Hardware-Software Co-design for Reliable and Efficient Heterogeneous Systems

by

Gefei Zuo

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

Doctoral Committee:

Professor Baris Kasikci, Co-Chair
Professor Manos Kapritsos, Co-Chair
Professor Peter Chen
Professor Hun Seok Kim
Professor Andrew Quinn

Gefei Zuo
gefeizuo@umich.edu
ORCID iD: 0000-0003-0861-5010

© Gefei Zuo 2024

DEDICATION

To Yirui, who has got my back in all these years.
And to my parents, Yi Zi (訾轶) and Xinling Zuo (左新岭), for all their support.

ACKNOWLEDGEMENTS

I would first like to thank my beloved Yirui, who provides unconditional support and makes this all possible. I am also grateful to my parents for their caring, encouragement and devotion, which have enabled me to pursue the Ph.D.

I'd like to thank my amazing advisor Baris, for spotting me in the pile of applicants back in 2018, then training me to become a better researcher in all aspects, from writing and communication to critical thinking and problem solving, etc.

I'd next like to thank all members from the Efeslab: Tanvir Khan, Jiacheng Ma, Kevin Loughlin, Ian Neal, Marina Minkin, Andrew Loveless and Andrew Quinn, with whom I felt a sense of belonging in the team. They provided me countless pieces of advice and help not only with my research projects, but also with day-to-day life as an international Ph.D. student. In particular, Tanvir, as the most senior student in the lab, essentially became my student mentor; and Jiacheng, who shared the most common language with me, also became my most frequent collaborator. The last shout-out is for Andrew Quinn, who joined the lab for a short period of time during my Ph.D., but since then started collaborating with me and eventually became my second academic advisor as well as a dissertation committee member.

I'd like to thank all professors who serve on my dissertation committee. Their support and insightful feedback helped me improve this dissertation and conclude this Ph.D. journey.

I'd like to thank my major funding sources, the Applications Driving Architectures (ADA) Research Center, and the Processing with In- telligent Storage and Memory (PRISM) Research Center, which generously supported my Ph.D. researches.

Reflecting things prior to but relevant for this Ph.D., I'd like to thank Bojie Li, who introduced me to research in undergrad and mentored me during the internship with Microsoft Research Asia.

I'd also like to thank those friends from the Linux User Group during my undergrad in the University of Science and Technology of China. I learned so many technical skills from that small group of people and was given a wonderful environment to practice. That experience has proven to be invaluable when carried over to my Ph.D., and I expect it will continue aiding me in the future.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES viii

ABSTRACT ix

CHAPTER

1 Introduction 1

1.1 Motivation 1
1.2 Wide Adoption of Heterogeneous Systems 2
1.3 Hardware-Software Co-design for Heterogeneous Systems 3
1.3.1 Hardware-assisted Software Failure Reproduction 3
1.3.2 Software-inspired Record Replay for Reconfigurable Hardware 4
1.3.3 Virtualization Support for Shared-Memory FPGA Platforms 5

2 Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction 6

2.1 Introduction 6
2.2 Motivation 9
2.2.1 Efficiency 10
2.2.2 Effectiveness 11
2.2.3 Accuracy 12
2.2.4 Summary 12
2.3 Design 13
2.3.1 Online Monitoring 14
2.3.2 Shepherded Symbolic Execution 14
2.3.3 Key Data Value Selection 17
2.3.4 Handling Concurrency 20
2.4 Implementation 21
2.5 Evaluation 22
2.5.1 Experimental Setup 23
2.5.2 Effectiveness and Accuracy of ER 24

2.5.3 Efficiency of ER . . . 26
2.5.4 Invariant-Based Failure Localization . . . 26
2.6 Related Work . . . 28
2.7 Chapter Summary . . . 29

3 Vidi: Record Replay for Reconfigurable Hardware . . . 30

3.1 Introduction . . . 30
3.2 Background and Observations . . . 33
3.2.1 Channels and Transactions . . . 33
3.2.2 Transaction Ordering in FPGAs . . . 34
3.3 Design . . . 35
3.3.1 Channel Monitor . . . 38
3.3.2 Trace Encoder . . . 39
3.3.3 Trace Store . . . 40
3.3.4 Trace Decoder . . . 41
3.3.5 Channel Replayer . . . 41
3.3.6 Handling Replay Divergence . . . 43
3.4 Implementation . . . 44
3.4.1 Hardware . . . 44
3.4.2 Software . . . 45
3.5 Evaluation . . . 45
3.5.1 Experimental Setup . . . 46
3.5.2 Debugging Case Study . . . 47
3.5.3 Testing Case Study . . . 48
3.5.4 Effectiveness of **VIDI** . . . 49
3.5.5 Efficiency of **VIDI** . . . 50
3.6 Discussion . . . 52
3.7 Related Work . . . 53
3.8 Chapter Summary . . . 54

4 Virtualization Support for Shared-Memory FPGA Platforms . . . 55

4.1 Introduction . . . 55
4.2 Background . . . 57
4.2.1 FPGA Programming Models . . . 57
4.2.2 FPGA Virtualization . . . 59
4.3 Goals and Challenges . . . 60
4.4 Design . . . 61
4.4.1 Userspace Library for Spatial Multiplexing . . . 62
4.4.2 Preemption Interface for Temporal Multiplexing . . . 63
4.5 Implementation . . . 64
4.6 Evaluation . . . 65
4.6.1 Experimental Setup . . . 66
4.6.2 Performance Overhead . . . 68
4.6.3 Scalability of Spatial Multiplexing . . . 69
4.6.4 Scalability of Temporal Multiplexing . . . 70

4.6.5 Fairness of Temporal Multiplexing 71
4.7 Related Work 72
4.8 Chapter Summary 74
5 Future Work **75**
6 Conclusion **77**
BIBLIOGRAPHY 78

LIST OF FIGURES

FIGURE

1.1 Microprocessor Trend Data from 1971 to 2021. 2
2.1 Prior techniques on a spectrum for each failure reproduction property. 9
2.2 The high-level design and usage model of ER 13
2.3 An example program for ER to analyze 15
2.4 Step-by-step construction of the constraint graph for the example in Fig. 2.3. 17
2.5 Number of instructions executed and the time spent on shepherded symbolic execution for PHP-74194. 25
2.6 Runtime performance overhead incurred by ER's control and data flow recording as well as recording using rr [39]. 27
3.1 An example waveform of a VALID/READY handshake on an AXI Channel. Each line represents the values of a different signal in the communication channel. 33
3.2 Example group of FPGA-CPU communication channels. 35
3.3 VIDI's design overview. 37
3.4 A channel monitor deployed on an input channel. 38
3.5 Architecture of VIDI's Trace Encoder. 40
3.6 Architecture of the Trace Decoder and Channel Replayers with an example trace. . . . 42
3.7 A break-down of VIDI resource overhead when monitoring different combinations of AWS F1 AXI interfaces. 51
4.1 Illustration of the host-centric programming model 58
4.2 Illustration of the shared-memory programming model 58
4.3 OPTIMUS design overview, shown with two physical accelerators for brevity. 62
4.4 Performance overhead of different benchmarks compared to pass-through. 68
4.5 Average memory access latency of LinkedList with different working set sizes and number of virtual machines. 69
4.6 Aggregate throughput of MemBench with different working set sizes and number of virtual machines. 70
4.7 The aggregate throughput of different real-world applications, normalized to the throughput of a single VM. 71
4.8 Normalized aggregate throughput in the presence of preemptive temporal multiplexing. All virtual accelerators are scheduled on a single physical accelerator. 72

LIST OF TABLES

TABLE
2.1 Bugs used in the evaluation of ER along with bug IDs and types. 23
3.1 The applications used to evaluate VIDI. 45
3.2 On-FPGA resource overhead of VIDI, broken-down by resource types (i.e., LUT, FF, and BRAM) and normalized to the resource available on the F1 FPGA. 50
4.1 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. . . . 66

ABSTRACT

Transistor scaling has been slowing down in the past decade, while the demand for computing continue rising up, especially the exponential growth recently driven by machine learning. As a result, system builders are increasingly turning to heterogeneous systems to accommodate the ever-growing demand. The design philosophy of Heterogeneous systems is to trade general purpose for better performance or energy efficiency, etc., which often results in integrating specialized and non-uniform hardware for different use cases. Such trend has been observed across the stack of computing infrastructure, from computing units to storage and networking devices, even heterogeneity within a single chip.

As specialized hardware emerges, it is important to rethink system design and expand existing system support to new heterogeneous setups. This dissertation presents my efforts to improve the reliability and efficiency of heterogeneous systems by applying hardware-software co-design to answer the following two questions: 1) How to best utilize new specialized hardware to improve solutions to an existing problem. 2) How to adapt existing systems support to new heterogeneous setups.

In particular, this dissertation will first introduce ER, which utilizes specialized on-CPU tracing hardware to improve failure reproduction, a classical problem at the heart of software reliability. ER combines on-CPU tracing acceleator with symbolic execution to address the path explosion and constraint complexity challenges, while still accurately reproducing the failure with low overhead.

The second part will introduce VIDI, which generalizes the record/replay system support to CPU/FPGA systems and improves hardware program reliability. VIDI utilizes widely-adopted communication protocols in hardware programming to hide cycle-specific details from developers while still enforcing the happens-before relationship and preserving communication semantics.

Finally, this dissertation will discuss my contribution to a hypervisor project , OPTIMUS, which targets a special type of cache-coherent shared-memory CPU-FPGA heterogeneous systems. It supports both spatial multiplexing (VMs simultaneously sharing different partitions of the FPGA) and temporal multiplexing (VMs time-sharing the same part of the FPGA), in order to improve the hardware utilization and efficiency in a cloud environment.

CHAPTER 1

Introduction

1.1 Motivation

With the slow down of transistor performance scaling (e.g., density, cost, power, frequency and energy efficiency), heterogeneous systems have become an increasingly important strategy to accommodate the evergrowing demand of computing resources. Historically, two transistor scaling trends have driven advances in computer systems for decades. First, the Moore's Law projects transistor density would double about every two years. Second, the Dennard scaling predicts that the transistor size scales down with generations of technology node, but the power density remains constant.

However, there is a limit, as we can observe from the trend of microprocessor specifications [173] (Fig. 1.1), which is one of the most important semiconductor devices in computer systems. The Dennard scaling started failing first at around 2004, when circuits frequency, as a proxy to single-thread performance, became capped by power consumption and dissipation. Such a power wall forced computer system designers to turn to multi-core designs, or parallel computing in general, to practically utilize the increasing transistor density. Unfortunately, parallel computing was fundamentally limited by Amdahl's Law and stopped delivering benefits after exhausting different sources of parallelism. To make things worse, formidable challenges also slowed down the Moore's Law. Although transistor density seems to still maintain an exponential growth almost 50 years later, the cost of leading-edge technology node has been rising drastically accordingly. Recent semiconductor market analysis has reported that the research and development costs for chip design and fabrication factory construction are soaring super linearly with each generation the technology node advances [118]. From the chip designers' perspective, every step in the design process costs more (e.g., validation, prototype, software, physical, verification, etc., data reported in [164]), while the cost per logic transistor drops slower [139]. From the fabricators' perspective, the skills and investment required to pursue advanced technology node are so enormous that there are fewer and fewer companies still in the competition of chasing Moore's Law (from more than 30 companies at

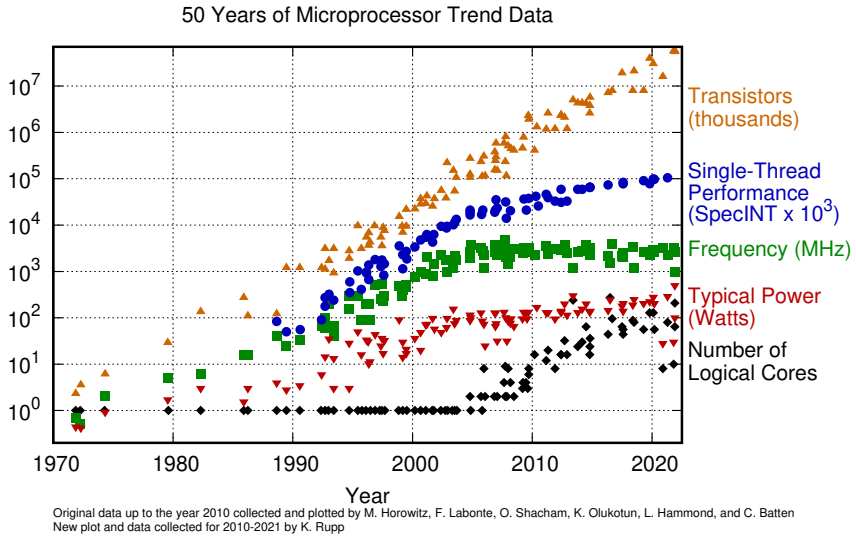


Figure 1.1: Microprocessor Trend Data from 1971 to 2021.

the 180 nm node to only 3 companies at the 5nm node) [258].

As a result, it is no longer feasible to soly rely on transistor scaling to catch up with the computing demand. One promising way forward is the so-called heterogeneous system, which consists of a set of specialized hardware/chips, each tailored towards a specific task, instead of using a general purpose hardware/chip for everything.

1.2 Wide Adoption of Heterogeneous Systems

Heterogeneous systems are becoming pervalent across the stack of computing infrastructure as more and more specialized hardware are designed and deployed to meet the evergrowing computing demand:

- For computing units, besides the classic general purpose CPUs, there are also GPUs specialized for massive parallel computing such as graphics and machine learning workloads; FPGA specialized for gate-level reconfigurable circuits; VCU specialized for video transcoding; TPU specialized for large matrix operations; etc.
- For storage devices, besides the regular DDR SDRAM, there are also High Bandwidth Memory (HBM), Non-volatile Memory (NVM), the Compute Express Link (CXL) that enables even more flexible memory expansions and programmable smartSSDs for near-memory computation.
- For networking devices, there are programmable switches, NICs that can offload certain network (IP/UDP/TCP, etc.) stack and programmable smartNICs for in-network computation.

- Heterogeneity also appears within a single chip. The traditional general purpose CPU has evolved to integrate various accelerators for vectorization, matrix operations, encryption/-compression, data movement, etc. The big/little core design, which are often used in mobile designs, are now adopted in desktop and server processors. The chiplet architecture, which leads to heterogeneity such as non-uniform cache and memory access even within one chip, is also widely adopted in current and future generations of commercial processors and GPUs.

Although specialized hardware is often more performant and energy efficienct than their general purpose counter parts for certain tasks, heterogeneity brings new challenges such as task placement, scheduling and different programming abstractions, etc.

1.3 Hardware-Software Co-design for Heterogeneous Systems

In this dissertation, I will aim to improve the reliability and efficiency of various types of heterogeneous systems by applying hardware-software co-design techniques to answer two questions:

1. How to best utilize new specialized hardware to improve solutions to an existing problem.
2. How to adapt existing system supports to new heterogeneous setups.

In particular, my work covers the following three aspects:

- Chapter 2: Utilize on-CPU specialized tracing hardware to improve failure reproduction, a classic problem in software reliability. This work combines hardware tracing with symbolic execution based software analysis to reconstruct program executions accurately and efficiently.
- Chapter 3: Generalize record replay, a classic software systems support, to CPU-FPGA heterogeneous systems, which has been widely deployed in data centers and available in numerous cloud providers. This work utilizes hardware programming abstractions and happens-before encoding to efficiently reproduce FPGA program executions.
- Chapter 4: Extend virtualization to a special type of cache-coherent shared-memory CPU-FPGA heterogeneous systems to improve utilization and cloud efficiency.

1.3.1 Hardware-assisted Software Failure Reproduction

Reproducing production failures is crucial for software reliability. Alas, existing bug reproduction approaches are not suitable for production systems because they are not simultaneously efficient, effective, and accurate. In this work, I survey prior techniques and show that existing approaches over-prioritize a subset of these properties, and sacrifice the remaining ones. As a result, existing

tools do not enable the plethora of proposed failure reproduction use-cases (e.g., debugging, security forensics, fuzzing) for production failures.

I propose Execution Reconstruction (ER), a technique that strikes a better balance between efficiency, effectiveness and accuracy for reproducing production failures. ER uses hardware-assisted control and data tracing to shepherd symbolic execution and reproduce failures. ER's key novelty lies in identifying data values that are both inexpensive to monitor and useful for eliding the scalability limitations of symbolic execution. ER harnesses failure reoccurrences by iteratively performing tracing and symbolic execution, which reduces runtime overhead. Whereas prior production-grade techniques can only reproduce short executions, ER can reproduce any reoccuring failure. Thus, unlike existing tools, ER reproduces fully replayable executions that can power a variety of debugging and reliabilty use cases. ER incurs on average 0.3% (up to 1.1%) runtime monitoring overhead for a broad range of real-world systems, making it practical for real-world deployment.

1.3.2 Software-inspired Record Replay for Reconfigurable Hardware

Developers are turning to heterogeneous computing devices, such as Field Programmable Gate Arrays (FPGAs), to accelerate data center workloads. FPGAs enable rapid prototyping and should facilitate an agile software-like development workflow to fix correctness bugs, performance issues, and security vulnerabilities. Unfortunately, hardware development still does not have a vast ecosystem of tools needed to support the agile hardware development vision. The capability to record and replay FPGA executions would constitute a key building block that will inspire the development of many tools, similar to what record/replay did for software. However, building a practical record/replay tool for FPGAs is challenging; existing approaches either record too much or too little information and cannot support real-world executions.

In this work, I present VIDI, the first record/replay system for real-world FPGA applications running on hardware. VIDI is based on the observation that widely-used communication protocols have well-defined input/output transactions to hide cycle-specific information from developers, which enables a more efficient design than heavyweight cycle-accurate record/replay approaches. VIDI proposes (1) the *transaction determinism* insight to track and enforce only necessary orderings of transaction events across record and replay, and (2) the *coarse-grained input recording* mechanism to record transaction-level information. VIDI is evaluated on Amazon EC2 F1 instances with 10 applications and two use cases (debugging, testing) and find that it incurs on average low performance slowdown (1.98%) and resource overhead (5.48%), making it practical for real-world deployments.

1.3.3 Virtualization Support for Shared-Memory FPGA Platforms

In cloud computing, field-programmable gate arrays (FPGAs) have become prevalent as dedicated accelerators in heterogeneous systems to enhance application-specific performance. Cloud providers are incentivized to harness FPGA virtualization to multiplex resources among tenants, thus optimize for high utilization and low operational costs. Traditional virtualization approaches have predominantly focused on host-centric FPGA programming model, which is also the most popular choices among other accelerators (e.g., GPU, TPU). However, the host-centric model prevents FPGA accelerators from initiating direct memory access (DMA) operations thus suffer from performance degradation on workloads manifesting pointer-chasing style behaviors. In response to such limitation, there is a shift towards incorporating a shared-memory programming model to FPGAs, permitting accelerators to arbitrarily access memory bypassing the CPU. Notwithstanding, the virtualization support extending to these shared-memory FPGAs remains scarce.

This project introduces OPTIMUS, a first-of-its-kind hypervisor designed explicitly to achieve scalable virtualization of shared-memory FPGAs. The core of OPTIMUS is a hypervisor spanning both software and hardware, aiming to isolate virtual memory address spaces between users while allowing the CPU and FPGA programs of the same user to share the same virtual memory address space. OPTIMUS supports both spatial and temporal multiplexing, enhancing the efficiency and versatility of accelerator sharing within an FPGA. For spatial multiplexing, OPTIMUS proposes the shadow page-table slicing technique, for which I build the guest driver and userspace libraries in software. For temporal multiplexing, I design and implement a preemption interface that allows the hypervisor to allocate and schedule virtual machines to use physical acceleators by time slices.

Empirical evaluations on the Intel-HARP platform demonstrate that OPTIMUS can concurrently support up to eight physical acceleators on a single FPGA. A suite of twelve real-world benchmarks shows that OPTIMUS significantly improves their aggregated throughput by 1.98x to 7x. I also built two microbenchmarks to characterize the latency (up to 24%) and bandwidth (up to 10%) overhead of OPTIMUS under stress tests.

CHAPTER 2

Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction

2.1 Introduction

Failure reproduction is critical for software reliability. Not only is failure reproduction the first step in effective debugging [8, 30, 181, 187, 212], it also enables tools across many software reliability domains including security forensics [95], configuration management [54], and directed testing and fuzzing [83]. These tools are particularly valuable when used on *production failures*, because such failures may be elusive and difficult to replicate in house, outside of specific deployment scenarios. For instance, eliminating a production bug can drastically improve availability [11–13, 115], or a security audit of a production breach can help with leak assessment [210, 218, 262].

Many techniques automate the process of reproducing failures in software, yet, none offer comprehensive support for production usage. Prior systems sacrifice at least one of three key properties that are necessary for failure reproduction in production settings: (1) *efficiency*, which relates to the amount of resources expended to reproduce failures, (2) *effectiveness*, which relates to the capability to reproduce different kinds of bugs (e.g., concurrency bugs [149] and bugs where the failure and root cause are distant [107]), and (3) *accuracy*, which relates to whether the control flow and the data values of an execution can be recovered correctly. Our motivating study (§2.2) reveals that existing bug reproduction techniques over-prioritize a subset of the efficiency, effectiveness and accuracy properties and thus over-sacrifice in the other properties. Consequently, when used in production systems, existing techniques cannot enable the plethora of use cases (e.g., debugging, security forensics, fuzzing, etc.) that are otherwise available.

For instance, record/replay systems [47, 119–121, 170, 185, 193, 203, 211, 244] can accurately reproduce any execution and are thus very effective in reproducing bugs, however, they incur high overheads (e.g., up to 2× for a state-of-the-art system [244]), making them unsuitable for production use cases. Many techniques reduce runtime overhead by reproducing failures using a combination of online (runtime) recording and offline program analysis [91, 108, 198, 254, 268, 281]. For instance,

REPT [254] and POMP [268] use hardware tracing to efficiently record a program's control flow at runtime and recover the program's data values via static analysis. However, since programs overwrite data values frequently, these tools cannot accurately reproduce full executions with only control-flow information and the final program state. It is difficult to debug programs with unreliable control flow and data information, and, even worse, these systems cannot be used in many domains including security forensics, directed testing and fuzzing, and configuration management.

We present Execution Reconstruction (ER), a hybrid offline/online failure reproduction technique that explores a sweet spot in the efficiency-effectiveness-accuracy trade-off space. Compared to prior work, ER's main contribution lies in determining a subset of the execution information that is both inexpensive to monitor online and useful for failure reproduction. ER is efficient enough to be deployed in production (i.e., it incurs on average 0.3% runtime overhead); effective in reproducing different kinds of bugs; and accurate in reproducing data values and control flow that lead to the original failure, which enables a variety of debugging and analysis use cases for production scenarios.

At the heart of ER lies *shepherded symbolic execution*, which uses dynamic control flow and data value information to reconstruct an execution that leads to a production failure. Rather than exploring exponentially many control-flow paths, ER follows a single recorded path that leads to a failure, thereby eliminating the notorious path explosion problem of symbolic execution. Symbolic execution computes constraints on program inputs that lead to the failure. Once ER completes symbolic execution along the entire path that was recorded during the failing production run, it invokes a constraint solver to compute failure-inducing program inputs.

Unfortunately, for real-world programs, eliminating path explosion is not sufficient to reconstruct an execution via symbolic execution. The constraints on input values grow very complex, which prevents symbolic execution from making progress (i.e., symbolic execution stalls) and eventually causes constraint solving to timeout (see §2.4 for how stalls occur in practice). Our evaluation shows that shepherded symbolic execution using only a control-flow trace stalls on 11/13 failures (§2.5). ER overcomes this challenge by carefully selecting and recording a minimal set of data values that simplify constraint solving and thus eliminate slowdowns in symbolic execution without rendering ER too inefficient for production use.

Ideally, ER would select the data values that optimally simplify constraint solving, but this is intractable. Instead, ER turns to a novel heuristic to identify data values that drastically simplify constraint solving in practice. ER builds and analyzes a constraint graph during symbolic execution to determine *key data values* (e.g., instruction operands, return values, addresses, parts of the input), which are involved in complex constraint dependencies. In particular, ER identifies two patterns in the constraint graph that often cause constraint solver timeouts: long chains of symbolic writes, in which each write in the chain is dependent upon a previous write in the chain, and accesses to large

symbolic memory objects, which can be difficult to reason about, as many memory locations may be accessed. If known, the data values from these patterns substantially simplify constraint solving. In our evaluation, the selection heuristic identifies a small set of data values that simplify constraint solving such that ER can reconstruct failures with negligible overhead.

