Methodologies for Accelerated Open-Source Hardware Verification and Optimization

Farzam Gilani

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Michael Taylor, Chair
Mark Oskin
Richard Shi
Mehran Mesbahi

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

Methodologies for Accelerated Open-Source Hardware Verification and Optimization

Farzam Gilani

Chair of the Supervisory Committee:
Michael Taylor
Department of Electrical and Computer Engineering

The rise in development of open-source hardware and the demand for energy-efficient, high-performance computing have led to increasingly complex processor and accelerator designs. While open-source tools streamline the design process, verification remains challenging and costly, as it requires extensive testing to avoid costly post-silicon faults. Conventional formal verification methods are limited by scalability, slow software simulators are slow, and FPGA prototyping offer limited design visibility. To address these challenges, Condominium is introduced to couple the speed of FPGA emulation with the transparency of RTL simulation. By enabling non-intrusive emulation and cycle-accurate data collection, Condominium provides real-time instruction-level bug localization and fine-grained performance profiling, enabling agile system evaluation methods for hardware engineers. Additionally, Condominium facilitates the precise emulation of peripherals and system calls, bypassing the need for extensive RTL development. Furthermore, this dissertation introduces a novel high-fidelity hardware

coverage metric for elevating the efficacy of modern coverage-guided hardware fuzzers. By providing an accurate representation of design exploration By incorporating the relative latency information of the cascaded coverpoints into the metric, this high-fidelity metric aims to provide an accurate representation of design exploration to coverage-guided fuzzers. Through specially designed coverage engines integrated into Condominium, this work enables FPGA acceleration of high-fidelity coverage, addressing the scalability and acceleration issues of previously proposed coverage-guided fuzzers. By providing an environment for accelerated and cycle-accurate hardware emulation and the fine-grained verification methodologies it enables, this dissertation aims to provide a framework that addresses common challenges in hardware verification and analysis and significantly reduces engineering time spent on design functional verification and performance optimization.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

The PhD program is a journey of constant learning and facing challenges and I am grateful for experiencing it at Bespoke Silicon group alongside a group of excellent engineers and researchers. First and foremost, I owe my sincerest thanks to my advisor, Prof. Michael Taylor, whose mentorship, insightful feedback, and positive enthusiasm fueled my work throughout this journey. Many of the projects I participated in during the program seemed insurmountable, but Michael's guiding insights always proved valuable. I want to also thank Prof. Mark Oskin and Prof. Ajay Joshi whose collaboration on the BlackParrot project provided a unique learning experience during the first years of my PhD.

I was also fortunate to work alongside and learn from many talented students at University of Washington. I want to thank Scott Davidson, Paul Gao, Huwan Peng, Tommy Jung, Mark Wyse, and Yuan-Mao Chueh for their insights and the great learning opportunity they provided me during our collaboration on BlackParrot and other projects. I want to specially thank Daniel Petrisko and Anoop Mysore Nataraja for their indispensable collaboration on the work on Condominium and High-Fidelity Coverage, which would have been not possible without their critical contributions.

I started my PhD journey with as an international student migrant and I want to acknowledge the support I received from friends that made facing the challenges of this transition possible. I want to thank Farzam Ebrahimnejad, Keivan Alizadeh, Artin Tajdini, and Diego Peña-Colaiocco for their support and friendship. I would also like to remember my late friend Ali Saffari whose fond memory will always stay with me.

I want to thank my parents, grandparents, and my brother Parham, and recognize their unconditional love and support throughout my life. They dedicated all they had so I can reach my goals and overcome life challenges, and I would not be here without their limitless support and sacrifice. While we have been separated for many years as I moved for my studies, they kept on supporting me from abroad. I look forward to finally reunite with them in the near future as I finish this chapter in my life.

In the end, I want to express my gratitude to my spouse, Mojdeh Kashani. I was extremely lucky to meet and marry someone who stood by my side and supported me during all of life's ups and downs. Mojdeh, thank you for always being there for me and having faith in me during times of hardship and uncertainty. I know building a life together during graduate studies has been challenging, but you always proved to be a pillar of unconditional love, encouragement, and stability in our lives. I had the privilege of receiving critical support from you and your family during the final years of my PhD journey which I will always appreciate. Mojdeh, thank you for believing in me, I would have never reached where I am today without you, and for this, I love you and thank you from the bottom of heart.

Chapter 1

Introduction

The emergence of open-source hardware, alongside the growing need for more energy-efficient and high-performance computers, has resulted in an explosion of increasingly complex processor and accelerator designs. As open-source tools and hardware libraries [131, 142] continue to streamline the design process to cater to more agile chip-design practices, verification of complex designs has remained a costly and time-consuming exercise [50]. Extensive verification is specially important in hardware development because as opposed to the software domain, post-release patches are not possible and post-silicon faults can be very expensive to find and resolve. While formal verification methods can be helpful, they're not scalable and a big portion of verification effort is spent on dynamic testing. However, the slow nature of software RTL simulators and the black-box nature of FPGA prototyping complicates design verification in the intermediate stages of design development. This problem is compounded by, in many domains, lack of well established software for custom designs which forces hardware designers to also design software in parallel which can be prone to bugs, extending the verification issue to another domain. Furthermore, as opposed to software development, lack of plug-and-play IDE environments with easy-to-use interfaces and bug localization further complicates hardware evaluation. In this dissertation, we inspect the challenges facing common techniques in design analysis and propose methodologies for bypassing the limitations on underlying experimental environment and streamline the overall development process.

Condominium is designed to address these challenges in hardware verification and optimization by providing an environment that joins the accelerated emulation speed of FPGAs with the design transparency of software RTL simulation. Condominium is implemented in a Zynq environment where design is emulated on a programmable logic and controlled by the Zynq processing system that handles tasks like DUT initialization, configuration, and collection of cycle-accurate microarchitectural information from DUT which can be used for debugging, performance profiling, and other experiments. Condominium aims to achieves the latter goal in a way that's non-intrusive to the DUT emulation to enable reproducibility and migration between FPGA and simulators by employing a clock-gating mechanism that can pause DUT emulation when backpressure is needed for processing streamed microarchitectural data. Condominium enables us to significantly accelerate the process of improving hardware designs, in this case BlackParrot, and establish efficient mechanisms for ensuring functionality and performance of BlackParrot throughout its development cycle. To ensure ISA compatibility, perform accelerated functional verification, and automate bug localization, we leverage Condominium to gather run-time instruction execution information and perform ISA cosimulation by cross-comparison of the execution stream with an ISA model hosted in the host. To enable accelerated and detailed performance analysis of design over complex and long-running real-life benchmarks, we leverage Condominium to gather cycle-accurate instruction and stall source cycle attribution of processor during benchmark execution. This information enables us to identify exactly how many cycles each instruction is contributing to overall execution and provide stall type categorization for each cycle. Finally, we use Condominium to emulate the timing and functionality of various peripherals and system-calls that can be employed in conjunction with the DUT without the need to explicitly implement them in RTL.

While relying on established benchmarks for testing helps designers moderately explore the design space triggered by those benchmarks, they are not extensive and heavily rely on programming patterns choices employed by software programmers. To further evaluate the design, researchers have opted for various shades of random testing. Coverage-guided Fuzzing is a method that aims to steer a random test generator during an iterative process to maximize the overall achieved design coverage, and by proxy, maximize the newly explored design states. However, as algorithms for coverage-guided fuzzing mature and become better at exploring the design space, they cannot fully flourish their verification potential due to poorly designed coverage metrics that guide these fuzzers. Contemporary coverage metrics often do not provide a representative feedback on the degree of design exploration and are also difficult to prototype for FPGA accelerated fuzzing. In this dissertation, we introduce a high-fidelity coverage metric that aims to provide an accurate feedback for modern fuzzers. By incorporating relative RTL latency information into covergroups, high-fidelity coverage disambiguates previous metrics by establish a deterministic mapping from coverage values to activated RTL datapaths. Furthermore, by introducing specialized coverage engines, that integrate into the Condominium infrastructure, this work enables FPGA acceleration of group-coverage guided fuzzing. Finally, we inspect the efficacy of proposed high-fidelity coverage by evaluating slowdown-utilization tradeoffs on FPGAs, presenting case studies on bugs uniquely identified by the metric, and performing comparative fuzzing experiments using other metrics.

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([12, 20, 24, 31, 54, 63, 64, 65, 94, 109, 111, 112,

113, 114, 126, 127, 133, 162, 163, 171, 183, 185]), ML ([161]), ASIC Clouds ([95, 96, 103, 140, 143, 151, 153, 172]), open source hardware ([48, 142, 152]) RISC-V ([2, 42, 44, 85, 87, 101, 107, 115, 120, 121, 124, 125, 160, 184]), Network-on-Chips ([86, 97, 116, 147, 186]), security ([6, 19, 34, 35, 71, 72]), benchmark suites ([21, 98, 155]), dark silicon ([26, 63, 66, 138, 139, 144, 162]), multicore ([38, 39, 67, 68, 69, 97, 115, 137, 141, 145, 146, 147, 148, 149, 150, 164]), compiler tools ([1, 15, 56, 57, 58, 81, 82, 83, 84, 182]) and FPGAs ([21, 76, 187]).

