DiffTest-H: Toward Semantic-Aware Communication in Hardware-Accelerated Processor Verification

Kunlin You
State Key Lab of Processors
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
youkunlin24s@ict.ac.cn

Yinan Xu
State Key Lab of Processors
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
xuyinan@ict.ac.cn

Kehan Feng
Beijing Institute of Open Source Chip
Beijing, China
fengkehan@bosc.ac.cn

Luoshan Cai
State Key Lab of Processors
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
cailuoshan22z@ict.ac.cn

Yaoyang Zhou
Beijing Institute of Open Source Chip
Beijing, China
zhouyaoyang@bosc.ac.cn

Yungang Bao
State Key Lab of Processors
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
baoyg@ict.ac.cn

Abstract

Verification has become the most time-consuming phase in chip development. Co-simulation frameworks simulate the design under test (DUT) with a golden reference model (REF) and compare their instruction-level results for verification, causing over 98% communication overhead: although hardware-accelerated platforms, such as FPGA and emulators, speed up DUT simulation by 300×–10000×, overall co-simulation speedup is still limited to 2.5×–20×.

In this paper, we propose DiffTest-H, a semantic-aware, hardware-accelerated co-simulation framework with three techniques reducing communication overhead while preserving debuggability: (1) Batch minimizes communication frequency by tightly packing structurally diverse verification events into a single transfer. (2) Squash reduces data transmission volume by fusing verification events with a decoupled checking order. (3) Replay preserves instruction-level debuggability by reprocessing the original, unfused verification events around the failure point.

DiffTest-H is deployed on both Palladium emulator and FPGA to verify a 6-wide, out-of-order RISC-V processor, XiangShan. It achieves simulation speeds of 478KHz and 7.8MHz respectively, with an 80× and 78× speedup over the baseline, 119× and 1945× faster than 16-thread Verilator, and uncovers 151 bugs in XiangShan.

CCS Concepts

• **Hardware → Functional verification**.

Keywords

Processor Verification, Simulation Acceleration, Co-simulation

ACM Reference Format:
Kunlin You, Yinan Xu, Kehan Feng, Luoshan Cai, Yaoyang Zhou, and Yungang Bao. 2025. DiffTest-H: Toward Semantic-Aware Communication in Hardware-Accelerated Processor Verification. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756108

1 Introduction

Verification has become the most time-consuming phase in modern chip development, accounting for over 50% of the overall workflow [17, 18]. The challenge becomes even more significant for industrial-scale processors with complex microarchitectures and instruction set architectures (ISAs), where exhaustive verification is essential for ensuring functional correctness.

Toward more efficient verification, co-simulation frameworks [14, 21, 23, 28, 42, 54] have been widely adopted in processor verification. In co-simulation, the design under test (DUT) and a golden reference model (REF) run in parallel, comparing their architectural states after each instruction. The co-simulation framework extracts verification events from the DUT, such as instruction commit and register updates, and compares them with REF. Additionally, the DUT-specific non-deterministic events (NDEs) [14, 21, 24, 39, 53, 54], such as external interrupts and MMIO access, must be fully synchronized to REF to align its architectural states with the DUT.

However, existing co-simulation frameworks are still inefficient. Traditional software-based solutions [21, 34, 54, 55] rely on RTL simulators [43, 46] to simulate the DUT. Despite extensive research working on its performance [3, 4, 6, 15, 16, 27, 38, 47–49, 58], the

Table 1: Verification Events in the DiffTest[34, 54].

Category	Types	Representative Examples
Control Flow	5	Exceptions and interrupts, Instruction commits, Traps, ...
Register Updates	9	CSRs, General-purpose registers, Floating-point registers, ...
Memory Access	3	Load/store operations, Atomic memory operations, ...
Memory Hierarchy	6	Cache refill operations, L1/L2 TLB operations, ...
RISC-V Extensions	9	Vector/Hypervisor CSRs, Vector registers, ...

simulation speed of large-scale DUTs is only at a few kHz, making it impractical for verification requiring billions of test cycles.

Hardware-accelerated platforms, including emulator [7, 8, 19, 41, 44] and FPGA [22, 24, 31, 40, 56, 57], offer promising simulation speed for better verification efficiency. Our evaluation shows that directly deploying the DUT on the emulator (Cadence Palladium) can yield a 300× speedup over RTL simulation, and a 10,000× speedup on the FPGA (Xilinx VU19P). In contrast, leveraging co-simulations, where the DUT and the REF are deployed on the hardware and software side respectively, the speedup drops to less than 2.5× on the emulator and 20× on the FPGA. The reason is that the hardware-software communication becomes a new bottleneck, with over 98% co-simulation time consumed by communication overhead.

The hardware-software communication, as a point-to-point interaction, can be modeled by the LogGP model [1, 12] and decomposed into three phases [10, 11, 25, 26, 30, 32, 37, 45]: communication startup, data transmission, and software processing. For example, in the co-simulation framework DiffTest [34, 54], which covers 32 verification events as shown in Table 1, each verification event requires a handshake to start up communication, and then transfer the DUT's architectural state to the REF for comparison, resulting in around 15 communications and 1.2 KB transmitted data per cycle.

Existing works explore optimizations across the three phases of communication. The frequency of communication startup can be reduced by packing all verification events within a cycle into a single transfer [8, 9, 19]. The data transmission volume can be reduced by fusing same-type events, such as N instruction commits into a single N-commit event [19, 40]. The software processing latency can be hidden through hardware-software parallelism [9, 24, 31, 56, 57]. However, existing works still face the communication bottleneck: the state-of-the-art Fromajo [57] achieves only 1 MHz co-simulation speed on a 100 MHz FPGA. Moreover, fusing verification events across cycles discards per-instruction details, weakening instruction-level debuggability.

To address communication challenges, Shannon and Weaver introduced *semantic communication* [52], emphasizing that understanding the information semantics improves communication efficiency. In co-simulation, verification events likewise carry three key semantic properties, which can be exploited to optimize communication while preserving debuggability:

(1) *Structural Semantics* denotes the length and data structure of verification events, which vary significantly across event types and increase the complexity of packing and unpacking. For example, the event lengths in DiffTest [34, 54, 55] differ by up to 170×. Existing packing schemes allocate fixed space for each verification event and pad invalid events with bubbles, resulting in more communications to transmit the same set of valid events. Leveraging structural semantics, we can tightly pack variable-length events with space allocated according to length, and extract packed events with their data structures. Tight packing eliminates bubbles and reduces the required packets with less communication frequency.

(2) *Order Semantics* denotes the specific checking order of verification events. For example, the NDEs, such as external interrupts, force updates to the REF's state, requiring prior instructions to be checked while subsequent ones remain unchecked. Existing fusion approaches couple communication with checking order: the NDEs break the fusion of other events, and the already fused ones are transmitted ahead to REF for ordered checking, causing frequent fusion breaks and a limited fusion ratio. Leveraging order semantics, we can decouple communication from checking order: NDEs are transmitted ahead with order tags while other events continue to be fused, and the software reorders events by these tags to restore the required checking order. Order-decoupled fusion reduces fusion breaks and improves fusion ratio with less data transmitted.

(3) *Behavioral Semantics* denotes the architectural behaviors checked by verification events, which help localize errors to specific microarchitectural components. However, fusing verification events weakens debuggability by discarding per-instruction behavioral details. Existing debugging methods rely on hardware snapshots to rerun the entire DUT for recovering the behavioral details, resulting in considerable resource and time overhead. Leveraging behavioral semantics, we can reprocess only the unfused verification events around the failure point rather than rerun the entire DUT, thereby enabling lightweight instruction-level debugging.

Building on the above three semantic properties, we propose **DiffTest-H**, *a semantic-aware, hardware-accelerated co-simulation framework* significantly reducing communication overhead while preserving instruction-level debuggability:

(1) Batch minimizes communication frequency by tightly packing structurally diverse verification events into a single transfer. Leveraging structural semantics, Batch computes the offset length of each valid event on hardware for tight packing, while the software parses packed events according to their data structures.

(2) Squash reduces data transmission volume by fusing verification events with a decoupled checking order. Leveraging order semantics, Squash allows NDEs to be transmitted ahead with order tags, while other events continue to be fused, and the software then reorders events by these tags to restore the required checking order.

(3) Replay preserves instruction-level debuggability by reprocessing the original, unfused verification events around the failure point. Leveraging behavioral semantics, Replay reprocesses only the unfused verification events rather than rerunning the entire DUT, enabling lightweight instruction-level debugging.

DiffTest-H is implemented and evaluated within the DiffTest [34, 54, 55] co-simulation framework to verify XiangShan [35, 50, 51, 54, 55], a 6-wide out-of-order RISC-V processor, covering 32 types of verification events, including instructions, cache coherence, TLB, vectorization, and virtualization. Deployed on both the Cadence Palladium emulator and FPGA, DiffTest-H achieves simulation speeds of 478 KHz and 7.8 MHz respectively, with an 80× and 78 × speedup over the baseline DiffTest, 273× and 1945× faster than 16-thread Verilator. DiffTest-H reduces communication overhead by 99.84% on the emulator, and is 7.8× faster than the state-of-the-art [56, 57] on the FPGA. DiffTest-H has uncovered over 151 complex bugs in XiangShan that require up to 2 months to identify with Verilator but are detected within 11 hours by DiffTest-H on Palladium. All of these bugs have been confirmed and fixed by XiangShan developers with more than 780 lines of code change across 19 pull requests.

