Systems and Debugging Supports for Hardware Designs

by

Jiacheng Ma

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Jiacheng Ma

jcma@umich.edu

ORCID iD: 0000-0001-9285-422X

To all the loves we give and receive.

或是——
下一个春天在骸骨上茁壮地生长
风里吹来故乡的歌谣

ACKNOWLEDGEMENTS

As I sit down to write these acknowledgements, I realized I was helped by a lot of people during my PhD study. From offering academic guidance to providing emotional and financial supports, these people have not only instilled in me the courage to step on this challenging path but also equipped me with the essential knowledge and skills required to navigate towards its completion.

First and foremost, I would like to thank my wonderful parents, Lie Ma and Hongping Mei, for their unwavering loves, supports, and sacrifices in every aspect of my life. Their understanding and patience have always been my backbone, even when I made the bold decision to pursue a PhD eight thousand miles away from home, or when I was a curious (or actually, stupid and trouble-making) ten-year-old who accidentally (or actually, semi-intentionally) wiped the operating system of the first computer of the family.

I would like to thank the Computer Science Engineering department of the University of Michigan, the Applications Driving Architectures (ADA) Research Center, and the Processing with Intelligent Storage and Memory (PRISM) Research Center, for their generous funding at various stages of my PhD journey.

I would like to thank Zhengwei Qi, Yaozu Dong, Yubin Xia, Binyu Zang, and Mochi Xue, who introduced me to the fabulous world of computer systems research during my undergrad, and encouraged me to pursue a PhD in this field.

I would like to thank my research collaborators and coauthors, particularly Andrew Quinn, Freddy Gabbay, Yanqiang Liu, and Majd Ganaiem, for their invaluable guidance and insightful feedback. Their expertise has significantly enriched my research experience. I share the same gratitude for Sanjay Kumar, Marcos Aguilera, Irina Calciu, and Dimin Niu, who mentored my internships at Intel, VMware, and Alibaba, respectively.

I thank my fellow PhD students and postdoc at Efeslab: Gefei Zuo, Tanvir Ahmed Khan, Kevin Loughlin, Ian Neal, Andrew Loveless, Marina Minkin, Andrew Quinn, and Shuwen Deng. It has been an honor and a privilege to be part of the team and to share an office space with these great people over the past few years. Without the wisdom and inner power of these people, I would not have survived the PhD (a.k.a., Permanent Head Damage) study.

Similarly, I would like to thank the undergraduate, master, and PhD students I mentored or worked close with during my PhD: Xiaohe Cheng, Abel Mulugeta Eneyew, Haoyang Zhang, Wentao Zhang, Yin Yuan, Yiwei Yang, Maddie Burbage, Rachel McAmis, and Theo Gregersen.

I next thank my dissertation committee—Dennis Sylvester, Scott Mahlke, George Tzimpragos, and my advisor, Baris Kasikci—for serving as committee members. Their insightful feedback and help are fundamentally important in the formation of this dissertation.

I am deeply grateful to Shibo Chen, Yunjie Pan, Shiyu Ding, Gefei Zuo, Yinlan Shao, and H[1]. These people brought a lot of fun to my life, and their wonderful friendship and mental support have been critical in maintaining my sanity above a crucial threshold throughout my PhD. I also thank other close friends—Yuwei Bao, Shuyang Cao, Xuefei Chen, Tianji Cong, Juechu Dong, Zekun Fan, Yan Ge, Yufeng Gu, Bowen Huang, Xinyi Ju, Jinyang Li, Yaxuan Li, Zun Li, Yin Lin, Yanqiang Liu, Bo Peng, Dandan Shan, Jiachen Sun, Shengpu Tang, Elisa Tsai, Yongyi Yang, Jialu Zhang, Xumiao Zhang, Yiwen Zhang, Nuda Zhang, Haizhong Zheng, Jiong Zhu, and Jiayun Zou—for the same reason.

Additionally, I would like to thank members of the "Have You Soloed Today" pilot group, especially Haoran Shi and Jianping Shen. Their special friendship, fostered 5,000 feet above the ground, has been a unique and cherished part of my life.

Finally, I would like to thank my amazing PhD advisor, Baris Kasikci, for his guidance and support throughout my PhD. Baris has been instrumental not only in teaching me how to identify good research directions and conduct cool research, but also in imparting valuable life lessons on how to be a better person. I feel incredibly fortunate and honored to have been among the first batch of students in his research group, which is an experience that has indelibly shaped my academic and personal growth.

[1]This person has chosen to remain anonymous in these acknowledgements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The development and deployment of hardware and software have traditionally been quite distinct. Software benefits from an agile development cycle, aided by a wide array of debugging tools—such as step-wise debuggers, logging frameworks, and both static and dynamic analyses—and is further simplified by its integration with multiple layers of systems—such as hypervisors, operating systems, and libraries. Unfortunately, such debugging and systems supports are less explored and usually not available in the hardware domain.

This dissertation envisions that by integrating software-like systems and debugging supports into the hardware domain, the development and deployment of hardware can be markedly improved, thereby aligning them more closely with software practices. Accordingly, this dissertation conducts preliminary explorations in designing and developing such supports tailored for different hardware designs including those based on Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). Specifically, it presents three studies and systems that demonstrate the feasibility and benefits of these supports.

First, this dissertation introduces OPTIMUS, the first hypervisor designed for shared-memory FPGA platforms. OPTIMUS incorporates both spatial and temporal multiplexing, enabling a cloud FPGA to be shared among different virtual machines in various manners. This sharing can be achieved either by partitioning the FPGA's area or by allocating specific time slots for its use, thus facilitating versatile and efficient resource utilization in a cloud environment. Moreover, OPTIMUS scales linearly and can enhance the aggregated performance of applications running in different virtual machines until the memory bandwidth of the FPGA platform reaches its limit.

Second, this dissertation conducts initial investigations into the debugging supports for FPGA-based hardware designs. It includes a comprehensive study of bugs commonly encountered in such platforms, and offers a testbed of 20 hardware bugs that can be easily reproduced in a push-bottom manner. Based on this study and the testbed, the dissertation also proposes a suite of specialized debugging tools designed to assist in the localization of these bugs.

The final part of this dissertation addresses the challenge of aging-related silent data corruptions (SDCs) that are increasingly observed in data centers. It introduces Vega, a bottom-up approach that constructs concise and effective tests for the detection of aging-related SDCs by examining the hardware's implementation details. To construct such tests, Vega identifies aging-prone signal propagation paths using a model for transistor aging, and then lifts these paths into software-executable test cases using a combination of formal methods and heuristics. Finally, Vega integrates the generated tests into applications, therefore allowing aging-related SDCs to be efficiently and effectively identified at application runtime.

CHAPTER 1

Introduction

The development cycles and deployment approaches of hardware and software have traditionally been quite distinct, reflecting the inherent differences between these two domains. Software benefits from its ability to receive post-deployment patches, resulting in an agile development cycle [59, 87, 124]. Initially, an application is developed and deployed. If any bugs emerge after the deployment, developers fix these bugs by applying patches to the deployed application. To address these bugs, developers may employ a wide variety of tools, including step-wise debuggers [255, 1], logging frameworks [2, 3, 210], and static and dynamic analyses [107, 125, 82, 242, 243, 245, 238]. In addition, software applications are usually deployed on top of layers of systems, such as hypervisors [181, 86], operating systems [4, 5, 6], language runtimes [7, 8, 9], and libraries [215, 189, 10, 141]. These systems encapsulate the intricate lower-level implementation details, thereby simplifying the development, testing, debugging, and deployment of software applications.

In contrast with the rich set of systems and debugging supports available for software applications, such supports have been relatively less explored in the hardware domain. This phenomenon can be attributed to the inherent nature of hardware designs. Traditionally, a hardware design is synthesized into gates and wires, ultimately being fabricated in a chip factory. As chip fabrication costs are typically determined by the silicon area occupied by the hardware design, hardware designs often omit systems supports in order to minimize the area usage. Moreover, because bugs that are found after fabrication are extremely costly to fix, traditional hardware development embraces extensive simulation [57, 237, 252, 277, 226, 108] and formal verification [278, 298, 163, 199, 147, 236, 235], striving to reduce the number of bugs prior to fabrication.

However, over the past few years, a noticeable trend has emerged where hardware development and deployment are increasingly aligning with their software counterparts. This shift can be attributed to various factors, including the widespread adoption of reconfigurable hardware and the growing concerns regarding transistor reliability issues. These changes underscore a growing need to enhance the current methodologies for the development and deployment of hardware designs.

The vision of this dissertation is that *by adopting and integrating software-like systems and debugging supports in the hardware domain, the development and deployment of hardware can* *be significantly improved and become more aligned to their software counterparts*. To pursue this vision, this dissertation conducts preliminary explorations into the design space of such supports for different hardware designs. It details three projects, each targeting a different aspect of this emerging field, collectively demonstrating the feasibility and benefits of introducing software-like systems and debugging supports to the hardware domain.

In the remainder of this chapter, we first discuss the reasons that cause the new trend (§1.1), and then introduce each of the three projects (§1.2). Finally, we give a roadmap that outlines the structure and content of the rest of this dissertation (§1.3).

1.1 Software-like Hardware Development and Deployment

The trend of software-like hardware development and deployment can be traced to various factors, including the emerging of reconfigurable hardware, the significant involvement of software companies in hardware development, the increasing adoption of abstractions, and the growing impact of transistor reliability issues that lead to post-deployment bugs with modern technology nodes.

The Emerging of Reconfigurable Hardware Reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), is becoming increasingly prominent in modern heterogeneous computer systems. FPGAs allow users to deploy (and redeploy) customized hardware designs, thus significantly accelerating their workloads. As the set of workloads changes over time, users can reconfigure their FPGAs into different designs, making FPGAs a cost-effective and flexible alternative to fabricating customized chips (i.e., ASICs). Consequently, cloud providers are integrating FPGAs into their data centers [66, 64] and implementing specialized hardware designs for a wide variety of workloads, such as machine learning [247, 248, 300, 293, 202, 299, 178], compression [232, 297], database operations [222, 240, 250], and networking [128, 269, 94].

With the emerging of FPGAs, hardware can now be deployed like software, as the reconfigurable nature of FPGAs empowers developers to patch hardware bugs. Thus, FPGA developers are moving towards a faster, software-like development cycle and adopting more agile development approaches that accelerate time-to-market by relaxing the traditional, extensive simulation in favor of lightweight simulation and on-FPGA testing. For example, Microsoft has adopted a software-like methodology for FPGA development, in which they perform relatively small amounts of simulation-based testing compared to traditional hardware [128].

The Increasing Involvement of Software and Cloud Companies In recent years, we have observed companies traditionally known for their software or cloud services actively engaging in hardware development. For example, Google designed TPU (tensor processing unit) [165, 166] and

VPU (video coding unit) [234] to improve the performance of specific workloads. Similarly, Alibaba designed its own processors, tailored for both data center and edge computing applications [103, 11]. These companies are adopting more aggressive and agile methodologies in terms of hardware development, debugging, and testing, drawing inspiration from their roots in software development.

The Adoption of Abstractions In order to further improve time-to-market efficiency, developers are increasingly adopting multiple levels of abstraction—including the use of high-level synthesis (HLS) and abstractions for various on-FPGA resources—to ease the development process. Typically provided by FPGA vendors, these abstractions offer simplified interfaces. For example, both Intel (Altera) and AMD (Xilinx) provide well-abstracted interfaces to their complex DMA (direct memory access) [12, 284] stacks on their FPGAs, as well as HLS supports that allow hardware development in software programming languages [152, 286]. This trend even extends to ASICs. For example, Google employs HLS to speed up the design process of their video transcoding ASICs [234].

These abstractions not only simplify hardware development but also blur the boundary between software and hardware, thereby aligning hardware development more closely with software development methodologies.

The Growing Impact of Transistor Reliability Issues While FPGA developers purposefully embrace an agile development cycle that leads to more bugs in production, ASIC developers are encountering a new challenge posed by post-fabrication errors. As semiconductor fabrication scales to smaller nodes, reliability issues such as transistor aging effects are becoming more widespread due to the reduced gate size and the increased transistor density [203, 36], eventually causing post-fabrication bugs that are difficult to detect.

Recently, data center operators have begun to observe such bugs in a small fraction of their deployed CPUs. These issues manifest as silent data corruptions (SDCs) [145, 121, 271, 122, 244, 84, 120], which are particularly problematic because they often go unnoticed until causing significant damage. Consequently, there is an increasing demand for debugging supports that can effectively detect and prevent these bugs after the hardware is deployed.

1.2 Systems and Debugging Supports for Hardware Designs

In this section, we introduce the three projects undertaken in the dissertation study. In these projects, we conduct comprehensive studies—that help us understand the nature of hardware designs, the taxonomy of hardware bugs, and the challenges encountered by hardware developers—and build a set of software-like systems and debugging tools for hardware designs.

Below, we elaborate on the aforementioned projects, organizing them into two categories: (1) how we build and improve the systems support for FPGAs (§1.2.1), and (2) how we understand hardware bugs and build tools to detect and localize these bugs (§1.2.2).

1.2.1 Building and Improving Systems for FPGAs

In the first part of this dissertation, we present our attempt to build better systems supports for hardware. Specifically, we showcase a hypervisor that we build for an emerging type of FPGA platform —namely shared-memory FPGAs—that is getting increasingly popular in the cloud.

OPTIMUS—Virtualizing a Shared-Memory FPGA Platforms Cloud providers widely deploy FPGAs as application-specific accelerators for customer use. These providers seek to multiplex their FPGAs among customers via virtualization, thereby reducing running costs. Unfortunately, most virtualization support is confined to FPGAs that expose a restrictive, host-centric programming model in which accelerators cannot issue direct memory accesses (DMAs). The host-centric model incurs high runtime overhead for workloads that exhibit pointer chasing. Thus, FPGAs are beginning to support a shared-memory programming model in which accelerators can issue DMAs. However, virtualization support for shared-memory FPGAs is limited.

This project presents OPTIMUS, the first hypervisor that supports scalable shared-memory FPGA virtualization. OPTIMUS offers both spatial multiplexing and temporal multiplexing to provide efficient and flexible sharing of each accelerator on an FPGA. To share the FPGA-CPU interconnect at a high clock frequency, OPTIMUS implements a multiplexer tree. To isolate each guest's address space, OPTIMUS introduces the technique of page table slicing as a hardware-software co-design. To support preemptive temporal multiplexing, OPTIMUS provides an accelerator preemption interface. We show that OPTIMUS supports 8 physical accelerators on a single FPGA and improves the aggregate throughput of twelve real-world benchmarks by 1.98x-7x.

1.2.2 Understanding and Finding Hardware Bugs

In the second part of this dissertation, we dive into the bugs in various hardware designs. Additionally, we develop novel debugging tools crafted to detect and localize these issues. Specifically, we study two sets of bugs. First, we focus on functional bugs on FPGA platforms that originate from errors made by hardware developers in hardware description languages (HDLs). Second, we explore silent data corruptions (SDCs) that manifest as a result of transistor aging, even within hardware that is correctly designed.

Understanding and Localizing Functional Bugs on FPGAs Hardware development and deployment are increasingly aligning with their software counterparts, thanks to the emerging of FPGAs. FPGAs allow post-deployment patches like software, enabling hardware developers to fix bugs that occur during on-chip testing and even in production. Unfortunately, FPGA programmers lack bug localization tools amenable to this rapid development cycle, since past tools mainly find bugs via simulation and verification. To develop hardware bug localization tools for a rapid development cycle, a thorough understanding of the symptoms, root causes, and fixes of hardware bugs is needed.

In this project, we first study bugs in existing FPGA designs and produce a testbed of reliably reproducible bugs. We classify the bugs according to their intrinsic properties, symptoms, and root causes. We demonstrate that many hardware bugs are comparable to software bug counterparts, and would benefit from similar techniques for bug diagnosis and repair. Based on our findings, we build a novel collection of hybrid static/dynamic program analysis and monitoring tools for debugging FPGA designs, showing that our tools enable a software-like development cycle by effectively reducing developers' manual efforts for bug localization.

Vega—Detecting Aging-Related SDCs at Application Runtime Recent advancements in semiconductor process technologies have unveiled the susceptibility of hardware circuits to reliability issues, especially those related to transistor aging. Transistor aging gradually degrades gate performance, eventually causing the hardware to behave incorrectly. Such misbehaving hardware can result in SDCs in software—a type of failure that comes without logs or exceptions, but causes miscomputing instructions, bitflips, and broken cache coherency. Alas, while design efforts can be made to mitigate transistor aging, complete elimination of this problem during design and fabrication cannot be guaranteed. This emerging challenge calls for a mechanism that not only detects potentially aged hardware in the field, but also triggers software mitigations at application runtime.

In this project, we propose Vega, a novel workflow that allows efficient detection of aging-related failures at software runtime. Vega leverages the well-studied gate-level modeling of aging effects to identify susceptible signal propagation paths that could fail due to transistor aging. It then utilizes formal verification techniques to generate short test cases that activate these paths and detect any failure within them. Vega integrates the test cases into a user application by directly fusing them together, or by packaging the test cases into a library that the application can invoke. We demonstrate our proposed techniques on the arithmetic logic unit and floating-point unit of a RISC-V CPU. We show that Vega generates effective test cases and integrates them into applications with an average of 0.8% performance overhead.

1.3 Road-map

In the subsequent chapters of this thesis, we provide details on the three projects regarding the systems and debugging supports for hardware designs. Chapter 2 is dedicated to detailing OPTIMUS, our proposed hypervisor designed for shared-memory FPGA platforms. Following this, Chapter 3 conducts a study of functional bugs in FPGAs and introduces the debugging tools developed to address these issues. In Chapter 4, we introduce Vega, a novel workflow that we have developed, aimed at detecting aging-related silent data corruptions in software runtime. Finally, we conclude and present possible future research directions in Chapter 5.

CHAPTER 2

A Hypervisor for Shared-Memory FPGA Platforms

2.1 Introduction

Field Programmable Gate Arrays (FPGAs) allow users to significantly accelerate custom workloads, including those of machine learning [247, 246, 300, 293, 250], compression [231], scientific computing [139], database operations [250, 222], and graph analytics [78, 302]. As the set of data center workloads changes over time, cloud providers can reconfigure their FPGAs into different accelerators, making FPGAs a cost-effective and flexible alternative to ASICs [251, 101].

Considering the high non-recurring engineering cost [177] of hardware design and the fact that most cloud application developers are software programmers, cloud providers such as Amazon and Microsoft configure their FPGAs into popular accelerators, which the providers then make available for customer use [66, 214].

As with other hardware devices, cloud providers desire the ability to multiplex their FPGAs among different customers via virtualization, thereby increasing resource utilization and return on investment (ROI) [176, 264]. Although multi-tenant FPGA hypervisors and operating systems exist [183, 272, 96, 273, 104, 176, 296, 126, 267, 223, 225], these solutions are restricted to FPGA platforms that expose a *host-centric* programming model, as opposed to a *shared-memory* model.

The key difference between host-centric and shared-memory FPGA programming models is whether or not accelerators can issue direct memory accesses (DMAs, via which an I/O device obtains data from system memory). In host-centric models, the host issues all DMAs via a CPU-configured DMA engine, which passes the accessed data to the necessary accelerator; the accelerators themselves cannot issue DMAs. Most FPGA manufacturers [283, 67, 260] adopt this programming model. Unfortunately, the host-centric model cannot efficiently support applications that exhibit pointer chasing (e.g., graph processing [275] and database acceleration [250]), as such applications require repeated communication between the CPU and FPGA to coordinate each DMA. In particular, the software programmer must either 1) initiate multiple data transmissions separately and sequentially, or 2) marshal the data every time before transmission, both of which hurt performance.

To overcome the performance penalties of the host-centric programming model, emerging FPGAs are alternatively exposing a lighter, more flexible shared-memory programming model [256, 153, 150, 77]. Under this new model, each accelerator can issue its own DMAs and shares an address space with a process on the CPU. The CPU is merely responsible for providing the accelerator with a pointer to its initial input data. Upon receiving the pointer, the accelerator can issue the initial and subsequent DMAs without CPU intervention. As we demonstrate in §2.2.1, the shared-memory model can outperform the host-centric model by 37%–85% in a virtualized environment.

Unfortunately, virtualizing system memory on shared-memory FPGA platforms is challenging. In particular, because both the CPU and FPGA can directly access system memory, virtualization solutions must provide consistent views to applications on the CPU and accelerators on the FPGA. For instance, if a software process updates a page's data/metadata, these changes must be immediately visible to its corresponding accelerator, and vice-versa.

Furthermore, while SR-IOV [191] (i.e., hardware-assisted IO virtualization) provides a method of isolating virtual DMAs on *PCIe* links, shared-memory platforms can expose an interface that encapsulates both a PCIe link and a UPI link (e.g., Intel HARP [150]). Thus, on such platforms, SR-IOV does not provide a comprehensive solution to virtual DMA isolation. Additionally, for the past five years, shared-memory platforms have been unable to support more than one VF per FPGA [153, 150], limiting SR-IOV's scalability on these platforms.

In this chapter, we introduce OPTIMUS, the first scalable hypervisor that virtualizes shared-memory FPGAs. Deployed by cloud providers, OPTIMUS can configure a single FPGA into well-isolated accelerators, simultaneously accelerating a variety of jobs and improving resource utilization.

OPTIMUS targets a use case in which cloud providers configure FPGAs as a set of popular accelerators for their customers (e.g., the accelerator libraries/registries of Amazon F1 [66] and others [176, 104]). Notably, OPTIMUS does *not* aim to virtualize an FPGA's reconfiguration capabilities, opting instead to schedule VMs on FPGAs pre-configured with the necessary accelerator(s). Such a model is desirable in a cloud setting, as it 1) avoids the high performance overheads—and therefore, revenue losses—of reconfiguration during accelerator context switches, and 2) still allows cloud providers to reconfigure their *physical* FPGAs as customer needs change over time.

OPTIMUS virtualizes shared-memory FPGAs via a composition of spatial multiplexing and temporal multiplexing. *Spatial multiplexing* partitions the physical FPGA into multiple accelerators that can be individually controlled by different VMs [104, 96, 126, 273, 184, 176, 267]. *Temporal multiplexing* then oversubscribes these accelerators—multiple VMs take turns running atop a fixed-configuration accelerator [272, 101]. To support temporal multiplexing, OPTIMUS offers a preemption interface for accelerator design, such that it can instruct virtual accelerators to swap their state to/from system memory on a context switch.

OPTIMUS is implemented atop Intel Skylake HARP [150], but its design can be generalized to different shared-memory FPGA platforms. OPTIMUS efficiently overcomes the DMA isolation limitations of existing shared-memory FPGAs with a virtualization technique called *page table slicing*. Page table slicing is inspired by prior software-only techniques on isolating DMAs [262, 280], but is instead implemented as a generic hardware-software co-design to provide virtualization independent of specific accelerator configurations. Using page table slicing, OPTIMUS configures the FPGA to include a *hardware monitor*, which assists in partitioning a single IO page table among all guests without incurring IO page table context switching overhead.

OPTIMUS spatially multiplexes up to eight unique physical accelerators and improves the aggregate throughput of twelve real-world benchmark workloads by 1.98x-7x. Additionally, OPTIMUS's hardware monitor occupies less than 7% of FPGA resources. Finally, OPTIMUS stringently enforces real-time bandwidth sharing policies for both spatially- and temporally-multiplexed accelerators.

In summary, this chapter makes the following contributions:

- We design OPTIMUS, the first scalable hypervisor to offer virtualization support for shared-memory FPGAs, using both spatial multiplexing and temporal multiplexing to provide efficient, fair, and flexible sharing of individual accelerators on an FPGA.

- We introduce a hardware-software co-design for IO virtualization—*page table slicing*—that isolates each virtual accelerator's DMAs via a combination of hypervisor and on-FPGA support.

- We provide an interface to support the inclusion of preemption capabilities in accelerator design.

A version of this work was previously published in the proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20) [207].

2.2 Background

Field Programmable Gate Arrays (FPGAs) are chips that can be configured (and reconfigured) into custom circuits (e.g., accelerators). FPGA developers often use hardware description languages such as Verilog [261] and VHDL [90] to describe their circuit designs. A *synthesizer* program translates these designs into native FPGA *bitstreams* (i.e., binaries).

In the rest of this section, we give detailed background on FPGA programming models as well as FPGA virtualization. We focus on FPGAs designed to be used as accelerators.

2.2.1 FPGA Programming Models

The software interface (i.e., programming model) for an FPGA is determined via a reserved portion of the FPGA called a *shell*, often provided by the manufacturer. The shell is responsible for sending, receiving, and processing I/O packets (such as those from the CPU, network, system memory, etc.), and generally presents one of two programming models to system software: *host-centric* or *shared-memory*. In both of these models, the shell exposes a memory-mapped IO (MMIO) control plane for software to manage the accelerator. The key difference between these models is whether accelerators can issue their own direct memory accesses (DMAs).

In the more widespread host-centric model, the accelerators are unaware of the system memory map and thus cannot issue DMAs. Instead, the CPU configures a DMA engine to transfer data from system memory to the accelerators. The host-centric model yields simpler hardware, as accelerator architects need not add DMA logic to their designs, instead relying on software programmers to manage DMAs.

However, the host-centric model incurs the latency of repeated communication between the CPU and accelerators for applications that exhibit pointer chasing. Specifically, the CPU must repeatedly configure the DMA engine to fetch new data for each accelerator. While scatter-gather DMA engines [283] can alleviate the penalty of certain non-contiguous access patterns (e.g., those where the sequence of DMA addresses is known prior to accelerator execution), they cannot alleviate the penalty of pointer chasing, as the sequence of DMA addresses is determined during accelerator execution.

In the emerging shared-memory model (e.g., that of Intel HARP [150]), each accelerator is cognizant of the system memory map and can issue its own DMAs. Therefore, shared-memory accelerators can engage in pointer chasing without interrupting the host to issue subsequent DMAs, avoiding the latency of host-centric platforms for such applications.

We use a graph processing application that uses the single source shortest path (SSSP) algorithm [302] to demonstrate the benefits of the shared-memory programming model. The algorithm needs to iteratively access a non-contiguous set of vertices and edges, thereby emulating the behavior of pointer chasing in the absence of scatter-gather DMA support (i.e., on our evaluation platform).

We implement this algorithm on Intel HARP, under the original shared-memory interface and a host-centric interface. Figure 2.1 shows the processing time of the algorithm on a set of graphs with 800K vertices and an increasing number of edges. "Host-Centric+Config" indicates that the host-centric FPGA's DMA engine has been configured to fetch each individual data segment, while "Host-Centric+Copy" indicates that the host copies all data segments to a contiguous buffer before invoking the DMA engine. As shown, the shared-memory implementation is 17%–60% faster than that of the host-centric. The benefit of the shared-memory model is even more striking in a virtualized environment (37%–85% faster execution), where control plane operations become more

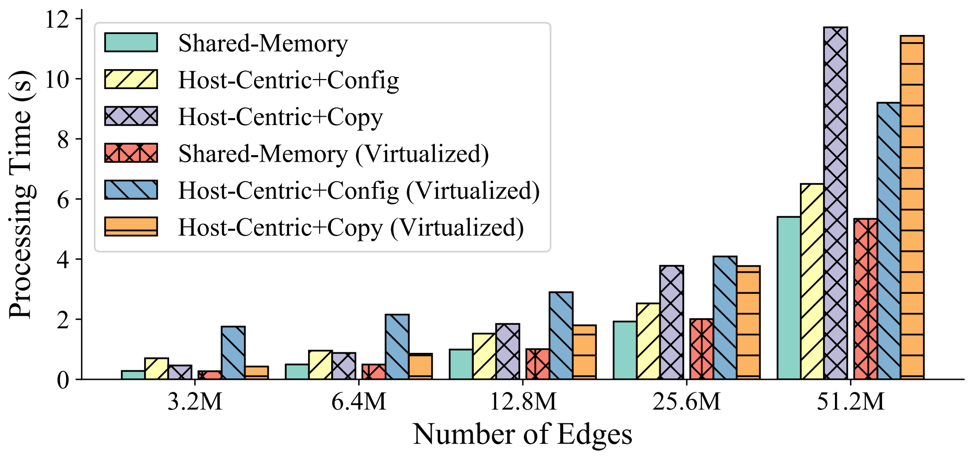


Figure 2.1: Graph processing time using the SSSP algorithm.

expensive due to hypervisor trap-and-emulate. In sum, the DMA capabilities of shared-memory accelerators allow workloads to engage in pointer chasing without CPU involvement, reducing communication costs and improving performance.

2.2.2 FPGA Virtualization

The accelerators on an FPGA can be multiplexed spatially [176, 104, 273, 223, 225, 96, 267] and temporally [176, 272, 290, 104, 223, 225]. Spatial multiplexing allows different accelerator configurations to simultaneously occupy the same FPGA. Temporal multiplexing allows each individual accelerator configuration on an FPGA to be shared by multiple VMs. Temporal multiplexing can either be non-preemptive (i.e., run-to-completion) [272] or preemptive (i.e., pause-and-resume) [176].

To virtualize an FPGA, each virtual accelerator's on-FPGA resources as well as IO channels must be isolated [176]. The FPGA synthesizer handles most on-FPGA resource isolation. Specifically, the synthesizer ensures that each accelerator on a spatially-multiplexed FPGA is provisioned a distinct portion of device resources. If a physical accelerator is additionally overprovisioned via preemptive temporal multiplexing, accelerator designs must include support for saving and restoring their execution states upon preemption.

As for IO channels, FPGAs utilize both an MMIO control plane and a DMA data plane. Since software initiates all MMIO accesses in both the host-centric and shared-memory programming models, a hypervisor can easily virtualize guest access to MMIO registers via trap-and-emulate. In the host-centric model, software also initiates all DMAs, meaning host-centric DMAs can also be virtualized via trap-and-emulate [104] or paravirtualization [272].

However, in the shared-memory model, accelerators issue their own DMAs without software intervention, posing a problem for DMA virtualization. The traditional virtualization solution for

DMA-capable IO devices has been a combination of SR-IOV [191] and PASID [154]. With SR-IOV, the IO memory management unit (IOMMU) provides a unique IO page table for each virtual device, thereby allowing the hypervisor to install unique address mappings that are enforced by the IOMMU at the time of DMA for each guest. With PASID, the IOMMU uses a CPU page table to translate DMAs, thereby allowing IO devices to directly access a process's address space. Each DMA is tagged with a process identifier, which the CPU uses to select the correct page table.

