Using Program Analysis and Testing to Facilitate Debugging and Optimization of Scientific Applications

By

DOLORES MIAO
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Cindy Rubio-González, Chair

Ignacio Laguna

Aditya V. Thakur

Committee in Charge

2026

To Evelyn Martin and 17β-estradiol.

Abstract

Scientific applications rely heavily on floating-point computations to model complex phenomena in physics, chemistry, biology, and other scientific domains. However, floating-point arithmetic is inherently imprecise and can lead to numerical inconsistencies that affect the correctness and reliability of scientific software. Additionally, optimizing these applications to efficiently utilize modern high-performance computing resources is a significant challenge.

This dissertation addresses these challenges through four complementary approaches using program analysis and testing techniques to facilitate debugging and optimization of scientific applications. First, we present Ciel, a tool that uses bisection search and precision enhancement for enhanced isolation of compiler-induced numerical inconsistencies in heterogeneous code, helping developers identify the expressions that cause unexpected numerical behavior. Evaluation demonstrated 99.4% precision across 330 GPU programs and real-world mini-apps. Second, we introduce CIGEN, a tool utilizing a multi-phased approach, combining sampling, input clustering, and function optimization techniques to find input ranges with compiler-induced numerical inconsistencies. On 175 GNU Scientific Library functions, CIGEN found 53.4% more functions with high inconsistencies than prior work. Third, we present FloatGuard, a dynamic testing approach for detecting floating-point exceptions in AMD GPUs. FloatGuard successfully detected floating-point exceptions in 89.7% of evaluated programs (507 out of 565). Finally, we develop Muppet, a framework to optimize the performance of OpenMP applications via source-based code transformation and mutation testing. Muppet achieved performance improvements in 75.9% of evaluated programs with a maximum speedup of 3.57x.

Together, these contributions provide developers and researchers with practical tools to improve the reliability and performance of scientific applications. Our evaluation on real-world scientific codes demonstrates the effectiveness of these approaches in detecting, localizing, and mitigating numerical issues while maintaining computational efficiency.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my academic advisor, Professor Cindy Rubio-González, for her invaluable guidance, support, and mentorship throughout my graduate studies. Her expertise and dedication have been instrumental in shaping my research and academic growth.

I am also grateful to my dissertation committee members, Professor Aditya V. Thakur and Dr. Ignacio Laguna, for their insightful feedback and suggestions that have greatly improved this work. Also, many thanks to my collaborating researchers and co-authors—Dr. Giorgis Georgakoudis and Dr. Konstantinos Parasyris—for their collaboration and contributions to this work. I would also like to acknowledge the support of our national funding agencies, including the National Science Foundation (NSF) and the Department of Energy (DOE), for providing the financial resources that made this research possible.

I am thankful to my colleagues and friends in the Computer Science department at UC Davis for their camaraderie and support. Their encouragement has been a constant source of motivation. I would like to give special thanks to Dr. Jackson Vanover for the days we spent discussing research ideas and for the invaluable experience we had collaborating on projects before I became a PhD student. Beyond that, I am very grateful to have met so many wonderful people in our department during my PhD journey, including Yutong Wang, Vivian Li, and Kai Lan. Furthermore, I also thank the university and its student organizations for providing a vibrant community and resources that have enriched my experience.

Finally, I would like to thank my family for their unwavering support and encouragement throughout my academic journey. I thank my parents who, unlike most of their peers, always supported my decision to pursue higher education and research at the tender age

of mid 30s. I am grateful to my cousin, an alumna of UC Davis, for her constant support and encouragement during my rapidly changing life in this strange yet familiar land. Most importantly, my wife, Evelyn Martin, has always stood by my side and has been my greatest source of motivation. She even helped me with specific aspects of my research using her expertise in entertainment software development.

Finally, I would like to give special attention to my feline companions, Cheese and Fluff. Their meowy presence and playful antics have provided me with comfort and joy during the challenging times of my PhD journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Scientific applications run on a variety of high-performance computing (HPC) systems to facilitate research in fields of theoretical and empirical sciences. In particular, scientific application development increasingly relies on heterogeneous computing, which uses a variety of processing cores, such as multicore CPUs and GPUs, to accelerate code execution [1]. Some prominent examples of such systems are modern exascale computing platforms from the US Department of Energy (DOE), such as El Capitan and Frontier, which include massive numbers of CPUs and GPUs to achieve unprecedented computational throughput via parallel processing. However, the use of heterogeneous computing paradigms introduces new challenges, both in maximizing the parallelism of systems that run these paradigms, and in ensuring the correctness of the resulting applications. These challenges are the focus of this dissertation.

1.1.1 Floating-Point Correctness Issues

An important aspect affecting the correctness of scientific applications is their use of floating-point computations. Floating-point arithmetic was invented around the same time as computers and is capable of representing a much wider range of values compared to fixed-point types with the same number of bits. Unfortunately, this characteristic of floating-point formats means that they are prone to errors. Floating-point arithmetic does not respect the laws

of association and may cause large errors from conditions such as catastrophic cancellation. Additionally, its behavior may differ due to differences in standard interpretations such as rounding modes and how to handle special values such as NaNs and infinities.

In particular, this dissertation addresses two specific types of floating-point correctness issues: **compiler-induced numerical inconsistencies**, where different compilers or optimization levels produce diverging numerical results, and **floating-point exceptions**, where operations encounter limits in the representable range of floating-point types, such as overflow or division by zero.

Compiler-Induced Numerical Inconsistencies

One issue stemming from the use of floating-point arithmetic is implementation-based: even though standards such as IEEE 754 define how floating-point types and arithmetic work, different hardware and compiler implementations may exhibit differences in terms of undefined behaviors. Compiler optimizations are often the first method software engineers consider when optimizing programs. Aggressive optimization options are also often invoked to push program performance as much as possible. Additionally, switching, upgrading, or adding support of compilers in the middle of a software project is a frequent industrial practice. Unfortunately, such modifications on a project global scale can have a negative impact on software reliability, particularly on floating-point arithmetic which could result in local errors that may propagate to the final program's output. Additionally, compiler flags that explicitly relax IEEE 754 compliance, such as `-ffast-math`, may further exacerbate these differences. When these differences manifest as diverging results in numerical programs, they are called **compiler-induced numerical inconsistencies**.

In cases found in the literature, these compiler-induced numerical inconsistencies have required significant effort and domain knowledge to isolate and fix. For example, a documented case [2] in the Laghos (LAGrangian High-Order Solver) application [3] at Lawrence Livermore National Laboratory occurred when the code was ported to the Sierra system using the IBM xlc compiler. This triggered an inconsistency in the energy computed by the application under `xlc -O3` but not with `xlc -O2`. In another documented case [4], the Community Earth System Model (CESM) failed its verification using the CESM-ECT quality assurance

framework when it was ported to the Mira machine at Argonne National Laboratory. Both cases required weeks to months for scientists and engineers to identify the source code that caused such failures.

Floating-Point Exceptions on GPUs

The implementation of floating-point arithmetic can differ across hardware and software platforms. In particular, GPU floating-point implementations may deviate slightly from strict IEEE 754 behavior, even when they nominally adhere to the standard. As a result, variations in precision, compiler optimizations, or handling of special cases such as floating-point exceptions (e.g., NaNs and infinities), can lead to discrepancies in results. Scientific applications rely on reproducibility across all architectures, including GPUs. If exceptions, such as NaNs, are not detected and handled, they can lead to inconsistent results across different runs of the same computation, undermining the credibility of the results. Different platforms may have different capabilities for detecting and handling floating-point exceptions, and thus it is crucial to have tools that can detect and analyze these exceptions on each platform.

1.1.2 Performance Optimization Challenges

Many scientific applications use a plethora of accelerator hardware such as GPUs and multi-CPU clusters, along with parallel programming paradigms such as OpenMP, CUDA, and MPI. It is important to ensure that hardware resources are sufficiently utilized by maximizing the parallelism of these scientific applications. However, many scientific application developers may not have the required knowledge to achieve optimal performance by following the best practices laid out in the latest versions of their respective parallel programming paradigms.

Most performance optimization techniques focus on highlighting performance hotspots in the program, but ultimately they rely on programmers to identify code modifications that fix a performance problem or improve overall performance. Compiler optimizations improve performance usually at the intermediate representation (IR) level; however, reasoning about correctness at the IR level is much more difficult than at the source level. As a result, compiler

optimizations can leave optimization opportunities on the table, and IR-level optimizations are not portable across compilers.

1.2 Dissertation Overview and Contributions

This dissertation addresses both correctness and performance challenges in scientific computing through program analysis and testing techniques. The first three technical chapters focus on debugging floating-point correctness issues, while the final technical chapter addresses performance optimization.

Chapter 2 presents CIEL (Compiler-induced Inconsistency Expression Locator), an approach and tool for isolating minimal code regions that cause compiler-induced numerical inconsistencies in heterogeneous programs at the *expression level*. Using hierarchical search and precision enhancement to pinpoint exact code expressions, CIEL provides a more efficient bisection search compared to the state of the art for CPU programs and offers finer search granularity than prior line-level approaches. The chapter includes motivating examples of compiler-induced inconsistencies, detailed description of the hierarchical search and precision enhancement approach, evaluation on synthetic and real-world GPU programs, and comparison with state-of-the-art CPU tools.

Chapter 3 presents CIGEN (a combination of Compiler-induced Inconsistencies and Input Generation), a tool that complements CIEL by finding *input ranges* that cause significant compiler-induced numerical inconsistencies in numerical programs. While CIEL answers "where in the code does the inconsistency occur?", CIGEN answers "which inputs trigger high inconsistency errors?" using a multi-phase approach combining input-partitioned and coverage-based random input sampling, input sample clustering, and differential evolution. The chapter covers the multi-phase approach in detail, presents evaluation on 175 GNU Scientific Library functions, and provides analysis of the characteristics of inconsistency-inducing input ranges.

Chapter 4 presents FLOATGUARD, the first tool to detect floating-point exceptions in HIP programs running on AMD GPUs. As HPC systems increasingly adopt AMD GPU architectures, this work addresses the critical need for AMD GPU numerical debugging by

utilizing hardware exception registers, assembly-level and source-level code instrumentation, and debugger-guided execution. The chapter provides background on AMD GPU architecture and exception handling, describes the design of the compiler wrapper and testing framework, and presents evaluation on benchmark and synthetic HIP programs.

After establishing correctness through these three chapters, Chapter 5 shifts focus to performance optimization. Chapter 5 presents MUPPET (Mutation-Utilized Parallel Performance Enhancement Tester), a mutation testing framework for optimizing OpenMP parallelism while maintaining correctness. MUPPET uses a multi-faceted approach combining source-level mutation testing with optimization algorithms such as delta debugging, Bayesian optimization, and decision tree optimization to automatically discover modifications to code constructs and OpenMP directives that provide performance speedups. By generating and evaluating source-level mutations applied to OpenMP directives, the tool helps developers reason about performance optimizations at the source level rather than at the intermediate representation or binary level. The chapter describes the mutation testing methodology for performance, the classes of OpenMP mutations, integration with various optimization algorithms, and comprehensive evaluation on benchmark programs and proxy applications.

Finally, Chapter 6 summarizes the contributions of this dissertation, discusses broader impact, and outlines future research directions.

Chapter 2

CIEL: Expression-level Isolation of Compiler-Induced Inconsistencies

2.1 Problem and Motivation

As discussed in Section 1.1, compiler-induced numerical inconsistencies can have significant impacts on scientific applications, requiring weeks to months of debugging effort. While prior work has made progress on isolating these inconsistencies, significant gaps remain for heterogeneous computing environments.

Limitations of Existing Approaches. FLiT [5] works at the function level, identifying which function in a program is affected by compiler-induced inconsistencies. pLiner [2] improves on this by isolating lines of code that cause inconsistencies. However, neither tool targets GPU code, and neither isolates at the expression level. Since program statements causing compiler-induced inconsistencies may include many floating-point operations involving different operators, variables, constants, or function calls, *isolating at the expression level* provides more precise insight into the inconsistencies, pointing users directly to their root cause and potential fix.

Challenges in Heterogeneous Programs. Heterogeneous programs that combine CPU and GPU code present unique challenges for inconsistency isolation: (1) GPU platforms lack floating-point arithmetic beyond double precision, requiring support for extended precision

libraries. (2) Built-in vector arithmetic on GPUs must be transformed to enhanced precision during isolation. (3) Code written for different target platforms (CPU or GPU) requires platform-specific transformations.

Our Approach. We present CIEL (Compiler-induced Inconsistency Expression Locator), the first tool that automatically isolates numerical inconsistencies in heterogeneous programs at the expression level. CIEL traverses the abstract syntax tree (AST) of each function and performs a bisection search for all adjacent sibling code blocks, maintaining their adjacency relationship. During precision enhancement, adjacent code blocks are either combined into a single code region or have variable checkpoints where redundant type conversions are removed. CIEL *is the first tool capable of isolating code regions in heterogeneous computing programs that, combined with compiler optimizations, produce inconsistent numerical results.*

Contributions. The contributions of this chapter are:

- An approach for isolating minimal code regions that cause compiler-induced numerical inconsistencies in heterogeneous programs, using a more efficient bisection search that operates on simplified ASTs and provides finer search granularity at the expression level (Sections 2.3.1 and 2.3.2).
- A precision enhancement strategy that more accurately reflects the resolvability of inconsistencies under precision enhancement, and addresses challenges specific to transforming heterogeneous code to higher precision (Section 2.3.3).
- An implementation of our approach in the tool CIEL, and an evaluation showing efficacy at isolating inconsistencies in 330 synthetic GPU programs, NAS and Rodinia GPU benchmarks, and the real-world mini-app CLOUDSC (Section 2.4.1), with higher precision and efficiency compared to the state of the art (Section 2.4.2).

2.2 Background

2.2.1 IEEE 754 Floating-Point Representation

IEEE floating-point types are represented by three fixed-point components: the sign bit, s, where 0 indicates a positive number and 1 indicates a negative number; the exponent

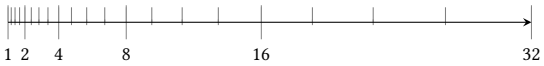


Figure 2.1: Density of floating-point numbers between 1-32.

e, which determines the exponential range of the floating-point value; and the mantissa m, which specifies the position of the value within the range defined by e. The interpretation of m differs depending on whether e is zero or non-zero; when e is zero, the floating-point number is within the *subnormal* range, which starts from zero, so that the floating-point type can represent numbers that are even closer to zero. The IEEE 754 standard also defines special values, such as infinity (Inf) and not-a-number (NaN), when e has all of its bits set to 1. These values correspond to situations where the limit of representing numbers is reached.

For example, a 64-bit double-precision floating-point number is represented by a 1-bit sign, 11-bit exponent, and 52-bit mantissa. The real value of a 64-bit floating-point number, using s, m, e, can be written as follows:

$$f_{64}(s,m,e) = \begin{cases} \text{Inf} & e = 2047,\ m = 0 \\ \text{NaN} & e = 2047,\ m > 0 \\ (1-2s)(1+m)2^{e-1023} & 0 < e < 2047 \\ (1-2s)m2^{e-1023} & e = 0 \end{cases} \tag{2.1}$$

An important property of a floating-point type is its logarithmic distribution. Figure 2.1 demonstrates that (a) small floating-point numbers are more dense than large numbers, and (b) the amount of unique floating-point numbers between two floating-point numbers is measured in logarithmic terms. These properties are important to consider when we design algorithms that operate on floating-point numbers.

2.2.2 Compiler Optimizations and IEEE 754 Compliance

Compilers for CPU and GPU code, such as GCC, Clang [6], and nvcc [7], offer various levels of optimization flags from `-O0` to `-O3`. With higher optimization levels, program performance is improved, sometimes significantly, but at the cost of potentially generating non-compliant IEEE 754-2008 floating-point code. There are optimization flags that explicitly violate the

IEEE 754-2008 standard, but in cases where precision is of a lesser concern, they could offer good speedups.

For example, the `-ffast-math` flag (or `-use_fast_math` in CUDA) enables a set of aggressive floating-point optimizations including:

- Assuming no NaN or infinity values will occur
- Allowing reassociation of floating-point operations
- Using reciprocal approximations instead of division
- Flushing subnormal numbers to zero

Since compilers have significant freedom when generating floating-point code—particularly when `-ffast-math` or other flags are used—operations can be re-ordered and/or replaced in different ways, which induce numerical inconsistencies. Such behavior is not limited to programs in native programming languages such as C/C++, and may in fact affect high-level languages and libraries that are built on top of them, as shown in [8].

Simply disabling compiler optimizations, or increasing precision uniformly across an application, may solve compiler-induced variability, but they are not practical solutions. Instead, developers strive to find the root cause of these compiler-induced inconsistencies and manually fix them to reduce their impact without disabling compiler optimizations. Currently, identifying the root cause of such issues in heterogeneous programs is a manual effort, requires domain knowledge, and is a time-consuming task.

Consider the CUDA version of the BT NAS program with input class S as a concrete example. Table 2.1 shows the program runtime and the maximum relative error for each compiler and optimization flag combination. Using `-O3 -use_fast_math` with nvcc yields 100% speedup compared to `-O0`, but at the cost of the error being 39% larger. Performance and error with Clang is generally worse, with the largest error in `clang -O3 -ffast-math` being 403% larger than `nvcc -O0`.

Table 2.1: Inconsistencies in BT.S.

Compiler Options	Runtime	Error
nvcc -O0	0.104s	6.98176E-13
nvcc -O3 -use_fast_math	0.052s	9.73738E-13
clang -O0	0.349s	8.32928E-13
clang -O3 -ffast-math	0.059s	3.50905E-12

Issues mentioned in the introduction chapter and above are bound to occur when real-world scientific applications are written or ported to new platforms. *Automatically resolving such issues without extensive domain knowledge would save a massive amount of time and increase programming productivity.*

2.3 Technical Approach

Problem Statement. Given *heterogeneous programs* with known compiler-induced numerical inconsistencies, a practical problem for software developers is how to isolate the *expressions* that cause such inconsistencies in a precise and efficient manner. CIEL is designed with the goal of tackling this problem. Specifically, CIEL takes as input a program P and its associated input, which under compilers $C_1, C_2, ..., C_n$ and optimization flags $O_{i1}, O_{i2}, ..., O_{ik}$ for each compiler C_i, produces inconsistent results. CIEL outputs the *minimal region* R_m in m searches, which means that by generating program variants $P'_1, P'_2, ..., P'_m$ it isolates the root cause of the inconsistency to a code region as narrow as possible. Below we present definitions that will be used throughout the rest of the chapter.

Definition 1. The output of program P given a specific error threshold ϵ under compiler C_i and optimization flags O_{ij} is written as $f(P, \epsilon, C_i, O_{ij})$.

Definition 2. Compiler-induced inconsistencies occur if there are two sets of compiler/optimization flag combinations C_i, O_{ij} and C_k, O_{kl}, where $f(P, \epsilon, C_i, O_{ij}) \neq f(P, \epsilon, C_k, O_{kl})$. When any two sets of combinations have the same output, the inconsistencies are considered to be resolved.

Definition 3. A region set R of a program P is defined as a set of regions in P, each of which is a straight-line code fragment with an entry point and one or more exit points.

Definition 4. A region set R_m is minimal if (a) the inconsistency is resolved when code in R_m is executed in higher precision, and (b) either R_m consists of only one expression, or leaving any expression in R_m in lower precision would result in unresolved inconsistencies.

CIEL's Workflow. The overall workflow of CIEL is illustrated in Figure 2.2. To find the minimal region that causes the numerical inconsistency, CIEL performs *hierarchical*

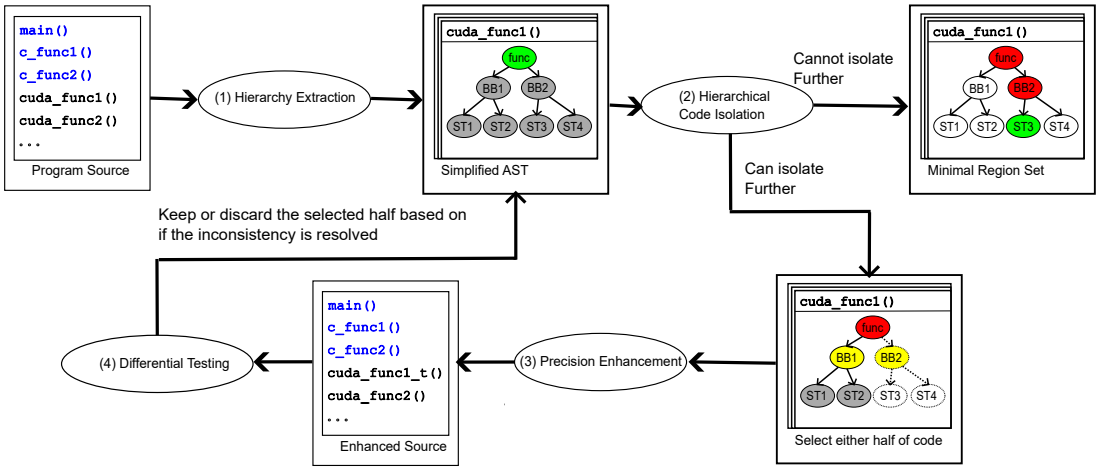


Figure 2.2: The workflow of CIEL.

bisection search on the source code—first between functions, then between code regions in the suspected functions. Each iteration increases the search granularity. The search algorithm identifies regions suspected of causing compiler-induced inconsistencies. For each region $R_1, R_2, ..., R_m$, CIEL then creates a mutated variant of the program $P'_1, P'_2, ..., P'_m$ for which code in the corresponding region is in enhanced precision. Whether the variant resolves these inconsistencies is then used to guide the further, narrower isolation of source code that triggers inconsistencies. The isolation process ends when region R_m satisfies the conditions of a minimal region. The modules in CIEL are described below:

(1) *Hierarchy Extraction* traverses the AST of the functions under analysis, extracting information relevant to floating-point operations, and generating a simplified AST for these functions. This is the entry point of the analysis.

(2) *Hierarchical Code Isolation* performs a hierarchical bisection search on the simplified AST to generate regions for subsequent precision enhancement.

(3) *Precision Enhancement* increases the precision of the code regions identified by hierarchical code isolation. The output is the transformed source code with the floating-point operations in specific code regions written in higher precision.

(4) *Differential Testing* compiles and runs the transformed program with specified compilers and optimization flags in parallel. The output of these combinations of compilers and flags is compared to determine if the compiler-induced inconsistencies are resolved.

The rest of the section describes modules 1-3 in more detail.

2.3.1 Hierarchy Extraction

The hierarchy extraction module traverses the program AST and extracts source code hierarchy information for each function in the form of a *simplified AST*. The simplified AST acts as a data exchange format between modules, and contains additional data specific to CIEL that includes whether a node should be enhanced in precision (enabled/disabled), and the list of all floating-point operations such as reads, writes, declarations, function calls, and constants in every statement under a node. The simplified AST classifies statement structure of a function into five node categories:

1. Each statement that ends with a semicolon (declaration, expression, and *return*/*break*) is a **statement node** on the simplified AST. Each statement node also contains its expression AST hierarchy.

2. A set of statements with only one entry point and one exit point is grouped as a **basic block (BB) node**.

3. For a selection statement such as *if-else* or *switch-case* statement, one BB node is assigned to each branch; and then a **conditional block node** is assigned for the whole selection statement as a code block.

4. For a loop statement such as a *for* or *do-while* statement, one BB node is assigned to the condition, another to the loop body, and then a **loop block node** is assigned for the whole loop statement as a code block.

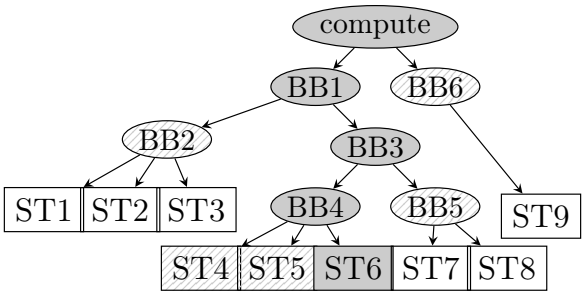
5. A **function node** is assigned to the whole function.

The relationship between a sample program and its simplified AST representation is shown in Figure 2.3. Each expression statement (for-loop header in Line 2; if condition in Line 6; statements in Lines 3, 4, 5, 7, 10) in Figure 2.3a has its own statement node, which is organized into block nodes. The corresponding simplified AST is shown in Figure 2.3b.[1]

[1]Statements and blocks with no floating-point operations are recorded but excluded from precision enhancement.

```
1 void compute(/*var args*/){
2   for(int i=0; i<n; ++i) { //ST1-3
3     comp = x-1.6f; //ST4
4     float t = +1.4697E36f; //ST5
5     comp += t+1.4E-41f; //ST6
6     if (comp < sinhf(y)) { //ST7
7       comp = tanf(z); //ST8
8     }
9   }
10  printf("%.17g",comp); //ST9
11 }
```

(a) Sample function `compute`. Variables `comp`, `x`, `y` and `z` are function arguments of type `float`.



(b) Simplified AST of `compute`. The inconsistency is in ST6. Dark filled nodes are hierarchically isolated. Stripe filled nodes are considered in the bisection search at each level of hierarchy, but not isolated. White nodes are not considered during bisection search.

Figure 2.3: Sample function and its simplified AST.

2.3.2 Hierarchical Code Isolation

When hierarchy extraction is complete, the simplified ASTs for all functions are output to the hierarchical code isolation module. As code isolation progresses, it marks nodes on the simplified ASTs as enabled or disabled depending on whether the node is still in consideration as a potential cause of inconsistencies.

Bisection search is the basis for the approach, followed by a 1-minimal check [9]. Bisection has shown to be an effective search strategy in the context of code isolation in CPU programs [2, 5]. CIEL bases its search algorithm on the same idea of partitioning the program into functions, code blocks, and statements, but improves on how the hierarchical search is performed to reduce search time and improve precision. In particular, CIEL proposes a refined hierarchical region isolation with the explicit goal of improving isolation accuracy by reducing unnecessary type conversions when enhancing precision. Furthermore, unlike previous work, CIEL explores expression-level granularity during the search.

CIEL isolates a minimal region of code amongst a set of code regions by recursively bisecting suspicious code regions into two halves and verifying if enhancing either half resolves the inconsistency (Line 1 in Algorithm 1). Hierarchical search first sets the whole program in enhanced precision (Line 17 in Algorithm 1), then finds the minimal region in increasing granularity, following two stages:

Algorithm 1: Hierarchical Code Isolation.

```
1  Function BisectionSearch(regions) :
2      if regions.size() > 1 then
3          regions_1, regions_2 = ArraySplit(regions, 2);
4          if HasResolvedInHighPrecision(regions_1) then
5              BisectionSearch(regions_1);
6          else if HasResolvedInHighPrecision(regions_2) then
7              BisectionSearch(regions_2);
8          else
9              BisectionSearch(regions_1);
10             BisectionSearch(regions_2);
11 Function RegionIsolation(regions) :
12     BisectionSearch(regions);
13     foreach region in regions do
14         if region.hasSubBlocks() && region.inHighPrecision() then
15             RegionIsolation(region.getSubBlocks());
16 Function FuncIsolation(Funcs ):
17     Funcs.setHighPrecision();
18     BisectionSearch(Funcs);
19     minFuncs = Funcs.getHighPrecisionFuncs();
20     foreach func in minFuncs do
21         RegionIsolation(func.getBlocks());
```

1. **Function Isolation.** During the function isolation stage (Line 16 in Algorithm 1), bisection search is performed at the function level, and the result is a minimal set of functions that cause the inconsistencies.

2. **Hierarchical Region Isolation.** For each function isolated in the first stage, during the hierarchical region isolation (Line 11 in Algorithm 1), bisection search is performed at increasingly granular levels, from code block level to statement level, and ultimately to expression level.

CIEL traverses the simplified AST of each function isolated in the function isolation stage, and isolates child nodes of the current node(s) that cause the compiler-induced inconsistency: from the child nodes of the function node, to child nodes of BB nodes, to all isolated statement nodes, until within the smallest subexpression in a statement node, e.g., a variable, constant, or function call. A difference of this code isolation method, compared to prior work, is that these child nodes are *continuous* blocks or statements, which are split into two continuous sets of code blocks or statements. For example, n continuous code blocks are split into the first $\lfloor n/2 \rfloor$ blocks and the remaining $n - \lfloor n/2 \rfloor$ blocks. Combined with the *region merge pass* (Section 2.3.3), all blocks are merged into as few continuous code regions as possible,

reducing redundant type conversions. Section 2.4.2 shows that by removing redundant type conversions, the transformed program can more accurately and efficiently reflect the resolvability of compiler-induced numerical inconsistencies under precision enhancement.

Furthermore, given how statements in loops could accumulate errors that could exacerbate compiler-induced inconsistencies, our bisection search prioritizes loop structures. Thus, loop BBs at the current level of the AST hierarchy are isolated first. If inconsistencies are resolved then the search is narrowed down to the identified loop BBs; otherwise the search proceeds normally.

We use the sample function from Figure 2.3a to illustrate hierarchical region isolation within a function. The statement that causes the compiler-induced inconsistency is in Line 5 (`ST6` in Figure 2.3b). The algorithm first searches in the loop BBs at the top level, between `BB1` and `BB6`. The inconsistency is resolved with `BB1` in enhanced precision, thus `BB1` is isolated and further split into `BB2` and `BB3`. `BB3` is isolated next, which is then split into `BB4` and `BB5`, with `BB4` then isolated and split into statements `ST4`, `ST5` and `ST6`, from which the constant expression $1.4E-41f$ in `ST6` is found to be the root cause of the inconsistency.

2.3.3 Source-to-Source Precision Enhancement

The precision enhancement module takes as input the marked simplified AST from the hierarchical code isolation module, and produces a transformed program where all floating-point operations in an enabled code region, whether it is a whole function or a continuous code segment, are in enhanced precision. Source-to-source program transformation allows the resulting programs to be successfully compiled by the same compilers that trigger the original inconsistencies. Furthermore, a source-level transformation, in contrast to IR or assembly level, is not affected by aggressive optimization passes such as instruction reordering which would obscure and obfuscate the boundaries between source code statements during binary generation.

CIEL detects and classifies CUDA host and device functions according to language-specific modifiers in the AST, such as the `__global__` and `__device__` modifiers in the function signature, and transforms code accordingly.

In terms of enhancing precision for CUDA kernels, while CPU programming platforms generally natively support floating-point types beyond double precision, GPU platforms do

not. Thus we design Ciel to support precision enhancement with custom extended precision floating-point types that support operator overloading and math functions. Some examples of extended precision libraries include CAMPARY [10] and CUMP [11], but only GPUprec [12] fits the above criteria for integration. GPUprec only requires modest effort to be integrated with Ciel. We use its quadruple precision type to perform precision enhancement because it offers the most support for math functions.

Ideally, all code executed is available to Ciel when isolating code within a code region. However, external functions whose code is not available may be called within a code region. Even though it would not be possible to isolate individual expressions within such external functions, isolating the function call site itself may still be helpful in isolating numerical inconsistencies. In cases where inconsistencies exist in external functions, and an enhanced precision version of the same function is available, replacing the original function calls with calls to their corresponding enhanced-precision functions is expected to resolve the inconsistencies. Thus, for functions called within enhanced code regions, Ciel automatically replaces those given in a customizable *replacement function list*, most of which are math library functions, with an enhanced precision version. In the example in Figure 2.3a, `sinhf` and `tanf` would be replaced with `sinh` and `tan`, respectively. On the other hand, precision enhancement of variables and constants consists of two stages: region and expression transformation, with targeted strategies for different categories of variables.

