Debugging and Detecting Numerical Errors in Computation with Posits

Sangeeta Chowdhary
Department of Computer Science
Rutgers University
USA
sangeeta.chowdhary@rutgers.edu

Jay P. Lim
Department of Computer Science
Rutgers University
USA
jpl169@cs.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
USA
santosh.nagarakatte@cs.rutgers.edu

Abstract

Posit is a recently proposed alternative to the floating point representation (FP). It provides tapered accuracy. Given a fixed number of bits, the posit representation can provide better precision for some numbers compared to FP, which has generated significant interest in numerous domains. Being a representation with tapered accuracy, it can introduce high rounding errors for numbers outside the above golden zone. Programmers currently lack tools to detect and debug errors while programming with posits.

This paper presents PositDebug, a compile-time instrumentation that performs shadow execution with high precision values to detect various errors in computation using posits. To assist the programmer in debugging the reported error, PositDebug also provides directed acyclic graphs of instructions, which are likely responsible for the error. A contribution of this paper is the design of the metadata per memory location for shadow execution that enables productive debugging of errors with long-running programs. We have used PositDebug to detect and debug errors in various numerical applications written using posits. To demonstrate that these ideas are applicable even for FP programs, we have built a shadow execution framework for FP programs that is an order of magnitude faster than Herbgrind.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Keywords: Posits, dynamic analysis, numerical errors

ACM Reference Format:
Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386004

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386004

1 Introduction

Representing real numbers is important in a variety of domains. The floating point (FP) representation is a widely used approximation of reals using a finite number of bits [1, 20]. Being an approximation, not every real value can be exactly represented in the FP representation. It has to be rounded to the nearest value according to the rounding mode. The IEEE standard represents a floating point number in the form: $(-1)^s \times M \times 2^E$, where s determines the sign of the number, significand M is a fractional binary number ranging either between $[1, 2)$ or $[0, 1)$, and exponent E multiplies the value with a power of 2. Further, there are numerous exceptional values (*i.e.*, NaNs and infinities).

Bugs in programs using FP have resulted in numerous mishaps with catastrophic consequences [52, 53, 56]. Hence, there is a large body of work on reasoning about the correctness and the accuracy of floating point programs [2–4, 7, 8, 12, 15, 17, 19, 22, 23, 28, 36, 37, 42, 43, 55, 58–62, 66]. Further, good FP performance is paramount in many application domains like machine learning and scientific computing. Given the need for varying performance and precision trade-offs, many accelerators for machine learning use non-standardized custom representations as replacements for FP [5, 11, 24, 29, 31, 33, 39, 40, 57, 63, 64]. There is active interest to explore new representations for better performance, flexible precision, and dynamic range.

Posit is a recently proposed stand-in replacement for FP by John Gustafson [26, 27]. Posit is a hardware friendly version of unums (universal numbers) [25]. A number in the posit representation consists of a sign bit, regime bits, optional exponent bits, and optional fraction bits. In a $\langle n, es \rangle$-posit representation, there are n-bits in total and at most es-bits for the exponent. Regime bits encode the super-exponent. When a minimum number of regime bits are needed to represent the number (*i.e.*, 2), the remaining bits can be used to increase precision, which provides tapered accuracy (Section 2 provides a detailed background).

Promise of posits. The posit representation provides two notable advantages compared to FP. (1) Given the same number of bits, posits can provide higher precision than FP for a range of numbers. It provides variable precision with tapered accuracy [45]. (2) An n-bit posit can represent more distinct values than the FP representation with the same number of bits. In contrast to FP, the posit representation has only one zero, one value to represent exceptions, and a single rounding mode. Initial research has demonstrated the promise of posits in high performance computing [27]. For example, the sigmoid function, which is commonly used in neural networks, can be approximated by bitwise operations: right shift of the posit bit-string by two bits after a negation of the sign bit [27]. There is excitement and interest in exploring posit as an alternative to FP in many domains [27, 31]. Further, there are ASIC and FPGA-based hardware proposals for posit arithmetic [30].

Challenges of programming with posits. Although posit provides better precision for a range of numbers with tapered accuracy, large and small numbers lose precision [13]. When such numbers are involved in subtraction, catastrophic cancellations are often common. Further, posit avoids overflows (underflows) by saturating all computations to the largest (or the smallest) possible posit value, which can produce counter-intuitive results when porting code using FP to use posits.

PositDebug. This paper proposes PositDebug, a compiler instrumentation that performs shadow execution to detect numerical errors in applications using posits. To detect numerical errors, it performs shadow execution with high-precision values. Every variable and memory location that stores a posit value is shadowed with a high-precision value in shadow memory. Whenever a program performs a posit arithmetic operation, a similar operation is performed in the shadow execution with higher precision on the operands from shadow memory. PositDebug detects numerical errors that are amplified due to posit's tapered accuracy: exceptions, cancellation, loss of precision, unexpected branch outcomes, and wrong outputs.

To enable the programmer to debug the error, PositDebug also provides a directed acyclic graph (DAG) of instructions in the set of active functions (*i.e.*, in the backtrace) that are responsible for the error. To provide such DAGs on an error, PositDebug has to maintain additional metadata with each memory location. A key contribution of this paper is the design of the metadata for each memory location and each temporary variable. PositDebug maintains a constant amount of metadata per memory location, which enables execution with long-running applications.

FPSanitizer. To show the generality of our approach, we have built FPSanitizer, a shadow execution framework for floating point programs. It uses the same metadata organization and design as PositDebug but works with FP programs. In contrast to Herbgrind [59] that crashes with large applications, FPSanitizer is able to successfully run them. Even with smaller applications that can execute with Herbgrind, FPSanitizer is more than 10× faster.

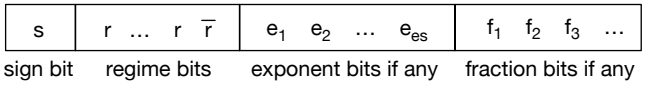

Figure 1. The posit bit-string for the $\langle n, es \rangle$ configuration to represent finite non-zero values, where n is the number of bits and es is the maximum number of exponent bits.

We have used the prototypes of PositDebug and FPSanitizer to detect and debug errors in both posit and FP applications. We initially developed PositDebug to help us in our effort to develop a math library for posits using the CORDIC (Coordinate Rotation Digital Computer) algorithm [65]. PositDebug's feedback helped us debug and eventually fix numerical errors in our math library implementation and also in many other programs. PositDebug's shadow execution that uses the Multiple Precision Floating-Point Reliable (MPFR) library with 256 bits of precision experiences a performance overhead of 12.3× compared to a software-only posit program without shadow execution. FPSanitizer's shadow execution detects numerical errors and provides DAGs to isolate the likely root cause of errors with 111× overhead compared to a program that uses hardware FP operations without any shadow execution.

2 Background on Posits

Posit is a recently proposed alternative to the IEEE-754 FP representation [26, 27]. There are two notable advantages with the posit representation compared to FP. Given the same number of bits, posit can represent numbers with a wider dynamic range and also provide higher precision for a certain range of values. An n-bit posit can represent more distinct values than FP.

2.1 The Posit Representation

A posit bit-string consists of a sign bit, regime bits, exponent bits, and fraction bits. The number of bits used for the regime, the exponent, and the fraction can vary depending on the number being represented. The regime is a super-exponent that is used to increase the dynamic range of the representation. In contrast, the IEEE-754 FP format consists of three parts: a sign bit, a fixed number of bits for the exponent and the fraction.

Decoding a posit bit-string. The posit environment is defined by a tuple of numbers, $\langle n, es \rangle$ where n is the total number of bits and es is the maximum number of bits to represent the exponent. Figure 1 depicts the format of a posit bit-string. The first bit, s, of a posit number represents the sign. If $s = 0$, then it is a positive number. If $s = 1$, then it is a negative number, in which case we have to compute the

two's complement of the bit-string before decoding the rest of the bits.

The next set of bits represents the regime. The length of the regime, m, can be $2 \leq m \leq n-1$. The regime bits consist of consecutive 1's (or 0's). It is terminated either by an opposite bit 0 (or a 1) or when there are no more bits left (when all $n-1$ bits are used to represent the regime).

If there are any remaining bits after the regime bits, the next x bits represent the exponent, where $x = min\{n-1-m, es\}$. If $x < es$, the exponent bits are extended to length es by appending $(es - x)$ 0's at the end. The last remaining bits after the exponent bits belong to the fraction.

Value of a posit bit-string. The sign of a posit number is determined using the sign bit: $(-1)^s$. Let r be the number of consecutive 1 or 0 bits without the terminating opposite bit in m. Let k be:

$$k = \begin{cases} -r & \text{if regime bits are 0's} \\ r-1 & \text{if regime bits are 1's} \end{cases} \quad (1)$$

Also, let useed be:

$$useed = 2^{2^{es}} \quad (2)$$

Abstractly, useed is a super-exponent. The exponent e is defined as the unsigned integer of the exponent bits. Note that the maximum value of 2^e is $2^{2^{es}-1} = \frac{useed}{2}$. The value of the fractional part f is calculated similar to IEEE-754 normal values:

$$f = 1 + \left(\frac{fraction}{2^{<\#\text{ of fraction bits}>}} \right) \quad (3)$$

Finally, the value that a posit bit-string represents is:

$$(-1)^s \times useed^k \times 2^e \times f \quad (4)$$

Let us consider a $\langle 8, 1 \rangle$-posit configuration, where there are 8-bits in total and at most 1-bit used for the exponent. The useed is $2^{2^1} = 4$. Consider the posit bit-string: 01101101. The sign bit is 0 and hence, it is a positive number. The regime is 110. The exponent is 1. The fractional part is 1 + 5/8. Hence, the value of the number is $(-1)^0 * 4^1 * 2^1 * (1 + 5/8) = 13$.

