugging numerical errors with FPSanitizer is better than what was previously possible, it is still challenging especially when the accumulation of rounding error spans function calls and loops. In such scenarios, the user will need to re-run the program numerous times with breakpoints before every function return or every loop iteration.

Error free transformations as the oracle in EFTSanitizer. We propose EFTSanitizer, a shadow execution framework that is usable with long-running applications because it is an order of magnitude faster than FPSanitizer. A key insight about the FP representation, which is surprisingly not used frequently by the developers of shadow execution tools, is that the rounding error of a primitive operation can be represented as an FP number in the same FP representation [Muller et al. 2018]. Further, the rounding error of a primitive operation can be computed with a sequence of regular FP arithmetic operations, which are known as error free transformations (EFTs). Section 2.2 provides background on EFTs. Rather than using high-precision computation as the oracle for comparing the FP execution with the shadow execution, we propose using error free transformations as the oracle for the shadow execution. Use of hardware supported FP operations to compute the error makes EFTSanitizer's shadow execution significantly faster than a shadow execution using the MPFR library.

EFTs have been previously used to extend the precision for geometric algorithms [Shewchuk 1996], to create libraries for encapsulating error with Shaman [Demeure 2020], and to generate compensation code. To the best of our knowledge, EFTSanitizer is first approach that advocates the use of error free transformations as an oracle for shadow execution.

EFTSanitizer's debugging support for long-running applications. To facilitate effective debugging with long-running applications, EFTSanitizer incorporates two key features. First, EFTSanitizer allows the user to perform shadow execution starting from an arbitrary point in the dynamic execution, which we call selective shadow execution. This is appealing for scientific simulations that execute for days. Second, EFTSanitizer provides a directed acyclic graph of instructions that spans multiple functions (many of which may have already completed) and multiple iterations of the loop while keeping the memory usage bounded.

EFTSanitizer is a compile-time instrumentation framework that instruments every FP variable in memory and registers to track additional information, which we call metadata. The key technical contribution of EFTSanitizer is the design of the metadata such that shadow execution can be started at an arbitrary point in the dynamic execution while detecting errors and providing a rich DAG of instructions to highlight the propagation of rounding errors.

To just detect errors, it is sufficient to propagate the rounding error computed using error free transformations with each FP variable. To produce DAGs, additional information about the operands needs to be maintained. To keep the memory usage bounded, DAGs produced by EFTSanitizer consists of last k dynamic instructions at the point of a numerical error. The threshold k can be configured by the user. These instructions can span functions that have already completed execution and various iterations of a loop. We found these DAGs useful to debug new numerical errors discovered by EFTSanitizer and also to validate existing bugs. However, it is important to note that just knowing the propagation of errors using the DAGs produced by our approach does not necessarily mean that the issue can be easily fixed. Repairing some errors may require different algorithms. In some cases, it may be feasible to use a modified expression using tools

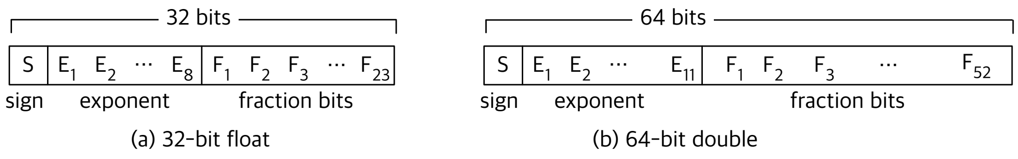


Fig. 1. The bit-string of the float and double formats in the IEEE-754 binary FP representation.

like Herbie [Panchekha et al. 2015] to reduce the error. The user will have to identify appropriate techniques to repair the program depending on the program in consideration.

Our prototype of EFTSanitizer is open source and publicly available [Chowdhary and Nagarakatte 2022a,b]. It is built on top of the LLVM-10 compiler and supports C/C++ programs. We have discovered new bugs in well tested applications (*e.g.*, Lulesh, AMG, and NAS IS) and validated that our tool detects existing bugs. EFTSanitizer is approximately 15× faster than FPSanitizer, which is the state-of-the-art for shadow execution.

In summary, this paper makes the following contributions.

- Proposes the first shadow execution framework that uses error free transformations as the oracle.
- Proposes a method that enables the user to run shadow execution starting from an arbitrary point in time in the dynamic execution.
- Proposes techniques for metadata management with shadow execution to produce DAGs, which highlights the propagation of rounding errors, that spans multiple function calls and loop iterations.
- Demonstrates that EFTSanitizer is effective in discovering previously unknown numerical errors and has an order of magnitude speedup compared to FPSanitizer.

2 BACKGROUND

We provide a brief overview of the FP representation and describe error free transformations, which will be used by EFTSanitizer as the oracle for shadow execution.

2.1 The Floating Point Representation

The floating point (FP) representation specified by the IEEE-754 standard is widely used to approximate real numbers. Two main attributes of any FP representation are its dynamic range (*i.e.*, range of values representable) and the precision with which each value is represented. The various formats (*e.g.*, half, float, and double) in the IEEE-754 standard provide reasonable dynamic range and precision appropriate for widely used applications. Most processors have hardware implementations for at least a few formats (*i.e.*, float and double).

In the IEEE-754 binary FP representation, the FP value is represented by a bit-string that consists of a sign bit (S), a set of bits that represent the unsigned (biased) exponent (E), and a set of bits that represents the fraction (F). We represent the number of bits used for the exponent field with $\mid E \mid$ and the number of bits used for the fraction field with $\mid F \mid$. The values represented by the FP representation are classified into normal values, subnormal values, and special values depending on the bit-pattern in the exponent. The exponent field is interpreted as an unsigned integer. Hence, a bias is used to represent FP values with negative exponents. The bias for a given format is $2^{|E|-1} - 1$, which is the midpoint of the set of feasible exponents representable with the FP representation. By interpreting the exponent field as an unsigned integer, two floating values with the same sign can be compared using integer comparisons.

When the exponent bits are not all zeros and not all ones (*i.e.*, E $\in [1, 2^{|E|} - 2]$), then the bit-string represents a normal value. The value represented by the FP bit-string is $(-1)^S \times 2^{E-bias} \times (1+F/2^{|F|})$.

When the exponent bits are all zeros, it represents subnormal values. It is used to represent values close to zero. The value represented by the bit-string is $(-1)^S \times 2^{1-bias} \times (F/2^{|F|})$. When the exponent bits are all ones, it represents special values. When, additionally, the fraction field is all 0's, it represents $+\infty$ if the sign bit is 0 and $-\infty$ otherwise. When the fraction field is not all zeros, then bit-string represents Not-a-Number (NaN), which is used to represent exceptional conditions.

The commonly used 32-bit float format has 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fraction. The 64-bit double format has 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. Figure 1 shows the bit-patterns used for the 32-bit float and 64-bit double formats.

Rounding modes. Most real numbers cannot be exactly represented in a FP representation. Hence, a real value is rounded to the nearest FP value according to the rounding mode. Given a real number x, x_l is the FP number less than or equal to x, and x_h is the FP number greater than x. Depending on the rounding mode, x is rounded either to x_l or x_h. The IEEE-754 standard specifies multiple rounding modes: round down (RD), round up (RU), round to zero (RZ), round to nearest ties to even (RN), and round to nearest ties to away (RA). The round to nearest ties to even mode is the default rounding mode. With the round to nearest ties to even mode, when x is less than the midpoint, it rounds to x_l and when x is greater than the midpoint, it rounds to x_h. When x is exactly at the midpoint between x_l and x_h, x is rounded to x_l if the last bit of x_l is 0. Otherwise, x rounds to x_h.

Rounding errors. Rounding a real value, which is is not exactly representable in a FP representation, to the nearest FP number results in rounding error. If x is a real value and x_{fp} is the rounded FP value, then the absolute error is $|x_{fp} - x|$. If x is in range of normal values, then the absolute error is less than the gap between two floating-point numbers x_l and x_h where $x_l \leq x \leq x_h$ for all rounding modes defined by the IEEE-754 standard. The absolute rounding error for the round to nearest modes (*i.e.*, RN and RA) is at most half of the gap between x_l and x_h. If x_{fp} has an exponent e, then $|x_{fp} - x| < 2^{-p} \times 2^e$, where p is the precision of the FP representation [Muller et al. 2018].

The IEEE-754 standard mandates correct rounding of primitive operations. Hence, the rounding error of any primitive operation is bounded. However, this rounding error can be amplified by operations such as subtraction. When two values close to each other are subtracted, it can cancel out all the leading bits such that remaining bits are influenced by rounding errors. This accumulation of errors with a sequence of operations can cause the program to produce totally different results, exceptional results such as NaNs and infinities, and can cause divergence in iterative algorithms [Benz et al. 2012; Higham 2002; Muller et al. 2018].

In the context of shadow execution, a common way to measure the absolute error and its propagation with various operations is to use a high-precision library such as MPFR [Fousse et al. 2007]. Next, we describe how we can compute this rounding error using FP operations itself.

2.2 Computing the Rounding Error with Error Free Transformations

An important, yet commonly unused, property of the floating point representation is that the rounding error of a primitive FP operation itself can be represented as a floating point number [Muller et al. 2018]. A sequence of FP operations to compute the rounding error of a primitive operation are called error free transformations (EFTs) [Ogita et al. 2005]. EFTs are appealing for shadow execution because we can use existing hardware supported FP operations to compute the rounding error. Given two FP operands a, b, and a primitive FP operation +, the error free transformations enables us to compute the floating point result $x = a + b$ and the rounding error δ_x such that $a +_R b = x +_R \delta_x$. Here, $+_R$ represents primitive operation with real numbers. Although the error of

```
1 Function TwoSum($a, b$):
2     $x \leftarrow a + b$
3     $b' \leftarrow x - a$
4     $a' \leftarrow x - b'$
5     $\delta_a \leftarrow a - a'$
6     $\delta_b \leftarrow b - b'$
7     $\delta_x \leftarrow \delta_a + \delta_b$
8     return $(x, \delta_x)$
```

```
1 Function PropSumError($(a, \delta_a), (b, \delta_b)$):
2     $(x, \delta_x) \leftarrow$ TwoSum($a, b$);
3     $\delta_x \leftarrow \delta_x + \delta_a + \delta_b$;
4     return $(x, \delta_x)$
```

Fig. 2. Error free transformations for addition. The function `TwoSum` computes the result (x) of FP addition and the rounding error (δ_x) from FP addition of two operands a and b with the assumption that operands do not have any error. All operations are performed using FP operations. The function `PropSumError` computes the FP result and the rounding error when the input operands also have some error (*i.e.*, δ_a and δ_b).

the primitive operation δ_x is representable in the FP representation, the value $x +_R \delta_x$ rounds to x in the FP representation.