Chapter 2

Background

2.1 Challenges in Verification and Optimization of Hardware Designs

The economic and engineering burdens imposed by modern computer-architecture projects now routinely exceed those of comparable software endeavors. Whereas software can be iterated, deployed, and patched at negligible marginal cost, hardware design demands large up-front non-recurring engineering (NRE) investments, extended verification cycles, and complex multidisciplinary coordination across RTL, physical, and manufacturing domains. Because this economic cliff exists, hardware teams front-load verification far more aggressively than software projects. Furthermore, unlike software, RTL describes thousands of state elements toggling concurrently, creating an exponential state-space that forces cycle-accurate simulators to run multiple orders of magnitude slower than real silicon, where even a one second of program time can consume a full day of wall-clock time. The result is a development process that is demonstrably more costly and time-consuming than software development. In this section we describe current methods in simulation, functional verification, and performance optimization of hardware projects and highlight the factors contributing to their long iteration times and complexity.

2.1.1 Hardware Simulation

Cycle-accurate RTL simulation is the recurring element in hardware design process, and while users rely on it for design, verification, and optimization, it routinely becomes the bottleneck affecting the process. Running single-threaded RTL simulators, such as Verilator [131], can yield and emulated clock speed of 1 KHz - 1000 KHz on modern x86 hosts [47]. On the other hand, native software can be executed on the same host at multi-gigahertz speeds. The multiple orders of magnitude slowdown from software to hardware simulation lends itself to complex nature of simulating RTL designs on a non-native software environment. The main source of this slowdown is due to the concurrent emulation of logical units on a sequential host. For every simulated RTL cycle, thousands of flip-flops and combinatorial computational units should be inspected and their output updated. To ensure the correct update order in the netlist, for every cycle the netlist graph of the RTL needs to be traversed and RTL units scheduled to be updated in the corresponding order. This has led to software RTL simulators to spend a dominant fraction of the cycle simulation time just for scheduling and ordering logic update events [23]. Furthermore, modern test-benches often include random test generation, data monitoring, waveform dumping, and DPI calls on every cycle that cross the HDL/C++ boundary and cost extra host execution cycles. While techniques like data reuse have been used to increase the performance of software RTL simulators by skipping inactive portions of hardware [110], the speedup is within one order of magnitude, and does not offer a solution for simulating real-life programs.

FPGA prototyping can be used as an alternative to accelerate RTL emulation, clocking designs at 10 MHz - 100 MHz for multiple orders of magnitude speedup over software RTL simulation. Suddenly, benchmarks that take days to simulate on software can be tested within minutes on FPGAs. This speedup, however, comes at the cost of full design visibility. This tradeoff can cause a significant hurdle when hardware design moves from early stages of reliance on software simulators to FPGA emulation. When designers move to FPGA emulation, the full design visibility that is crucial to quick debugging and evaluation becomes a challenge to be dealt with. Synthesized hardware probes [13] can be used to get a glimpse of a set of design signals, however this needs pre-planning and usually offer a limited visibility window as limited FPGA block-RAMs are used to store the data. The limited visibility forces users to decide on what set of signals to choose for monitoring and when to monitor them throughout emulation. The uncertainty cause by this limited visibility, in turn, may cost many extra iterations of trial-and-error if users lack the prior knowledge of what they are looking for, which is exacerbated by the long FPGA bitstream generation times, leading to hours of re-synthesis for locating an issue. When migrating from slow, but transparent software RTL simulators to fast, but hard to probe FPGA emulation, a framework must be developed for using FPGAs for design verification and optimization that bridges this gap in a way that designers can secure the acceleration without facing aforementioned challenges.

2.1.2 Hardware Functional Verification

As chips grow in both complexity and application variety, functional verification has grown to be both a crucial and time-consuming part of the hardware development process. A survey studying the state of ASIC design and verification industry in 2024 [50] highlights staggering statistics on the impact of verification on ASIC project timelines. According to this survey, while design engineering teams are supposed to be focused on developing new IP, they spend on average 49% of their time on functional verification. Additionally verification engineering teams spend, on average, 47% of their time on just debugging as opposing to developing new testing schemes and designing test-benches. Even with the significant man-hours invested into hardware verification, around 75% of ASIC projects report being behind schedule while only 14% of projects achieve first time silicon success. The significance of verification bottleneck, coupled with the multi-million dollar cost of re-manufacturing faulty chips highlights the need to development of agile and accurate functional verification techniques, specially in an open-source ecosystem, that can provide faster turnaround times for hardware verification.

Conventional methods of hardware verification include formal verification, unit testing, ISA compliance testing, benchmarking, and random constrained testing or fuzzing. Each of these methods can be effective in different stages of hardware development. For example, ISA compliance tests can ensure basic functional correctness in early stages, unit tests can be employed throughout the process for maintain individual logical units, and fuzzing can be used to catch unpredictable corner-case bugs as design matures and passes conventional benchmarks. Many of the aforementioned challenges facing software and FPGA emulation of the RTL also directly affect verification practices. As design verification moves away from hand-crafted short tests, software simulation suffers from long program runtime which is exacerbated when multiple re-runs are needed for discovering and resolving bugs. Due to delayed manifestation of bugs, localizing bug trigger points can be difficult even with the full transparency of software simulation. Moreover, the limited visibility window and signal bandwidth of FPGA emulation means they cannot easily be used for verification if user lacks prior knowledge on what signals to monitor and how to define a time point to capture the signal visibility window for debugging. Multiple FPGA synthesis reruns caused by misprediction of probed signals and trigger points for window capture in logic analyzers can cost days of idle time for verification engineers. Methods like ISA cosimulation [88, 89, 102, 154] are therefore needed to localize bug trigger points billions of cycles into programs, like Linux boot, saving verification engineers significant time in searching waveforms for sources of errors. Furthermore, since finding and fixing bugs usually includes running tests multiple times, methods like architectural checkpointing [16, 128] can be used to quickly reload simulation state to a snapshot of the design before the bug trigger point to test potential fixes. This checkpoint snapshot, in turn, can be used to move verification between FPGA and software simulation to leverage the transparency and speed tradeoff between the two environments. Finally, as engineers move towards fuzzing, verification challenges are compounded due to limited FPGA utilization and bandwidth for coverage collection and inaccurate coverage metrics misguiding the fuzzing algorithm and wasting testing iterations. This is specifically important because for an iterative verification process, a lower loop turnaround time and better guidance usually means higher design exploration and higher chance of revealing design bugs and vulnerabilities.

2.1.3 Hardware Performance Optimization

Hardware performance analysis usually is performed in conjunction with functional verification in later stages of development. Similarly, the same challenges with functional verification apply to performance optimization since both rely on extraction of microarchitectural information from RTL during benchmark execution. While custom designed tests alongside conventional benchmarks such as CoreMark [55], SPEC [32, 73, 74], and Linux kernel [156] can be used for stress testing different parts of hardware, simple metrics like instructions-per-clock (IPC) do not provide any indicators on the RTL and program bottlenecks contributing to overall performance. Modern processors provide built-in event-driven performance counters, usually for debugging purposes, that can be used for gaining better optimization insight [104], however, these counters are hardened to count predetermined events do not provide information on the extent that these events contribute to performance bottlenecks [166]. To obtain a fine-grained data on sources of performance bottlenecks, engineers have proposed sophisticated performance profilers that attribute each execution cycle to the corresponding offending instruction and logical stall sources [61, 62]. However, the use of these performance profilers has proven challenging in conjunction with FPGAs due the bandwidth required for streaming and processing the per-cycle attribution information. Solutions like software interrupt based performance sampling have been offered to mitigate the bandwidth issue, but they are prone to misattribution sampling errors and also perturb the normal benchmark flow during interrupt handling which can influence performance analysis. These challenges highlight the need for developing performance profiling techniques that provide fine-grained and cycle-accurate performance information without introducing sampling errors and perturbing the benchmark's execution flow.

2.2 BlackParrot

Much of the work in this dissertation was motivated by the need for development of agile functional verification and performance analysis methods during development of the BlackParrot RISC-V processor. In this section, we briefly introduce BlackParrot and go though the challenges that prompted the work on said methods.

