© 2022

Sangeeta Chowdhary

ALL RIGHTS RESERVED

FAST METHODS TO DETECT AND DEBUG NUMERICAL ERRORS WITH SHADOW EXECUTION

By

SANGEETA CHOWDHARY

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Santosh Nagarakatte

And approved by

____

____

____

____

New Brunswick, New Jersey

October, 2022

ABSTRACT OF THE DISSERTATION

Fast Methods to Detect and Debug Numerical Errors with Shadow Execution

by **Sangeeta Chowdhary**

Dissertation Director: Prof. Santosh Nagarakatte

The floating-point (FP) representation uses a finite number of bits to approximate real numbers in computer systems. Due to rounding errors, arithmetic using the FP representation may diverge from arithmetic with real numbers. For primitive FP operations, the rounding error is small and bounded. However, with the sequence of FP operations, rounding errors can accumulate. Such rounding errors can be magnified by certain operations. The magnified error can affect the program's control flow and the output compared to execution with infinite bits of precision. It is challenging for users to identify such bugs because the program does not crash but generates the incorrect output. Without any oracle in high precision, it is hard to differentiate between correct and incorrect output. Detecting such bugs in long-running programs becomes even more challenging.

This dissertation proposes a fast yet precise mechanism to detect and debug numerical errors in long-running programs. This dissertation makes the following contributions: First, we propose a selective shadow execution framework to detect and debug numerical errors. Our idea is to use shadow execution with high-precision computation for comprehensive numerical error detection. On every FP computation, an equivalent high precision computation is performed. If there is a significant difference between FP computation and high precision computation, the error is reported to the user. We use additional information about instructions to generate a directed acyclic graph (DAG) of them showing the

error propagation. The DAG helps the user identify the error's root-cause. Our prototype FPSanitizer for floating-point is an order of magnitude faster than prior work.

Second, we propose a novel technique to run shadow execution in parallel to further reduce performance overheads. In our approach, the user specifies parts of the program that need to be debugged. Our compiler creates shadow execution tasks that mirror these specified regions in the original program but perform equivalent high precision computation. To execute the shadow tasks in parallel, we break the dependency between them by providing the appropriate memory state and input. Moreover, to correctly detect the numerical errors in the original program, shadow tasks must follow the same control flow as the original program. Our key insight is to use FP values computed by the original program to start the shadow tasks from an arbitrary point in time. To ensure they follow the same control flow as the original program, our compiler updates every branch instruction in the shadow task to use the branch outcomes of the original program. As a result, the original program and shadow tasks execute in a decoupled fashion and communicate via a non-blocking queue. Our prototype PFPSANITIZER is significantly faster than the FPSANITIZER. On average, PFPSANITIZER provides a speedup of $30.6\times$ speedup over FPSanitizer with 64 cores.

Finally, we propose an alternative lightweight oracle to reduce the overheads of shadow execution. Executing the shadow execution on multiple cores in parallel reduces the performance overheads. However, the user must identify regions of the program to enable shadow execution in parallel. Often, users may not know where the numerical bugs are present. This thesis proposes a fast shadow execution framework, EFTSANITIZER, that uses error-free transformations (EFTs) to detect and debug numerical bugs comprehensively. EFTs are a set of algorithms that provide a mechanism to capture the rounding error using primitive FP instructions. For certain FP computations rounding error can be represented as an FP value. Based on this observation, EFTs transform FP computation to derive the rounding error. In our approach, we maintain the error with each memory location and compute the error for a sequence of FP operations using error composition with EFTs.

EFTSanitizer provides a trace of instructions that help users isolate and debug the root cause of the errors. In addition, EFTSanitizer is an order of magnitude faster ($14.72\times$) than FPSanitizer. In our experimental studies, EFTSanitizer detected all errors as detected by FPSanitizer. However, the error reported by EFTSanitizer may not be as precise as reported by FPSanitizer due to loss of precision with error accumulation.

ACKNOWLEDGMENTS

I would like to thank my advisor, Santosh Nagarakatte. Santosh has put tremendous effort into honing my critical thinking, reading, and writing skills for the last six years. I will always be thankful for all the effort he has put into building my research career.

I would like to thank my doctoral committee, Richard Martin, Mridul Aanjaneya, and Sreepathi Pai, for their insightful feedback that has improved my dissertation. I would also like to thank Srinath Setty and Kim Laine, who mentored me during my internships at Microsoft Research and helped me during my job search. In addition, I would like to thank the faculty members at Rutgers, Srinivas Narayan, and Rich Martin. When I felt lost during my Ph.D. journey, they helped me to see the bigger picture and motivated me to continue my journey. I would also like to thank Chris Kanich, who mentored me during one amazing year I spent at UIC.

During my Ph.D., I have made many friends who made my Ph.D. journey endurable. First, I would like to thank my lab-mates, Mohammedreza Soltaniyeh, Adarsh Yoga, Jay Lim, David Menendez, Matan Shachnai, Harishankar Vishwanathan, and Sehyeok Park, at the RAPL research group for their support and motivation during the time I spent at Rutgers. I would like to thank Saswat Padhi for his constant support throughout my Ph.D. journey. I would also like to thank Georgiana Haldeman for all the help and support during my job search.

Lastly, I would like to thank my parents, Darshan, Shaymveer, my brother Vivek, and sister-in-law Tanya. Without their support and encouragement, I can not imagine finishing my Ph.D. journey.

To my parents

TABLE OF CONTENTS

Abstract ii

Acknowledgments v

List of Tables xiii

List of Figures xiv

Chapter 1: Introduction 1

1.1 What are Rounding Errors? 1

1.2 Why is it Challenging to Debug Rounding Errors? 3

1.3 Inlined Shadow Execution 4

1.4 Thesis Statement 5

1.5 Contributions of This Dissertation 5

1.6 Inlined Selective Shadow Execution 6

1.7 Parallel Shadow Execution 8

1.8 Inlined Shadow Execution with a Light-Weight Oracle 10

1.9 Papers Related to this Dissertation 12

1.10 Organization of This Dissertation 12

Chapter 2: Background 14

2.1 The Floating Point Representation 14
2.1.1 A Tiny Floating-Point Number System 16
2.1.2 Rounding Modes 17
2.1.3 Rounding Errors 18
2.2 Inlined shadow Execution with Reals 19

Chapter 3: Inlined Selective Shadow Execution 21
3.1 Shadow Execution for Comprehensive Error Detection 22
3.2 Detection and Debugging Numerical Errors with Shadow Execution 23
3.3 Selective Shadow Execution 26
3.4 Instrumentation Mode 27
3.5 Metadata Design 28
3.5.1 Metadata for Temporaries 30
3.5.2 Metadata for FP Value Stored in Memory 34
3.6 Metadata Propagation 34
3.6.1 Creation of FP Constants 34
3.6.2 Metadata for FP Binary Operations 36
3.6.3 Metadata Propagation for Memory Store 36
3.6.4 Metadata Propagation for Memory Load 37
3.6.5 Metadata Propagation for Function Arguments and Function Return 39
3.6.6 Detection and Debugging of Errors 40
3.7 Running Example 42
3.8 Implementation Details of Our Prototype 42

3.8.1 Shadow Memory 43
3.8.2 Shadow Stack 43
3.8.3 Management of Lock and Key Metadata 44
3.8.4 Usage 45
3.9 Summary 45

Chapter 4: Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors 47

4.1 High-Level Overview of Our Approach 48
4.2 How Does Our Compiler Generate Shadow Tasks? 50
4.2.1 Modified original program 54
4.2.2 Shadow execution task 55
4.3 Dynamic Execution of Original Program and Shadow Tasks 57
4.3.1 Metadata to Detect and Debug Errors 58
4.3.2 Shadow Execution from an Arbitrary Memory State 59
4.3.3 Detecting and Debugging Errors 61
4.4 Illustration of Our Approach 61
4.5 Implementation Considerations 64
4.5.1 Shadow Memory Organization 65
4.5.2 Management of Temporary Metadata Space 65
4.5.3 Handling Indirect Function Calls 66
4.5.4 Support for Multithreaded Applications 66
4.5.5 Usage with Interactive Debuggers 66
4.6 Summary 67

Chapter 5: A Lightweight Oracle Using Error-Free-Transformations for Shadow Execution . . . 70

5.1 Computing the Rounding Error with Error Free Transformations . . . 72

5.1.1 Computing the Rounding Error for an FP Addition Operation . . . 73

5.1.2 Propagating the Error of the Operands with Addition . . . 74

5.1.3 EFTs for Subtraction . . . 75

5.1.4 Computing the Rounding Error for FP Multiplication . . . 75

5.1.5 Propagating the Rounding Error with Multiplication. . . . 76

5.1.6 Computing and Propagating the Rounding Error of the FP Division Operation . . . 76

5.1.7 Computing and Propagating the Error for Square Root . . . 77

5.2 The EFTSANITIZER Approach . . . 78

5.2.1 Error Free Transformations for Shadow Execution . . . 79

5.2.2 Debug Information to Illustrate the Propagation of Rounding Errors 79

5.2.3 Metadata Design and Organization of the Metadata Space . . . 80

5.2.4 What Should We Store in Each Metadata Entry? . . . 81

5.2.5 Organization of the Metadata Space . . . 82

5.2.6 Reusing the Entries of the Temporary Metadata Space . . . 82

5.2.7 Metadata Propagation . . . 84

5.2.8 Error Reporting and Debugging Interface . . . 89

5.3 Implementation Considerations . . . 90

5.3.1 Shadow Memory, Shadow Stack, and Temporary Metadata Space . 90

5.4 Illustrative Example . . . 91

5.5 Summary . . . 95

Chapter 6: Experimental Evaluation . . . 96

6.1 Experimental Evaluation of FPSANITIZER and EFTSANITIZER . . . 96

6.1.1 Prototype . . . 96

6.1.2 Methodology . . . 97

6.1.3 Effectiveness in Detecting and Debugging Numerical Errors . . . 98

6.1.4 Performance Evaluation of FPSANITIZER and EFTSANITIZER . . 104

6.2 Experimental Evaluation of PFPSANITIZER . . . 108

6.2.1 Prototype . . . 108

6.2.2 Methodology . . . 108

6.2.3 Placement of Directives . . . 109

6.2.4 Ability to Detect FP Errors . . . 110

6.2.5 Performance Evaluation of PFPSANITIZER . . . 111

Chapter 7: Related Work . . . 113

7.1 Static Analysis for Detecting Numerical Errors . . . 113

7.2 Dynamic Analysis for Detecting and Debugging Numerical Errors . . . 115

7.2.1 Shadow Execution Based Analysis . . . 115

7.2.2 Instruction-Based Analysis . . . 116

7.2.3 Prior Work on Error Free Transformations . . . 117

7.3 Parallel Dynamic Analysis . . . 118

7.4 Precision Tuning to Reduce Errors . . . 119

7.5 Identifying Inputs with High FP Error . . . 120

Chapter 8: Conclusion and Future Directions . . . 121

8.1 Dissertation Summary 121

8.1.1 Detecting and Debugging Numerical Errors in Computation with Floating-Point 122

8.1.2 Parallel Shadow Analysis To Accelerate the Debugging of Numerical Errors 123

8.1.3 Shadow Analysis With Error Free Transformations 125

8.2 Future Research Directions 126

8.2.1 C Program Reduction for Numerical Bugs 126

8.2.2 Detecting and Debugging Numerical Errors in Scripting Languages 126

8.2.3 Improving the Accuracy of an Oracle Based on EFTs 127

LIST OF TABLES

2.1 Bitstrings for 8-bit FP representation. 17

6.1 Summary of our experiments with EFTSANITIZER. 99

LIST OF FIGURES

1.1 Rounding error example. 2

1.2 Overall design of FPSanitizer. 8

1.3 Overall design of PFPSANITIZER. 9

2.1 Floating-Point 32-bit and 64-bit formats. 15

2.2 Normal and special numbers in FP-32 bit format. 16

2.3 Inlined Shadow Execution. 19

3.1 Dynamic shadow execution trace. 22

3.2 Multiple designs to store metadata. 25

3.3 Metadata stored for each FP variable. 28

3.4 Lock-and-key metadata. 29

3.5 Selective shadow execution. 31

3.6 Running example with FPSANITIZER. 41

3.7 DAG generated for the catastrophic cancellation. 41

4.1 Transformations done by the PFPSANITIZER's compiler. 51

4.2 Parallel execution of shadow execution tasks. 52

4.3 Timestamp execution graph of the original program. 53

4.4 Metadata maintained with temporaries and in shadow memory. 56

4.5 Execution of shadow tasks. 58
4.6 DAG of instructions generated by PFPSANITIZER. 62

5.1 Error free transformations for addition. 73
5.2 Differences in DAGs generated by PFPSANITIZER and FPSANITIZER. . . 80
5.3 Running example with EFTSANITIZER. 81
5.4 Reuse of the temporary metadata space entries. 93
5.5 DAG generated for the illustrative example. 94

6.1 DAG generated by EFTSANITIZER. 100
6.2 Spurious rounding error reported by EFTSANITIZER. 103
6.3 Slowdown experienced with FPSANITIZER. 105
6.4 Performance slowdown of FPSANITIZER. 105
6.5 Speedup achieved with EFTSANITIZER compared to FPSANITIZER. 105
6.6 Slowdown with EFTSANITIZER. 108
6.7 Speedup of PFPSANITIZER over FPSANITIZER. 110
6.8 Slowdown experienced with PFPSANITIZER. 111

CHAPTER 1
INTRODUCTION

The IEEE floating-point representation approximates real numbers in computer systems. The FP representation is used in numerous domains, such as weather simulations, aerospace engineering, machine learning, gaming, graphics, etc. The FP representation provides many advantages. First, the FP representation is standardized, allowing reproducibility across different machines. Second, hardware support for the FP representation in modern machines allows writing fast code compared to a software simulation of FP. Moreover, the FP representation allows for representing very large and tiny numbers while providing reasonable accuracy.

1.1 What are Rounding Errors?

The FP representation represents a finite set of real numbers. The real numbers outside this set are rounded to the nearest representable FP numbers. The FP representation is specified by a total number of bits divided into three components: a sign bit, exponent bits, and precision bits. The sign bit indicates whether the number is positive or negative. The number of exponent bits defines the dynamic range of such representation, i.e., the smallest representable number and the largest representable number. The number of precision bits defines the accuracy of such a representation. All real numbers within the dynamic range that are not FP numbers are rounded to the nearest FP number, resulting in a small rounding error. The rounding error per floating-point instruction is bounded. However, rounding errors can be accumulated or magnified with certain operations resulting in counter-intuitive results. For example, if two very close values are subtracted, the entire output can be influenced by the rounding error, which is known as catastrophic cancellation.

C Program	Float-32	Real Arithmetic
1. int main(){		
2. float a = 0.5f;		
3. float b = 0.0013f;		
4. float c = a + b;	0.50133997	0.50133999
5. float d = c - b;	0.4999	0.5
6. float e = d - a;	-2.9802E-08	0.0
7. float f = e * 2.0E+8;	-5.95	0.0
8. }		

Figure 1.1: This example shows the simple C program resulting in wrong output due to rounding errors in comparison to real arithmetic.

The real numbers outside the dynamic range are rounded to the special numbers. If the real number is greater than the largest representable number, it may overflow. If the real number is smaller than the smallest representable number, it may result in an underflow. In such cases, the overflowed value is rounded to a special number Inf, and underflowed value is rounded to zero. The FP format also defines a special value NaN (Not-a-Number) to represent invalid numbers, such as $0/0$, $\sqrt(-2)$.

The rounding errors and exceptional values may influence the control flow of the program, may lead to the slow convergence of numerical algorithms, and may result in the wrong output. Unsurprisingly, such rounding errors have resulted in many catastrophic incidents in the past. For example, the Patriot missile failure in 1991 due to rounding error resulted in a loss of 28 lives [7]. Recently, a driverless race car crashed into a wall due to a floating-point exception [10]. In this incident, the steering control signal resulted in a NaN, and subsequently, the steering locked to the maximum value to the right.

1.2 Why is it Challenging to Debug Rounding Errors?

The rounding errors may result in various numerical bugs in a program, such as wrong output, branch divergence, conversion errors, and floating-point exceptions. Debugging such errors is challenging for many reasons. First, numerical bugs are triggered by only a small set of inputs. Hence, if the program is not tested for problematic inputs, such errors can remain undetected. Second, almost every floating-point computation results in rounding errors, but only a small set of rounding errors are magnified due to accumulation. These magnified rounding errors become problematic if the program's output, branch instruction, or system call depends on them. Hence, it is challenging to identify if rounding errors are problematic or not. Third, rounding errors span function boundaries, memory locations, and function arguments before they are magnified, leading to a numerical bug. Hence, it is challenging to find the source of the error.

To highlight the challenges in detecting and debugging numerical errors, consider the program in Figure 1.1. This simple program computes the expression $((x+y)-y)-x)*z)$. This expression results in zero for all inputs in real arithmetic. However, in floating-point arithmetic, some inputs result in a different output. For example, if $x = 0.5$, $y = 0.0013$, and $z = 2.0E + 8$ with float-32 this program returns -5.95 rather than 0. Figure 1.1 shows the output for each instruction with real arithmetic and with FP arithmetic. Analyzing each instruction with FP-32 and real arithmetic shows a small rounding error in line 4 has propagated to line 5 and is amplified in 6, leading to the wrong result in line 7. Hence, a small rounding error in line 4 was magnified to produce the wrong output. Unfortunately, identifying such sources for numerical bugs is challenging without a fine-grained comparison of each FP instruction with high-precision computation. Further, debugging numerical errors becomes even more challenging with long-running programs as instructions responsible for the error could spread across multiple functions and memory locations.

Often numerical errors occur due to insufficient precision or dynamic range in the FP representation. Therefore, developers often rewrite the program with higher precision. Unfortunately, increasing the precision would reduce the program's performance. On the other hand, changing the precision for a small set of variables can often provide the desired accuracy. Hence, identifying the sequence of instructions or the expression responsible for the numerical bug would help the user meticulously increase the precision or rewrite the expression to avoid numerical bugs.

1.3 Inlined Shadow Execution

To identify sources of numerical bugs at the granularity of instructions, a promising technique is inlined shadow execution [6, 99]. In this technique, the high-precision instruction is inlined with the floating-point instruction and compared to detect the numerical bugs. For every FP operation, an equivalent high-precision value is retrieved from shadow memory, and the same operation is performed in high-precision. Further, additional information about the instructions is stored in the shadow memory for debugging purposes. Additional information about instructions helps in providing a backtrace of the program, showing the error propagation for root-cause analysis. However, prior shadow execution-based techniques focused on debugging numerical errors with small programs. In contrast to prior approaches, our goal is to enable shadow execution-based techniques to debug numerical errors for long-running programs. The main reasons for the performance overheads introduced by such a technique are the use of software simulated high-precision computations as an oracle and the design of metadata stored in shadow memory.

In summary, prior work has made significant progress in detecting and debugging numerical errors, but they incur huge overheads. Therefore, these approaches are not practical for long-running applications.

1.4 Thesis Statement

This dissertation develops novel abstractions to detect and debug numerical errors in long-running floating-point applications.

1.5 Contributions of This Dissertation

This thesis proposes various techniques to reduce the overheads of shadow execution to enable debugging numerical errors in long-running applications. The main contributions of this thesis can be summarized as follows:

This dissertation employs shadow analysis with real numbers to comprehensively detect numerical errors. In this approach, real computations are executed in lock-step with FP computations, providing a mechanism to check numerical errors. Moreover, each shadow variable stores extra information about the instructions to provide a backtrace once the error is detected. Hence, the approach enables comprehensive error detection and provides valuable feedback to the user.

1. Inlined selective shadow execution. This dissertation proposes a mechanism to detect and debug numerical errors comprehensively. In our approach, high-precision computations are executed in lock-step with FP computations, providing a mechanism to detect all numerical errors. To debug the numerical errors, we provide the directed acyclic graph (DAG) of instructions to show the error propagation. In contrast to prior approaches, we store a constant amount of information in shadow memory resulting in low overhead. We selectively perform shadow execution for code regions of interest to further reduce the overheads. To enable selective shadow execution we need to start the shadow execution at some arbitrary time. We use the computed value to reset the shadow memory state and start the shadow execution from an arbitrary point in the dynamic execution. Our shadow execution framework FPSanitizer is 10× faster than the state-of-the-art.

2. Parallel shadow execution. This thesis proposes a novel approach to accelerate the debugging of numerical errors by running shadow execution in parallel. In our approach, the user specifies parts of the program that need to be debugged. Then, our compiler creates shadow execution tasks that mirror the original program for these specified regions but perform the equivalent high precision computation. Since we are creating shadow tasks from a sequential program, shadow tasks are also sequential depending on prior tasks for memory state. To execute the shadow tasks in parallel, we need to break the dependency between them by providing the appropriate memory state and input arguments. Our key insight is to use FP values computed by the original program to start the shadow task from some arbitrary point. Finally, the runtime library automatically distributes the workload fairly among all cores resulting in significant speedups over the state-of-the-art.

3. A lightweight oracle to detect numerical errors. The parallel shadow execution framework requires user input to create the shadow task, and numerical errors are detected within the shadow task. This approach is not ideal when a user does not have the insight to create shadow tasks which can result in critical numerical bugs. This dissertation proposes a lightweight oracle with a smart metadata design to detect numerical errors. The oracle uses error-free transformations(EFTs) to capture the rounding errors. Since EFTs are designed using hardware-supported floating-point instructions, the oracle introduces lower overheads than high-precision computations. Our key insight is to use EFTs to automatically detect numerical errors for the sequence of operations with practical debugging support.

1.6 Inlined Selective Shadow Execution

Our goal is to enable developers to detect and debug numerical errors in long-running applications. We perform shadow execution with high-precision computations to detect numerical errors. To enable the user to debug the root-cause of the error, we provide a

directed acyclic graph (DAG) of instructions responsible for the error. Based on our observation, mostly instructions responsible for the error are local, hence, the DAG shows the instructions in an active stack frame. Our approach enables the user to diagnose the root-cause of the error using debuggers *i.e.* gdb. We export some functions where the user can set breakpoints and generate the backtrace (DAG) of the instructions responsible for the error.

Our approach transforms a program at compile-time to enable shadow execution with high-precision computations. At compile-time, FP variables are either resident in registers or memory. We shadow every FP variable with a high-precision value. The FP variables in the register are local to a function. Hence, we store equivalent high-precision values in the separate stack. The separate stack shadows the original program call stack and we call it a shadow stack. For FP variables in memory, we store equivalent high-precision values in shadow memory. The shadow memory is organized as a hash-map and the memory address is used as a key. For every FP computation, we retrieve the equivalent high-precision value from shadow memory and perform the same computation with higher precision. To detect the numerical error, we compare the high-precision value with the original computed value. If the difference between the computed value and the high-precision value is above the predefined threshold, we report the error to the user. As a result, we comprehensively detect numerical bugs such as rounding errors due to precision loss, catastrophic cancellation, unexpected branch outcomes, float-to-int conversions, and FP exceptions.

To enable users to debug numerical errors, we store additional metadata about the FP instructions in shadow memory. We reconstruct the directed acyclic graph of instructions using additional information in the active stack frame. The DAG of instructions shows the error propagation, enabling the user to localize the source of the numerical bug.

To further reduce the overheads in long-running applications, we realized that users often debug a small code region. Hence, we provide a mechanism to selectively instrument the interesting code regions within the application. In our approach, we use the computed

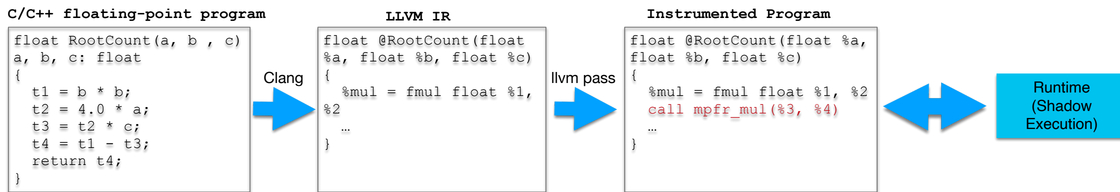

Figure 1.2: This figure shows the overall design of FPSanitizer.

value to reset the state at some arbitrary point in the dynamic execution trace of the program. Hence, using selective instrumentation, users can start the instrumentation at any random point in the program execution.

The contribution of our approach is the design of the metadata for each variable in register and memory. In our approach, we store the constant amount of metadata per memory location which reduces overheads significantly in contrast to prior approaches. In addition, we provide the mechanism to selectively instrument the program to further reduce the overheads. Figure 1.2 shows the overall design of our prototype FPSANITIZER. Our prototype, FPSANITIZER, is open-source and publicly available [19]. It is built on top of the LLVM compiler to instrument the program to enable inline shadow execution. FPSanitizer's runtime performs shadow execution. We get a slowdown of $111\times$ with FPSanitizer and $10\times$ speedup over the state-of-the-art. Chapter 3 describes our approach in detail.

The following sections describe the two orthogonal approaches to attain low overheads with comprehensive error detection and debugging.

1.7 Parallel Shadow Execution

Inline shadow execution enables debugging numerical errors with long-running applications. However, this approach still incurs significant overheads compared to the instrumented program. Based on our analysis, the primary source of overhead is the software emulation of high-precision computation using the MPFR library. Moreover, additional metadata stored along with the high-precision value adds to the performance over-heads.

