Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors

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

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

ABSTRACT

This paper proposes a new approach for debugging errors in floating point computation by performing shadow execution with higher precision in parallel. The programmer specifies parts of the program that need to be debugged for errors. Our compiler creates shadow execution tasks, which execute on different cores and perform the computation with higher precision. We propose a novel method to execute a shadow execution task from an arbitrary memory state, which is necessary because we are creating a parallel shadow execution from a sequential program. Our approach also ensures that the shadow execution follows the same control flow path as the original program. Our runtime automatically distributes the shadow execution tasks to balance the load on the cores. Our prototype for parallel shadow execution, PFPSanitizer, provides comprehensive detection of errors while having lower performance overheads than prior approaches.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

floating point, dynamic analysis, FPSanitizer, rounding errors

ACM Reference Format:
Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468585

1 INTRODUCTION

The floating point (FP) representation approximates a real number using a finite number of bits. Hence, some rounding error with each operation is inevitable. Such rounding errors can be amplified by certain operations (*e.g.*, subtraction) such that the entire result is influenced by rounding errors. Even the control flow of the program can differ compared to an oracle execution due to rounding errors, which can result in slow convergence or wrong results. In extreme cases, math libraries can produce wrong results, which can be amplified by other operations. Further, the program can produce exceptional values such as Not-a-Number (NaNs), which get propagated throughout the program to produce incorrect results. Such errors have caused well documented mishaps [33].

Given the history of errors with FP programs, there have been numerous proposals to detect [1, 3, 15, 16, 24], debug [9, 38], and repair errors [34]. There are numerous tools to detect specific errors [1, 15, 16, 24]. A comprehensive approach to detect errors in FP programs is to perform inlined shadow execution with real numbers [3, 9, 38]. In the shadow execution, every FP variable in memory and registers is represented with a type that has a large amount of precision (*e.g.*, using MPFR library [17]). When the FP value and the shadow value differ significantly, the error is reported to the user. Such shadow execution tools can comprehensively detect a wide range of errors: cancellation errors, branch divergences, and the presence of special values (*e.g.*, NaNs). To assist in debugging the error, Herbgrind [38] and FPSanitizer [9] provide a directed acyclic graph (DAG) of instructions. Herbgrind when coupled with Herbie [34] has been useful in rewriting FP expressions. The debugging features in our prior work, FPSanitizer [9], have been useful in our effort to develop correctly rounded math libraries [25–29].

Inlined shadow execution is useful in detecting errors with unit tests. However, it has more than 100× performance overhead. Software emulation of a real number using the MPFR library and additional information (*i.e.*, metadata) maintained with each memory location during inlined shadow execution are the primary sources of this performance overhead. These overheads prevent the usage of such debugging tools in many applications.

Our objective is to enable the use of shadow execution tools for debugging numerical errors with long running applications. This paper proposes a new approach for debugging numerical errors that performs shadow execution in parallel on the multicore machines. In our approach, the user specifies parts of the code that needs to be debugged (*i.e.*, with directives `#pragma pfpsan` in Figure 3(a)) similar to task parallel programs. Our compiler creates shadow execution tasks that mimic the original program but execute the FP computation with higher precision in parallel. The shadow execution tasks of a sequential program, by default, are also sequential because they need the memory state from prior tasks. To execute the shadow execution tasks in parallel, we need to provide these shadow execution tasks with appropriate memory state and input arguments. We also need to ensure that the parallel task follows

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468585

the same control-flow path as the original program to be useful for debugging.

Our key insight for parallel shadow execution from a sequential program 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 is partly inspired from prior efforts on speculative parallelization [44]. In our model, the shadow task can start execution even if prior shadow tasks (according to the order in the sequential program) have not completed as long as the original program has executed the corresponding FP computation. The original program pushes the live FP values, memory addresses, and the FP values loaded from those addresses to a queue that is used by the shadow execution task (see Figure 4). The original program and the shadow execution task execute in a decoupled fashion and communicate only through the queues. The shadow execution task reads the live FP values from the queue and executes the high-precision version of the program created by our compiler. It maintains a high-precision value (*i.e.*, a MPFR value) with each FP variable in both memory and in temporaries.

When the shadow execution task loads a value from a memory location, it needs to identify whether the memory location was previously written by that task. The high-precision value is available in memory when the task has previously written to that address. Otherwise, it needs to initialize the shadow execution using the FP value from the original program. To detect such scenarios, the shadow execution task stores both the high-precision value and the FP value produced by the program in its memory when it performs a store. Subsequently when the shadow execution task performs a load, it checks whether the loaded FP value from memory and the value produced by the program are 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 (see Figure 6). This technique to use the original program’s FP value as an oracle allows us to execute shadow execution tasks from an arbitrary state. To enable effective debugging of numerical errors, the shadow execution task also maintains information about the operation that produced the value in memory, which can be used to provide a DAG of instructions responsible for the error.

Our prototype, PFPSanitizer, is open-source and publicly available [11]. It 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, then the entire execution is a single task and PFPSanitizer can attain a maximum speedup of 2×. When the user creates sufficient number of shadow tasks, PFPSanitizer is approximately 30× faster on average on a machine with 64-cores when compared to FPSanitizer, which is the state-of-the-art for debugging FP programs.

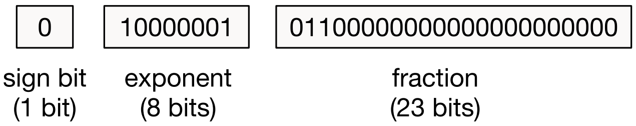


Figure 1: A 32-bit FP representation of 5.5 in decimal. First, the number 5.5 is converted to a binary fraction $(1.011)_2 \times 2^2$ and the actual bit patterns for the exponent and the fraction are determined. As $bias = 127$ for a 32-bit float and $E - bias = 2$, $E = 129$ which is 10000001 in binary. The least significant bits of the fraction are populated with zeros.