2.2.1 BlackParrot Architecture

BlackParrot [115] is a RV64GC processor designed at Bespoke Silicon Group that aims to become the default open-source Linux-capable RISC-V multicore used by the world. It's designed as a modular and highly configurable core with well defined and latency-insensitive interfaces that enable users to easily modify various components to be tuned for their application. Use cases range from a unicore serving as a host accelerator controller, to a multicore capable of executing complex programs, like Linux, with great performance. BlackParrot also provides more configurability by ensuring race-free programmable cache coherence through distributed directory-based coherence engines [170] that enable users to implement various coherence protocols, such as MESI, through micro-programming. BlackParrot is designed as a scalable, heterogeneously tiled SoC composed of different tiles designed for compute, L2 extension, and IO and accelerator tiles interfacing with the SoC network (shown in Fig-

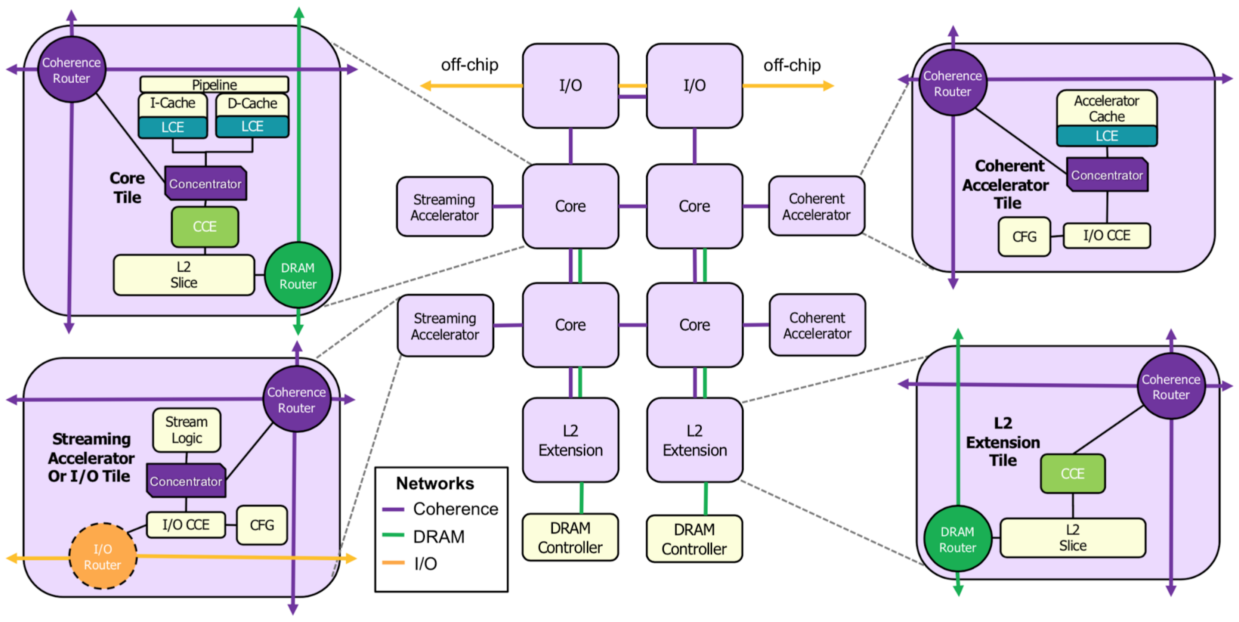


Figure 2.1: BlackParrot multi-core SoC comprise a mesh of heterogeneous tiles, allowing flexible composition of cores, accelerators, L2 cache slices, I/O, and DRAM controllers. Core tiles implement a processor, a cache coherence engine, and an L2 slice. Coherent accelerator tiles implement an accelerator that has access to the cache coherent memory system. L2 extension tiles allow the amount of L2 cache to be changed. Streaming accelerator or I/O tiles allow flexible interfacing of a common memory system via a shared non L1-cached address space that is routed over the coherence network.

ure 2.1). The BlackParrot core complex is functionally partitioned into three main sections, Front End, Back End, and Memory End, each fulfilling their specific purpose and communicating with each other through well defined interfaces.

BlackParrot Front End

The *Front End* (FE) is responsible for speculatively fetching instructions from the memory and providing the execution pipelines with stream of speculative PC-instruction pairs. To this end, FE consists of 2 major components: pc-generation and the instruction cache. The PC generation module provides speculative PCs to the instruction cache. It contains a Branch History Table (BHT), a Branch Target Buffer (BTB) and a Return Address Stack (RAS). The BHT and BTB together provide the next PC prediction on branching instructions. When the 2-cycle instruction cache returns the fetched instruction, it is partially decoded to determine whether it is a conditional branch, function call, or function return. This information is used in conjunction with the branch buffers and RAS to determine whether to use the currently predicted next PC or to override with a target calculated from the decoded instruction. The instruction cache is a Virtually-Indexed Physically-Tagged (VIPT) cache with two pipeline stages: Tag Lookup (TL) and Tag Verify (TV). There are 3 hardened memories in the instruction cache, the data, tag and stat memories which are implemented as single-port read-write synchronous RAMs to be amenable to most commercial SRAM generators. In TL, the data memory and tag memory are accessed. In TV, the data from these caches is selected based on the result of the tag comparison. A small stat mem contains the line access and replacement information for each set and is updated in TV as well. The instruction cache is non-stalling, requiring a replay on missed instructions. To enable virtual address translation , an instruction Translation Lookaside Buffer (TLB) is included as a small fully-associative buffer with a single-cycle access time. It is accessed in parallel with instruction cache and provides it with the translation physical tag for comparison. Note that the Front End does not independently modify the processor's architectural state and is logically controlled by the Back End through PC redirections caused by non-speculative resolution of branch mispredictions or trap handling.

BlackParrot Back End

The *Back End* (BE) is responsible for the non-speculative execution of RISC-V instructions. It receives a speculative instruction stream from the FE and processes them in order. To simplify physical design and reduce complexity for verification, the BE consists of a single stall point and a single commit point in the execution pipelines. The BE consists of multiple sections managing instruction execution and architectural state. The calculator unit hosts an array of non-stalling execution pipelines with varying latencies, each handling a subset of RISC-V instruction types. These include a 1-cycle integer pipe for handling simple ALU operations, a 1-cycle system pipe for handling CSR instructions, a 2/3-cycle memory pipe for handling memory operations include SV39 page-table handling, a 2-cycle auxiliary floating pipe for handling type conversions, a 4/5-cycle floating pipe for handling arithmetic operations, and a dynamic-latency pipe for handling long floating-point division and root operations. Furthermore, a hazard detector unit hosts logic for identifying data, structural, and control hazards within pipeline instructions, and stalling the pipeline when a dependency cannot be bypassed through data-forwarding. A director unit observers branch misprediction, traps, and other synchronization events and issues PC redirections to the FE to redirect the instruction fetch. Finally, a scheduler unit is tasked with receiving PC-instruction pairs from the FE, decoding them, reading register files, and dispatching them to execution pipes.

BlackParrot Memory End

The BlackParrot *Memory End* (ME) implements the Bedrock [170] protocol for providing cache coherence between processor cores, coherent accelerators, and the memory system. The Bedrock system consists of a coherence network, local cache engines (LCE), and cache coherence engines (CCE). The LCEs are cache controllers that manage coherence transactions for a single instruction or data L1 cache. The LCE interfaces on one side with the L1 cache and on the other side with the Bedrock coherence network. The LCE ensures coherence for its corresponding cache by issuing requests on a cache miss and responding to coherence update commands received from the network. The CCEs are coherence directories responsible for maintain coherence for independent subsets of memory address space. The CCE operate as programmable microcode engines executing both RISC-style general purpose operations and more complex coherence-specialized operations aimed at accelerating common cache coherence operations with most instructions executing in a single cycle. The engine is consists of a fetch stage for instruction decoding, branch prediction, and preforming redirections and an execute stage for handling all instruction execution and branch resolution. The CCE contains 8 64-bit general purpose registers, a miss status handling register to track the status of the current outstanding request, and coherence network ports for sending and receiving messages on the Bedrock network. The CCE also contains dedicated coherence directory storage, speculative memory access tracking storage, and pending bit storage.

2.2.2 BlackParrot's Bring-up Evolution

Maintaining a heterogeneous SoC such as BlackParrot with a high degree of configurability needs a robust framework for continuous verification and performance analysis so designers can quickly identify bugs and performance bottlenecks as the design matures throughout its development process. These verification methods can be integrated into the project's continuous integration (CI) [49] routines to ensure continuous design correctness with every major change in order to avoid back-tracking months into development when we face a bug. The first tier of testing used for early verification of BlackParrot has been the usage of simple ISA compliance benchmarks using RTL simulation alongside unit testing the various functional blocks of BlackParrot to ensure their functional correctness in isolation. The latter is specially effective because of the use of latency-insensitive interfaces between BlackParrot's functional units enables independent verification of various functional units without maintaining assumptions about handshake constraints with other units. However, as BlackParrot matured, the need arose for better testing methods that can handle longer and complex benchmarks and can deliver the performance and design transparency needed for more extensive verification.