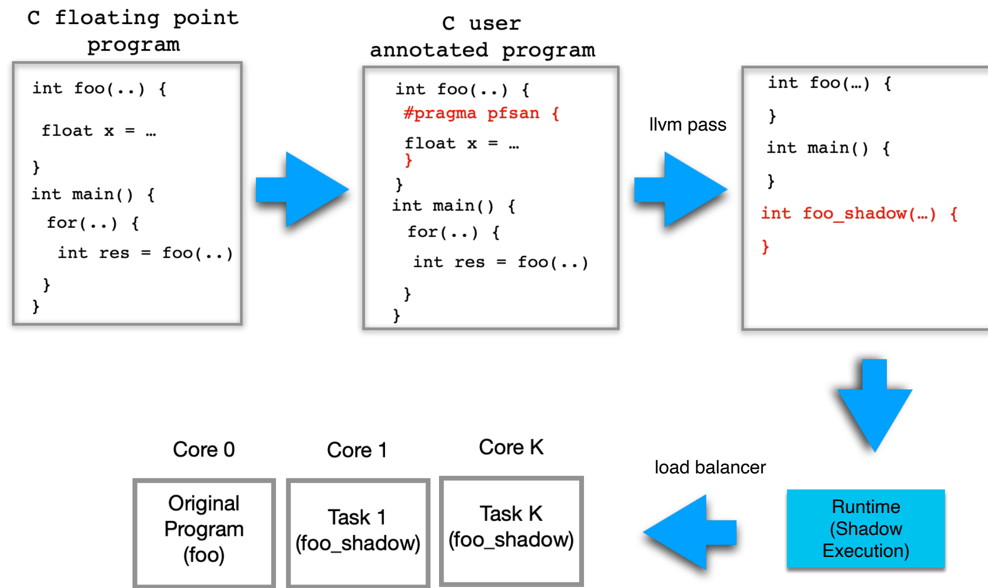

Figure 1.3: This figure shows the overall design of our approach to accelerate debugging of numerical errors by executing shadow execution in parallel on multiple cores.

Hence, high overheads prevent productive debugging of numerical bugs in long-running applications.

This dissertation proposes a parallel shadow execution framework PFPSanitizer to debug numerical errors with long-running applications. Our goal is to run a high-precision computation in parallel with the original FP program. Our compiler creates an equivalent program with high precision, mirroring the original program's FP computations. This design enables us to run the high-precision program parallel with the original FP program. However, the high-precision program is significantly slower than the original FP program giving us almost no speedup. We split the high-precision program into subprograms (shadow tasks) based on user input to get scalable speedups, as shown in Figure 1.3. Since we create shadow tasks from a sequential program, shadow tasks are also sequential. To break these dependencies between shadow tasks, we need to provide a memory state from prior tasks. Our key insight is to use the computed values from the original program to reset the memory state and break the dependencies making shadow tasks dependent on

the original program and not on each other. We also need to ensure that shadow tasks follow the same control flow as the original program. Our compiler updates the branch in the shadow task to take the same branch condition as the original program. The original program provides the branch conditions, original FP values, and memory addresses for the shadow task through the queues.

The key challenge is identifying when to reset the memory with the original computed values to break the dependencies between shadow tasks. Hence, on every read from shadow memory, we need to check if the value is dependent on the prior shadow task. To identify such scenarios, we store the original FP value in the metadata in shadow memory. Then, when the shadow execution task loads the value from the shadow memory, we check if the loaded FP value matches the stored FP value. If they match, then the high-precision value has been written by that task, and we do not need to reset the memory state. On the other hand, if they do not match, we reset the memory state and use the original FP value as the high-precision value. Hence, our key insight to use the original FP value to reset the metadata allows us to execute shadow execution tasks from an arbitrary point in time.

For practical debugging of numerical errors, we store additional information about the instructions in the metadata in shadow memory. Using additional metadata, we generate the DAG of instructions showing the error propagation for root-cause analysis.

Our prototype PFPSanitizer is publically available [21]. In addition, PFPSanitizer is approximately $30\times$ faster on average on a machine with 64-cores when compared to FPSanitizer, which is the state-of-the-art for debugging FP programs. Chapter 4 describes our approach in detail.

1.8 Inlined Shadow Execution with a Light-Weight Oracle

Parallel shadow execution helped us detect and debug numerical errors in long-running applications with low-performance overheads. However, PFPSanitizer requires user input to create the shadow tasks in high-precision at compile time. If the user creates enough

shadow tasks, we get low overheads. However, often users may not have insight into the code structure. Hence, this dissertation proposes an alternative mechanism to reduce overheads significantly to enable debugging numerical errors in production-based code.

Our key observation is that overheads come from using a high-precision software library as an oracle. Hence, in our approach, we use hardware-supported FP instructions to capture the rounding error of primitive FP instructions. We have designed a lightweight oracle using Error-Free-Transformations (EFTs) instructions for primitive FP computations and a high-precision math library for math functions. EFTs are a set of algorithms that uses properties of FP arithmetic to capture the rounding error of FP primitive instructions [78]. For primitive FP instructions, the rounding error is small and bounded. For certain FP operations, this rounding error is a FP number and can be stored precisely using the same FP representation. Using such FP properties, EFTs provide a mechanism to capture this rounding error for primitive FP operations. Using EFTs, a FP computation can be transformed such that $a \cdot b = x + y$, $x = a \odot b$, $y = err(a \odot b)$, where $\cdot$ is a real computation, and $\odot$ is a FP computation. For every FP primitive instruction, we compute the error by using EFTs. To capture the rounding error for math functions, we use a high-precision math library. The key challenge is to compose the rounding error for the sequence of FP operations. We use basic arithmetic to compose the error of the FP computation to get the first-order approximation of the error.

Our key contribution is the design of shadow execution using EFTs as an oracle. Similar to FPSanitizer, we transform the program at compile-time to enable shadow execution with our new oracle. We shadow every FP variable with a rounding error associated with that variable. For every FP value in a register and memory, we store the rounding error in the 64-bit FP variable in metadata in shadow memory. Using EFTs, the computed error is at least as good as the error computed with double-double arithmetic. Furthermore, we store additional information about FP instructions in our metadata to enable debugging of numerical errors. We provide the dynamic backward slice of a program to diagnose the

root cause of the error. FPSanitizer provides a DAG of static instructions for root cause analysis in an active stack frame. Hence, with FPSanitizer, the DAG does not provide the information after function calls and multiple iterations of a loop.

In contrast to FPSanitizer, we generate the DAG for fixed-size dynamic instructions for debugging of numerical errors. Using this approach, we do not lose the capability to generate the DAG of instructions if the stack frame is deallocated.

Our prototype EFTSanitizer achieves an average speedup of $14.72\times$ over FPSanitizer and detects almost all numerical errors as FPSanitizer without user input. Chapter 5 describes our approach in detail.

1.9 Papers Related to this Dissertation

The ideas and techniques presented in this dissertation are drawn from the following published papers written in collaboration with my advisor Santosh Nagarakatte and Jay Lim.

1. ”Debugging and Detecting Numerical Errors in Computation with Posits” [18], which introduces selective inlined shadow execution and metadata design to debug numerical errors in the context of floating-point and posits.

2. ”Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors” [20], which introduces parallel shadow execution to reduce the overheads of inline shadow execution.

3. ”Debugging Numerical Errors with Error Free Transformations” introduces the lightweight oracle with metadata design and effective debugging support to comprehensively detect and debug numerical errors with floating-point applications.

1.10 Organization of This Dissertation

Chapter 2 provides the background on FP representation, rounding errors, error-free-transformations. Chapter 3 presents our selective inline shadow execution with reals to

detect and debug numerical errors with FP applications. Chapter 4 presents our approach to significantly reduce the overheads of inline shadow execution. Chapter 5 presents the alternative approach to reduce overheads by using hardware-supported floating-point instructions to capture the rounding error. Chapter 6 evaluates the correctness and performance of our three main contributions - selective inline shadow execution, parallel shadow execution, and lightweight oracle. Chapter 7 discusses the prior work related to numerical error detection and debugging. Chapter 8 concludes this dissertation by presenting future directions.

CHAPTER 2
BACKGROUND

We provide a brief overview of the FP representation, the cause of rounding errors, an overview of inlined shadow execution, and a comparison of existing approaches.

2.1 The Floating Point Representation

The floating-point (FP) representation specified by the IEEE-754 standard [24] 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, 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, double).

In the IEEE-754 binary FP representation, the FP value is represented by a bitstring that consists of a sign bit (s), exponent bits that represent the unsigned (biased) exponent (e), and mantissa bits (p) that represent the fraction. The goal of the FP representation is to encode both large and very small values. 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.

In IEEE FP binary representation, an FP number x can be expressed as $(-1)^s \times m \times 2^e$, where s represents the sign of the number, m is the significand of the number, and e is the exponent. The bit pattern of m is expressed as $\pm(b_0.b1b2....b_{p-1})$ with precision p. Alternatively, x can be represented as $(-1)^s \times M \times 2^{e-p+1}$, where M is the integral significand. In this representation, 2^{e-p+1} is also called **ulp(x)**, where 2^{1-p} is **machine epsilon**.

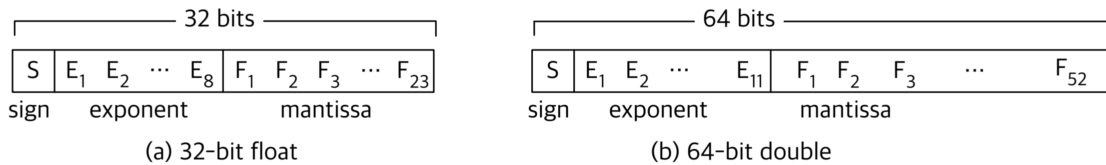

Figure 2.1: The bit-string of the float and double formats in the IEEE-754 binary FP representation.

In the IEEE-754 binary FP representation, biased representation is used to store the positive and negative exponent of a FP value. In this representation, a midpoint or bias is selected. Then, half of the values below the midpoint are used to encode negative exponent and another half to encode positive exponent values. For example, 8 bits are used to store the exponent in single precision. With 8 bits to store the exponent, the exponent ranges from 1 ... 254 (0 and 255 are used to store special values). IEEE-754 binary FP representation defines bias as $2^{|e|-1} - 1$, where $|e|$ represents the number of exponent bits. For the 8-bit exponent, the bias is 127. Using bias as 127, exponent values from -126 to +127 can be stored in the range 1 to 254. Using biased representation, the exponent field can be interpreted as an unsigned integer. Hence, two floating values with the same sign can be compared using an integer comparator.

When the exponent bits are not all zeros and not all ones (*i.e.*, e $\in [1, 2^{|e|} - 2]$, where $|e|$ represents the number of exponent bits), then the bit-string represents a normal value. For normal numbers, the leading bit b_0 is 1 and is not stored explicitly, called a hidden bit. The value represented by the FP bit-string is $(-1)^s \times 2^{e-bias} \times (1 + f/2^{|f|})$, where $|f|$ is the number of bits used to represent the fraction.

When the exponent bits are all zeros, it represents subnormal values and leading bit b_0 is 0. It is used to represent values close to zero. The value represented by the bitstring is $(-1)^s \times 2^{1-bias} \times (f/2^{|f|})$. When the exponent bits are all ones, it represents special values. When 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 the bitstring represents Not-a-number (NaN), which is used to represent exceptional conditions.

	sign	exponent	mantissa
x = 3.5	0	10000000	11000000000000000000000
x = Inf	0	11111111	00000000000000000000000
x = NaN	0	11111111	10000000000000000000000
	1-bit	8-bits	23-bits

Figure 2.2: This figure shows the representation of normal and special numbers in FP-32 bit format.

The commonly used float format has 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa or the fraction. The double format has 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. Figure 2.1 shows the bit-patterns used for the 32-bit float and 64-bit double formats. Figure 2.2 shows the representation for a 32-bit float, which has a sign bit, 8-bits for the exponent, and 23-bits for the fraction.

2.1.1 A Tiny Floating-Point Number System

To demonstrate the FP number system and its properties, let us consider a tiny floating-point number system with a total of 8 bits, with 1-bit for a sign, 4-bits for exponent, and 3-bits for a fraction. With this FP configuration, bias is $2^{(4-1)} - 1 = 7$, where 4 is the number of bits to store the exponent. For normalized FP numbers, the range for bitstrings for the exponent field is (0001 - 1110) in binary and (1 - 14) in decimal. Hence, in this representation the smallest exponent, e_{min}, is stored as $(0001)_2$ or 1_{10}. By using bias, we can drive the smallest exponent, $e_{min} = -6$. When we store the exponent, we add the bias to the exponent. Hence, we will store e_{min} as $(0001)_2$. For subnormal numbers, the exponent is fixed to $e_{min} = -6$, and the bitstring in the exponent field is all zero.

Table 2.1 shows the bitstring for positive FP numbers and how they are interpreted in decimal. In FP representation, if all bits in the exponent field are 1 and all bits in the precision field are 0, it represents $\pm infinity$. On the other hand, if all bits in the exponent field are 1 and all bits in the precision field are not zero, it represents $\pm NaN$ (Not-A-

Table 2.1: This table shows the bitstrings for a 8-bit FP representation and the numerical values these bitstrings represent.

Bit String	Numerical Value
0 0000 000	$0 \times 2^{-6} = 0$
0 0000 001	$\frac{1}{8} \times 2^{-6} = \frac{1}{512}$, smallest positive number
0 0000 010	$\frac{1}{4} \times 2^{-6} = \frac{1}{256}$
.	.
.	.
0 0001 000	$(1+0) \times 2^{-6} = \frac{1}{64}$, smallest positive normal number
0 0001 001	$(1+\frac{1}{8}) \times 2^{-6} = \frac{9}{512}$
.	.
.	.
0 0010 000	$(1+0) \times 2^{-5} = \frac{1}{32}$
.	.
.	.
0 1110 111	$(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}) \times 2^{7} = 240$, largest positive normal number
0 1111 000	Inf
0 1111 001	NaN
0 1111 010	NaN
.	.
.	.
0 1111 111	NaN

Number). Hence, more than one bitstring is used to represent NaN. An alternative Posit representation [48] avoids wasting more than one bitstring to represent NaN.

In floating-point representation, **machine epsilon** ϵ is defined as the gap between 1 and the number greater than 1, which is $2^{-(p-1)}$, where p is the precision. The precision for FP representation is defined as a number of fraction bits, including the hidden bit. For the tiny FP representation, $p = 1 + 3 = 4$, hence, $\epsilon = 2^{(-3)}$, and $ulp = 2^{e-p+1} = \epsilon \times 2^{e}$.

2.1.2 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), and round to nearest ties to even (RN).

In RD (round towards $-\infty$) rounding mode, x is rounded to x_l. In RU (round towards $+\infty$) rounding mode, x is rounded to x_h. In RZ rounding mode, x is rounded to x_l if $x > 0$ or x_h if $x < 0$. In RN rounding mode, x is rounded to either x_l or x_h, whichever is closer to x. When x is in the middle of x_l and x_h, it is a tie, and the ties-to-even approach is used by default to break the tie. 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.

2.1.3 Rounding Errors

Rounding a real value, which is not exactly representable in a FP representation, to the nearest FP number results in a 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 ties to even mode is at most half of the gap between x_l and x_h. If x_{fp} has an exponent e, then $|x_{fp} - x| < 2^{-p} \times 2^e$ or $|x_{fp} - x| < \frac{1}{2} \times 2^{e-p+1}$ or $|x_{fp} - x| < \frac{1}{2} \times ulp(x_{fp})$, where p is the precision of the FP representation [78].

The IEEE-754 standard mandates correct rounding of primitive operations. Hence, the rounding error of any primitive operation for the round to nearest ties to even mode is bounded by the gap between two adjacent FP values. However, this error can be amplified by operations such as subtraction that can cancel all the leading bits such that remaining bits are influenced by rounding errors. Hence, this accumulation of errors with a sequence

C/C++ floating-point program

```
float RootCount(a, b , c)
a, b, c: float
{
  t1 = b * b;
  t2 = 4.0 * a;
  t3 = t2 * c;
  t4 = t1 - t3;
  return t4;
}
```

Instrumented Program

```
float RootCount(a, b , c, a_h, b_h, c_h)
a, b, c: float
a_h, b_h,c_h: mpfr_t
{
  t1 = b * b;
  t1_h = h_mul(b_h, b_h);
  report_error(t1, t1_h);
  t2 = 4.0 * a;
  t2_h = h_mul(4.0, a_h);
  report_error(t2, t2_h);
  t3 = t2 * c;
  t3_h = h_mul(t2_h, c_h);
  report_error(t3, t3_h);
  t4 = t1 - t3;
  t4_h = h_sub(t1_h, t3_h);
  report_error(t4, t4_h);
  return t4;
}
```

Figure 2.3: This figure shows the original program and lock-step inlined high-precision computation to detect numerical errors.

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 [6, 52, 78].

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 [38]. Next, we describe how we can compute this rounding error using FP operations itself.

2.2 Inlined shadow Execution with Reals

One way to detect such numerical errors is by comparing the results of the FP program and the program that is rewritten with real numbers (*i.e.*, differential analysis). Such an approach can detect errors but does not help in debugging because it is infeasible to store all intermediate results and compare them. A lock-step inlined shadow execution [6, 18, 99] where the analysis performs real computation after each instruction, maintains the real value with each variable in registers and memory, and checks error after each instruction is useful, as shown in Figure 2.3. The real numbers are simulated with a widely used GNU MPFR library, which is a C library for multiple-precision floating point computations with

correct rounding. By maintaining appropriate information with each memory location, such lock-step shadow execution can provide a directed acyclic graph (DAG) of instructions (*i.e.*, a backward slice of instructions) to debug an error [18, 99].

Herbgrind and FPDebug perform inlined shadow execution with high-precision computation to detect numerical errors. However, both tools use heavyweight binary instrumentation to enable shadow execution and have significant overheads. Our work advances by reducing the overheads with inlined shadow execution frameworks. FPSANITIZER introduced in the next chapter reduces the overhead by keeping the memory usage bounded. PFPSANITIZER introduces a novel idea to perform shadow execution in parallel to reduce the overheads with inlined shadow execution. PFPSANITIZER reduces overheads by order of magnitude while providing comprehensive detection and debugging support. EFTSANITIZER uses a lightweight oracle to detect numerical errors in contrast to heavyweight oracle such as MPFR. EFTSANITIZER introduces an oracle based on hardware-supported FP arithmetic rather than software-simulated high-precision computation used by prior tools. EFTSANITIZER significantly reduces overheads of inlined shadow execution and provides a directed-acyclic-graph (DAG) of instructions for root-cause analysis.

CHAPTER 3

INLINED SELECTIVE SHADOW EXECUTION

This chapter describes our approach to detect and debug numerical errors in long-running floating-point programs with low-performance overheads. Our approach performs inlined shadow execution with high-precision computation for floating-point programs. Any shadow execution framework analyzes the program at runtime and requires storing additional information called metadata. The size of the metadata and the way it is propagated directly influences the overheads introduced by shadow execution. Our key contribution is the design of a metadata space to reduce memory overheads while providing comprehensive error detection and debugging support for floating-point programs. Additionally, our goal is to enable shadow execution for arbitrary code regions in the program to reduce performance overheads further. This design goal would help us avoid the performance cost of shadow execution of the entire program. However, it is challenging to start shadow executing from an arbitrary point in the dynamic execution of the program. Our technique maintains the original floating-point (FP) computed value in the metadata in a shadow memory, inspired by Intel MPX [53]. For every load of a FP value, we compare the loaded FP value with the stored FP value in shadow memory. If they do not match, we reset the metadata, and shadow execution starts from this point correctly. Otherwise, we read the metadata from shadow memory. This idea of selective shadow execution enables the user to focus on small code regions without paying the cost of shadow execution for the entire program. This approach also enables us to handle uninstrumented code due to external library calls. Based on these ideas, we have built a tool, FPSanitizer. Our tool FPSanitizer is a shadow execution framework for floating-point programs, and it enabled us to find bugs in long-running real-world applications.

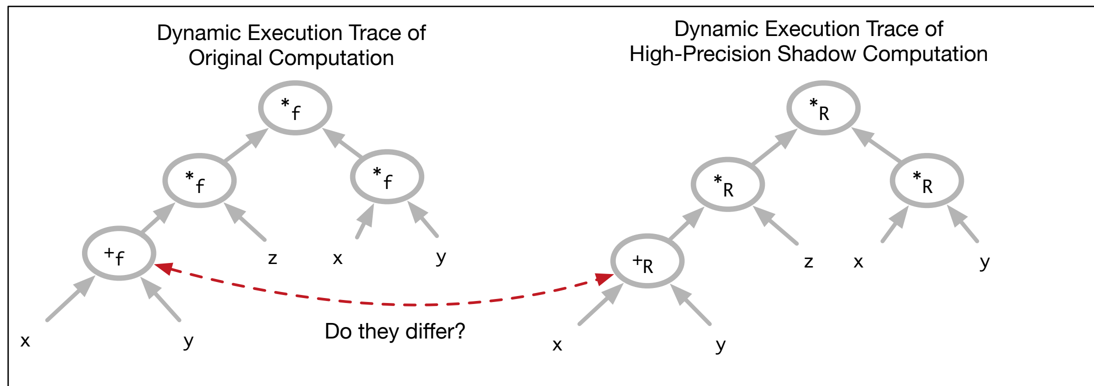

Figure 3.1: This figure shows the shadow execution of computations in high-precision side-by-side with the original program. In shadow execution, input variables are assumed with no error and stored in high-precision. A similar computation is performed side-by-side in high precision for every computation in the original program. To detect numerical errors, original computation is compared with the high-precision computation, and error is reported to the user if they differ by some user-defined threshold.

In the rest of the chapter, we describe the key ideas of our approach, how we encode the metadata for FP variables, how we propagate the metadata with each FP instruction, and how we enable debugging of numerical errors in FP programs.

3.1 Shadow Execution for Comprehensive Error Detection

A floating point number is an approximation of a real number using a finite number of bits. 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 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 [43]). Hence, such rounding errors can result in various numerical bugs - wrong outputs, branch divergences, slow convergence, wrong float-to-int conversions, and floating-point exceptions. One way to identify such

issues is by rewriting the program with higher precision and comparing the output. However, rewriting the program with higher precision requires effort. Further, just rewriting the program with higher precision will not help the user debug the root cause of the numerical bug, as numerical errors are not detected for each instruction. Detecting numerical errors for each instruction will help identify the set of instructions responsible for the error (root-cause analysis). Once a sub-expression responsible for the error is identified, the sub-regions of the program can be rewritten with higher precision [95]. Alternatively, the sub-expression responsible for the error can be rewritten to avoid numerical bugs [88].

Prior research [6, 99] has explored shadow execution with higher precision to detect and debug all numerical bugs for every instruction. In this approach, high-precision computation is performed side-by-side with the original FP computation and compared to detect numerical errors, as shown in Figure 3.1. In shadow execution, every FP instruction is monitored and compared with an oracle (high-precision computation). The availability of an oracle in high precision enables shadow execution to detect high-rounding errors due to accumulation and catastrophic cancellation. Moreover, branch divergences can be detected by comparing the result of a branch instruction in the original program and the equivalent high-precision value in shadow execution. Additionally, floating-point exceptions can be detected by checking the FP values in the original program. Hence, shadow execution with high-precision detects all numerical bugs compared to lightweight approaches that detect only a specific class of numerical errors [3, 36, 57, 63].

3.2 Detection and Debugging Numerical Errors with Shadow Execution

When one performs shadow execution with the high-precision value as an oracle, every variable in the program is shadowed with a high-precision value. For every original FP computation, an equivalent computation in high-precision is performed. The value computed by the original program is compared with the high-precision value to detect numerical errors with each instruction. If these two values differ significantly, the error is reported

to the user. Hence, we need to store high-precision values in shadow memory to detect numerical errors.

In floating-point arithmetic, the rounding error per primitive instruction is bounded [43]. However, rounding error accumulates with each instruction and magnifies due to catastrophic cancellation. Hence, instruction detected with high-rounding error is often not the root cause of the error. Beyond detection, we need to diagnose the root cause of the error and identify a set of instructions responsible for the error. This process of identifying why the numerical error has occurred and what is the set of instructions responsible for the error is the process of debugging. To help the user debug the error's root cause, we provide a backward slice of the program. To generate the backward slice of the program from the point of the error, we need to store additional information about the instructions. However, the key challenge is the design of metadata space to store additional information like high-precision value and information about instructions without incurring huge memory overheads.

Typically, we want to maintain metadata with variables. In any program, variables are either stored in memory or in registers. A single variable location can be written multiple times in a program by different dynamic instructions, *e.g.*, instruction within the loop body. While designing the metadata, we need to decide if we should maintain metadata of all dynamic writers or a single writer of a variable. Both of these design choices have pros and cons. For example, in Figure 3.2 (a), variable t1 can be stored in a register, or it can be stack-allocated. It is written multiple times within a loop. Do we maintain metadata for all dynamic writes to t1, or do we maintain metadata for just the last write of t1? If we maintain all writers, we would have more provenance for debugging the root cause of the error. However, metadata size would be directly proportional to the dynamic instructions in the program, as shown in Figure 3.2 (c). Hence, for long-running programs, we will quickly run out of memory. If we maintain just one writer, we will have information from just the last iteration of the loop, and metadata size will be proportional to the static instructions in

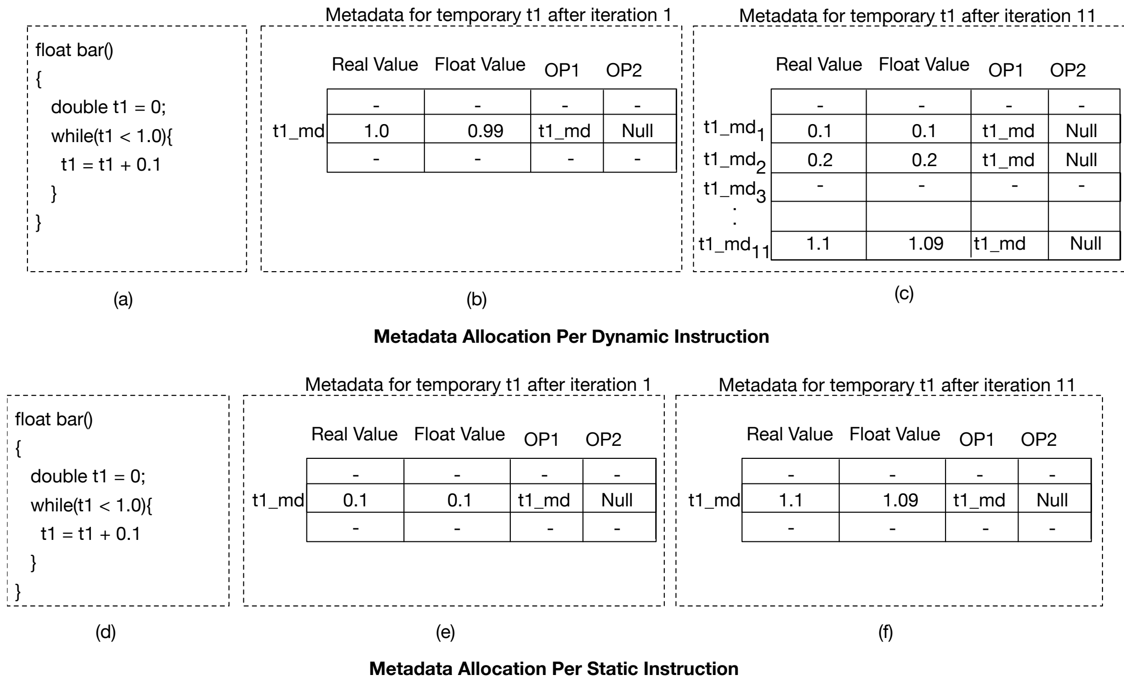

Figure 3.2: This figure shows that if metadata space is allocated for each dynamic instruction, then metadata size grows unboundedly. However, if the metadata space is allocated with each static instruction, metadata size is constant and proportional to the number of static instructions. In (a), instruction t1 is executed multiple times within the while loop body. In (b) and (e), we show the metadata state after the first iteration. In (c), metadata space is allocated for each dynamic instruction, and metadata size grows to the number of dynamic instructions in the program. In contrast, in (f), metadata space is allocated once and updated for the multiple executions of the instruction keeping the metadata space bounded.

the program. Prior approaches [6, 99] maintained metadata for all writers to provide more provenance to debug the root cause of the error. However, such a design works well for small programs, but not with long-running applications.