In summary, we make the following contributions in this paper:

- We identify three stages of hardware-software communication: communication startup, data transmission, software processing, and summarize three corresponding optimizations: packing, fusion, and hardware-software parallelism.
- We propose and open-source DiffTest-H[1], a semantic-aware, hardware-accelerated co-simulation framework: Batch minimizes communication frequency by tightly packing verification events. Squash reduces data volume by fusing events with a decoupled checking order. Replay preserves instruction-level debuggability by reprocessing events around failure.
- DiffTest-H, evaluated on XiangShan, an open-source 6-wide out-of-order RISC-V processor, achieves simulation speeds of 478 KHz on the Palladium emulator and 7.8 MHz on the FPGA, with an 80× and 78 × speedup over baseline DiffTest, 273× and 1945× faster than 16-thread Verilator, reducing 99.84% communication overhead on the emulator and is 7.8× faster than the state-of-the-art [56, 57] on the FPGA.
- DiffTest-H uncovers over 151 complicated bugs in XiangShan, all of which have been fixed by XiangShan developers with over 780 lines of code change across 19 pull requests.

2 Background

In this section, we present three key aspects of hardware-accelerated processor co-simulation: First, we introduce the fundamental principles and workflow of co-simulation, illustrating how it ensures verification sufficiency and instruction-level debuggability. Second, we demonstrate the general structure of co-simulation framework with an example of the DiffTest framework [34]. Third, we compare verification platforms of co-simulation, highlighting advantages and bottlenecks of hardware-accelerated co-simulation.

2.1 Processor Co-Simulation Verification

Processor co-simulation [21, 28, 42, 54] verifies functional correctness by running the design under test (DUT) in parallel with a software reference model (REF), and comparing their architectural states after each instruction.

As illustrated in Figure 1, a typical co-simulation workflow begins with the DUT and REF in the same initial state, and performs instruction-level comparison for processor verification: at each instruction, the co-simulation framework ① extracts DUT's verification events, such as instruction commits, ② notifies REF to ③ execute a corresponding instruction, and ④ compares architectural states of the DUT and REF. Once a mismatch is detected, the co-simulation aborts with a detailed bug analysis.

[1]https://github.com/OpenXiangShan/difftest

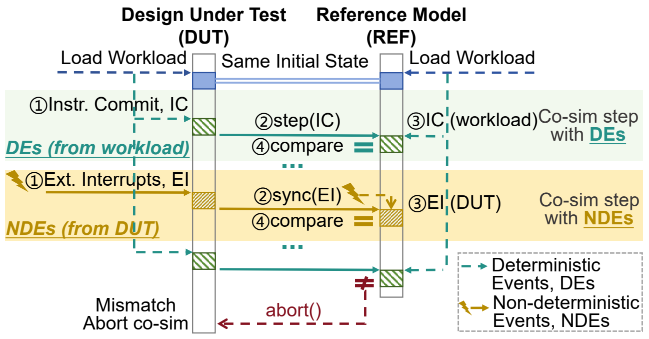

Figure 1: Co-simulation verification workflow. Each DUT event notifies the REF for execution and comparison: deterministic events are executed directly by the REF, while non-deterministic events are synchronized from the DUT.

Non-deterministic events (NDEs) [14, 21, 24, 39, 53, 54, 56, 57], such as interrupts and MMIO access, challenge co-simulation. These NDEs are specific to DUT and cannot be reproduced independently by the REF. To accommodate this, co-simulation frameworks fully synchronize these NDEs from DUT to REF at precise instructions to correctly align their architectural states.

With comprehensive checking of diverse verification events at each instruction, co-simulation offers two major advantages:

Verification Sufficiency. Covering a wide range of verification states, co-simulation ensures sufficient verification of DUT under ISA-level behaviors as well as complex non-deterministic scenarios.

Instruction-level Debuggability. Conducting comparisons after each instruction, co-simulation halts upon detecting any mismatch with a precise failure context, including mismatched verification events and cycle information for debugging.

2.2 Layout of Co-Simulation Framework

A typical processor co-simulation framework consists of three major components [21, 29, 42, 54, 55]: the monitor, the checker, and the communication unit, distributed across hardware and software to verify the correctness of the DUT.

On the hardware side, monitors are embedded into the processor to capture verification events such as instruction commits, register updates, and memory operations. Since these events are distributed across the DUT's microarchitecture, modern co-simulation frameworks [21, 54, 55] often implement monitors in high-level hardware description languages (HDLs) such as Chisel [2], enabling automated code generation to relieve manual wiring effort. The captured events are then formatted into structured data packets, which can be parsed by the software according to their data structure.

On the software side, the ISA checker operates alongside a REF, typically an Instruction Set Simulator (ISS) such as Spike [20] and NEMU [33]. The REF starts from the same initial state as the DUT, executes instructions accordingly, and is synchronized with non-deterministic events. The ISA checker uses the verification events monitored from the DUT to drive the REF's execution and performs comparisons after each instruction, as shown in Section 2.1.

Between the monitor and the checker, verification events are transmitted across the hardware-software interface, such as DPI-C.

Table 2: Comparison of Co-Simulation Platform.

Platform	Debuggability	Cost	Optimal Speed
RTL Simulator	Full visibility	Free	~ 3 KHz
Emulator	Waveform	Expensive	~ 500 KHz
FPGA	Limited	Affordable	~ 50 MHz

Given the diversity of verification events, modern co-simulation frameworks [21, 34, 54, 55] typically use individual DPI-C functions for each event, resulting in frequent communication calls and large data transfers. For example, in the co-simulation framework DiffTest [34, 54, 55] covering 32 types of verification events, each event is transmitted through a separate DPI-C interface with an aggregated size of 11,496 bytes, leading to substantial communication overhead and challenging communication-sensitive simulations.

2.3 Hardware-accelerated Co-simulation

Co-simulation, as discussed earlier, consists of two main components: the REF and the DUT. In general, the REF is implemented in software for flexibility and ease of maintenance, while the DUT is deployed through three distinct approaches: RTL simulation, hardware emulation, and FPGA prototyping. Each option presents a unique trade-off between simulation speed, debuggability, and deployment cost, as summarized in Table 2.

Depending on the DUT deployment platform, co-simulation can be categorized into two classes:

Software-based Co-simulation deploys both the REF and the DUT within a software environment, typically using RTL simulators such as Verilator [46] or Synopsys VCS [43]. In this setup, the DUT is translated into a high-level programming representation (e.g., C++), forming a directed graph where each node simulates a hardware signal. This method offers full design visibility and facilitates fine-grained debugging. However, the simulation speed is severely limited, typically reaching only a few KHz. As the complexity of the DUT increases, the performance of RTL simulators worsens, making software-based co-simulation impractical for verification requiring billions of test cycles.

Hardware-accelerated Co-simulation addresses the speed limitation by deploying the DUT onto hardware acceleration platforms, such as emulators (e.g., Cadence Palladium [7], Synopsys ZeBu [44], Siemens Veloce [41]) or FPGAs. Unlike software simulation, these platforms directly map the DUT onto physical hardware components, faithfully reproducing its behavior at much higher speeds—often achieving orders-of-magnitude improvements. The REF remains in software, preserving its flexibility.

However, this deployment across hardware and software introduces a new major bottleneck: communication overhead. Since the DUT and the REF are located on different physical platforms, verification states must be frequently transmitted across the hardware-software interface. As the amount of communication (both in terms of communication frequency and data volume) increases, the overall simulation speed becomes limited by the efficiency of the communication interface. For example, in the DiffTest framework applied to XiangShan [35], communication overhead accounts for over 98% of the total simulation time when running on Palladium emulator at 500 KHz. Similarly, the Fromajo co-simulation framework [56, 57] on a 100 MHz FPGA also experiences communication overhead exceeding 99%. These findings highlight that, while hardware acceleration significantly improves the theoretical speed of DUT emulation, the overall speed of co-simulation is fundamentally constrained by communication efficiency.

3 Analytical Overhead Model

To clarify and quantify the contributors to communication overhead in hardware-accelerated co-simulation, we introduce an analytical overhead model inspired by the LogGP model [1], which decomposes the overhead into three stages of hardware-software interaction: communication startup, data transmission, and software processing. These stages manifest differently across simulation platforms and DUT designs. To ground the model, we provide a quantitative case study across different DUTs and platforms, and identify three key optimization guidelines for communication.

3.1 Theoretical Analysis

The communication overhead in hardware-accelerated co-simulation can be modeled as the LogGP model [1, 12], where the overall latency between the FPGA/emulator and the software can be decomposed into three stages [10, 11, 25, 26, 30, 32, 37, 45]:

Communication Startup. This stage involves handshake and synchronization for each communication invocation, necessary to establish a data connection between the asynchronously running hardware and software. For example, emulator Cadence Palladium performs hardware-software synchronization at every DPI-C function calls [25], while FPGA platforms rely on valid-ready handshakes as dictated by protocols like XDMA [26]. The startup overhead is primarily determined by the communication frequency (N_{invokes}) and the per-invocation latency (T_{sync}).

Data Transmission. After the connection is established, data is transmitted over the hardware-software link in fixed-length protocol frames, and each frame incurs transmission and propagation delay. The transmission overhead scales with the total data volume (N_{bytes}) and the available bandwidth (BW).

Software processing. On the host side, software must receive data from buffers, drive the REF to execute the same instructions as the DUT (synchronize non-deterministic behavior such as external interrupts, if any), and compare their states for verifying correctness. In traditional step-and-compare strategies [34, 54, 55], hardware emulation pauses its clock until software processing completes. This part of latency is abstracted as T_{software}.

The overall communication overhead can be expressed as:

$$Overhead = N_{\text{Invokes}} \times T_{\text{sync}} + N_{\text{Bytes}} \times \frac{1}{BW} + T_{\text{software}} \quad (1)$$

3.2 Quantitative Analysis

The analytical model presented in Equation 1 provides a general framework for quantifying communication overhead on hardware-accelerated platforms with three key phases: communication startup, data transmission, and software processing. However, in practice, the relative contributions of the three phases vary significantly depending on both the verification coverage required by the DUT and the characteristics of the validation platform.