The Linux kernel serves as a stepping stone benchmark for functional verification and continuous maintenance of RISC-V cores as it utilizes almost every feature in the RV64G ISA. However, moving from simple ISA compliance tests to Linux for the purposes of BlackParrot verification turned out to be non-trivial and prompted a rethinking of our early verification infrastructure. First, a mechanism was needed to reliably find program diversions from the ISA specification. This was motivated by the fact that building a working Linux environment for an early-stage processor needed the exact cooperation of many building blocks, including a custom firmware [167] to handle BlackParrot-specific bare-metal responses to kernel system-calls, so there was a need to reliably distinguish program crashes due to a faulty kernel from bugs related to fault hardware as they should be dealt with completely differently. Second, since the Linux kernel is a complex program involving long-running tasks and context switches, an RTL bug could be triggered in a certain cycle but not manifest until millions of cycles after, or maybe not at all. Finally, software debugger tools, like GDB [59], hosted on the hardware are prone to the same bugs they are tasked to identify and therefore unreliable. In order to avoid wasting verification cycles on manually resolving above challenges, we employed an ISA cosimulation method to simulate BlackParrot in lockstep with a golden RISC-V reference model [88] to instantly identify RTL divergence from acceptable ISA behavior by cross-comparing instruction execution metadata with the reference model. Furthermore, as an intermediate method to cut-down the total co-simulation time, we leverage architectural state checkpointing [88, 102] in ISA simulators to break down program execution into several segments and perform ISA cosimulation for each segment independently and in parallel on different server threads.

While parallelizing software-based ISA cosimulation provided an opportunity to integrate maintenance of long benchmarks into the BlackParrot CI pipeline, achieving an interactable and real-time experience of Linux boot motivated the migration of this verification technique to the FPGA domain. What followed was development of a Zynq-based FPGA infrastructure that enabled acceleration of BlackParrot emulation while maintaining the ability to extract cycle-accurate execution data streams from the RTL without perturbing the design. Similarly, as the need for performance analysis and optimization of BlackParrot promoted the use of industry-standard benchmarks such as SPEC [32, 73, 74], and more fined-grained performance profilers were leveraged to provide us with a cycle-accurate breakdown of BlackParrot's performance bottlenecks, we employed the same Zynq-based FPGA infrastructure to perform the duty. Moreover, the same infrastructure proved to be useful in cycle-accurate timing modeling of BlackParrot peripheral, like DRAM, and Linux system-calls executed by proxy on an x86 host controller. In chapter 3, we introduce this FPGA emulation infrastructure and how it aids users in agile design analysis. Finally, we explored using coverage-guided hardware fuzzing for better random verification and catching corner-case bugs potentially missed by conventional benchmarks. As we made improvements to the current approaches and developed a high-fidelity coverage metric for better guidance of fuzzing loop, we employed the same FPGA infrastructure to enable high-throughput streaming of coverage data which was needed but overlooked in previously introduce fuzzing methodologies. In chapter 4, we introduce this high-fidelity coverage metric and how it can be effectively employed for better hardware fuzzing.

Chapter 3

Condominium

3.1 Acknowledgment

Research in computer architecture is an intensive collaborative effort, and the work on Condominium has relied very heavily on contributions from fellow BSG members: Dan Petrisko, Anoop Mysore Nataraja, Zoe Taylor, and Prof. Michael Taylor. I would like to thank Prof. Taylor and Zoe Taylor for the physical design and assembly of the Condominium cluster, Dan Petrisko for the design of the PS-PL interfaces and the Scale-down conceptualization, and Anoop Mysore Nataraja for the design of the clock-gating mechanism.

3.2 Motivation

The ever-increasing complexity of processors and the explosion of domain-specific accelerators motivated by the end of Dennard Scaling [28] have amplified the importance and the cost of fine-grained design analysis and verification. Hardware designers are faced with a diverse plethora of processors and accelerators, each presenting unique verification challenges both in terms of functional correctness and performance evaluation that, as opposed to software design, need to be carefully analyzed and ironed out before tapeout. Early in the hardware development process, architects can leverage the transparency of RTL waveform inspection to identify hardware units contributing to performance bottlenecks and functional vulnerabilities and quickly offer and verify fixes. However, cycle-accurate software RTL simulations are painfully slow, so as the design matures throughout its development process, the need for longer testing with real-life benchmarks renders relying on RTL simulation for verification unsustainable. Moreover, when analyzing target program performance on silicon, software engineers use simple performance counters which are decided on and built-in early in the hardware design. These performance counters are sampled and aggregated to be leveraged by software performance tools, like *perf* [165], to gain performance insight for a target program. On the other hand, when aiming to optimize a target hardware for conventional software benchmarks, hardware engineers need to gain cycle-accurate insights about various hardware units contributing to performance bottlenecks. This requires design of sophisticated RTL performance profilers that tightly couple with the design and replicate its events effecting software execution pipelines. Using these RTL profilers, hardware engineers can model how an instruction creates a bottleneck in a specific design unit and measure how many cycles it costs the overall benchmark execution. Similarly, this needs an accelerated hardware emulation approach that can maintain the ability to extract a high-throughput stream of profiler data for performance analysis.

Traditionally, architects have approached this dichotomy by performing early prototyping in FPGA. By doing so, RTL very similar to tapeout designs can be emulated with cycle accuracy at 10^2-10^4x [92] faster than simulation alone. For hardware designers to experience a smooth migration of verification methodologies from software RTL simulation to FPGA emulation, the underlying environment should provide a comparable level of cycle-accurate insight into the design. Furthermore, for purposes of conducting realistic performance analysis, the FPGA environment should be able to provide standardized guaranties on the behavior and timing of various IO peripherals interacting with the design. This ensures that the many tools and techniques developed for functional verification and performance evaluation in the early stages of the hardware development process can be ported into the new environment and continue to maintain the design quality with the same level of rigorous inspection. However, prototyping large hardware systems can often be challenging due to the limited available resources on conventional development boards. Large companies can Scale-Up their prototyping systems using large commercial emulation platforms [33, 105, 134], but these are usually unaffordable to academics and startups. Others have proposed Scale-Out solutions that leverage cloud FPGA clusters [3, 8] to emulate large SoCs as a more affordable option than scale-up solutions. However, his approach generally relies on regularity in the design. Furthermore, cloud clusters usually limits engineers to using certain standardized interface and memory systems such as PCIe [130] that offer higher access latencies, multiple microseconds, and further latency jitters due to network packets being transporter over Ethernet controllers and host software packet routing [92]. This is specially important if the design emulation relies on ensuring low-latency access to DRAM and other peripherals and can cause further slowdowns in real emulation speed. Also, since cloud FPGAs use hourly pricing [7], the scale-out solutions relying on them can become non-affordable for long-term design maintenance and continuous integration.

To provide an environment that addresses the mentioned challenges, we propose Condominium, a Scale-Down approach for architectural prototyping. Instead of unilaterally scaling up a chip design from an FPGA prototype to a full tapeout, it is more economical for academics to scale-down the design to prototype design subsystems for rigorous verification and reserve the scale-out approach for full system prototyping during tapeout events. Using local FPGA boards for scale-down prototyping of design subsystems can give deep debugging insights into effects of incremental design modifications and accelerate the design

Table 3.1: On a per-FPGA basis scale-down systems require a much smaller investment, allowing teams to incrementally build up their verification infrastructure using heterogeneous boards for a variety of target subsystems. The Ultra96v2 board cost breaks even after 4-months of smallest AWS-F1 instance usage.

Strategy	**$/year/FPGA**[0]	**Logic Unit**	**Required I/O**
Scale-Up[1]	~$6700	Full Design	Native PCIe
Scale-Out[2]	~$900	Tile	PCIe Tunnel
Scale-Down[3]	**~$100**	**IP Block**	**SSH/Serial**

[0] 2000 hours is equivalent to a year of 8-hour regressions.
[1] $3.3294 per AWS f1.16xlarge hour [7].
[2] $0.4202 per AWS f1.2xlarge hour [7].
[3] $300 per Avnet Ultra96v2 board [18], with a replacement rate of once per three years. Cluster MTBF is 100+ years [178].

verification process in a more economical way that the state of-the-art alternatives. Because iteration time is much faster than monolithic prototypes, small design teams can quickly bootstrap new subsystems, run long simulations, ensure functional correctness, analyze performance consequences, and quickly iterate on potential fixes. Previous FPGA emulation platforms [33, 41, 92, 134] are expensive, dependent on vendor IP, or cumbersome and prone to lock-up since they require expensive PCIe-capable accelerators and are built on top of proprietary PCIe IP and software layers such as Xilinx XDMA [173]. Failure to interface correctly to PCIe can lockup not only the DUT but also the host server machine, requiring remote restart capabilities. In contrast, by leveraging Zynq-based FPGAs [174], Condominium can be employed using only an SSH-capable machine running Vivado and builds upon the BaseJump STL [142] library to provide generic and open bridges to commonly available AXI and UART interfaces. As opposed to using PCIe tunnels for FPGA networking, hardened AXI interfaces on Zynq systems provide direct and low-latency access to design peripherals, such as host DRAM, which can help with high-performance emulation of design that often require low-latency guaranties for on-chip memory access. Furthermore, as opposed to other emulation platforms that, for long-term design development, can easily exceed tens of thousands of dollars, Condominium provides verification teams the ability to begin with the minimal possible Total Cost of Ownership (TCO) and scale costs alongside the design progress. Table 3.1 breaks-down the cost comparison between the prototyping strategies based on recent pricing data. By employing scale-down solutions, such as Condominium, for SW/HW co-design and long-term regression testing and continuous integration, engineers can lower TCO and benefit from higher flexibility and insight into the design, while reserving scale-out for occasional bursts of full-system testing during tape-out events.