Special values. There are two bit-patterns that do not follow the above rules. First, a bit-string of all 0's represents the number 0. Second, a bit-string that consists of a 1 followed by all 0's represents the *Not-a-Real* value (NaR).

Unique feature of posits. The posit representation provides tapered accuracy [45] using the Rice-Golomb encoding [21, 41]. The number of bits for the regime, the exponent, and the fraction can vary. Hence, posit can represent (1) a wider range of numbers by using more bits for the regime, which is a super-exponent and (2) provide higher precision in the interval $[\frac{1}{useed}, useed]$ by using more bits for the fraction. Large numbers or numbers extremely close to 0 can be represented by increasing the number of regime bits at the cost of losing precision. In contrast, numbers close to 1, which use a minimal number of regime bits (*i.e.*, 2-bits), provide higher precision by allocating a large number of bits to represent the fraction.

```
int RootCount(){
 a = 1.83090..E16;
 b = 3.24664..E12;
 c = 1.439239..E8;
 t1 = b * b;
 t2 = 4.0 * a * c;
 t3 = t1 - t2;
 if (t3 > 0.0)
  return 2;
 else if (t3 == 0.0)
  return 1;
 else return 0;
}
```

Posit Value	# frac. bits	Exact Value	Error Type
1.830..E16	14	1.83090..E16	
3.246..E12	17	3.24664..E12	
1.4392..E8	21	1.439239..E8	
1.057..E25	7	1.05406..E25	LP
1.057..E25	7	1.05404..E25	LP
0.0	N/A	2.40507..E20	CC
False		True	BF
True		False	BF
1		2	WR

Figure 2. A program to compute the number of roots for the equation $ax^2 + bx + c$. For each instruction, we report the result of the posit operation, the number of available fraction bits for the resulting posit value, the exact value if computed in an ideal environment, and the type of posit error: loss of precision (LP), catastrophic cancellation (CC), branch flips (BF), and wrong results (WR).

A $\langle 32, 2 \rangle$-posit configuration can represent all positive and negative numbers in the interval $[\frac{1}{2^{20}}, 2^{20}]$ with same or better precision than a 32-bit float, which is the golden zone for this posit configuration [13]. To avoid overflows and underflows, all values saturate at `maxpos` and `minpos`, where `maxpos` and `minpos` represent the largest and the smallest value representable in the posit configuration, respectively.

Fused operations with posits. The posit specification mandates the use of fused operations with higher precision using the quire data type, which is a high-precision accumulator [34, 35]. The posit standard defines several multiply-accumulator instructions using the quire data type (*e.g.*, fused sum, fused dot-product). The size of the quire is determined by the posit configuration. It should be large enough to support the computation of $maxpos^2 + minpos^2$ without any loss of precision.

2.2 Numerical Errors with Posits

The posit representation suffers from rounding error similar to FP, when a value is not exactly representable. However, some rounding errors can be amplified due to the loss of precision bits for numbers that lie outside the golden zone. Further, tapered accuracy with posits necessitates novel numerical analysis.

Catastrophic cancellation. Similar to FP, a posit configuration can experience catastrophic cancellation when two similar but inexact (rounded) values are subtracted from each other. Such two operands have the same bit-patterns in

the most significant digits and are canceled out. The result primarily depends on the inexact bits, amplifying the error of the operands.

Loss of precision bits. Posit enables variable precision with the use of regime bits, which increases the dynamic range. However, it also reduces the available precision bits. Posit values lose precision bits when the magnitude of the value increases towards `maxpos` or decreases towards `minpos`.

Posit's tapered accuracy by default ensures that values closer to `maxpos` and `minpos` can only be represented with a small number of precision bits. Additionally, the result of all basic operations (+, −, ×, and ÷) can lose precision bits depending on the magnitude of the result when compared to the operands, which can create issues similar to catastrophic cancellation. In FP, operations with denormalized values show such behavior.

Illustration of cancellation. Consider the posit program in Figure 2. Given three inputs a, b, and c, the function `RootCount` computes the number of roots for the equation $ax^2 + bx + c$. Mathematically, the quadratic equation with the coefficients

$$a = 1.8309067625725952 \times 10^{16}$$
$$b = 3.24664295424 \times 10^{12}$$
$$c = 1.43923904 \times 10^{8}$$

has two roots, since $b^2 - 4ac = 2.40507138275350151168 \times 10^{20} > 0$. However, a $\langle 32, 2\rangle$-posit evaluation of `RootCount` produces $t3 = 0.0$ and returns 1. The same computation with a 32-bit float evaluates $t3 = 2.40960594462831017984 \times 10^{20}$ and returns 2. There are two sources of error that contribute to this behavior. The intermediate result of b^2 and $4ac$ produces large values, $1.05406904723162347339776 \times 10^{25}$ and $1.05404499651779593838264 32 \times 10^{25}$, respectively. The posit configuration has only 8 bits of precision in that range. Both values are rounded to $1.05781009216280052786 7904 \times 10^{25}$. Hence, subtraction experiences catastrophic cancellation, ultimately resulting in 0.

An equivalent expression for $b^2 - 4ac$ reduces the dynamic range of the intermediate result, which increases the number of available fraction bits:

$$\left(b - 2\sqrt{a}\sqrt{c}\right)\left(b + 2\sqrt{a}\sqrt{c}\right)$$

Square root is an interesting operation with posits. For all $x \in \mathbb{R}^+, x \neq 1$, either $1 < \sqrt{x} < x$ or $x < \sqrt{x} < 1$. Hence, $\sqrt{x}$ is always closer to 1 than x is to 1:

$$\left|\sqrt{x} - 1\right| < |x - 1|$$

Hence, $\sqrt{x}$ has at least the same number of precision bits as x. Moreover, if the posit bit-string of x uses r bits for the regime, then $\sqrt{x}$ uses at most $\lceil \frac{r}{2} \rceil$ regime bits. A $\langle 32, 2\rangle$-posit evaluation of the above equation results in $2.17902164370694078464 \times 10^{20}$, which provides the best result with posits.

Saturation with `maxpos` and `minpos`. Unlike FP, posit values do not overflow or underflow. Any positive posit value greater than *maxpos* is rounded to *maxpos* and any positive value less than *minpos* (excluding 0) is rounded to *minpos*. Posit operations silently hide overflows (or underflows). It can produce results with large errors. Such computation will result in ∞ or 0 with FP, which can produce exceptions later in the program.

Different observable results. Similar to FP computation, numerical errors with posits can cause a program to produce different branch outcomes compared to the ideal execution. Figure 2 illustrates this behavior. Similar to branch conditions, the programs can also produce a wrong integer on posit-to-integer casts or wrong system call arguments due to rounding errors.

2.3 Metrics for Measuring Error

Unit in the last place (ULP) is a commonly used metric to measure error with normal values in FP computation. ULP is defined as the value of the last precision bit in a given FP value: 2^{e-p+1} where e is the exponent of the value and p is the precision of the given FP representation. The error in ULPs is also commonly interpreted as the number of distinct FP values between the computed result and the ideal result [46]. The relative error is typically expressed as ULP error with FP's normal values. The relative error corresponding to 1 ULP error in FP's normal values is bounded by $[2^{-p}, 2^{-p+1}]$ [20]. For example, in a 32-bit float, relative error of 1 ULP is bounded by $[2^{-24}, 2^{-23}]$ regardless of the magnitude of the value. Hence, it is commonly used to describe the precision limitation of a particular representation. For example, a 64-bit double normal value with a relative error of 2^{-23} has $\approx 2^{29}$ ULP error, indicating that the value is away from an accurately rounded value.

ULP error with posits. Measuring error with ULPs is not ideal for posits because of tapered accuracy (similar to denormal values in FP). An ULP of a posit value is 2^{e-p+1}, which is similar to FP, except that p depends on the magnitude of the posit value due to tapered precision. Hence, the relative error corresponding to 1 ULP error in posits can vary widely depending on the magnitude of the number.

Consider the $\langle 32, 2\rangle$-posit configuration where the smallest positive representable value is 2^{-120} and the next smallest representable value is 2^{-116}. Suppose that a posit computation produced the value 2^{-116} where the ideal value is 2^{-120}. In this case, the result has 1 ULP error. However, the relative error is $\frac{|2^{-120} - 2^{-116}|}{2^{-120}} = 15$. Similarly, the relative error of 1 ULP for posit values near 1 is bounded by 2^{-27}. The relative error represented in ULPs can widely vary for posits. Reporting 1 ULP error when a posit application has a relative error of 15 can be confusing to the users.

To address this issue, one approach is to measure the error in the posit value using the ULP error of a FP representation

that can represent all posit numbers as normal values. For example, every value in a ⟨32, 2⟩-posit configuration can be represented exactly as a normal value in the double data type. We can measure the ULP error of the double representation, which can be used to compare errors in posit computations.

3 PositDebug

The objective of our approach is to enable programmers to effectively detect numerical errors in posit applications and provide support for debugging them. To detect numerical errors with posit applications, our idea is to perform shadow execution with high-precision computation (*e.g.*, Multiple Precision Floating-Point Reliable Library (MPFR) [16]). In contrast to prior work [4, 59], PositDebug is a compile-time transformation that maintains a constant amount of metadata per memory location while performing shadow execution, which enables users to debug large programs.