An interesting aspect of EFTs is that for some computation they provide more precision when compared to arithmetic performed with the 128-bit format (*i.e.*, double-double) even when we maintain a single 64-bit error term. For example, the expression $(1.0 + (1.7 \times 10^{308})) - (1.7 \times 10^{308})$ will return 0 with double precision arithmetic. The addition $(1.0 + (1.7 \times 10^{308}))$ returns 1.7×10^{308} due to the loss of precision. This causes the final result to be 0. To capture this error with high precision computation (*e.g.*, MPFR library), we need at least 1024 bits of precision to store the result of addition precisely. By explicitly maintaining error with EFTs, we can easily capture this error. In essence, we can store the result of $(1 + (1.7 \times 10^{308}))$ as the sum of two floating-point numbers using EFTs. Hence, EFTs provide a mechanism to split the floating-point numbers as the sum of two non-overlapping floating-point numbers [Jeannerod et al. 2018].

Next, we describe error free transformations to compute the error of various primitive operations with the assumption that input operands do not have any error. Subsequently, we describe how to compose the error of the operands with error free transformations. We use $+_R$ to represent primitive operation + performed with real numbers. Otherwise, all operations are performed using floating point arithmetic operations.

Computing the rounding error for an FP addition operation. The sequence of FP operations to compute the rounding error of an FP addition operation with the round to nearest mode was proposed by Donald Knuth [Muller et al. 2018]. It was called `TwoSum` by Shewchuk [Shewchuk 1996]. Figure 2 provides the `TwoSum` algorithm. It assumes that there is no error in the input operands. It uses Sterbenz's lemma that states certain FP operations are exact without any rounding error. Specifically, if a and b are nonnegative FP numbers such that $b/2 \leq a \leq 2b$, then $a - b$ is exactly representable in the FP representation [Sterbenz 1974]. If $|a| \geq |b|$, then the subtraction in line 3 of `TwoSum` in Figure 2 is exact from Sterbenz's lemma. If $|a| < |b|$, then line 2 may have some rounding error, which is computed by computing $(a - a')$ and $(b - b')$ as shown in Figure 2. Other subtraction operations in the TwoSum algorithm in Figure 2 are exact from Sterbenz's lemma.

Provided there are no underflows or overflows in the computation of $a + b$, the `TwoSum` algorithm computes the error exactly representable as a FP value. The TwoSum algorithm may experience an overflow for some rare cases when the actual computation does not overflow [Boldo et al. 2017]. However, those cases are rare in practice [Muller et al. 2018]. Such error free transformations have also been explored to produce the rounding error for other rounding modes in the IEEE standard [Priest 1992].

If $|a| \geq |b|$, then a faster algorithm for computing the rounding error with FP operations can be used, which is also known as Dekker's Fast2Sum algorithm [Muller et al. 2018]. The rounding error can be computed exactly as $b - (x - a)$. We use the `TwoSum` algorithm for our shadow execution with error free transformations because we do not want to have an additional branch instruction and a swap of the operands for computing the error.

Propagating the error of the operands with addition. Using `TwoSum`, we can compute the rounding error of a single FP addition operation. The operands to this addition themselves may have been produced as a result of other FP operations. Hence, they will have some error. We need to propagate the error from the operands to the error of the result. The `PropSumError` function in Figure 2 shows the computation of the resultant error. We add the error in the operands to the error of the result of the FP addition. As the addition of error is performed with FP arithmetic, there will be some small rounding error corresponding to the error terms. It is possible to use the non-overlapping components method to compute this error [Shewchuk 1996]. For the purpose of shadow execution, we chose to ignore the second order error terms as they are extremely small. The computed error with this method is at least as good as the error computed with double-double arithmetic (*i.e.*, 128-bit FP format).

EFTs for Subtraction. To compute the error of the subtraction operation, we use `TwoSum` with sign of the second operand changed (*i.e.*, `TwoSum`$(a, -b)$). The propagated error is $\delta_x + \delta_a - \delta_b$, where δ_x is the error of the subtraction assuming no error in the operands. Here, δ_a and δ_b represents the error in the operands a and b, respectively.

Computing the rounding error for FP multiplication. Computing the rounding error of a single FP multiplication operation is easy when there exists a fused multiply-add (*fma*) operation in the system. Hardware vendors started adding fused multiply-add operations to processors in 1990 (*e.g.*, IBM POWER1). Now, almost all mainstream processors have hardware support for them. Fused multiply-add operations are also included in the 2008 version of the IEEE-754 FP standard. Further, recent processors also support SIMD versions of fused multiply-add (*i.e.*, FMA3 and FMA4 in the `x86` instruction set). Semantically, a correctly rounded fused multiply-add operation performs both the multiplication and the addition operation with infinite precision and the result is finally rounded to the FP representation (*i.e.*, only one rounding). Given operands a and b and the FP multiplication result x, the rounding error with FP multiplication can be computed as follows:

$$\delta_x = fma(a, b, -x)$$

The above method accurately computes the rounding error, which is representable as an FP value, for the round to nearest ties to even mode provided overflows and underflows do not occur. Specifically, error term δ_x is an exact FP number if $e_a + e_b \geq e_{min} + p - 1$, where e_a and e_b represent the exponents of a and b, and p is the precision of the FP representation. When this condition is not satisfied, then the error δ_x is below the underflow threshold. Hence, it is not exactly representable as an FP number [Muller et al. 2018].

When the system does not support fused multiply-add operations, then a more sophisticated algorithm called Dekker-Veltkamp splitting is used to compute the rounding error [Muller et al. 2018]. We use the fused multiply-add operation to compute the rounding error with a single multiplication operation.

Propagating the rounding error with multiplication. When the operands have error, we have (a, δ_a) and (b, δ_b) as the operands, we want to compute $(a +_R \delta_a) *_R (b +_R \delta_b)$. Hence,

$$x +_R \delta_x = (a +_R \delta_a) *_R (b + \delta_b) = (a *_R b) +_R (a *_R \delta_b) +_R (b *_R \delta_a) +_R (\delta_a *_R \delta_b)$$

Simplifying and ignoring the second order error terms (*i.e.*, $\delta_a *_R \delta_B$), we can perform the computation on the error terms using FP operations as shown below.

$$\delta_x = fma(a, b, -x) + a * \delta_b + b * \delta_a$$

By computing the operations on the error terms with FP operations, we will not be considering small amounts of rounding error in the computation on error terms, which is acceptable for debugging with shadow execution.

Computing and propagating the rounding error of the FP division operation. Similar to multiplication, computing the rounding error for the FP division operation can accomplished using the fused multiply-add operation.

$$x = a/b, \;\; \delta_x = fma(x, b, -a)$$

The above rounding error can be exactly computed using the fused multiply-add operation provided $e_b + e_x \geq e_{min} + p - 1$, where e_b, e_x, and e_{min} are the exponents of b, x, and the minimum exponent in the representation, respectively. Here, p is the amount of precision of the FP representation.

When the operands have some error $((a, \delta_a), (b, \delta_b))$, we want to compute $(a +_R \delta_a)/_R(b +_R \delta_b)$.

$$x +_R \delta_x = (a +_R \delta_a)/_R(b +_R \delta_b)$$

After rearranging the terms,

$$\delta_x = ((a +_R \delta_a)/_R(b +_R \delta_b)) -_R x = ((a +_R \delta_a) -_R (x *_R b) -_R (x *_R \delta_b))/_R(b +_R \delta_b)$$

After performing the computation of $x *_R b -_R a$ using the fused multiply-add operation and the rest of the computation on error terms using FP operations, the propagated error for division is

$$\delta_x = (\delta_a - fma(x, b, -a) - x * \delta_b)/(b + \delta_b)$$

Computing and propagating the error for square root. Similar to FP multiplication and division, the rounding error of a correctly rounded FP square root operation can be computed with the fused multiply-add operation as follows,

$$x = \sqrt{a}, \;\; \delta_x = fma(-x, x, a)$$

The rounding error $a - x^2$ is exactly representable with p bits of precision if $2e_x \geq e_{min} + p - 1$, where e_x is the exponent of x [Boldo and Daumas 2003; Muller et al. 2018].

When the operand has error (*i.e.*, (a, δ_a)), then we want to compute $x +_R \delta_x = \sqrt{a +_R \delta_a}$. After squaring both sides, rearranging the terms after ignoring the second order error term (δ_x^2) and computing $(a - x^2)$ with fused multiply-add , and performing the computation with FP operations, we have

$$\delta_x = (\delta_a + fma(-x, x, a))/2x$$

When we compute the error with the above formula for the square root operation, we also handle the case where $x = 0$ separately to avoid divide-by-zero exceptions in the computation of the error.

3 THE EFTSANITIZER APPROACH

Our goal is to develop an approach that is useful for detecting and debugging numerical errors in long-running programs during late stages of testing. To accomplish this goal, we need the resulting approach to have the following attributes. First, it should detect a wide range of errors such as exceptions (due to NaNs and infinities), cancellations where a large fraction (or all) of the bits are wrong, slow convergences, and/or significant rounding errors. Second, it should enable the debugging of reported errors with an execution trace that illustrates the propagation of errors. Third, we want to be able to debug the program from an arbitrary point of the execution. This is necessary with scientific simulation experiments that typically run for days. Finally, it must have low performance and memory overhead.

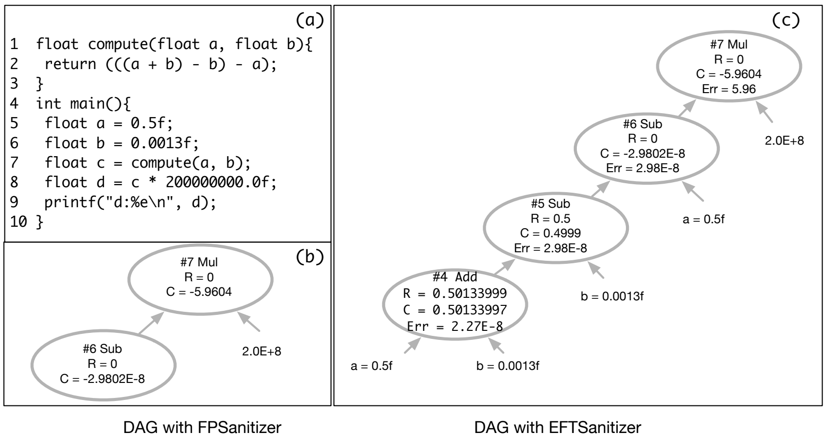
Error free transformations for shadow execution. This paper proposes EFTSanitizer, which is the first approach that performs inlined lock-step shadow execution with error free transformations (EFTs) as the oracle. Using error free transformations, EFTSanitizer computes the propagated rounding error with hardware supported FP operations. This use of hardware FP operations makes EFTSanitizer significantly faster compared to prior shadow execution tools [Benz et al. 2012; Chowdhary et al. 2020a; Sanchez-Stern et al. 2018].