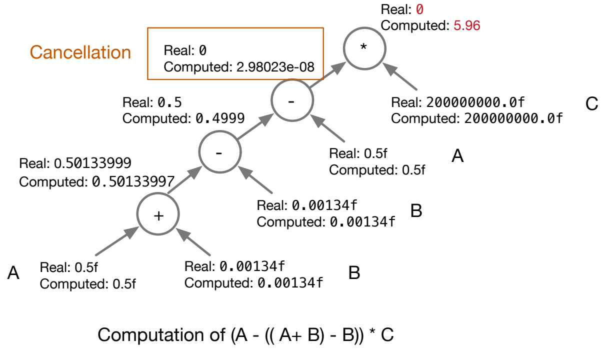


Figure 2: Error amplification with a sequence of operations, represented as a DAG. We show the real value and the FP result for each operation.

2 BACKGROUND

We provide a primer on the FP representation, the cause of errors and their accumulation with FP computation, an overview of inlined shadow execution, and a comparison of existing approaches.

The floating point representation. The floating point (FP) representation is the most widely used representation to approximate real numbers. It was standardized by the IEEE-754 standard. The FP representation is specified by the total number of bits (n) and the number of bits for the exponent ($|E|$). The representation has three components: a sign bit, $|E|$ exponent bits, and $n-1-|E|$ bits to represent the fraction. Figure 1 shows the representation for a 32-bit float, which has a sign bit, 8-bits for the exponent, and 23-bits for the fraction. A 64-bit double has a sign bit, 11-bits for the exponent, and 52-bits for the fraction. The goal of the FP representation is to encode both large and very small values.

The sign bit indicates whether the number is positive or negative. There are three classes of values depending on the bit pattern in the exponent. If the exponent bits are all 1’s, the bit pattern represents special values. It represents infinity if the fraction bits are all 0’s and NaNs (Not-a-Number) otherwise. These special values propagate with each operation. If the exponent bit pattern is all 0’s, it represents denormal values (*i.e.*, values very close to zero). The value of the number represented is $(-1)^s \times 0.F \times 2^{1-bias}$, where $bias = 2^{|E|-1} - 1$ and F is the value of the binary fraction (*i.e.*, from the fraction bits). If the exponent bit pattern is neither all 0’s nor

Table 1: Comparison of various dynamic analysis tools. We indicate whether they support parallel execution, type of instrumentation, performance overhead, kind of errors detected, and the oracle used for the analysis. Here, IL indicates that the tool provides debugging information at an instruction level, which is useful for debugging. FL represents information being presented at the granularity of a function.

Tool Name	Parallel Shadow Analysis	Instrumentation	Overhead	Rounding Errors	Branch Flips	Conversion Errors	Root Cause Analysis	Oracle
BZ [1]	✗	Compiler(GIMPLE IR)	7.91×	✓	✓	✓	✗	Canceled Bits
FPSpy [15]	✗	Binary(Valgrind)	127×	✓	✗	✗	✗	Condition Codes
Verrou [16]	✗	Binary(Valgrind)	7×	✓	✗	✗	✓(FL)	Randomized Rounding
FPDebug [3]	✗	Binary(Valgrind)	> 395×	✓	✗	✓	✓(IL)	MPFR
Herbgrind [38]	✗	Binary(Valgrind)	> 574×	✓	✓	✓	✓(IL)	MPFR
FPSanitizer [10]	✗	Compiler(LLVM IR)	> 100×	✓	✓	✓	✓(IL)	MPFR
PFPSanitizer	✓	**Compiler(LLVM IR)**	5.6×	✓	✓	✓	**✓(IL)**	**MPFR**

all 1's, then it represents normal values. The value represented is $(-1)^s \times 1.F \times 2^{E-bias}$.

Rounding error. When a real number is not exactly representable, then it must be rounded to the nearest FP number according to the rounding mode. The IEEE-754 has multiple rounding modes. Round-to-nearest-with-ties-to-even is the default rounding mode [18]. Hence, each primitive arithmetic operation has some error, which is bounded by 0.5 ULP (units in the last place) [18, 30].

Accumulation of errors. Although each primitive arithmetic operation has a small amount of rounding error ($\leq$ 0.5 ULP), the error can get amplified with a sequence of operations and produce wrong results, exceptions, and branch divergences. Figure 2 shows the accumulation of error while computing the expression $(A - ((A + B) - B)) * C$. An execution with reals produces zero but the FP execution produces a non-zero value. When this value is used as a branch condition, then the outcome of the branch will diverge from the ideal execution. One reason for this error is that when two numbers that are close to each to other are subtracted, the most significant precision bits can get canceled. If the remaining bits are influenced by rounding, then the rounding error gets amplified.

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 [3, 9, 38] 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. 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 [9, 38].

Comparison of prior approaches with PFPSanitizer. Table 1 compares various dynamic analysis tools to detect FP errors. Many of the prior approaches do not use the real execution as an oracle because it is expensive [1, 15, 16]. Hence, they detect likely errors rather than actual errors. Among the shadow execution approaches that use real numbers as an oracle, Herbgrind [38] and FPDebug [3] perform binary instrumentation and have significant overheads. Our prior work, FPSanitizer [9], reduces the overhead by keeping the memory usage bounded. In contrast to prior approaches, PFPSanitizer performs parallel shadow execution that reduces overheads by an order of magnitude while providing comprehensive detection and debugging support.