Anticipating the key data values and recording them before a failure is difficult. Key data values depend on the constraints in symbolic execution, which ER cannot predict without symbolically executing the program. Thus, to reproduce a production failure, ER introduces a novel iterative algorithm that leverages frequent failure reoccurrences in large scale deployments, similar to existing production monitoring systems [111, 140–142]. In each iteration, ER gathers more data about the reoccurring failure, uses symbolic execution to determine constraints of the failing execution, selects additional data values that can help symbolic execution complete, and collects more data from programs in the next iteration. To bootstrap this process, ER uses always-on control-flow recording. When symbolic execution completes, ER determines program inputs that lead to the original failure and stops iterating. Using this iterative process, ER guarantees reconstruction of an entire failing execution for any reoccuring failure. In certain cases (i.e., 2/13 cases in our evaluation), ER can reproduce a failing execution even after a single occurrence of a failure. In practice, ER reproduces failures in only a few occurrences (only requiring an average of ˜3.5 occurrences in our evaluation).

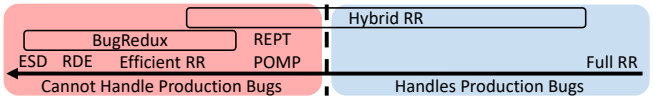
ER resides at the sweet spot in the efficiency-effectiveness-accuracy trade-off. By using hardware tracing, ER is *efficient* enough for production use, as it incurs on average 0.3% (up to 1.1%) runtime overhead during online monitoring. By using the iterative approach to shepherded symbolic execution, ER is *effective* in reproducing failures in long-running real-world program executions. Finally, by using shepherded symbolic execution, ER is *accurate* enough for a plethora of software reliability use cases (debugging, security forensics, fuzzing, etc.). ER generates program inputs that are guaranteed to lead to the original production failure. While ER-generated inputs may differ from actual inputs that led to the production failure, we empirically validate ER is useful for debugging and other software reliability use cases, including a case-study that uses likely program invariant computation to perform automated failure localization.

To summarize, we propose Execution Reconstruction (ER), an end-to-end technique for reproducing production failures and make the following contributions:

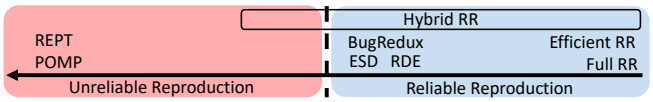
- *Shepherded Symbolic Execution*, which follows a control flow and data value trace recorded in production to eschew path explosion and solver stalls that hinder traditional symbolic execution.
- *Key Data Value Selection*, which identifies data values that drastically speed up constraint solving and can be monitored efficiently with existing hardware support.
- An evaluation of ER on 13 bugs in real systems (e.g., memcached, SQLite, python, etc.) shows that ER is efficient (on average 0.3% and up to 1.1% runtime overhead), effective, and accurate enough for production



(a) Efficiency. Systems on right have production-grade overhead.



(b) Effectiveness. Systems on right reproduce all production bugs.



(c) Accuracy. Systems on right reliably reproduce failures.

Figure 2.1: Prior techniques on a spectrum for each failure reproduction property. RR stands for record /replay. Hybrid RR and BugRedux are shown as boxes spanning a property range.

- An invariant-based failure localization case-study that shows that ER can enable production use of software reliability tools.

2.2 Motivation

Ideally, production-grade failure reproduction systems would achieve three key properties: (1) *efficiency*, which is the ability to incur low overhead and use few resources, (2) *effectiveness*, which is the ability to reproduce different kinds of failures (e.g., due to concurrency bugs or latent bugs where the failure (e.g., a crash) and the bug (e.g., an overflow) are distant), (3) *accuracy*, which is the ability to recover the control flow (branches executed) and data (values read/written) of the failing execution correctly. Unfortunately, all known approaches that achieve effectiveness and accuracy on multi-threaded applications (e.g., record/replay systems) record detailed execution information, which imposes high overhead. As a result, practical failure reproduction systems for production must trade-off one or more desirable properties.

We study prior failure reproduction systems and place them on a spectrum for each failure reproduction property (see Fig. 2.1). For each property, we identify a *usability-boundary* (shown as a vertical dotted-line in Fig. 2.1), an inflection point at which a failure reproduction system shifts from an undesirable trade-off to an acceptable one. While many systems are usable in one or two properties, no existing failure reproduction system is simultaneously efficient enough for production, able to handle common production bugs, and able to reliably reproduce an execution. Thus, no existing failure reproduction system provides *production* support for the powerful software

reliability tools that have been developed for debugging, security forensics, fuzzing, etc. Below we place systems onto each spectrum and construct a usability-boundary for each property.

2.2.1 Efficiency

Efficiency, or low runtime performance overhead, is paramount for a production-grade system. A few failure reproduction techniques maximize efficiency by using an offline analysis. For instance, ESD [274, 275] takes as input a failure (e.g., a deadlock) location, statically computes intermediate statements that the program must execute before the failure (e.g., locks before a deadlock), and steers symbolic execution towards intermediate statements and then towards the failure. Similarly, RDE [247] imposes negligible overhead because it uses existing or light-weight data from the production environment (application logs, system call sequences) to guide symbolic execution towards a failure-inducing input.

Purely offline approaches struggle to achieve effectiveness (see §2.2.2), so many systems record execution information at the cost of efficiency. Record/replay systems [39, 119–121, 169, 170, 180, 185, 203, 211, 244] record all program inputs and other non-deterministic events. Full record/replay systems [39, 244] record all inputs and all non-deterministic events resulting in high overhead that is unsuitable for production usage (up to 2× for a state-of-the-art system [244]); efficient record/replay systems [95, 169, 180] record fewer events but are less effective than full record/replay systems (i.e., they cannot handle data races, see §2.2.2).

Hybrid online/offline techniques provide a middle-ground; they achieve better effectiveness than offline techniques by recording some execution information, while achieving better efficiency than record/replay by offloading analysis to offline processing. BugRedux [137] uses symbolic execution to synthesize an execution that reaches the locations traced by one of two approaches: complete tracing records all control flow (up to 10× overhead), or the more efficient call sequence tracing, which records each function call (between 2%–50% overhead). Similarly, hybrid record/replay techniques (e.g., PRES [198] and ODR [47]) search for an execution that reaches execution states monitored using recording granularities that occupy different points in the efficiency-effectiveness-accuracy trade-off space: the finest-grained tracing records full program traces and has high overhead (3×-20×) but good effectiveness/accuracy, while coarse-grained modes impose lower overhead by only recording non-deterministic input and synchronization operations, but have poor effectiveness/accuracy. In Fig. 2.1, we plot ranges for BugRedux and hybrid record/replay systems.

Recent hybrid approaches, REPT [254] and POMP [268], use hardware-assisted tracing (e.g., Intel Processor Trace (PT) [133]) to record control-flow information with low overhead (below 5% [140, 254]). These approaches use static analyses that perform reverse and/or forward execution along the trace to reconstruct program state.

To identify a usability boundary, we use the analysis of always-on record/replay systems [95, 180], which suggest that run-time overhead of 10% or less is acceptable in production. We consider offline, recent hybrid approaches (REPT and POMP), and efficient record/replay systems to be efficient-enough for production use. Full record/replay and BugRedux are too inefficient for production use. Finally, hybrid record /replay systems are efficient-enough when using coarse-grained recording, but are too inefficient when using fine-grained recording. Fig. 2.1b depicts this spectrum.

2.2.2 Effectiveness

A failure reproduction system is only useful if it can effectively reproduce common production failures. Full record /replay systems [39, 244] are maximally effective since they can reproduce any error in any application. Efficient record /replay systems [95, 169, 180] have unacceptable effectiveness for production failure reproduction since they cannot replay executions that contain data-races, an important class of production bugs [107, 212]. REPT has limited effectiveness since it does not guarantee that it can reconstruct data for arbitrary failures. In particular, REPT can only reconstruct short execution fragments (15-60% of the data values are incorrectly recovered for traces longer than 100K dynamic instructions) because programs overwrite data values frequently. Thus, REPT is not effective for latent bugs, where failures are distant from the root cause. Latent bugs are an important and complex class of production incidents [89, 90, 107, 114, 234, 235]. Prior techniques that rely on symbolic execution (ESD, BugRedux, and RDE) are not guaranteed to reproduce failures because constraint solvers may timeout on real failures. Finally, hybrid record/replay systems are efficient when using fine-grained recording, but can struggle to reproduce an execution when using coarse-grained recording.

We find that very few solutions are effective enough for production usage. Existing solutions that rely on symbolic execution are not guaranteed to reproduce failures and are undesirably ineffective. In addition, the inability of efficient record/replay systems to handle data races and the latent bug restrictions of REPT render these tools similarly ineffective, since these bug classes are prevalent in production. We identify a usability-boundary inspired by the coarse interleaving hypothesis introduced by Snorlax [140] and adopted by REPT [254]. The coarse interleaving hypothesis observes that most events in concurrency bugs [143, 145] are interleaved coarsely (e.g., 10s of microseconds separate the events). Thus, we consider failure reproduction tools that can reproduce any bug not violating the coarse interleaving hypothesis to be effective-enough for production use, since these tools can faithfully reproduce errors that arise in production (shown in Fig. 2.1b). While REPT reproduces executions that satisfy the coarse interleaving hypothesis, it cannot support latent bugs and is thus not effective enough for production bugs.

2.2.3 Accuracy

All record/replay systems are accurate since they faithfully reproduce all state in an execution [39, 95, 169, 180, 244]. ESD and other offline tools achieve a useful accuracy property—while ESD does not always reproduce the same data and control flow as the failure, it reproduces an execution that will lead to the same failure. Hybrid record/replay systems offer accuracy between record/replay and offline tools, depending on the recording granularity. REPT guesses data values during post-mortem analysis, which yields incorrect register or memory values in the majority of the reconstructed executions—in its original evaluation, all failure reproductions contain incorrect values [254]. REPT is especially inaccurate for bugs with long traces; for traces longer than 100K instructions, 15%-60% of the values were incorrectly recovered by REPT. Even worse, incorrect values are not guaranteed to be consistent with recorded control-flow trace and may not be detectable without reproducing the failure, thereby misleading developers.

Record/replay systems, offline techniques like ESD, and hybrid record/replay techniques produce control flow and data that lead to the failure. Thus, developers can trust the output of these tools. In contrast, best-effort tools, such as REPT, produce output with missing and/or inconsistent data and control-flow. Inaccuracy is a major obstacle in practice—the reliability of the tools and the accuracy of their results is particularly important for developers [142], because testing and debugging often requires significant effort and time [181]. Moreover, the output from best-effort failure reproduction systems is not executable, which makes it impossible to leverage dynamic tools on top of these systems (see §2.2.4). We identify a usability-boundary at this inflection point: a tool is acceptably accurate if it is guaranteed to reproduce a replayable execution with the same failure.

2.2.4 Summary

Existing bug reproduction tools are not sufficiently efficient, effective, and accurate to (1) ensure low overheads that are compatible with the performance requirements of production settings, (2) reproduce complex failures found in production environments, and (3) provide reliable and executable output. We find that prior systems over-prioritize one or more properties at the expense of the others. Offline and hybrid solutions, like REPT, are overly efficient; they record little to no information, which imposes negligible run-time overhead but sacrifices efficiency/accuracy. Full record/replay systems are overly effective; they support bugs which violate the coarse interleaving hypothesis, even though those bugs do not appear in practice [140]. Efficient record/replay systems are overly accurate; they are guaranteed to reproduce the exact control flow and data values from a failure, but, in software reliability domains (e.g., debugging, fuzzing, security forensics), it is sufficient to reproduce any execution that leads to the same failure [274] (see §2.5).

In contrast, ER lies at the usability-boundary of each failure reproduction property. Thus, ER

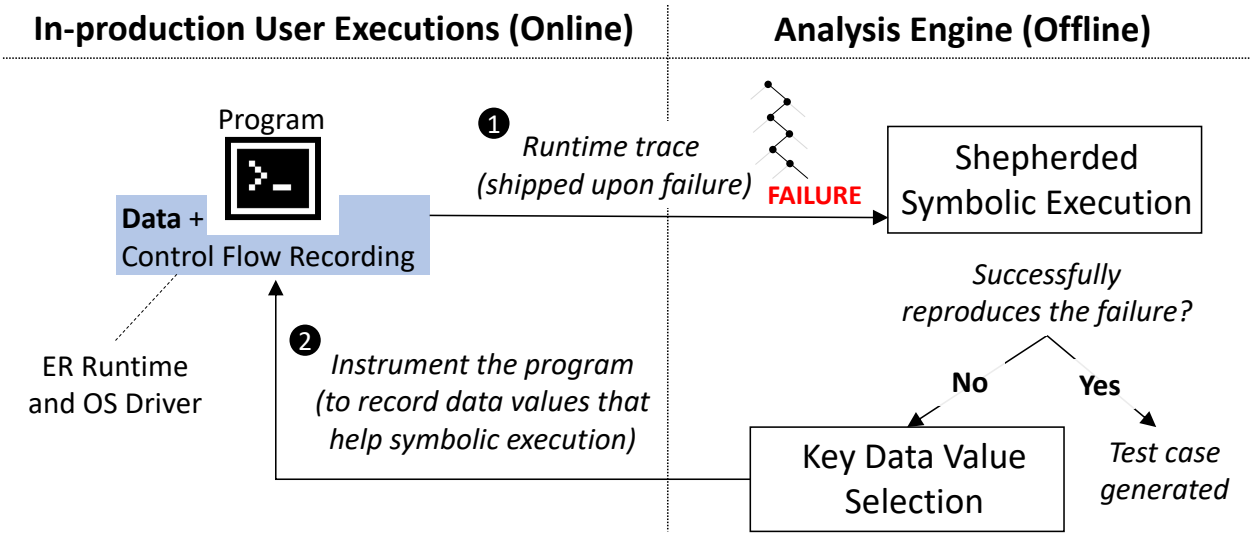


Figure 2.2: The high-level design and usage model of ER

reliably reproduces production failures with production-grade efficiency and adds production support for the wide-range of dynamic tools that are built on top of failure reproduction systems. For example, ER enables tools for debugging [119], configuration management [54], security forensics [95], automated failure localization [213, 291], large-scale parallelism [208], and bug detection [83]. In §2.5, we provide a case study that uses ER for invariant-based failure localization.

2.3 Design

ER resides at the sweet spot in the design space of bug reproduction systems with respect to the aforementioned key properties of efficiency, effectiveness, and accuracy. ER uses execution information (control flow and data values) in conjunction with symbolic execution to reconstruct a failure. Through careful combination of these techniques, ER simultaneously mitigates the efficiency problems of purely online approaches and the effectiveness problems of purely offline approaches. At a high-level, ER follows an iterative algorithm that monitors production failures and performs symbolic execution to either generate a failure-inducing input or identify the reason why inputs cannot be generated. In the latter case, ER selects new data values to monitor which allow symbolic execution to generate a failing input during future failure occurrences.

Fig. 2.2 shows the high-level design of ER. ER provides a runtime that monitors an execution to produce a trace of control flow and data values as per the direction of ER's Analysis Engine (§2.3.1). When a failure occurs, the runtime ships the failure trace to the analysis engine to perform shepherded symbolic execution. Shepherded symbolic execution uses the runtime trace recorded in production to reconstruct an execution that reproduces the failure (§2.3.2). Control-flow

information allows shepherded symbolic execution to eschew the notorious path-explosion problem in symbolic execution. However, the constraints on inputs nevertheless grow complex and stall symbolic execution, eventually leading to constraint solver timeouts. So, ER performs key data value selection, which analyzes the constraints from symbolic execution to identify a small set of data values that will simplify constraints and thus eliminate stalls in symbolic execution(§2.3.3). ER instruments the program to record selected data values, redeploys the program in production, and waits until the failure reoccurs. This process continues until ER successfully generates an input that leads to the failure.

We describe how ER performs online monitoring (§2.3.1), shepherded symbolic execution (§2.3.2), and key data value selection (§2.3.3), and how ER supports concurrency (§2.3.4).

2.3.1 Online Monitoring

ER uses the runtime and driver support to monitor a production failure using efficient hardware support (via Intel PT [133]). ER records the control flow and data values of production executions, as per the directions of ER's key data value selection (§2.3.3). By default, ER always traces control-flow, because control-flow information is useful to eliminate path explosion (§2.3.2) and existing hardware techniques enable non-intrusive control-flow recording (i.e., executions can be traced without any modifications). If desired, developers can configure ER to enable tracing only after a failure is observed multiple times.

When a failure is detected, ER ships the runtime trace including the control-flow trace and any data values to shepherded symbolic execution (§2.3.2, step ❶). The failure can either be fail-stop (e.g. crashes, hangs, etc.) or programmatically detectable by other criteria (e.g. semantic bugs checked by developer-specified assertions).

2.3.2 Shepherded Symbolic Execution

Shepherded symbolic execution generates program inputs that reproduce a production failure by following the runtime trace produced during online monitoring and gathering constraints on input variables. The runtime trace includes the control flow of the failing execution (i.e., the branches, calls, and returns) and any data values (e.g., arguments to a function, return values, etc.) selected by key data value selection (§2.3.3). The trace of a failure reoccurrences may have subtle discrepancies with previous traces for the same failure; shepherded symbolic execution always uses the latest trace.

To demonstrate ER's end-to-end operation, we use the listing in Fig. 2.3. If this program aborts (Line 11), then `x` equals `d`, due to the statements `V[V[x]] = x` (Line 8) and `if (V[V[d]] == x)` (Line 10). Although this program is simple, it shows the code patterns that cause symbolic execution stalls in real-world applications, as discussed in §2.3.3.

```
1 uint32 V[256] = {0};
2 foo(uint32 a, uint32 b, uint32 c, uint32 d) {
3   uint32 x = (a + b);
4   if (x < 256 && c < 256 && d < 256) {
5     V[x] = 1;
6     if (V[c] == 0) // x != c
7       V[c] = 512;
8     V[V[x]] = x;
9     if (c < d)     // d != c
10      if (V[V[d]] == x)
11        abort();
12  }
13 }
```

Figure 2.3: An example program for ER to analyze

In our example, we make a few assumptions: (1) `foo` is called as `foo(0,2,0,2)`, which satisfies all the branch conditions, (2) the program crashes on Line 11 due to the `abort`, and (3) this is the first time the failure has occurred (so the runtime trace only contains control-flow information). The control-flow trace of the failing execution (i.e., the line numbers of the executed statements after each taken branch) in Fig. 2.3 is **2→ 5 → 7 → 10 → 11 (FAILURE)**. The ER runtime ships this control-flow trace to the analysis engine.

By following a control-flow trace from a production failure, shepherded symbolic execution eliminates the path explosion challenge, since it only explores a single program state that follows the trace. ER still executes the program with symbolic inputs and builds up a *path constraint*. A path constraint is a collection of constraints on the program input at any given point in symbolic execution.

In our example, ER starts executing `foo` with four symbolic (unconstrained) arguments, λ_a, λ_b , λ_c and λ_d (Line 2). Since all the operands are symbolic, the value of `x` computed on Line 3 is also symbolic. The branch on Line 4 compares symbolic inputs (λ_c, λ_d) and the symbolic value `x` to 256. Since the next instruction executed in the recorded control-flow trace is the memory write at Line 5 (see the above trace in bold), the branch on Line 4 was taken and ER will update the path constraint to include the outcome of the branch condition, i.e. $((\lambda_a + \lambda_b) < 256) \wedge (\lambda_c < 256) \wedge (\lambda_d < 256)$.

Line 5 uses `x` to dereference an entry in `V` and creates a symbolic memory address (the base address of `V`, offset by $(\lambda_a + \lambda_b)$). ER cannot determine the addresses of `V[x]`, but the fact that the control-flow trace has more elements indicates that dereferencing `V[x]` does not lead to the failure. The control-flow trace indicates that the branch on Line 6 was also taken. Since `V` is updated at a symbolic address on Line 5, the branch condition on Line 6 needs to treat array `V` symbolically: $(\text{Read}(\text{Write}(V, x, 1), c) == 0)$. Here, we use a standard representation of accesses to symbolic memory [94, 109]: 1) $\text{Read}(A, i)$ returns the value at location $A[i]$, where A is an array and i is the index. 2) $\text{Write}(A, i, v)$ updates the i-th entry in the array A with value v.

ER continues shepherded symbolic execution by updating the path constraint until it reaches the failure, or until symbolic execution stalls because the underlying constraint solver cannot handle the complex constraints that have been gathered. During shepherded symbolic execution, ER invokes a constraint solver every time the program accesses symbolic memory (e.g., (Read(Write($V, x, 1$), c) in our example) to determine the set of concrete memory locations that may be accessed. This approach prevents the symbolic execution engine from assuming that a write operation may write to any address, which would complicate the path constraints and lead to other scalability issues in symbolic execution.

If shepherded symbolic execution reaches the failure that occurred in production (i.e., at the end of the trace), ER invokes a constraint solver to determine concrete program inputs that would lead to the failure. If constraint solving can determine a satisfying assignment to the path constraint, the failure is reproduced and a full test case is generated for the developer. The generated test case may not include the same inputs that caused the production failure, but is guaranteed to lead the program along the same recorded control flow and reproduce the same failure. As we show in our evaluation, in a few cases (2 out of 13), ER can reproduce the failure in the first attempt.

However, for most cases (11/13 in our evaluation), shepherded symbolic execution is not able to reproduce the failure in the first attempt. In these cases, ER identifies a solver timeout (§2.4) and constructs a constraint graph (explained below) that depicts the dependencies among data values and constraints. ER passes the constraint graph to key data value selection (§2.3.3).

Constraint Graph Construction When there is a solver timeout, ER constructs a constraint graph, which succinctly describes the dependencies between values and constraints that are established as the program is symbolically executed. This graph representation is inspired by the internal data structures of the STP constraint solver [109]. The nodes in the graph represent operations (arithmetic, logic, memory), constants, program inputs (which are symbolic and therefore unknown), objects in the symbolic memory model (e.g., an array allocated in the program), and symbolic memory addresses. The edges represent the dependencies among the nodes and point from a node to its input dependencies. The goal of a constraint solver is to identify a concrete assignment for each symbolic input node that simultaneously satisfies all the constraints in the graph.

We use our example from Fig. 2.3 to explain how ER constructs the constraint graph. Recall that shepherded symbolic execution follows the control-flow trace (**2→ 5 → 7 → 10 → 11 (FAILURE)**) recorded by running `foo`, and tries to reproduce the crash (due to `abort`) on Line 11. Fig. 2.4 shows the example constraint graph that is generated right before `foo` aborts. Since ER is executing the program symbolically, `foo`'s arguments are represented as four symbolic input nodes (λ_a, λ_b, λ_c, λ_d). As shown in step ①, the value computed by $x = (a + b)$ on Line 3 is an arithmetic "addition" node, `Add`, that depends on the values of the symbolic inputs λ_a and λ_b.

Using the recorded control-flow trace, ER knows that the branch on Line 4 was taken. Subsequently,

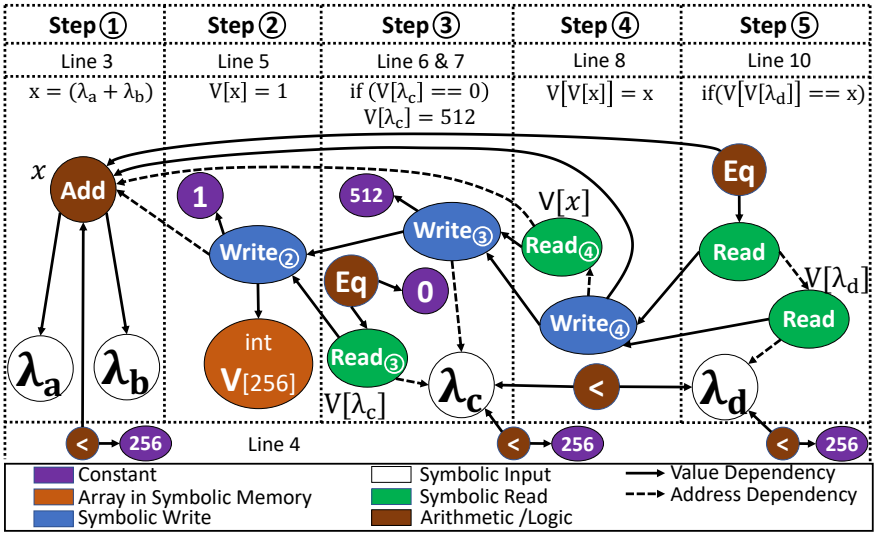


Figure 2.4: Step-by-step construction of the constraint graph for the example in Fig. 2.3.

in step ②, the value `x` is used to index into the array `V` and write 1 to an element (Line 5). The write operation ($\texttt{Write}_{②}$[1]) depends on the target array (`V`), the destination address (`x`), and the value to write (1). Note that the graph representation of the write operation is equivalent to the representation we introduced above, namely Write(`V`, `x`, 1).