Stage 1: Region Transformation. This stage enhances the precision of floating-point operations in a specific code region including scalar variables, built-in vectors and constants. Region transformation consists of three passes: region merge, variable categorization, and code transformation.

Pass 1: Region Merge. This pass merges all basic blocks and statements to be enhanced into as few continuous code regions as possible. Compared to prior work, this pass is added specifically as an improvement in removing unnecessary type conversions in precision enhanced code. If two adjacent code blocks on the same level of the AST hierarchy are to be enhanced, then these blocks are merged into one single block. For example, `ST5` and `ST6` in

Figure 2.3b are adjacent and on the same level in the AST, thus they are merged into one block. If two adjacent code blocks that are on different levels of the AST hierarchy are to be enhanced, we insert a *variable checkpoint* between them so that redundant type conversions can be detected and removed during the Code Transformation pass. For example, `ST6` and `BB5` in Figure 2.3b are adjacent but on different levels in the AST, a variable checkpoint is inserted here so that there would be no redundant type conversions in between for variables such as `comp`.

Pass 2: Variable Categorization. This pass iterates through all variable uses in a code region, and categorizes scalar and built-in vector floating-point variables[2] into four groups. Our variable categorization algorithm is based on [2] which, in essence, separates *read-after-write* variables in a code region that require allocating temporary storage from variables that just require casting when referenced. We then categorize the *read-after-write* variables into two groups based on whether the variable declaration is inside (*reviseVars*) or outside (*replaceVars*) the code region since they require different transformation strategies. Finally we group the variables that only require casting when referenced by checking if they are only read (*rdVars*) or only written (*wrVars*). The last category (*wrVars*) was added in CIEL to implement precision enhancement in GPU programs with custom extended precision floating-point types described later in this subsection.

Pass 3: Code Transformation. This pass transforms variables according to their categorization:

T1 For *reviseVars*, the declarations of these variables (originally inside the region) are replaced with a temporary variable in higher precision; any reference to this variable inside the code region is replaced with its corresponding temporary variable; the declaration of the original variable is moved prior to the region's exit points, and initialized with the temporary variable.

[2]Pointers and array references are not categorized; their dereferences are directly cast.

T2 For *replaceVars*, a temporary variable declaration is inserted at the entry of the region, initialized with the value of the original variable (declared outside the region); any reference to this variable inside the code region is replaced with the temporary variable; the value of the temporary variable is assigned back to the original variable at region exit points.

T3 For *rdVars*, any reference to the variable inside the code region is explicitly upcast to higher precision.

T4 For *wrVars*, any assignment to the variable inside the code region is explicitly downcast to lower precision (reads may occur only outside the region).

Lastly, type conversion statements from/back to original precision are inserted at the entry/exit point(s) of a code region. Note that calls to functions that are not included in the replacement function list are treated as special exit points of the code region, and their arguments are cast to original precision prior to the function call. Additionally, if the exit/entry of a code region is a *variable checkpoint*, CIEL finds all the variables shared between the two code regions, and simply assigns the enhanced-precision replacement variable in the first region to the one in the second region. By doing so, CIEL prevents redundant type conversions between these two code regions.

Custom extended-precision floating-point types present unique challenges compared to built-in floating-point types. C++ allows implicit conversions among floating-point types, even when such conversions incur precision loss, such as from `double` to `float`. However, such implicit conversions are not possible for custom floating-point types. For example (assuming the type name is `dd_real`):

```
dd_real a = 1.0; dd_real b = a + 2.0;
```

There is ambiguity in `a + 2.0`, which can either be interpreted as an addition of two `dd_real` values or two `double` values before assigning the result to `b`. For such code to pass compilation, CIEL inserts explicit casts back to original precision in value assignments, function arguments, and other possible situations. These explicit casts require CIEL to categorize variables that are only written in a code region, hence a new category, *wrVars*, was added.

Another challenge when enhancing the precision of CUDA kernels is built-in floating-point vector classes. These classes provide vertex and matrix calculation in 2 to 4 dimensions and are widely used. Ciel supports transforming built-in vector type operations to enhanced precision, including type conversions for function arguments passed by reference or by dereferencing. This requires creating temporary variables that are live only during the function call. For this purpose, we implemented converter template class instances as anonymous variables in function arguments. Upon construction, they accept a reference or a pointer of the source variable, convert it to the target type, and provide a reference or a pointer in the target type to the function calls. When the function call is finished, destructors for these converter classes are invoked, and we assign the return value of these references/pointers back to the original variable.

Stage 2: Expression Transformation. This transformation is only applied when the code has been successfully isolated at the statement level. Specified subexpressions in each isolated statement are converted to enhanced precision. Ciel traverses the AST starting from the subexpression node, cast all variable reads and constants from subexpressions to enhanced precision, and the whole subexpression is explicitly converted back to original precision. For example, the subexpression `b*2.0f` in expression `a = b*2.0f+c` would be transformed to `(float)((double)b*2.0)` in enhanced precision.

2.4 Experimental Evaluation

This experimental evaluation answers the following research questions:

RQ1 How effective is Ciel at isolating compiler-induced numerical inconsistencies in heterogeneous programs?

RQ2 How does Ciel compare with the state of the art in isolating compiler-induced numerical inconsistencies in CPU programs?

2.4.1 RQ1: Numerical Inconsistencies in Heterogeneous Programs

Benchmarks. We collected a total of 339 compiler-induced inconsistencies: 330 inconsistencies observed in floating-point synthetic GPU programs, 5 inconsistencies triggered in NAS Parallel Benchmarks for GPU (NPB-GPU) [13], 3 inconsistencies triggered in the CUDA version of the Rodinia Benchmark suite for heterogeneous computing [14], and a real-world inconsistency found in the C version of the ECMWF Cloud Physics mini-app CLOUDSC [15].

The synthetic GPU programs were generated with Varity [16], a framework that randomly generates small programs written in CUDA C along with an input for which a numerical inconsistency is observed when using `nvcc -O3 -use_fast_math` in comparison to `nvcc -O0`. These programs use single-precision floating-point arithmetic, various C syntax mechanisms such as `for-loop` and `if` statements, and calls to external math functions.

The CUDA NAS Parallel Benchmarks demonstrate the ability of Ciel to isolate compiler-induced floating-point inconsistencies in programs originally written for CPU architectures and ported to GPUs. These programs use double precision, which means the extended precision capabilities of Ciel are used. On the other hand, the Rodinia programs are originally written as heterogeneous applications for which single and double precision implementations are available. Therefore, we use the version in single precision.

Finally, CLOUDSC is a standalone mini-app of the ECMWF cloud microphysics parameterization, which tests the CLOUDSC cloud microphysics scheme of the ECMWF Integrated Forecasting System (IFS). We choose CLOUDSC as a candidate to demonstrate the efficacy of Ciel in finding compiler-induced inconsistencies in real-world applications, and show its capability of adapting to other software platforms and languages supported by Clang.

Experimental Environment. We use a PC with octa-core Intel(R) i7-11800H processors, and NVIDIA RTX 3070 GPU with 5120 CUDA cores, running Ubuntu 20.04 LTS. We use Clang version 14.0.6 to perform source-to-source transformation, which supports CUDA SDK versions up to 11.1 with Compute Capability up to 8.6. Clang 14.0.6 and nvcc 11.1 are also the compiler versions we use to compile transformed GPU programs. For CPU programs, we use gcc 9.4.0. Our methodology is independent of GPU models as long as they have the

Table 2.2: Numerical inconsistencies in NAS, Rodinia and CLOUDSC programs.

Benchmark	Program	LOC	Input	Epsilon	Compilation Command
NPB-GPU	BT	5062	S	3.0e-12	`clang -O3 -ffast-math`
NPB-GPU	CG	1868	S	1.1e-15	`clang -O3 -ffast-math`
NPB-GPU	CG	1868	W	4.0e-16	`clang -O3 -ffast-math`
NPB-GPU	LU	4437	S	1.9e-12	`nvcc -O3 -use_fast_math`
NPB-GPU	MG	2349	W	2.9e-14	`clang -O3 -ffast-math`
Rodinia	LUD	717	256	1.2e-5	`nvcc -O0`
Rodinia	CFD	647	097K	7.2e-2	`nvcc -O0`
Rodinia	CFD	647	193K	1.9e-1	`nvcc -O0` & `-O3 -ffast-math`
N/A	CLOUDSC	2593	N/A	1.0e-11	`gcc -O3 -ffast-math`

same Compute Capability. For all compilers, we considered two sets of compiler flags: `-O0`, and `-O3` with fastmath.

Methodology for Triggering Numerical Inconsistencies. The compiler-induced numerical inconsistencies in NAS, Rodinia and CLOUDSC were previously unknown, and were discovered through testing. Specifically, we rely on verification routines that compare the relative errors in output values to an epsilon value ϵ to determine whether the results meet accuracy constraints. A compiler-induced inconsistency exists if a program passes its verification routines for some compiler settings but not for others.

For six of the NAS programs (BT, CG, FT, LU, MP and SP) and Rodinia LUD, we utilize existing verification routines where results are either compared to precalculated ground truth embedded in the program source code, or in the case of Rodinia LUD, the resulting two matrices are multiplied and then compared against the original matrix. For the Rodinia CFD Solver, we calculate the total density energy (TDE) as specified in [17] and compare it to the reference TDE value precalculated by running the double-precision version of CFD Solver compiled with `nvcc -O0`. For CLOUDSC, we compare relative errors for the main variables at the end of program execution against ground truth precalculated by running the original cloud scheme from IFS in FORTRAN.

We follow an existing methodology to *trigger* numerical inconsistencies, first introduced in [2]. Specifically, we set the epsilon value ϵ between the minimum and maximum errors observed amongst all compiler/optimization flag combinations for a given program, and maximize the ϵ value such that the program passes its verification routines only for some

compiler settings but not for others. Table 2.2 lists the inputs, epsilon values, and compiler commands used to trigger each of the 9 numerical inconsistencies reported for these programs.

Evaluation Results. We find that CIEL is effective at isolating code responsible for the numerical inconsistencies in 337 out of 339 instances (99.4%). In terms of isolation granularity, CIEL isolates at expression level in 318 out of 339 instances (93.8%), while the rest of the inconsistencies are isolated at line, block, or function level. Below we describe the results per benchmark.

Synthetic GPU Programs. CIEL isolates all inconsistencies: individual expressions in 310 cases, a code block in 18, and a function in the remaining 2. We manually examined the source code, inputs, outputs, and in some cases the assembly code of each program. Our inspection revealed that CIEL correctly isolated 328 out of 330 (99.4%) inconsistencies while only 2 (0.6%) were false positives.

We identified six categories of *true* compiler-induced numerical inconsistencies isolated by CIEL. Table 2.3 lists these categories, the number of occurrences, and sample code. Note that an inconsistency may belong to multiple categories.

The first four categories are purely related to floating-point operations. *Subnormal arithmetic* indicates that subnormal numbers are involved in the floating-point operations. *Math functions* are often involved in which extreme values may be computed differently depending on the implementations. For example, nvcc compiles `sinf()` as a single fast approximation instruction instead of a full function. Also in some cases, `Inf` or `NaN` values are involved, which are not strictly IEEE 754-2008 compliant under fast math. We also found that the results of some operations differ under different optimization flags due to *rounding errors*.

The last two categories are related to the setup of the benchmark programs themselves. We observed cases where resolving compiler-induced inconsistencies also required enhancing the precision of their *program inputs*. And lastly, we found a few instances for which the final line of code where the computation result is *printed* byte by byte is the cause of the

inconsistency. This is because the result of the computation is subnormal when converted from enhanced precision.

Table 2.3: Categorization of inconsistencies found in synthetic GPU programs.

Categories	# Programs	Percentage	Sample Code
Subnormal Arithmetic	125	37.9%	`+1.8922E-42f + var_3`
`Inf` or `NaN` Arithmetic	53	16.0%	`+1.3797E-35f / -0.0f`
Math Functions	41	12.4%	`sinf(+1.0195E25f)`
Rounding Errors	18	5.5%	`-16458 / 1.67329e-16`
Program Inputs	164	49.7%	N/A
Print Statements	11	3.3%	N/A

As for false positives, we found two cases where CIEL isolates statements that have no effect on the computation. Specifically, a variable is assigned a value that is immediately overwritten by another value. These assignment statements are located inside a loop. When the precision of the entire loop is enhanced, the inconsistency is resolved; but if only the precision of the statements after the initial assignment is enhanced, the inconsistency persists because of type conversions inserted by CIEL at the end of the region inside the loop.

Overall, CIEL took a total of 3 hours and 2 minutes to analyze all 330 programs, and 33 seconds per program on average.

Table 2.4: NPB-GPU and Rodinia Experiment Results. Time is given in mm:ss.

	Statement Level				Expression Level		
Program	Isolated Function	Line(s)	# Cfgs	Time	Exp.	# Cfgs	Time
BT.S	`exact_solution`	1874-1886	10	1:23	`zeta`	20	2:10
CG.S	`sparse`	1710,1722	18	1:23	`size,shift`	24	1:52
CG.W	`sparse`	1710,1713,1765	19	1:34	`size,scale`	28	2:24
LU.S	`ssor_gpu_kernel_2`	4023	8	1:03	`tmp`	11	1:15
MG.W	`rprj3_gpu_kernel`	2045-2050	14	1:16	`x2,y2`	34	3:02
CFD 097K	`cuda_compute_step_factor`	283	14	6:01	`sqrtf`, `speed_sqd`	26	10:10
CFD 193K	`compute_speed_sqd`	252 257	10	7:29	`velocity`, `speed_sqd`	40	22:11
LUD 256	`lud_internal`	—	17	1:16	—	—	—

NAS and Rodinia. CIEL isolated all 8 numerical inconsistencies in NAS and Rodinia programs. Table 2.4 shows the results for each program for both statement and expression level isolation. For BT.S, LU.S, MG.W, CIEL isolates variable expression(s) in one statement that causes the compiler-induced inconsistency. In CFD 097K, CIEL isolates a function call

with a variable parameter. For CG and CFD 193K, CIEL isolates 2 variables across 2 to 3 statements as the cause of the inconsistencies. In all cases above, the isolated expressions are inside deeply nested loops, so even a slight offset can be accumulated into a larger inconsistency that exceeds the error threshold. The only exception is the LUD program where only a function, `lud_internal` is isolated. Upon inspection, the reason seems to be that a variable `sum` is read and written throughout the function, affecting the whole matrix, and any type conversion would cause the inconsistency to persist.

CIEL isolated each inconsistency within 22 minutes, used less than 20 searches (configurations) for statement isolation, and used no more than 40 searches for expression isolation. About 1%-5% of run time is used on code transformation.

CLOUDSC. CIEL isolated a constant expression `(float)0.4` as the cause of the inconsistency. After looking further into the code repository [18] and reporting the issue to ECMWF scientists, we confirmed CIEL's result. It turns out ECMWF scientists had meant to temporarily introduce a bug during testing with the type casting but had forgotten to remove it; CIEL correctly suggests increasing the precision of that same argument to resolve the inconsistency. CIEL took 7 minutes to isolate the inconsistency, from which 8% is spent on program transformation.

Answer to RQ1: CIEL isolated 337 out of 339 inconsistencies in minutes with a precision of 99.4%, which included 328 synthetic GPU programs, NAS and Rodinia programs, and the mini-app CLOUDSC. In 318 cases (93.8%), CIEL isolated expressions. Manual inspection revealed inconsistency characteristics, such as the involvement of `Inf`, `NaN`, or subnormal numbers in arithmetic.

2.4.2 RQ2: Comparison with the State of the Art

Baseline. We compare CIEL to pLiner [2], the prior work for isolating inconsistencies at the statement level in CPU programs.

Benchmarks. Due to pLiner capabilities, this evaluation is limited to CPU programs. We adopt benchmarks from the publicly available pLiner repository (SHA ef94b40)[3] originally

[3] https://github.com/LLNL/pLiner/commit/ef94b40

used to evaluate pLiner, which include 50 floating-point synthetic CPU programs on Intel CPU platforms, and 3 programs from the C version of the NAS Parallel Benchmark: CG.B, SP.A, and SP.B. We use the same compiler, optimization flags, and error thresholds as pLiner in our evaluation.

Evaluation Results. CIEL achieves more precise isolation results than pLiner for 84.9% of the programs benchmarked. When isolating at the same statement level as pLiner, CIEL is 24.5% more efficient in terms of number of searches. The rest of this section describes the results per benchmark.

Synthetic CPU Programs. In 42 out of 50 programs, CIEL successfully isolates code at the statement level, and subsequently at the expression level. In 36 of these programs, CIEL isolates the same statement (line) as pLiner. In the remaining 6 cases, CIEL isolates at the statement level while pLiner can only isolate at code block or function level. CIEL explores 29.7% fewer configurations to achieve this result. On average, CIEL explores 5.2 configurations for statement isolation compared to 7.4 configurations explored by pLiner. Expression level isolation requires exploring additional configurations: 16.5 on average.

Table 2.5: NPB CPU Experiment Results. Time is in minutes:seconds.

	CIEL Statement Level				pLiner Statement Level				CIEL Expression		
Prog.	Function	Line(s)	#Cfgs	T_{line}	Function	Line(s)	#Cfgs	T_{line}	Exp.	#Cfgs	T_{exp}
CG.B	`sparse`	814,819,876	19	16:23	`sparse`	—	7	3:53	—	—	—
SP.A	`tzetar`	65,69	16	6:56	`y_solve`	68	25	7:50	`r4,t2`	23	9:36
SP.B	`exact_solution`	44-47	9	17:28	`exact_solution`	44-47	17	24:51	`zeta`	19	34:57

For the remaining 8 programs, there are two cases in which CIEL isolates a smaller code block than pLiner. There are four programs for which neither CIEL nor pLiner can resolve the inconsistencies by using precision enhancement. Lastly, there are two programs for which we were not able to reproduce the numerical inconsistencies. Note that in these cases, pLiner still proceeded with the search while CIEL immediately detected the absence of an inconsistency.

NAS CPU Benchmarks. Results for the NAS CPU benchmark are shown in Table 2.5. In CG.B, CIEL isolates three statements in function `sparse`, which has 227 lines of code, while pLiner can only isolate the whole function. pLiner stopped after it failed to resolve the

inconsistency even when all basic blocks in `sparse` are in enhanced precision; CIEL prevents this by avoiding unnecessary type conversions between basic blocks. In SP.B, CIEL first isolates the same statement as pLiner in function `exact_solution`, and then further isolates a variable. Finally in SP.A, CIEL and pLiner isolate different functions (`tzetar` vs. `y_solve`) due to exploring different areas of the search tree. We confirmed that precision enhancement of either function resolves the inconsistency. If we were to limit the search in CIEL to only explore function `y_solve`, then CIEL would isolate the same statement as pLiner. Ultimately, CIEL isolates two variables.

In terms of efficiency, CIEL uses fewer configurations than pLiner to isolate inconsistencies in SP.A (16 vs. 25) and SP.B (9 vs. 17) at the statement level. CIEL uses more configurations for CG.B (19 vs. 7), but it isolates the same function with only 4 configurations, and isolates at a finer granularity. Expression isolation requires an additional 7 and 10 configurations for SP.A and SP.B, respectively.

Answer to RQ2: CIEL shows comparable or superior results in isolating the inconsistencies in 44 out of 48 (92%) synthetic CPU programs and the NAS programs. Overall, CIEL isolates inconsistencies at the same level of granularity as pLiner but with higher efficiency, or at a finer level of granularity with an additional cost, in particular in the case of expression isolation.

2.4.3 Threats to Validity

While our evaluation set of programs is large and diverse, our results may not generalize to all applications. Also, compiler-induced numerical inconsistencies are input dependent, thus it is possible that other inputs could trigger additional inconsistencies in the same code regions, or elsewhere. Complementary use of dynamic analysis or code coverage information may be useful. CIEL does not handle non-deterministic applications, but some of such programs could still be analyzed by removing certain sources of non-determinism for testing purposes [19].

CIEL's implementation only handles a subset of C/C++ and CUDA platform constructs. Features such as anonymous functions or the `auto` keyword, introduced in C++11, are not currently supported. Handling some of these features may require a new approach in

simplified AST generation and code isolation. Nevertheless, given how Ciel can differentiate between host and device code, it could be adapted to any platform supported by the Clang compiler frontend, such as OpenMP and OpenCL [20].

Code isolation may be further impacted by special floating-point values such as ±0.0, `Inf`, and `NaN`. The processing of these values, if consistent across precisions but inconsistent between different optimization flags, may become a blind spot for precision enhancement. The choice of extended precision library may also impact search results and efficacy. GPUprec, for example, has known issues with math functions when it should return `NaN` but returns `zero` instead, which could affect isolation results. Finally, Ciel requires source code when enhancing precision. However, if the source code for a function is not available, Ciel may still isolate the call site if an enhanced precision variant of the function exists.

2.5 Related Work

This section discusses related work in detecting and isolating numerical errors, and in testing compilers and numerical code.

2.5.1 Detecting and Isolating Numerical Errors

Several tools have been developed to detect or isolate numerical errors in programs. The most closely related to Ciel are those that target compiler-induced numerical inconsistencies.

Compiler-Induced Inconsistency Isolation. pLiner [2] isolates known compiler-induced numerical inconsistencies in C/C++ CPU programs at the line level. pLiner's approach also includes hierarchical code isolation and precision enhancement as a method to isolate inconsistencies. Unlike pLiner, Ciel works on heterogeneous programs, which pose unique challenges when isolating numerical inconsistencies, as described in Section 2.3.3. Furthermore, Ciel isolates inconsistencies to the expression level rather than lines. FLiT [5] generates and runs custom-made tests under different optimization levels to *trigger* compiler-induced numerical inconsistencies, which are then isolated at the function level only. Compared to Ciel, FLiT does not employ precision enhancement for inconsistency isolation, and focuses on CPU programs.

Other Numerical Error Detection Tools. There are also tools that automatically detect or isolate specific categories of numerical errors but not compiler-induced inconsistencies. FPChecker [21] is a tool that automatically detects floating-point exceptions in GPU applications, which also uses Clang to transform CUDA code, but it does so at the IR level. While FPChecker operates on GPU programs, it does not isolate compiler-induced numerical inconsistencies. FPChecker also inspired other tools, such as Predoo [22] in the field of precision testing for Deep Learning (DL) operators.

Shadow Execution Techniques. PFPSanitizer [23] detects numerical errors by performing shadow execution with higher precision in parallel. Shadow execution with precision enhancement is also employed by Herbgrind [24] and FPDebug [25] with the goal of finding floating-point precision errors. While these tools share the concept of precision enhancement with CIEL, they focus on detecting floating-point precision errors rather than isolating compiler-induced numerical inconsistencies.

2.5.2 Testing Compilers and Numerical Code

CIEL transforms source code in small increments and tests whether numerical inconsistencies are resolved. This approach is inspired by prior work on compiler mutation testing and differential testing.

Compiler Mutation Testing. Le et al. [26] introduce equivalent modulo inputs (EMI) which mutates programs on unexecuted paths to expose compiler bugs that incorrectly execute these paths. ClassFuzz [27] uses EMI by mutating Java classfiles on predefined mutation operators, and send them to various JVM implementations for differential testing. Zhu and Zaidman [28] propose new mutator operations alongside conventional ones to expose bugs in GPU programs, but their work does not involve floating-point arithmetic. HeteroFuzz [29] introduces a multi-pronged fuzzing approach to detect platform-dependent divergence in heterogeneous programs running on FPGAs, using techniques including dynamic probabilistic mutations to reduce the long latency between invocations to hardware simulators. Overall, none of the above tools focus on exposing or isolating compiler-induced numerical inconsistencies.

Differential Testing. CIEL performs differential testing to check whether compiler-induced inconsistencies exist by providing the same input to a series of programs compiled

from the same source code but with various compilers and optimization flags. Differential testing has been applied before to numerical programs. FPDiff [30] performs differential testing between automatically identified *synonymous functions* across various numerical libraries to identify inconsistencies between the results from these functions under certain inputs. Unlike Ciel, FPDiff tests *different* implementations of a given function, and it does not consider different compilers or optimization flags.

2.6 Conclusion

With scientific code ported or developed on GPUs, compiler-induced numerical inconsistencies can arise at various stages of development. Unfortunately, automatic tools to isolate such problems are nonexistent, which harms productivity in GPU computing. In this chapter, we demonstrate a practical method to identify the root cause of such inconsistencies in heterogeneous code. We implemented our approach in the tool Ciel based on the effective bisection search algorithm, and improved over the state of the art for CPU programs in both efficiency and accuracy. Most importantly, Ciel addresses a number of challenges to handle heterogeneous code. Our evaluation on synthetic GPU programs, GPU benchmarks, and real-world mini-app shows the effectiveness of Ciel at isolating inconsistencies in heterogeneous code with a precision of 99.4%.

Chapter Acknowledgements

This chapter is based on the paper "Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code" by Dolores Miao, Ignacio Laguna, and Cindy Rubio-González, presented at *ISC High Performance*, 2023. This paper received the **Hans Meuer Best Paper Award** at the conference. The code and experimental data for Ciel are publicly available at https://github.com/LLNL/Ciel/.

Funding: This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-846081), the U.S. Department of Energy, Office of Science, Advanced Scientific

Computing Research, under awards DE-SC0022182 and DE-SC0020286, and the National Science Foundation under award CCF-1750983.

Chapter 3

CIGEN: Input Range Generation for Compiler-Induced Inconsistencies

3.1 Problem and Motivation

Limitations of Existing Approaches. Previous research, including CIEL presented in the previous chapter, has studied how to diagnose compiler-induced inconsistencies for a *specific input*. Tools such as FLiT [5], pLiner [2], and CIEL [31] can identify the function, line, or expression in a program that is associated with an inconsistency. However, the main limitation of such tools is that they can only diagnose the inconsistency location for a *specific input* i out of all the program inputs I—usually i is given by the programmer to the tool, and it is confirmed that an inconsistency occurs for i between a specific set of compiler and optimization flag combinations.

The Input Range Question. A more general set of questions to ask are: Given a program $\mathcal{P}$, what inputs $j, k, \ldots \in I$, in addition to i, also activate compiler-induced numerical inconsistencies? How do we find such inputs for $\mathcal{P}$? For all the inputs in I that we know induce inconsistencies, which input induces the largest inconsistency (or error)? As HPC code is ported and tested in different environments, these are crucial questions to answer—having a better understanding of the inputs that cause such inconsistencies and

the error they introduce helps programmers better address such inputs in their code before testing, and effectively safeguard their programs against these inconsistencies.

Recently, CIV [32] addressed a similar question—finding *an* input which triggers high compiler-induced numerical inconsistencies in programs. However, CIV is limited to finding *one* inconsistency-inducing input, not input ranges, and its requirement of comparing LLVM IR-based instruction traces limits their use when it comes to compilers not built on LLVM.

Our Approach. We present CIGEN (Compiler-induced Inconsistencies and Input Generation), which finds *input ranges* that cause high compiler-induced inconsistencies in numerical programs. Given the observed characteristics of compiler-induced inconsistencies in numerous routines, we propose a multi-phase approach: first CIGEN performs input-partitioned and coverage-based random sampling to gather an initial set of input points that cause inconsistencies; then groups these input points into clusters and generates non-tight input ranges; finally, employs dense sampling and differential evolution (DE) [33] in each input range to discover inconsistency characteristics.

Contributions. The contributions of this chapter are:

- A multi-phase approach for finding input ranges with inputs that cause significant compiler-induced numerical inconsistencies in a program, combining input-partitioned and coverage-based random input sampling, input sample clustering, and differential evolution (Section 3.3).

- A platform and compiler independent implementation of our approach in the tool CIGEN. Users provide as input a program with floating-point inputs and a floating-point output, and CIGEN outputs a number of ranges in the input domain, each with statistics of inputs that cause compiler-induced inconsistencies.

- An evaluation on 175 functions from the GNU Scientific Library that shows that CIGEN compares favorably against the state of the art in terms of the number of functions found with inputs that trigger high compiler-induced inconsistency, time usage, and result stability (Sections 3.4.2 and 3.4.3).

- An investigation into characteristics of inconsistency-inducing inputs found by CIGEN, and into the difference in source code that causes these compiler-induced numerical inconsistencies (Section 3.4.4).

3.2 Background

3.2.1 Quantifying Numerical Inconsistency

It is important to establish a standard in how we calculate inconsistencies between outputs from two program variants. Since both outputs are of limited precision, metrics from comparing one limited precision number to an infinite precision counterpart, such as relative error or even unit in last place error (ulp error) does not accurately show the extent of the inconsistencies. Therefore we follow the method used in Herbie [34], originally from STOKE [35], where we simply calculate the amount of floating-point numbers between the two outputs, then get its base-2 logarithm. We call it *inconsistency error*.

$$\mathcal{E}(x, y) = log_2|\{z \in \mathbb{FP}|min(x, y) \leq z \leq max(x, y)\}| \tag{3.1}$$

Under this equation, the maximum inconsistency error possible for $\mathbb{FP}64$ is $\mathcal{E}(1.79e+308, -1.79e+308) = 63.99$. Additionally, how inconsistency errors are calculated when either x or y is a special value such as `inf` or `NaN` is shown in Table 3.1. Note that there is a baseline column in the table, even though unlike precision error calculation in tools such as Herbie, there does not exist a baseline calculated with infinite precision when calculating inconsistency error between two numbers; though in practice, we may still define a certain compiler and optimization flag combination as baseline to determine the degree of an inconsistency.

In this chapter we use `clang -O0` as a baseline. We choose this specification for two reasons: first, if the baseline result is a special number, the function input domain likely does not include the current input, thus the inconsistency from the current input is not of practical concern; second, `NaN` values usually result from instructions with `inf` as operands, therefore it is fair to treat them as `inf` when calculating inconsistency errors.

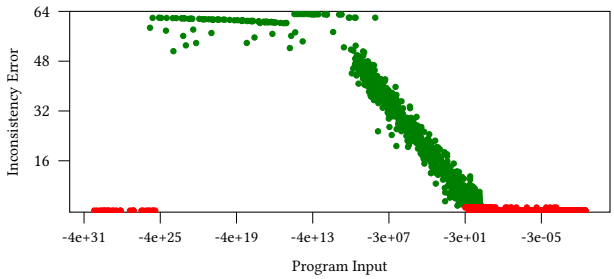


Figure 3.1: A subset of sampled inputs for the GSL function `gsl_sf_Airy_Ai` and the inconsistency error they have induced. A Red dot is an input that does not trigger inconsistency, while a green dot is an input that does. All negative input values outside of the plot have an inconsistency error of zero and are thus omitted.

3.2.2 Compiler-Induced Numerical Inconsistency

Table 3.1: Inconsistency error when at least one of the values is a special value.

baseline (x)	other (y)	inconsistency
inf or NaN	any	0.0
normal values	+/-inf	$\mathcal{E}$(x, y)
normal values	+/-NaN	$\mathcal{E}$(x, +/-inf)

A compiler-induced numerical inconsistency is an inconsistency shown in output values that come from running programs compiled from the same source code under different compilers and/or optimization flags. An example of a real-world function is the `gsl_sf_Airy_Ai()` GSL function. With input -973569893418508.1, if compiled with `clang -O0`, the return value would be $-6.314246267753519e+77$; but if compiled with `clang -O3` the return value would be 0.00010100276891802863. The inconsistency error introduced here is 63.161, which is very close to the maximum possible inconsistency error of 63.99 for $\mathbb{FP}64$.

