UNIVERSITY OF CALIFORNIA

Los Angeles

Hardware-Assisted Software Testing and Debugging for Heterogeneous Computing

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Jiyuan Wang

2025

ABSTRACT OF THE DISSERTATION

Hardware-Assisted Software Testing and Debugging for Heterogeneous Computing

by

Jiyuan Wang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2025
Professor Miryung Kim, Chair

There is a growing interest in the computer architecture community to incorporate heterogeneity and specialization to improve performance. Developers can write *heterogeneous applications* that consist of *host* code and *kernel* code, where compute-intensive kernels can be offloaded from CPU to GPU, FPGA, or quantum computer. However, the high complexity of these systems can pose challenges to developer productivity, particularly when it comes to understanding performance or ensuring correctness. Testing and debugging such heterogeneous applications and software stacks are also extremely challenging because the characteristics of the hardware in heterogeneous computing can vary from a traditional CPU. For example, the output of a quantum computer is completely different from a CPU.

By leveraging hardware accelerator capability and by accounting for hardware accelerator behavior, this thesis presents efficient debugging and testing techniques for heterogeneous computing. My first work QDiff is a differential testing framework for quantum software stacks to find *unexpected behavior*, such as crashes or unexpected divergences. By accounting for unique characteristics of quantum circuit execution and measurement, QDiff automatically finds abnormalities in quantum software stacks by detecting quantum circuit result divergence. QDiff proposes an input generation approach that applies gate-level equivalent transformation and explores the backends and compiler setting options to generate *semantic equivalent quantum circuits*. We then compare the measurement results of circuits with *K-S distance* and report unexpected results. For Cirq, Pyquil, and Qiskit, we found four new bugs in their simulators and two possible root causes for hardware execution divergence.

While QDiff targets the correctness of quantum software stacks, bugs in heterogeneous applications are another concern in this domain. My second work HFuzz designs a fuzz testing technique for heterogeneous applications. By designing hardware-level probes and offloading input mutations to hardware accelerators, HFuzz leverages in-kernel probes to retrieve hardware execution feedback, including channel usage and in-kernel variable's value range. HFuzz also uses FPGA to accelerate input mutations during fuzzing process. We conduct detailed experiments on seven benchmarks and demonstrate that HFuzz significantly improves the fuzzing efficiency and reveals 25 unique and unexpected behavior symptoms that could not be found by state-of-the-art testing techniques.

QDiff and HFuzz together provide automated testing approaches for heterogeneous computing. However, developers must be able to diagnose the detected errors. Compilation for heterogeneous applications is inherently complex. For example, CIRCT, an MLIR-based heterogeneous compiler, requires 13 compilation layers to translate high-level Python code into low-level RTL. Identifying which layers and which parts of the intermediate representation (IR) contribute to bugs is difficult. To tackle this issue, my third work, DuoReduce, introduces a novel dual-dimensional error localization approach. DuoReduce systematically analyzes errors across both IR code dimension and compilation path dimension within the compilation process. By combining delta debugging techniques with dependency-aware compilation path reduction, DuoReduce identifies the minimal subset of IR code and compilation passes responsible for triggering an error. Through evaluation on real-world scenarios, DuoReduce demonstrated significant improvements in debugging accuracy and efficiency, accelerating error localization by 901x compared to traditional techniques.

The dissertation of Jiyuan Wang is approved.

Harry Guoqing Xu

Jens Palsberg

Anthony John Nowatzki

Miryung Kim, Committee Chair

University of California, Los Angeles

2025

To my family and my passed grandma

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The past six years were the best years of my entire life. When I look back, I still remember the first day I met my advisors Miryung Kim and Harry Xu. It was during the Spring Festival, and at that time, I still hadn't received any PhD offers. I was deeply worried and sent emails to potential advisors to request interviews. That's when Miryung, like a superhero, stepped in and saved not only my winter holiday but also the next six years of my life. Though Miryung didn't give any immediate positive signals (I'm still striving to adopt to her style), Harry asked, "If we give you the offer, will you come?"

To be honest, I never felt like I was a "qualified" UCLA PhD student in the beginning. Before leaving for the US, all the students from my home university who had been admitted to UCLA gathered for a farewell dinner. I remember they were all either summa cum laude or cum laude graduates from Tsinghua—except me. My peers who joined the lab the same year, Haoran and Yifan, had exceptionally strong computer science backgrounds, and today they're both incredibly successful researchers. I want to prove I am qualified, both as a student, and a researcher.

I knew I did.

Life was never easy as a PhD student, especially during the covid. After the first quarter, just as I was beginning to learn how to conduct research, the world came to a halt. COVID-19 began, and suddenly everyone—including my advisors, Miryung Kim and Harry Xu, and my mentor, Qian Zhang—shifted to working from home. I found myself locked in a small apartment with my roommate and his girlfriend. Despite the isolation, I didn't feel lonely. My advisors provided me with more support during that difficult time than I could have ever imagined. I am also deeply grateful to Yifan Qiao and Haoran Ma, who often invited me over to their apartment for dinner

After two years in Los Angeles, I decided to visit my parents in China. It was a time of personal hardship; my father nearly lost his job and was in a very low emotional state.

Seeing him like that was heartbreaking. However, when I tried to return to the US, the visa approval process took nearly four months, and the delay deeply affected my productivity that summer and into the fall quarter. While everyone else returned to campus, I was still stuck in China. During this difficult period, my advisors showed me immense patience—patience that I couldn't even muster for myself.

If I list all the troubles I met during the PhD, then the acknowledgments will become 100 pages long. I want to express my gratitude to all the wonderful people who gave me support along the journey.

I would like to thank my committee members, Miryung Kim, Harry Xu, Jens Palsberg, and Tony Nowatzki. Thanks for your time and feedback on my research.

Thank you to my advisor, Miryung Kim. You once told me that your daughter is your first priority and your PhD students are your second priority. At the time, I thought it was a joke, but I realized you truly meant it. Your unwavering support has been invaluable. You taught me how to navigate the complexities of research, guided me when I felt lost in the dark, and stood by me during my most challenging times.

Thank you to my co-advisor, Harry Xu. I am deeply grateful for the invaluable insights you have provided for my research, as well as the thoughtful advice you have offered for life beyond academia. Being co-advised by you and Miryung has not only broadened my perspective on research but also enriched my understanding of life from different angles.

Thank you to my mentor and friend, Qian Zhang. I deeply appreciate all the guidance and countless hours you have dedicated to supporting me. I know I wasn't always the easiest collaborator, and I am grateful for your patience, understanding, and advocacy on my behalf. Working with you has been a privilege, and I am confident you will be an exceptional professor.

Thank you, my intern manager and my friend, Antonio Filieri. Amazon might not be the best company, but you are the best employee and best manager. Wish you all the best for your career and family.

Thank you to my collaborators, Yuxin Qiu, Hongbo Rong, and Rohan Padhye. Thanks for the brilliant ideas and valuable comments.

Thank you to my dear lab mates, Yifan Qiao, Haoran Ma, Jason Teoh, Muhammad Ali Gulzar, Yaoxuan Wu, Ben Limpanukorn, Zitong Zhou, and Hong Jin Kang. Thanks for the brainstorming and support as friends.

Thank you to my love, Jiayi. Your unwavering support—emotional, practical, and financial—has been a constant source of strength for me. The joy, compromises, and sacrifices you've made will forever leave an indelible mark on my heart.

Thank you to my cousin, Liang Shang. Having a family member in this foreign land gives me much more strength than I could ever imagine. Your presence means the world to me.

Thank you to my grandma. You raised me up for my childhood and taught me how to recognize the words one by one. I really want to say how much I love you in person, and I regret I did not use my last chance to say it. COVID took too much than I could bear.

Thank you to my parents. My name, "Jiyuan," means "to go far," and I hope I have lived up to the dreams you held for me. I can only wish to return even a fraction of the boundless love you have given me.

Thank you Jiyuan. No one is the arbiter of who does and doesn't have value, and you should always remember this.

VITA

2019 - 2025	Graduate Student Researcher/Teaching Assistant University of California, Los Angeles
2019 - 2022	M.S. in Computer Science University of California, Los Angeles
2015 - 2019	B.S. in Physics Tsinghua University

PUBLICATIONS

Jiyuan Wang, Yuxin Qiu, Ben Limpanukorn, Hong Jin Kang, Qian Zhang, Miryung Kim, DuoReduce: Bug Isolation for Multi-Layer Extensible Compilation, FSE, 2025

Ben Limpanukorn, **Jiyuan Wang**, Hong Jin Kang, Zitong Zhou, Miryung Kim, Fuzzing MLIR Compilers with Custom Mutation Synthesis, ICSE, 2025

Jiyuan Wang, Qian Zhang, Hongbo Rong, Guoqing Harry Xu, Miryung Kim, Leveraging Hardware Probes and Optimizations for Accelerating Fuzz Testing of Heterogeneous Applications, ESEC/FSE, 2023

Qian Zhang, **Jiyuan Wang**, Guoqing Harry Xu, Miryung Kim, HeteroGen: Transpiling C to Heterogeneous HLS Code with Automated Test Generation and Program Repair, ASPLOS, 2022

Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, Miryung Kim, QDiff: Differential Testing for Quantum Software Stacks, ASE, 2021, **SIGSOFT research highlight**

Qian Zhang, **Jiyuan Wang**, Miryung Kim, HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications, ESEC/FSE, 2021

Jiyuan Wang, Fuchen Ma, Yu Jiang, Poster: Fuzz Testing of Quantum Program, Poster, ICST, 2021, **Best Poster**

Qian Zhang, **Jiyuan Wang**, Muhammad Ali Gulzar, Rohan Padhye, Miryung Kim, Efficient Fuzz Testing for Apache Spark Using Framework Abstraction, Demonstrations, ICSE, 2021

Qian Zhang, **Jiyuan Wang**, Muhammad Ali Gulzar, Rohan Padhye, Miryung Kim, BigFuzz: Efficient Fuzz Testing for Data Analytics using Framework Abstraction, ASE, 2020

CHAPTER 1

Introduction

Specialized hardware accelerators like GPUs, FPGAs, and quantum computers have become a prominent part of the current computing landscape. As a result, an increasing number of applications and software stacks are constructed to leverage heterogeneous architectures. Developers can write *heterogeneous applications* that consist of *host* code and *kernel* code, where compute-intensive kernels can be offloaded from CPU to GPU, FPGA, or quantum computer. Similar to other software development scenarios, developers often deal with software underlying errors, such as divergent results produced by different hardware accelerators or crashes in compilers for heterogeneous computing. Heterogeneous computing systems provide increased computing ability and expressiveness through pragma and optimizations that specially designed for hardware accelerators. This consequently increases the complexity of testing and debugging heterogeneous applications and underlying software stacks.

Therefore, testing and debugging heterogeneous applications and software stacks can be an expensive and time-consuming process. For example, ensuring the correctness of FPGA programs, even seemingly-simple kernels, could take a substantial amount of time in terms of months [140]. Moreover, most software developers treat hardware accelerators as a black box, and failures in hardware accelerators like FPGAs occur silently without any raised exception. For example, when a divide-by-0 is generated in FPGA, there is no signal but only returned wrong values. In order to test software in the heterogeneous domain, we need to account for hardware execution behavior and leverage hardware accelerators to produce feedback.

Traditional software testing and debugging approaches do not work for heterogeneous applications, because they do not consider the hardware accelerators' characteristics. For example, it can takes AFL days and hours to trigger the bugs in heterogeneous applications [182, 13].

1.1 Thesis Statement

We synthesize ideas from software engineering and heterogeneous computing and answer the following research questions in the context of heterogeneous computing:

- What kinds of hardware accelerator characteristics can be leveraged to filter out hardware noise and accurately identify software bugs?
- How can we utilize hardware accelerators to generate execution feedback and accelerate test input generation?
- How can we systematically localize compilation layers and code segments responsible for bugs in heterogeneous computing?

Our hypothesis is that by leveraging hardware accelerator capabilities and accounting for execution behavior, we can develop efficient testing and debugging techniques that enhance the accessibility and reliability of heterogeneous computing.

To address the first research question, we introduce QDiff, a differential testing framework for quantum software stacks. QDiff enhances testing efficiency by filtering out noise circuits and leveraging quantum-specific properties such as qubit decay time (T1) and two-qubit error rates to optimize test case selection. Additionally, QDiff employs a Kolmogorov-Smirnov (K-S) based statistical comparison to detect quantum software bugs by analyzing deviations in measurement results.

For the second question, we design HFuzz, an automated fuzz testing framework for heterogeneous applications. HFuzz enhances test case generation by monitoring execution behavior across both software host code and hardware kernel code. Additionally, HFuzz offloads input mutation tasks directly to FPGAs, and accelerates mutation with FPGA-specific optimization, including loop unrolling, shannonization, and data preloading.

To further advance debugging for heterogeneous compilation, we focus on MLIR (Multi-Level Intermediate Representation), a flexible compiler infrastructure designed to unify and optimize compilation across diverse hardware backends. MLIR introduces a structured, extensible IR that enables progressive lowering from high-level domain-specific abstractions to hardware-specific representations. It serves as the foundation for several heterogeneous and ML compilation frameworks, including CIRCT, which provides specialized dialects for FPGA design. We introduce DuoReduce, a debugging methodology for heterogeneous compilation. Our third hypothesis is that by analyzing relationships between IR-level debugging and compilation-layer debugging, we can systematically identify which compilation layers contribute to a given bug. This approach enables more precise error localization within complex compilation pipelines, improving debugging efficiency and reliability.

1.2 Automated test generation by understanding hardware characteristics and leveraging hardware

The existing testing approach, including fuzz testing and differential testing, has been proved successful for traditional software. Google's OSS-Fuzz project alone has detected over 10,000 security vulnerabilities and 36,000 functional bugs across open-source projects since its inception [1]. Most fuzzing techniques, such as AFL [13], start from a seed input, generate new inputs by mutating the previous input, and add new inputs to the queue if they improve a given guidance metric, such as branch coverage.

However, applying these traditional testing approaches to specialized domains presents unique challenges, necessitating tailored solutions [185, 182, 28, 119]. Traditional software testing methods, including symbolic execution, fuzz testing, and differential testing, rely on assumptions that do not hold in heterogeneous computing. Symbolic execution, for example, struggles with the complexity of hardware-accelerated code due to its reliance on constrained symbolic reasoning, making it infeasible for analyzing optimized code running on GPUs, FPGAs, or quantum processors. Fuzz testing, which typically relies on random input mutation and branch coverage as a feedback metric, also faces fundamental limitations in heterogeneous environments. In traditional software, branch coverage is an effective measure of code exploration, but in hardware-accelerated computation, it often becomes meaningless. For example, in quantum computing, quantum circuits do not follow conventional branching logic—every operation in a quantum circuit executes simultaneously due to the inherent parallelism of quantum gates. This means that traditional branch coverage cannot provide useful feedback on test effectiveness, rendering conventional fuzzing techniques ineffective for exploring diverse execution behaviors in quantum and other heterogeneous systems.

Differential testing, which compares the outputs of multiple implementations to detect discrepancies, faces fundamental challenges in hardware-accelerated systems. Unlike traditional software, where execution is deterministic, heterogeneous computing involves non-deterministic factors such as hardware noise (e.g., quantum gate errors) and precision loss (e.g., floating-point rounding differences across CPUs and FPGAs). These factors introduce noise into differential outputs, making it difficult to distinguish true bugs from hardware-induced variations.

To address the above challenges, we investigate the following hypothesis:

Sub-hypothesis 1: By understanding the quantum hardware characteristics, we can build efficient differential testing for quantum software stacks. We redesign *differential testing* approach for quantum software stacks (QSSes) with three major innovations: (1) We generate input programs to be tested via semantics-preserving transformation to explore program variants. (2) We speed up differential testing by filtering out quantum circuits that are unnecessary to execute on quantum hardware by analyzing static characteristics such as the circuit depth, gate error rates, and T1 relaxation time. (3) We design an extensible equivalence-checking mechanism via distribution comparison functions such as Kolmogorov–Smirnov test.

We evaluate QDiff with three widely-used open-source QSSes: Qiskit from IBM, Cirq from Google, and Pyquil from Rigetti. Our hypothesis is that by accounting for unique characteristics of quantum circuit execution and measurement, QDiff can improve the efficiency of automated test input generation and the effectiveness of differential testing. By running QDiff on both real hardware and quantum simulators, we found several critical bugs revealing potential instabilities in these platforms. QDiff's source transformation is effective in producing semantically equivalent yet not-identical circuits (i.e., 34% of trials), and its filtering mechanism can speed up differential testing by 66%.

Understanding hardware accelerators is the first step toward enhancing the effectiveness of traditional testing approaches. Our work on QDiff demonstrated that leveraging domain-specific hardware characteristics—such as quantum noise, circuit depth, and measurement fidelity—can significantly improve the efficiency of test input generation and bug detection in quantum software stacks. However, the challenges posed by hardware-accelerated computing extend beyond quantum processors to other specialized architectures, including FPGAs, GPUs, and TPUs.

Similar to quantum circuits, FPGA-based computations follow a fundamentally different execution model from traditional CPUs. FPGA kernels operate as dataflow-driven accelerators, where execution is dictated by hardware-encoded pipelines rather than sequential software instructions. This hardware specialization introduces two major challenges for traditional software testing methods. First, limited observability: unlike conventional programs, FPGA kernels lack internal execution visibility, making it difficult to detect silent failures, hangs, or incorrect outputs. Second, inefficient test generation: fuzzing for FPGA applications is constrained by slow host-device communication and the highly inefficient mutation that modifies input data at the bit level.

We extended our focus to FPGA-based heterogeneous application testing, investigating the following hypothesis:

Sub-hypothesis 2: By understanding and leveraging the FPGA accelerators, we can build efficient fuzz testing for heterogeneous applications. Testing heterogeneous applications is extremely challenging: FPGA kernels are black boxes, revealing no information about the kernels' internal execution to help diagnose when the kernels silently hang or produce unexpected results. We propose HFuzz to leverage hardware probes and acceleration for testing heterogeneous applications. The essence of HFuzz is to introduce observability to FPGA kernels by combining host-side software monitors and device-side in-kernel hardware probes. Furthermore, HFuzz speeds up iterative test generation with hardware acceleration by offloading input mutation from the host-side to the kernel-side.

Our hypothesis is that by designing hardware-level probes and offloading input mutations to hardware accelerators, HFuzz can improve the efficiency and effectiveness of fuzz testing. We evaluate HFuzz on seven real-world heterogeneous applications. HFuzz speeds up fuzz testing by 4.7× times with HW accelerated mutations. By incorporating HW probes in tandem with SW monitors, HFuzz finds 33 kernel defects on the seven benchmarks within 24 hours and reveals 25 unique and unexpected behavior symptoms that could not be found by SW-based monitoring.

1.3 A bug isolation approach by considering multi-layer, extensible heterogeneous compilation

Exposing a bug is not the last step. Our collaboration with Intel partners revealed that merely triggering a bug is not enough—developers need more context to debug effectively. Heterogeneous compilers are specialized compilers designed to support diverse hardware architectures, such as CPUs, GPUs, FPGAs, and quantum accelerators. Unlike traditional compilers, which follow a linear pipeline to generate machine code for a single instruction set, heterogeneous compilers perform multiple layers of transformation and optimization, progressively lowering high-level abstractions to hardware-specific representations [88, 63].

MLIR (Multi-Level Intermediate Representation) [23] plays a crucial role in enabling this complexity by providing a flexible and extensible framework for heterogeneous compilation. MLIR structures compilation as a progressive lowering process, where code is transformed across multiple abstraction levels, from high-level domain-specific dialects to low-level hardware-specific IRs. This approach allows heterogeneous compilers to modularize transformations and optimizations, making it easier to support new hardware architectures. Such examples include Triton [155], CIRCT [63], and ONNX-MLIR [96]. Take CIRCT as an example; it offers domain-specific dialects for FPGA development, enabling high-performance hardware synthesis through structured transformations.

However, while MLIR improves the modularity and scalability of heterogeneous compilers, it also introduces new debugging challenges. The layered compilation flow makes it difficult to trace how a high-level operation is transformed through various IR stages, leading to bugs that are hard to localize. When a failure occurs in the low-level code, developers often lack visibility into which transformation, optimization pass, or lowering step introduced the issue. Manually inspecting the entire compiler stack—spanning multiple dialects, transformations, and hardware-specific optimizations—is impractical and time-consuming. A systematic approach is needed to localize the root cause at both the code dimension and the compilation dimension.

While existing delta debugging techniques can be used to identify a minimum subset of IR code that reproduces a given bug symptom, their naive application to MLIR is time-consuming, because real-world MLIR compilers usually involve a large number of compilation passes and compiler developers must also identify a minimized set of relevant compilation passes simultaneously, in order to reduce the footprint of MLIR compiler code to be inspected for a bug fix.

To address this challenge, we investigate the following hypothesis:

Sub-hypothesis 3: By understanding the relation between the IR code and multi-layer heterogeneous compilers, we can build efficient debugging tool. We propose DuoReduce, a dual-dimensional reduction approach for MLIR bug localization. DuoReduce leverages three key ideas in tandem to design an efficient MLIR debugger. First, DuoReduce reduces the bug-irrelevant compilation passes by identifying ordering dependencies among different compilation passes. Second, DuoReduce uses MLIR-semantics aware transformations to expedite IR code reduction. Finally, DuoReduce leverages cross-dependence between the IR code dimension and the compilation pass dimension by accounting for which IR code segments are related to which compilation passes to reduce the unused passes.

Experiments with three large-scale MLIR compiler projects find that DuoReduce outperforms syntax-aware reducers such as Perses and Vulcan in terms of IR code reduction by 31.6% and 21.5%, respectively. If one uses these reducers by enumerating all possible compilation passes (on average 18 passes), it could take up to 145 hours. By identifying ordering dependencies among compilation passes, DuoReduce reduces this time to 9.5 minutes. By identifying which compilation passes are unused for compiling reduced IR code, DuoReduce reduces the number of passes by 14.6%. This translates to not needing to examine 281 lines of MLIR compiler code on average to fix the bugs. DuoReduce has the potential to significantly reduce debugging effort in multi-layer extensible compilers, which serves as an important basis for the current landscape of machine learning and hardware accelerators.

1.4 Outline

The structure of this dissertation is outlined as follows. Chapter 2 introduces several related basic concepts. In Chapters 3 and 4, we delve into two testing methods for heterogeneous computing: QDiff for quantum software stacks, and HFuzz for heterogeneous applications, Chapter 5 presents DuoReduce, the dual-dimensional bug isolation method for multi-layer compilers. Finally, we conclude and discuss future directions in Chapter 6.

CHAPTER 2

Background

There has been a growing interest in developing specializable hardware accelerators for domain-specific workloads for various performance and energy benefits [55, 59, 62]. As an example, FPGA can be easily customized to accelerate applications across a wide variety of domains [48, 60] at lower power and higher performance than general-purpose CPUs [49, 136, 79]. Major hardware vendors are offering or plan to offer packages that include both CPUs and FPGAs [67, 30]. Such hardware packages have also been made into all major clouds to accelerate various analytic and learning tasks.

FPGAs, for instance, can be easily customized to accelerate applications across diverse domains [48, 60], delivering higher performance and lower power consumption compared to general-purpose CPUs [49, 136, 79]. Recognizing these advantages, major hardware vendors have begun integrating CPUs and FPGAs into unified packages [67, 30]. These hybrid architectures are now widely available in all major cloud platforms to accelerate analytics and machine learning tasks.

In this section, I will introduce the traditional testing and debugging approach, and illustrate why they cannot work for heterogeneous domains.

2.1 Heterogeneous Computing: Context and Challenge

With the slowdown of Moore's Law—the observation that transistor density doubles approximately every two years—traditional CPU performance scaling has become increasingly difficult [129, 71]. There has been a growing interest in developing specializable hardware accelerators for domain-specific workloads for various performance and energy benefits [55, 59, 62]. As an example, FPGA can be easily customized to accelerate applications across a wide variety of domains [48, 60] at lower power and higher performance than general-purpose CPUs [49, 136, 79, 67, 30]

Heterogeneous architectures are now widely adopted across various computing domains:

- Machine Learning and AI: GPUs accelerate deep learning training and inference [97, 53].
- Scientific Computing: Supercomputers leverage GPUs and FPGAs for high-performance simulations, molecular dynamics, and climate modeling [70].
- Embedded & Edge Computing: FPGAs and custom ASICs (Application-Specific Integrated Circuits) are used in low-power, high-throughput applications such as autonomous vehicles, IoT devices, and robotics [122, 157].
- Quantum Computing: Quantum accelerators, such as IBM's Q System and Google's Sycamore, are emerging as promising alternatives for problems involving cryptography, optimization, and quantum chemistry [34, 135].

Despite their advantages, heterogeneous architectures pose several challenges, particularly in software testing and debugging. Traditional testing and debugging techniques struggle to handle the complexity of heterogeneous computing because hardware characteristics are not considered. For example, Symbolic testing, a widely used program analysis technique that systematically explores all possible execution paths by treating inputs as symbolic variables instead of concrete values [99, 37] cannot work because it struggles to model parallel, event-driven, and hardware-level execution effectively.

2.2 Quantum Computing & Quantum Software Stack

Quantum Computing. Quantum computing emerges as a promising technology for many domains where quantum computers have been demonstrated to outperform classical computers by a large margin. For example, Grover's algorithm [77] can find a given item in an unsorted database with a $\sqrt{N}$-times speedup when running on a quantum computer, compared to a classical computer. Similar to programs executed on heterogeneous devices such as FPGAs, a quantum application is comprised of *host code* that runs on CPU and *quantum code* that runs on quantum hardware. Generally, quantum computers work as accelerators to execute the compute-intensive parts of the original application. For example, the Shor algorithm [145] can achieve an exponential speedup by decomposing integer factorization into a reduction to be executed on a classical computer and an order finding problem to be executed on quantum hardware.

Quantum Software Stack. A quantum software stack (QSS) includes (1) APIs and language constructs to express quantum algorithms, (2) a compiler that transforms and optimizes a given input quantum algorithm at the circuit level, and (3) a backend executor that either simulates the resulting gates on classical devices or executes directly on quantum hardware. Currently, three most widely used QSSes are Qiskit, Cirq, and Pyquil [104].

- **Qiskit [31]** is an open-source framework developed by IBM. Qiskit provides a software stack that is easy to use quantum computers and facilitates quantum computing research. Qiskit consists of `Qiskit Terra` (compiler), `Qiskit Aer` (several quantum simulators), `Qiskit Ignis` (which supports error correction and noise characterization), and `Qiskit Aqua` (APIs to help developers write applications).

- **Cirq [6]** is an open-source Python framework from Google. It enables a developer to create and simulate Noisy Intermediate-Scale Quantum (NISQ) circuits. Cirq consists of an `optimization` component to compile and transform circuits, a `simulator` component for emulating quantum computation on classical devices, and other `development` libraries for application development.

- **Pyquil** [58] is an open-source Python framework developed by Rigetti. It builds on Quil, an open quantum instruction language for near-term quantum computers, and uses a combined classical/quantum memory model. PyQuil is the main component of Forest, the overarching platform for Rigetti's quantum software. Pyquil consists of: `quilc` (compiler), `qvm` (quantum virtual machine with several kinds of simulators), and `pyquil` (a library to help users write and run quantum applications).

All three QSSes are similar to one another in that each includes a quantum programming language, an optimizing compiler that outputs quantum gate instructions, a quantum simulator that emulates these instructions on a classical device, and a software controller that executes gate instructions on quantum hardware.

As with any compiler framework, a QSS could be error-prone. Developers and users often report bugs on popular QSSes [7, 8, 31], and a simple search on StackOverflow with the keyword "quantum error" would bring up over 500 posts on various QSS components, ranging from compiler settings, simulation, and the actual hardware [14]. These posts often reveal deeper confusion that developers face due to the inherent probabilistic nature of quantum measurements—*if a program produces a result that looks different from what is expected, is it due to a bug or the non-determinism inherent in quantum programs? Is there divergence beyond expected noise coming from an input program, a compiler, a simulator, and/or hardware?*

2.3 MLIR in Heterogeneous Computing

LLVM [74] and the Multi-Level Intermediate Representation (MLIR) [64] framework are gaining significant traction across a wide array of compilers for machine learning and heterogeneous computing. They have revolutionized modern compiler architectures with layered and extensible compiler development, by enabling custom extension of IRs. For example, Triton [155], a cutting-edge language and compiler for machine learning developed by OpenAI, leverages the MLIR framework to translate Python kernels into Triton's IR representation. This IR is then optimized through various transformation passes before being lowered to PTX assembly, which is then fed to the CUDA compiler. The increasing adoption of MLIR compiler projects such as Triton underscores the importance of *extensible* compiler design, particularly in the domain of machine learning and hardware accelerators, where the underlying IRs and key optimization strategies are rapidly evolving.