In our approach, we maintain metadata for the last dynamic writer of a variable, as shown in Figure 3.2 (f) to reduce memory overheads for long-running programs. If the instruction is updated in the loop, we update the metadata associated with the instruction. This insight to store metadata just for the last writer helps us keep the metadata space bounded, resulting in low memory overheads. To debug the numerical errors within the loop, we provide an interface for the user to put a breakpoint and check metadata for every loop iteration.

3.3 Selective Shadow Execution

In long-running applications, small code regions are often responsible for numerical bugs. However, shadow execution performed for the entire program leads to high-performance costs making it infeasible for long-running applications. We perform selective shadow execution for small code regions to reduce such overheads. However, the challenge with selective shadow execution is to provide the correct state of metadata for variables.

To perform selective shadow execution for small code regions, the user provides the list of functions of interest. Our compiler pass instruments only these functions. At the runtime, shadow execution starts from an arbitrary point in the dynamic execution of the program. At this point in time, for the first instruction within the shadow execution boundary, the equivalent operation is performed with high precision for the first time. Therefore, we need to retrieve operands with high-precision to perform the equivalent high-precision operation. However, operands of the first instruction where shadow execution starts do not have an equivalent high-precision value in the metadata because these variables might be computed outside the shadow execution boundary. Therefore, we need to identify such sce-

narios during the shadow execution automatically and provide the correct state of metadata for variables.

We automatically identify such cases by storing original computed values in the metadata inspired by Intel MPX [53]. Then, whenever a floating-point variable is read from memory, we check if the computed value stored in metadata is the same as the value read. If these two values match, then we read the high-precision value from the metadata. Otherwise, we reset the high-precision value to the computed value, and shadow execution starts correctly from this point. This approach also works well even when some code regions are not instrumented (*e.g.* calls to third-party libraries).

3.4 Instrumentation Mode

The program is transformed by instrumenting instructions of interest to perform shadow execution. Programs can be instrumented at different levels during the program's transformation from source code to binary. One such mechanism is source-to-source transformation. In this approach, the program can be performed by parsing source code to abstract-syntax-tree (AST) and annotating the parse tree to add the instrumented code. The source-to-source transformation has rich syntax and semantics while instrumenting the program. However, it is language-dependent and requires more effort to support multiple languages.

Another mechanism to instrument the program is at the binary level. Binary instrumentation can be performed statically or dynamically. In static binary instrumentation, the executable is transformed to generate a new instrumented executable without execution. In dynamic binary instrumentation, the program is transformed on-the-fly at runtime. Prior tools transformed the program using Valgrind [85], a dynamic binary instrumentation framework. However, the Valgrind framework is a heavy-weight tool that transforms the program at runtime before it starts executing. Although instrumenting binary is useful when source code is unavailable, modifying the program and generating code at runtime results in huge overheads ranging from $10\times$ to $100\times$.

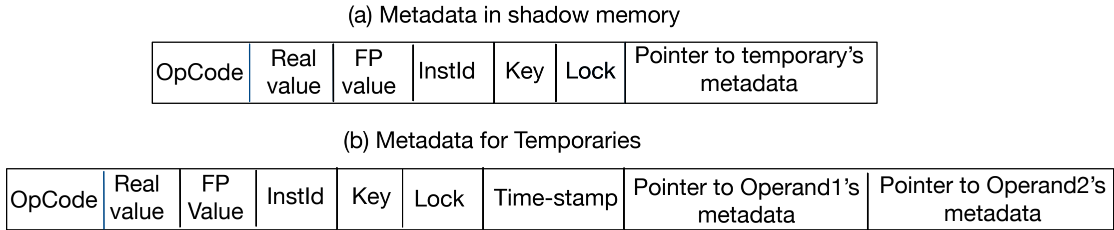

Figure 3.3: Metadata for (a) each FP value in shadow memory and (b) each FP temporary on the stack. Metadata in shadow memory maintains a pointer to the temporary metadata that was previously written to the memory location. It has the lock and key information to check the validity of the metadata pointer 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.

In contrast to prior approaches, we instrument the program at compile time to enable shadow execution with high precision. We have built our tool FPSanitizer on top of LLVM Intermediate Representation (IR) [71]. Our LLVM pass exploits the rich type system of LLVM IR to instrument the instructions of interest. Our approach to instrumenting the program at compile-time enables us to take advantage of low-level compiler optimizations and support multiple programming languages that can be compiled to LLVM-IR (C, C++).

In the next section, we describe our approach to creating and propagating metadata for different types of variables and instructions.

3.5 Metadata Design

Our idea is to transform a program at compile-time to enable shadow execution. At compile-time, FP values are either stored in memory or in registers. Since these variables have different lifespans, we design a different metadata space for variables in registers, memory, function arguments, and return values. For variables in registers (temporaries) and constants, we store metadata in the program stack. For variables in memory, we store metadata in shadow memory. Finally, for function arguments and return values, we store metadata in the shadow stack.

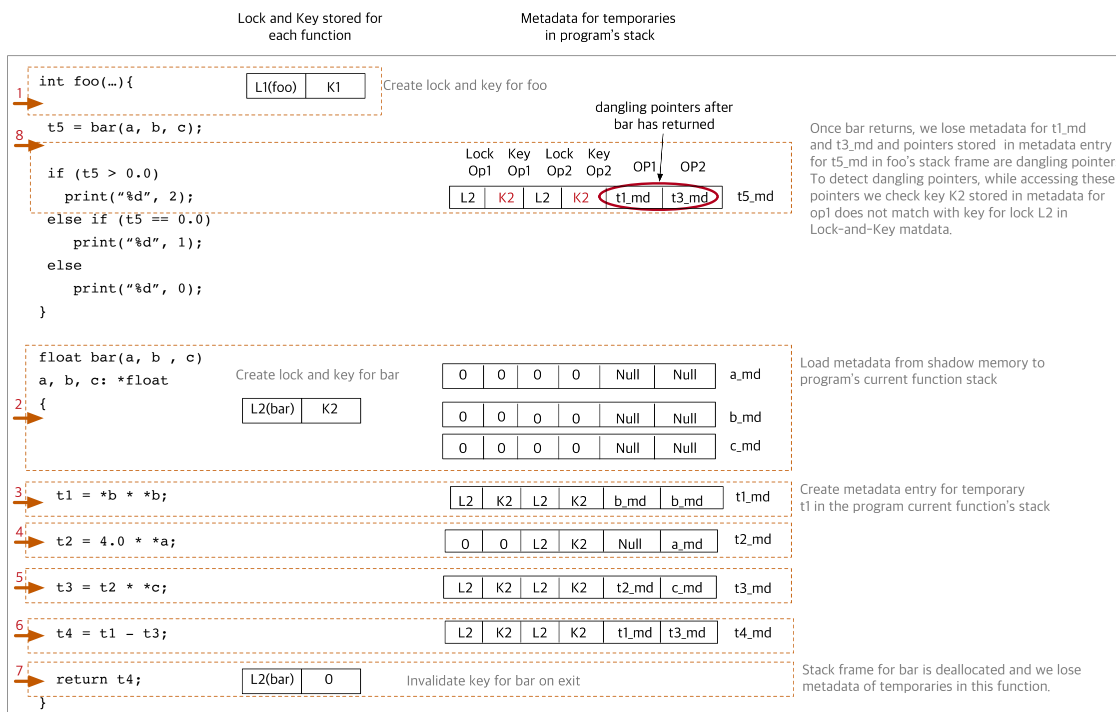

Figure 3.4: This figure shows the lock-and-key metadata to check for temporal safety. In the metadata for temporaries, we only show lock and key stored for both operands and pointers to operands' metadata. For step 1, function foo starts, and we create lock (function address) L1 and key (unique identifier) K1 in the global lock-and-key metadata space. All temporaries within this function will inherit the lock and key assigned for this function. In step 2, the function bar starts, and we create the lock-and-key for this function as L2 and K2. We also copy the metadata for function arguments from shadow memory to the current function stack. In step 3, we create the metadata for temporary t1. Similarly, in steps 4, 5, and 6, we create metadata for temporary t2, t3, and t4. Finally, in step 7, function foo returns, and we invalidate the key for function foo. Once the function foo returns, we lose the metadata stored in the foo stack frame for temporaries t1, t2, t3, and t4. In step 8, we create a metadata entry for t5 by copying the metadata for the return value from the shadow stack. The metadata for temporary t5 stores pointers to the metadata for t1 and t3 stored in the foo's stack frame. Since the foo's stack frame is deallocated, t1_md and t3_md are dangling pointers. Before we access the pointer t1_md, we compare the key K2 stored in the metadata space with key 0 stored in the lock-and-key metadata space for lock L2. Since they do not match, we do not access these pointers while generating a DAG of instructions for debugging.

3.5.1 Metadata for Temporaries

For FP variables in registers, we store metadata in the original program stack. By storing metadata in the program stack, we get metadata management for free. We allocate space for metadata in the program stack when the function starts, and it is reclaimed automatically once the function exits. The below section describes what metadata we store to detect the numerical errors, to enable selective shadow execution, and to enable debugging of the numerical errors.

Metadata For Error Detection

For every FP variable, we track high-precision values to detect rounding errors by comparing them with the original computed value. Hence, we store a high-precision value in the metadata for each FP value in a register to enable error detection. For every FP instruction in the original program, we retrieve the equivalent high-precision operands from the metadata and perform the same operation in high-precision. If the original computed value diverges from the high-precision value significantly then error is reported to the user.

Metadata to Enable Selective Shadow Execution

Using our approach, users can perform the shadow execution on particular code regions of interest in the program. Performing shadow execution for particular code regions of interest results in lower performance overheads than the entire program's shadow execution.

While performing selective shadow execution, certain variables will not have the correct metadata state if they were updated before the shadow execution started. Hence, any computation with such variables as operands within the shadow execution boundary will result in an incorrect high-precision result. Automatically identifying the state of such variables is challenging while running the shadow execution.

In our approach, we redundantly maintain the computed FP value in the metadata to reset the state of such variables while performing shadow execution. Whenever we perform

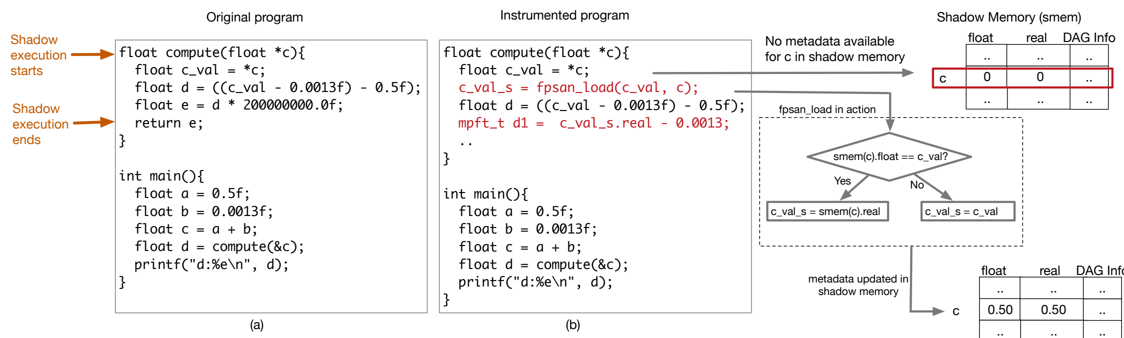

Figure 3.5: This figure shows our technique to reset the metadata state to enable selective shadow execution. In (a), the function main performs some computation and calls the function compute. This function dereferences a pointer variable c. The memory location stored in pointer c is updated in function main. This example assumes that the user marks only the compute function for shadow execution. Hence, our compiler only instruments the function compute to enable shadow execution, as shown in (b). However, the shadow location for the memory location stored in pointer c does not have a valid high-precision value because it was computed in the main function, which is outside the shadow-execution boundary. That is why high-precision computation in the next line would result in an incorrect value. Our compiler instruments the load instruction with a check to detect if metadata holds a valid high-precision value. This check is performed by comparing the computed value in metadata, which is 0 in this example, with the variable's original value, which is 0.50. Since these two values do not match, this check fails, and the metadata state is updated with the FP value pointed by variable c.

the equivalent high-precision computation, we store the high-precision value to detect numerical errors and the computed value for selective shadow execution. Whenever a FP variable is read in the program, we check if the variable read value is the same as the stored value. They would differ if the variable were written outside the shadow execution boundary. In such a case, the high-precision value of this variable will be reset to the original computed value, and shadow execution will start correctly. Our selective shadow execution technique is described in detail in Figure 3.5.

Metadata For Error Debugging

To enable debugging of numerical errors, we need to identify the instructions responsible for high rounding error. Hence, we store additional information about the instructions in the metadata for temporaries. Our goal is to generate a computation graph whose root is the instruction with high rounding error to show the error propagation with each node. To achieve this goal, we store a pointer to the operands' metadata for each temporary.

Additional metadata about operands enables us to generate a directed-acyclic-graph (DAG) of instructions by recursively accessing the pointers to the operands metadata in the active stack frame. If the stack frame is deallocated, accessing such a pointer would result in a memory safety error. Hence, we need a mechanism to detect temporal safety. Prior work [79, 80, 81, 82] has tackled this problem by associating extra information with each allocation and performing a runtime check if pointer access is safe. Prior work to check temporal safety can be broadly classified into location-based temporal checking and identifier-based temporal checking. In location-based temporal checking, with each memory allocation, some extra information is stored to record the status of each allocation and deallocation. At runtime, with each pointer access, the status of the memory location is consulted to flag safe or unsafe pointer access. This approach can detect unsafe pointer access if deallocated memory locations are never reallocated. For example, if some pointer tries to access the memory location that is being freed would be detected as a dangling pointer.

However, if the memory location is reallocated, the same pointer would be allowed to access the memory location due to insufficient information stored to detect a dangling pointer. Identifier-based temporal checking resolves this issue by providing the unique identifier for each memory allocation to differentiate it from the reallocated memory location with the same address.

In our approach, we want to detect such errors due to re-allocation. Otherwise, the DAG of instructions would result in the wrong information. Hence, we use identifier-based temporal checking from prior work to check temporal safety while accessing the pointers to generate the DAG.

Lock-and-Key Metadata for Temporal Safety

For temporal safety, we use the idea of Lock-and-Key from the prior work [79, 80, 81, 82]. In our approach, we need to detect the temporal safety of pointers to the metadata in the program stack. The pointer to the metadata is valid if it is pointing to the stack frame, which is not deallocated. If the pointer to the metadata is pointing to the deallocated stack frame then it is invalid. To detect such dangling pointers before accessing them, we provide the unique identifier for each stack frame in our approach. Hence, we provide a global unique identifier for every function at function entry (stack frame is allocated). All metadata entries within this function inherit a function address (lock) and the unique identifier (key). Once the function exits (stack frame is deallocated), the global unique identifier associated with this function is invalidated. Before any pointer access, we check if the unique identifier inherited by this pointer matches with the global unique identifier for the given function. If they match, then the pointer is safe to access. Otherwise, it is a dangling pointer. Our approach to detecting dangling pointers is described with an example in Figure 3.4.

We also store the timestamp in metadata space for temporaries. The timestamp monotonically increases for every FP instruction and records when the FP variable is updated. While generating the DAG of instructions to show error propagation, we do not report

operands of the instruction if they are updated after the instruction. Figure 3.3 (b) shows the metadata maintained with each temporary.

3.5.2 Metadata for FP Value Stored in Memory

For FP values in memory, we store metadata in shadow memory. For every FP temporary stored in memory, we store metadata associated with the temporary from stack to shadow memory. The metadata in shadow memory contains the original FP computed value, high-precision value, lock-and-key metadata, instruction identifier, and a pointer to the temporary's metadata in the stack. We store lock-and-key metadata to avoid accessing temporary metadata in a stack if the stack frame is deallocated. Storing high-precision values in shadow memory lets us detect numerical errors when a stack frame is unavailable. Hence, we cannot generate the backward slice of instructions when the stack frame is deallocated, but we detect numerical errors.

Figure 3.3 (a) shows the metadata in shadow memory for each FP value in memory.

3.6 Metadata Propagation

This section describes how we create and propagate metadata for variables in registers and memory.

3.6.1 Creation of FP Constants

For each FP constant in the function, we allocate the metadata space on the stack using the alloca instruction to create the metadata entry.

```
float t5 = a * 4.0f;
```

The above instruction in a C program will be compiled to LLVM IR as shown below.

```
t5 = fmul float a, 4.0;
```

We create metadata space in the program stack for FP constants using alloca instruction. We set the high-precision value as the original computed value, lock-and-key is set to the function's lock and key, and operands are set to null. The instruction identifier is set as the static instruction identifier. We set the timestamp to a monotonically increasing global variable. Finally, we store the mapping of the instruction and metadata entry to the compile-time instruction map (inst_map) in our LLVM pass. The inst_map is a std::map of type ⟨*Instruction**, *Instruction** ⟩. Since in LLVM IR, every variable is assigned only once (SSA), we use the instruction as the key to the map. In our metadata design, we store metadata for only the last writer of a variable. Hence, using the compile-time map, we retrieve metadata for high-precision computation if this instruction is used as an operand with some other instruction. In the below code snippet, *real* is an abstract type representing high-precision value, and the inst_id function returns a unique identifier of the instruction provided in our compiler pass.

```
t5 = fmul float a, 4.0;
```

```
t5_tmd = alloca %struct.fpsan_tmd
t5_tmd->real = real(4.0f);
t5_tmd->lock = func_lock;
t5_tmd->key = func_key;
t5_tmd->op1 = NULL;
t5_tmd->op2 = NULL;
t5_tmd->computed = t5;
t5_tmd->inst_id = inst_id(t5);
t5_tmd->ts = ts++;
inst_map.insert(4.0, t5_tmd);
```

If the FP value is copied from another temporary, we copy the metadata except for the timestamp.

3.6.2 Metadata for FP Binary Operations

For FP binary instructions, we retrieve the high-precision value associated with the operands, perform the same computation in high-precision, and update the real value with the high-precision result. The metadata entry is updated with the function's lock and key, and operands are updated with the pointer to the metadata entry associated with the operands.

```
float t6 = t5 * t1;
```

The above instruction in a C program will be compiled to LLVM IR as shown below.

```
t6 = fmul float t5, t1;
```

```
t5_tmd = inst_map.at(t5);
t1_tmd = inst_map.at(t1);
t6_tmd = alloca %struct.fpsan_tmd
t6_tmd->lock = func_lock;
t6_tmd->key = func_key;
t6_tmd->real = real_mul(t5_tmd->real, t1_tmd->real);
t6_tmd->op1 = t5_tmd;
t6_tmd->op2 = t1_tmd;
t6_tmd->computed = t6;
t6_tmd->inst_id = inst_id(t6);
t6_tmd->ts = ts++;
inst_map.insert(t6, t6_tmd);
```

3.6.3 Metadata Propagation for Memory Store

When the FP value is stored in the memory, we update the metadata in shadow memory from the shadow stack. We use the memory address value stored as the key to shadow

memory. We copy the real value, lock, and key associated with the temporary in shadow memory. We store the pointer to the temporary's metadata in the shadow stack.

```
*res = t8
```

It would be translated into LLVM IR as below.

```
store float %t8, float* %res
```

```
t8_tmd = inst_map.at(t8);
shadow_mem(res)->real = t8_tmd->real;
shadow_mem(res)->computed = t8;
shadow_mem(res)->lock = t8_tmd->lock;
shadow_mem(res)->key = t8_tmd-> key;
shadow_mem(res)->tmd = t8_tmd;
```

3.6.4 Metadata Propagation for Memory Load

To perform selective shadow execution, we check if metadata stored in shadow memory is consistent with the oracle on every load instruction. We perform this check by comparing the value read from the memory by the original program and the value stored in metadata in the shadow memory as described in 3.3. If they do not match, we reset the temporary's metadata similar to the assignment of a constant. If they match, we check if the temporary that has previously written to that location is still valid by checking the lock and 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.

```
float t1 = *a;
```

It would be translated to load instruction in the LLVM IR as shown below.

```
t1 = load float, float* %a
```

```
t1_tmd = alloca %struct.fpsan_tmd
t1_tmd->lock = func_lock;
t1_tmd->key = func_key;
if(t1 == shadow_mem(a)->computed){
  t1_tmd->real = shadow_mem(a)->real;
  t1_tmd->computed = shadow_mem(a)->computed;
  lock = shadow_mem(a)->lock;

  if(*lock == shadow_mem(a)->key){
    t1_tmd->op1 = shadow_mem(a)->tmd->op1;
    t1_tmd->op2 = shadow_mem(a)->tmd->op2;
  }
  else {
   t1_tmd->op1 = NULL;
   t2_tmd->op2 = NULL;
 }
}
else{
 t1_tmd->real = t1;
 t1_tmd->computed =  t1;
 t1_tmd->op1 = NULL;
 t2_tmd->op2 = NULL;
}
```

3.6.5 Metadata Propagation for Function Arguments and Function Return

We store metadata in the shadow stack for function arguments called by value and return values. For every call instruction in LLVM IR, we push the metadata in the shadow stack in the same order as passed in the original function call, as shown below.

```
t1_tmd = inst_map.at(t1);
t2_tmd = inst_map.at(t2);
push_shadow_stack(t1_tmd);
push_shadow_stack(t2_tmd);
```

```
%call = call i32 @foo(float %t1, float %t2), !dbg !34
```

We retrieve the arguments from the shadow stack and copy them to the program stack in the function body.

```
define dso_local i32 @foo(float %x, float %y) #0 !dbg !7 {

x_tmd = pull_shadow_stack();
inst_map.insert(x, x_tmd);
y_tmd = pull_shadow_stack();
inst_map.insert(y, y_tmd);

}
```

Currently, we do not support variadic functions and multiple return values (*e.g.*, values returned as a pair in LLVM IR). However, similar logic can be extended to support such functions and return values.

Figure 3.6 illustrates the metadata before and after the memory store (*i.e.*, instruction 4). Since a constant value is stored in memory, in shadow memory, we store real value as computed value, computed value, and a pointer to operands metadata as null. In instruction 5, a

function RootCount is called with two FP values passed as values. For function arguments, we store the metadata in the shadow stack in the same order as arguments are passed. In the function body of RootCount (instruction 7), we retrieve the function arguments from the shadow stack and save them in the function's stack. On a load instruction (instruction 10), we first check if the metadata state is valid by checking if the loaded value and the stored computed value match. Since they match, we copy the metadata from shadow memory to the program's stack.

3.6.6 Detection and Debugging of Errors

To detect FP errors, a high-precision value is converted to a double value and compared with the original computed double value. If the difference between these two values reaches a user-defined threshold, the numerical error is reported to the user. Similar checks are performed for all FP instructions or user-defined instructions. We also report branch divergence, incorrect float-to-int conversions, and FP exceptions. To enable debugging such errors once they are detected, FPSanitizer produces a DAG of instructions in the set of active functions showing the instruction identifier, and rounding errors occur compared to high-precision values. To generate the DAG of instructions, the FPSanitizer accesses the metadata of instruction that has been reported with an error. FPSanitizer runtime then accesses the operands' metadata by traversing the pointers stored in the instructions metadata. FPSanitizer traverses the operand's metadata if it is not a dangling pointer and its timestamp is lower than the instructions' timestamp. Hence, FPSanitizer generates the DAG of instructions in the active stack frame to diagnose the root cause of the error. If a certain node has an error, but no operands are shown due to the deallocation of the stack frame, then the user needs to put the breakpoint on that node to debug the error in the node's stack frame.