Next, we formally define compiler-induced numerical inconsistency in terms of input. Suppose we have two floating-point representations of the same function under two different compiler/optimization flag combinations, one baseline $y_0 = f_0(x)$, one $y_{opt} = f_{opt}(x)$. Then, we can define an inconsistency function with an input x as follows:

$$Inc_f(x) = \mathcal{E}(f_0(x), f_{opt}(x)) \tag{3.2}$$

Therefore, to study the characteristics of compiler-induced numerical inconsistencies is to study the characteristics of $Inc_f(x)$ itself; to find an input x that maximizes compiler-induced inconsistency is to find a local or global maximum of $Inc_f(x)$.

During our preliminary research, we found that $Inc_f(x)$ functions may exhibit complex characteristics. As an example, we compile the GSL function `gsl_sf_Airy_Ai()` with `clang -O0` and `clang -O3 -ffast-math` separately into two binaries, run both with 25,000 randomly generated inputs from the $[-\infty, 0]$ input domain, and calculated the inconsistency error triggered by each input. First we observe that for all sampled inputs, except for the input range shown in Figure 3.1, $Inc_f(x)$ is zero. This makes finding at least *one* input that causes inconsistency a prerequisite of finding input ranges that contain high inconsistency errors. Second, when we focus on inputs in the range [-1e-30, -1e-8] and the inconsistency errors they trigger which is shown in Figure 3.1, it is evident that the inconsistency function has numerous local minimums and maximums, and thus difficult to express in polynomial terms; simply employing widely used global optimization strategies such as Bayesian Optimization (BO) [36] or Markov Chain Monte Carlo (MCMC) [37] on the whole input domain may not be suitable for functions like this. Lastly, when the input value is around -3e+1, there is no clear boundary where all inputs inside the boundary cause inconsistencies, and all inputs outside the boundary cause no inconsistencies. This means that defining a clear-cut boundary with a limited number of samples is difficult.

To better address the characteristics of inconsistency functions as shown above, we propose a novel, multi-phase method to study the characteristics of compiler-induced inconsistencies which will be discussed in Section 3.3.

3.3 Technical Approach

The workflow of CIGEN consists of three phases: first, domain-partitioned sampling and coverage-based random sampling are performed to generate sampling points in the input domain whose inconsistencies are then calculated; second, input clustering is used to generate "candidate input ranges" (CIR) where sample inputs that cause inconsistencies exist; lastly, dense sampling and DE are performed for each CIR to find the statistics of inputs in the

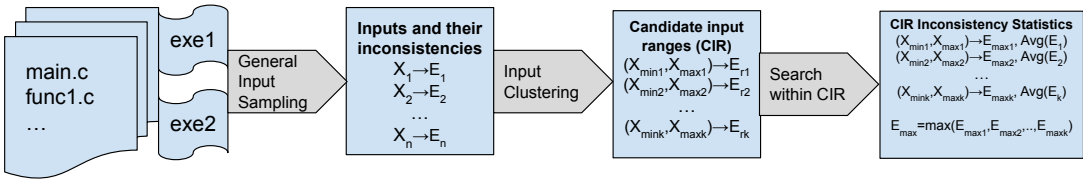


Figure 3.2: CIGEN: Tool Workflow.

CIR, such as the average and maximum inconsistency errors. The visualized workflow of CIGEN can be found in Figure 3.2.

Since floating-point numbers have logarithmic distribution, all subroutines in this algorithm are either run on the logarithmic scale such as the clustering algorithm and the optimization algorithm, or that exponents and mantissas are treated separately such as generating random floating-point numbers from randomized exponent and mantissa. We go into detail in the next subsections.

3.3.1 General Input Sampling

The first phase in our algorithm is general input sampling, which combines two different methods of generating inputs and testing them to determine whether they trigger compiler-induced inconsistencies: domain-partitioned sampling and coverage-based sampling. The goal of this phase is to find a set of inputs that trigger compiler-induced inconsistency before we can categorize them into groups in the next phase. The algorithm for this phase is shown starting from Line 16 in Algorithm 2.

In this phase, CIGEN generates a set of sample points which are deemed representative of the characteristics of the inconsistency function $Inc_f(x)$ in the input domain. The first step of the algorithm, called domain-partitioned sampling, is to partition the floating-point input domain of the program, $\mathbb{FP}$, into several partitions, based on the exponent part of the floating-point number. The partitions are predefined in the case of $\mathbb{FP}32$ and $\mathbb{FP}64$ in CIGEN, and can be customized depending on the floating-point types used in the program. For $\mathbb{FP}64$ programs which are evaluated in this chapter, the partition CIGEN uses is similar to the many-range partition used in XSCOPE [38]:

Algorithm 2: The multi-phase algorithm in CIGEN.

```
1  Function RandomFloat(Domain):
2      Sign_j = rand (0, 1);
3      Exp_j = rand (Domain);
4      Mant_j = randmantissa ();
5      return Sign_j, Exp_j, Mant_j
6  Function TestInc(x, S_a, S_z):
7      inc = Inc_f(x);
8      MaxInc = max (inc, MaxInc);
9      if inc > 0 then
10         S_a = S_a ∪ {(x, inc)};
11     else
12         S_z = S_z ∪ {(x, inc)};
13     return inc
14 Function MainAlgorithm(Inc_f, n):
15     // General Input Sampling
16     S_a, S_z, ExpSet_1...n = ∅;
17     for i ← 1 to K_1 do
18         for j ← 1 to n do
19             Sign_j, Exp_j, Mant_j = RandomFloat(FP_rand());
20             ExpSet_j = ExpSet_j ∪ {(Sign_j, Exp_j)};
21         x = combine (Sign_1...n, Exp_1...n, Mant_1...n);
22         TestInc(x, S_a, S_z);
23     for i ← 1 to n do
24         foreach (Sign, Exp) ∉ ExpSet_i do
25             for j ← 1 to n do
26                 if i = j then
27                     Mant_j = randmantissa ();
28                     Sign_j, Exp_j = Sign, Exp;
29                 else
30                     Sign_j, Exp_j, Mant_j = RandomFloat(FP_rand());
31             x = combine (Sign_1...n, Exp_1...n, Mant_1...n);
32             TestInc(x, S_a, S_z);
33     // Input Clustering
34     CIR = ∅;
35     for S_ai, S_zi in each orthant do
36         C_i1, ..., C_im = ClusteringByAbsLog (S_ai);
37         foreach C_ij in C_i1, ..., C_im do
38             (BB_j, MaxInc_j, MaxX_j) = MinBoundingBox (C_ij);
39             enlarge BB_j by points in S_zi;
40             CIR = CIR ∪ (BB_j, MaxInc_j, MaxX_j);
41     // Search Within Candidate Input Ranges
42     for CIR_i in CIR do
43         for j ← 1 to K_2 do
44             for k ← 1 to n do
45                 Sign_k, Exp_k, Mant_k = RandomFloat(CIR_i);
46                 ExpSet_j = ExpSet_j ∪ {(Sign_j, Exp_j)};
47             x = combine (Sign_1...n, Exp_1...n, Mant_1...n);
48             inc=TestInc(x, S_a, S_z);
49             replace MaxInc_i, MaxX_i if inc > MaxInc_i;
50         DifferentialEvolution (Inc_f, CIR_i, MaxX_i);
```

$$F = N_{max}^{-}, -2^{1020}, -2^{333}, -2^{32}, -2^{3}, -1,$$
$$-2^{-3}, -2^{-32}, -2^{-333}, -2^{-1018}, 0, 2^{-1018}, 2^{-333},$$
$$2^{-32}, 2^{-3}, 1, 2^{3}, 2^{32}, 2^{333}, 2^{1020}, N_{max}^{+} \quad (3.3)$$

$$fp_0 = N_{max}^{-}, fp_1 = -2^{1020}, \ldots \quad (3.4)$$

$$\mathbb{FP}_0 = |fp_0, fp_1|, \mathbb{FP}_1 = |fp_1, fp_2|, \ldots \quad (3.5)$$

CIGEN uses this input partition because the sampling rate is higher in partitions with very small (close to subnormal), close to 1.0, and very large (close to infinite) floating-point numbers. Our assumption is that numbers in these partitions are likely to trigger floating-point exceptions such as underflow or overflow, thus causing compiler-induced numerical inconsistencies. Compared to XSCOPE, CIGEN uses values which are powers of 2 as boundaries of each input partition instead of powers of 10. CIGEN then randomly chooses a particular partition $\mathbb{FP}_i = |fp_i, fp_{i+1}|$ and then generates a random floating-point number within $\mathbb{FP}_i$. Since floating-point numbers are logarithmically distributed, if floating-point numbers are generated in $\mathbb{FP}_i$ with uniform distribution, in cases where fp_i is much smaller than fp_{i+1}, for example, if $fp_i = 1$ and $fp_{i+1} = 32$ as shown in Figure 2.1, the value closest to fp_{i+1} is 16x more likely to be selected. To mitigate this issue, CIGEN instead randomly generates an exponent and a mantissa with uniform distribution, while keeping the resulting floating-point value within $\mathbb{FP}_i$. In this way it is guaranteed that any floating-point number in $\mathbb{FP}_i$ has the same probability of being selected, whether the number is closer to fp_i or fp_{i+1}. This step is then repeated K_1 times. The time complexity of this step is $O(2^n)$ where n is the number of input parameters.

Coverage-based Sampling. The next step is coverage-based sampling. CIGEN collects a set of exponent values for each input parameter that are not covered by the inputs generated in the first step. For example, if the exponent value 100 did not appear in previously generated inputs for input parameter 1, then 100 will be added to set 1. Then CIGEN selects an exponent value from set 1, combines this exponent with a random mantissa as input parameter

1, generates other input parameters using domain-partitioned sampling described in the first step, and tests this set of input parameters to see the inconsistency error it triggers. Continue this selection process until all exponents in set 1 have been selected, then repeat the same process in set 2, until all exponents in all sets are selected. The goal of this step is to make sure that for all input parameters, every possible exponent value can be found in at least one input. The time complexity for this step is $O(e \cdot n)$ where e is the number of possible exponents for the floating-point type, and n is the number of input parameters.

The inputs generated by both steps above are tested for compiler-induced numerical inconsistency. If an input triggers an inconsistency, the input is added to S_a, otherwise it is added to S_z. Both sets of inputs, along with the inconsistency errors they trigger, are sent to the next phase where they are used in determining the bounds of input ranges that contain inconsistency-inducing inputs.

3.3.2 Input Clustering

The input clustering phase is shown starting from Line 34 in Algorithm 2. It takes the two sets of inputs from the previous phase, S_a and S_z, categorizes them into various groups according to the relative proximity between inputs using hierarchical agglomerative clustering, and generate a candidate input range (CIR) for each group.

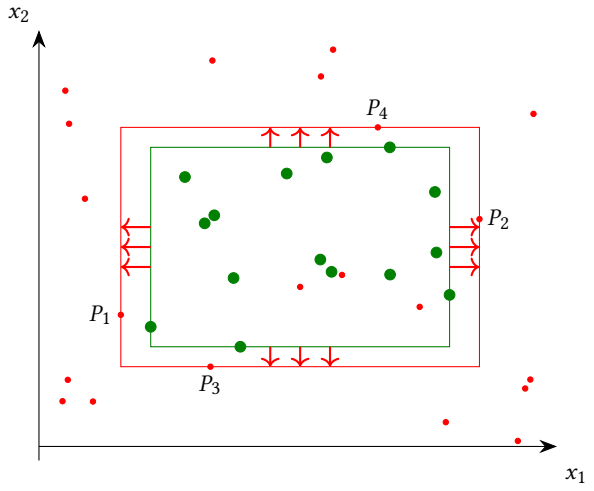


Figure 3.3: An example showing how to calculate a CIR from a set of input points that cause inconsistencies in larger green dots, and input points that do not cause inconsistencies in smaller red dost. Green box shows the minimum bounding box for the green points, while the red box shows the candidate input range calculated for this set of inconsistency-inducing inputs.

First, CIGEN categorizes S_a and S_z into multiple subsets according to the orthant each input belongs to. The reason for this is that the clustering algorithm used in this phase calculates the Euclidean distance between points *on the logarithmic scale*, and floating-point numbers in different orthants cannot be put on the same logarithmic scale. For example, if $X \in S_a \,\&\, sign(x) = (+,-)$, then $X \in S_{a(+,-)}$. This means that all inputs in this subset have the same sign configuration: the first input parameter is positive, and the second is negative. We now use the subsets $S_{a(+,-)}$ and $S_{z(+,-)}$ to describe the algorithm in this subsection.

Second, for each subset where all inputs belong to the same orthant, all parameters in $S_{a(+,-)}$ are transformed to the logarithm of the absolute value, then CIGEN runs a clustering algorithm to categorize them into different groups. Specifically, CIGEN uses hierarchical agglomerative clustering [39] due to the fact that this algorithm does not require a predefined number of clusters to run, and that CIGEN does not know beforehand how many clusters there are. The end result is a list of groups (subsets) $S_{a(+,-)1}$, $S_{a(+,-)2}$, ..., $S_{a(+,-)r}$.

Lastly, CIGEN calculates a CIR for each $S_{a(+,-)k}$ where $1 \leq k \leq r$. An example is shown in Figure 3.3. CIGEN first calculates a minimum bounding box (green rectangle in the figure) for the inputs in $S_{a(+,-)k}$ (green dots in the figure), then for each axis (x_1 and x_2) representing each input parameter, CIGEN finds one point in $S_{z(+,-)}$ outside of the minimum bounding box and closest to the lower bound on the current axis (P_1 and P_3 for axes x_1 and x_2 in the figure), then update the lower bound on the current axis to this point; then finds another point in $S_{z(+,-)}$ outside of the minimum bounding box and closest to the upper bound on the current axis (P_2 and P_4 for axes x_1 and x_2 in the figure), then update the upper bound on the current axis to this point. Repeat this process for all axes and the updated bounding box becomes the new CIR. CIGEN then gathers all CIRs and sends them to the next phase.

3.3.3 Search Within Candidate Input Ranges

The third phase of the CIGEN algorithm identifies characteristics of each CIR sent from the last phase, including average and maximum inconsistency. CIGEN achieves this by running a two-step optimization algorithm on each CIR. The algorithm can be found starting from Line 42 in Algorithm 2.

First, for all CIR, CIGEN runs dense sampling with the same input sampling method as domain-partitioned sampling in Section 3.3.1 K_2 times to gather data on statistics of inconsistency-inducing inputs in the CIR. The end results are the average of inconsistencies triggered by sampled points, and a maximum inconsistency along with its triggering input. This information is then sent to an optimization algorithm.

There are many optimization algorithms available for both approximating a function and finding the global maximum/minimum. We have previously shown in Section 3.2.2 that the inconsistency function $Inc_f(x)$ (a) has numerous local minimums and maximums, and thus it is difficult to express in polynomial terms; and (b) no bound can be inferred where all inputs inside the bound cause inconsistencies, while all inputs outside the boundary cause no inconsistencies. Thus CIGEN uses differential evolution [33] to locate the global maximum of the inconsistency function $Inc_f(x)$ in the CIR. DE is selected because it is a genetic algorithm where current candidates are improved upon by mutations.

CIGEN uses logarithmic scaling for each input range to be optimized. This makes the process of finding maximums more efficient when input ranges are large. Thus the inconsistency function is transformed as:

$$Inc'_f(x) = \mathcal{E}(f_0(2^x), f_{opt}(2^x)) \tag{3.6}$$

where x is between the logarithm of minimum and maximum values in the input range. The input in the CIR that CIGEN currently reports to trigger maximum inconsistency error is also sent to DE as an initial population, in order to speed up the optimization process. Once all DE processes are finished, the maximum inconsistency between all local maximums found will become the reported global maximum.

3.4 Experimental Evaluation

This evaluation answers the following research questions:

RQ1 How effective is CIGEN at finding inputs that trigger large compiler-induced inconsistencies in comparison to the state of the art and a general-purpose approach?

RQ2 How time-efficient is CIGEN in comparison to the state of the art?

RQ3 How many input ranges can CIGEN find with inconsistency-inducing inputs, and what information can we infer from these input ranges?

3.4.1 Experimental Setup

Benchmarks. We evaluate CIGEN on 175 functions from the GNU Scientific Library (GSL) [40] version 2.7. GSL is a numerical library for C and C++ programmers that provides a wide range of mathematical routines, such as linear algebra, differential equations and special functions. Our evaluation of GSL focuses on all library functions with floating-point input and output, consisting of 175 functions, most of which are special functions such as Bessel and hypergeometric functions. Amongst these functions, 122 are single-parameter functions, while the rest 53 are multi-parameter.

We assume that the input domain of these functions is $\mathbb{FP}64^n$, where n is the number of input parameters, unless the input domain of a function is explicitly stated in the GSL documentation. There are 47 functions out of 175 where there are input domain specifications in the GSL documentation. One example is the input domain for `gsl_sf_bessel_K1(x)` which is $x > 0$. This is a valid assumption because inputs outside of the input domain have no valid meaning in the mathematical sense and may return an `inf` or `NaN` value.

Programs under evaluation are compiled and tested with Clang 16.0.6. Inconsistencies are calculated by comparing results from two binary variants of the same program, one compiled with `-O0` and the other with `-O3 -ffast-math`. This applies to both CIGEN and CIV (both the original paper and our implementation), for a better comparison.

Baselines. The closest work to CIGEN in terms of finding inputs that trigger compiler-induced numerical inconsistencies is CIV [32], which for a given program generates an input that triggers a large compiler-induced inconsistency. Thus, we compare CIV and CIGEN in their effectiveness to find large inputs that *maximize* compiler-induced numerical inconsistencies. The source code of CIV is not available publicly, and the CIV authors declined our request for source code. Thus, for better comparison under the same runtime

environments, we have implemented the algorithm in CIV, and set up its parameters according to the specifications in this chapter.

Since our focus when comparing with the state of the art is in maximizing inconsistencies, we also adapt a general algorithm that finds inputs that cause large errors. Specifically, we use the Binary Guided Random Testing (BGRT) method, implemented in S3FP [41]. This baseline is also used by the CIV authors in their evaluation.

Algorithm Parameters. We specify the number of random samples in both domain-partitioned sampling steps (Section 3.3.1 and Section 3.3.3) with $K_1 = K_2 = 256 \cdot 2^n$, where n is the number of input parameters of the function. We use the scikit-learn [42] machine learning library to implement agglomerative clustering (Section 3.3.2), with parameters as follows: n_clusters=None, distance_threshold=0.2, metric='euclidean', linkage='single'. This means we use the minimum Euclidean distance between input points as the criteria for determining whether two points belong to the same cluster. Lastly, we use SciPy [43] to implement differential evolution (Section 3.3.3) with parameters: updating='deferred', popsize=20, maxiter=50.

CIGEN and CIV run until termination, thus do not require a budget. For BGRT, we follow the specification given in [32] and set a budget of $T = 60 \cdot 2^n$ seconds, where n is the number of input parameters of the function.

Runtime Environments. We use a PC with an 18-core Intel(R) i9-10980XE processor with 256 GiB RAM running Ubuntu 22.04 OS in our evaluation. While the core algorithms for each tool are run serially, the process of measuring the compiler-induced numerical inconsistencies for every input is run in parallel, utilizing all CPU resources.

Table 3.2: Number of GSL functions for which CIGEN, CIV, and BGRT have discovered input that causes an inconsistency error higher than 48, 32, 8, and 0, respectively.

Method	>48	>32	>8	>0
CIGEN	125 (71.4%)	127 (72.6%)	133 (76.0%)	163 (93.1%)
CIV	81 (46.3%)	94 (53.7%)	110 (62.9%)	155 (88.6%)
BGRT	20 (11.4%)	20 (11.4%)	22 (12.6%)	35 (20.0%)

3.4.2 RQ1: Effectiveness in Triggering Large Inconsistencies

We first evaluate the effectiveness of CIGEN in finding inputs that trigger large compiler-induced inconsistencies in comparison to CIV and BGRT. We consider two evaluation metrics: the number of GSL functions for which a compiler-induced inconsistency is triggered, and the average of maximum inconsistency errors discovered for these functions.

Functions with Triggered Inconsistencies. Table 3.2 shows the number of GSL functions for which the tools can find inconsistency-inducing inputs with error higher than 48, 32, 8, and 0, respectively. CIGEN finds inputs that cause high inconsistency error of 48 or more in 125 functions. This is 54.3% more functions than CIV, which triggers such error for 81 functions. On the other hand, BGRT can only find inputs triggering inconsistency error of 48 or more in 20 functions. Note that CIGEN's 125 functions are a superset of the 81 functions from CIV and the 20 functions from BGRT, which means that all high compiler-induced inconsistencies discovered by CIV and BGRT can also be found by CIGEN. Even when considering smaller inconsistencies, e.g., 8 or higher, CIGEN triggers inconsistencies in a larger number of functions: 133 vs. 110 in CIV and 22 in BGRT.

We further discuss the results based on whether a function is single- or multi-parameter. Out of 122 single-parameter GSL functions tested, CIGEN finds inputs that trigger 32 or higher compiler-induced numerical inconsistency error in 80 of them. In contrast, CIV can only find inputs for 51 functions. CIGEN shows 56.9% improvement over the state of the art. In contrast, BGRT can only find such inputs for 14 functions. On the other hand, among 53 GSL functions with multiple input parameters, CIGEN finds inconsistency-inducing inputs of error 32 or higher in 47 of them, while CIV triggers such errors in 43 functions, a 9.3% improvement. BGRT can only generate inputs for 6 functions. If we consider functions for which an error of 8 or higher is triggered, the improvement of CIGEN over CIV is smaller: CIGEN triggers inconsistencies in 84 single-parameter functions vs. 64 with CIV, a 31.3% improvement; and in 49 multi-parameter functions vs. 46 with CIV, a 6.5% improvement. And again, BGRT performs unfavorably against the other two, with 16 single-parameter functions and 6 multi-parameter functions.

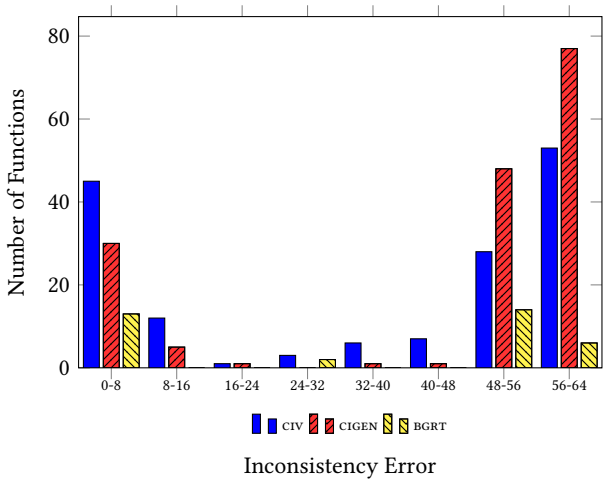


Figure 3.4: The number of functions with various ranges of maximum inconsistency error discovered by CIGEN, CIV and BGRT. CIGEN finds inputs that trigger high inconsistency error (>48.0) for more functions than CIV and BGRT.

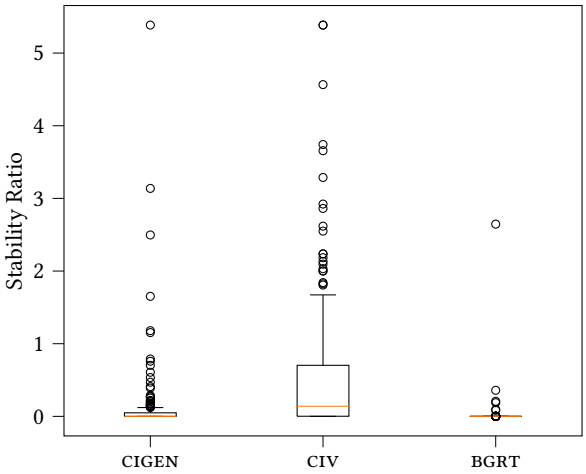


Figure 3.5: Box plot showing the statistics of stability ratio between the tools for all 175 functions tested. The boxes show the range between the first to the third quartile of results, and the whiskers show the interquartile range. The flier points show stability ratio outside of the interquartile range.

Maximum Average Inconsistency Error. The average inconsistency error triggered on average by CIGEN is 43.40 (38.25 for single-parameter functions and 55.26 for multi-parameter functions) vs. CIV's average of 32.33 (22.51 for single-parameter functions and 51.41 for multi-parameter functions), and BGRT's average of 6.81 (6.72 for single-parameter functions and 7.02 for multi-parameter functions). The statistics of peak compiler-induced inconsistencies of all functions can be found in histogram form in Figure 3.4. From the histogram, we observe that the maximum inconsistencies of most functions for all tools

are distributed in the 0-8 and 48-64 regions, with CIGEN successfully triggering higher inconsistencies for more functions (see the 48-64 region).

From our manual inspection of the results, we can group the compiler-induced inconsistencies CIGEN discover into three categories: (1) smaller (<8) inconsistencies that are accumulated from multiple rounding errors; (2) inconsistencies that are close to 52, where the output from `clang -O3 -ffast-math` is zero while the output from `clang -O0` is subnormal (or close to subnormal); and (3) inconsistencies close to the maximum possible inconsistency error of 64, either from an instruction that generates special numbers such as `inf` or `NaN`, or from sign flips.

Result Stability. Given CIGEN and our baselines utilize random testing, indeterminacy can be a potential issue affecting effectiveness. To evaluate how indeterminacy impacts the maximum inconsistency error triggered by the tools, we use the same method specified in [32] to compare the stability between CIGEN, CIV and BGRT. We run each of the three tools on the same 175 GSL functions 30 times and calculate the stability ratio for each function tested, which is defined as the ratio of the standard deviation ρ to the mean μ: $C_v = \frac{\rho}{\mu}$.

The result is shown in Figure 3.5. Judging from the box plot, we can first conclude that BGRT performs more stably than both CIGEN and CIV. However, since BGRT performs much less favorably than CIGEN and CIV, the stability shown here simply means it performs consistently worse than the other two. When we compare CIGEN and CIV, we can see that (1) the interquartile range from CIGEN is much smaller than the one from CIV; and (2) in terms of flier points, which indicate functions for which the tools are least stable, CIGEN still performs more stable than CIV. Thus we conclude that CIGEN is more stable than CIV in finding inputs that cause maximum inconsistencies.

Comparison with Original CIV Results. As noted earlier, we did not have access to the original implementation of CIV (Section 3.4.1), and thus all results discussed in this chapter were obtained through our own implementation. Nevertheless, it is important to note that when directly compared to the results presented in [32], CIGEN is even more effective than reported in this chapter. From Table 3.2 we know that CIGEN finds inputs that cause inconsistency error higher than 48 in 71.4% (125 out of 175) of functions, compared to 46.3% (81 out of 175) in our CIV implementation and 30.4% (48 out of 158) reported in the CIV

paper. Therefore, we can conclude that CIGEN is superior to both the original and our implementation of CIV in terms of maximum inconsistency error discovered.

Answer to RQ1: CIGEN and CIV perform closely in terms of finding an input that triggers compiler-induced inconsistencies *at all*, even though CIGEN is still slightly ahead; on the other hand, CIGEN is superior to CIV and BGRT in finding inputs that maximize compiler-induced inconsistencies in both single- and multi-parameter functions. Finally, simply adapting a general-purpose approach for maximizing error is not effective for compiler-induced inconsistencies.

3.4.3 RQ2: Time Efficiency

In this section we evaluate the efficiency of CIGEN and compare it with the state-of-the-art tool CIV.[1] CIGEN finds at least one inconsistency-inducing input for 163 out of 175 functions (see Table 3.2). For these functions, CIGEN takes an average of 17.95 seconds to generate inputs. For the remaining 12 functions, CIGEN takes an average of 1.31 seconds to conclude that no inconsistency-inducing input is found. On the other hand, our implementation of CIV takes on average 155.15 seconds to generate inputs that trigger inconsistencies in 155 out of 175 functions. For the remaining 20 functions, it takes 44.73 seconds on average to determine that no inconsistency-inducing input is found. Thus, CIGEN is 8× faster than CIV in finding inconsistency-inducing inputs, and determines that no such input is found 33× faster. Figure 3.6 plots the running time of each of the 175 functions, regardless of whether an inconsistency is triggered. Functions are sorted in ascending order by their running time using CIGEN.

To investigate how the tools compare when time is limited, we select one of the functions with the longest runtime for both tools: the multi-parameter function `gsl_sf_hyperg_2F1_renorm`. We evaluate how much time it takes for the tools to first discover an input that causes a high inconsistency error. Figure 3.7 shows the maximum inconsistency error discovered over time. From the figure we can observe that CIGEN finds an input that triggers an inconsistency error of 63 just after 1 second. Meanwhile it takes

[1] BGRT is given a fixed budget to run (see Section 3.4.1) and thus is not included it in this comparison. Figure 3.6 includes BGRT only for reference.

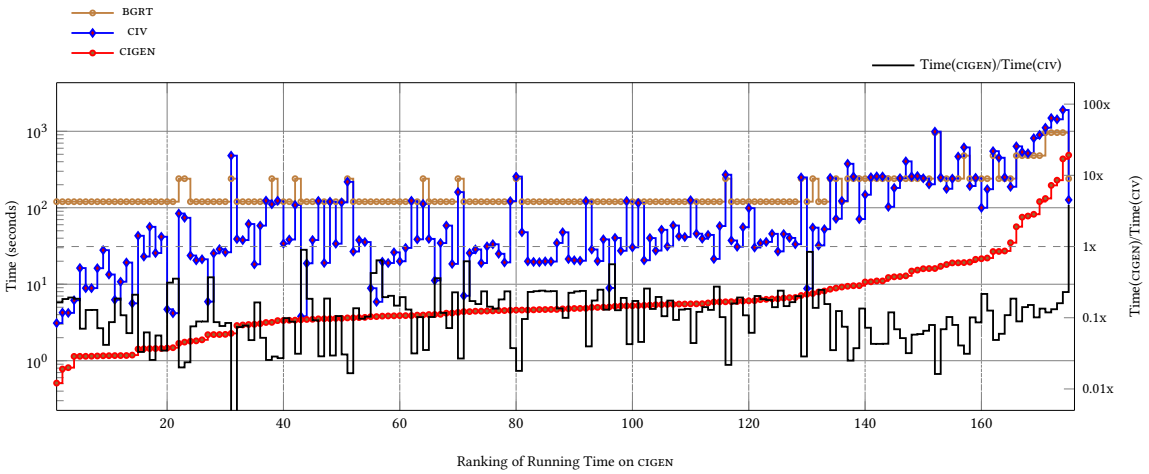


Figure 3.6: Running time for all 175 functions tested, in ascending order by the running time on CIGEN, along with the running time ratio of Time(CIGEN)/Time(CIV). Running time for BGRT is also displayed for reference.

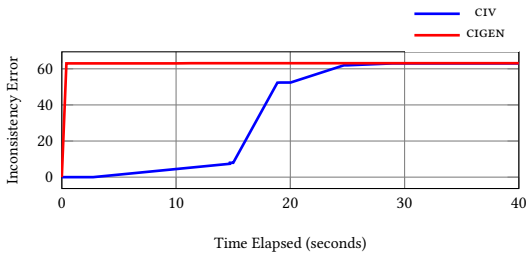


Figure 3.7: Maximum inconsistency error discovered over time for function `gsl_sf_hyperg_2F1_renorm` for the evaluated tools. Statistics after the first 40 seconds are omitted when the maximum inconsistency error for the tools no longer changes.