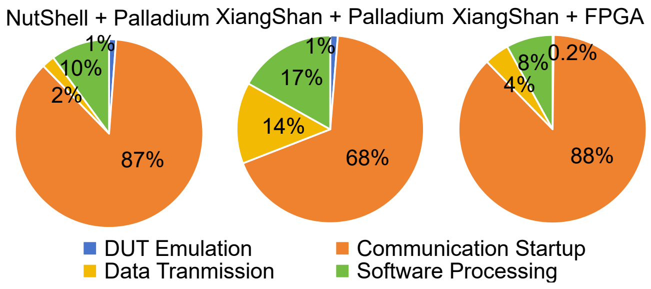

Figure 2: Overhead breakdown across DUTs and platforms.

To demonstrate the model's generality, we conduct evaluations based on NutShell [36], a scalar in-order processor, and XiangShan [35, 50, 51, 54], a 6-wide out-of-order processor, across both Palladium emulation and FPGA platforms. As shown in Figure 2, XiangShan incurs higher data transmission and software processing overhead than NutShell on the same Palladium platform, primarily due to its expanded verification events resulting in larger data volume and more complex checking. When comparing XiangShan across platforms, the FPGA setup shows higher communication startup but lower data transmission overhead, which results from the FPGA's PCIe interface exhibiting higher handshake latency yet greater bandwidth compared to Palladium's internal link.

3.3 Guiding Communication Optimizations

Based on Equation 1, the total overhead of software–hardware co-simulation can be decomposed into three phases: communication startup, data transmission, and software processing, which can be optimized through the following optimizations:

The frequency of communication startups can be reduced by packing multiple verification events into a single transfer [8, 9, 19]. For example, packing 256 16B events into a single 4KB transfer reduces startup cost by 256×.

The data transmission volume can be reduced by fusing same-type events [8, 19, 40, 57], such as N instruction commits into one N-commit event with $N\times$ reduction in data volume.

The software processing latency can be hidden through hardware–software parallelism [9, 24, 31, 56, 57], also known as non-blocking support. Such support is widely available: emulators like Palladium provide primitives such as GFIFO, while FPGAs can emulate non-blocking transmission using multi-buffer FIFOs.

4 Semantic-aware Communication Mechanism

To reduce communication overhead while preserving debuggability, we propose DiffTest-H, a semantic-aware, hardware-accelerated co-simulation framework. This section introduces the overall framework of DiffTest-H and three optimization strategies.

4.1 Overview

DiffTest-H is a semantic-aware, hardware-accelerated co-simulation framework, covering 32 types of verification events and preserving instruction-level debuggability. Figure 3 shows the DiffTest-H framework under a dual-core design. The framework comprises five components: monitor, acceleration unit, communication unit, debugging unit, and ISA checker.

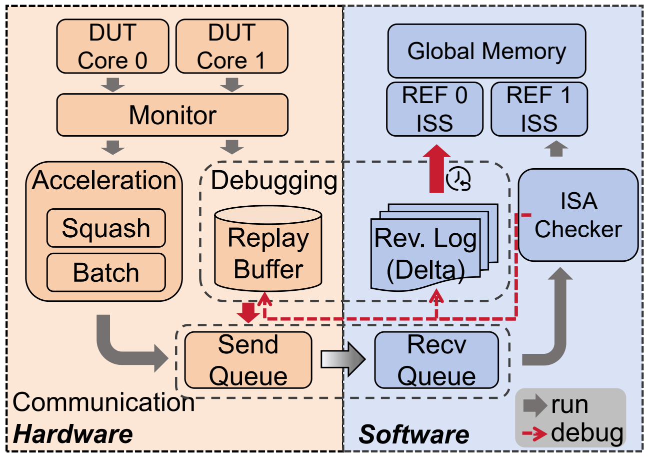

Figure 3: The DiffTest-H Framework.

From Figure 3, the monitor captures verification events from the DUT. The events are then optimized by the acceleration unit and buffering for potential debugging.

The acceleration unit applies two key optimizations: *Squash* reduces data volume by fusing events (Section 4.3) and *Batch* minimizes communication frequency by packing events into a single packet (Section 4.2). The packed data is then *non-blocking* transmitted for software processing (Section 4.5), while allowing the hardware to continue running at the same time.

Upon reception of verification events, the ISA checker runs the REF model accordingly and compares its state against the DUT to verify correctness. Once a mismatch is detected, the debugging flow is triggered: the *Replay* unit rolls back the fused buggy events and reprocesses the original unfused ones (Section 4.4), providing instruction-level debugging details around the failure point.

4.2 Batch: Packing with Structural Diversity

4.2.1 ***Why semantics matter?*** Minimizing communication frequency relies on effective packing of verification events. However, the structural diversity of events poses significant challenges to both hardware packing and software unpacking. As shown in Figure 4, the 32 types of verification events in the DiffTest co-simulation framework [34, 54, 55] exhibit size differences of up to 170×, along with highly variable transmission frequencies.

As illustrated in Figure 5, existing schemes [8, 9, 19] simplify packing by assigning each verification event with a fixed-offset region in the packet. On the hardware side, the packer writes valid events into the assigned region, while on the software side, the parser always reads from the same region and extracts the event according to its data structure. However, this fixed-offset method requires padding for invalid events to preserve offsets for others. Evaluation on DiffTest shows that such padding leads to more than 60% invalid bubbles in the packet, thereby resulting in 1.67× more communications to transmit the same set of valid events.

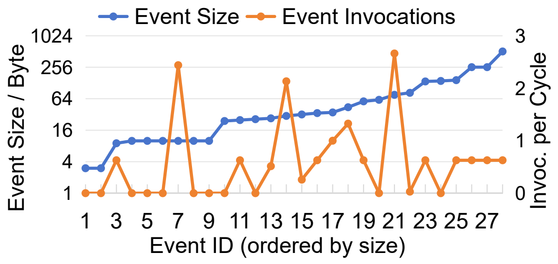

Figure 4: Verification event size and invocations in baseline DiffTest. Event IDs are ordered by increasing size, with transmission frequency measured as invocations per cycle.

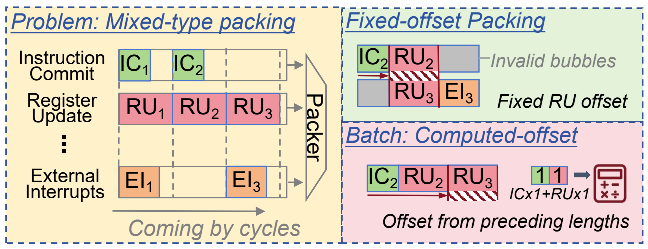

Figure 5: Comparison of packing schemes. Fixed-offset packing inserts bubbles to preserve offsets, while Batch computes offsets as the sum of preceding event lengths.

Verification events inherently contain ***structural semantics***, namely their length and data structure. By leveraging structural semantics, the hardware packer can dynamically allocate space according to the actual length of each event and tightly pack variable-length events of different types. As shown in Figure 5, the offset of a register update (RU) event can be computed by summing the lengths of prefix events. On the software side, the parser can also use the length information to compute offsets of specific events and reconstruct them according to their data structure. Such tight packing eliminates invalid bubbles in the packet, improving bandwidth utilization and allowing for transmitting the same set of valid events with fewer communications.

4.2.2 ***How semantics work?*** To minimize communication frequency, we propose Batch, a structure-wise mechanism that tightly packs events with different structures and supports dynamic unpacking on the software side. Batch is designed to address two main challenges: (1) how to tightly pack verification events with diverse structures and lengths; (2) how to dynamically unpack the packed events and reconstruct their original data structures.

As shown in Figure 6, the Batch workflow consists of two parts: Pack and Unpack. In the Pack stage, Batch applies a multi-level packaging strategy, which tightly packs events of different structures with metadata recording the structural information and numbers. In the Unpack stage, Batch will use the metadata to extract events of specific lengths from the packet and invoke the corresponding parser functions to recover their original structures.

Multi-level Packing Strategy. Batch leverages a 3-level strategy that exploits structural similarity to reduce packing complexity:

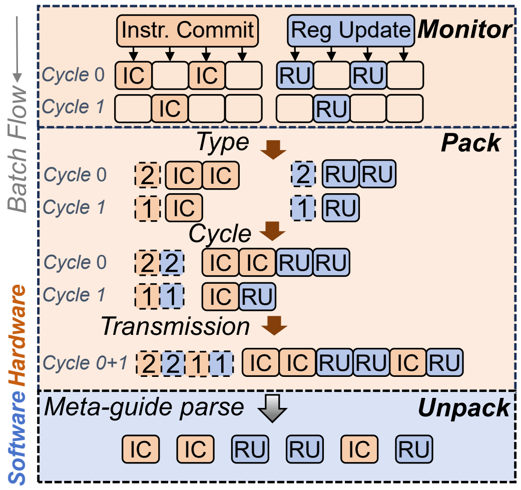

Figure 6: The Batch workflow. Verification events from different cycles are packed in three levels, accompanied by a meta recording its type and structure. The software then unpacks the events based on the meta.

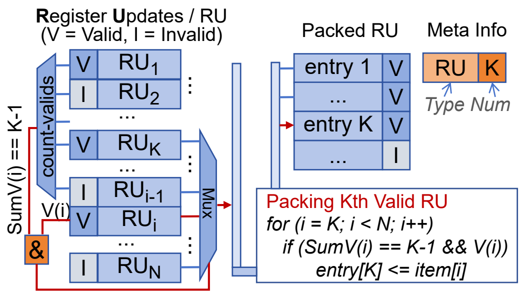

Figure 7: Type-level packaging in Batch. Packing K valid entries from N incoming semantics, and the K-th entry comes from the valid one whose prefix valids is exactly $K-1$.