By carefully designing the platform interfaces, Condominium provides a flexible environment that can support interactions of emulation design with the outside world without sacrificing emulation accuracy. To that end, great care must be taken to mimic the environment between the full design and the subsystem. First, gaining transparency into the emulated subsystem requite run-time streaming of architectural data from RTL to the host system. However, the limited streaming bandwidth can create backpressure that can introduce non-determinism into RTL emulation, making behavior reproducibility impossible. Condominium adheres to strict non-interference of internal design timings through strategic clock-gating during unpredictable host back-pressure. This allows subsystems to execute with the illusion that they are running in situ within the full system. This technique enables the runtime streaming of cycle-accurate RTL data from DUT for the purposes of verification without perturbing the emulation flow. Second, to achieve reproducible testing and realistic performance analysis, accurate timing models should be enforced on IO interfaces with simulated peripherals. Condominium leverages the same clock-gating mechanism to enforce timing guaranties by interfacing with both in-silicon RTL timing models and software hosted abstraction models. By enabling the software or RTL driven abstraction of other parts of the system, including the timing and functional behavior of various peripherals, Condominium can be easily used to fine-tune environmental emulation parameters for better design verification. In section 3.3, we go though Condominium architecture and how it provides an accelerated emulation system without losing cycle-accuracy guaranties.

By providing an accelerated emulation environment with cycle-accurate insight and control over the DUT and its surrounding abstracted system, Condominium has proven to be an extremely powerful tool for agile functional verification and performance optimization of BlackParrot throughout its development. A deep dissection of the subsystem can allow an architect to design experiments to identify subsystem bottlenecks and quickly iterate on potential changes without requiring major microarchitectural changes in the emulation system. In this context Condominium has been used on BlackParrot for instruction-granularity verification through ISA co-simulation, cycle-accurate and time-proportional performance profiling, and accelerated high-fidelity coverage collection for entablement of coverage-guided random verification. Condominium's non-invasive, instrumented measurements and host software abstraction layers were motivated and written for use in BlackParrot, but they are generally applicable to a wide range of hardware projects. In section 3.4, we will go through these use-cases and highlight how Condominium helps architects build and maintain high-quality hardware through facilitating techniques for accurate and agile evaluation.

3.3 Condominium Architecture

In this section, we describe the Condominium architecture and how it addresses these challenges cost-effectively and with lower maintenance than previous solutions. The Condominium architecture was designed with the following goals in mind:

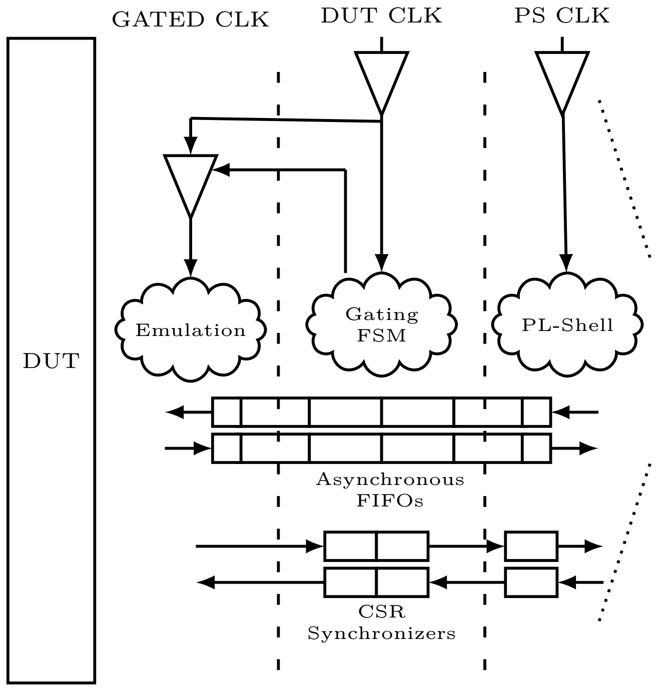
- Enable an FPGA-accelerated emulation framework for agile DUT analysis with tight emulation configuration and control mechanism.

- Provide flexible interfaces for precise interfacing between emulated subsystem logic and the abstracted subsystem peripherals.
- Provide cycle-accuracy guaranties to enable reproducibility between RTL simulator and FPGA domains.
- Build an FPGA cluster based on Condominium with remote use capability and robustness against system freezes.

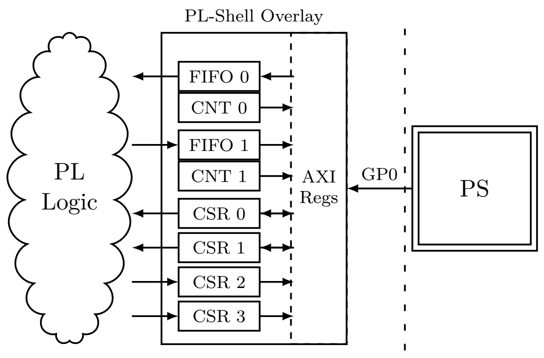
3.3.1 Zynq-based Architecture

Zynq FPGA boards couple hardened ARM cores (PS) with a programmable fabric (PL). Common peripherals such as USB, Ethernet, and DRAM are connected to the PS, while the PS communicates with the PL via hardened AXI [10] interfaces. The PS master ports are general-purpose (GP) AXI ports and cover a small address space. PS client ports are larger and higher performance (HP), allowing the PL to directly access dedicated DDR controllers ports for the PS DRAM, bypassing the ARM L2 cache coherence system and providing DRAM access without PS Linux stack interference. Condominium leverages the Zynq architecture to decompose prototyped system into these functional domains by mapping the subsystem DUT to the PL fabric, while leveraging the PS software for memory access, modeling abstracted peripherals, and emulation configuration. Bitstream generation can be done on any machine with a compatible Vivado version to the particular IP. From there, users can login to the PS over a standard Ethernet or UART connection, copy over the compiled bitstream and using the PYNQ API [175], program the overlay and DUT on the PL. Condominium provides an overlay (shown in Figure 3.1) that includes the PL-Shell, the main interface between the host emulation and the DUT user logic. The PL-shell provides a parameterizable array of input/output Control and Status Registers (CSRs), as well as an array of semi-blocking FIFOs (SB-FIFO). An SB-FIFO exposes blocking ready/valid [142] interfaces to the PL side to support latency-insensitive interfaces, while the PS interacts with a non-blocking credit/valid polling interface to prevent system lockup. While non-blocking interfaces require multiple transactions for each read and write, they generally have little overall performance impact as the PS outpaces the PL during large system prototyping.

Condominium aims to provide reproducible emulation with identical host controller code on both FPGA and software RTL simulator targets. This is motivated by the need to migrate and reproduce identified DUT issues triggered by regression runs back and forth between thee two targets to profit from their respective speed-transparency tradeoff. The common C++ host code is used for controlling the emulation flow by interfacing with PL-shell CSRs and FIFOs for tasks such as DUT and peripheral configuration, loading test programs, and processing streamed DUT microarchitectural data for verification. To interface with the PL-shell, as shown in Figure 3.2 , target-specific APIs are used to initiate AXI read and write requests to various MMIO locations in PL-shell. When using the Zynq PS as the emulation target, these APIs perform MMIO read and writes that are translated to AXI transaction over hardened AXI interfaces by the Zynq PS. When using software RTL simulators, such



(a) Condominium sub-components interface with the PL-Shell through a parameterizable set of NB-FIFOs and CSRs. The PL-Shell logic is run asynchronously to the DUT, allowing for decoupled co-emulation. Clock gating logic ensures accurate co-emulation by maintaining internal timings of the DUT.