Unfortunately, the applicability of these techniques to shared-memory platforms is currently limited for two reasons. First, SR-IOV and PASID only virtualize PCIe links. Thus, on shared-memory platforms that expose both a UPI link and a PCIe link (e.g., Intel HARP [150]), SR-IOV and PASID cannot provide complete virtualization.

Second, the scalability of SR-IOV implementations in shared-memory FPGAs is severely limited. Although the SR-IOV standard supports thousands of VFs [191], shared-memory FPGAs have only supported one VF for the past five years [153, 150]. Because SR-IOV implementations are proprietary, our knowledge of the factors restricting scalability in shared-memory FPGAs is limited. However, certain shared-memory platforms such as Intel HARP [150] currently implement both the SR-IOV and the (related) IOMMU as soft IP in the FPGA shell, restricting scalability as compared to that of more resource-efficient hard IP implementations.

2.3 Goals and Challenges

OPTIMUS targets a use case in which cloud providers configure FPGAs as a set of popular accelerators for their customers, avoiding the penalty of virtual accelerator reconfiguration in favor of increased uptime [66, 176, 104]. To enable efficient and flexible sharing of accelerators on FPGAs, OPTIMUS utilizes spatial multiplexing [176, 104, 273, 223, 225, 96, 267] to partition an FPGA into a fixed set of accelerators, and temporal multiplexing [176, 272, 290, 104, 223, 225] to overprovision each of these accelerators. Because OPTIMUS novelly virtualizes shared-memory FPGAs, OPTIMUS tailors the goals of FPGA virtualization to shared-memory platforms as follows:

Programmability Unlike virtualization solutions for host-centric platforms [104, 273, 223, 96, 272, 176, 267, 225], OPTIMUS aims to share a unified virtual memory address space between software and hardware, similar to the original HARP interface [150]. However, programmability implies that cloud application developers should not have to deal with low-level platform details such as memory isolation, and should instead rely on straightforward memory abstractions of unified address spaces [200, 38, 239, 301]. Therefore, OPTIMUS must provide user-friendly abstractions for its unified CPU and FPGA address spaces to achieve programmability.

Isolation While host-centric FPGA virtualization solutions focus on the isolation of on-FPGA DRAM [104, 273, 223, 96, 272, 176, 267], OPTIMUS must consider the isolation of system memory in the presence of accelerator DMAs. Given limited support for hardware-assisted virtualization, OPTIMUS must provide strong DMA isolation within a single IOMMU address space.

We note that OPTIMUS assumes the synthesizer places each physical accelerator on isolated pieces of the FPGA fabric. Additionally, OPTIMUS does not consider side channels, which are an interesting direction for future work.

Scalability As the number of accelerators on an FPGA increases, the FPGA's *multiplexers* (i.e., the hardware components that propagate signals between the set of accelerators and the singular system interconnect) must process data from a greater number of sources within timing constraints (e.g., a given number of cycles). At some point, a flat multiplexer arrangement physically cannot process all the signals under timing constraints; a multiplexer tree hierarchy must instead be used [176]. Given that OPTIMUS targets hardware operating at higher frequencies than state-of-the-art solutions—thereby placing tighter constraints on timing—OPTIMUS must provide a multiplexer tree by default to achieve scalability.

Efficiency OPTIMUS must have low virtualization overhead to provide sufficient performance to each VM. Specifically, the sum of each virtual accelerator's bandwidth must be as close as possible to the FPGA's total bandwidth. Furthermore, the latency added by hypervisor and hardware monitor execution must be minimized. Given the frequent occurrence of DMAs as compared to MMIOs, the primary challenge is ensuring that DMAs occur with minimal overhead. Unfortunately, traditionally-efficient DMA isolation methods such as SR-IOV and PASID do not currently provide a comprehensive and scalable DMA virtualization solution. Therefore, OPTIMUS must synthesize virtualization support into the FPGA to achieve the efficiency of hardware-assisted virtualization.

Fairness In line with prior work [176], OPTIMUS aims to ensure that each accelerator receives a fair share of the FPGA's total bandwidth. Given N spatially multiplexed physical accelerators, each accelerator must receive at least $1/N$ of the total real-time bandwidth when transmitting data. In temporal multiplexing, the physical accelerator must be assigned to each virtual accelerator for the same amount of time.

2.4 Design

OPTIMUS follows a mediated pass-through [262] architecture in which control plane operations are trapped by the hypervisor, while data plane operations bypass the hypervisor. Figure 2.2 shows

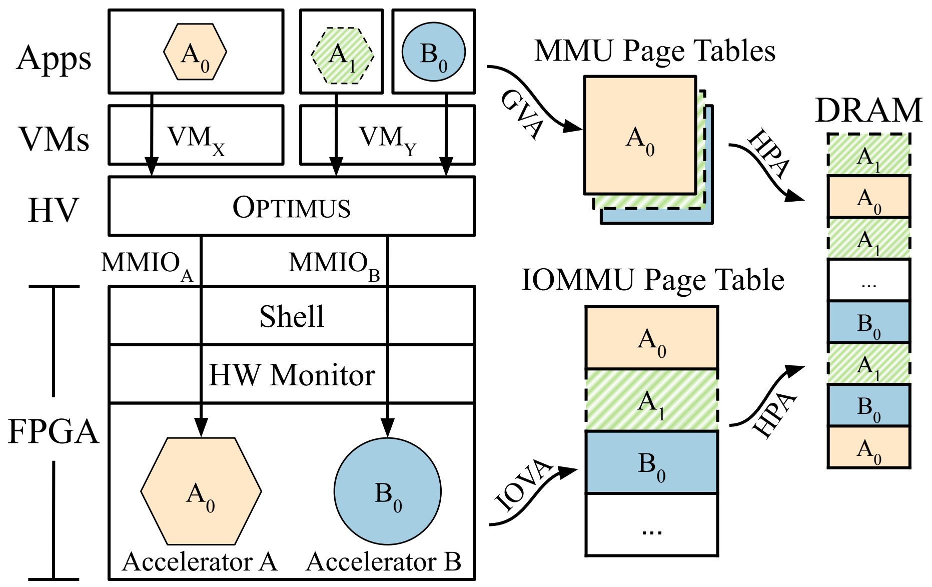


Figure 2.2: OPTIMUS design overview, shown with two physical accelerators for brevity. OPTIMUS spatially multiplexes a shared-memory FPGA as physical accelerators (*A* and *B*), and temporally multiplexes physical accelerators as virtual accelerators (A_0, A_1, and B_0).

the high-level architecture of OPTIMUS, limited to two accelerators for brevity. OPTIMUS uses the FPGA's shell to configure a shared-memory FPGA as a fixed set of *physical accelerators* (*A* and *B*), thereby offering spatial multiplexing. OPTIMUS can additionally expand its virtualization scalability by temporally sharing a physical accelerator among several *virtual accelerators* (A_0 and A_1). For example, in Figure 2.2, virtual accelerator A_0 is scheduled on physical accelerator *A* (meaning *A* holds A_0's execution state), while OPTIMUS stores virtual accelerator A_1's execution state in DRAM until re-scheduling A_1 on physical accelerator *A*.

MMIO Control Plane OPTIMUS traps all virtual accelerator control plane operations (MMIOs) to redirect the operations to the correct physical location. For scheduled virtual accelerators (A_0 and B_0), OPTIMUS adds an offset to the trapped MMIOs in order to address the appropriate physical accelerator, forwarding the adjusted MMIOs to the FPGA. The hardware monitor then routes each MMIO to the appropriate physical accelerator (*A* or *B*) based on the offset MMIO address. For a queued virtual accelerator (A_1), OPTIMUS postpones the MMIO access until the virtual accelerator is re-scheduled on a physical accelerator. The details of MMIO operations in temporal multiplexing will be discussed in §2.4.2.

DMA Data Plane Guest applications and their accelerators interact with DRAM using virtual addresses, which are translated to host physical addresses by the MMU and IOMMU respectively.

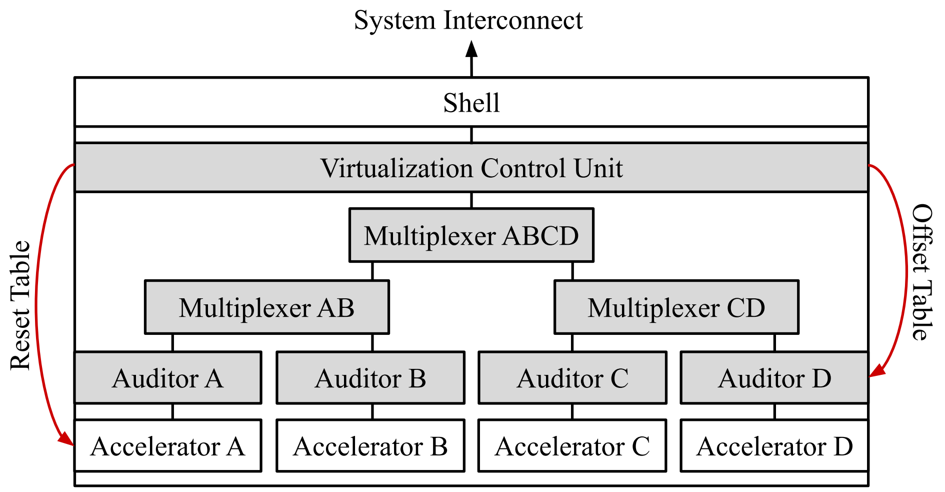


Figure 2.3: An example OPTIMUS FPGA architecture, with the hardware monitor components shaded in gray. A two-level binary multiplexer tree is shown for brevity, but the multiplexer tree arrangement is configurable.

However, the IO virtual addresses (IOVAs) used for virtual DMAs are offset versions of guest virtual addresses (GVAs). Although the CPU can provision a separate hardware page table in the MMU (i.e., an extended page table) for each application, only a single hardware page table is available to the FPGA in the IOMMU. Thus, OPTIMUS must partition the single IO virtual address space among virtual accelerators using a technique called *page table slicing*, where each virtual accelerator's DMA region begins at a unique offset within the IO virtual address space. OPTIMUS stores an offset table within the hardware monitor to translate from guest virtual addresses to IO virtual addresses during DMAs.

2.4.1 Hardware Monitor

Figure 2.3 shows the FPGA configuration to support OPTIMUS. The manufacturer provides the shell, which serves as the IO interface for the FPGA. OPTIMUS uses the shell to load the cloud provider's desired accelerator configurations onto the FPGA. OPTIMUS also includes a hardware monitor (shown in gray) on the FPGA.

Virtualization Control Unit OPTIMUS uses the virtualization control unit (VCU) to configure the runtime behavior of the hardware monitor. Specifically, VCU presents an accelerator management interface to allow OPTIMUS to configure the offset and reset tables. The offset table stores offsets between guest virtual addresses and IO virtual addresses for each accelerator (necessary to support page table slicing). The reset table is used to specify the reset signal for each accelerator, thus enabling OPTIMUS to reset individual accelerators to clear state for isolation purposes on a VM context switch.

OPTIMUS reserves a special region of MMIO for communication with VCU. If the incoming packets fall in this range, the virtual control unit intercepts the packets to configure the hardware monitor. Otherwise, VCU forwards the packets to the multiplexer tree.

Multiplexer Tree The multiplexer tree is responsible for propagating input packets from the shell to each accelerator, and transmitting output packets from each accelerator to the shell. Each multiplexer in the multiplexer tree operates on a round robin scheduling policy, thereby ensuring equal bandwidth for each accelerator on the same path through the multiplexer tree (and thus, fair real-time bandwidth sharing as mentioned in §2.3). However, if cloud providers seek to provide greater bandwidth to some accelerator A, the multiplexer tree can be configured to place fewer accelerators under the multiplexers on A's path.

Auditors Unlike AXI or Avalon interconnects [281, 157], the multiplexer tree does not make routing decisions based on the accessed address. Instead, the multiplexer tree propagates packets to a set of *auditors* (one per physical accelerator), where each auditor determines whether incoming packets are intended for its associated accelerator. This lazy packet routing (i.e., waiting until the packets arrive at the auditor to make routing decisions) results in simpler circuitry than eager packet routing (i.e., including routing logic within the multiplexer tree).

If the incoming packet is an MMIO, the auditor checks that the MMIO offset falls within the accelerator's MMIO range. If so, the auditor forwards the packet to its associated accelerator. If not, the packet is discarded.

If the incoming packet contains DMA data, the auditor must determine if the packet is a response to a DMA that the accelerator initiated. When an accelerator wishes to perform a DMA, the auditor tags the outgoing packet with an accelerator ID, which is preserved in the response packet. Thus, an auditor can verify if a DMA packet is intended for its accelerator by checking the packet's accelerator ID field. If so, the packet is forwarded to the accelerator. If not, the packet is discarded.

Page Table Slicing For simplicity, guest applications and their virtual accelerators would both access memory using guest virtual addresses, which would ultimately be translated to host physical addresses by the MMU and IOMMU respectively. However, given the limitation of a single IO virtual address space, the guest virtual addresses of different applications would conflict if used as keys in the IO page table. To isolate guest memory, OPTIMUS introduces a hardware-software co-design called page table slicing, which adapts prior software-only techniques for virtualizing GPUs [262] and wireless NICs [280].

Page table slicing configures the auditors with a *linear* address mapping policy, where guest virtual addresses (GVAs) map to IO virtual addresses (IOVAs). OPTIMUS allows each accelerator to access a contiguous *DMA memory* range $[g, g+p)$ in the application's address space. It also divides IOVAs into several p-sized partitions, and assigns each partition to a unique (virtual) accelerator. For a given IOVA partition $[i, i+p)$, OPTIMUS stores the offset value $(i-g)$ in the corresponding accelerator's entry in the offset table. Afterward, the accelerator's auditor can convert between IOVAs and GVAs during DMAs within a single cycle, ensuring efficient memory isolation. In the presence of temporal multiplexing (i.e., oversubscription of individual accelerators), OPTIMUS updates the physical accelerator's offset table entry with the newly-scheduled virtual accelerator's offset entry.

We consider page table slicing as a lightweight isolation method which is complementary to SR-IOV. Specifically, even if SR-IOV scalability increases for future shared-memory FPGAs, page table slicing would allow for nested virtualization on SR-IOV enabled devices; a cloud provider could use SR-IOV to provide a "vFPGA" to a VM acting as a nested hypervisor. The nested hypervisor could then use page table slicing to share this vFPGA among its own guests.

Shadow Paging An important goal of OPTIMUS is to share a contiguous range of virtual memory between software and hardware, which requires the IOMMU (together with page table slicing) to directly map GVAs to HPAs. Since the IOMMU does not support nested paging, OPTIMUS maintains a shadow page table for each accelerator.

2.4.2 Preemption Interface

While spatial multiplexing allows different accelerators to run on the same FPGA, OPTIMUS uses temporal multiplexing to share a fixed accelerator configuration among different VMs, with each VM's virtual accelerator occupying the physical accelerator for a short time-slice. OPTIMUS must be able to preempt acceleration jobs to provide fair temporal multiplexing, and therefore exposes a *preemption interface* similar to that of AmorphOS [176].

A preemption-capable accelerator should implement a set of *control registers* which serves two purposes: 1) saving and restoring internal execution states, and 2) starting, preempting, and resuming acceleration jobs. Control registers are privileged resources, thus should not be accessible by virtual machines directly. The hypervisor traps and emulates accesses to control registers, and hides the hardware status of the physical accelerator. Registers besides the control registers are called *application registers*. Accesses to application registers are postponed until the virtual accelerator is scheduled. Specifically, if the register does not have side effects (i.e., read/write to the register is idempotent), the hypervisor can cache the register's value in software and synchronize the cache and the physical register while scheduling.

During virtual accelerator initialization, the accelerator informs OPTIMUS how much memory is needed to store internal execution states. OPTIMUS then allocates a memory buffer for the states and informs the physical accelerator of the buffer's base address via the control registers.

When OPTIMUS wishes to schedule a virtual accelerator on a physical accelerator, OPTIMUS reads the current job status from the physical accelerator. If the physical accelerator is occupied, OPTIMUS sends a preempt command, causing the physical accelerator to write the virtual accelerator's execution state to the system memory buffer. Once all in-flight transactions have been processed, the accelerator notifies OPTIMUS that context has been successfully saved and a new job may be scheduled, as in prior work [176]. If an accelerator fails to cede control, OPTIMUS can forcibly reset the accelerator after a configurable timeout period.

Later, when OPTIMUS re-schedules the original virtual accelerator job on the physical accelerator, it issues a resume command that instructs the physical accelerator to load execution state from its memory buffer and continue execution.

OPTIMUS's decision to leave the implementation of preemption to accelerator designers is a complexity-performance trade-off. On one hand, designers using OPTIMUS must reason about the state to save upon preemption, in contrast to automatic mechanisms such as Cascade [241]. On the other hand, designers using OPTIMUS can identify the minimal amount of state to save. For example, when preempting a linked-list walker, saving the address of the next node can be sufficient. In contrast, Cascade conservatively requires all latches to be saved. This results in a more complex circuit, consuming more resources, inhibiting a circuit's ability to scale to higher frequencies, and ultimately hurting performance. Thus, given OPTIMUS's performance and scalability goals, OPTIMUS relies on accelerator designers to implement the preemption interface.

2.4.3 Userspace API

Because native platform APIs can be complex [155], OPTIMUS offers a simplified API for software application developers. OPTIMUS provides a separate implementation of the same simplified API to accelerator developers for use in Verilog simulations.

From the guest's perspective, each accelerator is a PCIe device. OPTIMUS offers a customized driver and a userspace library that work in tandem to allow for application-level programming of accelerators. The driver is responsible for initializing the virtual accelerator, including mapping MMIO regions to userspace and registering DMA memory with the hypervisor. The userspace library allows the programmer to easily connect to and disconnect from a virtual accelerator, reset the accelerator, program the virtual accelerator through its MMIO region, and manage DMA memory.

2.5 Implementation

OPTIMUS is implemented atop the Intel HARP shared-memory FPGA platform [150] using Intel's Core Cache Interface (CCI-P) [149]; however, OPTIMUS's design can be applied to any shared-memory FPGA platform with IOMMU support (which is necessary to implement page table slicing). OPTIMUS is implemented as a kernel module in 3,199 lines of C code, using the *vfio-mdev* [164] framework for device mediation and KVM [182] for CPU and memory virtualization. The guest FPGA driver and user API library are an additional 2,033 lines of C code, not including a ported memory allocation library [198] used to help manage DMA regions for accelerators. The Verilog implementation of the hardware monitor relies on Intel's open-source multiplexer (MUX) module [159], which adds 1,237 lines of code. Altogether, the hardware monitor occupies less than 7% of on-FPGA configurable resources.

FPGA Interface HARP's shell provides a request/response interface called CCI-P for memory access [149], which encapsulates PCIe and UPI transactions. In order to access CPU memory, an accelerator sends a request packet and then waits for a corresponding response packet. While waiting, the accelerator may send out other requests to saturate the bandwidth.

MMIO Slicing The MMIO address space of OPTIMUS consists of three portions. The first portion of the MMIO space is reserved for the HARP shell. The next 4 KB is reserved for the virtualization control unit's accelerator management interface, via which the hypervisor can configure the hardware monitor (e.g., the offset and reset tables) and obtain the FPGA configuration information (e.g., the number of physical accelerators on the device and whether or not the configuration is compatible with OPTIMUS). Finally, each physical accelerator receives a 4 KB page for its individual MMIO state, with isolation enforced by the accelerator's auditor.

Guest-MMIO Layout From a guest's perspective, a virtual accelerator is a PCIe device. PCIe BAR0 points to the accelerator MMIO space, and PCIe BAR2 points to the hypervisor MMIO space (used to communicate with the hypervisor).

Page Table Slicing By default, OPTIMUS uses a 64 GB slice of the 48-bit IO virtual address space for each virtual accelerator. However, this can be increased on systems where more than 64 GB of RAM is needed per virtual accelerator.

OPTIMUS's guest library uses the `mmap()` system call with the `MAP_NORESERVE` flag to reserve a 64 GB slice without allocating physical memory or swap. OPTIMUS writes the base address of each slice to a register in BAR2 (the hypervisor MMIO space). The slicing offset is calculated based on the value stored in this register.

Shadow Paging For prototype simplicity, OPTIMUS currently features a hypercall-style shadow paging mechanism, reserving a register in the hypervisor MMIO space. During the initialization of each accelerator, OPTIMUS allocates a 2 MB page, and initializes the IOPT entries of the accelerator to map to the physical address of the page. When a guest wants to make a page FPGA-accessible, it uses this register to notify the hypervisor of the GVA and GPA for the page. The hypervisor then checks page permissions, calculates the correct IOVA and HPA, pins the HPA in memory, and inserts the IOVA$\rightarrow$HPA mapping into the IO page table.

Multiplexer Tree Hierarchy OPTIMUS uses a three-level binary tree which supports up to 8 physical accelerators. We experimented with different hierarchies for the multiplexer tree (e.g., more layers and more nodes per layer); however, for some benchmarks, the synthesizer was unable to synthesize greater than eight accelerator instances on the FPGA without lowering the multiplexer tree frequency below 400 MHz, which is necessary to fully utilize the memory bandwidth. Hence, we limited the tree's support to eight physical accelerators.

AMORPHOS [176]—a prior FPGA virtualization solution—uses a flat multiplexer to avoid the complexity and latency of a multiplexer-tree when there are eight or fewer accelerators, and uses a layered multiplexer-tree when there are greater than eight accelerators. However, in OPTIMUS, a flat multiplexer is not feasible even with a smaller number of accelerators, as it prevents OPTIMUS from multiplexing the accelerators at a high frequency (400 MHz).

Huge Pages In line with prior work [81, 45, 44, 39, 193, 194, 212], OPTIMUS uses huge pages to avoid IOTLB (IO translation lookaside buffer) thrashing and improve DMA performance. To the best of our knowledge, on the Intel HARP platform, the IOTLB for both 4 KB pages and 2 MB pages can only store 512 IOVA to HPA mappings. Only using 4 KB pages may cause frequent IOTLB misses, which hurts performance on HARP. OPTIMUS uses 2 MB huge pages for DMA memory, thereby allowing the IOTLB to cache 2 MB $*$ 512 $=$ 1 GB worth of mappings.

We do not see 2 MB pages as a significant drawback for three reasons. First, hypervisors are already unable to oversubscribe memory in the presence of pass-through or SR-IOV-enabled devices; the device-accessible memory pages must be pinned due to the IOMMU's inability to handle page faults. Second, as opposed to pass-through or SR-IOV, OPTIMUS only pins FPGA-accessible pages once they are allocated by the guest. Third, data center servers are often equipped with hundreds of gigabytes of memory; therefore, 2 MB pages are relatively small.

IOTLB Conflict Mitigation When using our original page table slicing technique (in which each 64 GB slice is laid out contiguously in the IO virtual address space), we discovered that IOTLB mappings for different virtual accelerators were frequently evicting each other, hurting system performance.

While the exact eviction policy for the IOTLB is unknown, we believe the problem stems from a conflict in the set indices of IOVAs for different virtual accelerators. To the best of our knowledge, when the page size is 2 MB, the IOTLB uses 9 bits after the 21-bit huge page offset as the set index (bits 21-29). We believe each set consists of a single entry. Thus, if a virtual accelerator accesses a virtual page with the same set index as another virtual accelerator's page, an IOTLB conflict will occur. More precisely, a given page p_1 will conflict with any page p_2 where $p_1 \equiv p_2 \mod 2^9$.

To work around this problem in software (given the IOTLB could not be altered), we added an extra 128 MB of address space between each 64 GB IOVA slice to offset the set indices of different virtual accelerator pages. Because OPTIMUS supports eight physical accelerators and the IOTLB can address 1 GB of memory without conflicts, OPTIMUS divides this 1 GB of memory evenly among the accelerators, yielding 128 MB per accelerator. Thus, each virtual accelerator's working set must exceed 128 MB before IOTLB conflicts potentially occur among accelerators. If sequential accesses are performed, IOTLB misses are rare, regardless of the working set size.

Tiling and Partial Reconfiguration Like other FPGAs [229, 101, 66], HARP FPGAs can be reconfigured at tile granularity (i.e., a manufacturer-defined portion of the fabric). The reconfiguration of an individual tile is known as partial reconfiguration. However, HARP only provides a single tile, and therefore would require re-flashing all spatially-multiplexed accelerators to reconfigure an individual accelerator. As such, OPTIMUS does not support partial reconfiguration.

Temporal Multiplexing Interface For flexible memory management, each guest application allocates a buffer in host DRAM for storing accelerator state upon preemption.

Time Slice in Temporal Multiplexing The time slice used for temporal multiplexing is configurable; however the default value is 10 ms. A 10 ms time slice is possible because OPTIMUS does not reconfigure the FPGA upon preemption, since the temporally-multiplexed accelerators on a given physical accelerator share the same configuration. If partial reconfiguration support is added in the future, the time slice would need to be increased to allow for sufficient time to reconfigure individual tiles.

Temporal Multiplexing Scheduling OPTIMUS uses unweighted round-robin (i.e., equal time slices) as the default scheduling algorithm. However, OPTIMUS also implements a scheduler with weighted time slices and a priority-based scheduler.

2.6 Evaluation

In this section, we evaluate our prototype implementation of OPTIMUS and answer the following questions:

Efficiency What is the overhead of the hardware monitor in terms of FPGA resource utilization? To what extent does spatial multiplexing improve FPGA resource utilization (§2.6.2)? How much virtualization overhead does OPTIMUS incur compared to pass-through (i.e., direct assignment) (§2.6.3)? How does the use of huge pages influence memory throughput and latency? (§2.6.5)

Scalability How does OPTIMUS scale with respect to the number of acceleration jobs concurrently executing on the FPGA (§2.6.4)? How does OPTIMUS scale with respect to the oversubscription factor of each accelerator (i.e., the number of virtual accelerators per physical accelerator) (§2.6.6)?

Fairness How similar is the DMA bandwidth for each physical accelerator (§2.6.7)? Does OPTIMUS enforce different scheduling policies among its virtual accelerators (§2.6.8)?

2.6.1 Experimental Setup

Hardware We evaluate OPTIMUS on Intel Skylake HARP [150]. The platform features a 2.8 GHz Xeon CPU and a 400 MHz Arria 10 FPGA [158] located in the same package. The CPU and FPGA are connected via a single UPI [217] link as well as two PCIe 3.0 links. The server has 188 GB of DRAM.

Software OPTIMUS runs CentOS 7.5 with Linux kernel version 5.1.0-rc6 as the host OS, using QEMU version 3.0.1. Each guest also runs CentOS 7.5 and is allocated 10 GB of the server's 188 GB of DRAM.

Baseline We compare OPTIMUS's performance with virtualization via pass-through (i.e., direct assignment). To allow the FPGA to directly access the application's virtual address space, we enable vIOMMU [289] (virtual IOMMU) support in QEMU. To our knowledge, there are no shared-memory FPGA hypervisors to which we can compare OPTIMUS.

Configuration Unless mentioned specifically, OPTIMUS uses 2MB huge pages with IOTLB Conflict Mitigation enabled.

Accelerators	Description	LoC	Frequency (MHz)
AES	AES128 Encryption Algorithm	1,965	200
MD5	MD5 Hashing Algorithm	1,266	100
SHA	SHA512 Hashing Algorithm	2,218	200
FIR	Finite Impulse Response Filter	1,090	200
GRN	Gaussian Random Number Generator	1,238	200
RSD	Reed Solomon Decoder	5,324	200
SW	Smith Waterman Algorithm	1,265	100
GAU	Gaussian Image Filter	2,406	200
GRS	Grayscale Image Filter	2,266	200
SBL	Sobel Image Filter	2,451	200
SSSP	Single Source Shortest Path	3,140	200
BTC	Bitcoin Miner	1,009	100
MB	Random Memory Accesses	1,020	400
LL	Linked List Walker	695	400

Table 2.1: The benchmarks used to evaluate OPTIMUS, the number of lines of Verilog code used to implement benchmarks, and the frequencies at which benchmarks are executed.

Benchmarks Table 2.1 shows the fourteen benchmarks with which we evaluate OPTIMUS. Ten of these benchmarks are ported from HardCloud [102], an open-source framework that offloads OpenMP [113] computation tasks to the FPGA. Our HardCloud benchmarks are all compute-intensive; they include signal processing, cryptography, scientific computing, and image processing applications. We port these benchmarks to our virtualization platform, and use their default configuration during synthesis. Besides, we also port an FPGA based graph processing application (single source shortest path or SSSP) [302], and a bitcoin miner [42] to our virtualization platform. Unlike in §2.2.1, we only evaluate the shared-memory implementation of SSSP in this section, while configuring the benchmark to use a graph with 800K vertices and 12.8M edges. HardCloud benchmarks, SSSP, and Bitcoin are chosen to represent real-world applications.

Since no open-source benchmarks for HARP place sufficient strain on OPTIMUS's bandwidth and latency for a single acceleration job, and because no existing benchmarks conform to OPTIMUS's preemption interface, we provide two benchmarks ourselves. Both of these benchmarks implement OPTIMUS's preemption interface in order to evaluate OPTIMUS's temporal multiplexing capabilities.

MemBench (MB) *concurrently* issues random DMA read and write requests in order to saturate HARP's bandwidth. The random reads and writes result in the worst-case effects of IOTLB misses, and thus minimize throughput benefits from memory locality.

LinkedList (LL) *sequentially* fetches cache line sized nodes from a linked list distributed randomly in DRAM, connecting the performance of LinkedList to worst-case DMA patterns and thus creating a latency bottleneck. Because shared-memory FPGAs are an emerging technology, there are currently few open-source benchmarks that leverage this model. However, LinkedList represents the fundamental limitations for irregular parallel applications (i.e., with a lot of pointer chasing), and prior work [276] has demonstrated that linked lists are sufficient to study the overhead of latency-bound workloads on shared-memory FPGA platforms.

The latency sensitivity of LinkedList requires special treatment due to the intricacies of the HARP platform. All HardCloud benchmarks allow the HARP shell to automatically select the interconnect channel (PCIe or UPI) used for each IO packet; for throughput-bound workloads, this configuration generally yields optimal performance [149]. However, for a highly latency-sensitive benchmark such as LinkedList, automatic channel selection yields unstable performance. HARP's channel selector is optimized for throughput rather than latency. Thus, although UPI has lower latency for reads [149], the channel selector places some reads on PCIe, leading to wide performance variation for latency-sensitive benchmarks. As such, we measure the performance of LinkedList under two configurations: PCIe-only and UPI-only.