Our goals with PositDebug are to (1) detect instances of numerical error such as cancellation, branch flips, incorrect results, and exceptions, (2) produce a directed acyclic graph (DAG) of instructions from the set of active functions (*i.e.*, backtrace) on a numerical error to provide concrete feedback to the user, and (3) enable debugging with debuggers (*i.e.*, gdb) with low-performance overheads.

3.1 High-Level Sketch

PositDebug is a compile-time transformation that instruments posit operations for shadow execution with higher-precision. Each memory location that stores a posit value is shadowed with an MPFR value in shadow memory and every temporary with a posit value is shadowed with an MPFR value on the stack. On every posit operation, the shadow execution retrieves the corresponding high precision values and computes the result in high-precision. When the error in the result exceeds a threshold or the result is used with branches/integer casts/system calls, PositDebug detects the error and also reports a DAG of instructions, which is likely responsible for the error, in the set of active stack frames at that instant to the user.

When the program and its shadow execution differ with respect to branch outcomes (*i.e.*, branch flip), the shadow execution follows the original program's execution. After a branch flip, the shadow execution uses the program's posit values to re-initialize the metadata entries in shadow memory and on the stack, which enables PositDebug to provide useful feedback for subsequent operations after a branch flip.

PositDebug exports a few functions that the programmer can use to diagnose the error with debuggers (*i.e.*, gdb). It allows the user to insert a conditional breakpoint depending on the amount of the error and obtain a DAG of dependent instructions. The nodes of the DAG are results of other binary/unary operations in the set of active functions at that instant. The root of this DAG is the instruction experiencing an error that exceeds the threshold.

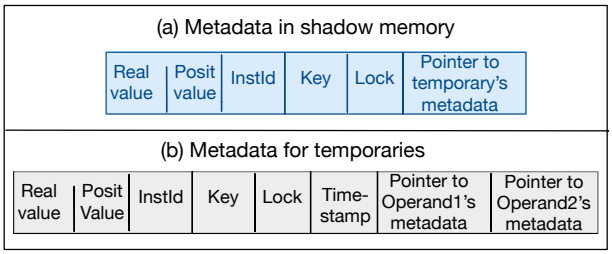

Figure 3. Metadata for (a) each posit value in shadow memory and (b) each posit temporary on the stack. Metadata in shadow memory maintains a pointer to the metadata of the temporary that previously wrote to the memory location. It has the lock and key information to check the validity of the metadata pointer for temporaries. Metadata in shadow memory also stores the real value, instruction identifier, and the posit value to detect errors when the pointer to temporary's metadata is not valid.

The primary challenge lies in designing the metadata for memory locations and temporaries for shadow execution. Although we present the key ideas in the context of posits, PositDebug can easily be re-purposed to debug and diagnose numerical errors with FP programs. We have also built a similar shadow execution framework for FP programs, which we call FPSanitizer [9].

3.2 Metadata Organization

As PositDebug is a compiler instrumentation, posit values are either resident in memory or are temporaries (*e.g.*, LLVM has temporaries with posit values or the values are resident in memory that are accessed through loads/stores). Every posit value in memory has metadata in shadow memory. Every temporary posit value has metadata on the stack. PositDebug also uses a shadow stack to store metadata for arguments and return values in a function call.

Minimal amount of metadata to detect errors. To detect numerical errors, we need to track a high-precision value with each posit in memory and for posit temporaries. Further maintaining the posit value and the information about the instruction that produced the posit value is useful to debug the error. We maintain the posit value in the metadata for two reasons. First, we can compute the error by comparing it to the MPFR value. Second, we can check if the value in memory was changed by a library function that was not instrumented by PositDebug. In the latter case, PositDebug uses the posit value from the program to initialize the metadata entry for subsequent use. Hence, each temporary and memory location with a posit value has to maintain at least the MPFR value, instruction identifier, and the posit value.

Constant amount of metadata per memory location. Identifying a set of instructions responsible for a particular error is useful for debugging. In contrast to prior work [59], PositDebug tracks a constant amount of metadata for each memory location, which enables its use with long-running applications while providing a trace of instructions responsible for the error. We observe that *most errors are local and the set of instructions responsible for the error are typically available in the set of active functions (*i.e.*, backtrace) when the error is encountered.*

Based on this observation, the metadata in shadow memory maintains a pointer to the metadata for the temporary on the stack that previously wrote to the memory location, temporal safety metadata for that pointer, the real value, instruction identifier, and the posit value. If the temporary that wrote to the memory location of interest is not available in the set of active stack frames (*i.e.*, when the stack frame has been deallocated on a function return), then it would be a memory safety error to access that temporary's metadata pointer. Hence, we need to maintain temporal safety metadata for that pointer. We use the lock-and-key metadata from our prior work on checking temporal safety errors [47–51].

Lock-and-key metadata for temporal safety. To ensure temporal safety, each pointer to the metadata for a temporary maintains a lock (address of a location) and a key (a unique identifier) [47–51]. On memory allocation, a new lock is allocated and a unique identifier is written to that lock. Any pointer that points to the newly allocated memory inherits the lock and the key. As long as the memory allocation is valid, the unique identifier at the lock and the key associated with the pointer will match. On deallocation, the identifier at the lock is invalidated. Any dangling pointer will find that the identifier at the lock and the key associated with the pointer does not match.

Figure 3(a) shows the metadata in shadow memory for each posit value in memory. When the pointer to the temporary's metadata is invalid, PositDebug is still able to detect numerical errors because it maintains the high-precision value with each metadata entry. However, it will not be able to provide a detailed trace of instructions that wrote to that location. In this scenario, the user can set a breakpoint on the instruction that previously wrote to the location and obtain a trace of instructions.

Metadata for temporaries. Metadata for temporaries with posit values are maintained on the stack. The additional stack usage is proportional to the original program's stack usage. Temporary posit variables either contain the result of posit arithmetic operations or are constants. Hence, the metadata for temporary posit values contains the high-precision value produced in the shadow execution, actual posit value, instruction identifier, pointers to the metadata for the operands, and lock-and-key metadata to ensure the temporal safety information for the stack frame. Metadata for each temporary gets the lock-and-key of the function because the stack frame is allocated on function entry and deallocated on function return.

To keep the stack usage bounded, we maintain a single metadata entry for each static temporary in the code. When a temporary is repeatedly updated in a loop, the metadata corresponding to the temporary in the stack is updated. We maintain a timestamp in the temporary's metadata that records when it was updated. When we report the DAG of instructions responsible for the error, we do not report the operands of an instruction if the operand's metadata timestamp is greater than the timestamp of the instruction in consideration. Figure 3(b) illustrates the metadata for temporaries.

3.3 Metadata Propagation

This section describes the creation of the metadata and its propagation from the stack to shadow memory. Each function has a lock and a key for temporal safety, which we refer to as `func_lock` and `func_key`. We omit the instruction identifier, the posit value, and timestamp updates for space reasons. The added instrumentation code is shaded.

Creation of temporary constants. When the program creates a temporary posit variable with a constant, we create metadata on the shadow stack that uses the concrete constant in high-precision, inherits the function's lock and key, and has its operand pointers set to NULL.

```
posit32_t t5 = 4.0f; //instruction 5 in Figure 4
```

```
t5_tmd = get_temporary_metadata(t5);
t5_tmd->real = Real(4.0f);
t5_tmd->lock = func_lock;
t5_tmd->key = func_key;
t5_tmd->op1 = NULL;
t5_tmd->op2 = NULL;
```

Assignment of temporaries. When a posit value is copied from another temporary, all fields of the metadata for the new temporary are also copied. Only the timestamp is updated.

Posit binary and unary operations. On binary or unary arithmetic operations with posit values, we look up the metadata of the operands, compute the high-precision value with shadow execution, and update the metadata of the result. The result's metadata is updated with the high-precision result, pointers to operand's metadata, and lock-and-key metadata for the stack frame.

```
posit32_t t6 = t5 * t1; //instruction 6 in Fig 4.
```

```
t5_tmd = get_temporary_metadata(t5);
t1_tmd = get_temporary_metadata(t1);
t6_tmd = get_temporary_metadata(t6);
t6_tmd->lock = func_lock;
t6_tmd->key = func_key;
t6_tmd->real = t5_tmd->real *R t1_tmd->real;
```