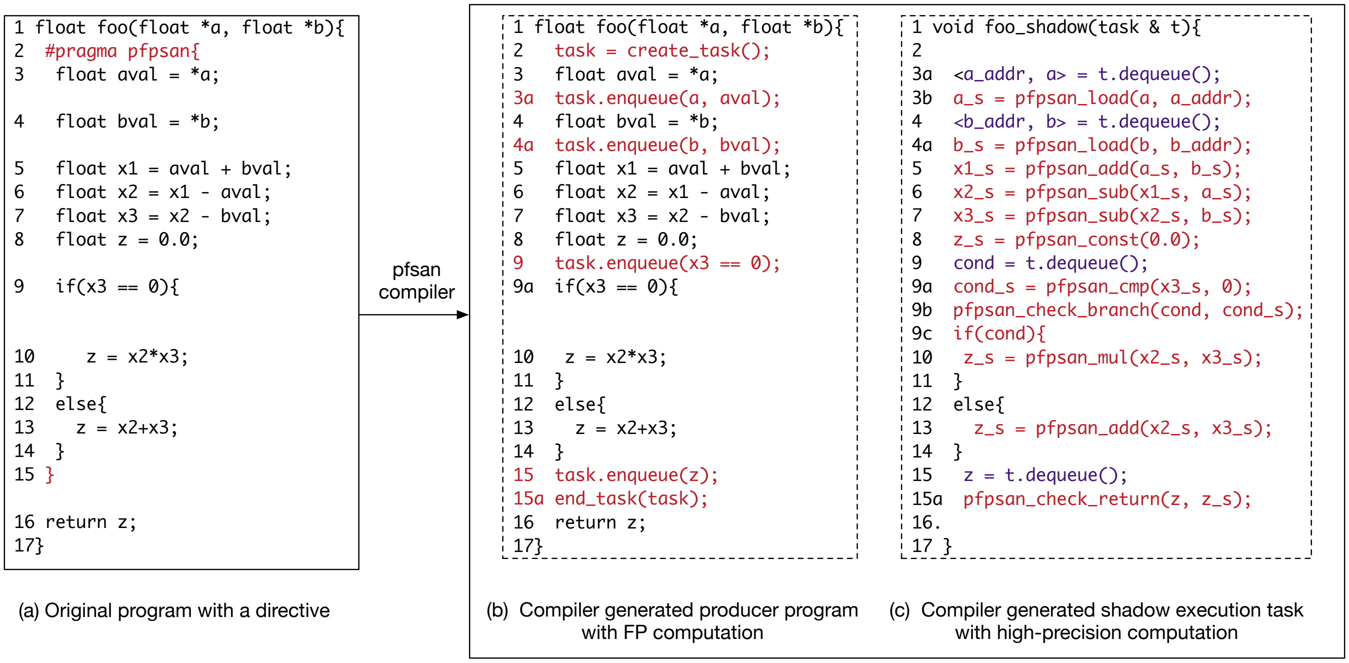
3 PARALLEL SHADOW EXECUTION

Our objective is to facilitate detection and debugging of numerical errors by performing a fine-grained comparison of a program's execution with a parallel shadow execution carried out with higher-precision. In contrast to approaches that detect specific errors [1, 15, 16], a shadow execution with higher precision can enable comprehensive detection and debugging of errors such as cancellation, exceptions, precision loss, and control-flow divergences. Typically FP data types such as `float` and `double` are supported in hardware and the higher-precision execution is performed using a software library (*e.g.*, MPFR). Hence, inlined shadow execution will be significantly slower than the program's execution. To address these overheads, we propose to perform parallel shadow execution.

Performing shadow execution in parallel is challenging for the following four reasons. First, we have to create a higher-precision version of the program automatically, which checks its execution with the original program in a fine-grained manner. Second, significant communication between the original program and the shadow execution will reduce performance. Third, executing the entire shadow execution on another core (*i.e.*, a single core) will not provide significant speedup compared to inlined shadow execution because the shadow execution is significantly slower than the original program. Hence, we need a mechanism to identify and execute fragments of shadow execution in parallel (*i.e.*, parallelize the shadow execution). Finally, we need to initialize the memory state appropriately for each such parallel fragment of shadow execution.

3.1 High-Level Overview of PFPSanitizer

We propose a new approach for debugging numerical errors where the user specifies parts of the program that needs to be debugged. Our compiler automatically creates a high-precision version of



```
float foo(float *a, float *b){
 #pragma pfpsan{
   float aval = *a;

   float bval = *b;

   float x1 = aval + bval;
   float x2 = x1 - aval;
   float x3 = x2 - bval;
   float z = 0.0;

   if(x3 == 0){


      z = x2*x3;
   }
   else{
     z = x2+x3;
   }
 }

 return z;
}
```

(a) Original program with a directive

```
float foo(float *a, float *b){
  task = create_task();
  float aval = *a;
  task.enqueue(a, aval);
  float bval = *b;
  task.enqueue(b, bval);
  float x1 = aval + bval;
  float x2 = x1 - aval;
  float x3 = x2 - bval;
  float z = 0.0;
  task.enqueue(x3 == 0);
  if(x3 == 0){


    z = x2*x3;
  }
  else{
    z = x2+x3;
  }
  task.enqueue(z);
  end_task(task);
  return z;
}
```

(b) Compiler generated producer program with FP computation

```
void foo_shadow(task & t){

  <a_addr, a> = t.dequeue();
  a_s = pfpsan_load(a, a_addr);
  <b_addr, b> = t.dequeue();
  b_s = pfpsan_load(b, b_addr);
  x1_s = pfpsan_add(a_s, b_s);
  x2_s = pfpsan_sub(x1_s, a_s);
  x3_s = pfpsan_sub(x2_s, b_s);
  z_s = pfpsan_const(0.0);
  cond = t.dequeue();
  cond_s = pfpsan_cmp(x3_s, 0);
  pfpsan_check_branch(cond, cond_s);
  if(cond){
   z_s = pfpsan_mul(x2_s, x3_s);
  }
  else{
    z_s = pfpsan_add(x2_s, x3_s);
  }
   z = t.dequeue();
   pfpsan_check_return(z, z_s);
.
}
```

(c) Compiler generated shadow execution task with high-precision computation

Figure 3: 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`)

the original program corresponding to it, which we call as the shadow execution task. There are two requirements for the shadow execution: (1) it has to follow the same control-flow path as the original program and (2) we should be able to check the shadow execution's value with the original program's value at various points of interest. Our goal is to execute multiple shadow execution tasks in parallel. However, these shadow execution tasks are derived from a sequential program and they are dependent on each other. To execute them in parallel, we need to break dependencies between these shadow execution tasks.