Table 2.1 shows the frequency at which each benchmark is run. Ideally, each benchmark would be run at the highest frequency that the FPGA board supports (400 MHz). However, a number of the benchmarks are too complex for HARP's current synthesizer to be able to ensure that their circuits can correctly operate at this maximum frequency; the synthesizer cannot place the FPGA logic elements sufficiently close in order to propagate signals quickly enough. We therefore synthesize each benchmark at the highest frequency achievable with OPTIMUS's maximum number of physical accelerators (eight). As synthesis algorithms improve, we anticipate being able to run the benchmarks at higher frequencies.

2.6.2 FPGA Resource Utilization

In this section, we evaluate the impact of OPTIMUS on FPGA resource utilization as reported by Intel's FPGA toolchain. We measure the percent of on-FPGA resources consumed by the hardware monitor (indicating virtualization overhead), and we explore the extent to which spatial multiplexing can improve FPGA resource utilization.

Table 2.2 displays the percentage of Adaptive Logic Modules (ALMs) and Block RAM (BRAM) that each major FPGA component utilizes on (1) a single accelerator pass-through baseline versus (2) eight accelerators under OPTIMUS. The FPGA shell is an inherent component in both OPTIMUS and the pass-through baseline, and consumes 23.44% of ALMs and 6.57% of BRAM. The hardware monitor is only present in OPTIMUS, but utilizes just 6.16% of the ALMs and 0.48% of the BRAM, indicating low virtualization overhead in terms of resource utilization.

Without the spatial multiplexing of OPTIMUS, benchmarks in the pass-through accelerator configuration utilize no more than 5% of available FPGA resources. OPTIMUS's spatial multiplexing increases aggregate accelerator resource utilization roughly linearly. With eight accelerators,

FPGA Component		ALM Usage (%)		BRAM Usage (%)	
		OPTIMUS	PT	OPTIMUS	PT
Shell		23.44	23.44	6.57	6.57
Hardware Monitor		6.16	0.00	0.48	0.00
Accelerators	AES	27.80	3.62	23.01	2.82
	MD5	34.27	4.35	23.01	2.82
	SHA	18.16	2.16	22.46	2.82
	FIR	15.77	1.92	22.46	2.82
	GRN	12.53	1.76	7.98	1.02
	RSD	17.93	2.21	22.87	2.87
	SW	10.34	1.42	11.67	1.47
	GRS	9.92	1.32	18.15	2.28
	GAU	25.28	3.41	21.24	2.60
	SBL	18.49	2.39	20.30	2.55
	SSSP	15.73	1.96	22.47	2.82
	BTC	8.99	1.32	4.16	0.48
	MB	4.84	0.83	0.00	0.00
	LL	-0.24	0.15	0.00	0.00

Table 2.2: Breakdown of FPGA resource utilization by component (ALM and BRAM). Each component's utilization is reported as a percentage of the total amount of each resource type available on the FPGA. The pass-through (PT) baseline features a single instance of the accelerator benchmark, while OPTIMUS features eight instances in order to compare resource utilization in the presence of spatial multiplexing.

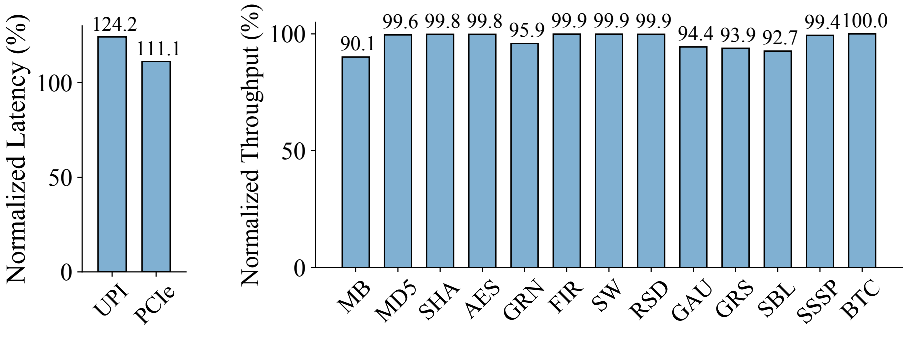
OPTIMUS's slight overhead beyond 8x stems from increased circuit complexity as the number of accelerators increases. Specifically, the synthesizer must consume extra resources in order to route signals to different locations on the FPGA chip under timing requirements.

MemBench and LinkedList are sufficiently simple that the synthesizer is able to optimize the FPGA configuration, yielding a sublinear relationship. MemBench only uses 6x the number of ALMs as the pass-through baseline. As for LinkedList, overall resource usage actually decreases, and is thus listed as using a negative portion of resources in Table 2.2.

2.6.3 Performance Overhead

To measure the virtualization overhead introduced by OPTIMUS, we compare the performance of an accelerator virtualized via pass-through (i.e., direct assignment) with an accelerator virtualized via OPTIMUS, as shown in Figure 2.4.

Latency Figure 2.4a shows the latency overhead for LinkedList—a microbenchmark which represents the worst-case for latency-bound applications—when running in PCIe-only mode and UPI-only mode. The 24% latency overhead of LinkedList stems from a decision to favor scalability



(a) Latency (b) Throughput

Figure 2.4: Performance overhead of different benchmarks compared to pass-through.

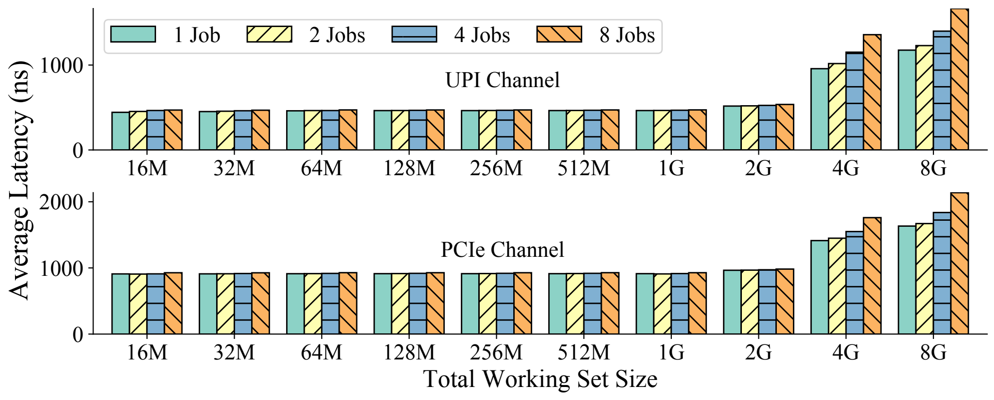
over latency in the arrangement of our hardware multiplexers. In order to pass timing requirements when scaling to eight accelerators, we require a three-level binary tree (as opposed to a single multiplexer with eight child accelerators). Unfortunately, each added layer of the tree adds approximately 33 ns of latency; therefore, our design induces approximately 100 ns of latency on the path through the multiplexer tree in order to provide scalability.

Throughput Figure 2.4b displays the throughput overhead for the remaining benchmarks. For MemBench (a microbenchmark which represents the worst-case for bandwidth-intensive applications), the relative throughput overhead is 9.9%. MemBench is specifically designed to stress the interconnection as much as possible, and therefore issues memory requests at every possible FPGA cycle. However, given the routing complexity of the multiplexer tree, the accelerator can only transmit a memory request packet every two cycles. Thus, the multiplexer tree is again the primary source of overhead. Despite this worst-case scenario, our HardCloud benchmark results indicate that the throughput overhead of OPTIMUS is less than 5% for realistic applications.

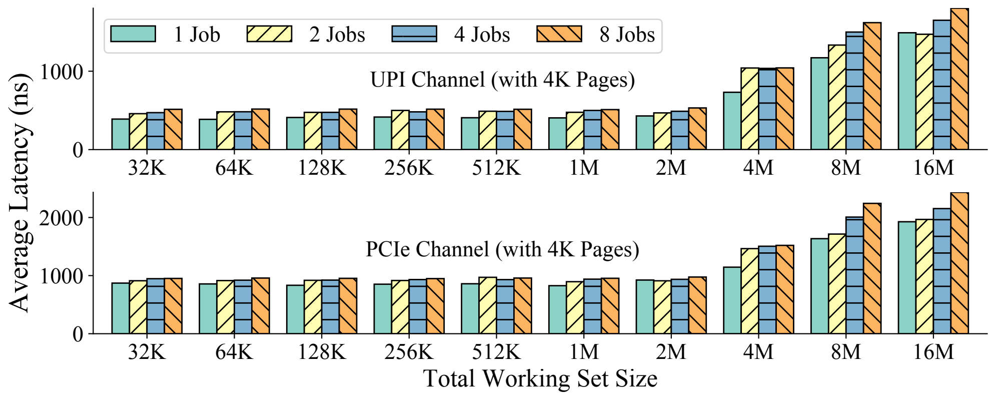
2.6.4 Scalability of Spatial Multiplexing

In this section, we assess OPTIMUS's ability to scale with respect to the number of acceleration jobs executing concurrently on the FPGA. For each benchmark, we place eight instances of the accelerator on the FPGA (i.e., the maximum number of physical accelerators that can be synthesized on our platform). We measure the performance of each benchmark as the number of concurrent acceleration jobs increases.

Latency Because LinkedList is highly sensitive to memory access latency, we measure the benchmark's execution time as the number of acceleration jobs increases to determine the effects



(a) With 2M Pages



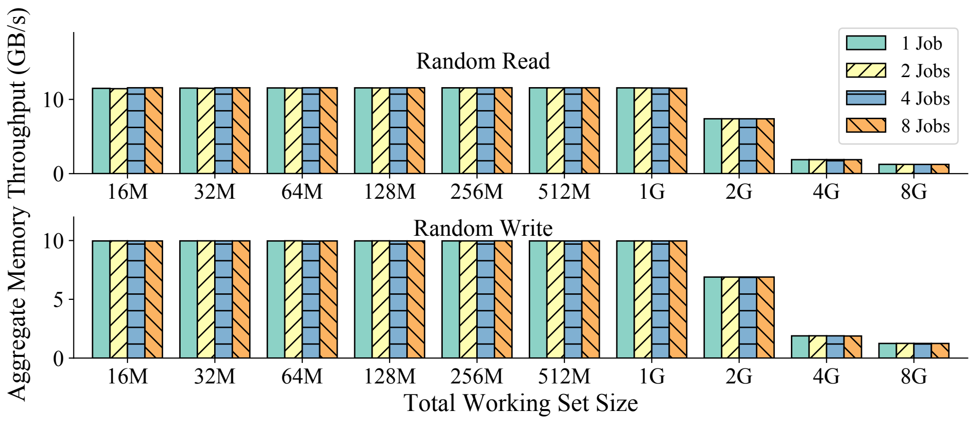
(b) With 4K Pages

Figure 2.5: Average memory access latency of LinkedList with different working set sizes and number of virtual machines.

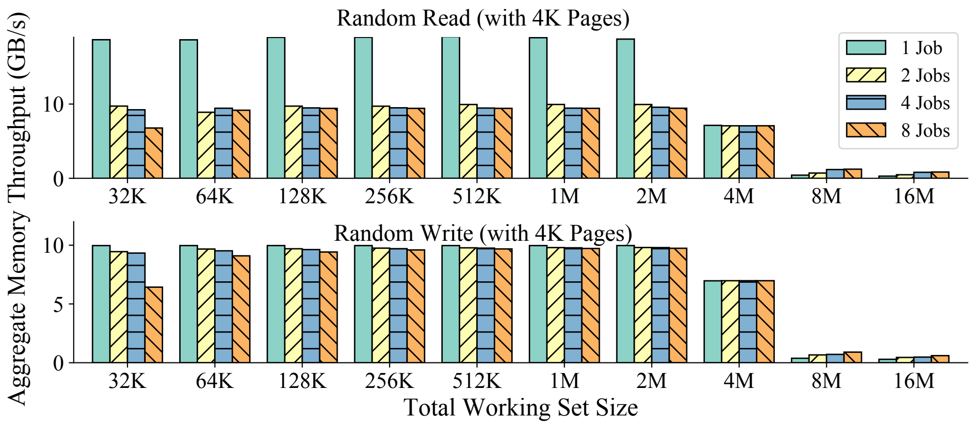
of scaling on latency. As shown in Figure 2.5a, increasing the number of acceleration jobs has negligible effect on aggregate latency if the working set does not exceed IOTLB capacity. The slight ($< 6\%$) increase from 1 job to 8 jobs is due to IO queuing delays.

When the working set barely exceeds IOTLB capacity (2G), latency only suffers a slight increase, since address translation is not overwhelmed. However, once the working set reaches 4G, the queuing delay is exacerbated by frequent address translation, resulting in a rapid increase in average latency as the number of jobs grows.

Throughput Since a single instance of MemBench saturates the platform's bandwidth, MemBench indicates the worst-case measurement of throughput scalability. Figure 2.6a shows the aggregate throughput of MemBench as the number of acceleration jobs and aggregate working set



(a) With 2M Pages



(b) With 4K Pages

Figure 2.6: Aggregate throughput of MemBench with different working set sizes and number of virtual machines.

size are increased. As demonstrated, increasing the number of acceleration jobs does not diminish the aggregate throughput. Thus, OPTIMUS scales well in terms of memory access throughput.

The drop-off in throughput beyond 1 GB is not due to OPTIMUS, but rather due to the limitations of the current HARP IOMMU. Since we believe that the IOTLB only contains 512 entries when the page size is 2 MB, the IOTLB is limited to only caching the mappings of 1 GB virtual address space. Thus, when the aggregate working set size exceeds 1 GB, throughput degrades as a result of IOTLB misses.

In HARP, the IOMMU is not integrated into the CPU in order to minimize CPU modifications needed to support the experimental platform. As a result, upon each IOTLB miss, the IOMMU must go through the system interconnection to fetch the required IO page table from the CPU. We argue that in future generations of shared-memory FPGA platforms, the manufacturer should increase the number of IOTLB entries and integrate the IOMMU into the CPU in order to mitigate the frequency and severity of IOTLB misses. Additionally, supplementing a CPU-integrated IOMMU with hard-wired support for SR-IOV could potentially allow SR-IOV to scale on shared-memory platforms. Further modifying the IOMMU to support SR-IOV on UPI links could even allow SR-IOV to virtualize encapsulated PCIe and UPI transactions.

Figure 2.7 shows the aggregate throughput (normalized to a single acceleration job) of our real-world applications as the number of acceleration jobs is increased. Unlike MemBench, none of these applications fully utilize the bandwidth for a single acceleration job. As a result, the aggregate throughput increases as the number of acceleration jobs increases. Except for Gaussian, Grayscale, Sobel, and Bitcoin, whose working set sizes are relatively small, the total working set sizes of other applications vary from 2GB to 32GB, which means the capacity of IOTLB is exceeded. However, since all these applications are well-designed to have good memory locality, performance is not impacted due to IOTLB thrashing.

2.6.5 Benefit of Using Huge Pages

To measure the performance benefit of "huge" (2M) pages, we compare the throughput and latency when using 2M versus 4K pages. Figure 2.5 and Figure 2.6 compare the results of 2M versus 4K paging in terms of latency and throughput, respectively.

OPTIMUS suffers from a performance drop when the aggregate working set exceeds the IOTLB capacity (512 pages); a 2M TLB entry can serve 512 times more memory than a 4K entry. Using 2M pages can thus postpone the performance drop from a 4M aggregate working set to 2G, which is beneficial for applications with a large working set.

As shown in Figure 2.6b, we discovered an unusually-high read throughput when (a) there is only one accelerator, and (b) the working set does not exceed 2M. We noticed a similar phenomenon with 2M pages, which is not pictured due to spacing constraints. While we cannot definitively determine the source of this behavior, we believe the phenomena arise due to a speculative optimization in the IOTLB pipeline, which assumes that subsequent memory accesses will access the same 2 MB region as previous accesses.

2.6.6 Scalability of Temporal Multiplexing

In this section, we evaluate how OPTIMUS scales with respect to the oversubscription factor (i.e., the number of virtual accelerators per physical accelerator). Since only MemBench and LinkedList conform to OPTIMUS's preemption interface, we are limited to directly evaluate these benchmarks. However, preemption overhead is correlated to the amount of execution state that must be saved. Therefore, because we know the total set of resources consumed by each accelerator configuration,

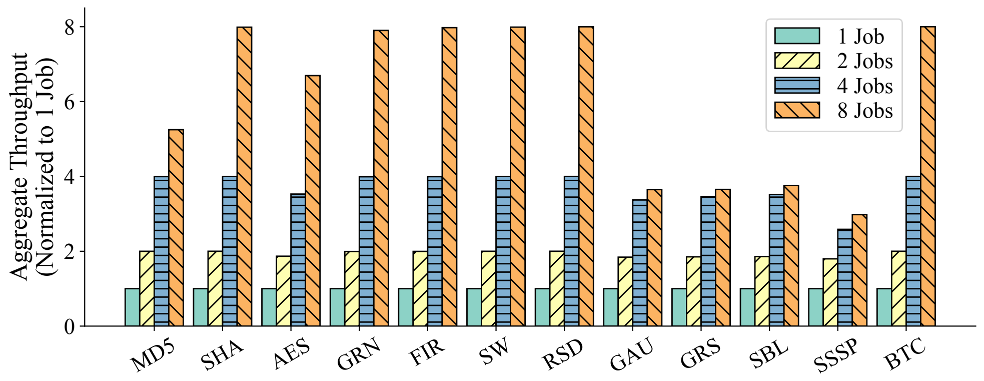


Figure 2.7: The aggregate throughput of different real-world applications, normalized to the throughput of a single VM. GAU, GRS, SBL, and SSSP fail to scale because the interconnection bandwidth becomes saturated, creating a performance bottleneck beyond four accelerators.

Accelerators	AES	MD5	SHA	FIR	GRN	RSD	SW
Normalized Throughput Range (10^{-4})	21.9	11.9	4.40	30.1	108	1.77	3.79
Accelerators (Cont.)	GAU	GRS	SBL	SSSP	BTC	MB	LL
Normalized Throughput Range (Cont.) (10^{-4})	63.1	1.60	147	595	0.468	1.83	3.25

Table 2.3: Normalized throughput range among eight homogeneous physical accelerators.

we can use this percentage as an upper bound on the amount of state that must be saved, thus establishing an upper bound on context-switching overhead.

Figure 2.8 presents the aggregate throughput of running a varying number of virtual accelerators on a physical accelerator, normalized against a single job on an accelerator. Theoretically, OPTIMUS does not have a hard limitation on the scalability of temporal multiplexing. Our evaluation stops at 16 because we are able to show that the context-switching overhead does not increase as the number of jobs increases.

As indicated by the drop-in throughput between 1 and 2 jobs, the overhead of preemption for LinkedList is approximately 0.5%. For MemBench, this number is 0.7%. The overhead remains constant beyond 2 jobs because preemption occurs at a fixed interval in the presence of temporal multiplexing, regardless of the number of jobs being multiplexed.

We estimate the worst-case overhead of temporal multiplexing for real-world applications by

Co-located Accelerator	AES	MD5	SHA	FIR	GRN	RSD	SW
Normalized Throughput	0.86x	0.50x	0.77x	0.75x	1.00x	0.78x	0.78x
Co-located Accelerator (Cont.)	GAU	GRS	SBL	SSSP	BTC	MB	LL
Normalized Throughput (Cont.)	0.80x	0.80x	0.79x	0.75x	1.00x	0.5x	1.00x

Table 2.4: MemBench's throughput when co-located with a second active accelerator, normalized against a standalone instance.

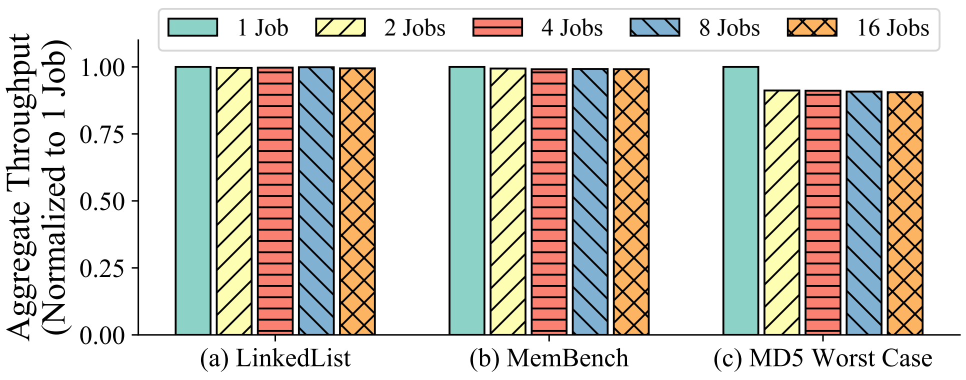


Figure 2.8: Normalized aggregate throughput in the presence of preemptive temporal multiplexing. All virtual accelerators are scheduled on a single physical accelerator.

simulation. Since MD5 occupies the most on-FPGA resources of any real-world application, we use this benchmark to establish an upper bound. Our estimation yields 9% temporal multiplexing overhead in the worst case (i.e., assuming all resources occupied by MD5 must be saved on a context switch).

We stress that the amount of state that must be saved is application-dependent. If the amount of state is large, the length of each time slice can be increased to reduce the number of context switches, thereby mitigating the penalty.

2.6.7 Fairness of Spatial Multiplexing

In this section, we measure the fairness of the hardware scheduler in terms of its ability to guarantee at least 1/*N* of the total real-time bandwidth to each of *N* physical accelerators, assuming those accelerators are actively transmitting data. We assess the bandwidth fairness in both homogeneous configurations (where the FPGA is configured with multiple instances of the same accelerator) and heterogeneous configurations (where the FPGA is configured with various accelerators).

Homogeneous Configurations For each benchmark, we configure the FPGA with eight homogeneous accelerators and measure the per-accelerator throughput. Table 2.3 presents the *normalized throughput range* (i.e., the difference between the maximum and minimum accelerator throughput divided by the average throughput) for each benchmark. The maximum normalized throughput range is approximately 1%, demonstrating that the difference in throughput between any two accelerators is at most 1%. In other words, given eight homogeneous accelerators, each accelerator achieves roughly 1/8 of the aggregate throughput. Thus, the hardware monitor fairly multiplexes the FPGA among physical accelerators in homogeneous FPGA configurations.

Heterogeneous Configurations MemBench is designed to saturate HARP's bandwidth for a single job. Therefore, we use it as a baseline for full throughput, and measure the relative decrease in MemBench's throughput in the presence of a second active accelerator benchmark.

Table 2.4 shows the normalized throughput reported by the MemBench accelerator for each configuration. In the presence of a second active accelerator, MemBench is guaranteed to receive at least half of the original bandwidth.

Upon first glance, MemBench receiving more than half of the total bandwidth may appear to be unfair. However, most accelerators do not transmit data as often as MemBench. For instance, in the cases where data is rarely transmitted by the other accelerator (e.g., LinkedList), MemBench receives a near-complete share of the bandwidth. When the second accelerator is also bandwidth-hungry (e.g., MD5 and a second instance of MemBench), the bandwidth is evenly split.

2.6.8 Fairness of Temporal Multiplexing

Enforcing fairness in the context of a software scheduler means being able to enforce the cloud provider's custom time-sharing policy. OPTIMUS implements an unweighted round-robin scheduler (i.e., equal time slices), a weighted scheduling policy (i.e., weighted time slices), and a priority scheduler (i.e., the job with the greatest priority runs at each time slice). We verify that the software scheduler successfully enforces each policy by measuring the execution time of each virtual accelerator across varying oversubscription factors, time slice lengths, and job weights/priorities. On average, the actual execution times are within 0.32% of the expected times, with the greatest difference being 1.42%. Thus, OPTIMUS successfully enforces each of its software scheduling policies.

2.7 Discussion

2.7.1 OPTIMUS vs. AMORPHOS

AMORPHOS [176] targets OS management of FPGAs. Like OPTIMUS, AMORPHOS enables both spatial and temporal multiplexing of FPGAs. AMORPHOS overcomes the static limitations of partial reconfiguration (i.e., forcing accelerator designs to fit into a fixed-size FPGA partition) through an abstraction called morphlets. Specifically, AMORPHOS virtualizes an FPGA as a set of morphable tasks, which can alter their resource requirements at runtime to dynamically accommodate a greater or lesser number of accelerators on the same FPGA. OPTIMUS does not support dynamic scalability on a single FPGA. However, since OPTIMUS supports acceleration preemption, OPTIMUS's virtual accelerators can theoretically be migrated in the event that a cloud provider wishes to alter an FPGA configuration.

The fundamental difference between AMORPHOS and OPTIMUS is that they target different FPGA platforms (host-centric vs shared-memory, respectively). The differences between these platforms are substantial (e.g., different software/hardware programming interfaces, memory latencies/capacities, hardware topologies, and so forth).

Most importantly, these platforms necessitate significantly different forms of memory management. Because AMORPHOS targets host-centric platforms—where accelerators *cannot* issue their own DMAs—it focuses on virtualizing each accelerator's view of on-FPGA DRAM. Thus, AMORPHOS's memory protection logic only needs to manage on-FPGA DRAM, and can do so with segment-based translations.

On the other hand, OPTIMUS targets platforms in which the FPGA uses the system DRAM. Thus, OPTIMUS must integrate accelerator memory protection with the host's page-level memory management, while maintaining consistent views of each address space for the CPU and FPGA. Nonetheless, given that platforms such as Intel PAC [153] give FPGAs access to both system and on-FPGA DRAM, our approaches to memory virtualization are complementary to those of AMORPHOS.

2.7.2 Key Takeaways

We believe our work highlights two key areas for improvement in systems and architectural support for heterogeneous computing. First, there is a need for new OS abstractions. Currently, each FPGA vendor uses a different programming interface. Thus, standard OS abstractions (e.g., to send messages to the CPU and access different memories) would immensely increase program portability. FPGA manufacturers can hasten the arrival of such OS abstractions by providing a standardized hardware interface.

Second, a hard-wired multiplexer tree is needed to provide more efficient and scalable packet routing. Like AMORPHOS, OPTIMUS also confirms that a flat multiplexer becomes a bottleneck for scalability. Furthermore, OPTIMUS shows that even a programmer-synthesized multiplexer tree can be a bottleneck at higher frequencies. These bottlenecks arise due to the difficulty of placing multiplexer resources sufficiently close to pass timing constraints, but could be mitigated via a hard-wired multiplexer tree.

2.8 Related Work

Accelerator Libraries Amazon F1 [66] and Microsoft Brainwave [95, 214] offer accelerator libraries to their customers. The customer chooses from among these accelerators, ultimately running their acceleration job on an FPGA that has been configured accordingly. OPTIMUS is targeted for this use case, and allows the cloud provider to spatially and temporally multiplex their FPGAs among customers.

Sharing On-FPGA Memory Asiatici et al. propose a hypervisor featuring a high-level framework to facilitate FPGA application development [76]. The hypervisor provides a framework to share on-FPGA memory among multiple accelerators. CoRAM [109] and CoRAM++ [274] similarly allow software to read and write on-FPGA BRAMs. Unlike OPTIMUS, none of these designs grant the CPU and FPGA a unified view of memory.

Sharing System Memory FPGAs can share system memory with the CPU on platforms such as Intel PAC [153] (PCIe-only), Intel HARP [150] (PCIe and UPI), and Enzian [77] (forthcoming). GPUs from Intel [156, 262] and NVIDIA [200, 38, 239, 301] can transparently share memory regions with the CPU, using both software-only and hardware-assisted techniques. OPTIMUS's page table slicing is inspired by such GPU page table partitioning techniques (as well as those of Virtual WiFi [280]) in a hardware-software co-design that is independent of accelerator design and behavior.

Overlays FPGA overlays [92, 186, 161, 162] provide an abstraction of FPGA hardware such that configurations can be made architecture-agnostic. Unfortunately, the abstractions of overlays sacrifice throughput and resource utilization compared to configurations built for specific FPGA architectures. Given that the burden of developing accelerators is not placed on the customer in OPTIMUS, we believe that cloud providers and customers would prefer the efficiency of native builds over the ease of cross-platform porting.

Virtualizing FPGA Pools Xilinx SDAccel [285], Tarafdar et al. [259], and Microsoft Catapult [229, 101] target the virtualization of FPGA pools, allowing jobs to be scheduled on available accelerators within the pool. Unlike these systems, OPTIMUS targets the virtualization of individual FPGAs.

Virtualizing Individual FPGAs Prior work explores spatial multiplexing [104, 273, 223, 225, 96, 267] and temporal multiplexing [272, 290, 104, 223, 225] of FPGAs. While most of these works focus on host-centric FPGAs, OPTIMUS focuses on shared-memory FPGAs. An exception is AvA [290], which uses API remoting to virtualize accelerators. Unlike OPTIMUS, AvA targets a higher level of abstraction (e.g., OpenCL), and virtualizes the userspace library instead of low-level hardware.

FPGA OSes BORPH [253, 254] supplements software processes with *hardware processes*, which communicate with other processes via standard UNIX interfaces. ReconOS [205] and Hthreads [224] extend the domain of multi-threaded programming to an FPGA, and provide support for inter-thread communication and synchronization. LEAP [130] offers reliable and latency-insensitive communication channels between different hardware modules. AMORPHOS [176] provides support for sharing different on-FPGA resources. Unlike these works, OPTIMUS is a hypervisor that focuses on virtualizing shared-memory FPGAs as a set of accelerators.

SR-IOV for FPGAs Intel [151] and Xilinx [282] both offer IP to support hardware-assisted FPGA virtualization of PCIe transactions via SR-IOV [191]. However, state-of-the-art shared-memory FPGA platforms that use SR-IOV do not support more than one VF [150, 153]. OPTIMUS supports up to eight physical accelerators, which can each support both UPI and PCIe transactions as well as an arbitrary number of virtual accelerators.

Partial Reconfiguration A number of FPGA virtualization solutions [176, 104, 96, 273] target partial reconfiguration capabilities of FPGAs, where an individual accelerator can be reconfigured without needing to reconfigure the entire FPGA. Because Intel HARP currently only provides a single reconfigurable region on the FPGA, OPTIMUS does not support partial reconfiguration; doing so would overwrite the hardware monitor.

2.9 Conclusion

In this project, we presented OPTIMUS, the first scalable hypervisor for shared-memory FPGA platforms. OPTIMUS provides both spatial and preemptive temporal multiplexing of FPGAs, such that individual accelerators on an FPGA can be fairly overprovisioned to guests. OPTIMUS offers efficient virtual DMA isolation via page table slicing. Our experiments show that OPTIMUS can support eight physical accelerators on a single FPGA, and improves the aggregate throughput of twelve realistic benchmark workloads by 1.98x-7x.