EFTSanitizer is a compile-time instrumentation framework that automatically adds code after each FP operation to compute and propagate the error using EFTs. EFTSanitizer maintains this propagated rounding error for both variables in memory and in registers. For operations that do not have EFTs available (*e.g.*, elementary functions), we use corresponding high-precision operations from the MPFR library [Fousse et al. 2007] to compute the error. In the case of elementary functions, the user can configure EFTSanitizer to use correctly rounded elementary functions [Aanjaneya et al. 2022; Daramy-Loirat et al. 2006; Lim et al. 2020, 2021; Lim and Nagarakatte 2021a,b,c, 2022; Muller 2016] to compute the error, which improves performance.

Debug information to illustrate the propagation of rounding errors. The propagation and accumulation of rounding errors with a sequence of operations causes exceptions and wrong results in programs. Hence, EFTSanitizer provides a dynamic trace of instructions represented as a directed acyclic graph that demonstrates the propagation of errors. Prior research on shadow execution such as FPSanitizer [Chowdhary et al. 2020a] and Herbgrind [Sanchez-Stern et al. 2018] also provide DAGs to assist debugging. However, the DAG information is lost after function calls and multiple iterations of a loop with FPSanitizer. Similarly, the Herbgrind's metadata to produce DAGs grows linearly with the number of dynamic instructions and the program crashes with out-of-memory errors for almost all programs beyond unit tests.

In contrast, the directed acyclic graphs reported by EFTSanitizer span multiple functions and provides information about functions that have already completed execution (*i.e.*, no longer in the set of active stack frames) and also across multiple iterations of the same loop. We develop novel methods to manage the metadata for FP values in registers and in memory to generate such DAGs while having low performance/memory overheads. Figure 3 compares the DAGs generated by EFTSanitizer and FPSanitizer for a sample program for illustration.

Selective shadow execution. Developers of many long-running scientific applications prefer to test and debug numerical errors in specific parts of the application rather than the entire execution.



```
1  float compute(float a, float b){
2   return (((a + b) - b) - a);
3  }
4  int main(){
5   float a = 0.5f;
6   float b = 0.0013f;
7   float c = compute(a, b);
8   float d = c * 200000000.0f;
9   printf("d:%e\n", d);
10 }
```

Fig. 3. (a) A sample program to illustrate differences in the DAGs generated by EFTSanitizer and our prior work FPSanitizer [Chowdhary et al. 2020a] for the variable d after executing line 8. Here, 0.013 is not exactly representable as a 32-bit float. This rounding error is amplified by other operations. (b) The DAG generated by FPSanitizer where the DAG information is lost after the function call to `compute` completes. (c) The DAG generated by EFTSanitizer. Each node in DAG reports the operation, oracle real value (R), computed value (C), and the error for that node. The oracle real value is computed as the sum of the error and the computed value with FP arithmetic in EFTSanitizer. The oracle real value is computed using the high-precision MPFR library in FPSanitizer.

EFTSanitizer supports such selective shadow execution, which requires us to distinguish memory locations that have been previously accessed by the shadow execution and those that have not been previously accessed by the shadow execution. To distinguish such cases and facilitate such selective shadow execution, we maintain the floating-point value in the metadata along with the error. On a memory access, we check if the FP value produced in the program and the FP value in the metadata for the load instruction are identical. If they are identical, then the memory location was previously written by the shadow execution. Otherwise, we start tracking the rounding error from that point in time. This support for selective shadow execution enables EFTSanitizer to not only debug entire executions but also small fragments of a large program. It also helps in interacting with third-party libraries that cannot be recompiled with EFTSanitizer.

In summary, the use of EFTs as an oracle, the design of the metadata to provide rich traces of instructions to highlight the accumulation of rounding errors, and selective shadow execution enables EFTSanitizer to detect errors with long-running applications.

3.1 Metadata Design and Organization of the Metadata Space

Given that EFTSanitizer performs compile-time instrumentation, the FP values are resident either in memory locations or in registers/temporaries. To perform inlined shadow execution, we need to store the propagated error and some additional information with each memory location and temporaries (stack allocated variables or registers) that hold an FP value. The lifetime of the FP values in memory locations and in temporaries are different. Hence, we design different metadata spaces for FP values in memory and for those in temporaries.

Organization of the metadata space. We store metadata for FP values in memory in shadow memory, which is organized as a hash map. The lookup of the metadata is performed using the address of the memory access. For temporaries with FP values, we maintain the metadata in a small

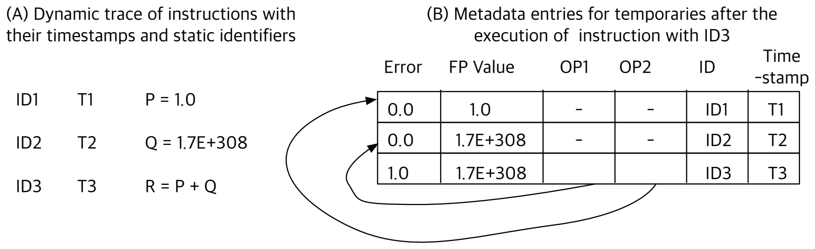

(A) Dynamic trace of instructions with their timestamps and static identifiers
ID1 T1 P = 1.0
ID2 T2 Q = 1.7E+308
ID3 T3 R = P + Q
(B) Metadata entries for temporaries after the execution of instruction with ID3
Error
FP Value
OP1
OP2
ID
Time -stamp
0.0 1.0 - - ID1 T1
0.0 1.7E+308 - - ID2 T2
1.0 1.7E+308 ID3 T3

Fig. 4. The metadata maintained with each temporary (and also with each memory location). (A) We show the dynamic trace of the executed instructions with the timestamp of the execution and the static identifier of the instruction. (B) We show the temporary metadata entries after the execution of instruction with ID3 (*i.e.*, R = P + Q). Here, we show pointers in the metadata space with arrows from the field to their corresponding metadata entries.

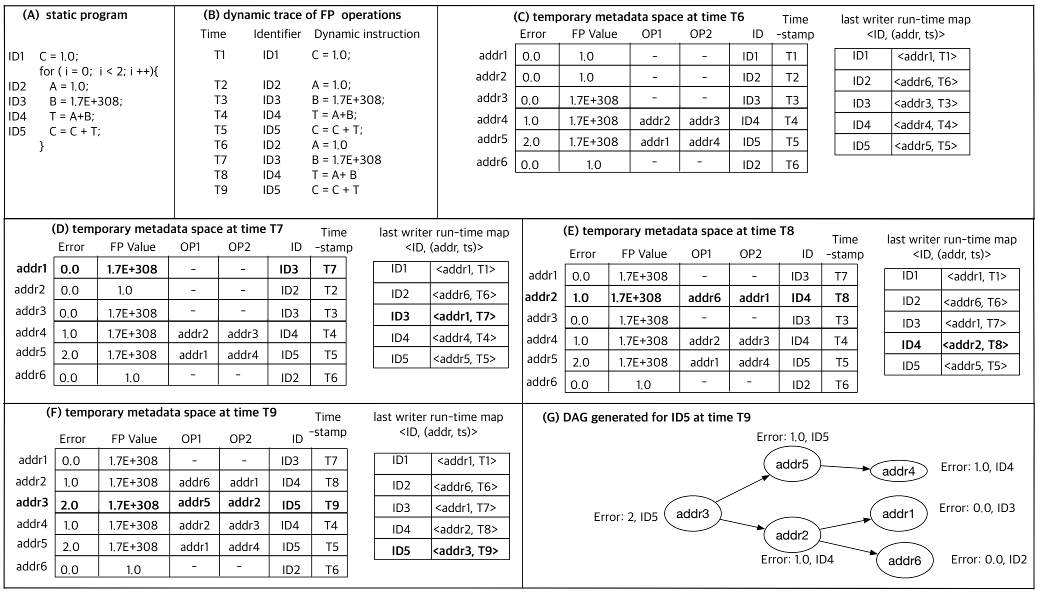
circular queue, which we call the temporary metadata space. Given a temporary that has an FP value, we need to maintain the mapping between the temporary and its corresponding entry in the temporary metadata space. We maintain a runtime map, which we call the last writer map, that maps the temporary that holds an FP value to its entry in the temporary metadata space.

One requirement on any such map is that we do not want this runtime map to grow proportional to the number of dynamic instructions. In the context of shadow execution, we observe that we just need to know the last writer to a temporary for establishing dataflow from the definition of the temporary to its use. We use the static instruction identifier generated by the compiler for the temporary to index into the last writer map.

What should we store in each metadata entry? To detect numerical errors, we store the propagated rounding error that is computed using error free transformations. We use the 64-bit double format to store the propagated rounding error. To facilitate selective shadow execution, we store the FP value generated by the program in the metadata entry. To produce a directed acyclic graph that highlights the accumulation of rounding error, we also store the pointers to the temporary metadata space entries of the operands. As the temporary metadata space entries can be reused, we store a monotonically increasing timestamp in the metadata entry to detect instances of reuse of the temporary metadata entries. We also store the compiler generated static instruction identifier of the instruction producing the FP value in the metadata entry to help debugging.

Figure 4 illustrates the information maintained with each metadata entry. When EFTSanitizer finds an instruction that exceeds the error threshold set by the user or observes an exceptional condition, it produces a DAG of instructions that shows error propagation by following the pointers to the operands in the metadata space.