The CIRCT project [63] is an effort to apply MLIR and the LLVM development methodology to the domain of hardware design tools. They aim to have reusable infrastructure that is modular, uses library-based design techniques, is more consistent, and builds on the best practices in compiler infrastructure and compiler design techniques.

The multi-layer nature of MLIR-based compilers, where high-level representations are progressively lowered through multiple IR transformations, brings significant challenges to testing and debugging. Existing fuzzing techniques such as MLIRSmith [160], HirGen [121], CLSmith [113], NNSmith [116], Neuri [117], and Tzer [118] have made strides in compiler testing and validation.

2.4 Automated Testing Techniques for Heterogeneous Application

In software engineering, symbolic execution is a program analysis technique that systematically explores program paths by treating inputs as symbolic variables [37]. For example, KLEE [46] is an open-source symbolic execution engine designed for automatically generating test cases and detecting bugs in programs written in C and C++. GKLEE [109] extends symbolic execution to CUDA programs, enabling the analysis of GPU kernels for correctness and performance issues. SmarTest leverages symbolic execution for smart contract testing [147]. However, symbolic execution's applicability to heterogeneous computing is limited due to differences in execution models, memory architectures, and programming paradigms. Alternative approaches, such as fuzz testing, have gained more traction for validating software on GPUs, FPGAs, and quantum processors.

Traditional fuzzing starts from a seed input, runs the program on the selected input, generates new inputs by mutating the previous input, and adds new inputs to the queue if they improve a given guidance metric such as branch coverage. Instead of using coverage as guidance, several techniques use custom guidance mechanisms. UAFL [161] incorporates typestate properties and information flow analysis to detect the use-after-free vulnerabilities. BigFuzz [181] monitors dataflow operator coverage in tandem with branch coverage for dataflow-based analytics. For example, MemLock [166] employs both coverage and memory consumption metrics. AFLgo [41] extends AFL to direct fuzzing toward user-specified target sites. SiliFuzz [142] finds CPU defects by fuzzing software proxies, like CPU simulators or disassemblers, and then executing the accumulated test inputs (known as the corpus) on actual CPUs on a large scale. PerfFuzz [107] uses the execution counts of exercised instructions together with branch coverage to identify inputs revealing pathological performance. HeteroFuzz [182] generates concrete test inputs for heterogeneous applications to perform differential testing between CPU vs. CPU+FPGA. However, Although these techniques work effectively in their domains, none of them leverage hardware to speed up their approach with parallelism. In other words, they overlook the opportunities for hardware optimizations, as the mutations often consist of independent tasks that can be parallelized efficiently when offloaded to the hardware accelerator.

A fuzzing loop consists of multiple invocations of a target program with different inputs in an independent manner; thus, it provides a natural opportunity for parallelism. AFL++ [73] injects a fork server, which tells the target to fork itself to run, and thus realizes parallel fuzzing across multiple CPU cores or across a fleet of systems. For example, P-Fuzz [148] distributes unique seeds to run fuzzing in parallel, and PAFL [112] maintains global and local guiding information for synchronizing parallel fuzzing jobs. While these techniques accelerate fuzz testing through distributed computation on CPUs, their underlying principles can be adapted to hardware accelerators like FPGAs to design a more efficient and scalable fuzzing approach.

Coverage-guided greybox fuzzing adds test cases into the set of seeds if they exercise the new path or new behavior. However, most seeds exercise the same "high-frequency" paths. To explore more paths with the same number of tests, researchers develop strategies to select seeds wisely. AFLFast [42] models coverage-based greybox fuzzing as a Markov chain, and assigns different selection probabilities for different seeds. EcoFuzz [174] improves AFLFast's Markov chain model and presents a variant of the Adversarial Multi-Armed Bandit model. EcoFuzz sets three states of the seeds set and develops a unique adaptive scheduling algorithm. However, these traditional coverage-feedback techniques do not translate well to heterogeneous computing. Unlike conventional software, heterogeneous kernels execute at the hardware level, where all branches are typically exercised, providing no meaningful code coverage signal. Furthermore, kernel code running on accelerators (e.g., GPUs, FPGAs) is often treated as a black box, offering no direct execution feedback that fuzzers can use for guidance. To enable effective fuzzing for heterogeneous computing, a new approach is required—one that extracts meaningful execution feedback from kernel code and provides guidance for exploring diverse execution behaviors beyond simple control-flow coverage.

2.5 Testing and Verification for Quantum

Zhao [184] introduces a quantum software life cycle and lists the challenges and opportunities we face. Long et al. [120] introduce quantum-specific testing principles and criteria for quantum program testing. Ying et al. [173] formally reason about quantum circuits by representing qubits and gates using matrix-valued Boolean expressions, and verify them using a combination of classical logical reasoning and complex matrix operations. Huang et al. [86] introduce quantum program assertions, allowing programmers to decide if a quantum state matches its expected value. They define a logic to provides ϵ-robustness to characterize the possible distance between an ideal program and an erroneous one. Proq [111] is a runtime assertion framework for testing and debugging quantum programs. It transforms hardware constraints to executable versions for measurement-restricted quantum computers. QPMC [72] applies classical model checking on quantum programs based on Quantum Markov Chain. Ali et al. [32] propose a new testing metric called quantum input-output coverage, a test generation strategy, and two new test oracles for testing quantum programs. Two test oracles include *wrong output oracle*, which checks whether a wrong output has been returned, and *output probability oracle* which checks whether the quantum program returns an expected output with its corresponding expected probability. However, their work targets at quantum program testing, and the measurement they used might not be sufficient.

Verified quantum compilers guarantee gate transformation and circuit optimization is correct by construction. CertiQ [144] is a verified Qiskit compiler by introducing a calculus of quantum circuit equivalence to check the correctness of compiler transformation. VOQC [82] provides a verified optimizer for quantum circuits by adapting CompCert [108] to the quantum setting. Smith and Thornton [146] present a compiler with built-in translation validation via QMDD equivalence checking. However, verifying large-scale quantum circuits remains a major challenge. However, verifying large-scale quantum compilers remains a major challenge. As quantum circuits grow in size, their unitary representations expand exponentially, making equivalence checking computationally infeasible [186]. Without scalable verification techniques, ensuring correctness for real-world, large-scale quantum compilers and applications remains an open problem.

Given the scalability limitations of traditional verification techniques for quantum compilers, differential testing emerges as a superior alternative due to its ability to validate compiler correctness without requiring full formal verification or exponential equivalence checking [124, 151]. It has been used to test large software systems and to find bugs in various domains such as SSL/TLS [45, 134], machine learning applications [78], JVM [54], and clones [183], etc. Equivalence modulo inputs (EMI) [105] is such an example that tests compilers by generating equivalent variants. Many random program generators are used for compiler testing [52]. Csmith [172] randomly generates C programs and checks for inconsistent behaviors via differential testing. Quest [115] focuses on argument passing and value returning, while testing with randomly generated programs. Different from Csimth-like tools, refactoring-based testing systematically modifies input programs with refactorings, as opposed to random program generation. Orion [105] adapts EMI to test GCC and LLVM compilers. Christopher et al. [113] combine random differential testing and EMI-based testing to test OpenCL compilers. Orison [106] uses a guided mutation strategy for the same purpose. Mucerts [51] applies differential testing to check the correctness of certificate validation in SSL/TLS. It uses a stochastic sampling algorithm to drive its input generation while tracking the program coverage. DLFuzz [78] does fuzz testing of Deep Learning systems to expose incorrect behaviors. Chen et al. [54] perform differential testing of JVM with input generated from Markov Chain Monte Carlo sampling with domain-specific mutations with the knowledge of Java class file formats.

Such classical compiler testing is not directly applicable to quantum software stacks due to the three challenges: (1) how to generate variants, (2) how to test simulators and hardware together with compilers, and (3) how to interpret quantum measurements for differential testing.

Based on my work QDiff [164], MorphQ [133] leverages equivalent quantum gate transformations to detect bugs in quantum compilers such as Qiskit [29], demonstrating the effectiveness of differential testing in identifying miscompilations in quantum software stacks.

2.6 Automated Debugging in Heterogeneous Domains

To ease the development of heterogeneous applications, HLS tools automatically generate RTL descriptions from C/C++ programs. To help debugging HLS-generated circuits, *Inspect* [47] introduces software debugger-like capabilities, including gdb-like breakpoints, step, and data inspection. It tracks file names and line numbers in HLS code, so that HW probes at the level of wires and registers could be linked to specific lines in the HLS code. A user can monitor each variable for its data width and the number of elements in an array. Monson and Hutchings [128] design a debugger for HLS-generated FPGA-based circuits via source instrumentation by connecting C expressions to top-level ports that serve as debug signals. HLScope [56] is a performance debugger that traces the cause of stalls for HLS-generated circuits. Curreri et al. realize in-circuit assertions for timing analysis and stall-relate bugs [66].

In traditional software engineering, taint analysis and program slicing are widely used in automated debugging [130, 156, 40, 168]. For example, Wang et al. [165] used taint analysis to localize bugs in configuration options. Soremekun et al. [149] did an empirical study on how program slicing can be applied on locating real-world faults. Badihi et al. [36] developed a program slicing based debugger for java. While program slicing and taint analysis are effective in traditional debugging, they fail in multi-layer heterogeneous compilation debugging, whereas delta debugging succeeds. Taint tracking assumes a direct mapping between input and output variables, but heterogeneous compiler optimizations break this assumption. Constant propagation, register allocation, and strengthening reduction may directly rewrite expressions with hardware resource specification. Program slicing works best when explicit control-flow dependencies exist. However, many compiler passes eliminate, reorder, or replace control structures, making it impossible to extract an accurate slice. Delta debugging, however, instead of following control/data dependencies, simply removes pieces of the input program until the bug disappears.

Delta debugging is a seminal work for input reduction to identify the minimal failure-inducing changes between two program versions [178, 176]. Based on *ddmin*, Ghassan et al. developed Hierarchical Delta Debugging (HDD) [127]. It applies delta debugging at each level of a program's input, working from the coarsest to the finest levels. Recent work proposed several enhancements on HDD [83, 84, 100]. ProbDD [159] is a probabilistic DD algorithm that learns from testing history to select elements based on probabilities. RCC [153] uses ZIP and SHA to compress the generated variants to speed up program reduction. Perses [150] ensures that each reduction step considers only syntactically valid variants to avoid futile effort on syntactically invalid variants and was later extended to specific domains [180, 154]. None of these reducers can handle bug isolation across two dimensions: code and compilation pass. Existing DD is inefficient for today's extensible multi-layer compilation and can take up to 145 hours to isolate culprit compilation passes along with the minimized IR program.

Many delta debugging tools consider the underlying language feature and apply various program transformations to decompose an input program into fine-granular units [69]. Vulcan [171] applies general code program transformations including identifier and subtree replacement. LPR [179] combines LLMs and language-generic reduction tools to refine the results of program reduction. C-Reduce [137] tackles this problem by leveraging domain-specific program transformations to reduce C/C++ programs. J-Reduce [98] and ddSMT [132] serve as the domain-specific reducers for Java and SMT-LIBv2. CHISEL [81] uses reinforcement learning in the C programming to select steps in *ddmin* that are more likely to satisfy the target oracle.

MLIR is widely adopted in heterogeneous compilation because it provides a modular IR infrastructure that spans multiple abstraction levels, from high-level computational graphs to low-level hardware instructions. However, debugging MLIR-based compilers poses unique challenges.

- Multi-Layer Transformations: MLIR enables a layered compilation flow where high-level representations progressively transform into low-level IRs, making it difficult to trace how errors propagate across different passes.

- Interaction Between Passes: Compiler bugs often emerge due to unintended interactions between optimization passes, making it essential to consider dependencies between transformations rather than debugging individual passes in isolation.

The MLIR and CIRCT projects also develop their own debugger utilities [63, 64]. They provide useful mechanisms for isolating faults but lack dependency-aware debugging approaches that account for the interactions between compilation passes and their effect on program correctness.

Given these challenges, a new debugging approach is needed—one that explicitly considers the dependency relationships between compilation passes and between compiled code and its transformations.

CHAPTER 3

QDiff: Redesign Differential Testing for Quantum Computing

As a first step toward building software support for heterogeneous systems, we focus on providing compiler testing support for quantum software stacks. A quantum software stack (QSS) includes (1) APIs and language constructs to express quantum algorithms, (2) a compiler that transforms and optimizes a given input quantum algorithm at the circuit level, and (3) a backend executor that either simulates the resulting gates on classical devices or executes directly on quantum hardware. Quantum computing, as an emerging field, currently lacks robust compiler and system testing frameworks. However, testing quantum software stacks poses significant challenges. The probabilistic nature of quantum computing results makes it difficult to determine whether unexpected outputs are due to the inherent quantum randomness or actual bugs in the software stack. Furthermore, the noisy nature of contemporary quantum hardware complicates the detection of software bugs from hardware-induced errors.

In this chapter, we address the following research question: *What kinds of hardware accelerator characteristics can be leveraged to filter out hardware noise and accurately identify software bugs?* To address this problem, we investigate the sub-hypothesis: *By understanding the quantum hardware characteristics, we can build efficient differential testing for quantum software stacks.* Building on the insights discussed in Section 3.4, we redesign differential testing frameworks to account for T1 time and 2-qubit gate number, which are the main sources of the quantum hardware noise, enabling more effective identification of software bugs while mitigating the impact of hardware noise.

3.1 Introduction

As with any heterogeneous compiler framework, a Quantum Software Stack could be error-prone. Developers and users often report bugs on popular QSSes [7, 8, 31]. A bug in a QSS can be just as severe as a miscompilation in a traditional compiler, potentially leading to incorrect quantum computations or silent failures. For example, In Qiskit, the `transpile` function is used to transform quantum circuits into a form that is compatible with a specific quantum backend. However, a bug was identified where the transpilation process fails when handling parameterized delay instructions [16].

We evaluated QDiff with the latest versions of three widely-used QSSes: Qiskit, Cirq, and Pyquil. With six seed quantum algorithms, QDiff generates 730 variant algorithms through semantics-modifying mutations. Starting from the generated algorithms, it generates a total of 14799 program variants using semantics-preserving source transformations. This generation process took QDiff 14 hours. With the filtering mechanism, QDiff reduces its testing time by 66%. Using QDiff, we determined total 6 sources of instabilities. These include 4 software crash bugs in Pyquil and Cirq simulation, and 2 potential root causes that may explain 25 out of 29 cases of divergence beyond expected noise on IBM hardware.

3.2 Background

Quantum Bit. A *quantum bit*, or *qubit* for short, is the basic unit of quantum computation. Unlike a classical bit that is either 0 or 1, a qubit's state is a probabilistic function of $|0\rangle$ and $|1\rangle$, represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \text{where } |\alpha|^2 + |\beta|^2 = 1 \tag{3.1}$$

The state of a qubit is unknown until a *measurement* is completed, resulting in $|\alpha|^2$ to be 0 and $|\beta|^2$ to be 1. Naturally, the sum of the probabilities (i.e., the modulus squared of amplitudes) is 1: $|\alpha|^2 + |\beta|^2 = 1$.

Quantum Gate. Classical computers use logic gates to transform signals from input wires. For example, a *NOT* gate, also known as an inverter, takes a single signal as input and outputs the opposite value. The quantum gate analogous to *NOT* is an X gate, which transforms a qubit $\alpha|0\rangle + \beta|1\rangle$ to $\beta|0\rangle + \alpha|1\rangle$.

An X gate has the following matrix-based representation:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{3.2}$$

Using a vector to represent the quantum state $\alpha|0\rangle + \beta|1\rangle$, applying an X gate has the following effect:

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \begin{bmatrix} \alpha' \\ \beta' \end{bmatrix} \tag{3.3}$$

Other commonly-used quantum gates include H, T, $CNOT$, Z, CZ, S, *U1*, and *U3*; a full explanation of these gates is elsewhere [131].

Since all quantum gates can be represented as *matrices*, we can transform a sequence of gates to another logically equivalent sequence without altering its outcome, as long as *the multiplication of the matrices for gates in each sequence* produces the same result. As an example, a *SWAP* gate, a two-qubit gate that swaps the two qubits' states, is semantically equivalent to a sequence of three *CNOT* gates.

$$SWAP(q_1, q_2) = CNOT(q_1, q_2)\,CNOT(q_2, q_1)\,CNOT(q_1, q_2) \tag{3.4}$$

This observation forms the foundation of QDiff's program variant generation procedure, detailed in Section 3.4.

```python
def make_circuit() -> QuantumCircuit:
    qubit = QuantumRegister(2,"qc")
    bit = ClassicalRegister(2, "qm")
    prog = QuantumCircuit(qubit, bit)
    prog.h(qubit[0])
    prog.x(qubit[1])
    for i in range(2):
        prog.measure(qubit[i], bit[i])
    return prog
```

(a) A quantum circuit with an H gate and an X gate.

```python
    prog = make_circuit()
    backend = BasicAer.get_backend('qasm_simulator')
    info = execute(prog, backend=backend, shots=1000).result().get_counts()
```

(b) The host code using the circuit in (a).

Figure 3.1: An example Qiskit program.

Quantum Circuit. A quantum circuit consists of a set of connected quantum gates. Since quantum gates can be represented as unitary matrices, a quantum circuit is essentially the multiplication of the matrices.

Figure 3.1a shows a program in IBM's Qiskit [31]. It first registers two qubits and initializes them to $|0\rangle$. Next, an H gate sets the first qubit into state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and an X gate flips the second qubit from $|0\rangle$ to $|1\rangle$. Finally, a measurement is performed and stored in a classical array. The function returns this circuit as a function-type value.

Figure 3.1b shows host code that calls this quantum circuit. Users can execute this circuit on different backends such as real quantum hardware or simulators. The circuit is executed on *qasm_simulator* for a thousand times (`shots=1000`). Since the first qubit state is $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and the second qubit state is $|1\rangle$, each run produces a result of either $|01\rangle$ or $|11\rangle$ with the equal probability of $0.5 = (\frac{1}{\sqrt{2}})^2$.

Table 3.1: Different layers that bugs appear

Layers	**Percentage**	**Example**
Compiler (optimizations & settings)	53.9%	`MergeInteractions()` returns different results for the same circuit [9].
Backend (simulators & hardware)	19.7%	Simulators change a global random state on Cirq [2].
API and quantum gate	11.8%	PhasedXPowGate raised to a symbol power fails the `is_parameterized` protocol [3].

T1 relaxation time. In quantum computing, a qubit can retain data for only a limited amount of time, referred to as *relaxation Time* because a qubit in a high-energy state (state $|1\rangle$) naturally decays to a low-energy state (state $|0\rangle$). The time span for this decay is referred to as *T1 Relaxation Time*. For a physical circuit, its measurement results are unreliable if its execution time is longer than *T1*.

3.3 Motivation

To understand real-world QSS bugs, we collected 76 latest issues reported on Github Pyquil, Cirq, and Qiskit. After excluding 11 issues related to installation and other tools, we categorized the remaining 65 issues by the layer where each issue appears: compilers, backends, and APIs.

Table 3.1 summarizes the percentage of the corresponding layer and a representative example. Most bugs appear at the compiler level—compiler optimizations and settings. For example, Cirq produces incorrect circuits when using `MergeInteractions()` [9]. The second most common bugs appear at the backend level—simulators and hardware execution, *e.g.*, the initial state is unexpectedly modified by the Cirq simulator in [2]. The other issues are with respect to the API implementation of high-level gates. For example, in

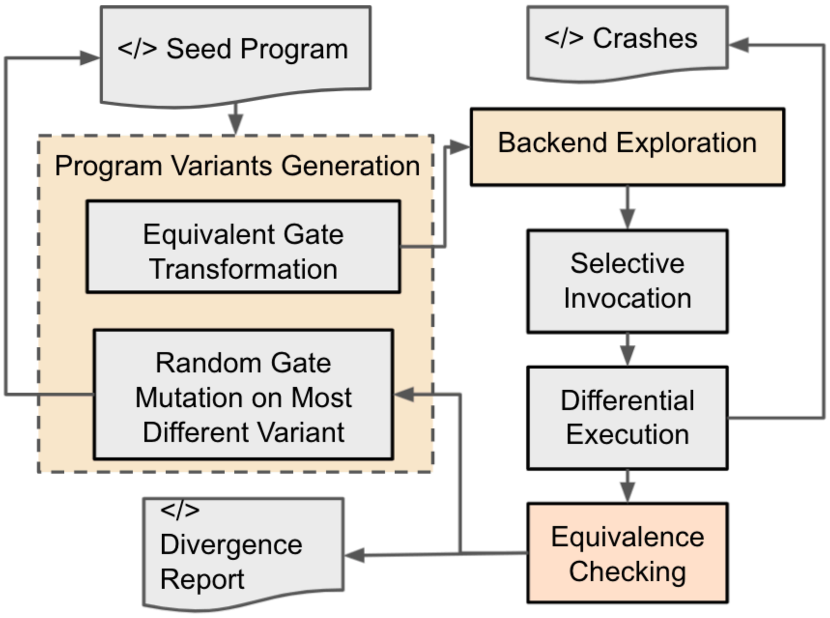


Figure 3.2: QDiff overview

Cirq, when `cirq.PhasedXPowGate` is invoked with an input argument `exponent`, invoking `is_parameterized` on the resulting gate should return `true` but returns `false` due to a bug in `cirq.PhasedXPowGate`.

The aforementioned bugs are hard to find due to quantum indeterminacy. The following excerpt illustrates a concrete example of how one google tutorial user is confused whether a problem is a bug or due to inherent non-determinism. Cirq's VQE (Variational-Quantum-Eigensolver) tutorial had a mistake—the orders of mix layer and cost layer were described in a wrong order. The tutorial user `D` (shortened) posted a question on StackExchange [4] with the embedded code from Cirq's VQE tutorial. A Cirq developer `C` (shortened) noticed that "The strange thing is the example output shows the output probabilities varying with *gamma* (the CZ parameter)", where *gamma* should not have any effect on the measurement results." `D` actually believed the disagreement is due to quantum indeterminacy, until `C` explicitly labeled it as a compiler bug.

Table 3.2: Explored Compiler Configurations

Framework	**Options**	**Description**
Qiskit	`BasicSwap,` make the circuit `LookaheadSwap,` `StochasticSwap`	Specify how swaps should be inserted to compatible with the coupling map. QDiff checks if BasicSwap has the most *SWAP* gates.
	`Optimization_level=0,1,2,3`	Specify the optimization level—the higher the level, the simpler the resulting circuit. QDiff checks if a higher-level optimization generates a more complex circuit.
Cirq	`DropEmptyMoment()`	Remove empty moments from a circuit. QDiff checks empty moments in the circuit.
	`MergeInteractions()` the applicability of G6 in Table 3.3.	Merge adjacent gates; QDiff checks
	`PointOptimizationSummary()`	User-defined optimization.
Pyquil	`PRAGMA INITIAL_REWIRING {"NAIVE" ,"GREEDY", "PARITIAL"}`	Change the optimization/mapping style
	`PRAGMA {COMMUTING_BLOCKS ,PERSERVE_BLOCKS}`	Change the optimization style for certain part of the code.

Key Takeway. This study of Github issues shows that QSS bugs are not just traditional compiler bugs and may come from various sources at different layers: API implementation of high-level gates, backend simulators, or hardware execution. This problem of detecting QSS bugs is further complicated by the probabilistic and noisy nature of quantum indeterminacy.

3.4 Approachh

QDiff contains three novel components to detect meaningful instabilities in QSSes. Figure 3.2 shows *program variant generation* using equivalent gate transformation and mutations

(Section 3.4.1); *backend exploration* leveraging selective invocation of quantum simulators and hardware (Section 3.4.2); and *equivalence checking* via distribution comparison (Section 3.4.3). Starting from a seed program, QDiff iterates these steps until a time limit is reached. Our key insight is that (1) we can generate semantically equivalent but syntactically different circuits, and (2) we can speed-up the differential execution process by filtering out certain circuits, because they will definitely lead to unreliable divergence and thus are *not worthwhile to run* on hardware or noisy simulators.

3.4.1 Program Variant Generation

Prior work [105] finds that testing compilers with equivalent programs is highly effective. QDiff adapts this idea to the domain of quantum compiler testing by creating logically equivalent gate sequences. It then checks whether the corresponding equivalent circuits produce the same results (in this case, a similar statistical distribution with some noise) on quantum simulators or hardware. For this purpose, QDiff generates program variants by repeating the two-fold process of *applying semantics-preserving gate transformation* to each generated program in each iteration and *applying semantics-modifying mutations* to diversify the pool of input programs in the next iteration.

Equivalent Gate Transformation (EGT). As discussed earlier in Section 3.2, one quantum gate sequence is semantically equivalent to another sequence, if they both yield the same unitary-matrix representation. In fact, complex quantum gates, without altering the outcome, can be described as a combination of basic quantum gates. QDiff leverages seven gate transformation rules to map complex gates into sequences of simple gates, as shown in Table 3.3. In G7, a Toffoli gate (*i.e.* $CCNOT$) can be replaced with 6 $CNOT$ gates and 9 one-qubit gates. To explore the alternative representation of a given gate sequence, QDiff finds the applicable transformation rules by matching the gate names. For an example circuit $S(q2)Z(q1)$, QDiff identifies the complex gate $Z(q1)$, applies rule G2 to construct gate

Table 3.3: QDiff generates program variants based on EGT rules G1-G7.

Rule ID	Original Construct	Equivalent Construct

G1	SWAP(q_1,q_2)	CNOT(q_1,q_2)
		CNOT(q_2,q_1)
		CNOT(q_1,q_2)

G2	H(q_1)H(q_1)	Merged to Identity Matrix
G3	X(q_1)	H(q_1)S(q_1)S(q_1)H(q_1)
G4	Z(q_1)	S(q_1)S(q_1)
G5	CZ(q_1,q_2)	H(q_2)CNOT(q_1.q_2)H(q_2)
G6	CZ(q_1,q_2)CZ(q_1,q_2)	Merged to Identity Matrix
G7	CCNOT(q_1,q_2,q_3)	6 CNOT gates with 9 one-qubit gates

$Z(q1)$ as a sequence $S(q1)S(q1)$, and generates a final variant $S(q2)S(q1)S(q1)$. As opposed to an optimizing compiler that applies transformation rules to reduce the number gates or the total gate depth [82, 31], QDiff aims to diversify the pool of input programs through source to source transformation.

Seed Diversification via Mutations. While generation of logically equivalent programs can find bugs through observing disagreements, they are unlikely to exercise various programming constructs. To diversify the seed input programs and to explore hard-to-reach corner cases in quantum compilers. QDiff borrows the idea of mutation-based fuzz testing [13] and designs a set of semantics-modifying mutations.

After generating multiple logically equivalent programs in each iteration, QDiff calculates the average distributions among the equivalent programs as the reference distribution, then picks the variant that leads to the largest comparison distance (e.g., K-S distance) from the reference distribution, and randomly applies one of the following four mutation operations to the variant in order to generate a new algorithm. This is based on the insight

Table 3.4: IBM quantum hardware's gate-level error rates, gate time, and T1 relaxation (decoherence) time.

IBM quantum computer	**T1 time** $/\mu s$	**Gate time** $/ns$	**2-qubit gate error rate**	t_m	δ_{2qubit}
ibm_santiago	155.19	408.89	1.79%	379	5
ibm_yorktown	56.81	476.44	2.03%	119	5
ibm_16_melbourne	53.37	928.71	3.29%	57	3
ibm_belem	74.45	552.89	1.07%	135	11
ibem_quito	85.35	353.78	1.03%	241	11

that the program with the most deviating results has a higher chance to expose unseen behavior [105, 113]. Please note that these mutations do not preserve semantics; instead, the goal of mutations is to resume the next round of differential testing with a different algorithm. We start with four mutation operators listed below, used in quantum mutant generation [125, 32].