In step ③ (Lines 6-7), the branch condition includes a read operation ($\texttt{Read}_{③}$) , which depends on: (i) a symbolic array to read from (which is represented by $\texttt{Write}_{②}$, as this node represents the last state of the array `V` that was written to in step ② and (ii) a source address (λ_c). Note that the graph representation of the read operation is equivalent to the representation we introduced above, namely Read($\texttt{Write}_{②}$, λ_c). The path constraint that ER builds by following the control-flow trace (i.e., the branches that are taken) is represented in the graph using the logic `Eq` nodes. $\texttt{Eq}_{③}$ denotes that $\texttt{Read}_{③}$ was equal to 0. Finally, $\texttt{Write}_{③}$ represents the value 512 being written to the symbolic array `V` (whose final state before being written to is again represented by $\texttt{Write}_{②}$).

Constraint graph generation is similar for steps ④, ⑤.

2.3.3 Key Data Value Selection

Key data value selection eliminates symbolic execution stalls by analyzing the constraint graph and determining the key data values that will simplify the path constraints. We describe the main sources of constraint solving complexity (in §2.3.3.1) and how ER determines the set of key data values that will remove stalls in symbolic execution (in §2.3.3.2).

[1]We use subscripts to distinguish different nodes in different steps.

2.3.3.1 Sources of Constraint Complexity

Recall from §2.3.2 that ER invokes the constraint solver every time a symbolic memory location is accessed to simplify reasoning about subsequent memory accesses. We now describe how constraints on symbolic memory accumulate and provide intuition about how these constraints lead to symbolic execution stalls. We then explain the two key contributors to constraint complexity, namely (1) the length of symbolic write chains, and (2) the size of the accessed symbolic memory.

Examples of complex constraints. Consider the example in Fig. 2.3, and the associated constraint graph in Fig. 2.4. On Line 5, (step ② in the constraint graph), the program writes the value 1 to a symbolic address (`V[x]`), because x is a symbolic value, `x`= $(\lambda_a + \lambda_b)$. Depending on the value of x, the access will either be within the bounds of the array `V` and simplify reasoning, or, outside the bounds of the array and require that ER update other memory objects. Since the branch on Line 4 was taken, the path constraint contains the constraint $\lambda_a + \lambda_b < 256$, and the solver deduces that the array access on Line 5 is within the bounds of `V`, which has 256 elements. ER represents the state of the symbolic memory in step ② as Write$(V, x, 1)$, or $\texttt{Write}_{②}$ for short.

Symbolic execution can stall when encountering chains of symbolic addresses. Consider the statement `V[V[x]]=x` on Line 8 (step ④ in the constraint graph). This statement first reads `V[x]` from the most recent update to `V` (i.e. $\texttt{Write}_{③}$) with offset `x`, where $x = \lambda_a + \lambda_b$. By the logic of the previous paragraph, the solver determines that the offset `x` is always within the bounds of the array `V`. We denote this read operation in step ④ as $\texttt{Read}_{④}$ = Read($\texttt{Write}_{③}, x$), where $\texttt{Write}_{③}$ = Write($\texttt{Write}_{②}, \lambda_c, 512$). Expanding $\texttt{Write}_{②}$ provides $\texttt{Write}_{③}$ = Write(Write($V, x, 1$), $\lambda_c, 512$). Together with the initialization of `V` on Line 1, this chain of writes identifies that `V[x]` is 0, 1 or 512. Thus, it is difficult to determine if the access on line 8 (`V[V[x]]`) is within the bounds of the array `V`; the solver will have to combine constraints gathered from Lines 5 (`V[x] = 1`) and 6 (`if (V[c]==0)`) to determine that `c` cannot be equal to `x` and thus `V[x]` cannot be equal to 512. The memory read and the branch condition on Line 10 (`if(V[V[d]] == x)`) involves similar non-trivial reasoning on the solver's part.

While modern solvers, and hence ER, are able to reason about cases such as Lines 8 and 10, as chains of symbolic memory accesses increases, constraint solving becomes a challenge. In addition, constraint solving is complicated by accesses to large memory objects, since the solver needs to reason about accesses to more memory locations. Therefore, for the purpose of illustration, we assume that the accesses on Line 8 and Line 10 are challenging for solvers to resolve and thus cause symbolic execution to stall.

Key contributors of constraint complexity. An obvious way to avoid all constraint solving complexity would be to record all program inputs, similar to what record/replay engines do. Unfortunately, record/replay can incur high overhead, therefore, ER aims to determine key data values that causes the constraints to become complex and record them. We observe that there are

two key contributors to the complexity of symbolic memory constraints:

1) Length of symbolic write chains. Updates (i.e., writes) to symbolic memory add constraints to the memory state, complicating constraint solving. In our example, the chain of three `Write` operations in Fig. 2.4 can be represented as:

$$\texttt{Write}_{④} = \text{Write}(\text{Write}(\text{Write}(V, x, 1), \lambda_c, 512), \\ \text{Read}((\text{Write}(\text{Write}(V, x, 1), \lambda_c, 512), x), x)$$

This chain bottlenecks shepherded symbolic execution, when the solver gets invoked for the read operations in steps ④ and ⑤ (Lines 8 and 10).

2) Size of the accessed symbolic memory. If the size of the symbolic memory that is accessed is large, the solver needs to reason about a large number of locations that can be accessed, which is complicated. An example is the three `Write` nodes in Fig. 2.4 that modify the 1024-byte array `V` (256×4), which is larger than any other memory object in the graph.

2.3.3.2 Computing the Bottleneck and Recording Sets

ER exhaustively searches in the constraint graph to identify the longest symbolic write chain and the write chain that updates the largest symbolic memory object. (Note that these two chains can be the same.) We call the set of all symbolic values that are read/written by operations in these two chains as the *bottleneck set*; collectively, this set represents key data values that should be recorded to simplify constraints and resolve symbolic execution stalls. In our example, where we assume constraint solving stalls on Line 8 in step ④, there is a single write chain ($\texttt{Write}_{④} \rightarrow \texttt{Write}_{③} \rightarrow \texttt{Write}_{②}$) that updates the symbolic array `V`, which is the largest symbolic memory object. The bottleneck set comprises all the symbolic values that are referred to by these three writes (see the dashed address dependency arrows depicted in Fig. 2.4), namely {x, λ_c, `V[x]`}.

Reducing the Cost of Recording. A naive strategy to simplify constraint solving would be to record all the elements in the bottleneck set the next time the failure occurs. In Fig. 2.4, this strategy would correspond to recording the concrete values for node `x`, λ_c and `V[x]` ($3 \times 4 = 12$ bytes total), which will make it easier for the constraint solver to satisfy all the constraints in the first four steps.

Unfortunately, this approach has high overhead. So, ER reduces the amount of data it records by identifying an alternative set of data values to record from which the bottleneck set can be inferred. ER achieves this with the following algorithm: Initially, ER assigns the recording set of data values, $\mathbf{E} = \{E_0, E_1, \ldots, E_k\}$, to be the bottleneck set. ER assigns a cost (C_i) to recording each element that is equal to the size of the element times the number of times the node is referenced in the recorded control-flow trace, i.e., $C_i = sizeof(E_i) \times Count_{E_i}$. ER's goal is to minimize the total cost of recording, $C = \sum_{i=1}^{k} C_i$.

For each element E_i in **E**, ER performs a depth-first search in the constraint graph to determine if it can record a node (or multiple nodes) with a lower cost than C_i that ER can use to determine the value of E_i. ER continues searching until it can no longer reduce C, the total cost of recording.

Consider the symbolic elements of the bottleneck set {`x`, λ_c, `V[x]`} in Fig. 2.4, where each element has a reference count of 1 since all the statements in `foo` executed once. The algorithm first considers the element `x`, which has a recording cost of 4 (4 bytes of data × referred 1 time). ER continues the depth first search and determines that it cannot record `x` with a lower cost (the cost of recording both λ_a and λ_b—sum of which equals `x`—is 8). Similarly, when ER does a depth-first search starting with λ_c, it does not update the recording set (since λ_c is a leaf node, i.e., an input). Finally, when ER performs a search starting with `V[x]`, it finds out that `V[x]` can be deduced given the values of `x` and λ_c. This happens, because given the values of `x` and λ_c, all the memory writes in steps ②–④ write a concrete value to a concrete address, therefore, $\texttt{Read}_④$ reads a concrete value from a concrete address as well. Consequently, ER removes `V[x]` from the recording set. At this stage of the search, the elements in the recording set do not change anymore and the set of key values ER will record is {`x`, λ_c}.

2.3.3.3 Recording Key Data Values

Based on the final recording set, ER instructs its runtime to record the elements (e.g., values, addresses). The runtime records an element at a point in the execution corresponding to where the element is introduced in the constraint graph. For the recording set {`x`, λ_c} in our example, ER records the value `x` (to concretize $\lambda_a + \lambda_b$), when `x` is first computed (Line 3 in Fig. 2.3, step ① in Fig. 2.4). ER records the input value `c` (to concretize λ_c) when it is introduced in the graph in the expression `if (V[c] == 0)` (Line 6 in Fig. 2.3 and step ③ in Fig. 2.4).

2.3.3.4 Iterative Operation

ER continues the iterative process of recording runtime information, shepherded symbolic execution, and key data value selection until it can reproduce the failure. In our example, a second occurrence of the failure will ship the control-flow trace and values of x and λ_c to shepherded symbolic execution. Symbolic execution will stall when calculating the memory read on Line 10 and key data selection will identify a recording set of {x, λ_c, λ_d}. After a third occurrence of the failure, shepherded symbolic execution is able to reproduce the failure on Line 11.

2.3.4 Handling Concurrency

Traditionally, symbolic execution reasons about different thread interleavings in a program by encoding the thread schedule in the path constraint of an execution [71]. Unfortunately, in order to

explore the potential thread interleavings, a symbolic execution engine has to fork program states after every instruction in the program, exacerbating the path explosion problem. Although this naive method can be improved by identifying data races in the program and only forking program states at racing memory accesses [45, 144, 146], it is still not a tractable approach for large systems [274].

ER instead relies on observations of prior works [140, 254] that suggest that in many cases, it is possible to use a coarse-grained timer to track the execution order of shared memory accesses in a multi-threaded program. Specifically, ER relies on time tracking capabilities of modern hardware (timestamp packets in Intel PT [133]) to record a partial order of instructions across multiple threads. During shepherded symbolic execution, ER executes chunks of instructions from different threads according to the partial order described by the timer packets in the recorded trace. If ER is unable to establish a total order among the chunks of instructions because the associated timestamps overlap, ER arbitrarily selects a sequence of instructions and tries to reconstruct the execution.

If the granularity of the timer packets is fine enough to capture all the data races and synchronization operations, ER reconstructs the execution reliably. For the multi-threaded programs in our evaluation (§2.5), ER is able to reconstruct failing executions. However, if a program exhibits a high-degree of fine-grained racing accesses, ER may not be able to reconstruct the execution. In such cases, ER could use state-space exploration techniques [47, 198] to determine the order of finer grained racing accesses.

2.4 Implementation

Runtime and OS Driver. ER uses hardware-assisted control and data tracing in Intel PT [133] to collect the runtime traces to quantify performance overhead. ER configures the Intel PT Linux driver and records the control-flow, timing information, key data values, in a 64MB ring buffer for each monitored application. The buffer size is decided by the largest trace we collect from the evaluated failures (§2.5.1). Mapping x86_64 control-flow traces into LLVM IR (which is required by KLEE to symbolically execute code) introduces inaccuracies due to optimizations. In our evaluation, we observe that only 91.5% of the control-flow events (branches, calls, returns) in the x86_64 executions can be mapped back to LLVM IR. Meanwhile, recorded key data values can be mapped fully accurately. Our prototype currently makes up for the control-flow inaccuracy by tracing both control-flow and key data values within KLEE.

Even though symbolic execution in KLEE can deal with partially-recovered LLVM traces at the expense of slight path explosion [240], we intend to tackle this problem by either (1) instrumenting the `clang` optimization passes that cause information loss to save metadata about optimizations in order to increase the accuracy of x86_64-to-LLVM mapping, or (2) turning to a binary symbolic execution engine [87].

Recording Key Data Values. We implemented an LLVM compiler pass (156 LoC) to instrument programs with the `ptwrite` instructions [35], which records data values identified by key data value selection into the Intel PT trace. Instrumentation requires redeploying a new version of the application to record additional values, which suits the rapid development cycles of modern software [216] well. Dynamic binary instrumentation [68, 70, 172] presents a potentially less invasive solution for less frequently updated applications.

Shepherded Symbolic Execution. We implemented a prototype shepherded symbolic execution engine on top of KLEE [75] in 13.8K LoC. We modified KLEE to follow a control-flow trace and extended its POSIX environment model based upon the changes from Cloud9 [71]. The extended POSIX environment treats thread-interleavings and system calls (e.g. the content of input files, packets from network sockets, clock information, etc.) as sources of non-determinism. Our prototype treats these non-deterministic values as symbolic (unknown). The shepherded symbolic execution engine detects the reoccurrence of a failure based on matching the program counter and the call stack where the failure occurs.

Detecting Symbolic Execution Stalls Complex constraints prevent symbolic execution from progressing. Some constraints (e.g. floating-point) are not expressible in KLEE; certain classes of constraints are known to be undecidable[60]. For constraints supported and solvable in KLEE, ER determines if shepherded symbolic execution is stalled by identifying solver timeouts. The ideal timeout depends upon the frequency of a failure. For failures that occur frequently, ER should be configured with a relatively short timeout so that shepherded symbolic execution can quickly simplify symbolic constraints using recorded data values; for infrequent failures, ER should use a longer timeout to reduce the number of reoccurrences required to reproduce a failure. For the failures in our evaluation, we found that a 30 second timeout provides a good balance between the time spent on symbolic execution and the number of failure reoccurrences required to reproduce a failure.

Key Data Value Selection. We implemented the key data value selection algorithm in Python (1.3K LoC). When the solver times out, ER passes the path constraint from KLEE to the analyzer. The analyzer generates a list of LLVM IR registers as the key data values to record.

2.5 Evaluation

In this section, we first describe our experimental setup (§2.5.1) as well as our benchmark selection. We then evaluate our prototype of ER by answering the following questions:

Effectiveness and Accuracy (§2.5.2) Can ER reproduce failures in complex, long-running executions? Is ER able to accurately reproduce failing executions? Does key data value selection choose to record useful data values? How useful is data value recording for shepherded symbolic execution?

Table 2.1: Bugs used in the evaluation of ER along with bug IDs and types. † denotes that the program is multithreaded. "LoC" denotes the lines of code of each application. "#Instr(x86_64)" denotes the total number of executed x86_64 instructions in the failing execution."#Occur" denotes the number of failure occurrences ER needs to reproduce the failure. "Symbex Time" represents the total time (sum for all iterations) ER spends on shepherded symbolic execution.

Application-BugID	Bug Type	LoC	#Instr (x86_64)	#Occur	Symbex Time	Performance Benchmark
PHP-2012-2386	Integer overflow [10]	969K	5,460K	6	1.7 min	Benchmark Script [9]
PHP-74194	Heap buffer overflow [18]	1,304K	5,791K	10	111 min	Benchmark Script [9]
SQLite-7be932d	NULL dereference [22]	414K	1,408K	3	3.3 min	Official fuzz test
SQLite-787fa71	Inconsistent data [26]	222K	1,115K	4	61.3 min	Official fuzz test
SQLite-4e8e485	NULL dereference [27]	303K	1,349K	3	21.8 min	Official fuzz test
Nasm-2004-1287	Stack buffer overrun [5]	224K	1,480K	3	12.5 min	Assemble a large asm file
Objdump-2018-6323	Integer overflow [24]	1,078K	324K	3	0.06 min	Disassemble a large binary
Matrixssl-2014-1569	Stack buffer overrun [29]	160K	4,449K	6	6.5 min	Official test
Memcached-2019-11596†	NULL dereference [28]	152K	1,840K	2	3.1 min	memtier_benchmark [37]
Libpng-2004-0597	Buffer overflow [4]	73K	72K	1	0.2 min	resvg-test-suite [38]
Bash-108885	NULL dereference [16]	335K	867K	1	0.3 min	Quicksort in Bash script
Python-2018-1000030†	Shared data corruption [23]	1,021K	36,109K	2	23.5 min	From PyPy benchmarks [31]
Pbzip2†	Use-after-free [14]	13K	6,938K	2	2.6 min	Compress a .tar file

Efficiency (§2.5.3) What is the runtime performance overhead incurred by ER due to online control and data recording? How does ER's runtime overhead compare against record/replay? What is ER's offline computational and memory overhead?

Case Study (§2.5.4) Can ER provide production support for software reliability tools? How does ER's support for these tools differ from existing systems?

2.5.1 Experimental Setup

Target Programs and Bugs. As shown in Table 2.1, we evaluate ER using 13 failures from a broad range of 10 real-world programs including the PHP interpreter; the Python interpreter; SQLite database; memcached, a widely-used distributed object store; the Bash shell; the binary analysis tool objump; NASM, a popular assembler in Linux; libpng, a pervasive image processing library; MatrixSSL, a TLS/SSL implementation, and pbzip2, a parallel compression tool.

Bug IDs and bug types are also shown in Table 2.1. We choose programs and bugs from closely-related work [140, 142, 254] and CVE exploits [33, 34] that were supported by the POSIX environment model of the KLEE symbolic execution engine [75], which ER uses for shepherded symbolic execution. The failures are caused by inconsistent data-structures, integer overflow, buffer overflow, shared data corruption, use-after-free, etc.

Workloads for Performance Evaluation. When available, we use existing benchmarking and testing suites to assess the performance impact of data and control-flow recording by ER. To evaluate the performance overhead of NASM and objdump, we assemble and disassemble the SQLite binary, respectively. To evaluate pbzip's performance, we compress the SQLite codebase which is a 71 MB

.tar file. We ran each performance experiment 10 times and report averages and standard error.

Software and Hardware Configuration. We evaluate ER on two servers with Intel Xeon Silver 4114 CPU and Intel Pentium Silver J5005 CPU. The servers have 187 GB and 8 GB of memory, respectively. The Linux kernel version on both machines is 5.5.2.

Baseline. We compare ER against REPT [254], a state-of-the-art deployed bug reproduction system, and rr [193], a state-of-the-art record-and-replay system. Since REPT is not publicly available, we are only able to compare with the results reported in the REPT paper.

2.5.2 Effectiveness and Accuracy of ER

Length and Complexity of Reproduced Executions. Table 2.1 shows all the failures that ER reproduced along with the number of instructions in the executions that ER reconstructed. As shown in the "#Instr(x86_64)" column, ER was able to reconstruct failures in executions of up to ˜36 million dynamic x86_64 instructions, which is more than two orders of magnitude (361×) longer than the executions that REPT [254] can reproduce (˜100K).

Accuracy of Reproduced Executions. ER accurately reconstructs all data values of a failing execution, given a trace of the control flow and a few key data values recorded in production. Even though the input generated by ER may not be the same input that caused the in-production failure, ER guarantees the generated input will lead to the same control flow as the failure and reproduce the same failure. Thus, ER is able to provide developers a concrete test case (input + control flow) that they can run in a debugger to debug root causes. For example, the SQL queries recovered by ER in three SQLite bugs ([22, 26, 27]) differ from the original input that leads to these bugs in terms of SQL keywords (e.g. `sEleCT` instead of `SELECT`), identifier names (e.g. different table/field names), data values, etc. Despite these differences, the generated input follows the same control-flow because 1) keywords are case-insensitive; 2) renaming identifiers does not change query semantics; 3) recovered data satisfies all control-flow conditions in the trace.

In some instances, developers may be able to debug an issue using an inaccurate execution, such as the one provided by REPT, without relying on a full test case. However, REPT has a substantial number of unknown and inaccurate values as trace length extends beyond 100K instructions. To make matters worse, a developer cannot know which values are inaccurate, since they lack the ground truth. To demonstrate when this may be an issue, we investigated the MatrixSSL bug (for which thorough developer patches and documentation were available) to determine the root cause of the failure. We found the last instruction from the patch that fixes the bug to be executed 3 million instructions prior to the failure. So, REPT would likely not have been able to provide information about the data values of the variables used by the patch and not accurate enough for a developer to debug this issue (we only provide a qualitative comparison, as REPT is closed-source and only

available on Windows).

Key Data Value Selection Effectiveness. To determine the effectiveness of key data value selection, we compared key data value selection with a version of ER that uses a random data recording strategy. The random recording strategy records the same amount of data that ER records, but selects the data randomly among all the data elements in the constraint graph. ER with random data recording only reproduces one failure among the failures that require data value recording (Nasm-2004-1287). In all other cases, ER with random data recording encounters symbolic execution stalls which it is not able to simplify with the random data, and thus cannot complete symbolic execution.

Benefits of Data Value Recording. We show the benefits of data value recording on shepherded symbolic execution. Fig. 2.5 compares the progress of shepherded symbolic execution on the PHP-74194 bug when using a control-flow trace compared to using traces containing control flow and the data values selected during the first and second iterations of key data value selection. We disable the solver timeout and let shepherded symbolic execution execute the same number of instructions in all three cases. Our results show that shepherded symbolic execution with no data values, the first iteration data values and the second iteration data values take 11468, 5006, and 1800 seconds, respectively, to symbolically execute the same number of instructions. We conclude that shepherded symbolic execution drastically benefits from data values and that ER's iterative approach is effective in current production environments where bugs often reoccur.

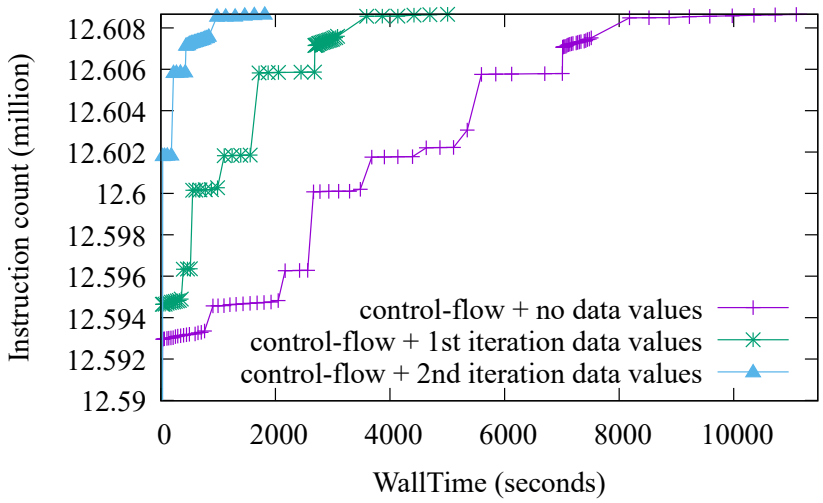


Figure 2.5: Number of instructions executed and the time spent on shepherded symbolic execution for PHP-74194.

2.5.3 Efficiency of ER

Runtime Performance Overhead. We measure the runtime performance overhead of online control flow and data value tracing. For each program, we report the recording overhead of ER for the last occurrence of the failure needed to reproduce the failure. We choose the last occurrence because ER records the most data in the last iteration. Similar to prior work [110], our sensitivity analysis shows no statistical difference in the runtime overhead for buffer sizes of 4KB, 64KB, 1MB, 16MB, 64MB (using a 90% confidence interval).

As shown in Fig. 2.6, ER incurs on average 0.3% (and up to 1.1%) runtime performance overhead across all the programs we evaluate. The overhead numbers for some applications are sufficiently low to be affected by the variability in other factors such as disk I/O (e.g., Libpng's performance benchmarks open and read about 1000 files). These results are in line with prior work that demonstrated that Intel PT is efficient enough to be deployed in production [140, 142, 254]. In our work, we also demonstrate that recording a few key data values does not increase the overhead incurred by Intel PT.

Comparison to Record/Replay. A key question we set out to explore in this paper is whether we can achieve the same level of effectiveness (i.e., failure reproduction ability) as a record replay engine, with better efficiency (i.e., by incurring lower runtime performance overhead).