Reusing the entries of the temporary metadata space. The number of temporaries generated at runtime by the program is proportional to the number of dynamic instructions. To keep the memory usage bounded for the temporary metadata space, we use a circular queue of a fixed size (*i.e.*, say k entries). When the queue becomes full, the next instruction that produces an FP value as the temporary uses the slot of the next entry in the queue (*i.e.*, the entry which was previously used for the oldest instruction in the temporary metadata space). Given the reuse of the temporary metadata space entries, the DAG generated to highlight the propagation of rounding error has at most k entries, where k is the size of the circular queue used for the temporary metadata space. Unlike prior work such as FPSanitizer [Chowdhary et al. 2020a], the metadata entries can



(A) static program

```
ID1   C = 1.0;
      for ( i = 0;  i < 2; i ++){
ID2     A = 1.0;
ID3     B = 1.7E+308;
ID4     T = A+B;
ID5     C = C + T;
      }
```

(B) dynamic trace of FP operations

Time	Identifier	Dynamic instruction
T1	ID1	C = 1.0;
T2	ID2	A = 1.0;
T3	ID3	B = 1.7E+308;
T4	ID4	T = A+B;
T5	ID5	C = C + T;
T6	ID2	A = 1.0
T7	ID3	B = 1.7E+308
T8	ID4	T = A+ B
T9	ID5	C = C + T

(C) temporary metadata space at time T6

	Error	FP Value	OP1	OP2	ID	Time -stamp
addr1	0.0	1.0	-	-	ID1	T1
addr2	0.0	1.0	-	-	ID2	T2
addr3	0.0	1.7E+308	-	-	ID3	T3
addr4	1.0	1.7E+308	addr2	addr3	ID4	T4
addr5	2.0	1.7E+308	addr1	addr4	ID5	T5
addr6	0.0	1.0	-	-	ID2	T6

last writer run-time map <ID, (addr, ts)>

ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr3, T3>
ID4	<addr4, T4>
ID5	<addr5, T5>

(D) temporary metadata space at time T7

	Error	FP Value	OP1	OP2	ID	Time -stamp
addr1	**0.0**	**1.7E+308**	-	-	**ID3**	**T7**
addr2	0.0	1.0	-	-	ID2	T2
addr3	0.0	1.7E+308	-	-	ID3	T3
addr4	1.0	1.7E+308	addr2	addr3	ID4	T4
addr5	2.0	1.7E+308	addr1	addr4	ID5	T5
addr6	0.0	1.0	-	-	ID2	T6

last writer run-time map <ID, (addr, ts)>

ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	**<addr1, T7>**
ID4	<addr4, T4>
ID5	<addr5, T5>

(E) temporary metadata space at time T8

	Error	FP Value	OP1	OP2	ID	Time -stamp
addr1	0.0	1.7E+308	-	-	ID3	T7
addr2	**1.0**	**1.7E+308**	**addr6**	**addr1**	**ID4**	**T8**
addr3	0.0	1.7E+308	-	-	ID3	T3
addr4	1.0	1.7E+308	addr2	addr3	ID4	T4
addr5	2.0	1.7E+308	addr1	addr4	ID5	T5
addr6	0.0	1.0	-	-	ID2	T6

last writer run-time map <ID, (addr, ts)>

ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr1, T7>
ID4	**<addr2, T8>**
ID5	<addr5, T5>

(F) temporary metadata space at time T9

	Error	FP Value	OP1	OP2	ID	Time -stamp
addr1	0.0	1.7E+308	-	-	ID3	T7
addr2	1.0	1.7E+308	addr6	addr1	ID4	T8
addr3	**2.0**	**1.7E+308**	**addr5**	**addr2**	**ID5**	**T9**
addr4	1.0	1.7E+308	addr2	addr3	ID4	T4
addr5	2.0	1.7E+308	addr1	addr4	ID5	T5
addr6	0.0	1.0	-	-	ID2	T6

last writer run-time map <ID, (addr, ts)>

ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr1, T7>
ID4	<addr2, T8>
ID5	**<addr3, T9>**

(G) DAG generated for ID5 at time T9

$$\begin{tikzcd}
 & \text{addr5} \arrow[r] & \text{addr4} \\
\text{addr3} \arrow[ru] \arrow[rd] & & \text{addr1} \\
 & \text{addr2} \arrow[ru] \arrow[r] & \text{addr6}
\end{tikzcd}$$

Fig. 5. Illustration to highlight the reuse of the temporary metadata space entries. We assume the there are only 6 entries in the temporary metadata space for this illustration. (A) An example program with loops. (B) Dynamic execution trace of the program in (A). We present the static instruction identifier and the time at which the program is executed. Multiple dynamic instances of the same instruction will have the identifier. (C) The snapshot of the temporary metadata space at time T6. We also show the last writer runtime map that maps a static instruction to its temporary metadata entry and the timestamp at which it was written. (D) Snapshot of the temporary metadata space after the operation B=1.7E+308 at time T7. The changes to the temporary metadata entries are highlighted in bold. (E) Temporary metadata space after the operation T=A+B at time T8. (F) Temporary metadata space after the operation C= C+T at time T9. (G) The DAG generated for the instruction with ID5 at time T9. From the last writer's map, it is mapped to addr3. When we follow the operand nodes while creating the DAG entries, we check if the operand's timestamp is greater than the current instruction's timestamp. If so, we do not print that node. For example, when we access the metadata entry at addr4 that was written at time T4, it operands are at addr2 and addr3. The timestamp of metadata entry at addr2 is greater than T4 because of the reuse of temporary metadata space entries. Hence, we do not print the operands of the entry at addr4.

span function calls that have already completed execution and multiple loop iterations, which significantly helps in the task of debugging numerical errors.

Given that the metadata entries in the temporary metadata space can be reused, we need a mechanism to map a temporary to its metadata entry. Subsequently when we generate the DAG on observing significant rounding error, we need to stop following the pointers to the operands when the metadata entries are reused. We use a map, which we call last writer runtime map, that maps a static instruction that produces a temporary to its metadata entry and the timestamp of the instruction when it was written. Multiple dynamic instances of the same static instruction can be present in the temporary metadata space. They will be linked as the operands of the other temporary metadata space entries. It is important to note that the last writer runtime map only maintains information about the last writer for a given static instruction.

When an instruction with compiler generated static identifier ID produces a temporary, a new entry for the instruction is created in the temporary metadata space. We add the address of the newly created metadata entry and the current timestamp to the last writer runtime map corresponding to ID. Next, we need to populate the operands for the newly created metadata entry. First, we check if the metadata for the operand is still available by checking last writer runtime map to obtain a tuple `(addr, ts)` for the operand, where `addr` is the address of the temporary metadata entry for the operand and `ts` is the timestamp when the operand was written to the temporary metadata space. Now, we check if the timestamp in the metadata entry at address `addr` is equal to `ts` from the last writer runtime map. If so, the metadata for the operand is available. Otherwise, the operand's metadata is not available because the operand's metadata entry has been reused. We use the `null` value for the operand's metadata entry. The DAG generated to highlight the propagation of rounding errors will be limited until this operand.

The use of monotonically increasing timestamps and the last writer runtime map that maps a temporary to its metadata entry enables us to keep the temporary metadata space bounded (*i.e.*, by reusing entries) and still provide DAGs proportional to the number of entries in the temporary metadata space.

Further when we print the DAG to highlight the propagation of errors, we follow the operands of an instruction I when the timestamp in the metadata entry of the operands is smaller than the timestamp of instruction I. This idea of using timestamps in the metadata and the last writer runtime map is inspired by the lock-and-key approach for detecting temporal memory safety errors in CETS [Nagarakatte 2012; Nagarakatte et al. 2010]. Rather than maintaining explicit lock locations and keys, we accomplish the detection of metadata reuse with timestamps and the last writer runtime map. Unlike CETS, this decision to detect reuse with timestamps and the last writer runtime map ensures that size of the runtime map is proportional to the number of static instructions in the program rather than number of dynamic instructions (number of active memory allocations in the case of CETS).

3.2 Metadata Propagation

We now describe when the metadata is created and how it is propagated with various FP operations, load/store operations, and function calls. We show the added instrumentation in the shaded region in the code snippets below.

Initialization with compile-time constants. When the program generates a new temporary and initializes it with a constant, we create a new temporary metadata space entry. The temporary metadata space is internally represented as a circular queue implemented in an array. It just wraps around after reaching the end of the array. We add the address of the new metadata entry and the timestamp to the last writer runtime map corresponding to the compiler generated identifier associated with this instruction. Compile-time constants are assumed to have zero error. Hence, we initialize the rounding error as zero. We set the FP value in the metadata entry to the initialized constant. We also set the operands in the metadata entry to `null`.

```
double x = 1.0; // instruction with compiler generated identifier x_id
```

```
x_meta   = allocate_temporary_metadata_space_entry();
x_meta -> error = 0.0;
x_meta -> fpvalue = 1.0;
x_meta -> operand1 = null;
x_meta -> operand2 = null;
x_meta -> id = x_id;
```

```
x_meta->timestamp = timestamp++;
last_writer_map.insert(x_id, <x_meta, x_meta->timestamp>);
```

FP Operations. Given that EFTSanitizer operates on the intermediate representation of the compiler, all FP operations are either binary operations or unary operations. Further, memory accesses happen with explicit load and store instructions. When an FP operation produces a value in a temporary, we first allocate a new metadata entry in the temporary metadata space. We retrieve the metadata of the operands by looking up the last writer runtime map. The metadata entries of the operands might have been reused. Hence, we check if the timestamp in the last writer runtime map matches the timestamp at the metadata entry. If so, the metadata entries are valid. Otherwise, we do not have information about the operands. In such cases, we assume that the operands do not have any error. We also set the operands in the metadata entry for the current operation as shown below.

When the operand's metadata entries have not been reused, we read the error from the metadata entries for the operands and compute the propagated rounding error using error free transformations as described in Section 2.2. We use PropSumError in the listing below to compute the propagated rounding error after addition. For operations without corresponding EFTs, we use the high-precision computation using the MPFR library to compute the error.

```
double z = x + y; // with identifer z_id and operand identifiers x_id and y_id
```

```
z_meta = allocate_temporary_metadata_space_entry();
<x_meta, x_ts> = last_writer_map(x_id);
<y_meta, y_ts> = last_writer_map(y_id);
// check if x and y metadata entries are valid
z_meta->op1 = (x_meta->timestamp != x_ts) ? NULL: x_meta;
z_meta->op2 = (y_meta->timestamp != y_ts) ? NULL: y_meta;
x_error = (x_meta->timestamp != x_ts) ? 0.0: x_meta->error;
y_error = (y_meta->timestamp != y_ts) ? 0.0: y_meta->error;
z_meta-> error = PropSumError(x, x_error, y, y_error);
z_meta->fpvalue = z;
z_meta->id = z_id;
z_meta->timestamp = timestamp++;
last_writer_map.insert(z_id, <z_meta, z_meta->timestamp>);
```

Handling stores of FP values to memory. When we store a FP value to a memory, we need to propagate the metadata to memory locations. Each memory location that holds an FP value is shadowed with metadata in shadow memory. We first obtain the temporary metadata space entry of the FP operand that is being stored to memory. We check if the temporary metadata entry is still valid. If so, we copy the temporary metadata space entry to a shadow memory location corresponding to the address where the FP value is being stored.

```
*x = y; // x's type is double* and y's type is double with ID: y_id
```

```
<y_meta, y_ts> = last_writer_map(y_id);
shadow_addr  = shadow_memory(x);
memcpy(shadow_addr, y_meta, SIZE);
timestamp++;
```

Here, SIZE is the size of the metadata space entry.