- **Gate Insertion/ Deletion (M1)**inserts/deletes random quantum gates: *e.g.*, insert `prog.x(qubit[1])`;
- **Gate Change (M2)** changes a quantum gate to another gate: *e.g.*, from `prog.x(qubit[1])` to `prog.h(qubit[1])`;
- **Gate Swap (M3)** swaps two quantum gates;
- **Qubit Change (M4)** changes the qubits: *e.g.*, from `prog.x(qubit[1])` to `prog.x(qubit[2])`.

3.4.2 Quantum Simulation and Hardware Execution

Compiler Configuration Exploration. QDiff automatically explores different compiler configurations. In Qiskit, compiler settings can be specified by the arguments passed to

Table 3.5: Explored Backends (Simulators and Hardware)

Framework	Backends	Description
Qiskit	`statevector_simulator`	noiseless sim.
	`qasm_simulator`	noiseless sim.
	`FakeSantiago` `FakeYorktown` `FakeMelbourne`	noisy sim.
	`ibmq_santiago` `ibmq_yorktown` `ibmq_16_melbourne`	quantum hardware
Cirq	`Simulator`	noiseless/noisy sim.
	`DensityMatrixSimulator`	noiseless/noisy sim.
Pyquil	`Aspen-x-yQ-noisy-qvm`	noisy sim.
	`WavefunctionSimulator`	noiseless sim.
	`Aspen-x-yQ-qvm,Pyqvm`	noiseless sim.

the backends e.g., users can apply `optimization level=1` to collapse adjacent gates via light-weight optimization, while `optimization level=3` does heavy-weight optimization to resynthesize two-qubit blocks in the circuit. In Cirq, compiler settings must be specified using API invocations: e.g., users can write their own optimization with `PointOptimizationSummary()`. In Pyquil, compiler settings are specified using inlined pragmas similar to how FPGA developers specify high level synthesis options using pre-processor directives, e.g., a region denoted by `PRAGMA PRESERVE_BLOCK` will not be modified by a compiler. There are in total 2, 3, and 2 configuration types for Qiskit, Cirq, and Pyquil respectively, as shown in Table 3.2. When executing a variant with a specific compiler setting, QDiff records both thrown exceptions and program timeouts.

Backend Exploration. Backend exploration runs the same input program on different backends, shown in Table 3.5. In terms of real hardware execution, QDiff uses the free version of IBM hardware only, because other platforms are currently proprietary. QDiff is extensible by specifying a different backend configuration. Noisy simulators and state-vector simulators are both included in QDiff's backend exploration.

Filtering and Selective Invocation on Hardware. QDiff is focused on isolating software defects, not hardware defects. Because hardware imperfections such as decoherence is present, it is important to filter out circuits that would invoke errors due to the inherent hardware limitations. With the above observations, QDiff filters out unnecessary circuits in two steps. First, QDiff examines the final gate sequences after all compiler optimizations and logical-to-physical mappings, filtering out exactly identical physical circuits by moment-by-moment comparison. Second, QDiff analyzes the static characteristics of circuits to remove those that certainly produce unreliable executions (i.e. results dominated by hardware-level noise such as gate errors and relaxation errors and hence unreliable), while leaving those that may produce *meaningful divergences* using Definition IV.1.

As discussed in Section 3.2, for a physical circuit, its measurement results are unreliable if its execution time is longer than T1, implying that the number of circuit moments n_m (the depth of the circuit) in any circuit should not exceed a threshold t_m. Moreover, different kinds of quantum gates have different inherent error rates. For publicly available IBM quantum computers, error rates of single-qubit operations are in the order of 10^{-3}, while error rates of 2-qubit gate operations are in the order of 10^{-2}. A typical quantum program contains a significant number of 2-qubit gates, whose errors contribute the most to the overall error rate because such gates are more error-prone than 1-qubit gates [152]. Taking this into consideration, d_{2qubit} (the difference in the number of 2-qubit gates from the original circuit) should not exceed an application-specific threshold δ_{2qubit} to avoid unreliable results. Leveraging the above observations, we define the *worthiness* of invoking a quantum circuit.

Definition 3.4.1 *A circuit is worth invoking on quantum hardware or noisy simulator, if it satisfies the following condition:* $n_m < t_m$ *and* $d_{2qubit} < \delta_{2qubit}$.

Table 3.6: Cumulative probability of KS test

Measurement	**State**		**Cumulative**	**State**	
Distribution	**'0'**	**'1'**	**Probability**	**'0'**	**'1'**
A_1	464	546	EDF_{A1}	0.464	1
A_2	500	500	EDF_{A2}	0.500	1

The threshold t_m is determined empirically by two factors: (1) IBM computers' average $T1$ time, and (2) the average *gate* execution time for all gates, as listed in Table 3.4. QDiff computes t_m by dividing $T1$ by the average *gate* execution time. It then filters out those whose n_m is greater than t_m. Take `ibm_16_melbourne` as an example: with $T1 = 53.57\mu s$ and the average gate execution time = $928ns$, t_m is 53570/928=57. A circuit whose total number of moments is above 57 is filtered out for `ibm_16_melbourne`.

The threshold δ_{2qubit} is determined in the following way. Suppose a user is willing to tolerate an addition error rate of t for the entire quantum program's final measurements (with 0.1 as the default). Using t, we compute δ_{2qubit} as the maximum number of 2-qubit gates to be added or deleted from the number of 2-qubit gates in the original circuit. Suppose that CNOT's error rate for this IBM computer is 1.07%. If CNOT is used d_{2qubit} times in a row additionally, its updated error rate would be by $1-(1-e)(1-0.0107)^{d_{2qubit}}$, where e is the original error rate. Since both e and $(1-0.0107)^{d_{2qubit}}$ are relatively small, we can regard the error rate change as $1-(1-0.0107)^{d_{2qubit}}$. Therefore $|d_{2qubit} - \delta_{2qubit}|$ should be less than $log_{(1-0.0107)}(1-t)$. δ_{2qubit} is 11 when t is 0.1. The above thresholds and filtering condition in Definition 3.4.1 are customizable according to hardware's published error rates, supported gate types, and T1 relaxation time.

3.4.3 Equivalence Checking via Distribution Comparison

Nondeterministic nature of quantum programs makes it difficult for equivalence checking. Developers usually reason about the output of a quantum circuit by executing it multiple times to obtain a distribution. While numerous distribution comparison methods are well studied in statistics, one consequent yet over-looked question for quantum computing is that *how many measurements do we need for a reliable evaluation to ensure the relative error between two distributions is within a given threshold t with confidence p?* We design a novel equivalence checking component, which consists of: (1) a particular distribution comparison method C, and (2) an estimation of the required number of measurements for C, given a threshold t and confidence p. QDiff is equipped with *K-S test* and *Cross Entropy*, but is also extensible to other comparison methods by providing a new comparison-specific measurement estimation.

K-S Test [102] has been used to check the equality of distributions by measuring the largest vertical distance between empirical distribution functions (EDFs) in two steps. First, it creates the EDF for a given distribution by calculating the cumulative probability of different outcome states with respect to the total number of samples. In Table 3.6, A_1 and A_2 are two state distributions for 1000 samples. A_1 has 464 samples in state '0' while 546 samples in state '1'. Thus, the consequent EDF_{A1} indicates that the cumulative probability of A_1 samples in state '0' and state '1' are 0.464 (464/1000) and 1 ((464+546)/1000) respectively. Next, K-S test calculates the largest distance D of EDFs for each state, and uses such distance to quantify the difference of the original distributions under comparison. In Table 3.6, the largest distance is 0.06 ($|0.464 - 0.500|$) for state '0'.

QDiff evaluates this K-S distance with a user-defined threshold t. If the K-S distance of two results is less than t, QDiff regards them as similar results. QDiff provides a statistical guarantee on this comparison by estimating the required number of samples. For two distributions $d1$ and $d2$ over m outcome states, prior work [50] theoretically ensures that, with $n = \Omega(m^{1/2} \cdot t^{-2})$ samples, a sample-optimal tester can check if the relative error of $L1$ distance is within a threshold t when using a default confidence level of p=2/3. QDiff estimates n using Equation 3.5. This estimated number is directly applicable to QDiff

```
backend = BasicAer.get_backend('qasm_simulator')
info = execute(prog, backend=backend, shots=1024).result().get_counts()
```

```
Result: {'0': 475, '1': 549}
```

(a) Quantum simulator.

```
backend = BasicAer.get_backend('statevector_simulator')
info = execute(prog, backend=backend).result().get_statevector()
```

```
Result: [0.70710678+0.j, 0.70710678+0.j]
```

(b) State-Vector simulator.

Figure 3.3: A quantum circuit in Qiskit with two backends: quantum simulator and state-vector simulator.

because K-S distance is bounded by $L1$ distance [102]. In other words, Equation 3.5 calculates the number of measurements required, parameterized with respect to p. We empirically set p as 2/3, as it is a commonly used default in quantum volume measurement [27, 65], bioinformatics, and other statistics comparison.

$$n = A \cdot \frac{1}{\sqrt{1-p}} \cdot m^{1/2} \cdot t^{-2} \tag{3.5}$$

where A is a platform-related constant and m is the number of qubit states. In Figure 3.3, 2828 measurement samples are needed, when we empirically set $t = 0.1$, $p = 2/3$, and $A = 12$ for Qiskit. In our evaluation, we empirically measure the constant A for each platform by repeatedly running the same programs and compare the results.

Cross Entropy measures the difference between distributions via the total entropy. It represents the average number of bits needed to encode data coming from an underlying distribution q_1 when we use an estimated target distribution q_2.

$$H(q_1, q_2) = -\sum_{x=0}^{max} q_1(x) \log q_2(x) \tag{3.6}$$

Prior work ensures that, with $n = \Omega(m^{2/3} \cdot t^{-4/3})$ samples, the expected cross entropy of two similar distributions over m outcome states can be bounded with t [50, 143]. t is the difference from $H(q_1, q_1)$. Similar to K-S test, QDiff estimates the number of required samples to reliably satisfy this bound, as shown in Equation 3.7.

$$n = A \cdot \frac{1}{\sqrt{1-p}} \cdot m^{2/3} \cdot t^{-4/3} \tag{3.7}$$

For Figure 3.3, we need 420 measurements with $t = 0.1$, $p = 2/3$, and $A = 7$ for Qiskit. This measurement trials are different from K-S test, because we are using different distance metrics.

Comparison with Reference Distribution. After generating a group of equivalent programs and filtering out worthless circuits, QDiff executes the remaining circuits and calculates the average distribution from their results. QDiff will regard this average distribution as the *reference distribution*. With the distribution comparison methods (eg. K-S Test, Cross-entropy, etc), QDiff compares this reference distribution with each result distribution and reports divergence when the distance is larger than the threshold t.

Reporting Divergence Explanation. QDiff reports the potential source of the divergence in program P:

1. If P finds divergence when using different backends while keeping a frontend's options unchanged, QDiff reports this as a potential backend source;

2. If P finds divergence when using a specific backend while varying a frontend's options, QDiff reports this as a potential frontend source;

3. Otherwise, QDiff reports this as other sources, such as a potential bug in the API gate implementation.

Table 3.7: Seed subject programs

ID	Program	# of Qubits	Moments	2-qubit gate	Description	Iteration Number	Measurement trial with $t = 0.1$
P1	X gate	1	1	0	one-qubit X gate	46	2000∼2828
P2	Deutsch-Jozsa	4	39	33	check if a function is balanced	95	5293∼7998
P3	Bernstein-Vazira	4	41	32	find a and b for f(x) = ax+b	121	5293∼7998
P4	Grover	5	84	53	find a unique input in a database	129	8000∼11312
P5	VQE	4	36	28	approximate the lowest energy level	171	5293∼7998
P6	QAOA	5	29	19	QAOA algorithm	168	8000∼11312

3.5 Evaluation

We evaluate the following research questions:

RQ1 How many syntactically different programs can be generated by QDiff's mutation and equivalent gate transformation?

RQ2 How much speedup can we achieve via filtering and obviating the need of invoking a quantum simulator or hardware?

RQ3 What has QDiff found via differential testing of the widely-used QSSes?

Benchmarks. QDiff starts differential testing with five well known quantum algorithms as seed programs [77, 145, 39, 123, 68] (Deutsch-Jozsa, Berstein-Vazira, VQE–Variational-Quantum-Eigensolver, Grover, and QAOA–Quantum Approximate Optimization Algorithm) and one additional program X Gate listed in Table 3.7. We do not use the relatively large algorithms like Shor's[145] because IBM's public access can support up to 16 qubits only. Large algorithms also require many moments and 2-qubit gates, often producing unreliable results on quantum hardware.

Experimental Environment. We evaluate QDiff with three widely-used QSSes: Pyquil 2.19.0 with Quilc 1.19.0, Qiksit 0.21.0, and Cirq 0.9.0.

We run the circuits on five different hardware versions based on their availability, including `ibmq_santiago`, `ibmq_yorktown`, `ibmq_16_melbourne`, `ibmq_belem`. and `ibmq_quito`. The details of hardware can be found on IBM's quantum computing website [31]. We use K-S test with $t = 0.1$ as the distribution comparison method.

3.5.1 RQ1: Variant Generation via S2S transformation

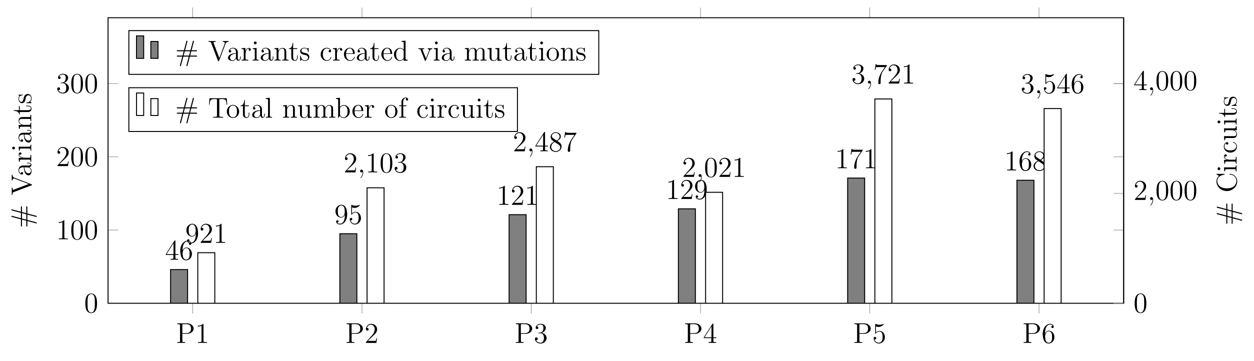
As shown in Figure 3.4a, for all six seed algorithms together, QDiff generates 730 program variants through semantics-modifying mutations and generates the total of 14799 circuits with equivalent gate transformation to each generated variant. This total circuit generation process takes around 14 hours.

Take P2 Deutsch-Jozsa algorithm as an example. 95 different program variants are generated through semantics-modifying mutations. For each variant program, QDiff generates 20 logically equivalent circuits. For P2, the total generation for 2103 circuits takes around 2 hours, while the rest of differential execution via simulation or hardware execution takes around 2 days. This implies that the bottleneck of testing is not about input program generation but the execution of the generated programs, which justifies our approach to select which circuits are worthwhile to run.

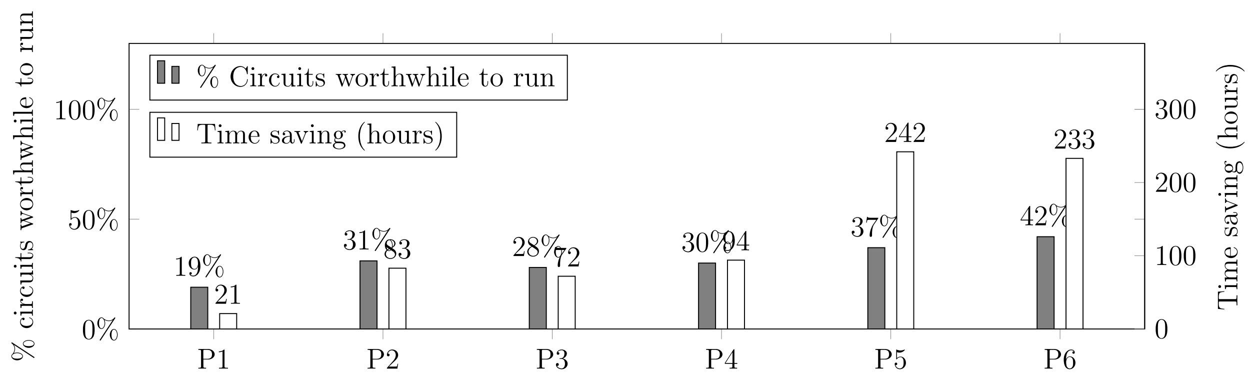
3.5.2 RQ2: Speed Up

As shown in Figure 3.4b, after filtering, only 19%-42% of the generated circuits are retained for differential execution on quantum hardware, leading to a 66% reductions in quantum hardware or noisy simulator invocations. QDiff finishes the entire testing process within around 17 days by leveraging its circuit selection process, which means QDiff would have saved an additional 30 days of testing time by filtering.

Take QAOA as an example, when running on IBM quantum hardware, the experiment would take around 7 minutes to wait in line on average (as launching a quantum job uses



(a) Total circuits and variants originating from each seed algorithm



(b) The percentage of circuits that are worth running

Figure 3.4: Statistics of QDiff-generated circuits.

a shared web service for a few IBM computers in the world) and 10 seconds to execute on hardware. If users run all 3546 circuits, the total clock time would be 17 days. With filtering, QDiff removes 68% circuits that are not worthwhile to run and finishes the execution in 7 days.

3.5.3 RQ3: What has QDiff found?

From the 730 sets of semantically equivalent circuits, QDiff found 33 differing outcomes out of 730. 4 out of 33 are crashes in simulators. The remaining 29 cases are divergence beyond expected noise on IBM hardware. By inspecting all 33 cases carefully, we determined total 6 sources of instabilities: 4 simulator crashes and 2 potential root causes that may explain

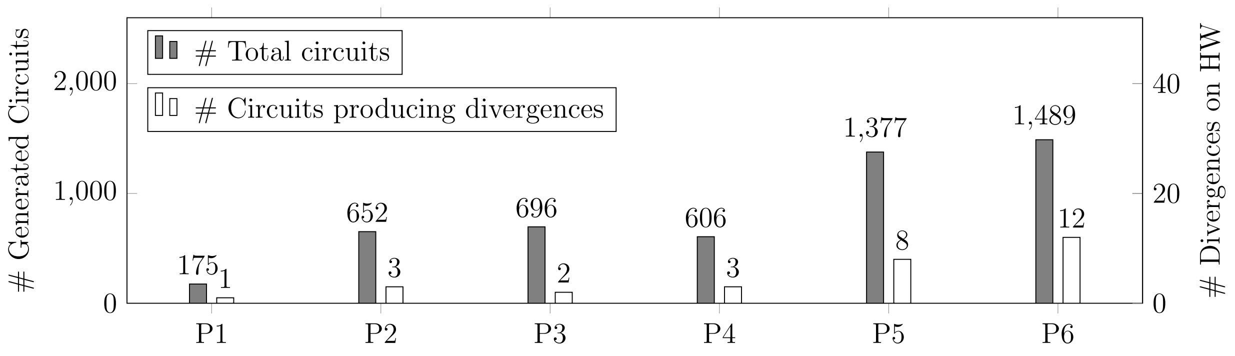


Figure 3.5: Circuits producing divergences on IBM hardware

Table 3.8: Bugs found by QDiff when executing generated programs with simulation only

Platform	**Bugs Description**	**Source of Bugs**
Cirq	Program runs endlessly with some compiler settings	Compiler Setting
pyquil	Simulator register wrong number of qubits	Simulator Backends
	Crashes on control gates on certain backends	Gate implementation
	Simulator stuck into a bad state	Simulator Backends

25 out of 29 cases of divergence on IBM hardware. For the remaining 4 divergence cases, we could not easily determine their underlying root causes.

3.5.3.1 Crash bugs in simulators

QDiff reports four crashes during differential testing with both noiseless and noisy simulators. All divergences involve clear failure signals. All are due to bugs in compiler or simulator implementations, summarized in Table 3.8. 2 out of 4 crashes were already reported by developers [5, 10] and 2 out of 4 crashes were confirmed by developers, when we filed crash reports [11, 12].

Compiler option error: Given an arbitrary program in Cirq, when the compiler option `clear_span` is set to a negative number or `clear_qubits` is set to an unregistered qubit, the execution does not terminate. Cirq does not check the boundary values of compiler options

```
1 qc = get_qc('Aspen-0-3Q-A-qvm')
2 result_rm = qc.run_and_measure(p)
```

(a) `run_and_measure` measures all 16 qubits in device `Aspen-0` when a simulator allocates 3 qubits, resulting in an exception.

```
p = Program(X(0), X(1).controlled(0))
qvm = PyQVM(n_qubits=2)
qvm.execute(p)
```

(b) Controlled X gate raises an error when 2 qubits are allocated for simulation.

Figure 3.6: Bugs in Pyquil found by QDiff.

and attempts to reset non-existing qubits. This bug was detected during explorations of compiler settings. When these compiler options are set to the aforementioned values, QDiff notices that the execution does not finish in a reasonable time, indicating a potential infinite loop. QDiff found this bug on the 74th iteration with P3.

Backend registers a wrong number of qubits: QDiff found that Pyquil's measurement crashed on `Aspen-0-xQ-A-qvm`. In Pyquil, users can specify the quantum simulator to have the same topology as a real 16 qubit device `Aspen-0-16Q-A` by setting the backend to be `Aspen-0-16Q-A-qvm` (`16Q` refers to using 16 qubits in simulation). However, QDiff found that when a user allocates to use 3 qubits in simulation (line 1 in Figure 3.6a) but attempts to conduct measurements at line 2 using `run_and_measure`, Pyquil simulator crashes due to a wrong number of allocated qubits. This bug was found with QDiff's backend exploration. Another user reported the same issue on Pyquil's Github [5].

Simulator is stuck into a bad state: Pyquil's simulator raises an exception when taking an empty circuit as input. Afterward, all subsequent invocations to the simulator crash even when the input circuit is valid and not empty. QDiff detects this bug by mutating an input program to an empty program and executing it on both `pyqvm` simulator and state-vector simulator. While the state-vector simulator runs this empty circuit normally, `pyqvm` throws an exception. Then, QDiff generates an arbitrary non-empty program. In the next few iterations, QDiff finds this bug because `pyqvm` crashes, while the state-vector simulator does not.

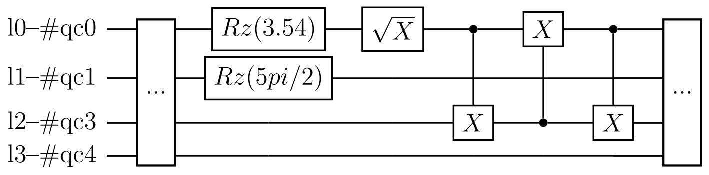
Wrong type of the controlled gate: Figure 3.6b shows a scenario where Pyquil crashed when a control X gate was used on `pyqvm` with 2 qubit allocation. The exception message shows *"ValueError: cannot reshape array of size 4 into shape (2,2,2,2)"*. A control X gate should be a 2-qubit gate (which is 4×4 matrix), but the gate was represented as 1×4 by `pyqvm`, which is a bug. This bug was found in the 15^{th} iteration with P1 as a seed.

3.5.3.2 Divergences on real hardware

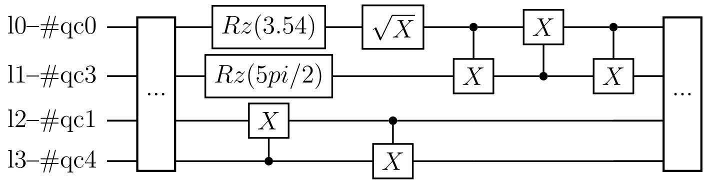
Figure 3.5 reports the total numbers of circuits executed on quantum hardware and those that exhibit behavioral divergence. The rate of latter is relatively low (0.3%-0.8%), which demonstrates the robustness of the quantum software and hardware we tested—this is not surprising since such software and hardware has been widely used and continuously improved for a number of years. On the other hand, these results also highlight the need of a systematic testing framework such as QDiff for quantum developers—since quantum bugs are rare and hard-to-detect, developers should test their programs/tools exhaustively with a QDiff-like approach before releasing them.

For 29 divergence cases on IBM hardware, we manually inspected the corresponding circuits. We then determined 2 root causes that may explain 25 out of 29 cases. For the remaining 4 cases, we could not easily determine underlying root causes. We discuss each root cause with examples in this subsection.

Divergence due to 2 qubit gate errors We concluded that 1, 2, 6, and 7 divergences in P1, P3, P5, and P6 respectively—55% of all divergences detected on hardware in total—can be explained by placing many 2 qubit gates between so called *couplers* qubits. IBM released the reliability of connection between each qubit pair on their hardware [31]. For example,



(a) "Divergent" circuits



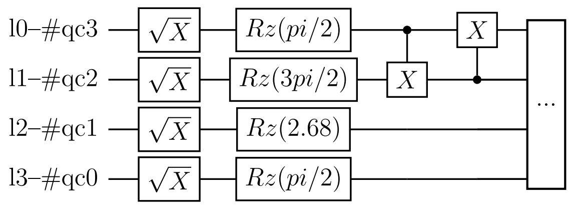
(b) "Good" circuits

Figure 3.7: Divergence on IBM `ibmq_yorktown`: no operation on qc4 for a long time.

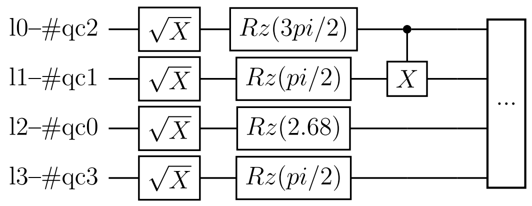
in hardware `santiago`, mapping CX between physical qubits {2, 3} is less reliable than mapping CX between qubits {1, 2} [31]. We found 16 out of 29 divergences could have the same underlying cause of using CX between known, unreliable qubit connections. These errors could be reduced through improved mapping to 2-qubit gates or using other strategies such as randomized compilation [158, 80].

Consider the two equivalent circuits shown in Figure 3.8, generated from P6 QAOA. These two circuits generated divergent measurements on `santiago`, although both circuits have nearly the same moments and the same number of 2-qubit gates. This is because the CX error rate of physical qubits 2 and 3 is much higher than qubits 1 and 2 (77% higher according to published information from IBM [31]). It appears that Qiskit's logical to physical qubit mapping procedure does not always avoid the use of CX on qubits 2 and 3 in their compilation and qubit allocation steps. Thus, Figure 3.8a produces divergence from Figure 3.8b.

Divergence due to qubit dephasing & decoherence: 9 of 29 divergences—2, 3, 2, and 2 divergences in P2, P4, P5, and P6—could have the same underlying cause of qubit



(a) Divergent circuits (CX between physical qubits 2 and 3)



(b) "Good" circuits (CX between physical qubits 1 and 2)

Figure 3.8: Divergence detected on IBM hardware: bad connection between qc_2 and qc_3.

dephasing & decoherence. Qubits that remain idle for long periods tend to dephase and decohere [15]. Figure 3.7 shows a pair of circuits with a similar depth and a similar number of 2-qubit gates. However, when run on hardware `ibm_belem`, the pair produces divergences beyond expected noise. We speculate that, when no operation is applied to physical qubit 4 for 18 moments, it may increase dephasing and decoherence possibilities. This can by fixed by adding two successive Pauli Y gates [131] on idle qubits during the compilation phase [15].

Others: For the other 4 divergences, we could not easily determine the underlying root causes. Because the circuit moments and the number of CX gates are roughly the same with their equivalent groups, stochastic errors in hardware can mostly be ruled out. The problem could be low-level quantum control software bugs that emerge from different combinations of gates, resulting in different control / coherent errors introduced at the pulse level.