(1) *Type-Level.* At the first level, Batch collects valid events from multiple same-type entries within a cycle and tightly packs them together for subsequent packing. Since events of the same type have identical structures and lengths, Batch employs a muxtree to aggregate them in parallel, where the K-th packed entry corresponds to the K-th valid incoming event of the cycle. As illustrated in Figure 7, Batch instantiates a prefix counter for each incoming event to count the number of preceding valid ones in parallel; when the count reaches $K-1$ and the current event is valid, that event is selected as the K-th packed entry. In this stage, Batch also generates metadata for each packed event, recording its structure and count to support subsequent hardware packing and software unpacking.

(2) *Cycle-Level.* At the second level, Batch further packs different types of events within the same cycle. For each type of packed event, Batch dynamically allocates a region in the packet, with its offset computed by the length sum of preceding packed events. Since the valid count of packed events varies across cycles, both the offset and the region length need to be computed dynamically according to the actual counts of valid events recorded in the metadata.

(3) *Transmission-Level.* At the third level, Batch assembles packed data from different cycles into fixed-size transmission packets, with metadata and payloads concatenated separately. Because cycle packets are variable in length, the residual space of a packet may be insufficient to hold a full cycle packet, leaving unused space and wasting bandwidth. To address this, Batch uses metadata to split a cycle packet at event-type boundaries, filling the remaining space of the current packet and placing the rest into the next, thereby reducing required communications with fewer packets.

Dynamic Unpacking with Meta. Packing events with diverse structures and lengths complicates unpacking, as the parser must both locate variable-length events within a mixed packet and reconstruct their original structures. Batch resolves this challenge through a meta-guided dynamic unpacking mechanism. Alongside each packet, Batch generates a meta that records event types, counts, and offsets, and links the meta to its parsing function. Guided by this metadata, the software parser extracts variable-length events and invokes the corresponding reconstruction functions to restore their structures. In this way, Batch enables accurate and efficient unpacking of events despite flexible, tight packing.

4.3 Squash: Fusing with Order Decoupling

4.3.1 ***Why semantics matter?*** Reducing transmission data volume calls for fusing same-type verification across instructions. For instance, a sequence of N instructions can be fused into a single N-commit event with the final PC and the instruction count. However, non-deterministic events (NDEs) challenge fusion with a strict checking order requirement. The NDEs, such as external interrupts and MMIO access, are specific to the DUT and need to be synchronized to REF at precise instructions, forcing updates to the REF's architectural states. As a result, any instructions prior to an NDE should be checked ahead while subsequent ones remain unchecked.

As illustrated in Figure 8, existing fusion approaches [8, 19, 40, 57] couple fusion to the checking order. Once an NDE is detected, they terminate the ongoing instruction fusion and transmit the fused instructions to the REF, ensuring the required checking order by consistent transmission order. The order coupling design causes frequent fusion breaks and a limited fusion ratio. In real-world workloads with substantial device interaction and frequent exceptions, such as OS boot, device drivers, and I/O-intensive applications, the fusion breaks more often, markedly reducing the overall fusion ratio.

Verification events inherently carry ***order semantics***, which reflects the required checking order between verification events. By leveraging order semantics, the fusion and communication order can be decoupled from the checking order: NDEs can be transmitted ahead with order tags, indicating after which instruction they should be checked. Meanwhile, other events continue to be fused across instructions. On the software, the checker reorders events by the tags to restore the required checking order. Such order-decoupled fusion reduces NDE-induced fusion breaks and improves fusion efficiency with less data volume.

Moreover, the verification events exhibit natural repetitiveness and locality. For example, some control and status registers (CSRs)

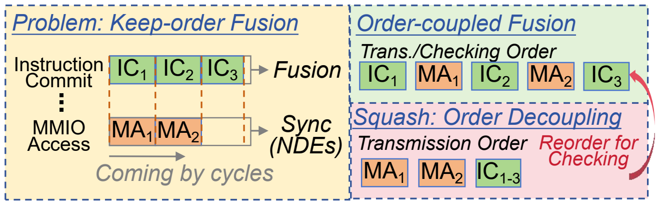

Figure 8: Comparison of fusion schemes. Order-coupled fusion breaks instruction fusion at each MMIO access (NDEs), while Squash decouples fusion from checking order, transmitting MMIO accesses ahead and then reordering them.

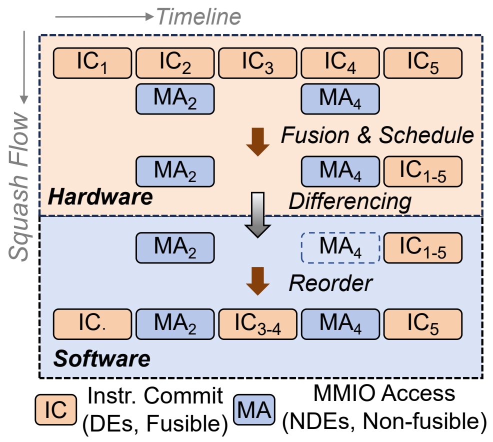

Figure 9: The Squash workflow. Verification events are fused across cycles, with the NDEs scheduled ahead for continuous fusion of DEs. Differencing removes the redundancy in MA_4 relative to MA_2. The software then completes MA_4 and restores the checking order by inserting MA back into IC.

remain unchanged over long instruction sequences. Such repetitiveness allows for referencing unchanged parts of prior events to avoid redundant transmission.

4.3.2 ***How semantics work?*** To reduce transmission data volume while preserving correct checking order, we propose Squash, an order-aware fusion scheme that fuses verification events from different instructions with decoupled checking order. Squash addresses two key challenges: (1) how to preserve checking order and enable continuous fusion without frequent breaks; (2) how to eliminate redundant event contents for further data reduction.

As illustrated in Figure 9, the Squash workflow consists of three stages: fusion and scheduling, differencing, and reordering. On the hardware side, same-type events are fused across instructions, while non-fusible NDEs are scheduled ahead with order tags. Both the fused events and NDEs are then differenced to remove redundancy. On the software side, events are completed from the preceding event and reordered to the original checking sequence.

Fusion and Scheduling. Squash fuses verification events of the same type into a single event that represents the collective effect of a sequence. For instance, a sequence of instruction commits (ICs) can be fused into one fused IC containing dense information such as the final PC, the number of committed instructions, and corresponding register updates. Meanwhile, non-fusible NDEs are scheduled for transmission ahead with order tags, notifying the checker to check them at the precise instruction order. By binding each event to its nearest preceding instruction commit, Squash preserves the checking order and ensures that the REF verifies events in the same order as the DUT, especially for non-deterministic behaviors such as external interrupts and MMIO access.

Differencing. To exploit event repetitiveness, Squash applies differencing to remove unchanged fields in each event. Events are decomposed into smaller units (e.g., CSR entries), and only modified ones are transmitted (e.g., via XOR operations). On the software side, the checker keeps the latest record and completes events by filling unchanged fields from the previous ones, as in completing MA_2 from MA_1 in Figure 9.

4.4 Replay: Debugging with Instruction-level Behaviors

4.4.1 ***Why semantics matter?*** Debugging requires localizing processor errors to specific microarchitectural components. Verification events enable this by checking the DUT's architectural state after each instruction, with each event corresponding to a specific architectural behavior and covering the relevant microarchitectural components. For example, bugs in the DUT's memory subsystem can be exposed by register updates comparison from load/store instructions, as well as the refill value check from cache accesses.

As shown in Figure 10, existing work [9, 24, 31, 56, 57] relies on verification events to detect errors, but still falls back to waveforms for debugging. To recover waveforms near the failure point, they snapshot the entire DUT and re-execute from the nearest checkpoint. Since the root cause may precede the observed failure, snapshots must be taken periodically, incurring substantial resource and time overhead.

Verification events also carry ***behavioral semantics***, referring to architectural behaviors and their mapping to specific microarchitectural components. These semantics can guide debugging by localizing faults more precisely. To address the loss of per-instruction detail caused by fusion, we reprocess only the unfused verification events around the failure, rather than re-executing the entire DUT. This restores instruction-level behavioral details and pinpoints the faulty instruction and related microarchitectural component, enabling lightweight and effective debugging.

4.4.2 ***How semantics work?*** To restore instruction-level debuggability, we propose Replay, a lightweight debugging mechanism that localizes bugs by reprocessing unfused events around the failure point. Replay addresses two key challenges: (1) how to determine the range of retransmission; (2) how to recover the REF's state for reprocessing.

As illustrated in Figure 11, the Replay workflow contains a hardware retransmission module and a software checking module. On the hardware side, verification events are buffered during fusion and retransmitted upon notification. On the software side, once a fused event mismatches, the REF reverts its state, requests retransmission, and reprocesses the unfused events for debugging.

Range Determination. Optimizations and communication latency make it difficult to identify the range of unfused events that

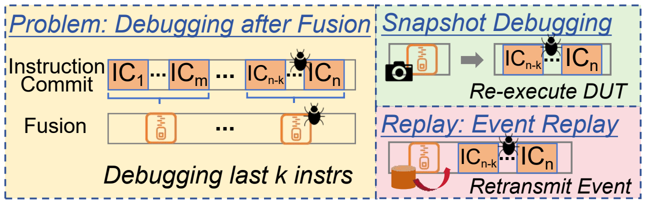

Figure 10: Comparison of debugging schemes after fusion. Snapshot debugging re-executes the entire DUT from a snapshot, while Replay retransmits buffered verification events to restore instruction-level debuggability.