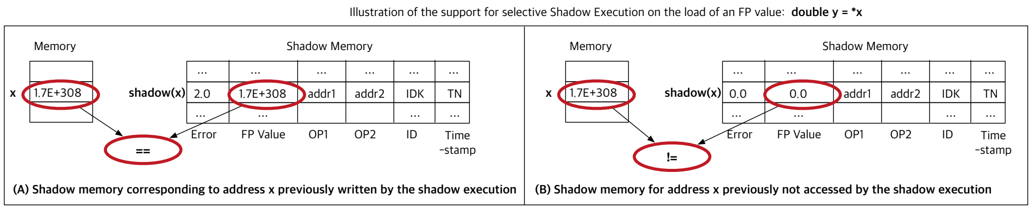


Illustration of the support for selective Shadow Execution on the load of an FP value: double y = *x
Memory
Shadow Memory
x
1.7E+308
shadow(x)
2.0
1.7E+308
addr1
addr2
IDK
TN
Error
FP Value
OP1
OP2
ID
Time -stamp
==
(A) Shadow memory corresponding to address x previously written by the shadow execution
0.0
0.0
!=
(B) Shadow memory for address x previously not accessed by the shadow execution

Fig. 6. Illustration of the support for selective shadow execution in EFTSanitizer. When the program loads an FP value load from an address x, we check if the FP value in the metadata entry in the shadow memory corresponding to address x is equal to the FP value produced by the program. (A) If the shadow execution has previously written to the metadata entry, then the FP values in the program and in the metadata entry will match. (B) When the shadow memory for the location has not been previously updated (*i.e.*, it happened in uninstrumented code), the FP values produced by the program and the one in shadow memory will not match. We consider the value not to have any error and start error propagation from that point.

Handling the load of an FP value from memory. On every load operation that loads an FP value, we read the metadata from shadow memory corresponding to the address where the FP value is being loaded. Since we want to enable selective shadow execution from an arbitrary point in time, the metadata in the shadow memory corresponding to the address may not have been written previously by the shadow execution. As we store the FP value previously produced by the program in the metadata entry, we check if the FP value in the metadata entry and the one produced by the program match. If so, the shadow memory entry was previously written by the shadow execution and we copy the metadata entry from shadow memory to the temporary metadata space (*i.e.*, with memcpy). If the FP value in shadow memory and the FP value produced by the program do not match, then we create a new temporary metadata space entry, initialize the error to 0.0, and initialize the FP value with the value produced by the program. Figure 6 illustrates the check performed to support selective shadow execution from an arbitrary point in the execution.

```
y = *x // where x's type is double * and y's identifier is y_id
```

```
y_meta = allocate_temporary_metadata_space_entry();
shadow_addr = shadow_memory(x);
if(shadow_addr->fpvalue == y){    //check for selective shadow execution
  memcpy(y_meta, shadow_addr, SIZE);
}
else{
  y_meta-> error = 0.0;
  y_meta-> fpvalue = y;
  y_meta->op1 = null;
  y_meta->op2 = null;
}
y_meta->id = y_id;
y_meta->timestamp = timestamp++;
last_writer_map.insert(y_id, <y_meta, y_meta->timestamp>);
```

Metadata propagation with function arguments and returns. We use a shadow stack to propagate the metadata for arguments and return values. Our compiler adds instrumentation at the call site to add metadata entries for arguments in the shadow stack. The compiler adds instrumentation in the beginning of the callee to retrieve the metadata for the arguments. Similarly, the compiler also adds instrumentation to propagate the metadata for return values. This method of propagating the metadata for arguments and return values using the shadow stack enables us to handle both regular function calls and calls through function pointers (*i.e.*, indirect calls).

3.3 Error Reporting and Debugging Interface

Most FP instructions have some rounding error. They do not change the program's output. When a variable (x) is marked by the user as a variable of interest, EFTSanitizer checks if the FP value produced by the program when added to the error (δ_x) in the metadata for the variable x significantly differs from the value produced by the program. In essence, if $x + \delta_x$ is significantly different from x while using FP arithmetic operations, we report it to the user. Recall, $x + \delta_x$ in FP arithmetic is equal to x while rounding a primitive operation. The user provides the threshold for an error to be considered significant. By default, EFTSanitizer reports errors where all bits (both fraction and exponent bits) between x and $x + \delta_x$ are different. In such cases, EFTSanitizer generates a DAG to highlight the propagation of the error.

When FP values are used in branches, EFTSanitizer checks if adding the error to the FP value changes the result of the branch predicate and reports such branch divergences along with the respective DAGs to the user. Finally, EFTSanitizer checks the FP value to determine if it is a special value such as a NaN or an infinity. EFTSanitizer provides the DAG for the first instance where such NaNs or infinities occur because any operation on a NaN results in a NaN.

To debug interactively using debuggers such as gdb, EFTSanitizer provides publicly exported auxiliary functions that can be used by the user to examine the metadata with breakpoints and watchpoints.

Illustrative example. We illustrate the propagation of metadata with a simple example where all operations are performed with temporaries in Figure 5. To illustrate reuse, we show the temporary metadata space with 6 entries. We show the updates to the timestamp and the last writer runtime map after adding each instruction.

4 PROTOTYPE AND IMPLEMENTATION

Prototype. We built a prototype of EFTSanitizer as a module pass of the LLVM-10 compiler infrastructure. EFTSanitizer takes as input C/C++ programs and generates the LLVM intermediate representation (IR) of the input program using the Clang++ frontend. The instrumentation is performed over the LLVM IR. All instrumentation for the computation of error using EFTs, metadata propagation, and metadata creation is inlined by EFTSanitizer's compiler instrumentation to reduce the overhead of function calls. The snippets of code that are not inlined correspond to the initial creation of shadow memory and temporary metadata space, which is done with calls to the `mmap` function in the runtime. EFTSanitizer uses a module pass rather than a function pass because we need to provide unique compile-time identifiers to all instructions in the program. We can support separate compilation by providing a unique starting identifier for each translation unit.

The LLVM IR is in static single assignment form, which partly helps the identification of the last writer (*i.e.*, definition) for any variable. When the FP value is involved in a PHI node, we define corresponding PHI nodes that maintain the pointer to the temporary metadata space entry.

By default, we handle elementary functions, which is provided by math libraries, using the corresponding versions from the MPFR library [Fousse et al. 2007]. Similarly, we use the MPFR versions of the operation for LLVM's intrinsics. The user can configure EFTSanitizer with a command-line option to use correctly rounded functions from the RLIBM project [Aanjaneya et al. 2022; Lim et al. 2020, 2021; Lim and Nagarakatte 2021a,b,c, 2022] for the float type, CR-LIBM [Daramy et al. 2003; Daramy-Loirat et al. 2006] for the double type, and glibc's libraries for the double-double type when the corresponding functions are available. Using correctly rounded elementary functions improves performance.

EFTSanitizer is open source and is publicly available [Chowdhary and Nagarakatte 2022a,b].

Shadow memory, shadow stack, and temporary metadata space. We organize the shadow memory in EFTSanitizer as a best effort hash map (*i.e.*, a direct mapped cache) with 64 million entries (*i.e.*, 64* 1024*1024 entries). Further, shadow memory is allocated with the `mmap` system call. We use `MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE` flags, which just creates virtual memory mappings without reserving physical memory on Linux. Hence, the program experiences memory overhead only when it touches memory. Each metadata entry is 56 bytes as shown in Figure 4. The hash map is indexed by the memory address of the FP value. If two addresses map to the same entry in shadow memory (*i.e.*, a collision), the old entry will be overwritten with the new entry similar to a direct-mapped cache. Hence, it is a best-effort hash map.

In contrast to shadow memory, the temporary metadata space is organized as a circular queue implemented with an array. By default, it has 64 entries. Hence, the number of dynamic instructions in the directed acyclic graph is at most 64. The next slot to use (*e.g.*, to allocate a new entry in the temporary metadata space) is implemented as an increment operation modulo the size of the temporary metadata space.

We use a shadow stack of 16 entries to pass metadata for arguments and return values. We have not seen functions with more than 16 arguments in our evaluation. If necessary, these can be customized with a large size by the user. The shadow stack also detects implicit casts from FP values to integers and vice versa through incorrect function signatures.

5 EXPERIMENTAL EVALUATION

This section presents the results of our experimental evaluation. We attempt to answer two questions. First, is EFTSanitizer effective in the task of debugging numerical errors? Second, is EFTSanitizer efficient compared to other state-of-the-art tools?

Methodology. To measure the effectiveness of EFTSanitizer in detecting numerical errors, we use a collection of 46 tests with known numerical errors from correctness test suites of FPSanitizer and Herbgrind. For these test suites, we compared the results with FPSanitizer and Herbgrind. These tests are 50-100 lines of code, which can be executed by all three tools. We also developed a suite of algorithms widely used in numerical methods (*e.g.*, Gaussian elimination with partial pivoting). To demonstrate the usability of EFTSanitizer with large applications and to perform performance evaluation, we use C/C++ FP applications from the SPEC-2006 and SPEC-2017 suites, applications from the Lawrence Livermore National Laboratory's (LLNL) Coral benchmark suite, and NAS-3.0 benchmarks.

We generated two versions of EFTSanitizer: (a) tracing mode of EFTSanitizer that generates DAGs and (b) a non-tracing mode where it just detects errors but does not produce DAGs. The difference between these two versions is the number of fields in the metadata entry. In the non-tracing mode, the metadata entry in both the temporary metadata space and shadow memory does not maintain information about that operands and the timestamp.

For our performance experiments, we perform shadow execution for the entire execution. EFT-Sanitizer can also be executed with selective shadow execution where the overhead can be significantly lower than for the entire execution.

To facilitate the comparison of EFTSanitizer with FPSanitizer, we use the publicly available version of FPSanitizer [Chowdhary et al. 2020b]. The publicly available version of FPSanitizer did not support C++ applications. We added C++ support to FPSanitizer, which enabled us to compare FPSanitizer and EFTSanitizer with C++ applications. We configured FPSanitizer to use 128-bits of precision for the MPFR value in our experiments. We chose 128-bits because the

Table 1. Summary of our experiments to detect and debug numerical errors in various scientific computing applications and NAS Parallel Benchmarks 3.0 [NAS 2022] with EFTSanitizer. The table reports the various kinds of errors that EFTSanitizer detects for these applications (high rounding error, NaNs, infinities-Inf, and divergences in branch outcomes). We report the number of dynamic instances of such error and the unique static program locations that correspond to these dynamic instances. We also report the overall performance overhead of EFTSanitizer to detect and generate DAGs for these applications when compared to an uninstrumented program. In these experiements, we report high rounding error when the FP value produced by the program has 45-bits in error when compared to the sum of the FP value and the propagated rounding error in shadow execution.