3.5.4 Threats to Validity

Lack of Error Correction: The number of divergences on quantum hardware found by QDiff would depend on the reliability of hardware and its error correction capability. If it were to run on an error-corrected quantum hardware, which does not exist yet, it may report fewer divergences and it would be easier to disambiguate whether divergences are caused by software-level defects as opposed to hardware-level defects.

Similarly, 2-qubit gate errors depend on which qubit connections that the gates are applied to, Therefore, it may be necessary to adjust the divergence threshold t based on the empirical 2 qubit error rates for each connection and how many times the 2 qubit gates are used on that connection. Such impact of 2 qubit error rates must be investigated further.

Time Out: In fuzz testing, longer experimentation periods tend to expose more errors or new program execution paths [101]. The total time taken for all our experiments was limited to seven days.

Number of Qubits: The maximum number of qubits that we used for our experiment was 5 qubits, because the only publicly available hardware limits public access up to 16 qubits and the waiting time tends to increase significantly, as you request more qubits. Running experiments on Google's sycamore processor with 53 qubits and 1000+ 2q gates may produce different results [126].

3.6 Discussion

Quantum computing has emerged to be a promising computing paradigm with remarkable advantages over classical computing. QDiff is the first to reinvent differential testing for quantum software stacks. It adapts the notion of equivalence checking to the quantum domain, redesigns underlying program generation and mutation methods, and optimizes differential testing to reduce compute-intensive simulation or expensive hardware invocation. It is effective in generating variants, reduce 66% unnecessary quantum hardware or noisy simulator invocations, and uses divergence to isolate errors in both the higher and lower levels of the quantum software stack.

MorphQ [133] and QuteFuzz [95] have built upon the foundations laid by QDiff to enhance the reliability of quantum software stacks. MorphQ introduces a program generator that produces a diverse set of valid quantum programs while leveraging the same equivalent program transformation techniques as QDiff. QuteFuzz, on the other hand, focuses on generating random quantum programs with higher-level abstractions, such as subroutines and complex control flows (e.g., if-else, switch statements). While QuteFuzz operates at a different level of abstraction, it shares the same test generation strategy as QDiff, reinforcing the effectiveness of structured test generation in detecting quantum compilation issues.

Beyond testing quantum software stacks, I have also explored fuzz testing techniques specifically targeting quantum programs [162]. Potential future directions for debugging quantum software are discussed further in Section 6.2.

A key insight from QDiff is that a deep understanding of hardware enables more efficient testing approaches. But can we go beyond merely understanding the hardware? Instead of just optimizing for hardware constraints, can we actively leverage the hardware itself to accelerate the testing process? Given the current resource limitations and access challenges associated with quantum hardware, we shift our focus to FPGAs, which offer greater accessibility, flexibility, and opportunities for hardware-accelerated testing. In the next chapter, we introduce HFuzz, an approach that pushes this idea further—utilizing hardware acceleration to enhance the efficiency of software testing.

CHAPTER 4

HFuzz: Redesign Fuzz Testing for Heterogeneous Computing

Testing methodologies like QDiff have demonstrated that a deep understanding of hardware can significantly enhance software testing efficiency. By minimizing unnecessary hardware invocations and strategically leveraging quantum-specific properties such as qubit decay time (T1), error rates, and circuit depth, QDiff optimizes the testing process for quantum software stacks. However, while these optimizations reduce overhead and improve efficiency, they primarily focus on making better use of existing hardware constraints. A natural next step is to move beyond passive optimizations and actively harness hardware acceleration to improve the testing process itself.

Traditional software testing, particularly fuzz testing, often struggles with heterogeneous architectures due to the complex interplay between software and hardware execution models. Many existing fuzzing techniques rely on iterative input mutation, which can be computationally expensive and inefficient when applied to hardware-accelerated applications. Furthermore, testing heterogeneous applications involves both host-side execution (CPU) and kernel execution (accelerators like FPGAs and GPUs), creating a disconnect between software-driven fuzzing strategies and hardware execution behavior. To bridge this gap, we need a fuzzing approach that is hardware-aware and hardware-accelerated.

In this chapter, we address the following research question: *How can we utilize hardware accelerators to generate execution feedback and accelerate test input generation?* To address this problem, we investigate the sub-hypothesis: *By understanding and leveraging the FPGA* *accelerators, we can build efficient fuzz testing for heterogeneous applications.* We introduce HFuzz, a novel approach that leverages FPGA hardware not only as a testing target but also as a **computational resource** to accelerate the fuzzing process.

4.1 Introduction

Ensuring the correctness of heterogeneous applications, even seemingly simple kernels run on FPGA, could take a substantial amount of time in terms of months [140]. As such, FPGA programming can be done by only a small handful of hardware experts [103, 141, 35]. Automatic fuzz testing of heterogeneous applications, together with root cause analysis of failures, can greatly simplify FPGA programming, thereby making FPGAs accessible to the masses. However, testing these heterogeneous applications can remain a significant challenge due to the following reasons:

Lack of Observability. FPGA is a device of massive parallelism. Little debugging support exists to help high-level programmers. Kernels run on an FPGA device as black boxes, and it often confuses programmers, e.g., when the kernels silently deadlock.

Costly Transfer of Data with High Redundancy. Traditional iterative fuzzing techniques often mutate a small part of a seed input to generate new inputs. While this approach works well for many CPU programs, it is extremely ineffective for applications that are run on heterogeneous architectures. Figure 4.1 illustrates the latency breakdown of running applications on Intel's heterogeneous architecture. On average, data transfer from CPU to hardware kernels takes 60% of the execution time.

Overlooked Opportunities for FPGA-level Optimizations.
Fuzzing heterogeneous applications may be approached in a naïve manner by treating hardware kernel invocations as analogous to software function calls and repeatedly invoking them from an iterative input mutation loop. However, this approach ignores the potential optimizing capability of FPGA, as the mutations often consist of independent tasks that can be

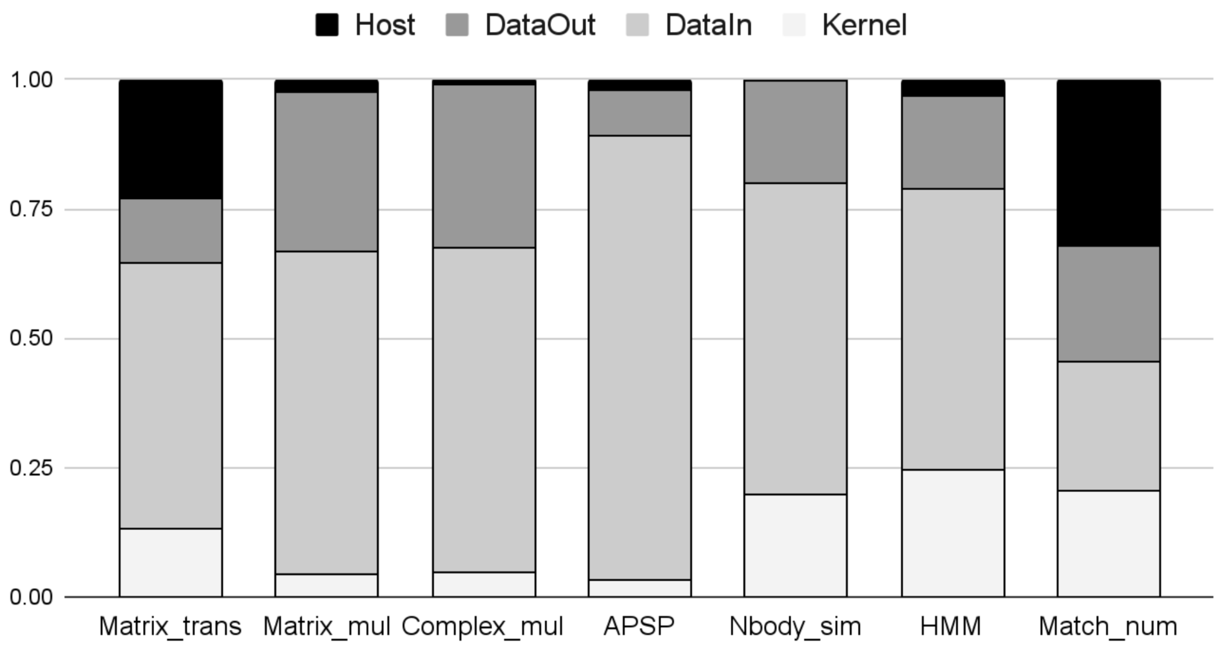


Figure 4.1: **Latency breakdown of running applications on heterogeneous architectures. On average, data transfer into kernels takes 60% of execution time, highlighted in gray.**

parallelized efficiently when offloaded to the FPGA side.

To address these challenges, we propose a novel fuzz testing technique, HFuzz, to enable efficient testing on real heterogeneous architectures. HFuzz aims to increase both the observability of hardware kernels and testing efficiency through a three-pronged approach. First, HFuzz automatically generates test guidance by inserting device-side in-kernel hardware probes in addition to host-side software monitors. Second, it performs rapid input space exploration by offloading compute-intensive input mutations to hardware kernels. Third, HFuzz parallelizes fuzzing and enables fast on-chip memory access, by utilizing four FPGA-level optimizations including loop unrolling, shannonization, data preloading, and dynamic kernel sharing.

We evaluate HFuzz on seven open-source OneAPI subjects from Intel. HFuzz speeds up fuzz testing by 4.7× with HW-accelerated input space exploration. By incorporating HW probes in tandem with SW monitors, HFuzz finds 33 defects within 4 hours and reveals 25 unique, unexpected behavior symptoms that could not be found by SW-based monitoring alone. HFuzz is the first to design hardware optimizations to accelerate fuzz testing.

4.2 Background

```
for(int s = 1; s <= nsteps; ++s) {
  ...
// Kernel: calculate velocity
   h.parallel_for(n, [=](item<1> i){
      acc0=0; acc1=0; acc2=0;
      #pragma unroll factor=2
      for(int j=0; j<n; j++) {
         if (j==i) {continue};
         int8 dx, dy, dz;
         dx = p[j].pos[0]-p[i].pos[0];
         dy = p[j].pos[1]-p[i].pos[1];
         dz = p[j].pos[2]-p[i].pos[2];
         int8 sqr=dx*dx+dy*dy+dz*dz;
         acc0+=(kG*p[j].mass/sqr)*dx; //calculate acceleration
         acc1+=(kG*p[j].mass/sqr)*dy;
         acc2+=(kG*p[j].mass/sqr)*dz;}
       p[i].vel[0]+=acc0*dt; //calculate velocity
       p[i].vel[1]+=acc1*dt; p[i].vel[2]+=acc2*dt;});
});
```

Figure 4.2: **Nbody-simulation: a heterogeneous version with DPC++ high-level synthesis.**

4.2.1 Heterogeneous Applications with FPGA

Driven by performance and energy benefits, heterogeneous computing applications [43] contain code that is executed on different kinds of processors such as CPU, GPU, and FPGA.

FPGAs are field programmable gate arrays. Modern FPGAs include millions of look-up tables (LUTs), thousands of embedded block memories (BRAMs), thousands of digital signal processing blocks (DSPs), and millions of flip-flop registers (FFs) [169]. Intel provides CPU+FPGA multi-chip packages; with its recent acquisition of Altera, such integration is expected to be even tighter in the future. FPGA has made its way into modern data centers, including Microsoft's Azure, Amazon F1, and Intel DevCloud [175, 33, 89].

A heterogeneous application typically consists of *host* code executed on the CPU and *kernel* code to be synthesized and executed on FPGA or GPU. Host code initializes the device, allocates the device memory, transfers data to the device, and invokes the compute-intensive kernel on the device side. After the execution, it transfers the kernel output back to the host and deallocates the memory.

To simplify kernel development, high-level-synthesis (HLS) [61, 75] lifts the abstraction of hardware development by automatically generating register-transfer level (RTL) descriptions from code written in C-like dialects. One example of HLS C/C++ dialects is Intel's Data Parallel C++ (DPC++), a cross-platform abstraction layer that enables code to be targeted to different CPUs, GPUs, and FPGAs [139, 138]. With DPC++, users can specify which hardware platform to implement a kernel on. For example, a user may use a compiler flag `-Xsboard=intel_s10sx_pac` to select Intel's FPGA S10. The user can develop a kernel function `f`, calling `h.parallel_for(n,f)` with a job handler `h`. This handler executes `f` with `n` degree parallelism on FPGA S10. Consider the example in section 4.2.2.

4.2.2 An Illustrating Example: Nbody-simulation

Figure 4.2 illustrates the simulation of n particles moving over a sequence of `nsteps`. Lines 10-12 calculate the distance between particles, while Lines 14-16 calculate the acceleration. In lines 17-19, the program subsequently updates the particles' velocities based on the acceleration. These computations are extracted as compute-intensive kernels and offloaded to an FPGA. To enable parallelism and speed up the velocity calculation, the developer uses `h.parallel_for` and loop unrolling `#pragma unroll factor=2` (highlighted in red) at Lines 4 and 6.

When writing a heterogeneous application, a user must conservatively estimate the limit of hardware resources and specify bitwidths for custom types and the size of buffers and pipes because all hardware resources are finite. Due to the need to finite hardware resources, a heterogeneous application often contains defects that cannot be detected statically via static analysis. This is a problem that universally exists with all HLS languages. To illustrate, consider the real defects in the Nbody-simulation.

Divide By Zero in Nbody-Simulation. For code in Figure 4.2, with the input `p.pos=[(1,2,4),...,(1,2,4)]`, the velocity calculation on an FPGA A10 device using Intel DevCloud produces absurdly large numbers `p.vel=[(-214748364,..),..]`. This is because, when the kernel inputs contain two particles with the same position, a divide-by-zero may happen inside the kernel in Lines 16-18 due to `sqr=0` at Line 15.

Overflow in Nbody-Simulation. When the kernel calculates the acceleration of two particles in Figure 4.2, an in-kernel overflow could occur if two particles are close to each other (i.e., `sqr≈0` at Line 15). This is because when `sqr` is close to zero, `acc` becomes large. When the inputs `p.pos=[(81,0,0),(81,1,0),(81,0,1),...]` are sent to the velocity calculation kernel, it produces a small value for `sqr=1`, leading to overflow for the variables `acc1`; finally, the wrong result is sent back to the host.

State-of-the-Art. Grey-box fuzzing [182] generates program inputs based on per-iteration execution feedback. Suppose that a user uses grey-box fuzzing to monitor the value range of the inputs and outputs of kernels on the host-side (CPU) code. For the divide-by-zero bug that could occur in Figure 4.2, because `sqr` is an in-kernel variable and does not appear in the host code, software-side grey-box fuzzing [182] cannot easily reveal defects that originate from the inside of the kernel.

HFuzz addresses the limitations of existing work by utilizing hardware probes to monitor the intermediate states of kernels. HFuzz identifies the in-kernel local variable `sqr` at Line 13 and inserts hardware probes to track its value range. The input generation process is then optimized by prioritizing inputs that result in new minimum or maximum values of `sqr`. As a result, HFuzz is able to effectively detect overflow when `sqr` reaches the small value `sqr=1` and divide-by-zero defects when `sqr` reaches its minimum value 0.

4.3 Approach

HFuzz aims to find inputs that can trigger both in-kernel errors and host-side errors for heterogeneous applications written in Intel's DPC++ HLS [88]. HFuzz contains three novel components that work in concert: (1) in tandem monitoring of software and hardware feedback by injecting software monitors and in-kernel probes (Section 4.3.1); (2) offloading input mutations to hardware kernels (Section 4.3.2), and (3) FPGA-level optimizations to speed up iterative input generation and kernel invocation (Section 4.3.3). HFuzz's design builds on two key insights. First, hardware-level parallelism can bring notable performance enhancement for iterative fuzzing, which is often characterized by independent task-level parallelism. Second, grey-box fuzzing's effectiveness can be significantly improved by observing feedback signals from both hardware and software.

The Fuzzing Process. The overall workflow of HFuzz is shown in Algorithm 1. HFuzz takes as input a program p written in Intel's DPC++ and produces concrete inputs that trigger defects in p. HFuzz first applies a source-to-source transformation to p to produce

Algorithm 1 Fuzzing Workflow

```
Require: Program $p$, Input Generator Set $S$, Mutation Operator Set $O$
 1: procedure FuzzingLoop($p$, $S$)
 2:     $p' \leftarrow$ instrument($p$)
 3:     $Feedback \leftarrow \emptyset$
 4:     for $i = 1$ to max do
 5:         $G \leftarrow S$.select_input_generator()
 6:         $in' \leftarrow$ random_select($G$)
 7:         $F_{HW}, F_{SW} \leftarrow p'.host$, in_kernel_mutate_execute($in', O$)
 8:         for all $F \in \{F_{HW} \cup F_{SW}\}$ do
 9:             if $F \notin Feedback$ then
10:                 increase_prob($S$, $G$)
11:                 $good\_input \leftarrow$ regenerate($F.m, in'$)
12:                 $G \leftarrow G \cup \{good\_input\}$
13:                 $Feedback \leftarrow Feedback \cup \{F\}$
14:             end if
15:         end for
16:     end for
17: end procedure
Require: Kernel Input $k_s$, Mutation Operator Set $O$
Ensure: $F_{HW}$: Queue of triples $(f, m, out)$ where $f$ is kernel feedback, $m$ is mutation, and $out$ is
    kernel output
18: procedure In_Kernel_Mutate_Execute($k_s$, $O$)
19:     $In_{queue} \leftarrow \emptyset$
20:     for $i = 1$ to MAX do
21:         $o \leftarrow$ select_op($O$)
22:         $s \leftarrow$ random_generate()
23:         $e \leftarrow$ random_generate()
24:         $m \leftarrow \{(o, s, e)\}$
25:         $In_{queue} \leftarrow In_{queue} \cup$ mutate_input($o, s, e, k_s$)
26:     end for
27:     $F_{HW} \leftarrow \emptyset$
28:     for all $in \in In_{queue}$ do
29:         $(f, m, out) \leftarrow$ ExecuteOnDevice($in$)
30:         $F_{HW} \leftarrow F_{HW} \cup (f, m, out)$
31:     end forreturn $F_{HW}$
32: end procedure
```

Table 4.1: Mutations accelerated by hardware.

Category	Description	SW Mutations	In-kernel Mutations	Average Speedup
M1 Sparsity Mutation	Replace non-zeros with zeros from index s to e, or do the opposite	`for i in s..e do {vector[i]=0}`	`# pragma unroll for i in s..e{vector[i]=0});`	4.31×
M2 Copy Mutation	Replace each element from index s to e with element at s	`for i in s..e do {vector[i]=vector[s]}`	`# pragma unroll for i in s..e{vector[i]=vector[s]});`	3.98×
M3 Addition Mutation	Add constant a to each element from index s to e	`for i in s..e do {vector[i]+=a}`	`# pragma unroll for i in s..e{vector[i]+=a});`	3.21×
M4 Bit Mutation	Mutate an element with binary XOR given a constant x	`for i in s..e do {vector[i]^= (1<<x)}`	`# pragma unroll for i in s..e {vector[i]^= (1<<x)});`	4.42×

an instrumented version $p\prime$, by inserting in-kernel probes and software monitors that can guide fuzz testing. HFuzz selects an input generator G from a set of generator S. It then randomly offloads a random seed input in' from G's seed queue into the kernels. To generate new inputs, HFuzz creates a new mutation kernel job in addition to the original kernel, and utilizes parallelism within FPGAs to mutate the input locally. The target function directly accesses the new input from local memory. In this process of input mutation and target execution, HFuzz incorporated four FPGA level optimizations for performance efficiency. As shown in Algorithm 1 at Lines 10-15, inputs that advance either software or hardware feedback are saved to the input queue as *good_input* for the next fuzzing iteration. If a new input generated by generator G results in new feedback, G will be considered a favored generator and its activation probability will be increased with $INCREASE\_PROB(S, G)$ at Line 11.

4.3.1 Injecting HW Probes in addition to SW Monitors

HFuzz, *for the first time*, directly introduces application-specific observability to hardware kernels by inserting hardware probes. It leverages these kernel probes in tandem with software-level monitors to form effective feedback signals to stretch heterogeneous application behavior.

```
//First kernel...
  h.parallel_for(range(M, P), [=](auto index) {
    int sum = 0;
    #pragma unroll factor=2
    for (int i = 0; i < num_element; i++) {
      sum += a[index[0]][i] * b[i][index[1]];
      if (min_sum>sum) min_sum=sum;
      if (max_sum<sum) max_sum=sum;}
    bool flag;
    KToKPipe::write(sum, flag);
    KToKPipeSize++;//Pipe usage Probe
    DeviceToHostKToKPipe::write(KToKPipeSize);
    DeviceToHostMax_sum::write(max_sum);//sum's Value Range Probe
    DeviceToHostMin_sum::write(min_sum);});});
//Second kernel...
  h.single_task([=]() {
  for (size_t i = 0; i < number_element; ++i) {
    out[i] = KToKPipe::read();
    KToKPipeSize--;//Pipe usage Probe
    DeviceToHostKToKPipe::write(KToKPipeSize);
    out[i] = reciprocalTransform(output[i]); }});});
for(int i=0; i<number_element; i++) {//SW monitor for kernel output
  outmin=min(outmin, output[i]);
  outmax=max(outmax, output[i]);}
```

Figure 4.3: Matrix transform: inserted Value Range Probes are in the green rectangle. InsertedPipe Usage Probes are in the red rectangles. Inserted SW Monitors are in the orange rectangle.

Hardware Probes. While OS virtualization could provide the appearance of unbounded resources for the code executed on traditional CPUs, kernel functions are physically mapped to *resource-limited* heterogeneous architectures. This distinction leads to unique failures that are often induced by *resource limitations* on the device-side, which are not easily detectable when running software simulators. For example in Figure 4.2, a local variable `sqr` customizes regular integers to 8-bit integers for resource efficiency. Overflow conditions can occur if the variable's value exceeds its customized bitwidth. As another example, pipe saturation between two consecutive kernel functions can lead to read and write failures. In fact, such incorrect *intermediate computation states* within hardware kernels have been identified as the primary reason for hardware-originated bugs. HFUZZ takes advantage of this observation, identifies local variables within kernels that hold intermediate states, and injects hardware probes to expose potential failures in kernel.

HFUZZ automates the process of hardware probe insertion through source to source transformation, creating an instrumented kernel. From such instrumented kernel, intermediate states in the HW device are sent directly to the host code using dedicated host-kernel communication channels. The channels are implemented as global FIFO buffers and can be accessed from both the host and the kernel. The kernel side writes hardware feedback into the channels, while the host side reads information from the channels. Both read and write operations are non-blocking, in order to minimize any additional overhead to the original kernel logic. To expose intermediate computation states, HFUZZ identifies in-kernel local variables and pipe usage via a C/C++ AST analysis [38]. As shown in Figure 4.3, in-kernel variable `sum` is highlighted in green, and pipe usage is highlighted in red. With a focus on in-kernel local variable and pipe monitoring, HFUZZ aims to uncover the two most commonly seen errors in custom hardware accelerators: overflows resulting from the resource and bitwidth finitization, as well as read/write failures caused by communication pipe saturations.

- *Value Range Probe*: HFUZZ creates a value range monitor that checks the maximum and minimum value for each in-kernel variable. In Figure 4.3, HFUZZ inserts probes on the intermediate variable `sum` which saves the cumulative sum of the product `a[index[0]][i]*b[i][index[1]]`. These probes monitor the minimum and maximum value of `sum`. HFuzz also constructs channels `DeviceToHostMax_sum` and `DeviceToHostMin_sum` to send these captured values back to the host at Line 13-14.

- *Pipe Usage Probe*: HFuzz creates a pipe usage monitor for each communication pipe. Consider the same example in Figure 4.3. HFuzz uses an AST analysis tool [38] to identify the locations of two kernel functions: `matrix_multiply` at Line 1-14 and `transformer` Line 16-21. We identify the variable name, `KToKPipe` used for pipe-based data transfer between the two kernels. By using `KToKPipe::write()` and `KToKPipe::read()`, the first kernel writes its result `sum` at Line 10 and the second kernel reads the value from this pipe at Line 18 in Figure 4.3. HFuzz applies source to source transformation to inject a counter-based usage monitor for this pipe and update the counter `KToKPipeSize` at Line 11 and Line 19 in Figure 4.3. Then HFuzz sends this counter value to the host by creating another direct communication channel, called `DeviceToHostKToKPipe` at Line 12 and Line 20.

Software Monitors. In addition to in-kernel probes, HFuzz inserts a set of software monitors on the host side, specialized to the custom FPGA accelerator synthesized on the device. We monitor: (1) the number of loop iterations, because it is related to pipelining and loop unrolling, common optimizations for parallelization implementation on FPGA; (2) the value range of each kernel input and output; (3) the kernel execution time, as hang or unexpectedly slow execution could be an indicator of failures. HFuzz retrieves the time and loop unrolling information from the HLS compilation report generated by DPC++. Besides, to monitor the value range of each kernel input and output, HFuzz inserts a value range monitor before and after each kernel, as shown in Lines 22-24 of Figure 4.3.

4.3.2 Offloading Input Mutations to Kernels

The traditional fuzzing process involves repeatedly mutating seed inputs and feeding them into a target program. The implicit assumption underlying such mutations is that seed inputs can be mutated and sent to the target program fast. Unfortunately, this assumption does not hold true for heterogeneous applications. Inputs to heterogeneous applications are often large matrices, leading to significant data transfer overheads between CPU and FPGA. We observe that *local* data transfer—data transfer within FPGAs, consumes less than 89% of the time required for data transfer between the fuzzer and the kernel. Additionally, in the process of fuzzing, a variety of *independent* mutation operations are frequently employed on small segments of the same seeds with the aim of exploring the input space. Thus, we can avoid repetitive data transfer by offloading the seed inputs to hardware kernels and mutating them directly within FPGAs. To achieve this, HFuzz creates a dedicated kernel for mutations *in parallel to* the original kernel, as well as a segment of on-chip memory for the storage of seeds and newly generated inputs. The mutation kernel and the original kernel function are both synthesized to the FPGA hardware concurrently. Table 4.1 shows four supported mutation operators. Because mutation operators are all order-independent and deterministic, HFuzz modifies all elements in the seed input at once. A resulting input can be re-generated given the seed and a concrete instance of mutation.

Consider Figure 4.3 as an example. The first kernel code computes the matrix product with two input matrices. We show how HFuzz tracks the feedback and mutates the input step by step in Table 4.2. With the initial seed input offloaded to the kernel, HFuzz tracks hardware feedback from the in-kernel variable `sum` at Line 2 by the inserted in-kernel probes in the green rectangle (column *Hardware Probes* in Table 4.2). After we apply the *M3 Addition Mutation* with loop unrolling optimization, from the starting offset `s=1` to the ending offset `e=4` on array `a`, a greybox fuzzer that only monitors the value range for the kernel interface variables `a` and `b` would discard the input `[-20,5,7,7,9,20]` because it does not achieve a new value spectra at the software level. However, HFuzz saves the corresponding mutation information, since this input registers a new feedback at the hardware level for the in-kernel variable `sum`.

Table 4.2: Example execution of input generator G.

	Mutation	Kernel		Hardware Probes		Software Monitors		New Value	Over-	Save	Memorization		
ID	Operator	Inputs	Variable	Min	Max	Min	Max	Range	flow	Input	HW Range	SW Range	P_G
Seed	N/A		sum	-56	168			N/A	No	N/A	[-56,168]		0.25
		a[][1]=[-20, 2, 4, 4, 6, 20]	a			-20	20	N/A				[-20, 20]	
		b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	N/A				[-14, 28]	
1	**M3**		sum	-202	54			**Yes**	No	**Yes**	**[-202, 168]**		0.3
	start s=1	a[][1]=[-20, 5, 7, 7, 9, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	No				[-14, 28]	
2	**M2**		sum	-70	140			No	No	No	[-202, 168]		0.25
	start s=1	a[][1]=[-20, 5, 5, 5, 5, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	No				[-14, 28]	
3	**M3**		sum	20	-140			**No**	**Yes**	**Yes**	**[-202, 168]**		0.3
	start s=1	a[][1]=[-20, 8, 10, 10, 12, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-11	28	No				[-14, 28]	

4.3.3 FPGA Optimizations for Fuzzing

Traditional fuzz testing can be naïvely applied to heterogeneous applications by treating hardware kernel invocations as equivalent to software function calls. However, such straightforward application of software-style fuzzing results in severe performance inefficiencies. In heterogeneous applications, there is a distinct *opportunity* to utilize hardware micro-architecture level optimizations to accelerate the traditional fuzzing process. Both iterative matrix mutations and target executions involve independent tasks, enabling task-level parallelism.