CHAPTER 3

Debugging in the Brave New World of Reconfigurable Hardware

3.1 Introduction

Field Programmable Gate Arrays (FPGAs) are increasingly prominent in modern heterogeneous computer systems. Specialized hardware designs provide unprecedented efficiency in domains such as machine learning [247, 248, 300, 293, 202, 299, 178], compression [232, 297], database operations [222, 240, 250], graph processing [78, 303, 106, 270], networking [128, 269, 94], and storage virtualization [192]. To realize the benefits of FPGAs, systems researchers have built operating systems [131, 254, 176, 187], virtualization support [207, 96, 104, 272, 295, 196, 294, 291], just-in-time compilers [241], and high-level synthesis tools [100, 99, 285, 152, 286]. The proliferation and benefits of FPGAs have even prompted major cloud vendors to provide FPGA instances on their platforms [66, 64].

Compared to traditional hardware development, FPGA development has many similarities to software development. Since post-fabrication bugs are extremely costly to fix, traditional hardware development invests massive resources into simulation-based testing and formal verification to eradicate bugs *before* silicon fabrication. In contrast, reconfigurability allows a developer to patch hardware bugs in an FPGA, even those caught during on-FPGA testing or in production. As a result, FPGA developers are moving towards an agile development approach that accelerates time to market by relaxing cumbersome verification in favor of lightweight simulation and on-FPGA testing. For example, Microsoft has adopted a software-like methodology for FPGA development, in which they perform relatively small amounts of verification compared to traditional hardware [128].

Unfortunately, relaxed verification leads to more bugs in FPGA designs, with most FPGA projects experiencing bugs that escape testing and end up in production [132]. Alas, while there are many hardware tools that help developers *find* bugs using simulation-based testing and verification [252, 277, 57, 278, 195, 263, 298, 163, 199], very few hardware debugging tools help a developer *localize the root cause* of a bug. Existing fault localization tools only apply to specific protocols and algorithms [55, 211]. Other tools, such as checkpointing [249, 80, 37, 79, 180] and tracing [160, 288, 138, 137, 146, 241, 196], can be used to localize the cause of a hardware failure, but require substantial manual effort to do so. Finally, existing software fault localization techniques, such as data-race detectors [243] and undefined memory use detectors [242], cannot be immediately applied to hardware programming models. Consequently, debugging an FPGA design today is a highly manual process that either involves inspecting a massive waveform (i.e., a trace of the state of the circuit over time) or iterative rounds of synthesis in which a developer selects and analyzes key data signals. Unsurprisingly, a majority of FPGA developers in a recent study indicate a need for better debugging tools [50].

Reaping the full benefits of rapid FPGA development will require constructing FPGA debugging tools that help localize the root cause of a hardware fault—similar to the rich set of tools available to software developers. Towards this goal, we first study bugs in open-source FPGA designs. We then introduce a novel root-cause-based classification of the bugs we study inspired by a prior bug taxonomy [201] and document the intrinsic properties and symptoms of these bugs. We augment our study with a testbed in which each hardware bug is reliably reproducible. We demonstrate that each class of hardware bugs mirrors a counterpart class of software bugs and would benefit from similar techniques for bug diagnosis and repair.

Guided by the intrinsic properties and symptoms of bugs in FPGA designs, we build a collection of hybrid static/dynamic program analysis and monitoring tools to help developers of reconfigurable hardware systems follow a software-like development and debugging process. Because hardware bugs may be detected during simulation or when executing on an FPGA, our tools are designed to operate in either scenario. Thus, we consider the effects of our debugging logic on real circuit synthesis and behavior, as opposed to only accounting for a simulator environment where resource and timing constraints are far less stringent. At a high level, our tools allow selectively recording and analyzing targeted execution information using limited on-FPGA storage, and consist of the following:

1- SignalCat unifies hardware debugging during simulation and when deployed on an FPGA by providing a single interface for tracing state in a hardware design. The tool converts "`printf`"-like statements embedded in a hardware description into logic that records the arguments of these statements in a hardware deployment or during simulation. After an execution, SignalCat reconstructs a log containing the output of the `printf` statements.

2- FSM Monitor helps a developer identify and track finite state machines (FSMs), which are a widespread component in a hardware design. It uses SignalCat to support both simulation and on-FPGA scenarios.

3- Dependency Monitor enables a developer to trace the provenance of the value of a variable in their hardware design. The tool identifies the dependency chain of each developer-specified variable (i.e., the registers upon which the variable depends), and tracks all updates made to these variables during a simulation or on-FPGA execution using SignalCat.

4- Statistics Monitor helps a developer identify anomalous behavior by recording statistics about various execution events, such as the number of times that an interrupt is triggered or the number of packets that arrive in a communication channel. Developers specify an event of interest; Statistics Monitor instruments the hardware design with new logic that uses SignalCat to track statistics during simulation or on-FPGA scenarios.

5- LossCheck helps a developer localize the root cause of data loss (e.g., an unintended packet drop). A developer who suspects data loss in their design uses LossCheck to check for—and potentially identify the source of—data loss between a specified source (e.g., an input to a hardware module) and sink (e.g., an output). The tool instruments the hardware design with new logic that monitors all data propagation paths between the source and sink by using SignalCat.

We show how a developer can use the aforementioned tools—either individually or in various combinations—to debug the bugs in our study. In particular, we show that our tools help diagnose the cause of each bug in our study by automatically generating and executing dozens to thousands of lines of analysis code, which the developer would otherwise need to write. Additionally, we evaluate the resource overhead of our debugging tools and demonstrate that they are feasible for production use. Among the 20 bugs we evaluated, 18 cases maintain the design's original target frequency after debugging instrumentation; all cases incur at most linear resource overheads with increased recording buffer sizes.

Overall, we make the following contributions:

- We provide the first study of bugs in open-source FPGA designs, a root-caused-based bug classification, and a description of typical bug symptoms to guide developers in their debugging efforts.
- We design a collection of hybrid static/dynamic analyses that developers can use in simulation and real hardware deployments to debug FPGA designs.
- We develop an open-source testbed [13] that includes reproducible hardware bugs and our tools to facilitate future FPGA debugging research.

A version of this work was previously published in the proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22) [208].

3.2 Background

In this section, we discuss the FPGA development concepts that are necessary for understanding the bugs and debugging tools presented in this paper.

3.2.1 Languages for Hardware Programming

Developers program FPGAs by implementing a digital circuit in a hardware description language (HDL), such as Verilog [261], SystemVerilog [56], or VHDL [90]. HDLs enable developers to describe the behavior of a circuit in a cycle-by-cycle manner. For instance, the simple statement $c <= a + b$ subscribes to a broad programming paradigm: right hand expressions ($a + b$) are computed and propagate to left hand operands (c) via an appropriate assignment operator ($<=$) at each clock cycle.

Emerging high level synthesis (HLS) tools enable hardware development using software programming languages, but impose significant performance and resource penalties compared to HDLs. For example, state-of-the-art HLS-implemented image processing is 6.6$\times$ slower and uses 5$\times$ more resources than an HDL-implementation [216]. As such, HDLs continue to dominate hardware development.

3.2.2 FPGA Debugging Stages

FPGA debugging contains two stages: simulation and on-FPGA testing. Simulation avoids lengthy hardware synthesis and is thus faster to iterate, but executes orders of magnitudes slower than on-FPGA testing [241]. In practice, developers simulate FPGA designs and iteratively fix any bugs they find before employing on-FPGA methods to test their design against more complex workloads (e.g., via stress testing).

3.2.3 FPGA Programming Techniques and Constructs

Hardware developers leverage a number of common techniques and constructs to implement FPGA designs.

Buffers and Queues Hardware developers use buffers and queues to temporarily store values. Hardware buffers and queues are similar to their software equivalents, except they must be constant-sized, since all hardware components occupy a fixed area in a circuit.

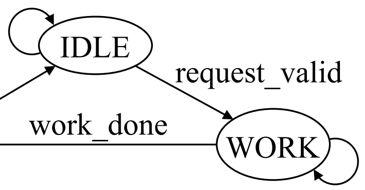


Figure 3.1: An example FSM with states represented by nodes, and transition conditions represented by edges. This FSM has three states: `IDLE`, `WORK`, and `FINISH`. A state transforms to another state when a certain condition is satisfied.

```verilog
reg [1:0] state;
always @(posedge clk) begin
  case(state)
    IDLE: if (request_valid) state <= WORK;
    WORK: if (work_done) state <= FINISH;
    FINISH: state <= IDLE;
  endcase
end
```

Listing 1: Verilog code implementing of the state transition of the FSM in Figure 3.1.

Communication Control: Valid Interface Logically, hardware circuits continually process data, with one or more input signals consumed every clock cycle. However, an input signal may not always be meaningful. For instance, a module may only receive a "packet" every 5 cycles. Thus, developers use valid interfaces that indicate whether a particular input is valid (i.e., a "valid bit" variable associated with one or more inputs).

Communication Control: Backpressure In a communication channel where a source repeatedly sends data to a destination, the destination may use a backpressure or "ready" signal to inform the source that it needs time to process inputs. These signals indicate to the source that the destination can only receive x new packets, where x is defined by the communication protocol (e.g., $x = 1$ for a binary ready signal). In the event of backpressure, the source should stop sending packets or reduce the sending rate to avoid bugs at the destination.

Finite State Machines Hardware developers frequently incorporate finite state machines (FSMs) in their designs [287, 65]. Figure 3.1 demonstrates an example FSM; Listing 1 shows the Verilog code that implements the FSM. In Verilog, an FSM is implemented using conditional assignments (e.g., an assignment inside a switch case); once a condition is satisfied, the "state" transfers along the arrows in the next clock cycle.

Bug Class	Bug Subclass	#	Common Symptoms			
			App Stuck	Data Loss	Incorrect Output	Ext.
Data Mis-Access	Buffer Overflow	5		✓		
	Bit Truncation	12			✓	✓
	Misindexing	5		✓	✓	
	Endianness Mismatch	1			✓	
	Failure-to-Update	5		✓	✓	✓
Communication	Deadlock	3	✓			
	Producer-Consumer Mismatch	3	✓	✓	✓	
	Signal Asynchrony	10			✓	
	Use-Without-Valid	1			✓	
Sementic	Protocol Violation	3	✓		✓	✓
	API Misuse	3			✓	
	Incomplete Implementation	7			✓	
	Erroneous Expression	10			✓	

Table 3.1: The result of our bug classification, including 3 main classes, 13 different subclasses, the number of bug instances observed in each subclass, and the common symptoms of each subclass. "#" stands for the number of bugs, and "Ext." indicates that the error is reported by an external component.

Module A module is a sub-component of a Verilog circuit with a group of input and output signals, akin to a software function.

Intellectual Property (IP) A hardware intellectual property (IP) block is a "blackbox" module that implement commonly-used or platform-specific functionality, akin to a static software library. Like a software function, an IP block accepts user-controlled inputs and produces a set of outputs.

3.3 Study of Bugs in FPGA Designs

To identify useful FPGA debugging tools, we study 68 hardware bugs across 19 FPGA designs and build a testbed [13] that reliably reproduces 20 of these bugs[1] in a push-button manner to enable their detailed study (§3.6.1). The study explores functional bugs, i.e., bugs in the HDL code that lead to functional issues rather than timing-related issues, since most production FPGA bugs are functional bugs [132]. Our methodology for gathering bugs is as follows:

Target Systems First, we study bugs in four applications that we used in prior work. In particular, these applications use the Intel HARP platform [150], which uses the FPGA as a reconfigurable accelerator and provides an end-to-end acceleration stack. Specifically, we identify bugs in a SHA512 accelerator [51], Reed-Solomon decoder [52], and grayscale image accelerator [53]

[1]We select these 20 bugs because they occur in an application/platform with which we have familiarity. The rest of the bugs could be reproduced with additional effort.

applications from HardCloud [102] (a framework with applications using HARP-based FPGA acceleration). Additionally, we find bugs in Optimus [207] (a HARP-based FPGA hypervisor).

Second, we examine bugs in hardware designs described in the ZipCPU website, a popular hardware design blog [14]. We identify bugs in SDSPI [15] (a library that drives an SD card through a Serial Peripheral Interface), Xilinx's two example AXI endpoint implementations [16, 17], and an FFT implementation [54].

Third, we study bugs found in hardware components from the most popular FPGA projects on GitHub, including a WiFi controller [18], a GPGPU processor [19], two RISC-V CPUs [20, 21], a Bitcoin Miner [22], a NIC [23, 24], and two hardware libraries [25, 26].

Finally, we examine a floating-point adder [27] that was provided to us by a hardware developer upon consultation about their experiences debugging hardware.

Bug Collection Bugs in FPGA designs are difficult to collect, reproduce, and study due to the relative dearth of open-source hardware. Exacerbating this problem, among the 50 most popular FPGA projects on GitHub, 56% do not have a publicly-accessible bug tracker and 88% do not include test cases to reproduce bugs.

Therefore, rather than analyzing hardware bugs from bug trackers, we resorted to searching commit histories/issues of FPGA projects on GitHub to identify hardware bugs. In some cases, we found bugs through direct communication with developers (Optimus and FADD) and the ZipCPU website.

For each identified bug, we manually inspect related commit messages and discussions in GitHub Issues to understand the bug's root cause and symptoms. Sometimes, the commit messages and issues do not provide sufficient information for a thorough understanding; in these cases, we inspect the hardware design's codebase as well as bug-related patches to understand the bug.

3.3.1 Bug Classification

We cluster bugs with similar root causes and symptoms into 3 main classes and 13 subclasses. Table 3.1 shows the classification results, identifying each bug subclass, the bug class to which each subclasses belongs, the number of bugs in the study that belong to each subclass, and the most common symptoms of each bug subclass.

The three bug classes roughly correspond to the three classes of software bugs from Li et al.'s software bug study [201] and are as follows: data mis-access bugs (§3.3.2), which arise when data is accessed without proper consideration for properties of the data format and are similar to software memory bugs; communication bugs (§3.3.3), which arise when a circuit violates inter-component communication standards and are similar to software concurrency bugs; and semantic bugs (§3.3.4), which arise from other remaining violations of a circuit's intended functionality and correspond to software semantic bugs. Some bugs could be classified into multiple classes/subclasses (e.g., a buffer overflow may arise because of an erroneous expression); we assign such multi-class bugs to the most related and specific subclass to which they could be assigned.

In the rest of this section, we provide a detailed description of each subclass of bug including their intrinsic properties, root causes, and common symptoms. We identify similarities between the hardware bugs and well-studied software bugs, which provide inspiration for the hardware debugging tools that we propose.

3.3.2 Data Mis-Access Bugs

Data mis-access bugs occur when the developer accesses data without proper considerations for size, endianness, and other properties of data. These bugs are similar to software memory bugs [201] (e.g., buffer overflows, our first example).

3.3.2.1 Buffer Overflow

A buffer overflow in an FPGA design occurs when a buffer is accessed with an offset that is greater than the size of the buffer. We identify 5 real-world examples of buffer overflow bugs in our bug study. We present a basic code snippet for simplicity.

```
1 reg mybuf [N-1:0]; // a buffer with N 1-bit elements
2 always @(posedge clk)
3   mybuf[offset] <= value; // offset >= N
```

Line 1 defines a buffer named `mybuf` consisting of N single-bit elements; `mybuf [N-1:0]` can be legally indexed from 0 to $N-1$ (inclusive). On Line 3, the snippet uses `offset` to assign a bit of `mybuf` to a `value`; however, the value of `offset` is greater than N and therefore overflows `mybuf`.

Accordingly, a buffer overflow in an FPGA design is similar to a software buffer overflow. However, unlike software buffer overflow bugs, which can corrupt memory by overwriting adjacent addresses, there is no notion of address adjacency beyond a buffer in hardware logic. Instead, hardware buffer overflows yield two possible outcomes: (1) the highest bits of `offset` are truncated, so an incorrect position in buffer is assigned (when the buffer size is a power of two), or (2) the assignment is ignored (when the buffer size is not a power of two). In select cases, hardware developers rely on truncation of the high bits of `offset` in their circuits for correctness, but this approach does not work for common data structures such as heaps and queues.

Symptoms Data loss from truncation or ignored assignment.

Fixes Hardware buffer overflows are fixed similarly to software buffer overflows: Developers enlarge the buffer or change the behavior of the FPGA design to avoid the overflow.

3.3.2.2 Bit Truncation

Bit truncation bugs in FPGA designs occur when assigning a variable to another variable with fewer bits. We identify 12 bit truncation bugs in 7 different FPGA designs.

The software equivalent of a bit truncation bug occurs when casting a variable to another variable that is represented with fewer bits. As in software, bit truncation in hardware may be used to intentionally discard part of a variable, which makes precise bug detection challenging.

In the following code snippet, `left` is a 42-bit variable and `right` is a 64-bit variable whose 42 bits from `[47:6]` contain meaningful data. On Line 4, `right` is cast into a 42-bit variable via `42’(right)` and then right-shifted by 6 bits before being assigned to `left`. As a result, bits $[47:42]$ are truncated unintentionally.

```
1 reg [41:0] left;  // left is a 42-bit register
2 reg [63:0] right; // bits [47:6] are meaningful
3 always@(posedge clk)
4     left <= 42'(right) >> 6;
```

Symptoms An incorrect value or an error (e.g., a page fault) reported by an external monitor (such as an FPGA shell).

Fixes Depending on the developer’s intentions, one technique for fixing truncation bugs is to perform shifts before bit-width casts. In our example, this means the developer would change Line 4 to: `left <= 42'(right >> 6);`. Another potential fix is to grow the variables that can cause truncation. For instance, a developer can change the width of `left` to 48 bits, which prevents trucation of meaningful bits in `right`. In this case, Line 4 would be updated to: `left <= 48'(right) >> 6;`.

3.3.2.3 Misindexing

A misindexing bug occurs when a developer uses an incorrect index to extract information from a variable. We identify 5 misindexing bugs in our study. For example, the IEEE-754 [49] standard defines the binary layout of 32-bit floating point, where the bits `[22:0]` are the fraction and the bits `[30:23]` are the exponent. However, in an implementation of floating point adder, the developer incorrectly extracted bits `[23:0]` as the fraction in a floating point adder, which lead to the wrong output value.

Symptoms Incorrect output or data loss, if the misindexed data used for a control signal.

Fixes Misindexing bugs are fixed by correcting the index.

3.3.2.4 Endianness Mismatch

Endianness mismatches occur when an FPGA design assumes the wrong endianness for a particular piece of data (e.g., register arrays, off-chip DRAM, and disks), similar to how kernel code may assume the wrong endianness for device driver data. One instance of endianness mismatch bug is identified in our study.

In the simplified code snippet below, the circuit stores the least significant bits of an input in `data[7:0]` (on Line 2) and the most significant bits in `data[15:8]` (on Line 3). As a consequence, the input is stored in `data` in the little endian format. On Line 5, `data` is passed to a function expecting a big endian input, causing `out` to have the wrong result.

```
1 // Store data as little endian
2 data[7:0] <= least_significant_byte;
3 data[15:8] <= most_significant_byte;
4 // Pass data to function expecting big endian input
5 out <= big_endian_function(data);
```

Symptoms A wrong value following assignment.

Fixes Developers fix endianness mismatch bugs by manipulating bytes to account for the endianness difference. For example, the bug in the above code snippet is fixed by replacing Lines 2-3 with the following code:

```
data[7:0] <= most_significant_byte;
data[15:8] <= least_significant_byte;
```

3.3.2.5 Failure-to-Update

A failure-to-update bug occurs when a developer forgets to put (including reset and initialization) a signal; we identify 5 failure-to-update bugs in our study.

Below, we provide a simple example code snippet of a failure-to-update bug. In this example, `input_counter` is incremented when the `input_valid` signal is set, while `output_counter` is incremented when `output_ready` is set. However, upon `reset`, only `input_counter` is set to 0, so `output_counter` may contain incorrect data after `reset`.

```
if (input_valid) input_counter <= input_counter + 1;
if (output_ready) output_counter <= output_counter + 1;
if (reset) input_counter <= 0;
```

Symptoms Invalid output, data loss, or violation of communication interfaces if the failure-to-reset occurs on ready/valid signals (§3.2.3).

Fixes The developer will reset each relevant signal in the system.

Takeaway #1. Data mis-access bugs can often be localized to a specific assignment, so stepping through dependency chains/FSM transitions can help localize the bug.
Takeaway #2. Data mis-access often results in data loss, so data loss detection (e.g., counting inputs received versus outputs sent) is crucial for finding bugs.

3.3.3 Communication Bugs

Communication bugs occur when the developer violates inter-component communication standards (e.g., inter-module interfaces, different clock domains, pipeline stages, etc.). They are similar to concurrency bugs in the software [201].

3.3.3.1 Deadlock

A deadlock in an FPGA design occurs when two (or more) variables have a circular control dependency on each other. Hardware deadlocks are similar to software deadlocks, where a circular dependency among resources (e.g., locks) causes the program to stall. In hardware, deadlocks are triggered due to conditional assignments (e.g., assignments inside if-statements) that execute in parallel. We identify 3 deadlock bugs in our study.

In the following code snippet, if `a` and `b` are both initialized to 0, the assignment to `out` on Line 3 will never execute.

```
1 if (a) b <= 1;
2 if (b) a <= 1;
3 if (a) out <= result;
```

Symptoms Infinite stall.

Fixes To fix the bug in the above code snippet, a developer could initialize either `a` or `b` to 1. Fixing a deadlock bug in a complex circuit is often difficult because it is challenging to identify circular dependencies.

3.3.3.2 Producer-Consumer Mismatch

When a collection of consumer registers cannot process the data values produced by a collection of producer registers, a producer-consumer bug occurs. For example, if the producers yield more valid data in a cycle than the consumers can process and store, data will be lost. Hence, a producer-consumer mismatch bug is similar to the classic "bounded-buffer" [118] producer-consumer problem in software, in which consumer threads can only process/store a limited quantity of output from producer threads. We identify 3 real-world examples of producer-consumer mismatch bugs in our study.

For a simple example, consider the following code snippet that uses a *valid interface* (§3.2.3), where producers generate and overwrite `x` and `y` at every cycle. If both `x_valid` and `y_valid` are *true* in the same cycle, then the value of `y` may be lost, since only the code on line 1 will execute.

```
1 if (x_valid) out <= x;
2 else if (y_valid) out <= y;
```

Symptoms Data loss, invalid output, or an infinite stall (if the consumer FSM logic waits for a lost producer value).

Fixes In software, locks and condition variables are used to force producer threads to wait until the consumer threads are ready to receive new values. In hardware, an analogous solution is to pause a producer by adding a back-pressure signal throughout the circuit. However, pausing a producer is invasive, since nearly all components of the circuit must be altered to accommodate the pause. Instead, an easier solution is creating a larger buffer for produced values that have not been consumed, assuming the maximum needed queue size is bounded.

3.3.3.3 Signal Asynchrony

A signal asynchrony bug occurs when two variables that are supposed to be used together—such as a data variable and its valid/backpressure interface signals (§3.2.3) or the two operands of a mathematical operation—are not updated synchronously. We identify 10 signal asynchrony bugs in our study.

The following code snippet shows a simplified example of a signal asynchrony bug. The code responds to requests from a module that requires a minimum 2 cycle difference between requests and responses. Accordingly, upon receiving a request, the code buffers the response (calculated in a single cycle) in `buffered_response` for an extra cycle (Line 1), before outputting `final_response` (Line 2). Unfortunately, the `final_response_valid` signal (indicating the validity of the response data) is set immediately following receipt of `request` (Line 3), meaning final_response and final_response_valid are out of sync. For simplicity, we omit the code resetting final_response_valid to 0.

```
1 if (request) buffered_response <= input_data + 1;
2 final_response <= buffered_response;
3 if (request) final_response_valid <= 1;
```

Symptoms An incorrect output value.

Fixes The signal asynchrony bug in the snippet can be fixed by properly delaying the final_response_valid signal to be synchronous with the final_response signal. For instance, the developer may replace Line 3 with the following lines to fix the bug.

```
1 if (request) delayed_response_valid <= 1;
2 final_response_valid <= delayed_response_valid;
```

3.3.3.4 Use-Without-Valid

A use-without-valid bug occurs when a data variable guarded by a valid signal (§3.2.3) is used when the valid signal is in an invalid state. Use-without-valid bugs are similar to signal asynchrony bugs, but occur when data is *used* erroneously, as opposed to signal asynchrony bugs which occur when data is *updated* erroneously. We identify one instance of use-without-valid bug in our study.

In the following code snippet, if data is a variable using a valid interface (e.g., with data_valid as its valid signal), sum may not be calculated correctly because it can use an invalid data as input.

```
1 // data is associated with a valid variable (data_valid)
2 sum <= sum + data;
```

Symptoms An incorrect output value.

Fixes Developers fix use-without-valid bugs by updating their code to use the correct valid interface. For example, the bug in the above code snippet is fixed by replacing Line 2 with the following two lines:

```
1 if (data_valid) sum <= sum + data;
2 else sum <= sum;
```

Takeaway #3. Given the proliferation of FSMs, circular dependencies, and infinite stalls in communication bugs, localizing the bugs would be easier with ability to record key states and statistics at arbitrary points in the circuit.
Takeaway #4. Like data mis-access bugs, debugging communications bugs would benefit from localized data loss detection.

3.3.4 Semantic Bugs

Semantic bugs occur due to remaining violations that cause the circuit to incorrectly perform its intended functionality. Semantic bugs include bugs where a developer does not correctly implement the entire high-level circuit specification (e.g., the protocol or FSM logic), misuses the API of a pre-implemented module, or does not implement special cases in complex logic. They are similar to semantic software bugs [201].

3.3.4.1 Protocol Violation

Components of an FPGA design (e.g., modules) communicate through industry-standard communication protocols such as AXI4 [74]. However, such protocols are complex and contain corner cases that are difficult to cover in testing. If a developer fails to handle all cases correctly, a protocol violation occurs and escapes from simulation-based testing. We identify 3 instances of protocol violations.

Symptoms Invalid outputs, infinite stall, or a protocol violation error reported by an external monitor (e.g., an FPGA shell).

Fixes Fixing protocol violations requires correcting a mismatch between the high-level specification and implementation or adding logic for an unhandled corner case.

3.3.4.2 API Misuse

FPGA designers use a hierarchy of modules to organize code and simplify the FPGA design process. An API misuse bug occurs when developers fail to use a pre-implemented module or IP block correctly. A hardware design may have an API misuse bug even if it implements all the involved communication protocols correctly, as it may pass wrong parameters to the module or configure it improperly. We identify 3 API misuse bugs in our study.

The following code snippet shows an example of an API misuse bug. Suppose that a developer wants to determine whether signal `a` is greater than signal `b` using a module, `greater_than`, which takes two parameters, `x` and `y`, and returns `x>y`. However, when instantiating the module, the developer erroneously connects signal a to the module's input port y and signal b to the module's input port x. Consequently, the module instance (i.e., a_greater_than_b) computes b>a instead of a>b, resulting in an incorrect output value.

```
// The greater_than module calculates whether x>y
greater_than a_greater_than_b(.x(b), .y(a), .result(out));
```

Symptoms An incorrect output value.

Fixes Fixing API misuse bugs involves correcting the mismatch between a module's API definition and how the module is used, usually by changing signal connections and the module's configuration.

3.3.4.3 Incomplete Implementation

Hardware designs can be exceedingly complex, so hardware developers omit logic to handle corner cases, either intentionally or unintentionally. Such omissions are incomplete implementation bugs and often occur in corner cases that are difficult to trigger during testing. We identify 7 instances of incomplete implementation bugs in our study.

Symptoms Incorrect and invalid output.

Fixes Developers fix incomplete implementation bugs by implementing the missing functionality, which may involve a redesign of certain components of the hardware design. Developers may also add additional test cases to cover the newly-added code.

3.3.4.4 Erroneous Expression

An erroneous expression bug occurs when hardware developers use a wrong expression in a control-flow statement (e.g., an if-statement) or data-flow statement (e.g., an assign-statement). Erroneous expression bugs are different from incomplete implementation bugs in that they involve an *incorrect* expression rather than *omitted* expressions. A wrong expression in a control-flow statement steers the hardware's control-flow to a wrong direction; a wrong expression in a data-flow statement generates an incorrect data value, which is used in other statements. In our study, we include 5 erroneous expression bugs in control-flow and 5 such bugs in data-flow.

Symptoms Incorrect and invalid output.

Fixes Developers fix erroneous expression bugs by correcting the erroneous expression in the control-flow or the data-flow.

Takeaway #5. Corner cases that trigger semantic bugs are difficult to detect, especially in simulation; runtime data recording enables debugging these scenarios.

3.4 Design of FPGA Debugging Tools

Our bug study in §3.3 demonstrates that FPGA debugging can benefit from debugging tools similar to those used in software (e.g., flexible logging capabilities and program analysis). In contrast, past hardware debugging tools have emphasized airtight verification, and do little to help a developer diagnose the cause of a bug after its symptoms have been observed.

Therefore, we propose a set of hybrid static/dynamic analysis tools that simplify root cause diagnosis in FPGA designs. In this section, we describe the tools; the evaluation demonstrates their applicability to the bugs in our study (§3.6).

First, we unify simulation and on-FPGA debugging with SignalCat (§3.4.1). While "`printf`"-like statements have traditionally only been available in HDL simulators or required platform-specific IP to implement on FPGAs, SignalCat synthesizes these statements for actual FPGA deployments across multiple platforms. The infrastructure provided by SignalCat serves as a cornerstone upon which developers can build symptom-specific tools without needing to consider the execution context of the circuit and applies directly to all 5 of the takeaways from our bug study.

Using SignalCat, we build three monitoring tools that gather targeted information based upon insights from our bug study. First, FSM Monitor (§3.4.2) statically detects FSM variables and records them at runtime, automatically reconstructing FSM state-transition traces to aid developers in debugging. Second, Dependency Monitor (§3.4.3) statically analyzes the dependencies of user-specified variables and dynamically records the updates to each dependency, allowing developers to backtrace and localize the source of an incorrect output-of-interest. Third, Statistics Monitor (§3.4.4) provides counters for user-specified events, helping users identify bugs reflected in statistical metadata (e.g., data loss is often indicated by fewer outputs generated than inputs received).

Finally, given the commonality of data loss in our bug symptoms, we develop an additional tool for the event that a developer suspects or detects data loss. In particular, LossCheck (§3.4.5) pinpoints the location of data loss within a hardware design. LossCheck statically analyzes an FPGA design and instruments it with logic that dynamically checks for data loss in suspected locations.