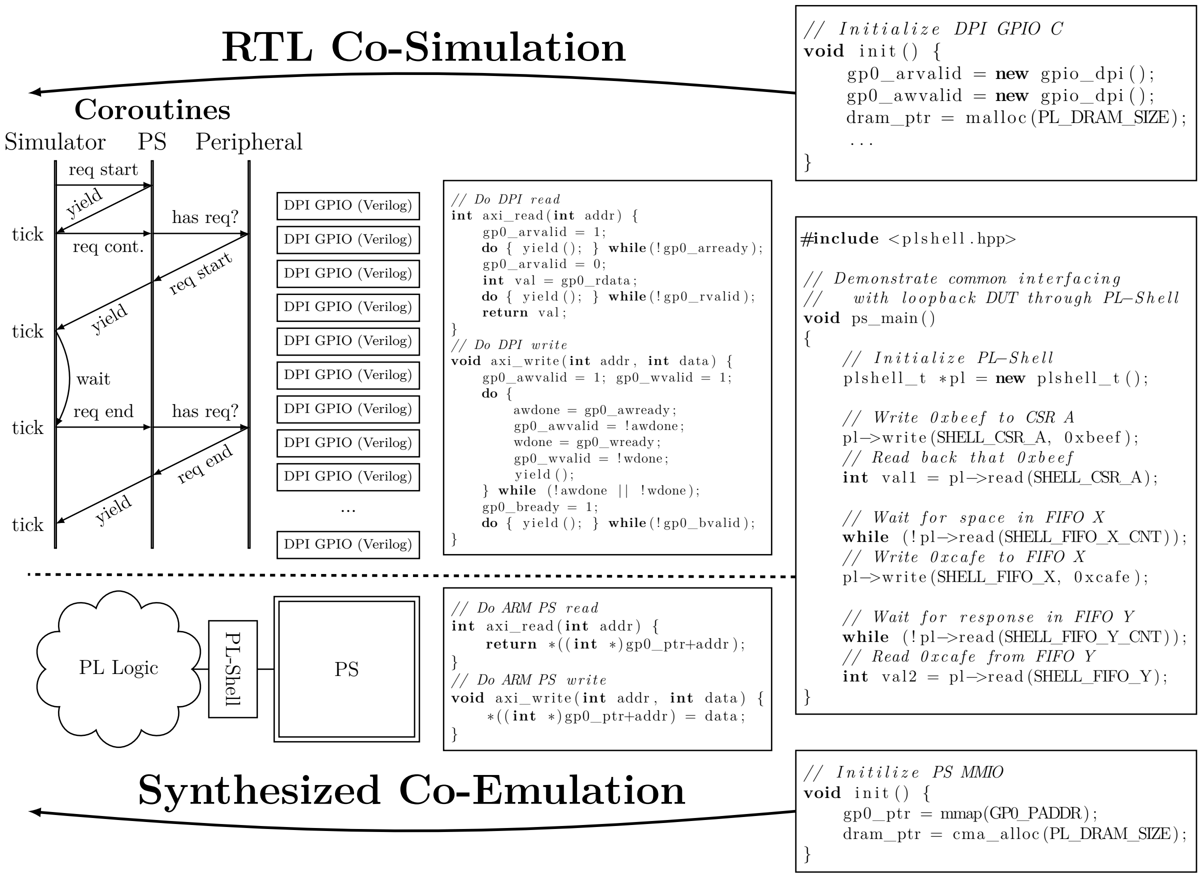
(b) As DUT logic may be buggy during design, it is essential to not hang GP0, which could lead to PS lockup. In Condominium, the PL-Shell prevents lockup regardless of DUT state, by lifting generic DUT interfaces to a set of non-blocking FIFO and read/write CSRs.

Figure 3.1: The Condominium system provides system architects with full co-emulation capabilities through simple C++ MMIO interfaces identically accessible from simulation or on deployed systems. Users parameterize the PL-Shell to for control or execution monitoring, while the PS runs necessary software functional models.

as Verilator, as the emulation target, these APIs leverage DPI-C interfaces [78] with the simulated Verilog to issue AXI transactions directly at the PL-shell ports. Also, since many simulators do not allow for native multi-threading, to handle parallel AXI transactions and design simulation, we use C++ coroutines [79] to service each transaction and to avoid deadlock in the system.

3.3.2 Cycle-Accurate Emulation Layer

When prototyping ASICs, the DUT clock is often limited by poor mapping of standard cells to FPGA primitives [169], limiting the overall emulation performance. While some efficiency may be regained by explicit manual remapping of problematic primitives, this du-



```
// Initialize DPI GPIO C
void init() {
    gp0_arvalid = new gpio_dpi();
    gp0_awvalid = new gpio_dpi();
    dram_ptr = malloc(PL_DRAM_SIZE);
    ...
}
```

```
// Do DPI read
int axi_read(int addr) {
    gp0_arvalid = 1;
    do { yield(); } while(!gp0_arready);
    gp0_arvalid = 0;
    int val = gp0_rdata;
    do { yield(); } while(!gp0_rvalid);
    return val;
}
// Do DPI write
void axi_write(int addr, int data) {
    gp0_awvalid = 1; gp0_wvalid = 1;
    do {
        awdone = gp0_awready;
        gp0_awvalid = !awdone;
        wdone = gp0_wready;
        gp0_wvalid = !wdone;
        yield();
    } while (!awdone || !wdone);
    gp0_bready = 1;
    do { yield(); } while(!gp0_bvalid);
}
```

```
#include <plshell.hpp>

// Demonstrate common interfacing
//   with loopback DUT through PL-Shell
void ps_main()
{
    // Initialize PL-Shell
    plshell_t *pl = new plshell_t();

    // Write 0xbeef to CSR A
    pl->write(SHELL_CSR_A, 0xbeef);
    // Read back that 0xbeef
    int val1 = pl->read(SHELL_CSR_A);

    // Wait for space in FIFO X
    while (!pl->read(SHELL_FIFO_X_CNT));
    // Write 0xcafe to FIFO X
    pl->write(SHELL_FIFO_X, 0xcafe);

    // Wait for response in FIFO Y
    while (!pl->read(SHELL_FIFO_Y_CNT));
    // Read 0xcafe from FIFO Y
    int val2 = pl->read(SHELL_FIFO_Y);
}
```

```
// Do ARM PS read
int axi_read(int addr) {
    return *((int *)gp0_ptr+addr);
}
// Do ARM PS write
void axi_write(int addr, int data) {
    *((int *)gp0_ptr+addr) = data;
}
```

```
// Initilize PS MMIO
void init() {
    gp0_ptr = mmap(GP0_PADDR);
    dram_ptr = cma_alloc(PL_DRAM_SIZE);
}
```

Figure 3.2: Condominium enables designs to run identical C++ code on the PS of a Zynq ARM core, over a UART bridge, or in vendor-agnostic simulation. Instead of relying on Verilog tasks to interact with the DUT, Condominium exposes pins on the PL-Shell through a DPI-C interface. The result is fine-grained control over DUT execution, enabling software flow-control and thorough verification. As multi-threading is disallowed by many commercial simulators, C++ co-routines are used to co-simulate the DUT with blocking transactions such as AXI requests, providing parallelism and deadlock avoidance.

plicates design efforts and forces dependencies between FPGA and ASIC teams. To alleviate performance bottlenecks, as shown in Figure 3.1, we decouple the DUT and emulation logic clock domains with asynchronous FIFOs and CSR synchronizers bridging the PL emulation fabric and DUT domains. This decoupling allows the PS to interface with an independently PL emulation fabric clocked, usually at a higher frequency, to run ahead of the prototyped design and prevent it from constraining the emulation layer.

When extracting cycle-accurate data from DUT or enforcing virtualized peripheral timing models, the PS may need to process information while also handling context switching, network bridging, or other asynchronous processing while DUT is going though new execution cycles and generating new emulation metadata. If an asynchronous FIFO fills up and the PS is not ready to accept a new packet from the PL, either the FIFO must backpressure the DUT, such that cycle-accuracy is lost, or the packet is dropped. Most systems using latency-insensitive I/O constructs use ready/valid handshakes to backpressure the DUT operation when buffers are full. However, doing so perturbs the system and eliminates cycle-accuracy, making the emulation non-reproducible due to the unpredictable nature of emulation system's data processing bandwidth and the resulting backpressure on the DUT.

Another approach to avoid degradation is to run a Real-Time Operating System (RTOS) on the PS. However hard real-time guarantees are difficult to meet, restricting maximum performance; and proofs would need to be rewritten for each DUT interface, slowing iteration time. During performance profiling, for example, the information bandwidth needed varies dramatically based on the specific performance aspect being monitored, so these real-time guarantees will need to be re-tuned for each different experiment as well as for running in simulation modes that have dramatically different wall-clock timing characteristics. In addition, compiling arbitrary programs is much more difficult on a specialized RTOS compared to a full POSIX operating system.

Condominium leverages the PS-DUT asynchrony to implement cycle-accurate emulation by gating the DUT clock upon interfering backpressure. Once gated, the asynchronous FIFOs are drained and execution can safely resume. This approach masks non-determinism in the PS, which may be running a full PetaLinux [176] operating system. Clock-gating in the PL-shell means that both PS software and DUT logic can be completely unaware of the other side of the interface, operating in an ideal environment. This ensures gathering and processing of fine-grained, cycle-accurate DUT data in PS can enable various techniques ranging from instruction-level cycle attribution for performance bottleneck analysis to ISA cosimulation for agile bug localization. Also, clearly defined boundaries between PS and DUT domains simplify necessary timing constraints during synthesis and standardized, validated asynchronous primitives for clock domain crossing shield users from the unwanted issues of multi-clock systems.

Furthermore, modeling exact IO timing is an essential functionality in a scale-down system.