HFuzz applies four FPGA optimizations to accelerate iterative matrix mutations and target execution, including loop unrolling, shannonization, local memory access, and dynamic kernel sharing. These optimizations are not specific to HFuzz or Intel's heterogeneous architecture, and thus also are applicable to other applications on other FPGAs. For instance, loop unrolling is a technique that can be used to optimize iterative computations that do not have significant data dependencies between iterations, and it can be applied independently of the specific FPGA platform.

1. Dynamic Kernel Sharing. In traditional fuzzing, the difficulty of testing often arises from the need to explore deep branches within the program. However, when testing het-

```
for (int i = s; i < e; i++) {
  if (A[i]==0) {A[i] = generate_number(seed);}}
```

(a) **Original mutation**

```
int local_A[e-s];
#pragma unroll factor=4
for (int i = 0; i < e-s; i++) {local_A[i] = A[i+s];}
int t = generate_number(seed);
for (int i = 0; i < e-s; i++) {
  if (local_A[i]==0) {
  local_A[i] = t;
  t = generate_number(seed);}}
#pragma unroll factor=4
for (int i = 0; i < e-s; i++) {A[i+s] = local_A[i];}
```

(b) **Optimized mutation in kernel**

Figure 4.4: Sparsity mutation: replace the zero elements to non-zero elements from index `s` to index `e`.

erogeneous applications, errors tend to occur due to variations in the range of values for in-kernel variables and resource usage. This presents a significant challenge of rapid input space exploration especially when inputs are large matrices.

We propose a dynamic, probabilistic kernel-sharing method to interleave the exploration of input search space originating from multiple seeds in heterogeneous applications. To implement this method, HFuzz employs four input generators that share the same target kernel and each has its own seed queue. These input generators start with different seed inputs and, during each iteration, one generator is chosen based on an activation probability array. The selected generator then picks a seed input from its queue, mutates it within the kernel, and sends the generated input to the target kernel function via on-chip memory on the device. If the generated input results in new feedback, it is saved in the generator's seed queue for use in future fuzzing iterations.

HFuzz utilizes an adaptive approach to input generation by selecting an input generator and its associated seed queue based on an activation probability array. The selection process involves evaluating the performance of each generator and adjusting its probabilities accordingly. For instance, if a new input generated by generator G results in new feedback, it will be considered a favored generator and its activation probability will be increased. Otherwise, it will be labeled as an inactive generator and its activation probability will be decreased. This approach allows for efficient input space exploration and ensures that the test generation is focused on areas that are likely to yield new feedback:

$$P_G = \begin{cases} P_G + \alpha & \text{if } G \text{ is chosen and HFuzz} \\ & \text{gets new feedback} \\ P_G - \frac{\alpha}{l-1} & \text{if } G \text{ is not chosen and HFuzz} \\ & \text{gets new feedback} \\ P_G - \alpha & \text{if } G \text{ is chosen and HFuzz} \\ & \text{gets no new feedback} \\ P_G + \frac{\alpha}{l-1} & \text{if } G \text{ is not chosen and HFuzz} \\ & \text{gets no new feedback} \end{cases} \tag{4.1}$$

In our experiment, we set the number of generators l to be 4. The initial activation probability for each generator P_G is set to $1/l = 0.25$. The update factor α is predefined as 0.05. In Table 4.2, in the second execution (ID 2), inputs generated by generator G increased the hardware monitor range. As a result, HFuzz increases the activation probability of G from 0.25 to $0.25+\alpha = 0.3$.

2. Data Preloading [91]. Matrix mutation on large matrices requires a significant amount of data read and write operations. To improve efficiency, it is crucial to minimize memory access time for input vectors or matrices. Many heterogeneous computing systems, such as Intel oneAPI, have both *global memory* that can be accessed by both kernel and host code, and on-chip *local memory* that is only accessible by kernel code. Accessing local memory within the kernel typically has a shorter latency than accessing global memory. We thus apply data preloading to transfer data from global memory to local memory.

In Figure 4.4b, HFuzz reduces memory access costs (highlighted in red) by transferring data from array `A` to the local array `local_A`. This results in a reduction of memory access cost as seen at Lines 6-7 in the optimized code, compared to the original code in Figure 4.4a at Line 2. This optimization leads to a 1.31x speedup in the mutation process.

3. Shannonization [90]. Sparsity mutation replaces zero elements with non-zero elements. It necessitates the implementation of a null check for each element in the matrix. As shown in Line 2 of Figure 4.4a, an `if` statement is added to accomplish this. However, this `if` statement induces extra hardware overhead, as it increases the delay in the critical path. Each time the `if` condition is satisfied (i.e. `A[i]==0`), the operation `generate_number` needs to be computed, which can slow down the overall performance.

Shannonization improves performance by precomputing operations within a loop and removing them from the critical path. In this example, HFuzz applies shannonization (highlighted in green in Figure 4.4b) by precomputing the operation `generate_number` at Line 4, and removing it from the critical path inside the branch at Line 6. Then HFuzz precomputes the next value of `t = generate_number` at Line 8 for a later iteration of the loop to use when required (that is, the next time `local_A[i]==0`). This precomputation can be done simultaneously within the loop, allowing for a reduction in the critical path delay and leading to a 1.24x speedup in the sparsity mutation process.

4. Loop Unrolling [92]. Software-style mutations on large vectors and matrices are often performed by modifying one or some particular elements. Line 2 in Figure 4.4a shows an example mutation based on a `for` loop. Such direct application of loops on hardware neglects the potential for hardware parallelism, resulting in inefficient use of hardware resources.

Loop unrolling improves performance by creating multiple copies of the loop body, thus

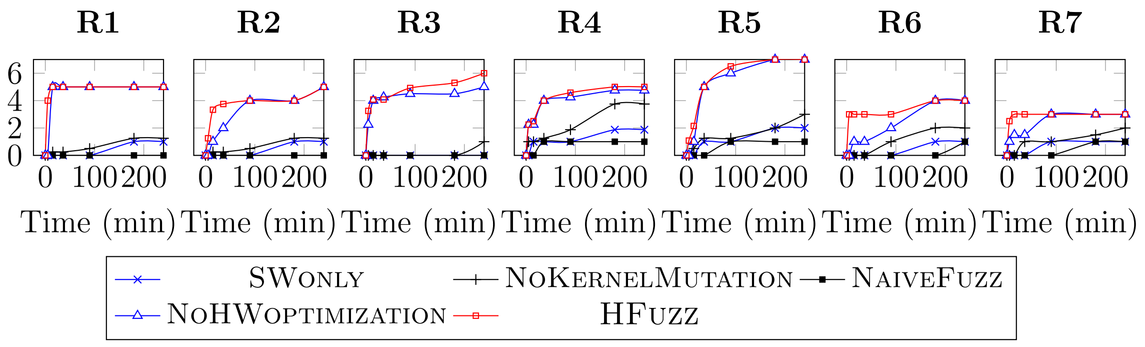


Figure 4.5: Number of Defects

the required number of iterations is reduced. In the example shown in Figure 4.4b, the `#pragma unroll` directive (highlighted in orange) causes the kernel to unroll the loop by a factor of 4, as specified by the `factor=4` argument. The compiler then expands the pipeline by quadrupling the number of operations and loading three times more data. This results in a 4x speedup of the loop process.

4.4 Evaluation

We evaluate the following research questions:

RQ1 How much improvement in defect detection capability is achieved by incorporating both device-side feedback and host-side feedback in HFuzz?

RQ2 How much speed-up is achieved by in-kernel input mutations?

RQ3 How much speed-up is achieved by FPGA-level optimizations for fuzzing?

RQ4 How much overhead is incurred by injecting hardware probes in HFuzz?

To assess the improvement in defect detection and fuzzing acceleration, we compare HFuzz against four baselines.

1. Alternative 1 NaiveFuzz: This option uses branch-coverage guided fuzzing similar to AFL and performs input mutations on CPU side.

2. Alternative 2 SWonly: This option is a replication of the state-of-art work SWonly [182] for Intel DPC++. Compared to HFuzz, it does not have in-kernel probes on FPGA devices and considers only software monitoring feedback.

3. Alternative 3 NoKernelMutation: This option disables in-kernel mutations and performs input mutations on the CPU.

4. Alternative 4 NoHWoptimization: This option disables hardware optimizations and only uses one input queue instead.

Benchmarks. We choose seven applications from Intel's OneAPI GitHub repositories [87]: (R1) Matrix-transform. It has two kernels—one for matrix multiplication M=A*B and the other for reciprocal transformation on each element of M; (R2) Matrix-mul: multiplication of two matrices; (R3) Complex-mul: multiplication of two vectors of complex numbers in parallel; (R4) APSP: the Floyd-Warshall algorithm to find the shortest path between the pairs of vertices in a graph; (R5) Nbody-sim: Simulation of a dynamical system of particles under the influence of gravity; (R6) Hidden-Markov-model: a statistical model using a Markov process; (R7) Match-num: reading data from the host and sending the numbers that match a set of pre-defined constants back to the host.

These benchmarks are widely used in hardware acceleration literature [140] and cover a representative set of optimizations used in kernels (e.g., custom bitwidth, loop unrolling, etc.) and exhibit different memory usage patterns (e.g., buffer memory and unified shared memory for kernel input and output, kernel-to-kernel pipe and kernel-to-host pipe, local memory for in-kernel variables, etc.). Testing difficulties for heterogeneous applications do not depend on the code size; rather, it depends on how hardware resources are synthesized (e.g., in-kernel variables, loop unrolling) and the communication channel details between software and hardware and between hardware kernels. These benchmarks' kernels are widely used and their code size is similar to commercial HLS benchmarks. They are complex in both optimizations and memory arrangements and hard to get right.

Experimental Environment. All experiments were conducted on Intel DevCloud A10 nodes [89]. The automated kernel probe insertion was implemented using DPC++ compiler and Pycparser [38]. The refactored programs were synthesized to RTL and targeted to Intel Arria 10 GX FPGA [93]. We also tried HFuzz on other FPGAs like Intel Stratix 10 SoC FPGA [94] and achieved similar results.

4.4.1 Defect Detection by HW and SW Feedback

We assess the effectiveness of HFuzz's feedback guidance by comparing the number of defects detected through combined hardware probes and software monitors to that of SWonly, which relies solely on software monitors. For each benchmark, we generate test inputs using HFuzz and SWonly for 4 hours. We tried longer time (24 hours) but no more defect is found after 4 hours. Using the generated inputs, we then perform differential testing between CPU-only executions and CPU+FPGA executions and measure the number of defects (i.e., diverging outcomes) found.

Figure 4.5 shows the average experimental results from ten runs. HFuzz is able to detect 3.1× more defects than SWonly. For example, for R5 Nbody-simulation, without monitoring in-kernel variable `sqr`, SWonly cannot find the divide-by-zero error we mentioned in Section 4.2.2 at Lines 16-18 in Figure 4.2. When using SWonly, the value range of kernel inputs does not reflect the change in the square of distance between particles `sqr`. HFuzz, instead, directly monitors the value range of in-kernel variable `sqr`, and finds the defects when `sqr` reaches its minimum value 0. In total, SWonly finds 8 unique defects in 16.5 hours, while HFuzz finds the same defects in 1.6 hours—almost 90% reduction in the testing time.

Table 4.3 lists five defects found by HFuzz in R1 `Matrix-transform`.

First, S1 shows an overflow occurred in the FPGA execution due to the in-kernel variable

Table 4.3: Example symptoms of kernel defects in R1.

ID	**Symptom**	**Description**	**SWonly Find**
S1	Kernel Runtime Overflow	The value of intermediate variables `sum` at line 2 of Figure 4.3 exceeds its bitwidth capacity, leading to a wrong result.	✓
S2	Pipe Write Failure	Pipe write failure happens when FPGA attempts to write into a pipe when the pipe is full.	×
S3	Pipe Read Hang	Pipe read hang happens when FPGA attempts to read synchronously from an empty pipe.	×
S4	Division by Zero	`sum` in line 5 of Figure 4.3 equals 0, leading to divide by zero at line 21.	×
S5	Incorrect Loop Unrolling	CPU and FPGA produce different results when the input array size `num_element` is not multiple of 2.	✓

`sum` at Line 3 in Figure 4.3. It happens when the input vector `a` includes a large number such as 2090401586. By monitoring in-kernel variable `sum`'s value range, HFuzz increases the chance of generating a new vector with large numbers.

Second, two kernels in R1 use a 128-byte pipe to facilitate direct data transfer. As mentioned in Section 4.1, when the first kernel produces results faster than the second kernel can consume, the pipe may become saturated. Consequently, a pipe write failure occurs silently and the newly written value is lost, shown as `S2` in Table 4.3. This may further lead to another defect `S3`: pipe read hang. The second kernel in Figure 4.3 reads values from the pipe for `number_elements` times. However, if the number of values successfully written to the pipe is less than `number_elements`, the second kernel will hang at this pipe read. Both defects cannot be detected by prior work SWONLY because host-side software monitors cannot detect the saturation of commutation pipes.

Third, `S4` depicts a divide-by-zero error caused by the intermediate result `sum` in the second kernel `reciprocalTransform` at Line 21 in Figure 4.3. It happens when both two input matrices are sparse matrices. On CPU, this execution may raise a division-by-zero exception; however, it silently returns an unexpected number on FPGA instead. By monitoring `sum`'s value range, HFUZZ triggers this defect by generating inputs using *Sparsity Mutation*.

Fourth, since R1 makes two copies of the loop body at Line 4 in Figure 4.3 by using `#pragma unroll factor=2`, a wrong result happens if the number of loop iterations `num_elements` is not a multiple of the unroll factor 2.

HFUZZ achieves 10.3× speed-up and finds 25 new defects compared to SWONLY, demonstrating the combined benefit of hardware probes and software monitors.

4.4.2 Speed-up from In-kernel Input Mutations

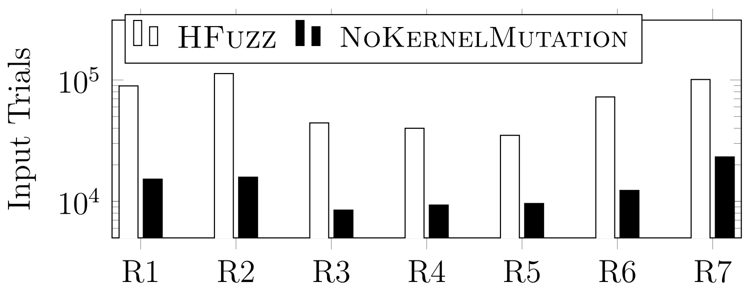


Figure 4.6: Number of Input Trials

To assess speed-up enabled by offloading input mutations to FPGA devices, we compare HFUZZ with a downgraded version NOKERNELMUTATION. We measure the number of generated inputs and defects found within the same 4-hour budget.

Figure 4.6 reports the average number of input trials within 4 hours. For example, in R7, NoKernelMutation generates 23225 inputs, while HFuzz generates 100918 inputs (5.3× speed-up) by avoiding redundant data transfer and parallelizing input mutations. In R2, NoKernelMutation and HFuzz enumerate 15824 and 112940 inputs respectively, leading to 7.1× speed-up. R2 achieves higher speedup than R7 because its performance is more dominated by data transfer as shown in Figure 4.1.

Figure 4.5 shows the number of defects found by NoKernelMutation. While NoKernelMutation reports 14 unique defects in 24 hours, HFuzz detects the same defects in 5.1 hours, which translates to 4.7× speed-up in defect detection. These defects are not found by NoKernelMutation, because it wastes time in sequentially mutating inputs in CPU and sending the large data to the kernel.

HFuzz reduces the need for data transfer by offloading mutations into kernels and thus speeds up fuzzing by 4.7×.

4.4.3 Speed-up from FPGA-level Optimizations

To evaluate the effectiveness of FPGA-level optimizations for input generation, we created a downgraded version of our tool NoHWoptimization, which disables this feature. We evaluated the time taken to find the same defects. The results are shown in Figure 4.5. Compared to NoHWoptimization, HFuzz finds the same 33 bugs 3.4x faster, taking only 8.3 hours as opposed to 28 hours.

In R1 (e.g., Figure 4.3), the detected defects include (1) a divide-by-zero error when the kernel takes as input two sparse matrices and (2) an overflow error when the kernel takes as input two dense matrices with large elements. Because inputs leading to these defects are distinct from each other, traditional mutational fuzzers with a single input queue may be inefficient to find them. In fact, it takes 2 hours to mutate two sparse matrices into dense ones. HFuzz uses one hardware optimization technique, called dynamic kernel sharing, to enable simultaneous exploration of input subspaces originating from different seeds. For that, HFuzz utilizes multiple input generators. One generator A starts with dense matrices and another generator B starts with sparse matrices. HFuzz can detect these two bugs by interleaving generator A and generator B based on runtime feedback. For example, when generator A reaches its maximum value and triggers an overflow, it can no longer provide any new feedback. HFuzz will switch to generator B and detect the divided-by-zero error. HFuzz reduces the detection time to 5 mins.

HFuzz achieves 3.4× speed-up in the detection of detects by implementing hardware optimizations. Loop unrolling, shannaization, and fast memory access directly speed up the mutation process. Dynamic kernel sharing enables efficient input space exploration.

4.4.4 Probe Overhead

Inserting hardware probes into the original kernels may cause extra overhead on hardware resources, as reported in Table 4.4. We measure four types of hardware resource, including ALUT (a lookup table implementing the boolean function), FF (flip flops for storing temporary data), RAM (random access memory blocks), and DSP (a digital signal processing unit for common fixed-point and floating-point arithmetic). The inserted kernel probes incur a relatively large overhead for a simple kernel because the inserted probes significantly increase kernel logic complexity compared to the original kernel. In R2, compared to the original kernel with 9592 ALUTs and 14466 FFs, inserted probes used 22% more ALUTs and 33% more FFs. For a relatively complex kernel R4, the overhead is 6% ALUT and 10% FFs. The extra resource usage mainly comes from (1) the probe computation including read and write, and (2) the kernel dispatch logic establishes the communication between kernel and host.

Table 4.4: Resource overhead from injecting hardware probes.

ID/Program		#LUT	#FF	#RAM	#DSP	Freq /MHz
R1/	Orig	15932	25088	137	4.5	247
Matrix_trans	Probe	17905	34320	192	4.5	246
R2/	Orig	9592	14466	492	16	259
Matrix_mul	Probe	12032	19443	492	16	247
R3/	Orig	11545	18494	106	6	273
Complex_mul	Probe	11203	27117	106	6	253
R4/	Orig	60468	92249	555	195	221
APSP	Probe	64327	101229	558	195	212
R5/	Orig	23642	44352	309	34	270
Nbody_sim	Probe	27612	50549	317	34	260
R6/	Orig	48706	64987	395	67	257
HMM	Probe	56562	87392	491	67	247
R7/	Orig	2239	1357	67	12	279
Match_num	Probe	3828	2033	73	12	259

Such overhead could be further reduced by manual optimizations. For example, Curreri [66] performs resource sharing by using the same FIFO probe for multiple feedback signals.

Hardware probe insertion uses 24% extra LUT, 29% extra FF, and 8% extra RAM, and reduces frequency by 5% on average. However, it enables an overall 10.3× speed-up in defect detection by providing hardware feedback.

4.5 Conclusion

HFuzz is the first grey-box testing approach leverages the *capability of heterogeneous hardware* for testing *heterogeneous applications*. In particular, HFuzz injects hardware probes in addition to injecting software monitors to better guide input generation and offloads iterative input generation to hardware accelerators. HFuzz speeds up fuzzing by offloading input mutations to FPGAs by 4.7× without sacrificing any defect detection capability. It speeds up testing 10.3× on average by gathering meaningful signals from hardware execution directly by injecting in-kernel probes. This work fits the domain of software testing, as it targets HLS C/C++ dialects and it has the potential to significantly improve correctness in the new era of *heterogeneous computing*, where regular software developers write code in HLS C/C++ to exploit custom hardware acceleration.

However, detecting a bug is only the beginning. While applying HFuzz, we identified bugs in Intel's DevCloud. Yet, our industry collaborator at Intel, Hongbo Rong, emphasized the need for deeper insights. Heterogeneous compilers involve multiple compilation layers, making it difficult for humans to pinpoint the root cause of bugs.

While HFuzz effectively uncovers heterogeneous application bugs, understanding why these failures occur remains a significant challenge. **Simply knowing that a bug exists is not enough—we need efficient techniques to isolate the exact sequence of transformations responsible for the failure.** The complexity of heterogeneous compilation demands an **automated debugging approach** to reducing failing test cases and compilation layers while preserving their failure-triggering behavior.

To address this challenge, my next work, DuoReduce, introduces **dependency-aware delta debugging** to automate and refine bug isolation across complex compiler layers. By leveraging structural dependencies in compilation, DuoReduce minimizes test cases and compilation path while maintaining failure conditions, making root cause analysis significantly more efficient.

CHAPTER 5

DuoReduce: Redesign Delta Debugging for Multi-layer Heterogeneous Compiler

While HFuzz efficiently exposes bugs in heterogeneous compilers, it leaves an important question unanswered: **how to isolate the root cause of these failures?** Detecting a bug is only the first step—understanding **why** it occurs is often more challenging, especially in heterogeneous compilation pipelines where multiple transformations interact in complex ways. We asked the following research question, *How can we systematically localize compilation layers and code segments responsible for bugs in heterogeneous computing?* In this chapter, we investigate the sub-hypothesis: *By understanding the relation between the IR code and multi-layer heterogeneous compilers, we can build an efficient debugging tool.*

We introduce DuoReduce, a dependency-aware delta debugging framework designed specifically for heterogeneous compilation. Unlike traditional delta debugging, which reduces test cases indiscriminately, DuoReduce leverages **compilation dependencies** to systematically isolate the minimal sequence of transformations responsible for a failure. By preserving structural integrity while reducing test cases, DuoReduce makes root cause analysis significantly more efficient.

5.1 Introduction

Modern heterogeneous compilers need to support a diverse range of hardware platforms, including CPUs, GPUs, FPGAs, and domain-specific accelerators. To address the complexity inherent in targeting multiple hardware backends, MLIR (Multi-Level Intermediate Representation) has emerged as a powerful infrastructure, providing a unified and modular framework that enables systematic compilation and optimization across heterogeneous architectures. While MLIR's layered design facilitates hardware-agnostic optimizations and translation to platform-specific backends, the increasing complexity and multiple abstraction levels within MLIR compilation introduce new debugging challenges. Bugs or miscompilations can arise at any of these abstraction layers. This motivates the critical need for advanced debugging techniques and effective bug isolation methods explicitly tailored to MLIR-based heterogeneous compilation flows.

Consider the Circuit IR Compilers and Tools (CIRCT) [63] project that leverages the MLIR framework to build a compiler for heterogeneous hardware accelerator compilation. As shown in Figure 5.1, CIRCT transforms high-level language code such as Python and C into Verilog. It defines 30 different hardware-related custom dialects and 206 compilation passes. Similarly, other MLIR-based compilers use a significant number of compilation passes. For instance, GPU stencil optimization [44] has 27 compilation passes. This size of available compilation passes introduces significant complexity in bug isolation. Suppose that a developer attempts to reproduce the same compiler error by exhaustively testing all 2^{27} combinations naively by turning on and off each compilation pass. It could take more than 1000 days to complete. Take the CIRCT Python frontend, PyCDE [24] as another example. It converts high-level Python constructs into optimized Verilog code by utilizing 15 distinct compilation passes. These optimization passes include crucial tasks like resource sharing, pipelining, and clock domain management. A naive attempt would require testing 2^{15} combinations, taking more than 18 hours to complete.

We propose DuoReduce, a dual-dimensional reduction approach for MLIR bug localization. DuoReduce leverages three key ideas in tandem to design an efficient MLIR delta debugger. First, DuoReduce reduces compiler passes that are irrelevant to the bug by identifying ordering dependencies among the different compilation passes. Second, DuoRe-

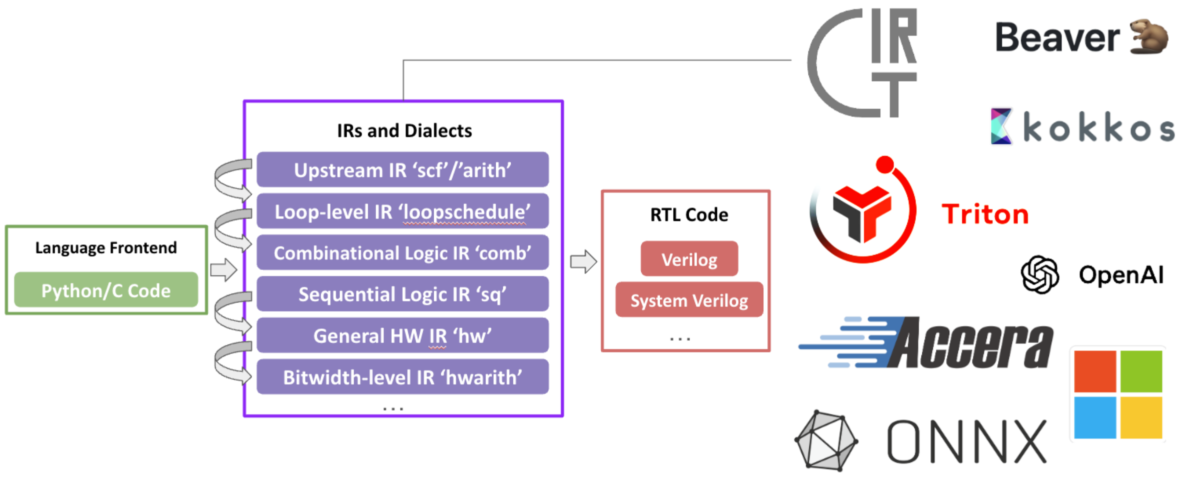


Figure 5.1: CIRCT's multi-layer compilation involves 4 core dialects and 26 optimizing dialects such as `hwarith` and `sq`. An MLIR input program goes through on average 13 transformation passes and produces the resulting Verilog code. The MLIR community is growing and as of Sep. 2024, there are 30 MLIR-based compiler projects on GitHub.

duce uses MLIR-semantics-aware transformations to expedite IR code reduction. Finally, DuoReduce leverages cross-dependence between the IR code dimension and the compilation pass dimension by accounting for which IR code segments are related to which compilation passes to reduce unused passes.

In terms of IR code reduction, DuoReduce outperforms Perses and Vulcan by 32% and 22% respectively. In terms of reducing compilation passes, DuoReduce achieves 14% pass reduction than DuoReduce without dual-dimensional reduction. This translates to not needing to inspect 281 lines of MLIR compiler code per bug. Suppose that each IR program goes through 18 compilation passes on average and each compilation trial takes 2 seconds to finish. Naively enumerating all pass combinations to find buggy passes needs 2^{18} compilation attempts, taking 145 hours. By reasoning about compilation pass dependencies, DuoReduce reduces the IR debugging time from 145 hours to 572.35 seconds.

In the current AI era, where companies such as AWS, Microsoft, and Google invest significant resources for in-house AI processor and hardware compiler development, DuoReduce has the significant potential to expedite the overall development time of extensible optimizing compilers. The remainder of this paper is organized as follows. Section 5.2 introduces MLIR and a motivating example. Section 5.3 presents the design of DuoReduce. Section 5.4 provides the design of our experiments and their results. Section **??** introduces the related work. We draw the conclusions of our work in Section 5.5.

