Vidi: Record Replay for Reconfigurable Hardware

Gefei Zuo
gefeizuo@umich.edu
University of Michigan
USA

Jiacheng Ma
jcma@umich.edu
University of Michigan
USA

Andrew Quinn
aquinn1@ucsc.edu
University of California, Santa Cruz
USA

Baris Kasikci
barisk@umich.edu
University of Michigan & Google
USA

ABSTRACT

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 paper, we 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. We evaluate Vidi 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.

CCS CONCEPTS

• **Hardware** → **Reconfigurable logic and FPGAs**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

FPGA, Record Replay, Debugging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582040

ACM Reference Format:
Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. 2023. Vidi: Record Replay for Reconfigurable Hardware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582040

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 [47, 55, 81, 92, 100, 103, 104], databases [70, 77, 83, 94], graph processing [17, 29, 90, 106], networking [25, 36, 89], storage [50], remote memory [28, 39] and compression [75, 101]. Cloud vendors have begun providing FPGA instances on their platforms due to the promise that these resources show [12, 14].

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 [36].

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 [79], data-race detectors [80]), performance profilers (e.g., perf, Coz [33]), and comprehensive logging infrastructure (e.g., Nanolog [99], log20 [105]). The ecosystem of FPGA debugging tools is comparatively lacking—while the research community has built FPGA development tools [19, 48, 52, 59, 63, 87, 102], studies show that most FPGA developers desire more and better debugging tools [1]. 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 [31], debugging [66], performance profiling [16], security auditing [34], and replication [24]. 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 [2], SignalTap [6], and Panopticon [37, 82]), 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 §6 for further discussion). Other tools operate in simulation (e.g., VCS [86], Vivado [8]), 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 §5.2 and §5.3).

Other record/replay approaches (e.g. DebugGovernor [63]) 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 (§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 (§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 (§3.6).

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 (§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 §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 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.

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).

FPGAs often send and receive data from components in a heterogeneous system (e.g., CPUs, NICs) using *communication channels*

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

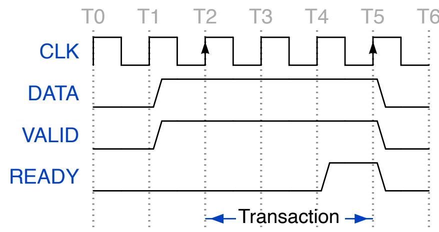

Figure 1: The waveform for a VALID/READY handshake of an AXI Channel. Each line represents the values of a different signal in the communication channel.

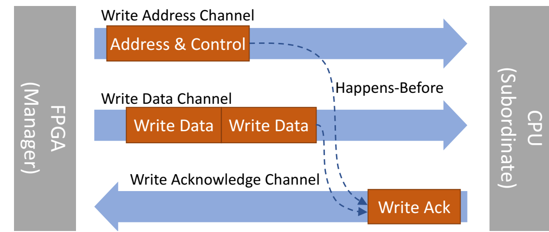

Figure 2: Example group of FPGA-CPU communication channels. The displayed roles of the FPGA and CPU in this example can be reversed. The orderings between transactions (the dashed arrows between orange boxes in each channel) are important for correct operation and hence a record/replay system must capture these orders.

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).

2.1 Channels and Transactions

Fig. 1 shows an example waveform diagram of an instance of *VALID/READY* handshaking in AXI [15], 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. 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 [15, 26, 45, 69, 84]. Transactions identify the start and end events between which signals in a communication channel are unmodified and meaningful.

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 [15, 96].

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 [15]. Fig. 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 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[15, 84] 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 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 §3.6, 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 §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. 1, T2, T5, and DATA at T2), which reduces storage and performance overhead compared to existing work (see §5.5). Note that VIDI assumes a proper application to at least implement the single-channel handshaking (§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 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 (§4) treats the CPU as the external environment and the entire FPGA application as the FPGA