28.95 seconds for CIV to achieve the same result. It is possible to use CIGEN to find *an* input that triggers a compiler-induced inconsistency as high as possible within a given budget.

We found only one function, `gsl_sf_multiply`, for which CIGEN takes longer than CIV. From our observation, this is because DE is constantly updating its population with inputs that cause a minuscule amount of increase in inconsistency error, therefore the optimization algorithm takes a long time to finish. However, even when its total running time is higher, CIGEN still finds a large inconsistency-inducing input faster than CIV. Specifically, CIGEN only takes 5 seconds to find an input that causes an inconsistency error higher than 60, while CIV takes 47 seconds to achieve the same result.

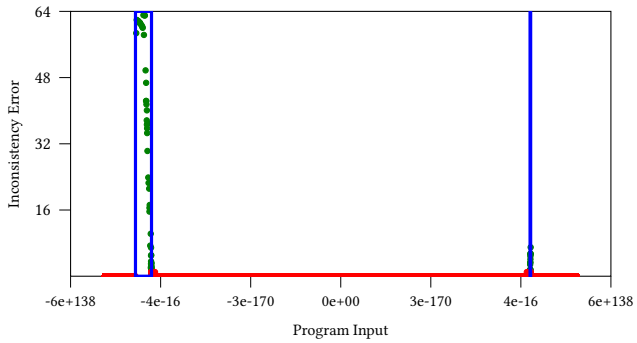


Figure 3.8: Sampled inputs, the inconsistency error triggered by these points, and detected candidate input ranges of `gsl_sf_Airy_Ai`. Red and green dots indicate sampled inputs, while blue rectangles indicate candidate input ranges.

Answer to RQ2: CIGEN shows 8× improvement over CIV in running time in finding inputs that trigger compiler-induced inconsistencies. It is 33× faster in determining that no such input is found. It also takes a shorter time than CIV to find *an* input that triggers inconsistencies as high as possible within a specified budget.

3.4.4 RQ3: Input Ranges With Inconsistency-Inducing Inputs

A unique feature of CIGEN is its ability to find input ranges (candidate input ranges, CIR) in which inputs that cause compiler-induced numerical inconsistencies are found. This section further describes the characteristics of these CIRs in terms of the number of CIRs reported by CIGEN, and the root cause of the compiler-induced inconsistencies triggered by the maximum-error input in each CIR.

Input Range Statistics. We first investigate the CIRs found by CIGEN in single-parameter GSL functions. Amongst 113 functions in which at least one inconsistency-inducing input is found, the number of candidate input ranges are between 1 and 5. A simple example function with only 1 CIR is `gsl_sf_log`, where only positive subnormal input values between [0, 2.225e-308] trigger an inconsistency. A more complex example with 2 CIRs is `gsl_sf_Airy_Ai` in Figure 3.8. We can see that the inputs we have found to cause compiler-induced inconsistencies can be grouped into two input ranges: one between [-6.63e+26, -1.90], the other between [0.63, 291.13]. For the first input range, among the dense sampling points tested (Section 3.3.3), 89.1% of sampled inputs trigger inconsistencies,

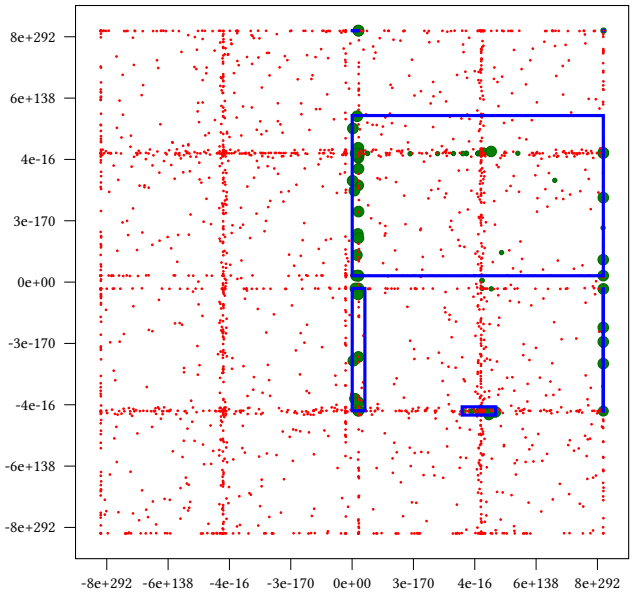


Figure 3.9: Sampled inputs and detected input ranges of `gsl_sf_bessel_Jnu()`. Red dots denote an input that does not trigger inconsistency, while green dots do. Size of the green dots denotes the inconsistency error triggered. Blue rectangles indicate candidate input ranges.

with an average inconsistency error of 46.54. For the second input range, 19.5% of sampled inputs trigger inconsistencies, with an average inconsistency error of 5.49. Developers can utilize this information when they consider improving the accuracy of a function.

In the case of multi-parameter GSL functions, the situation is more complex: among the 50 functions in which we have found inputs that cause compiler-induced inconsistencies, the number of CIRs varies between 1 and 102, with an average of 15.02 CIRs per function. The number of input ranges is especially higher amongst 3 and 4 input parameter functions, likely because there are exponentially more conditions where they satisfy the requirements of introducing a compiler-induced inconsistency. Figure 3.9 shows an example corresponding to the 2-parameter function `gsl_sf_bessel_Jnu`, for which CIGEN reports 6 CIRs. Each dot in the figure represents a sampled input. Red dots denote inputs that do not trigger inconsistencies. Green dots denote inputs that trigger inconsistencies, and their size is proportional to their inconsistency error. Each CIR is indicated by a blue rectangle.

Root Cause of Triggered Inconsistencies. To further investigate the nature of the compiler-induced inconsistencies captured by each CIR, we use the state-of-the-art tool for isolating compiler-induced inconsistencies, CIEL [31]. Given a program and an input known to

trigger a compiler-induced inconsistency in that program, CIEL applies a bisection algorithm and precision enhancement to isolate the expression(s) in the program responsible for causing such inconsistency. We consider all 70 single-parameter functions for which CIGEN reports more than one CIR, and select the input with maximum error from each CIR. We then run CIEL for each program and input. For 21 out of 70 functions, CIEL isolates distinct expressions for different inputs, suggesting that the programs suffer from multiple sources of compiler-induced inconsistency which are captured by the distinct CIRs. An example of these 21 functions is the two inconsistency-inducing inputs in `gsl_sf_Airy_Ai`, in two different input ranges shown in Figure 3.8: $x_1 = 7.663693956591586$ and $x_2 = -997404629288211.9$. For x_1, CIEL isolates a statement in the function `gsl_sf_airy_Ai_e`:

```
double s = exp(-2.0*x32/3.0);
```

Meanwhile for x_2, CIEL isolates a statement in a different function `gsl_sf_cos_e`:

```
z = ((abs_x - y * P1) - y * P2) - y * P3;
```

After investigation at the binary level, it turns out that both inconsistencies are triggered by reassociation of floating-point operations introduced by the `-ffast-math` flag: in the first case, it combines `-2.0/3.0` into one constant; in the second case, it adds `P1,P2,P3` into one constant. This example shows that finding these input ranges can help software developers isolate as much source code that causes compiler-induced numerical inconsistencies as possible, and deal with them accordingly.

On the other hand, CIEL isolated only one source of inconsistency across all CIRs in 28 functions, 25 of which have exactly 2 CIRs. This is still acceptable in common scenarios, for example, positive and negative subnormal numbers trigger an inconsistency from the same source code location, even though the inputs belong to different CIRs. In such cases it is still helpful to know about both CIRs to be able to handle both corner cases. Finally, there were 8 functions for which CIEL only isolated expressions for some inputs, and 13 functions for which CIEL was unable to isolate any expression. This does not invalidate our findings as the inputs generated by CIGEN are used to test the programs and verify the presence of a compiler-induced inconsistency. Rather, these inconsistencies may involve numbers such as

±0 and `NaN`, which remain unchanged regardless of precision and are known sources of false negatives for CIEL.

Answer to RQ3: CIGEN finds between 1 to 102 candidate input ranges for all GSL functions tested. Using the state-of-the-art tool for isolating compiler-induced inconsistencies, CIEL, we find 21 single-parameter functions where multiple sources of compiler-induced inconsistency have been captured by distinct candidate input ranges.

3.4.5 Limitations

First, our experimental evaluation only considered an assortment of GSL functions with floating-point inputs and output. The number of floating-point inputs is also relatively small, with the maximum being 4 input parameters. In terms of time complexity, the time used for CIGEN to analyze any program is a function of T and I, where T is the execution time of the program and I is the number of program inputs. For a given input $i \in I$, T could be a function of several parameters, such as the number of loops or the number of functions in a program. Thus, for programs with more inputs and longer execution time, CIGEN may still find some inconsistency-inducing inputs under time limits, but it may suffer from scalability limitations mapping CIRs. An improved algorithm that is more aware of the relationship between the source code and the compiled binaries, optimized or not, may be necessary to explore the characteristics of inconsistency-inducing inputs of such programs. Despite the above, our work represents a step forward in automated generation of inputs that trigger compiler-induced numerical inconsistencies, and our evaluation is still representative and on par with the state of the art, considering a large number of real-world functions from the popular GNU Scientific Library.

Second, CIGEN focuses on numerical inconsistencies observed in a deterministic program when compiled with different compilers and/or optimization options. CIGEN does not handle cases where the programs are non-deterministic, or when programs are subject to malicious code injections from outside sources.

Third, as mentioned throughout the chapter, we implemented CIV from scratch. In the implementation process we found that many parameters, such as how many partitions

the algorithm uses for floating-point input domain (P) or the number of tryouts for each experiment run, are not mentioned in the CIV paper. To minimize the risk of discrepancies between the original implementation and ours, we contacted the authors to request additional information about their implementation. While some questions were answered, there is still room for potential differences. In particular, the results in the CIV paper show a non-exponential relationship between the number of input parameters n and the running time of CIV of a function, but the time complexity of the CIV algorithm should be $O(P^n)$ which is exponential. This in turn means our implementation of CIV may have higher time complexity than the original implementation. The above observations make a direct comparison between CIGEN and the results in the CIV paper impossible. To mitigate this issue, in Section 3.4.2 we compared the results indirectly, first comparing between our implementations of CIGEN and CIV, then comparing our implementation of CIV against the results in the CIV paper. We also make the source code and data of our work publicly available for further research and validation.

Last, our results reveal an overlap between triggering compiler-induced inconsistencies and triggering floating-point exceptions, such as underflow and overflow. Given this, it may be interesting to see if tools such as XSCOPE [38] can be customized for our purpose.

3.5 Related Work

This section discusses related work in floating-point input generation, optimization algorithms for floating-point program analysis, and input partitioning techniques.

3.5.1 Floating-Point Input Generation to Maximize Errors

Several tools aim to find inputs that trigger high floating-point errors or inconsistencies. These tools can be categorized into the following groups:

Compiler-Induced Inconsistency Detection. CIV [32] generates inputs that cause high compiler-induced numerical inconsistencies via input space partition and Markov Chain Monte Carlo (MCMC) sampling. It compares LLVM Intermediate Representation (IR) floating-point execution traces between binary programs compiled with different compiler

and/or optimization flags in order to determine if an input is a candidate input for MCMC, but this technique limits its use to compilers based on LLVM. In contrast, CIGEN utilizes the logarithmic distribution of floating-point numbers and the importance of exponents in terms of triggering high compiler-induced numerical inconsistencies while being both architecture and compiler independent.

Precision Error Maximization. Other tools related to our work are limited to finding high floating-point precision errors in programs with a limited number of variables. S3FP [41] uses binary guided random testing (BGRT) to find inputs that trigger large precision errors in numerical programs. Input ranges for each variable are halved and for every iteration, it picks the upper or lower half of the variable range and creates random test input within these ranges. It then chooses the one range configuration with the largest error; and repeats the random testing process.

Shadow Execution Techniques. Some tools use shadow execution techniques in order to track how precision errors occur and how they are propagated and amplified throughout the program data flow. For example, FPSanitizer [44] and PFPSanitizer [23] run parts of a program in higher precision with posits [45]. The posit version of the program is run in parallel with the same code snippets in original precision, in order to facilitate the debugging of floating-point errors. RAIVE [46] on the other hand uses vectorization to implement shadow execution. Each scalar variable is expanded to a 4-dimension vector, in order to artificially inject errors that are upper bounds of actual errors; RAIVE then detects branch divergence along the execution. We have considered shadow execution during the design of CIGEN. However, we find that optimized and unoptimized versions of the same program can diverge significantly in terms of the order and the types of floating-point instructions which makes shadow execution difficult to implement.

Other Dynamic Analysis Techniques. Tools using other dynamic analysis techniques to detect precision errors include ATOMU [47], which uses the condition numbers of each floating-point atomic operation to build an error model to effectively guide the search for large precision errors. FPGen [48] on the other hand treats maximizing floating-point errors as a problem of maximizing code coverage. It uses symbolic execution with KLEE [49] on programs injected with error checks to detect precision loss and catastrophic cancellation.

Compared to FPGen, CIGEN uses a black-box approach, and is not limited by the compiler used to implement KLEE—Clang.

Our method of guided random sampling in CIGEN is similar to FPED [50]. It first generates inputs more evenly on the predefined input range, then it partitions the ranges according to error value distribution, and then generates more inputs where precision error is high. Aceso [51] uses an oracle-free approach to estimate floating-point errors. It samples the program and examines the pattern statistics (micro structures) of results of floating-point programs within an input range to estimate the floating-point errors of a program. CIGEN also uses a clustering algorithm to look for patterns in the results of floating-point programs, and also uses an oracle-free approach except when special values are involved.

3.5.2 Use of Optimization Algorithms in Floating-Point Program Analysis

Several tools use optimization algorithms to search for inputs that trigger numerical errors.

Differential Evolution and MCMC. AutoRNP [52] uses differential evolution (DE) and Monte Carlo Markov Chain (MCMC) on the condition number to detect large precision errors, uses a point-to-bound algorithm to expand the point that triggers large precision errors into an input range that triggers large errors. CIGEN also makes use of DE to find inputs that cause compiler-induced numerical inconsistencies.

Genetic Algorithms. Similar to CIGEN, genetic algorithm is used in [53] to trigger high precision errors. It uses an example to show that exponents in a floating-point input are important in determining precision errors, and runs genetic algorithm that generates and mutates exponents so that an input value that triggers high precision errors can be found. CIGEN proposes that the exponent is also important in determining compiler-induced numerical inconsistency, and is implemented based on this idea.

Bayesian Optimization. XSCOPE [38] utilizes Bayesian optimization to identify inputs that cause floating-point exceptions. Both XSCOPE and CIGEN have the advantage of not requiring the source code of the program to function, and thus not limited to compilers that have source-to-source components.

3.5.3 Input Partition

Input partitioning is a technique for understanding how numerical behavior varies across input domains. Herbie [34] is one of the first tools that perform input partition. It first randomly samples program inputs and uses these sample inputs to gauge the extent of precision improvement, and then rewrites numerical expressions under an input range partition scheme (called regime), so that precision can be improved across the whole input domain of an expression. A newer tool, Regina [54], is based on Herbie, and introduces a regime inference algorithm for a multiple of input parameters within respective input ranges. It uses a two-phase, bottom-up then top-down approach to find the most efficient regimes, which are then used in various optimization problems, such as mixed precision tuning, or program rewriting to improve precision.

3.6 Conclusion

We proposed in this chapter a multi-phase approach to generate input ranges that trigger compiler-induced numerical inconsistencies by combining input-partitioned and coverage-based input sampling, input clustering, and optimization algorithms. We implemented our approach in the tool CIGEN. Our experimental evaluation showed a 53.4% improvement over the state of the art in detecting inputs triggering high inconsistency errors in 175 GSL functions 8× faster. Further analysis of a subset of inconsistencies revealed their characteristics and possible root causes, which can be helpful for developers to determine the best course to mitigate numerical issues caused by such inconsistencies.

Chapter Acknowledgements

This chapter is based on the paper "Input Range Generation for Compiler-Induced Numerical Inconsistencies" by Dolores Miao, Ignacio Laguna, and Cindy Rubio-González, presented at *the 38th ACM International Conference on Supercomputing (ICS)*, 2024. The code and data for CIGEN are publicly available at https://github.com/LLNL/CIGEN/.

Funding: This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-859884), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under awards DE-SC0022182 and DE-SC0020286, and the National Science Foundation under award CCF-1750983.

Chapter 4

FloatGuard: Detecting Floating-Point Exceptions via Runtime Testing

4.1 Problem and Motivation

This chapter addresses the challenge of detecting floating-point exceptions in AMD GPU programs, which is a critical need as HPC systems increasingly adopt AMD GPU architectures.

The AMD GPU Challenge. While NVIDIA has long dominated the GPU market, AMD has made significant strides in recent years and has started to be used in many HPC clusters. Exascale computing platforms from the US Department of Energy (DOE), such as El Capitan and Frontier—the two fastest supercomputers[1]—use AMD GPUs (MI300A and MI250X GPUs, respectively).

Scientific applications rely on reproducibility across all architectures. If floating-point exceptions such as NaNs are not detected and handled, they can lead to inconsistent results across different runs of the same computation, undermining the credibility of the results. Previous studies indicate that significant numerical differences may arise when code is compiled and executed on NVIDIA GPUs and later run on AMD GPUs, or the reverse [55]—

[1] https://top500.org/lists/top500/2024/11/

these differences include the occurrence of exceptions in GPU A and normal results in GPU B, even when using the same program inputs.

Limitations of Existing Approaches. Previous works on detecting floating-point exceptions have focused on NVIDIA GPUs, using various methods such as compiler instrumentation [56, 57] or binary instrumentation [58, 59]. This presents a significant gap in the ability to detect and analyze floating-point exceptions in AMD GPU programs which, as mentioned above, are increasingly prevalent in HPC systems.

A study of floating-point exceptions detected in a comprehensive set of CUDA programs is presented in [59]; however, currently, no tools or frameworks are available to enable a comparable study for AMD (or HIP) programs. Developing such a tool is far from trivial and requires addressing several challenges: (1) Reverse engineering the low-level mechanisms that the AMD architecture provides to enable exception traps (if they exist); (2) Enabling such mechanisms efficiently in all GPU kernels. (3) Providing the ability to detect *all* exceptions while ensuring correct execution.

Our Approach. We present FloatGuard, the first tool to detect floating-point exceptions in HIP programs running on AMD GPUs. FloatGuard relies on the AMD GPUs hardware registers (e.g., trap status and mode registers) to enable floating-point exception detection and efficiently gather and analyze all the exceptions.

While the mechanism provided by AMD to enable such traps is useful, it is not practically feasible to use it to detect exceptions in *all* kernels and to ensure correct execution when such modes are enabled. If a developer enables the provided trap manually (e.g., with the ROCgdb debugger), it must be done for *every* kernel thread, and it must be enabled at the beginning of the kernel, which would only trap *one* exception, namely, the first one that occurs in the kernel.

FloatGuard addresses these challenges using code instrumentation at two levels—assembly-level and source-level—and a debugger-guided execution that runs iteratively until *all* floating-point exceptions have been reported.

Contributions. The contributions of this chapter are:

- The first approach for detecting all floating-point exceptions in HIP programs running on AMD GPUs, which utilizes registers provided by the GPU hardware, assembly-level

and source-level code instrumentation, and debugger-guided execution (Sections 4.2 and 4.3).

- An implementation of our approach in the tool FloatGuard, which requires a minimal number of modifications to program build systems (Section 4.3).
- An evaluation on 565 HIP programs, where FloatGuard detects floating-point exceptions in 507 out of 565 programs, with a slowdown ratio that increases at most linearly with the number of exceptions found (Section 4.4.2).
- An empirical analysis of the impact of compiler optimizations on floating-point exceptions showing a decrease of exceptions when using aggressive compiler optimizations (Section 4.4.3).
- A comprehensive comparison of floating-point exceptions triggered in both AMD HIP and NVIDIA CUDA versions of the same program, revealing differences in numerical characteristics between the two platforms (Section 4.4.4).

4.2 Background

4.2.1 AMD GPU Toolchain

AMD provides a software stack called ROCm [60] to enable developers to create, compile and run programs on AMD GPUs. Apart from kernel drivers and low level libraries, ROCm provides a programming API and kernel language called HIP. A HIP program has a structure similar to that of CUDA programs on NVIDIA GPUs, where host code runs on CPUs and may call AMD GPU kernels to run on the GPU device. These kernels can create numerous GPU threads that run in parallel.

An AMD HIP program is compiled into host and device binaries using a compiler interface—HIPCC—which calls an underlying Clang compiler with an AMD GPU backend. In terms of debugging, ROCm offers ROCm Debugger [61] (ROCgdb), the AMD source-level debugger for Linux based on GDB [62], which provides support for debugging AMD GPU kernels. AMD ROCm supports a range of AMD GPUs with GCN, RDNA and CDNA architectures. FloatGuard is tested to work with both RDNA2/CDNA2 GPU architectures [63]. The evaluation of FloatGuard in Section 4.4 is performed on an RDNA2 GPU, but our approach

is also applicable to other existing or future RDNA/CDNA architectures as long as they keep the existing mechanism to access the same set of registers related to floating-point exceptions.

4.2.2 Floating-Point Exceptions

Given the limits of IEEE floating-point types in representing the infinite set of real numbers, IEEE 754 defines certain situations, where the limits of these types are reached, as *floating-point exceptions*. Hardware and software manufacturers may define additional types of floating-point exceptions beyond the IEEE 754 standard. The AMD RDNA2/CDNA2 GPU architecture defines seven types of floating-point exceptions, listed in Table 4.1. In addition to the 5 types of floating-point exceptions defined by IEEE 754, AMD GPUs also define two additional exception types: input denormal, and integer divide by zero, which are also useful in analyzing abnormal numerical program behavior. FloatGuard focuses on detecting all floating-point exceptions listed in Table 4.1 except for the inexact exception, which is a rounding error that occurs in virtually all numerical programs.

Table 4.1: Exception types in AMD GPUs. An asterisk in the exception type denotes that it is an AMD-specific exception.

Exception Type	EXCP bit	Description
invalid operation	0	Result of an operation is not-a-number (NaN), such as $0/0$, $\infty/0$, or $\infty \mod 0$.
input denormal*	1	At least an operand is a subnormal number.
divide by zero	2	Division of a non-zero floating-point number by floating-point zero.
overflow	3	Absolute value of an operation result exceeds maximum value in floating-point type.
underflow	4	An operation result is a subnormal number.
inexact	5	Operation result cannot be precisely represented and rounding is involved.
integer divide by zero*	6	Division of a non-zero integer number by integer zero.

The RDNA2/CDNA2 GPU architecture provides a 32-bit mode register to check and control whether floating-point exceptions are trapped or not in a GPU kernel program. The layout of the mode register is presented in Table 4.2, adapted from [63]. Bits 12-18 in the mode register set which exception types are enabled. Additionally, there is also a trap status register, with its `EXCP` bits 0-6 (corresponding to bits 12-18 in the mode register) set when

Table 4.2: Bit ranges of the 32-bit mode register (value 0x5F2F0).

Bit Range	Purpose	Value
0–3	Round mode	0000
4–7	Denormal mode	1111
8	DX10_CLAMP	0
9	IEEE	1
10–11	Reserved	00
12–18	Exception bits	1011111
19–31	Others	0000000000000

```
1 #include "hip/hip_runtime.h"
2
3 __global__ void k_fp(int *mem, float a, float b) {
4   int fret = a / b;
5   *mem = fret;
6 }
7
8 __global__ void k_mixed (int *mem, int a, int b) {
9   int fret = a / b;
10  fret = fret + (float)a / (float)b;
11  *mem = fret;
12 }
13
14 int main (int argc, char* argv[]) {
15   int *mem;
16   hipMalloc(&mem,4);
17   k_fp<<<1,1>>>(mem,1,0);
18   hipDeviceSynchronize();
19   k_mixed<<<1,1>>>(mem,1,0);
20   hipDeviceSynchronize();
21   return 0;
22 }
```

Figure 4.1: A sample HIP program for testing AMD GPU kernel floating-point exception behavior, where the lines of code triggering exceptions are highlighted.

a floating-point exception of a respective type is triggered during kernel execution. If the GPU kernel is set to trap these exceptions during runtime, program execution stops when an exception occurs, and a floating-point exception signal (SIGFPE) is sent; if the GPU kernel is set to not trap these exceptions, execution continues when an exception occurs, and the corresponding `EXCP` bit in the trap status register is set. By default, bits 12-18 in the mode register are all set to 0 (as in disabled) when each kernel starts execution; to enable them so that floating-point exceptions can be trapped, these mode register bits must be set to 1. In addition, to achieve IEEE 754 compliance, bits 8 (`DX10_CLAMP` bit: DX10-style treatment of NaNs) and 9 (`IEEE` bit) should be set to disabled and enabled, respectively.

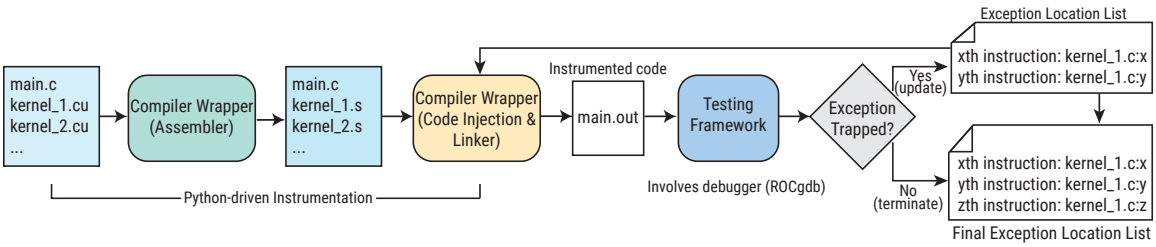


Figure 4.2: The workflow of FloatGuard.

To observe the runtime floating-point exception characteristics of AMD GPUs, a preliminary test is conducted using ROCgdb on a sample program listed in Figure 4.1. The program triggers four floating-point exceptions: a divide by zero floating-point exception in function `k_fp` in line 4, two divide by zero exceptions in function `k_mixed` in lines 9 and 10, and an invalid exception from the addition in line 10. Breakpoints are set at the beginning and end of each kernel. At the start of each kernel, the values of the mode and trap status registers are recorded. If floating-point exceptions are enabled for the kernel, the mode register is updated accordingly. The value of the trap status register is then observed either at the end of the kernel or when an exception occurs.

Three preliminary observations can be drawn from the above example. (1) By default, exception trapping is disabled in AMD GPUs; theoretically, developers can check the trap status register at the end of each kernel's execution to know which exception occurred, however, it is not possible for developers to know which line of code caused the exception, and it is impractical for developers to check the trap status of each GPU thread. (2) The memory location where the program counter stops after a floating-point exception is trapped does not always correspond to the actual instruction that causes the exception, possibly due to delayed exception reporting on GPUs. (3) Once a floating-point exception is triggered in a kernel, the state of the program is not recoverable. This means that even if the SIGFPE signal is cleared and program execution continues, the program state after this point does not exactly correspond to when floating-point exceptions are disabled. These observations about AMD HIP program execution on AMD GPUs influence the design and implementation of FloatGuard.

4.2.3 Overview of FloatGuard

The workflow of FloatGuard is illustrated in Figure 4.2. FloatGuard consists of two main components: a compiler wrapper and a testing framework.

The **compiler wrapper** acts as an in-place replacement of the HIPCC compiler interface. When an HIP source file is compiled using the compiler wrapper, it replaces output object files with text-based assembly files instead; then, during link time, the compiler wrapper injects input assembly files with instructions for enabling or disabling floating-point exceptions at injection points in a list maintained by both modules in FloatGuard, then calls the actual link command to generate the instrumented binary. The instrumented program is then sent to the **testing framework**, which runs the instrumented program, gathers information on exceptions encountered, then updates the list of injection points as mentioned above, to ensure that FloatGuard can report all floating-point exceptions observed in an AMD HIP program at runtime.

A typical usage scenario of FloatGuard is as follows: a software developer would like to know whether their HIP program that runs on an AMD GPU suffers from any floating-point exceptions. The developer then needs to modify their build script by simply replacing the original compiler with our compiler wrapper, and enabling debug information in the build by using the `-g` flag in the compiler command line. The developer can optionally select the types of exceptions to be captured, as some types of exceptions, such as input denormal, may be of less interest because of their lower impact towards numerical correctness of the program. Finally, the developer can invoke FloatGuard, which will test the program to output the source code locations that trigger floating-point exceptions during runtime. The developer can then inspect the source code and devise solutions to mitigate the exceptions, such as reordering operations or increasing floating-point precision.

The next section presents the design and implementation of these modules and how they interact to detect floating-point exceptions.

4.3 Design of FloatGuard

This section describes the design and implementation of FloatGuard for detecting floating-point exceptions in AMD GPU kernels.

4.3.1 Floating-Point Exception Detection Control

As mentioned in Section 4.2.2, setting the individual bits in the mode register in each kernel thread allows a floating-point exception to be trapped or not during program execution. A method to write to the mode register is via debugging commands using the ROCgdb debugger. However, this method requires using breakpoints to pause the program at the beginning of every GPU kernel thread execution. For most AMD GPU kernels, thousands of GPU threads run in parallel. To set the mode register for each GPU threads, the ROCgdb debugger must stop the program thousands of times just for one kernel, which introduces significant overhead (according to our testing), and makes the tool impractical to use.

Assembly Injection. An improved solution is to inject assembly instructions that write to the mode register during program compilation. Below are two sets of instructions that set the mode register. They are interchangeable, but FloatGuard uses them both for the two kinds of code injections described in Section 4.3.2. The first set of instructions moves an immediate value, first into a scalar general purpose register `s31`, which has no specific reservations in kernel code, then into the mode register.

```
1    s_mov_b32 s31, 0x5F2F0
2    s_setreg_b32 hwreg(HW_REG_MODE), s31
```

On the other hand, the second set of instructions directly assigns values to the mode register. Since only 16-bit values are allowed, FloatGuard uses two instructions for the value assignment.

```
1    s_setreg_imm32_b32 hwreg(HW_REG_MODE, 0, 16), 0xF2F0
2    s_setreg_imm32_b32 hwreg(HW_REG_MODE, 16, 16), 0x5
```

FloatGuard also uses a similar instruction to set the EXCP bits (0-6) in the trap status register to all zero. This instruction is injected whenever floating-point exceptions

are enabled during program execution, so that only the floating-point exception type that is trapped is recorded.

```
1    s_setreg_imm32_b32 hwreg(HW_REG_TRAPSTS, 0, 7), 0
```

A naive method for detecting floating-point exceptions with these instructions is to simply enable floating-point exceptions at the *beginning* of all AMD GPU kernels. However, as mentioned in Section 4.2.2, it is not possible to recover an AMD HIP program from a trapped exception. Thus, if FloatGuard simply enables exceptions at the beginning of each kernel call, the program would stop at the point where the first floating-point exception is triggered, irrevocably, and no further exception would be triggered. To work around this issue, FloatGuard injects code in multiple locations of the program. This is further explained in the next section.

4.3.2 Compiler Wrapper and Code Injection

To ensure that floating-point exceptions are enabled or disabled in the original program, injecting exception control instructions (discussed in Section 4.3.1) is necessary. The compiler wrapper module performs such injection alongside with program compilation. It injects code in the two kinds of locations listed below:

- **Entry points:** at the beginning of each AMD GPU kernel, instructions to enable floating-point exceptions are injected. This ensures that exception capture is always enabled when an AMD GPU kernel starts execution.