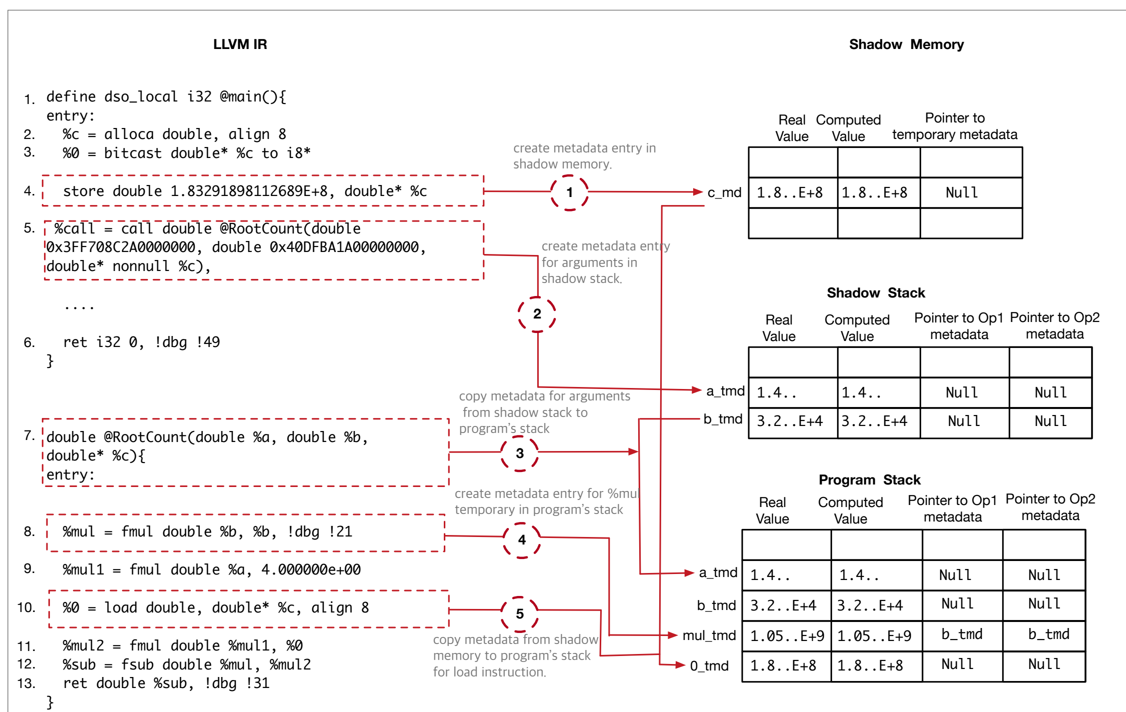

Figure 3.6: This figure shows the original program, LLVM IR, and metadata state for each LLVM IR instruction in a shadow memory, shadow stack, and program stack. In step 1, for store instruction, we store the real value, computed value, and pointer to the temporary's metadata in shadow memory. We do not show instruction-identifier and lock-and-key for simplicity. In-store instruction, a constant value is stored, and hence the pointer to temporary metadata is set to null. In step 2, for call instruction, we push the metadata for function arguments in the shadow stack. In step 3, in the function body, we retrieve the metadata for function arguments from the shadow stack to the program's stack. In step 4, for fmul instruction, we computed the multiplication in high-precision and stored the real value, computed value, and pointers to operands metadata. In step 5, for load instruction, we load the metadata from shadow memory to the program's stack. To detect errors, we compare the high-precision value with the computed value and report if the difference is above the user-defined threshold.

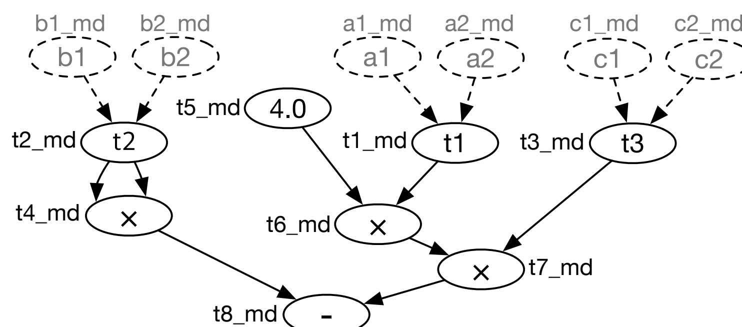

Figure 3.7: DAG generated for the catastrophic cancellation with the operation: `t8 = t4 - t7` in Figure 3.6.

3.7 Running Example

Figure 3.6 shows the program to compute the number of roots for a quadratic equation. To find the roots of the quadratic equation, we compute the determinant of the equation, and if it is greater than 0, then we have two distinct real roots. If $b^2 - 4ac > 0$, then there are two real distinct roots. If the determinant is equal to 0, we have one real root. That is, if $b^2 - 4ac == 0$, then, one real root. If the determinant is less than zero, we have no real root, $b^2 - 4ac < 0$, then no real root. The C program with float precision returns a number of roots as one. However, with double precision, the number of roots returns as two.

In this example, for input *a = 1.8309067625725952E+16, b = 3.24664295424E+12, and c = 1.43923904E+8* , the expression $b^2 - 4ac$ experiences catastrophic cancellation, and, hence, the number of roots returned for the given input is incorrect. The result of computation b^2 experiences a rounding error due to loss of precision. The computation $4 \times a \times c$ also experiences a small rounding error due to loss of precision. Also $4 \times a \times c$ results in a very close value to b^2. The most significant digits are canceled and result in 0 during subtraction of these expressions in comparison to 42.2 in high-precision. Since the result of the subtraction is equal to 0, the number of roots is returned as 1. Figure 3.6 shows the state of metadata in a shadow memory, shadow stack, and the program's stack. Using our approach, we detected the high rounding in line 18, and we diagnosed that the root cause of the error is the rounding error in line 15 due to loss of precision. This rounding error is magnified in line 18 due to catastrophic cancellation. Figure 3.7 shows the DAG generated for this program using FPSanitizer.

3.8 Implementation Details of Our Prototype

In this section, we describe the implementation details for the FPSanitizer prototype. FPSanitizer instruments the program at compile-time to transform the program to enable shadow execution with high-precision computation. FPSanitizer prototype consists of a

llvm-pass to instrument the program at compile-time. Our LLVM pass adds function calls defined in our runtime library to perform high-precision computation using the MPFR library. Our runtime library maintains the metadata in shadow memory.

3.8.1 Shadow Memory

Our goal is to map every memory address to a shadow memory location. We have explored two different designs to implement shadow memory: a trie-based data structure [81] and a hash-table based shadow memory organization [83, 110]. These designs provide a tradeoff to the user between the performance and accuracy of our error detection framework. A trie-based data structure is implemented similar to a page table. In such an implementation, a set of memory address bits is used to access each trie level. A trie-based implementation provides a mechanism to shadow the entire virtual address space. However, we need to shadow the virtual address space for floating-point variables. Hence, the FPSanitizer prototype shadow memory is implemented as a hash table with a fixed number of entries similar to a directly-accessed cache.

In a hash-table based implementation, the memory address is used as a key to access the hash table. Each entry in the hash-table is the metadata stored associated with the memory address used as a key. If two addresses map to the same shadow memory location (hash-table conflict), FPSanitizer updates the shadow memory with the last writer. We handle this loss of information on conflicts using our technique to perform shadow execution from an arbitrary memory state (Section 3.3). In such cases, we lose the information associated with the first writer. However, if the hash-table size is big enough, then information loss due to collisions can be avoided leading to high-performance overheads.

3.8.2 Shadow Stack

FPSanitizer prototype maps each FP variable in register with high-precision value in program stack. We allocate the space for metadata for variables stored in the registers on the

program stack, and space is reclaimed on function exit. We need to propagate the metadata for function arguments and return values. However, passing the metadata on a program stack would require changing the calling conventions. Hence, we propagate metadata for function arguments (pass by value) and return values on a shadow stack. We allocate the space for the shadow stack before the program starts executing using mmap. To access the stack, we use a global variable that points to the top of the stack. On a function call, we increment the global variable by a number of arguments in a function body and decrement it by the same amount on a function exit. To propagate the metadata for the arguments passed by value in a function call instruction, we store the equivalent high-precision value in the shadow stack. Once the function body starts executing, we retrieve the metadata from the shadow stack for the arguments.

3.8.3 Management of Lock and Key Metadata

We store lock and key metadata for the temporal safety of the pointers. We store pointers to the metadata stored in a stack frame in our metadata. However, accessing such pointers once the stack frame is deallocated can result in a crash or a generation of a wrong DAG. Hence, before accessing such a pointer, we need to check if the pointer is valid. To enable this check, we associate the lock and key with each stack frame. At the beginning of a program, we map a region and call it a set of locks. On a function call (*i.e.*, the stack frame is allocated), we assign one free lock from the set of locks and store the unique identifier at the lock. We associate the address of the lock and the unique identifier with each function. Any pointer that points to the location in the stack inherits the lock and the key of that function. On a function exit, we invalidate the identifier by setting it to zero stored at the lock. Hence, the identifier stored at the lock won't match the identifier inherited by the function for the same lock. Hence, this check enables us to access only safe pointers while generating a DAG of instructions. The size of the lock space is bounded by the number of active functions.

3.8.4 Usage

To use FPSanitizer, the user should compile the program using clang with our LLVM pass. Our runtime library is implemented as a shared library and linked with the program. The users can run FPSanitizer in two modes - error detection mode and debugging mode. An error file is generated in the error detection mode once the program finishes the execution. The error report file contains the number of instances for high-rounding error (above user-defined threshold), instances of branch flips, FP exceptions, and incorrect FP to int conversions. In debugging mode, users can run FPSanitizer with interactive debuggers like gdb. Using FPSanitizer in the debug mode, the developer can insert breakpoints/watchpoints on the exported functions and generate the DAG of instructions for root-cause analysis. However, FPSanitizer has low-performance overheads in error detection mode as we do not store additional information about instructions in the metadata.

3.9 Summary

This chapter proposes an FPSanitizer prototype to detect and debug numerical errors in floating-point applications. FPSanitizer LLVM IR passes instruments to the program at compile time to enable shadow execution with high precision. FPSanitizer uses the LLVM IR type system to only instrument FP instructions. Every FP variable in LLVM IR is shadowed with a high-precision value in shadow memory. In addition, we store extra information about instructions to generate the backtrace of the program for root-cause analysis. Our key contribution is the design of metadata space for different types of variables in LLVM IR and the propagation of metadata with each FP instruction. Our approach stores a constant amount of metadata per instruction in shadow memory to reduce memory overheads.

To further reduce the overheads of shadow execution, our metadata encoding enables the user to start the shadow execution at an arbitrary point during the dynamic execution

of the program. This technique helps the user detect and debug numerical errors for small regions in the program.

To debug numerical errors with each instruction, we store pointers to the metadata entry for operands. Using this information, we access the metadata of operands recursively and generate the directed acyclic graph of instructions. The DAG of instructions shows the backtrace of the program responsible for the error detected. In addition, our debugging support helps the user diagnose the error's root cause.

Our experimental studies found the debugging support useful while implementing and debugging a wide range of floating-point applications. In addition, our prototype FPSANITIZER is an order of magnitude faster than the state-of-the-art as shown in Chapter 6.

CHAPTER 4
PARALLEL SHADOW EXECUTION TO ACCELERATE THE DEBUGGING OF NUMERICAL ERRORS

In Chapter 3 we discussed inlined shadow execution with real numbers to detect numerical errors comprehensively. In shadow execution, every floating-point instruction is inlined with equivalent high-precision computation (*e.g.*, using MPFR library [39]). Numerical errors are reported to the user if the FP and high-precision values differ significantly. Shadow execution with high-precision computation can detect a wide range of numerical errors: rounding errors due to accumulation and catastrophic cancellation, branch diverges, incorrect float-to-int conversions, and FP exception values. However, inline shadow execution is a heavy-weight technique and introduces huge overheads of more than $100\times$ [6, 99]. The main reason for the high-performance overheads is the software emulation of a real number using the MPFR library. The high overheads due to inlined shadow execution with high-precision computation prevents the use of such tools with long-running applications. Our goal is to enable shadow execution tools to debug numerical errors with long-running applications. This chapter proposes a parallel shadow execution technique to debug numerical errors with long-running applications with low-performance overheads.

Our goal is to perform a parallel shadow execution and still provide a fine-grained comparison of high-precision computation with floating-point computation to detect numerical errors. In our approach, we automatically create a new high-precision program that mirrors the original program's floating-point computation. The original program provides the FP computed values to the high-precision program to provide the fine-grained comparison of each FP instruction with high-precision instruction. However, running the original and high-precision programs on different cores won't provide significant speedup compared to inlined shadow execution because the high-precision execution is significantly slower than

the original program. Hence, we need a mechanism to split the high-precision program into various fragments to run in parallel to provide scalable speedups. However, each fragment is dependent on the prior fragment as they are derived from a sequential program. Hence, we need a mechanism to break this dependency by initializing the memory state appropriately for each such parallel fragment of shadow execution.

4.1 High-Level Overview of Our Approach

In our approach, the user specifies parts of the code that needs to be debugged, similar to task-parallel programs as shown in Figure 4.1 (a). Our compiler creates shadow execution tasks based on user input that mirror the original program but execute the FP computation with higher precision in parallel. The shadow execution tasks are sequential since they are created from the sequential program. Hence, they depend on each other and need the memory state from the prior tasks. To execute the shadow tasks in parallel, we need to break the dependencies between these tasks by providing the memory state and input arguments. We also need to ensure that the parallel task follows the same control-flow path as the original program to be useful for debugging.

Our key insight is to use the FP value produced by the original program as the oracle whenever we do not have the high-precision information available. Our approach uses the FP value as an oracle to break the dependency between shadow tasks. Hence, shadow tasks are independent of each other, but they depend on the original program for FP values to reset the state. Since the original program is an order of magnitude faster than the high-precision program, we get scalable speedups, as shown in Figure 4.3. The original program provides the required FP values, memory addresses, and branch conditions to the high-precision shadow tasks via the queue. The original program and the shadow execution task execute in a decoupled fashion and communicate only through the queues. Our compiler instruments the original program to enqueue the required values and instruments the

shadow tasks to dequeue these values. Our compiler also updates the branch instructions in the shadow tasks to branch conditions from the original program.

When the shadow execution task loads a value from a memory location, it needs to identify whether that task previously wrote the memory location. The high-precision value is available in memory when the task has previously been written to that address. Otherwise, it needs to initialize the shadow execution using the FP value from the original program. Our mechanism to perform such a check is similar to selective shadow execution in Chapter 3. We identify such cases using the original computed FP value as the oracle. The shadow execution task stores both the high-precision value and FP value produced by the program in its memory when it performs a store. Then when the shadow execution task performs a load, it checks whether the loaded FP value from memory and the value produced by the program is identical. The FP values from the program and the ones in the memory of the shadow task will mismatch when the shadow task is accessing the memory address for the first time or when the memory address depends on values from prior shadow tasks. In such cases, the shadow task uses the FP value from the program as the oracle and re-initializes its memory. This technique of using the original program's FP value as an oracle allows us to execute shadow execution tasks from an arbitrary state. Furthermore, to enable effective debugging of numerical errors, the shadow execution task also maintains information about the operation that produced the value in memory. This additional information can be used to provide a directed-acyclic-graph (DAG) of instructions responsible for the error, similar to FPSanitizer in Chapter 3.

To get scalable speedups, we need to automatically run these tasks on multiple cores and balance the workload. The execution time of the shadow task depends on the user annotations to create shadow tasks. Hence, shadow tasks could vary in execution time, and we need a mechanism to automatically balance the workload on multiple cores. To achieve this goal, we map a shadow execution task to one of the cores in the system, similar to the work-stealing algorithm, dynamically balancing the load. In our approach, we maintain an

active task list and a team of threads equal to the number of cores in the system. All these threads wait on a task queue and steal a task to execute it till completion. The shadow tasks are embarrassingly parallel as they are not dependent on each other and do not share any data structure. Hence, shadow tasks are executed till completion without any synchronization. The decoupled execution of the original program and shadow tasks with dynamic load balancing provides significant speedups with the increase in the number of cores.

Our prototype, PFPSanitizer, enhances the LLVM compiler to instrument the program and generate shadow execution tasks. PFPSanitizer's runtime creates a team of threads for shadow execution, allocates bounded queues to communicate values for shadow execution tasks, and dynamically assigns shadow execution tasks to the cores to balance the load. The speedup with PFPSanitizer over inlined shadow execution depends on the number of shadow tasks. If the user does not create any task, the entire execution is a single task, and the PFPSanitizer can attain a maximum speedup of $2\times$. However, when the user creates a sufficient number of shadow tasks, PFPSanitizer is approximately $30\times$ faster on average on a machine with 64-cores compared to FPSanitizer, which is the state-of-the-art for debugging FP programs. Our performance results are evaluated in chapter 6.

4.2 How Does Our Compiler Generate Shadow Tasks?

Our goal is to run shadow execution with high-precision in parallel to reduce the performance overheads of inlined shadow execution with long-running applications. Our design goal is to minimize the communication between threads. Hence, we separate the high-precision computation from the original program in our design and create a high-precision program. The original program and the high-precision program can run in parallel. However, we want to provide a fine-grained comparison of each FP instruction with the equivalent high-precision instruction to assist the user in debugging the numerical errors. Hence, the original program provides the original FP values to the high-precision program. In our model, the original program works as a producer, and the shadow program works as a con-

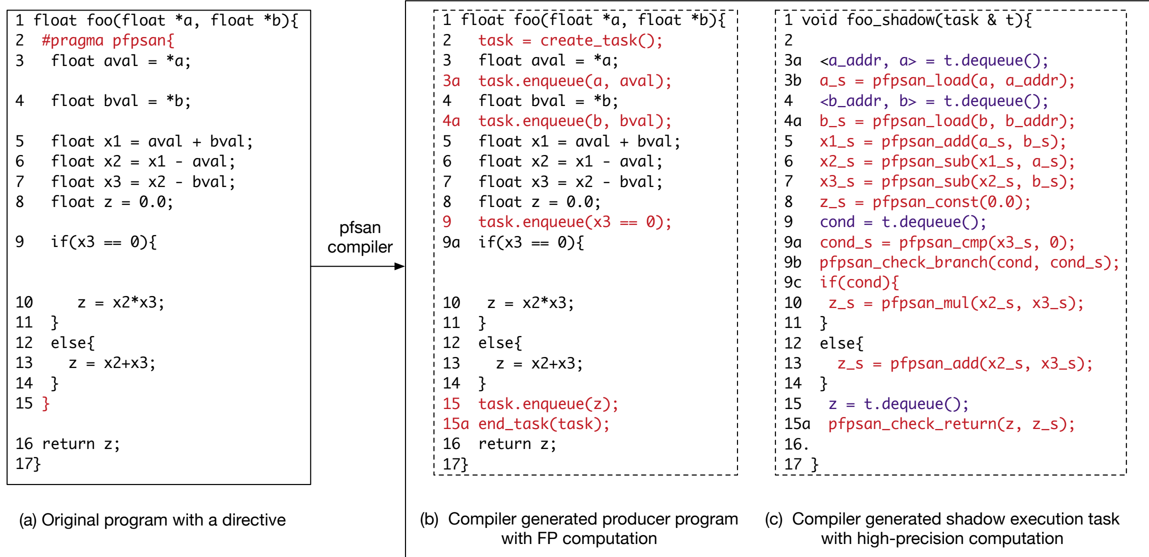

Figure 4.1: Transformations done by the PFPSANITIZER's compiler. (a) Program with `pfpsan` directive. (b) The producer (original program) with additional instrumentation to write FP values and addresses to the queue. The producer passes the address of the memory read and the actual FP value because it enables the shadow execution task to map the address to a shadow memory address. The FP value enables it to check if the shadow task is starting from an arbitrary memory state. (c) The consumer (shadow execution task) that performs high-precision computation. By default, PFPSANITIZER checks error on every branch condition and return value (*i.e.*, `pfpsan_check_branch` and `pfpsan_check_return`)

sumer. Our compiler instruments the original program to push the computed FP values to the queue and instruments the shadow program to read these FP values. Our compiler adds an instruction in the shadow task to compare the FP and high-precision values to detect numerical errors. This model with one producer and one consumer gives us almost 2× speedup compared to inlined shadow execution. However, our goal is to achieve higher speedups with the addition of multiple cores. Hence, we need a mechanism to split the shadow execution into various fragments. To achieve this goal, we take the input from the user. More often, the user has insights into the critical code regions with floating-point computations. Using these insights, the user marks the code regions to debug numerical errors. Based on user input, our compiler creates shadow tasks.

In our approach, the programmer marks parts of the program that need to be debugged with the pfpsan directive (i.e., pragma pfpsan in Figure 4.1 (a)). Each pfpsan directive

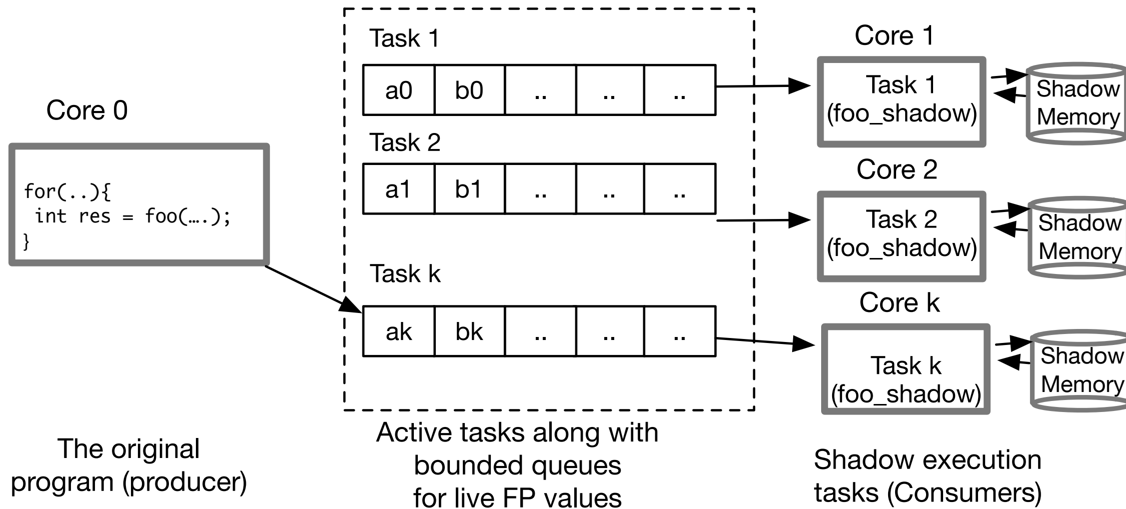

Figure 4.2: Parallel execution of shadow execution tasks during dynamic execution on a multicore machine. The producer (original program) and the consumer communicate live FP values, addresses of memory accesses, and branch outcomes using queues.

represents a scoped block where the programmer suspects the presence of numerical errors and wants to debug them with shadow execution. Our compiler generates a shadow execution task for each such directive. Each such directive corresponds to a single shadow task, which can be executed on another core. Our design does not support nested directives. If the dynamic execution encounters nested directives, the nested directives are ignored, and the shadow execution task corresponds to the outermost directive. If the user marks the loop body as one shadow task, there would be k shadow tasks at run time if the loop body is executed k times. If the user marks the entire program as one shadow task, such as the user places the directive at the beginning of the main method, the entire program will be a single shadow execution task. It can at most get a speedup of $2\times$ over inlined shadow execution. As the programmer introduces more directives, more shadow execution tasks can be executed in parallel. The introduction of additional non-nested directives decreases the window of instructions tracked to debug an error. Numerical errors have a relatively small window of dynamic instructions that are useful to debug and fix the error [18, 99]. Hence, when the programmer uses a sufficient number of directives, the programmer can

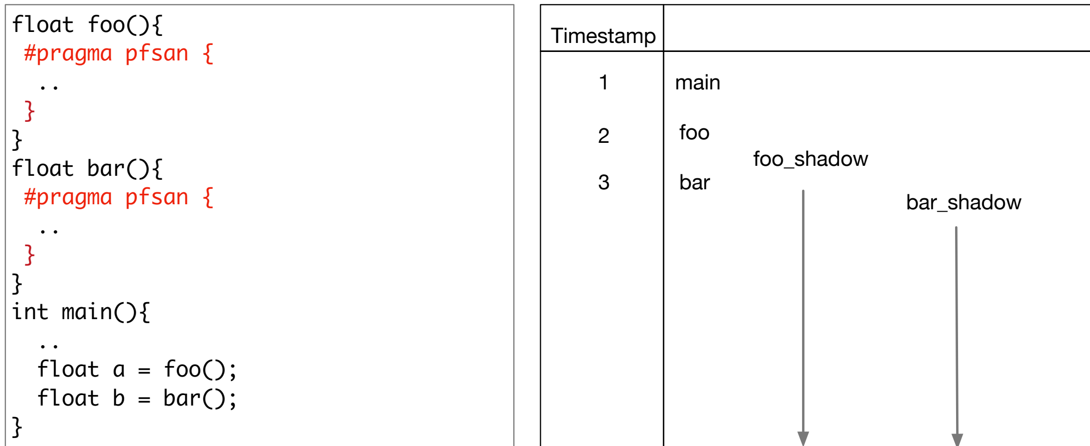

Figure 4.3: In this figure, we show the timestamp execution of the original program. In (a), we show the user annotated program. The user has marked function foo and bar as the shadow tasks. Our compiler creates foo_shadow, a high-precision program that mirrors the function foo, and bar_shadow, a high-precision program that mirrors the function bar in the original program. In our approach, foo_shadow and bar_shadow are independent, but they depend on the original function foo and bar, respectively. In the example, function main starts executing at timestamp 1, then foo starts at timestamp 2, and shadow task foo_shadow starts after timestamp 2. Then, the bar starts executing at timestamp 3, and bar_shadow starts executing after timestamp 3. Since the original program is significantly faster than the high-precision shadow tasks, we achieve executing shadow tasks foo_shadow and bar in parallel.

obtain sufficient speedup and relatively rich DAG of instructions to debug the error using our approach.

We want the original program and the shadow task to execute independently with minimum communication. In our design, there is one producer (original program) and many consumers (shadow tasks), and they communicate through bounded queues. There is no communication or dependency between shadow tasks. Consumers do not stall since the producer is significantly faster than the consumers.

Our shadow tasks perform high-precision computation and have no information about integer operations. Therefore, it is challenging to mirror memory operations and branch instructions in the shadow tasks without integer operations. However, we need to ensure that the shadow task follows the same control-flow path as the original program for effective debugging. We also need to ensure that high-precision computations are propagated through memory as FP values in the original program are. The original program provides the outcome of branch instructions and memory addresses touched by the original program accessing FP values to handle such scenarios. Hence, we instrument the load/store instructions with FP values and branch instruction to enqueue memory addresses and branch outcomes. Similarly, our compiler instruments the shadow execution tasks to dequeue these values. Figure 4.1 (b) and (c) shows the modified original program and the shadow execution task corresponding to the directive.

4.2.1 Modified original program

Our compiler modifies the original program to facilitate communication with shadow tasks. Whenever our compiler encounters the pfpsan pragma, it adds a call to the runtime to obtain a unique task identifier and a queue associated with it. For each FP computation within the scope of the pfpsan directive, our compiler instruments the original program to enqueue the live FP values. Additionally, our compiler instruments the load/store instruction and branch instructions. For each load instruction, our compiler instruments the original pro-

gram to provide the memory address and the value read from that address. Our compiler instruments the store instruction to provide the memory address to the shadow task. For each branch instruction, our compiler enqueues the branch condition to provide information about the branch conditions.