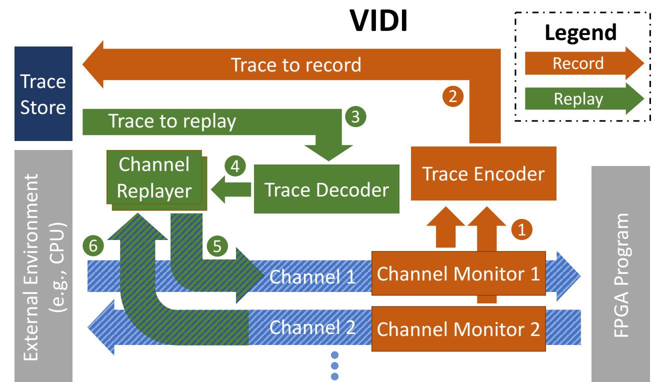

Figure 3: VIDI's design overview. For simplicity, we show a single input channel (Channel 1) and output channel (Channel 2). The green arrow (❺) from the channel replayer to Channel 1 indicates that, during replay, the channel replayer creates input transactions on input channels. The green arrow (❻) from Channel 2 to the channel replayer indicates that, during replay, the channel replayer receives output transactions on output channels.

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 §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 (§3.1) 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 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 (§3.2). 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).

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 (§3.4). 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 (§3.5): input channel replayers (i.e., channel replayers that are senders) control when each input transaction starts (e.g. the VALID signal in Fig. 1) and its content (e.g. the DATA signal in Fig. 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. 1) ❻. All channel replayers coordinate using vector clocks [53] 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 §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 §5.4). In §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 §5.2. Below, we describe how each Vidi component works.

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 (§3.6), the system configures output channel monitors to track the content of each completed transaction, in addition to the transaction end event. Fig. 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 (§3.5).

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. 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).

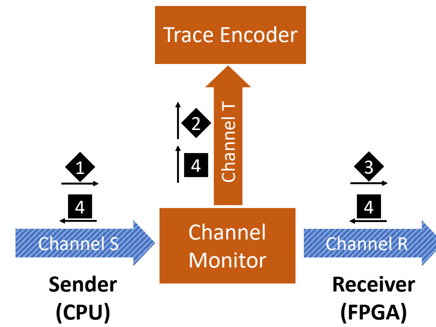

Figure 4: A channel monitor deployed on an input channel. Black arrows, boxes, and diamonds identify the steps needed to transparently perform coarse-grained input recording between the sender and receiver.

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. 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 §6).

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 (§3.5) and provide transaction determinism.

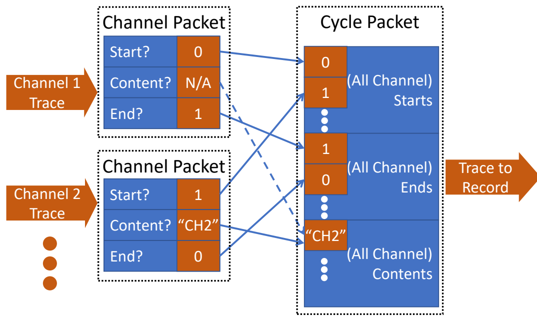

Figure 5: Architecture of the trace encoder.

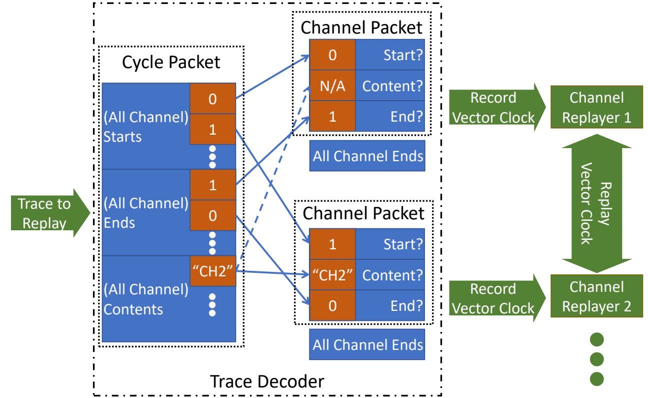

Figure 6: Architecture of the Trace Decoder and Channel Replayers with an example trace.

At each clock cycle, the trace encoder adds the content from all channel packets into a *cycle packet* (Fig. 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 nth field is set in Starts, then the nth 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 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.

3.4 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. 5). The trace decoder then decomposes the fields of each cycle packet (i.e., Starts, Ends, and Contents) into individual channel packets (see §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 §3.2) 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 (§3.5).