3.4.1 SignalCat for Unified Logging

Our bug study shows that hardware debugging would benefit from the ability to log arbitrary runtime information, just as software debugging does [292]. Today, while developers can use debug statements (e.g., `$display`) to log values during HDL simulation, similar tools are not pervasively available on deployed FPGAs without specific FPGA virtualization or IO support [241, 196]. In lieu of generic "`printf`"-like statements, developers typically use vendor-provided data recording IPs (e.g., Intel SignalTap [160] and Xilinx ILA [288]) to record a subset of variables when debugging a deployed FPGA design. Thus, developers must maintain two different versions of their FPGA design when debugging, one that uses simulation-based deubgging primitives, and one that uses on-FPGA primatives.

SignalCat bridges this gap by unifying simulation-based and on-FPGA debugging through automatic generation of on-FPGA recording logic (e.g., using FPGA vendors' IPs) from debugging statements (e.g., `$display`). SignalCat incorporates a static and a dynamic component. The static component analyzes the path constraints of debugging statements and generates an IP instance for on-FPGA data collection, while the dynamic component records the trace via the IP instance in an on-FPGA scenario.

SignalCat searches the abstract syntax tree (AST) of an FPGA design for debugging statements. For each such statement, SignalCat determines the arguments (i.e., the variables that the developers want to print) and the path constraint (i.e., the conditions under which the statement is reached) of the statement. Then, SignalCat generates an instance of a vendor-provided data recording IP to record the collected arguments and path constraints, encoding path constraints as a 1-bit `bool` per debugging statement. At each cycle, The system stores all arguments and encoded path constraints in the recording IP buffer if at least one path constraint is true. SignalCat reconstructs and prints debugging logs after execution allowing the same format for on-FPGA debugging and simulation.

SignalCat requires that developers specify the size (i.e., the number of data entries) of the IP's recording buffer and events that start and stop data recording (e.g., when the first packet arrives or an assertion is triggered). Developers can also configure the buffer to capture a fixed interval before and/or after the user-provided event.

Since SignalCat provides a single interface for simulation and on-FPGA logging, developers of debugging tools can instrument an HDL design with a "`printf`"-like statement and support simulation and on-FPGA debugging with a single code-base. In fact, all of our subsequent debugging tools (§3.4.2–§3.4.5) leverage SignalCat for runtime data recording.

3.4.2 FSM Monitor for State Machine Traces

Hardware circuits often use finite state machines (FSMs) in their design (§3.2.3). When this design paradigm is used, an FSM (state-transition) trace provides a user-friendly abstraction for circuit execution and debugging, especially in comparison to a low-level waveform (i.e., a graph of all signals at every cycle). Therefore, we propose FSM Monitor to help developers automatically generate FSM traces. FSM Monitor detects FSMs in a circuit and generates logic that monitors state changes for each detected FSM.

Hardware FSMs employ fixed code patterns that are detectable with static analysis [65, 287], unlike software FSMs, which are difficult to detect without complex online tracing tools [88]. In an FSM, a state transforms to another state when certain condition(s) are satisfied. State transitions usually conditionally assign (e.g., an assignment inside a switch case) to FSM variables and include FSM variables as a part of the condition. Additionally, circuits rarely perform mathematical operations (e.g., addition or subtraction) on FSM variables and rarely select individual bits of FSM variables.

Accordingly, FSM Monitor traverses the abstract syntax tree (AST) of a circuit and searches for FSM variables by using the aforementioned heuristics. For each identified FSM variable, FSM Monitor generates Verilog code that displays a log message when the variable is updated.

FSM Monitor's heuristics can incur both false positives and false negatives, but we find a high degree of accuracy in our evaluation (of the 32 manually-identified FSMs in our benchmark suite, FSM Monitor has 0 false positives and 5 false negatives). Furthermore, more sophisticated FSM detection approaches, like those used by the Intel and Xilinx synthesizers, could further increase accuracy. Finally, FSM Monitor allows developers to patch mistakes by adding undetected FSMs and filtering out FSMs that are inaccurate or irrelevant for their current bug.

3.4.3 Dependency Monitor for Provenance Tracking

Our bug study indicates that the only symptom of many hardware bugs is one or more incorrect output values (Table 3.2). Since the root cause of a bug can occur many cycles prior to output generation, it is useful to build the dependency chains for a specific variable and trace updates to variables in the dependency chain during execution.

We therefore build Dependency Monitor to statically analyze the dependencies of a variable and generate the necessary logic to monitor their updates. Dependency Monitor first statically finds all registers that may propagate to a variable v within the previous k cycles (where v and k are specified by the developer). Dependency Monitor then generates logic that logs each update to variables in the dependency chain at runtime.

Dependency Monitor handles partial assignments (i.e., assignment to a strict subset of a variable's bits) by logically splitting a partially assigned variable to multiple variables. Similarly, Dependency Monitor splits constant-indexed arrays into individual variables. If an array is accessed with at least one variable index, Dependency Monitor considers the whole array as an individual register and an assignment to/from the array as a special assignment that only occurs when the index matches. To track dependencies through a blackbox IP, Dependency Monitor requires the developer to provide a model of data and control dependencies within the IP. An IP model describes the relationship between the input signals and the output signals of the IP, which is included in the IP specification and typically well-understood by developers before using an IP. Developers can reuse IP models across projects that share the same IPs.

By default, Dependency Monitor analyzes both control and data dependencies; however, it can be configured to only analyze one type of dependency.

3.4.4 Statistics Monitor for Counting Events-of-Interest

Collecting hardware statistics (i.e., event counters) provides insight into program execution without requiring cycle-by-cycle recording of numerous variables. Furthermore, per-component (e.g., per pipeline stage) counters help a developer localize a statistical anomaly (indicative of a bug) to a small region of a complex circuit.

Accordingly, we propose Statistics Monitor, a tool to help developers collect statistics for events of interest when debugging an FPGA design. Statistics Monitor generates Verilog code that counts occurrences of single-bit signals specified by a developer and adds logging code that emits messages when counts change.

Statistics Monitor is particularly useful when developers suspect that 1) it is too expensive (i.e., with regard to resource consumption) or unnecessary to record all variables of interest on an FPGA deployment (especially cycle-by-cycle), and 2) the bug's symptoms can be inferred via statistical anomalies (e.g., unexpected differences between valid input and valid output counts, indicating potential data loss).

3.4.5 LossCheck for Precise Data Loss Localization

While Statistics Monitor may indicate the presence of data loss (among other bug symptoms) and may localize it to a portion of the circuit, the pervasiveness of bugs manifesting as data loss in our bug study indicates that precise data loss localization would be helpful for hardware debugging.

We therefore design LossCheck, a tool that localizes the root cause of data loss symptoms. A developer specifies a SOURCE register, a SINK register, and a valid signal for SOURCE (§3.2.3). Then, LossCheck instruments the HDL code to monitor the propagation of valid data between SOURCE and SINK. If a valid register is overwritten before its value is propagated from SOURCE to

SINK (i.e., overwritten before being used as a right-hand variable), LossCheck indicates potential data loss.

We note that the tracking of data propagation logic in LossCheck shares similarities with that of Dependency Monitor. However, unlike Dependency Monitor, LossCheck does not yield a trace of updates to variables of interest in a dependency chain. Rather, LossCheck indicates the precise location of a potential data loss. Ultimately, LossCheck's dynamic analysis conveniently enables automatic localization of data loss bugs without recording a large number of data propagation events.

We now describe how LossCheck statically analyzes HDL code (§3.4.5.1), instruments the code (§3.4.5.2), and dynamically detects data loss while mitigating false alerts (§3.4.5.3). We then discuss the limitations of LossCheck (§3.4.5.4).

3.4.5.1 Static Analysis of Data Propagation

LossCheck statically analyzes data propagation in an FPGA design and builds a table of *propagation relations*. It uses these relations to calculate metadata variables that indicate potential data loss (§3.4.5.2).

A propagation relation $X \rightsquigarrow_\sigma Y$ implies that the data value stored in register X will propagate to register Y when the condition σ is satisfied. In other words, the value stored in Y at cycle $k+1$ (i.e., Y_{k+1}) will be influenced by the value stored in X at cycle k (i.e., X_k), if σ is true at cycle k (i.e, σ_k).

At a high level, LossCheck uses logic similar to Dependency Monitor to detect propagation relations and thereby build the propagation relation table. More specifically, LossCheck first identifies a set of data propagation sequences through which a value stored in SOURCE can propagate to SINK. LossCheck then analyzes the control and data dependencies for each register R in the propagation sequences, and adds each identified propagation relation into the table.

We use the following code snippet as a running example of how LossCheck works, where `in` is the SOURCE register, `out` is the SINK register, and `in_valid` is the valid bit for `in`:

```
always @(posedge clk) begin
  // buggy code (b's value can be lost)
  if (cond_a) out <= a;
  else if (cond_b) out <= b;
  if (in_valid) b <= in;
end
```

To analyze the dependencies of b in this example, LossCheck first detects the propagation sequence: `in` $\rightarrow$ `b` $\rightarrow$ `out`. LossCheck then analyzes the dependencies for `b` and `out`, building the following table with 3 propagation relations.

Line	Propagation Relations
3	$\texttt{a} \rightsquigarrow_{\texttt{cond\_a}} \texttt{out}$
4	$\texttt{b} \rightsquigarrow_{\neg\texttt{cond\_a} \wedge \texttt{cond\_b}} \texttt{out}$
5	$\texttt{in} \rightsquigarrow_{\texttt{in\_valid}} \texttt{b}$

Similar to Dependency Monitor, if the source code for an IP is unavailable, LossCheck inserts propagation relations into the table based upon developer-provided IP models.

3.4.5.2 Instrumentation of HDL Code

LossCheck uses the propagation relations to guide circuit instrumentation that enables data loss detection at runtime. The instrumentation process of LossCheck contains two phases: 1) inferring various loss-related metadata for each register in each propagation sequence, and 2) inserting corresponding logic to check for potential data loss via this metadata.

Assignment, Validity, and Propagation Statuses Intuitively, potential data loss occurs when the *assignment* of a *valid* register occurs before its value is *propagated* to another register, thereby overwriting (unused) valid data. So, to detect potential data loss for a register R, LossCheck generates assignment $A(R)$, valid-assignment $V(R)$, and propagation $P(R)$ shadow variables for the register.

For some cycle k, a register's assignment status $A(R)_k$ indicates whether R is assigned a value during cycle k. The value of $A(R)_k$ is inferred at runtime from the propagation relation table. Specifically, $A(R)_k$ evaluates to *true* if at least one register R' propagates its value to R at cycle k. More formally, the condition σ for some propagation relation $R' \rightsquigarrow_\sigma R$ must be satisfied at cycle k.

Similarly, $V(R)_k$ indicates whether R is specifically assigned a *valid* value during cycle k. $V(R)_k$ is therefore determined by combining the logic for calculating $A(R)_k$ with runtime information about data validity. In simple cases (such as our code example), data validity status is trivially available for the variable of interest (e.g., via a corresponding valid signal); in more complex cases, LossCheck calculates validity status for each variable of interest according to the initial input validity value and propagation relations.

Finally, a register's propagation status $P(R)_k$ indicates whether R is used to compute another register's value during cycle k. Similar to how $A(R)_k$ represents assignment *to* register R, $P(R)_k$ represents assignment *from* register R. Thus, $P(R)_k$ evaluates to *true* if R can propagate its value to at least one register R' at cycle k (i.e., if the condition σ for some propagation relation $R \rightsquigarrow_\sigma R'$ is satisfied at cycle k).

After LossCheck determines the values of $A(R)$, $V(R)$, and $P(R)$, it instruments the circuit with the logic to compute the values of these variables at each cycle. Below, we apply these rules to variable b from the original code snippet:

```
1 always @(posedge clk) begin
2   // update shadow vars for next cycle
3   A_b <= in_valid;
4   V_b <= in_valid;
5   P_b <= ~cond_a & cond_b;
6 end
```

Lines 3–5 calculate the values of $A(\texttt{b})$, $V(\texttt{b})$, and $P(\texttt{b})$ for the next cycle based on the propagation relations. We note that, in this example, $A(\texttt{b}) = V(\texttt{b})$ because assignment to b is guarded by the valid signal in_valid.

Inserting Checking Logic Given a register's shadow variables, data loss for register R at cycle k occurs if the following 3 conditions hold: (1) R is assigned at cycle k—i.e., $A(R)_k = \textit{true}$, (2) R is not simultaneously propagated at cycle k—i.e., $P(R)_k = \textit{false}$, and (3) R was assigned a valid value in some previous cycle, which has not yet propagated.

The first two conditions are trivially calculated for R at the current cycle via aforementioned logic. For the third condition, LossCheck keeps track of an additional "Needs-Propagation" variable $N(R)$, which is set to *true* when a valid value is assigned to R and reset to *false* when the value propagates. In mathematical terms, $N(R)_0 = \textit{false}$ (since no valid value has been assigned at cycle 0), and for $k > 0$,

$$N(R)_k = V(R)_{k-1} \vee [N(R)_{k-1} \wedge \neg P(R)_{k-1}] \ . \tag{3.1}$$

Potential data loss at cycle k is then calculated as:

$$\texttt{Loss} = A(R)_k \wedge \neg P(R)_k \wedge N(R)_k \quad . \tag{3.2}$$

Notably, while the shadow variables (i.e., $A(R)$, $P(R)$, and $N(R)$) have a unique value at each cycle, k, LossCheck can detect loss in R at cycle k using only the most recent value of each shadow variable, (i.e.,$A(R)_k$, $P(R)_k$, and $N(R)_k$). Consequently, the amount of state that LossCheck tracks is bounded, so LossCheck can be realized on hardware.

LossCheck generates code that calculates $N(R)$ and checks Equation 3.2. The instrumented circuit that checks for data loss on b is:

```
always @(posedge clk) begin
  // calculate N_b for next cycle from shadow vars
  if (reset) N_b <= 0;
  else N_b <= V_b | (N_b & ~P_b);
```

```
5    // check for data loss at current cycle
6    if (A_b & ~P_b & N_b)
7      $display("LossCheck: potential data loss at b");
8  end
```

Lines 3–4 calculate $N(\mathtt{b})$ for the next cycle according to Equation 1, and Lines 6–7 perform the check for potential data loss at cycle k based on Equation 2.

3.4.5.3 Filtering False Positives and Final Analysis

Notably, LossCheck's design can generate false positives due to an intentional data *drop* (as opposed to an unintentional data *loss*). For example, an FPGA may intentionally drop a network packet input that fails a checksum; LossCheck would flag the packet as data loss. Accordingly, LossCheck uses an FPGA design's test cases—presumably passed during simulation testing— as "ground-truth" test programs; LossCheck suppresses warnings triggered by these test cases. We note that pre-existing test programs for the open-source designs in our study filter 23/24 false positive registers (i.e., those with intentional data drops).

Like the monitors, LossCheck leverages SignalCat to transform the filtered debugging statements (indicating unintentional data loss) into log messages for either simulation or on-FPGA scenarios. Thus, if potential data loss is detected for some register R, a log message indicates R as the source of the loss, and the bug can be precisely localized.

3.4.5.4 Limitations of LossCheck

While LossCheck can accurately localize data losses to a specific register, it cannot distinguish intentional data drops from unintentional data losses. As a consequence, if an unintentional data loss and an intentional data drop occur at the same place, the data loss may be filtered by LossCheck, resulting in a false negative. We identify a single such false negative (out of 7 data loss bugs) in our testbed (§3.6.3).

3.5 Implementation

We build our static analyses using Pyverilog [258], a toolbox for Verilog analysis and instrumentation. We use Pyverilog's dataflow analysis framework to analyze data dependencies and its Verilog code generator to output the instrumented circuit. Furthermore, to analyze circuits developed in SystemVerilog (i.e., an extension of Verilog with more language features), we augment Pyverilog to use the more modern SystemVerilog parser of Verilator [252], a SystemVerilog simulator. Verilator parses SystemVerilog files and performs optimizations such as inline expansion and module instantiation, resulting in an analysis-friendly abstract syntax tree (AST) that Pyverilog can analyze. We modify and add 269 lines of C++ code and 1,750 lines of Python code to integrate Verilator and Pyverilog.

We implement the debugging tools (i.e., SignalCat, FSM Monitor, Dependency Monitor, Statistics Monitor, and LossCheck) as a collection of analysis and instrumentation passes on Pyverilog ASTs. These passes are implemented with 3,797 lines of Python code.

Dependency Monitor and LossCheck require developers to implement a model that describes the relation between the inputs and outputs for each closed-source IP. In our testbed, three IPs are used: `altsyncram`, a block RAM implementation; `scfifo`, a single clock queue implementation; and `dcfifo`, a double clock queue implementation. We implement the models for these IPs in Python and Verilog, resulting in 394 lines of code in total.

3.6 Evaluation

In this section, we first present our testbed (§3.6.1) and experimental setup (§3.6.2). Then, we evaluate the effectiveness of our debugging tools at helping developers debug the bugs in our study (§3.6.3). Finally, we present the resource usage and performance overhead when using the debugging tools to diagnose the study bugs (§3.6.4).

3.6.1 Testbed of Reproducible FPGA Bugs

We built and released a testbed consisting of 20 bugs that we reproduced to facilitate further study of FPGA bugs and FPGA debugging tools [13]. The bugs span the 3 major classes of bugs we identified—data mis-access, communication, and semantic—and multiple development platforms (e.g., Intel HARP and Xilinx). For each bug, we identify the subclass, application, symptom, and the tools that are helpful when debugging each bug, as shown in Table 3.2. The artifact also includes a simplified code snippet for each bug for explanation purposes and provides instructions for reproducing the bug in a push-button manner with the open-source Verilator simulator [252] . Using a simulator eliminates the need for testbed users to spend substantial time and effort acquiring design-specific knowledge that would otherwise be necessary to reproduce each bug.

Although each bug in the testbed is reproducible on real hardware, but, we opt to reproduce the bugs in Verilator for 3 reasons. First, a Verilator-compatible testbed demonstrates that both the fundamental properties of the bugs and the logic of our debugging tools are broadly-applicable in FPGA development. Second, other developers can reason about these bugs and a range of development platforms without purchasing expensive hardware. Third, Verilator simplifies the environmental

ID	Subclass	Application	Platform	Symptom			
				Stuck	Loss	Incor.	Ext.
D1	Buffer Overflow	RSD	HARP	✓	✓		
D2	Buffer Overflow	Grayscale	HARP	✓	✓		
D3	Buffer Overflow	Optimus	HARP	✓	✓		
D4	Buffer Overflow	Frame FIFO	Generic		✓	✓	
D5	Bit Truncation	SHA512	HARP	✓			✓
D6	Bit Truncation	FFT	Generic			✓	
D7	Misindexing	FADD	Generic			✓	
D8	Misindexing	AXI-Stream Switch	Generic			✓	
D9	Endianness Mismatch	SDSPI	Generic			✓	
D10	Failure-to-Update	SHA512	HARP				✓
D11	Failure-to-Update	Frame FIFO	Generic		✓		
D12	Failure-to-Update	Frame FIFO	Generic			✓	
D13	Failure-to-Update	Frame Length Measurer	Generic			✓	
C1	Deadlock	SDSPI	Generic	✓			✓
C2	Producer-Consumer Mismatch	Optimus	HARP	✓	✓		
C3	Signal Asynchrony	SDSPI	Generic			✓	
C4	Signal Asynchrony	AXI-Stream FIFO	Generic		✓		
S1	Protocol Violation	AXI-Lite Demo	Xilinx				✓
S2	Protocol Violation	AXI-Stream Demo	Xilinx				✓
S3	Incomplete Implementation	AXI-Stream Adapter	Generic			✓	

Table 3.2: The testbed of reproducible bugs, including their classes, subclasses, application, platforms, and symptoms. Bug D1–D13 are data mis-access bugs, Bug C1–C4 are communication bugs, and Bug S1–S3 are semantic bugs. A "Generic" platform means that the application does not target on a specific platform and can be synthesized to different FPGAs. For bug symptoms, "Stuck" indicates a symptom of infinite waiting; "Loss' indicates a data loss; "Incor." means the FPGA design gives an incorrect output; and "Ext." means an external monitor (such as an FPGA shell) reports an error.

conditions required to reproduce each bug—crucially, without changing the buggy programs themselves. For the key platform-specific recording IP primitives used by SignalCat (SignalTap [160] and ILA [288]), we provide support for simulating their behavior. Unless specifically mentioned, the buffer size for these data recording IPs is fixed at 8,192 entries.

3.6.2 Experimental Setup

Platform for Overhead Measurement We evaluate the resource and performance overhead of our debugging tools using Quartus 17.0 [40] and Vivado 2020.2 [46], the official synthesizers for Intel's and Xilinx's FPGAs, respectively. We synthesize all Intel HARP-specific designs to the HARP platform [150] (using Quartus), with the remaining designs synthesized to the Xilinx KC705 [48] platform (using Vivado).

ID	Helpful Tools				
	SignalCat	FSM Monitor	Stats. Monitor	Dep. Monitor	LossCheck
D1	✓	✓	✓		✓
D2	✓	✓	✓		✓
D3	✓	✓	✓	✓	✓
D4	✓	✓			✓
D5	✓	✓		✓	
D6	✓			✓	
D7	✓				
D8	✓				
D9	✓				
D10	✓	✓		✓	
D11	✓			✓	
D12	✓			✓	
D13	✓		✓	✓	
C1	✓			✓	
C2	✓	✓	✓	✓	✓
C3	✓				
C4	✓				✓
S1	✓				
S2	✓				
S3	✓				

Table 3.3: The tools used during the debugging process of each bug.

Use Cases We evaluate our tools in two use cases. In the first case, we use SignalCat and the three monitors (FSM Monitor, Dependency Monitor, and Statistics Monitor) to debug all bugs in our study; in the second one, we use LossCheck to localize the source of data loss symptoms for the 7 relevant bugs. Table 3.3 shows the tools used during the debugging process of each bug.

3.6.3 Effectiveness of Debugging Tools

We evaluate the effectiveness of our debugging tools by assessing how much they simplify root cause diagnosis for the bugs in our study (§3.3). An experienced developer could diagnose, localize, and fix the bugs in our study without extra tooling; this is what occurred when these bugs were first reported. But, we find that the localization process is simpler when using our tools. We specifically answer two questions (1) How often is each tool useful when debugging the bugs in our study? and (2) How much work do the debugging tools automate? Additionally, we provide a case study that demonstrates how a developer would use the tools to localize a data loss bug in an Intel HARP application.

SignalCat and Monitors SignalCat is useful for debugging every bug in our study, serving as the fundamental cross-platform logging infrastructure. Each of the 3 monitors assists with debugging at

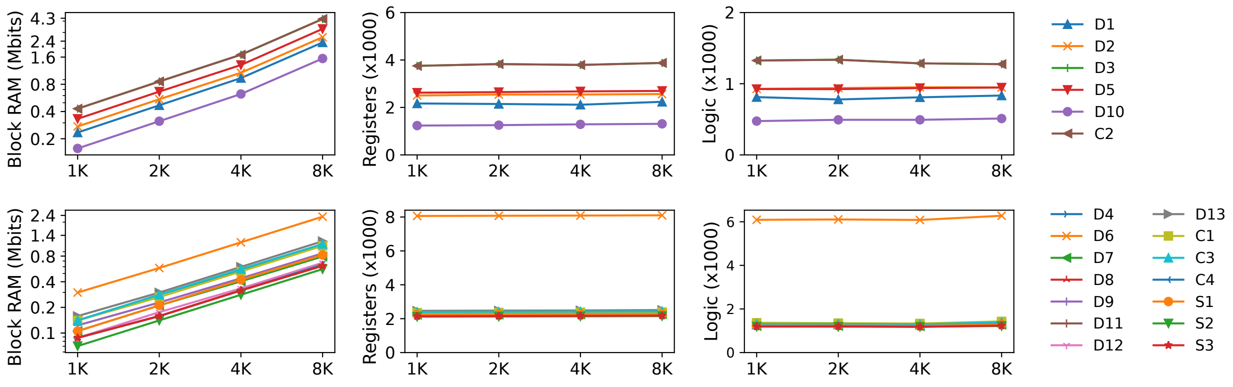


Figure 3.2: The resource overhead of manual debugging using SignalCat, FSM Monitor, Statistics Monitor, and Dependency Monitor on Intel HARP (top) and Xilinx KC705 (bottom) platforms. Resource overheads (y-axes) are shown in terms of block RAM, registers, and logic (i.e., the three types of resources on an FPGA) with an increasing recording buffer size (x-axes). The buffer size and block RAM overhead are shown in log-scale.

least four bugs from the testbed. During debugging (with SignalCat and the 3 monitors), we often find FSM Monitor to be the most helpful in an initial debugging iteration when one or more FSMs were present in the design. Statistics Monitor is generally most usefully deployed in subsequent iterations, where developers try to narrow down the search space of a bug's root cause. Finally, upon encountering a variable with an unexpected value, SignalCat is useful for directly recording updates to the specific variable, while Dependency Monitor supplements this with an analysis of the variable's dependencies. On average, SignalCat and the monitors generate and insert 72 lines of Verilog code to help with root cause localization.

LossCheck LossCheck precisely locates the root cause of data loss (i.e., a specific register) for 6 out of 7 bugs exhibiting data loss (i.e., Bugs D1, D2, D3, D4, C2, and C4) in our study. For 2 of these bugs (D4 and C4), LossCheck uniquely identifies the root cause of the bug without using the false positive filtering technique in §3.4.5.3. For 3 of these bugs (D2, D3, and C2), LossCheck uses the false positive filtering technique to localize the bug without reporting false positives. For the Reed-Solomon decoder buffer overflow (D1), LossCheck reports 1 false positive (i.e., it mistakenly identifies an intentionally dropped register as unintentional data loss), because the developer-provided test case does not perform an intentional data drop at the mis-reported register, so LossCheck does not silence the warning. LossCheck cannot localize the data loss in Bug D11 because the unintentional data loss occurs in a register where the data value may be dropped intentionally under certain conditions; as a result, the data loss is mis-filtered by the LossCheck's false positive filtering. LossCheck generates and inserts 522–19,462 lines of Verilog code to analyze data propagation and detect data loss at runtime, which helps developers avoid the time-consuming manual implementation of data loss checking logic.

Case Study: Debugging Grayscale's Buffer Overflow We describe a case study in which a developer uses the new tools to debug a buffer overflow in the Grayscale application [53]. Grayscale is an end-to-end application written for Intel HARP [150] that includes an FPGA accelerator and a software component. The CPU-side software component reads an image from the file system and programs the FPGA accelerator to read the image from CPU-side memory, perform the grayscale transformation, and write the result back to CPU-side memory. The software component identifies that the acceleration task hangs when the bug occurs.

Grayscale consists of multiple FSMs, so the developer first uses FSM Monitor to identify the state of each FSM when the hang occurs. The developer re-executes the application to trigger the bug. FSM Monitor's output identifies that the accelerator finished reading data from the CPU, since the `read` FSM—which controls how the accelerator reads CPU memory—is in the `RD_FINISH` state. However, the circuit has not finished writing data to the CPU, since the `write` FSM—which controls how the accelerator writes CPU memory—is in the `WR_DATA` state. The developer concludes that the hang occurs in write-related logic.

Next, the developer inspects the state transition logic of the `write` FSM. They find that the state of the write FSM only transfers from `WR_DATA` to `WR_FINISH` after the accelerator writes the whole transformed image to the CPU-side memory. Since the accelerator has already read all data from the CPU (i.e., the `read` FSM is in the `RD_FINISH` state), the hang indicates data loss in the accelerator during the propagation between a memory read and its corresponding memory write.

Finally, the developer uses LossCheck to identify the source of the data loss. They re-execute the application with LossCheck enabled. LossCheck identifies the source of the data loss as a specific register in the accelerator.

3.6.4 Efficiency of Debugging Tools

In this section, we assess the efficiency of the debugging tools by measuring (1) the additional resources consumed when circuits are instrumented using our tools—i.e., the resource overhead, and (2) the necessary clock frequency slowdown stemming from the augmented logic that must execute each cycle—i.e., the runtime performance overhead.

SignalCat and Monitors Figure 3.2 shows the resource overhead (in terms of block RAM, registers, and logic) of SignalCat and the monitors, applied to each buggy design. The most significant resource overhead lies in block RAM usage, which increases linearly as the developer-specified recording buffer size increases. The register and logic overheads tend to be stable for each

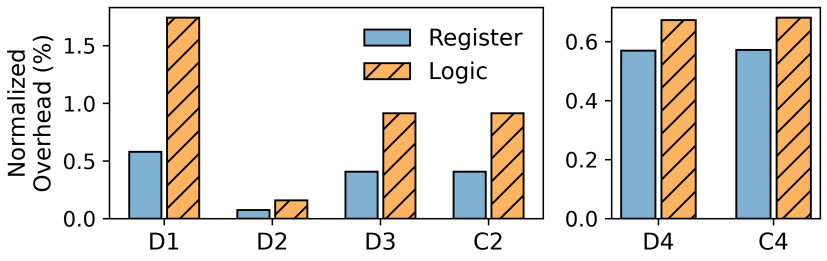


Figure 3.3: LossCheck's overhead in terms of registers and logic, normalized to the total resources available on Intel HARP (left) and Xilinx KC705 (right) platforms.

bug, regardless of the recording buffer size. Among our benchmarks, the two bugs on the Optimus hypervisor and the Bit Truncation bug on the FFT accelerator incur the largest register and logic overheads consuming approximately 0.23% and 0.3% of register and logic resources on the Intel platform (3.08% and 1.99% on Xilinx).

Runtime performance overhead is only incurred for 1 design; namely, Optimus fails to achieve its targeted clock frequency (400 MHz) after the debugging instrumentation. As a result, we reduce its frequency to 200 MHz for debugging. While SHA512 also targets a 400 MHz frequency, it still achieves this frequency after instrumentation. Other designs target a 200 MHz frequency and likewise do not incur performance overhead to account for debugging logic.

LossCheck Figure 3.3 shows LossCheck's resource overhead in terms of registers and logic for the data loss bugs in our study. LossCheck's instrumentation uses less than 1.7% of the total register and logic resources for the four data loss bugs on the Intel platform, and uses less than 0.7% of total resources for the two data loss bugs on Xilinx.

As with SignalCat and the monitors, LossCheck reduces the frequency of Optimus from 400 MHz to 200 MHz. The 200 MHz target frequency of other FPGA designs remain unchanged.