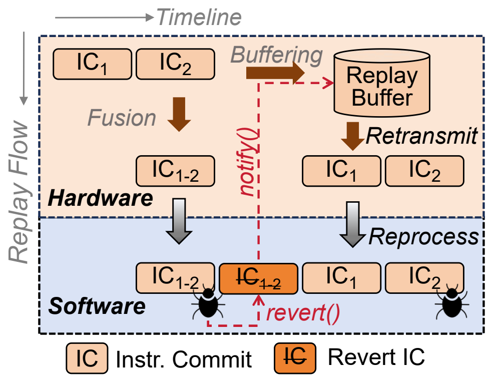

Figure 11: The Replay workflow. Verification events are buffered during fusion. When a fused event mismatches, the REF reverts it, notifies the hardware to retransmit the buffered unfused ones, and reprocesses them for debugging.

need to be replayed. Replay introduces a token-based management mechanism: tokens are assigned to buffered verification events before fusion, and fused together during optimization. Upon detecting a mismatch, Replay uses these tokens to locate the exact range of events and notifies the hardware to retransmit only the necessary buffered events. Tokens also filter out irrelevant events that may arrive between the bug occurrence and replay notification, ensuring consistent replay.

Revert Reference Model. Since mismatches may occur at any check, Replay must revert the REF to the latest checkpoint before reprocessing events. Directly snapshotting the REF at each checkpoint would be prohibitively expensive, especially in memory usage. Instead, Replay adopts a compensation-based strategy: it records only the modifications between consecutive checkpoints. When a fused event is found buggy, Replay restores the REF by rolling back these changes. For example, the original values of memory updates between two checkpoints are logged; reverting simply writes back these logs in reverse order to achieve lightweight state recovery.

4.5 Other Design Issue

Software processing latency can be hidden through hardware–software parallelism enabled by non-blocking communication. In co-simulation, all checks are assumed to match until the first mismatch occurs, so the DUT does not need to wait for software results and can speculatively continue execution in parallel with software processing, thereby hiding software latency. If the checker later detects a mismatch, it asynchronously notifies the ahead-running DUT and terminates the co-simulation.

However, non-blocking transmission introduces new challenges such as out-of-order delivery and transmission bursts. To ensure correct processing, we employ a unified hardware–software interface (Section 4.2), where all verification events are transmitted in structured packets for ordered parsing. To handle bursts, the communication unit incorporates sending and receiving queues with backpressure, ensuring reliable and balanced transmission.

4.6 Put It All Together

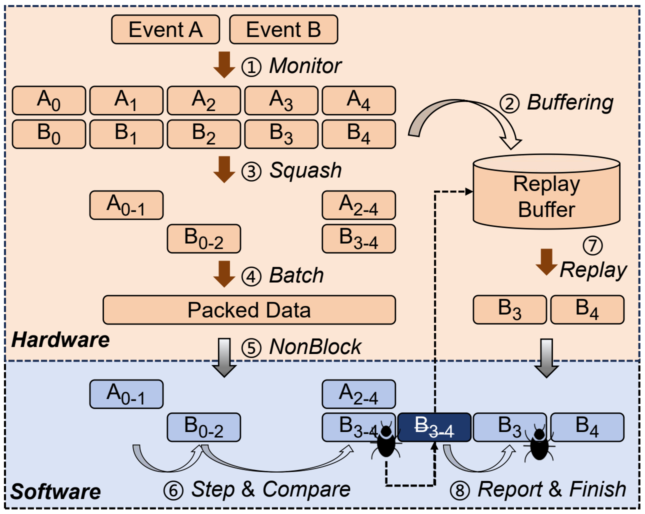

Figure 12: The DiffTest-H workflow.

Putting the acceleration and debugging units together, DiffTest-H effectively optimizes hardware–software communication overhead while preserving instruction-level debuggability. Figure 12 illustrates the workflow of DiffTest-H and the eight procedures.

On the hardware side, the verification events are captured and collected by the monitor unit inserted into DUT (see ①). The events are buffered for potential debugging purposes (see ②), and then optimized by the acceleration unit (see ③-④). Specifically, Squash performs both fusion and differencing: it fuses verification (e.g., B_3–B_4) and filters out unchanged fields between successive events (e.g., $B_{0\text{-}2}$ and $B_{3\text{-}4}$) to reduce data transmission volume (see ③). Batch further packages diverse events across multiple cycles to minimize communication frequency (see ④).

Through the non-blocking communication unit, the packed events are transmitted to the software side (see ⑤), where they are extracted with computed offset and reconstructed to their original data structure. The events are then checked step-by-step to verify the DUT against the REF (see ⑥).

Upon detecting a mismatch, the replay unit recovers the REF's state by rolling back the last faulty events (e.g., $B_{3\text{-}4}$) and notifies the hardware to retransmit the corresponding buffered data (e.g., B_3–B_4) (see ⑦). Finally, the checker reprocesses the pre-fusion events up to the specific buggy point and completes the co-simulation with a detailed debugging report, thereby preserving instruction-level debuggability (see ⑧).

5 Implementation

DiffTest-H is implemented with high-level hardware description language (HDL), Chisel. This section introduces DiffTest-H 's compatibility across platforms and designs. To optimize DiffTest-H in different scenarios, we have also proposed an open-source tuning toolkit, supporting performance evaluation, SQL analysis, and iterative debugging of DiffTest-H.

Design/Platform Compatibility. As mentioned earlier, the DiffTest-H framework mainly consists of five units besides DUT: monitor unit, acceleration unit, communication unit, check unit, and replay unit. By simply instantiate a piece of probe logic from the DUT, the DiffTest-H framework can automatically generate matching logic for all five units, providing compatibility for different designs and platforms: we have deployed DiffTest-H on both NutShell, a scalar in-order processor, and XiangShan, a 6-wide out-of-order dual-core processor, across platforms including emulator (Cadence Palladium), FPGA, and RTL Simulator (Verilator, VCS), and speeds up co-simulation with the hardware acceleration.

Tuning Toolkit. To further explore the optimization space for different designs and co-simulation strategies, we have constructed a complete open-source toolkit, which mainly includes three parts:

(1) Performance evaluation support: DiffTest-H integrates performance counters in both software and hardware. On the software side, the counters collect performance statistics, such as the transmission times and data volume. On the hardware side, the counters monitor performance-related indicators, including Squash fusion ratios and Batch packet utilization. These metrics will be used to guide the adjustment of optimization for better performance.

(2) SQL analysis support: DiffTest-H records online transmission data in an SQL database for offline analysis. With this SQL backend, DiffTest-H can also simulate order-decoupled fusion and differencing strategy on the software, thereby fully exploiting event correlations and reducing data transmission volume.

(3) Iterative debugging support: When debugging DiffTest-H's verification logic, it is time-consuming and resource-wasting to include the unchanged DUT during compilation and execution. To support independent iteration, DiffTest-H decouples the DUT and verification logic by trace dumping and reloading. The mechanism dumps the original verification events captured from the DUT during the first run, which is also called the DUT trace. Based on the traces, DiffTest-H generates and drives verification logic independently, supporting lightweight and rapid iterative debugging.

6 Evaluation

In this section, we evaluate the performance and resource utilization of DiffTest-H across various DUT scales. We further perform an optimization breakdown to quantify the contribution of each strategy within DiffTest-H. Finally, we demonstrate the effectiveness of DiffTest-H in the development of XiangShan, a 6-wide out-of-order processor. Our results are highlighted as follows:

- On Cadence Palladium, DiffTest-H achieves 80× speedup over baseline, and is 119× faster than a 16-thread Verilator simulation, reducing communication overhead by 99.8%.
- On FPGA, DiffTest-H achieves 78× speedup over baseline, and is 1945× faster than a 16-thread Verilator simulation, reducing communication overhead by 98.8%.
- DiffTest-H incurs a maximum resource overhead of 26%, reduced to 6% when disabling Batch packing.
- DiffTest-H uncovers over 151 complex bugs in XiangShan that require up to 2 months to identify with Verilator but are detected within 11 hours by DiffTest-H on Palladium.

6.1 Experimental Setup

Table 3: Experimental Setup.

Feature	Configuration
DUT	• NutShell, scalar, inorder • XiangShan (Minimal), 2-wide, out-of-order • XiangShan (Default), 6-wide, out-of-order • XiangShan (Default, dual-core), 6-wide, out-of-order
Platform	Emulator: Cadence Palladium FPGA: Xilinx VU19P
Workload	Linux boot (~1.7B instruction) KVM, XVISOR, RVV_TEST, SPEC CPU 2006

Table 4: Scales and verification coverage across DUTs.

DUT	Gates	Event Types	Avg. Bytes per Instr.
NutShell	0.6 M	6	93
XiangShan (Minimal)	39.4 M	32	692
XiangShan (Default)	57.6 M	32	1437
XiangShan (Default, 2C)	111.8 M	32	3025

To demonstrate the generalizability of DiffTest-H across DUTs and platforms, we evaluate it on both Palladium and FPGA using NutShell and XiangShan across different configurations. To further validate DiffTest-H's effectiveness under full-system workloads, we employ benchmarks including Linux boot and SPEC CPU2006, covering most verification scenarios involving control flow, register updates, memory access, hierarchy, and optional ISA extensions listed in Table 1. The experimental setup is listed in Table 3 with scales and verification coverage across DUTs listed in Table 4.

6.2 Performance Evaluation

We evaluate DiffTest-H's performance against both the state-of-the-art RTL simulator and emulation platforms under various setups. The performance results are obtained on realistic benchmarks, including Linux, KVM, XVISOR, RVV_TEST, and SPEC CPU 2006.