Benchmark	Rounding Errors		Inf		NaN		Branch Flips		Overhead
	Dyn.	Stat.	Dyn.	Stat.	Dyn.	Stat.	Dyn.	Stat.	
HPCG	147	4	0	0	0	0	118	3	14.95×
Laghos	0	0	40800	2	0	0	553	1	2.42×
Quicksilver	0	0	0	0	0	0	0	0	9.06×
LULESH	285246131	3	0	0	138	1	373615808	33	23.69×
Kripke	0	0	0	0	0	0	0	0	1.05×
AMG	0	0	5	5	0	0	0	0	2.57×
NAS BT	20	4	0	0	0	0	10	4	33.79×
NAS CG	31	3	0	0	0	0	1	1	13.62×
NAS EP	2	2	0	0	0	0	0	0	3.58×
NAS FT	2	2	0	0	0	0	0	0	16.13×
NAS IS	0	0	4	4	0	0	0	0	7.97×
NAS LU	26	5	0	0	0	0	11	3	45.86×
NAS MG	96	2	0	0	0	0	0	0	11.58×
NAS SP	89020284	4	0	0	0	0	1041064223	15	15.42×

effective data type size with both FPSanitizer and EFTSanitizer will be identical. In EFTSanitizer, we use double precision both for the FP value and the error in the metadata (*i.e.*, 64 + 64 = 128-bits). Herbgrind crashed with out-of-memory errors on almost all the applications used for the performance experiments. Hence, we do not report Herbgrind for the performance experiments.

Our experiments are performed on a 2.10GHz Intel Xeon Gold 6230R machine with 192GB of RAM running Ubuntu 20.04.4. We measure the wall clock execution time of the application with shadow execution frameworks and with the uninstrumented application. We repeated the experiments multiple times to minimize the noise in the performance experiments. We report the number of bits of the result that are erroneous compared to the oracle, which is a double precision value. For example, when we say 52-bits of error in the rest of evaluation, the FP value represented in double precision and the sum of the FP value and the propagated rounding error in double precision differ in the least significant 52-bits (*i.e.*, all precision bits are wrong).

5.1 Effectiveness in Detecting and Debugging Numerical Errors

Evaluation with existing correctness suites. When we evaluated EFTSanitizer with the correctness suite with 46 C/C++ micro-benchmarks, it detected all errors in these micro-benchmarks without false positives. Among these 46 micro-benchmarks, 21 of them have some numerical errors (*i.e.*, 19 of them have high rounding error where all the precision bits are wrong, 2 of them produce infinities, and the rest do not have any numerical error). This experiment with micro-benchmarks shows that EFTSanitizer detects numerical errors similar to existing shadow execution tools.

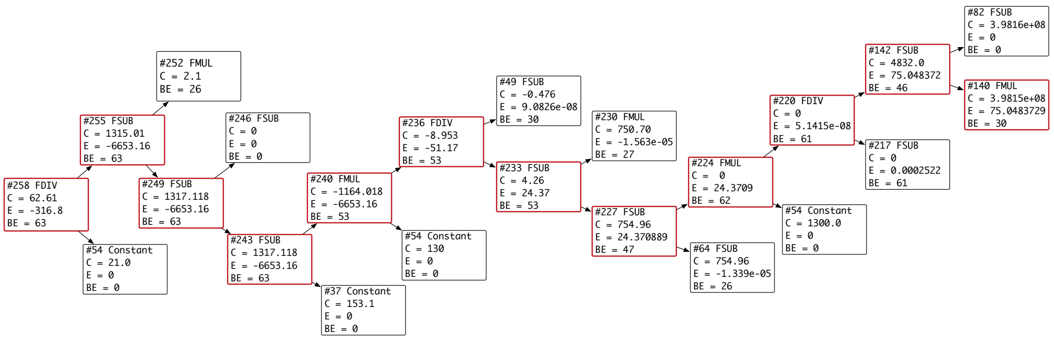


Fig. 7. The DAG generated by EFTSanitizer for debugging the root cause of the error for the case study with GEPP. Each DAG node show the instruction opcode, computed value, propagated rounding error with EFTs, and the number of bits in error in the computed value in comparison to the shadow execution. The node #258 (FDIV) produces the observed wrong result. The root cause of this error is caused by instruction at node #142 (FSUB).

Detecting numerical errors in applications. Beyond micro-benchmarks, we also evaluated EFTSanitizer by executing it with various applications from the LLNL application suite and NAS parallel benchmarks. In this process, we detected previously unknown bugs in many applications. Table 1 summarizes our experiments in finding numerical errors in long-running applications. Table 1 reports the total number of dynamic and static instances with more than 45-bits of error in the results, NaNs, infinities, differing branch outcomes, conversion errors, and total overhead we have experienced with EFTSanitizer. Multiple instances of the dynamic instruction can be mapped to the same static instruction. By default, EFTSanitizer generates rounding errors for variables of interest such as return values, arguments of system calls, and input/output routines. It can be configured to generate an error report for any FP instruction. We also identified new floating-point exceptions (NaN/infinities) in four applications: Laghos [LLNL 2022b], Lulesh [Karlin et al. 2013], AMG [LLNL 2022a], and NAS IS. The DAGs provided by EFTSanitizer helped us reason about the propagation of errors and identify the potential reason for these exceptions.

Case study 1: Debugging a numerical error in Gaussian elimination with partial pivoting (GEPP). In this case study, we demonstrate that we can detect and debug high rounding errors using EFTSanitizer much more productively than previous shadow execution tools using high-precision computations. More importantly, we show how the DAGs reported are effective to debug the bug. We chose this case study for illustration because it was previously used by the developers of the CADNA tool [Cadna 2022].

Gaussian elimination (GE) is a direct method to solve a system of linear equations of form $Ax = b$. In this method, a system of linear equations is represented by an augmented matrix [A | b] of size $N \times N + 1$, where N is the number of unknowns in the system of linear equations. In this method, matrix A is reduced to the upper triangular matrix using row operations followed by back-substitution. Due to rounding errors with FP arithmetic, the GE method can return wrong results. Using partial pivoting with the GE method can reduce the rate of increase in the error. In Gaussian elimination with partial pivoting (GEPP), the maximum absolute value in the first column, which is called the pivot, is selected. If the first row does not have the maximum absolute value, the row containing the pivot is swapped with the first row. This technique is shown to reduce numerical errors. However, if the pivot is influenced by rounding error, it can also lead to wrong results.

Let us consider the code snippet below (from [Cadna 2022]).

$$A = \begin{bmatrix} 21.0 & 130.0 & 0.0 & 2.1 \\ 13.0 & 80.0 & 4.74E+8 & 752.0 \\ 0.0 & -0.4 & 3.9816E+8 & 4.2 \\ 0.0 & 0.0 & 1.7 & 9E-9 \end{bmatrix} \tag{1}$$

$$b = \begin{bmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6E-8 \end{bmatrix} \tag{2}$$

When we implement GEPP using the 32-bit float format, we get the following solution:

$$x = \begin{bmatrix} 62.62 \\ -8.95 \\ 0.00 \\ 1.0 \end{bmatrix} \tag{3}$$

This solution does not match the reference output of the program. To diagnose the cause of wrong results and to debug it, we ran the float version of this program with EFTSanitizer. It detected the error and the DAG of instructions generated by EFTSanitizer for the 32-bit float version corresponding to the first element of the column vector x is shown in Figure 7. The first element with the float version produces 62.62, which is the wrong result. The root node of the DAG (#258 FDIV in Figure 7) has 63-bits of error. We analyzed the nodes of the DAG with significant error. While following the nodes with significant rounding error, we identified that during the elimination of variables, A[3][3] is computed as 4832 (*i.e.*, node #142 in Figure 7). The error in this computation is the primary cause of the wrong result.

In contrast to EFTSanitizer, the DAG generated by FPSanitizer is a single node because of the loss of DAG information with loop iterations, which is not useful in debugging this error. Although EFTSanitizer does not generate the exact real value due to the loss in precision while composing the errors, the error information was sufficient to detect and debug this error.

Case study 2: Debugging a numerical error in Laghos. Our goal with this case study is to show that EFTSanitizer is efficient and effective in debugging numerical errors in long-running applications. Laghos is an application from LLNL's CORAL suite. We discovered a divide-by-zero error in the Laghos benchmark. We used EFTSanitizer's exported API to debug the application using gdb. When we ran the program with EFTSanitizer, it reported a division by zero resulting in infinity (*i.e.*, the root node #4872 Div in Figure 8). EFTSanitizer generated the DAG of instructions shown in Figure 8. From analyzing the DAG and following the nodes in the DAG that have a large number of bits in error, we identified that this error was generated in the second iteration of the loop. The node that performs square root (#122 Sqrt in Figure 8) has produced a value of 0.0, which results in the eventual production of infinity. The DAG in Figure 8 does not report for any child nodes for this instruction probably due to reuse of the temporary metadata space entries. This square root operation corresponds to `laghos_solver.hpp:122`, which can be obtained from the identifier. We set a breakpoint in laghos_solver.hpp:122 and generated the DAG for the square root operation and identified the root cause of the bug.

In contrast, the DAG generated by FPSanitizer was a single node because the DAG information was lost across iterations. Similarly, we tested this application with FPSpy [Dinda et al. 2020] whose tracing information was not helpful in diagnosing the root cause of this problem. The DAG

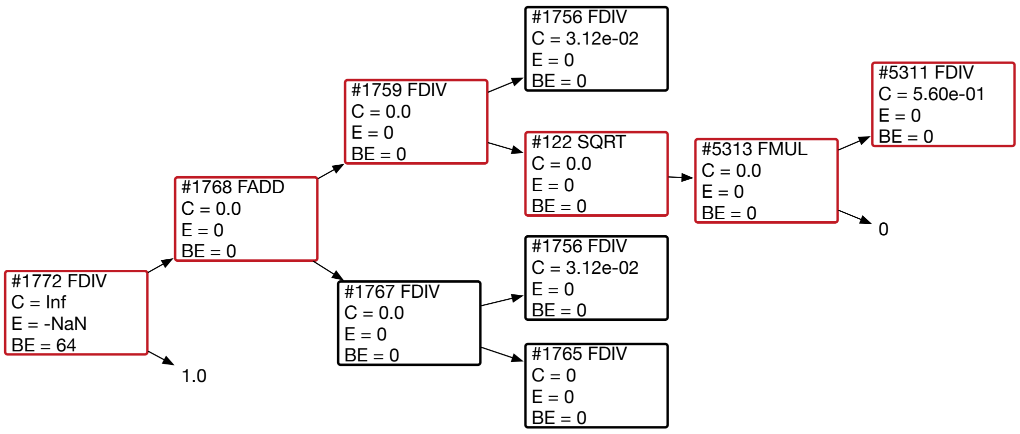


Fig. 8. DAG of instructions generated by EFTSanitizer for the Laghos benchmark to diagnose the reason for the infinity. The node #122 (SQRT) instruction that produces a result of 0.0 is the root cause of the error.