PFPSanitizer's runtime maintains the active task list. At the end of the scoped block corresponding to the directive, the compiler adds a runtime call to enqueue the shadow task in the task list with the task identifier.

4.2.2 Shadow execution task

PFPSanitizer creates a shadow execution task that performs the higher-precision execution based on the pfpsan pragma directive provided by the user. To detect numerical errors, the shadow task compares the high-precision computation with FP computation. Furthermore, to provide support to analyzing the root cause of the error, the shadow task also provides the DAG of instructions showing the error propagation similar to FPSanitizer as shown in Chapter 3.

The PFPSanitizer compiler clones the original function and replaces the FP computations with equivalent high-precision computations to create a shadow task. To improve performance, our compiler deletes all other non-FP instructions except the branch instructions. Furthermore, to propagate the metadata with load/store instruction, our compiler replaces FP load and stores operations with loads and stores of MPFR data type in shadow memory.

For each load instruction in the shadow task, PFPSanitizer's compiler introduces the dequeue operation to read the memory address, and the value read from that memory address. Subsequently, the PFPSanitizer's compiler inserts a runtime call in the shadow execution task to access the shadow memory corresponding to the address of the load instruction. Our compiler copies the metadata from shadow memory to the stack. Additionally, our compiler checks if metadata is valid using the original FP computed value for each load

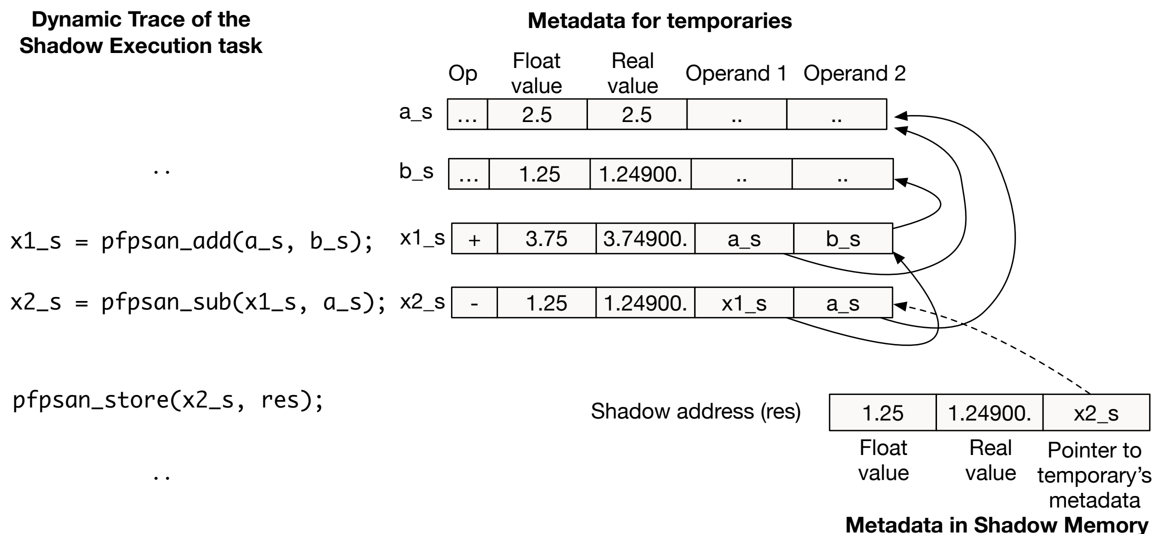

Figure 4.4: Metadata maintained with temporaries and in shadow memory. Every temporary's metadata has the operation (op), float value, real value (MPFR), pointers to operands that produced it, and lock-key metadata for temporal safety similar to Chapter 3, which we do not show here for simplicity. Every FP value in memory has metadata in shadow memory that has the FP value, the real value, and the pointer to the previous writer's metadata. The arrows indicate how a DAG can be constructed using the metadata.

instruction. This check enables us to start shadow execution at an arbitrary point in the program by using the original program as the oracle. For each store instruction, our compiler stores the metadata associated with the FP value being stored in shadow memory.

The shadow execution task does not perform any integer operations. Since the shadow task needs to follow the same control flow path as the original program, PFPSanitizer inserts a dequeue operation from the queue to obtain the branch outcome of the producer. Subsequently, it changes the branch condition in the shadow execution task to branch based on the producer's branch outcome. Figure 4.1 (c) shows the shadow execution task created by the PFPSanitizer compiler for the program in Figure 4.1(a). Overall, PFPSanitizer's compiler generates a high-precision version of the program that executes FP operations with a MPFR type and will follow the exact same control-flow path as the original program during execution.

4.3 Dynamic Execution of Original Program and Shadow Tasks

PFPSanitizer's runtime maintains a list of empty buffers double the size of the number of cores in the processor to facilitate the communication between the original program and the shadow tasks. As the producer executes and encounters the pragma directive, it selects the buffer identifier and begins enqueuing FP values, memory addresses, and branch conditions to the buffer. Once the producer reaches the end of the scoped block corresponding to the directive, it enqueues the shadow task and the buffer identifier associated with it to the active task list.

PFPSanitizer's runtime creates a pool of threads at the start of the producer's execution, which executes the shadow execution tasks. These threads wait on the task list. Any available thread dequeues the shadow task with the buffer identifier from the task list and starts executing the shadow task. PFPSANITIZER's runtime employs a work-stealing algorithm to dispatch a shadow execution task to a thread in the pool. The thread executes a shadow execution task to completion, which is similar to task parallel runtimes [92]. To keep the resource (memory) usage bounded, there are a fixed number of entries in the task queue. If the producer (*i.e.*, the original program) creates more tasks than the size of the task queue, then the producer stalls until there is space in the queue. Also, the original program uses `float` or `double` types that have hardware support, and it is substantially faster than the software MPFR library. Hence, the original program dequeues more tasks than are executed by threads. That means there are sufficient shadow execution tasks for the pool of threads to execute. To minimize contention, the queue used to communicate values from the producer to the shadow execution task uses non-blocking data structures. The use of non-blocking tasks and the work-stealing algorithm ensures dynamic load balancing and provides scalable speedups.

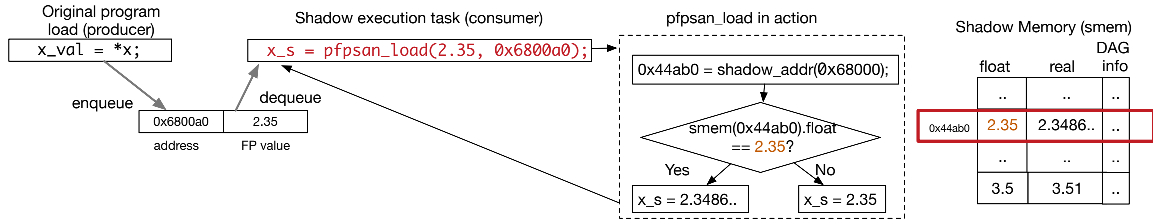

Figure 4.5: Our approach to execute shadow tasks from an arbitrary memory state. PFPSANITIZER maintains the FP value in shadow memory and checks if the program's FP value is exactly equal to the value in shadow memory. If so, it uses the real value for subsequent shadow execution. Otherwise, it uses the program's FP value as the oracle. Here, `pfpsan_load` first maps the producer's address to a shadow address and retrieves the metadata from shadow memory.

4.3.1 Metadata to Detect and Debug Errors

Our compiler generates the shadow task in high precision to detect and debug numerical errors. At compile-time, FP variables are either resident in registers or in memory. We need to store additional information for each FP variable to facilitate error detection and debugging. Similar to FPSanitizer in Chapter 3, for each FP variable in the register, we store metadata in the program stack. For each FP variable stored in memory, we store metadata in shadow memory.

We maintain the real value (*i.e.*, the MPFR data type) with each temporary to detect errors in the FP program compared to an oracle execution with real numbers (i.e., the MPFR data type). Figure 4.4 shows the metadata maintained with each temporary. For each FP computation in the original program, the shadow task reads the FP value from the queue provided by the original program and compares it with the high-precision computation. The metadata for temporaries also maintains information about the operation and the pointers to the metadata of the instruction's operands. For each detected rounding error, PFPSanitizer provides the DAG of instructions. The instruction detected with the high-rounded error is the root of the DAG, and the next level shows the operands of this instruction. Figure 4.4 also illustrates the construction of the DAG using the metadata in shadow memory and for the temporaries, which is similar to our design in FPSanitizer 3.

To enable the execution of shadow tasks from an arbitrary memory state, we also maintain the computed value in the metadata. Using the computed value in metadata, PFPSanitizer performs the check to identify if metadata in shadow memory is valid or not. If it is not valid, then metadata is reset to the original computed value provided by the original program through a queue.

For every memory location, we maintain the real value and the pointer to the metadata of the temporary that was previously written to that memory location. For each store instruction, the metadata associated with the temporary being stored is copied to the shadow memory. On a memory operation that reads a FP value from memory to a variable, the shadow execution task creates a new metadata entry for the variable (i.e., a temporary). It copies the real value from shadow memory to the temporary's metadata. Further, it copies the information about the previous writer and its operands to facilitate the subsequent construction of the DAG. It also performs the check by comparing the computed value stored in the shadow memory and the original FP value provided by the producer to start the shadow execution from an arbitrary state.

4.3.2 Shadow Execution from an Arbitrary Memory State

Since shadow tasks are created from a sequential program, shadow tasks are also sequential, and they depend on the prior task for the memory state. Hence, we need to provide an appropriate memory state for the shadow execution tasks. Our key insight is to use FP values from the original program (the producer) to reset the memory state and start the shadow execution from an arbitrary memory state. Using FP values from the original program as an oracle, we reset the memory state whenever shadow execution lacks information (*i.e.*, either due to an uninstrumented library call or the task is accessing an untracked location for the first time). Hence, shadow tasks are dependent on the original program to reset the state whenever required but not on each other. Furthermore, using our insight, shadow tasks can execute when prior shadow tasks have not been completed. Hence, shadow tasks can run in

parallel on multiple cores as long as the original program has executed the corresponding instructions.

In addition to the live FP values and addresses for the memory accesses, the producer also provides the FP value loaded by the program on every memory read instruction. The shadow execution task maintains the FP value in the metadata for both temporaries and shadow memory locations, as shown in Figure 4.4. For every memory read operation, the shadow task reads the metadata from the shadow memory associated with the memory address. Before reading the metadata, the shadow task performs a check to identify if the metadata is valid or not. The metadata will be valid if written by the same shadow task. However, it would be invalid if it was written by the prior shadow task. If metadata is invalid, it is reset to the original value read from memory. This idea which we call selective shadow execution enables the execution of shadow tasks from an arbitrary memory state. To perform this check, the shadow task retrieves the address of the memory operation and the FP value produced by the producer from the queue. Then, it accesses the shadow memory location corresponding to the address provided by the producer and checks if the FP value in the metadata is exactly equal to the FP value from the producer If they match, PFPSanitizer continues to use the real value in the metadata because the shadow task previously wrote to that location. Otherwise, PFPSanitizer uses the FP value from the producer as the oracle and reinitializes the shadow memory for that memory location with the producer's FP value. If the FP values do not match, then the previous writer to the particular memory location did not update metadata. Such mismatches happen when an update occurs in uninstrumented code, or the update happens in other shadow tasks. Figure 4.5 illustrates our approach to starting shadow execution from an arbitrary memory state with this technique. This idea of starting a shadow task from an arbitrary memory state is similar to selective shadow execution described in Chapter 3.

4.3.3 Detecting and Debugging Errors

The shadow tasks perform computation in high precision and get the equivalent original FP value from the producer. To detect FP errors, PFPSanitizer's runtime converts the MPFR value in the shadow task to a double value and compares it to the double value generated by the producer. If the error exceeds some threshold, it can be reported to the user. Such checks are performed on branch conditions that use FP values, arguments to system calls, return values from functions, and user-specified operations. This fine-grained comparison of the FP program and the high-precision execution enables comprehensive detection of numerical errors. Once the shadow task finishes the execution, it generates the error report with the number of instances found with rounding errors above some user-defined threshold. Using the error report, the user can run the application with gdb and put a breakpoint in the dynamic execution where the error was reported. Once execution reaches the breakpoint, the user can call a function defined PFPSanitizer's runtime to generate the DAG of instructions responsible for the error. Using the DAG of instructions, the user can diagnose the root cause of the error, as illustrated below with an example.

4.4 Illustration of Our Approach

This section describes how PFPSANITIZER helped us detect infinities and NaNs in Cholesky decomposition from the Polybench benchmark suite. We also describe how we have debugged the root cause of the error using PFPSANITIZER debugging support. PFPSanitizer detected infinities and NaNs (*i.e.*, exceptional conditions) at various places in the application. Unfortunately, the program's code or documentation did not explain the exception.

Cholesky decomposition [50] is a widely used algorithm in various domains and problems such as Monte Carlo simulation and Kalman filters. Cholesky takes a $N \times N$ positive-definite matrix A as input and outputs a lower triangular matrix where $L \times L^T = A$, where

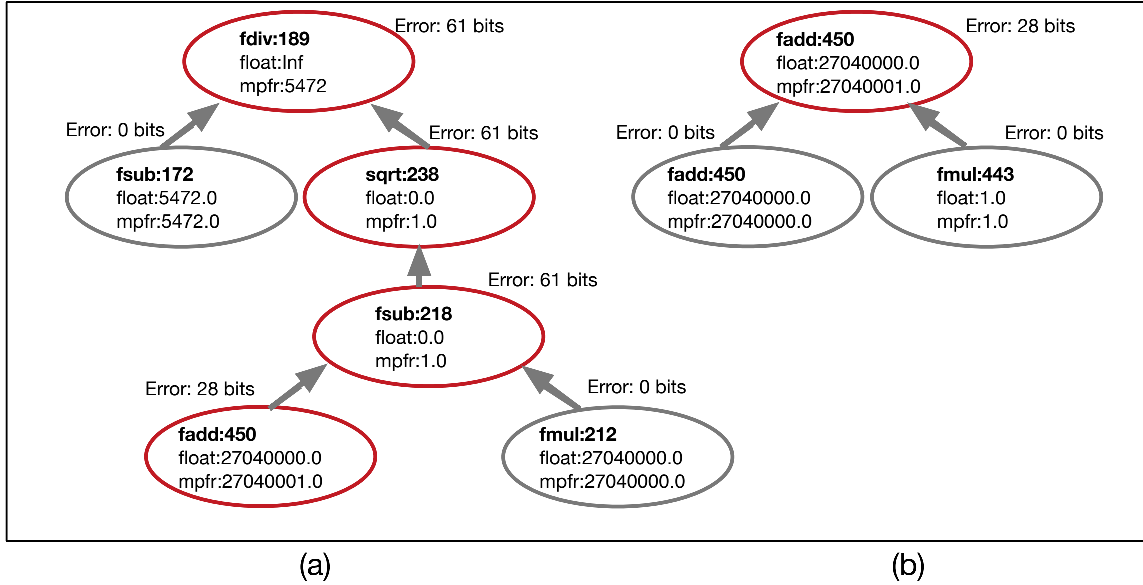

Figure 4.6: A DAG of instructions generated by PFPSANITIZER while debugging the error in Cholesky. Each node shows the opcode, instruction id, computed value, real value and the numerical error occurred. (a) The DAG for the `fdiv` instruction that results in infinities (inf). (b) The DAG for the `fadd` instruction that is the root cause of the error.

L^T is the transpose of L. Lower triangular matrix L is computed as shown below, where i and j represent matrix indices.

$$L_{i,j} = \begin{cases} i = j: & \sqrt{A(i,i) - \sum_{k=0}^{i-1} L(i,k)^2} \\ i > j: & \frac{A(i,j) - \sum_{k=0}^{j-1} L(i,k)L(k,j)}{L(j,j)} \end{cases} \tag{4.1}$$

It can be observed that the computation can produce infinities (and NaNs when infinities get propagated) when $L(j,j)$ evaluates to zero, which happens when the matrix A is not positive semi-definite. To make the matrix positive semi-definite, Cholesky in Polybench computes $A = A \times A^T$. When this computation is performed with reals, the resulting matrix A is positive semi-definite for all inputs.

We generated inputs to this application using an input generator and ran the application with PFPSanitizer using those inputs. Specifically, when we generated the input matrix.

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 5200.0 & 1.0 & 0.0 \\ 0.0 & 5472.0 & 1.0 \end{bmatrix} \tag{4.2}$$

PFPSanitizer detected NaNs and infinities in the program. Next, we describe the process we used to debug this error.

When the matrix A is adjusted to make it positive semi-definite (*i.e.*, $A \times A^T$), the resultant matrix A in real numbers is

$$\begin{bmatrix} 1.0 & 5200.0 & 0.0 \\ 5200.0 & \mathbf{27040001.0} & 5472.0 \\ 0.0 & 5472.0 & 29942785.0 \end{bmatrix} \tag{4.3}$$

Using PFPSANITIZER, we observed that the program computes the following matrix.

$$\begin{bmatrix} 1.0 & 5200.0 & 0.0 \\ 5200.0 & \mathbf{27040000.0} & 5472.0 \\ 0.0 & 5472.0 & 29942785.0 \end{bmatrix} \tag{4.4}$$

Specifically, when computed with real numbers, $A[1][1]$ cannot be exactly represented in a 32-bit float. Hence, it is rounded to 27040000. PFPSANITIZER identified that the computation of $A[1][1]$ in the lower triangular matrix differs from the oracle execution. Specifically, $A[1][1]$ is computed as $A[1][1] - (A[1][0] * A[1][0])$. The FP program produces a 0, whereas the oracle execution with real arithmetic produces 1. Subsequent division operation results in infinities for the 32-bit float version.

We used the `gdb` debugger to insert a conditional breakpoint in the PFPSanitizer's runtime when the program produces an infinity or a NaN in the result of any operation. We

observed that the breakpoint was triggered with a `fdiv` instruction. We generated the DAG in the debugger. Figure 4.6 provides the DAG, where each node provides the instruction (instruction opcode:instruction id) and a number of bits of error with it. Figure 4.6(a) shows that error occurs in `fadd:450` and is amplified by `fsub:218`. To identify why `fadd:450` has any error, we set a breakpoint on the `fadd` instruction if the error is greater than or equal to 28 bits. Figure 4.6(b) shows the DAG generated by PFPSanitizer. The real execution with the MPFR type computed 27040001 while the FP computation produced 27040000. The value 27040001 cannot be exactly represented in a 32-bit float, and it is rounded to 27040000. We reported this bug to the maintainers of the PolyBench suite. They have acknowledged the error. For performance reasons, all kernels in the PolyBench suite avoid such checks. They delegate the responsibility of checking invalid inputs to the user. Our experience demonstrates that PFPSanitizer will be useful in debugging errors that result from such implicit preconditions.

4.5 Implementation Considerations

PFPSANITIZER enhances the LLVM compiler to add instrumentation to the original program and create shadow execution tasks. PFPSANITIZER's runtime is in C++, linked with the binary when the program is compiled. Specifically, the runtime manages task creation, management of queues associated with tasks, creation of worker threads, and the implementation of the work-stealing algorithm to dynamically balance the load among the threads. Although the producer creates numerous shadow tasks, the number of threads created by the runtime equals the number of cores in the system to avoid unnecessary context switches. We describe important implementation decisions in building the PFPSanitizer prototype. The design and implementation of metadata space for variables in registers and in memory are similar to FPSanitizer's approach discussed in Chapter 3.

4.5.1 Shadow Memory Organization

A shadow execution task accesses shadow memory, which maps each memory address with an FP value to its corresponding real value. Each worker thread has its own shadow memory, which is completely isolated from the shadow memory of other threads. To bound the memory usage, PFPSanitizer uses a fixed-size shadow memory for each thread that is organized as a best-effort hash table (similar to a direct-mapped cache). In a conflict, when two addresses map to the same shadow memory location, PFPSanitizer overwrites the shadow memory location with the information about the latest writer. We handle this loss of information on conflicts using our technique to perform shadow execution from an arbitrary memory state.

4.5.2 Management of Temporary Metadata Space

PFPSANITIZER maintains metadata with each temporary in the LLVM intermediate representation. PFPSANITIZER allocates metadata space for temporaries in the program stack at function entry. Once the function exits, metadata space for temporaries is reclaimed automatically. In metadata, we store a pointer to the operands metadata. Once the stack frame is deallocated, then the metadata for some temporaries might store a dangling pointer. Accessing such a pointer while generating the DAG of instructions would crash. Hence, PFPSanitizer's runtime also checks the validity of the temporary metadata pointer in shadow memory before dereferencing it, which is similar to FPSanitizer's temporal safety checking described in Chapter 3. Rather than maintaining a unique metadata entry for each dynamic instruction, PFPSanitizer maintains a unique entry for each static instruction. As a result, PFPSanitizer produces DAGs restricted to the last iteration in a program with loops similar to FPSanitizer.

4.5.3 Handling Indirect Function Calls

PFPSanitizer's compiler creates a high-precision version for each function in the program. PFPSanitizer's runtime maintains the mapping between the address of the original function and the address of the corresponding shadow function. PFPSanitizer's compiler replaces all direct function calls with corresponding shadow functions in the shadow execution task. However, for indirect function calls, we do not know the equivalent shadow function at compile time.

To handle indirect functions (i.e., calls through a function pointer), for every indirect function call in the original program, PFPSANITIZER introduces the dequeue operation to provide the function address to the shadow task. Then, at the equivalent location in the shadow task PFPSANITIZER compiler introduces a call to the runtime that uses the address of the original function provided by the producer on the queue and calls the corresponding shadow function using the mapping maintained by the runtime.

4.5.4 Support for Multithreaded Applications

Although we describe our approach as assuming a single-threaded program, our approach will work seamlessly with multithreaded applications. As PFPSanitizer treats the FP value produced by the program as the oracle, it can detect errors even in programs with races. However, it will not detect errors specifically due to data races. One challenge with multithreaded applications is the allocation of cores to the original program and the shadow execution tasks. Parallel shadow execution with PFPSanitizer will be beneficial compared to inlined shadow execution when at least one core is unused by the original multithreaded application.

4.5.5 Usage with Interactive Debuggers

PFPSanitizer supports debugging with interactive debuggers like gdb. We propagate debugging symbols from the original program to the shadow execution task to enable such

debugging. We export functions defined in the PFPSANITIZER's runtime for the user to call to enable debugging numerical errors with gdb. Hence, the developer can insert breakpoints/watchpoints on functions in the shadow execution task. We support two modes in PFPSANITIZER- error detection and error debugging. In detection mode, PFPSANITIZER generates the error report once the execution finishes. In detection mode, we do not store the pointer to operands metadata to enable debugging of numerical errors. Hence, the error detection mode has lower overheads than the debugging mode. In error debugging mode, the user can generate the DAG of instructions by calling a runtime function for root-cause analysis with gdb. The backward slice of the instructions with the DAG and the detection enabled us to find and debug errors with the Cholesky application.

4.6 Summary

In shadow analysis, real numbers are typically simulated with a high-precision software library (*i.e.*, MPFR library). Hence, a software simulation of real numbers is the major reason for high overheads. One way to reduce the overheads is to perform shadow analysis in parallel on multicore machines.

This chapter proposed a novel approach to detecting and debugging numerical errors in long-running applications that perform shadow analysis in parallel. In our model, the user specifies parts of the program that need to be debugged. Our compiler creates shadow execution tasks that mirror the original program for these specified regions but performs FP computations with high precision in parallel. Since we are creating shadow tasks from a sequential program, shadow tasks are also sequential depending on prior tasks for memory state. To execute the shadow tasks in parallel, we need to break the dependency between them by providing the appropriate memory state and input arguments. Moreover, to correctly detect the numerical errors in the original program, shadow tasks need to follow the same control flow as in the original program. Our key insight is to use FP values computed by the original program to start the shadow task from some arbitrary memory state. Our

compiler introduces the additional instrumentation in the original program to provide live FP values, branch outcomes, and memory addresses. To ensure shadow tasks follow the same control flow as the original program, our compiler updates every branch instruction in the shadow task to use the branch outcomes of the original program. The original program and shadow tasks execute in a decoupled fashion and communicate via a non-blocking queue. Our shadow tasks do not have any information about integer operations in the original program. Hence, shadow tasks get the memory address from the queue and map it to the shadow memory location with a high-precision value. On every memory load in the original program, shadow tasks access the shadow memory and check if it has a valid high-precision value. If this check fails, the shadow task initializes the shadow memory with a computed FP value. Hence, using the FP value from the original program enables us to perform parallel shadow execution from a sequential program. To detect the numerical error, the shadow task compares the high precision value with the actual computed value and reports it to the user if the difference is above some threshold. Similarly, to detect branch flips, the shadow task compares the FP branch outcome in the actual program with the high precision branch outcome in the shadow task. To run shadow tasks in parallel, our runtime maps shadow tasks to one of the available cores using a work-stealing algorithm to get scalable speedups. Once the shadow task reports the error, a directed acyclic graph of instructions is generated to give feedback to the user. Using the DAG of instruction, users can identify the root cause of the error.

Our tool PFPSANITIZER is an order of magnitude faster than the state-of-the-art and comprehensively detects numerical errors within the specified regions. PFPSANITIZER helped us to detect and debug numerical errors in the Cholesky benchmark from the Polybench suite. Although we have shown our approach for sequential programs, our technique seamlessly works for parallel programs. Also, this technique can be applied to a wide variety of shadow analysis such as data race detection, memory safety analysis, and taint analysis.

In our approach, the user annotates the program to help the compiler to generate the shadow tasks. However, often the user does not have insights to mark the code regions for debugging. In such a scenario user can mark the entire program as one shadow task, giving almost $2\times$ speedup over FPSanitizer. An alternative approach could automatically generate the shadow tasks without any user input. In this work, we do not tackle the problem of identifying the code regions automatically for numerical error debugging, and we leave it for future work.

