# Research Statement

Sangeeta Chowdhary

Modern systems are complex and built by developers spanning across multiple teams. To ensure the correctness of these complex systems testing and debugging are critical parts of the software development cycle. Moreover, developers spend almost 50% of their time in just testing and debugging. Major software companies spend billions of dollars in testing in debugging software systems. My broader research vision is to develop well-defined interfaces and novel techniques leading to efficient and effective testing and debugging of software. My thesis research has taken one step towards my broader research goal by making a case for fast yet precise mechanisms to detect and debug numerical errors.

All modern applications compute with real numbers to represent various quantities continuously. 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 Floating-Point (FP) representation. In FP arithmetic, with each primitive 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. During my Ph.D. research, I have noticed that current work lacks a clear interface to detect and debug numerical errors in FP applications. Moreover, current tools either introduce significant slowdown or detect a specific class of numerical errors. My thesis research has proposed approaches to reduce overheads while providing clean interfaces to test and debug numerical programs.

My approach employs shadow analysis with real numbers to comprehensively detect numerical errors. In this approach, real computations are executed side-by-side with FP computations, providing a mechanism to check numerical errors. Moreover, each shadow variable stores extra information (metadata) about the instructions to provide a slice of the program to debug the error. The key contribution of my thesis is to store the constant amount of metadata with each instruction to reduce the overheads and yet provide a mechanism to debug numerical errors. Hence, my approach enables comprehensive error detection and provides valuable feedback to the user with low overheads. To further reduce the overheads, I 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. Alternatively, I have proposed a lightweight oracle enabled by hardware-supported FP data type to measure the error significantly reduces the overheads. The lightweight oracle is designed using error-free transformations (EFTs), which keep track of the error by transforming FP arithmetic instructions. The key contribution in designing such an oracle is the composition of the error for the sequence of FP instructions.

**Detecting and Debugging Numerical Errors in Computation with Floating-Point and Posits:** One way to detect numerical errors is to use shadow analysis. Shadow analysis with high precision computations enables comprehensive numerical error detection. In this approach, the FP variable in register and memory is shadowed with a high precision value. On every FP computation, a high precision computation is done 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. Our prototype FPSanitizer [2] for floating-point is an order of magnitude faster than the prior work. We have applied the same ideas to detect and debug numerical errors with posits. Our prototype PositDebug helped in building math libraries for posits.

**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 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 needs 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 report 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 PFSan [4] is an order of magnitude faster than the state-of-the-art and comprehensively detects numerical errors within the specified regions. PFSan 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.

**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, to reduce overheads significantly, we are designing a shadow analysis technique with a lightweight oracle. In this approach, we use 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. Our tool EFTSanitizer [3] is an order of magnitude faster than FPSanitizer and Herbgrind. Using EFTSanitizer, we could debug long-running applications effectively and efficiently.

## Future Research Directions

In future work, I would like to apply novel techniques to build a set of tools to detect, debug, and correct numerical errors in a broader range of programming languages. I would also like to expand my research area by focusing on building a compiler for outsourced computations.

**C Program Reduction for Numerical Bugs**: To debug numerical bugs in long-running applications, my approach 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 zero overheads. An alternative approach could be to generate a smaller program that triggers the same numerical bug automatically. This approach has been applied in the past to generate a smaller program for compiler optimization bugs. This problem could be seen as a search problem, and the key challenge is to minimize the search space and direct the searching algorithm in the right direction. I generate the directed acyclic graph of instructions in my current work to enable the root cause analysis. I would like to direct the search using DAG of instructions to generate smaller test cases.

**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. In my current work, I have applied PL ideas to significantly reduce overheads to detect and debug numerical errors in long-running applications written in system languages. My techniques work for programming languages whose compiler targets LLVM IR, including C, C++, Fortran, Julia, Rust, etc. In my future work, I would like to apply these ideas to scripting languages. Python is a widely used scripting language for application domains such as machine learning, computer graphics, etc. 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.

**General Compilers for Outsourced Computations**: Outsourcing computations is currently a hot trend with the advance in cloud computing. However, this also leads to the question of data privacy and correctness. Fully Homomorphic Encryption (FHE) focuses on data privacy by computing encrypted data without revealing the private data to the remote machine. On the other hand, verifiable computations with probabilistic proofs (also called zero-knowledge proofs) focus on the correctness of cloud computations. In this paradigm, the remote machine gives a short proof of the computations, which lets the local machine verify that computation is correct. During my internship at Microsoft Research, I worked with two different teams to write the compiler for outsourced computations. First, I worked on designing a new compiler for zero-knowledge proofs called Pickle. Then, in another team, I worked on a compiler EVA for Fully Homomorphic Encryption [1]. Building these compilers required a significant amount of time, and still, they are far from what is expected. For example, these compilers still lack novel optimization techniques accustomed to arithmetic circuits targetted by these compilers. Moreover, they currently also lack type rules to provide a correctness guarantee. Additionally, there are many different types of protocols providing different security guarantees for outsourced computations. Currently, we lack a generic compiler targetting different protocols for outsourced computations while providing correctness and security guarantees. In my future work, I will design a generic compiler for outsourced computations.

My thesis research focused on designing novel and practical techniques to debug numerical programs. I have also spent time designing compilers for outsourced computations. I would like to build more robust and novel techniques to debug numerical errors in scripting languages in my future work. Furthermore, I will focus on designing a generic compiler for outsourced computations providing correctness and security guarantees.

## References


[1] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi. EVA Improved: Compiler and Extension Library for CKKS. WAHC, 2021.
[2] S. Chowdhary, J. P. Lim, and S. Nagarakatte. Debugging and Detecting Numerical Errors in Computation with Posits. PLDI, 2020.
[3] S. Chowdhary and S. Nagarakatte. EFTSanitizer - A Fast Debugger to Detect and Diagnose Numerical Errors, In Submission.
[4] S. Chowdhary and S. Nagarakatte. Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors. FSE, 2021.