To break dependencies between two shadow execution tasks, we need to provide appropriate state for memory locations that depend on values produced by prior shadow tasks. Our key insight is to use the FP values from the original program as the oracle to break dependencies. In our model, we treat a shadow execution task to be independent of other shadow tasks and use the values produced by the corresponding regions of the original program to initialize the memory state. Hence, our compiler introduces additional instrumentation to the original program to provide live FP values, addresses of memory accesses, FP values read from each memory access, and outcomes of branches to the queue. Figure 3(b) presents the instrumented version of the original program. The shadow execution tasks created by our compiler read FP values and addresses from the queue. It executes the FP computation with higher precision. To ensure that the shadow execution task follows the same control-flow path as the original program, PFPSanitizer's compiler changes every branch in the shadow task to use the outcome of the original program. Figure 3(c) illustrates the shadow execution task created by our compiler for the program in Figure 3(a).

The shadow task maps every memory location with an FP value in the original program to a shadow memory location that has a high-precision value. When the shadow task executes a memory access, it needs to determine whether that location has been previously accessed by it. If it has previously accessed the memory location, then the high-precision value is available in shadow memory. Otherwise, it needs to initialize the shadow memory with the FP value from the program. Hence, every shadow memory location maintains the high-precision value and the FP value produced by the original program. When the shadow execution task performs a load, we check if the FP value loaded from shadow memory and the FP value produced by the original program are identical. If they match, we use the high-precision value for shadow execution. Otherwise, we use the FP value produced by the program to reinitialize shadow memory for that location. This technique to use the FP value from the original program as the oracle enables us to perform parallel shadow execution from a sequential program. It limits the detection of errors to instructions in the region provided by the programmer, which we found to be sufficient to debug various FP errors. To assist debugging, each shadow memory location maintains information about the operations that produced the value. This information enables PFPSanitizer to detect errors and provide a DAG of instructions for those errors.

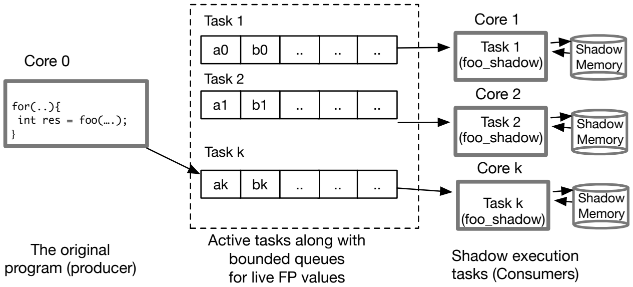


Figure 4: Parallel execution of shadow execution tasks during dynamic execution on a multicore machine. The producer (original program) and the consumer communicate live FP values, addresses of memory accesses, and branch outcomes using queues.

PFPSanitizer's runtime maps a shadow execution task to one of the cores in the system dynamically balancing the load. The original program and the shadow execution task operate in a decoupled fashion and communicate only through the queues (see Figure 4). This decoupled execution with dynamic load balancing provides significant speedups with the increase in the number of cores.

3.2 Our Model for Debugging FP Errors

As our goal is to enable programmers to debug numerical errors in long running programs, performing an expensive shadow execution for the entire program may not be feasible. In our approach, the programmer marks parts of the program that needs to be debugged with the `pfpsan` directive (*i.e.*, `pragma pfpsan` in Figure 3(a)). Each `pfpsan` directive 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. We do 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.

Two non-nested directives in the dynamic execution result in two shadow execution tasks that can execute in parallel. If the programmer 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× over inlined shadow execution. As the programmer introduces more directives, more shadow execution tasks can be executed in parallel. With the introduction of additional non-nested directives, the window of instructions tracked to debug an error decreases. Numerical errors have a relatively small window of dynamic instructions that are useful to debug and fix the error [9, 38]. Hence, when the programmer uses a sufficient number of directives, the programmer can obtain sufficient speedup and relatively rich DAG of instructions to debug the error using our approach.

3.3 Compiler Generated Shadow Tasks

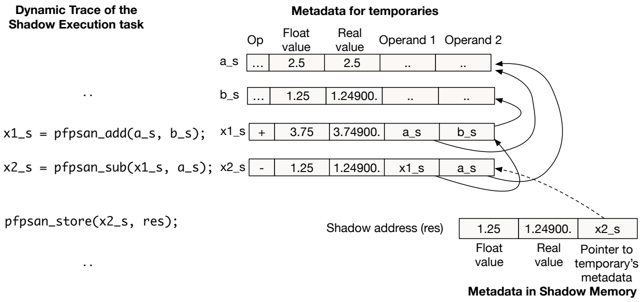
Given a program with directives, PFPSanitizer's compiler automatically creates a shadow higher-precision version of the program that can operate in parallel with the original program on a separate core. We want the original program and the shadow task to execute independently with minimum communication. In our design, they communicate through bounded queues. The original program is the producer and the shadow execution task is the consumer. To enable effective debugging, we need to ensure that the shadow task follows the same control-flow path as the original program. PFPSanitizer's compiler identifies the `pfpsan` directive and creates the modified original program and the shadow execution task corresponding to the directive. Figure 3(b) shows the modified original program and the shadow execution task corresponding to the directive.