- **Previously-triggered exception locations:** the compiler wrapper may also accept a list of assembly code locations at which previously triggered exceptions have been observed. The compiler wrapper injects exception disabling instructions before, and exception enabling instructions after these locations, so that no exceptions are triggered when the instructions within these locations are executed. This is to prevent previously-triggered exceptions from being triggered again. Additionally, the trap status register is cleared when floating-point exceptions are re-enabled, because the trap status register is still set when an instruction that causes a floating-point exception at the location is executed. Clearing the trap status register ensures that the next floating-point exception trapped will return the correct exception type.

Two widely-used methods for code injection in floating-point research are Clang plugins and LLVM passes. Calls to a function containing the inline instructions from Section 4.3.1 can be injected either into the source code, or into its LLVM intermediate representation (IR). There are advantages and disadvantages of these two methods. For example, Clang plugins apply source-to-source transformations and thus are compiler agnostic; meanwhile LLVM passes can be adapted to other language front-ends that HIP supports, such as HIP FORTRAN. However, during the implementation of FloatGuard, we found that neither Clang plugins nor LLVM passes can solve the problem of instruction reordering: optimization passes in the AMD GPU back-end of the default HIPCC compiler reorders assembly code *after* code injection is performed. In particular, compiler optimizations may alter code regions bracketed by function calls enabling and disabling floating-point exceptions, and undermine their effectiveness. Unfortunately, using software barriers does not appear to work as intended either, as their presence may also interfere with optimization passes and lead to numerical behaviors different from those observed without them.

Assembly Files Injections. To solve the above issue, FloatGuard instead implements code injection by directly inserting assembly instructions into assembly files, after all compilation and optimization steps are performed, but before the linker combines all assembly files into a single binary executable. This method combines the advantages of both previously mentioned methods: it does not require source code to inject code into the program, and thus it works regardless of what kind of high-level language features exist in the program source code; it also preserves source location information when exceptions occur, as long as such debug information is built into the assembly files from compiler options `-g` in HIPCC; it is compiler and language agnostic, as it uses Python scripts to read, process, and modify AMD kernel assembly code, without any other software dependency; it is transparent to the end user, as it does not require them to drastically modify the build process of their program, and all the resulting instrumented code exists in a human-readable assembly form; and lastly it can be easily adapted to a different AMD GPU architecture as long as the equivalent of mode register control instructions are still available.

Instructions controlling the mode register are injected as follows. First, an assembly file is read, and the compiler wrapper scans for the listing of AMD GPU kernel functions, which

Algorithm 3: The testing algorithm in FLOATGUARD.

```
1  Function TestAlgo(Program):
2      ExceptionLocations = ∅;
3      while ProgramFinished == false do
4          InstrumentedProgram = Program;
5          foreach Location ∈ ExceptionLocations do
6              InstrumentedProgram =InjectException (InstrumentedProgram, Location);
7          RunProgramUntilStop (InstrumentedProgram);
8          if InstrumentedProgram has finished running then
9              ProgramFinished = true;
10         else
               // signalled exception
11             ExceptionLocations = ExceptionLocations∪GetExceptionInfo ();
12             OutputExceptionInfo ();
```

is marked by `__CLANG_OFFLOAD_BUNDLE` markers; then the compiler wrapper scans for the beginning of a kernel function, which is marked by a mangled function name beginning with `_Z`. The instructions are then injected into the body of these kernel functions.

An additional issue arises from the delayed exception reporting problem described in Section 4.2.2. To address this, FLOATGUARD associates each trapped exception with a *set* of instructions rather than one instruction. We have empirically found that delayed exception reporting often is off by at most 5 instructions. Thus, FLOATGUARD associates a reported exception with the instruction at which execution halted and its four preceding instructions. These are treated as a set of instructions for which floating-point exception should be disabled in subsequent testing. Consequently, the compiler wrapper injects instructions to disable exceptions before, and to re-enable them after this set of instructions.

4.3.3 Testing Framework

FLOATGUARD runs the instrumented program and aims to capture all floating-point exceptions in AMD GPU kernels. This testing component is built with Python spawning a ROCgdb instance with the `pexpect` library, which attaches to the instrumented program and controls whether floating-point exceptions are enabled/disabled before every AMD GPU kernel's execution. When a floating-point exception is trapped, the testing framework outputs the crash site information such as source file and lines, function name, adjacent assembly instructions, and exception types.

The testing algorithm of FloatGuard is illustrated in Algorithm 3. It is designed and fine-tuned to adapt to the characteristics of AMD GPU kernel behavior with floating-point exceptions described in Section 4.2.2. The detailed steps are as follows:

▷Step 1: Initially, FloatGuard compiles the original program using the compiler wrapper, inserting instructions at the beginning of each AMD GPU kernel to enable exceptions for all code inside all HIP kernels, then attaches ROCgdb to the instrumented program. Also, the list of injection sites that trigger floating-point exceptions is set to empty. Furthermore, the linker command used during the build process is saved separately so that future injection of assembly instructions can be performed without running the whole compilation process again, improving the efficiency of the tool.

▷Step 2: If no exception is found and the program exits normally, FloatGuard finishes running and reports that no floating-point exception is found; otherwise, when a floating-point exception occurs and ROCgdb stops, it parses and outputs exception information from ROCgdb output. Information that is saved includes source code location, the number of instructions from the beginning of the function to the instruction that the program stops at, and the exception type. If the source code location is from an AMD HIP library, FloatGuard goes up the call stack until it finds a source code location that is from the program source code rather than from an existing library, and saves this location as well.

▷Step 3: FloatGuard also records the location of the exception-occurring kernel in the execution trace (kernel name and the number of instructions from the beginning of the function to the exception site) to a list of injection sites.

▷Step 4: FloatGuard then runs the linker command saved in step 1; this time the compiler wrapper reads the list of injection sites saved in Step 3, and injects code around these sites, so that floating-point exceptions from instructions around the injection sites are not trapped the next time, since these sites are already recorded as locations that trigger exceptions.

▷Step 5: The program is then restarted with ROCgdb, and return to Step 2. As more sites that trigger floating-point exceptions are found and recorded, all code that triggers such

exceptions will eventually be included in the list of injection sites; at this point, the program finishes without triggering any other exceptions.

Consider again the sample program from Section 4.2.2. Exceptions are enabled at the beginning of both kernels in the program. When the program is run with ROCgdb attached, an exception occurs in `k_fp` line 4, whose function location is recorded in the list of injection sites. The linker command is executed again, and the compiler wrapper injects code around the location previously recorded so that floating-point exceptions are disabled before reaching the location, and enabled after it. The program is re-run with ROCgdb attached, rinse and repeat, until the program no longer triggers any exception before exiting normally. The list of injection sites then corresponds to the list of floating-point exceptions detected.

Slowdown. FloatGuard undoubtedly introduces slowdown, given the overhead introduced by the various components, and the need to run the program multiple times to detect all floating-point exceptions. The slowdown introduced by FloatGuard consists of the following parts: spinning up ROCgdb and running the program every time it needs to recover from a previous floating-point exception; collecting exception information when it is trapped; and running the linker every time a new exception location is added. The overhead introduced by the instructions that set the mode and trap status registers is negligible. The slowdown ratio can be calculated using the following equation:

$$\text{Slowdown} = \frac{\sum_{i=1}^{N} \mathrm{P_i} + \mathrm{OH} \cdot (\mathrm{N}+1) + \mathrm{P}}{\mathrm{P}} \tag{4.1}$$

where P is the program running time; P_i is a strictly increasing sequence of the time of the ith run of the program, from beginning of program execution to when the exception occurs; OH is a relative constant cost of the overhead for each re-run of the program, including time running the linker, spinning up ROCgdb, and collecting exception information after it occurs, all of which are constants depending on the program; and N is the number of floating-point exceptions triggered. When P is small (less than 1s in programs tested in Section 4.4.2), OH is relatively large compared to P, and the slowdown ratio increases in an approximately linear manner with the number of floating-point exceptions. Meanwhile, when P is large,

OH is relatively small, and the relationship between slowdown ratio and number of exception detected depends on the ratio between P_i and P, which is discussed in Section 4.4.2.

4.4 Experimental Evaluation

This evaluation answers the following research questions:

RQ1 How many floating-point exceptions are detected by FloatGuard in AMD HIP programs, and how much slowdown does FloatGuard introduce?

RQ2 How do compiler optimization options affect the number of floating-point exceptions detected by FloatGuard?

RQ3 How do floating-point exceptions differ when the same GPU programs are executed on AMD HIP and NVIDIA CUDA?

4.4.1 Experimental Setup

Table 4.3: Benchmark programs tested with FloatGuard to detect floating-point exceptions.

Benchmark Set	Benchmarks
Rodinia	backprop, cfd, gaussian, heartwall, hotspot, hotspot3D, lavaMD, lud, myocyte, nn, nw, particlefilter, streamcluster
PolyBench-ACC	correlation, covariance, 2mm, 3mm, atax, bicg, doitgen, gemm, gemver, gesummv, mvt, syr2k, syrk, gramschmidt, lu, adi, convolution-2d, convolution-3d, fdtd-2d, jacobi-1d-imper, jacobi-2d-imper
Parboil	bfs, cutcp, histo, lbm, mri-gridding, mri-q, sad, sgemm, spmv, stencil, tpacf
SHOC	bfs, fft, gemm, md, md5hash, reduction, scan, sort, spmv, stencil2d, triad, qtclustering, s3d
GPGPU-Sim	cp, libor, lps, mum, rayTracing, wp
HPCG	HPCG
Varity	500 randomly-generated programs containing numerical inconsistencies

Benchmark Selection. FloatGuard is evaluated on 565 programs, which are categorized in Table 4.3. The first 65 programs are from five heterogeneous benchmark suites: 13 programs are from Rodinia [64], where only programs with floating-point operations are

selected; 21 programs are from PolyBench-ACC [65]; 6 programs are from benchmarks distributed with the GPGPU-Sim simulator [66]; 11 programs are from the Parboil benchmark suite [67]; 13 programs are from the Scalable HeterOgeneous Computing (SHOC) benchmark suite [68]; And lastly, we also include the HIP version of the HPCG benchmark [69] based on the original HPCG implementation [70]. These benchmarks cover a wide range of uses for heterogeneous computing, such as linear algebra for machine learning, data mining, medical imaging, and fluid dynamics.

We also include 500 programs generated using Varity [16], a floating-point program random generator. The programs are publicly available in Varity's GitHub repository[2] and have been previously identified as exhibiting numerical inconsistencies when compiled with different optimization settings—specifically, comparing `-O3 -use_fast_math` against `-O0` using the CUDA NVCC compiler. These inconsistencies often arise from compiler optimizations that affect the precision, associativity, or evaluation order of floating-point operations. Such optimizations can also influence whether a floating-point exception is triggered. By including these synthetic programs, we aim to capture edge cases where aggressive optimization interacts with exception conditions in a nontrivial fashion, and may not be well represented in real-world HPC workloads but are critical for evaluating the robustness and coverage of floating-point exception detection tools.

All programs in this evaluation, with the exception of ROC HPCG, are only available as CUDA programs; thus we convert them for AMD HIP using the HIPIFY tool [71] and then fix minor issues manually, so that the end results of the program are either comparable with the original CUDA versions, or they pass in-built verification routines, when available.

Benchmark Parameters. Some benchmarks, such as the Varity synthetic programs, and the programs from GPGPU-Sim and SHOC, have no selection of input parameters and datasets; these benchmarks are run once with the default parameters. While others, such as the `cfd` program in Rodinia, have multiple input datasets; some other benchmarks, such as all PolyBench-ACC programs, can be run with various input data sizes. In our evaluation we do the following: for Rodinia, all inputs listed in the `run` script in each program are included; for PolyBench-ACC, all problem sizes from mini, small, default, large to extra large sizes for

[2] https://github.com/LLNL/Varity/tree/master/cuda_examples

each benchmark are included; for Parboil, all input datasets are included; and for HPCG, we use the parameter `128 128 64`.

Baselines. FloatGuard is the first tool that detects floating-point exceptions in AMD HIP programs. There are works detecting floating-point exceptions in other architectures such as x64 and CUDA, which are described in Section 4.5.1, and we include a cross-platform comparison for the same programs between AMD and NVIDIA GPUs in Section 4.4.4.

Runtime environment. We run FloatGuard on a workstation PC, running with a 6-core AMD(R) Ryzen 5 7500F processor, 32 GiB RAM, and an AMD RX 6650 XT GPU with 8GiB of VRAM. The workstation PC runs Ubuntu 22.04 OS, with AMD ROCm 6.1.0 installed. FloatGuard is run on Python 3.10.12 with pexpect 4.9.0 for automatically controlling ROCgdb process input and output, with no other external dependencies.

4.4.2 RQ1: Exception Detection Effectiveness

Table 4.4: A list of benchmarked programs that have floating-point exceptions detected by FloatGuard. The exception type abbreviations NAN, I_SUB, DIV0, INF, and O_SUB correspond to exception types invalid operation, input denormal, divide by zero, overflow, and underflow, respectively. No integer divided by zero exceptions were detected.

Benchmark	Exceptions	NAN	I_SUB	DIV0	INF	O_SUB
Rodinia: cfd	12	6	0	6	0	0
Rodinia: myocyte	50	26	3	0	15	9
PolyBench-ACC: correlation	1	0	0	1	0	0
PolyBench-ACC: gramschmidt	2	1	0	1	0	0
PolyBench-ACC: lu	1	1	0	0	0	0
PolyBench-ACC: adi	4	4	0	0	0	0
SHOC: s3d	823	6	681	0	7	264
Parboil: stencil	2	0	1	0	0	1
GPGPU-Sim: wp	68	2	50	0	5	18

In the 65 benchmark programs evaluated, FloatGuard detects floating-point exceptions in 9 of them, spanning across 5 benchmark suites. The number of each type of floating-point exceptions detected by FloatGuard in each of these 9 programs are listed in Table 4.4. Note that the number of total exceptions may be smaller than the sum of all types of exceptions; this phenomenon occurs in `myocyte`, `s3d` and `wp`, in which input denormal exceptions occur.

This is due to the fact that one floating-point operation may trigger both input denormal and other exceptions, for example, dividing a subnormal number with zero.

Also, amongst these programs, there is only one—`stencil` in Parboil—which FloatGuard only detects floating-point exceptions related to subnormal numbers. Meanwhile, all other 8 benchmarks have exceptions related to special values—Inf and NaN—which are considerably more serious floating-point exceptions and have a higher impact on the precision error of these programs.

In the 500 synthetic GPU programs, FloatGuard detects floating-point exceptions in 498. If only exceptions related to special values are counted, 291 programs trigger them. On average, 6.056 floating-point exceptions are detected per program, or 2.088 when considering only those related to special values.

As for the remaining 2 programs which do not have floating-point exceptions detected: the first one, `case_152`, has its set of input parameters all in normal range (which means none of them is subnormal), and all subnormal and division by zero operations are constant, which are likely pre-calculated during compile time. Meanwhile, the other program, `case_473`, has a `compute` function that looks like the following excerpt:

```
1 for(int i=0;i<var_1;++i){
2   comp=var_2*1.5271E-42f*(var_3/floorf(-1.5532E35f-1.8528E35f-var_4*-1.4807E36f));
3   comp=var_5+1.1129E35f+var_6/powf(expf(1.4350E-36f),var_7/-1.7503E11f*var_8/1.5272E
        -19f);
4 }
```

The expression in line 2, from its variable to constant operands, all contain floating-point exceptions; but the whole expression is removed from the program binary due to dead code, and only the expression in line 3 is kept. The operands in this expression, from `var_5` to `var_8`, are all normal numbers, and the whole expression is indeed not expected to produce any floating-point exceptions. We confirm our hypothesis for the two programs by running FloatGuard again but with the programs compiled with `-O0` instead, which does not perform either compile-time constant calculation or dead code removal. FloatGuard successfully detects floating-point exceptions in this case.

In terms of slowdown ratio, first for the 65 benchmark programs: the original runtime for these 65 programs ranges from 0.17s which includes the spin-up time for AMD HIP libraries,

to 196s in the case of the `correlation` program in PolyBench-ACC. The slowdown ratio introduced by FloatGuard ranges from 1.0× for some benchmarks where no floating-point exceptions are detected, to 6888× in the case of `s3d` from the SHOC benchmark, in which the original program runtime is 0.2s, and 823 floating-point exceptions are detected. As for the 500 synthetic GPU programs, the runtimes for these programs are all between 0.17s and 0.18s, and the slowdown ratio introduced by FloatGuard ranges from 2.53× for `case_152` where no floating-point exception is detected, to 118× for `case_365` where 30 floating-point exceptions are detected.

The slowdown ratios for 564 out of 565 programs[3] are shown in Figure 4.3. We use linear regression to find the line of best fit, and find that for most programs that run for a short time (less than 1s), their slowdown ratios are approximately linearly related to the number of exceptions detected, which corroborates the assertions made in Section 4.3.3. On the other hand, for programs running for a longer period of time, the slowdown ratio is sublinearly related to the number of exceptions, since the exceptions occur early during program execution, therefore $\sum_N P_i$ in Equation (4.1) is negligible compared to P. The sole outlier is `s3d` which has a higher slowdown ratio due to its longer linker time (1s) compared to shorter running time (0.2s).

Answer to RQ1: FloatGuard successfully detects floating-point exceptions in 9 out of 65 benchmark programs. For the 500 synthetic GPU programs, FloatGuard detects exceptions in 498 programs (99.6%), with an average of 6.056 exceptions per program. The slowdown introduced by FloatGuard is linearly related to the number of exceptions detected for short-running programs, and sublinearly related for longer-running programs.

4.4.3 RQ2: Effect of Aggressive Floating-Point Optimization Flags on Exceptions

The impact of compilers and their optimization flags on the correctness of heterogeneous numerical programs has been well documented [72], and the underlying causes of these effects have been studied in [2, 5, 31]. Optimization flags related to floating-point operations can

[3]Except `s3d` which has over 800 exceptions detected and is not shown here to preserve the clarity of the chart.

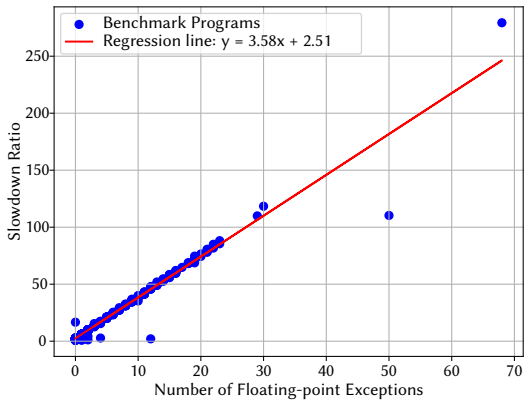


Figure 4.3: Slowdown ratio compared to the number of floating-point exceptions in the programs tested.

Table 4.5: How the number of floating-point exceptions detected changes when `-ffast-math -fdenormal-fp-math=preserve-sign` is enabled in the compiler. The exception type abbreviations are the same as in Table 4.4.

Benchmark	Exceptions	Exc. (excl. SUB)	NAN	I_SUB	DIV0	INF	O_SUB
cfd	12 → 14	12 → 14	6 → 8	0	6	0	0
myocyte	50 → 28	41 → 23	26 → 24	3 → 0	0	15 → 2	9 → 2
correlation	1	1	0	0	1	0	0
gramschmidt	2	2	1 → 0	0	1 → 2	0	0
lu	1	1	1	0	0	0	0
adi	4 → 6	4 → 6	4 → 2	0	0 → 4	0	0
s3d	823 → 739	13 → 5	6 → 0	681 → 608	0	7 → 5	264 → 309
stencil	2 → 3	0	0	1	0	0	1 → 2
wp	68 → 87	4	2 → 1	50 → 78	0	5 → 3	18 → 16

greatly affect the occurrence of floating-point exceptions in a numerical program. One of the main optimization flags related to floating-point operations in the HIPCC compiler is `-ffast-math`, which does the following:

1. Floating-point operations are treated as infinite-precision real numbers in terms of algebraic rules such as associative and distributive properties. This may negatively impact floating-point round-off errors, which could cause catastrophic cancellation or loss of significance for some floating-point operation reassociations.

2. No NaN or infinite values will be operands or results of floating-point operations. However, this does not mean that floating-point hardware does not generate floating-point exceptions about NaN or infinite values internally. These exceptions are still recorded on AMD GPUs.

3. Fast approximation math functions, which are less accurate, are applied instead of the IEEE 754 standard versions. Fast constant reciprocal instructions are also applied.

If developers use optimization flags such as `-ffast-math`, not only may their program have higher precision errors, they may also encounter different floating-point exceptions to be triggered, due to changed results in operations and these results possibly affecting program execution paths. Thus, studying how these optimization flags affect floating-point exceptions is important. To do so, we run the same programs with the optimization flags: `-ffast-math` and `-fdenormal-fp-math=preserve-sign`. The inclusion of the second flag is to investigate how exception behaviors related to subnormal numbers change; simply enabling `-ffast-math` does not enable this flag.

In total, when aggressive floating-point optimization flags are enabled, the average number of floating-point exceptions for all tested programs drops by 37.3%. First we look at the 9 benchmark programs in which FloatGuard detects floating-point exceptions, when aggressive floating-point optimization flags are enabled, the number of exceptions detected changes, as shown in Table 4.5. The results are mixed: 2 programs have the number of exceptions decreased, 4 programs increased, and the remaining 3 have their numbers unchanged. If we only count the number of exceptions related to special values, then 2

programs have the number of exceptions decreased, 1 program increased, and the remaining 6 have their numbers unchanged. On one hand, there are programs like `myocyte` where all types of floating-point exceptions have their numbers decreased. On the other hand, programs like `wp` have their floating-point exceptions numbers change up or down, indicating that aggressive optimization flags in the HIPCC compiler change rather than reduce the characteristics of exceptions in a program.

Next we look at the 500 synthetic GPU programs. We observe that for the 472 programs[4] that successfully execute when optimized, the average number of floating-point exceptions fall from 6.023 to 3.19 when aggressive floating-point optimization flags are enabled. FloatGuard reports 297 (62.9%) programs with fewer floating-point exceptions, 45 (9.5%) programs with more exceptions, and 130 (27.5%) with an unchanged number of exceptions. An example of fewer reported exceptions when aggressive floating-point optimization flags are enabled is `case_68`, where the single floating-point exception detected is an input denormal exception, which is not triggered when aggressive floating-point optimization flags are enabled. On the contrary, an example of more floating-point exceptions detected when aggressive floating-point optimization flags are enabled is `case_387`. When aggressive floating-point optimization flags are disabled, the first floating-point exception likely comes from the first line of code in `compute`, where we verify that this `if` statement always returns `false` regardless of compiler options:

```
if(comp==cosf(var_1-1.8906E35f-(1.1216E14f/var_2))) {
```

Variable `var_2` is assigned a subnormal number from program input, triggering an overflow exception with the result of the division. When the aggressive optimization flags are enabled, `var_2` is flushed to zero, therefore triggering a divide by zero exception instead. There is also a new exception detected when aggressive floating-point optimization flags are enabled; it is of the input denormal type, and it is likely that different ways to compile the equal operator in this line of code is the cause of this new exception.

[4]The remaining 28 programs cause runtime exceptions unrelated to floating point when compiled and run with aggressive floating-point optimization flags, and are thus excluded from discussion in this subsection.

Answer to RQ2: Overall, the average number of floating-point exceptions drops by 37.3% when aggressive floating-point optimization flags are enabled. For the 500 synthetic GPU programs, the average exceptions drop from 6.023 to 3.19 when optimizations are enabled, with 62.9% of programs showing fewer exceptions, 9.5% showing more exceptions, and 27.5% unchanged. These results indicate that aggressive optimization flags change rather than uniformly reduce exception characteristics.

4.4.4 RQ3: Comparing AMD HIP and NVIDIA CUDA Exception Behavior

Floating-point operations are inherently sensitive to variations in how they are compiled and executed. This has been shown in previous studies described in Section 4.5.2. Many heterogeneous GPU programs, including most programs used in our evaluation, are written for specific hardware and software environments, most commonly to run on NVIDIA GPUs and CUDA, and then ported to the AMD HIP platform. Studying the differences in floating-point exceptions between NVIDIA CUDA and AMD HIP versions of the same program can reveal how each platform handles IEEE 754 compliance, compiler optimizations, and exception propagation. It helps identify numerical divergences that are less obvious, guides platform-specific debugging, and informs trade-offs between program performance and correctness. Such understanding subsequently ensures portability and reliability of scientific applications across GPU vendors. Thus, in this section, we investigate differences with respect to floating-point exceptions when running a same program on both AMD and NVIDIA platforms.

To study the differences of program behavior between AMD and NVIDIA GPUs, we use the same set of 565 programs described in Section 4.4.1, running on both HIP and CUDA platforms. We use the CUDA version of HPCG [73], while using the same codebase for other programs, and only differing in HIP/CUDA function calls. To detect floating-point exceptions in CUDA programs, we use the state-of-the-art tool GPU-FPX [59]. We use the latest GitHub revision as of writing[5] and build it using CUDA 11.8 and NVBit 1.7.2. We run the detector module of GPU-FPX to detect floating-point exceptions on CUDA. Our

[5] https://github.com/LLNL/GPU-FPX/tree/ab39d7f286fed890c394af014175721b619b8d28

experiments are run on an 18-core Intel(R) i9-10980XE processor workstation with 256 GiB RAM, an NVIDIA RTX 4090 GPU, and with Ubuntu 22.04 OS installed.

Since both tools use different hardware, software, algorithms, and possibly different metrics to track floating-point exceptions, a direct comparison of slowdown ratio or the amount of exceptions triggered is not practical. Instead, we simply run both tools separately to detect floating-point exceptions on both environments, on the same set of programs, with the same optimization level, no aggressive floating-point optimization flags like `use_fast_math` or `-ffast-math`. We compare them for any differences in both detected exceptions and the end results, to gain an insight on diverging behavior for the same programs across platforms.

First we run both tools with the same 500 synthetic GPU programs. These programs are compiled with `-O3` on both platforms. We find that GPU-FPX only finds 404 out of 500 in the CUDA versions, compared to AMD HIP versions where 498 programs out of 500 are detected with floating-point exceptions. We look into the GPU-FPX source code and find that GPU-FPX does not detect input denormal exceptions. After removing input denormal exceptions from the FloatGuard results, the number of HIP programs with detected floating-point exceptions drops from 498 to 365. We look into a few example programs where only the CUDA or HIP version triggers floating-point exceptions. We analyze their source code, compiled assembly, program control flow during execution, and try out different compiler optimization flags to determine possible cause for these differences.

The first program we analyze is `case_450`. GPU-FPX does not detect any floating-point exceptions in this program, while FloatGuard detects an underflow exception from the first line of the `compute` function:

```
if (comp < (-1.2964E-35f / var_2)) {
```

where `var_2` is assigned a value of -1.9453E27f. The result of the division operation is too small to be represented by a 32-bit floating-point type, regardless of whether subnormal numbers are allowed as operation results, and is rounded to zero. Since GPU-FPX simply checks the results of operations, which is zero in this case, it does not treat this as an exception. Still, reporting such underflow exceptions is helpful to developers, since with a slight change to `var_2`, this division operation can still generate subnormal results.

The second program we analyze is `case_350`. GPU-FPX detects an underflow exception while FloatGuard does not detect any exception. To investigate the cause, we find that the `if` statement at the beginning of the `compute` function is as follows:

```
if (comp<=(-0.0f-var_1-1.9945E-44f/-1.8945E36f)){
  float tmp_1=(1.6933E34f-(-1.3140E20f/+1.2012E-26f));
  float tmp_2=-1.5160E-43f;
  comp=tmp_2/tmp_1*var_2*var_3;
}
```

We investigate the PTX code generated by NVCC and find that under `-O3` optimization, all constant expressions are calculated at compile time, and the `if` statement is compiled as a selection instruction. Since the `comp` variable has a subnormal value before this code snippet, and the `if` condition is false, the subnormal value in turn becomes the result of the selection instruction thus triggering GPU-FPX to report an underflow exception. Meanwhile, in the AMD HIP version, even though it is still compiled as a selection instruction, the hardware registers do not trigger exceptions here. It only triggers an input denormal exception when input denormal exceptions are enabled. Our conclusion is that FloatGuard, utilizing hardware registers, can more correctly identify the types of exceptions than GPU-FPX.

As for the 65 benchmark programs, GPU-FPX detects floating-point exceptions in 11 of them: `rayTracing` and `wp` in GPGPU-Sim benchmarks, `s3d` in SHOC, `stencil` in Parboil, `correlation, gesummv, gramschmidt, lu` in PolyBench-ACC, `cfd` and `myocyte` in Rodinia, and the HPCG benchmark. As comparison, FloatGuard detects floating-point exceptions in 9 programs. Next we focus on the following programs where only one tool detects exceptions in its respective platform.

rayTracing in GPGPU-Sim: FloatGuard does not report any floating-point exceptions; meanwhile GPU-FPX reports 10 floating-point exceptions. After investigation, we find that all of them result from CUDA implementations of `__saturatef` and `pow` functions, frequently used in color manipulations to avoid using branching statements in GPUs. It is likely that the AMD HIP version of these function implementations avoids triggering floating-point exceptions. We also compare the rendered images for both versions of this program and find that they are indeed slightly different.

gesummv in PolyBench-ACC: when the dataset is extra large, there are two floating-point exceptions captured with GPU-FPX in the `gesummv_kernel` function. One is an underflow exception from line 115, which is a multiply-add operation; the other is an overflow exception from line 118, which are two multiply-add operations. We manually add debug code after the two expressions, to check if the results are subnormal or infinite, and we find that the CUDA version shows subnormal or infinite results while the AMD HIP version does not. This is possibly due to aggressive optimizations from CUDA.

adi in PolyBench-ACC: FloatGuard reports 4 floating-point exceptions in 4 different kernels (corresponding to expressions in lines 164, 177, 201, 212 of the original CUDA version). All four kernels involve dividing by an element in matrix B. If we put `isnan` calls for the results for these 4 expressions we can find that NaNs do occur in these kernels when running on AMD GPUs. However, they do not occur in the CUDA version.

HPCG: FloatGuard reports no floating-point exceptions in the HIP version; meanwhile, GPU-FPX reports two exceptions, both in the `reduce_1Block_kernel` which is a closed-source kernel in the cuBLAS library. It is likely that the AMD implementation of the library, hipBLAS, avoids these exceptions.

Answer to RQ3: Floating-point exceptions vary between AMD HIP and NVIDIA CUDA platforms. FloatGuard detects exceptions in 498/500 synthetic HIP programs, compared to 404/500 synthetic CUDA programs for GPU-FPX. For benchmarks, FloatGuard detects 9/65 programs while GPU-FPX detects 11/65. The differences in how exceptions arise are due to distinct implementations of library functions, compiler optimizations, and hardware-specific exception handling.

4.4.5 Limitations

The evaluation results of FloatGuard, in terms of the floating-point exceptions captured by the tool, are contingent upon hardware and software environments and the inputs of the program. Different AMD GPUs, different compiler versions, or a different set of inputs, may trigger a different set of exceptions in a same program. The code coverage of a program when different inputs are tested can also vary, and may trigger floating-point exceptions

on different code paths. FloatGuard users need to be cautious when they interpret exception information from the output of FloatGuard; proposed fixes to these floating-point exceptions may impact the correctness of the program when it is run under different environments and inputs. Lastly, subsequent exceptions captured by FloatGuard may be causally dependent on earlier ones; modifying code to prevent an earlier exception may also suppress others. Therefore, thoroughly evaluating all potential floating-point exceptions during execution may require analyzing all instructions that can trigger exceptions, along with an input generation strategy to explore those conditions.