Fig. 6 is an example of decoding a cycle packet from the trace encoded in Fig. 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.

3.5 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.

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* [53] 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 ith 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 ith 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 ith channel, the channel replayer for that channel sends a message to all channel replayers; each channel replayer increments the ith element in their T_{current} by 1 after receiving the message.

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).

3.6 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 §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 (§3.1). 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 §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.

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 (§3.6).

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 §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 [9]. 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 (§3.1). Ensuring the ordering

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

Table 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 [9]	1.66	5.93±0.45	0.81	97x
(2) 3D [107]	4.14	0.54±2.88	0.14	1,439x
(3) BNN [107]	6.43	0.63±1.68	0.31	966x
(4) DigitR [107]	9.56	0.03±0.14	0.97	468x
(5) FaceD [107]	17.41	-0.05±1.28	0.12	7,011x
(6) SpamF [107]	1.56	10.54±0.40	0.83	88x
(7) OpFlw [107]	13.79	1.91±0.27	1.33	490x
(8) SSSP [3]	397.83	0.00±0.01	0.002	10,149,896x
(9) SHA [4]	31.75	0.64±0.06	1.23	1,219x
(10) MNet [5]	110.71	0.11±0.27	0.51	10,163x

properties required for a correct channel monitor proved extremely challenging. For example, we found that Debug Governor [63], 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 [61]. 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 [27] to formally prove that channel monitors (Fig. 4) enforce critical properties (e.g., intercepted transactions handshake correctly and are not reordered nor dropped).

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.

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 §3.6. Vidi also includes a trace mutation tool that can reorder a trace's transaction events to aid testing §5.3.

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 (§5.2). How does Vidi help FPGA debugging?

Testing (§5.3). How does Vidi help FPGA testing?

Effectiveness (§5.4). How well does the transaction determinism provided by Vidi preserve the same output across recording and replaying?

Efficiency (§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?

5.1 Experimental Setup

Benchmark Selection. We first port a buggy Frame FIFO implementation from a recent survey of FPGA bugs [59] to AWS F1 to demonstrate how Vidi assists debugging. Then, we port a buggy component of an open-source AXI communication library [7] 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 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 [107], 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 (§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 (§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 (§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.

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 [59]. 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 [59], 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.

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` [7], which belongs to an academic open-source multi-core computing platform [74]. 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. 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 (§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.

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 §3.6: 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 behavior, which we replace with cycle-independent interrupts to eliminate all content divergences (see §3.6).

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

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 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

Table 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 [9]	6.18	4.34	6.92
3D [107]	5.57	3.82	6.92
BNN [107]	5.67	3.82	6.92
DigitR [107]	5.65	3.82	6.92
FaceD [107]	5.64	3.82	6.92
SpamF [107]	5.63	3.82	6.92
OpFlw [107]	5.73	3.86	6.92
SSSP [3]	5.58	3.82	6.92
SHA [4]	5.60	3.82	6.92
MNet [5]	5.61	3.81	6.92

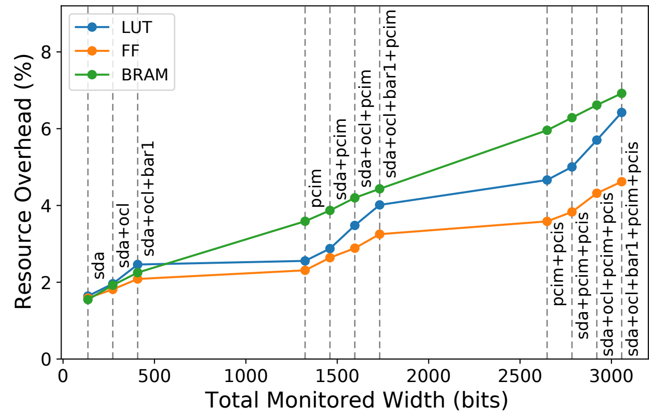

Figure 7: A break-down of VIDI resource overhead when monitoring different combinations of AWS F1 AXI interfaces.

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 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 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. 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 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.