CHAPTER 5
A LIGHTWEIGHT ORACLE USING ERROR-FREE-TRANSFORMATIONS FOR SHADOW EXECUTION

In Chapter 4 we discussed a novel approach to run shadow execution in parallel to reduce overheads with inlined shadow execution. Our tool PFPSANITIZER, based on this idea, helped us debug numerical errors with long-running applications. In this approach, the user annotates the source code using the pragma directive, and our compiler generates shadow tasks for these code regions. These shadow tasks are run in parallel on multiple cores. This approach provided scalable speedups and reduced overheads with inline shadow execution. However, there are a few caveats with this approach that we will discuss next. First, the user often does not know which code regions to mark for debugging. In such scenarios, the user can mark an entire program for shadow execution, which would be similar to running inline shadow execution. Second, this approach limits the detection of errors to the region defined by the user. For example, if the numerical error propagates between two shadow tasks, this approach would miss such errors.

This chapter proposes an alternative mechanism to reduce the performance overheads with inlined shadow execution while detecting and debugging numerical errors. Our studies found that the major reason for high overheads with inlined shadow execution is due to the use of software simulated high-precision oracle. In our approach, we use an oracle designed with floating-point representation instead of software simulated high-precision computation. To design such an oracle, we exploit the properties of the floating-point representation. A key insight about the FP representation is that the rounding error of a primitive operation can be represented in the FP representation itself [78]. Furthermore, 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) [86]. We have designed a shadow execution framework called as EFTSANITIZER with EFTs as an oracle.

Using hardware-supported FP operations to compute the error makes EFTSANITIZER's shadow execution significantly faster than the inlined shadow execution framework, FPSANITIZER. Our prototype, EFTSANITIZER, is usable with long-running applications because it is an order of magnitude faster than FPSANITIZER. EFTs have been previously used to extend the precision for geometric algorithms [100], to create libraries for encapsulating error with Shaman [33], and to generate compensation code. With EFTSANITIZER, we advocate using error-free transformations as an oracle for shadow execution.

To facilitate effective debugging with long-running applications, EFTSANITIZER allows the user to perform shadow execution from an arbitrary point in the dynamic execution (selective shadow execution), similar to FPSANITIZER. It is appealing for scientific simulations that execute for days. Also, EFTSANITIZER provides a directed acyclic graph of instructions that spans multiple functions (many of which may have already been completed) and multiple iterations of the loop while keeping the memory usage bounded in contrast to FPSANITIZER, which provides DAGs only when the instructions in the DAG belong to functions in the current calling context and only for the last iteration in the presence of loops.

EFTSANITIZER is a compiler-instrumentation framework that instruments every FP variable in memory and registers to track additional information, which we call metadata. To detect errors, it is sufficient to propagate the rounding error computed with 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 the last k dynamic instructions at the point of a numerical error (see Section 5.2.3 for details on the design of the metadata). These instructions can span functions that have already completed execution and various iterations of a loop. We found

these DAGs useful for debugging new numerical errors discovered by EFTSANITIZER and validating existing bugs.

Our prototype of EFTSANITIZER 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 14.72× faster than FPSANITIZER, which is the state-of-the-art for shadow execution. Our experiment results are shown in Chapter 6.

5.1 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 [78]. A sequence of FP operations to compute the rounding error of a primitive operation is called error-free transformations (EFTs) [86]. 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 a primitive operation with real numbers. Although the error of 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 they provide more precision than double-double arithmetic for some computations, even when we maintain a single error term. For example, the expression $((1.0 + 1.7E + 308) - 1.7E + 308)$ will return 0 with double-precision arithmetic. The addition $(1.0+1.7E+308)$ returns $1.7E+308$ 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 precisely store the result of the addition. By explicitly maintaining error with EFTs, we can easily capture this error. In essence, we can store the result of $(1 + 1.7E + 308)$ as the sum of two floating-point

```
1 Function TwoSum(a, b):
2     x ← a + b
3     b′ ← x − a
4     a′ ← x − b′
5     δ_a ← a − a′
6     δ_b ← b − b′
7     δ_x ← δ_a + δ_b
8     return (x, δ_x)
```

```
1 Function PropSumError((a, δ_a),
  (b, δ_b)):
2     (x, δ_x) ← TwoSum(a, b);
3     δ_x ← δ_x + δ_a + δ_b;
4     return (x, δ_x)
```

Figure 5.1: 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).

numbers using EFTs. Hence, EFTs provide a mechanism to split the floating-point numbers as the sum of two non-overlapping floating-point numbers [56].

Next, we describe the 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 $+$ with real numbers. Otherwise, all operations are performed with floating-point arithmetic operations.

5.1.1 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 the nearest mode was proposed by Donald Knuth [78]. It was called TwoSum by Shewchuk [100]. Figure 5.1 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 [93]. If $|a| \geq |b|$, then the subtraction in line 2 of TwoSum in Figure 5.1 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 5.1. Other subtraction operations in the TwoSum algorithm in Figure 5.1 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 [10]. However, those cases rarely appear in practice [78]. Such error-free transformations have also been explored to produce the rounding error for other rounding modes in the IEEE standard [90].

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 [78]. 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.

5.1.2 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 due to 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 5.1 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 such error [100]. 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 a double-double arithmetic [51].

5.1.3 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.

5.1.4 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. Recent hardware has support for fused-multiply-add operations. 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, the FP multiplication result in 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, the error δ_x is below the underflow threshold. Hence, it is not exactly representable as an FP number [77].

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 er-

ror [78]. We use the the fused-multiply-add operation to compute the rounding error with a single multiplication operation.

5.1.5 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 $(a +_R \delta_a) * (b +_R \delta_b)$. Hence,

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

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.

5.1.6 Computing and Propagating the Rounding Error of the FP Division Operation

Similar to multiplication, computing the rounding error for the FP division operation can be 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,

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

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

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

5.1.7 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 [9, 78].

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 the operation, we also handle the case where $x = 0$ separately to avoid divide-by-zero exceptions in the computation of the error.

5.2 The EFTSanitizer Approach

Our goal is to develop a shadow execution framework with EFTs for detecting and debugging numerical errors in long-running programs during the late stages of testing. We need the resulting approach to have the following attributes to accomplish this goal. First, it should detect errors comprehensively, such as exceptions (due to NaNs and infinities), cancellations where a large fraction (or all) of the bits are wrong, slow convergences, and significant rounding errors. Second, it should enable the debugging of reported errors with an execution trace that illustrates the propagation of errors.

5.2.1 Error Free Transformations for Shadow Execution

Our prototype EFTSANITIZER performs inlined 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 FPSANITIZER and prior shadow execution tools [6, 18, 99].

EFTSANITIZER is a compiler instrumentation framework that automatically adds code after each FP operation to compute and propagate the error with 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.*, some elementary functions), we use high-precision operations (*e.g.*, MPFR library) to compute the error.

5.2.2 Debug Information to Illustrate the Propagation of Rounding Errors

Accumulation of rounding errors with a sequence of operations is the root cause of numerical errors. Hence, EFTSANITIZER provides a dynamic trace of instructions represented as a directed acyclic graph that demonstrates the propagation of errors to help the user debug the error. Prior research in improving shadow execution such as FPSANITIZER [18] and Herbgrind [99] 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. 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 calling context) 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 5.2 com-

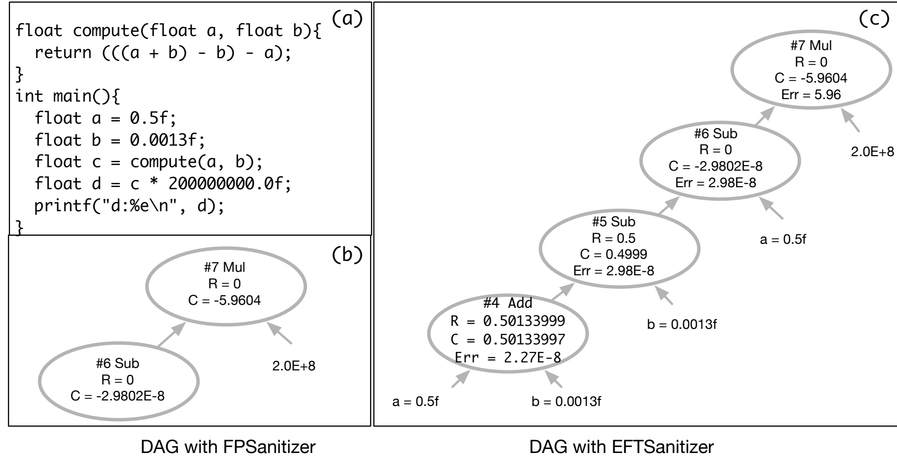

Figure 5.2: (a) A sample program to illustrate differences in the DAGs generated by PFPSANITIZER, which is proposed in this paper, and prior work FPSanitizer [18] for the variable d used in the print statement. (b) The DAG generated by FPSanitizer where the DAG information is lost after the function call completes. (c) The DAG generated by PFPSANITIZER. Each node in DAG reports the operation, real value (R), computed value (C), and the error for that node. The real value is computed as the sum of the error and the computed value with FP arithmetic in PFPSANITIZER. The real value is computed using the high-precision MPFR library in FPSanitizer.

pares the DAGs generated by EFTSANITIZER and FPSanitizer for a sample program for illustration.

Our tool EFTSANITIZER supports selective shadow execution similar to FPSANITIZER to support testing and debugging of numerical errors in specific parts of the application rather than the entire execution.

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 comprehensively detect errors with long-running applications.

5.2.3 Metadata Design and Organization of the Metadata Space

Given that EFTSANITIZER performs compiler-based instrumentation, the FP values are resident either in memory locations or in registers/temporaries. Therefore, we need to

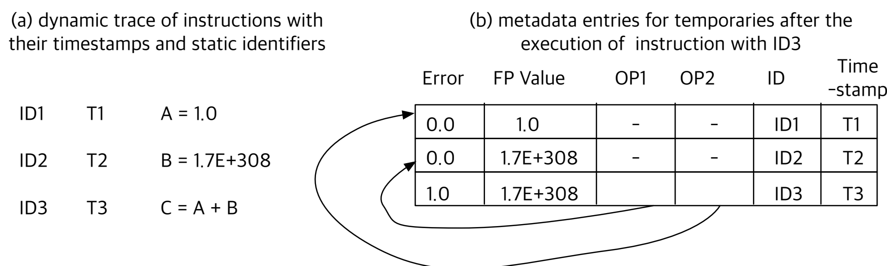

Figure 5.3: The metadata maintained with each temporary and each memory location. In (a) we show the dynamic trace of the executed instructions with the timestamp of the execution and the static identifier of the instruction. In (b) we show the temporary metadata entries after the execution of instruction with ID3 (*i.e.*, C = A+ B). Here, we show pointers in the metadata space with arrows from the field to their corresponding metadata entries.

store the propagated error and additional information with each memory location and temporaries (stack-allocated variables or registers) that hold an FP value to perform inlined shadow execution. However, 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 those in temporaries.

5.2.4 What Should We Store in Each Metadata Entry?

To detect numerical errors, we compute the rounding error using error-free-transformations and store it using double-precision for each FP variable in the program. 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 errors, 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 to debug.

Figure 5.3 illustrates the information maintained with each metadata entry. When EFT-SANITIZER 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 an error propagation by following the pointers to the operands in the metadata space.

5.2.5 Organization of the Metadata Space

We store metadata for FP values in memory in a shadow memory, organized as a hash map similar to FPSANITIZER. The metadata lookup is performed using the address of the memory access. For temporaries with FP values, we use an alternative design space to provide the backward slice of dynamic instructions for debugging numerical errors. Hence, we maintain the metadata in a small circular queue for temporaries with FP values, which we call the temporary metadata space. Given a temporary with 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 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.

5.2.6 Reusing the Entries of the Temporary Metadata Space

Providing the dynamic trace of instructions is useful in debugging when a temporary is being updated or when error propagation spans across multiple functions. However, maintaining the metadata for all the dynamic instructions would make such an approach infeasible with long-running programs. Hence, to keep the memory usage bounded for the temporary metadata space, we use a circular use 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 FPSanitizer, the metadata entries can span function calls that have already completed execution and multiple loop iterations, which significantly helps in the task of debugging numerical errors (see Section 6).

While generating the DAG of instructions, we first access the metadata entry of instruction with a high-rounding error which is the root of the DAG. Next, we access the pointers to the operand's metadata from the root node. Similarly, we recursively traverse pointers to the operand's metadata to show the error propagation. However, if the operand's metadata entry is being reused by some other temporary, we need to stop further traversing. Below we describe our approach to avoid generating incorrect DAGs due to the reuse of metadata space. In our approach, we use a map, which we call the 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 the 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 [82]. Rather than maintaining an 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 last writer runtime map ensures that the size of the runtime map is proportional to the number of static instructions in the program rather than the number of dynamic instructions (number of active memory allocations in the case of CETS).

5.2.7 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.

Metadata Initialization for 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 repre-

sented 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`.

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

```
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>);
```

Metadata Entry Creation for 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 could have been reused. Hence, we check if the timestamp in the last writer runtime map matches the timestamp at the metadata entry. If so, then the

metadata entries are valid. Otherwise, we do not have information about the operands. Therefore, we consider 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 after the FP operations using error free transformations as described in Section 5.1. We use PropSumError in 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.

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

```
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->ts != x_ts) ? NULL: x_meta;
z_meta->op2 = (y_meta->ts != y_ts) ? NULL: y_meta;
x_error = (x_meta->ts != x_ts) ? 0.0: x_meta->error;
y_error = (y_meta->ts != 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 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's type is double* and y's type is double with ID: y_id
*x = y;
```

```
<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.

Handling the Load of an FP Value From Memory

On every load operation that loads an FP value, we read the metadata from the 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 similar to FPSanitizer described in Chapter 3.

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

```
y_meta = allocate_temporary_metadata_space_entry();
shadow_addr = shadow_memory(x);
//check for selective shadow execution
if(shadow_addr->fpvalue == y){
 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 since we do not want to change the calling conventions. 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).

5.2.8 Error Reporting and Debugging Interface

Most FP instructions have some rounding error. They do not always 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 the program produces exceptional conditions such as NaNs and infinities. 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.

5.3 Implementation Considerations

We built a prototype 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 corresponds 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. We handle elementary functions, which is provided by math libraries, using correctly rounded functions from RLIBM [66, 67, 68] for the float type, CR-LIBM [26, 27] for the double type, and glibc's libraries for the double-double type when the corresponding functions are available. We use the MPFR library otherwise [38]. Similarly, we use the MPFR versions of the operation for LLVM's intrinsics.

5.3.1 Shadow Memory, Shadow Stack, and Temporary Metadata Space

We organize the shadow memory in EFTSANITIZER as a best-effort hash map with 64 million entries (*i.e.*, 64* 1024*1024 entries). Further, the shadow memory is allocated with the `mmap` system call, which creates virtual memory mappings without reserving physical memory on Linux. Hence, the program experiences memory overhead only when

the program touches memory. Each metadata entry is 56 bytes, as shown in Figure 5.3. The hash map is indexed by the memory address of the location where the FP value is stored. 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.

5.4 Illustrative Example

To illustrate the metadata propagation, let us consider a simple example where all operations are performed with temporaries. In the comments, we have shown the static identifier for each instruction.

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

```
    C = C + T; // ID5
}
```

Below we have shown the dynamic execution trace of this program. We show the static instruction identifier and the time the instruction was executed. The multiple dynamic instances of the same instruction will have the same identifier.

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

In Figure 5.4, 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. In this example, instructions with identifiers as ID1, ID2, and ID3 are constants. Hence in temporary metadata space, we create an entry for these constants, set the error to 0, and set pointers' to operands' metadata to Null, as shown in Figure 5.4 (A). In the last writer runtime map, we create an entry to map a static instruction to its temporary metadata entry and the timestamp at which it was written.

The next instruction with identifier ID4 and timestamp T4 results in an error due to loss of precision. More precisely, the expression $1 + 1.7E + 308$ is rounded to 1.7E+308 due to loss of precision. Hence, the error for this instruction is 1.0. In temporary metadata space, we create an entry for this instruction. We set the error to 1.0 and pointer to operand1

(A) 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)>

ID	(addr, ts)
ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr3, T3>
ID4	<addr4, T4>
ID5	<addr5, T5>

(B) 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)>

ID	(addr, ts)
ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	**<addr1, T7>**
ID4	<addr4, T4>
ID5	<addr5, T5>

(C) 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)>

ID	(addr, ts)
ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr1, T7>
ID4	**<addr2, T8>**
ID5	<addr5, T5>

(D) 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**	**addr2**	**addr5**	**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)>

ID	(addr, ts)
ID1	<addr1, T1>
ID2	<addr6, T6>
ID3	<addr1, T7>
ID4	<addr2, T8>
ID5	**<addr3, T9>**

Figure 5.4: This figure shows the snapshot of temporary metadata space at various timestamps. We assume that there are only 6 entries in the temporary metadata space for this illustration. We identify these entries using the memory address as(addr$i$). (A) 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. (B) 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. (C) Temporary metadata space after the operation T=A+B at time T8. (D) Temporary metadata space after the operation C= C+T at time T9.

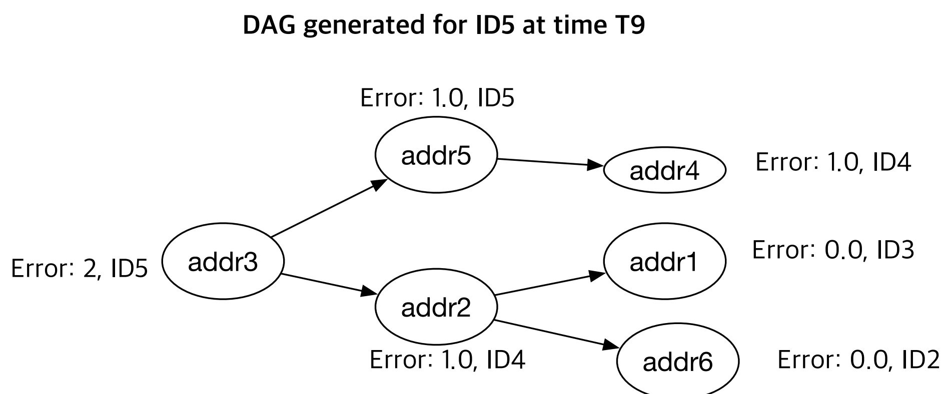

Figure 5.5: The DAG generated for the instruction with ID5 at time T9 Figure 5.4 (D). From the last writer's map, it is mapped to addr3. When we follow the operand nodes while creating the DAG entries, we check if the operand's timestamp is greater than the current instruction's timestamp. If so, we 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.

and operand2 to addr2 and addr3. The next instruction with identifier ID5 and timestamp T5 results in error 1.0. However, the total error at this instruction is 2.0, considering the error in the operand. Then, in the next iteration, the instruction with identifier ID2 and timestamp T6 is executed. Figure 5.4 (A) shows the snapshot of the temporary metadata space at time T6. At this point, all the entries in the temporary metadata space are used. The execution of the next three instructions with timestamps T7, T8, and T9 would result in a reuse of the metadata space. While generating the DAG of instructions, we avoid printing operands node if the operand's timestamp is greater than the current instruction's timestamp, as shown in Figure 5.5.

5.5 Summary

This chapter has described an inlined shadow analysis with a lightweight oracle. Our oracle uses hardware support FP arithmetic to capture the rounding error. We transform FP computations to capture the rounding error accurately. For example, FP addition $a + b$ where $|a| \geq |b|$ is transformed into $x + y$. In this transformation, $x = FP(a + b)$ represents the computed FP addition with round-to-nearest mode and $y = FP(b - (a + b) - a)$ represents the exact rounding error occurred during this computation. Such transformations are based on how rounding errors in FP representation can be accurately stored in an FP data type. These transformations are called as Error Free Transformations (EFTs), and they can extend the accuracy beyond the available hardware primitive type [55]. 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 order of magnitude faster than state-of-the-art, enables selective shadow execution for arbitrary fragments of dynamic execution, and enables effective debugging of numerical errors.

CHAPTER 6
EXPERIMENTAL EVALUATION

In the prior chapters, we have presented different techniques to reduce the overheads of shadow execution to detect and debug numerical errors. Based on these ideas, we have introduced different frameworks: FPSANITIZER, PFPSANITIZER, and EFTSANITIZER. In this chapter, we evaluate performance and effectiveness of these frameworks and compare them with state-of-the-art tools.

6.1 Experimental Evaluation of FPSANITIZER and EFTSANITIZER

This section presents the results of our experimental evaluation for FPSANITIZER and EFTSANITIZER. We show how effective are both of these tools in detecting and debugging numerical bugs. Then, we compare the efficiency in terms of performance for both of these tools.

6.1.1 Prototype

FPSANITIZER takes as input C/C++ programs and generates the LLVM intermediate representation (IR) of the input program using the Clang++ frontend. FPSANITIZER consists of two components. (1) A standalone LLVM-9.0 pass to instrument a FP 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. Our tool currently supports precision of 128, 256, 512, and 1024 bits of precision. We provide compile-time flags to run the tool in the detection and debugging mode.

Similar to FPSANITIZER, EFTSANITIZER supports C/C++ applications with floating-point computations. 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 in contrast to FPSANITIZER.

6.1.2 Methodology

To measure the effectiveness of these tools in detecting numerical errors, we use a collection of 46 tests with known numerical errors from correctness test suites of Herbgrind and from numerical analysis textbooks [78]. For these test suites, we compared the results with 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 FPSANITIZER and 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.

Both of these tools support two versions: (a) tracing mode 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 the operands and the timestamp.

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

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 4-core Intel Core i7-7700K machine with 32GB of main memory. 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).

6.1.3 Effectiveness in Detecting and Debugging Numerical Errors

To evaluate the effectiveness in detecting and debugging numerical errors, we ran our tools with micro-benchmarks with known numerical bugs. We also show that EFTSANITIZER effectively debugs numerical errors in long-running applications due to low-performance overheads. Further, we demonstrate our EFTSANITIZER's technique and compare its debugging support with FPSANITIZER with a case study.

Evaluation with Existing Correctness Suites

When we evaluated FPSANITIZER and 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 both of these tools detect numerical errors similar to existing shadow execution tools.

Table 6.1: Summary of our experiments to detect and debug numerical errors in various scientific computing applications and NAS Parallel Benchmarks 3.0 [84] 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 experiments, 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
	D	S	D	S	D	S	D	S	
HPCG	147	4	0	0	0	0	118	3	14.95X
Laghos	0	0	40800	2	0	0	553	1	2.42X
Quicksilver	0	0	0	0	0	0	0	0	9.06X
LULESH	285246131	3	0	0	138	1	373615808	33	23.69X
Kripke	0	0	0	0	0	0	0	0	1.05X
AMG	0	0	5	5	0	0	0	0	2.57X
NAS BT	20	4	0	0	0	0	10	4	33.79X
NAS CG	31	3	0	0	0	0	1	1	13.62X
NAS EP	2	2	0	0	0	0	0	0	3.58X
NAS FT	2	2	0	0	0	0	0	0	16.13X
NAS IS	0	0	4	4	0	0	0	0	7.97X
NAS LU	26	5	0	0	0	0	11	3	45.86X
NAS MG	96	2	0	0	0	0	0	0	11.58X
NAS SP	89020284	4	0	0	0	0	1041064223	15	15.42X

Detecting Numerical Errors in Large Applications

EFTSANITIZER and FPSANITIZER are able to detect numerical errors in long-running applications. However, EFTSANITIZER has low performance overheads and more effective in detecting numerical bugs in large applications. Hence, we evaluated EFTSANITIZER by executing it with various large applications from the LLNL application suite and NAS parallel benchmarks. In this process, we detected previously unknown bugs in many applications. Table 6.1 summarizes our experiments in finding numerical errors in long-running applications. Table 6.1 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

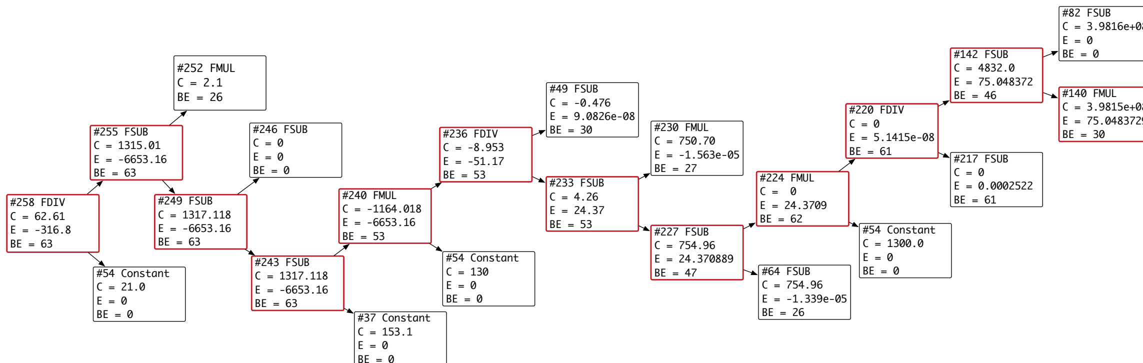

Figure 6.1: The DAG generated by EFTSANITIZER for debugging the root cause of the error for the case study with GEPP. Each DAG node show the instruction opcode, computed value, propagated rounding error with EFTs, and the number of bits in error in the computed value in comparison to the shadow execution. The node #258 (FDIV) produces the observed wrong result. The root cause of this error is caused by instruction at node #142 (FSUB).

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 [70], Lulesh [59], AMG [69], and NAS IS. We identified the root cause of these bugs using the debugging support.

A Case Study of 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 FPSANITIZER. 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 [103].

Gaussian elimination (GE) is a direct method to solve a system of linear equations of form $qAx = b$. 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 GEPP, a column with a maximum absolute value called pivot is selected and swapped with the first row if the first row does not have the maximum absolute value. This technique reduces 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 [103]).

$$A = \begin{bmatrix} 21.0 & 130.0 & 0.0 & 2.1 \\ 13.0 & 80.0 & 4.74E+8 & 752.0 \\ 0.0 & -0.4 & 3.9816E+8 & 4.2 \\ 0.0 & 0.0 & 1.7 & 9E-9 \end{bmatrix} \tag{6.1}$$

$$b = \begin{bmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6E-8 \end{bmatrix} \tag{6.2}$$

When we implement GEPP using the single precision float type, we get the following solution:

$$x = \begin{bmatrix} 62.62 \\ -8.95 \\ 0.00 \\ 1.0 \end{bmatrix} \tag{6.3}$$

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 EFTSAN-

ITIZER. 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 6.1. 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 6.1) 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 6.1). The error in this computation is the root 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.