Modified original program. PFPSanitizer's compiler modifies the original program to account for the creation of the shadow task. It adds a call to the runtime to obtain an unique task identifier and a queue associated with it. Subsequently, PFPSanitizer captures the FP values that are live to the directive and introduces enqueue operations. Our shadow tasks do not have information about integer operations. Hence, PFPSanitizer enqueues the address and the FP value loaded/stored for every memory operation. To provide information about the branch conditions, the compiler also enqueues the branch condition. At the end of the scoped block corresponding to the directive, the compiler adds a runtime call to indicate the end of the shadow task.

Shadow execution task. PFPSanitizer creates a shadow execution task that performs the higher-precision execution to facilitate detection and debugging of FP errors. To improve performance, PFPSanitizer does not create a high-precision replica of the entire scoped block indicated by the directive. Further, rewriting an entire program especially global data structures with indirect references is a challenging task. Instead, PFPSanitizer's compiler removes all non-FP operations (except the branch conditions), changes FP arithmetic operations to use the corresponding higher-precision operations in the MPFR library, and replaces FP load and store operations with loads and stores of MPFR data type in shadow memory.

For each live FP value in the directive, PFPSanitizer introduces a dequeue operation to read the input FP value from the queue associated with the shadow task. All FP arithmetic operations use the corresponding high-precision values. For example, `pfpsan_add` performs high-precision arithmetic using the MPFR library. For each memory operation (*i.e.*, a load or a store), PFPSanitizer's compiler inserts a runtime call in the shadow execution task to access the shadow memory corresponding to the address of the memory operation. The FP value produced by the original program is maintained as metadata in shadow memory along with the high-precision value. It enables us to start shadow execution at an arbitrary point in the program by using the original program as the oracle.

The shadow execution task does not perform any integer operations. As 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 3(c) presents the shadow execution task created by the PFPSanitizer compiler for the program in Figure 3(a). Overall, PFPSanitizer's compiler generates a high-precision version of the program that


Dynamic Trace of the Shadow Execution task
Metadata for temporaries
Metadata in Shadow Memory

```
..
x1_s = pfpsan_add(a_s, b_s);
x2_s = pfpsan_sub(x1_s, a_s);

pfpsan_store(x2_s, res);

..
```

	Op	Float value	Real value	Operand 1	Operand 2
a_s	...	2.5	2.5	..	..
b_s	...	1.25	1.24900.	..	..
x1_s	+	3.75	3.74900.	a_s	b_s
x2_s	-	1.25	1.24900.	x1_s	a_s

	Float value	Real value	Pointer to temporary's metadata
Shadow address (res)	1.25	1.24900.	x2_s

Figure 5: 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. 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.

executes FP operations with a MPFR type and will follow the exact same control-flow path as the original program during execution.

3.4 Dynamic Execution of Shadow Tasks

PFPSanitizer's runtime creates a pool of threads at the start of the producer's execution, which execute the shadow execution tasks. As the producer executes, the instrumentation corresponding to the directives create tasks and their associated queues. 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 [35]. As the original program uses `float` or `double` types that have hardware support, it is substantially faster than the software MPFR library. Hence, there are sufficient shadow execution tasks for the pool of threads to execute. To keep the resource (memory) usage bounded, there are fixed number of entries in the task queue. If the producer (*i.e.*, original program) creates more tasks than the size of the task queue, then the producer stalls until there is space in the queue. 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.

Metadata to detect and debug errors. The shadow execution task has all the live FP values from the queue and the runtime calls introduced by the compiler performs high-precision execution using real numbers (*i.e.* the MPFR data type). The shadow execution task stores high-precision values in shadow memory (*i.e.*, an address mapped to the original address produced by the program). The shadow memory of two different tasks are completely isolated from each other.

To detect errors in the FP program compared to an oracle execution with real numbers (*i.e.*, the MPFR data type), we maintain the real value with each temporary and each memory location. The temporaries are typically register allocated or allocated on the stack. The metadata for temporaries also maintains information about the operation and the pointers to the metadata of the operands of the instruction. For every memory location, we maintain the real value and the pointer to the metadata of the temporary that previously wrote to that memory location. Figure 5 shows the metadata maintained with each temporary. This metadata about operands in temporaries enables us to construct the DAG on an error (*i.e.*, backward slice responsible for the error). Figure 5 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 [9].

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.

Shadow execution from an arbitrary memory state. As we are creating a parallel shadow execution from a sequential program, we need to provide appropriate memory state for the shadow execution tasks. Our key insight is to use FP values from the original program (the producer) as the oracle 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, we can execute the shadow execution task even when prior shadow execution tasks have not completed 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 5. To enable the shadow execution task to start from an arbitrary memory state, PFPSanitizer takes the following actions whenever the task wants to perform a memory (read) access. First, the shadow task retrieves the address of the memory operation and the FP value produced by the producer from the queue. Second, it accesses the shadow memory location corresponding to the address provided by the producer. Third, it checks if the FP value in the metadata is exactly equal to the FP value from the producer. If so, 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 6 illustrates our approach to start shadow execution from arbitrary memory state with this technique.