Figure 13 presents DiffTest-H's performance when running Linux Boot across the DUT configurations mentioned in Table 4. For comparison, we measure the performance under identical DUT and

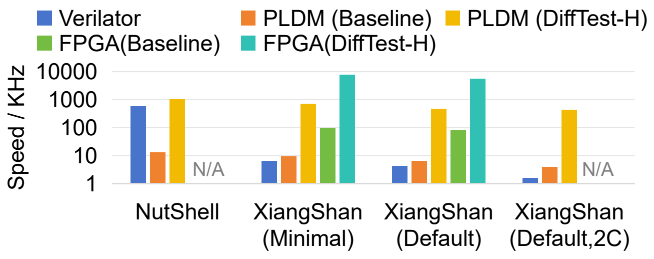

Figure 13: Performance comparison.

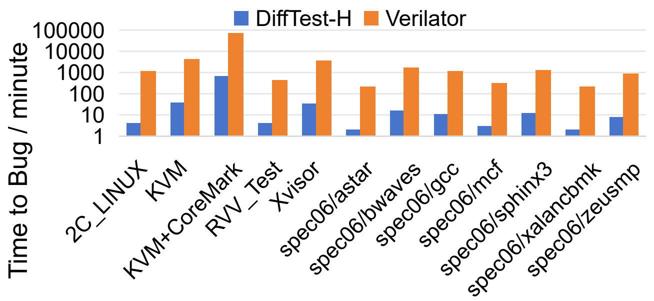

Figure 14: Bug detection time.

benchmark across the following setups: (a) 16-threads Verilator, the current state-of-the-art RTL simulator; (b) Unoptimized Palladium setup, serving as the baseline for DiffTest-H; (c) DUT-only Palladium setup, representing the theoretical maximum simulation speed without any co-simulation overhead. The performance results are quantified in kiloCycles per second (KHz).

On the large-scale DUT co-simulation, DiffTest-H demonstrates significant acceleration, achieving an 80× speedup over the unoptimized Palladium baseline and 119× faster than a 16-thread Verilator simulation. Across all DUT scales, including small and mid-sized configurations, DiffTest-H consistently delivers over 74× speedup compared to the baseline, highlighting its effectiveness across a range of design complexities.

Furthermore, DiffTest-H's acceleration capability significantly improves the efficiency of functional debugging. As illustrated in Figure 14, complex bugs that require millions to billions of simulation cycles to manifest can be detected within 11 hours on Palladium using DiffTest-H, whereas traditional simulation with Verilator would take up to 2 months under the same conditions. These bugs, uncovered during the verification of the XiangShan project, have been officially reported to and acknowledged by the XiangShan development team, demonstrating the practical effectiveness of DiffTest-H in real-world chip development scenarios.

By greatly accelerating bug discovery and iteration, DiffTest-H enables designers to quickly identify and fix bugs, enhancing the productivity and reliability of the chip development.

6.3 Optimization Breakdown

To evaluate the effectiveness of DiffTest-H's optimization techniques across different designs and platforms, Table 5 presents incremental performance results on NutShell and XiangShan with

Table 5: Optimization breakdown across DUTs and platforms.

Setup	NutShell on Palladium	XiangShan on Palladium	XiangShan on FPGA
Baseline	14 KHz	6 KHz	0.1 MHz
+Batch	102 KHz (7×)	24 KHz (4×)	1.3 MHz (13×)
+NonBlock	389 KHz (28×)	71 KHz (12×)	2.2 MHz (22×)
+Squash	1030 KHz (74×)	478 KHz (80×)	7.8 MHz (78×)

Palladium, and XiangShan with FPGA. Each row shows the benefit brought by progressively applying Batch, NonBlock, and Squash.

Batch significantly improves performance by reducing communication frequency through tight packing of structurally diverse events, achieving up to 4×–13× speedup over the baseline. NonBlock further accelerates co-simulation by masking software processing latency with hardware-software parallelism, and provides an additional 2×–4× speedup over Batch. Squash greatly reduces data volume by fusing events with a decoupled checking order, contributing the final boost to a total of 74× speedup on NutShell, 80× on XiangShan (Palladium), and 78× on XiangShan (FPGA).

Overall, these optimizations reduce co-simulation time to about 1%–2% of the unoptimized baseline, cutting communication overhead by 99.8% on Palladium and 98.8% on FPGA. This demonstrates that DiffTest-H effectively eliminates the primary performance bottleneck in hardware-accelerated co-simulation, achieving both high speed and minimal extra overhead beyond DUT emulation.

6.4 Resource Analysis

We evaluate the additional resource usage introduced by the complete DiffTest-H framework across different configurations of XiangShan. In our setup, DiffTest-H monitors XiangShan by inserting 128 probes within each core, covering 32 types of verification states. The basic resource usage for both DiffTest-H and the DUTs is summarized in Figure 15, with area results estimated using Cadence Palladium and quantified in million gates.

As shown in Figure 15, DiffTest-H incurs approximately a 6% area overhead without Batch across different DUT configurations. In this setup, DiffTest-H can operate on platforms with software-like communication support, such as Cadence Palladium, achieving accelerated co-simulation with minimal additional area cost. When Batch is enabled, the area overhead increases to an average of 25%. This configuration introduces a unified hardware-software communication interface, significantly simplifying the migration to platforms lacking software-like communication support.

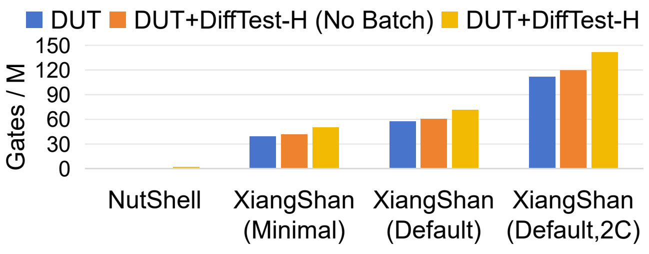

Figure 15: Resource usages.

Table 6: Summary of pull requests fixing bugs detected by DiffTest-H in XiangShan.

Bug Category	Pull Requests
Exception and interrupt handling errors	#3639, #4239, #4263, #3991, #3778, #4157
Memory hierarchy and coherence issues	#3964, #3685, #3621, #4037, #3719, #4442
Vector and control logic errors	#3876, #3965, #3690, #3643, #3646, #3664, #4361

6.5 Finding Bugs

To demonstrate the effectiveness of DiffTest-H in verification with millions of test cycles and a wide range of verification states, we deployed DiffTest-H on XiangShan, an open-source 6-wide out-of-order dual-core processor within the DiffTest framework. DiffTest-H supports 32 types of verification state, including instructions, cache coherence, TLB, vectorization, and virtualization.

DiffTest-H was extensively used during XiangShan's development to run real-world benchmarks such as SPEC06 for error detection. These workloads trigger complex microarchitectural corner cases between pipeline stages, memory systems, and exception logic, with many bugs only manifesting after millions or billions of cycles. Compared to baseline DiffTest, DiffTest-H achieved significantly shorter runtime to detect the same errors at similar cycle counts, demonstrating higher co-simulation efficiency without sacrificing debuggability.

Over the past six months, DiffTest-H helped XiangShan uncover over 151 complex bugs. All 151 complex bugs were confirmed and fixed by the XiangShan development team, involving a total of 780 lines of code modifications across 19 pull requests. These bugs span three categories: (1) exception and interrupt handling errors, such as incorrect virtual address generation, misaligned load/store wakeup, and improper interrupt responses; (2) memory hierarchy and coherence issues, including TLB deadlocks during guest page faults, StoreQueue condition mismatches, and cache inconsistencies under specific faults; (3) vector and control logic errors, such as wrong `vstart` updates, incorrect `vs.dirty` settings, and faulty vector exception tracking.

Table 6 summarizes 19 pull requests categorized by bug type, while Figure 14 presents the time savings achieved by DiffTest-H compared to Verilator in detecting these bugs.

6.6 Comparison with Prior Work

DiffTest-H supports deployment on both emulator and FPGA. As illustrated in Table 7, IBI-check [8] and SBS-check [19] represent state-of-the-art emulator-based solutions, achieving low communication overhead (~2%) and moderate area overhead (~20%). However, their verification states are limited to basic events such as instruction commits and register updates, still incapable of detecting more complex architectural behaviors such as non-determinism discussed in Section 4.3. In contrast, DiffTest-H expands the verification states to 32 architectural behaviors while reducing the communication overhead to just 0.4% with similar area overhead.

Table 7: Comparison of hardware-accelerated co-simulation frameworks.

Work	Platform	Verification States/Bytes †	Communication Overhead	Area Overhead	DUT-only Speed	Co-sim Speed
IBI-check [8]	IBM AWAN [13]	2 / 7	20 %	20 %	100 KHz	80 KHz
SBS-check [19]	Gem5 [5] (for estimation‡)	2 / 7	2 %‡	22%‡	100 KHz‡	98 KHz‡
DiffTest-H	Cadence Palladium [7]	32 / 1200	0.4 %	26%	480 KHz	**478 KHz**
Fromajo [56, 57]	FireSim [22]	7 / 24	99 %	Unknown	100 MHz	1 MHz
DiffTest-H	Xilinx VU19P	32 / 1200	84 %	24 %	50 MHz	**7.8 MHz**

† The number of verification state types and the average byte size of verification states per retired instruction before optimization.

‡ Speed and overhead of SBS-check is estimated using Gem5, with IBI-check serving as the baseline.

On FPGA-accelerated platforms, Fromajo [56, 57] is the state-of-the-art framework that runs the DUT on FireSim [22] and compares its execution against the reference model Dromajo [21]. It supports 7 types of architectural states and detects a subset of non-deterministic behaviors. In contrast, DiffTest-H, with a more comprehensive set of 32 verification states, achieves a simulation speed of 7.8 MHz, 7.8× faster than Fromajo.