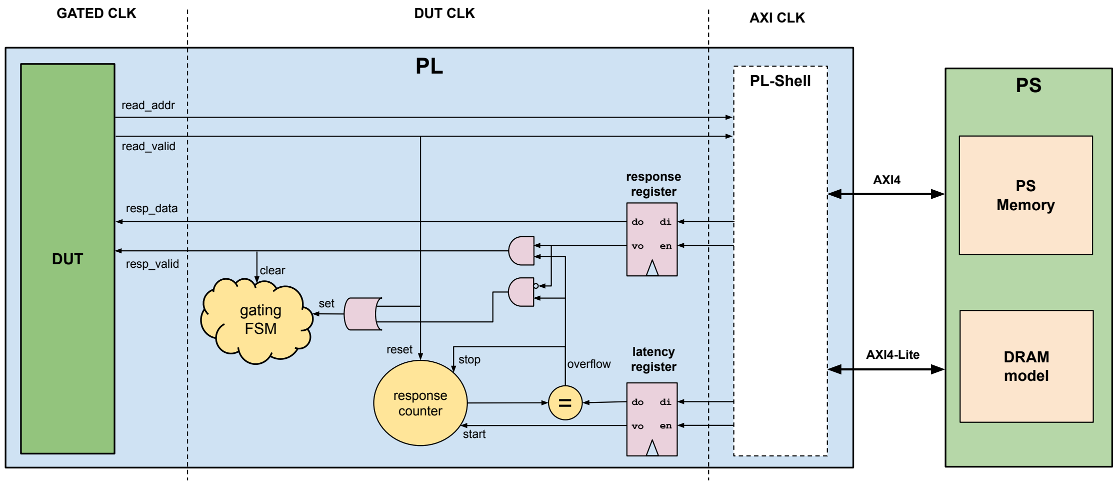


Figure 3.3: Condominium leverages the clock-gating to enforce exact timing behavior for abstracted peripherals. In the case of modeling DRAM access latency, a Condominium pauses DUT emulation on a memory request, PS receives the request and programs a PL register with the desired latency which resumes the emulation. If the response packet from the ARM memory arrives early, its withheld and release to DUT at the desired latency. If the response is late, emulation is paused until packet is received from the ARM memory.

In Condominium, hardware model timers exist in the DUT clock domain, but timing information is stored in the PS program where it is exchanged via a simple handshake. For instance, to prototype a system with cutting-edge HBM DRAM, the DUT may emit a DRAM request which causes DUT execution to stop. As show in Figure 3.3, the PS receives the request, calculates the predicted timing of the specific HBM model, and programs the expected timing through PL-shell CSRs. The DUT clock then resumes waiting for the DRAM request to return by executing any other parallel tasks. If the DRAM request returns before the hardware model timer finishes, it will be held until the correct cycle and then released to the DUT. If the hardware model timer expires before the DRAM request returns, the DUT will return to a gated state. This event-driven co-emulation maintains cycle-accuracy while ensuring there are minimal wasted cycles. Alternatively, for peripherals with simpler timing model, designers can opt to implement the model on the PL in RTL by incorporating it into the existing clock-gating state machine.

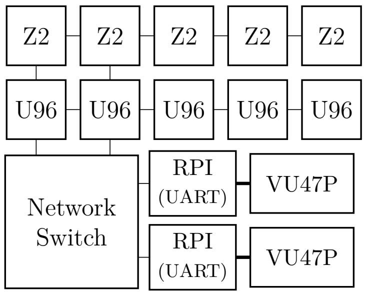
3.3.3 Heterogeneous Prototyping Cluster

Condominium was initially developed for FPGA prototyping of BlackParrot and also for computer architecture students performing full-stack analysis and optimizations on it. The initial vision was a cluster of heterogeneous FPGA development boards aimed at fulfilling the following goals:

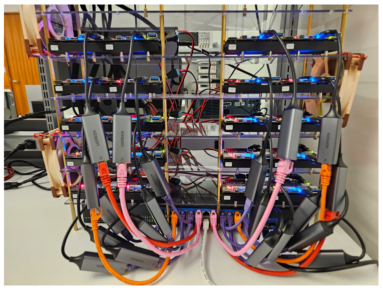
- Provide a cheap, flexible platform for designing and analyzing microarchitectural modifications to the core.
- Avoid supporting all laptop to FPGA mappings by standardizing the host Zynq PS system.
- Synchronize the software simulator and Zynq target environments to enable domain migration and minimize FPGA debugging.
- Provide a platform robust to fatal RTL bugs by construction, prevent the host system from hanging, and enabling remote reboot of the cluster.

The first iteration of Condominium was built on PYNQ-Z2 [158], inexpensive educational boards available at an academic discount. Z2 boards are out-of-box compatible with the open-source Xilinx Pynq [175] SDK, providing a Python-based interface for bitstream programming, peripheral management and PS configuration, among many other convenience features. Because the Pynq software makes interaction with the Z2 boards so convenient, students can buy and develop on their own device. Needing a more structured approach to coherently integrate a large number of boards, we designed a scalable cluster of various network-attached FPGAs all based on the Condominium environment. All prototyping and emulation data is stored on a host system and shared with the boards through a network-based distributed file-system. This ensures the occasionally large evaluation data files extracted by the PS for future analysis are not limited by the board memory and can easily be accessed, aggregated, and processed by the host. For parallel development, team members log into each board to independently program and run experiments. Bulk regression can be run from standard job-scheduling software. The setup shown in Figure 3.4(b) is built with commodity components: USB-Ethernet controllers, a network switch, and hand-cut plexiglass shelves. Comprising 20 Ultra96v2 [18] boards, this cluster cost around $4500 and supports multiple projects and CI pipelines for a modestly sized research group. Based on Table 3.1, we estimate that this cluster outperforms in TCO after less than a full year of usage.

Maintaining a heterogeneous FPGA cluster is typically as simple as ensuring the central network switch is remotely accessible. Moreover, the PetaLinux configuration on Zynq boards allows for watchdog timers [168] which forces a reset upon hanging the board. To enable automatic watchdog reboot on PYNQ boards, many segments from the hardware to the software stack need to be modified [46]. First, the Zynq PS needs to be configured to have hardware watchdog timers ticking and be able to reset the system. Second, the PYNQ system device tree must be modified to recognize the watchdog timer at the correct PS address. Third, the linux kernel and boot-loader should be configured to enable software support for the watchdog timer. Finally, a watchdog daemon should be created to periodically reset the watchdog timer from user-sapce and prevent overflow. If everything is properly configured, any temporary glitch with the board or a system freeze caused by inexperienced users has



(a) Condominium clusters connect to a standard network switch to enable remote connections. While homogeneous clusters of Pynq boards is the lowest maintenance options, some labs may be restricted to non-Zynq FPGAs and use small controllers such as Raspberry Pi to bridge to a PS interface.



(b) A twenty-server Ultra96v2 cluster. Students can time-share boards for parallel builds and serialized, private experiments. By connecting the cluster to a network switch, students are able to work fully remotely, important during events such as the COVID-19 pandemic.

Figure 3.4: A heterogeneous cluster can be configured for a variety of Pareto frontiers along cost, capacity and design parallelism. For design space exploration of heterogeneous components, a fleet of small FPGAs may minimize build times, whereas for a suite of long-running benchmarks, medium-sized FPGAs may be able to complete overnight regressions on a full system.

a fail-safe backup and connections can generally be restored after board is automatically rebooted by a watchdog overflow. For further robustness, a remote network-attached reset switch (or Raspberry Pi [51]) removes the need to physically reset the system even upon unlikely watchdog failures. A centralized job-scheduler can dynamically prevent interference between users and regression jobs. As shown in Figure 3.4(a), Condominium easily supports heterogeneous boards as there are only two classes of network interface to maintain. Standard Zynq boards connect directly via Ethernet while non-Zynq parts tunnel through a network-attached UART-capable device such as a Raspberry Pi. In this way, clusters can simultaneously service a wide range of IP blocks that each may leverage specific board features. A diverse setup is ideal for a large continuous integration server as generic jobs can be assigned to minimally-sized boards, reducing regression time and improving energy efficiency.

3.4 Condominium Usage

As mentioned before, Condominium was initially developed for FPGA prototyping of BlackParrot. However, it soon became clear that the ability to perform accelerated emulation of a design and while extracting cycle-accurate microarchitectural information without perturbing the emulation flow and losing reproducibility opens more doors to develop methods for agile hardware development. In this section we go through how Condominium has been used for enabling agile and accurate functional verification and performance analysis techniques.

3.4.1 Accelerated Functional Verification

One of the important ways Condominium has been leveraged is to accelerate methods developed for agile hardware functional verification. In this section we go explain how ISA cosimulation can be used to significantly reduce debugging time as hardware designs mature and migrate to longer testing regressions. We explain the limitations of the technique, how it can be parallelized while still relying on software RTL simulators, and how Condominium can be used to accelerate the process without losing debugging transparency.