Detecting errors. To detect FP errors, PFPSanitizer's runtime needs to convert the MPFR value in the shadow task to a double value and compare it to the double value generated by the producer. If the error exceeds some threshold, then 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

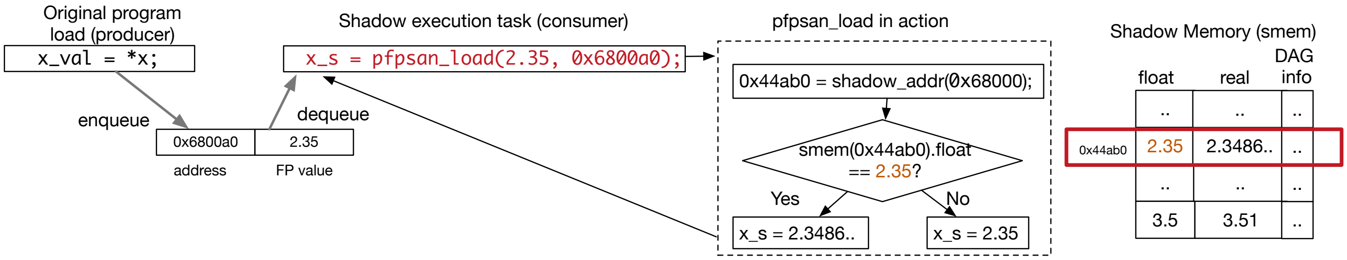


Figure 6: Our approach to execute shadow task from an arbitrary memory state. PFPSanitizer maintains the FP value in shadow memory and checks if the program's FP value is exactly equal to the value in shadow memory. If so, it uses the real value for subsequent shadow execution. Otherwise, it uses the 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.

program and the high-precision execution enables comprehensive detection of numerical errors.

4 IMPLEMENTATION CONSIDERATIONS

PFPSanitizer enhances the LLVM compiler to add instrumentation to the original program and to create shadow execution tasks. PFPSanitizer's runtime is in C++, which is 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 is equal to the number of cores in the system to avoid unnecessary context switches. We describe important implementation decisions in building the PFPSanitizer prototype.

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). On 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.

Management of temporary metadata space. PFPSanitizer maintains metadata with each temporary in the LLVM intermediate representation. PFPSanitizer uses a separate bounded space to maintain temporary metadata. When this space is completely utilized, PFPSanitizer automatically reclaims the space allocated to oldest entry in this metadata space. 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 [9]. Rather than maintaining unique metadata entry for each dynamic instruction, PFPSanitizer maintains a unique entry for each static instruction. As a result, PFPSanitizer produces DAGs that are restricted to the last iteration in a program with loops.

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 in the shadow execution task with their corresponding shadow functions. To handle indirect functions (*i.e.*, calls through a function pointer), PFPSanitizer's compiler introduces a call to the runtime in the shadow execution task 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.

Support for multithreaded applications. Although we describe our approach 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 there is at least one core unused by the original multithreaded application.

Usage with interactive debuggers. PFPSanitizer supports debugging with interactive debuggers like gdb. To enable such debugging, we propagate debugging symbols from the original program to the shadow execution task. Hence, the developer can insert breakpoints/watchpoints on functions in the shadow execution task. The backward slice of the instructions with the DAG and the detection enabled us to find and debug errors with the Cholesky application.

5 EXPERIMENTAL EVALUATION

This section briefly describes our prototype, methodology, and performance evaluation.

5.1 Prototype and Experimental Methodology

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 [17]. 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 [11].

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 application 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 [9, 38]. 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 [9, 10] , 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 [9, 38]. If the exponent of the two such values differ, then all the precision bits are in error. If all the bits differ, then entire double is influenced by rounding error.

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× 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 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.

5.2 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.

Table 2: Table reports the various kinds of bugs that we found in the programs used for our performance experiments. We report the number of static instructions that experience branch divergences in the second column, the number of instances where all bits are wrong in the third column (*i.e.*, sign, exponent, and precision bits), number of instances where all the precision bits are wrong in the fourth column (*i.e.*, 52 bits of precision with double), and number of instances where than more than fifty percent of the precision bits are wrong (fifth column). The last column provides the number of non-comment and non-blank lines of code in the application.

Name	Branch Flips	All Bits Wrong	All Precision Bits Wrong	50% Precision Bits Wrong	# Lines
art	2	0	0	0	1070
ammp	0	0	0	0	9791
equake	0	0	310	605	1125
lbm	4	0	609	1197	721
milc	1	0	378	607	8568
sphinx	0	0	2	56	11029
amg	0	0	4	14	58679
milcmk	0	0	0	0	88064

Table 2 provides information on the errors that we detected in applications from SPEC and CORAL, which have several thousand lines of code. As these applications are well tested for exceptions, we did not find exceptional conditions such as NaNs/infinities and instances where all bits are wrong. However, we detected branch divergences in the SPEC application `art`. When we investigated the root cause of these branch divergences, we identified the FP equality comparison as the culprit. Similarly, we observed many instances of precision loss where all precision bits are wrong (52 bits in a double) or more than 50% of the precision bits are wrong. Our investigation indicates that these are real divergences from an oracle execution with reals. These bugs need to be validated by the developer of these applications.

5.3 Debugging an Error in Cholesky

We discovered an error in the Cholesky decomposition program from the Polybench benchmark suite. PFPSanitizer detected infinities and NaNs (*i.e.*, exceptional conditions) at various places in the application. The program's code or documentation did not provide the reason for the exception. We describe how PFPSanitizer was helpful in both detecting and debugging the root cause of this error.

Cholesky decomposition [21] 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 $L \times L^T = A$,

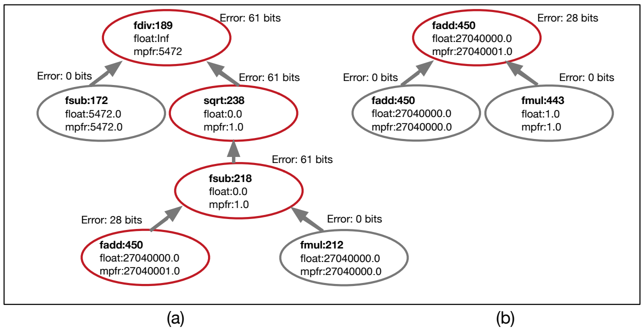


Figure 7: A DAG of instructions generated by PFPSanitizer while debugging the error in Cholesky. Each node shows the opcode, instruction id, computed value, real value and the numerical error occurred. (a) The DAG for the `fdiv` instruction that results in infinities (inf). (b) The DAG for the `fadd` instruction that is the root cause of the error.