Overall, compared to the state-of-the-art approaches on both emulator and FPGA, DiffTest-H delivers higher simulation speed, expanded verification coverage, and comparable area overhead.

7 Related Work

Improved RTL Simulators. RTL simulators, such as open-source Verilator [46] and commercial VCS, translate RTL circuits written in Verilog into dataflow graphs, where nodes represent combinational logic and edges represent data values. Recent optimizations for CPU-based RTL simulation mainly focus on the sequential latency of the dataflow graph, including ESSENT [4], RepCut [48], and Khronos [58]. Some other efforts in accelerating RTL simulation fully leverage the task parallelism in the dataflow graph, such as RTLFlow [27] and SAGA [47] running on GPUs, as well as Manticore [16], ASH [15], and Nexus [6] accelerated on the FPGA.

Despite these advances, dataflow-based RTL simulation executes instructions rather than circuit logic, requiring multiple host cycles for one design cycle and thus limiting speed to orders of magnitude below FPGA prototyping.

Hardware-Accelerated Co-Simulation. Differing from RTL simulators, hardware emulators synthesize RTL circuits into gates in specialized ASICs or FPGAs, reaching a speed of MHz over industrial-scale designs. Traditional emulators include Cadence Palladium, Synopsys Zebu, Siemens Veloce, and Xilinx FPGA.

Considering the ease of maintenance and running speed, existing co-simulation mainly adopts software-implemented ISA reference models such as Spike [20] and NEMU [33]. To adopt hardware emulators for accelerating co-simulation, it is necessary to consider the cross-platform communication overhead, which consumes over 98% of co-simulation time. According to the communication direction, existing works can be categorized into two groups:

Software-to-hardware Communication. ENCORE [40] runs the DUT on the emulator and the REF on the host server independently, and transmits software data to the emulator for comparison. However, as mentioned in Section 2.2, the reference model relies on the data from design for state alignment under external interrupts. Running software independently will lead to divergence in the execution path of the reference model.

Hardware-to-software Communication. Due to the large amount of hardware verification data, communication overhead accounts for more than 98% of overall co-simulation time. Recent approaches, including IBI-check [8] and ArChiVED [19], employ static data packaging and checksum-based compression to optimize communication. However, these works neglect non-deterministic behaviors in co-simulation, which is critical for aligning the reference model state with the DUT under external interrupts and stimulus. While DESSERT [24], ZP Cosim [31], and Fromajo [57] identify several key sources of non-determinism and make some optimizations toward communication, they are inefficient for handling the large-scale, diverse verification events typical of industrial designs.

8 Conclusion

We propose DiffTest-H, a semantic-aware, hardware-accelerated co-simulation framework. It enhances verification efficiency while maintaining verification completeness and instruction-level debuggability by three semantic-aware communication optimizations. Batch minimizes communication frequency by tightly packing structurally diverse verification events into a single transfer. Squash reduces data transmission volume by fusing verification events with a decoupled checking order. Replay preserves instruction-level debuggability by reprocessing the original, unfused verification events around the failure point. DiffTest-H is deployed on both emulator and FPGA to verify XiangShan [35, 50, 51, 54, 55], a 6-wide out-of-order RISC-V processor. DiffTest-H achieves a 478KHz and 7.8MHz simulation speed respectively, 80× and 78× faster than the baseline, and uncovers 151 bugs in XiangShan. We have open-sourced DiffTest-H to the community, promoting verification efficiency for broader chip designs.

Acknowledgments

This work is co-authored by Shoulin Zhang, Ziqing Zhang, and Kan Shi, with valuable support for the FPGA-based experiments. The authors would like to thank the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Natural Science Foundation of China (Grant No. 62090022, 62090023) and the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDA0320000, XDA0320300).

References

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. 1995. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures.* 95–105.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference.* 1216–1225.

[3] Scott Beamer. 2020. A case for accelerating software RTL simulation. *IEEE Micro* 40, 4 (2020), 112–119.

[4] Scott Beamer, Thomas Nijssen, Krishna Pandian, and Kyle Zhang. 2021. ESSENT: A high-performance RTL simuator. In *Workshop on Open-Source EDA Technology (WOSET), at International Conference on Computer-Aided Design (ICCAD).*

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A Wood. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[6] Peter Birch. 2022. Open source FPGA-based emulation with nexus. In *Workshop on Open-Source EDA Technology (WOSET)*, Vol. 1.

[7] Cadence. n.d.. Palladium. https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html

[8] Debapriya Chatterjee, Anatoly Koyfman, Ronny Morad, Avi Ziv, and Valeria Bertacco. 2012. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proceedings of the 49th Annual Design Automation Conference.* 955–961.

[9] Yuxiao Chen, Yisong Chang, Ke Zhang, Mingyu Chen, and Yungang Bao. 2023. REMU: Enabling Cost-Effective Checkpointing and Deterministic Replay in FPGA-based Emulation. In *2023 IEEE 41st International Conference on Computer Design (ICCD).* 21–29. doi:10.1109/ICCD58817.2023.00014

[10] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 53rd Annual Design Automation Conference.* 1–6.

[11] Ryan A Cooke and Suhaib A Fahmy. 2020. Characterizing latency overheads in the deployment of FPGA accelerators. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* IEEE, 347–352.

[12] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming.* 1–12.

[13] J. Darringer, E. Davidson, D.J. Hathaway, B. Koenemann, M. Lavin, J.K. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. 2000. EDA in IBM: past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 12 (2000), 1476–1497.

[14] Simon Davidmann and Lee Moore. 2022. Introduction to the 5 Levels of RISC-V Processor Verification. In *Design and Verification Conference and Exhibition (DVCon).*

[15] Fares Elsabbagh, Shabnam Sheikhha, Victor A Ying, Quan M Nguyen, Joel S Emer, and Daniel Sanchez. 2023. Accelerating rtl simulation with hardware-software co-design. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture.* 153–166.

[16] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R Larus. 2023. Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4.* 219–237.

[17] Harry D. Foster. 2022. Part 3: The 2022 Wilson Research Group Functional Verification Study. https://blogs.sw.siemens.com/verificationhorizons/2022/10/30/part-3-the-2022-wilson-research-group-functional-verification-study/

[18] Harry D. Foster. 2024. Wilson Research Group IC/ASIC functional verification trend report. https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/

[19] Chang-Hong Hsu, Debapriya Chatterjee, Ronny Morad, Raviv Gal, and Valeria Bertacco. 2014. ArChiVED: architectural checking via event digests for high performance validation. In *Proceedings of the Conference on Design, Automation & Test in Europe* (Dresden, Germany) *(DATE '14).* European Design and Automation Association, Leuven, BEL, Article 317, 6 pages.

[20] RISC-V International. 2025. Spike, a RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim

[21] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 667–678. doi:10.1145/3466752.3480092

[22] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 29–42.

[23] Michael Katrowitz and Lisa M Noack. 1996. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DEC chip 21164 Alpha microprocessor. In *Proceedings of the 33rd Annual Design Automation Conference.* 325–330.

[24] 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).* 76–764. doi:10.1109/FPL.2018.00021

[25] Sunwoo Kim, Jooho Wang, Youngho Seo, Sanghun Lee, Yeji Park, Sungkyung Park, and Chester Sungchung Park. 2020. Transaction-level model simulator for communication-limited accelerators. *arXiv preprint arXiv:2007.14897* (2020).

[26] Zhiwei Li, Boyan Ding, Haoyang Wu, and Tao Wang. 2017. A Flexible Frame-Oriented Host-FPGA Communication Framework for Software Defined Wireless Network. In *2017 International Conference on Networking and Network Applications (NaNA).* IEEE, 118–124.

[27] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus. In *Proceedings of the 51st International Conference on Parallel Processing.* 1–12.

[28] lowRISC. 2025. Ibex. https://ibex-core.readthedocs.io/en/latest/03_reference/verification.html

[29] S Marconi, E Conti, P Placidi, J Christiansen, and T Hemperek. 2017. IEEE Standard for Universal Verification Methodology Language Reference Manual.

[30] Romina Soledad Molina, Veronica Gil-Costa, María Liz Crespo, and Giovanni Ramponi. 2022. High-level synthesis hardware design for fpga-based accelerators: Models, methodologies, and frameworks. *IEEE Access* 10 (2022), 90429–90455.

[31] Anoop Mysore Nataraja. 2023. *A Research-Fertile Co-Emulation Framework for RISC-V Processor Verification.* Master's thesis. University of Washington.

[32] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* 327–341.

[33] OpenXiangShan. 2025. NEMU. https://github.com/OpenXiangShan/NEMU

[34] OpenXiangShan. n.d.. DiffTest. https://github.com/OpenXiangShan/difftest

[35] OpenXiangShan. n.d.. XiangShan. https://github.com/OpenXiangShan/XiangShan

[36] OSCPU. n.d.. NutShell. https://github.com/OSCPU/NutShell

[37] Lakshmanan Ponnambalam. 2017. Efficient SCE-MI Usage to Accelerate TBA Performance. In *Design, Verification & Test of Low Power and Secure Systems (DVCon).* IEEE, 2–2. https://dvcon-proceedings.org/document/efficient-sce-mi-usage-to-accelerate-tba-performance/ DVCon Proceedings Archive.

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

[39] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. 2021. iDEV: Exploring and exploiting semantic deviations in ARM instruction processing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 580–592.