Fig. 2.6 also displays the recording overhead incurred by a state-of-the-art record/replay engine, Mozilla rr. rr imposes an average of 48.0% (and up to 142.2%) runtime performance overhead. The overhead of rr is prohibitive, even for single-threaded applications (objdump, libpng, etc.). We conclude that ER is able to effectively reconstruct all the failures in our evaluation, and incur lower overhead than a state-of-the-art record/replay system.

Offline Memory and Computational Overhead of ER. The maximum amount of memory consumed by shepherded symbolic execution and iterative constraint reduction is 10 GB, which we believe is a reasonable amount given the decreasing trends of memory cost [36, 117]. The largest constraint graph in our evaluation had about 40K nodes, from which computing bottleneck sets and recording sets took at most 15 seconds. The average and the maximum total shepherded symbolic execution time was 19 and 111 minutes, respectively.

2.5.4 Invariant-Based Failure Localization

ER allows software reliability tools across many domains (e.g., security forensics, fuzzing, etc.) to leverage production failures. In this case study, we show how ER provides production-support to MIMIC [291], an invariant-based automated failure localization technique. MIMIC uses Daikon [101] to calculate likely invariants, predicates which are observed during successful executions. When presented with a failure, MIMIC identifies the likely invariants that are violated

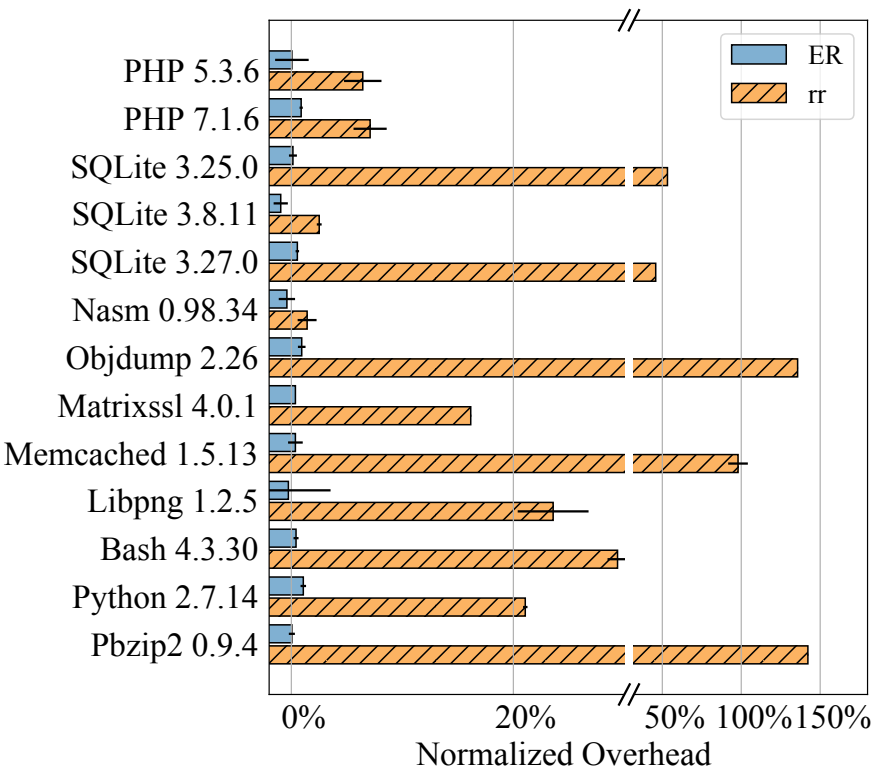


Figure 2.6: Runtime performance overhead incurred by ER's control and data flow recording as well as recording using rr [39]. The error bars display standard error.

by the failure and proposes these as potential root causes. In our case study, MIMIC gathers likely invariants offline, using existing integration and unit tests. When a production failure occurs, ER reconstructs the failure and passes it to MIMIC, which identifies potential root causes.

We test this approach using the coreutil bugs (od [6] and pr [7]) from MIMIC. For both applications, we generate likely invariants using 4 successful executions. We then use ER to reconstruct a failing execution that exhibits the bugs, and pass the reconstructed execution to Daikon. Daikon identifies the same potential root causes (i.e., invariant violations) when using the reproduced execution from ER as it does when using the failing test case directly.

As discussed in Fig. 2.1, existing techniques for reproducing bugs cannot support tools like MIMIC in production scenarios (although they may help with the relatively simple programs, od and pr, discussed above). Full record/replay systems, hybrid record/replay systems, and BugRedux are too expensive for production deployment. Offline approaches (e.g., ESD) and efficient record/replay cannot handle all production bugs and are not guaranteed to reproduce the failure to pass to MIMIC. Finally, REPT does not produce output that MIMIC can execute to produce invariants. If MIMIC could generate invariants from the inaccurate data values recovered by REPT, MIMIC might identify root causes that are both false positives (i.e. inaccurate values violate a likely invariant that should be

satisfied) and false negatives (i.e. inaccurate values satisfy a likely invariant that should be violated).

2.6 Related Work

Reverse execution of failed executions. REPT [254] is the state-of-the-art approach to recover the data flow of a program given the control flow. We compare our work with REPT in §2.2 and subsequent sections, in particular, we discuss its limitations in terms of effectiveness and accuracy. Similarly, RETracer [91] reverse-executes a program to determine the root causes of memory corruption. RETracer is efficient in that it only uses a core dump, without data or control-flow information, however its scope is limited to reasoning about memory corruption bugs within a stack trace.

Record-replay. iDNA [62], PinPlay [199], ReVirt [98], rr [193] and many others [47, 119–121, 169, 170, 180, 185, 203, 211, 244] record at run-time all non-deterministic events (i.e., system calls, thread scheduling, etc.) that impact program execution. They allow users to replay and analyze the execution afterwards. Although useful during development, they are not commonly used in production due to high overheads, e.g., rr has an overhead that can range from 49% to 685% [193]. Other approaches [189] propose specialized hardware for record-replay, but adoption is slow and difficult.

Schedule record-replay. Several approaches specifically target the challenges posed by multi-threaded applications [106], which depend not just on program input but also on the thread interleavings. For instance, H3 [122] records the control flow of programs (with overheads between 1.4%-14.7%) and uses this information to determine the thread interleavings that lead to concurrency bugs. Notably, PRES [198] and HOLMES [86] record select run-time information, such as the total order of synchronization calls, system calls, or function call invocations, to constrain the interleaving space exploration during post-failure analysis. Unlike ER, these tools assume that the failure-inducing input is already known by employing uni-processor record-replay techniques, hence these approaches are orthogonal to ours.

Symbolic execution. Symbolic execution techniques [152] reason about programs symbolically using SMT solvers (e.g., Z3 [94]) and are often used to explore the input space of programs and conduct path-sensitive analysis [75, 87, 112, 192]. To reproduce a particular failure, state-of-the-art approaches [81, 137, 274] take a stack trace where the program crashed, and steer symbolic execution towards the failure location, which traditionally leads to path explosion. Symbolic backward execution [97, 176] is a variant technique that tries to reason about programs from a source code location (e.g., an assertion) backwards to the initial program state. Such approaches are not able to address the constraint complexity challenges like ER. Concolic testing approaches augment symbolic execution with concrete executions to automatically generate test cases [112, 177, 219].

Although concolic testing can simplify the path constraints by recording concrete values from random testing, it is generally designed for path exploration, and does not focus on a specific failure trace like ER.

Failure analysis and diagnosis. A large body of work has developed techniques for failure diagnosis and root cause analysis [44, 51, 136, 138, 140, 142, 168, 213, 242, 251, 276, 284], which can complement ER, since failure reproduction can assist failure diagnosis. Other work [268] has explored different approaches to analyze core dumps of failed executions and extract useful information. Pensieve [283] reproduces the set of events that are relevant to a failure in a distributed system. ER can be used in conjunction with Pensieve to thoroughly reason about failures in a single node.

2.7 Chapter Summary

In this chapter, we presented Execution Reconstruction (ER), a technique to reproduce production failures. ER strives for a sweet spot among efficiency, effectiveness and accuracy. ER uses shepherded symbolic execution to leverage dynamic control flow to eschew path explosion. In addition, to avoid solver stalls, ER uses key data value selection, which analyzes the constraints generated by shepherded symbolic execution to identify a set of data values that can accelerate symbolic execution. Using these techniques, ER will eventually generate concrete test cases that reproduce complex failures. We have implemented an end-to-end prototype to demonstrate the effectiveness of our approach based on Intel PT and KLEE. Real world applications show that ER is able to reproduce failures with similar efficiency to state-of-the-art efficient failure reproduction tools (on average 0.3%) while providing similar accuracy/effectiveness as state-of-the-art accurate/effective failure reproduction tools. In addition, we show that ER allows software reliability tools to leverage production failures.

CHAPTER 3

Vidi: Record Replay for Reconfigurable Hardware

3.1 Introduction

With the end of Moore's Law and Dennard Scaling, system builders are increasingly turning to heterogeneous computing elements such as Field Programmable Gate Arrays (FPGAs) to build efficient computer systems. For example, recent proposals offload computation to FPGAs to improve application performance for machine learning [150, 167, 223, 252, 277, 282, 285], databases [196, 215, 226, 260], graph processing [55, 82, 248, 289], networking [69, 103, 246], storage [159], remote memory [77, 116] and compression [207, 279]. Cloud vendors have begun providing FPGA instances on their platforms due to the promise that these resources show [46, 49].

FPGAs offer the appeal of rapidly prototyping hardware applications without the costly design/validate/tape-out cycle of application specific integrated circuits (ASICs). In theory, developers of FPGA applications could rapidly fix issues such as correctness bugs, performance bottlenecks, security vulnerabilities, and information leaks in their designs. FPGA developers could then redeploy their improved applications and keep iterating—similar to agile software development—until they are satisfied with the outcome. Industry teams are beginning to adopt such software-like workflows for FPGA application development [103].

However, more work is needed to close the gap between software and hardware development workflows to fully realize the vision of agile hardware development. Software developers have a vast ecosystem of tools, including bug finders (e.g., memory-violation detectors [220], data-race detectors [221]), performance profilers (e.g., perf, Coz [92]), and comprehensive logging infrastructure (e.g., Nanolog [272], log20 [286]). The ecosystem of FPGA debugging tools is comparatively lacking—while the research community has built FPGA development tools [57, 151, 161, 175, 182, 241, 280], studies show that most FPGA developers desire more and better debugging tools [15]. Until more comprehensive tools are created, developers will struggle to fix issues that arise in their designs.

We argue that the capability to record and replay executions of an FPGA application would

constitute a foundational building block that would enable the development of further FPGA tools. Record/replay techniques identify and capture the non-deterministic inputs to an execution. At a later time, a developer can use record/replay techniques to reproduce the non-deterministic inputs and recreate the original execution. Record/replay enables a wide variety of use-cases, including testing [84], debugging [188], performance profiling [54], security auditing [96], and replication [67]. FPGA record/replay records and replays the input signal values to an FPGA program that affect the semantics of the output signal values. Using FPGA record/replay, developers could build FPGA development tools that improve reliability through better testing/debugging support, optimize performance through better profiling, and improve security through forensics.

Unfortunately, existing FPGA record/replay systems present unfortunate trade-offs in either efficiency or effectiveness since they are at extreme ends of the record/replay design-space. Most existing approaches employ cycle-accurate record/replay, which is fundamentally inefficient and degrades the usefulness of record/replay. Such systems guarantee that a replay execution produces the same output in the same cycle as the recorded execution by recording and replaying a trace of all input signals at every clock-cycle to the circuit. Some cycle-accurate tools target hardware deployments (e.g., ILA [17], SignalTap [25], and Panopticon [105, 225]), which enables developers to recreate production executions but only for short periods of execution due to the high storage demands of recording cycle-accurate information (see §3.6 for further discussion). Other tools operate in simulation (e.g., VCS [233], Vivado [40]), which enables cycle-accurate recreation of long executions by using software to model hardware behavior, but limits the executions that a developer can record/replay because many executions (some of which are buggy) cannot be observed by simulation due to inaccurate modelling of hardware behavior (see §3.5.2 and §3.5.3).

Other record/replay approaches (e.g. DebugGovernor [182]) employ order-less record/replay, which is fundamentally ineffective at recreating the recorded behavior of a circuit during replay. Such systems capture and can recreate the data sent on each input communication channel of a circuit, but not the ordering of data sent across communication channels. As a result, these systems impose little performance overhead, but cannot support applications whose behavior depends upon the ordering of inputs sent on different input channels. Unfortunately, many applications, including all of those used in our evaluation (§3.5.1), depend upon such orderings and cannot be supported by order-less record/replay tools.

Finally, it is difficult to use software record/replay tools for FPGA record/replay, because CPU-side tools cannot observe hardware events that are only visible to the FPGA.

In this paper, we present, VIDI, which strikes a better balance in the design space of FPGA record/replay to offer both efficiency (deployability on hardware) and effectiveness (support for many FPGA applications). VIDI uses the observation that nearly all FPGA applications communicate using well-defined *transactions* over communication channels, hiding cycle-specific information from

applications to simplify development (§3.2). VIDI uses this observation to relax the granularity at which it captures application behavior, and thus improve efficiency without sacrificing effectiveness.

Specifically, VIDI enforces *transaction determinism*, a novel property that ensures that transaction content and the ordering between transactions are the same across a recorded execution and its replay. Transaction determinism supports arbitrary transaction ordering requirements, which are required by either communication protocols or application semantics. Our results indicate that even though it does not guarantee equivalence of FPGA applications' internal states between record and replay, transaction determinism is sufficient and effective for many use-cases across many hardware designs. In rare cases, transaction determinism can yield output divergences across a recorded execution and its replay; one of VIDI's contributions is a method and mechanisms to automatically detect divergences and techniques that can remove them (§3.3.6).

To ensure transaction determinism, VIDI employs a novel mechanism called *coarse-grained input recording*, which utilizes the abstraction of transactions to identify the signals in a communication channel that effect an FPGA application's semantics. In particular, coarse-grained input recording captures signal values associated with the start/end events and the content of a transaction, which is more efficient than recording/replaying signals at every clock cycle and more effective than completely forgoing transaction ordering.

We implemented and deployed VIDI on Amazon EC2 F1[1]. VIDI supports both F1's simulation framework and its hardware. We evaluated VIDI using one debugging use case, one testing use case, and 10 other FPGA applications and show that transaction determinism and coarse-grained input recording accurately record/replay the FPGA applications and impose low runtime (avg. 1.98% slowdown) and resource overheads (avg. 5.68% LUT, 3.87% registers, 6.92% BRAM of the whole FPGA), making VIDI practical for real-world FPGA deployments. Our evaluation also demonstrates that coarse-grained input recording achieves a median trace size reduction of 1092x when compared to a cycle-accurate approach.

While our prototype targets end-to-end FPGA applications on F1, VIDI's design supports record/replay of individual FPGA components (e.g., DDR4 or app-internal traffic) with little effort (§3.4.1). Moreover, the observations underpinning transaction determinism and coarse-grained input recording apply to other FPGA ecosystems (e.g., Intel FPGAs, RISC-V designs), so we believe that the VIDI design will apply to other use-cases beyond Cloud FPGA offloading (see §3.2).

Overall, VIDI makes the following contributions:

- VIDI determines opportunities for relaxing the granularity and cycle-accurate ordering requirements used in existing record/replay approaches.
- VIDI is the first record/replay system for real-world FPGA applications that run on hardware. VIDI leverages the recording relaxation opportunities identified above by enforcing transaction

[1]Available at https://github.com/efeslab/aws-fpga

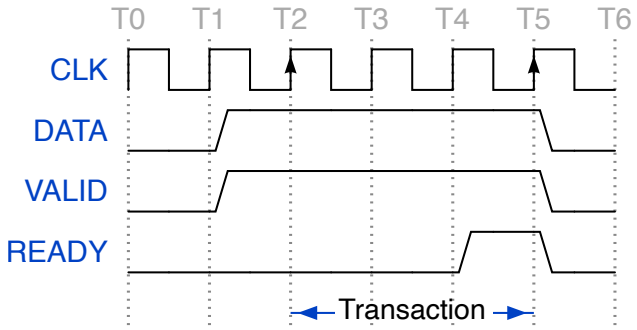


Figure 3.1: An example waveform of a VALID/READY handshake on an AXI Channel. Each line represents the values of a different signal in the communication channel.

determinism, a novel property for FPGA record/replay, through coarse-grained input recording, a novel technique.

- An evaluation of VIDI shows that it works for real-world FPGAs on a real Cloud deployment with low performance and resource overhead.

3.2 Background and Observations

In this section, we provide background regarding FPGAs and the key observations about common FPGA communication primitives that inform VIDI's design (§3.3).

FPGAs often send and receive data from components in a heterogeneous system (e.g., CPUs, NICs) using *communication channels* comprised of multiple shared signals (i.e., variables). In this paper, we use the term *channel* to refer to a unidirectional communication channel that uses *handshaking* techniques on the shared signals to coordinate message transmission between a single sender and a single receiver. We focus on FPGA deployments in which the sender and receiver of each channel share a clock, since these deployments are commonly provided by current cloud computing platforms (e.g., the Amazon EC2 F1 platform).

3.2.1 Channels and Transactions

Fig. 3.1 shows an example waveform diagram of an instance of *VALID/READY* handshaking in AXI [50], the de facto communication protocol used in Xilinx FPGAs. This diagram shows the values (i.e., low or high) of the shared signals in the communication channel over time. CLK is the clock signal, which oscillates from high to low repetitively, VALID and READY are AXI control signals that the endpoints use to coordinate communication, and DATA is a single-bit data value that is transmitted from the sender to the receiver. The sender takes READY as an input signal and

sets VALID, DATA as output signals, while the receiver takes VALID, DATA as input signals and sets READY as an output signal.

In this example, the sender initiates a handshake at T2 by setting DATA to the desired value and assigning the VALID signal to be high before T2. The receiver observes that VALID is set to high and waits until it is ready to receive the data. In this example, the receiver is ready at T5, so it sets READY to high between T4 and T5. At T5, the sender and receiver observe that both VALID and READY are high, indicating that the handshake is complete and DATA is transmitted. The receiver must use or store the value of DATA before T6, as DATA is only valid when the VALID signal is high.

A *transaction* is the transmission of DATA via a handshaking process; the rest of the paper uses the terms transaction and handshaking interchangeably. Transactions have clearly defined start and end events (e.g., the transaction in Fig. 3.1 starts at T2 and ends at T5). For correct operation, handshaking protocols specify that VALID and DATA signals must be constant throughout the duration of the transaction and that the receiver must not use DATA until the transaction is complete.

We make the following observation:

Observation #1: FPGA applications typically use communication protocols that employ handshakes and transactions [50, 73, 132, 195, 227]. Transactions identify the start and end events between which signals in a communication channel are unmodified and meaningful.

3.2.2 Transaction Ordering in FPGAs

Transactions hide cycle-specific information from an FPGA application. Nevertheless, FPGA applications and protocols often depend on multiple channels grouped by semantics (e.g., a group of read/write, data/address channels). Their correctness relies on specific ordering requirements across transactions on multiple channels. Violation of transaction ordering requirements can cause incorrect results, deadlock, or one communication party to enter an unrecoverable error-state [50, 267].

In particular, the correct operation of an FPGA application depends on the happens-before relationships of the start and end events of each transaction. We define a happens-before relationship between events as follows: a transaction event A (either start or end) *happens before* a transaction event B if A happened at an earlier time, based on wallclock, than B.

For example, consider the AXI communication protocol [50]. Fig. 3.2 shows an example using the protocol, where an FPGA (manager) sends one memory write operation to a CPU (subordinate). The FPGA communicates the memory address and the data to be written to the CPU in the top and middle channels, respectively. The CPU communicates the write acknowledgement to the FPGA in the bottom channel.

The AXI protocol requires that the end events of the address and data (in the top and the middle

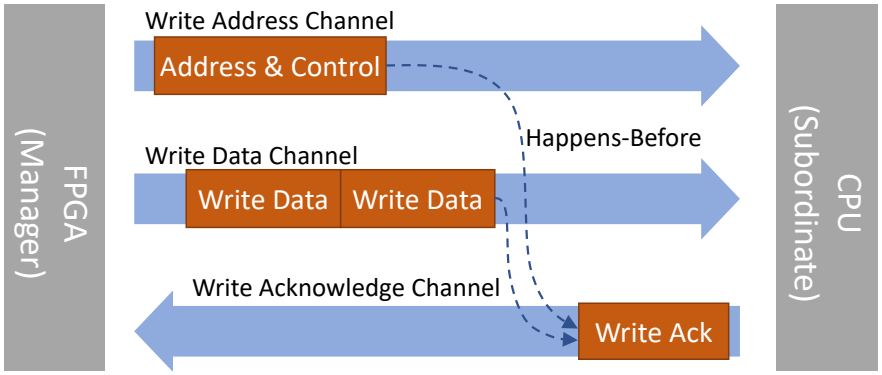


Figure 3.2: Example group of FPGA-CPU communication channels. The displayed roles of the FPGA and CPU in this example can be reversed. The orderings between transactions (the dashed arrows between orange boxes in each channel) are important for correct operation and hence a record/replay system must capture these orders.

channel) transactions must *happen before* the start event of the corresponding acknowledgement (in the bottom channel) transaction. However, the protocol does not place ordering requirements on transactions at the finest possible (i.e., individual clock cycle) granularity.

We study existing communication protocols[50, 227] and applications to determine the ordering requirements upon which FPGA applications typically depend. We then observe that an FPGA application's behavior typically depends on the ordering of transaction end events with respect to all other transaction events (start and end), but rarely depends on the ordering of transaction start events with respect to all other transaction events (start and end). The observation is based on transaction semantics; transactions dictate that a receiver refrain from accessing and using data until the transaction end event. For example, memory consistency models dictate that memory operations *complete* in a certain order—i.e., that the end event of one write acknowledgment transaction must happen before the end event of another write acknowledgment transaction. In sum:

> **Observation #2:** FPGA applications often depend on specific orderings among their transaction start/end events, but rarely depend on the specific cycle in which a transaction starts or ends. Furthermore, FPGA applications often depend on the ordering of transaction end events with respect to all other transaction start/end events due to transaction semantics.

3.3 Design

VIDI exploits observations #1 and #2 from the previous section to relax the granularity and cycle-accurate record/replay for efficiency while preserving the necessary ordering of transaction events to ensure effectiveness (i.e., ability to record/replay).

Based on observation #2, VIDI introduces a novel property, *transaction determinism*, which preserves the happens-before relationship between transaction end events and other transaction start/end events across a recorded execution and its replay. Transaction determinism guarantees that FPGA applications will produce the same output if their executions depend only on the content and partial ordering of transactions at the I/O boundary. Enforcing transaction determinism is generally sufficient for successful record/replay. In rare cases, this relaxation causes VIDI's replay to diverge from the original execution if program behavior depends on the exact clock cycle when a signal changes (e.g., the bug is only triggered when input signal X is 1 at cycle 1). In §3.3.6, we discuss how VIDI detects and fixes such divergences and provide a real-world example. Thus, VIDI makes a tradeoff in enforcing transaction determinism: it favors practical utility over stronger guarantees which would come with higher overhead (see §3.6).

To ensure transaction determinism, VIDI introduces a novel mechanism, *coarse-grained input recording*, based on observation #1. Rather than recording input signal values in a channel at all clock cycles, VIDI records the time when a transaction starts/ends, and the transaction content at the start (e.g., in Fig. 3.1, T2, T5, and DATA at T2), which reduces storage and performance overhead compared to existing work (see §3.5.5). Note that VIDI assumes a proper application to at least implement the single-channel handshaking (§3.2.1) correctly, thus can not handle applications that do not use handshaking or do not implement it correctly. VIDI enforces transaction determinism by ensuring that the orderings of all transaction end events and other transaction events (start and end) are consistent with the orderings observed during recording. The ubiquity of the transaction abstraction suggests that coarse-grained input recording is generalizable across FPGA platforms.

Fig. 3.3 shows VIDI's design. VIDI intercepts all transaction-based communication channels across a user-defined record/replay boundary between an FPGA program and the external environment with which it interacts. Our implementation (§3.4) treats the CPU as the external environment and the entire FPGA application as the FPGA program, and hence records/replays transactions that occur on all input channels (e.g., Channel 1) and all output channels (e.g., Channel 2). During recording, VIDI performs coarse-grained input recording and stores input signals from the environment in a storage resource accessible to the FPGA (either internal or external). During replay, VIDI redeploys the FPGA program, either in hardware or simulation, and replays the input signals in each channel (see §3.2.1). While VIDI only records/replays transactions at the boundary with the external environment, the system recreates internal execution states of the FPGA program (e.g., computation logic and internal traffic among different modules) during replay.