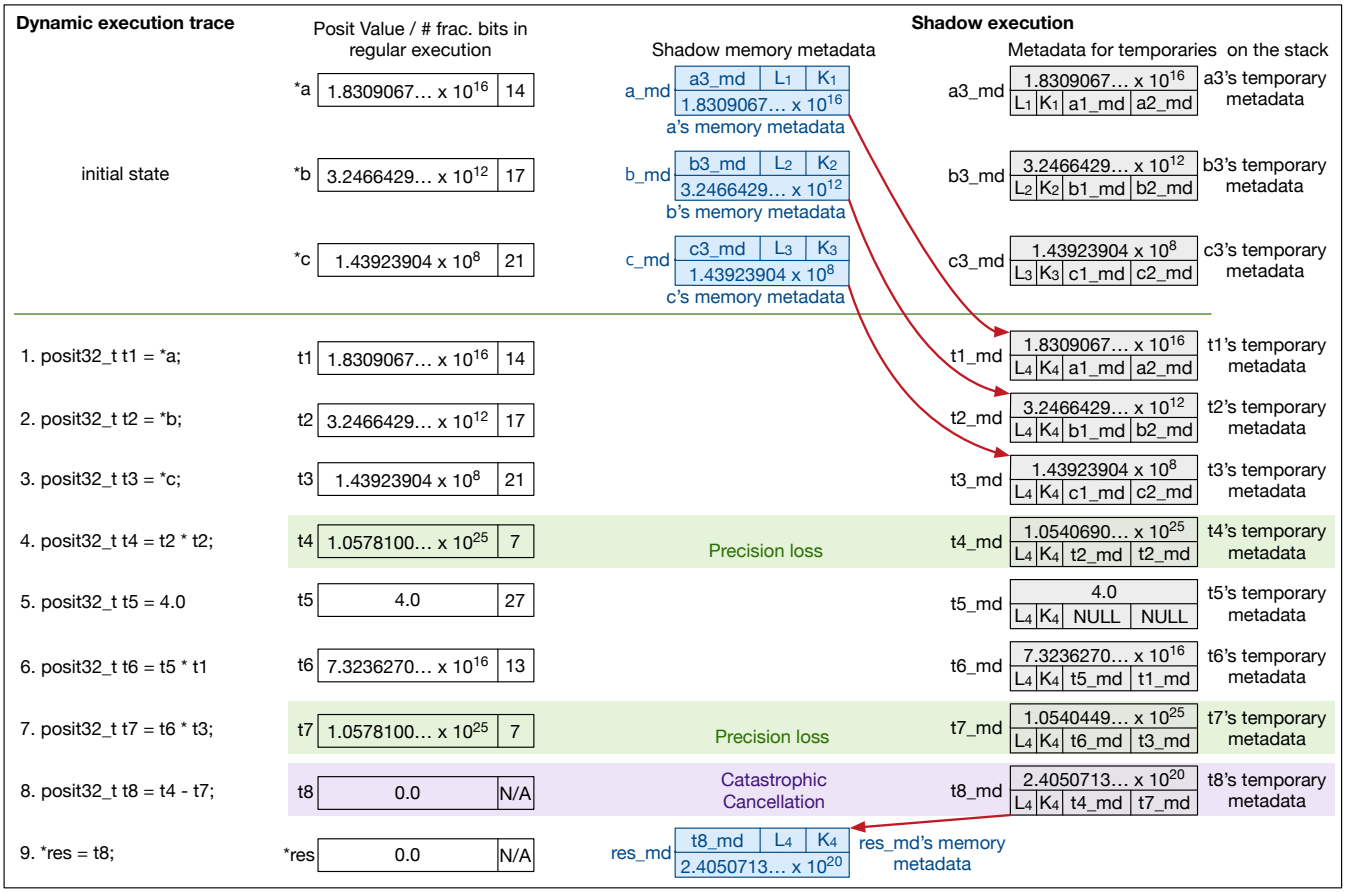

Figure 4. Execution trace, program's memory, and the content of the metadata. We show the posit value and the number of available fraction bits after each instruction. The state of shadow memory and the metadata for temporaries on the stack are on the right. The movement of metadata between shadow memory and the stack is shown with red arrows. Instances of precision loss and catastrophic cancellation are highlighted in green and purple, respectively. Here, all lock and key metadata are valid. (L4, K4) represents the lock and key for the function being executed.

```
t6_tmd->op1 = t5_tmd;
t6_tmd->op2 = t1_tmd;
```

Memory stores. When a posit value is stored to memory, we update the shadow memory with the high-precision value, pointer to the temporary's metadata that is being stored, and the lock and key associated with the temporary.

```
*res = t8 //instruction 9 in Figure 4
```

```
t8_md = get_temporary_metadata(t8);
shadow_mem(res)->real = t8_md->real;
shadow_mem(res)->tmd = t8_md;
shadow_mem(res)->lock = t8_md->lock;
shadow_mem(res)->key = t8_md-> key;
```

Memory loads. When a posit value is loaded from memory into a temporary, the metadata from the shadow memory is accessed and the temporary's metadata is updated with information from shadow memory. First, we check if the temporary that previously wrote to that location is still valid by checking the lock and the key. If so, the entire temporary metadata of the previous writer is copied except the lock and the key. The lock and the key of the new temporary are initialized to the executing function's lock and key. If the pointer to the previous writer is invalid, we initialize the temporary metadata similar to the assignment of a constant value.

```
posit32_t t1 = *a; //instruction 1 in Figure 4
```

```
t1_md = get_temporary_metadata(t1);
t1_md->real = shadow_mem(a)->real;
t1_md->lock = func_lock;
t1_md->key = func_key;
lock = shadow_mem(a)->lock;
if(*lock == shadow_mem(a)->key){
  t1_md->op1 = shadow_mem(a)->tmd->op1;
  t1_md->op2 = shadow_mem(a)->tmd->op2;
}
```

```
else {
  t1_md->op1 = NULL;
  t2_md->op2 = NULL;
}
```

Figure 4 illustrates the metadata before and after three loads from memory (*i.e.*, instruction 1-3). When a value is being loaded from address a, the shadow memory of a (*i.e.*, a_md) is accessed, which has a pointer to the temporary metadata of the last writer (a3_md). The lock and key of a3_md are L_1 and K_1. As it is valid, the entire contents of a3_md are copied to the temporary metadata space for $t1$ (*i.e.*, t1_md). The lock and key of t1_md are set to the lock and key of the executing function (*i.e.*, L_4 and K_4).

3.4 Detection and Debugging of Errors

When PositDebug detects a significant error in the result of the computation, it classifies the cause of error into various categories: catastrophic cancellation, loss of precision, changes in branch outcomes, and wrong values with casts.

Catastrophic cancellation. PositDebug detects whether any cancellation that occurred during the subtraction operation is catastrophic. When a subtraction operation cancels a number of significant digits of the operands such that every digit of the normalized result is affected by the error in the operands, it is considered to be a catastrophic cancellation. The relative error cannot be bounded in the presence of catastrophic cancellation. The computed result v and the real result r can differ considerably. In this scenario, if $v \geq 2r$ or $v \leq \frac{r}{2}$, then the exponents of v and r are guaranteed to differ and also all the precision bits are influenced by error.

In PositDebug, we say that a subtraction operation experiences catastrophic cancellation if (1) it experiences cancellation and (2) $v \geq 2r$ or $v \leq \frac{r}{2}$ where v and r are the computed and the real result, respectively. In a 64-bit FP representation that can represent all $\langle 32, 2\rangle$-posit values as normalized FP values exactly, the result has at least $\geq 2^{52}$ ULP error.

PositDebug detects whether the subtraction $v_1 = v_2 - v_3$ has catastrophic cancellation using the following formula:

$$cbits > 0 \wedge \left(v_1 \geq 2r_1 \vee v_1 \leq \frac{1}{2} r_1\right)$$

where r_1 is the real result and $cbits$ is defined as:

$$cbits = max\{exp(v_2), exp(v_3)\} - exp(v_1)$$

Here, $exp(v)$ represents the exponent of v. If the exponent of the result is smaller than that of the operands, then some bits of the operand are canceled. Along with cancellation, if the error in the final result is above a threshold, then it is catastrophic cancellation.

As there is no widely accepted error threshold for catastrophic cancellation, the users can choose the error in our second criteria for catastrophic cancellation using the expression:

$$v \geq \epsilon r \vee v \leq \frac{r}{\epsilon}$$

In our definition, $\epsilon = 2$. The magnitude of ϵ defines the amount of error tolerance: larger ϵ widens the error tolerance and smaller ϵ narrows it.

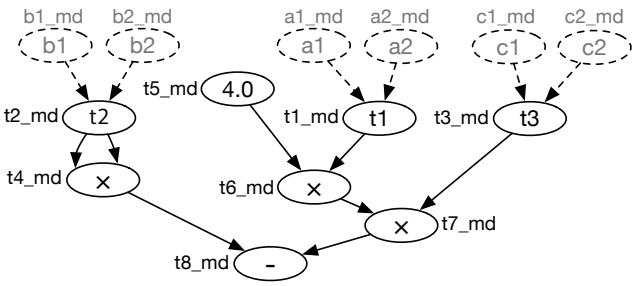

Figure 5. DAG generated for the catastrophic cancellation with the operation: t8 = t4 - t7 in Figure 4.

Loss of precision bits and saturation. PositDebug allows the detection of significant precision loss, which is important with posits because of tapered precision. PositDebug detects if the result of an operation has more regime bits than the operands and reports if such precision loss is beyond a user-specified threshold. Similarly, PositDebug reports whenever operations use maxpos and minpos as they are cases of likely overflows/underflows with FP operations.

Branch outcomes and casts to integers. PositDebug instruments branch conditions with posit values and checks if the branch condition evaluates with a different outcome in the shadow execution. As we do not want to track metadata with integers, PositDebug also instruments all posit-to-integer cast operations and reports it to the user if the shadow execution produces a different result. It is possible that an error in posit-to-integer conversion is benign. However, the result of a posit-to-integer conversion can also be used with operations that can influence the result (*e.g.*, array indices, pointer offsets). PositDebug does not triage these cases into benign or malicious. We let the user decide.

3.5 Debugging Errors with Tracing

Whenever PositDebug detects the errors mentioned above, it provides a DAG of instructions in the set of active functions that are responsible for the error. The temporary metadata of the instruction experiencing the error becomes the root of the DAG. It has information about the temporary metadata of its operands. PositDebug checks the temporal validity of the operand's pointers. If they are valid and the timestamp of the operands are lower than the timestamp of the current instruction, those operands are recursively traversed.