In terms of tool compatibility, since FloatGuard modifies assembly code, any inserted code is dependent on the GPU architecture of interest. Users may need to modify the instructions used in case a newer architecture changes register names, bit fields, and/or the instructions used to set registers, according to the instruction set architecture manuals for each GPU architecture.

In terms of reported floating-point exception information, FloatGuard outputs source code locations at which floating-point exceptions are triggered; however, these source code locations are based on the ROCgdb's output, which in turn is based on compiler debugging information. Compiler optimization passes, both from the Clang front-end and the AMD GPU back-end, may cause inaccuracies. Furthermore, the accuracy of the reported floating-point exceptions is dependent on the existing GPU exception triggering mechanism; the binary code locations of floating-point exceptions given by ROCgdb may be delayed, as described in Section 4.2.2, also resulting in potentially inaccurate information. Lastly, due to ROCgdb's limitations, FloatGuard only provides source code line information. Developers must investigate the source code along with assembly locations around the reported locations to pinpoint the cause of these floating-point exceptions.

Similar to the exceptions triggered by FloatGuard, slowdown ratio of FloatGuard is contingent upon the environment it runs on. Different AMD GPU models, ROCm versions, and input parameters may affect P in Equation (4.1); meanwhile CPU models and RAM speed may affect OH in the same equation. Both variables affect the slowdown ratio. Furthermore, as described in Section 4.4.2, the slowdown of using FloatGuard increases as the number of floating-point exceptions increases; and some programs have a massive

amount of floating-point exceptions, likely due to the fact that exceptional values, both subnormal values and special values like Inf or NaN, can be passed downstream and continue to trigger floating-point exceptions. In cases like these, FloatGuard introduces greater slowdown with many reported exceptions being secondary. It is recommended that users of FloatGuard either (1) focus on resolving the first exception found in a kernel, then re-run FloatGuard to uncover any remaining exceptions, or (2) filter out floating-point exceptions of less impact, such as input denormal and underflow. Lastly, FloatGuard's overhead is also affected for programs with longer link times.

Overall, the slowdown ratio may be reduced by minimizing the overhead of repeatedly re-linking and re-executing the program upon each floating-point exception by methods such as preserving the program state on the host side across successive kernel invocations. As AMD's ROCm platform continues to evolve, current limitations in floating-point exception handling may be alleviated. For instance, future iterations of ROCm may allow recovery of the program state from the GPU thread where the exception occurred. In such cases, FloatGuard could be modified to eliminate or reduce the need for repeated execution, thereby improving performance. Additionally, the development of tools similar to NVBit [74] but for AMD HIP code, such as Luthier [75], could enable new approaches to floating-point exception detection on AMD GPUs.

4.5 Related Work

This section discusses related work in floating-point exception analysis and floating-point error analysis in general.

4.5.1 Floating-Point Exception Analysis

Several tools have been developed to detect floating-point exceptions across a variety of platforms with different black box and white box testing techniques.

CPU Exception Detection. Similar to FloatGuard, FPSpy [76] leverages existing condition codes inside status registers in x64 processors to detect floating-point exceptions. It performs binary instrumentation of existing Linux x64 applications without modifying

the binary or requiring extra work by developers, while FloatGuard injects code into the intermediate assembly files. However, it is important to note that it is possible to rewrite FloatGuard to directly inject code into binaries given appropriate tools.

Symbolic Execution Approaches. Ariadne [77] transforms a numerical program to check for exception triggering conditions and uses symbolic execution on the transformed program to find inputs that can trigger floating-point exceptions. More recently, EXCVATE [78] spoofs floating-point exceptions in numerical libraries to detect instances in which the resulting special values are not propagated according to exception-handling policy. For those, EXCVATE then uses an off-the-shelf SMT solver to generate concrete inputs that induce the buggy behavior. However, none of the works above detects floating-point exceptions for GPU programs.

NVIDIA GPU Exception Detection. Floating-point exceptions detection tools on GPUs adapt to its heterogeneous, parallel execution characteristics. Various methodologies are developed on CUDA programs running on NVIDIA GPUs due to the fact that there is no exception status hardware presented in those GPUs. FPChecker [57] uses LLVM passes to inject checking functions for each floating-point operation into the LLVM intermediate representation (IR) of the program. These checking functions are then executed alongside the original program during runtime analysis, and detects and records exceptional events. FloatGuard ultimately does not utilize the approach of injecting code into LLVM IR because of extensive backend optimizations done to the program after the post-optimization LLVM IR code is generated, and instead opt for assembly level injection. BinFPE [58] and GPU-FPX [59] both use the NVBit [74] framework provided by NVIDIA to perform binary instrumentation on floating-point code. Both of them detect floating-point exceptional events during GPU kernel execution, and transmit data from GPU to the host program. GPU-FPX provides superior performance and better analysis of floating-point exceptions with its detector and analyzer modules.

4.5.2 Floating-Point Error Analysis In General

There are other floating-point error detection and analysis tools that work with specific errors in floating-point programs. These conditions found in these tools may correspond to

floating-point exceptions that happen in the tested programs, but the tools themselves do not directly identify these exceptions.

Precision Error Detection. Select tools that track precision errors are as follows: S3FP [41] finds high precision errors in floating point programs with limited number of variables and predefined value range restrictions; it uses binary guided random testing on a black-box program executable to trigger the conditions that may cause high precision errors. FPGen [48] is similar but uses symbolic execution while injecting error check functions in the LLVM IR to trigger high precision errors, thus it is superior to S3FP in terms of high precision errors.

Shadow Execution Techniques. FPSanitizer [44] and PFPSanitizer [23] use shadow execution to run program snippets in an enhanced precision using posits [79], a replacement format of IEEE floating-points, to achieve the same goal. Herbgrind [80], on the other hand, analyzes program binaries and records the intermediate error of floating-point operations and tracks how these operations influence program outputs and control flow. It also uses symbolic execution to facilitate numerical analysis of floating-point errors. However, it does not explicitly track floating-point exceptions as defined by IEEE 754.

Compiler-Induced Numerical Inconsistencies. On the other hand, there are also tools that track compiler-induced numerical inconsistencies, which is characterized by the difference in outputs for the same numerical program when it is compiled with different compilers and/or compiler optimization flags. FLiT [5] uses bisection search to identify the function in a program that is impacted by such inconsistencies; pLiner [2] and Ciel [31] explore the algorithm further and can identify the line of code, or even an expression in the case of Ciel, in a program that causes these inconsistencies. These inconsistencies found are often related to how different floating-point exceptions are handled under different compilers and/or optimization flags, but they themselves do not explicitly identify floating-point exceptions; also, even when the source code locations found in these tools may correspond to a floating-point exception, it does not track *all* floating-point exceptions encountered during program execution.

4.6 Conclusion

Modern HPC systems increasingly rely on AMD GPUs for scientific computing, yet existing tools for detecting floating-point exceptions have been limited to NVIDIA architectures. This work addresses this gap by presenting FloatGuard, the first framework for efficiently detecting floating-point exceptions in HIP programs running on AMD GPUs. Our approach leverages AMD GPU hardware registers and combines assembly- and source-level instrumentation with debugger-guided execution to overcome the limitations of AMD's built-in trapping mechanisms.

Our comprehensive evaluation on 565 HIP programs demonstrates that FloatGuard can detect floating-point exceptions in 507 cases with a slowdown ratio that increases at most linearly with the number of exceptions discovered. We also analyze the impact of compiler optimizations on exceptions trapped and provide a detailed comparison with state-of-the-art tools for NVIDIA GPUs, revealing important differences in floating-point exception behavior between AMD and NVIDIA architectures. These insights are crucial for ensuring the robustness and reliability of scientific applications across different GPU vendors.

Chapter Acknowledgements

This chapter is based on the paper "FloatGuard: Efficient Whole-Program Detection of Floating-Point Exceptions in AMD GPUs" by Dolores Miao, Ignacio Laguna, and Cindy Rubio-González, presented at *the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2025. The source code of FloatGuard is publicly available at https://github.com/LLNL/FloatGuard.

Funding: This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2005079), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under awards CCF-2119348 and CNS-2346396.

Chapter 5

MUPPET: OpenMP Performance Optimization via Mutation Testing

5.1 Problem and Motivation

While the previous chapters focused on numerical correctness, this chapter addresses a complementary challenge: performance optimization at the source level. As discussed in Section 1.1.2, compiler optimizations at the IR level are not portable across compilers and make reasoning about correctness more difficult.

Limitations of Existing Approaches. A lot of work has been proposed to identify performance issues, and several tools are used in current HPC production environments to analyze the performance of applications [81–84]. However, the process of isolating performance problems and/or generating tests to identify them is still mostly a manual process. Most performance optimization techniques focus on highlighting performance hotspots in the program, but ultimately they rely on programmers to identify code modifications that fix a performance problem or improve overall performance.

Other approaches are based on the concept of quantifying hardware or runtime system events [85–87], but metrics of these events do not directly relate to program performance, and these approaches do not explicitly inform the programmer how to modify the code to improve performance.

Performance models could potentially solve performance problems given accurate models for each available platform and application. However, performance models are notoriously difficult to build accurately given the complexity of the HPC software stack and underlying hardware. While there are solutions to build performance models for specific aspects of the hardware and applications [88–90], they are usually not composable and thus of little practical use in modeling an entire application and platform.

Our Approach. We present an approach based on *mutation testing* [91] to identify source code changes, or *mutations*, that (1) improve performance, and (2) help developers reason about performance at the source-code level, in contrast to IR- or assembly-level like in existing methods. Since such an approach is based on source modifications, it is portable across compilers.

Mutation testing has been proposed to identify correctness faults [91], and assumes that a syntactic change (a mutant) along with an exploration campaign of multiple mutants can help discover programs' defects faster than traditional methods. While some previous work has applied mutation testing to solve performance defects [92], mutation testing for performance has not been applied on parallel code and/or HPC programs. We demonstrate our approach in the OpenMP programming model, although our approach is also applicable to other HPC programming paradigms such as CUDA and OpenACC.

We implement our approach in the framework named MUPPET (Mutation-Utilized Parallel Performance Enhancement Tester). MUPPET generates a list of OpenMP mutations—changes in existing OpenMP directives or additions of new OpenMP directives that could change performance without likely changing correctness. MUPPET then uses optimization algorithms such as delta debugging [93], Bayesian optimization [36], and decision tree optimization [94] to find a subset of mutations that yields the highest speedup.

Contributions. The contributions of this chapter are:

- A source-level approach that uses mutation testing to optimize HPC code. Our approach considers five classes of source mutations and applies them in OpenMP directives (Section 5.3).

- A design and implementation of our idea in the MUPPET framework via the Clang/LLVM front-end. Our approach integrates MUPPET with several optimization algorithms, including delta debugging, Bayesian optimization, and decision tree optimization. The output of MUPPET is a set of source modifications that produce a maximum speedup among the explored mutations, without affecting correctness (Section 5.4).

- An evaluation on several benchmarks and proxy applications demonstrating that MUPPET is capable of identifying mutations that improve performance in 75.9% of the evaluated programs, with the best speedup of 3.57x (Section 5.5).

5.2 Overview

In this section, we describe the philosophy of our approach, provide background information on mutation testing, and provide a simple mutation example in a matrix multiply kernel that improves performance.

5.2.1 Approach's Philosophy

Existing approaches to isolate performance issues are difficult to use in practice. A number of performance problems can be fixed by changes in the source code; however, existing methods do not directly point to developers' source modifications that fix such issues. Compilers optimize code at the IR level but such solutions are not portable across compilers and make it harder to reason about correctness than solutions based on source modifications.

We believe that tools and techniques for performance optimization should have the following features:

- **Fine granularity detection:** tools should pinpoint, with fine granularity, the location (code line) of performance issues or potential performance improvements.

- **Guided fixes:** the approach should help programmers understand and reason about performance defects—without a good understanding, it is hard to solve the problem or avoid it in the future.

- **Automatic recommendations:** the approach should automatically suggest code modifications that improve performance or fix a performance problem.

We designed MUPPET using the above criteria to identify changes in OpenMP directives that improve performance.

5.2.2 Mutation Testing for Performance

Challenges

The key idea of MUPPET is to perform small changes in the code, called *mutations*, and use exploratory algorithms to search for cases where mutations improve performance or fix a performance problem. Mutation testing has been studied before to detect faulty programs by injecting small syntactical changes that expose correctness defects [91]. The idea of mutation testing is to generate sufficient data to expose real software defects in the code. However, it is challenging to use traditional mutation testing in isolating performance defects because the syntactic changes could create faults, i.e., breaking the semantics of the program and producing incorrect programs.

Our Solution

Inspired by the previous work on mutation testing, we propose a different approach: *to inject only mutations that are semantically correct and do not yield an incorrect program for the purpose of exposing performance defects or speedup opportunities.* Semantically correct mutants, or *equivalent mutants*, are considered problematic for traditional mutation testing because by definition, they cannot fail the test suite, so they should be avoided to increase the effectiveness of mutation testing. In contrast, our approach explores semantically correct mutations, or a weaker form of mutations that successfully pass correctness tests, to identify any mutations that increase performance, thus indicating performance defects.

5.2.3 Mutation Example

Here, we present a synthetic matrix-multiplication example, shown in Listing 5.1, that demonstrates MUPPET's capabilities—when we apply MUPPET, it can find a set of mutations that yields faster code execution.

Listing 5.1: Example code with a mutation found by MUPPET that improves performance.

```
1  #define ARRAY_SIZE (2048)
2  double A[ARRAY_SIZE][ARRAY_SIZE];
3  double B[ARRAY_SIZE][ARRAY_SIZE];
4  double C[ARRAY_SIZE][ARRAY_SIZE];
5
6  int main(void) {
7    // initialize array and timer setup omitted
8    float var = 2.3f;
9    #pragma omp parallel for shared(var)
10 // mutation adds an OpenMP directive
11 #pragma omp tile sizes(16,16,16)
12   for (int i = 0; i < ARRAY_SIZE; ++i)
13     for (int j = 0; j < ARRAY_SIZE; ++j)
14       for(int k = 0; k < ARRAY_SIZE; ++k) {
15         C[i][j] += var*A[i][k]*B[k][j];
16       }
17   // end processing omitted
18 }
```

Originally, the code has only the OpenMP `parallel for` directive to parallelize the loop. MUPPET applies mutations to the existing OpenMP directives found in the code. Note that while MUPPET only considers semantically correct mutations (and are likely to produce a correct program), it relies on existing correctness checks of the program, as shown in Section 5.5.1 for the evaluated programs. Possible mutations are shown in Figure 5.2 as references. When we run MUPPET on this example with delta debugging, after 20 tryouts, MUPPET reports a mutation that, when applied to the program, improves performance. With BO, it takes 66 tryouts to finish the optimization process; but the mutation was reported with 11 tryouts. The identified mutation is highlighted in the source code. In this simple

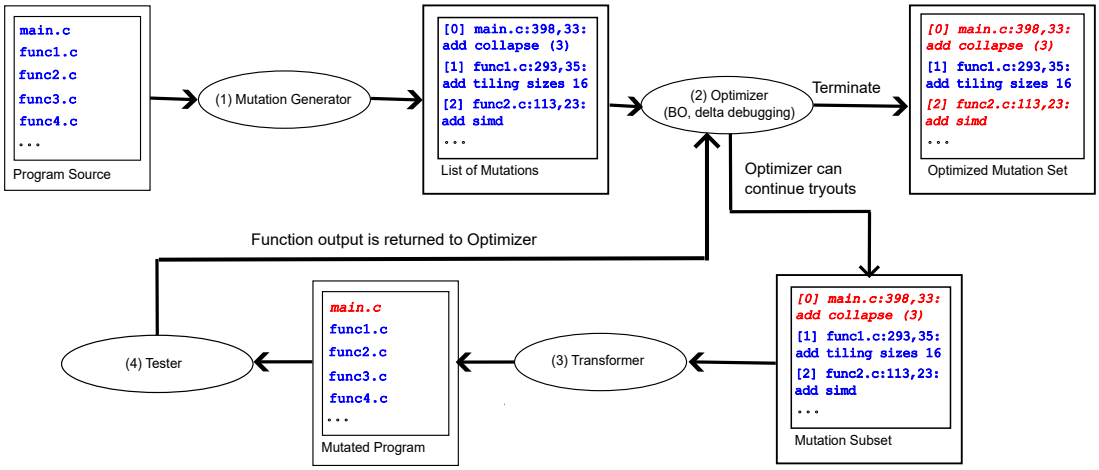


Figure 5.1: The workflow of MUPPET. *Red texts in italic* indicates the mutation is applied, or the source file is changed.

example, the mutation is the addition of the OpenMP tile construct, which converts the three-dimensional loop space in this program into smaller-sized "tiles" in 16x16x16 increments and suggests to the compiler that each tile is to be assigned to one OpenMP thread. In the end, MUPPET reports to the developer that adding this construct to the loop introduces an 18.84x speedup, from 7.116801 seconds to 0.377674 seconds.

5.3 Approach

5.3.1 Problem Statement

Given an OpenMP program P with running time T, MUPPET analyzes the program and generates a set of mutations, $M = \{m_1, m_2, ..., m_n\}$, which potentially could induce program speedup. We define the program running time for the original variant program as:

$$T = P(\emptyset)$$

We define the running time for a variant program as:

$$T' = P(M'), \text{where} \quad M' \subseteq M, \text{and} \quad accurate(P, M') = True.$$

We define the ideal minimum program running time as:

$$
\begin{aligned}
& T_{min} = P(M_{min}), \\
& \text{where} \quad M_{min} \subseteq M, \\
& \text{and} \quad accurate(P, M_{min}) = True, \\
& \text{and} \quad \forall M' \subseteq M, T' >= T_{min}
\end{aligned}
$$

The goal of MUPPET is find a subset of M, $M_{min'}$, with $T_{min'}$ as close to T_{min} as possible.

5.3.2 Tool Workflow

The overall workflow of MUPPET is illustrated in Figure 5.1. The purposes of these modules are described below:

- *Mutation generator* analyses the program and finds a set of source code mutations, which can potentially be applied to change the OpenMP parallelism of the program.
- *Transformer* generates a program variant with a subset of mutations found in the *Mutation generator* module.
- *Tester* runs the mutated programs from *Transformer* and tests the performance speedup and correctness of the mutated variant.
- *Optimizer* applies a user-specified optimization algorithm to find the minimum of the function $T' = P(M')$.

In the next subsections, we describe the details of these modules, following the order as they appear in Figure 5.1.

5.3.3 Mutation Generator

The Mutation Generator module traverses the abstract syntax tree (AST) of the program, looking for source code locations that potentially can be mutated so that program parallelism is changed. The time complexity of this step is $\mathcal{O}(n)$ where n is the number of statement

nodes on the AST. The mutators in MUPPET focus on mutating parallel/loop OpenMP constructs such as the `parallel` directive, `for` directive, or the `parallel for` directive. All of these directives specify a source code region to be executed in parallel, but the parallelism may not be high enough to utilize all available cores for the OpenMP program. MUPPET also looks for the beginning of `for` loops for both SIMD mutations. Lastly, `tiling` mutations may be added in code locations where tiling is possible to be performed in an inner part of a multiple-dimension parallel loop.

Mutation Classes

There are five classes of mutations possible to apply to certain source code locations:

- **Collapse Mutations** add a `collapse` clause to a multiple-dimension `parallel for` loop. Collapse clauses may potentially improve parallelism by having more iterations, thus higher hardware thread usage, at the top level of the loop.

- **SIMD Mutations** have two forms: adding a `simd` clause to an OpenMP parallelism-related directive such as a `parallel for` loop or an `omp for` loop; or adding a `simd` directive to a `for` loop. SIMD clauses or directives hint at the compiler to check if there is a possibility to vectorize the loops and apply SIMD vectorization if possible.

- **Tiling Mutations** add `tile` directives at the top of a multiple-dimension OpenMP loop. Tile directives split the loop space into smaller-sized "tiles", and each tile is ideally only accessed by one OpenMP thread. This design can potentially improve cache locality depending on how the data within memory is accessed within the loops, and thus may also introduce performance speedup. Due to the difficulty in determining loop size at compile time, MUPPET only supports setting a fixed set of differently sized tiles as different mutations. For example, we can only set the tile size as a power of 8, 16, or 32. Given the limitations, users can still see from the optimization results whether using a smaller or larger-sized tile can have a higher speedup.

- **Firstprivate Mutations** put read-only shared variables into a `firstprivate` clause for an OpenMP parallel region, so that these variables are kept a copy in each parallel

thread. These mutations are designed to reduce data dependency between parallel threads when these variables are accessed.

- **Schedule Mutations** add a `schedule` clause to an OpenMP `parallel` or `parallel for` directive. OpenMP by default statically assigns loop iterations to OpenMP threads when running a parallel block. This mutation modifies the scheduling behavior of OpenMP programs. Similar to tiling mutations, MUPPET supports two different schedule mutations: dynamic- and auto-scheduling, implemented as two different mutations for each parallel block.

For all mutations, once a language construct of interest is detected, the Mutation Generator module then checks the associated source code around the current language construct. If the source code around it satisfies certain statically defined criteria (see Section 5.3.3), then unique information regarding the current mutation, such as source location, the way source code is modified with this mutation (insert before a source code location, insert after a source code location, modify a source code range), and the class of the mutation is added to the list of mutations. The algorithm for this process is shown in Algorithm 4.

Criteria Selection

The criteria for each class of mutation follow the syntax of OpenMP language specifications. These criteria can be expanded for any new class of mutations added. A list of criteria is below:

- Mutations should not be added in a `for` loop that has statements that change the control flow, such as a `break`, `continue` and a `return` statement. OpenMP parallel loops in existing code already follow this rule, but other standalone `for` loops may not, so condition checking is necessary.

- Tiling and SIMD mutations should not be added when the affected OpenMP block contains specific OpenMP directives such as a `critical`, `barrier`, or a `master` directive.

- There should not be OpenMP blocks inside SIMD directives, otherwise such mutations should not be added.

Algorithm 4: The mutation generator algorithm.

```
1  Function GenerateMutations(AST):
2      M = ∅;
       // Traverse the AST.
3      foreach Statement in AST do
4          if Statement is an OpenMP directive then
5              if can add collapse mutation then
6                  M = M ∪ CollapseMutation(Statement);
7              if can add SIMD mutation then
8                  M = M ∪ SIMDMutation(Statement);
9              if can add tiling mutation then
10                 M = M ∪ TilingMutation(Statement);
11             if can add firstprivate mutation then
12                 M = M ∪ FirstprivateMutation(Statement);
13             if can add schedule mutation then
14                 M = M ∪ ScheduleMutation(Statement);
15         if Statement is a for loop then
16             if can add SIMD mutation then
17                 M = M ∪ SIMDMutation(Statement);
18             if can add tiling mutation then
19                 M = M ∪ TilingMutation(Statement);
20     return M
```

- In general, there should only be SIMD directives inside a SIMD directive, and no other directives are allowed.

We illustrate applicable OpenMP mutations in the previously shown *matmul* example in Figure 5.2. The one that shows the highest speedup in *matmul* is the tiling mutation.

5.3.4 Optimizer

Once a list of mutations is generated, it is exported to the Optimizer. This module runs an optimization algorithm specified by the end user to find the minimum point of $T' = P(M')$. During the optimization process, it finds specific points on the $T' = P(M')$ function by selecting a subset of mutations, sending these mutations to the Transformer and Tester module, and receiving T' from the Transformer and Tester module once the mutated program has finished execution and running time statistics are collected.

MUPPET supports three optimization algorithms: delta debugging [93], Bayesian Optimization [36], and decision tree optimization [94]. The goal of all three algorithms, albeit vastly different in implementation, is to find the subset of source mutations that would introduce maximum speedup. The inclusion in MUPPET of a variety of algorithms shows how

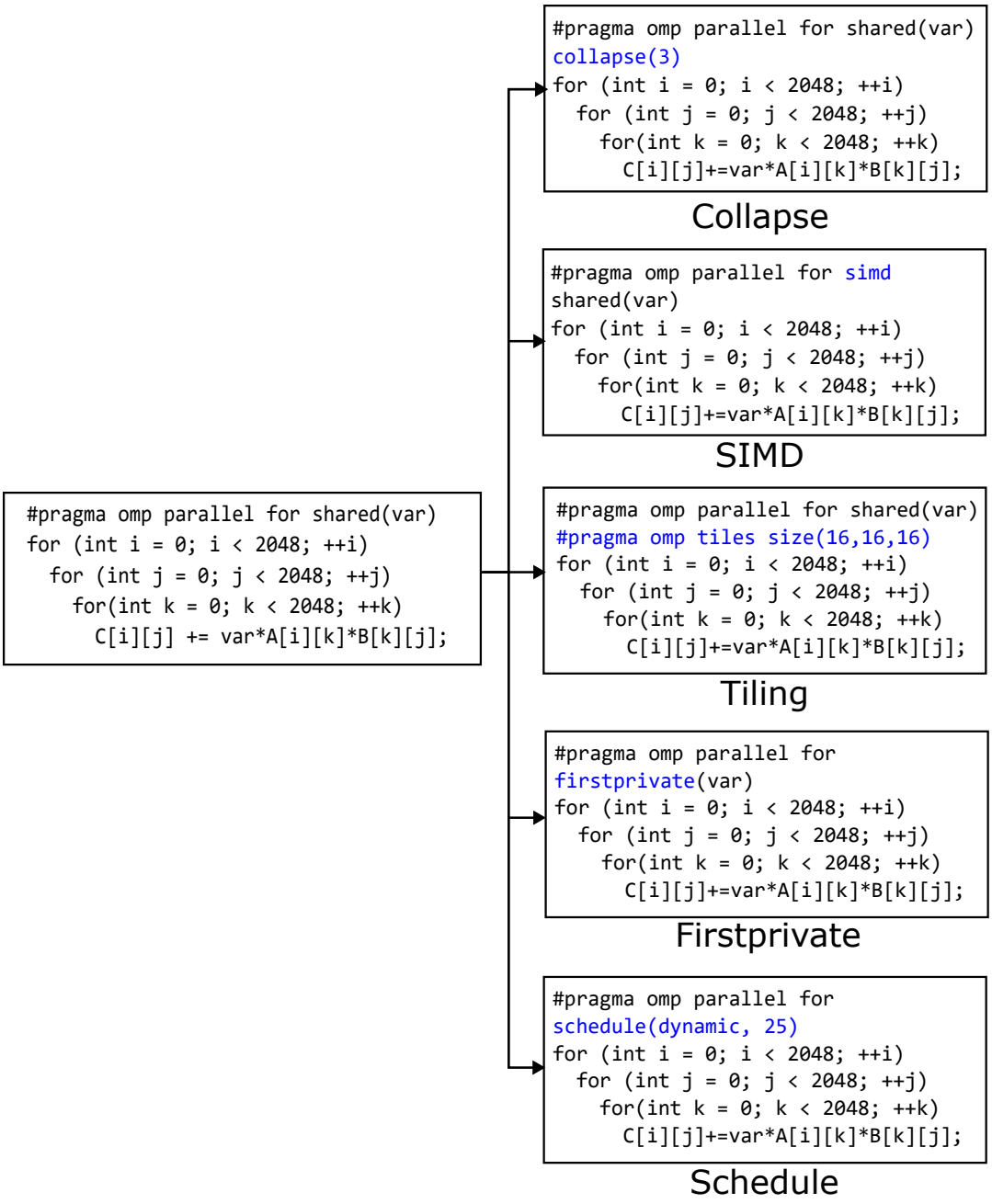


Figure 5.2: Classes of mutations in MUPPET.

algorithms with vastly different original purposes can solve the same problem in different ways with varying efficacy. MUPPET can also be extended to run other optimization algorithms such as differential evolution or simulated annealing. The details for each algorithm and how they are adapted to MUPPET are detailed below.

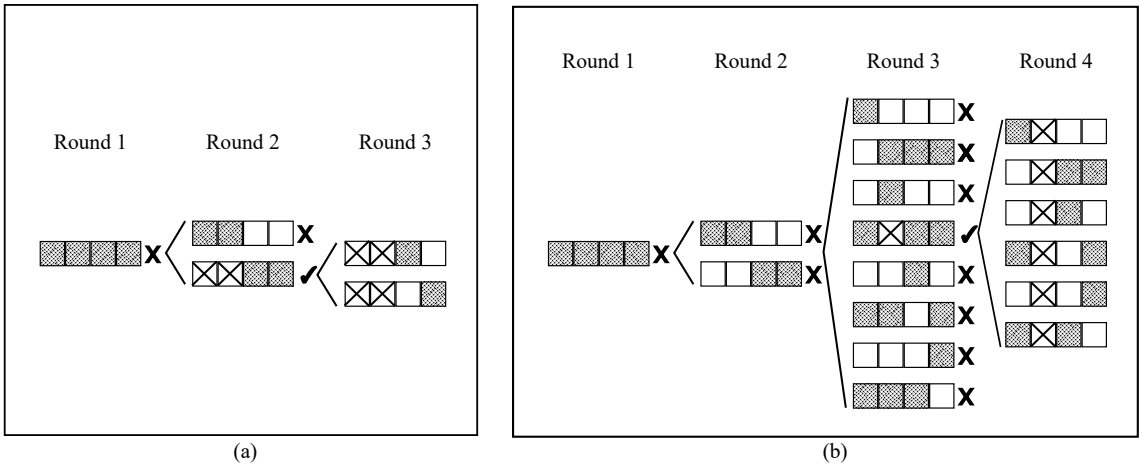


Figure 5.3: Two simple examples illustrating the delta debugging algorithm.

Delta Debugging

Delta debugging (DD) is an algorithm that was originally developed as a software testing algorithm to isolate bugs inside a program, which is then adapted into finding speedup in program variants in previous work such as Precimonious [95] with regards to precision tuning. We follow the LCCSearch algorithm in [95], where a change set in our adaptation of the algorithm is defined as the set of mutations that are applied to the original program, and the outputs are a minimal change set which causes speedup.

To illustrate how delta debugging is adapted to Muppet, we show two simple examples in Figure 5.3. The change set is initially assigned as the set of all mutations. Then Muppet starts testing mutated program variants with all mutations, with the first half of mutations, and with the second half of mutations. In (a), the second half of mutations cause speedup, thus the second half of mutations become the new change set, and then this new change set is split in two halves and test their performance speedup respectively. In (b), none of the halves cause speedup, thus Muppet split the mutations into 4 parts instead of 2 and test each part and its complement set, respectively, as shown in Round 3. If a single part among these 4 parts causes speedup, then similar to (a), that part becomes the new change set and is split again in two for further testing. If a complement set (3 parts in this case) causes speedup, as shown in the figure, this means that excluding that one part improves performance, thus the change set becomes the remaining 3 parts for further performance testing, as shown

in Round 4. The algorithm finishes running when every part in the algorithm is a single mutation and cannot be split further.

The time complexity of delta debugging is $O(n \cdot log(n))$ where n is the number of possible mutations in the average case; in the worst case, the time complexity is $O(n^2)$.

Bayesian Optimization

Bayesian Optimization (BO) is a common optimization algorithm that approximates a computationally expensive function, such as the programs tested in Section 5.5. In MUPPET, Gaussian processes are used as surrogate models to approximate program time characteristics, an acquisition function is used to predict the next input to be tested for performance, and the result of the next input (running time) is then sent back to improve the surrogate model. BO does not have the assumption of the function forms, which makes it an appropriate algorithm to use in MUPPET.