5.2 Background

5.2.1 Multi-Level Intermediate Representation

```
hw.module @Test(in %x : i8, in %clock: i1) {

  %regvar = sv.reg : !hw.inout<i8>
  sv.assign %regvar, %x : i8

  %regwithinit = sv.reg init %x : !hw.inout<i8>
}
```

(a)

Original MLIR code

```
module {
  hw.module @Test(in %x : i8, in %clock : i1) {
    %regvar = sv.reg {hw.verilogName = "regvar"} : !hw.inout<i8>
    sv.assign %regvar, %x : i8
    %regwithinit = sv.reg init %x {hw.verilogName = "regwithinit"} : !hw.inout<i8>
    hw.output}}
```

(b)

MLIR code after pass "lower-hw-to-sv"

```
module Test( // o.mlir:1:1
  input [7:0] x, // o.mlir:1:20
  input       clock // o.mlir:1:32
);
  reg [7:0] regvar; // o.mlir:3:13
  assign regvar = x; // o.mlir:4:3
//correct: reg [7:0] regvar = x;
  reg  [7:0] regwithinit = x;
endmodule
```

(c)

Wrong verilog code after pass "export-verilog"

Figure 5.2: The CIRCT bug # 6317 from GitHub [17] does not include which compilation passes were used to detect this bug. If a user want to compile program (a) with pass `export-verilog` only to reproduce the bug, the compiler will crash with message *"Error: Unsupported operation found in design"* This shows the need of identifying a correct ordered set of compilation passes to reproduce the same bug. The original MLIR code (a) in the `hw` dialect must go through a first compilation pass `lower-hw-to-sv` to produce code (b) and go through a second compilation pass `export-verilog` to produce the Verilog code (c). There are 205 compilation passes available in CIRCT; thus, it is extremely challenging to isolate 2 out of 205 passes.

MLIR [64] aims to offer a unified infrastructure that can represent code at multiple levels of abstraction. Its core idea is to define extensible IRs that can be customized with domain-specific *dialects* [64] using *compilation passes* specific to particular domains, such as machine learning, high-performance computing, or embedded systems. All 30 MLIR projects listed on this website [23] have a large number of compilation passes; Triton [155] from NVIDIA has 258 passes, CIRCT has 205 passes, and ONNX-mlir has 34 passes. As such, MLIR enables code optimization and translation from high-level domain-specific abstractions to lower-level architecture-specific machine code through multiple compilation passes. We use the term *compilation path* P to refer to a sequence of *compilation passes* $P = [P_i, 1 \leq i \leq n]$ For each compilation path P, we define two relations.

- **Execution Order Relation**: we define a binary relation $\leq$ to indicate the *execution order*, such that $P_i \leq P_j$ means P_i is executed before P_j in the compilation path P.

- **Dependence Relation**: A compilation pass P_j depends on another pass $P_i \rightharpoondown P_j$ if executing P_j without executing P_i leads to a different compilation outcome in terms of reproducing the same bug symptom. It is important to note executing P_j after P_i (i.e., $P_i \leq P_j$) does not directly imply P_j depends on P_i (i.e., $P_i \rightharpoondown P_j$). However, $(P_i \rightharpoondown P_j)$ $\Rightarrow$ $(P_i \leq P_j)$.

DuoReduce finds the shortest sublist P' that can preserve the same bug as the original compilation path P. For each pass $P_i \in P'$, each P_k that P_i depends on (i.e., $P_k \rightharpoondown P_i$) must be included in P'. For example, in the GitHub issue 82382 [21], the original post contains P with 10 passes. DuoReduce finds an alternative shortest path P' with 2 passes: `affine-loop-tile` $\rightharpoondown$ `convert-parallel-loops-to-gpu`.

CIRCT [63] has 196 passes to compile high-level Python or C code to custom hardware expressed in low-level RTL. Figure 5.2 shows an example compilation process based on two passes. The registers `%regvar` (line 3 in Figure 5.2a) and `%regwithinit` (line 6) are assigned and initialized with value `%x`. The original MLIR code is first converted to dialect `sv` in Figure 5.2b with pass `lower-hw-to-sv` and then lowered to System Verilog code with pass `export-verilog` in Figure 5.2c: P_1=`lower-hw-to-sv` and P_2=`export-verilog`, where $P_2 \nrightarrow P_1$. When a bug occurs in a compilation path with n passes, developers may naively enumerate all 2^n *combinations* of passes. Such a naive attempt can easily take up to 100 hours when n is over 18. DuoReduce addresses this very problem by accounting for dependencies among compilation passes and by applying IR code reduction in tandem, reducing such time to less than 10 minutes.

5.2.2 Motivating Example

Figure 5.3a illustrates a real-world MLIR GitHub issue [18], which resulted in a segmentation fault. The result of matrix multiplication `@matmul` at line 3 is stored in `%2` and checked at line 8. The original compilation path has 9 passes: `convert-linalg-to-loops`, `affine-loop-unroll`, `convert-scf-to-cf`, `convert-arith-to-llvm`, `convert-linalg-to -llvm`, `convert-memref-to-llvm`, `convert-func-to-llvm`, `reconcile-unrealizedcasts`, and `mlir-cpu-runner`.

The crash message's stack dump in Figure 5.3c indicates that the last used pass is `mlir-cpu-runner`. However, the message does not display the preceding faulty pass `convert -func-to-llvm` responsible for the issue.

For Figure 5.3a, no further IR reduction is possible with existing delta debuggers such as Perses or Vulcan. However, this minimized IR presents ambiguity. *Is the crash caused by flawed implementation of `@matmul` called at line 4? Is the crash due to incorrect transformation of the nested `for` loop at lines 5-6?*

DuoReduce performs dual-dimensional fault localization based on two insights:

1. MLIR compiler bugs are often caused by *operation invocations* rather than the values of operands. For example, in Figure 5.3, using `cf.assert` at line 9 with the `convert-func-to`
`-llvm` pass causes the bug, and the value of operand `%4` is irrelevant.

2. IR code reduction can eliminate unnecessary passes. For example, a loop optimization

pass `affine-loop-unroll` becomes unnecessary when `for` loops are removed.

```
func.func @main() {
  func.func @matmul(%arg0, %arg1,
      %arg2)
    ...
  call @matmul(%2, %0,
      %1):(memref<128x128xf32>,
      memref<128x128xf32>,
      memref<128x128xf32>) -> ()
  scf.for %arg0 = %c0 to %c128 step
      %c1 {
    scf.for %arg1 = %c0 to %c128 step
        %c1 {
      %3 = memref.load %2[%arg0,
          %arg1] : memref<128x128xf32>
      %4 = arith.cmpf oeq, %3, %cst_0
          : f32 -> i1
        cf.assert %4, "Matmul does not
            produce the right output"}}
...
return}}
```

(a) A buggy IR code reported in the MLIR GitHub issue 56914 [18]: compute matrix-multiplication and verify.

```
1     func.func @main() {
2   - ~~func.func matmul(%arg0, %arg1,~~
        ~~%arg2)~~
3       ...
4   - ~~call matmul(%2, %0,~~
        ~~%1):(memref<128x128xf32>,~~
        ~~memref<128x128xf32>,~~
        ~~memref<128x128xf32>) -> ()~~
5   - ~~scf.for %arg0 = %c0 to %c128 step~~
        ~~%c1 {~~
6   - ~~  scf.for %arg1 = %c0 to %c128 step~~
        ~~%c1 {~~
7   - ~~%3 = memref.load %2[%arg0, %arg1] :~~
        ~~memref<128x128xf32>~~
8   - ~~%4 = arith.cmpf oeq, %3, %cst_0 :~~
        ~~f32 -> i1~~
9   + %4 = arith.constant 1: i1
10      cf.assert %4, "Matmul does not
            produce the right output"}
11      }
12  ...
13  return}}
```

(b) After applying constant replacement at lines 8-9, DuoReduce removes lines 2-7.

```
Stack dump:
0.      Program arguments: mlir-cpu-runner -e main -entry-point-result=void
 #0 0x000055d56f369ee0 PrintStackTraceSignalHandler(void*)
 #1 0x000055d56f367904 SignalHandler(int)
 ...
```

(c) LLVM's crash message shows only the last pass `mlir-cpu-runner`, which is not the root cause of the crash.

Figure 5.3: MLIR bug #56914 [18]: the original input IR code in Fig 5.3a and 9 compilation passes to reproduce the crash in Fig 5.3c. If she compiles the input IR with all available 9 passes, 4480 lines of compiler code should be inspected. In Fig 5.3b, DuoReduce successfully removed the `@matmul` with "constant replacement" transformation in Section 5.3.2. DuoReduce removes 6 irrelevant compilation passes out of 9, leaving only 1245 lines of compiler code to inspect.

After performing syntax-aware IR reduction, DuoReduce applies *constant replacement* and *return operand rollback* transformations, introduced in Section 5.3.2. For example, replacing `%4` in line 8 with the constant `0` retains the bug, allowing us to remove related IR using `%3` in line 7 and `@matmul` in line 4. By further removing IR constructs in the nested `for` loop (lines 5-6), DuoReduce eliminates the associated `affine-loop-unroll` pass.

Figure 5.3b shows the minimized code, reproducing the same error reported in Figure 5.3a. This example shows that it is possible to exclude the `@matmul` function from the culprit IR code and remove 6 out of 9 passes. It suggests that the underlying error is caused by 3 remaining passes: `convert-arith-to-llvm`, `convert-func-to-llvm`, or `mlir-cpu-runner`.

5.3 Approach

DuoReduce takes the original IR code C, the compilation path P, and a separate oracle O to check if the error message remains identical to the original. It outputs the minimized IR code C_ans and the reduced path P' for reproducing the bug.

At the heart of DuoReduce is a three-fold approach.

1. **C**ompilation Path Reduction: It reduces unnecessary compilation passes by identifying dependence among involved passes, described in Section 5.3.1.

2. **I**R Code Reduction: In the IR code dimension, it performs syntax-aware reduction in conjunction with MLIR-specific transformation, described in Section 5.3.2.

3. **D**ual-Dimensional Reduction: As the IR code is reduced, DuoReduce's dual-dimensional reduction approach narrows down necessary compilation passes by analyzing the relation between IR code and passes, described in Section 5.3.3.

Algorithm 2 describes DuoReduce's process. In lines 1-7, DuoReduce removes as many unnecessary compilation passes as possible (Section 5.3.1). Once a minimized compilation

Algorithm 2 DuoReduce takes as input the original IR program C, the original compilation path P, and an oracle O. DuoReduce first gets the reduced compilation path P' by identifying the dependency of each pass. Then DuoReduce applies delta debugging with MLIR transformations on the IR code C. In the end, it applies dual-dimensional reduction on the reduced path P' to get P_ans.

```
Require:
    • $C \leftarrow$ Original IR Program
    • $P \leftarrow$ Original Compilation Path
    • $O \leftarrow$ Oracle that checks whether the bug is preserved
Ensure:
    • $D_i \leftarrow$ Shortest path ending with $P_i$ that can reproduce the bugs with the original $C$
    • $P' \leftarrow$ Shortest path that can reproduce the bugs with the original IR program $C$
    • $C\_ans \leftarrow$ Reduced IR Program
    • $P\_ans \leftarrow$ Shortest compilation path that can reproduce the bugs with $C\_ans$
1: for $n = N$ downto 1 do
2:     $D_n$ = DEPEND_GEN($P[1:n]$) // return the shortest path ending with $P_n$.
3:     if Len($P'$) > Len($D_n$) then
4:         $P' = D_n$ //Update $P'$ with the shorter path if possible
5:     end if
6: end for
7: $C\_ans \leftarrow$ DD_IR($P'$, $C$, $O$)
8: $Flag \leftarrow$ True
9: while $Flag$ do //Flag is used to check whether the reduction still happens
10:     $C\_cand \leftarrow$ IR_Trans($C\_ans$, $O$)
11:     $Flag \leftarrow$ False
12:     if Len(DD_IR($P'$, $C\_cand$, $O$)) < $C\_ans$ then
13:         $C\_ans \leftarrow$ DD_IR($P'$, $C\_cand$, $O$)
14:         $Flag \leftarrow$ True
15:     end if
16: end while
17: $P\_ans$ = DUO_REDUCTION($P'$, $C\_ans$, $O$)//Applies dual-dim reduction to further reduce $P'$
18: return $C\_ans$, $P\_ans$
```

path is established, DuoReduce applies IR code reduction at line 8 until no further reduction is possible. Next, DuoReduce applies MLIR-specific transformations at line 11 and applies IR code reduction again on the transformed code at line 14 (Section 5.3.2). DuoReduce then performs dual-dimensional reduction on the compilation path in line 18 (Section 5.3.3).

5.3.1 Dependency-Aware Compilation Path Reduction

MLIR compilers have dependencies among compilation passes, as shown in Figure 5.2. Naively, one may examine all possible pass combinations—2^n combinations where n is the number of passes. However, this approach may lead to a large number of invalid compilation attempts, due to dependencies among passes. In fact, our evaluation showed that exhaustively examining all 2^9 combinations resulted in 89% invalid compilation attempts for a GitHub issue involving 9 passes [21]. Such exhaustive attempts are inefficient. Therefore DuoReduce recognizes dependencies among passes to avoid invalid compilation attempts.

DuoReduce performs this (1) pass reduction phase before the (2) IR code reduction phase and the (3) joint reduction in both dimensions. Our evaluation shows that, without the initial path reduction (1), it takes 16.7% more time to reach the same IR code reduction rate.

Problem Definition. Given a compilation path $P = [P_i, 1 \leq i \leq n]$, DuoReduce identifies the minimum sublist of P as $P' = [P_{i_1}, .., P_{i_m}]$, where $1 \leq i_1 < \ldots < i_m \leq n$, preserving the same bug, meaning (1) P' is a sublist of P; (2) the number of passes included in P' is minimal; and (3) for each pass $P_i \in P'$, the passes that P_i depends on (i.e., $P_k \rightharpoondown P_i$) must also be included in P'.

$$P' = \underset{P' \subseteq P}{\operatorname{argmin}} |\{P_i \in P' | (P_k \rightharpoondown P_i) \Rightarrow (P_k \in P')\}| \tag{5.1}$$

Algorithm 3 `DEPEND_GEN`: Dependency-Aware Reduction Algorithm

Require:

- $Code \leftarrow$ IR Program
- $P \leftarrow$ A Compilation Path ends with P_n, where P_i is each compilation pass, and P_i is execute sequentially after P_{i-1} on this path P.
- $O \leftarrow$ Oracle that checks whether the bug is preserved

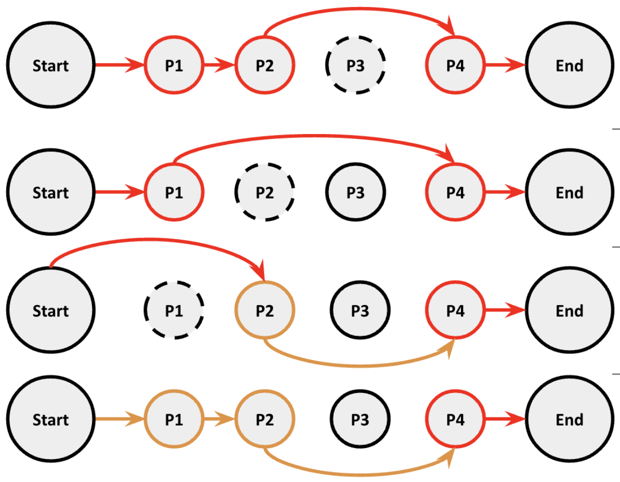
Ensure:

- $D_n \leftarrow$ A sequence of all the compilation passes that P_n depends on.
- $C_{shortest} \leftarrow$ The shortest compilation path ends with P_n that can satisfy the oracle O with $Code$.

```
1:  $D_n = []$
2:
3:  for $i = n - 1$ to 1 do
4:      $C_i = [P_1, \dots, P_{i-1}]$
5:      $C'_i = \text{Concat}(C_i, D_n).\text{add}(P_n)$
6:      if $O(C'_i, Code)$==false then
7:          $D_n.add(P_i)$
8:      end if
9:  end for
10: return $C_{shortest} = D_n.add(P_n)$
```

Identifying Pass Dependencies. Given the original compilation path $[P_1, P_2, \dots, P_N]$, DuoReduce identifies the dependent passes for each pass in a *backward order* in lines 1-3 of Algorithm 2, in the decreasing order of P_N to P_1. We denote the dependencies of P_n as D_n, which is initialized to an empty set. DuoReduce checks P_n's preceding passes (*i.e.,* $P_1, ..., P_{n-1}$) and adds the pass to D_n, if it is necessary for preserving the bug with a compilation path ending with P_n. As shown in line 3 of Algorithm 3, DuoReduce first constructs a set of compilation paths $\{C_1, ..., C_n\}$, where each C_i is the path executing from P_1 up to P_{i-1} (i.e., $C_i = [P_1, \dots, P_{i-1}]$) and is later used to determine whether P_n depends on P_i. For example, $C_3 = [P_1, P_2]$. Next, DuoReduce tests whether P_n depends on P_i, $i < n$, by constructing a modified path C'_i. C'_i concatenates C_i and current D_n and adds P_n at the end, as shown in line 4 of Algorithm 3. In this way, C'_i reserves all passes that P_n depend on, except for P_i. If C'_i fails to retain the same behavior, it indicates that P_n

depends on P_i, and P_i is added to D_n, as shown in lines 5-7 of Algorithm 3. Otherwise, we can safely remove the P_i.



Path to try	Our finding
Try to skip P3; Run P1→P2→P4	P3 can be discard
Try to skip P2; Run P1→P4	P2 cannot be discard
Try to skip P1; Run P2→P4	P1 and P2 cannot be discard
Find shortest path ends with P4: P1→P2→P4	

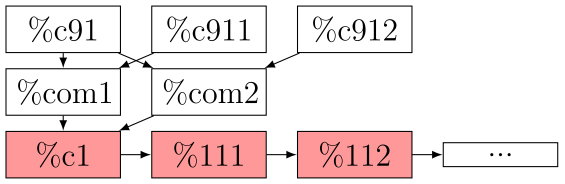
Figure 5.4: Red edges indicate the current passes being tested, and the golden vertices highlight dependent passes. DuoReduce finds the shortest path P from **Start** to **P4**. P4=`export-verilog` does not depend on P3=`-inline` but depends on P1=`lower-hw-to-sv` and P2=`lower-calyx-to-hw`.

Figure 5.4 shows a running example. We start with finding the dependency D_4 for P_4, where D_4 is initially an empty list []. For that, DuoReduce incrementally constructs this list by testing the removal of preceding passes P3 to P1. First, DuoReduce checks if P_4 depends on P_3 by removing it from the compilation passes. Thus, DuoReduce executes $[P_1, P_2, P_4]$. $[P_1, P_2, P_4]$ is constructed by concatenating $C_3 = [P_1, P_2]$ and $D_4 = []$, and adding the target pass P_4 together. Because it retains the same bug, P_4 does not depend on P_3 and P_3 can be safely removed. Next, DuoReduce checks if P_4 depends on P_2 by executing $[P1, P4]$, which is constructed by concatenating $C_2 = [P_1]$ and $D_4 = []$, and adding the target pass P_4. However, it does not reproduce the same bug; thus, DuoReduce adds P_2 into D_4. DuoReduce then checks the dependency against P_1 by removing P_1 and executing $[P_2, P_4]$, which is constructed by concatenating $C_1 = []$ and $D_4 = [P_2]$, and adding P_4. If the same bug is not reproduced, DuoReduce adds P_1 to D_4. Finally, DuoReduce identifies the passes that P_4 depends on as $D_4 = [P_1, P_2]$, and constructs the shortest path that ends with P_4, $[P_1, P_2, P_4]$. This algorithm has the time complexity of $O(n^2)$, where n is the total number of compilation passes.

5.3.2 Transformation-Based Code Reduction

```
1   func.func @func1(%arg0:
        tensor<5x5xf16>, %arg1:
        tensor<?x12xi16>, %arg2:
        vector<5x5xi1>) -> i1 {
2     %c91 = arith.constant 91 : index
3     %c911 = arith.constant 911 : index
4     %c912 = arith.constant 912 : index
5     %com1 = index.divs %c911, %c91
6     %com2 = index.divs %c912, %c91
7     %c1 = index.divs %com1, %com2
8    //Replace with "%c1 = arith.constant
        42 : index" still trigger the
        crash
9     %111 = vector.broadcast %c1 : index
          to vector<21x12x12xindex>
10    %112 = vector.fma %111, %111, %111
          : vector<21x12x12xf32>
11  //Insert "return %112 :
        vector<21x12x12xf32>" + replace
        the function return type to
        "vector" still trigger the crash
12    vector.print %112 :
          vector<21x12x12xf32>
13    %ans = arith.constant true
14    return %ans : i1}
```

(a) This IR code crashes LLVM due to the invocation of `vector.broadcast` at line 9 and `vector.fma` at line 10, with pass `vector-unrolling`.



(b) Dataflow graph for (a): DuoReduce applies constant replacement for `%c1` and return operand rollback for `%112`. It then removes the operands that `%c1` depends on such as `%com1`, and all the operands after `%112` such as `%ans`.

```
func.func @func1(...) ->
    vector<21x12x12xf32> {
  %c1 = arith.constant 42 : index
  %111 = vector.broadcast %c1 :
      index to vector<21x12x12xindex>
  %112 = vector.fma %111, %111, %111
      : vector<21x12x12xf32>
  return %112 : vector<21x12x12xf32>}
```

(c) Constant replacement on `%c1` and return operand rollback on `%112` enabled removal of lines 2-7 and lines 12-13 in (a).

Figure 5.5: Code example from MLIR GitHub issue 64074 [19]. DuoReduce applies program transformation on the 1-minimal IR code and enables a further 75% reduction.

Prior work has found that applying program transformations can improve delta debugging [171, 132, 98, 137]. In MLIR, we observe that compilation crashes are often caused by *operation invocations* rather than the values of operands. We thus hypothesize that altering MLIR operands, can offer new opportunities to further IR code reduction. We design two operand-specific transformations—*constant replacement* and *return operand rollback*. If altering an operand successfully retains the bug behavior, we can eliminate all dependent operands.

1. Constant Replacement. The *Constant Replacement* transformation substitutes operands in the IR code with constants of the same type. In Figure 5.5a, a GitHub issue [19] causes a compiler crash occurs due to the invocations of `vector.broadcast` for the operand `%111` at line 9 and `vector.fma` for the operand `%112` at line 10. The data flow graph in Figure 5.5b shows the code is already 1-minimal. DuoReduce replaces the operand `%c1` at line 7 with a random constant 42 of the same type `index` at line 8. Because the bug is induced by the operation `vector.broadcast` rather than the specific value of `%c1`, replacing line 7 with line 8 triggers the same crash.

DuoReduce then begins another DD cycle. All precedent operands before `%c1` are eliminated because `%c1` is a constant. DuoReduce safely removes all operands that lead to the previous `%c1` and that are not used later—operand `%c91`, `%c911`, `%c912`, `%com1`, and `%com2`. This removes line 2 to line 6, in Figure 5.5c.

2. Return Operand Rollback. The *Return Operand Rollback* transformation returns intermediate operands instead of the last operand. DuoReduce updates the return statement and the function's return type. In Figure 5.5, instead of returning `%ans` at line 14, DuoReduce inserts a `return` statement after each operand to determine if the bug persists. For example, DuoReduce produces a transformation to return at `%112` and updates the return type from `i1` at line 1 to the type of operate `%112`, which is a `vector`. After returning `%112`, DuoReduce safely remove all subsequent operands because they are no longer invoked, removing lines 12-13.

5.3.3 IR-Path Dual-Dimensional Reduction

```
...
//IR Dump Before AffineLoopUnroll(affine-loop-unroll)
module {
  func.func @main() {
  ...
  %4 = arith.constant false
  cf.assert %4, "Matmul does not"}}
  ...
//IR Dump Before ArithToLLVMConversionPass(convert-arith-to-llvm)
module {
  func.func @main() {
  ...
  %4 = arith.constant false
  cf.assert %4, "Matmul does not"}}
  ...
//IR Dump Before ConvertFuncToLLVMPass(convert-func-to-llvm)
module {
  func.func @main() {
  ...
  %4 = llvm.mlir.constant(false) : i1
  cf.assert %4, "Matmul does not"}}
```

Figure 5.6: After removing `@matmul` in Figure 5.3, the pass `affine-loop-unroll` is no longer relevant. DuoReduce removes 4 out of 7 passes, and reduces the inspection scope from 2279 to 1245 lines.

The existing MLIR framework provides the ability to display the impacts of each compilation pass on the IR code by the option `mlir-print-ir-before-all`. When we compile Figure 5.6 with this option and (`affine-loop-unroll`, `convert-arith-to-llvm`, `convert-func-to-llvm`), MLIR prints the IR code before each pass. By comparing the IR code before and after each pass, DuoReduce identifies which passes are no longer necessary. In this example, the IR remains unchanged after the `affine-loop-unroll` pass but is altered following the `convert-func-to-llvm` pass. It indicates that the latter pass has an effect on the IR code, while the former does not.

For example, in Figure 5.3a, the `for` loops in the original IR code necessitate the `affine-loop-unroll` pass. However, after the IR code is reduced to Figure 5.3b which no longer contains these loops, the `loop-unrolling` becomes useless. Similarly, this approach effectively reduces the passes `convert-linalg-to-loops`, `convert-scf-to-cf`, `convert-lin` `alg-to-llvm`, `convert-memref-to-llvm`, and `affine-loop-unroll`, resulting in 63% pass reduction by removing those unnecessary passes. This in turn reduces the inspection scope from 2279 lines of compiler code to 1245 lines.

5.3.4 Overall Time Complexity

DuoReduce operates in three phases: Phase 1 performs compilation path reduction, Phase 2 focuses on IR code reduction, which involves IR code transformation, and Phase 3 applies dual-dimension reduction on the compilation path.

Phase 1 has a time complexity of $O(c^2)$, where c represents the number of compilation passes, as previously discussed.

Phase 2 consists of two key components: IR transformation and syntax-guided delta debugging reduction. Let n denote the number of operands in the IR code. In the worst-case scenario, delta debugging reaches 1-minimal n times, with each step the IR transformation enabling further reduction. For IR transformation, Both constant replacement and return operand rollback traverse each operand once, leading to a time complexity of $O(n)$, where n is the number of operands in code. Since IR transformation may be applied n times in the worst case, the total cost accumulates to $O(n^2)$. For delta-debugging, since we reaches 1-minimal n times, it means we apply delta-debugging n times. While the time cost of each delta-debugging is linear [177], the overall cost for syntax-guided delta debugging is $O(n^2)$. Thus, the total time complexity for phase 2 is $O(n^2) + O(n^2) = O(n^2)$.

Phase 3 iterates over each compilation pass and checks the compilation results, leading to a time complexity of $O(c)$.

Summing the complexities of all three phases, the total time complexity for DuoReduce is $O(c^2) + O(n^2) + O(c) = O(n^2) + O(c^2)$, where c is the number of compilation passes and n is the number of operands in code.

5.4 Evaluation

DuoReduce has three main components: MLIR code transformation, dual-dimensional reduction, and dependency-aware pass reduction. We examine the following research questions to answer the effectiveness of each component:

RQ1: How effective is DuoReduce in terms of *IR code reduction*? How much reduction can be achieved by DuoReduce's MLIR code transformation?

RQ2: How effective is DuoReduce in terms of *compilation pass reduction*? How much reduction can be achieved by DuoReduce's dual-dimensional reduction?

RQ3: How much speedup can DuoReduce achieve with dual-dimensional reduction?

5.4.1 Experiment Design

We select three large MLIR compiler projects, listed in Table 5.1. The systems used in our evaluation, including MLIR [64], CIRCT [63], and ONNX-MLIR [96], are significant-sized real-world projects (444k, 171k, and 96k LOC for MLIR, CIRCT, and ONNX-MLIR, respectively). We examined all issues related to these systems and filtered out 62 out of 134 issues due to challenges in reproducing bugs, as these issues lacked MLIR code inputs or corresponding compilation passes. The GitHub issues are chosen for the latest version of each project. Out of the remaining 72 issues, we removed 41 issues because they were too easy to debug: they involved less than 10 lines of IR code. In the end, we selected 31 reproducible issues for our evaluation. The detailed benchmarks are in the replication package with the GitHub issue IDs and links.