In Figure 4, there is a catastrophic cancellation with the instruction t8 = t4 - t7, which becomes the root of the DAG. The t8's metadata (*i.e.*, t8_md) has pointers to

temporary metadata of the operands: `t4_md` and `t7_md`. They are recursively traversed. The resultant DAG reported to the programmer is shown in Figure 5.

4 Implementation and Prototypes

We describe the implementation decisions, PositDebug prototype, and the FPSanitizer prototype for FP programs.

4.1 Implementation Issues

To realize a prototype of shadow execution for large programs, we need to design mechanisms to shadow memory, manage the lock and key metadata space, and interface with uninstrumented code.

Shadow memory. We use a two-level trie data structure for shadow memory. It maps every virtual address that holds a posit value to its corresponding shadow memory address. The first-level trie entries are allocated during program initialization. The second-level trie entries are allocated on demand. Hence, the shadow memory usage is proportional to the actual memory usage of the program. The shadow memory entry contains a pointer to the MPFR value. The MPFR runtime allocates high-precision values on the heap.

Management of lock and key metadata. Our lock and key metadata ensure the temporal safety of the pointers to various stack frames. The lock locations are organized as a stack. On a function entry, a new key is pushed to the top of this stack. This key is associated with any temporary metadata that belongs to the current function. On a function exit, the lock entry is invalidated and popped from the stack. These lock entries can be subsequently reused. As our keys are monotonically increasing, PositDebug accurately identifies all valid and invalid pointers to the stack frames even when the lock space is subsequently reused. The size of the lock space is bounded by the number of active functions.

Interfacing with uninstrumented code. To support large applications, we have to support interfacing with libraries, which is a challenging task for any dynamic monitoring tool [47, 50]. Inspired by Intel MPX, we maintain the posit value in the metadata in shadow memory. When the program loads a posit value from program's memory, the corresponding metadata is loaded from shadow memory. The posit value generated by the program and the one in the metadata space is compared. If they do not match, we can conclude that some library or uninstrumented function updated program's memory without updating shadow memory. In those cases, we use the program's posit value to initialize the high-precision value. This approach also enables us to incrementally use our tools with large applications.

4.2 PositDebug Prototype

PositDebug prototype detects numerical errors in C programs that use posits. It is publicly available [10]. As there is no mainstream hardware support yet, we use the official software library for posits (*i.e.*, SoftPosit). PositDebug consists of three components. (1) A standalone LLVM-9.0 pass to instrument a posit program with calls to our runtime to maintain metadata, propagate metadata, and perform shadow execution. (2) A runtime written in C++ that performs shadow execution and propagates the metadata. The runtime by default uses the MPFR library to perform high-precision execution. It can be customized to run with any data type. (3) A clang-based refactorer that traverses the abstract syntax tree of a FP program and creates a posit program that uses the SoftPosit API. Given that posit is a stand-in replacement for FP, we wanted to experiment with large applications and existing numerical code without manually rewriting it. Although the refactorer automated most of this translation, we had manually change indirect function calls and global initializers in SPEC applications.

Usage. If the program already uses the SoftPosit API, the user of PositDebug prototype will use the instrumenter directly, link with the runtime, and generate an executable. Otherwise, the user will use our refactorer to rewrite the FP program to use the SoftPosit API and then use PositDebug's LLVM instrumentation to inject calls to the runtime and generate the executable. The user sets an error and reporting threshold (as an environment variable) and executes the binary. The prototype generates an overall summary of instances of error exceeding the threshold. It also generates a DAG of instructions for each error instance until the reporting threshold is met. The user of PositDebug can compile the runtime with debugging symbols and reason about the error using the DAG of instructions using the debugger.

Reporting error. Due to tapered accuracy with posits, reporting relative error as posit ULP error is misleading (see Section 2.3). We use the ULP error of the double representation to report the error because double can represent all $\langle 32, 2\rangle$-posit values exactly as normal values. Hence, PositDebug converts the posit value to a double, converts the MPFR value to a double, and then computes the ULP error of the two double values. PositDebug also reports the number of bits with error, which is $\lceil log_2(ulperror)\rceil$. It is important to note that a $\langle 32, 2\rangle$-posit has at most 27 fraction bits whereas double has 52 fraction bits. Therefore, even if a $\langle 32, 2\rangle$-posit is rounded correctly, it can have up to 25 bits of error.

4.3 FPSanitizer Prototype

To show the generality of the proposed ideas, we also built FPSanitizer, a shadow execution framework to detect numerical errors in FP programs. It consists of an LLVM pass that adds calls to our runtime and a runtime that performs high-precision shadow execution. The metadata organization, implementation details, and the usage mode of FPSanitizer are exactly identical to PositDebug. It does not support vectorization yet. The FPSanitizer prototype is also publicly available [9].

5 Experimental Evaluation

This section describes our experiments to evaluate PositDebug and FPSanitizer's ability to detect errors and the performance overhead. We illustrate the usefulness of the debugging support with case studies.

Methodology and applications. To evaluate PositDebug for its ability to detect numerical errors, we use a set of thirty two micro-benchmarks with known numerical errors. This set consists of twelve C programs that use FP from the Herbgrind suite [59] and 20 C programs written with posits that feature commonly used numerical algorithms. We converted FP C programs from the Herbgrind to use posits using PositDebug's refactorer. All these programs use the SoftPosit library for posits.

To evaluate performance, we use all C applications from PolyBench's linear algebra suite, SPEC-FP-2000, and SPEC-FP-2006. These applications are written with FP. We used PositDebug's refactorer to create posit versions of the application using the SoftPosit library. We use the $\langle 32, 2\rangle$-posit configuration for all our experiments as it is the recommended type in the SoftPosit library. We ran all experiments on an Intel Core i7-7700K machine that has four cores and 32GB of main memory. We measure the wall clock execution time of the application with PositDebug and without any shadow execution. As there is no publicly available hardware implementation of posits, our baseline uses the SoftPosit library. On average, these baseline applications with posits are 11× slower compared to the hardware FP versions.

5.1 Effectiveness in Detecting Numerical Errors

To test the effectiveness in detecting errors, we ran PositDebug's shadow execution with all thirty two programs in our test suite with numerical errors. PositDebug was able to detect errors in all of them. Among them, the output of 28 programs have more than 35 bits of error (at most 17 exact fraction bits with posit), the output of 24 programs are reported to have more than 45 bits of error (at most 7 exact fraction bits with posit), and the output of 18 programs are reported to have more than 52 bits of error (all fraction bits are wrong). PositDebug identified that 18 programs have catastrophic cancellation, 10 programs experience significant loss of precision, 5 programs have branch flips, 1 program has an incorrect integer cast, 2 programs produce *NaR* values, and two programs have saturation errors. We also observed numerical errors in six PolyBench and all the SPEC-FP applications.

Next, we inspected the DAG reported by PositDebug. The largest reported DAG had 12 instructions. It reported DAGs with a single instruction for 8 programs. When we further examined the program with the debugger, we observed that the operands of the single instruction DAG were not in the set of active functions. It was a function that had recently completed execution. We put a breakpoint at the return instruction of that function and we could obtain DAGs with 4 to 8 instructions for those 8 programs. Overall, the instructions in the DAG enabled us to understand the cause of error.

5.2 Case Studies of Debugging with PositDebug

This section describes our experience using PositDebug to debug and understand the cause of numerical error.

5.2.1 Posit Math Library with the CORDIC Method.

Posit still does not have an implementation of the math library for the $\langle 32, 2\rangle$-posit configuration. Hence, programmers have to use the math library for FP (*i.e.*, libm) by casting the posit value to double, use the libm function with double, and convert the double result back to a posit value. To build a math library for posits, we were exploring the feasibility of using the CORDIC (Coordinate Rotation Digital Computer) algorithm [65], which is attractive given that it can be implemented with just addition and subtraction operations. Our initial motivation for developing PositDebug was to debug our implementation of *sin* and *cos* function.

CORDIC is a class of iterative add-shift algorithms that can compute `sin` and `cos` of $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$:

$$x_0 = \prod_{i=0}^{k-1} \frac{1}{\sqrt{1+2^{-2i}}},\ \ y_0 = 0,\ \ z_0 = \theta$$
$$x_{n+1} = x_n - d_n y_n 2^{-n}$$
$$y_{n+1} = y_n + d_n x_n 2^{-n}$$
$$z_{n+1} = z_n - d_n tan^{-1}(2^{-n})$$
$$d_n = \begin{cases} 1 & if\ \ z_n \geq 0 \\ -1 & otherwise \end{cases}$$

As the total number of iterations k approaches ∞, $x_k = cos(\theta)$ and $y_k = sin(\theta)$. The value of x_0 for a given k and the values of $tan^{-1}(2^{-n})$ for $n \in [0, k-1]$ are compile-time constants. All other operands can be computed with addition, subtraction, and shift operations, which can be easily implemented on small FPGAs and ASICs. Further, they can be extended to compute other transcendental functions.

We implemented *sin* and *cos* functions with posits using CORDIC. Our implementation performs 50 iterations. To have high precision for compile-time constants, we precomputed values for the initial x_0 and tan^{-1} values using the MPFR library with 2000 bits of precision. Our posit implementation outperformed a similar implementation with float on 97% of the inputs in the range $[0, \frac{\pi}{2}]$. We observed that there was significant error for $sin(\theta)$ when θ was close to 0 and for $cos(\theta)$ for θ near $\frac{\pi}{2}$. We wanted to debug the program and identify the cause of error, which motivated us to build PositDebug.