Recording. When recording is enabled, VIDI uses a channel monitor (§3.3.1) on each channel to transparently observe potentially concurrent transactions (❶). Channel monitors deployed on input channels perform coarse-grained input recording, i.e., they capture the start/end events and the content of each transaction. By default, channel monitors deployed on output channels only track

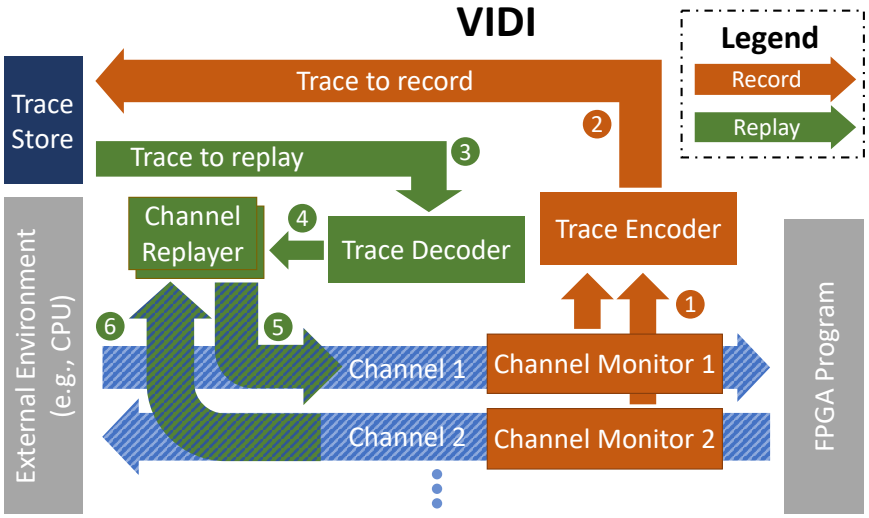


Figure 3.3: VIDI's design overview. For simplicity, we show a single input channel (Channel 1) and output channel (Channel 2). The green arrow (❺) from the channel replayer to Channel 1 indicates that, during replay, the channel replayer creates input transactions on input channels. The green arrow (❻) from Channel 2 to the channel replayer indicates that, during replay, the channel replayer receives output transactions on output channels.

transaction end events. Together, the channel monitors produce data suitable for identifying the happens-before relationships between transaction end events and either transaction start events of input transactions or transaction end events of output transactions. These relationships are exactly those that are required for transaction determinism.

The channel monitors then send this information to the trace encoder (❷). The trace encoder generates a compacted trace containing the content of input transactions and happens-before relationships between input and output transaction events (§3.3.2). The happens-before relationships identified by the trace encoder are required during replay to ensure transaction determinism. Finally, the trace encoder forwards the trace of events to a Trace Store, which saves the trace into auxiliary storage (§3.3.3).

Replaying. When replaying is enabled, the trace store forwards a previously-recorded trace to the trace decoder (❸). The trace decoder reverses the work of the trace encoder by decompressing the trace to identify transaction content and happens-before relationships across transaction events (§3.3.4). The trace decoder creates a separate trace for each channel which it forwards to the channel's replayer (❹). Each channel replayer communicates with the FPGA application over its assigned channel (§3.3.5): input channel replayers (i.e., channel replayers that are senders) control when each input transaction starts (e.g. the VALID signal in Fig. 3.1) and its content (e.g. the DATA signal in Fig. 3.1) ❺, while output channel replayers (i.e., channel replayers that are receivers) control when each output transaction ends (e.g. the READY signal in Fig. 3.1) ❻. All channel

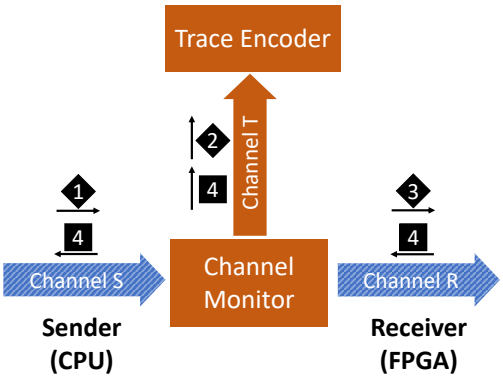


Figure 3.4: A channel monitor deployed on an input channel. Black arrows, boxes, and diamonds identify the steps needed to transparently perform coarse-grained input recording between the sender and receiver.

replayers coordinate using vector clocks [162] to ensure transaction determinism. Although VIDI is designed to support replay on hardware, it can be run in simulation to replay traces collected from hardware executions (see §3.5.2).

Divergences. If the FPGA application's behavior is cycle-dependent, transaction determinism may be insufficient for deterministically reproducing an execution's output content (we observe about one divergence every one million transactions in 1/10 applications in our evaluation in §3.5.4). In §3.3.6, VIDI provides a two-step mechanism for detecting such divergences. Based on VIDI's divergence report, the developer can resolve these divergences by eliminating cycle-specific dependencies from their program. VIDI provides a reusable solution for the only source of divergence we observe (i.e., a communication construct that uses polling).

We provide a concrete example of VIDI's workflow in §3.5.2. Below, we describe how each VIDI component works.

3.3.1 Channel Monitor

VIDI deploys a channel monitor for each channel used by the FPGA program. The monitor transparently intercepts the transactions on a channel. Monitors on input channels (i.e., channels in which the FPGA is a receiver) perform coarse-grained input recording; they send messages identifying the start, end, and content of each transaction to the trace encoder. By default, channel monitors on output channels (i.e., channels in which the FPGA is a sender) send messages identifying the end of each transaction to the trace encoder. When using VIDI to validate output (§3.3.6), the system configures output channel monitors to track the content of each completed transaction, in addition to the transaction end event. Fig. 3.4 illustrates a channel monitor on an input channel that intercepts a transaction between a sender (CPU) and a receiver (FPGA program).

When the sender begins a transaction on an input channel ❶, the monitor sends data to the trace encoder ❷, which will log both the start event and content of the transaction. The channel monitor uses transactions to communicate with the trace encoder, since the trace encoder depends upon downstream resources (e.g., the trace store) that may not be ready to receive more events. After the data is safely stored on the trace encoder, the channel monitor starts a transaction with the receiver ❸. A channel monitor on an output channel elides this work on transaction start, since the start time and content of output transactions are not needed when enforcing transaction determinism (§3.3.5).

Managing the end of the transaction is complex for channel monitors, regardless of whether the monitor is deployed on an input or output channel. A channel monitor must ensure that it completes three transactions (denoted by 4 in Fig. 3.4) simultaneously: the transactions to the sender and receiver (so that there is a clearly defined ending to the original transaction between the sender and the receiver) and a new transaction to the trace encoder to log the end event (so that the encoded trace correctly identifies when the transaction completes). The simultaneous completion of these three transactions cannot always be completed without additional machinery, since the trace encoder may need to block due to a full downstream buffer (e.g. in the trace store).

To create a single-cycle transaction with the trace encoder, the channel monitor makes an eager reservation with the trace encoder before starting the transaction ❷ with the receiver. This reservation pre-allocates a buffer in the trace encoder and ensures that the trace encoder can instantaneously accept the end event from the channel monitor at a later clock cycle.

The channel monitor uses a single fixed-sized channel packet (see the left-hand-side of Fig. 3.5) to send transaction start/end events and content to the Trace Encoder. A channel packet consists of three elements: `Start`, a boolean field indicating that a new handshake started on the channel in the current clock cycle; `Content`, binary data sent by the transaction; and `End`, a boolean field representing that a handshake completed on the channel in the current clock cycle. VIDI uses a special channel packet format instead of recording physical timestamps (i.e. cycle counters) because physical-timestamp-based approaches either limit record/replay to short traces or make record/replay prohibitively expensive, as observed by existing cycle-accurate tools (see §3.6).

3.3.2 Trace Encoder

The trace encoder consumes traces from the channel monitors, encodes the happens-before relationships between the start/end event of each transaction and the end events of all other transactions (input and output), and produces a compact trace for efficient storage. The resulting trace efficiently encodes a vector clock for each transaction event, which can be used to enforce happens-before relationships during replay (§3.3.5) and provide transaction determinism.

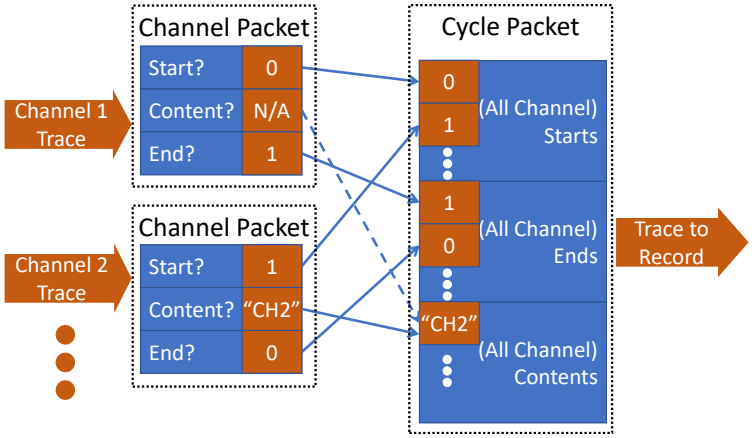


Figure 3.5: Architecture of VIDI's Trace Encoder.

At each clock cycle, the trace encoder adds the content from all channel packets into a *cycle packet* (Fig. 3.5). The cycle packet contains two fixed-size bit-vectors, `Starts` and `Ends`, and one variable-sized field, `Contents`. `Starts` identifies whether each input channel started a handshake during the cycle (i.e., if the n^{th} field is set in `Starts`, then the n^{th} input channel started a handshake), while `Ends` identifies whether each input *or* output channel completed a handshake during the cycle. Including input and output transaction end events in `Ends` is critical to enforce transaction determinism. Finally, the trace encoder constructs the `Contents` field in the cycle packet using a binary-tree structure to compact the `Content` fields from all channel packets. The compact format only includes the `Content` of channel packets on input channels that indicate the start of a transaction.

3.3.3 Trace Store

The trace store performs two actions, depending on whether VIDI is configured to perform recording or replaying. When VIDI is recording, the trace store stores cycle packets that are generated by the trace encoder to storage resources (e.g., CPU-side DRAM). When VIDI is replaying, it retrieves cycle packets that are consumed by the trace decoder from storage resources. To improve resource utilization, the trace store converts variable-sized cycle packets into the fixed-size storage interface packets available to FPGA applications (e.g., the AWS F1 platform exposes CPU-side DRAM to FPGA programmers using 64-byte granular read/write operations via the AXI protocol). The trace store packs multiple cycle packets into a single read or write operation to the storage resources when possible (e.g., placing 48-byte and 16-byte packets into the same 64-byte cacheline). Additionally, the trace store collaborates with other entities in a heterogeneous system (e.g., the operating system kernel on the CPU side) to manage storage allocation, buffer management, etc.

External storage resources may operate slower than the execution trace is generated or consumed on FPGA, so the trace store also manages a back-pressure signal to pause the recording/replaying when necessary. Since VIDI uses handshaking and transactions, it can easily pause the recording/replaying without disrupting any necessary happens-before relationships. Such design enables VIDI to support arbitrarily long execution traces.

3.3.4 Trace Decoder

During replay, the trace decoder reconstructs the content of each input transaction and provides channel replayers with the information necessary to ensure transaction determinism. This process involves decomposing the trace from the trace store into channel-specific traces.

The trace decoder receives a sequence of cycle packets from the trace store (i.e., the same format as the output of the trace encoder in Fig. 3.5). The trace decoder then decomposes the fields of each cycle packet (i.e., `Starts`, `Ends`, and `Contents`) into individual channel packets (see §3.3.1). For each input channel, the trace decoder generates the `Start` and `End` fields of the channel packet by identifying the corresponding element in the `Starts` and `Ends` bit-vectors in the cycle packet and decompresses `Contents` (via the binary tree used in §3.3.2) to generate the `Content` field corresponding to each `Start` field. For each output channel, the trace decoder only generates the `End` field of the channel packet.

In addition to the channel packets, the trace decoder also sends each channel replayer the `Ends` field from each cycle packet. The `Ends` field is critical for reconstructing the vector clock to identify the happens-before relations among transaction events and ensure transaction determinism (§3.3.5).

Fig. 3.6 is an example of decoding a cycle packet from the trace encoded in Fig. 3.5. The decoder creates a channel packet for the first input channel by inspecting the first element in `Starts` (i.e. 0) and the first element in `Ends` (i.e. 1). Since this is not a start packet, there is no content field (i.e., `Content` is N/A). The decoder then creates the channel packet for the second input channel by inspecting the second elements in `Starts` and `Ends`. Since the resulting channel packet contains the first start event in the cycle packet in the example, the decoder assigns the first element of the `Contents` field to the channel packet for the second input channel.

3.3.5 Channel Replayer

VIDI deploys a channel replayer for each input and output channel used by the FPGA program. Together, the channel replayers provide transaction determinism by reproducing the content of each input transaction and ensuring that each recreated transaction event satisfies the recorded happens-before relationships with all other transactions.

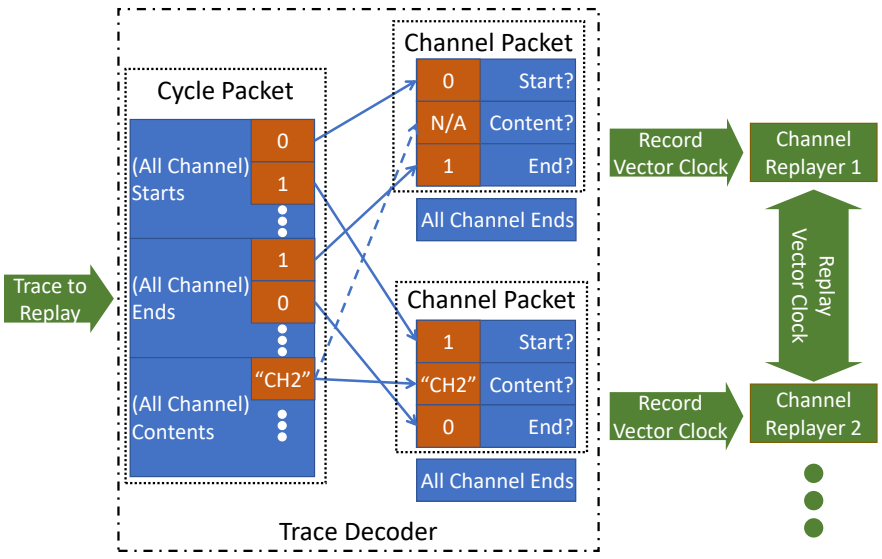


Figure 3.6: Architecture of the Trace Decoder and Channel Replayers with an example trace.

Each channel replayer receives a sequence of ⟨channel packet, `Ends`⟩ pairs from the trace decoder. Each replayer recreates the transaction events contained in pairs from the sequence in the correct order as required to enforce transaction determinism. The channel replayers use *vector clocks* [162] for this task. VIDI associates a logical timestamp, $\langle t_1, t_2, ..., t_n \rangle$, with each transaction event (start or end). Each entry, t_i, represents the number of completed transactions in the i^{th} channel. VIDI determines and enforces the happens-before relationships of two events by maintaining and comparing the partial order ($\geq$) of their logical timestamps[2].

Each channel replayer maintains two vector clocks during the replay execution: (1) T_{expected}, which tracks the expected logical timestamp of the next event, and (2) T_{current}, which tracks the current progress of the replay. The replayers initialize T_{current} and T_{expected} to contain 0 in each field. Before processing each transaction event from the sequence of pairs, the channel replayers ensure that $T_{\text{current}} \geq T_{\text{expected}}$.

Each replayer maintains T_{expected} using `Ends`. Specifically, after processing an element in its sequence, each channel replayer advances T_{expected}'s i^{th} element by 1 if `Ends` indicates that a new transaction is expected to finish in the i^{th} channel. The updated T_{expected} indicates the logical timestamp at which the required happens-before relationship of the next channel packet is satisfied. Each replayer maintains T_{current} by communicating with the other channel replayers. Specifically, when a transaction completes on the i^{th} channel, the channel replayer for that channel sends a message to all channel replayers; each channel replayer increments the i^{th} element in their T_{current} by 1 after receiving the message.

[2]For two timestamps T_1 and T_2, $T_1 \geq T_2$ if and only if the i^{th} element of T_1 is greater than or equal to the i^{th} element of T_2 for all i.

After $T_{\text{current}} \geq T_{\text{expected}}$ is satisfied at a given channel replayer, the replayer processes any events contained in the channel packet. If the channel packet refers to the `Start` of a transaction, the input channel replayer starts a transaction using the `Content` field of the channel packet. If the channel packet refers to the `End` of a transaction, the output channel replayer attempts to end a transaction (e.g. by setting the READY signal to high).

3.3.6 Handling Replay Divergence

In rare cases transaction determinism fails to deterministically record and replay an execution because the FPGA program has cycle-dependent behavior (about one in one million transactions differs between recording and replaying in only 1/10 of our evaluated applications §3.5.4).

VIDI follows a two-step process to identify divergences. First, VIDI records a *reference trace* by configuring output channel monitors to record the content of output transactions in addition to the normal recording workflow (§3.3.1). Second, VIDI replays the *reference trace* while simultaneously recording the replayed transactions as a *validation trace*. VIDI compares the *reference trace* and the *validation trace* to identify divergences between record and replay.

Developers must convert the cycle-dependent behavior in their application into cycle-independent logic to resolve divergences. In our evaluation, we observe that one application, DRAM DMA, has one content divergence about every one million transactions. All the divergences that we observe are caused by the same cycle-dependent logic (polling). Below, we describe how we used VIDI to automatically identify the cycle-dependent behavior in that application and manually convert its cycle-dependent polling in to a cycle-independent implementation that uses interrupts.

DRAM DMA uses polling to determine progress; the CPU polls a value every 500ms to identify whether the FPGA application has finished an acceleration task. Since the task completion depends on real-time behavior, VIDI replays may produce the polling request too early or too late relative to the task completion and change the execution's behavior. VIDI automatically identifies the problem when configured to test for replay divergences (every application in the eval §3.5.1 is configured in this way). It reports transaction content, the output channel, and the context (e.g., which transactions completed on the offending channel before the divergence). Using VIDI's report, we identify the code causing cycle-dependent behavior and create a 10-line patch (out of 4.3K lines in the application) that instead sends a cycle-independent interrupt upon task completion. Other applications that use polling could reuse our approach to eliminate cycle-dependent behavior.

3.4 Implementation

We implemented VIDI on Amazon EC2 F1 as a shim module supporting the same programming interface as AWS F1 instances. Thus, AWS F1 FPGA applications can seamlessly use VIDI. For heterogeneous designs that deploy CPU-side applications, VIDI provides a runtime library that can be used to enable and disable record/replay. VIDI also includes a software component that detects replay divergences (§3.3.6).

3.4.1 Hardware

VIDI's hardware shim module consists of 7318 lines of SystemVerilog using Xilinx Vivado Design Suite 2020.2 with F1 shell_v04261818 and a high-performance 250 MHz clock.

Our prototype uses the boundary between the CPU and FPGA as the record/replay boundary. Specifically, our prototype provides transaction determinism for the transactions issued on all 5 AXI interfaces on AWS F1 (the de facto communication mechanism between CPUs and FPGAs on F1). Replaying the AXI transactions recreates DDR4 traffic (which provide access to on-FPGA memory), so recording DDR4 traffic offers no benefit in our use-cases (see §3.5) and our prototype refrains from recording DDR4 traffic by default. If preferred, a developer can customize VIDI to include or exclude other AXI-like interfaces; we demonstrate such customizations by extending VIDI to record/replay the aforementioned DDR4 interface and application internal buses with only 13 additional lines of code per interface.

VIDI requires that recorded/replayed AXI interfaces use the same clock, which is enforced by the AWS F1's programming interface [41]. We describe our implementation below.

Channel Monitor. A channel monitor transparently interposes on a channel between the CPU and FPGA by coordinating transactions across three channels: one between the CPU and channel monitor, one between the channel monitor and trace encoder, and one between the channel monitor and FPGA (§3.3.1). Ensuring the ordering properties required for a correct channel monitor proved extremely challenging. For example, we found that Debug Governor [182], which aims to support equivalent functionality to a channel monitor, violates the handshaking protocol with the receiver when the trace encoder delays transaction completion despite the developers carefully considering a truth-table of 128 elements during their implementation [179]. Moreover, subtle issues in the protocol can push the FPGA into an unrecoverable error state that is extremely difficult to debug on hardware.

In our implementation, we applied formal verification to ensure the correctness of this critical design component. Specifically, we applied SystemVerilog Assertions (SVA) via JasperGold v2021.06 [76] to formally prove that channel monitors (Fig. 3.4) enforce critical properties (e.g., intercepted transactions handshake correctly and are not reordered nor dropped).

Table 3.1: The applications used to evaluate VIDI. We provide their execution time without VIDI (ET w/o VIDI), average performance overhead and standard deviation of VIDI's recording (Overhead±std), the size of VIDI trace generated during recording (TS), and the reduction of trace size.

App	ET w/o VIDI (s)	Overhead ±std (%)	TS (GB)	Trace Reduction
(1) DMA [41]	1.66	5.93±0.45	0.81	97x
(2) 3D [290]	4.14	0.54±2.88	0.14	1,439x
(3) BNN [290]	6.43	0.63±1.68	0.31	966x
(4) DigitR [290]	9.56	0.03±0.14	0.97	468x
(5) FaceD [290]	17.41	-0.05±1.28	0.12	7,011x
(6) SpamF [290]	1.56	10.54±0.40	0.83	88x
(7) OpFlw [290]	13.79	1.91±0.27	1.33	490x
(8) SSSP [19]	397.83	0.00±0.01	0.002	10,149,896x
(9) SHA [20]	31.75	0.64±0.06	1.23	1,219x
(10) MNet [21]	110.71	0.11±0.27	0.51	10,163x

Trace Store. The trace store uses the PCIe Direct Memory Access (DMA) programming interface, which is also used by the FPGA application, to store and fetch the trace. The prototype uses the AXI-Interconnect Xilinx library to multiplex the PCIe interface between VIDI and the application.

3.4.2 Software

We implemented VIDI's software runtime library in 772 lines of C on Ubuntu 20.04 with kernel 5.11. During recording, the runtime reserves huge-pages for trace buffering, initializes VIDI's shim module before the CPU-side application invokes the FPGA-side application, and saves the recorded trace to disk when the application finishes. During replay, the software runtime copies the trace into huge-pages and initializes VIDI's shim module to replay the trace.

VIDI's offline trace analysis tools consist of 1396 lines of C++. The trace validation tool detects divergences by comparing the content and ordering of the transactions in two traces §3.3.6. VIDI also includes a trace mutation tool that can reorder a trace's transaction events to aid testing §3.5.3.

3.5 Evaluation

In this section, we first describe our experimental setup and benchmark selection. We then evaluate our prototype of VIDI by answering the following questions:
Debugging (§3.5.2). How does VIDI help FPGA debugging?
Testing (§3.5.3). How does VIDI help FPGA testing?
Effectiveness (§3.5.4). How well does the transaction determinism provided by VIDI preserve the same output across recording and replaying?

Efficiency (§3.5.5). What is the performance overhead of ggVIDI? What is the resource overhead of VIDI? How much does VIDI benefit from the transaction abstraction?

3.5.1 Experimental Setup

Benchmark Selection. We first port a buggy Frame FIFO implementation from a recent survey of FPGA bugs [175] to AWS F1 to demonstrate how VIDI assists debugging. Then, we port a buggy component of an open-source AXI communication library [32] to AWS F1 to demonstrate how VIDI can enable new testing techniques.

Finally, we evaluate VIDI on 10 other applications on the AWS F1 platform to demonstrate its efficiency and effectiveness in general. As shown in Table 3.1: (1) DRAM DMA is an example application written by AWS in SystemVerilog that demonstrates many of the features and resources on the F1 platform, including PCIe register access, bidirectional PCIe DMA between CPU and FPGA, etc. The rest of the applications are generated via High Level Synthesis (HLS). (2)-(7) are from the Rosetta FPGA benchmark [290], including graphics rendering applications and machine learning accelerators. (8)-(10) are open-source FPGA applications that accelerate graph processing, hashing and image classification.