For each IR input reported for a given compiler bug, we determined the super-set of candidate passes based on two criteria: (1) the structure of the IR program, and (2) the dialects used. For instance, if an IR program contains loops and vector operations, we assume that vector unrolling passes such as *-test-vector-unrolling-patterns=unroll-based-on-*

Table 5.1: The systems used in our evaluation are significant-sized real-world projects (444k, 171k, and 96k LOC for MLIR, CIRCT, and ONNX-MLIR, respectively). They suffer from an overwhelming number of compilation passes: 234, 206, and 34 respectively. On the left, BugLoC stands for lines of IR code and BugLoP stands for the number of compilation passes for each bug on average.

Project	LoC	# Compilation Pass	Commit Message	# GitHub Issues	Bug LoC	Bug LoP
MLIR [64]	444k	234	1730	23	132.3	17.9
CIRCT [63]	171k	206	2051	6	32.1	20.2
ONNX-mlir [96]	96k	34	2915	2	23.2	17

type should be included in the scope. Similarly, if the IR program uses the `arith` dialect, lowering passes such as *-convert-arith-to-llvm* should be included. Column **Bug LoP** in Table 5.1 shows the number of candidate passes for the reported bugs.

Evaluation metrics. The following metrics are typically used in previous work on DD [176, 83, 171].

- Average reduction: reduction in IR code size and the number of compilation passes. It is calculated as $\frac{|x|-|x'|}{|x|}$, where $|x|$ and $|x'|$ are the original and reduced IR code size or the length of the compilation path.

- Successful reduction: the number of cases in which buggy compilation passes and IR code segments have been successfully localized by the debugging tool in a 4-hour time limit.

- Time usage: time in seconds that each tool takes.

Baselines. To answer RQ1, we evaluate DuoReduce against Perses+ [25], Vulcan+ [171], `mlir-reduce+` (`circt-reduce+` for the CIRCT bug issue), and DuoReduce_NoTran (DuoReduce without MLIR code transformations).

Perses is a syntax-aware delta debugger by leveraging the ANTLR grammar and prunes the search space by avoiding generating syntactically invalid programs. We configured Perses to use the base MLIR grammar [114]. We then construct **Perses+** by adding DuoReduce's compilation path reduction, since Perses does not account for the dimension of compilation passes at all and it would be unfair to compare DuoReduce against Perses directly. In short, **Perses+** = Perses + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction.

Vulcan, an enhanced version of Perses, introduces additional code transformation rules, such as identifier and subtree replacement, to achieve further reduction. These general transformation rules are not designed for IR code transformations and are thus less effective than DuoReduce. Similar to Perses+, we configured Vulcan with the base MLIR grammar [114]. Vulcan+ is different from the original Vulcan by adding compilation path reduction: **Vulcan+** = Vulcan + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction. In short, **Vulcan+** replace DuoReduce's MLIR transformation with Vulcan's general code transformations.

`mlir-reduce` is a utility provided by the MLIR compiler community to reduce the size of the IR code, and `circt-reduce` [26] is built on top of `mlir-reduce` by considering CIRCT dialects. These domain-specific reducers employ multiple strategies to minimize the input: directly removing operations (akin to Perses), and applying optimization rewrites [64]. However, since the optimization rewrites are designed for compilation, not reduction, they are conservative in terms of reducing IR inputs. Similarly, we construct **mlir-reduce+** = `mlir-reduce` + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction.

To answer RQ2, we compare DuoReduce against its downgraded version DuoReduce_No2Dim without dual-dimension reduction. We demonstrate why the LLVM compiler crash message cannot give correct information to help developers localize the buggy pass. To answer RQ3, we evaluate DuoReduce against its downgraded version DuoReduce_NoDep, DuoReduce without identifying compilation pass dependences. DuoReduce_NoDep exhaustively tries all pass combinations, instead of considering the dependences among passes.

Table 5.2: Effectiveness and Efficiency for DuoReduce. DuoReduce achieves the highest IR code reduction compared to Perses+ and Vulcan+. MLIR transformations enable 31.6% additional IR size reduction. DuoReduce_NoDep cannot finish the reduction for 29 out of 31 GitHub issues in the 4-hour time limit, since it requires 2^n trials, where n is the number of compilation passes.

ID	Tool Name	IR Code		Compilation Path		Time
		Successes Reduction	Average Reduction	Successful Reduction	Average Reduction	Usage
1	mlir-reduce+	27	26.1%	27	84.2%	530.65s
2	Perses+	31	47.2%	31	87.2%	414.37s
3	Vulcan+	31	51.1%	31	89.3%	1336.55s
4	DuoReduce _NoDEP	31	47.2%	31	87.2%	397.45s
5	DuoReduce _NoTRAN	2	/	2	/	∼ 145h
6	DuoReduce _No2DIM	31	62.1%	31	77.1%	570.77s
7	Compiler Crash Msg	/	/	15	94.4%	N/A
8	DuoReduce	**31**	**62.1%**	**31**	**91.7%**	**572.35s**

Experimental environment. All experiments are performed on a machine with an AMD Ryzen 2950X 16-Core Processor with 32 GB RAM running on Ubuntu 22.04.

5.4.2 RQ1: Effectiveness of MLIR Code Transformations

As shown in Table 5.2, `mlir-reduce+` has the worst performance, since it applies the naive ddmin [176] approach. DuoReduce outperforms `mlir-reduce+`, Perses+/DuoReduce_NoTran, and Vulcan+ by 138%, 31.6%, and 21.5%, respectively, in terms of the IR reduction rate. Although DuoReduce shares the same compilation pass reduction with the above baselines, the better IR code reduction leads to a better compilation pass reduction rate. As shown in the column **C**ompilation Path - Average Reduction, compared to `mlir-reduce+`, Perses+/DuoReduce_NoTran, and Vulcan+, DuoReduce achieves 7.5%, 4.5%, and 2.4% higher reduction rate respectively.

```
1  module {
2    func.func @func2(...) {
3      %c11 = arith.constant 11 : index
4      %c12 = arith.constant 12 : index
5      %com1 = index.divs %c11, %c11
6      %com2 = index.divs %c12, %c12
7      %c2 = index.divs %com1, %com2
8      ...
9      %92 = affine.apply affine_map<(d0,
           d1, d2, d3) -> (d0 - 16)>(%46,
           %28, %c2, %43)
10     scf.index_switch %92
11     default {....}}
```

(a) The 1-minimal code example: the compilation crash is due to `scf.index_switch`.

```
1  module {
2    func.func @func2(%arg0:
         tensor<?x?x?xi1>, %arg1:
         tensor<5x5xi32>) {
3      %c2 = arith.constant 2 : index
4      ...
5      %92 = affine.apply affine_map<(d0,
           d1, d2, d3) -> (d0 - 16)>(%46,
           %28, %c2, %43)
6      scf.index_switch %92
7      default {....}}
```

(b) DuoReduce applies constant replacement for `%c2`, enabling further removal of all the operands `%c2` depends on.

Figure 5.7: MLIR GitHub issue 64071 [20]. DuoReduce achieves 4 more lines of code reduction with the constant replacement compared to Perses+ and Vulcan+, and only takes 617.4 seconds compared to Vulcan+ which takes 2604 seconds, resulting in 3.87× speedup.

Figure 5.7 is an example of how DuoReduce outperforms Perses+ and Vulcan+. The compilation crash is caused by the logic in `scf.index_switch` at line 10 in Figure 5.7a, which works like the `case` statement in C/C++. Perses+ and `mlir-reduce+` reach 1-minimal in Figure 5.7a; however, the code can be further reduced if we replace the operand `%c2` at line 7 with a constant, since the crash is due to the invocation of `scf.index_switch` instead of the value of `%c2`. DuoReduce replaces the operands `%c2` with a random constant 2 and further reduces the IR code to Figure 5.7b.

Compared to Perses+, DuoReduce takes 38.1% more time to finish the entire DD process on average, as shown in the **Time Usage** column of Table 5.2. It is because after reaching the fixed point, DuoReduce applies IR program transformations and enables further reduction that Perses+ cannot reach. However, by using these transformations, DuoReduce achieves a 14.9% higher reduction rate and outperforms Perses+ by 31.6%.

Although Vulcan+ also applies code transformations, its general code transformations are not suitable for MLIR and thus less effective. For example, in Figure 5.7a, Vulcan+ cannot eliminate the logic dependency of the operand `%c2`, which prevents it from removing the code in lines 3-6. Vulcan+'s identifier replacement technique, which substitutes one identifier with another, fails to break the logic dependency on the operand in line 7. Similarly, its subtree replacement strategy, which shares the same logic as identifier replacement, cannot address this dependency either. Additionally, the time complexity of Vulcan+'s transformations is $O(n^2)$ [171], which is significantly higher than DuoReduce's transformation complexity of $O(n)$, as discussed in Section 5.3. For instance, in Figure 5.7, Vulcan+ takes 2604 seconds to complete the transformation, whereas DuoReduce only requires 617.4 seconds. As indicated in the **Time Usage** column of Table 5.2, DuoReduce consistently outperforms Vulcan+, reducing the time required from 1336.55 seconds to 572.35 seconds on average. DuoReduce also achieves an 11.0% higher reduction rate and outperforms Vulcan+ by 21.5%.

With IR code transformations, DuoReduce outperforms the state-of-the-art DD methods, Perses and Vulcan, in terms of IR code reduction by 31.6% and 21.5%, respectively.

5.4.3 RQ2: Effectiveness of Dual-Dimensional Reduction

```
1 func.func @main() {
2   %in_buf = memref.alloc() :
        memref<16x230x230x3xf32>
3   %filter_buf = memref.alloc() :
        memref<64x7x7x3xf32>
4   %out_buf = memref.alloc() :
        memref<16x112x112x64xf32>
5   linalg.conv_2d_nhwc_fhwc {dilations
        = dense<1> : tensor<2xi64>,
        strides = dense<2> :
        tensor<2xi64>}
6     ins (%in_buf, %filter_buf:
          memref<16x230x230x3xf32>,
          memref<64x7x7x3xf32>)
7     outs (%out_buf:
          memref<16x112x112x64xf32>)
8   return}
```

(a) The above code example crashes because of the loop tiling. It takes two tensors as input and computes the 2d convolution.

```
Stack dump:
0. Program arguments: mlir-opt
      --convert-linalg-to-affine-loops
      --affine-loop-tile=tile-sizes=4,28,28,..
      --affine-loop-unroll=unroll-factor=4
      --canonicalize --affine-parallelize
      --lower-affine --canonicalize
      --gpu-map-parallel-loops
      --convert-parallel-loops-to-gpu conv2d.mlir
...
#22 0x00005c544cb547a3 processParallelLoop...
...
#25 0x00005c544cb53b93 ... const
      /home/mlir/lib/Conversion/SCFToGPU/
      SCFToGPU.cpp:642:11
```

(b) Compiler crash message localizes the crash to the wrong pass `convert-parallel-loops-to-gpu`.

Figure 5.8: MLIR GitHub issue 82382 [21]. DuoReduce finds the buggy pass `affine-loop-tile` in the results while the compiler crash message doesn't, which motivates the need for DuoReduce.

We evaluate DuoReduce against its downgraded version DuoReduce_No2Dim and LLVM compiler crash message to show the benefit of dual-dimensional reduction.

As shown in Table 5.2, comparing row **6** and row **8** shows that DuoReduce achieved 14.6% higher pass reduction while only taking 1.58 seconds longer than DuoReduce_-No2Dim. LLVM compiler crash messages cannot handle non-crash bugs like wrong code generated for CIRCT GitHub issue 6317 [17] in Figure 5.2. In total, LLVM fails to localize correct buggy passes for 52% GitHub issues.

Take Figure 5.8 as an example. The original code in Figure 5.8a transforms a 2-dimension tensor using `linalg.conv_2d` at line 5 and stores the result in `out_buf` at line 7. LLVM reports a crash in pass `convert-parallel-loops-to-gpu`, as shown at `#22` in Figure 5.8b. However, the crash is actually triggered by the parameter chosen in pass

```
1  module {
2    func.func @omp_target() {
3      %alloca = memref.alloca() :
           memref<64x64xf64>
4      ...
5      %0 = omp.map_info var_ptr(%alloca
           : memref<64x64xf64>,
           tensor<?xi32>) map_clauses(to)
           capture(ByRef) ->
           memref<64x64xf64>
6      ...
7      omp.parallel {
8        ...
9        %2 = vector.load %arg0[%arg2,
             %arg3] : memref<64x64xf64>,
             vector<16xf64>
10       ...}}
11     return}}
```

(a) The code example that crashes with a compilation path including 4 passes. This results in the inspection of 2423 lines of MLIR compiler code for debugging.

```
module {
  func.func @omp_target() {
    %alloca = memref.alloca() :
        memref<64x64xf64>
    %0 = omp.map_info var_ptr(%alloca
        : memref<64x64xf64>,
        tensor<?xi32>) map_clauses(to)
        capture(ByRef) ->
        memref<64x64xf64>
    return}}
```

(b) This reduced IR code enables DuoReduce reduce the compilation path from 4 passes: `convert-vector-to-llvm`, `finalize-memref-to-llvm`, `convert-arith-to-llvm`, and `convert-openmp-to-llvm`, to a single pass: `convert-openmp-to-llvm`, and results in the inspection of 284 lines of MLIR compiler code for debugging.

Figure 5.9: MLIR GitHub issue 76579 [22]. With dual-dimensional reduction, DuoReduce removes 3 redundant passes and achieves better reduction than DuoReduce_No2Dim, which translates to not needing to inspect 2139 lines of MLIR compiler code.

`affine-loop-tile`. Unlike LLVM crash report, DuoReduce localizes all four buggy passes: `convert-linalg-to-affine-loops`, `affine-loop-tile=...`, `affine-loop-unroll`, `gpu-map-parallel-loops`.

Some compilation passes are not related to the crash but are essential for code compilation, and with dual-dimensional reduction, DuoReduce may remove them after IR code reduction. For example, in Figure 5.9a, the vector at line 9 needs the pass `convert-vector-to-llvm` to lower the `vector` to `llvm` dialect and make the code executable. However, after removing line 9, which is unrelated to the crash, DuoReduce finds that the pass `convert-vector-to-llvm` can be safely removed while the crash remains. In the end, compared to DuoReduce_No2Dim, DuoReduce achieved an additional 75% reduction rate on compilation passes. On average, DuoReduce achieved a 91.7% reduction rate compared to DuoReduce_No2Dim with 77.1%.

Compiler crash messages cannot correctly isolate culprit passes in 16 out of 31 GitHub issues. DuoReduce outperforms DuoReduce_No2Dim and achieves a higher pass reduction of 14.6%, translating to not needing to examine 281 lines of MLIR compiler code on average.

5.4.4 RQ3: Effectiveness of Compilation Pass Dependence-Aware Reduction

As shown in Table 5.2 row **5** and row **8**, DuoReduce_NoDep cannot finish the compilation pass reduction for 29 out of 31 GitHub issues in the 4-hour time limit, while DuoReduce finishes all the tasks with an average time of 572.35 seconds. In our evaluation, DuoReduce_NoDep cannot finish the reduction tasks in the 4-hour time limit when the number of compilation passes reaches 14.

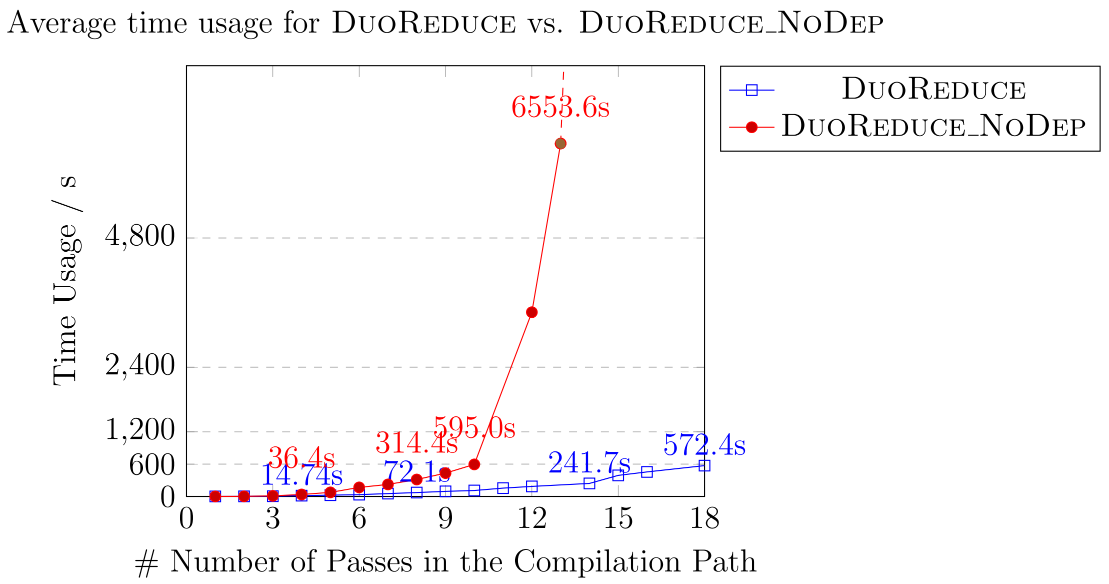


Figure 5.10: Time usage for DuoReduce and DuoReduce_NoDep. The x-axis represents the number of passes, and the y-axis represents the average time usage. DuoReduce achieves higher speedup when more compilation passes are involved. For example, for 13 compilation passes, DuoReduce takes 241.7s, while DuoReduce_NoDep takes 6553.6s, resulting in 27× speedup.

Take the compilation path from GitHub issues [21] in Figure 5.8 as an example. The original compilation path has 9 passes, as listed in the stack dump after the *Program arguments*. DuoReduce first figures out the dependency relation in the compilation path. For example, `affine-loop-unroll` depends on `covert-linalg-to-affine-loops` and `affine-loop-tile=...`. With DuoReduce's dependency-aware reduction, DuoReduce achieves the same reduction rate with 101 compilation trials, compared to DuoReduce_NoDep with 2^9=512 compilation trails, resulting in a 5.1× speedup.

We compare DuoReduce against DuoReduce_NoDep to examine DuoReduce's scalability as the number of compilation passes grows. Figure 5.10 shows that DuoReduce's efficiency escalates with the increase in the number of compilation passes. This improvement is because more passes lead to a denser web of dependencies among the passes. With the compilation pass dependency, DuoReduce effectively reduces the number of unnecessary compilation trials. For example, for the GitHub issues with 4 passes like the example in Figure 5.9, DuoReduce requires 14.7 seconds on average to complete the debugging, compared to DuoReduce_NoDep, which takes 36.4 seconds, resulting in a 2.47× speedup. The efficiency gains are more pronounced with 13 passes, where DuoReduce averages 241.7 seconds, outperforming DuoReduce_NoDep's 6553.6 seconds, thereby achieving a 27× speedup. DuoReduce_NoDep cannot finish the delta debugging process within the time limit (4 hours) when the number of compilation passes reaches 14, while DuoReduce finished its process for 18 compilation passes in 572.4 seconds.

DuoReduce finished the compilation path reduction with an average time of 572.35 seconds, while DuoReduce_NoDep needs an estimated 145 hours, achieving a 901× speedup. The result shows the effectiveness of compilation pass dependency aware reduction.

5.5 Conclusion

Multi-Level Intermediate Representation (MLIR) is increasingly gaining prominence in compiler development in the domain of machine learning, high-performance computing, and embedded systems. By serving as a bridge between high-level languages and low-level machine code, MLIR facilitates optimizations across multiple layers of abstraction. Some MLIR compilers have over 200 compilation passes available. As of September 2024, the top 3 MLIR projects Triton, CIRCT, and tensorflow-mlir have 258, 206, and 364 compilation passes available. Due to the complexity of multi-pass compilation, debugging extensible compilers is challenging.

We present DuoReduce, a novel dual-dimensional debugging approach designed for such extensible compiler development. Its innovation centers around handling dependencies among compilation passes to streamline the debugging search space and utilizing MLIR program transformation to further decompose the IR code into fine-granular units. Demonstrated through experiments on three large MLIR projects, DuoReduce significantly outperforms all seven baselines. Compared to Perses and Vulcan, it improves the IR reduction rate by 31.6% and 21.5% respectively. While none of the baselines isolate culprit passes, DuoReduce achieves an estimated 901× speedup with dependency-aware compilation pass reduction. The experiment results prove DuoReduce's effectiveness in debugging multi-layer extensible compilers, showing significant potential to reduce the development cost of extensible optimizing compilers.

CHAPTER 6

Conclusion and Future Direction

In this concluding chapter, we reflect on the key findings, discuss the broader implications of our work, and identify potential directions for future research.

6.1 Conclusion

Moore's law is reaching its limit, pushing modern computing systems toward heterogeneous architectures that integrate FPGAs, GPUs, and quantum processors. As software increasingly relies on these specialized hardware accelerators, ensuring its correctness and reliability becomes a critical challenge. To bridge the gap between heterogeneous computing and software developers, robust testing and debugging tools are essential.

However, traditional software testing and debugging approaches fall short in heterogeneous computing environments. These methods were designed with conventional CPU architectures in mind and fail to account for the unique execution behaviors and hardware characteristics of accelerators. For example, quantum computers produce probabilistic outputs, fundamentally differing from deterministic CPU execution. To address these challenges, my research leverages hardware accelerator capabilities and integrates hardware-aware execution behaviors into efficient testing and debugging methodologies for heterogeneous computing.

Chapter 3 introduces QDiff, a differential testing framework for quantum software stacks. By considering the unique execution and measurement characteristics of quantum circuits, QDiff automatically detects bugs in quantum software by identifying result divergence in quantum circuit executions. Chapter 4 presents HFuzz, a fuzz testing technique designed to enhance testing efficiency for heterogeneous applications running on FPGAs. HFuzz accelerates testing by inserting in-kernel probes to gather hardware execution feedback and offloading input mutations to FPGA hardware, significantly improving testing throughput.

Moving beyond testing, Chapter 5 explores debugging heterogeneous compilers. DuoReduce introduces a dual-dimensional error localization strategy that systematically analyzes errors across both the IR code dimension and the compilation path dimension. By combining delta debugging with dependency-aware compilation path reduction, DuoReduce identifies the minimal subset of IR code and compilation passes responsible for triggering an error, significantly reducing debugging effort and improving compiler reliability.

By incorporating hardware execution feedback, leveraging acceleration capabilities, and redesigning traditional software approaches, these works effectively bridge the gap between software development and specialized hardware architectures. This dissertation not only addresses the fundamental challenge of ensuring software correctness in heterogeneous environments but also lays the foundation for more scalable, automated, and hardware-aware software techniques in the future.

6.2 Discussion

Q1: Is it possible to apply delta-debugging to quantum circuit, to identify the root cause of the hardware noise? Yes, delta debugging can be adapted for quantum computing to identify the root cause of hardware noise by reducing quantum circuits while preserving the observed noise-induced errors. However, applying delta debugging in this domain presents unique challenges due to probabilistic measurement outcomes. Delta debugging relies on reproducible failures, but quantum noise is inherently stochastic. We can adopt QDiff [164]'s approach and define failure as a statistical deviation in output distributions rather than a single incorrect result.

Q2: If we want to redesign fuzzing like HFuzz [163], what kind of new work do we need to target a new hardware accelerator? Fuzzing optimizations must be highly tailored to the target hardware architecture. The hardware-specific optimizations used in HFuzz [163], such as Shannolization and loop unrolling, are commonly applied in FPGA compilation to enhance parallelism and resource utilization. However, when redesigning fuzzing for a different hardware accelerator, the fuzzing strategy must align with the accelerator's computational strengths. Take GPU as an example. maximizing parallelism is crucial. The fuzzing framework should be designed to launch thousands of threads simultaneously, distributing test cases efficiently across CUDA warps or OpenCL workgroups. For TPUs, test case generation should be transformed to align with tensor-based computation, as TPUs excel at matrix operations.

Q3: You design software approach in a very specialized way. What about using an OS abstraction that abstracts the given hardware accelerators and using a general software tool?

Our goal is to test low-level hardware platforms that can be used to build OS abstractions. Current OS abstractions do not expose low-level hardware behavior. Even if we have a good OS, different hardware accelerators require different testing approaches. GPUs require testing for warp divergence and shared memory conflicts. TPUs require testing for tensor precision loss and under-utilized matrix units. FPGAs require testing for timing violations, channel usage, and memory issues. A one-size-fits-all approach cannot efficiently cover these hardware-specific issues.

6.3 Future Direction

My PhD research introduces a broad suite of software tools to support heterogeneous computing, addressing key challenges in testing, debugging, and refactoring. However, despite these advancements, developing heterogeneous applications remains a difficult task for developers. A significant source of this complexity lies in the compiler infrastructure, which plays a crucial role in optimizing and transforming code for specialized hardware. To further ease the development of heterogeneous computing, this dissertation concludes with several open questions and potential future directions inspired by our findings.

6.3.1 Challenges in Quantum Computing: Embracing Noise in Compilation

Quantum hardware is inherently noisy. For example, the error rate of a two-qubit gate remains around 0.1%, posing a significant challenge for large-scale quantum programs, which commonly involve 128 qubits or more [29]. Given these constraints, quantum compilers primarily focus on reducing the number of two-qubit gates and optimizing circuit depth to minimize the impact of noise. Significant research efforts have been devoted to quantum compiler optimizations [57, 170, 76, 110].

While existing quantum compilers rely on strict mathematical matrix equivalence transformations, the presence of heavy hardware noise calls this strictness into question. Exact transformations may not always yield the highest fidelity when executing on real quantum hardware. This insight parallels the concept of approximate compilation in traditional scientific computing, where relaxing mathematical precision can lead to better overall performance. By extending this principle to quantum compilation, relaxing matrix transformations may actually improve execution fidelity on noisy quantum devices.

However, approximation should be both application-aware and hardware-aware. Different applications and different quantum hardware favor different approximations. For example, quantum error models vary significantly between superconducting quantum computers and neutral-atom-based quantum computers. As shown in Figure 6.1, superconducting qubits typically suffer from limited qubit connectivity and short coherence times, making gate count reduction a primary optimization goal [85]. In contrast, neutral-atom-based quantum systems have longer coherence times but different connectivity constraints, requiring optimizations tailored to their unique hardware properties [167]. Understanding these differences is critical for designing adaptive compilation strategies that select the most effective approximations for a given hardware architecture and application domain.

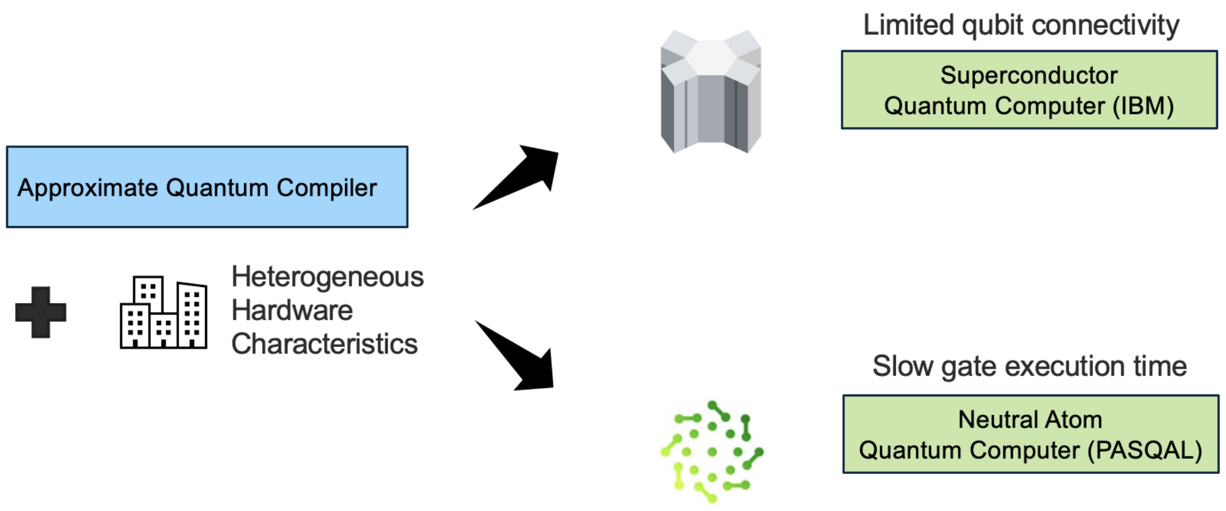

Limited qubit connectivity
Superconductor Quantum Computer (IBM)
Approximate Quantum Compiler
Heterogeneous Hardware Characteristics
Slow gate execution time
Neutral Atom Quantum Computer (PASQAL)

Figure 6.1: A quantum compiler should employ diverse compilation strategies, since different quantum computers have different hardware constraints.