To debug our implementation, we chose a specific input $\theta = 1.0 \times 10^{-8}$. Our implementation produced $sin(\theta) =$

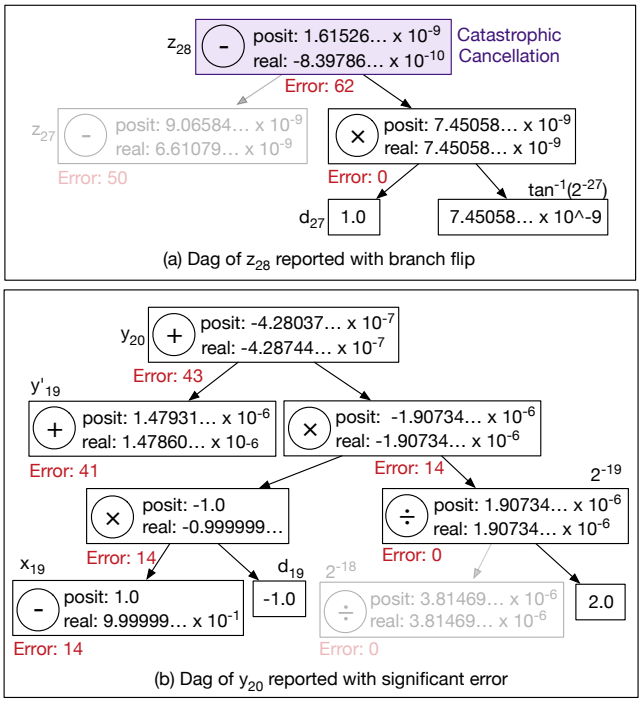

Figure 6. (a) The DAG of z_{28} reported by PositDebug when the branch flip occurred. There is a catastrophic cancellation for the variable used in the branch condition. (b) The DAG of y_{20} on the 20th iteration of CORDIC. The nodes in gray are not reported by PositDebug. We include it to understand the source of error.

$1.3162\cdots\times 10^{-8}$ as the result. We used libm's implementation and MPFR's implementation of *sin* as the oracle. These libraries do not use CORDIC but use polynomial approximation using the mini-max method. The libm implementation with double reported $1.0000\cdots\times 10^{-8}$ and the MPFR math library with 2000 precision bits reported $1.0000\cdots\times 10^{-8}$. Hence, the relative error of our implementation of *sin* with CORDIC is $0.3162\ldots$.

We wanted to debug this error and find the source of error for the same input. When we ran our implementation with PositDebug, it reported that the result of the shadow execution is $sin(\theta) = 1.2455\cdots\times 10^{-8}$ and the result had 48 bits of error (*i.e.*, it has at most 4 exact fraction bits in the $\langle 32, 2\rangle$-posit). It also reported that there were 4 instructions with less than 17 exact bits and 14 branch flips. On a branch flip, the shadow execution follows the execution path of the posit program. It is probably the reason that PositDebug's shadow execution produces a different result when compared to the result from MPFR's math library.

When we ran the shadow execution within the debugger (*i.e.*, gdb), we identified that our posit implementation encountered a branch flip in the 29th iteration while checking the condition $z_{28} \geq 0$ due to catastrophic cancellation. Hence, d_{28} was assigned 1 instead of -1.

Figure 6(a) shows the DAG produced by PositDebug for z_{28}, the cause of branch flip. It experiences catastrophic cancellation with 62 bits of error (all bits are inexact). The operands were z_{27} and $d_{27}\times tan^{-1}(2^{-27})$. The result of $d_{27}\times tan^{-1}(2^{-27})$ was exact with d_{27} being 1. By using gdb and tracing z_i for the previous iterations, we were able to find that the operand z_{27} had 50 bits of error (at most 2 fraction bits are exact). Additionally, we were able to infer that z_i's were slowly and constantly accumulating error. Hence, we identified that branch flip is one reason for our posit implementation to produce a different result compared to the oracle result.

Upon investigating other instructions before the branch flip with PositDebug and gdb, we identified that the value of y_{20} was incurring 43 bits of error (at most 9 exact fraction bits) and the value of y_{28} had 48 bits of error (at most 4 exact fraction bits). Figure 6(b) shows the DAG for y_{20}. Our initial θ is close to 0 and the value of d_n changes each iteration. The value of y, which is initially 0, is added and subtracted with values (*i.e.*, $x_n 2^{-n}$) that are gradually decreasing. Hence, the relative error of y increases gradually each iteration. The value of x_n also changes in a similar fashion. However, it starts with a relatively large value compared to the terms that are being added/subtracted. Hence, the final x does not exhibit much error. We are still identifying ways to rewrite our computation to avoid the branch flip and error accumulation with y. PositDebug was helpful in understanding the root cause of error while debugging our CORDIC implementations.

5.2.2 Computation of Integrals with Simpson's Rule.

Simpson's rule is an approximation algorithm to compute the integral of a function. Given an even number of intervals, n, an integral can be approximated by:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3}(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n))$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x$. As n approaches ∞, the result from this approximation converges to the value of the integral.

We implemented a program to compute the integral with the Simpson's rule using $\langle 32, 2\rangle$-posits. With $n = 20,000,000$, we computed the integral for the following input

$$\int_{13223113}^{14223113} x^2 dx = 1.8840\cdots\times 10^{20}$$

The output of our program was $1.5372\cdots\times 10^{17}$.

PositDebug reported that there were 4 instructions with more than 55 bits of error (no exact fraction bit in the answer) and 7 instructions with more than 35 bits of error (at most 17 exact fraction bits).

PositDebug's DAG also indicated that three of the four instructions with more than 55 bits of error were addition

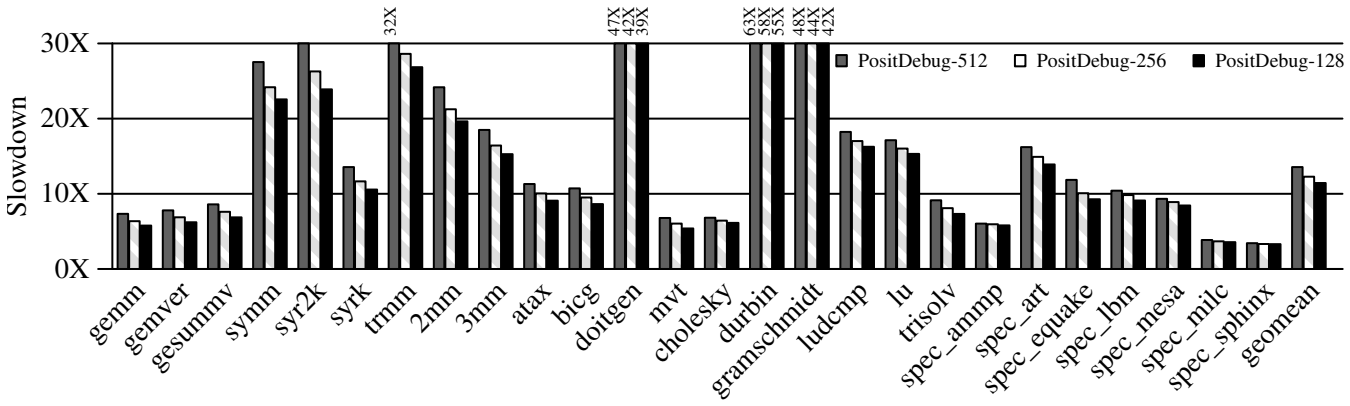

Figure 7. Performance overhead of PositDebug compared to a baseline that uses the SoftPosit library for various applications. It reports PositDebug's overhead with 512, 256, and 128 bits of precision for the shadow execution's MPFR value.

instructions, which was accumulating the terms. When we inspected the shadow execution with gdb, the program produced a value of 2^{63}, which has only 12 fraction bits available. All subsequent terms added to the above result were smaller than 2^{50} and the result was rounded down to 2^{63}. Even though the accumulation was slowly increasing the error, PositDebug still was able to identify the existence of error and accurately locate the source of error.

The cause of error is the lack of available precision bits for the accumulation operation. We replaced this accumulation operation with a fused dot product that uses the quire datatype. After this change, the benchmark produced the value $1.8850\cdots\times 10^{20}$, which is similar to the shadow execution value.

5.2.3 Root-Finding of a Quadratic Formula.

The formula to compute the roots of a quadratic equation is given by:

$$root = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

When computing with FP, there are two subtractions that can result in significant error: computation of $b^2 - 4ac$ and the computation of $-b + \sqrt{b^2 - 4ac}$.

In the case of posits, there is additional instruction that can cause error apart from two subtractions: division by $2a$. We identified this when we ran PositDebug with our implementation of the root-finding program with the following inputs:

$$a = 1.4396470127131522076524561271071 \times 10^{-14} \quad (5)$$

$$b = 3.24884063720703125 \times 10^{2} \quad (6)$$

$$c = 1.822878072832 \times 10^{12} \quad (7)$$

The roots returned by our $\langle 32, 2\rangle$-posit implementation are:

$$-5.29948672 \times 10^{9} \quad (8)$$

$$-2.2566376648474624 \times 10^{16} \quad (9)$$

PositDebug reported that the first root had 48 bits of error (*i.e.*, at most 4 exact bits). The DAGs generated by PositDebug indicated that $-b$ had 0 bits of error, $\sqrt{b^2 - 4ac}$ had 28 bits of error (at most 24 exact fraction bits), and $-b + \sqrt{b^2 - 4ac}$ had 48 bits of error (at most 4 exact fraction bits). Although the subtraction did not cause catastrophic cancellation in this instance, the cancellation of bits in the subtraction caused a significant increase in error.