Methodology. We integrate VIDI into each application and synthesize the resulting combination to a bitstream that can be loaded to the FPGA on AWS F1. Specifically, we modify the software component of each application to enable and disable VIDI record/replay around the invocation of each FPGA-side application (§3.4.2), requiring less than 15 lines of code for each application. Note, VIDI instruments the hardware design automatically (i.e., without any developer annotations) by placing a shim layer (§3.4.1) between the accelerator and the FPGA shell. We conclude that VIDI is easy to apply to real-world FPGA applications.

To measure the effectiveness and the efficiency of VIDI, we run each application using three different configurations: (R1) disable recording and disable replaying, which makes VIDI transparent to the transactions on all channels; (R2) enable recording and disable replaying for both input channels and output channels; and (R3) enable replaying and enable recording for output channels. Although individual benchmarks use at most 3 interfaces, VIDI is configured to record/replay on all 5 interfaces (25 channels in total), which provides evaluation results of the worst-case scenarios. VIDI is also configured to record additional information (i.e., the content of output transactions) for divergence detection (§3.5.4). In a real-world deployment, developers could restrict recording to the used interfaces/channels or opt out of divergence detection for better performance and lower resource overhead.

3.5.2 Debugging Case Study

We demonstrate the effectiveness of VIDI in a debugging case study based upon a bug presented in a survey of FPGA bugs [175]. Specifically, we build an echo server (i.e., loopback test) on AWS F1 that uses an existing buggy Frame FIFO library. This Frame FIFO groups 32-bit data fragments into frames and enqueues/dequeues data fragments one at a time. The FIFO should block incoming data when the FIFO is full, but mistakenly drops data fragments if the incoming frame size is unaligned with the remaining FIFO capacity.

The FPGA component of the echo server receives PCIe DMA-Write requests from a CPU, converts each 512-bit DMA-Write operation (a frame) into 16 32-bit data fragments, feeds the fragments to the Frame FIFO, and stores the FIFO outputs to on-FPGA DRAM. The CPU component of the application uses two threads, `T1`, which validates the FPGA component by issuing DMA-Writes to the FPGA and checking the FIFO output using DMA-Reads, and `T2`, which modifies a control-register to initiate the FPGA component.

We observe two bugs in the echo server, which cause `T1` to observe inconsistency in the data written to and read-back from the FPGA component. In addition, these bugs escape simulation and only arise when the application is deployed to an FPGA.

As a typical workflow of using VIDI to reliably reproduce the buggy behavior for further diagnosis, we instrument the application with VIDI and trace transactions on all AXI-interfaces, including the DMA-Write, the DMA-Read and the control-register bus. First, we configure VIDI to enable recording. VIDI records the number of DMA transactions that completed before/after the control-register update transaction, as well as the contents of these transactions. Once `T1` observes inconsistency in the DMA data, it saves the corresponding trace for replay. Second, we enable VIDI to replay the buggy trace. VIDI provides the exact same DMA transactions (i.e., the same number with the same content) to the FPGA component before it replays the control-register transaction, which triggers the same bug from the recording. We confirm that the recorded bug was reproduced by checking the data inconsistency pattern observed by `T1`. In the end, we can replay the buggy trace as many times as necessary to investigate the buggy behavior with third-party diagnosis tools.

Below, we describe the two bugs and how VIDI could help debug them:

Unaligned DMA access. Unaligned addresses cause the FPGA's DMA engine to use bitmasks indicating that certain bytes are "invalid". The FPGA component of the echo server does not handle bitmasks properly leading to bugs. Unfortunately, simulation does not model the behavior of unaligned addresses, so the bug is not observable using current simulators. VIDI enables a developer to debug this issue by collecting a trace from a buggy hardware execution and replaying it in simulation, enabling a developer to observe bitmasks that were missing in the original simulation.

Delayed Start. The echo server works as expected if T2 starts the echo server before T1 starts DMA. However, if T2 starts the echo server after T1 begins operating, the buggy Frame FIFO is

quickly filled; the Frame FIFO drops incoming DMA data, and T1 observes data loss. The AWS F1 simulation framework cannot find this bug since it does not support multi-threaded CPU programs (the simulator segfaults). With VIDI, we debug this issue by first using LossCheck, a third party debugging tool [175], to instrument and redeploy the FPGA component to the AWS, then using VIDI to replay a buggy execution trace on hardware (simulation-based replay could not finish within a reasonable time). LossCheck reports which elements in the FIFO are overwritten and identifies the root cause of the data loss.

We conclude that VIDI is a useful tool for diagnosing hardware bugs since it enables developers to debug issues that only appear on hardware with sophisticated debugging tools.

3.5.3 Testing Case Study

We demonstrate the versatility of VIDI by using the system as a testing tool. VIDI enables better testing by allowing developers to capture real-world workloads of their systems during production. Offline, the developer can mutate the production trace to ensure that they test their circuit on inputs that are "similar" to what they expect to find in production.

We demonstrate this use case by showing how a developer can use VIDI to reorder transactions in a trace to find an existing bug in an open-source AXI communication library. Specifically, we build an end-to-end echo server application on AWS F1 that uses an existing, unchanged buggy AXI transaction filtering library, `axi_atop_filter` [32], which belongs to an academic open-source multi-core computing platform [204]. The FPGA component of the echo server receives PCIe DMA-Write requests (i.e., "pings") from a CPU program, stores the data to on-FPGA DRAM, and sends PCIe DMA-Write requests (i.e., "pongs") that write the data in on-FPGA DRAM back to CPU-side DRAM. The `axi_atop_filter` library is configured to intercept the PCIe DMA writeback requests (i.e., "pongs"), but does not filter out any transactions. It is placed after the PCIe DMA writeback logic and directly connected to the I/O interfaces that VIDI records and replays.

The `axi_atop_filter` implementation assumes that the end event of the address transaction always happens before the end events of data transactions during the PCIe DMA-Write. However, the AXI protocol does not require such an ordering (see Fig. 3.2). When the address transaction occurs after the data transaction, the `axi_atop_filter` deadlocks. Unfortunately, this case is quite rare: we have not observed it in simulation nor on real hardware, which makes it difficult to observe using traditional testing workflows.

We use VIDI to test the echo server in the following way. First, we deploy the echo server on hardware and use VIDI to capture an execution trace. We use the mutation tool (§3.4.2) in software to reorder the recorded PCIe DMA-Write related transactions. In particular, we reorder the end event of the first write data transaction in a DMA-Write operation so that it happens before the end

event of the write address transaction. The reordering models correct AXI behavior in which a CPU-side DMA controller only completes a write address transaction if it has received at least one write data transaction. When replaying with the mutated trace, VIDI observes the deadlock: the echo server never completes the writeback DMA operation. We confirm that the bugfix proposed in the repository eliminates the issue, since VIDI no longer observes deadlock when replaying the mutated trace.

We conclude that VIDI is a useful building block for testing tools, since it enables replay with carefully mutated execution traces that are closely based on real production traces.

3.5.4 Effectiveness of VIDI

In this experiment, we evaluate the number of divergences across record and replay when using transaction determinism enforced by VIDI. We use VIDI's divergence detection workflow described in §3.3.6: First, VIDI records a *reference trace* using configuration (R2). Then, VIDI records a *validation trace* using configuration (R3). VIDI finally compares the *reference* with the *validation trace* for divergences.

For this divergence detection approach to work correctly, the recording has to work transparently, i.e., without altering the functionality of the program when it is not being recorded. We first describe our process that gives us confidence that the recording is transparent and correct. Then, we evaluate and describe the divergences that are observed when using VIDI.

Recording: We identify if any errors arise on our workloads by comparing the application output when using the (R1) configuration (i.e., disabled recording, disabled replaying) and the (R2) configuration (i.e., enabled recording, disabled replaying). We observe that each application produces the same result (i.e., renders the same image, makes the same classification, produces results that cross-check with a software implementation, etc.) and that no deadlocks or protocol violations occur. Consequently, we are confident that our implementation transparently records the transactions on our workloads, and therefore it is suitable for recording the output trace for divergence detection.

Replaying: Next, we evaluate whether the replay output diverges from the original output. In particular, we run each execution using (R2) and (R3), as described above, and check that each output channel produces the same number of transactions, that each transaction has the same content, and that the ordering of replayed transaction events is the same. The number and the happens-before relationships of replayed transaction events are equivalent across record and replay for all applications. The content of all output transactions is equivalent across recording and replay for all but one application, DRAM DMA, which has about one content divergence every one million transactions. We use VIDI's divergence report to locate the application's cycle-dependent polling

Table 3.2: On-FPGA resource overhead of VIDI, broken-down by resource types (i.e., LUT, FF, and BRAM) and normalized to the resource available on the F1 FPGA.

App	LUT (%)	FF (%)	BRAM (%)
DMA [41]	6.18	4.34	6.92
3D [290]	5.57	3.82	6.92
BNN [290]	5.67	3.82	6.92
DigitR [290]	5.65	3.82	6.92
FaceD [290]	5.64	3.82	6.92
SpamF [290]	5.63	3.82	6.92
OpFlw [290]	5.73	3.86	6.92
SSSP [19]	5.58	3.82	6.92
SHA [20]	5.60	3.82	6.92
MNet [21]	5.61	3.81	6.92

behavior, which we replace with cycle-independent interrupts to eliminate all content divergences (see §3.3.6).

We conclude that VIDI is an effective record/replay tool that enforces transaction determinism for real-world applications.

3.5.5 Efficiency of VIDI

In this section, we evaluate the efficiency of VIDI by measuring the runtime performance overhead—i.e., the slowdown of end-to-end performance due to VIDI's recording—and the resource overhead—i.e., the additional hardware resources that are used by VIDI for record/replay. In addition, we quantify the benefit of coarse-grained input recording by comparing the size of trace captured by VIDI with the size of the trace that would be captured by a cycle-accurate record/replay tool.

Runtime Performance Overhead. We measure VIDI's runtime performance overhead by comparing our evaluated applications' native end-to-end performance (i.e. configuration R1) against their performance with VIDI's recording (i.e. configuration R2). For each application, we run the experiment 10 times and report the average overhead and the standard deviation. As shown in Table 3.1, most applications encounter negligible overhead (i.e., ¡2%) when using VIDI's recording, with the largest overhead of 10.58%. Noise in the experimental setup (standard deviation), caused by, e.g., sharing in the AWS cloud environment, even outweighs the average recording overhead for 3D Rendering, BNN, etc. Notably, VIDI is configured to record additional output transaction contents for divergence detection in this experiment (i.e. using configuration R2). Developers can opt out of divergence detection to further reduce the performance overhead.

We conclude that VIDI has low performance overhead that makes it suitable for a production deployment.

Resource Overhead. We collect the resource overhead of VIDI in terms of three types of on-FPGA

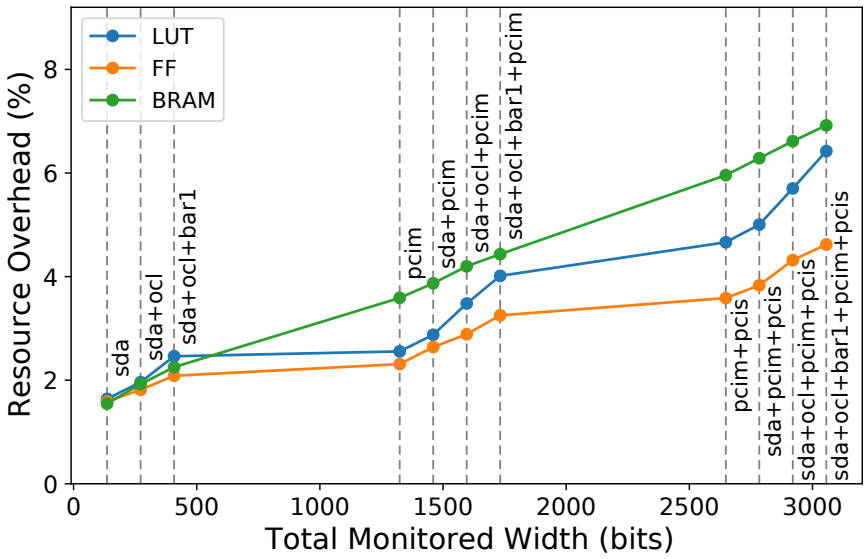


Figure 3.7: A break-down of VIDI resource overhead when monitoring different combinations of AWS F1 AXI interfaces.

resources: LUT (logic resource), FF (register resource), and BRAM (on-chip memory resource). F1's synthesis toolchain (Vivado), reports the overhead normalized to the resource utilization afforded to each accelerator on AWS F1. While VIDI's implementation remains unchanged across benchmarks, different Vivado optimizations may result in varying resource overhead. Table 3.2 identifies that VIDI incurs less than 7% resource overhead across all types of resources on all benchmarks, which is small enough to enable a production deployment. Notably, VIDI is configured to record all 5 AXI interfaces on F1, while each application uses at most 3 interfaces. In a real-world setup, developers can configure VIDI to only record/replay the AXI interfaces used by the application and to only perform record or replay, which further reduces the resource overhead.

To better understand how VIDI's resource overhead scales when recording different AXI interfaces, we also configure VIDI to record different combinations of the 5 AXI interfaces on F1. Fig. 3.7 visualizes this scalability analysis. The total width of monitored interfaces ranges from 136 bits (i.e., single 32-bit AXI-Lite interface such as the sda/ocl/bar1 MMIO bus) to 3056 bits (i.e., all three AXI-Lite buses plus all 512-bit AXI interfaces, the pcim and pcis DMA buses).

Results show that VIDI has low resource overhead and scales roughly linearly with the width of monitored interfaces.

Benefit of Coarse-Grained Input Recording. To demonstrate the benefit of coarse-grained input recording, we compare size of traces captured by VIDI to the size of traces that would be captured on the same I/O interfaces by a cycle-accurate record/replay approach. In this experiment, we first record each benchmark and calculate the size of the trace that is produced. Then, we determine the size of a cycle-accurate trace by multiplying the total size of all input signals to the circuit by the

number of cycles executed by the circuit. We determine the number of cycles executed by each benchmark by running the benchmark natively (i.e., without VIDI).

Table 3.1 shows the average size of the VIDI traces and the trace size reduction compared with cycle-accurate recording. VIDI delivers a median of 1092x reduction on trace sizes thanks to coarse-grained input recording. We conclude that VIDI drastically increases resource efficiency.

3.6 Discussion

In this section we explain the rationale behind the design decision to have VIDI use a custom packet format (§3.3.1) rather than using physical timestamps.

A physical-timestamp-based record/replay approach seems alluring, since the timestamps are readily available and straightforward to use in a design. However, our investigation shows that physical-timestamp-based approaches overwhelm storage (e.g., PCIe or DRAM), which can lead to data loss. Prior work [105] observed the same limitation; it encountered trace loss after 2ms when tracing ˜2000 bits at 100MHz, which translates to a 25 GB/s peak bandwidth.

We perform a back-of-the-envelop calculation to determine how quickly a physical-timestamp-based approach, namely Panopticon, would encounter trace loss in our experimental setup (i.e., the setup of VIDI from §3.5). We assume that: (a) the application only traces the largest AXI channel, which is 593 bits running at 250MHz and is mainly used for burst PCIe communications. (b) the tool can use all 43MB of BRAM avaialbe on the AWS FPGA for its trace buffer (cycle-accurate tools typically use such BRAM). (c) the trace store has a maximum effective bandwidth of 5.5 GB/s (this is the effective bandwidth of PCIe storage reported on the AWS FPGA [178, 250]). Under these conditions, Panopticon would need to support a peak tracing bandwidth of 18.5 GB/s, so a 3.3ms burst traffic would cause Panopticon to stop sending data to the trace store and begin losing trace data. The trace buffer (43MB of BRAM) is not large enough to prevent loss in a real application: our evaluation shows that 9 out of 10 real-world benchmarks have trace sizes that are larger than the BRAM buffer.

VIDI will also overwhelm trace storage in similar conditions to Panopticon[3]. However, the transaction abstraction enables VIDI to avoid trace loss via a back-pressure mechanism. VIDI's back-pressure mechanism causes additional overhead (§3.5.5) but does not affect the correctness of record/replay since the transactions are asynchronous and robust to delays. On the contrary, a physical timestamp-based scheme such as Panopticon cannot use back-pressure both correctly and efficiently. Delays caused by back-pressure invalidate cycle-accurate physical timestamps. Alternatively, a physical-timestamp-based approach could pause the application's clock when storage

[3]VIDI's packet format is slightly more compact than recording physical timestamps and could support slightly longer bursts

is overwhelmed and thus guarantee that transactions are recorded/replayed at the correct moment. However, such pauses are challenging and expensive to implement, especially when interacting with closed-source IPs [57]. For example, Synergy [163] in §3.7 can pause application clocks, but reports an overhead of a factor of 3-4, which renders it impractical for production scenarios.

Therefore, we opt for transaction determinism instead of physical-timestamp-based (or cycle-accurate in general) approaches in Vidi.

3.7 Related Work

Software Record/Replay. VIDI is inspired by prior software record /replay tools, which record all non-deterministic events (e.g., system calls and thread interleavings) [47, 61, 66, 86, 99, 106, 119–122, 160, 169, 170, 180, 185, 194, 198, 200, 203, 211, 244] during execution and reproduce these events to replay an execution. Some systems investigate using specialized hardware to assist and accelerate record/replay [59, 85, 184, 190, 269, 270, 292]. Unfortunately, these systems cannot record and replay FPGA applications because they cannot observe hardware events that are only visible by the FPGA.

Hardware Simulation. Simulation is widely used to test and verify an FPGA application before it is deployed. Most simulators [42, 43, 229, 233, 259] can record a cycle-accurate trace of an accelerator execution and visualize the execution in a waveform showing the value of each signal at each cycle. While simulation provides full signal visibility and is useful for debugging, it can be orders of magnitude slower than a hardware execution on FPGA [217], which limits practicality. In contrast, VIDI imposes just 1.98% runtime overhead during recording and can thus be practically used on real hardware.

Record/Replay on FPGAs. FPGA vendors provide libraries, such as Intel's SignalTap [131] and Xilinx's ILA [266], that enable a developer to perform cycle-accurate recording of specified signals during FPGA application's execution. Unfortunately, these tools cannot record all dynamic signal values, a requirement of cycle-accurate record/replay, because the resulting trace would be too large to practically store. In contrast, VIDI uses coarse-grained input recording to reduce the size of the trace and uses external resources (e.g., CPU-side DRAM) so that it can store the entire trace. Panopticon [105, 225] records input data on the FPGA, which it reproduces in simulation to replay the execution. Not only do these approaches not support replay on real hardware, they also drop recorded inputs if the trace is generated too quickly. In contrast, VIDI records an FPGA's execution without dropping transactions (§3.3.1) and can replay much larger program executions on actual FPGAs.

Debug Governors [182] provide an interactive debugging tool to record and replay the traffic of a single-channel streaming interface. Debug Governors do not consider happens-before relationships

among transactions on multiple channels, which prevents the replay of real-world applications, including all benchmarks evaluated in §3.5.

Runtimes for FPGAs. AmorphOS [147], Optimus [174], and Coyote [157] add virtualization shim layers between the FPGA shell and the user-provided accelerators for resource multiplexing; VIDI adopts a similar architecture. Cascade [217] and Synergy [163] propose compiler transformations for FPGA designs. Their concept of virtual clocks could be used to implement cycle-accurate record/replay that would have no replay divergences but have significantly more runtime overhead.

FPGA Checkpointing. Checkpointing-based tools [56, 57, 151, 163, 217] can take a snapshot of a running FPGA accelerator and use the snapshot for future analyses. Recently, StateLink [58] also leveraged the transaction abstraction to extend checkpointing support to more complex FPGA designs and build a co-simulation framework. StateLink and VIDI could create a strong synergy, where VIDI allows users to partially record an execution starting from a checkpoint, or provides transaction ordering information for StateLink to simulate with higher fidelity.

3.8 Chapter Summary

In this chapter, we presented VIDI, the first record/replay system for real-world FPGA applications running on real hardware. VIDI' design is informed by: (1) the *transaction determinism* insight, which only tracks and enforces necessary orderings of transaction events across record and replay. (2) the *coarse-grained input recording* mechanism, which only captures the start/end events and the content of each transaction. We designed and evaluated VIDI on Amazon EC2 F1 instances using one debugging case study, one testing case study, and 10 other applications. We found that VIDI incurs low performance slowdown and resource overhead, and is practical for real-world deployments.

CHAPTER 4

Virtualization Support for Shared-Memory FPGA Platforms

4.1 Introduction

Field-Programmable Gate Arrays (FPGAs) present a significant advantage for accelerating custom workloads, spanning domains such as machine learning [222, 223, 226, 277, 285], data compression [206], scientific computing [113], database operations [196, 226], and graph processing [55, 288]. The agility of FPGAs enables cloud providers to adapt to evolving application workloads by reprogramming the FPGA chip to various accelerators, resulting a cost-effective and versatile alternative to ASICs [78, 228].

Given the significant non-recurring engineering costs associated with hardware design [148] and the prevalence of software developers among cloud customers, one of the FPGA deployment models is that cloud services, such as Amazon and Microsoft, pre-configure FPGAs to be commonly-used accelerators for customers to choose from [49, 183]. In the interest of maximizing FPGA resource utility and achieving higher return of investment, cloud services seek to virtualize FPGAs, allowing for shared use between customers [147, 243]. Albeit the existence of multi-tenant FPGA virtualization techniques [74, 80, 102, 147, 154, 197, 202, 245, 249, 253, 278], they are restricted to FPGA platforms offering a *host-centric* programming model, rather than a *shared-memory* model.

The main distinction between host-centric and shared-memory programming model hinges on the ability of accelerators to indepenently perform direct memory access (DMA) to acquire data from the CPU-side DRAM. In host-centric FPGA devices, DMAs are managed solely by the host CPU, which configures the DMA engine to forward data from CPU-side host memory to the associated accelerator on FPGA, inhibiting these acceleartors from independently initiating DMA transfers. The host-centric programming model is favored by many FPGA vendors [48, 237, 264] but is not ideal for applications that necessitate fine-grained and irregular memory interactions, like pointer chasing in graph processing [256] and database [226] accelerators, due to the need for perpetual coordinations between CPU and FPGA for each DMA. FPGA developers are thus forced to either

sequentially transfer small data segments or serialize the data every time to an acceleration-friendly format, both scenearios would hurt performance.

To address the inefficiencies associated with the host-centric programming model, emerging FPGAs propose a more versatile shared-memory programming model [53, 124, 126, 232]. On shared-memory FPGA devices, individual accelerators can independently initiate DMAs and share an memory address space with the host CPU processes, requiring the CPU to simply provide a pointer to the accelerator's initial input data. Then, the accelerator can manage subsequent DMA requests autonomously, bypassing CPU. A motivating example will be presented in §4.2.1, where the shared-memory model demonstrates performance improvement over the host-centric model by 37% to 85%.

However, virtualization support for the shared-memory FPGA platforms is challenging, mainly because the hypervisor in such setup is reponsible for providing a consistent views to shared virtual memory address spaces across software running on the CPU and accelerators running on the FPGA. Virtual memory address spaces used by application are often hidden behind layers of page-table translations that CPUs with mature virtualization support are particularly good at. For typical I/O devices, there is also Single Root Input/Output Virtualization (SR-IOV) [158], which supports hardware-accelerated IO virtulization to isolate virtual DMAs on *PCIe* links. On the contrary, shared-memory FPGA platforms lacks of mature virtualization support. For example, the Intel HARP platform [124, 126] has both a PCIe and a UPI (for cache-coherence) interface, beyond the capability of SR-IOV. Such shared-memory FPGA platforms also don't work with the nested page-tables used by CPU-side virtualization architecture extensions.

In this project, we designed Optimus, a pioneering scalable hypervisor designed for shared-memory FPGA virtulization. Optimus enables cloud providers to partition a single FPGA into isolated accelerators to concurrently accelerate various tasks, improving the FPGA resource utilization. Optimus focuses on the deployment model where cloud providers pre-configure FPGAs as common fixed-function accelerators for customers to choose from (e.g., Amazon F1's libraries/registries or services alike [80, 147]). Optimus enables scheduling of virtual machines to use pre-configured accelerators, while cloud providers may periodically reconfigure those accelerators in response to customers' evolving demands.

Optimus supports a combination of *spatial multiplexing* and *temporal multiplexing*. *Spatial multiplexing* partitions the physical FPGA into isolated accelerators, assignable to distinct VMs [74, 80, 102, 147, 155, 245, 253]. Meanwhile, *Temporal multiplexing* facilitates a preemptible time-shared usage of one accelerator by numerous VMs [78, 249]. To enable temporal multiplexing, Optimus introduces a preemption interface as a contract between the hypervisor and the accelerator, allowing virtual accelerators to swap their program states to and from system memory upon context switching.