3.7 Related Work

Hardware Bug Studies HardFails [117] performs a bug study of security bugs in CPUs that include real-world and synthetic bugs and creates a testbed by injecting bugs into an open-source CPU design. HardFails only includes security bugs, which are representative of few bugs that make it to production [132]. In contrast, our study examines real-world functionality bugs in FPGA designs.

Simulation-Based FPGA Debugging Developers usually simulate an FPGA design before deploying on-FPGA. Most simulators [252, 257, 277, 47, 43] can generate a waveform—a visualization of signal values—during simulation to aid with debugging. Previous research accelerates simulation-based debugging using language features [226, 108] and by offloading simulation to an FPGA [241, 167, 168] or a GPU [230]. Our debugging tools are designed for both on-FPGA and simulation-based debugging.

Trace-Based FPGA Debugging Trace-based FPGA debugging tools allow developers to collect the value of a selected set of signals in an FPGA deployment. FPGA vendors provide IPs (e.g., Intel SignalTap [160] and Xilinx ILA [288]) that export manual interfaces (e.g., GUIs). To use these tools, developers manually specify the signals that they wish to trace and triggering conditions that should enable tracing output. In contrast, SignalCat automates the selection of signals and corresponding trigger conditions (by statically analyzing "`printf`"-like statements and their path constraints) and provides a natural, vendor-agnostic debugging interface. Prior work reduces the runtime recording overhead of platform-specific IPs by reducing buffer usage [204, 185, 148, 138, 137, 146, 249, 229]; SignalCat can benefit from these optimizations when applicable.

Checkpointing-Based FPGA Debugging Checkpointing-based FPGA tools [80, 79, 180, 241, 196] allow a developer to capture the state of an FPGA deployment for later analysis or debugging, but do not help with localizing the root cause of bugs. Our debugging infrastructure could benefit from similar checkpoint-based functionality.

Synthesizing Traditionally-Unsynthesizable HDL Cascade [241] and Synergy [196] enable traditionally "unsynthesizable" Verilog, including "`printf`"-like statements, to execute on an FPGA. Cascade and Synergy can store arbitrarily-long logs in off-FPGA storage (e.g., in CPU-side memory or disk), but may slow down the circuit since they pause circuit execution when executing "`printf`"-like statements. In contrast, SignalCat offers a different tradeoff: SignalCat imposes lower overhead since it does not pause circuit execution, but can only store limited information since it uses on-FPGA storage (e.g., block RAM).

Interactive FPGA Debugging Interactive FPGA debugging tools allow a developer to interactively manipulate packets in their FPGA's communication channels [211] and provide GDB-like interfaces for FPGA debugging [72]. These tools are useful during simulation but are not applicable for on-FPGA debugging and do not directly help a developer localize the root-cause of a hardware bug.

Traditional Hardware Testing Traditionally, hardware developers implement test suites with industry standard frameworks [57] to extensively test hardware designs in simulation. Hardware fuzzing techniques [195, 263] and formal verification [278, 298, 163, 199, 147, 236, 235] help developers find and eliminate bugs before fabrication, but do not help a developer identify the root-cause of a bug and are resource-intensive. In contrast, our work explores bug localization tools designed for both simulation and on-FPGA scenarios.

Hardware-Assisted Testing and Debugging A plethora of tools [305, 112, 169, 171, 172, 170] have used efficient hardware tracing techniques (typically used in profiling and optimization of hardware/software designs [175, 174, 173]) for testing and debugging. In this paper, we show how reconfigurable hardware can be leveraged to instead design more targeted debugging support by designing and implementing foundational debugging tools. We expect future work to use the reconfigurable nature of FPGAs to design advanced debugging support.

Software Bug Detection at Runtime Our work on FPGA bug localization is inspired by software debugging tools and techniques such as AddressSanitizer [242], ThreadSanitizer [243], Memcheck [245], and dynamic slicing [238]. Particularly, LossCheck's key building block–tracking data propagation dynamically—is closely inspired by such work. Since our own work shows that software techniques are useful for hardware debugging, we believe that the core data propagation logic of LossCheck could be generalized and adapted to other sophisticated FPGA debugging tools.

3.8 Conclusion

The proliferation of reconfigurable hardware has enabled a software-like rapid development cycle in which teams relax verification efforts. While the community has expended effort into bug finding tools (e.g., simulation-based testing tools), very little work has focused on localizing the root cause of hardware bugs. In this work, we performed a study of bugs in open-source FPGA designs and showed that hardware bugs follow a similar taxonomy to software bugs. We argue that hardware bugs are amenable to software-style hybrid static/dynamic program analysis and monitor tools and provide a toolset that aids FPGA debugging and facilitates greater confidence in emerging test-deploy-patch FPGA development cycles.

CHAPTER 4

Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach

4.1 Introduction

In recent decades, the semiconductor industry has made remarkable technological progress. Continuous advancements in process nodes have ensured a consistent downsizing of transistors to nanoscale dimensions, yielding improvements in performance and reductions in energy consumption. However, these advances have also bared circuits to reliability challenges [203, 36], notably evidenced by the emergence of silent data corruptions in data centers [145, 121, 271, 122, 244, 84, 120].

Silent data corruptions, or SDCs, are a form of undetected failures that occur without generating logs, exceptions, or causing immediate program crashes. Instead, they silently introduce incorrect data into applications. As a result, the error can spread far from its point of origin, potentially leading to failures that are difficult to predict, prevent, and troubleshoot. Recently, cloud providers have identified CPUs with such SDCs in their data centers [145, 121, 271].

SDCs are risky, as they challenge the fundamental "fail-stop" assumption of hardware failures that software developers have been accustomed to for decades. Most software applications in data centers and personal computers assume that a circuit, having undergone correct design, fabrication, and testing, will either function correctly or not work at all. Unfortunately, SDCs usually involve hardware malfunctions like miscomputing instructions and broken cache coherency, which are typically not considered by applications. Worse, these faults may be transient or persistent, which increases the difficulty of monitoring and mitigating SDCs.

Transistor aging is believed to be one of the causes of SDCs [60, 73, 69, 71]. This gradual performance degradation of transistors over time steadily increases signal propagation delays. Eventually, this aging results in timing violations inside a circuit, thus causing certain components to malfunction. Alibaba observed that a significant portion of SDCs in their CPUs appear only after a period of usage [271], suggesting that these SDCs may be attributed to transistor aging.

A straightforward strategy for mitigating the risks associated with SDCs is to conduct frequent and proactive testing using well-designed test cases, therefore enabling quick identification and removal of malfunctioning hardware. Following this strategy, previous research has explored the formulation of test cases specifically for the detection of SDCs, as well as the development of frameworks for test management and scheduling. However, in order to stress individual components inside the circuit, these works tend to create complex tests with a long execution time, preventing them from being frequently scheduled. For example, in Alibaba, such tests are only scheduled once every three months [271].

The detection of SDCs—especially these attributed to aging—would be markedly improved by increasing testing frequency. Transistor aging occurs progressively and a circuit may exhibit SDCs at any stage of its lifecycle, so more frequent testing helps ensure more timely detection. Ideally, test cases should be selectively integrated within applications, guaranteeing routine execution and enabling immediate error-handling for detected SDCs. However, such an integration is traditionally impractical because of the long execution times associated with SDC tests developed by previous work. For example, Google's SiliFuzz [244] generates around 500,000 test cases, and a full execution of DCDiag [41]—Intel's official CPU diagnosis tool—takes 45 minutes.

Previous research tends to yield tests with a long execution time because of their adoption of a "top-down" approach. These works treat the hardware as a black box and generate test cases atop an abstract model of it. For example, Google's SiliFuzz [244] generates test cases by fuzzing the instruction set architecture (ISA) of a CPU, while Intel designed OpenDCDiag [28]—an open-source variation of DCDiag [41]—on top of popular libraries such as zlib [29] and eigen [140]. Due to the lack of implementation details in the abstract hardware model, these methods must generate a set of complex tests to ensure that all components inside the hardware are stressed.

However, hardware is not a black box. Circuit design is detailed in hardware description languages (HDLs) and subsequently synthesized into a netlist, which comprises a complex placement of gates and wires. This netlist serves as a blueprint during chip fabrication. Unsurprisingly, as a dominant factor in circuit reliability, transistor aging has been extensively studied at the level of gates and netlists [60, 219, 220, 83, 143]. Particularly, prior research has identified key electrical effects that contribute to transistor aging and developed comprehensive models to estimate these effects [63, 133].

In this paper, we present Vega, a novel "bottom-up" workflow designed to bridge the gap between the gate-level understanding of transistor aging and the proactive detection of aging-related SDCs in software. Vega empowers frequent and routine detection at application runtime, thereby improving the effectiveness of transistor aging failure detection. Specifically, Vega is comprised of three phases:

1– Aging Analysis identifies susceptible signal propagation paths that can potentially fail due to transistor aging. This is achieved through the use of aging-aware static timing analysis, supplemented by the well-studied gate-level models for transistor aging [63, 133].

2– Error Lifting transforms aging-prone paths into short test cases that are executable in a software environment and integrable into an application. This conversion leverages a combination of formal methods, logical modeling for timing errors, and heuristics based on the hardware's microarchitecture to ensure precise test case generation. As a byproduct, this phase additionally yields a number of failure models for the analyzed hardware, which can be valuable for future research in circuit and software reliability.

3– Test Integration combines test cases with an application. We showcase two approaches for such integration: a profile-guided method for automated test instrumentation, and a manual method for a more controlled integration.

We demonstrate Vega on the arithmetic logic unit (ALU) and the floating-point unit (FPU) of a RISC-V CPU, synthesized into a 28nm cell library. We show that Vega can identify aging-susceptible signal paths and generate effective test cases to target faults arising from them: these test cases incur negligible runtime performance overhead while ensuring routine aging detection.

Overall, we make the following contributions:

- We design Vega, a novel workflow that bridges the gap between the physical understanding of transistor aging and the proactive detection of aging-related SDCs in software.
- We evaluate Vega with a circuit synthesized into a real-world cell library, demonstrating the capability of frequent aging-related failure detection with negligible runtime overhead.
- We provide a set of circuit-level failure models for the analyzed hardware to facilitate future research into silent data corruptions.

4.2 Background and Motivation

This section begins with context about recent observations of silent data corruptions (SDCs) on data center-deployed hardware circuits (§4.2.1). Next, we summarize the development process of such circuits (§4.2.2), and explore how transistor aging—a common cause of SDCs—can impact the performance and reliability of hardware circuits (§4.2.3). After each subsection, we present key takeaways that motivate Vega.

4.2.1 Silent Data Corruptions

Silent data corruptions (SDCs) are a form of undetected failure that silently introduces incorrect data into applications. These occur without generating logs, triggering exceptions, or causing immediate program crashes. Consequently, SDCs can propagate beyond their point of origin, leading to issues that are challenging to prevent, predict, or troubleshoot.

Recently, major data center operators, including Meta, Google, and Alibaba, have reported incidents of SDCs within their clusters [145, 121, 271]. Investigations into these SDCs uncovered that they stem from faults within computational circuits (i.e., CPUs), rather than the more typical suspect, memory devices. Occasionally, a CPU, despite having been correctly designed, fabricated, and tested, may still consistently produce incorrect results during certain operations, leading to various misbehaviors including miscomputing instructions and disruptions in cache coherency.

SDCs may manifest at any point during the lifecycle of a circuit, but only a limited subset of them can be detected during factory testing. Alibaba's data indicates that a significant 73.5% of the SDCs they identified occur in CPUs that have already been in use, either during system re-installations (63.9%) or while in production (9.6%) [271].

Currently, data center operators detect SDCs through extensive, long-running test cases that proactively stress the underlying hardware. However, given the low probability of SDC occurrence, it is impractical to run these tests frequently. For instance, in Alibaba's data centers, these tests are conducted only once every three months [271]. Consequently, there is a growing need for a more efficient and practical mechanism to identify SDCs.

Takeaway #1. Increasing the frequency of SDC testing can lead to more timely detection of SDCs. Shrinking the size of the test cases can make frequent testing more practical.

4.2.2 Hardware Development

Digital circuits such as CPUs and GPUs are initially developed in hardware description languages (HDLs) like Verilog, SystemVerilog, and VHDL. HDLs empower developers to precisely describe the functionality of each component in a hardware design. Functionalities can then be tested through circuit simulations, allowing developers to verify and debug a design before it progresses to the physical fabrication stage.

Subsequently, in a process akin to software compilation, a hardware synthesizer transforms the circuit's functional description from an HDL into a netlist. The netlist is structured as a directed graph comprised of a large number of *cells* from a standard cell library, with wires that describe the electrical connections between the cells. These libraries, provided by chip manufacturers, describe the functionality and timing behavior of predefined circuit components such as logical gates and flip-flops, allowing the synthesized design to be practically implemented in hardware.

Following circuit synthesis, the hardware design progresses to a stage known as place-and-route. This stage involves strategically positioning cells into designated locations on a silicon die, and creating the wires that interconnect these cells. Moreover, it also ensures the clock signal reaches all parts of the chip in a timely and synchronized manner, which is crucial for the proper functioning of synchronous logic. Additionally, static timing analysis (STA) is employed to evaluate the compliance of digital signals with timing constraints, thereby determining the circuit's maximum operating frequency and ensuring its reliable operation.

Takeaway #2. Hardware is not a black box: its implementation details can provide insights that guide SDC detection.

4.2.3 Transistor Aging

Transistor aging, which is believed to be a significant cause of SDCs [60, 73, 69, 71], refers to the gradual degradation of the performance and reliability of transistors over time. This process leads to an increase of a transistor's threshold voltage, which in turn causes a higher switching delay. As a result, signal propagation through an aged circuit may take longer than anticipated, potentially violating the circuit's timing constraints and resulting in malfunctions.

In modern circuits, this aging process predominantly stems from a physical phenomena called *bias temperature instability* (BTI), occurring when a static voltage is applied to a transistor for a long period of time [63, 190]. In other words, when a transistor remains in a constant state without regular switching, it is more likely to experience aging.

4.2.3.1 The Nonuniform Nature of Transistor Aging

Transistor aging in a circuit is a nonuniform process [133], and several factors vary degradation rates. A key factor is the different BTI stress each transistor experiences during operation. Rarely-used circuit components tend to have more transistors idling in a fixed state, increasing their vulnerability to BTI effects. This variation is heightened in CMOS-based technologies due to their inherent design and operational characteristics. Specifically, as p-type transistors are more susceptible to BTI effects than n-type transistors, logical gates that consistently idle in a "0" state tend to age faster than those that idle in a "1" state or that switch regularly.

There are several additional causes of non-uniform transistor aging [134]. For example, clock gating, a standard power-saving technique, has been identified as a primary cause of uneven transistor aging. Clock gating inadvertently introduces varying levels of BTI stress across different areas of the clock network. Therefore, it leads to different aging rates in different regions of the network.

4.2.3.2 Timing Violations Caused by Transistor Aging

For a circuit to operate correctly, it is crucial that signals comply with their timing constraints, reach their intended destinations, and remain stable within a critical time interval, as shown in Figure 4.1.

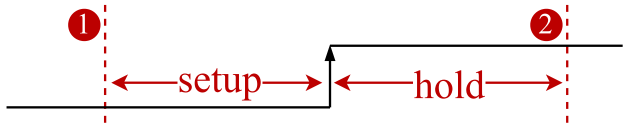


Figure 4.1: The setup and hold windows of a clock edge. Signals should arrive at its destination flip-flop before ❶ and hold stable until ❷.

Unfortunately, an aged circuit may potentially breach these requirements, leading to two types of timing violations: *setup violations* and *hold violations*.

A setup violation happens when a signal arrives at a flip-flop too late (i.e., after ❶), failing to meet the required setup time before the clock edge. In contrast, a hold violation happens when a signal changes too soon (i.e., before ❷), failing to hold stable for the required window after the clock edge. Both setup violations and hold violations can result in incorrect data being captured by the flip-flop, thereby causing circuit malfunctions. When this arises due to transistor aging in a previously-working circuit, this can result in SDCs or other application-level misbehavior. While setup violations can be addressed by lowering the clock frequency, this approach is ineffective for hold violations, as the clock frequency does not affect the required hold time. Consequently, hold violations are considered more severe than setup violations, as they necessitate chip repair.

4.2.3.3 The Physical Model for Transistor Aging

The reaction-diffusion model, widely accepted for transistor aging, effectively describes the increase in a transistor's threshold voltage under BTI stress [62, 61, 89, 97]. With this model, the threshold voltage increase of a transistor, denoted as ΔV_{th}, can be determined via the following equation:

$$\Delta V_{\text{th}} \propto e^{\frac{E_a}{kT}} (t - t_0)^{1/6}, \tag{4.1}$$

where E_a is a constant related to process technology, T is the operating temperature, k is Boltzmann's constant, and $t - t_0$ represents the duration for which the transistor undergoes stress due to BTI effects.

Using this equation, we can calculate the changes in a transistor's threshold voltage based on the duration of its exposure to BTI stress. Once the stress is removed, some of the degradation can be reversed, and a similar equation can be employed to quantify the recovery process.

4.2.3.4 Profiling BTI Stresses with Signal Probability

A common method for profiling the BTI stress of a logical element is to use *signal probability* (SP). SP calculates the probability of a signal being in the logical "1" state, determined by the ratio of

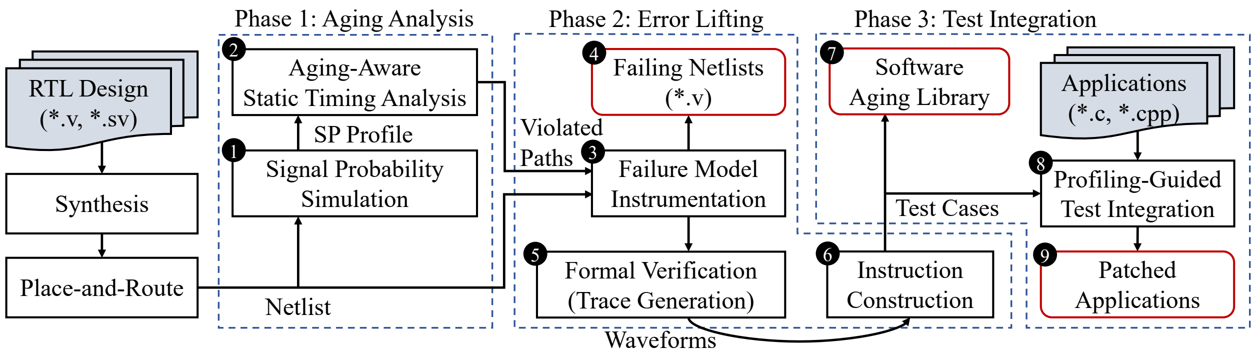

Phase 1: Aging Analysis
Phase 2: Error Lifting
Phase 3: Test Integration
RTL Design (*.v, *.sv)
Synthesis
Place-and-Route
Netlist
2 Aging-Aware Static Timing Analysis
SP Profile
1 Signal Probability Simulation
Violated Paths
4 Failing Netlists (*.v)
3 Failure Model Instrumentation
5 Formal Verification (Trace Generation)
Waveforms
7 Software Aging Library
6 Instruction Construction
Test Cases
Applications (*.c, *.cpp)
8 Profiling-Guided Test Integration
9 Patched Applications

Figure 4.2: Overview of Vega's workflow, comprising three key phases: Aging Analysis, Error Lifting, and Test Integration. Each step in the workflow is outlined with a black box, with the inputs enclosed in gray boxes and the outputs in red boxes.

the time spent in the "1" state to the total time. For example, an SP of 0.25 means the signal is in the logical "1" state for 25% of the time (while in "0" state for the remaining portion of time). Usually, an SP profile is gathered by conducting functional simulations for the circuit, using a set of representative workloads that the circuit is expected to process.

With a given SP profile, we can calculate the ΔV_{th} for each transistor within a given cell (e.g., a `NOT` gate) using the reaction-diffusion model. Subsequently, analog simulation techniques, e.g., SPICE [218], can be employed to determine the change in the cell's switching delay. This change can then be considered in static timing analysis to identify timing violations that may occur after the impact of transistor aging.

Takeaway #3. Signal probability profiles, along with aging-aware static timing analysis, can help identify logical elements prone to timing violations and potential SDCs, thereby producing effective targets for test cases.

4.3 Design of the Vega Workflow

Vega leverages the takeaways from the previous section to *enable frequent and proactive detection of aging-related SDCs at application runtime*. It targets a CPU's critical functional units, such as ALU and FPU, and generates software-executable instructions to test them. Specifically, Vega adopts a bottom-up approach, where precise test cases for likely aging-related faults are crafted by analyzing the detailed implementation of the CPU. As a result, Vega yields a sufficiently compact set of test cases that can be seamlessly integrated into an application's runtime. This targeted approach makes it practical to conduct frequent and timely detection of aging-related SDCs.

Figure 4.2 illustrates the workflow of Vega, which is composed of three distinct phases. In the first phase, *Aging Analysis*, Vega identifies locations within a circuit that are most vulnerable to transistor aging. These locations are potential origins of SDCs that Vega aims to detect. To precisely identify the locations, Vega simulates the circuit with a set of representative workloads, and utilizes the gate-level modeling of transistor aging to determine each components' timing degradation.

In the second phase, *Error Lifting*, Vega transforms the potential timing violations identified by the first phase into tiny test cases, consisting of several instructions that are executable in a software environment. This transformation is achieved in two steps. First, Vega employs formal verification techniques to create a cycle-accurate trace of module-level input that is capable of triggering a specific timing violation inside the CPU. Then, it leverages heuristics based on the CPU's microarchitecture to transform this trace into a sequence of instructions, which is carefully crafted to activate the same signals as outlined in the trace. As a byproduct, this phase also yields a set of circuit-level failure models. These models, formatted as gate-level netlists, simulate failures and describe the possible misbehavior of the CPU as it ages.

In the third and final phase, *Test Integration*, Vega performs application-level integration of the previously-generated test cases. We implement two approaches for such integration, enabling different degrees of control over test invocation and allowing test integration without code modification. The first approach produces a software library supporting different strategies of transistor aging detection and response, along with wrappers compatible with various programming languages. The second method minimizes a developer's integration effort by employing profile-guided techniques to embed these test cases directly into an application, while incurring minimal performance overhead.

In the rest of this section, we explain the design details of these three phases using an example circuit.

4.3.1 Preparation for the Workflow

Consider the hardware module presented in Listing 2, written in System Verilog, as a representative example. This module implements a pipelined 2-bit adder that calculates the sum of two 2-bit integers, denoted as `a` and `b`. The computation is segmented into two cycles. In the first cycle, `a` and `b` are sampled in `aq` and `bq`, respectively (Line 5). In the second cycle, the sum of `aq` and `bq` is calculated, with the result stored in `o` (Line 6).

As we described in Section 4.2.2, this module will proceed through a sequence of processes during development, including circuit synthesis and place-and-route, which eventually transform the circuit into the netlist illustrated in Figure 4.3. For simplicity, we exclude components used solely for timing correction, such as clock buffers, and employ a minimal standard cell library. This library consists of three cell types: `AND` and `XOR` cells, which respectively perform the "and" and "xor" operations on their inputs, and the `DFF` cell, a D-type flip-flop that registers its input `D` for one clock cycle. For our example, we also assume the maximum propagation delay of the `AND`, `XOR` and

```
module adder (clk, a, b, o);
  input wire clk; input wire [1:0] a, b;
  output reg [1:0] o; reg [1:0] aq, bq;
  always @(posedge clk) begin
    aq <= a; bq <= b;
    o <= aq + bq;
  end
endmodule
```

Listing 2: An example hardware module.

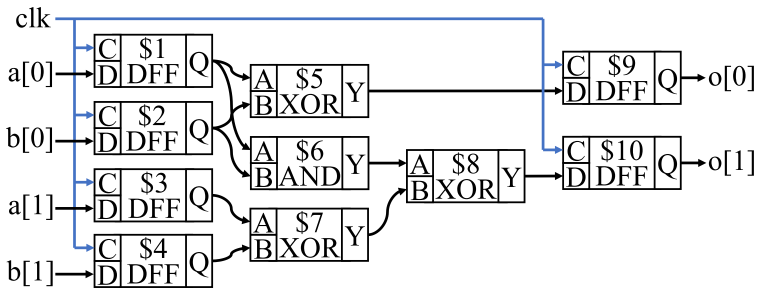


Figure 4.3: The netlist associated with Listing 2. Components used for timing correction (e.g., clock buffers) are excluded.

DFF cells is 0.3ns, and the minimum delay is 0.1ns. The DFF cell requires a setup time of 0.06ns and a hold time of 0.03ns. Additionally, the module targets an operation frequency of 1GHz; therefore, each cycle spans 1ns.

Our example netlist satisfies the cells' timing constraints: The longest path ($4 → $7 → $8 → $10) accumulates a maximum delay of 0.9ns, indicating signal arrival at $10 more than 0.06ns (setup time) before the next clock edge. Conversely, the shortest path ($1 → $5 → $9) has a total minimum delay of 0.2ns, ensuring $9's input is stable for greater than 0.03ns (hold time) after the clock edge. However, transistor aging may disrupt these constraints. To evaluate aging's impact on the circuit, the netlist is forwarded to the Aging Analysis phase for further examination.

4.3.2 Aging Analysis

In the Aging Analysis phase, Vega focuses on identifying where transistor aging is likely to impact a hardware circuit, by identifying signal propagation paths that will potentially violate timing constraints after experiencing realistic transistor aging. First, Vega instruments the circuit's netlist and simulates a set of representative workloads on it (➊). For each cell in the design, we record the *signal probability*, representing the likelihood of a signal being in a given logical state, which may correlate with its susceptibility to BTI. This profile is then consumed by an *Aging-Aware Static*

Signal	SP	Signal	SP	Signal	SP
`DFF$1.Q`	0.85	`DFF$2.Q`	0.54	`DFF$3.Q`	0.38
`DFF$4.Q`	0.27	`XOR$5.Y`	0.46	`AND$6.Y`	0.48
`XOR$7.Y`	0.13	`XOR$8.Y`	0.52	`DFF$9.Q`	0.44
`DFF$10.Y`	0.54				

Table 4.1: An SP profile associated with the netlist in Figure 4.3.

Timing Analysis to determine the most likely locations of future timing violations (❷).

4.3.2.1 Signal Probability Simulation

To determine which of a circuit's cells will experience the largest effect from transistor aging, Vega needs to estimate the likelihood that a cell will rest at different signals during its lifetime. Vega does this through simulation of an instrumented circuit. It attaches a counter to the output port of each cell in the circuit's netlist, recording the incidence of logical "0" and "1" states from that cell. For our example circuit, these counters attach to the `Q` port for `DFF` cells and the `Y` port for `AND` cells and `XOR` cells. Notably, these counters are driven by a separate free running clock generated by Vega. Vega assures this clock continues toggling even if the clock within the circuit is paused.

Vega then simulates the instrumented circuit, post-place and route, with an HDL simulator and a set of representative workloads. After the simulation is completed, Vega aggregates the values of each cell's counters to determine what fraction of the time the cell remained at logical "1": together, this forms a signal probability (SP) profile for the circuit. Table 4.1 shows an example SP profile corresponding to a simulation of the netlist in Figure 4.3. Notably, the `XOR` cell `$7` has a particularly extreme SP value; therefore, it is under the highest BTI pressure and more susceptible to transistor aging. The `clk` signal is omitted from this example, because clock distribution in a placed and routed design involves the use of numerous clock buffers. In a real-world SP profile, each of these clock buffers is profiled individually.

4.3.2.2 Aging-Aware Static Timing Analysis

Vega can now identify timing violations that may emerge in the circuit due to transistor aging. By harnessing the SP profile generated in the last step along with an *aging-aware timing library*, Vega can quantify the performance degradation of each logical cell in the circuit. This timing library characterizes how signal probability affects a cell's timing characteristics, such as maximum and minimum propagation delay, over a period of time. Vega generates this library by conducting analog simulation with SPICE [218] for each cell in the standard cell library, determining how changes in a cell's physical property correspond to changes in its timing characteristics. Notably, as multiple circuit designs may use the same standard cell library, this work is pre-computed to

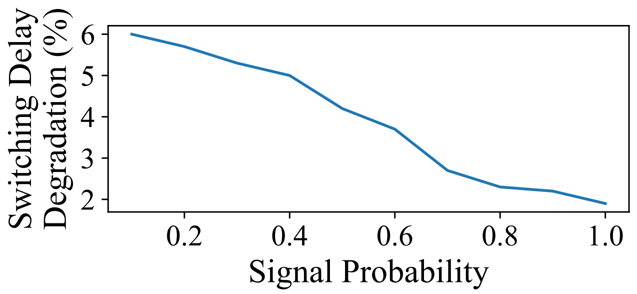


Figure 4.4: The switching delay degradation of a 28nm `XOR` cell under different levels of SP over a 10-year period.

accelerate Aging-Aware Static Timing Analysis (STA). Figure 4.4 shows an example entry from a pre-computed timing library, showcasing the speed degradation of a typical `AND` cell.

Once the aging-aware timing library is generated, Vega looks up cell data by their signal probabilities and updates the timing characteristics of the netlist under test. Vega performs static timing analysis to identify signal propagation paths that exhibit timing violations. These paths are considered aging-prone, with a higher risk of experiencing timing violations due the impact of transistor aging. The Aging-Aware STA is based on a 10-year assumed lifetime that is commonly adopted by mission critical systems [111]. Notably, during the Aging-Aware STA, Vega also analyzes the effect of aging on the clock distribution network. This analysis can reveal phase shifts of the clock signals in different locations, which could potentially lead to hold violations.

Based on the SP profile in Table 4.1 and the timing library demonstrated in Figure 4.4, Vega finds that the propagation delay of the path `$4` $\rightarrow$ `$7` $\rightarrow$ `$8` $\rightarrow$ `$10` accumulates to 0.946ns after considering transistor aging. Therefore, it violates the required setup window (0.946ns ¿ 1ns - 0.06ns) and incurs a violation. For demonstration purpose, we also assume that a phase shift is detected between the clock signals connected to `DFF $1` and `DFF $9`, causing a hold violation in path `$1` $\rightarrow$ `$5` $\rightarrow$ `$9`. These violating paths are provided for the next phase, Error Lifting, to help test case formulation.