Further, PositDebug reported loss of precision with the second root, which had 36 bits of error (at most 16 exact fraction bits). The result of $-b - \sqrt{b^2 - 4ac}$ incurred 26 bit of error (at most 26 exact fraction bits) and the result of $2a$ had 0 bits of error. However, the result of the division lost precision bits due to an increase in regime bits, which resulted in 36 bits of error (at most 16 exact fraction bits). Although multiplication and division in FP does not amplify the relative error with normal values, posit multiplication and division can amplify the relative error. We found PositDebug's output to be helpful in both identifying the amount of error and the cause of precision loss.

5.3 Performance Overhead with PositDebug

Figure 7 reports the performance overhead experienced by various applications with PositDebug's shadow execution compared to an uninstrumented posit application. For each benchmark, we report the slowdown of PositDebug with three different configurations based on the number of precision bits used for the MPFR value in the metadata: 512, 256, and 128 precision bits. The rightmost cluster of bars (geomean) reports the geometric mean of all benchmarks for each precision. On average, PositDebug's shadow execution has a slowdown of 13.6× with 512 bits of precision, 12.3× with 256 bits of precision, and 11.4× with 128 bits of precision compared to a baseline without any instrumentation.

As the overheads increase with an increase in precision for the MPFR value used in the metadata, MPFR computation is one major source of overhead. The baseline software

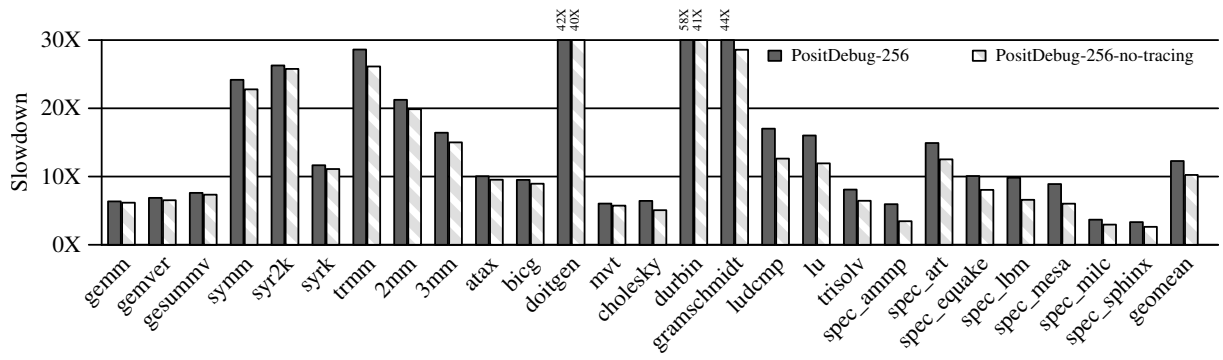

Figure 8. Performance slowdown of PositDebug with and without tracing for shadow execution with 256 bits of precision.

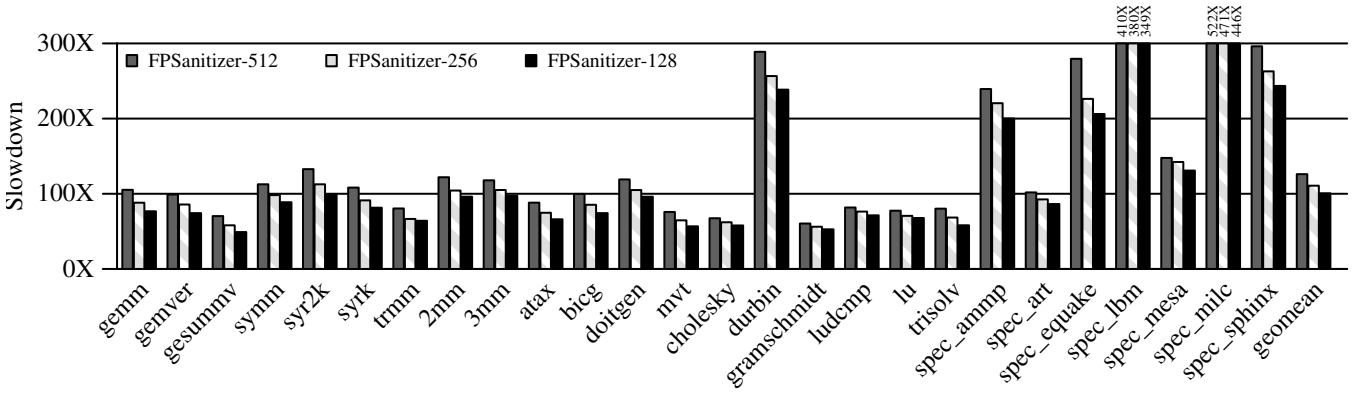

Figure 9. FPSanitizer's performance overhead with varying precision bits (512, 256, and 128) for the shadow execution compared to an uninstrumented baseline application that uses hardware FP operations.

posit version of SPEC applications is significantly slower compared to the hardware FP version. Hence, the additional overhead of PositDebug over the baseline software posit version is small. On average, PositDebug's overhead when compared to a baseline with hardware FP operations is 225×, 174×, and 157× for 512, 256, and 128 precision bits for the MPFR value, respectively.

To understand the overhead of the additional mechanisms for tracing, we evaluate a version of PositDebug that detects numerical errors but does not provide tracing information. Figure 8 reports the overhead of PositDebug with and without tracing support when the shadow execution is performed with 256 bits of precision. The removal of tracing reduces the overhead on average from 12.3× to 10.2× for PositDebug in this configuration. Some benchmarks with compact loops such as `durbin` and `gramschmidt` see a significant reduction in overhead (*i.e.*, from 58× to 41× for `durbin` and from 44× to 29× for `gramschmidt`). Overall, we found these overheads to be acceptable while debugging applications. An effective strategy is to run PositDebug without tracing, detect the error, and subsequently run PositDebug with tracing selectively for the functions of interest rather than the entire program, which we found useful in our experience.

5.4 Performance Overhead with FPSanitizer

Figure 9 reports the performance overhead of FPSanitizer's shadow execution, which uses the same metadata organization as PositDebug, compared to an uninstrumented hardware FP baseline. Similar to PositDebug, we evaluate FPSanitizer's overheads with increasing precision for the shadow execution's MPFR value. On average, FPSanitizer's shadow execution has a performance overhead of 126×, 111×, and 101× with 512, 256, and 128 bits of precision, respectively.

Herbgrind [59], which is the state-of-the-art shadow execution framework for FP applications, crashed with all these applications in our evaluation. We were able to run Herbgrind with PolyBench applications with smaller inputs. In these cases, we observed that FPSanitizer was more than 10× faster than Herbgrind on average.

All SPEC applications have a higher memory footprint than PolyBench applications and have higher overhead. Applications `spec_milc`, `spec_sphinx`, and `spec_lbm` have a large number of cache misses even in the baseline without FPSanitizer. Accesses to metadata increase the memory footprint causing more cache misses at all levels. Further, the software high-precision computation prevents memory-level

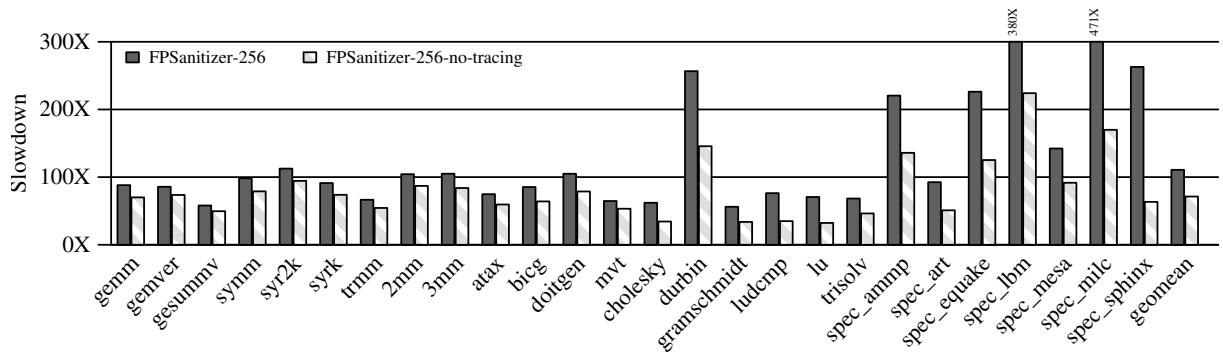

Figure 10. Performance slowdown of FPSanitizer with and without tracing for shadow execution with 256 bits of precision.

parallelism that overlaps misses and reduces the effectiveness of the prefetcher.

Figure 10 reports the overhead with and without tracing for FPSanitizer. On average, the performance overhead decreases from 111× to 71×. Additional overhead with metadata for tracing is significant for applications with a higher memory footprint. Overall, we found FPSanitizer to be usable with long-running SPEC applications.