6 DISCUSSION

In this section we explain the rationale behind the design decision to have VIDI use a custom packet format (§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 [37] 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 §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 [60, 91]). 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 (§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 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 [19]. For example, Synergy [54] in §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.

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) [13, 22, 23, 32, 35, 38, 40–43, 51, 56, 57, 62, 65, 68, 71–73, 76, 88] during execution and reproduce these events to replay an execution. Some systems investigate using specialized hardware to assist and accelerate record/replay [21, 30, 64, 67, 97, 98, 108]. 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 [10, 11, 85, 86, 93] 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 [78], 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 [44] and Xilinx's ILA [95], 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 [37, 82] 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

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

FPGA's execution without dropping transactions (§3.1) and can replay much larger program executions on actual FPGAs.

Debug Governors [63] 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 §5.

Runtimes for FPGAs. AmorphOS [46], Optimus [58], and Coyote [49] add virtualization shim layers between the FPGA shell and the user-provided accelerators for resource multiplexing; Vidi adopts a similar architecture. Cascade [78] and Synergy [54] 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 [18, 19, 48, 54, 78] can take a snapshot of a running FPGA accelerator and use the snapshot for future analyses. Recently, StateLink [20] 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.

8 CONCLUSION

In this paper, 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.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Christopher J. Rossbach, for their valuable feedback. We thank YosysHQ for an academic licence of the SymbiYosys toolchain. This work is supported by generous gifts from Intel Labs, Google, Microsoft, NSF/Intel joint grant #2010810, the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides the source code and related scripts of the full implementation of Vidi and all of our evaluated benchmarks. These allow simulation, synthesis and experiments on the actual FPGA hardware to be reproduced.

In particular, we first walk through the interactive debugging (§5.2) and testing (§5.3) case studies. Then we demonstrate how to run larger-scale experiments on the rest of 10 applications to evaluate Vidi's effectiveness and efficiency. Finally we provide data analysis scripts to reproduce Table 1, Table 2 and Fig. 7.

The artifact requires access to an FPGA development server (with Xilinx FPGA toolchains), an AWS EC2 F1 instance (with one Xilinx UltraScale+ VU9P FPGA), prebuilt FPGA images and synthesis reports (the synthesis workflow will be provided but it is optional).

A.2 Artifact Check-list

- **Program:** Rosetta FPGA benchmark and 6 other individual applications. All of them are publicly available and have been included in this artifact.
- **Compilation:** [open-source]: make, g++ (with c++17 support) [commercial:] Vivado 2020.2, VCS-2020.12
- **Run-time environment:** Artifact was prepared on Ubuntu 20.04. Root access are not needed for simulation and synthesis, but needed for experiments on the actual FPGA hardware.
- **Hardware:** Need a special FPGA that can be rented via AWS EC2 `f1.2xlarge` instances.
- **Execution:** Exclusive access to the actual FPGA is needed. Running all experiments is expected to take for 4-5 hrs.
- **Metrics:** Execution time, Trace size, FPGA resource utilization breakdown.
- **Output:** For benchmarks, outputs are console logs and files; expected outputs are included in the artifact. For the data analysis scripts, outputs are tables and graphs; expected results are reported in the paper.
- **Experiments:** Most experiments is automated via the Makefile. Manual steps are required in the interactive case studies. Reproducing results reported in the paper are automated via python scripts. We expect empirical results to have < 5% variation.
- **How much disk space required (approximately)?:** < 10GiB
- **How much time is needed to prepare workflow (approximately)?:** < 30min
- **How much time is needed to complete experiments (approximately)?:** < 5hr
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7680535

A.3 Description

A.3.1 How to access. The source code and the tutorial are available via GitHub[4] and Zenodo[5]. Information regarding remote access to preinstalled commercial toolchains and prebuilt FPGA images will be sent to AE chairs.

A.3.2 Hardware dependencies. AWS EC2 `f1.2xlarge` instances, which include a Xilinx Virtex UltraScale+ VU9P FPGA.

A.3.3 Software dependencies. All experiments are conducted under Ubuntu 20.04. The artifact was tested using g++ 9.4.0, python-3.8, make 4.2.1, Vivado 2020.2 and VCS-2020.12.