4.3.3 Error Lifting

In the Error Lifting phase, Vega formulates test cases for aging-related hardware faults, targeting these tests to the aging-sensitive signal paths identified in the previous phase. Vega crafts test cases in two steps. First, it instruments the hardware module's netlist with a failure model propagating the effect of a previously-identified timing violation (Figure 4.2, ❸). Then Vega uses a formal verification tool to produce a sequence of cycle-accurate, module-level inputs that provokes the failure (❺). These inputs, represented in a hardware waveform, are constrained to ensure an incorrect value will be generated in the module's output. Then, Vega processes and analyzes the

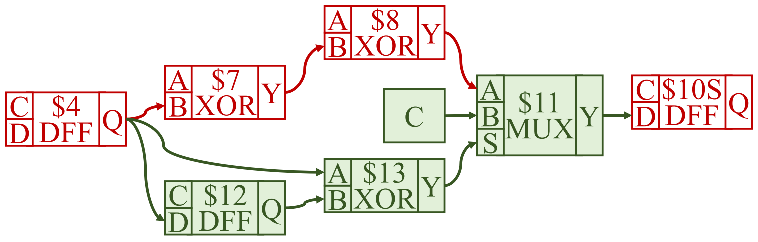


Figure 4.5: Failure model for a setup violation in the path `$4` → `$7` → `$8` → `$10`, with red highlighting the failing path and green indicating the instrumented cells. Unrelated signals are omitted for clarity.

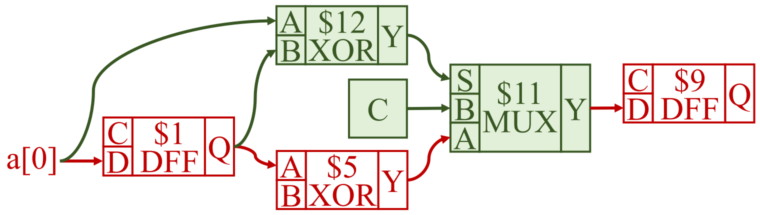


Figure 4.6: Failure model for hold violation path `$1` → `$5` → `$9`.

input sequence to generate a series of software-executable instructions that activates the exact waveform inside the circuit (❻).

We opt for formal verification in test case generation because it provides a systematic way to explore infrequently used components inside the circuit. Formal verification enables us to create a concise set of test cases that target specific potential failures inside the circuit, therefore avoiding the long execution time associated with tests crafted from top-down approaches like fuzzing.

However, since formal verification only operates within the logical domain and does not consider timing violations, we need to introduce a model that logically describes the misbehaviors associated with these timing violations. Moreover, formal verification proves more practical at the scale of individual hardware modules (e.g., an FPU), rather than an entire hardware design (e.g., a CPU) with cache, memory, and software running on top of it. Consequently, we choose to only apply formal verification on specific hardware modules, and then construct the instructions by leveraging our understanding of the hardware design's microarchitecture.

4.3.3.1 Logical Models for Timing Violations

As we described in Section 4.2.3, transistor aging may cause setup violations and hold timing violations. During a setup violation, the signal reaches the flip-flop too late, failing to meet the setup window. As a result, the flip-flop may sample an incorrect value during the clock tick. However, it is important to note that even if a signal path fails to meet its required setup time, the flip-flop may still sample the correct value in some cycles, provided the values previously held in the path

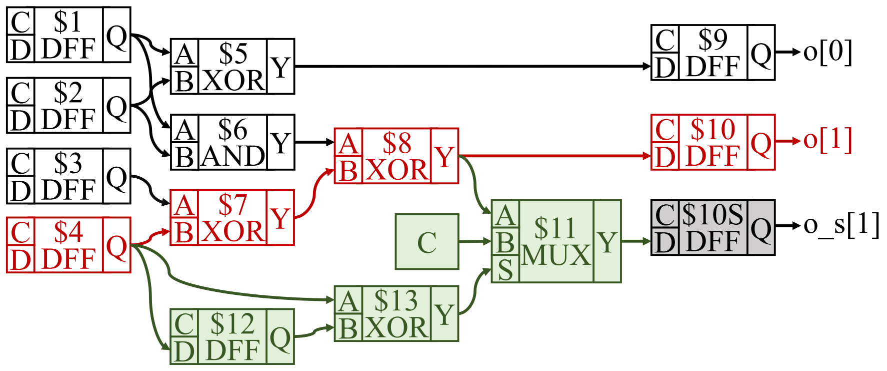


Figure 4.7: The netlist instrumented with the shadow replica (in gray) and the failure model (in green).

remain unchanged [228]. As a result, for a signal path between a pair of DFFs $X \rightsquigarrow Y$ that violates its required setup time, Y's output at cycle $t+1$ is logically modeled as:

$$Y(t+1) = \begin{cases} Y_{original}(t+1) & \text{if } X(t) = X(t-1) \\ C & \text{otherwise} \end{cases}, \tag{4.2}$$

where $Y_{original}$ represents the value of Y assuming no violation occurs, and C denotes the wrong value sampled by Y when the timing violation occurs. For formal verification, C is set to a constant value—either 0 or 1—to limit the search space the formal verification tool is required to explore. Nonetheless, the tool can conduct separate rounds of verification for each case of C, allowing Vega to generate test cases for different scenarios that could occur in an actual circuit.

Similarly, in a hold violation, the flip-flop may still sample a correct value by chance, as long as the values in the path are not changing for the next clock cycle. Therefore, for a signal path $X \rightsquigarrow Y$ that violates its required hold time, Y's output is modeled as:

$$Y(t+1) = \begin{cases} Y_{original}(t+1) & \text{if } X(t) = X(t+1) \\ C & \text{otherwise} \end{cases}. \tag{4.3}$$

In the special case where the path starts from and ends at the same flip-flop, we consider Y to always produce the value C. This approach is adopted because, in these situations, the value captured by Y relies on its own value in the same cycle. As a result, Y will consistently be in a meta-stable state.

4.3.3.2 Failure Model Instrumentation

To integrate this logical model of timing violations into the circuit's netlist, we introduce a `MUX` cell. `MUX` functions as a selector, outputting either of its inputs, `A` or `B`, depending on the value of a select signal, `S`. Figure 4.5 shows the instrumented failure model for the setup violation occurring in path

`$4` (X) $\rightsquigarrow$ `$10` (Y). In this instrumentation, `DFF $12` is used to retain the output value of `$4` for a cycle, thereby allowing $X(t) = X(t-1)$ to be calculated. Similarly, Figure 4.6 shows the failure model for the hold violation in path `$1` (X) $\rightsquigarrow$ `$9` (Y). In this instrumentation, $X(t+1)$ is derived from the input of `$1`, since `$1` is a `DFF` and its input value will be output in the next cycle.

Failure Model Instrumentation can function in one of two modes. In one mode, the instrumentation phase generates a *failing netlist*, which is a Verilog file that describes the behavior of the circuit component after the impact of transistor aging (❹). This Verilog file can be synthesized for a range of targets, including simulation environments and FPGAs, rendering it useful as a circuit-level failure model in future reliability research. Additionally, this circuit-level failure model enables us to validate the effectiveness of test cases constructed by Vega, as detailed in Section 4.5.2.2.

Alternately, the instrumentation phase can prepare the netlist for trace generation to support the crafting of targeted test cases for the modeled failure. Instead of directly integrating the failure model into the netlist, Vega first generates a shadow replica for a portion of the netlist. Specifically, for an aging-prone path $X \rightsquigarrow Y$, the instrumentation conducts a static analysis of the circuit and identifies all cells that can potentially be influenced by Y—this includes Y itself. Based on this analysis, it creates copies of these identified cells, with the same interconnections between copied cells that the original cells have. The failure model is then integrated into this shadow replica, so the module-wide effects of a targeted timing violation can be tracked.

Figure 4.7 shows the instrumentation for the setup violation path `$4` $\rightsquigarrow$ `$10`. As shown in the figure, a shadow cell, `$10S`, is created, with its input linked to the failure model and its output to a shadow wire named `o_s[1]`. In the subsequent step, this shadow wire will be used by the formal verification tool to hint the generation of module-level inputs.

A similar instrumentation will be generated for the hold violation path `$1` $\rightsquigarrow$ `$9`. In this case, a shadow replica will be created for cell `$9`, with its output linked to a shadow wire.

4.3.3.3 Trace Generation using Formal Methods

After the shadow replica is created and connected to the failure model, Vega incorporates a formal verification tool to produce a sequence of module-level inputs that provokes the instrumented failure. Specifically, it formulates a *cover property*—a System Verilog primitive—that requires that the value in the shadow replica differs from the values in its corresponding original copy. For example, Vega generates the below property for the instrumented netlist in Figure 4.7:

```
cover property (@(posedge clk) o[1] != o_s[1]);
```

Formal verification tools are designed to interpret such properties and synthesize a sequence of inputs that ensures the expression in the property evaluates to true for at least one cycle. In the case where the constant C is set to 1, the formal verification tool finds the trace described in

Cycle	1	2	3
`a[1:0]`	`'b01`	`'b11`	`'b11`
`b[1:0]`	`'b11`	`'b00`	`'b01`
`o[1]`	`'b0`	`'b0`	`'b0`
`o_s[1]`	`'b0`	`'b0`	`'b1`

Table 4.2: An example trace that provokes the instrumented failure in Figure 4.7. `o[1]` and `o_s[1]` mismatch at cycle 3.

Table 4.2. When the circuit's input signals align with the trace, the expression in the cover property will evaluate to true at cycle 3. The trace is then captured and saved as a waveform, which proceeds to step ❻ for instruction generation.

In some scenarios, it is necessary to apply extra restrictions on the module's input to prevent unrealistic traces from being generated. These restrictions are described using the *assume property* primitive of System Verilog, and writing useful restrictions requires that developers have some knowledge of the target microarchitecture's behavior. For instance, when analyzing a hardware module like an ALU, we may restrict the range of input to include only valid operations.

4.3.3.4 Mitigation for Initial Value Dependency

In some instances, the traces produced by the formal verification tool may not reliably trigger failures in a real-world execution. This issue occurs because the tool assumes that the circuit's initial state has been perfectly reset. Specifically, the tool first simulates the circuit's reset behavior to obtain the initial values for each signal within the circuit, before beginning its symbolic exploration of the design. Consequently, it may generate traces that are effective only under these specific initial values. However, in a real-world execution, these initial values may be modified by a previous instruction, potentially making the generated trace ineffective.

To mitigate this issue, Vega allows configuring the failure model to activate only when detecting a rising or falling edge in the value of X (i.e., the starting point of the violated path). For example, in Figure 4.7, it may replace cell $13 with logics that determines ¬`$12.Q`∧`$4.Q` (i.e., a rising edge) or `$12.Q`∧¬`$4.Q` (i.e., a falling edge).

4.3.3.5 Instruction Construction

Reliable activation of aging-related failures is now brought up to the software level in this step, which aims at generating a sequence of instructions that can activate the module's input signals as described in the trace. Vega leverages expert knowledge of the CPU's microarchitecture to achieve this generation. This step is the most labor-intensive part of Vega, but only has to be done once:

For each CPU microarchitecture and hardware component under analysis, developers must write a dedicated script to facilitate instruction construction.

To create this script, developers use their understanding of how each instruction activates signals in the analyzed hardware component; therefore, they can create a look-up table that reverse-maps the activation of signals to an instruction. In some cases, additional steps like mapping constant values to specific registers are necessary.

During instruction construction, Vega determines the values of the input registers and the expected value of the output registers. However, the allocation of these registers is deferred to the next phase (i.e., Test Integration), to allow a more seamless integration of test cases with applications.

4.3.4 Test Integration

In this phase, Vega crafts the constructed instruction sequences into test cases that can be run from applications. There are two methods of integration, allowing flexibility in how SDCs can be monitored. First, Vega can create a software aging library for detecting aging-related SDCs (❼). Second, Vega provides a profile-guided method to automatically integrate the test cases into an application (❽).

4.3.4.1 Generation of Software Aging Library

In this approach, Vega combines the generated test cases together in a C file, using a set of pre-defined templates. In this C file, each test case is specified with the standard inline assembly format, while the registers are designated as variables for clarity. Furthermore, Vega generates support files for the compilation, as well as a set of helper functions and language-specific wrappers. These helper functions are designed to support different scheduling methods for the test cases, allowing them to be executed either sequentially or in a random order. Additionally, for programming languages that support exceptions, this library can be configured to trigger an exception when failing a test case. This allows detected hardware faults to be handled by an exception handler (e.g., a catch block) within the call stack.

4.3.4.2 Profile-Guided Test Integration

To enable test integration without the need to modify the application's source code, Vega employs a profile-guided approach for embedding test cases. Specifically, Vega first instruments the application with a series of counters, which track and record the invocations of application code (i.e., at the granularity of basic blocks) throughout the application runtime. Vega then executes the application with representative inputs to collect a *profile* that reflects the characteristics of the application's execution. Using this profile, Vega identifies a location in the program that, while not frequently invoked, is still routinely accessed. This location is chosen to be the point of test integration.

Subsequently, Vega integrates its generated SDC test cases into the chosen location. Vega estimates their expected performance overhead by considering both the profile data and the size of the test cases. During the estimation, it employs the number of executed Intermediate Representation (IR) instructions as a proxy for performance impact. If the estimated performance overhead exceeds a threshold, Vega restricts the invocation of test cases so that they trigger with a certain probability, which reduces the overhead and enables controlling SDC testing frequency at a finer granularity. Thus, Vega ensures that the overall performance overhead remains within manageable limits.

4.4 Implementation

We prototype and demonstrate Vega for the arithmetic logic unit (ALU) and floating-point unit (FPU) of the CV32E40P [136, 114, 209]—an open-source, 32-bit, in-order RISC-V CPU—and a real-world 28nm process technology. However, Vega's design can be applied to other instruction sets, microarchitectures, and process technologies.

During Aging Analysis, to construct the aging timing library, Vega uses SPICE [218] to conduct the analog simulation which determines the gate delay degradation of cells with varying SP profiles. Cadence Innovus [30] is then used to perform the timing analysis. Subsequently, we employ a set of TCL scripts to post-process the timing report and update the cells' timing characteristics to those affected by aging. Error Lifting is primarily implemented atop of Yosys [279], adding $\sim$3,700 lines of C++ for Failure Model Instrumentation and $\sim$400 lines of Python for Instruction Construction. JasperGold [31] is used to conduct the formal verification. Moreover, Profile-Guided Test Integration is implemented as a set of LLVM [197] passes with $\sim$800 lines of C++.

4.5 Evaluation

We evaluate Vega along the following dimensions:

Effectiveness Can Vega identify signal propagation paths prone to transistor aging (§4.5.2.1)? Can Vega generate test cases that are executable in a software environment (§4.5.2.2)? Can these test cases detect aging-related SDCs (§4.5.2.3)?

Efficiency How much performance overhead do these test cases incur when integrated into an application (§4.5.3)?

4.5.1 Experimental Setup

Hardware We evaluate Vega on the ALU and FPU of the CV32E40P. These components are synthesized for a 28 nm process technology using Cadence Genus [32] and Synopsys Design Compiler [33], and placed-and-routed using Cadence Innovus [30]. The ALU targets an operating frequency of 167 MHz and the FPU targets a frequency of 250 MHz.

Software We evaluate Vega's performance impact with EEMBC benchmarks [227, 34], a benchmark set for embedded CPUs like the CV32E40P. These benchmarks are compiled using OpenHW-Group's clang fork [35] with an `-O2` flag. This set of benchmarks are also used as representative workloads during Signal Probability Simulation (§4.3.2.1).

Failing Netlists To evaluate Vega's effectiveness in identifying aging-related SDCs, we use the failing netlists generated during failure model instrumentation (Section 4.3.3.2). We configure these failing netlists to operate in three distinct modes: by setting C (i.e., the value sampled by the ending point of the failing path as we discussed in §4.3.3.1) to 0, setting it to 1, or allowing it to take a random value in each cycle.

Simulation Environment All experiments are carried out using the official simulation framework of the CV32E40P with Verilator [252]. To speed up simulation and focus on the primary evaluation targets, i.e., the ALU and FPU, only these components are replaced with the placed-and-routed netlist. The remainder of the CPU is simulated in System Verilog.

4.5.2 Effectiveness of Vega

4.5.2.1 Potential Aging Identification

Despite the ALU and FPU being correctly placed-and-routed and meeting the required timing constraints initially, Vega reveals that over an extended period of usage, transistor aging has the potential to break these constraints. Table 4.3 summarizes the worst negative slack (WNS) and the number of identified timing violations within the ALU and FPU after 10 years of aging. In summary, Vega identifies 11 aging-prone paths in the ALU, and 1,366 such paths in the FPU.

However, many of these aging-prone paths share the same pairs of starting and ending points, indicating that these paths would exhibit the same misbehavior under the failure model we employ (§4.3.3.1). From the identified timing-violated paths, Vega recognizes 6 unique pairs of starting and ending points for the ALU and 41 pairs for the FPU. Therefore, for the rest of our analysis and test case generation, we only use one representative failing path for each unique pair of starting and ending points.

Unit	WNS / # of Violated Paths	
	Setup	Hold
ALU	-76ps / 11	— / 0
FPU	-157ps / 1,363	-1ps / 3

Table 4.3: STA Result with Aging-Aware Timing Libraries.

4.5.2.2 Test Case Construction

For each unique pair of starting and ending points, Vega invokes formal verification to produce a set of waveforms that activate this failing path in an observable manner, and then converts these waveforms into a few test cases. Depending on configuration, different numbers of test cases may be generated. When the mitigation for initial value dependency (Section 4.3.3.4) is disabled, Vega produces a maximum of 2 test cases for each pair, attributable to the failure model's constant, C, which can be either 0 or 1. With the mitigation enabled, Vega generates a maximum of 4 test cases per pair, in order to account for different signal transitions (i.e., rising or falling) in the starting point.

Table 4.4 presents the effectiveness of this process. Without the mitigation, Vega can construct the test cases for 66.7% and 51.2% of these unique pairs of endpoints identified in the ALU and FPU respectively. Additionally, it formally proves that 33.3% of the pairs in the ALU and 43.9% of the pairs in the FPU will not cause an actual error—no allowable set of inputs to the module can trigger the timing violation for these paths. Enabling the mitigation reduces the proportion of test cases that can be successfully generated. However, because it generates up to twice as many test cases, it can produce a more robust test suite.

In some instances, we observed that Vega may produce a waveform that is not convertible into a practical test case. These instances are indicated as "FC" in Table 4.4. All such instances occur with the FPU. This situation happens because certain failures—characterized by a pair of endpoints along with a particular failure model—require multiple instructions to propagate an error to the output; moreover, the only detectable erroneous output is a status flag (e.g., a flag indicating an overflow), which is already altered by a prior instruction. As a result, Vega cannot compare this output against a correct value, making the conversion impossible.

Table 4.5 shows the total number of test cases generated by Vega and the corresponding CPU cycles required for their execution, both in scenarios with and without the mitigation for initial value dependency. Notably, a complete execution of these test cases consumes only a few hundred to a couple thousand cycles, thereby making frequent testing practical.

Unit	w/o Mitigation (%)				w/ Mitigation (%)			
	S	UR	FF	FC	S	UR	FF	FC
ALU	66.7	33.3	0	0	33.3	66.7	0	0
FPU	51.2	43.9	4.9	0	40.2	43.9	8.5	7.3

Table 4.4: Result of Test Case Construction. "S" denotes the successful construction of a test case; "UR" indicates that the formal verification tool proves that the failing path cannot cause an actual error; "FF" indicates a timeout occurred in the formal verification tool; "FC" indicates a waveform is generated while Vega fails to convert it to a test case.

Unit	w/o Mitigation		w/ Mitigation	
	Test Cases	Cycles	Test Cases	Cycles
ALU	8	124	8	134
FPU	60	685	96	1202

Table 4.5: The quantity of test cases generated and the number of CPU cycles required for their execution.

4.5.2.3 Quality of Test Cases

We evaluate the quality of these test cases by simulating them against the failing netlists produced by failure model instrumentation (Section 4.3.3.2). For each failing netlist associated with one of the generated test cases, we run the entire set of test cases to see whether it can detect the failure. As mentioned in the evaluation setup, we configure each failing netlist to fail in three modes: with C setting to 0, 1, or taking a random value at each cycle.

Table 4.6 shows the result of this experiment. In summary, the test cases generated by Vega are are generally effective in detecting their intended failures. Interestingly, in many cases, a failure is identified by a test case designed for another failure, before its own corresponding test case is scheduled to execute. In rare cases, a failure may be missed by its own test case; however, it is very likely to be identified by a subsequent test case. In two instances, the failure caused the CPU to become stuck, making it detectable. In one particular instance, a failure remained undetected after the execution of the entire set of test cases. This occurs because the test cases generated for this failure depends on certain initial signal values to be effective. Unfortunately, these values are modified by a prior instruction, thereby preventing the failure from being detected. However, using the mitigation technique described in Section 4.3.3.4, Vega can generate a set of test cases that successfully detects this failure.

Unit	FM	w/o Mitigation (%)				w/ Mitigation (%)			
		Det.	**B**	**L**	**S**	**Det.**	**B**	**L**	**S**
ALU	**0**	100.0	50.0	0	0	100.0	50.0	0	0
	1	100.0	75.0	0	0	100.0	50.0	0	0
	R	100.0	50.0	0	0	100.0	50.0	0	0
FPU	**0**	95.4	72.7	4.5	9.1	100.0	72.7	0	9.1
	1	95.4	81.8	9.1	0	100.0	81.8	4.5	0
	R	95.4	72.7	4.5	9.1	100.0	72.7	0	9.1

Table 4.6: The quality of the generated test cases measured by their ability to detect failures. "FM" refers to the failure mode used in the experiment; "Det." indicates the failures that are detectable by one of the test cases; "B" represents the failures detected by a test case that executed before the test case designed to detect it; "L" represents the failures that are not detected by their corresponding test case, but are identified by later test case; "S" indicates cases where the failure results in the CPU becoming stuck.

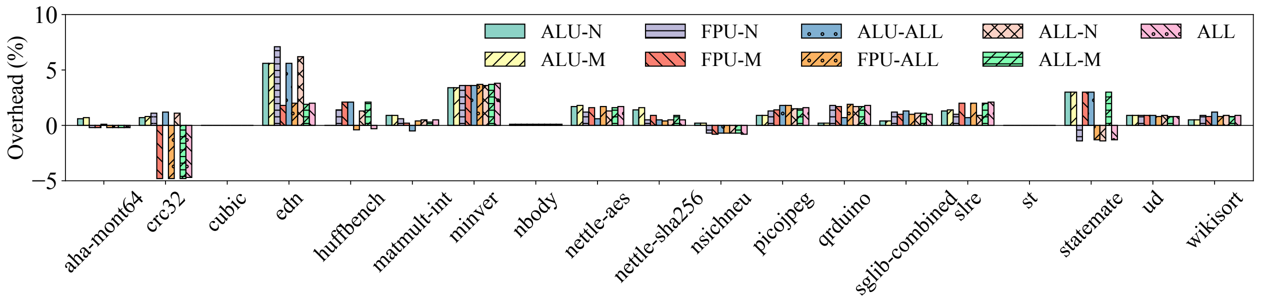


Figure 4.8: Performance overhead of the EEMBC benchmark set with Vega's Profile-Guided Test Integration. The "-M" and "-N" labels indicate that only the test cases generated with and without the mitigation technique are enabled, respectively.

4.5.3 Efficiency of Vega

We evaluate the performance overhead of Vega's Profile-Guided Test Integration by comparing the execution time of the benchmarks instrumented with the test cases against their baseline. We adopt a variety of configurations, with different configuration enabling different sets of test cases.

Figure 4.8 presents the overhead. On average, Vega's Profile-Guided Test Integration introduces 0.8% overhead to the application's execution time. In many instances, this overhead is so negligible that it becomes indistinguishable with environmental noise (e.g., the compiler optimizations that are accidentally triggered as a side effect of the instrumentation), resulting in a negative overhead. Therefore, we conclude that Vega's Profile-Guided Test Integration can effectively manage and minimize its performance overhead.

4.6 Related Work

Analyses of SDCs Previous work studied radiation-induced SDCs, including in storage systems [85], memory devices [206], FPGAs [233], HPC systems [213, 127, 123], and satellite processors [304]. Recently, hyperscalers have reported SDCs caused by malfunctioning CPUs in both case studies and comprehensive analyses at the scale of data centers [145, 121, 271, 122, 244, 84, 120]. Vega is inspired by these studies, and focuses on the detection of aging-related SDCs inside CPUs.

Detection of SDCs Data center operators identify SDCs in their CPU populations by conducting tests that are scheduled infrequently or with a low priority [271, 244]. In certain systems, SDCs are detected through checksums or by verifying whether the computation result is reasonable [84, 142]. Additionally, SDCs can be detected by introducing redundancies in either software or hardware designs [115, 75, 93, 268]. Vega focuses on generating compact test cases for frequent, at-scale SDC testing in data centers.

Test Case Construction for SDCs OpenDCDiag [28] detects SDCs by using popular software libraries. Google's SiliFuzz [244] synthesizes test cases by fuzzing a functional model of the CPU. Unlike these works, Vega adopts a bottom-up approach that constructs test cases by analyzing the CPU's low-level implementation.

Analyses of Transistor Aging Previous work has analyzed the effect of transistor aging on various hardware designs, such as CPUs [133], GPUs [135], and FPGAs [68], as well as a variety of hardware building blocks, such as standard cell libraries [179, 70] and SRAMs [119]. In this paper, we focuses on transistor aging within CPUs; however, Vega's design and insights can be applied to other hardware designs.

Hardware Mitigations for Transistor Aging Prior work has explored transistor aging mitigation techniques that operate across multiple phases of the hardware design process. Gabbay et al. proposed an aging-aware microarchitecture aimed at mitigating the effects of nonuniform aging on various components of microprocessors [133]. Calimera et al. introduced an aging-resistant SRAM cache design [98]. Other studies have focused on designing EDA tools for simulating and analyzing the impact of transistor aging [221, 105], and on refining the physical design of a circuit to enhance aging resistance [266, 265, 188]. Unlike these works, Vega analyzes a hardware design for the sake of building better application-level detection techniques for transistor aging.

Software Mitigations for Transistor Aging Firouzi et al. proposed the insertion of NOP instructions in MIPS processors to reduce the impact of transistor aging [129]. Abbas et al.

proposed running anti-aging programs during processors' idle periods for aging mitigation [58]. Vega is orthogonal to these works, and provides a systematic way to provoke specific aging-related failures within a CPU design.

4.7 Conclusions

In this work, we presented Vega, a novel workflow that bridges the gap between the physical understanding of transistor aging and the software detection of aging-related SDCs. Vega adopts a bottom-up approach and targets the most aging-prone components within a CPU, thus yielding a compact set of test cases that only take hundreds to thousands of cycles to execute. Therefore, Vega enables frequent detection of aging-related SDCs at application runtime. Our experiments show that Vega can effectively construct test cases for complex CPU elements, and that these test cases can effectively identify aging-related failures.

CHAPTER 5

Conclusion and Future Work

In this dissertation, we propose, introduce, and examine software-like systems and debugging supports tailored for different hardware designs. Specifically, this dissertation conducts preliminary explorations into this field, and demonstrates the feasibility and benefits of such supports via three projects, each focusing on a different aspect of hardware development and deployment.

The first project focuses on improving the deployment of reconfigurable hardware by introducing software-like systems (Chapter 2). In this project, we build OPTIMUS, the first hypervisor for shared-memory FPGA platforms. OPTIMUS implements both spatial multiplexing and temporal multiplexing, thus allowing a shared-memory FPGA to be shared among different virtual machines either by partitioning the FPGA's area or by allocating time slots for its use. OPTIMUS demonstrates that software-like systems can not only be applied to reconfigurable hardware, but also yield comparable benefits to those observed in the software counterparts.

The second project focuses on studying bugs that occur on reconfigurable hardware (Chapter 3). In this project, we first conduct a comprehensive study on FPGA bugs, in which we find that hardware bugs and software bugs share a number of similarities. Furthermore, based on the findings in the study, we designed a suite of debugging tools for bugs that occur on an FPGA. With the study and tools, we demonstrate that software-like debugging tools can be beneficial to hardware bugs in FPGA-based designs.

The third project focuses on the detection of aging-related silent data corruptions, an emerging hardware reliability issue that is increasingly observed in data centers (Chapter 4). In this project, we present Vega, a workflow that enables the detection of aging-related SDCs during application runtime. Vega adopts a bottom-up approach that generates concise test cases targeting the most aging-prone components within a CPU, thus yielding a compact enough test suite that can be frequently invoked at application runtime. With Vega, we show that certain hardware reliability issues can be detected using software-like techniques.

Collectively, these projects present a viable pathway toward simplifying the development and deployment processes of hardware designs. By building software-like systems and tooling supports for hardware designs, the development and deployment of hardware can be significantly enhanced and more efficiently managed.

In the remainder of this chapter, we outline potential future research directions that are opened up by the findings of this dissertation.

5.1 Extending Systems Supports for FPGAs

While OPTIMUS conducted preliminary investigations into systems support for FPGAs, there is significant potential to expand these supports even further.

Resource Sharing among Accelerators Although OPTIMUS enables the deployment of multiple shared-memory accelerators on a single FPGA, it is designed so that the on-FPGA resources (e.g., BRAM and ALM) occupied by each accelerator do not overlap. This design choice is primarily driven by security considerations. However, in the software domain, it's common for different applications to share components like language runtimes, libraries, and even hardware elements such as caches and registers, while still maintaining adequate isolation. This presents an intriguing contrast and potential area for exploration.

Context-Switch for Accelerators OPTIMUS supports temporal multiplexing that allows different virtual machines to take turns using the same accelerator. However, this functionality necessitates the implementation of a preemption interface within the accelerator, which requires additional effort from developers. While compiler-driven techniques such as Synergy [196] allow automatically preempting the context of an accelerator, they often incur significant performance and area overheads. How to co-design the FPGA and its systems support for more efficient preemption remains an open research question. Addressing this challenge would involve finding a balance between the programming flexibility, the granularity of temporal multiplexing, and the efficiency of hardware, which can lead to more robust, elastic, and scalable FPGA-based systems.

Data Center Aware Accelerator Scheduling OPTIMUS allows multiple tenants in the public cloud to share the same FPGA. However, effectively co-locating different tenants while taking into account their specific FPGA usage is still an open area of research.

5.2 Exploring More Debugging Tools for Hardware Designs

In Chapter 3, we introduce a suite of debugging tools, which includes SignalCat, a collection of monitors, and LossCheck. These tools instrument the hardware design with logic that actively monitors specific events. They function by either generating human-readable logs for analysis or pinpointing the location of a failure as soon as it is detected. Following this methodology, more debugging tools can be designed to target different kinds of bugs.