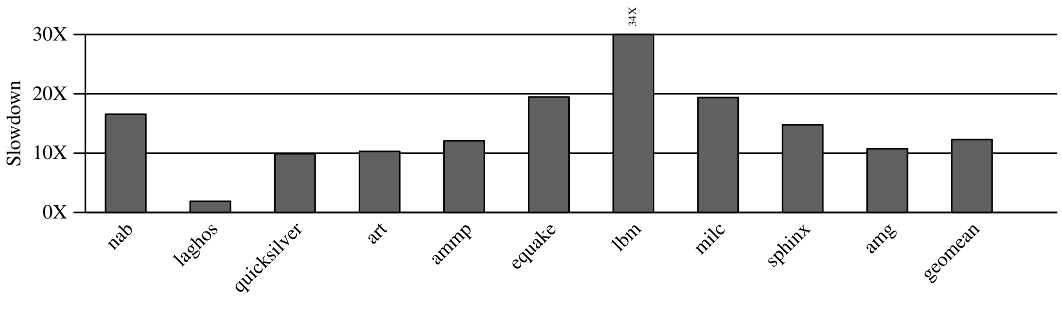


Fig. 9. This figure reports the slowdown with EFTSanitizer in the tracing mode compared to a baseline without any instrumentation.

generated by EFTSanitizer was helpful to debug the root cause of the error, validate the bug, and create a reduced test case.

5.2 Performance Evaluation

Performance overhead of EFTSanitizer. We measure the total slowdown of EFTSanitizer compared to a baseline without any instrumentation for shadow execution. Figure 9 reports the EFTSanitizer's slowdown in the tracing mode that produces DAGs compared to a baseline (*i.e.*, total height of each bar). On average, EFTSanitizer slowdowns the program by 12.3× in comparison to an uninstrumented application. A variant of EFTSanitizer that uses correctly rounded elementary functions instead of the MPFR math library reduces the total performance overhead from 12.3× to 11.3×. This variant detects all errors in the correctness test suite described earlier.

To understand the source of the overheads, we measured the overheads for computing the error using EFTs, performing checks for selective shadow execution, performing metadata propagation with load and store instructions, and copying metadata between shadow memory and the temporary

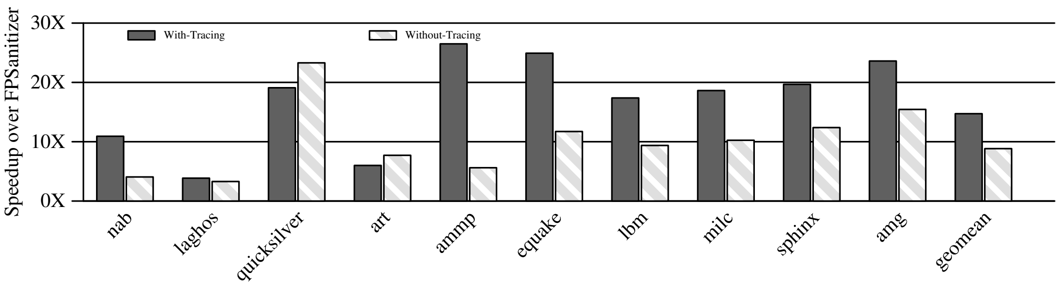


Fig. 10. The first bar of this graph shows the speedup achieved with EFTSanitizer when compared to shadow execution with FPSanitizer when tracing is enabled. The second bar of this graph shows the speedup achieved with EFTSanitizer compared to shadow execution with FPSanitizer when tracing is disabled.

metadata space. Computing the error using EFTs for primitive FP instructions slows down the execution by 1.71×.

For every load instruction, we compare the program's FP value and the FP value stored in shadow memory for selective shadow execution. If they mismatch, we reset the metadata with the FP's program value. Otherwise, we copy metadata from shadow memory to temporary metadata space. These operations performed on every load instruction (*i.e.*, check and metadata copy) introduces 5.20× slowdown. For each store instruction, we copy the metadata from temporary metadata space to shadow memory. Handling store instructions slows down the program by 0.44×. For each FP arithmetic instruction, we allocate the temporary metadata space entry and store the address and the timestamp in the last writer runtime map. For each operand of an FP instruction, we load the address of the temporary metadata space entry and the timestamp from the last writer runtime map to access the metadata of the operands. Together, performing the metadata updates on FP arithmetic operations introduces an additional 1.45× overhead. Handling other FP instructions such as FPToSIInst, FPToUIInst, and function arguments/returns introduces the remaining overheads.

EFTSanitizer's detection mode that does not produce DAGs slows down the execution by 6.55× on average compared to an execution without any instrumentation. We measured the slowdowns where we instrumented every FP operation in the program. EFTSanitizer's overhead is significantly lower when the user selects certain regions for selective shadow execution, which we found useful to debug numerical errors in long-running applications.

Speedup of EFTSanitizer over FPSanitizer. Figure 10 shows the speedup of EFTSanitizer's shadow execution when compared to FPSanitizer. We compare both EFTSanitizer and FPSanitizer with debugging support for DAGs (*i.e.*, tracing mode) and without support for DAGs where it just detects errors (*i.e.*, non-tracing mode). For each application, we report the speedup of EFTSanitizer's execution compared to the corresponding FPSanitizer's execution. On average, EFTSanitizer's execution tracing mode that provides debugging support is 14.7× faster than FPSanitizer's tracing mode. When we repurpose both tools where they detect errors but do not provide DAGs, EFTSanitizer was faster than FPSanitizer by 8.8× on average.

To understand the reason for these significant performance speedups, we measure the total user-mode dynamic instructions executed by the application instrumented with EFTSanitizer and FPSanitizer using Linux's `perf` utility when compared to a baseline without any instrumentation. Table 2 reports the dynamic instruction overhead with both EFTSanitizer and FPSanitizer with the tracing modes. On average, EFTSanitizer has a dynamic instruction overhead of 14×, which results in a performance overhead of 12.3×. In contrast, the dynamic instruction overhead with

Table 2. Dynamic instruction overhead with EFTSanitizer and FPSanitizer compared to a baseline without any instrumentation. We also report the geometric mean of the dynamic instruction overhead for both EFTSanitizer and FPSanitizer.

Application	Instruction Overhead	
	EFTSanitizer	FPSanitizer
nab	28×	320×
laghos	2×	12×
quicksilver	9×	152×
art	12×	153×
ammp	20×	489×
equake	19×	496×
lbm	26×	857×
milc	21×	450×
sphinx	15×	267×
amg	13×	401×
geomean	**14 ×**	**249 ×**

FPSanitizer is 249× on average. FPSanitizer executes an order of magnitude more instructions when compared to EFTSanitizer. This large instruction overhead with FPSanitizer is primarily due to the software simulation of FP operations by the MPFR library. In contrast, EFTSanitizer's computation of error with error free transformations, which that uses hardware FP operations, as the oracle provides significant speedup over FPSanitizer. In summary, EFTSanitizer is not only faster than FPSanitizer but also provides better debugging information to diagnose and debug errors.

6 RELATED WORK

There is a large body of prior work to detect and debug numerical errors. We focus our attention on closely related prior work.

Static analysis techniques [Barr et al. 2013; Darulova et al. 2018; Darulova and Kuncak 2014; de Dinechin et al. 2006; Delmas and Souyris 2007; Feliú et al. 2018; Ghorbal et al. 2012; Goubault 2001; Goubault et al. 2007; Solovyev et al. 2018; Zhang et al. 2020] use abstract interpretation or interval arithmetic to reason about numerical errors. Most static analysis tools provide a correct over-estimation of the rounding error that may occur in a program for the provided input ranges for the input variables. The inaccuracy of the deduced bounds is due to the use of abstract domains (such as intervals in the case of Gappa [de Dinechin et al. 2006] or zonotopes in the case of Fluctuat [Goubault et al. 2007]) or due to the error expression used to bound the error. Tools such as PRECiSA [Feliú et al. 2018] and FPTaylor [Solovyev et al. 2018] use a combination of symbolic error expressions and global optimization techniques to compute the round-off error bounds. They also provide formal guarantees in the form of proof certificates. In addition, PRECiSA uses abstract interpretation techniques to reason about conditionals and iterative structures. These tools provide more precise estimation than previous static analysis tools. Scaling these tools to large programs with easy specifications is still an open research problem.

Dynamic analysis techniques monitor the program behavior at run time for a single input. Such techniques compare the actual execution with some oracle. Depending on the oracle used, these techniques can be classified into heavy-weight techniques that comprehensively detect errors and light-weight techniques that detect specific errors. FPDebug [Benz et al. 2012], Herbgrind [Sanchez-Stern et al. 2018], FPSanitizer [Chowdhary et al. 2020a], and PFPSanitizer [Chowdhary and Nagarakatte 2021] are examples of approaches that use high precision computation (*i.e.*, using the MPFR library) as the oracle. FPDebug and Herbgrind perform binary instrumentation using Valgrind [Nethercote and Seward 2007]. In contrast, FPSanitizer instruments programs at compile time on the LLVM intermediate representation. Hence, it reduces the overheads compared to FPDebug. FPSanitizer also produces DAGs but loses DAG information after a function completes execution and with multiple iterations of a loop. PFPSanitizer requires the user to mark parts of the shadow execution that needs to be executed in parallel. Once the user provides such annotations, it reduces the overhead of shadow execution by running distinct parts of the shadow execution in parallel on multiple cores.

In contrast to shadow execution, BZ [Bao and Zhang 2013] and RAIVE [Lee et al. 2015] monitor the exponent of the operands and the result of the FP computation. Monitoring the exponents to detect numerical errors avoids using real numbers as an oracle. RAIVE monitors if the final output is affected by numerical errors and uses vectorization to reduce the overheads. FPSpy [Dinda et al. 2020] uses hardware condition flags and uses exception handling to detect FP errors in binaries. FPSpy incurs huge overheads if every FP instruction is monitored. However, the use of sampling can reduce the performance overheads. CADNA [Jézéquel and Chesneaux 2008] and Verrou [Févotte and Lathuilière 2016] use random rounding to detect the sensitivity of the program to rounding errors. Similarly, AtomU [Zou et al. 2019] uses condition numbers to detect numerical errors and instability in FP applications.

Prior work on EFTs. The idea of error free transformation is rather old and has been used in the past for compensated summation [Kahan 1965; Rump 2009], compensated Horner Scheme [Langlois et al. 2006], and robust geometric algorithms [Shewchuk 1996].

The Fast2Sum algorithm was first used in accurate summation [Kahan 1965] in 1965. Then it was described by Dekker [Dekker 1971] in 1971 as a technique to extend the precision. Fast2Sum requires three floating-point instructions and one branch instruction. Hence, it can be costly due to branch mispredictions. To avoid the branch instruction, TwoSum algorithm was introduced in [Knuth 1997]. TwoSum requires six floating-point instructions and no comparison of operands. Hence, TwoSum is cheaper than the Fast2Sum algorithm on modern machines. It has been shown that Fast2Sum and TwoSum algorithms are robust if underflow occurs. Fast2Sum is immune to overflow, and TwoSum is almost immune to overflow. For some corner cases, TwoSum can overflow when actual computation does not [Boldo et al. 2017]. Fast2Sum and TwoSum algorithms described in the paper work with the round-to-nearest rounding modes. Similar algorithms for different rounding modes have also been developed [Priest 1992].