A Case Study Showing Shortcomings of EFTs as an Oracle

We have evaluated EFTSANITIZER for various benchmarks showing promising results in detecting and debugging numerical errors with long-running applications. However, there are shortcomings in using EFTs as an oracle to detect numerical errors. This section demonstrates such use cases for completeness. For that purpose, let us consider the below program.

```
int main(){
  double a = -2.99376e+307;
  double b = 1.7976931348623157E+308;
  double x = a + b;
}
```

For this program, EFTSANITIZER reported a spurious error. However, FPSANITIZER, with 128 bits of precision, reported no error. For this program, the error computed using

```
1. double TwoSum(double a, double b){
2.   double x = a + b;
3.   double b` = x - a;
4.   double a` = x - b`;
5.   double da = a - a`;
6.   double db = b - b`;
7.   double dx = da + db;
8.   return dx;
9. }
```

Line	FP Value
2	x = 1.4983171348623158e+308
3	b` = inf
4	a` = inf
5	da = inf
6	db = inf
7	dx = nan

Figure 6.2: This figure demonstrates a case of spurious rounding error reported by EFTSanitizer. In this case, rounding error computed using TwoSum results in an overflow when actual FP computation does not. Due to the overflow, a spurious high error is reported by the EFTSanitizer.

TwoSum overflows as shown in Figure 6.2. Due to this overflow, b' results in Inf, and the error computed in line 7 results in a NaN (Not-A-Number), leading to a high error reported by EFTSanitizer.

Another example of spurious errors reported by both EFTSanitizer and FPSanitizer is when error free transformations are used within the program to improve the accuracy. These errors are reported only when error detection is enabled for all instructions. Such spurious errors can be avoided if the user avoids instrumenting such functions using selective shadow execution.

For example, consider the below program, which implements Kahan Summation [52].

```
double kahanSum(vector<double> &input)
{
  double sum = 0.0;
  double c = 0.0;
```

```
  for(double x : input)
  {
    double y = x - c;
    double t = sum + y;
    double k = t - sum;
    c = k - y;
    sum = t;
  }
  return sum;
}
```

For this program, EFTSANITIZER reports a high rounding error for subtraction $(k - y)$. This subtraction computes the rounding error that occurred in the summation $(sum + y)$ using EFTs. For example, for $x = 0.2$ and no rounding error associated with it, $y = 0.1$ with no rounding error, the sum is computed as $t = x + y$, results in a small rounding error, $-2.7E - 17$. Kahan Summation extracts this rounding using EFTs and adds it back to the final sum. Hence, to get the rounding error, in the next instruction, 0.2 is subtracted from t, which gives $k = 0.1$, and the rounding error is $-2.7E - 17$, which is propagated from t. In the next instruction, 0.1 is subtracted from k, which results in $c = 2.7E - 17$ and rounding error as $-2.7E - 17$, which is propagated from k. Since error and computed values are the same in magnitude, EFTSANITIZER reported a high-rounding error for c.

However, as we mentioned, users could disable parts of the program that use EFTs using selective shadow execution in both EFTSANITIZER and FPSANITIZER.

6.1.4 Performance Evaluation of FPSANITIZER and EFTSANITIZER

Figure 6.3 reports the performance overhead of FPSANITIZER's shadow execution, compared to an uninstrumented hardware FP baseline. We evaluate FPSANITIZER's overheads

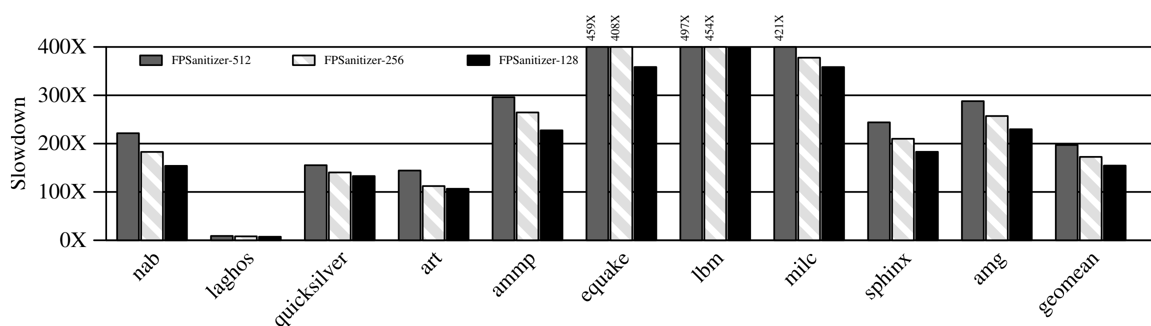

Figure 6.3: FPSANITIZER 's performance overhead with varying precision bits (512, 256, and 128) for the shadow execution compared to an uninstrumented baseline application that uses hardware FP operations.

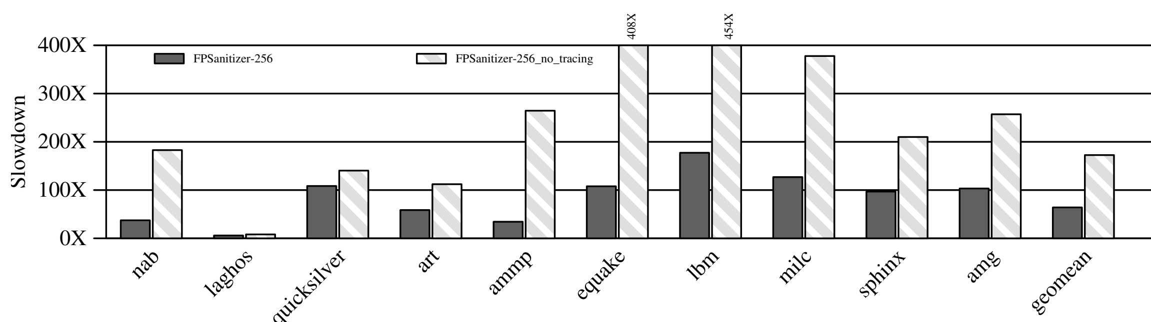

Figure 6.4: Performance slowdown of FPSANITIZER with and without tracing for shadow execution with 256 bits of precision.

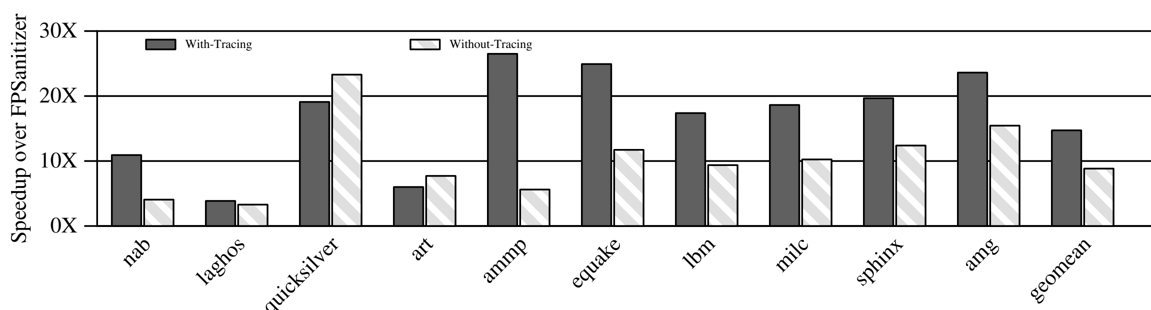

Figure 6.5: The first bar of this graph shows the speedup achieved with EFTSANITIZER when compared to shadow execution with FPSANITIZER when tracing is enabled. The second bar of this graph shows the speedup achieved with EFTSANITIZER compared to shadow execution with FPSANITIZER when tracing is disabled.

with increasing precision for the shadow execution's MPFR value. On average, FPSANITIZER's shadow execution has a performance overhead of 197×, 172×, and 154× with 512, 256, and 128 bits of precision, respectively.

All SPEC applications have a higher memory footprint and have higher overhead. Applications `milc`, `equake`, and `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 parallelism that overlaps misses and reduces the effectiveness of the prefetcher.

Figure 6.4 reports the overhead with and without tracing for FPSANITIZER. On average, the performance overhead decreases from $172\times$ to $64\times$. 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 applications.

Figure 6.7 shows the speedup of EFTSANITIZER's shadow execution when compared to FPSANITIZER. We compared both EFTSANITIZER and FPSANITIZER with the debugging support for DAGs (*i.e.*, tracing mode) and without support for DAGs where it 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 $14.72\times$ 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 $8.83\times$ on average. This significant speedup is attributed to the use of error free transformations that uses hardware FP arithmetic to compute the error as the oracle. In summary, EFTSANITIZER is not only faster than FPSANITIZER but also provides better debugging information to diagnose the root cause of errors and debug them.

We wanted to understand the total slowdown of EFTSANITIZER compared to a baseline without any instrumentation for shadow execution. Figure 6.6 reports the EFTSANI-

TIZER's slowdown in the tracing mode that produces DAGs compared to a baseline without any instrumentation (*i.e.*, total height of each bar). On average, EFTSANITIZER slowdowns the program by $10.75\times$ in comparison to an uninstrumented application. Computing the error using EFTs for primitive FP instructions and high-precision computation for math functions slowdowns the execution by $1.71\times$.

For every load instruction, we compare the program's FP value and the FP value stored in the 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 an additional $5.20\times$ slowdown (*i.e.*, load stack in Figure 6.6). For each store instruction, we copy the metadata from temporary metadata space to shadow memory. Handling store instructions additionally slows down the program by $0.44\times$ (*i.e.*, store stack in Figure 6.6). For each FP instruction, we allocated 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 $1.45\times$ overhead (*i.e.*, FP Ops stack in Figure 6.6. Handling other FP instructions such as FPToSIInst, FPToUIInst, and function arguments and returns introduces the remaining overheads (*i.e.* Other stack in Figure 6.6).

EFTSANITIZER's detection mode that does not produce DAGs slows down the execution by $6.55\times$ 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.

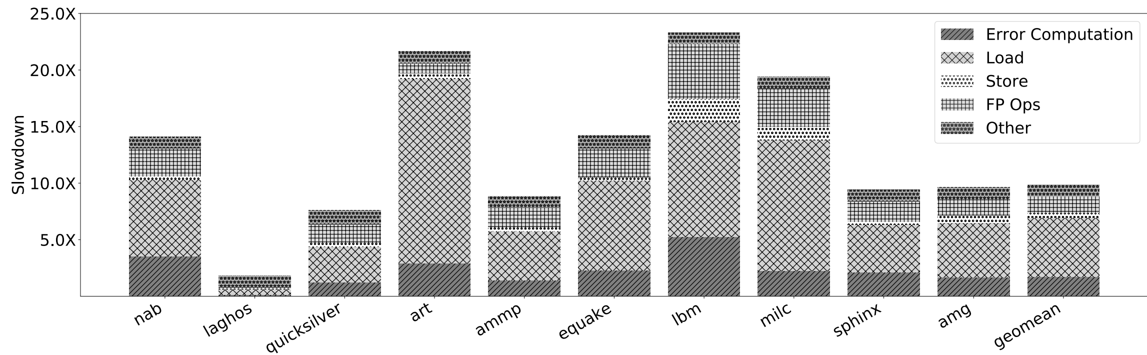

Figure 6.6: This figure shows the breakdown of the slowdown with EFTSANITIZER in the tracing mode compared to a baseline without any instrumentation (*i.e.*, the total height of each bar). We show the individual components for this overhead: (a) computation of propagated error using EFTs, (b) metadata propagation for load operations, (c) metadata propagation with store operations, (d) metadata accesses with FP operations, and (e) metadata management for the remaining FP operations.

6.2 Experimental Evaluation of PFPSANITIZER

This section briefly describes our prototype PFPSANITIZER, methodology, and performance evaluation. PFPSANITIZER requires users' input to specify regions to debug and hence we evaluate it separately.

6.2.1 Prototype

We built the prototype of PFPSANITIZER with two components: (1) an LLVM-9.0 compiler pass that takes C programs as input and creates binaries with shadow tasks and (2) a runtime written in C++ that manages worker threads, shadow memory, and performs the high-precision computation using the MPFR library [39]. PFPSANITIZER can be customized to perform shadow execution with a wide range of precision bits and also check error at various granularities. PFPSANITIZER is open source and publicly available [21].

6.2.2 Methodology

To evaluate the detection abilities and performance of PFPSANITIZER, we perform experiments using C applications from the SPEC 2000, SPEC 2006, PolyBench, and CORAL ap-

plication suites. SPEC is widely used to test the performance of compilers and processors. CORAL is a suite of applications developed by Lawrence Livermore National Laboratory to test the performance of supercomputers. Specifically, AMG is a C application that is an algebraic multi-grid linear system solver for unstructured mesh physics packages. To test the detection abilities, we used a test suite with 43 micro-benchmarks that contain various FP errors that have been used previously by prior approaches [18, 99]. We performed all our experiments on a machine with AMD EPYC 7702P 64-Core Processor and 126GB of main memory. We disabled hyper-threading and turbo-boost on our machines to minimize perturbations. We measure end-to-end wall clock time to evaluate performance. We report speedups over our prior work FPSanitizer [18, 19] , which is the state-of-the-art shadow execution tool for inlined shadow execution. We use the exact same precision both for FPSANITIZER and PFPSANITIZER when we report speedups. We use the uninstrumented original program to report slowdowns with PFPSANITIZER. To compute the error in the double value produced by the program in comparison to the real value, we convert the MPFR value to double and compute the ULP error between the doubles [18, 99]. If the exponent of the two such values differs, then all the precision bits are in error. If all the bits differ, then the entire double value is influenced by rounding error.

6.2.3 Placement of Directives

To create tasks for parallel shadow execution, we profiled applications to identify loops with independent iterations and placed directives. In the absence of such fragments, we placed directives following the approach that one typically takes to debug a large program. When the programmer does not know if a bug exists in the program, it may be beneficial to run it with a single directive (*i.e.*, entire program), which can provide a maximum speedup of $2\times$ over inlined shadow execution. Once we are certain about the existence of the bug, we use the following procedure to debug it. We profile the application using the `gprof` profiler, identify the top-n functions, and place the directives at the beginning of these

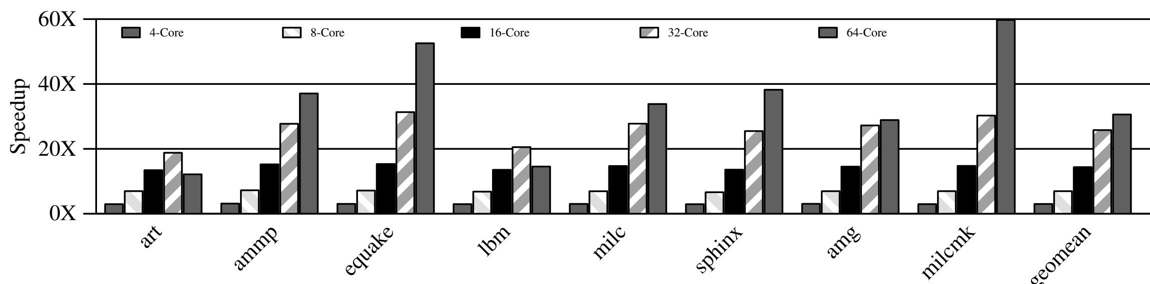

Figure 6.7: This graph reports the speedup of PFPSANITIZER over FPSanitizer when the program is executed with 4 cores, 8 cores, 16 cores, 32 cores, and 64 cores, respectively. As these report speedups, higher bars are better.

functions. If this has sufficient parallelism and we can debug the error, then the process ends. Otherwise, we remove the old directives, insert new $n/2$ directives corresponding to the top $n/2$ long-running functions, and repeat this process. This process continues until we either debug the root cause of the bug with sufficient parallelism or end up with a single directive. For our performance experiments, we placed directives using the above procedure to ensure that the application had enough parallelism for execution on 64-cores.

6.2.4 Ability to Detect FP Errors

To test the effectiveness of PFPSANITIZER in detecting existing errors, we tested it with a test suite used by previous tools. Out of the 43 tests, 12 test cases are from the Herbgrind test suite, and the rest are from the FPSanitizer test suite. These test cases include 16 cases of catastrophic cancellation (*i.e.*, all the bits are wrong between the real value and the FP value), 5 cases of branch divergences, and 2 cases of exceptional conditions such as NaNs and infinities. Rest of them do not have any numerical error but have tricky FP computation that test dynamic tools. PFPSANITIZER detects all errors without reporting any spurious errors.

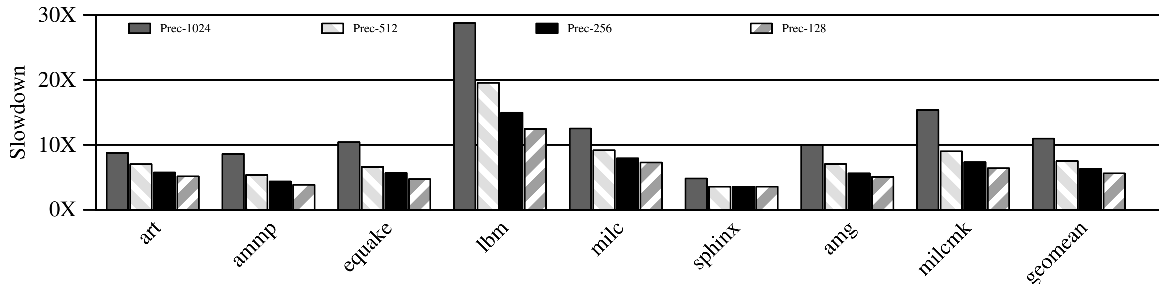

Figure 6.8: This graph reports the slowdown experienced due to parallel shadow execution with PFPSANITIZER when compared to a baseline without any instrumentation. We report the slowdowns when we vary the number of bits used for the precision in the MPFR data type: 1024 (Prec-1024), 512 (Prec-512), 256 (Prec-256), and 128 (Prec-128) bits of precision.

6.2.5 Performance Evaluation of PFPSANITIZER

Figure 6.7 reports the speedup with PFPSANITIZER that uses 512-bits of precision for the MPFR type when compared to `FPSanitizer`, which is the state of the art for shadow execution of FP programs, with the increase in the number of cores. On average, PFPSANITIZER provides a speedup of $30.6\times$ speedup over FPSanitizer with 64 cores. PFPSANITIZER provides speedups of $3.0\times$, $7.0\times$, $14.3\times$, and $25.8\times$ speedup over FPSanitizer with 4 cores, 8 cores, 16 cores, and 32 cores, respectively. This increase in speedup with the increase in the number of cores highlights PFPSANITIZER's scalability. We observe that some applications provide more speedup with 32 cores than 64 cores because there is not enough work in the application to utilize all cores when executed with 64 cores.

Figure 6.8 shows execution time slowdown of PFPSANITIZER with varying precisions for the MPFR type (128, 256, 512, and 1024 bits of precision) over a baseline that does not perform any shadow execution. On average, PFPSANITIZER experiences a slowdown of $5.6\times$, $6.2\times$, $7.5\times$, and $10.9\times$ compared to the baseline without any shadow execution for the MPFR types with 128, 256, 512, and 1024 bits of precision, respectively. In contrast, prior work FPSanitizer has slowdowns of $232\times$ on average with 512 bits of precision over the same baseline with these applications. This order of magnitude decrease in slow-

down from FPSanitizer to PFPSANITIZER enables effective debugging with long-running applications.

We also investigated the cause of the remaining overheads with parallel shadow execution. First, the producer has to provide values to the queue and has to wait when all the tasks are active, which causes an overhead of $3\times$ over the baseline. Second, accesses to shadow memory and the queues by the consumer task introduces an additional overhead of $3\times$. Third, the high precision computation with the MPFR library introduces additional $1.5\times$ overhead on average. All these overheads together add up to $7.5\times$ slowdown with PFPSANITIZER using 512 bits of precision over the baseline. In summary, PFPSANITIZER reduces the performance overhead of shadow execution significantly, which enables the use of shadow execution with long-running applications.

To summarize, our evaluation shows if a user can tolerate false positives due to limited precision with EFTs, then our tool EFTSANITIZER can detect numerical errors with very low-performance overheads with reasonably long-running applications. For comprehensive error detection with high-precision, FPSANITIZER is a great fit. However, the user has to pay in terms of performance costs. With FPSANITIZER, the user can choose the precision for shadow execution and pay in terms of performance cost as the number of precision bits increases. In contrast to these approaches, PFPSANITIZER is appealing when the user wants to debug multiple code regions independent of each other in parallel.

CHAPTER 7
RELATED WORK

Given the history of errors with FP programs, there have been numerous proposals to detect and debug numerical bugs. These proposals can be classified into static or dynamic analysis techniques. Static analysis techniques analyze the program for all input at compile time. The dynamic analysis techniques analyze the program at run-time for the given input. The dynamic analysis provides more precise and accurate information for the given input. However, analyzing the program at the run time results in high overheads. The dynamic analysis can be further classified into a lightweight and heavyweight approaches to detect numerical errors. The heavyweight approach comprehensively detects all errors, whereas the lightweight approaches detect a class of error. Further, there has been prior work to reduce the performance overheads of heavyweight approaches by running the dynamic analysis in parallel. In this chapter, we described the related prior work to detect and debug numerical errors. We also describe the prior work on using EFTs to extend the precision for various algorithms and for encapsulating the error.

Later in the chapter, we describe work to generate input that causes high rounding error and work related to precision tuning. Such work complements our work and enables techniques to detect numerical errors effectively.

7.1 Static Analysis for Detecting Numerical Errors

There is a large body of work on detecting numerical errors using both static analysis and dynamic analysis. Static analysis techniques [4, 11, 25, 28, 32, 37, 42, 44, 54, 73, 74, 75, 76, 98, 101] use abstract interpretation or interval arithmetic to reason about numerical errors for all inputs.

Approaches based on interval arithmetic [11, 37] compute the upper and lower bound for FP instruction, and these bounds are propagated with each instruction in the program. The final result using interval arithmetic shows the lower and upper bound of the result. These bounds can be calculated using two rounding towards minus infinity to get the lower bound and rounding towards plus infinity to get the upper bound. The actual output value lies in between these bounds. Hence, interval arithmetic provides a good estimation of the error. However, computing bounds require the input ranges, which are often unavailable. Moreover, with multivariate input and applications with loops, such bounds often become too large to provide any meaningful error estimation to the user.

An alternative approach to estimate the rounding error is modeling the errors using Symbolic Taylor Expansions [30, 101]. In this approach, rounding error is overapproximated using abstract models of floating-point arithmetic, and the maximum roundoff error is calculated by solving an optimization problem. The optimization problem is simplified using Taylor expressions, and the global optimization technique is used to get rigorous bounds for error expressions. Although approaches based on Taylor Expansions provide tight bounds on error in contrast to interval arithmetic, solving optimization problems results in high overheads in practice. Moreover, such approaches do not work well for programs with branches, function calls, and pointers.

One way to avoid floating-point errors is to let the user write a program in real-valued specification with maximum tolerable error for the output and let the compiler automatically generate the code with a data type that gives results within the error threshold. Rosa [29] is based on this idea and is a source-to-source compiler that takes the program in real-valued specification language and generates the code in finite precision for floating-point and fixed-point programs. Rosa performs a search on data type and applies affine and interval arithmetic to get sound estimates of the roundoff error to generate the code with finite precision. Such an approach to let the compiler automatically choose the finite preci-

sion to get the result within a pre-specified error is appealing. However, such an approach does not scale well with loops, functions, and pointers.

To summarize, error bounds for all inputs from the static analysis are appealing. However, the bounds can be too large, especially in the presence of loops, function calls, and pointer-intensive programs. Our goal is to detect numerical errors in long-running applications. Enabling approaches based on static analysis for long-running applications is still an open research problem. Our work focuses on enabling dynamic program analysis to detect and debug numerical errors in long-running applications.

7.2 Dynamic Analysis for Detecting and Debugging Numerical Errors

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 heavyweight techniques that comprehensively detect errors and lightweight techniques that detect specific errors.

7.2.1 Shadow Execution Based Analysis

Dynamic analysis needs some oracle to compare against the FP execution. Inlined shadow execution with a real number, which is approximated with a high-precision MPFR data type, is one such oracle. FPDebug [6] and Herbgrind [99] are examples of approaches with inlined shadow execution. FPSANITIZER described in Chapter 3 is based on a similar idea. On the contrary, NSan [23] performs shadow execution with finite-precision data type. FPDebug and Herbgrind perform shadow execution with dynamic binary instrumentation using Valgrind [85], which introduces significant overheads. FPSANITIZER addresses this issue with LLVM IR-based instrumentation. Among these approaches, FPDebug and NSan, do not provide additional information for debugging errors. On the contrary, Herbgrind and FPSANITIZER provide DAGs to debug errors. Such DAGs can be used with tools like Herbie [88] to rewrite expressions. To provide DAGs, Herbgrind stores metadata that is

proportional to the number of dynamic instructions with each memory location. Hence, it runs out of memory with long-running applications. In contrast, FPSANITIZER bounds the usage of memory and can run with large applications without encountering out-of-memory errors. However, large performance overheads (*i.e.*, $100\times$ or more) make it challenging for debugging. Further, expert-crafted code (*e.g.*, error free transformations [78]) is a challenge for these approaches as they can report spurious errors with them. PSO [107] tackles this problem by building heuristics to detect such instructions and assist tools to avoid such scenarios.

Instead of using the MPFR library, real arithmetic has also been approximated with constructive reals [7, 8, 62]. Using constructive reals, we are more likely to detect all numerical errors, which is not possible using limited precision with the MPFR library. However, they will likely be as slow as the MPFR library.

7.2.2 Instruction-Based Analysis

Each floating-point instruction results in a small and bounded rounding error. However, rounding errors can be accumulated with the sequence of operations or magnified due to certain operations. Rounding errors are magnified when two very close values with rounding errors are subtracted (catastrophic cancellation). Hence, BZ [3] detects such catastrophic cancellation with floating-point subtraction. BZ monitors just the exponent of the operands and the result of the FP computation to detect catastrophic cancellation. If the exponent of the operands exceeds the exponent of the result, then it flags those operations as errors. Although approximate, such checks can be performed without the real execution as an oracle. The propagation of such likely errors can be tracked to see if they affect branch predicates. RAIVE [63] uses a similar approximation, computes the impact of such likely errors on the final output of the program, and uses vectorization to reduce performance overheads. FPSpy [35] relies on hardware condition flags and uses exception handling to detect FP errors in binaries. It has low overheads as long as the program is monitored rarely,

and overhead can exceed shadow execution tools when such exceptions are monitored on each instruction.