[40] Kan Shi, Shuoxiang Xu, Yuhan Diao, David Boland, and Yungang Bao. 2023. ENCORE: Efficient Architecture Verification Framework with FPGA Acceleration. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23).* Association for Computing Machinery, New York, NY, USA, 209–219. doi:10.1145/3543622.3573187

[41] Simens. n.d.. Veloce. https://eda.sw.siemens.com/en-US/ic/hav/veloce-cs/

[42] Synopsys. 2025. ImperasDV. https://www.synopsys.com/verification/imperasdv.html

[43] Synopsys. n.d.. VCS. https://www.synopsys.com/verification/simulation/vcs.html

[44] Synopsys. n.d.. ZeBu. https://www.synopsys.com/verification/emulation-prototyping/emulation/zebu-200.html

[45] Bill Jason Tomas, Yingtao Jiang, and Mei Yang. 2014. Co-Emulation of Scan-Chain Based Designs Utilizing SCE-MI Infrastructure. *arXiv preprint arXiv:1409.3276* (2014).

[46] Verilator. n.d.. Verilator. https://github.com/verilator/verilator

[47] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, and Franco Fummi. 2012. SAGA: SystemC acceleration on GPU architectures. In *Proceedings of the 49th Annual Design Automation Conference.* 115–120.

[48] Haoyuan Wang and Scott Beamer. 2023. Repcut: Superlinear parallel rtl simulation with replication-aided partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* 572–585.

[49] Haoyuan Wang, Thomas Nijssen, and Scott Beamer. 2024. Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 79–93.

[50] Kaifan Wang, Jian Chen, Yinan Xu, Zihao Yu, Wei He, Dan Tang, Ninghui Sun, and Yungang Bao. 2025. XiangShan: An Open-Source Project for High-Performance RISC-V Processors Meeting Industrial-Grade Standards. *IEEE Micro* (2025).

[51] Kaifan Wang, Jian Chen, Yinan Xu, Zihao Yu, Zifei Zhang, Guokai Chen, Xuan Hu, Linjuan Zhang, Xi Chen, Wei He, Dan Tang, Ninghui Sun, and Yungang Bao. 2024. XiangShan: An Open-Source Project for High-Performance RISC-V Processors Meeting Industrial-Grade Standards. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. 1–25. doi:10.1109/HCS61935.2024.10665293

[52] Warren Weaver. 1953. Recent contributions to the mathematical theory of communication. *ETC: a review of general semantics* (1953), 261–281.

[53] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1307–1324.

[54] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199. doi:10.1109/MICRO56248.2022.00080

[55] Yi-Nan Xu, Zi-Hao Yu, Kai-Fan Wang, Hua-Qiang Wang, Jia-Wei Lin, Yue Jin, Lin-Juan Zhang, Zi-Fei Zhang, Dan Tang, Sa Wang, Kan Shi, Ning-Hui Sun, and Yun-Gang Bao. 2023. Functional Verification for Agile Processor Development: A Case for Workflow Integration. *Journal of Computer Science and Technology* 38, 4 (2023), 737–753.

[56] Jiahan Zhang, Varun Koyyalagunta, Joe Rahmeh, and Divyang Agrawal. 2023. Integrating a High-Performance Instruction Set Simulator with FireSim to Co-simulate Operating System Boots. In *First FireSim and Chipyard User/Developer Workshop at ASPLOS 2023 (ASPLOS '23 Workshops)*. https://fires.im/workshop-2023-pdf/04_integ_isa_sim_FireSim_Zhang.pdf

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, Vol. 5. International Symposium on Computer Architecture Valencia, Spain, 1–7.

[58] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Khronos: Fusing memory access for improved hardware RTL simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 180–193.

A Artifact Appendix

A.1 Abstract

DiffTest-H is an open-source, hardware-accelerated co-simulation framework for processor verification. It deploys the design under test (DUT) on Palladium or FPGA, while comparing its instruction-level architectural state with a golden reference model (REF) on the host server. The artifact includes all code and workflow of DiffTest-H to demonstrate FPGA/Palladium-based simulation speed.

A.2 Artifact check-list (meta-information)

- **Hardware:** x86-64 Ubuntu servers, Xilinx VU19P FPGA, Cadence Palladium Z1
- **Metrics:** Simulation Speed.
- **Output:** Performance report.
- **Experiments:** (1) FPGA-based simulation speed evaluation with XiangShan. (2) Palladium-based Optimization breakdown with XiangShan/NutShell.
- **How much disk space required (approximately)?:** About 128 GB.
- **How much time is needed to prepare workflow (approximately)?:** About 18 hours. (Minimal if use pre-built bitstream).
- **How much time is needed to complete experiments (approximately)?:** Less than 1 hour.
- **Publicly available?:** Yes. GitHub link: https://github.com/OpenXiangShan/xs-env/tree/micro2025-ae
- **Code licenses (if publicly available)?:** Mulan Permissive Software License, Version 2
- **Archived (provide DOI)?:** Yes. DOI link: https://doi.org/10.5281/zenodo.16637351

A.3 Description

A.3.1 How to access. DiffTest-H is open-sourced on GitHub and archived on Zenodo. For reference, we provide runtime logs and performance reports on both platforms. To reduce setup time for FPGA-based experiments, we also include pre-built bitstreams. Please refer to `README.md` for more details.

A.3.2 Hardware dependencies. Xilinx VU19P FPGA (for FPGA-based simulation), Cadence Palladium (for Palladium-based simulation), x86-64 server with 128GB RAM (host in simulation).

A.3.3 Software dependencies. Vivado 2020.2 (FPGA synthesis and implementation), Mill 0.11 (RTL generation from Chisel).

A.3.4 Data sets. Linux Boot, Microbench.

A.4 Installation

```
## Get latest artifacts from GitHub.
$ git clone -b micro2025-ae \
    https://github.com/OpenXiangShan/xs-env.git
## Install required software dependencies
$ sudo -s ./setup-tools.sh
## Init environment with submodule mechanism.
$ make init
```

A.5 Experiment workflow

For the most up-to-date and detailed instructions, please refer to `README.md`. Below is a brief workflow of the experiments.

A.5.1 FPGA-based Co-simulation Speed with XiangShan. The experiment demonstrates DiffTest-H's co-simulation speed on Xilinx VU19P FPGA as Figure 13 and Table 7. We recommend users to use the **Step 0 Quick Start**, which directly leverages our pre-built bitstream, host, and workloads for reliable results in minutes.

Recommended (Steps 0): Quick start with pre-built artifacts.

```
make write_bitstream
make write_ddr
make fpga-run
```

Fully Rebuild (Steps 1-5): From Chisel RTL generation to FPGA execution (~18 hours).

• Steps 1: Generate RTL from Chisel.

```
make fpga-rtl DUT=XiangShan
```

• Step 2: Build Host Executable Binary.

```
make fpga-host DUT=XiangShan
```

• Step 3: Generate Bitstream via Vivado.

```
make vivado    ## Setup Vivado Project
make bitstream ## Synthesis, Implementation and Bitstream
```

• Step 4: Write bitstream and workload to FPGA. (Please check README.md for more details, especially FPGA reset.)

```
# Step 4.1: Write bitstream to FPGA
make write_bitstream FPGA_BIT_HOME=...
# Step 4.2: Write workload to DDR via tcl
make write_ddr WORKLOAD=microbench
```

• Step 5: Run XiangShan Co-simulation

```
make fpga-run Host=... WORKLOAD=microbench
```

A.5.2 Palladium-based Optimization Breakdown with XiangShan/NutShell. The experiment demonstrates incremental impacts of optimization as shown in Table 5.

Step 1: Generate RTL from Chisel.

```
## DIFF_CONFIG options:
#   Z       for Baseline,
#   EBI     for Batch,
#   EBIN    for Batch+NonBlock
#   EBINSD  for Batch+NonBlock+Squash
## DUT options: XiangShan or NutShell
make sim-rtl DUT=XiangShan DIFF_CONFIG=EBINSD
```

Step 2: Compile for Palladium.

```
## Build on Palladium, requiring XCELIUM, IXCOM, VXE...
make pldm-build DUT=XiangShan
```

Step 3: Run XiangShan/NutShell Co-simulation

```
## WORKLOAD options: linux or microbench
make pldm-run DUT=XiangShan WORKLOAD=linux
```

A.6 Evaluation and expected results

Please check reference/ folder for detailed log. Below are some critical results of the experiments.

(1) Result of A.5.1: FPGA-based co-simulation speed with XiangShan as shown in Fig.13 and Table 7, detailed in reference/perf-log.

```
Core 0: HIT GOOD TRAP at pc = ...
Simulation speed: 7780.71 KHz
```

(2) Result of A.5.2: Palladium-based simulation speed with different optimization, as shown in Table 5, detailed in reference/perf-log.

```
## Speed of XiangShan-PLDM
Simulation speed: 6.49 KHz    # Baseline
Simulation speed: 23.84 KHz   # Batch
Simulation speed: 71.22 KHz   # Batch+NonBlock
Simulation speed: 478.12 KHz  # Batch+NonBlock+Squash

## Speed of NutShell-PLDM
Simulation speed: 13.67 KHz   # Baseline
Simulation speed: 101.65 KHz  # Batch
Simulation speed: 389.09 KHz  # Batch+NonBlock
Simulation speed: 1030.93 KHz # Batch+NonBlock+Squash

## Speed of XiangShan-FPGA
Simulation speed: 1278.07 KHz # Batch
Simulation speed: 2198.00 KHz # Batch+NonBlock
Simulation speed: 7780.71 KHz # Batch+NonBlock+Squash
```

A.7 Notes

DiffTest-H is developed in the open-source community and will keep updating the latest code and document. Any feedback and issues are welcome via GitHub or the author's emails. The usage and reference results of both Palladium and FPGA are included in README.md and reference/ folder. We are delighted to assist users in reproducing the experiment results.