A similar algorithm for multiplication was introduced by Dekker based on Veltkamp splitting [Muller et al. 2018]. Veltkamp splitting algorithm splits a floating-point number into two $\lceil p/2 \rceil$-bit numbers so that they can be multiplied without any error. Using Veltkamp splitting, Dekker [Dekker 1971] proposed an algorithm to compute the error of FP multiplication in 1971. However, Dekker's multiplication requires 17 floating-point instructions. An alternative algorithm 2MultFMA [Muller et al. 2018] using the *fma* instruction computes the error of FP instruction with just two instructions. Similarly, the error for division and the square root operations can be easily computed using the *fma* instruction [Boldo and Daumas 2003].

In the context of numerical error detection, EFTs have been recently used by Shaman [Demeure 2020]. Shaman uses EFTs as an oracle and implements a C++ library using operator overloading. SHAMAN is attractive for developing new applications and measuring their error. To use SHAMAN with an existing application, the user will need to rewrite the program to change the types of the variables and math functions. It will likely have higher overheads when additional debugging mechanisms are added. In contrast, EFTSanitizer does not require changing the source code and incurs low overheads when compared to Shaman. EFTSanitizer also provides debugging support by generating a DAG of instructions.

7 CONCLUSION

EFTSanitizer is the first approach for shadow execution that uses error free transformations as the oracle. Given that the accumulated rounding error is computed with hardware supported FP operations, it is significantly faster than prior approaches. The directed acyclic graph generated by EFTSanitizer spans multiple function calls and loop iterations, which we found extremely helpful in debugging numerical errors. EFTSanitizer includes a novel metadata management scheme that makes the resulting tool an order of magnitude faster than the state-of-the-art, enables selective shadow execution for arbitrary fragments of dynamic execution, and enables effective debugging of numerical errors.

ACKNOWLEDGMENTS

We thank OOPSLA reviewers, Laura Titolo, Matan Shachnai, Harishankar Vishwanathan, and Mridul Aanjaneya for their feedback. This material is based upon work supported in part by the National Science Foundation under Grant No. 1908798 and Grant No. 2110861. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2022. Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries. In *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'22)*. https://doi.org/10.1145/3519939.3523447

Tao Bao and Xiangyu Zhang. 2013. On-the-Fly Detection of Instability Problems in Floating-Point Program Execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 817–832. https://doi.org/10.1145/2509136.2509526

Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-Point Exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 549–560. https://doi.org/10.1145/2429069.2429133

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 453–462. https://doi.org/10.1145/2254064.2254118

S. Boldo and Marc Daumas. 2003. Representable correcting terms for possibly underflowing floating point operations. *Proceedings - Symposium on Computer Arithmetic*, 79– 86. https://doi.org/10.1109/ARITH.2003.1207663

Sylvie Boldo, Stef Graillat, and Jean-Michel Muller. 2017. On the Robustness of the 2Sum and Fast2Sum Algorithms. *ACM Trans. Math. Softw.* 44, 1, Article 4 (jul 2017), 14 pages. https://doi.org/10.1145/3054947

Cadna. 2022. *The gaussian method.* https://www-pequan.lip6.fr/cadna/Examples_Dir/ex6.php

Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020a. Debugging and Detecting Numerical Errors in Computation with Posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 731–746. https://doi.org/10.1145/3385412.3386004

Sangeeta Chowdhary, Jay P Lim, and Santosh Nagarakatte. 2020b. *FPSanitizer - A debugger to detect and diagnose numerical errors in floating point programs.* Retrieved March 23rd, 2020 from https://github.com/rutgers-apl/fpsanitizer

Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3468264.

3468585

Sangeeta Chowdhary and Santosh Nagarakatte. 2022a. *Artifact for Fast Shadow Execution for Debugging Numerical Errors using Error Free Transformations*. https://doi.org/10.5281/zenodo.7080559

Sangeeta Chowdhary and Santosh Nagarakatte. 2022b. *EFTSantizer: Fast Shadow Execution for Debugging Numerical Errors using Error Free Transformations*. Retrieved August, 2022 from https://github.com/rutgers-apl/EFTSanitizer

Catherine Daramy, David Defour, Florent Dinechin, and Jean-Michel Muller. 2003. CR-LIBM: A correctly rounded elementary function library. In *Proceedings of SPIE Vol. 5205: Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, Vol. 5205. https://doi.org/10.1117/12.505591

Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. 2006. *CR-LIBM A library of correctly rounded elementary functions in double-precision*. Research Report. Laboratoire de l'Informatique du Parallélisme. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804

Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy-framework for analysis and optimization of numerical programs (tool paper). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 270–287. https://doi.org/10.1007/978-3-319-89960-2_15

Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 51, 14 pages. https://doi.org/10.1109/SC41405.2020.00055

Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2006. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*. ACM, 1318–1322. https://doi.org/10.1145/1141277.1141584

T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (1971), 224–242. https://doi.org/10.1007/BF01397083

David Delmas and Jean Souyris. 2007. Astrée: From Research to Industry. In *Proceedings of the 14th International Conference on Static Analysis* (Kongens Lyngby, Denmark) *(SAS'07)*. Springer-Verlag, Berlin, Heidelberg, 437–451. https://doi.org/10.1007/978-3-540-74061-2_27

Nestor Demeure. 2020. *Compromise between precision and performance in high-performance computing*. Ph. D. Dissertation. Université Paris-Saclay. https://tel.archives-ouvertes.fr/tel-03116750

Peter Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/3369583.3392673

Marco A Feliú, Mariano Moscato, César A Muñoz, et al. 2018. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

François Févotte and Bruno Lathuilière. 2016. VERROU: Assessing Floating-Point Accuracy Without Recompiling. (Oct. 2016). https://hal.archives-ouvertes.fr/hal-01383417 working paper or preprint.

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. In *ACM Transactions on Mathematical Software*, Vol. 33. ACM, New York, NY, USA, Article 13. https://doi.org/10.1145/1236463.1236468

Khalil Ghorbal, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2012. Donut Domains: Efficient Non-convex Domains for Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science, Vol. 7148)*. Springer, 235–250. https://doi.org/10.1007/978-3-642-27940-9_16

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. In *ACM Computing Surveys*, Vol. 23. ACM, New York, NY, USA, 5–48. https://doi.org/10.1145/103162.103163

Eric Goubault. 2001. Static Analyses of the Precision of Floating-Point Operations. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer, 234–259. https://doi.org/10.1007/3-540-47764-0_14

Eric Goubault, Sylvie Putot, Philippe Baufreton, and Jean Gassino. 2007. Static analysis of the accuracy in control systems: Principles and experiments. In *Revised Selected Papers from the 12th International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 3–20. https://doi.org/10.1007/978-3-540-79707-4_3

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Claude-Pierre Jeannerod, Jean-Michel Muller, and Paul Zimmermann. 2018. On Various Ways to Split a Floating-Point Number. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. 53–60. https://doi.org/10.1109/ARITH.2018.

8464793

Fabienne Jézéquel and Jean-Marie Chesneaux. 2008. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications* 178, 12 (June 2008), 933–955. https://doi.org/10.1016/j.cpc.2008.02.003

William Kahan. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. In *Communications of the ACM*, Vol. 8. ACM, New York, NY, USA. https://doi.org/10.1145/363707.363723

Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA.

Philippe Langlois, Stef Graillat, and Nicolas Louvet. 2006. Compensated Horner Scheme. In *Algebraic and Numerical Algorithms and Computer-assisted Proofs (Dagstuhl Seminar Proceedings (DagSemProc), Vol. 5391)*, Bruno Buchberger, Shin'ichi Oishi, Michael Plum, and Sigfried M. Rump (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/DagSemProc.05391.3

Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: Runtime Assessment of Floating-Point Instability by Vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 623–638. https://doi.org/10.1145/2814270.2814299

Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2020. A Novel Approach to Generate Correctly Rounded Math Libraries for New Floating Point Representations. arXiv:2007.05344 Rutgers Department of Computer Science Technical Report DCS-TR-753.

Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 29 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434310

Jay P. Lim and Santosh Nagarakatte. 2021a. High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*. https://doi.org/10.1145/3453483.3454049

Jay P Lim and Santosh Nagarakatte. 2021b. RLIBM-32: High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations. arXiv:2104.04043 Rutgers Department of Computer Science Technical Report DCS-TR-754.

Jay P. Lim and Santosh Nagarakatte. 2021c. RLIBM-ALL: A Novel Polynomial Approximation Method to Produce Correctly Rounded Results for Multiple Representations and Rounding Modes. arXiv:2108.06756 [abs] Rutgers Department of Computer Science Technical Report DCS-TR-757.

Jay P. Lim and Santosh Nagarakatte. 2022. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 3 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498664

LLNL. 2022a. *AMG*. https://asc.llnl.gov/codes/proxy-apps/amg2013

LLNL. 2022b. *High-order Lagrangian Hydrodynamics Miniapp*. https://github.com/CEED/Laghos

Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation*. Springer, 3rd edition. https://doi.org/10.1007/978-1-4899-7983-4

Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2nd ed.). Birkhäuser Basel. https://doi.org/10.1007/978-3-319-76526-6

Santosh Nagarakatte. 2012. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. Ph. D. Dissertation. University of Pennsylvania.

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*. https://doi.org/10.1145/1806651.1806657

NAS. 2022. *NAS Parallel Benchmarks 3.0*. https://github.com/benchmark-subsetting/NPB3.0-omp-C

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

Takeshi Ogita, Siegfried Rump, and Shin'ichi Oishi. 2005. Accurate Sum and Dot Product. *SIAM J. Scientific Computing* 26 (01 2005), 1955–1988. https://doi.org/10.1137/030601818

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2813885.2737959

Douglas M. Priest. 1992. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations.* Ph. D. Dissertation. USA. UMI Order No. GAX93-30692.

Siegfried M. Rump. 2009. Ultimately Fast Accurate Summation. *SIAM Journal on Scientific Computing* 31, 5 (2009), 3466–3502. https://doi.org/10.1137/080738490

Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/3192366.3192411

Jonathan Shewchuk. 1996. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete and Computational Geometry* 18 (07 1996). https://doi.org/10.1007/PL00009321

Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (dec 2018), 39 pages. https://doi.org/10.1145/3230733

Pat H Sterbenz. 1974. *Floating-point computation.* Prentice-Hall, Englewood Cliffs, NJ.

US-GAO United States General Accounting Office. 1992. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.* https://www.gao.gov/products/IMTEC-92-26

Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting Numerical Bugs in Neural Network Architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 826–837. https://doi.org/10.1145/3368089.3409720

Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proc. ACM Program. Lang.* 4, POPL, Article 60 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371128