There is one difficulty in adapting BO to MUPPET: BO requires the function to be in the form of $y = f(x)$ where x is a list of number inputs, while y is a floating-point output (in our case, program run time). Meanwhile the input parameter of the function to be optimized, $T' = P(M')$, is a subset of mutations. Therefore we optimize $T' = P(Mb')$ instead where:

$$Mb' = \{mb_1, mb_2, ..., mb_n\}$$

$$mb_i = \begin{cases} 1, & \text{if} \quad m_i \in M' \\ 0, & \text{otherwise} \end{cases}$$

In this way, we assign 0 or 1 for each element in the set of mutations according to whether the mutation is in the subset. For example, if $M = \{m_1, m_2, m_3, m_4\}$, $M' = \{m_1, m_3\}$, then $Mb' = \{1, 0, 1, 0\}$. The converted list of 0 and 1 can be accepted by BO as input parameters.

The computational complexity of Bayesian optimization is $O(n^3)$ where n is the number of mutations, however, some methods reduce the computation time, as shown in [96].

Decision Tree optimization

Similar to BO, decision tree optimization (FO) also uses a surrogate model to approximate an expensive function, and an acquisition function to estimate the next input for running time evaluation, and the running time output is also sent back to improve the surrogate model. However, in this case, a decision tree regression model is used instead. We decide to use random forest [97] as the surrogate model here. Also similar to BO, FO optimizes $T' = P(Mb')$ function by assigning 0 or 1 for each mutation in the mutation set.

The computational complexity of decision tree optimization using random forest is $O(n \cdot log(n) \cdot d \cdot k)$ where n is the number of points in the training set, d is the number of mutations, and k is the number of decision trees in the forest.

5.3.5 Transformer and Tester

The Transformer and Tester modules read the list of mutations from the Optimizer module, mutate the program into a variant, and run the variant to see if there is any speedup while maintaining the correctness of the program.

Compilation and Conflicts Checks

Even though there are already criteria placed in the Mutation Generator module for each mutation class to ensure that all mutations generated are syntactically correct, there are still situations where two mutations, when applied to the same programs at the same time, cause conflicts between them. If MUPPET lets these conflicts pass without checking during the transformer phase, it would cause a large number of mutated program variants that do not compile. Thus, to save execution time, when the Transformer module traverses the program, it also statically checks and circumvents certain conflicts. These conflict checks can also be customized in the case where new classes of mutations are implemented or new conflicts are discovered during testing. Existing conflict checks in MUPPET are listed below:

- An active SIMD directive mutation should not be inside an active collapse or tiling mutation. Any mutation that causes this conflict will not be applied.

- An active tiling directive mutation should not be inside an active SIMD mutation. Any mutation that causes this conflict will also not be applied.
- Every variable inside an OpenMP parallel region is checked. If the variable is read-only, and not in a list of existing `private` variables, then it will be included in the list of variables in the firstprivate mutation.

5.4 Implementation Details

5.4.1 Tools Used to Implement MUPPET

MUPPET is implemented with a variety of programming languages and toolsets. The Mutation Generator and the Transformer modules are implemented via Clang plugins. Given the nature of our work on source-level OpenMP mutations, MUPPET is compatible with programs compiled with any C/C++ compiler as long as it supports OpenMP 5.0[1]. There are a few components in Clang that are capable of performing source-to-source code transformation besides Clang plugins, such as libtooling and libclang. Sending Clang plugin calls as compiler parameters to the build system ensures that all source files are processed by the Mutation Generator and the Transformer modules. MUPPET only requires minimal changes to the build systems for it to work on new programs, which is described in Section 5.4.3.

The Optimizer and Tester modules, and the overarching framework managing the communication between modules, on the other hand, are implemented in Python 3.10, to leverage the existence of a mature set of Python optimization modules such as scikit-optimize [98].

5.4.2 Modular Extension

Since MUPPET uses a modular approach, each of the four modules shown in Figure 5.1 can be replaced to implement an analogous functionality. Optimization algorithms can be replaced, as mentioned in Section 5.3.4. Even language support can be extended; while currently MUPPET targets C/C++ programs with OpenMP language constructs, it is possible to target

[1]By disabling certain classes of mutations such as tiling, MUPPET is compatible with programs built with compilers that only support lower versions of OpenMP such as 4.5, although the speedup discovered would potentially be lower.

FORTRAN programs by rewriting the Mutation Generator and Transformer modules with a source-to-source FORTRAN compiler such as ROSE [99].

5.4.3 Making MUPPET Work with Your Own Program

For better management of programs, MUPPET calls a customized version of the FAROS build system [100]. MUPPET accepts a YAML config file that contains program entries, with commands for a variety of functionalities such as analyzing, transforming, building and running the specified program. With FAROS, it is easy to add new programs to perform OpenMP mutation testing for speedup by simply adding new entries into the config file.

Entries and Correctness

An example entry for a locally stored simple matrix multiplication program is shown in Listing 5.2. It sets up commands for each step used in MUPPET, such as building, calling plugins for mutations, running the program, extracting running time statistics from program output, and cleaning. Note that the `call_plugin` part of the YAML file was not originally a part of FAROS and is an addition of MUPPET to FAROS to analyze and transform program source code. In addition to the YAML file, the only changes required for the *matmul* source code is (a) modify the build scripts (Makefile targets `func_analysis` and `trans_mutations` in this case) so that it accepts parameters for calling the Clang plugins, and (b) add correctness check code that parses program output and determines if the mutated program variant still runs correctly.

Listing 5.2: YAML config file for *matmul*.

```
1 matmul: fetch: 'cp -r ../../../extra/matmul .'
2   build_dir: 'matmul'
3   build: {
4     omp: ['make CC=clang++ OPT_LEVEL=3 OMP=1'],
5   }
6   call_plugin: {
7     analysis: ['make func_analysis OMP=1'],
8     mutate: ['make trans_mutations OMP=1'],
9   }
```

```
10    copy: ['matmul' ]
11    bin: 'matmul'
12    run: './matmul'
13    input: ''
14    measure: 'Work consumed (\d+\.\d+) seconds'
15    clean: 'rm -r *.*; cp ../../../../extra/matmul/*.* .'
```

5.4.4 Customizing MUPPET Runtime Parameters

User can select between delta debugging, Bayesian optimization and decision tree optimization when using MUPPET. Bayesian optimization and decision tree optimization are implemented with `gp_minimize` and `forest_minimize` functions in scikit-optimize, while delta debugging is implemented from scratch, adapting the algorithm described in Precimonious [95], since it has no publicly available Python implementations.

Since the running time for each program run may have variability that should not be counted as speedup, to reduce the impact of such variability affecting the reported speedup, users can customize MUPPET parameters to change how it measures running time. The *times* parameter instructs MUPPET to run a specified number of repetitions for each variant, and collect running times for each run; the *shuffle* switch, only available for delta debugging, randomly shuffle the order of mutations so that the delta debugging algorithm partitions these mutations differently each time (users can still specify the same random number generator seed for the same shuffle result). Lastly, users can choose between using the minimum running time in all repetitions, or using the average running time, as the fitness function for all optimization algorithms. Our evaluation of MUPPET uses some of these parameters which are discussed later in Section 5.5.1.

5.5 Experimental Evaluation

This experimental evaluation answers the following research questions:

RQ1 Does MUPPET discover source code mutations that induce speedup for OpenMP programs, and how does it perform with different algorithms?

Table 5.1: Problem size and running time for evaluated programs.

(a) Benchmarks (Rodinia, NPB-CPP, HPCGs)

Program	Parameters	Min. Time	Avg. Time
backprop	16777216	13.270s	13.506s
cfd	fvcorr.domn.193K	18.527s	18.535s
b+tree	file mil.txt command command.txt	1.536s	1.539s
heartwall	test.avi 20 14	2.259s	2.266s
hotspot	1024 1024 16384 14 temp_1024 power_1024	3.896s	3.921s
hotspot3D	512 8 1000 power_512x8 temp_512x8	2.726s	2.737s
kmeans	kdd_cup	1.727s	1.752s
lavaMD	-cores 14 -boxes1d 32	4.177s	4.191s
leukocyte	5 14 testfile.avi	1.481s	1.549s
lud	-n 14 -s 3200	5.638s	5.847s
myocyte	1000 500 1 14	5.186s	5.221s
nn	filelist.txt 10000 30 90	1.825s	1.835s
nw	32000 10 14	1.332s	1.346s
particlefilter	-x 512 -y 512 -z 100 -np 10000	2.936s	2.991s
pathfinder	1000000 100	2.992s	3.066s
srad	1024 1024 0 127 0 127 1 0.5 100	3.443s	3.476s
streamcluster	10 20 256 65536 65536 1000 none output.txt 14	10.780s	11.066s
FT	CLASS=A	1.257s	1.262s
LU	CLASS=A	4.933s	4.956s
MG	CLASS=A	3.617s	3.704s
SP	CLASS=A	31.526s	31.675s
BT	CLASS=A	42.574s	42.590s
CG	CLASS=B	22.512s	22.735s
EP	CLASS=B	6.244s	6.248s
HPCG	96 96 96	15.548s	15.620s

(b) Scientific Applications

Program	Parameters	Min. Time	Avg. Time
LULESH	-i 1500 -s 35	11.346s	11.740s
CoMD	-e -i 1 -j 1 -k 1 -x 20 -y 20 -z 20	2.304s	2.418s
CoSP2	–hmatName hmatrix.1024.mtx –N 12288 –M 16384	4.266s	4.312s
CLOUDSC	14 100 4 (500 iterations)	8.160s	8.187s

RQ2 How does MUPPET compare when using different optimization algorithms, with limited time budget/tryouts?

RQ3 What kind of mutations does MUPPET discover that cause significant speedup?

5.5.1 Evaluation Setup

Benchmarks

We use a variety of C/C++ OpenMP programs to evaluate MUPPET. These programs are from different fields of scientific computing, utilize a variety of computational kernels, and have different levels of OpenMP parallel optimizations: some are reference implementations with the purpose of maintaining the correctness of the program, while others are manually optimized code. The list of programs tested includes benchmark programs like Rodinia benchmarks [101], NPB-CPP [102], HPCG [70], and scientific applications such as LULESH [103], CoMD [104], CoSP2 [105], and the CLOUDSC cloud physics mini-app [15].

Among the benchmarks tested, Rodinia is a benchmark suite designed to test heterogeneous accelerators, but it also contains OpenMP version of their benchmarks for evaluation on the CPU side. In our evaluation, we choose 17 OpenMP benchmarks which run for a long enough time for performance measurement to be possible. They cover a wide range of topics, from medical imaging, fluid dynamics, data mining, to linear algebra. NPB-CPP is the C++ version of NAS Parallel Benchmarks [106] and supports various programming frameworks on shared-memory architectures including OpenMP. These benchmark programs focus on computational fluid dynamics (CFD). HPCG is a benchmark program that performs multigrid preconditioned conjugate gradient iterations, and it is a standard program to measure the performance of HPC systems. The problem sizes and original running time for these benchmark programs tested are shown in Table 5.1 (a). We use a combination of existing and customized result verification routines to determine the correctness of the results from mutated program variants.

Among the scientific applications tested, LULESH is a proxy application simulating the Shock Hydrodynamics Challenge Problem. CoMD is a proxy application implementing classical molecular dynamics algorithms and workloads as used in materials science. CoSP2

is a reference implementation for quantum molecular dynamics (QMD) electronic structure code. Lastly, CLOUDSC is a standalone mini-app of the ECMWF cloud microphysics parameterization for its Integrated Forecasting System (IFS). Evaluating these programs may show the efficacy of MUPPET in helping software developers in scientific computing optimize the parallel performance of their programs. Again, the problem sizes and original running time for these applications tested are shown in Table 5.1 (b). As for correctness checks, for both LULESH and CoMD, we use the approach presented in [17] to determine the correctness of the program. For LULESH, we consider iteration count, final origin energy, and TotalAbsDiff as the output; for CoMD, we use the final energy as output. For CoSP2, we consider the AAN and fraction values to determine whether the mutated program variant runs correctly. And lastly, for CLOUDSC, we use its internal verification routine for this purpose.

Algorithm Parameters

We use delta debugging, Bayesian optimization, and decision tree optimization in our evaluation. Given the fact that program running time varies across the programs being evaluated, we put a tryout limit of 100 for all programs tested across all three algorithms, instead of using a total time limit. Algorithms may finish before the tryout limit. Parameters for Bayesian optimization are `n_calls=100`, `n_initial_points=10`, and `noise="Gaussian"`, and parameters for decision tree optimization are `n_calls=100`, `n_initial_points=10`, `acq_func=`
`"LCB"`, `kappa=1.6`, `base_estimator="RF"`.

Evaluation Environment

We use a workstation computer with two 14-core Intel Xeon E5-2694v3 CPUs and 32GiB of RAM, running Ubuntu 22.04. We use Clang 16.0.6 with OpenMP 5.1 support as the compiler for both source-to-source code transformation. We use both `gcc` (Rodinia, CLOUDSC) and `clang` (others) to compile the original programs and their mutated variants, to demonstrate the adaptability of our tool to different compilers. Using OpenMP 5.1 enables us to build programs with `tiling` clauses as well.

We also ensure that performance variation is minimized between program runs. We avoid CPU context switching by limiting the programs to run on hardware threads on the second CPU by forcing the `taskset -c 14-27` command in FAROS. Hardware quiescing, as defined by [107], is also performed to reduce performance fluctuations, such as turning off both simultaneous multithreading and dynamic frequency scaling.

As for running time statistic collection, each mutated variant is run 3 times and use the minimum running time as the program running time T. As a comparison, we also record the average running time for each tryout and evaluate if there is any possible discrepancy between average and minimum running time, but this statistic is not used as the fitness function output for optimization algorithms. We use the minimum running time as the output of the fitness function because as stated in [107] it is best at rejecting noise introduced by the evaluation environment, since any running time higher than the minimum must be due to such noise. However, we still measure speedup for average running time to evaluate how performance variability affects running time across all programs.

5.5.2 RQ1: Speedup Discovered by MUPPET

Even though we take various measures to reduce performance variability in our evaluation system, it is still not completely removed. Therefore, to determine if a mutated program variant shows speedup, we use the 1% threshold. If among the 3 runs, the time improvement between the minimum running time or between the average running time is less than 1%, we do not consider the current subset of mutations as speedup-inducing.

The time improvement discovered by all three algorithms with MUPPET is shown in Table 5.2, where columns 2-4 show the time improvement when comparing the minimum running time of the best variant against the original, and columns 5-7 show the time improvement for the average running time. Our evaluation shows that there are 75.9% of programs (22 out of 29 programs; in which there are 18 out of 25 benchmark programs and 4 out of 4 scientific applications) in which delta debugging can find a subset of mutations that, when applied, can cause speedup while maintaining the correctness of the program. The other 7 programs show negligible (<1.01x) or no speedup, as shown in red background in Table 5.2. The largest speedup observed is from the `hotspot` benchmark in Rodinia, with a 257.59% time

Table 5.2: Mutation speedup discovered by delta debugging (DD), Bayesian optimization (BO), and decision tree optimization (FO). Results that are not considered as having speedup are highlighted in red.

	Min. Time Improvement			Avg. Time Improvement			No. of Mutations (collapse/simd/firstprivate/tile/schedule)			
Program	DD	BO	FO	DD	BO	FO	Original	DD	BO	FO
backprop	3.39%	3.60%	4.07%	5.09%	5.31%	5.67%	1/28/1/6/4	1/0/0/1/0	1/16/0/1/1	1/10/0/1/2
cfd	2.84%	2.84%	3.64%	2.60%	2.39%	3.20%	1/16/5/39/10	0/0/0/1/1	1/10/1/11/3	0/8/3/11/4
b+tree	0.38%	0.43%	0.47%	0.41%	0.47%	0.55%	0/30/2/6/4	0/2/0/0/0	1/13/1/0/1	0/11/2/2/2
heartwall	4.76%	4.97%	5.00%	4.70%	5.02%	5.04%	0/50/1/3/2	0/2/0/0/0	0/27/1/1/1	0/25/1/1/0
hotspot	9.34%	9.42%	9.42%	8.84%	9.53%	9.52%	0/5/0/0/0	0/3/0/0/0	0/5/0/0/0	0/4/0/0/0
hotspot3D	254.58%	257.59%	255.60%	254.64%	256.71%	255.26%	0/7/1/3/2	0/3/1/1/1	0/4/1/1/1	0/4/1/0/1
kmeans	2.13%	3.55%	1.00%	2.79%	3.03%	1.15%	0/22/0/3/2	0/16/0/0/0	0/12/0/0/0	0/10/0/1/1
lavaMD	0.58%	0.90%	0.78%	0.55%	0.83%	0.82%	0/12/1/18/2	0/6/0/3/0	0/7/1/6/1	0/5/0/3/1
leukocyte	2.63%	3.04%	3.47%	6.40%	6.37%	6.45%	0/135/3/54/4	0/9/0/0/0	0/53/1/17/2	0/67/1/16/1
lud	36.48%	38.33%	26.01%	27.80%	35.12%	22.02%	0/34/2/33/4	0/4/1/4/0	0/21/1/9/2	0/14/1/8/2
myocyte	3.67%	3.64%	4.38%	2.99%	3.10%	3.21%	0/41/2/66/2	0/40/2/22/1	0/20/0/16/0	0/24/1/19/1
nn	16.65%	17.03%	16.90%	16.52%	16.85%	16.66%	0/4/1/3/2	0/2/1/1/0	0/2/0/1/1	0/2/1/1/0
nw	0.39%	0.27%	1.56%	-0.52%	0.02%	1.72%	0/19/0/33/2	0/0/0/0/1	0/13/0/9/0	0/6/0/10/0
particlefilter	0.31%	0.15%	0.01%	1.73%	1.60%	1.51%	0/35/10/30/20	0/32/10/10/10	0/17/7/7/9	0/16/7/10/9
pathfinder	0.38%	0.40%	0.35%	2.58%	2.53%	2.38%	0/7/1/3/2	0/7/1/1/1	0/4/1/1/1	0/3/1/1/1
srad	1.74%	1.26%	1.26%	1.62%	1.62%	1.70%	2/9/2/6/4	0/1/0/0/1	0/7/0/2/2	0/6/2/2/1
streamcluster	2.30%	4.59%	3.12%	4.73%	4.85%	4.53%	0/25/2/0/0	0/2/0/0/0	0/15/0/0/0	0/13/1/0/0
BT	0.24%	-1.86%	-1.38%	0.14%	-1.84%	-1.40%	44/218/2/381/108	13/118/2/66/23	24/111/1/111/40	22/103/2/109/40
CG	1.28%	5.86%	5.02%	1.57%	2.92%	2.41%	0/18/11/27/14	0/18/9/9/7	0/9/5/9/7	0/9/8/9/5
EP	0.12%	0.11%	0.10%	0.11%	0.10%	0.10%	0/9/1/24/2	0/5/1/4/0	0/7/0/8/1	0/1/0/6/1
FT	1.80%	1.88%	1.25%	1.91%	1.88%	1.43%	1/42/5/45/12	0/6/1/1/1	0/27/3/11/3	1/23/4/10/3
LU	1.55%	2.62%	2.96%	1.43%	2.96%	2.88%	3/100/6/186/20	1/23/1/15/2	0/54/4/58/7	0/51/5/59/9
MG	15.30%	15.28%	15.67%	17.85%	17.90%	18.23%	7/66/8/39/20	3/28/4/6/3	5/34/5/12/7	4/32/6/10/9
SP	5.86%	-1.71%	-0.34%	5.89%	-1.65%	-0.81%	64/267/3/396/140	0/1/0/2/1	30/143/3/115/56	23/135/1/114/59
HPCG	4.19%	13.91%	13.21%	2.34%	8.09%	7.73%	0/63/13/81/26	0/4/0/5/0	0/36/7/25/12	0/36/7/25/12
LULESH	3.87%	2.32%	2.49%	6.29%	5.36%	5.66%	0/95/0/222/76	0/13/0/12/8	0/42/0/60/28	0/49/0/62/32
CoMD	3.34%	3.91%	4.36%	7.37%	7.86%	9.03%	0/78/13/132/30	0/24/3/10/4	0/32/8/36/12	0/47/6/36/12
CoSP2	3.80%	4.55%	6.27%	4.22%	5.10%	5.46%	0/67/9/117/22	0/17/0/12/1	0/32/5/34/10	0/44/4/36/8
CLOUDSC	1.07%	1.07%	1.20%	1.16%	3.46%	1.28%	0/58/0/111/0	0/28/0/19/0	0/34/0/34/0	0/33/0/33/0

improvement or 3.57x speedup. Three other benchmark programs have over 1.1x speedup (`lud` and `nn` in Rodinia, and `MG` in NPB-CPP).

On the other hand, both Bayesian optimization and decision tree optimization find the same number of programs—72.4% (21 out of 29 programs) in which a subset of mutations is found to induce speedup. They also find one more program, `HPCG`, with >1.1x speedup, compared to delta debugging. The programs with discovered speedup are the same across three algorithms, except in `SP` where delta debugging finds a subset of mutations that cause speedup, but the other two algorithms cannot.

Next, we compare the speedup discovered in individual programs by the three algorithms. Even though the speedup discovered by these algorithms looks roughly the same for a majority of programs, after comparing speedup for all programs, we find that in the 21 programs in which all algorithms find speedup in average running time, delta debugging can only find the maximum speedup among three algorithms in 2 programs, while Bayesian optimization and decision tree optimization finds the maximum speedup in 10 and 9 programs respectively. This shows that Bayesian optimization and decision tree optimization are slightly superior in finding the maximum speedup (when they can find any speedup at all). Also interesting to note, in some programs, such as `backprop` and `LULESH`, the speedup discovered for average running time is higher than the speedup discovered for minimum running time, which may suggest a reduction in performance variability in mutated program variants.

Answer to RQ1: Of all 29 programs tested, delta debugging can discover a subset of mutations that cause speedup in 22 (75.9%) of them; Bayesian optimization and decision tree optimization can discover speedup in 21 (69%) of them. The highest speedup discovered is 3.57x in `hotspot`. Bayesian optimization and decision tree optimization are slightly better than delta debugging in finding the highest speedups.

5.5.3 RQ2: Time Comparison between Different Algorithms

The time complexity of all algorithms in MUPPET is defined in Section 5.3.4 but they are all subject to performance variability in the fitness function which in MUPPET is the

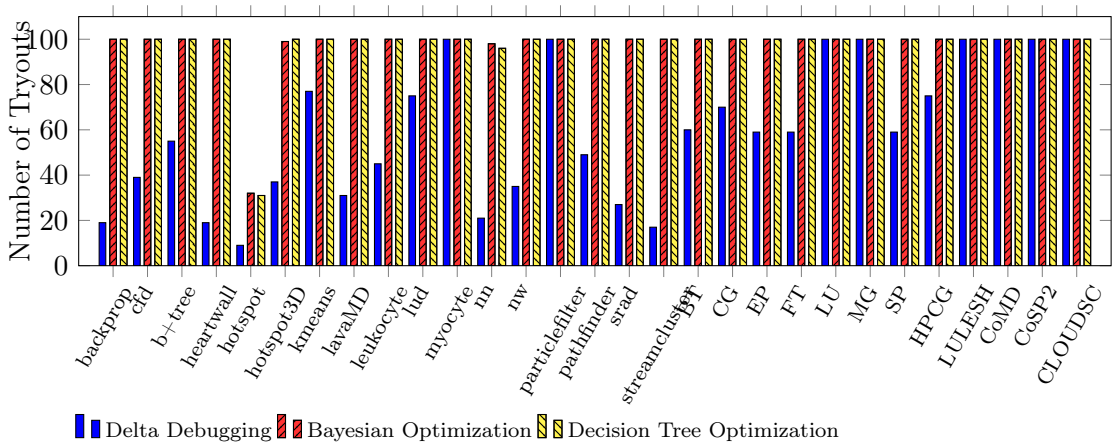


Figure 5.4: The number of tryouts used by each algorithm in MUPPET to terminate the optimization algorithm (maximum is 100).

measured running time. Therefore it is preferable to look into the actual performance of these algorithms in programs tested.

We collect information on the number of total tryouts attempted for each program by each algorithm with the 100 tryout limit, and the statistics are shown in Figure 5.4. For programs with a smaller amount of possible mutations, such as most benchmarks in Rodinia except `myocyte` and `particlefilter`, delta debugging can terminate before 100 tryouts. Even for NAS Parallel Benchmarks, there are still programs like `FT` and `SP` that terminate early. Meanwhile, for the other two algorithms, only 2 (`hotspot` and `nn` in Rodinia benchmarks) out of 29 programs terminated before 100 tryouts. This shows delta debugging is better than the other two algorithms in total time used when all three algorithms can find mutations that cause speedup in a program. In total, in 21 out of 29 programs, delta debugging uses fewer tryouts to finish its algorithm. In the remaining 8 programs, all algorithms terminate at the 100 tryout limit.

Our observation of logs shows that even when speedups can be discovered in relatively few tryouts for both BO and FO, the algorithms do not terminate and they continue to look for the next input that causes greater speedup. This is likely because of the noisy nature of the fitness function introduced by software and hardware environments.

Next, we evaluate how many tryouts it takes for different algorithms to first discover a subset of mutations that cause a non-negligible speedup. Figure 5.5 shows the maximum

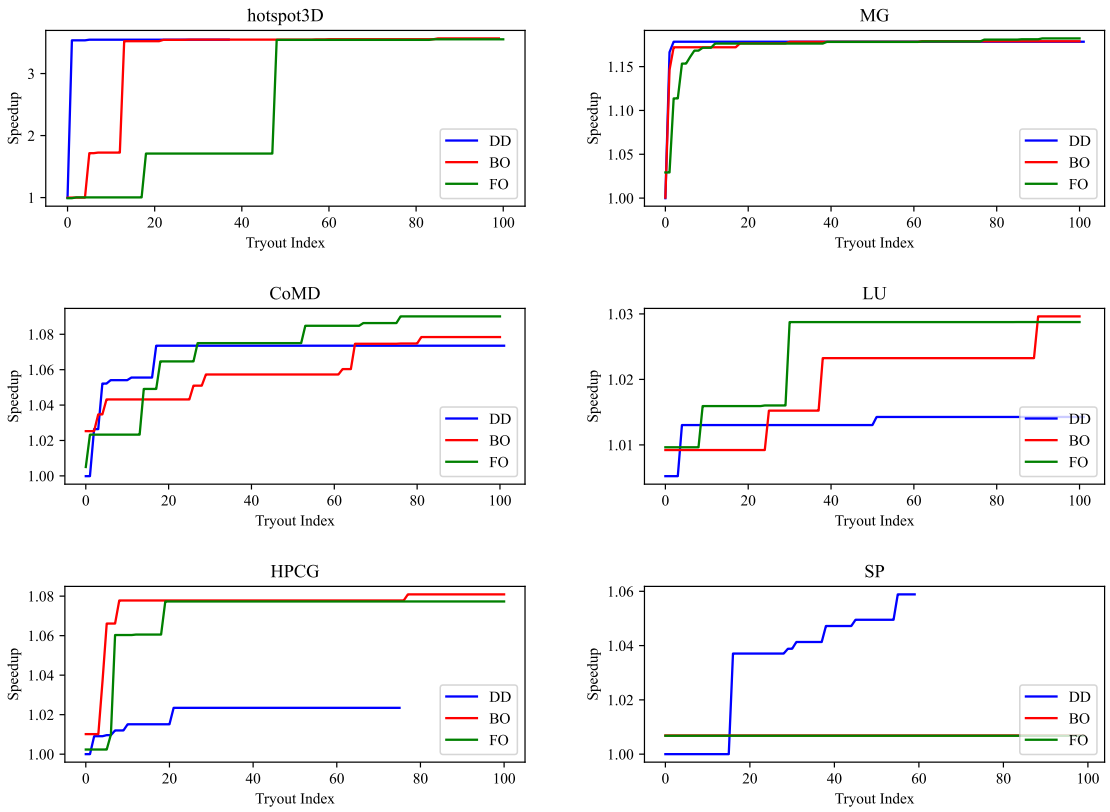


Figure 5.5: Time lapse graphs showing the speedup discovered for selected programs by MUPPET, using different algorithms.

speedup discovered by each algorithm for 6 programs over time. For the two programs in the top row, `hotspot3D` and `MG`, all three algorithms find a subset of mutations that cause roughly the same amount of high speedup, even though it takes decision tree optimization more tryouts to achieve that. For the two programs in the middle row, `CoMD`, and `LU`, delta debugging uses fewer tryouts to find a subset of mutations that cause relatively high speedup, but eventually, Bayesian optimization and decision tree optimization catch up and find a subset of mutations that has an even higher speedup. For `HPCG`, Bayesian optimization and decision tree optimization find much higher speedup than delta debugging almost immediately and even improved their highest speedup throughout the process. The findings for these three programs are consistent with Section 5.5.2. Lastly, we look into an example `SP` where only delta debugging finds speedup. It takes delta debugging almost 20 tryouts to find the first subset that contains the speedup-inducing mutations. Before that, delta debugging can only find subsets of mutations that slow down the program.

Answer to RQ2: In 21 out of 29 programs, delta debugging can finish its search algorithm faster than both Bayesian optimization and decision tree optimization. For the other 8 programs, MUPPET terminates at the 100 tryout limit. However, the other two algorithms, after more tryouts, can usually find the same or even higher speedup, except for `SP`.

5.5.4 RQ3: Analysis of Mutations Found by Different Algorithms

Next, we look at the subset of mutations that cause the highest speedup found by each algorithm. The number of possible mutations for each program, and the number of remaining mutations in those subsets are shown in columns 8-11 in Table 5.2. Note that except in programs like `hotspot` or `hotspot3D` where the number of possible mutations itself is small, the number of remaining mutations in the subset of mutations found by delta debugging is always much smaller than both Bayesian optimization and decision tree optimization. Since we know from Section 5.5.2 that delta debugging is slightly inferior when finding a subset of mutations that cause maximum speedup, it is likely that the mutations not included in the subset found by delta debugging causes minor speedup that contributes to the maximum.

Since the number of mutations found by delta debugging is small, next we look into them for some individual programs with the highest speedups discovered to find out what kind of mutations we can find that cause significant speedup. We achieve this by looking into the runtime logs of delta debugging. In `hotspot3D`, the mutations that cause speedup consist of a firstprivate clause at the `omp parallel` directive in the `computeTempOMP` function, and some SIMD directives in the multiple-dimension loop. In `MG`, two tiling mutations, when applied to the parallelized loops in functions `psinv()` and `resid()` induce 1.03x and 1.08x speedup, while some other mutations cause minor speedup.

Lastly, we also look into `SP`, a program where only delta debugging finds speedup. The runtime log of both Bayesian optimization and decision tree optimization shows that a lot of subsets of mutations these algorithms generate slow down the program, sometimes the running time is longer by as much as 25%. These two algorithms would likely take significantly more than 100 tryouts to eliminate all mutations with negative speedup, while delta debugging quickly isolates the 4 out of 870 mutations that cause speedup with only 60

tryouts. These few mutations are from SIMD and tiling mutations in the first two parallel loops in the `lhsz()` function. This shows that delta debugging is superior when there are many mutations that slow down the program.