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

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

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.

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 5200.0 & 1.0 & 0.0 \\ 0.0 & 5472.0 & 1.0 \end{bmatrix} \quad (2)$$

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

$$\begin{bmatrix} 1.0 & 5200.0 & 0.0 \\ 5200.0 & \mathbf{27040001.0} & 5472.0 \\ 0.0 & 5472.0 & 29942785.0 \end{bmatrix} \quad (3)$$

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

$$\begin{bmatrix} 1.0 & 5200.0 & 0.0 \\ 5200.0 & \mathbf{27040000.0} & 5472.0 \\ 0.0 & 5472.0 & 29942785.0 \end{bmatrix} \quad (4)$$

Specifically, $A[1][1]$ when computed with real numbers 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 where as 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 breakpoint was triggered with a `fdiv` instruction. We generated the DAG in the debugger. Figure 7 provides the DAG, where each nodes provides the instruction (instruction opcode:instruction id) and number of bits of error with it. Figure 7(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 7(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.

5.4 Performance Evaluation of PFPSanitizer

Figure 8 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× speedup over FPSanitizer with 64 cores. PFPSanitizer provides speedups of 3.0×, 7.0×, 14.3×, and 25.8× 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 9 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×, 6.2×, 7.5×, and 10.9× 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× on average with 512 bits of precision over the same baseline with these applications. This order of magnitude decrease in slowdown 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× over the baseline. Second, accesses to shadow memory and the queues by the consumer task introduces an additional overhead of 3×. Third, the high precision computation with the MPFR library introduces additional 1.5× overhead on average. All these overheads together add up to 7.5× slowdown

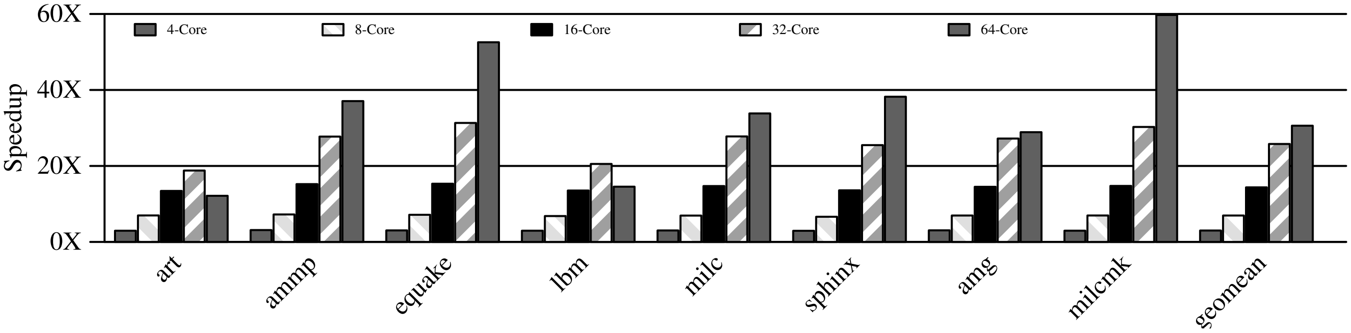


Figure 8: This graph reports the speedup of PFPSanitizer over FPSanitizer when the program is executed with 4 cores, 8 cores, 16 cores, 32 cores, and 64 cores, respectively. As these report speedups, higher bars are better.

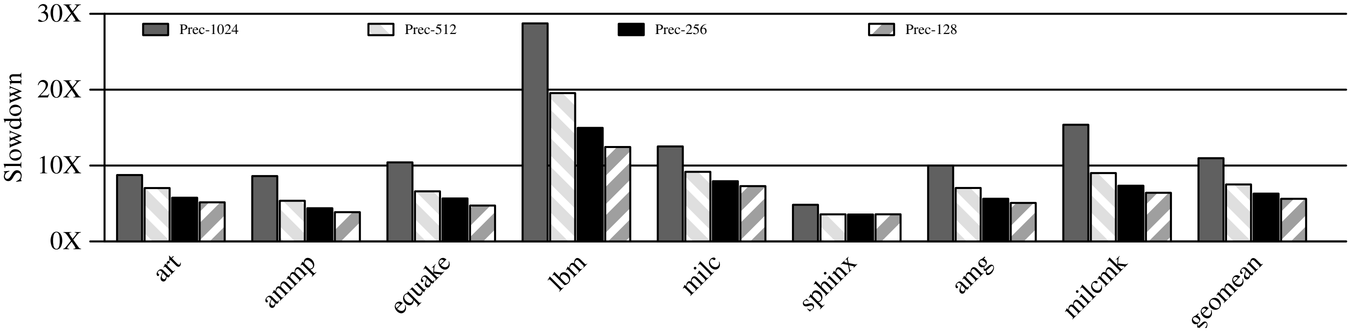


Figure 9: This graph reports the slowdown experienced due to parallel shadow execution with PFPSanitizer when compared to a baseline without any instrumentation. We report the slowdowns when we vary the number of bits used for the precision in the MPFR data type: 1024 (Prec-1024), 512 (Prec-512), 256 (Prec-256), and 128 (Prec-128) bits of precision.

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.

6 RELATED WORK

The related work can be broadly classified into two main categories: (a) those that detect numerical errors and (2) those that accelerate dynamic analysis with parallelism.

Detecting numerical errors. There is a large body of work on detecting numerical errors using both static analysis and dynamic analysis. Static analysis techniques [2, 12–14, 19, 39] use abstract interpretation or interval arithmetic to reason about numerical errors for all inputs. Error bounds for all inputs from static analysis is appealing. However, the bounds can be too large especially in the presence of loops, function calls, and pointer-intensive programs.