Another approach to estimate rounding errors is to perform a dynamic analysis using random perturbations on the floating-point operations [22, 34, 36, 57, 89, 102, 108]. CADNA [57] and Verrou [36] use random rounding, whereas MCALIB [40] and Verificarlo [34] use Monte Carlo Arithmetic. Similarly, the condition number of individual operations can be used to detect numerical errors and instability in FP applications [41, 113]. Similar efforts have been made to detect numerical errors for machine learning [13, 60].

In contrast to these approaches that detect likely errors, our work EFTSANITIZER has similar or lower performance overheads when compared to them while providing detection and debugging support using shadow execution with EFTs. Similarly, our tool PFPSANITIZER incurs lower overheads and provides a mechanism to detect and debug numerical errors by running shadow execution with real numbers in parallel.

7.2.3 Prior Work on Error Free Transformations

The idea of error free transformation is rather old and has been used in the past for compensated summation [58, 97], compensated Horner Scheme [45], robust geometric algorithms [100, 104], and printing floating-point numbers [2].

The Fast2Sum algorithm was first used in accurate summation [58] in 1965. Then it was described by Dekker [31] 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 in modern machines. To avoid the branch instruction, the TwoSum algorithm was introduced in [61]. 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 the actual computation does

not [10]. Fast2Sum and TwoSum algorithms work with round-to-nearest rounding mode. However, similar algorithms for different rounding modes are proposed by Priest [90].

A similar algorithm for multiplication was introduced by Dekker based on Veltkamp splitting [78]. The 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 [31] 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 [78] using fma instruction computes the error of FP instruction with just two instructions. In 2003 Boldo [9] presented the algorithm to compute the error for division and sqrt using the fma instruction.

In the context of numerical error detection, EFTs have been recently used by Shaman [33]. 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. However, 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. In addition, 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.

Instead of using a lightweight oracle, there has been prior work to reduce the overheads of dynamic analysis by running it in parallel, as discussed below.

7.3 Parallel Dynamic Analysis

In the context of dynamic analysis for detecting memory safety errors and race detection, numerous parallel analysis techniques have been explored [14, 49, 105, 111, 112]. Approaches that perform fine-grained monitoring use hardware support as with a dedicated operand queue in Log-Based Architecture (LBA) [14]. Further, dataflow analyses have

been modified to accelerate dynamic analyses with LBA [106]. Other approaches for parallel data race detection and deterministic execution monitor programs at the granularity of epochs [105]. Our tool PFPSANITIZER is closest related to Cruiser [112], which is a heap-based overflow detector. Cruiser performs validity checks for each memory access on a separate core. It has a single producer and a consumer, which is acceptable when the checks are lightweight. Cruiser just needs to pass the memory address of the access to another core performing the check. In contrast to Cruiser, PFPSANITIZER addresses the issues of monitoring errors even on arithmetic instructions, parallel execution from a single-threaded dynamic execution, and a relatively heavyweight dynamic analysis with support for debugging.

7.4 Precision Tuning to Reduce Errors

One way to avoid common numerical errors is to select the appropriate precision for each variable. Previous approaches have explored tuning the precision of FP variables for all inputs and for a specific execution to improve performance and to reduce the occurrence of FP errors [1, 5, 15, 94, 96]. Precimonious [96] recommends lower precision for variables while providing the expected accuracy guarantees. However, Precimonious tunes the precision for the set of inputs but not all inputs. BLAME [94] speeds up precision tuning by using blame analysis. BLAME executes FP instructions using different floating-point precision for their operands and constructs a blame set. Once execution finishes, it analyzes the blame set to find the precision of variables so that the final result achieves the required accuracy. In contrast to Precimonious, the FPTuner [15] approach is based on static analysis and provides precision tuning for all inputs using Symbolic Taylor Expansions to model the error.

Such work complements our approach, such as these techniques can be employed to correct the program once numerical errors are detected using our tools. Further, this work

is dependent on dynamic analysis techniques, and our work on parallel shadow execution can help reduce the performance overheads of such techniques.

7.5 Identifying Inputs with High FP Error

Dynamic analyses need inputs that exercise operations with FP error. The problem of input generation for floating-point programs is challenging for the following reasons. First, the input space for floating-point is quite large, and only a small set of input domains cause rounding errors in the program. Hence, searching the error-inducing input in large input space results in slowdowns. Second, as the number of inputs in the program increases, the search space increases resulting in even more slowdowns. Third, the input for real applications is not often floating-point numbers. For example, the input could be an image or an audio file. Generating such inputs to trigger rounding errors in the application is even more challenging. Finally, often one search strategy doesn't work well for all applications, and different applications might require different mechanisms to generate input that trigger rounding errors. Hence, designing a tool that works for different kinds of multivariate inputs is a challenging problem.

Prior work has explored this problem using techniques such as symbolic execution [46, 47, 87] and random input generation [16, 17, 109]. Advances in symbolic execution frameworks [12, 64, 65] enabled input generation techniques for floating-point programs [46, 47, 87]. The other line of work uses search-based techniques to find an input that generates rounding errors. For example, BRGT [17] uses a binary search over input to find the input that maximizes the error.

Techniques to generate inputs complement our approach and can enable us to detect and debug FP errors efficiently.

CHAPTER 8
CONCLUSION AND FUTURE DIRECTIONS

In this chapter, we summarize this dissertation's key technical contributions and present future work directions.

8.1 Dissertation Summary

Real numbers are approximated using a finite number of bits in computer systems for performance reasons. However, fitting all real numbers with a finite number of bits results in rounding errors. One such widely used approximation for real numbers is the Floating-Point (FP) representation. With each primitive FP arithmetic operation, rounding errors are negligible. However, with a sequence of operations, rounding errors can be magnified, resulting in a wrong output of the program. Unsurprisingly, rounding errors have caused various catastrophic incidents in the past. Hence, there is a large body of work on reasoning about the correctness of FP programs. Prior approaches perform inlined shadow execution with real numbers to comprehensively detect numerical errors. To debug numerical errors, additional information about instructions is stored in shadow memory. Using additional information, a backward slice of the program is generated for root-cause analysis. However, inlined shadow execution with real numbers introduces significant overheads making such approaches infeasible with long-running applications. Moreover, additional information stored in memory for root-cause analysis introduces additional memory overheads. This dissertation proposes various approaches to reduce overheads with inlined shadow execution while providing clean interfaces to test and debug numerical programs.

The key contribution of this dissertation is to store the constant amount of metadata with each instruction to reduce the overheads and yet provide a mechanism to debug numerical errors. To further reduce the overheads, we have developed a novel approach to execute

parts of the shadow execution in parallel. The key contribution is designing a mechanism to run the parts of shadow execution from an arbitrary memory state. In our approach, the user marks the program regions for inlined shadow execution. Our compiler generates shadow execution tasks that can run in parallel on multiple cores, accelerating the inlined shadow execution. However, our technique depends on the user to mark the code regions for parallel shadow execution. Hence, we explored an alternative mechanism to reduce the overheads of inlined shadow execution. This dissertation proposed a lightweight oracle enabled by hardware-supported FP data type to measure the error and significantly reduce the overheads compared to shadow execution with high-precision computation (using MPFR library). The lightweight oracle is designed using error-free transformations (EFTs) to capture the rounding error with primitive FP instructions. The key contribution in designing such an oracle is the error composition for the sequence of FP instructions. Moreover, we provide a backward slice of k dynamic instructions to enable the users to debug numerical errors.

8.1.1 Detecting and Debugging Numerical Errors in Computation with Floating-Point

One way to detect numerical errors is to use shadow analysis. Shadow analysis with high precision computation enables comprehensive numerical error detection. In this approach, any FP value in register and in memory is shadowed with a high precision value. On every FP computation, a high precision computation is executed with high precision values. If there is a significant difference between FP computation and high precision computation, then the error is reported to the user. Similarly, if branch outcomes are different, branch flip is reported to the user. To detect errors, we maintain the real value for each memory location and each temporary. To debug errors, we need to produce the backtrace of instructions responsible for the error. Hence, for every FP instruction, we also maintain

information about the metadata for its operands. Once the error is detected, metadata is traversed recursively to provide the DAG of instructions responsible for the error. The key point is that we store metadata proportional to the static instructions in the program. Hence, a constant amount of metadata for each memory location enables detecting and debugging numerical errors in long-running applications in contrast to prior approaches. We also provide a mechanism to start the shadow execution at any arbitrary point during the execution of the program. Using our selective shadow execution technique, users can instrument critical parts of the program for shadow execution, reducing overheads significantly. Our prototype FPSANITIZER based on these ideas detects numerical errors with floating-point applications and is an order of magnitude faster than the prior work.

8.1.2 Parallel Shadow Analysis To Accelerate the Debugging of Numerical Errors

In shadow analysis, real numbers are typically simulated with a high-precision software library. Hence, a software simulation of real numbers is the primary reason for high overheads. One way to reduce the overheads is to perform shadow analysis in parallel on multicore machines. We proposed a novel approach in this dissertation to detect and debug numerical errors in long-running applications that perform shadow analysis in parallel. In our model, the user specifies parts of the program that need to be debugged. Our compiler creates shadow execution tasks that mirror the original program for these specified regions but performs FP computations with high precision in parallel. Since we are creating shadow tasks from a sequential program, shadow tasks are also sequential depending on prior tasks for memory state. To execute the shadow tasks in parallel, we need to break the dependency between them by providing the appropriate memory state and input arguments. Moreover, to correctly detect the numerical errors in the original program, shadow tasks need to follow the same control flow as in the original program.

Our key insight is to use FP values computed by the original program to start the shadow task from some arbitrary point. Our compiler introduces the additional instrumentation in the original program to provide live FP values, branch outcomes, and memory addresses. To ensure shadow tasks follow the same control flow as the original program, our compiler updates every branch instruction in the shadow task to use the branch outcomes of the original program. The original program and shadow tasks execute in a decoupled fashion and communicate via a non-blocking queue. Our shadow tasks do not have any information about integer operations in the original program. Hence, shadow tasks read the memory address with FP value from the queue and map it to the shadow memory location with a high-precision value. On every memory load in the original program, shadow tasks access the shadow memory and check if it has a valid high-precision value. If this check fails, then the shadow task initializes the shadow memory with a computed FP value. Hence, using the FP value from the original program enables us to perform parallel shadow execution from a sequential program. To compute the error, the shadow task compares the high precision value with the actual computed value and reports it to the user if the difference is above the threshold. Similarly, to detect branch flips, the shadow task compares the FP branch outcome in the actual program with the high precision branch outcome in the shadow task. To run shadow tasks in parallel, our runtime maps shadow tasks to one of the available cores inspired by work-stealing algorithms to get scalable speedups. Once the shadow task reports the error, a directed acyclic graph of instructions is generated to give feedback to the user. Our tool PFPSANITIZER is an order of magnitude faster than the state-of-the-art and comprehensively detects numerical errors within the specified regions. PFPSANITIZER helped us to detect and debug numerical errors in the Cholesky benchmark from the Polybench suite.

8.1.3 Shadow Analysis With Error Free Transformations

Parallel shadow analysis is an effective approach to reduce the overheads significantly. However, it requires user input to direct the compiler to create shadow tasks. Alternatively, in this dissertation, we proposed a shadow analysis technique with a lightweight oracle to reduce overheads significantly. This approach uses hardware-supported FP data type to capture the numerical error. We transform FP computations to capture the rounding error accurately. For example, FP addition $a + b$ where $|a| \geq |b|$ is transformed into $x + y$. In this transformation, $x = FP(a + b)$ represents the computed FP addition with round-to-nearest mode and $y = FP(b - (a + b) - a)$ represents the exact rounding error occurred during this computation. Such transformations are based on the fact that rounding errors in FP representation can be accurately stored in an FP data type. These transformations are called Error Free Transformations (EFTs), and they can extend the accuracy beyond the available hardware primitive type. Our key insight is that using EFTs for FP arithmetic instructions can provide a lightweight oracle to capture the rounding error due to available hardware support. However, the key challenge is to capture the error for the sequence of operations, including math functions that do not have support for EFTs. We have also designed a novel mechanism to provide a DAG of instructions for k dynamic instructions. The DAG generated by EFTSanitizer spans multiple function calls and loop iterations, enabling developers to debug numerical errors effectively. EFTSanitizer enables selective shadow execution for arbitrary fragments of dynamic execution and enables effective debugging of numerical errors. EFTSanitizer is an order of magnitude faster than FPSanitizer and Herbgrind. Using EFTSanitizer, we detected various numerical bugs in long-running applications.

8.2 Future Research Directions

This dissertation makes a case for fast approaches to detect and debug numerical errors with long-running applications. The ideas presented in this dissertation can be applied to different domains to detect other bugs such as data race detection, memory errors, and taint analysis. This section also presents new ideas to improve the line of work proposed in this dissertation for numerical error detection and debugging.

8.2.1 C Program Reduction for Numerical Bugs

To debug numerical bugs in long-running applications, we focused on reducing the overheads by designing smart metadata, parallel shadow execution, and lightweight oracle design. However, if a program runs for 30 minutes, it is challenging to debug numerical bugs with even zero overheads. An alternative approach could be generating a smaller program that automatically triggers the same numerical bug. This approach has been applied in the past to generate a smaller program for compiler optimization bugs [72, 91]. This problem could be seen as a search problem, and the key challenge is minimizing the search space and directing the search algorithm in the right direction. In our work, we generated the directed acyclic graph of instructions to enable the root cause analysis. The search algorithm can be directed using DAG of instructions to generate smaller test cases.

8.2.2 Detecting and Debugging Numerical Errors in Scripting Languages

Application development is moving from system languages to scripting languages for its easy-to-use interface and faster app development. This dissertation proposed various ideas to significantly reduce overheads to detect and debug numerical errors in long-running applications written in system languages. These techniques work for programming languages whose compiler targets LLVM IR, including C, C++, Fortran, Julia, and Rust. In future

work, these ideas can be applied to scripting languages. Python is a widely used scripting language for machine learning and computer graphics applications. Python is an interpreted language that interacts with native libraries for heavy-lifting computations. The main challenge in designing such a system is detecting numerical errors in python interpreters and native libraries at the granularity of instruction while providing feedback to users for root cause analysis.

8.2.3 Improving the Accuracy of an Oracle Based on EFTs

EFTSANITIZER proposed in this dissertation uses a lightweight oracle based on EFTs. EFTSANITIZER detects most of the errors, but it does not generate the exact real value due to the loss of precision while composing the errors. Hence, the error value returned by EFTSANITIZER could be misleading to the user. An approach based on extended precision [100] can be applied to improve the accuracy of the error value returned by EFTSANITIZER. In the prior work [100], such an approach is used to improve the accuracy of geometric applications. However, the key challenge is to design algorithms for primitive FP operations using EFTs with a configurable number of variables for storing the error without significant performance overheads.

BIBLIOGRAPHY

[1] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. 2021. Fast and Efficient Bit-Level Precision Tuning. In *Static Analysis*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 1–24.

[2] Marc Andrysco, Ranjit Jhala, and Sorin Lerner. 2016. Printing Floating-Point Numbers: A Faster, Always Correct Method. *SIGPLAN Not.* 51, 1 (jan 2016), 555–567. https://doi.org/10.1145/2914770.2837654

[3] 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

[4] 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

[5] Dorra Ben Khalifa and Matthieu Martel. 2021. An Evaluation of POP Performance for Tuning Numerical Programs in Floating-Point Arithmetic. In *2021 4th International Conference on Information and Computer Technologies (ICICT)*. 69–78. https://doi.org/10.1109/ICICT52872.2021.00019

[6] 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. https://doi.org/10.1145/2345156.2254118

[7] Hans-J. Boehm. 2005. The constructive reals as a Java library. In *The Journal of Logic and Algebraic Programming*, Vol. 64. 3–11.

[8] Hans-J. Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. 1986. Exact Real Arithmetic: A Case Study in Higher Order Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 162–173.

[9] 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

[10] 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

[11] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. 2006. The design of the Boost interval arithmetic library. *Theoretical Computer Science* 351, 1 (2006), 111–118. https://doi.org/10.1016/j.tcs.2005.09.062

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[13] Yohan Chatelain, Nigel Yong, Gregory Kiar, and Tristan Glatard. 2021. Py-Tracer: Automatically profiling numerical instabilities in Python. *arXiv preprint arXiv:2112.11508* (2021).

[14] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *2008 International Symposium on Computer Architecture (ISCA 2008)*. 377–388. https://doi.org/10.1109/ISCA.2008.20

[15] 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.

[16] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Practical Floating-Point Divergence Detection. In *Languages and Compilers for Parallel Computing*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.). Springer International Publishing, Cham, 271–286.

[17] 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.

[18] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. https://doi.org/10.1145/3385412.3386004

[19] Sangeeta Chowdhary, Jay P Lim, and Santosh Nagarakatte. 2020. *FPSanitizer - A debugger to detect and diagnose numerical errors in floating point programs*. Retrieved June, 2021 from https://github.com/rutgers-apl/fpsanitizer

[20] 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

[21] Sangeeta Chowdhary and Santosh Nagarakatte. 2021. *PFPSanitizer - Parallel Shadow Execution to Detect and Diagnose Numerical Errors in Floating Point Programs*. Retrieved June, 2021 from https://github.com/rutgers-apl/PFPSanitizer

[22] Michael P. Connolly, Nicholas J. Higham, and Theo Mary. 2021. Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM Journal on Scientific Computing* 43, 1 (2021), A566–A585. https://doi.org/10.1137/20M1334796 arXiv:https://doi.org/10.1137/20M1334796

[23] Clement Courbet. 2021. NSan: A Floating-Point Numerical Sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/3446804.3446848

[24] Mike Cowlishaw. 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE 754-2008. IEEE Computer Society. 1–70 pages. https://doi.org/10.1109/IEEESTD.2008.4610935

[25] Nasrine Damouche and Matthieu Martel. 2018. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Automated Formal Methods (Kalpa Publications in Computing, Vol. 5)*, Natarajan Shankar and Bruno Dutertre (Eds.). 63–76. https://doi.org/10.29007/j2fd

[26] 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

[27] 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. LIP,. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804

[28] 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

[29] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (mar 2017), 28 pages. https://doi.org/10.1145/3014426

[30] 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.

[31] 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

[32] 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.

[33] Nestor Demeure. 2020. *Compromise between precision and performance in high-performance computing*. https://tel.archives-ouvertes.fr/tel-03116750

[34] Christophe Denis, Pablo Castro, and Eric Petit. 2016. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. 55–62. https://doi.org/10.1109/ARITH.2016.31

[35] 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

[36] 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.

[37] Oliver Flatt and Pavel Panchekha. 2021. An Interval Arithmetic for Robust Error Estimation. *arXiv preprint arXiv:2107.05784* (2021).

[38] 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.

[39] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[40] Michael Frechtling and Philip H. W. Leong. 2015. MCALIB: Measuring Sensitivity to Rounding Error with Monte Carlo Programming. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 5 (apr 2015), 25 pages. https://doi.org/10.1145/2665073

[41] Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated Backward Error Analysis for Numerical Code. *SIGPLAN Not.* 50, 10 (oct 2015), 639–654. https://doi.org/10.1145/2858965.2814317

[42] Khalil Ghorbal, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2012. Donut Domains: Efficient Non-convex Domains for Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 235–250.

[43] 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.

[44] 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

[45] Stef Graillat, Philippe Langlois, and Nicolas Louvet. 2005. Compensated horner scheme. *Univ. of Perpignan, France, Tech. Rep* (2005).

[46] Yijia Gu, Thomas Wahl, Mahsa Bayati, and Miriam Leeser. 2015. Behavioral Non-portability in Scientific Numeric Computing. In *Euro-Par 2015: Parallel Processing*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 558–569.

[47] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020)*. 1261–1272.

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

[49] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. 2009. A Concurrent Dynamic Analysis Framework for Multicore Hardware. *SIGPLAN Not.* 44, 10 (oct 2009), 155–174. https://doi.org/10.1145/1639949.1640101

[50] Caroline N. Haddad. 2009. *Cholesky Factorization*. Springer US, Boston, MA, 374–377. https://doi.org/10.1007/978-0-387-74759-0_67

[51] Yozo Hida, Xiaoye S Li, and David H Bailey. 2007. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory* (2007), 19.

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

[53] Intel. 2020. *Intel Memory Protection Extensions Enabling Guide*. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-mpx-enablingguide.pdf

[54] Maxime Jacquemin, Sylvie Putot, and Franck Védrine. 2018. A Reduced Product of Absolute and Relative Error Bounds for Floating-Point Analysis. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 223–242.

[55] Claude-Pierre Jeannerod, Nicolas Louvet, and Jean-Michel Muller. 2013. Further analysis of Kahan's algorithm for the accurate computation of 2 x 2 determinants. *Math. Comp.* 82 (10 2013). https://doi.org/10.1090/S0025-5718-2013-02679-8

[56] 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

[57] 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

[58] William Kahan. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. In *Communications of the ACM*, Vol. 8. ACM, New York, NY, USA.

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

[60] Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. 2022. DeepStability: A Study of Unstable Numerical Methods and Their Solutions in Deep Learning. *arXiv preprint arXiv:2202.03493* (2022).

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

[62] Vernon A. Lee, Jr. and Hans-J. Boehm. 1990. Optimizing Programs over the Constructive Reals. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. ACM, New York, NY, USA, 102–111.

[63] 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.

[64] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/3338906.3338921

[65] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-Point Symbolic Execution: A Case Study in n-Version Programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 601–612.

[66] 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

[67] Jay P. Lim and Santosh Nagarakatte. 2021. 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

[68] 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

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

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

[71] LLVM. 2022. *LLVM Language Reference Manual*. https://llvm.org/docs/LangRef.html

[72] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

[73] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4, Article 34 (jan 2017), 31 pages. https://doi.org/10.1145/3015465

[74] Guillaume Melquiond. 2019. *Gappa*. http://gappa.gforge.inria.fr

[75] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. 2017. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security*, Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 213–229.

[76] Mariano M. Moscato, Laura Titolo, Marco A. Feliú, and César A. Muñoz. 2019. Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 21–37.

[77] Jean-Michel Muller. 2006. Elementary Functions, Algorithms and Implementation, 2nd Edition. (01 2006).

[78] 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.

[79] 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)*.

[80] 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*.

[81] 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*.

[82] 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*.

[83] Santosh Ganapati Nagarakatte. 2012. *Practical Low-Overhead Enforcement of Memory Safety for c Programs*. Ph.D. Dissertation. University of Pennsylvania, USA. Advisor(s) Martin, Milo M. AAI3551723.

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

[85] 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

[86] 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

[87] Gabriele Paganelli and Wolfgang Ahrendt. 2013. Verifying (In-)Stability in Floating-Point Programs by Increasing Precision, Using SMT Solving. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 209–216. https://doi.org/10.1109/SYNASC.2013.35

[88] 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*, Vol. 50. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2813885.2737959

[89] D.S. Parker, B. Pierce, and P.R. Eggert. 2000. Monte Carlo arithmetic: how to gamble with floating point and win. *Computing in Science & Engineering* 2, 4 (2000), 58–68. https://doi.org/10.1109/5992.852391

[90] Douglas M. Priest. 1992. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. Dissertation. University of California at Berkeley, USA. UMI Order No. GAX93-30692.

[91] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. 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, 335–346. https://doi.org/10.1145/2254064.2254104

[92] James Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly Associates, Inc., USA.

[93] John H. Rowland. 1976. Floating-Point Computation (Pat H. Sterbenz). *SIAM Rev.* 18, 1 (jan 1976), 138–139. https://doi.org/10.1137/1018026

[94] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1074–1085. https://doi.org/10.1145/2884781.2884850

[95] 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)*. Association for Computing Machinery, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

[96] 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.

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

[98] Rocco Salvia, Laura Titolo, Marco A. Feliú, Mariano M. Moscato, César A. Muñoz, and Zvonimir Rakamarić. 2019. A Mixed Real and Floating-Point Solver. In *NASA Formal Methods*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer International Publishing, Cham, 363–370.

[99] 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. https://doi.org/10.1145/3296979.3192411

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

[101] 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. https://doi.org/10.1145/3230733

[102] Enyi Tang, Xiangyu Zhang, Norbert Th. Müller, Zhenyu Chen, and Xuandong Li. 2017. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing. *IEEE Transactions on Software Engineering* 43, 10 (2017), 975–994. https://doi.org/10.1109/TSE.2016.2642956

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

[104] Laurent Thévenoux, Philippe Langlois, and Matthieu Martel. 2015. Automatic Source-to-Source Error Compensation of Floating-Point Programs. In *2015 IEEE 18th International Conference on Computational Science and Engineering*. 9–16. https://doi.org/10.1109/CSE.2015.11

[105] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing Sequential Logging and Replay. *ACM Trans. Comput. Syst.* 30, 1, Article 3, 24 pages. https://doi.org/10.1145/2110356.2110359

[106] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. 2010. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/1736020.1736051

[107] 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.

[108] Jackson H.C. Yeung, Evangeline F.Y. Young, and Philip H.W. Leong. 2011. A Monte-Carlo Floating-Point Unit for Self-Validating Arithmetic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/1950413.1950453

[109] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 11–20. https://doi.org/10.1109/APSEC.2017.7

[110] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. 2016. Parallel Data Race Detection for Task Parallel Programs with Locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 833–845. https://doi.org/10.1145/2950290.2950329

[111] Zhibin Yu, Weifu Zhang, and Xuping Tu. 2011. MT-Profiler: A Parallel Dynamic Analysis Framework Based on Two-Stage Sampling. In *Advanced Parallel Processing Technologies*, Olivier Temam, Pen-Chung Yew, and Binyu Zang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 172–185.

[112] Qiang Zeng, Dinghao Wu, and Peng Liu. 2011. Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-Free Data Structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 367–377. https://doi.org/10.1145/1993498.1993541

[113] 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