Answer to RQ3: Among three algorithms, delta debugging isolates mutations causing significant speedup most quickly. Investigation into mutations that cause the highest speedups in programs evaluated shows that tiling mutations cause some of the most significant speedups discovered, while other mutations also contribute.

5.5.5 Discussion of Results and Limitations

Our evaluation shows that delta debugging is the fastest among all algorithms while discovering speedups in more programs tested. However, Bayesian optimization and decision tree optimization discover the highest speedups. Therefore, the choice between algorithms depends on the intended purpose of the end user. In a time-limited situation, it is preferable to use delta debugging to discover mutations that result in speedup; the other two algorithms are more suitable in situations where finding a speedup as high as possible is a priority.

The experimental evaluations in this chapter are performed with programs that are compiled with a fixed compiler and optimization flags, both for the original program and its mutated variants. We also use fixed problem sizes and input files for all programs. Even though we use different compilers to evaluate the performance of evaluated programs and the problem sizes for our programs are varied, we cannot ensure that the mutations discovered with MUPPET would also induce the same amount of speedup when either the compiler, optimization flags or runtime parameters are changed; the speedup may become larger or smaller. It is also possible that under different environments, a new performance hotspot may emerge so that running MUPPET in such an environment would return different mutations to achieve speedup. Furthermore, our evaluation is performed on a workstation with a limited number of CPUs and cores. Using different hardware (CPU, RAM, etc.) or using OpenMP offload may also change the speedup and/or the mutations discovered. Nonetheless, our approach is compatible with any C/C++ compiler in any hardware and software environment, and the mutations discovered are portable across compilers.

MUPPET is a dynamic tool that relies on mutation testing and it does not perform static performance modeling, nor does it perform instrumentation. The Mutation Generator, Translator, and the optimization algorithm take up negligible time during the running of MUPPET, thus the running time of MUPPET can be approximated as $T \cdot I$ where T is the running time of the original program and I is the number of tryouts, differed by algorithms. Thus the running time of MUPPET is linearly correlated with T, and would be much longer when T is larger, such as when the program has more possible OpenMP mutations, mutations that significantly slow down the mutated variant such as those in SP described in Section 5.5.4, and/or when the program is run with larger problem sizes. Even with `HPCG`, it takes MUPPET 5.5 hours to complete running with Bayesian optimization (4 hours with delta debugging) on the reference computer; while its original program only runs for 15.6 seconds. A combination of mutation testing and heuristics from program analysis may be needed to improve the search performance of MUPPET.

The efficacy of MUPPET also varies due to factors in program source code. MUPPET is a source-level approach; this means that some mutations may already be applied manually by software developers and are no longer possible mutations as seen by MUPPET. The amount of parallelism involved in the mutations may impact its efficacy as well, according to Amdahl's Law [108]. However, our experimental evaluation shows that MUPPET can still assist developers in finding previously unknown optimization opportunities, even in widely used, long-established performance benchmark programs such as `HPCG`. MUPPET can also be useful at finding optimization opportunities when migrating programs to newer OpenMP versions, as shown in examples like `MG` where tiling mutations that improve performance are discovered.

Lastly, due to time limitation, and given the fact that IR-level compiler optimizations are possibly already integrated into the compilers available, a comprehensive comparison between MUPPET and IR-level compiler optimizations is not performed. As stated in Section 5.1, since MUPPET is a portable, source-level approach, it can and should be used in addition to compiler optimizations. Software developers can use MUPPET to discover and apply mutations into their source code to speed up their programs and avoid leaving optimization opportunities on the table, regardless of the utilized compilers and their optimizations, which may be specific to a compiler and thus not portable.

5.6 Related Work

This section discusses related work in mutation testing, general auto-tuning techniques, and OpenMP-specific auto-tuning.

5.6.1 Mutation Testing

Mutation testing has already been proposed to identify correctness defects [91]. The assumption in mutation testing is that a syntactic change (a mutant) can help discover programs' defects. Mutation testing, however, has not been applied deeply in HPC programs and on performance defects.

Mutation Testing in Parallel and HPC Systems. Some attempts to build mutation testing for cloud systems have been reported [109]. Mutation operators (i.e., syntactic changes) have been proposed to reveal faults in small-size MPI programs [110]. With the increased use of LLVM, researchers are exploring the support of mutation testing in LLVM [111].

Mutation Testing for Performance. To the best of our knowledge, the only work that considers mutation testing for performance is [92]. However, this work does not consider parallelism and mutations in numerical (floating-point) code—these two aspects are critical to HPC applications. To the best of our knowledge, we are the first to explore using mutation testing for performance in OpenMP scientific codes.

5.6.2 General Auto-tuning

There are many past works on auto-tuning techniques.

Classic Auto-tuning Frameworks. Typical examples include ATLAS [112], Active Harmony [113], FFTW [114], POET [115], CHILL [116], GEIST [117], OpenTuner [118], CLTune [119], Apollo [120, 121], and Dutta et al. [122, 123]. Their common theme is that they tune compile-time, such as tiling, or runtime parameters, such as the number of threads, presupposing a given source code representation of a program. Typical search algorithms for tuning they propose include random, grid, or Bayesian search, or various machine learning-based search models.

By contrast, MUPPET mutates the source code of the program, which exposes a large, combined set of both source code modifications as compile-time parameters and their possible configurations as runtime parameters to tune for. Furthermore, MUPPET automates the generation of tuned source code variants without user intervention and it is the first to propose the delta debugging search algorithm for tuning. Integrating machine-learning techniques for fast searching in MUPPET is an interesting future extension.

Domain-Specific Tuning. A number of papers research domain-specific tuning using code generation, alternate data layouts, or algorithmic parameters, such as [124–128] for linear algebra kernels and [129–132] for stencils. Those approaches require users to express the programs in specialized domain-specific languages amenable to tuning, which limits their generality. MUPPET tunes unaltered, user-provided, general OpenMP code to generate tuning source code variants and optimizing runtime parameters.

5.6.3 Auto-tuning OpenMP

Specifically on OpenMP, several approaches have been proposed.

OpenMP Language Extensions. Adaptive OpenMP [133, 134] and Sreenivasan et al. [135] propose OpenMP language extensions to support auto-tuning on OpenMP regions, such as scheduling policies of parallel loops, number of threads or teams. Those approaches require significant refactoring of the code and domain-specific knowledge from the programmer to successfully integrate tuning extensions and their possible configuration parameters in their OpenMP code. Instead, MUPPET treats source code modifications as a tunable parameter and independently explores the runtime configuration space.

Bayesian Optimization for OpenMP. Bliss [136] proposes probabilistic Bayesian optimization to tune hardware (core frequency, hyperthreading) and software execution parameters (OpenMP threads, algorithmic alternatives) for the whole application, specified by the user. Bliss does not modify the program's source code and tunes all regions in unison; by contrast, MUPPET both enables source code modifications and specializes tuning to each region, since mutations are region-specific.

LLVM IR-Level Tuning. Scalable Record-Replay [137] is a mechanism that extracts the LLVM IR of OpenMP GPU target region kernels to tune for each kernel in parallel the

GPU launch bounds as compile-time parameters, by modifying the IR to re-compile, and the number of threads/teams as runtime parameters. Performing the kind of mutations in MUPPET on LLVM IR is challenging compared to source code, which motivates our choice of a source code mutation tool. Nevertheless, the idea of extracting OpenMP regions and tuning them independently is a possible extension to MUPPET to speed up search time.

5.7 Conclusion

We presented MUPPET, a novel application of mutation testing aimed at improving the performance of OpenMP programs. MUPPET uses different search algorithms to apply and compose program mutations to reduce application execution time. Because program transformations are performed at the source level, MUPPET's mutations are transferable across different OpenMP implementations and compilers. We demonstrate that MUPPET is capable of identifying mutations that improve performance in 75.9% of the evaluated programs achieving a maximum speedup of 3.57x.

In the future, we plan to extend MUPPET to automatically update OpenMP code bases with the latest OpenMP features that improve performance while maintaining correctness. Currently, it is the responsibility of the code maintainer to manually update their code base to use newly available OpenMP features, which require significant manual effort.

Chapter Acknowledgements

This chapter is based on the paper "MUPPET: Optimizing Performance in OpenMP via Mutation Testing" by Dolores Miao, Ignacio Laguna, Giorgis Georgakoudis, Konstantinos Parasyris, and Cindy Rubio-González, presented at *the 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM @ PPoPP)*, 2024, and extended in "An Automated OpenMP Mutation Testing Framework for Performance Optimization" in *Journal of Parallel Computing*, 2024. The source code and data of MUPPET are publicly available at https://github.com/LLNL/MUPPET/.

Funding: This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

(LLNL-JRNL-863899), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under award CCF-2119348.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

This dissertation has presented four complementary approaches to facilitate debugging and optimization of scientific applications through program analysis and testing techniques.

Chapter 2 presented Ciel, the first tool that automatically isolates numerical inconsistencies in heterogeneous programs at the expression level. Ciel employs a hierarchical search approach that operates on simplified ASTs to provide finer granularity than prior line-level approaches, coupled with a precision enhancement strategy that addresses challenges specific to transforming heterogeneous code to higher precision. Evaluation demonstrated 99.4% precision in isolating inconsistencies across 330 synthetic GPU programs, NAS and Rodinia benchmarks, and the real-world CLOUDSC mini-app, while achieving 24.5% fewer searches compared to pLiner, the state of the art for CPU programs.

Chapter 3 presented CIGEN, a tool that finds input ranges that cause high compiler-induced inconsistencies in numerical programs. CIGEN combines a multi-phase approach with input-partitioned sampling, clustering, and differential evolution, and provides a platform and compiler independent implementation that works as a black-box tool. Evaluation on 175 GNU Scientific Library functions found 53.4% more functions with high inconsistencies than the state of the art, and revealed important characteristics of inconsistency-inducing input ranges.

Chapter 4 presented FloatGuard, the first tool to detect floating-point exceptions in HIP programs running on AMD GPUs. FloatGuard utilizes AMD GPU hardware registers for exception detection and combines assembly-level and source-level code instrumentation with debugger-guided execution. The tool detected floating-point exceptions in 89.7% of evaluated programs (507 out of 565) and provided a comprehensive comparison between AMD HIP and NVIDIA CUDA exception behavior.

Together, these three tools provide a comprehensive workflow for debugging floating-point correctness: CIGEN helps discover which inputs trigger inconsistencies, Ciel localizes the root cause in the source code, and FloatGuard validates correctness at runtime by detecting floating-point exceptions.

Chapter 5 presented Muppet, a mutation testing framework for optimizing OpenMP parallelism while maintaining correctness. Muppet applies five classes of source mutations to OpenMP directives and integrates with delta debugging, Bayesian optimization, and decision tree optimization to suggest performance improvements. The framework achieved performance improvements in 75.9% of evaluated programs, with the best speedup of 3.57x, while maintaining a source-level approach that is portable across compilers.

6.1.1 Broader Impact

The tools and techniques presented in this dissertation address fundamental challenges in scientific computing that affect researchers across many domains:

Numerical Reliability. As scientific applications are ported to new architectures and compiled with different toolchains, the tools in this dissertation help ensure that numerical results remain consistent and reliable. This is particularly important for reproducibility in scientific research.

Cross-Platform Computing. With the increasing diversity of HPC architectures, including the adoption of AMD GPUs in exascale systems like Frontier and El Capitan, tools like FloatGuard enable developers to validate numerical correctness across different platforms.

Developer Productivity. By automating the isolation of numerical issues (from weeks of manual effort to minutes) and suggesting performance optimizations, these tools significantly improve developer productivity in scientific computing.

Performance and Correctness Balance. The dissertation demonstrates that aggressive compiler optimizations can improve performance but may introduce numerical inconsistencies. The tools help developers understand and manage this tradeoff.

6.2 Future Research Directions

Several promising directions for future research emerge from this work:

Extended Platform Support. Extending Ciel to additional programming models such as HIP and SYCL and extending Muppet to support other parallel programming models such as MPI and OpenMP target offloading would broaden their applicability. Furthermore, extending Ciel to support isolation of cross-GPU-vendor inconsistencies would be valuable for developers porting applications to new architectures.

Gray- or white-box Approaches. Integrating more information about the program structure to guide existing algorithms, and using methodologies that operate on program source such as symbolic execution and shadow value could further improve the efficiency of the search process in Ciel and CIGEN.

Machine Learning Approaches. Machine learning techniques could further enhance tools in this dissertation. For example, graph neural networks could analyze code structure to predict where inconsistencies are likely in Ciel, improving search efficiency beyond AST-based approaches. Reinforcement learning could optimize the mutation search strategy in Muppet, adapting over time to discover more effective optimizations.

Integration and Extension of Tools. A unified framework that combines input generation, fault isolation, and exception detection could provide a comprehensive solution framework for implementation-specific numerical debugging. Beyond integration, moving toward automated repair of numerical issues by suggesting and automatically applying code transformations that eliminate inconsistencies would further enhance developer productivity.

6.3 Concluding Remarks

Reliable scientific software is fundamental to advancing research across all domains that depend on computational methods. This dissertation has contributed new tools and techniques that help developers build more reliable and efficient scientific applications. As computing systems continue to evolve in complexity, the need for automated tools to ensure numerical correctness and optimize performance will only increase. The approaches presented here provide a foundation for addressing these challenges in current and future HPC systems.

Bibliography

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, 2010.

[2] H. Guo, I. Laguna, and C. Rubio-González, "pliner: isolating lines of floating-point code for compiler-induced variability," in *SC*, p. 49, IEEE/ACM, 2020.

[3] CEED, "Laghos (lagrangian high-order solver)." https://computing.llnl.gov/projects/co-design/laghos, 2018.

[4] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping science on keel when software moves," *Commun. ACM*, vol. 64, no. 2, pp. 66–74, 2021.

[5] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn, "FLiT: Cross-platform floating-point result-consistency tester and workload," in *IISWC*, pp. 229–238, IEEE Computer Society, 2017.

[6] "Compiling CUDA with Clang," 2022.

[7] "CUDA LLVM compiler," Oct 2018.

[8] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *ASE*, pp. 509–519, IEEE Computer Society, 2017.

[9] A. Zeller, "Yesterday, my program worked. today, it does not. why?," in *ESEC / SIGSOFT FSE*, vol. 1687 of *Lecture Notes in Computer Science*, pp. 253–267, Springer, 1999.

[10] M. Joldes, J. Muller, V. Popescu, and W. Tucker, "CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications," in *ICMS*, vol. 9725 of *Lecture Notes in Computer Science*, pp. 232–240, Springer, 2016.

[11] T. Nakayama and D. Takahashi, "Implementation of multiple-precision floating-point arithmetic library for GPU computing," in *PDCS*, pp. 343–349, 2011.

[12] M. Lu, B. He, and Q. Luo, "Supporting extended precision on graphics processors," in *DaMoN*, pp. 19–26, ACM, 2010.

[13] G. A. de Araujo, D. Griebler, M. Danelutto, and L. G. Fernandes, "Efficient NAS parallel benchmark kernels with CUDA," in *PDP*, pp. 9–16, IEEE, 2020.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, pp. 44–54, IEEE Computer Society, 2009.

[15] ECMWF, "Standalone mini-app of the ECMWF cloud microphysics parameterization," 2022.

[16] I. Laguna, "Varity: Quantifying floating-point variations in HPC systems through randomized testing," in *IPDPS*, pp. 622–633, IEEE, 2020.

[17] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "Gpumixer: Performance-driven floating-point tuning for GPU scientific applications," in *ISC*, vol. 11501 of *Lecture Notes in Computer Science*, pp. 227–246, Springer, 2019.

[18] ECMWF, "CLOUDSC-V3: Re-create the single-exponent bug in the c variant," 2019.

[19] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz, "Clock delta compression for scalable order-replay of non-deterministic parallel applications," in *SC*, pp. 62:1–62:12, ACM, 2015.

[20] "Clang 14.0.0 documentation," 2022.

[21] I. Laguna, "FPChecker: Detecting floating-point exceptions in GPU applications," in *ASE*, pp. 1126–1129, IEEE, 2019.

[22] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: precision testing of deep learning operators," in *ISSTA*, pp. 400–412, ACM, 2021.

[23] S. Chowdhary and S. Nagarakatte, "Parallel shadow execution to accelerate the debugging of numerical errors," in *ESEC/SIGSOFT FSE*, pp. 615–626, ACM, 2021.

[24] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," in *PLDI*, pp. 256–269, ACM, 2018.

[25] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *PLDI*, pp. 453–462, ACM, 2012.

[26] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *PLDI*, pp. 216–226, ACM, 2014.

[27] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *PLDI*, pp. 85–99, ACM, 2016.

[28] Q. Zhu and A. Zaidman, "Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to GPU programs," in *ICST*, pp. 209–219, IEEE, 2020.

[29] Q. Zhang, J. Wang, and M. Kim, "HeteroFuzz: fuzz testing to detect platform dependent divergence for heterogeneous applications," in *ESEC/SIGSOFT FSE*, pp. 242–254, ACM, 2021.

[30] J. Vanover, X. Deng, and C. Rubio-González, "Discovering discrepancies in numerical libraries," in *ISSTA*, pp. 488–501, ACM, 2020.

[31] D. Miao, I. Laguna, and C. Rubio-González, "Expression isolation of compiler-induced numerical inconsistencies in heterogeneous code," in *ISC*, vol. 13948 of *Lecture Notes in Computer Science*, pp. 381–401, Springer, 2023.

[32] H. Yu, X. Yi, B. Yin, F. Li, Z. Chen, and C. Huang, "Efficient generation of floating-point inputs for compiler-induced variability," in *SANER*, pp. 224–235, IEEE, 2023.

[33] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Trans. Evol. Comput.*, vol. 15, no. 1, pp. 4–31, 2011.

[34] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *PLDI*, pp. 1–11, ACM, 2015.

[35] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, pp. 53–64, ACM, 2014.

[36] J. Mockus, "Application of bayesian approach to numerical methods of global and stochastic optimization," *J. Glob. Optim.*, vol. 4, no. 4, pp. 347–365, 1994.

[37] C. J. Geyer, "Practical markov chain monte carlo," *Statistical science*, pp. 473–483, 1992.

[38] I. Laguna and G. Gopalakrishnan, "Finding inputs that trigger floating-point exceptions in gpus via bayesian optimization," in *SC*, pp. 33:1–33:14, IEEE, 2022.

[39] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *CoRR*, vol. abs/1109.2378, 2011.

[40] M. Galassi, J. Davies, J. Theiler, B. Gough, and G. Jungman, *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12*. Network Theory Ltd, 2009.

[41] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *PPoPP*, pp. 43–52, ACM, 2014.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[43] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy, "Scipy 1.0-fundamental algorithms for scientific computing in python," *CoRR*, vol. abs/1907.10121, 2019.

[44] S. Chowdhary, J. P. Lim, and S. Nagarakatte, "Debugging and detecting numerical errors in computation with posits," in *PLDI*, pp. 731–746, ACM, 2020.

[45] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.*, vol. 4, no. 2, pp. 71–86, 2017.

[46] W. Lee, T. Bao, Y. Zheng, X. Zhang, K. Vora, and R. Gupta, "RAIVE: runtime assessment of floating-point instability by vectorization," in *OOPSLA*, pp. 623–638, ACM, 2015.

[47] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 60:1–60:27, 2020.

[48] H. Guo and C. Rubio-González, "Efficient generation of error-inducing floating-point inputs via symbolic execution," in *ICSE*, pp. 1261–1272, ACM, 2020.

[49] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, pp. 209–224, USENIX Association, 2008.

[50] Y. Xia, S. Guo, J. Hao, D. Liu, and J. Xu, "Error detection of arithmetic expressions," *J. Supercomput.*, vol. 77, no. 6, pp. 5492–5509, 2021.

[51] D. Zou, Y. Gu, Y. Shi, M. Wang, Y. Xiong, and Z. Su, "Oracle-free repair synthesis for floating-point programs," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 957–985, 2022.

[52] X. Yi, L. Chen, X. Mao, and T. Ji, "Efficient automated repair of high floating-point errors in numerical libraries," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 56:1–56:29, 2019.

[53] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *ICSE (1)*, pp. 529–539, IEEE Computer Society, 2015.

[54] R. Rabe, A. Izycheva, and E. Darulova, "Regime inference for sound floating-point optimizations," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, pp. 81:1–81:23, 2021.

[55] A. H. Zahid, I. Laguna, and W. Le, "Testing GPU numerics: Finding numerical differences between NVIDIA and AMD gpus," in *SC Workshops*, pp. 547–557, IEEE, 2024.

[56] I. Laguna, "Fpchecker: Detecting floating-point exceptions in GPU applications," in *ASE*, pp. 1126–1129, IEEE, 2019.

[57] I. Laguna, T. Tirpankar, X. Li, and G. Gopalakrishnan, "Fpchecker: Floating-point exception detection tool and benchmark for parallel and distributed HPC," in *IISWC*, pp. 39–50, IEEE, 2022.

[58] I. Laguna, X. Li, and G. Gopalakrishnan, "Binfpe: accurate floating-point exception detection for GPU applications," in *SOAP*.

[59] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, and G. Gopalakrishnan, "Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA gpus," in *HPDC*, pp. 59–71, ACM, 2023.

[60] AMD, "ROCm: Platform for GPU-Enabled High-Performance Computing," 2025. Accessed: 2025-02-05.

[61] AMD, "ROCgdb, the ROCm source-level debugger for Linux," 2024. Accessed: 2024-10-03.

[62] GNU Project, *GDB: The GNU Project Debugger*, 2024.

[63] ""AMD Instinct MI200" Instruction Set Architecture Reference Guide," 2022. [Accessed 16-09-2024].

[64] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, pp. 44–54, IEEE Computer Society, 2009.

[65] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*, pp. 1–10, Ieee, 2012.

[66] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *ISCA*, pp. 473–486, IEEE, 2020.

[67] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, no. 7.2, 2012.

[68] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *GPGPU*, vol. 425 of *ACM International Conference Proceeding Series*, pp. 63–74, ACM, 2010.

[69] AMD, "rocHPCG," 2017.

[70] J. J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 3–10, 2016.

[71] AMD, "Hipify: Convert cuda to portable c++ using hip," 2023.

[72] G. Gopalakrishnan, I. Laguna, A. Li, P. Panchekha, C. Rubio-González, and Z. Tatlock, "Guarding numerics amidst rising heterogeneity," in *Correctness*.

[73] NVIDIA Corporation, "NVIDIA HPCG Benchmark," 2024. Accessed: 2024-10-03.

[74] O. Villa, M. Stephenson, D. W. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for NVIDIA gpus," in *MICRO*, pp. 372–383, ACM, 2019.

[75] M. R. Ardakani, A. Nguyen, I. Rosales, D. Xu, Y. Sun, Y. Sun, D. Kaeli, and N. Rubin, "Luthier: A dynamic binary instrumentation framework targeting AMD gpus," in *ISPASS*, pp. 137–149, IEEE, 2025.

[76] P. A. Dinda, A. Bernat, and C. Hetland, "Spying on the floating point behavior of existing, unmodified scientific applications," in *HPDC*, pp. 5–16, ACM, 2020.

[77] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *POPL*, pp. 549–560, ACM, 2013.

[78] J. Vanover, J. Demmel, X. S. Li, and C. Rubio-González, "Excvate: Spoofing exceptions and solving constraints to test exception handling in numerical libraries," in *ARITH*, pp. 109–116, IEEE, 2025.

[79] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.*, vol. 4, no. 2, pp. 71–86, 2017.

[80] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error with herbgrind," *CoRR*, vol. abs/1705.10416, 2017.

[81] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 685–701, 2010.

[82] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 702–719, 2010.

[83] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," in *HPCS*, pp. 898–907, IEEE, 2017.

[84] S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[85] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, Citeseer, 1999.

[86] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *Green Computing Conference*, pp. 115–122, IEEE Computer Society, 2010.

[87] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: an openmp tools application programming interface for performance analysis," in *IWOMP*, vol. 8122 of *Lecture Notes in Computer Science*, pp. 171–185, Springer, 2013.

[88] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[89] Y. J. Lo, S. Williams, B. van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *PMBS@SC*, vol. 8966 of *Lecture Notes in Computer Science*, pp. 129–148, Springer, 2014.

[90] N. Ding and S. Williams, "An instruction roofline model for gpus," in *PMBS*.

[91] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[92] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Software Testing, Verification and Reliability*, p. e1728, 2020.

[93] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[94] J. Fürnkranz, *Decision Tree*, pp. 263–267. Boston, MA: Springer US, 2010.

[95] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *SC*, pp. 27:1–27:12, ACM, 2013.

[96] G. Lan, J. M. Tomczak, D. M. Roijers, and A. E. Eiben, "Time efficiency in optimization with a bayesian-evolutionary algorithm," *Swarm Evol. Comput.*, vol. 69, p. 100970, 2022.

[97] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[98] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, "scikit-optimize/scikit-optimize," Oct. 2021.

[99] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, p. 1, Citeseer, 2011.

[100] G. Georgakoudis, J. Doerfert, I. Laguna, and T. R. W. Scogland, "FAROS: A framework to analyze openmp compilation through benchmarking and compiler optimization analysis," in *IWOMP*, vol. 12295 of *Lecture Notes in Computer Science*, pp. 3–17, Springer, 2020.

[101] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proc. Workload Characterization (IISWC), 2010 IEEE International Symposium on*, (Atlanta, GA, USA), pp. 1–11, IEEE Computer Society, 2010.

[102] J. Löff, D. Griebler, G. Mencagli, G. A. de Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures," *Future Gener. Comput. Syst.*, vol. 125, pp. 743–757, 2021.

[103] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. D. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. F. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IPDPS*, pp. 919–932, IEEE Computer Society, 2013.

[104] T. E. C.-D. C. for Materials in Extreme Environments (ExMatEx), "Comd - classical molecular dynamics proxy application." https://github.com/ECP-copa/CoMD, 2013.

[105] ExMatEx, "CoSP2 proxy application." https://proxyapps.exascaleproject.org/app/cosp2/.

[106] D. H. Bailey, "NAS parallel benchmarks," in *Encyclopedia of Parallel Computing*, pp. 1254–1259, Springer, 2011.

[107] "Performance engineering of software systems," 2018.

[108] D. P. Rodgers, "Improvements in multiprocessor system design," in *ISCA*, pp. 225–231, IEEE Computer Society, 1985.

[109] P. C. Cañizares, A. Núñez, and M. G. Merayo, "Mutomvo: Mutation testing framework for simulated cloud and HPC environments," *J. Syst. Softw.*, vol. 143, pp. 187–207, 2018.

[110] R. A. Silva, S. do Rócio Senger de Souza, and P. S. L. de Souza, "Mutation operators for concurrent programs in MPI," in *LATW*, pp. 1–6, IEEE Computer Society, 2012.

[111] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on LLVM," in *ICST Workshops*, pp. 25–31, IEEE Computer Society, 2018.

[112] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC*, p. 38, IEEE Computer Society, 1998.

[113] C. Tapus, I. Chung, and J. K. Hollingsworth, "Active harmony: towards automated performance tuning," in *SC*, pp. 43:1–43:11, IEEE Computer Society, 2002.

[114] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[115] Q. Yi, K. Seymour, H. You, R. W. Vuduc, and D. J. Quinlan, "POET: parameterized optimizations for empirical tuning," in *IPDPS*, pp. 1–8, IEEE, 2007.

[116] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," tech. rep., Citeseer, 2008.

[117] J. J. Thiagarajan, N. Jain, R. Anirudh, A. Giménez, R. Sridhar, A. Marathe, T. Wang, M. Emani, A. Bhatele, and T. Gamblin, "Bootstrapping parameter space exploration for fast tuning," in *ICS*, pp. 385–395, ACM, 2018.

[118] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe, "Opentuner: an extensible framework for program autotuning," in *PACT*, pp. 303–316, ACM, 2014.

[119] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *MCSoC*, pp. 195–202, IEEE Computer Society, 2015.

[120] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *IPDPS*, pp. 307–316, IEEE Computer Society, 2017.

[121] C. Wood, G. Georgakoudis, D. Beckingsale, D. Poliakoff, A. Giménez, K. A. Huck, A. D. Malony, and T. Gamblin, "Artemis: Automatic runtime tuning of parallel execution parameters using machine learning," in *ISC*, vol. 12728 of *Lecture Notes in Computer Science*, pp. 453–472, Springer, 2021.

[122] A. Dutta, J. Alcaraz, A. TehraniJamsaz, A. Sikora, E. César, and A. Jannesari, "Pattern-based autotuning of openmp loops using graph neural networks," in *AI4S*, pp. 26–31, IEEE, 2022.

[123] A. Dutta, J. Alcaraz, A. TehraniJamsaz, E. César, A. Sikora, and A. Jannesari, "Performance optimization using multimodal modeling and heterogeneous GNN," in *HPDC*, pp. 45–57, ACM, 2023.

[124] M. Gates, J. Kurzak, P. Luszczek, Y. Pei, and J. J. Dongarra, "Autotuning batch cholesky factorization in CUDA with interleaved layout of matrices," in *IPDPS Workshops*, pp. 1408–1417, IEEE Computer Society, 2017.

[125] R. Nath, S. Tomov, J. Dongarra, and E. Agullo, "Autotuning dense linear algebra libraries on gpus and overview of the magma library," in *6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*, 2010.

[126] J. Kurzak, H. Anzt, M. Gates, and J. J. Dongarra, "Implementation and tuning of batched cholesky factorization and solve for NVIDIA gpus," *IEEE Trans. Parallel Distributed Syst.*, vol. 27, no. 7, pp. 2036–2048, 2016.

[127] W. A. Abu-Sufah and A. A. Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," in *ISC*, vol. 7905 of *Lecture Notes in Computer Science*, pp. 151–164, Springer, 2013.

[128] J. J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, and Y. M. Tsai, "Autotuning numerical dense linear algebra for batched computation with GPU hardware accelerators," *Proc. IEEE*, vol. 106, no. 11, pp. 2040–2055, 2018.

[129] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance GPU code for stencil computations," *Proc. IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[130] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "AN5D: automated stencil framework for high-degree temporal blocking on gpus," in *CGO*, pp. 199–211, ACM, 2020.

[131] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, and P. Sadayappan, "On optimizing complex stencils on gpus," in *IPDPS*, pp. 641–652, IEEE, 2019.

[132] X. You, H. Yang, Z. Jiang, Z. Luan, and D. Qian, "Drstencil: Exploiting data reuse within low-order stencil on GPU," in *HPCC/DSS/SmartCity/DependSys*, pp. 63–70, IEEE, 2021.

[133] C. Liao, D. J. Quinlan, R. W. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *LCPC*, vol. 5898 of *Lecture Notes in Computer Science*, pp. 308–322, Springer, 2009.

[134] G. Georgakoudis, K. Parasyris, C. Liao, D. Beckingsale, T. Gamblin, and B. R. de Supinski, "Machine learning-driven adaptive openmp for portable performance on heterogeneous systems," *CoRR*, vol. abs/2303.08873, 2023.

[135] V. Sreenivasan, R. Javali, M. W. Hall, P. Balaprakash, T. R. W. Scogland, and B. R. de Supinski, "A framework for enabling openmp autotuning," in *IWOMP*, vol. 11718 of *Lecture Notes in Computer Science*, pp. 50–60, Springer, 2019.

[136] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models," in *PLDI*, pp. 1280–1295, ACM, 2021.

[137] K. Parasyris, G. Georgakoudis, E. Rangel, I. Laguna, and J. Doerfert, "Scalable tuning of (openmp) GPU applications via kernel record and replay," in *SC*, pp. 28:1–28:14, ACM, 2023.