Dynamic analysis for detecting and debugging numerical errors. Dynamic analyses focus on the program's behavior for a given input. They can be classified into approaches that target a specific class of errors [1, 16, 22, 24] and those that are comprehensive detectors [3, 9, 38]. Any such 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 [3], Herbgrind [38], and FPSanitizer [9] are examples of approaches with inlined shadow execution. FPDebug and Herbgrind perform shadow execution with dynamic binary instrumentation using Valgrind [32], which introduces significant overheads. FPSanitizer addresses this issue with a LLVM IR based instrumentation. Among these approaches, FPDebug does not provide additional information for debugging errors. Herbgrind and FPSanitizer provide DAGs to debug errors. Such DAGs can be used with tools like Herbie [34] 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× or more) makes it challenging for debugging. Further, expert-crafted code (*e.g.*, error free transformations [31]) is a challenge for these approaches as they can report spurious errors with them. PSO [42] tackles this problem by building heuristics to detect such instructions and assist tools to avoid such scenarios. Our approach is inspired by FPSanitizer but reduces the performance overheads significantly with parallel execution so that the shadow execution can be performed with long-running applications.

Instead of using the MPFR library, real arithmetic has also been approximated with constructive reals [4, 5, 23]. However, they will likely be as slow as the MPFR library. To address the issue of slow oracles, BZ [1] monitors just the exponent of the operands and the result of the FP computation. 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 [24] uses similar approximation, computes the impact of such likely errors on the final output of the program, and uses vectorization to reduce performance overheads. FPSpy [15] 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. To check the sensitivity of the program to the rounding mode, another approach is to perform dynamic analysis with random rounding. CADNA [22] and Verrou [16] use random rounding. Similarly, condition number of individual operations can be used to detect numerical errors and instability in FP applications [45]. In contrast to these approaches that detect likely errors, our approach has similar or lower performance overheads when compared to them while providing comprehensive detection and debugging support using shadow execution with real numbers.

Precision tuning to reduce errors. One way to avoid common numerical errors is to select 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 [7, 37].

Identifying inputs with high FP error. Dynamic analyses need inputs that exercise operations with FP error. Hence, prior research has explored symbolic execution, forward and backward error analysis, and random input generation to generate such inputs [8, 20, 36]. Techniques to generate inputs complement our approach and can enable us to detect and debug FP errors efficiently as we illustrated with our Cholesky case study.

Parallel dynamic analysis. In the context of dynamic analysis for detecting memory safety errors and race detection, numerous parallel analysis techniques have been explored [6, 40, 43]. Approaches that perform fine grained monitoring use hardware support as with a dedicated operand queue in Log-Based Architecture (LBA) [6]. Further, dataflow analyses have been modified to accelerate dynamic analyses with LBA [41]. Other approaches for parallel data race detection and deterministic execution monitor programs at the granularity of epochs [40]. The closest related work is Cruiser [43], 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 heavy-weight dynamic analysis with support for debugging.

7 CONCLUSION

This paper advances the state-of-the-art in debugging numerical errors by performing shadow execution with higher precision. To enable the use of such shadow execution with long-running applications, we perform shadow execution in parallel. As we are creating parallel execution from a sequential program, we need to provide appropriate memory state for shadow execution. Our key insight is to use the FP values from the original program as the oracle for initializing memory state. The resulting tool is an order of magnitude faster than existing shadow execution tools. We believe comprehensive detection with these overheads can enable their usage in late stages of development and debugging. Our experience suggests that other dynamic analyses (*e.g.*, race detectors) can also benefit from this approach, which we plan to explore in future work.

ACKNOWLEDGMENTS

We thank the members of the Rutgers Architecture and Programming Languages group for their feedback. This material is based upon work supported in part by the National Science Foundation under Grant No. 1908798, Grant No. 1917897, and Grant No. 2110861. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 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

[2] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-point Exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 549–560. https://doi.org/10.1145/2480359.2429133

[3] 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

[4] Hans-J. Boehm. 2005. The constructive reals as a Java library. In *The Journal of Logic and Algebraic Programming*, Vol. 64. 3–11. https://doi.org/10.1016/j.jlap.2004.07.002

[5] 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. https://doi.org/10.1145/319838.319860

[6] 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

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

[8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-point Errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/2555243.2555265

[9] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*

(London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 731–746. https://doi.org/10.1145/3385412.3386004

[10] 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

[11] 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

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

[13] Marc Daumas and Guillaume Melquiond. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Softw.* 37, 1, Article 2 (Jan. 2010), 20 pages. https://doi.org/10.1145/1644001.1644003

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

[15] 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

[16] 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.

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

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

[19] 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

[20] 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. https://doi.org/10.1145/3377811.3380359

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

[22] 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

[23] 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. https://doi.org/10.1145/93548.9355

[24] 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. https://doi.org/10.1145/2814270.2814299

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

[26] 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

[27] 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

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

[29] Jay P. Lim, Matan Shachnai, and Santosh Nagarakatte. 2020. Approximating Trigonometric Functions for Posits Using the CORDIC Method. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (Catania, Sicily, Italy) *(CF '20)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3387902.3392632

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

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

[32] 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)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

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

[34] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

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

[36] 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

[37] 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. https://doi.org/10.1145/2503210.2503296

[38] 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/10.1145/3192366.3192411

[39] 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/10.1007/978-3-319-19249-9_33

[40] 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

[41] 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

[42] 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. https://doi.org/10.1145/2950290.2950355

[43] 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

[44] Craig Zilles and Gurindar Sohi. 2002. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture* (Istanbul, Turkey) *(MICRO 35)*. IEEE Computer Society Press, Washington, DC, USA, 85–96. https://doi.org/10.1109/MICRO.2002.1176241

[45] 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