The key technique in OPTIMUS is called *page table slicing*, which overcomes the DMA isolation challenges in existing shared-memory FPGA platforms. Page table slicing is inspired by prior software-only DMA isolation techniques [239, 261], but generalized to a software-hardware co-design technique in OPTIMUS that provides virtualization support agnostic to specific accelerators. Page table slicing is achieved by spanning the hypervisor design across CPU and FPGA. The CPU-side is derived from typical hypervisor software stack while the FPGA-side *hardware monitor* collaborates on the partition of a single IO page table among all virtual machines without the expensive IO page table context switching. Although OPTIMUS is soly implemented and evaluated on the Intel Skylake HARP [124] platform, we believe its underlying design principles are compatible with other shared-memory FPGA platforms.

Our evaluation demonstrates that OPTIMUS can spatially multiplex up to eight distinct physical accelerators and improve the aggregated application-level throughput of twelve real-world benchmarks by 1.98x-7x. For temporal multiplexing, OPTIMUS can efficiently and fairly schedule virtual accelerators, showing at most 9% runtime overhead in the worst case.

In this project, I mainly worked on the following things:

- For spatial multiplexing, I built the guest device driver and userspace library to manage and support the page table slicing.
- For temporal multiplexing, I designed and implemented a preemption interface that allows the hypervisor to allocate and schedule virtual accelerators to time-share the same physical accelerator.
- For evaluation, I built two microbenchmarks to characterize the latency and bandwidth performance overhead of OPTIMUS and interated them with the temporal multiplexing interface.

4.2 Background

Field Programmable Gate Arrays (FPGAs) are chips that can be programed (also called reconfigured) into custom circuits (e.g., accelerators). FPGA developers often use hardware description languages such as Verilog [238] and VHDL [63] to describe their circuit designs. A *synthesizer* program translates these designs into native FPGA *bitstreams* (i.e., binaries).

The rest of this section will introduce necessary background of FPGA programming models as well as FPGA virtualization.

4.2.1 FPGA Programming Models

The programming model of an FPGA is determined via a reserved portion of the FPGA called a *shell*, which serves like the operating system in the software world and is often provided by

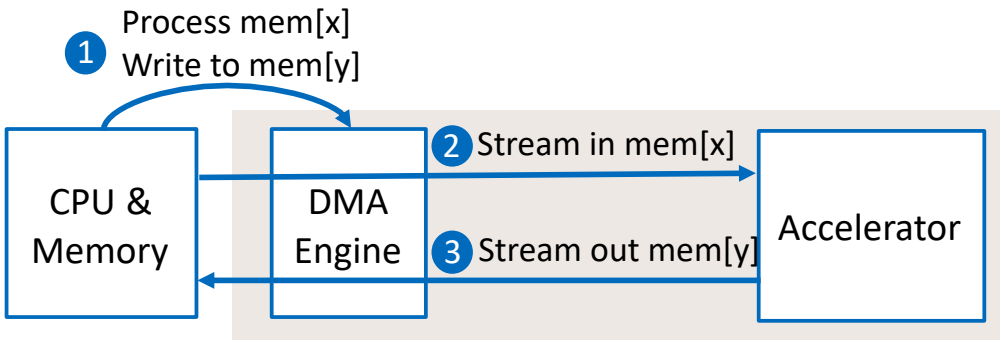


Figure 4.1: Illustration of the host-centric programming model

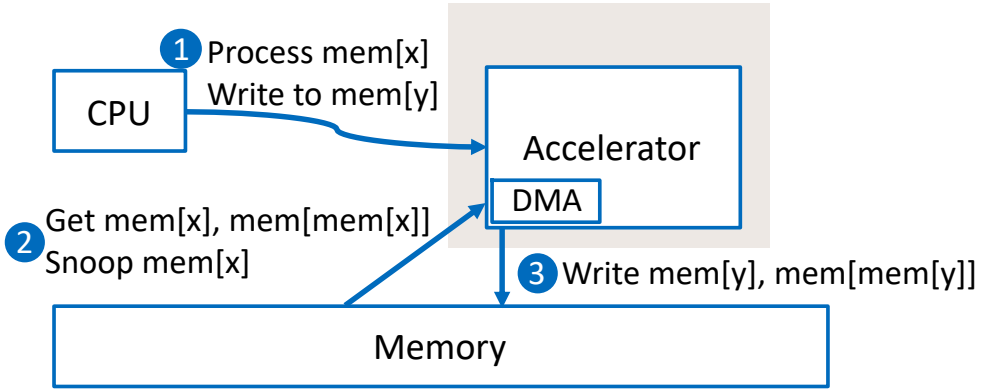


Figure 4.2: Illustration of the shared-memory programming model

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 employs one of two programming models: *host-centric* or *shared-memory*. In both programming models, the shell exposes a memory-mapped IO (MMIO) control plane for software on the CPU to manage the accelerator. The key difference between these models is whether accelerators can independently initiate direct memory accesses (DMAs).

In the more widespread host-centric model, the accelerators are unaware of the system memory layout and thus cannot issue DMAs. Instead, the CPU configures a DMA engine to transfer data from system memory to the accelerators, as shown in Fig. 4.1. The host-centric model yields simpler accelerator design, as accelerator developers only work with simple data flow on FPGAs and offload DMA management mostly to the software.

However, the host-centric model incurs unnecessary latency for applications that exhibit pointer chasing behavior. Specifically, the CPU must repeatedly configure the DMA engine to fetch new data for each accelerator, because the addresses of new data depend on accelerator results from the previous iteration.

As for the emerging shared-memory model (e.g., that of Intel HARP [124]), each accelerator is cognizant of the system memory layout and can initiate its own DMAs, as shown in Fig. 4.2. For example, the accelerator may dereference pointers or even snoop cache-coherence traffic to have fine-grained interactions with the CPU. Therefore, shared-memory accelerators can engage in pointer chasing without relying on the host to manage subsequent DMAs. The abilities to avoid the latency on host-centric platforms and efficiently parse complex application data structures, make the shared-memory model attractive to certain workloads such as graph processing and database acceleration.

4.2.2 FPGA Virtualization

The accelerators on an FPGA can be multiplexed spatially [74, 80, 147, 197, 202, 245, 253] and temporally [80, 147, 197, 202, 249, 273]. 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) [249] or preemptive (i.e., pause-and-resume) [147].

To virtualize an FPGA, each virtual accelerator's on-FPGA resources as well as IO channels must be isolated [147]. 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 [80] or paravirtualization [249].

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 [158] and PASID [127]. 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 [124]), 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 [158], shared-memory FPGAs have only supported one VF for the past five years [124, 126]. 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 [124] 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.

4.3 Goals and Challenges

OPTIMUS targets a deployment model that cloud providers configure FPGAs as a set of popular accelerators for their customers, avoiding the downtime penalty of frequent accelerator reconfiguration [49, 80, 147]. To enable efficient and flexible sharing of accelerators on FPGAs, OPTIMUS aims to support spatial multiplexing [74, 80, 147, 197, 202, 245, 253] to partition an FPGA into a fixed set of accelerators, and temporal multiplexing [80, 147, 197, 202, 249, 273] to oversubscribe 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 [74, 80, 147, 197, 202, 245, 249, 253], OPTIMUS aims to share a unified virtual memory address space between software (CPU programs) and hardware (FPGA programs), preserving the same programmability as developing on bare-metal shared-memory FPGA devices. However, such programmability implies that cloud application developers should not deal with low-level platform details such as memory isolation, and should instead rely on straightforward memory abstractions of virtualized unified address spaces [1, 166, 214, 287] provided by OPTIMUS.

Isolation While host-centric FPGA virtualization solutions focus on the isolation of on-FPGA DRAM [74, 80, 147, 197, 245, 249, 253], OPTIMUS must consider the isolation of system memory in the presence of accelerator-initiated DMAs. Given the limited vendor-provided built-in support of virtualization, OPTIMUS must provide strong DMA isolation within a single IOMMU address space. We note that OPTIMUS relies on the synthesizer to place each physical accelerator on isolated pieces

of the FPGA fabric, safely without interferences. As a result, OPTIMUS does not protect accelerators from on-FPGA side channels, which is an interesting direction for future work.

Scalability As the number of accelerators on an FPGA increases, the FPGA I/O interfaces have to be multiplexed for each accelerator via implementation of *multiplexers*. In the case of OPTIMUS, such multiplexer design is especially important because our target FPGA devices operate at higher frequencies (thus higher performance) than state-of-the-art FPGA virtualization solutions – there by placing tigher constraints on the hypervisor design. The scalability of such multiplexers is critical to the scalability of the hypervisor. As shown in prior work [147], we should use a hierarchical multiplexer tree rather than a flat multiplexer structure to scale better, espeically given the tigher design constraints of OPTIMUS.

Efficiency OPTIMUS must have low virtualization overhead, without sacrificing too much performance when compared to bare-metal accelerator. Specifically, the sum of each virtual accelerator's bandwidth is supposed to match the FPGA's total bandwidth. Furthermore, the extra latency added by the hypervisor must be minimized. Given the frequent occurrence of DMAs as compared to MMIOs, the primary challenge in OPTIMUS is to reduce the virtualized DMA overhead. Unfortunately, typical efficient DMA isolation methods such as SR-IOV and PASID do not currently provide a comprehensive and scalable DMA virtualization solution to the shared-memory FPGA platforms. Therefore, OPTIMUS must synthesize virtualization support into the FPGA to customize an efficient hardware-assisted hypervisor.

Fairness In line with prior work [147], 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.

4.4 Design

In brief, OPTIMUS follows a mediated pass-through [239] architecture, in which control plane operations (i.e., MMIO) are intercepted by the CPU-side software hypervisor, while data plane operations (i.e., DMA) are processed by the FPGA-side hardware hypervisor and bypassing the CPU. Fig. 4.3 shows the high-level architecture of OPTIMUS, featuring 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

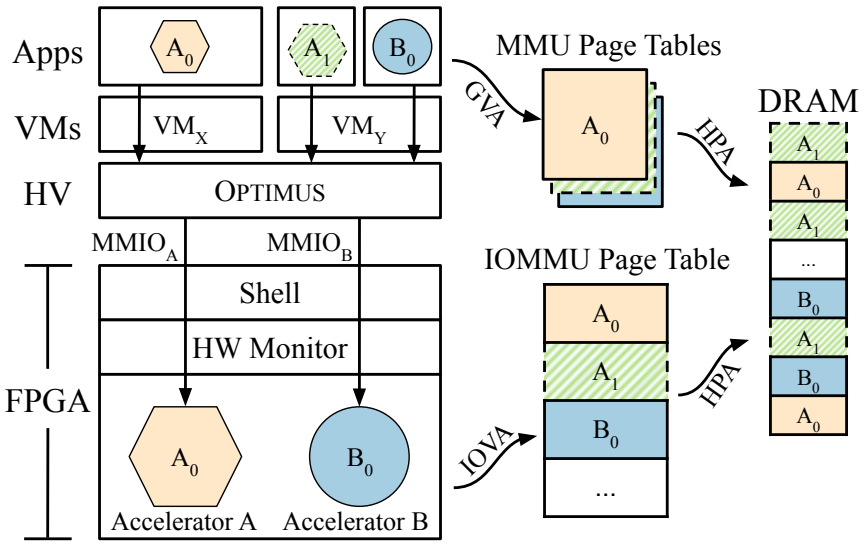


Figure 4.3: OPTIMUS design overview, shown with two physical accelerators for brevity. OPTIMUS spatially multiplexes a shared-memory FPGA as physical accelerators (A and B), and temporally multiplexes physical accelerators as virtual accelerators (A_0, A_1, and B_0).

its virtualization scalability by temporally sharing a physical accelerator among several *virtual accelerators* (A_0 and A_1). For example, in Fig. 4.3, 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.

In the rest of this section, I will mainly focus on the design components of which I contributed the majority. For a complete explanation of the OPTIMUS design, please refer to our publication [174].

4.4.1 Userspace Library for Spatial Multiplexing

To interface with the physical or virutal acceelrator in guest virtual machine userspace, OPTIMUS offers a new wrapper of the underlying native platform APIs [2]. OPTIMUS also provides a separate implementation of the guest userspace API tailored for Verilog simulations, which is needed in typical accelerator development workflow.

From the guest's perspective, each accelerator is a PCIe device. OPTIMUS offers a customized FPGA driver and a userspace library that work in tandem to allow easy integration with existing shared-memory applications and accelerators. The driver is responsible for initializing the virtual accelerator, including mapping MMIO regions to userspace, allocating 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.

A noteworthy component in the userspace library is a customized memory allocator, which confines all shared-FPGA related memory regions inside a pre-defined fixed-size virtual memory address space. This is to assist page table slicing, the key technique in OPTIMUS's design, which uses linear address mapping policy to isolate and convert guest virtual addresses to I/O virtual addresses. For each accelerator managed by OPTIMUS, either via spatial multiplexing or temporal multiplexing, it can only access a contiguous partition of the hypervisor-managed I/O page table. As a result, all shared virtual memory addresses, which are accessible by both the application running on CPU and the virtualized accelerator on FPGA, must reside in a contiguous range of the guest virtual memory address space. That continuous range of virtual address space will be eventually assigned to an equally-sized partition in the I/O page table. Upon virtualized FPGA accelerator initialization, the customized memory allocator will register the memory-pool start address with the hypervisor via hypercalls, such that all future memory (de)allocation operations will be transparently synchronized to the corresponding partition in the I/O page table.

4.4.2 Preemption Interface for Temporal Multiplexing

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 [147].

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 [147]. 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 [217]. 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.

4.5 Implementation

OPTIMUS is implemented atop the Intel HARP shared-memory FPGA platform [124] using Intel's Core Cache Interface (CCI-P) [123]. However, OPTIMUS's design can be applied to any shared-memory FPGA platform with IOMMU support (which is necessary to implement page table slicing).

Implementations authored by other collaborators will be omitted here and can be found in [174]. In brief, the CPU-side software hypervisor is based on the linux kernel KVM infrastructure [153], and the hardware monitor is based on Intel's open-source multiplexer (MUX) module [130].

Besides, I implemented the guest FPGA driver and userspace API library with 2,033 lines of C code and customized a memory allocation library [165], collaborating with the hypervisor to manage DMA regions for accelerators. I also implemented two microbenchmarks (MM and LL in Table 4.1) with 1715 lines of Verilog, which will be used in §4.6 to evaluate latency and bandwidth overhead of OPTIMUS.

Guest-MMIO 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, the size of page table slices is configurable and can easily adapt to systems where more than 64 GB of RAM is needed per virtual accelerator.

Given the pre-defined size of a page table slice, OPTIMUS's guest library uses the `mmap()` system call with the `MAP_NORESERVE` flag to reserve the full-size of a 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 required by the hardware monitor is calculated based on the value stored in this register.

Temporal Multiplexing Interface The temporal multiplexing is transparent from the guest perspective: accelerator state swap in/out will be automatically handled by the hypervisor and guest driver. For flexible memory management, each guest application uses the guest userspace API to allocate a buffer in host DRAM for storing accelerator state upon preemption. Meanwhile, the accelerator developers are responsible for supporting the temporal multiplexing interface (§4.4.2) in their FPGA design. I implemented temporal multiplexing support in the aforementioned two microbenchmarks (MB and LL in Table 4.1).

4.6 Evaluation

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

Efficiency How much virtualization overhead does OPTIMUS incur compared to pass-through (i.e., direct assignment) (§4.6.2)?

Scalability How does OPTIMUS scale with respect to the number of acceleration jobs concurrently executing on the FPGA (§4.6.3)? How does OPTIMUS scale with respect to the oversubscription factor of each accelerator (i.e., the number of virtual accelerators per physical accelerator) (§4.6.4)?

Fairness Does OPTIMUS enforce different scheduling policies among its time-shared virtual accelerators (§4.6.5)?

In our publication [174], we also discussed FPGA resource overhead, utilization improvements and DMA bandwidth fairless of the spatial multiplexing, etc., which are omitted here since irrelevant to my contribution.

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 4.1: 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.

4.6.1 Experimental Setup

Hardware We evaluate OPTIMUS on Intel Skylake HARP [124]. The platform features a 2.8 GHz Xeon CPU and a 400 MHz Arria 10 FPGA [129] located in the same package. The CPU and FPGA are connected via a single UPI [186] 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 [271] (virtual IOMMU) support in QEMU.

Benchmarks Table 4.1 shows the fourteen benchmarks with which we evaluate OPTIMUS. Ten of these benchmarks are ported from HardCloud [79], an open-source framework that offloads OpenMP [93] 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) [288], and a bitcoin miner [3] to our virtualization platform. HardCloud benchmarks, SSSP, and Bitcoin are chosen to represent real-world applications.

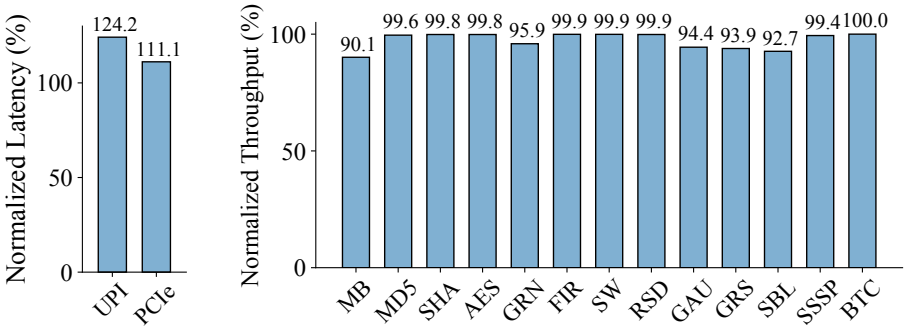
Since no open-source benchmarks for HARP place sufficient strain on OPTIMUS's bandwidth and latency for a single acceleration job, and because no existing benchmarks conform to OPTIMUS's preemption interface, I implemented two microbenchmarks specifically for the HARP platform from scratch. I also implemented OPTIMUS's preemption interface for both of them 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 [257] 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 [123]. 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 [123], 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 4.1 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.



(a) Latency

(b) Throughput

Figure 4.4: Performance overhead of different benchmarks compared to pass-through.

4.6.2 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 Fig. 4.4.

Latency Fig. 4.4a 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 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 Fig. 4.4b 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.

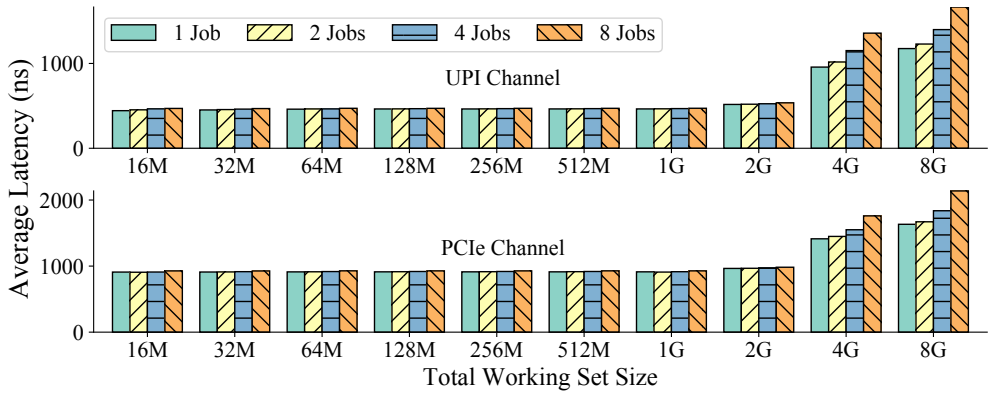


Figure 4.5: Average memory access latency of LinkedList with different working set sizes and number of virtual machines.

4.6.3 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 of scaling on latency. As shown in Fig. 4.5, 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. §4.6.3 shows the aggregate throughput of MemBench as the number of acceleration jobs and aggregate working set 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

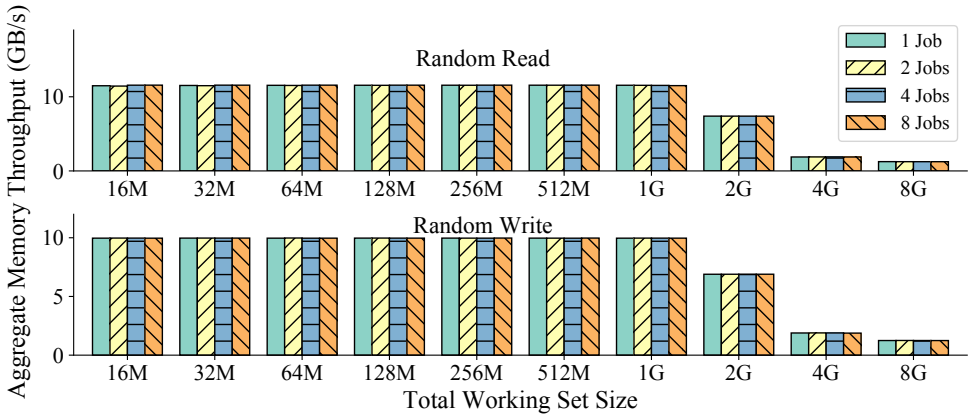


Figure 4.6: Aggregate throughput of MemBench with different working set sizes and number of virtual machines.

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.

Fig. 4.7 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.

4.6.4 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, 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.

Fig. 4.8 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

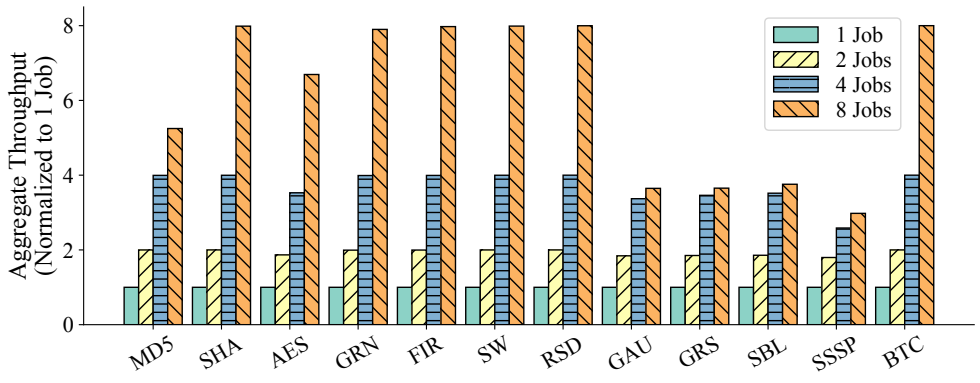


Figure 4.7: The aggregate throughput of different real-world applications, normalized to the throughput of a single VM. GAU, GRS, SBL, and SSSP fail to scale because the interconnection bandwidth becomes saturated, creating a performance bottleneck beyond four accelerators.

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 throughput drop 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 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.

4.6.5 Fairness of Temporal Multiplexing

In addition to the preemption interface, temporal multiplexing also requires Optimus to schedule virtual accelerators used in guest to run on physical accelerators. The fairness of such scheduling is essentially the cloud provider's time-sharing policy among its customers. 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

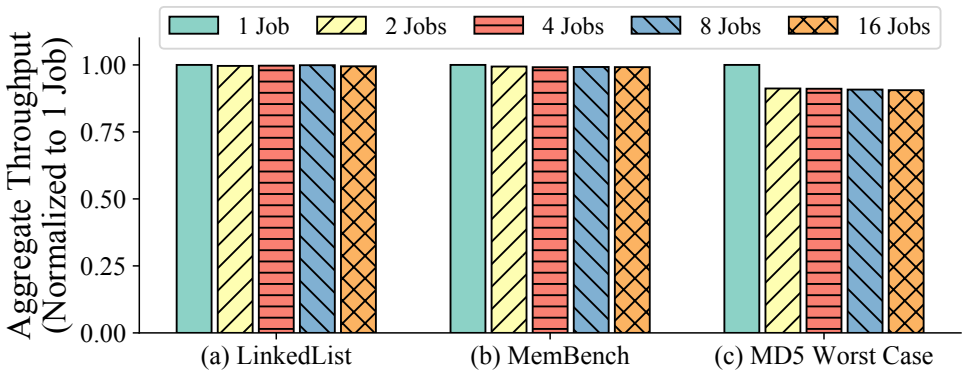


Figure 4.8: Normalized aggregate throughput in the presence of preemptive temporal multiplexing. All virtual accelerators are scheduled on a single physical accelerator.

the execution time of each virtual accelerator across varying oversubscription factors, time slice lengths, and job weights/priorities. On average, the fluctuation of actual execution time of each virtual accelerator is within 0.32% of the expected runtime, with the greatest difference being 1.42%. Thus, OPTIMUS successfully enforces each of its software scheduling policies.