A.4 Installation

Refer to the `artifact-eval/README.md` in the artifact. In brief, you need to `source hdk_setup.sh` and `source sdk_setup.sh`. Three basic tests are provided to confirm that the installation is

[4] https://github.com/efeslab/aws-fpga
[5] https://doi.org/10.5281/zenodo.7680535

complete: one simulation test, one synthesis test and one test on the actual FPGA.

A.5 Experiment Workflow

Refer to the `artifact-eval/README.md` in the artifact for more details.

For the debugging case study, we first run the target application on the AWS F1 instances, then debug the first bug in simulation and finally interact with multiple Xilinx tools to debug the second bug.

For the testing case study, we first collect trace of a successful execution, then mutate the trace to represent certain corner cases, then replay the mutated trace on the original buggy application and finally replay the same mutated trace on the application with a proper bugfix.

For the effectiveness and efficiency experiments, we run each application multiple times under different Vidi record/replay configurations. Metrics about their execution time, performance and storage overhead, etc. will be logged and later analyzed.

A.6 Evaluation and Expected Results

Refer to the `artifact-eval/README.md` in the artifact for more details.

For the debugging case study, you are expected to observe the buggy behavior of the target application and confirm its root cause during replay.

For the testing case study, you are expected to observe the mutated trace expose a corner case that is allowed by the protocol but one that causes the buggy application to stall.

For the effectiveness and efficiency experiments, you are expected to reproduce the results reported in §5.4 and §5.5.

A.7 Experiment Customization

Refer to the `artifact-eval/README.md` in the artifact.

REFERENCES

[1] 2015. https://www.exostivlabs.com/fpga-debug-flow-should-be-improved/.
[2] 2016. Xilinx Integrated Logic Analyzer v6.2 (LogiCORE IP). https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.
[3] 2018. https://github.com/aeonstasis/sssp-fpga.
[4] 2018. https://github.com/dowenberghmark/FPGA-SHA256.
[5] 2018. https://github.com/onioncc/iSmartDNN.
[6] 2018. Intel Quartus Prime Pro Edition User Guide: Debug Tools. (2018), 196.
[7] 2020. Buggy axi_atop_filter. https://github.com/pulp-platform/axi/commit/a8a3a2a322602399bfa3b6bda2e1f754994751f4.
[8] 2020. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.
[9] 2021. Official Repository of the AWS EC2 FPGA Hardware and Software Development Kit. https://github.com/aws/aws-fpga.
[10] 2022. Questa Verification & Simulation. https://eda.sw.siemens.com/en-US/ic/questa/simulation.
[11] 2022. Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.
[12] Alibaba. 2018. 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.
[13] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594
[14] Amazon. 2017. Amazon EC2 F1 Instances - Run Customizable FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.
[15] ARM Limited. 2021. AMBA AXI and ACE Protocol Specification.
[16] Mona Attariyan, MIchael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 307–320. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/attariyan
[17] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 228–235. https://doi.org/10.1109/IPDPSW.2014.30
[18] Sameh Attia and Vaughn Betz. 2020. Feel Free to Interrupt: Safe Task Stopping to Enable FPGA Checkpointing and Context Switching. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 1 (2020), 1–27. https://doi.org/10.1145/3372491
[19] Sameh Attia and Vaughn Betz. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/3373087.3375307
[20] Sameh Attia and Vaughn Betz. 2022. Toward Software-Like Debugging for FPGAs via Checkpointing and Transaction-Based Co-Simulation. *ACM Transactions on Reconfigurable Technology and Systems* (Aug. 2022). https://doi.org/10.1145/3552521
[21] David F Bacon and Seth Copen Goldstein. 1991. Hardware-assisted replay of multiprocessor programs. *ACM SIGPLAN Notices* 26, 12 (1991), 194–206. https://doi.org/10.1145/122759.122777
[22] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*. 154–163. https://doi.org/10.1145/1134760.1220164
[23] Manuel Bravo, Nuno Machado, Paolo Romano, and Luís Rodrigues. 2013. Towards effective and efficient search-based deterministic replay. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*. 1–6. https://doi.org/10.1145/2524224.2524228
[24] Thomas C Bressoud and Fred B Schneider. 1996. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 80–107. https://doi.org/10.1145/225535.225538
[25] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 973–990. https://doi.org/10.1145/3543668
[26] Burkeley Architecture Research. 2014. TileLink 0.3.3 Specification. https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8/pub.
[27] Cadence. 2021. 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.
[28] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project pberry: Fpga acceleration for remote memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 127–135. https://doi.org/10.1145/3317550.3321424
[29] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80. https://doi.org/10.1145/3431920.3439290
[30] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. 2010. LReplay: A pending period based deterministic replay scheme. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 187–197. https://doi.org/10.1145/1815961.1815985
[31] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002
[32] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 34–44. https://doi.org/10.1109/ICSE.2009.5070506
[33] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/2815400.2815409