Invasive Instrumentation All debugging tools proposed in Chapter 3 instruments hardware designs in a non-invasive manner—i.e., they operate without altering the per-cycle behavior of the hardware under analysis. However, it is worth considering that allowing modifications to the cycle-level behavior could potentially lead to more powerful and general-purpose debugging tools, as it may offer deeper insights into the hardware's operation. How to safely conduct such invasive instrumentation while ensuring the correct functionality of the hardware remains a research challenge for future exploration.

Co-design FPGA and Debugging Tools Many software debugging techniques, such as GDB [255] and REPT [112], rely on specific hardware supports for seamless operation. The current lack of such built-in support in FPGA architectures highlights the need for a co-design between the FPGA and external debugging tools.

Software-Hardware Co-debugging In certain scenarios, diagnosing the source of a bug can be challenging when it is unclear whether the issue originates from the hardware itself or from the software running on top of it. Future work may explore debugging techniques that allow software and hardware to be instrumented and debugged together.

5.3 Enhancing the Detection of Unreliable Hardware

In Chapter 4, we designed Vega, which employs a bottom-up workflow and allows aging-related silent data corruptions to be detected at application runtime with low overhead. Vega opens future research in two directions. The first direction focuses on expanding the capability of detecting a broader range of reliability issues, while the second direction concentrates on enhancing the detection mechanism itself.

Electromigration Like transistor aging, electromigration is another important physical effect that affects circuit reliability. Like transistor aging, previous research has explored the mechanism and modeling for electromigration [144, 91, 110, 116]. Such understanding can be integrated into Vega's workflow to enable a more comprehensive circuit reliability testing at application runtime.

Testing-Aware Microarchitecture Vega leverages a compiler pass to instrument the test cases into applications, employing a profile-guided approach for the selection of instrumentation points. While this method enables the detection of aging-related SDCs with low overhead, it has the limitation of only conducting tests when the specific code sections are invoked. On the other hand, there are typically idle units present in each clock cycle within a CPU. These units, not engaged in active computation, could potentially be leveraged for continuous or periodic testing that does not depend on the execution of a user application. By designing a microarchitecture that harnesses these idle units for reliability-related testing, we can further improve the testing frequency, therefore enhancing the overall reliability of the system.

BIBLIOGRAPHY

[1] https://lldb.llvm.org/.

[2] https://logging.apache.org/log4j/2.x/.

[3] https://logback.qos.ch/.

[4] https://www.kernel.org/.

[5] https://en.wikipedia.org/wiki/Microsoft_Windows.

[6] https://en.wikipedia.org/wiki/MacOS.

[7] https://gcc.gnu.org/wiki/Libstdc++.

[8] https://www.python.org/.

[9] https://en.wikipedia.org/wiki/Java_(programming_language).

[10] https://sourceware.org/glibc/.

[11] https://www.tomshardware.com/news/alibaba-unveils-128-core-server-cpu.

[12] https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/multichannel-dma-mcdma.html.

[13] https://github.com/efeslab/hardware-bugbase.

[14] https://zipcpu.com.

[15] https://github.com/ZipCPU/sdspi.

[16] https://zipcpu.com/formal/2018/12/28/axilite.html.

[17] https://zipcpu.com/dsp/2020/04/20/axil2axis.html.

[18] https://github.com/open-sdr/openwifi-hw.

[19] https://github.com/jbush001/NyuziProcessor.

[20] https://github.com/openhwgroup/cva6.

[21] https://github.com/SpinalHDL/VexRiscv.

[22] https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.

[23] https://github.com/corundum/corundum.

[24] https://github.com/alexforencich/verilog-ethernet.

[25] https://github.com/analogdevicesinc/hdl.

[26] https://github.com/alexforencich/verilog-axis.

[27] https://github.com/mjc0608/really-simple-fadd.

[28] https://github.com/opendcdiag/opendcdiag.

[29] https://www.zlib.net/.

[30] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html.

[31] https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[32] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[33] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[34] https://github.com/embench/embench-iot.

[35] https://github.com/openhwgroup/corev-llvm-project.

[36] Aging problems at 5nm and below. https://semiengineering.com/aging-problems-at-5nm-and-below/.

[37] Axi hardware icap. https://www.xilinx.com/products/intellectual-property/axi_hwicap.html.

[38] Gp100 pascal whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[39] Hugetlbfs reservation. https://www.kernel.org/doc/html/v4.18/vm/hugetlbfs_reserv.html.

[40] Intel quartus prime software suite. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[41] Intel® data center diagnostic tool for intel® xeon® processors. https://www.intel.com/content/www/us/en/support/articles/000058107/processors/intel-xeon-processors.html.

[42] Open-source fpga bitcoin miner. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.

[43] Questa verification & simulation. https://eda.sw.siemens.com/en-US/ic/questa/simulation.

[44] Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/.

[45] Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[46] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[47] Xcelium logic simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[48] Xilinx kintex-7 fpga kc705 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[49] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[50] https://www.exostivlabs.com/fpga-debug-flow-should-be-improved/, 2015.

[51] https://github.com/omphardcloud/hardcloud/tree/master/samples/sha512, 2018.

[52] https://github.com/omphardcloud/hardcloud/tree/master/samples/reed_solomon_decoder, 2018.

[53] https://github.com/omphardcloud/hardcloud/tree/master/samples/grayscale, 2018.

[54] https://zipcpu.com/dsp/2018/10/02/fft.html, 2018.

[55] Axi protocol checker v2.0. https://www.xilinx.com/support/documentation/ip_documentation/axi_protocol_checker/v2_0/pg101-axi-protocol-checker.pdf, 2018.

[56] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[57] Ieee standard for universal verification methodology language reference manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pages 1–458, 2020.

[58] Haider Muhi Abbas, Mark Zwolinski, and Basel Halak. Aging mitigation techniques for microprocessors using anti-aging software. *Ageing of Integrated Circuits: Causes, Effects and Mitigation Techniques*, pages 67–89, 2020.

[59] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[60] Mridul Agarwal, Bipul C Paul, Ming Zhang, and Subhasish Mitra. Circuit failure prediction and its application to transistor aging. In *25th IEEE VLSI Test Symposium (VTS'07)*, pages 277–286. IEEE, 2007.

[61] M. A. Alam and C. Augustine K. Roy. Reliability- and process-variation aware design of integrated circuits—a broader perspective. In *2011 International Reliability Physics Symposium, Monterey, CA, USA*, pages 4A.1.1–4A.1.11, 2011. doi: 10.1109/IRPS.2011.5784500.

[62] M. A. Alam and S. Mahapatra. A comprehensive model of pmos nbti degradation. *Microelectronics Reliability*, 45:71–81, 2005. https://doi.org/10.1016/j.microrel.2004.03.019.

[63] M.A. Alam, H. Kufluoglu, D. Varghese, and S. Mahapatra. A comprehensive model for pmos nbti degradation: Recent progress. *Microelectronics Reliability*, 47(6):853–862, 2007. Modelling the Negative Bias Temperature Instability.

[64] Alibaba. Deep dive into alibaba cloud f3 fpga as a service instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057.

[65] Altera. Implementing state machines (verilog hdl). https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vlog/vlog_pro_state_machines.htm, 2013.

[66] Amazon. Amazon ec2 f1 instances - run customizable fpgas in the aws cloud. https://aws.amazon.com/ec2/instance-types/f1.

[67] Amazon. Official repository of the aws ec2 fpga hardware and software development kit. https://github.com/aws/aws-fpga.

[68] Abdulazim Amouri, Florent Bruguier, Saman Kiamehr, Pascal Benoit, Lionel Torres, and Mehdi Tahoori. Aging effects in fpgas: An experimental analysis. In *2014 24th international conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[69] Hussam Amrouch. *Techniques for aging, soft errors and temperature to increase the reliability of embedded on-chip systems*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2015, 2015.

[70] Hussam Amrouch, Behnam Khaleghi, Andreas Gerstlauer, and Jörg Henkel. Reliability-aware design to suppress aging. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[71] Hussam Amrouch, Javier Martin-Martinez, Victor M van Santen, Miquel Moras, Rosana Rodriguez, Montserrat Nafria, and Jörg Henkel. Connecting the physical and application level towards grasping aging effects. In *2015 IEEE International Reliability Physics Symposium*, pages 3D–1. IEEE, 2015.

[72] Hari Angepat, Gage Eads, Christopher Craik, and Derek Chiou. Nifd: Non-intrusive fpga debugger–debugging fpga'threads' for rapid hw/sw systems prototyping. In *2010 International Conference on Field Programmable Logic and Applications*, pages 356–359. IEEE, 2010.

[73] Md Toufiq Hasan Anik, Sylvain Guilley, Jean-Luc Danger, and Naghmeh Karimi. On the effect of aging on digital sensors. In *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, pages 189–194. IEEE, 2020.

[74] ARM. Amba axi and ace protocol specification, 2021.

[75] Sanem Arslan and Osman Unsal. Efficient selective replication of critical code regions for sdc mitigation leveraging redundant multithreading. *The Journal of Supercomputing*, 77(12):14130–14160, 2021.

[76] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. Virtualized execution runtime for fpga accelerators in the cloud. *Ieee Access*, 5:1900–1910, 2017.

[77] Systems Group at ETH Zurich. Enzian is a research computer built by the systems group at eth zurich. http://www.enzian.systems/.

[78] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 228–235. IEEE, 2014.

[79] Sameh Attia and Vaughn Betz. Feel free to interrupt: Safe task stopping to enable fpga checkpointing and context switching. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(1):1–27, 2020.

[80] Sameh Attia and Vaughn Betz. StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, pages 175–185, New York, NY, USA, February 2020. Association for Computing Machinery.

[81] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: Enabling application-transparent support for multiple page sizes in throughput processors. *SIGOPS Oper. Syst. Rev.*, 52(1):27–44, August 2018.

[82] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[83] Altug Hakan Baba and Subhasish Mitra. Testing for transistor aging. In *2009 27th IEEE VLSI Test Symposium*, pages 215–220. IEEE, 2009.

[84] David F Bacon. Detection and prevention of silent data corruption in an exabyte-scale database system. 2022.

[85] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):1–28, 2008.

[86] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[87] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[88] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D Ernst. Synoptic: Studying logged behavior with inferred models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 448–451, 2011.

[89] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula. Predictive modeling of the nbti effect for reliable design. In *Proceedings of the Custom Integrated Circuits Conference*, pages 189–192, 2006. https://doi.org/10.1109/CICC.2006.320885.

[90] Jayaram Bhasker. *A Vhdl Primer*. Prentice-Hall, 1999.

[91] James R Black. Electromigration—a brief survey and some recent results. *IEEE Transactions on Electron Devices*, 16(4):338–347, 1969.

[92] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*, pages 93–96. IEEE, 2012.

[93] Yuriy Brun, George Edwards, Jae Young Bang, and Nenad Medvidovic. Smart redundancy for distributed computation. In *2011 31st International Conference on Distributed Computing Systems*, pages 665–676. IEEE, 2011.

[94] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on {FPGA} nics. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 973–990, 2020.

[95] Doug Burger. Microsoft unveils project brainwave for real-time ai. *Microsoft Research, Microsoft*, 22, 2017.

[96] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116. IEEE, 2014.

[97] A. Calimera, M. Loghi, E. MacIi, and M. Poncino. Aging effects of leakage optimizations for caches. In *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, pages 95–98, 2010. https://doi.org/10.1145/1785481.1785504.

[98] Andrea Calimera, Mirko Loghi, Enrico Macii, and Massimo Poncino. Dynamic indexing: Leakage-aging co-optimization for caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(2):251–264, 2014.

[99] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[100] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, 2013.

[101] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Press, 2016.

[102] Ciro Ceissler, Ramon Nepomuceno, Marcio Pereira, and Guido Araujo. Automatic offloading of cluster accelerators. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 224–224. IEEE, 2018.

[103] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, et al. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64. IEEE, 2020.

[104] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 3. ACM, 2014.

[105] Kueing-Long Chen, Stephen A Saller, Imelda A Groves, and David B Scott. Reliability effects on mos transistors due to hot-carrier injection. *IEEE Transactions on Electron Devices*, 32(2):386–393, 1985.

[106] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. Thundergp: Hls-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–80, 2021.

[107] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[108] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. Flash: Fast, parallel, and accurate simulator for hls. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4828–4841, 2020.

[109] Eric S Chung, James C Hoe, and Ken Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 97–106. ACM, 2011.

[110] J Joseph Clement. Electromigration modeling for integrated circuit interconnect reliability analysis. *IEEE Transactions on Device and Materials Reliability*, 1(1):33–42, 2001.

[111] Automotive Electronics Council. *LaTeX: Failure mechanism based stress test qualification for integrated circuit*. AAEC – Q100 – REV-G standard.

[112] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 17–32, 2018.

[113] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[114] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[115] Ádria Barros de Oliveira, Lucas Antunes Tambara, and Fernanda Lima Kastensmidt. Applying lockstep in dual-core arm cortex-a9 to mitigate radiation-induced soft errors. In *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–4, 2017.

[116] RL De Orio, Hajdin Ceric, and Siegfried Selberherr. Physically based models of electromigration: From black's equation to modern tcad models. *Microelectronics Reliability*, 50(6):775–789, 2010.

[117] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 213–230, 2019.

[118] Edsger W. Dijkstra and DIJKSTRA EW. Information streams sharing a finite buffer. 1972.

[119] Jie Ding, Dave Reid, Plamen Asenov, Campbell Millar, and Asen Asenov. Influence of transistors with bti-induced aging on sram write performance. *IEEE Transactions on Electron Devices*, 62(10):3133–3138, 2015.

[120] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. Detecting silent data corruptions in the wild. *arXiv preprint arXiv:2203.08989*, 2022.

[121] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[122] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[123] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault tolerance techniques for high-performance computing*. Springer, 2015.

[124] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9-10):833–859, 2008.

[125] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[126] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 430–435. IEEE, 2015.

[127] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[128] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[129] Farshad Firouzi, Saman Kiamehr, and Mehdi B Tahoori. Nbti mitigation by optimized nop assignment and insertion. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 218–223. IEEE, 2012.

[130] K. Fleming, H. Yang, M. Adler, and J. Emer. The leap fpga operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2014.

[131] Kermin Fleming and Michael Adler. The leap fpga operating system. In *FPGAs for Software Programmers*, pages 245–258. Springer, 2016.

[132] Harry Foster. 2020 wilson research group functional verification study: Fpga functional verification trend report, 2020.

[133] Freddy Gabbay and Avi Mendelson. Asymmetric aging effect on modern microprocessors. *Microelectronics Reliability*, 119:114090, 2021.

[134] Freddy Gabbay, Firas Ramadan, and Majd Ganaiem. Clock tree design considerations in the prescence of asymmetric transistor aging. In *20203 10th Design and Verification Conference - Europe (DVCON)*, 2023.

[135] Freddy Gabbay, Firas Ramadan, Majd Ganaiem, Ofrie Rosenthal, and Lior Bashari. Effect of Asymmetric Transistor Aging on GPGPUs. In *Proceedings of the 5th International Conference on Microelectronic Devices and Technologies (MicDAT '2023)*, pages 52–56, 2023.

[136] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[137] Jeffrey Goeders and Steve JE Wilton. Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas. In *2015 IEEE 23rd annual international symposium on field-programmable custom computing machines*, pages 127–134. IEEE, 2015.

[138] Jeffrey Goeders and Steven JE Wilton. Effective fpga debug for high-level synthesis generated circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[139] Gokul Govindu, Ronald Scrofano, and Viktor K Prasanna. A library of parameterizable floating-point cores for fpgas and their application to scientific computing. In *Proc Int'l Conf. Eng. Reconfigurable Systems and Algorithms (ERSA'05)*. Citeseer, 2005.

[140] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[141] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[142] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[143] Jeffrey Hicks, Daniel Bergstrom, Mike Hattendorf, Jason Jopling, Jose Maiz, Sangwoo Pae, Chetan Prasad, and Jami Wiedemer. 45nm transistor reliability. *Intel Technology Journal*, 12(2), 2008.

[144] Paul S Ho and Thomas Kwok. Electromigration in metals. *Reports on Progress in Physics*, 52(3):301, 1989.

[145] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021.

[146] Daniel Holanda Noronha, Ruizhe Zhao, Jeff Goeders, Wayne Luk, and Steven JE Wilton. On-chip fpga debug instrumentation for machine learning applications. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 110–115, 2019.

[147] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ila) a uniform specification for system-on-chip (soc) verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(1):1–24, 2018.

[148] Eddie Hung and Steven JE Wilton. Scalable signal selection for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6):1103–1115, 2012.

[149] Intel. Acceleration stack for intel xeon cpu with fpgas core cache interface (cci-p) reference manual. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl-ias-ccip.pdf.

[150] Intel. Hardware Accelerator Research Program. https://software.intel.com/en-us/hardware-accelerator-research-program.

[151] Intel. Intel arria 10 avalon-st interface with sr-iov pcie solutions user guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_a10_pcie_sriov.pdf.

[152] Intel. Intel high level synthesis compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[153] Intel. Intel programmable acceleration card with intel arria 10 gx fpga. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html.

[154] Intel. Intel virtualization technology for directed i/o. https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf.

[155] Intel. Open programmable acceleration engine. https://opae.github.io/latest/index.html.

[156] Intel. Intel open source hd graphics and intel iris plus graphics programmer's reference manual for the 2016 - 2017 intel core processors, celeron processors, and pentium processors based on the "kaby lake" platform. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol05-memory_views.pdf, 2017.

[157] Intel. Embedded peripherals ip user guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf, 2019.

[158] Intel. Intel arria 10 fpgas. https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html, 2019.

[159] Intel. Intel FPGA Basic Building Blocks (BBB). https://github.com/OPAE/intel-fpga-bbb, 2019.

[160] Intel. Intel quartus prime pro edition user guide: Debug tools. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-debug.pdf, 2020.

[161] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. Efficient overlay architecture based on dsp blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2015.

[162] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Throughput oriented fpga overlays using dsp blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1628–1633. IEEE, 2016.

[163] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In *Proceedings of the 42nd annual Design Automation Conference*, pages 445–450, 2005.

[164] Neo Jia and Kirti Wankhede. Vfio mediated devices. https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt.

[165] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023.

[166] 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.

[167] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[168] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. Fireperf: Fpga-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 715–731, 2020.

[169] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *SOSP*, Shanghai, China, October 2017.

[170] Baris Kasikci, Cristiano Pereira, Gilles Pokam, Benjamin Schubert, Malandal Musuvathi, and George Candea. Failure sketches: A better way to debug. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[171] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *SOSP*, Monterey, CA, October 2015.

[172] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced data race detection. In *SOSP*, Farmington, PA, November 2013.

[173] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[174] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[175] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture*, 2021.

[176] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127. USENIX Association, 2018.

[177] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: Nre optimization in asic clouds. *SIGPLAN Not.*, 52(4):511–526, April 2017.

[178] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. S2n2: A fpga accelerator for streaming spiking neural networks. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 194–205, 2021.

[179] Saman Kiamehr, Farshad Firouzi, Mojtaba Ebrahimi, and Mehdi B Tahoori. Aging-aware standard cell library design. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014.

[180] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. Dessert: Debugging rtl effectively with state snapshotting for error replays across trillions of cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 76–764. IEEE, 2018.

[181] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[182] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, 2007.

[183] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. *Reconfigurable Architectures, Tools and Applications, RECATA*, 2017.

[184] Oliver Knodel and Rainer G Spallek. Rc3e: provision and management of reconfigurable hardware accelerators in a cloud environment. *arXiv preprint arXiv:1508.06843*, 2015.

[185] Ho Fai Ko and Nicola Nicolici. Automated trace signals selection using the rtl descriptions. In *2010 IEEE International Test Conference*, pages 1–10. IEEE, 2010.

[186] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. An efficient fpga overlay for portable custom instruction set extensions. In *2013 23rd international conference on field programmable logic and applications*, pages 1–8. IEEE, 2013.

[187] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do {OS} abstractions make sense on fpgas? In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 991–1010, 2020.

[188] Anand T Krishnan, Frank Cano, Cathy Chancellor, Vijay Reddy, Zhangfen Qi, Palkesh Jain, John Carulli, Jonathan Masin, Steve Zuhoski, Srikanth Krishnan, et al. Product drift from nbti: Guardbanding, circuit and statistical effects. In *2010 International Electron Devices Meeting*, pages 4–3. IEEE, 2010.

[189] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[190] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. An analytical model for negative bias temperature instability. In *2006 IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, USA*, pages 493–496, 2006. doi: 10.1109/ICCAD.2006.320163.

[191] Patrick Kutch. Pci-sig sr-iov primer: An introduction to sr-iov technology. *Intel application note*, pages 321211–002, 2011.

[192] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. {FVM}: Fpga-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 955–971, 2020.

[193] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 705–721, 2016.

[194] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Ingens: Huge page support for the os and hypervisor. *ACM SIGOPS Operating Systems Review*, 51(1):83–93, 2017.

[195] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[196] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. Compiler-driven fpga virtualization with synergy. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 818–831, 2021.

[197] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[198] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[199] Suho Lee and Karem A Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *International Conference on Computer Aided Verification*, pages 849–865. Springer, 2014.

[200] W. Li, G. Jin, X. Cui, and S. See. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098, May 2015.

[201] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006.

[202] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072–1086, 2018.

[203] Changze Liu, Yongsheng Sun, Pengpeng Ren, Dan Gao, Weichun Luo, Zanfeng Chen, and Yu Xia. New challenges of design for reliability in advanced technology node. In *2020 4th IEEE Electron Devices Technology & Manufacturing Conference (EDTM)*, pages 1–4. IEEE, 2020.

[204] Xiao Liu and Qiang Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 1338–1343. IEEE, 2009.

[205] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, 2009.

[206] Lucas Matana Luza, Daniel Söderström, Helmut Puchner, Rubén García Alía, Manon Letiche, Alberto Bosio, and Luigi Dilillo. Effects of thermal neutron irradiation on a self-refresh dram. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2020.

[207] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 827–844, 2020.

[208] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. Debugging in the brave new world of reconfigurable hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 946–962, 2022.

[209] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(4):774–787, 2020.

[210] Raffael Marty. Cloud application logging for forensics. In *proceedings of the 2011 ACM Symposium on Applied Computing*, pages 178–184, 2011.

[211] Marco Antonio Merlini, Isamu Poy, and Paul Chow. Interactive debugging at ip block interfaces in fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 138–144, 2021.

[212] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 121–131. ACM, 2019.

[213] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, 2005.

[214] Microsoft. What are fpgas and project brainwave? `https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-accelerate-with-fpgas`, 2019.

[215] Ali Mili, Rym Mili, and Roland T Mittermeir. A survey of software reuse libraries. *Annals of software engineering*, 5(1):349–414, 1998.

[216] Roberto Millón, Emmanuel Frati, and Enzo Rucci. A comparative study between hls and hdl on soc for image processing applications. *arXiv preprint arXiv:2012.08320*, 2020.

[217] David Mulnix. Intel xeon processor scalable family technical overview, 2017.

[218] Laurence W. Nagel and D.O. Pederson. Spice (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[219] S Novak, C Parker, D Becher, M Liu, Marty Agostinelli, M Chahal, P Packan, P Nayak, Stephen Ramey, and S Natarajan. Transistor aging and reliability in 14nm tri-gate technology. In *2015 IEEE International Reliability Physics Symposium*, pages 2F–2. IEEE, 2015.

[220] Fabian Oboril and Mehdi B Tahoori. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[221] Shigeo Ogawa and Noboru Shiono. Generalized diffusion-reaction model for the low-field charge-buildup instability at the si-sio$_2$ interface. *Phys. Rev. B*, 51:4218–4230, Feb 1995.

[222] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 211–218. IEEE, 2017.

[223] Michele Paolino, Sébastien Pinneterre, and Daniel Raho. Fpga virtualization with accelerators overcommitment for network function virtualization. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2017.

[224] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4. IEEE, 2006.

[225] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino, and Daniel Raho. vfpgamanager: A virtualization framework for orchestrated fpga accelerator sharing in 5g cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.

[226] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, and Adam Chlipala. Effective simulation and debugging for a high-level hardware language using software compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 789–803, 2021.

[227] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, 29(5):18–29, 2009.

[228] C. L. Portmann and T. H. Y. Meng. Metastability in cmos library elements in reduced supply and technology scaled applications. *IEEE J Solid-State Circuits*, 30:39–46, 1995. https://doi.org/10.1109/4.350196.

[229] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[230] Hao Qian and Yangdong Deng. Accelerating rtl simulation with gpus. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 687–693. IEEE, 2011.

[231] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, April 2018.

[232] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[233] H. Quinn and P. Graham. Terrestrial-based radiation upsets: a cautionary tale. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 193–202, 2005.

[234] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 600–615, 2021.

[235] Alastair Reid. Trustworthy specifications of arm® v8-a and v8-m system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168. IEEE, 2016.

[236] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[237] James A Rowson. Hardware/software co-simulation. In *Proceedings of the 31st Annual Design Automation Conference*, pages 439–440, 1994.

[238] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 139–152, 2013.

[239] Nikolay Sakharnykh. Everything you need to know about unified memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[240] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. Nascent: Near-storage acceleration of database sort on smartssd. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 262–272, 2021.

[241] Eric Schkufza, Michael Wei, and Christopher J Rossbach. Just-in-time compilation for verilog: A new technique for improving the fpga programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–286. ACM, 2019.

[242] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Address-sanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[243] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

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

[245] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 2, USA, 2005. USENIX Association.

[246] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[247] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[248] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[249] David Sidler and Ken Eguro. Debugging framework for fpga-based soft processors. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 165–168. IEEE, 2016.

[250] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 403–415. ACM, 2017.

[251] Any Silicon. Fpga vs asic, what to choose? https://anysilicon.com/fpga-vs-asic-choose/.

[252] Wilson Snyder. https://www.veripool.org/verilator/, 2021.

[253] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):14, 2008.

[254] Hayden Kwok-Hay So and Robert W Brodersen. *Borph: An operating system for fpga-based reconfigurable computers*. University of California, Berkeley, 2007.

[255] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[256] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[257] Synopsys. Vcs functional verification solution. https://www.synopsys.com/verification/simulation/vcs.html, 2021.

[258] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[259] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Enabling flexible network fpga clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246. ACM, 2017.

[260] Terasic and Altera. De5a-net fpga development kit user manual. https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1804382103-de5a-net-user-manual.pdf.

[261] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[262] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *USENIX Annual Technical Conference*, pages 121–132, 2014.

[263] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[264] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE, 2018.

[265] Rakesh Vattikonda, Wenping Wang, and Yu Cao. Modeling and minimization of pmos nbti effect for robust nanometer design. In *Proceedings of the 43rd annual Design Automation Conference*, pages 1047–1052, 2006.

[266] Jyothi Bhaskarr Velamala. Compact modeling and simulation for digital circuit aging. PhD dissertation, 2012. https://repository.asu.edu/attachments/97628/content//tmp/package-sAOTlT/Velamala_asu_0010E_12271.pdf.

[267] Duy Viet Vu, Oliver Sander, Timo Sandmann, Steffen Baehr, Jan Heidelberger, and Juergen Becker. Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using pci express single-root i/o virtualization. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6. IEEE, 2014.

[268] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[269] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135. ACM, 2017.

[270] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. Grasu: A fast graph update library for fpga-based dynamic graph processing. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 149–159, 2021.

[271] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 216–230, 2023.

[272] Wei Wang, Miodrag Bolic, and Jonathan Parri. pvfpga: accessing an fpga-based hardware accelerator in a paravirtualized environment. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–9. IEEE, 2013.

[273] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*, pages 1078–1086. IEEE, 2015.

[274] Gabriel Weisz and James C Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[275] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C Hoe. A study of pointer-chasing performance on shared-memory processor-fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 264–273. ACM, 2016.

[276] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273, New York, NY, USA, 2016. ACM.

[277] Stephen Williams. Icarus verilog. http://iverilog.icarus.com/.

[278] Clifford Wolf. Yosys open synthesis suite, 2016.

[279] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[280] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. Virtual wifi: bring virtualization from wired to wireless. In *Acm sigplan notices*, volume 46, pages 181–192. ACM, 2011.

[281] Xilinx. Axi interconnect. https://www.xilinx.com/products/intellectual-property/axi_interconnect.html.

[282] Xilinx. Designing with sr-iov capability of xilinx virtex-7 pci express gen3 integrated block. https://www.xilinx.com/support/documentation/application_notes/xapp1177-pcie-gen3-sriov.pdf.

[283] Xilinx. Dma for pci express (pcie) subsystem. https://www.xilinx.com/products/intellectual-property/pcie-dma.html.

[284] Xilinx. Pcie solution portfolio. https://www.xilinx.com/products/technology/pci-express.html.

[285] Xilinx. Sdaccel development environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[286] Xilinx. Vitis high-level synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[287] Xilinx. Finite state machines. https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab10.pdf, 2015.

[288] Xilinx. Integrated logic analyzer v6.2. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf, 2016.

[289] Peter Xu. Device assignment with nested guest and dpdk. https://www.linux-kvm.org/images/a/a6/KVM_Forum_2018_viommu_vfio.pdf, 2018.

[290] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. Automatic virtualization of accelerators. In *17th Workshop on Hot Topics in Operating Systems (HotOS {XVII})*, 2019.

[291] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. Ava: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 807–825, 2020.

[292] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112. IEEE, 2012.

[293] H. Zeng, C. Zhang, and V. Prasanna. Fast generation of high throughput customized deep learning accelerators on FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, December 2017.

[294] Yue Zha and Jing Li. Virtualizing fpgas in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.

[295] Yue Zha and Jing Li. When application-specific isa meets fpgas: a multi-layer virtualization framework for heterogeneous cloud fpgas. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–134, 2021.

[296] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 22. ACM, 2017.

[297] Min Zhang, Linpeng Li, Hai Wang, Yan Liu, Hongbo Qin, and Wei Zhao. Optimized compression for implementing convolutional neural networks on fpga. *Electronics*, 8(3):295, 2019.

[298] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 815–827. IEEE Press, 2018.

[299] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. Fracbnn: Accurate and fpga-efficient binary neural networks with fractional activations. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 171–182, 2021.

[300] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[301] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.

[302] S. Zhou and V. K. Prasanna. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144, October 2017.

[303] Shijie Zhou and Viktor K Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.

[304] Kenneth M. Zick, Chien-Chih Yu, John Paul Walters, and Matthew French. Silent data corruption and embedded processing with nasa's spacecube. *IEEE Embedded Systems Letters*, 4(2):33–36, 2012.

[305] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *ACM SIGPLAN conference on Programming language design and implementation*, 2021.