4.7 Related Work

Accelerator Libraries Amazon F1 [49] and Microsoft Brainwave [72, 183] 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 [52]. The hypervisor provides a framework to share on-FPGA memory among multiple accelerators. CoRAM [88] and CoRAM++ [255] 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 [126] (PCIe-only), Intel HARP [124] (PCIe and UPI), and Enzian [53] (forthcoming). GPUs from Intel [128, 239] and NVIDIA [1, 166, 214, 287] 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 [261]) in a hardware-software co-design that is independent of accelerator design and behavior.

Overlays FPGA overlays [65, 134, 135, 156] 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 [265], Tarafdar et al. [236], and Microsoft Catapult [78, 205] 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 [74, 80, 197, 202, 245, 253] and temporal multiplexing [80, 197, 202, 249, 273] of FPGAs. While most of these works focus on host-centric FPGAs, OPTIMUS focuses on shared-memory FPGAs. An exception is AVA [273], 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 [230, 231] supplements software processes with *hardware processes*, which communicate with other processes via standard UNIX interfaces. ReconOS [171] and Hthreads [201] extend the domain of multi-threaded programming to an FPGA, and provide support for inter-thread communication and synchronization. LEAP [104] offers reliable and latency-insensitive communication channels between different hardware modules. AMORPHOS [147] 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 [125] and Xilinx [263] both offer IP to support hardware-assisted FPGA virtualization of PCIe transactions via SR-IOV [158]. However, state-of-the-art shared-memory FPGA platforms that use SR-IOV do not support more than one VF [124, 126]. 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 [74, 80, 147, 253] 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.

4.8 Chapter Summary

In this project, I presented my contributions to 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. In extreme cases, the microbenchmarks show that OPTIMUS may incur up to 24% latency and up to 10% bandwidth overhead when comparing with bare-metal setups. Experiments with twelve more realistic benchmarks show that OPTIMUS can support eight physical accelerators on a single FPGA, and improves the aggregate application-level throughput by 1.98x-7x.

CHAPTER 5

Future Work

Reverse Execution Reconstruction In Chapter 2, the symbolic execution I used is still forward program analysis. Although ER outperforms the state-of-the-art work from Microsoft, REPT [254] in certain cases, I believe reverse execution, the key idea of REPT, could even further improve failure reproduction if combined with my symbolic execution based analysis and constraint simplification. For example, there could be a reverse symbolic execution starting from the core dump with the control flow suffix trace of the failure. Then constraint simplification techniques could be applied when such reverse symbolic execution gets stuck. In that case, we might reconstruct the suffix of an interesting execution more accurate than REPT, while avoid reasoning about the potential long and irrelevant execution prefixes.

Heterogeneous Architeture Aware CPU Scheduling CPU architecture has evolved to be heterogeneous, with the recent chiplet design trend pushing the heterogeneity even further. In the early days, when the single core performance scaling was stalled due to the power wall, CPU architects turned to parallelism to continue driving performance scaling. In particular, Simultaneous Multithreading (SMT) was introduced to improve the single core efficiency/utilization via cheap hardware threads that share many functional units (e.g., TLB, L1, L2, executuon units, etc.). Later, Non-Uniform Memory Access (NUMA) architecture expands parallism further to CPUs on separate chips, while also scaling to more memory capacity and bandwidth proportionally to the overall core count. More recently, major CPU vendors (AMD since Zen2, Apple since M1 and Intel since Xeon 4th-gen) also adopted the chiplet architecture. A chiplet CPU packages multiple smaller, reusable chip dies, which even potentially use different technology node, to a single chip.

The above heterogeneous architectures share a common challenge: the trade-off between locality and load-balance, which is essentially application-dependent. Depending on the application behavior, co-locating two threads to hyperthreads on the same physical core may benefit from (e.g., locality reduces cache miss) or be penalized by (e.g., contention causes more stall) shared resources such as L1/L2 cache, TLB or execution units. Similarly, co-locating application threads to the same NUMA

node or same chiplet has the same issue: the shared LLC would only be benefitial if achieving good locality, otherwise the application is likely to be limited by the shared LLC capacity or the shared memory controller bandwidth. People have already been aware of these issues and have proposed improvements for a specific heterogeneity: (1) [224] discussed the challenges on chiplet architecture and proposed userspace hinting for scheduler [191] (2) [209] proposed to use memory address profiling on the classic NUMA balancing problem.

Another aspect of the locality/load-balance trade-off is related to the fair queueing in the CPU scheduler. Based on preliminary analysis of Google Production Workload Traces [100] and experiments with the parsec benchmarks [64], I found that existing CPU scheduling (e.g., linux kernel CFS) tends to spread an application execution to all available cores for extreme load-balancing. As a result, we observe that the number of contexts live (i.e., unique threads that have executed) on one physical core is significantly greater than the amortized average number of contexts on each physical core (e.g., total number of threads divided by total number of cores). Such symptom worsens even further when the CPU is oversubscribed with more application threads, which is a common practice to improve CPU utilization. I have a hypothesis that for some applications, sacrificing load-balancing to restrict certain threads in a smaller set of cores could yield better end-to-end performance than spreading all application threads to all available cores.

Following this direction, it might be worth exploring how to utilize runtime statistics, such as hardware performance monitors (especially the address sampling support) and threads wake-up dependencies, etc., to quantify the sharing and contention between different application threads. There could be a generic approach, like automatically clustering threads according to their microarchitecture level metrics, that can improve the scheduling decisions for all types of CPU heterogeneity (i.e., SMT, NUMA and chiplet). Such customized scheduling policies may also need to be fine-tuned for different hardware platform specifications and applications.

CHAPTER 6

Conclusion

In this dissertation, I have explored three previous projects that share a common thread: co-design hardware and software to enhance the reliability and efficiency of heterogeneous computing systems. With the introduction of diverse and novel specialized hardware, my work focused on identifying persisting challenges that could benefit from the new hardware capabilities; or broadening the applicability of existing system supports to accommodate emerging heterogeneous architectures.

In Chapter 2, I described how the combination of on-CPU specialized tracing hardware with classic symbolic execution techniques could improve the failure reproduction problem, thus enhancing software reliability. In Chapter 3, I showed how a software system-level support, record replay, can be generalized and adapted to CPU-FPGA heterogeneous systems, resulting in an efficient way to reproduce FPGA program executions, thus enhancing hardware reliability. In Chapter 4, I worked on virtualization, an important technique in cloud systems, which can be generalized to emerging cache-coherent shared-memory CPU-FPGA systems, thus improving FPGA utilization and cloud efficiency.

BIBLIOGRAPHY

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

[2] Open programmable acceleration engine - documentation. https://opae.github.io/.

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

[4] CVE Detail. https://nvd.nist.gov/vuln/detail/CVE-2004-0597, 2004.

[5] NASM 0.98.x - Error Preprocessor Directive Buffer Overflow. https://www.exploit-db.com/exploits/25005, 2004.

[6] Coreutils, fault in od. https://lists.gnu.org/archive/html/bug-coreutils/2007-08/msg00034.html, 2007.

[7] Coreutils, fault in pr. https://lists.gnu.org/archive/html/bug-coreutils/2008-04/msg00177.html, 2008.

[8] Fixing bugs - if you can't reproduce a bug, you can't fix it - dzone agile. https://dzone.com/articles/if-you-cant-reproduce-bug-you, 09 2012.

[9] Free php benchmark performance script. http://www.php-benchmark-script.com, 2012.

[10] Sec bug #61065: Secunia sa44335 - arbitrary code execution. https://bugs.php.net/bug.php?id=61065, 2012.

[11] https://issues.apache.org/jira/browse/HBASE-10210, 2013.

[12] https://issues.apache.org/jira/browse/HBASE-10237, 2013.

[13] https://issues.apache.org/jira/browse/HDFS-5690, 2013.

[14] pbzip2. https://github.com/jieyu/concurrency-bugs/tree/master/pbzip2-0.9.4, 2013.

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

[16] sr #108885: 4-byte script triggers null ptr deref and segfault. https://savannah.gnu.org/support/index.php?108885, 2015.

[17] Xilinx Integrated Logic Analyzer v6.2 (LogiCORE IP). https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf, 2016.

[18] Sec bug #74194: a heap-buffer-overflow when serializing arrayobject. https://bugs.php.net/bug.php?id=74194, 2017.

[19] https://github.com/aeonstasis/sssp-fpga, 2018.

[20] https://github.com/dowenberghmark/FPGA-SHA256, 2018.

[21] https://github.com/onioncc/iSmartDNN, 2018.

[22] Adverse interaction between .stats and .eqp in the cli. https://www.sqlite.org/src/tktview/7be932d, 2018.

[23] CVE-2018-1000030: Python 2.7 readahead feature of file objects is not thread safe. https://bugs.python.org/issue31530, 2018.

[24] CVE Detail. https://www.cvedetails.com/cve/CVE-2018-6323/, 2018.

[25] Intel Quartus Prime Pro Edition User Guide: Debug Tools. page 196, 2018.

[26] Assertion fault when multi-use subquery implemented by co-routine. https://www.sqlite.org/src/tktview/787fa71, 2019.

[27] Crash on query using an or term in the where clause. https://www.sqlite.org/src/tktview/4e8e485, 2019.

[28] CVE Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-11596, 2019.

[29] MatrixSSL <4.0.2 - Stack Buffer Overflow Verifying x.509 Certificates. https://www.exploit-db.com/exploits/46435, 2019.

[30] What does debugging a program look like? - Julia Evans. https://jvns.ca/blog/2019/06/23/a-few-debugging-resources/, 06 2019.

[31] https://foss.heptapod.net/pypy/benchmarks/-/tree/branch/default/unladen_swallow/performance, 2020.

[32] Buggy axi_atop_filter. https://github.com/pulp-platform/axi/commit/a8a3a2a322602399bfa3b6bda2e1f754994751f4, March 2020.

[33] CVE database. https://www.cvedetails.com, 2020.

[34] Exploit Database Archive. https://www.exploit-db.com/, 2020.

[35] Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, 2020.

[36] Memory prices decreasing with time. https://jcmit.net/mem2015.htm, 2020.

[37] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark, 2020.

[38] Resvg-test-suite. https://github.com/RazrFalcon/resvg-test-suite/tree/master/png, 2020.

[39] rr. https://rr-project.org, 2020.

[40] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html, 2020.

[41] Official repository of the AWS EC2 FPGA Hardware and Software Development Kit. https://github.com/aws/aws-fpga, 2021.

[42] Questa verification & simulation. https://eda.sw.siemens.com/en-US/ic/questa/simulation, 2022.

[43] Xcelium logic simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, 2022.

[44] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, 2006.

[45] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, ASPLOS'21, March 2021.

[46] 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, 2018.

[47] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, October 2009. Association for Computing Machinery.

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

[49] Amazon. Amazon ec2 f1 instances - run customizable fpgas in the aws cloud. https://aws.amazon.com/ec2/instance-types/f1, 2017.

[50] ARM Limited. AMBA AXI and ACE Protocol Specification, 2021.

[51] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*,

ASPLOS '13, pages 101–112, New York, NY, USA, 2013. Association for Computing Machinery.

[52] 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.

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

[54] Mona Attariyan, MIchael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, Hollywood, CA, October 2012. USENIX Association.

[55] 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.

[56] 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.

[57] 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.

[58] Sameh Attia and Vaughn Betz. Toward Software-Like Debugging for FPGAs via Checkpointing and Transaction-Based Co-Simulation. *ACM Transactions on Reconfigurable Technology and Systems*, August 2022.

[59] David F Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. *ACM SIGPLAN Notices*, 26(12):194–206, 1991.

[60] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, May 2018.

[61] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, 2006.

[62] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, page 154–163, New York, NY, USA, 2006. Association for Computing Machinery.

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

[64] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques - PACT '08*, page 72, Toronto, Ontario, Canada, 2008. ACM Press.

[65] 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.

[66] Manuel Bravo, Nuno Machado, Paolo Romano, and Luís Rodrigues. Towards effective and efficient search-based deterministic replay. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, pages 1–6, 2013.

[67] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[68] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275, San Francisco, CA, USA, 2003. IEEE Comput. Soc.

[69] 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.

[70] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 2000.

[71] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, 2011.

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

[73] Burkeley Architecture Research. TileLink 0.3.3 Specification. https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8/pub, 2014.

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

[75] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08,

pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. http://dl.acm.org/citation.cfm?id=1855741.1855756.

[76] Cadence. Jasper FPV App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html?CMP=SVG_JasGApp_IntDgn, December 2021.

[77] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project pberry: Fpga acceleration for remote memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 127–135, 2019.

[78] 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.

[79] 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.

[80] 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*, pages 1–10, 2014.

[81] N. Chen and S. Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, 2015.

[82] 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.

[83] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596, May 2020.

[84] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[85] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. Lreplay: A pending period based deterministic replay scheme. *ACM SIGARCH Computer Architecture News*, 38(3):187–197, 2010.

[86] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 34–44, USA, 2009. IEEE Computer Society.

[87] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in vivo multi-path analysis of software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2011.

[88] 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.

[89] Marcello Cinque, Catello Di Martino, and Alessandro Testa. Analyzing and modeling the failure behavior of wireless sensor networks software under errors. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International*, pages 1136–1141. IEEE, 2012.

[90] Matthew Tan Creti. *Software and hardware approaches for record and replay of wireless sensor networks*. PhD thesis, Purdue University, April 2015.

[91] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering*, 2016.

[92] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 184–197, New York, NY, USA, 2015. Association for Computing Machinery.

[93] 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.

[94] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[95] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[96] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 525–540, USA, 2014. USENIX Association.

[97] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 31–36, New York, NY, USA, 2014. Association for Computing Machinery.

[98] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[99] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[100] DynamoRIO. Google Workload Traces, 2022.

[101] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[102] 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.

[103] 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)*, pages 51–66, 2018.

[104] 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.

[105] Daniel Foisy and Sunil K. Shukla. Cycle-accurate replay and debugging of running FPGA systems, December 2015. https://patents.google.com/patent/US9217774/en.

[106] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys'11, 2011.

[107] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, 2010.

[108] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[109] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[110] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 585–598, New York, NY, USA, 2017. ACM.

[111] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22Nd*

Symposium on Operating Systems Principles, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[112] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[113] 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.

[114] Wei Guan and KJ Ray Liu. Mitigating error propagation for wireless network coding. *IEEE Transactions on Wireless Communications*, 11(10):3632–3643, 2012.

[115] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[116] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. *arXiv preprint arXiv:2108.03492*, 2021.

[117] Jim Handy. Dram prices hit historic low. https://thememoryguy.com/dram-prices-hit-historic-low/, 2019.

[118] Peter Kenevan Stephanie Lingemann Klaus Pototzky Harald Bauer, Ondrej Burkacky and Bill Wiseman. Semiconductor design and manufacturing: Achieving leading-edge capabilities. McKinsey and Advanced Electronics Practice, 2020.

[119] Nima Honarmand and Josep Torrellas. Replay debugging: Leveraging record and replay for program debugging. *SIGARCH Comput. Archit. News*, 42(3), June 2014.

[120] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 265–276, USA, 2008. IEEE Computer Society.

[121] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. *SIGPLAN Not.*, 48(6), June 2013.

[122] Shiyou Huang, Bowen Cai, and Jeff Huang. Towards production-run heisenbugs reproduction on commercial hardware. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 403–415, Santa Clara, CA, July 2017. USENIX Association.

[123] 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.

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

[125] 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.

[126] 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.

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

[128] 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.

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

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

[131] 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.

[132] Intel. Avalon® Interface Specifications. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf, May 2021.

[133] Intel Corporation. Intel processor trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[134] 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.

[135] 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.

[136] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[137] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 474–484. IEEE Press, 2012.

[138] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[139] Scotten W. Jones. Economics in the 3D Era, 2020. Data from IC Knowledge Inc. Strategic Cost Model.

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

[141] 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.

[142] 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.

[143] Baris Kasikci, Cristian Zamfir, and George Candea. CORD: A collaborative framework for distributed data race detection. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*, Hollywood, CA, October 2012. USENIX Association.

[144] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *ASPLOS*, London, UK, March 2012.

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

[146] Baris Kasikci, Cristian Zamfir, and George Candea. Automated classification of data races under both strong and weak memory models. *ACM Trans. Program. Lang. Syst.*, 2015.

[147] 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, 2018.

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

[149] Ali Kheradmand, Baris Kasikci, and George Candea. Lockout: Efficient testing for deadlock bugs. In *5th Workshop on Determinism and Correctness in Parallel Programming*, Salt Lake City, UT, March 2014.

[150] 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.

[151] 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.

[152] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[153] 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.

[154] 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.

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

[156] 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.

[157] 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.

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

[159] 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.

[160] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 155–166, 2010.

[161] 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.

[162] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[163] 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.

[164] Mark Lapedus. Big Trouble At 3nm, 2018. Data from International Business Strategies.

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

[166] 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.

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

[168] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

[169] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. ireplayer: In-situ and identical record-and-replay for multithreaded applications. *SIGPLAN Not.*, 53(4), June 2018.

[170] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. Light: Replay via tightly bounded recording. *SIGPLAN Not.*, 50(6), June 2015.

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

[172] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200, June 2005.

[173] O. Shacham K. Olukotun L. Hammond C. Batten M. Horowitz, F. Labonte and K. Rupp. 50 Years of Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data, 2022.

[174] 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.

[175] 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*, ASPLOS 2022, page 946–962, New York, NY, USA, 2022. Association for Computing Machinery.

[176] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, page 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[177] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, 2007.

[178] Piyush Manavar, Manoj Nambiar, Nupur Sumeet, Rekha Singhal, Sharod Choudhary, and Amey Pandit. Experience with PCIe streaming on FPGA for high throughput ML inferencing, October 2021.

[179] Marco Merlini. DebugGovernor at Github. https://github.com/esophagus-now/ye_olde_verilogge/blob/fe24224938814a799789ff159c659e857f76a8b2/dbg_guv/axis_governor/axis_governor.v#L105, May 2020.

[180] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[181] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[182] 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.

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

[184] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. *ACM SIGARCH Computer Architecture News*, 36(3):289–300, 2008.

[185] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *International Symposium on Computer Architecture*, 2008.

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

[187] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[188] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, page 216–227, New York, NY, USA, 2006. Association for Computing Machinery.

[189] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005.

[190] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 284–295. IEEE, 2005.

[191] K Prateek Nayak. Userspace Hinting for Task Placement. The Linux kernel mailing list, 2022.

[192] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, November 2020.

[193] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.

[194] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 377–389, 2017.

[195] OpenCores. WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores, 2010.

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

[197] 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.

[198] Soyeon Park, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, Shan Lu, and Yuanyuan Zhou. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[199] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 2–11, New York, NY, USA, 2010. Association for Computing Machinery.

[200] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11, 2010.

[201] 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.

[202] 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.

[203] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[204] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, 2019.

[205] 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. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[206] 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.

[207] 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.

[208] Andrew Quinn, Jason Flinn, and Michael Cafarella. Sledgehammer: Cluster-fueled debugging. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 545–560, Carlsbad, CA, October 2018. USENIX Association.

[209] Bharata B Rao. Memory access profiler(IBS) driven NUMA balancing. The Linux kernel mailing list, 2023.

[210] Vaibhav Rastogi, Zhengyang Qu, Jedidiah McClurg, Yinzhi Cao, and Yan Chen. Uranine: Real-time privacy leakage monitoring without system modification for android. In *International Conference on Security and Privacy in Communication Systems*, pages 256–276. Springer, 2015.

[211] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2), May 1999.

[212] Caitlin Sadowski and Jaeheon Yi. How developers use data race detection tools. In *Workshop on Evaluation and Usability of Programming Languages and Tools*, 2014.

[213] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. *SIGPLAN Not.*, 48(4), March 2013.

[214] 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.

[215] 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.

[216] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[217] 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, 2019.

[218] Julian Schütte, Dennis Titze, and José María De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 370–379. IEEE, 2014.

[219] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 571–572, New York, NY, USA, 2007. Association for Computing Machinery.

[220] 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.

[221] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, December 2009.

[222] 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.

[223] 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.

[224] Gautham R. Shenoy and K. Prateek Nayak. Linux Kernel Scheduling and split-LLC architectures: Overview, Challenges and Opportunities. Linux Plumbers Conference, 2022.

[225] Sunil Shukla and David F. Bacon. Cycle-Accurate Replay and Debugging of Running FPGA Systems. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 30–30, May 2015.

[226] 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.

[227] SiFive. SiFive TileLink Specification 1.8.0. https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf, August 2019.

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

[229] Wilson Snyder. https://www.veripool.org/verilator/, 2021.

[230] 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.

[231] Hayden Kwok-Hay So and Robert W Brodersen. *Borph: An operating system for fpga-based reconfigurable computers*. Citeseer, 2007.

[232] 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.

[233] Synopsys. Vcs functional verification solution. https://www.synopsys.com/verification/simulation/vcs.html, 2021.

[234] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pages 288–301. ACM, 2011.

[235] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. TARDIS: software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 286–297. ACM, 2015.

[236] 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.

[237] 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.

[238] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[239] 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.

[240] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360, 2018.

[241] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[242] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. *SIGOPS Oper. Syst. Rev.*, 41(6), October 2007.

[243] 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.

[244] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[245] 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.

[246] 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, 2017.

[247] Peipei Wang, Hiep Nguyen, Xiaohui Gu, and Shan Lu. RDE: Replay DEbugging for Diagnosing Production Site Failures. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 327–336, September 2016.

[248] 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.

[249] Wei Wang, Miodrag Bolic, and Jonathan Parri. pvfpga: Accessing an fpga-based hardware accelerator in a paravirtualized environment. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–9. IEEE, 2013.

[250] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When FPGA Meets Cloud: A First Look at Performance. *IEEE Transactions on Cloud Computing*, 10(2):1344–1357, April 2022.

[251] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[252] Yuke Wang, Boyuan Feng, Gushu Li, Georgios Tzimpragos, Lei Deng, Yuan Xie, and Yufei Ding. Tiacc: Triangle-inequality based hardware accelerator for k-means on fpgas. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 133–142. IEEE, 2021.

[253] 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.

[254] Xinyang Ge Weidong Cui, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: reverse debugging of failures in deployed software. In *OSDI*, 2018.

[255] 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.

[256] 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.

[257] 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.

[258] WikiChip. Table of Semiconductor Manufacturers with a Cutting Edge Logic Fab, 2022.

[259] Stephen Williams. Icarus verilog. http://iverilog.icarus.com/, 2022.

[260] Louis Woods, Jens Teubner, and Gustavo Alonso. Real-time pattern matching with fpgas. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1292–1295. IEEE, 2011.

[261] 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.

[262] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914. IEEE, 2015.

[263] 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.

[264] Xilinx. Dma for pci express (pcie) subsystem.

[265] Xilinx. Sdaccel development environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[266] Xilinx. Integrated logic analyzer v6.2. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf, 2016.

[267] Xilinx. AXI Protocol Checker v2.0. https://www.xilinx.com/support/documentation/ip_documentation/axi_protocol_checker/v2_0/pg101-axi-protocol-checker.pdf, April 2018.

[268] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, 2017.

[269] Min Xu, Rastislav Bodik, and Mark D Hill. A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135, 2003.

[270] Min Xu, Mark D Hill, and Rastislav Bodik. A regulated transitive reduction (rtr) for longer memory race recording. *ACM SIGARCH Computer Architecture News*, 34(5):49–60, 2006.

[271] Peter Xu. Device assignment with nested guest and dpdk. https://www.linux-kvm.org/images/a/a6/KVM_Forum_2018_viommu_vfio.pdf, 2018.

[272] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A nanosecond scale logging system. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 335–350, Boston, MA, July 2018. USENIX Association.

[273] 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.

[274] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[275] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. Automated debugging for arbitrarily long executions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM, May 2013. USENIX Association.

[276] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Transactions on Software Engineering*.

[277] 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.

[278] 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.

[279] 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.

[280] 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.

[281] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. *SIGARCH Comput. Archit. News*, 39(1), March 2011.

[282] 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.

[283] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19–33, 2017.

[284] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 131–146, 2019.

[285] 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.

[286] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery.

[287] 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.

[288] 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.

[289] 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.

[290] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[291] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. MIMIC: Locating and understanding bugs by analyzing mimicked executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 815–826, New York, NY, USA, September 2014. Association for Computing Machinery.

[292] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1155–1170, 2021.