[34] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 525–540. https://doi.org/10.1038/scientificamerican0469-36
[35] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224. https://doi.org/10.1145/1060289.1060309
[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.
[37] Daniel Foisy and Sunil K. Shukla. 2015. Cycle-Accurate Replay and Debugging of Running FPGA Systems. https://patents.google.com/patent/US9217774/en.
[38] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-threaded Applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*. https://doi.org/10.1145/1966445.1966465
[39] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2021. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. *arXiv preprint arXiv:2108.03492* (2021). https://doi.org/10.1145/3503222.3507762
[40] Nima Honarmand and Josep Torrellas. 2014. Replay Debugging: Leveraging Record and Replay for Program Debugging. *SIGARCH Comput. Archit. News* 42, 3 (June 2014). https://doi.org/10.1145/2678373.2665737
[41] Derek R. Hower and Mark D. Hill. 2008. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 265–276. https://doi.org/10.1109/ISCA.2008.26
[42] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. *SIGPLAN Not.* 48, 6 (June 2013). https://doi.org/10.1145/2491956.2462167
[43] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards Production-Run Heisenbugs Reproduction on Commercial Hardware. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 403–415.
[44] Intel. 2020. 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.
[45] Intel. 2021. Avalon® Interface Specifications. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.
[46] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 107–127.
[47] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. 2021. S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 194–205. https://doi.org/10.1145/3431920.3439283
[48] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. 2018. 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)*. IEEE, 76–764. https://doi.org/10.1109/FPL.2018.00021
[49] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 991–1010.
[50] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. 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)*. 955–971.
[51] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. 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*. 155–166. https://doi.org/10.1145/1811039.1811057
[52] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8. https://doi.org/10.1145/3240765.3240842
[53] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563
[54] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. 2021. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 818–831. https://doi.org/10.1145/3445814.3446755
[55] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* 275 (2018), 1072–1086. https://doi.org/10.1016/j.neucom.2017.09.046
[56] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications. *SIGPLAN Not.* 53, 4 (June 2018). https://doi.org/10.1145/3192366.3192380
[57] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. 2015. Light: Replay via Tightly Bounded Recording. *SIGPLAN Not.* 50, 6 (June 2015). https://doi.org/10.1145/2737924.2738001
[58] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 827–844. https://doi.org/10.1145/3373376.3378482
[59] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. 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* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 946–962. https://doi.org/10.1145/3503222.3507701
[60] Piyush Manavar, Manoj Nambiar, Nupur Sumeet, Rekha Singhal, Sharod Choudhary, and Amey Pandit. 2021. Experience with PCIe Streaming on FPGA for High Throughput ML Inferencing. https://doi.org/10.48550/arXiv.2110.11719 arXiv:2110.11719 [cs]
[61] Marco Merlini. 2020. DebugGovernor at Github. https://github.com/esophagus-now/ye_olde_verilogge/blob/fe24224938814a799789ff159c659e857f76a8b2/dbg_guv/axis_governor/axis_governor.v#L105.
[62] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. 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)*. https://doi.org/10.1145/3037697.3037751
[63] Marco Antonio Merlini, Isamu Poy, and Paul Chow. 2021. Interactive Debugging at IP Block Interfaces in FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 138–144. https://doi.org/10.1145/3431920.3439305
[64] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 289–300. https://doi.org/10.1145/1394608.1382146
[65] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*. https://doi.org/10.1145/1394608.1382146
[66] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. 2006. 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* (Saint Malo, France) *(SIGMETRICS '06/Performance '06)*. Association for Computing Machinery, New York, NY, USA, 216–227. https://doi.org/10.1145/1140277.1140303
[67] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 284–295. https://doi.org/10.1109/ISCA.2005.16
[68] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*. 377–389.
[69] OpenCores. 2010. WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores.
[70] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218. https://doi.org/10.1109/FCCM.2017.37
[71] Soyeon Park, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, Shan Lu, and Yuanyuan Zhou. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*. https://doi.org/10.1145/1629575.1629593
[72] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. 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*. 2–11. https://doi.org/10.1145/1772954.1772958
[73] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. 2011. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *IEEE/ACM International Symposium on Microarchitecture*. https://doi.org/10.1145/2155620.2155646
[74] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. 2019. Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for