Integrating hardware-specific noise models and application-driven approximations into quantum compilers will be essential for advancing the field and enabling scalable quantum computation.

6.3.2 Challenges in Heterogeneous Application Development: Optimization

Optimizing heterogeneous applications is inherently complex due to the multiple compilation paths available in heterogeneous compilers. Each path applies different optimizations and transformations, and selecting the most efficient path depends on various factors such as memory hierarchy, execution parallelism, and data movement [63]. However, developers often lack visibility into which compilation path is best suited for their application, making manual tuning both time-consuming and error-prone.

As shown in Figure 6.2, we propose a profiling-guided allocator where profiling and hardware feedback dynamically influence the selection of compilation paths. Instead of relying on static optimization heuristics, our approach continuously monitors runtime performance metrics, such as memory utilization, cache efficiency, and computation bottlenecks. Based on this feedback, the compiler can adaptively adjust optimizations to improve performance.

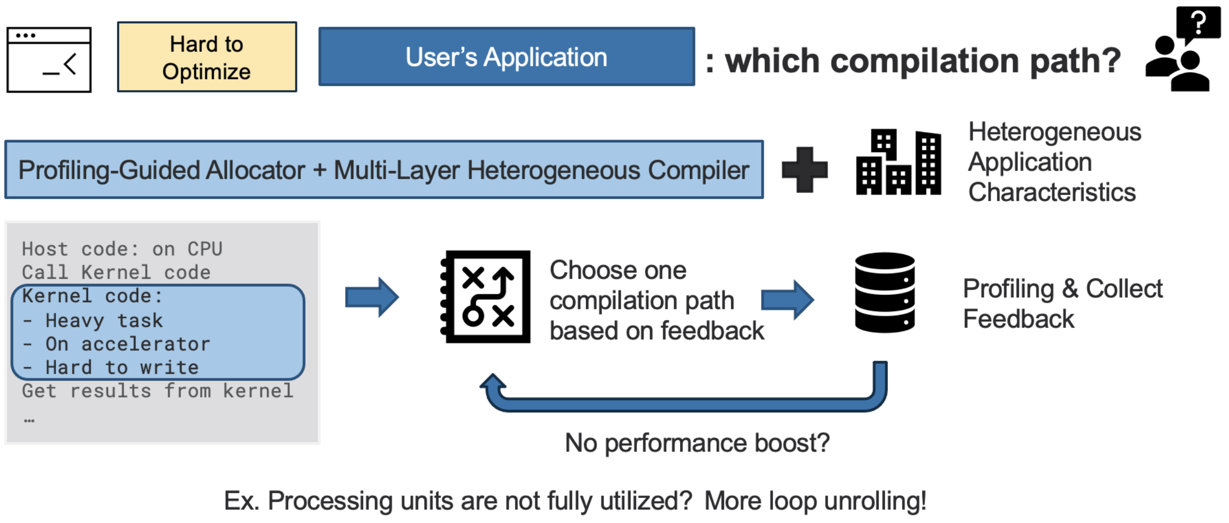

Hard to Optimize
User's Application
: which compilation path?
Profiling-Guided Allocator + Multi-Layer Heterogeneous Compiler
Heterogeneous Application Characteristics
Host code: on CPU
Call Kernel code
Kernel code:
- Heavy task
- On accelerator
- Hard to write
Get results from kernel
…
Choose one compilation path based on feedback
Profiling & Collect Feedback
No performance boost?
Ex. Processing units are not fully utilized? More loop unrolling!

Figure 6.2: Overview for a Profiling-Guided Allocator

For example, consider an application that initially follows a specific compilation path. If hardware profiling detects low memory utilization, this suggests that the workload could benefit from higher parallelism. In response, the compiler can enable more aggressive loop unrolling or adjust data partitioning strategies to better utilize hardware resources. Similarly, if the profiling reveals high register pressure or excessive memory stalls, the compiler can choose an alternative path that prioritizes register reuse and optimized data movement.

By closing the loop between compilation and execution, our approach enables adaptive optimization that tailors compilation decisions to the actual hardware behavior. This reduces manual tuning efforts and allows heterogeneous applications to achieve near-optimal performance across diverse architectures without requiring extensive hardware expertise from developers.

REFERENCES

[1] Google oss-fuzz. `https://google.github.io/oss-fuzz/`.

[2] `https://github.com/quantumlib/Cirq/issues/2240`, 2019.

[3] `https://github.com/quantumlib/Cirq/issues/1713`, 2019.

[4] `https://quantumcomputing.stackexchange.com/questions/8694/is-there-a-mistake-in-the-vqe-ansatz-in-cirqs-tutorial`, 2019.

[5] `https://github.com/rigetti/pyquil/issues/1050`, 2019.

[6] Cirq: A python framework for creating, editing, and invoking noisy intermediate scale quantum (nisq) circuits, 2019.

[7] `https://github.com/rigetti/pyquil/issues`, 2020.

[8] `https://github.com/quantumlib/Cirq/issues`, 2020.

[9] `https://github.com/quantumlib/Cirq/issues/673`, 2020.

[10] `https://github.com/rigetti/pyquil/issues/1034`, Nov,20 2020.

[11] `https://github.com/quantumlib/Cirq/issues/3907`, Nov,20 2020.

[12] `https://github.com/rigetti/pyquil/issues/1259`, Nov,20 2020.

[13] American fuzz loop. `http://lcamtuf.coredump.cx/afl/`, 2020.

[14] `https://stackoverflow.com/search?q=quantum++error`, April,20 2021.

[15] `https://quantumai.google/cirq/google/best\_practices#keep\_qubits\_busy`, 2021.

[16] `https://github.com/Qiskit/qiskit/issues/11010`, Oct 2023.

[17] `https://github.com/llvm/circt/issues/6317`, 2024.

[18] `https://github.com/llvm/llvm-project/issues/56914`, 2024.

[19] `https://github.com/llvm/llvm-project/issues/64074`, 2024.

[20] `https://github.com/llvm/llvm-project/issues/64071`, 2024.

[21] `https://github.com/llvm/llvm-project/issues/82382`, 2024.

[22] `https://github.com/llvm/llvm-project/issues/76579`, 2024.

[23] `https://mlir.llvm.org/`, 2024.

[24] `https://circt.llvm.org/docs/PyCDE/basics/`, 2024.

[25] `https://github.com/uw-pluverse/perses`, 2024.

[26] `https://github.com/llvm/circt/tree/main/test/circt-reduce`, 2024.

[27] Scott Aaronson and Lijie Chen. Complexity-theoretic foundations of quantum supremacy experiments. *arXiv preprint arXiv:1612.05903*, 2016.

[28] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[29] Héctor Abraham, AduOffei, Rochisha Agarwal, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Eli Arbel, Arijit02, Abraham Asfaw, Artur Avkhadiev, Carlos Azaustre, AzizNgoueya, Abhik Banerjee, Aman Bansal, Panagiotis Barkoutsos, Ashish Barnawal, George Barron, George S. Barron, Luciano Bello, Yael Ben-Haim, Daniel Bevenius, Arjun Bhobe, Lev S. Bishop, Carsten Blank, Sorin Bolos, Samuel Bosch, Brandon, Sergey Bravyi, Bryce-Fuller, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adrian Chen, Chun-Fu Chen, Edward Chen, Jielun (Chris) Chen, Richard Chen, Jerry M. Chow, Spencer Churchill, Christian Claus, Christian Clauss, Romilly Cocking, Filipe Correa, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Tareq El Dandachi, Marcus Daniels, Matthieu Dartiailh, DavideFrr, Abdón Rodríguez Davila, Anton Dekusar, Delton Ding, Jun Doi, Eric Drechsler, Drew, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem EL-Safty, Eric Eastman, Grant Eberle, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Axel Hernández Ferrera, Romain Fouilland, FranckChevallier, Albert Frisch, Andreas Fuhrer, Bryce Fuller, MELVIN GEORGE, Julien Gacon, Borja Godoy Gago, Claudio Gambella, Jay M. Gambetta, Adhisha Gammanpila, Luis Garcia, Tanya Garg, Shelly Garion, Austin Gilliam, Aditya Giridharan, Juan Gomez-Mosquera, Gonzalo, Salvador de la Puente González, Jesse Gorzinski, Ian Gould, Donny Greenberg, Dmitry Grinko, Wen Guan, John A. Gunnels, Mikael Haglund, Isabel Haide, Ikko Hamamura, Omar Costa Hamido, Frank Harkins, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Stefan Hillmich, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Junye Huang, Rolf Huisman, Haruki Imai, Takashi Imamichi, Kazuaki Ishizaki, Raban Iten, Toshinari Itoko, JamesSeaward, Ali Javadi, Ali Javadi-Abhari, Wahaj Javed, Jessica, Madhav Jivrajani, Kiran Johns, Scott Johnstun, Jonathan-Shoemaker, Vismai K, Tal Kachmann, Akshay Kale, Naoki Kanazawa, Kang-Bae, Anton Karazeev, Paul Kassebaum, Josh Kelso, Spencer King, Knabberjoe, Yuri Kobayashi, Arseny Kovyrshin, Rajiv Krishnakumar, Vivek Krishnan, Kevin

Krsulich, Prasad Kumkar, Gawel Kus, Ryan LaRose, Enrique Lacal, Raphaël Lambert, John Lapeyre, Joe Latone, Scott Lawrence, Christina Lee, Gushu Li, Dennis Liu, Peng Liu, Yunho Maeng, Kahan Majmudar, Aleksei Malyshev, Joshua Manela, Jakub Marecek, Manoel Marques, Dmitri Maslov, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Dan McPherson, Srujan Meesala, Thomas Metcalfe, Martin Mevissen, Andrew Meyer, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Abby Mitchell, Nikolaj Moll, Jhon Montanez, Gabriel Monteiro, Michael Duane Mooring, Renier Morales, Niall Moran, Mario Motta, MrF, Prakash Murali, Jan Müggenburg, David Nadlinger, Ken Nakanishi, Giacomo Nannicini, Paul Nation, Edwin Navarro, Yehuda Naveh, Scott Wyman Neagle, Patrick Neuweiler, Johan Nicander, Pradeep Niroula, Hassi Norlen, NuoWenLei, Lee James O'Riordan, Oluwatobi Ogunbayo, Pauline Ollitrault, Raul Otaolea, Steven Oud, Dan Padilha, Hanhee Paik, Soham Pal, Yuchen Pang, Vincent R. Pascuzzi, Simone Perriello, Anna Phan, Francesco Piro, Marco Pistoia, Christophe Piveteau, Pierre Pocreau, Alejandro Pozas-iKerstjens, Milos Prokop, Viktor Prutyanov, Daniel Puzzuoli, Jesús Pérez, Quintiii, Rafey Iqbal Rahman, Arun Raja, Nipun Ramagiri, Anirudh Rao, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Julia Rice, Matt Riedemann, Marcello La Rocca, Diego M. Rodríguez, RohithKarur, Max Rossmannek, Mingi Ryu, Tharrmashastha SAPV, SamFerracin, Martin Sandberg, Hirmay Sandesara, Ritvik Sapra, Hayk Sargsyan, Aniruddha Sarkar, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Zachary Schoenfeld, Travis L. Scholten, Eddie Schoute, Joachim Schwarm, Ismael Faro Sertage, Kanav Setia, Nathan Shammah, Yunong Shi, Adenilton Silva, Andrea Simonetto, Nick Singstock, Yukio Siraichi, Iskandar Sitdikov, Seyon Sivarajah, Magnus Berg Sletfjerding, John A. Smolin, Mathias Soeken, Igor Olegovich Sokolov, Igor Sokolov, SooluThomas, Starfish, Dominik Steenken, Matt Stypulkoski, Shaojun Sun, Kevin J. Sung, Hitomi Takahashi, Tanvesh Takawale, Ivano Tavernelli, Charles Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Miroslav Tomasik, Enrique de la Torre, Kenso Trabing, Matthew Treinish, TrishaPe, Davindra Tulsi, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Almudena Carrera Vazquez, Victor Villar, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Johannes Weidenfeller, Rafal Wieczorek, Jonathan A. Wildstrom, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, Steve Wood, James Wootton, Daniyar Yeralin, David Yonge-Mallo, Richard Young, Jessie Yu, Christopher Zachow, Laura Zdanski, Helena Zhang, Christa Zoufal, Zoufalc, a kapila, a matsuo, bcamorrison, brandhsn, nick bronn, brosand, chlorophyll zz, csseifms, dekel.meirom, dekelmeirom, dekool, dime10, drholmie, dtrenev, ehchen, elfrocampeador, faisaldebouni, fanizzamarco, gabrieleagl, gadial, galeinston, georgios ts, gruu, hhorii, hykavitha, jagunther, jliu45, jscott2, kanejess, klinvill, krutik2966, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sagar pahwa, rmoyard, saswati qiskit, scottkelso, sethmerkel, shaashwat, sternparky, strickroman, sumitpuri, tigerjack, toural, tsura crisaldo, vvilpas, welien, willhbang, yang.luh, yotamvakninibm, and Mantas Čepulkovskis. Qiskit: An open-source framework for quantum computing, 2019.

[30] Paul Alcorn. AMD to Fuse FPGA AI Engines Onto EPYC Processors, Arrives in 2023. `https://www.tomshardware.com/news/amd-to-fuse-fpga-ai-engines-onto-epyc-processors-arrives-in-2023`, May 2022.

[31] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, D Bucher, FJ Cabrera-Hernández, J Carballo-Franquis, A Chen, CF Chen, et al. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar*, 16, 2019.

[32] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 13–23. IEEE, 2021.

[33] Amazon.com. Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud. `https://aws.amazon.com/ec2/instance-types/f1`, 2021.

[34] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[35] David F. Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, apr 2013.

[36] Sahar Badihi, Sami Nourji, and Julia Rubin. Slicer4d: A slicing-based debugger for java. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2407–2410, 2024.

[37] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[38] E Bendersky. PyCParser C Parser and AST Generator Written in Python, 2012.

[39] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on computing*, 26(5):1411–1473, 1997.

[40] David W Binkley and Keith Brian Gallagher. Program slicing. *Advances in computers*, 43:1–50, 1996.

[41] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In David Evans, Tal Maklin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, United States of America, 2017. Association for Computing Machinery (ACM). ACM Conference on Computer and Communications Security 2017¡br/¿, CCS 2017 ; Conference date: 30-10-2017 Through 03-11-2017.

[42] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[43] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[44] Nick Brown, Maurice Jamieson, Anton Lydike, Emilien Bauer, and Tobias Grosser. Fortran performance optimisation and auto-parallelisation by leveraging mlir-based domain specific abstractions in flang. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 904–913, 2023.

[45] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129. IEEE, 2014.

[46] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[47] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. Source-level debugging for fpga high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[48] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 151–160, New York, NY, USA, 2014. Association for Computing Machinery.

[49] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods,

Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[50] Siu-On Chan, Ilias Diakonikolas, Paul Valiant, and Gregory Valiant. Optimal algorithms for testing closeness of discrete distributions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1193–1203. SIAM, 2014.

[51] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. Rfc-directed differential testing of certificate validation in ssl/tls implementations. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 859–870. IEEE, 2018.

[52] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*, pages 180–190, 2016.

[53] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH computer architecture news*, 44(3):367–379, 2016.

[54] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[55] Andrew A Chien, Allan Snavely, and Mark Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4:1987–1996, 2011.

[56] Young-Kyu Choi and Jason Cong. Hlscope: High-level performance debugging for fpga designs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–128, 2017.

[57] Frederic T Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, 2017.

[58] Rigetti Computing. Pyquil documentation. *URL http://pyquil. readthedocs. io/en/latest*, 2019.

[59] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[60] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. Smem++: A pipelined and time-multiplexed smem seeding accelerator for dna sequencing. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–206, 2018.

[61] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[62] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design Test of Computers*, 28(2):6–15, 2011.

[63] CIRCT Contributors. Circt: Circuit ir compilers and tools, 2023.

[64] LLVM Contributors. Mlir language reference, Jun 2023.

[65] Andrew W Cross, Lev S Bishop, Sarah Sheldon, Paul D Nation, and Jay M Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, 2019.

[66] John Curreri, Greg Stitt, and Alan D. George. High-level synthesis techniques for in-circuit assertion-based verification. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[67] Ian Cutress. Intel Shows Xeon Scalable Gold 6138P with Integrated FPGA, Shipping to Vendors. `https://www.anandtech.com/show/12773/intel-shows-xeon-scalable-gold-6138p-with-integrated-fpga-shipping-to-vendors`, May 2018.

[68] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[69] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1017–1032, 2021.

[70] Jack Dongarra, Laura Grigori, and Nicholas J Higham. Numerical algorithms for high-performance computational science. *Philosophical Transactions of the Royal Society A*, 378(2166):20190066, 2020.

[71] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 365–376, 2011.

[72] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. Qpmc: A model checker for quantum programs and protocols. In *International Symposium on Formal Methods*, pages 265–272. Springer, 2015.

[73] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. *AFL++: Combining Incremental Steps of Fuzzing Research*. USENIX Association, USA, 2020.

[74] Juliana Freire, D Koop, Emanuele Santos, Carlos Scheidegger, Cláudio Silva, and Vo Huy. *The Architecture of Open Source Applications*, volume 1, chapter LLVM. 01 2011.

[75] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[76] Yan Ge, Wu Wenjie, Chen Yuheng, Pan Kaisen, Lu Xudong, Zhou Zixiang, Wang Yuhan, Wang Ruocheng, and Yan Junchi. Quantum circuit synthesis and compilation optimization: Overview and prospects. *arXiv preprint arXiv:2407.00736*, 2024.

[77] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[78] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[79] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–135, 2019.

[80] Akel Hashim, Ravi K Naik, Alexis Morvan, Jean-Loup Ville, Bradley Mitchell, John Mark Kreikebaum, Marc Davis, Ethan Smith, Costin Iancu, Kevin P O'Brien, et al. Randomized compiling for scalable quantum computing on a noisy superconducting quantum processor. *arXiv preprint arXiv:2010.00215*, 2020.

[81] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394, 2018.

[82] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[83] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, pages 31–37, 2016.

[84] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 194–203. IEEE, 2017.

[85] He-Liang Huang, Dachao Wu, Daojin Fan, and Xiaobo Zhu. Superconducting quantum computing: a review. *Science China Information Sciences*, 63:1–32, 2020.

[86] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[87] Intel. Dense linear algebra. `https://github.com/oneapi-src/oneAPI-samples/tree/6901f7203b549a651911fec694ffefad82ed0b35/DirectProgramming/C\%2B\%2BSYCL/DenseLinearAlgebra`, 2021.

[88] Intel. Dpc++ reference. `https://oneapi-src.github.io/DPCPP_Reference/`, 2021.

[89] Intel. Devcloud. `https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html`, 2022.

[90] Intel. FPGA Optimization Guide for Intel® oneAPI Toolkits - Shannonization to Improve FMAX/II. `https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/shannonization-to-improve-fmax-ii.html`, 2022.

[91] Intel. Fpga optimization guide for intel® oneapi toolkits - transfer loop-carried dependency to local memory. `https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/transfer-loop-carried-dependency-to-local-memory.html`, 2022.

[92] Intel. FPGA Optimization Guide for Intel® oneAPI Toolkits - Unroll Loops. `https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/unroll-loops.html`, 2022.

[93] Intel. Intel® Arria® 10 GX FPGA Overview. `https://www.intel.com/content/www/us/en/products/details/fpga/arria/10/gx/products.html`, 2022.

[94] Intel. Intel® Stratix® 10 GX FPGA Overview. `https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html`, 2022.

[95] Ilan Iwumbwe, Benny Zong Liu, and John Wickerson. Qutefuzz: Fuzzing quantum compilers using randomly generated circuits with control flow and subcircuits. *PLanQC*, 2025.

[96] Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. Compiling onnx neural network models using mlir. *arXiv preprint arXiv:2008.08272*, 2020.

[97] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[98] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 556–566, 2019.

[99] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[100] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. Hddr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 16–22, 2018.

[101] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[102] Andrey Kolmogorov. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 4:83–91, 1933.

[103] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.*, 14(4), sep 2021.

[104] Ryan LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[105] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[106] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[107] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[108] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[109] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. Gklee: concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 215–224, 2012.

[110] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 1001–1014, 2019.

[111] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Proq: Projection-based runtime assertions for debugging on a quantum computer, 2020.

[112] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 809–814, New York, NY, USA, 2018. Association for Computing Machinery.

[113] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[114] Ben Limpanukorn, Jiyuan Wang, Hong Jin Kang, Eric Zitong Zhou, and Miryung Kim. Fuzzing mlir by synthesizing custom mutations. *arXiv preprint arXiv:2404.16947*, 2024.

[115] Christian Lindig. Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12, 2005.

[116] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, Volume 2, pages 530–543, 2023.

[117] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 657–669, 2023.

[118] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–26, 2022.

[119] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security*, 18:1237–1251, 2023.

[120] Peixun Long and Jianjun Zhao. Testing multi-subroutine quantum programs: From unit testing to integration testing. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–61, 2024.

[121] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 248–260, 2023.

[122] Alexander Magyari and Yuhua Chen. Review of state-of-the-art fpga applications in iot networks. *Sensors*, 22(19):7496, 2022.

[123] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[124] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[125] Enaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. Muskit: A mutation analysis tool for quantum software testing.

[126] Xiao Mi, Pedram Roushan, Chris Quintana, Salvatore Mandra, Jeffrey Marshall, Charles Neill, Frank Arute, Kunal Arya, Juan Atalaya, Ryan Babbush, Joseph C. Bardin, Rami Barends, Andreas Bengtsson, Sergio Boixo, Alexandre Bourassa, Michael Broughton, Bob B. Buckley, David A. Buell, Brian Burkett, Nicholas Bushnell, Zijun Chen, Benjamin Chiaro, Roberto Collins, William Courtney, Sean Demura, Alan R. Derk, Andrew Dunsworth, Daniel Eppens, Catherine Erickson, Edward Farhi,

Austin G. Fowler, Brooks Foxen, Craig Gidney, Marissa Giustina, Jonathan A. Gross, Matthew P. Harrigan, Sean D. Harrington, Jeremy Hilton, Alan Ho, Sabrina Hong, Trent Huang, William J. Huggins, L. B. Ioffe, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Cody Jones, Dvir Kafri, Julian Kelly, Seon Kim, Alexei Kitaev, Paul V. Klimov, Alexander N. Korotkov, Fedor Kostritsa, David Landhuis, Pavel Laptev, Erik Lucero, Orion Martin, Jarrod R. McClean, Trevor McCourt, Matt McEwen, Anthony Megrant, Kevin C. Miao, Masoud Mohseni, Wojciech Mruczkiewicz, Josh Mutus, Ofer Naaman, Matthew Neeley, Michael Newman, Murphy Yuezhen Niu, Thomas E. O'Brien, Alex Opremcak, Eric Ostby, Balint Pato, Andre Petukhov, Nicholas Redd, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vladimir Shvarts, Doug Strain, Marco Szalay, Matthew D. Trevithick, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, Igor Aleiner, Kostyantyn Kechedzhi, Vadim Smelyanskiy, and Yu Chen. Information scrambling in computationally complex quantum circuits, 2021.

[127] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151, 2006.

[128] Joshua S. Monson and Brad Hutchings. Using source-to-source compilation to instrument circuits for debug with high level synthesis. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 48–55, 2015.

[129] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[130] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[131] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[132] Aina Niemetz and Armin Biere. ddsmt: a delta debugger for the smt-lib v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*, pages 8–9, 2013.

[133] Matteo Paltenghi and Michael Pradel. Morphq: Metamorphic testing of the qiskit quantum computing platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2413–2424. IEEE, 2023.

[134] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632. IEEE, 2017.

[135] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[136] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59(11):114–122, October 2016.

[137] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[138] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021.

[139] Ruyman Reyes and Victor Lomüller. Sycl: Single-source c++ accelerator programming. In *Parallel Computing: On the Road to Exascale*, pages 673–682. IOS Press, 2016.

[140] Hongbo Rong. Programmatic control of a compiler for generating high-performance spatial hardware. *CoRR*, abs/1711.07606, 2017.

[141] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*, pages 1102–1105, 2011.

[142] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519*, 2021.

[143] Yang Shi, Mengqiao Wang, Weiping Shi, Ji-Hyun Lee, Huining Kang, and Hui Jiang. Accurate and efficient estimation of small p-values with the cross-entropy method: applications in genomic data analysis. *Bioinformatics*, 35(14):2441–2448, 2019.

[144] Yunong Shi, Xupeng Li, Runzhou Tao, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. Contract-based verification of a realistic quantum compiler. *arXiv preprint arXiv:1908.08963*, 2019.

[145] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[146] Kaitlin N Smith and Mitchell A Thornton. A quantum computational compiler and design tool for technology-specific targets. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 579–588, 2019.

[147] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378, 2021.

[148] Congxi Song, Xu Zhou, Qidi Yin, Xinglu He, Hangwei Zhang, and Kai Lu. P-fuzz: A parallel grey-box fuzzing framework. *Applied Sciences*, 9(23), 2019.

[149] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26:1–45, 2021.

[150] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 361–371, 2018.

[151] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science*, 14(1):1–20, 2020.

[152] Swamit S Tannu and Moinuddin K Qureshi. Not all qubits are created equal: a case for variability-aware policies for nisq-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 987–999, 2019.

[153] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. On the caching schemes to speed up program reduction. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–30, 2023.

[154] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. On the caching schemes to speed up program reduction. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–30, 2023.

[155] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[156] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[157] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE transactions on neural networks and learning systems*, 30(2):326–342, 2018.

[158] Joel J Wallman and Joseph Emerson. Noise tailoring for scalable quantum computation via randomized compiling. *Physical Review A*, 94(5):052325, 2016.

[159] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 881–892, 2021.

[160] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1555–1566, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society.

[161] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 999–1010, New York, NY, USA, 2020. Association for Computing Machinery.

[162] Jiyuan Wang, Fucheng Ma, and Yu Jiang. Poster: Fuzz testing of quantum program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 466–469. IEEE, 2021.

[163] Jiyuan Wang, Qian Zhang, Hongbo Rong, Guoqing Harry Xu, and Miryung Kim. Leveraging hardware probes and optimizations for accelerating fuzz testing of heterogeneous applications. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1101–1113, 2023.

[164] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM international conference on automated software engineering (ASE)*, pages 692–704. IEEE, 2021.

[165] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. Conftainter: Static taint analysis for configuration options. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1640–1651. IEEE, 2023.

[166] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 765–777, 2020.

[167] Karen Wintersperger, Florian Dommert, Thomas Ehmer, Andrey Hoursanov, Johannes Klepsch, Wolfgang Mauerer, Georg Reuber, Thomas Strohm, Ming Yin, and

Sebastian Luber. Neutral atom quantum computing hardware: performance and end-user perspective. *EPJ Quantum Technology*, 10(1):32, 2023.

[168] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

[169] Xilinx. Ultrascale architecture and product data sheet: Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf`, 2021.

[170] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 625–640, 2022.

[171] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. Pushing the limit of 1-minimality of language-agnostic program reduction. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):636–664, 2023.

[172] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[173] Mingsheng Ying and Zhengfeng Ji. Symbolic verification of quantum circuits. *arXiv preprint arXiv:2010.03032*, 2020.

[174] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2307–2324, 2020.

[175] Mohamed Zahran. Heterogeneous computing: Here to stay. *Communications of the ACM*, 60(3):42–45, 2017.

[176] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[177] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[178] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, page 1–10, New York, NY, USA, 2002. Association for Computing Machinery.

[179] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. Lpr: Large language models-aided program reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[180] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. Ppr: Pairwise program reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 338–349, 2023.

[181] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[182] Qian Zhang, Jiyuan Wang, and Miryung Kim. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 242–254, 2021.

[183] Tianyi Zhang and Miryung Kim. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 665–676. IEEE, 2017.

[184] Jianjun Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.

[185] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[186] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2018.