6 Related Work

Posit is gaining significant attention both from software developers and hardware vendors. There is an active community that has developed various software implementations [38, 54] and FPGA implementations [30]. Posits and its variants have been used in a wide variety of contexts: deep neural network [6, 14, 31, 44] and weather simulation [32]. For these applications, it has been shown that posit produces more accurate results than the floating point or the fixed point representations while also being computationally efficient.

Although posits are shown to be accurate under certain circumstances, it is prone to new types of errors. Dinechin et. al. [13] provide an analysis of advantages, disadvantages, and some sources of error with posits, making a case for debugging tools for posits. Posit community is still in its infancy and lacks debugging tools as of now. PositDebug is the first tool that helps programmers debug sources of numerical error, which includes programmers who are either writing new applications with posits or are porting FP code to use posits.

PositDebug is also related to prior tools for detecting and debugging FP errors. There is a large body of tools to analyze error in FP programs [2–4, 7, 8, 12, 15, 17–19, 22, 23, 28, 36, 37, 42, 43, 55, 58–62, 62, 66, 67]. Among them, Herbgrind [59] and FPDebug [4] are the most closely related to PositDebug. Both Herbgrind and FPDebug perform binary instrumentation with Valgrind to perform shadow execution with higher precision. FPDebug does not provide much support for debugging numerical errors. Herbgrind also stores the error and the dynamic traces for each memory location in the metadata space, analyzes the offending instruction that causes a program instability, and provides expression DAGs that could possibly be rewritten with Herbie [55]. However, the size of the metadata space per memory location is proportional to the number of dynamic FP instructions, which restricts its use with long-running applications. PositDebug is inspired by Herbgrind. However, it addresses the limitations of Herbgrind by maintaining a constant amount of metadata per-memory location and with compiler instrumentation that enables effective debugging with gdb in the context of posits.

7 Conclusion

Posit is a usable approximation of real numbers with tapered accuracy. Posit can provide higher precision than floating point for a certain range of values (*i.e.*, golden zone), which can be pivotal in some applications. Posit, like any representation with tapered accuracy, also introduces new concerns while programming with them. PositDebug is the first tool that performs shadow execution with high-precision values, detects errors, and provides DAGs of instructions that are responsible for the error in applications using posits. We built a shadow execution framework for floating point programs, FPSanitizer, using the same design, which is an order of magnitude faster than the state-of-the-art. We found the debugging support to be useful while implementing and debugging a wide range of posit applications.

Acknowledgments

We thank our shepherd Zachary Tatlock and the anonymous PLDI reviewers for their feedback. We also thank John Gustafson for numerous clarifications and inputs on the Posit representation. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, NSF Award CCF-1908798, and NSF Award CCF-1917897.

References

[1] 2008. *IEEE Standard for Floating-Point Arithmetic.* IEEE 754-2008. IEEE Computer Society. 1–70 pages.

[2] 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)*. ACM, New York, NY, USA, 817–832.

[3] 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)*. ACM, New York, NY, USA, 549–560.

[4] 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)*. ACM, New York, NY, USA, 453–462.

[5] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep Learning with Low Precision by Half-Wave Gaussian Quantization. In *2017 IEEE Conference on Computer Vision and Pattern Recognition*. 5406–5414.

[6] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks. In *Proceedings of the Conference for Next Generation Arithmetic 2019*. ACM Press.

[7] 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.

[8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-point Errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 43–52.

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

[10] Sangeeta Chowdhary, Jay P Lim, and Santosh Nagarakatte. 2020. *PositDebug.* Retrieved March 23rd, 2020 from https://github.com/rutgers-apl/PositDebug

[11] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. In *CoRR*, Vol. abs/1602.02830.

[12] 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)*. ACM, New York, NY, USA, 235–248.

[13] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: the good, the bad and the ugly. In *CoNGA'19 - Conference for Next Generation Arithmetic*. ACM Press, Singapore, Singapore, 1–10. https://hal.inria.fr/hal-01959581

[14] Seyed H. Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. 2018. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 19–23.

[15] 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.

[16] 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.

[17] Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated Backward Error Analysis for Numerical Code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 639–654.

[18] Zhoulai Fu and Zhendong Su. 2019. Effective Floating-point Analysis via Weak-distance Minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 439–452.

[19] 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.

[20] 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.

[21] S. Golomb. 2006. Run-length Encodings (Corresp.). In *IEEE Transactions on Information Theory*, Vol. 12. IEEE Press, Piscataway, NJ, USA, 399–401.

[22] 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.

[23] 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.

[24] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning* (Lille, France) *(ICML'15)*. JMLR.org, 1737–1746.

[25] John Gustafson. 2015. *End of Error: Unum Computing.* Chapman & Hall/CRC Computational Science.

[26] John Gustafson. 2017. *Posit Arithmetic.* https://posithub.org/docs/Posits4.pdf

[27] John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. In *Supercomputing Frontiers and Innovations: an International Journal*, Vol. 4. South Ural State University, Chelyabinsk, Russia, Russia, 71–86.

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

[29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[30] Manish Kumar Jaiswal and Hayden K.-H So. 2018. Universal number posit arithmetic generator on FPGA. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1159–1162.

[31] Jeff Johnson. 2018. *Rethinking floating point for deep learning.* http://export.arxiv.org/abs/1811.01721

[32] Milan Klöwer, Peter D. Düben, and Tim N. Palmer. 2019. Posits As an Alternative to Floats for Weather and Climate Models. In *Proceedings of the Conference for Next Generation Arithmetic 2019* (Singapore, Singapore) *(CoNGA'19)*. ACM, New York, NY, USA, Article 2, 8 pages.

[33] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint:

An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. abs/1711.02213.

[34] Ulrich W. Kulisch. 2002. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units.* Springer-Verlag, Berlin, Heidelberg.

[35] Ulrich W. Kulisch. 2013. *Computer Arithmetic and Validity: Theory, Implementation, and Applications.* De Gruyter. https://books.google.com/books?id=kFR0yMDS6d4C

[36] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying Bit-manipulations of Floating-point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI âĂŹ16)*. ACM, New York, NY, USA, 70–84.

[37] 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)*. ACM, New York, NY, USA, 623–638.

[38] Cerlane Leong. 2018. *SoftPosit.* https://gitlab.com/cerlane/SoftPosit

[39] Fengfu Li and Bin Liu. 2016. Ternary Weight Networks. In *CoRR*, Vol. abs/1605.04711.

[40] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 2849–2858.

[41] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. 2018. Universal Coding of the Reals: Alternatives to IEEE Floating Point. In *Proceedings of the Conference for Next Generation Arithmetic* (Singapore, Singapore) *(CoNGA '18)*. ACM, New York, NY, USA, Article 5, 14 pages.

[42] Matthieu Martel. 2009. Program Transformation for Numerical Precision. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Savannah, GA, USA) *(PEPM '09)*. ACM, New York, NY, USA, 101–110.

[43] David Monniaux. 2008. The Pitfalls of Verifying Floating-Point Computations. In *ACM Transactions on Programming Languages and Systems*, Vol. 30. Association for Computing Machinery, New York, NY, USA, Article 12, 41 pages.

[44] RaÃžl Murillo Montero, Alberto A. Del Barrio, and Guillermo Botella. 2019. Template-Based Posit Multiplication for Training and Inferring in Neural Networks. arXiv:cs.CV/1907.04091

[45] Robert Morris. 1971. Tapered Floating Point: A New Floating-Point Representation. In *IEEE Transactions on Computers*, Vol. C-20. 1578–1579.

[46] Jean-Michel Muller. 2005. *On the definition of ulp(x).* Research Report RR-5504, LIP RR-2005-09. INRIA, LIP. 16 pages. https://hal.inria.fr/inria-00070503

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

[48] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*.

[49] Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. 2013. Hardware-Enforced Comprehensive Memory Safety. In *IEEE MICRO Top Picks of Computer Architecture Conferences of 2012*.

[50] Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. 2015. Everything You Want to Know about Pointer-Based Checking. In *Proceedings of SNAPL: The Inaugural Summit On Advances in Programming Languages*.

[51] 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*.

[52] Thomas R. Nicely. 2011. *Pentium FDIV flaw FAQ.* http://www.trnicely.net/pentbug/pentbug.html

[53] 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

[54] Theodore Omtzigt. 2018. *Universal Number Arithmetic.* https://github.com/stillwater-sc/universal

[55] 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.

[56] Kevin Quinn. 1983. Ever Had Problems Rounding Off Figures? This Stock Exchange Has. *The Wall Street Journal* (Nov. 1983).

[57] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *European Conference on Computer Vision (ECCV '16)*, Vol. 9908. Springer, 525–542.

[58] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. ACM, New York, NY, USA, Article 27, 12 pages.

[59] 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)*. ACM, New York, NY, USA, 256–269.

[60] Geof Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 229–238.

[61] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 53–64.

[62] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *Formal Methods (Lecture Notes in Computer Science)*, Vol. 9109. Springer, 532–550.

[63] Andrew Tulloch and Yangqing Jia. 2017. High performance ultra-low-precision convolutions on mobile devices. In *ArXiv*.

[64] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

[65] Jack Volder. 1959. The CORDIC Computing Technique. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference* (San Francisco, California) *(IRE-AIEE-ACM '59 (Western))*. ACM, New York, NY, USA, 257–261.

[66] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and Fixing Precision-specific Operations for Measuring Floating-point Errors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 619–630.

[67] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-point Errors in Numerical Libraries. In *Proceedings of the ACM on Programming Languages*, Vol. 3. ACM, New York, NY, USA, Article 56, 29 pages.