IoT Edge Processing. *IEEE Journal of Solid-State Circuits* 54, 7 (2019), 1970–1981. https://doi.org/10.1109/JSSC.2019.2912307

[75] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 37–44. https://doi.org/10.1109/FCCM.2018.00015

[76] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (May 1999). https://doi.org/10.1145/312203.312214

[77] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 262–272. https://doi.org/10.1145/3431920.3439298

[78] Eric Schkufza, Michael Wei, and Christopher J Rossbach. 2019. 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*. 271–286. https://doi.org/10.1145/3297858.3304010

[79] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[80] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. https://doi.org/10.1145/1791194.1791203

[81] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[82] Sunil Shukla and David F. Bacon. 2015. Cycle-Accurate Replay and Debugging of Running FPGA Systems. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 30–30. https://doi.org/10.1109/FCCM.2015.68

[83] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 403–415. https://doi.org/10.1145/3035918.3035954

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

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

[86] Synopsys. 2021. VCS Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html.

[87] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308* (2021).

[88] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/1950365.1950370

[89] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. 122–135. https://doi.org/10.1145/3050220.3050234

[90] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 149–159. https://doi.org/10.1145/3431920.3439288

[91] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. 2022. When FPGA Meets Cloud: A First Look at Performance. *IEEE Transactions on Cloud Computing* 10, 2 (April 2022), 1344–1357. https://doi.org/10.1109/TCC.2020.2992548

[92] Yuke Wang, Boyuan Feng, Gushu Li, Georgios Tzimpragos, Lei Deng, Yuan Xie, and Yufei Ding. 2021. 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)*. IEEE, 133–142. https://doi.org/10.1109/CCGrid51090.2021.00023

[93] Stephen Williams. 2022. Icarus Verilog. http://iverilog.icarus.com/.

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

[95] Xilinx. 2016. Integrated Logic Analyzer v6.2. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.

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

[97] Min Xu, Rastislav Bodik, and Mark D Hill. 2003. A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*. 122–135. https://doi.org/10.1145/859618.859633

[98] Min Xu, Mark D Hill, and Rastislav Bodik. 2006. A regulated transitive reduction (RTR) for longer memory race recording. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 49–60. https://doi.org/10.1145/1168857.1168865

[99] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 335–350. https://www.usenix.org/conference/atc18/presentation/yang-stephen

[100] H. Zeng, C. Zhang, and V. Prasanna. 2017. Fast Generation of High Throughput Customized Deep Learning Accelerators on FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2017.8279792

[101] Min Zhang, Linpeng Li, Hai Wang, Yan Liu, Hongbo Qin, and Wei Zhao. 2019. Optimized compression for implementing convolutional neural networks on fpga. *Electronics* 8, 3 (2019), 295. https://doi.org/10.3390/electronics8030295

[102] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. 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* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 815–827. https://doi.org/10.1109/MICRO.2018.00071

[103] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 171–182.

[104] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 15–24. https://doi.org/10.1145/3020078.3021741

[105] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 565–581. https://doi.org/10.1145/3132747.3132778

[106] Shijie Zhou and Viktor K Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 137–144. https://doi.org/10.1109/SBAC-PAD.2017.25

[107] 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. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb 2018). https://doi.org/10.1145/3174243.3174255

[108] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1155–1170. https://doi.org/10.1145/3453483.3454101

Received 2022-10-20; accepted 2023-01-19