ISA Cosimulation for Verification

To ensure the correctness of hardware functionality whether when adding new features during development or experimenting as part of an academic project, the user needs to be able to verify the correct execution of a benchmark during its runtime. They also need to be able to identify silent benchmark fails and quickly localize the bugs to the instructions that triggered them to effectively reproduce and fix the bug. Additionally the offending instruction sequence can be added to added into the design's verification regression suite and integrated into the CI pipeline to avoid further resurfacing of the same issue. During the initial stages of hardware design, since engineers rely on short and targeted unit tests, potential bugs can usually be identified by browsing the RTL waveform. However, as the benchmarks and the RTL mature, the previous verification methodology fails to keep up with the increase in simulation times and complexity. When simulating benchmarks, such as the Linux kernel, hidden bugs can manifest millions of cycles after they have originally been triggered and to find their source using waveforms and simple execution tracing can prove to be inefficient and time-consuming. This is compounded by the fact that, due to the slow nature of RTL simulators, rerunning the benchmarks to verify a potential fix is also time-consuming. Software debuggers, like GDB [59], can offer tools for bug-localization, but will cause even more slowdown in simulation, may need special hardware support, and if hosted on the DUT itself, can be prone to the same bugs that they are aiming to reveal, making them unreliable for hardware verification.

To enable bug-localization for BlackParrot verification, we implemented a system of RTL co-simulation where we utilize a RISC-V ISA golden model, Dromajo [88], for runtime verification of the RISC-V executed instructions on BlackParrot. Integrated into the BlackParrot Verilator test-bench, on every instruction commit or taken trap, a co-simulation module passes the execution information through DPI-C interfaces to the C++ ISA model. The ISA model executes the benchmark in lock-step with DUT and cross-compares execution information, such as PC, instruction, register writeback data, and RISC-V status CSRs, with the DUT. By run-time cross-comparison of program execution data, we can immediately catch hidden bugs on a possible RTL and ISA model divergence and inspect the waveform

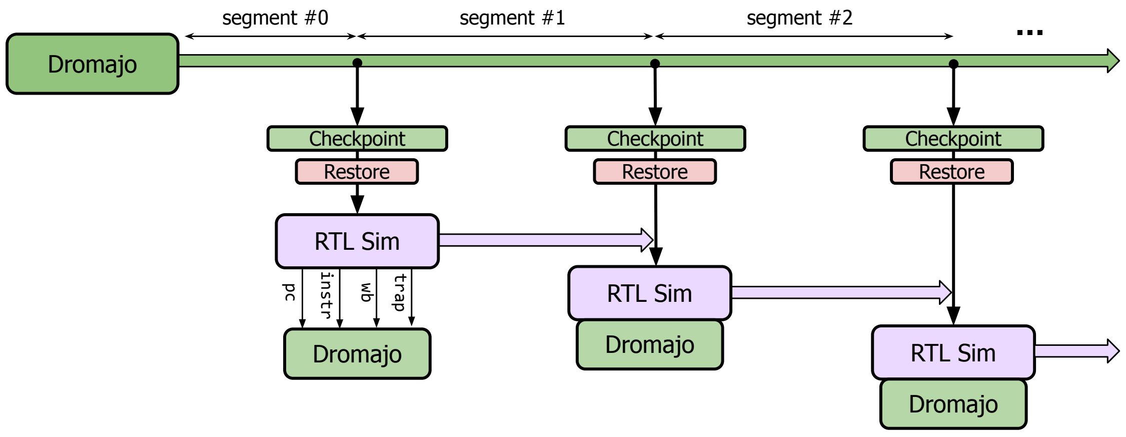


Figure 3.5: By generating periodic checkpoints using an ISA model, RTL co-simulation can be broken down into independent segments and parallelized after restoring the checkpoint state in the simulator.

at the triggered clock cycle for further insights on how to reproduce and fix the issue. Furthermore, the instruction sequences manifesting these bugs can be compiled and added to the design's verification suite to ensure such bugs do not resurface.

While ISA cosimulation largely automates the bug localization problem and significantly reduces debugging cycles by eliminating the need to browse long waveforms for RTL faults, it faces challenges and limitation that affect its efficacy as a functional verification method. First, it is only as accurate as the granularity of RTL microarchitecture modeled in the ISA simulator. This means it cannot certain issues such as RTL hangups and bugs that corrupt a microarchitectural unit, like branch prediction, but does not manifest in the RISC-V architectural state. Furthermore, timing race events between cores and accelerators are usually not modeled in ISA models, meaning reference models, like Dromajo, should defer to the DUT on the correct resolution for issues like atomic operation failures in a multicore. Also, while ISA cosimulation cuts down on manual debugging time, it is still bounded to the slow nature of software RTL simulators.

Accelerated ISA Cosimulation

To cut down the cosimulation time for longer benchmarks, the program execution can be broken down into multiple segments for further parallelization. As shown in Figure 3.5, we use the ISA model simulator to run the benchmark and create periodic checkpoints of the processor's architectural state space. The state includes the contents of memory, register-files, CSRs, and PC. Then, for each state checkpoint, an independent RTL co-simulation thread can be invoked by first loading in the architectural state into the processor by loading in the memory image, booting into a generated initialization boot-rom that sets up registers and CSRs, and returning to the checkpoint PC to resume cosimulation for the duration of that segment. By eliminating the backward dependencies for the program segments, we can run parallel independent cosimulation threads for each segment. This method reduces the cosimulation time for longer benchmarks while still maintaining the transparency of RTL simulation.

While parallelizing the ISA cosimulation can reduce benchmark verification by one order of magnitude, it requires a compute-heavy server with a high multi-processing capability. Since Condominium's infrastructure can be used to extract fine-grained instruction commit information, we can use it to accelerate ISA cosimulation. By hosting the reference model on the PS, we can feed and extract the corresponding execution information to the SB-FIFOs and perform cross-comparisons to catch program divergence in the DUT. The execution information can be passed through asynchronous FIFOs that can gate DUT clock before being streamed to PS by the PL-shell. This ensures cycle-accuracy is maintained and we can reproduce a caught bug on RTL simulation for further debugging. Furthermore, once the bug convergence point is identified, a prior checkpoint can be created by the ISA model to facilitate the bug reproduction in RTL simulation. Condominium has been used to accelerate Dromajo cosimulation of BlackParrot by an average of 33.8x on Ultra96v2 boards compared to Verilator [108]. Using this feature, Condominium can be integrated into the CI as part of the chip development cycle, and on a major change to the design, an FPGA cluster can be used to quickly run a diverse and evolving set of regression benchmarks for design verification.

3.4.2 Cycle-Accurate Performance Profiling

While Condominium accelerates design emulation, performance analysis and optimization at a scale-down granularity additionally requires deep introspection. Processors rely on built-in performance counters and metrics such as IPC to perform aggregated performance evaluation of running a benchmark. However, these methods offer limited insight into the various sources of inefficiencies in the hardware, and outside of analyzing the efficacy of solutions to previously known bottlenecks through A-B testing, provide no intuitive way of attributing elapsed clock cycles to their corresponding functional unit and program instruction.

Recent work on time-proportional performance profiling [61, 62] have proposed an RTL golden reference profiler, Oracle, that by tightly coupling with the core pipeline, can attribute every simulated clock cycle to the exact instruction that the processors is either committing or is stalling the execution. To correctly identify the instruction that's responsible for a delayed cycle of execution, Oracle employs a dynamic attribution strategy that improves on static methods such as last-committing-instruction (LCI) and next-committing-instruction (NCI). This dynamic attribution strategy (Figure 3.6) is based on the insight that the commit stage

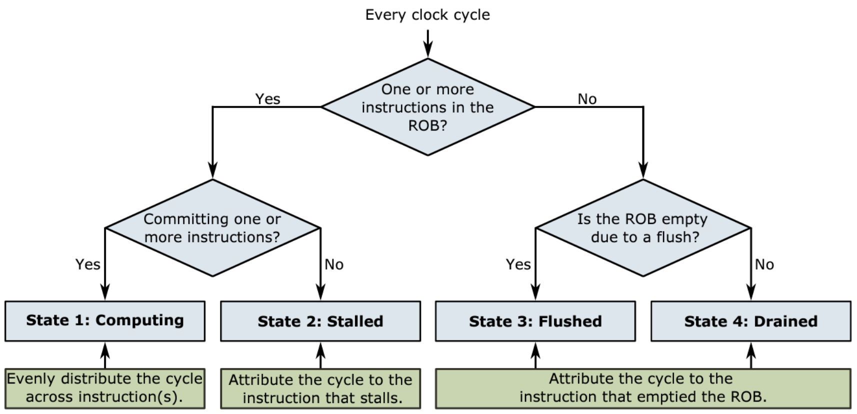

Every clock cycle
One or more instructions in the ROB?
Yes
No
Committing one or more instructions?
Is the ROB empty due to a flush?
Yes
No
Yes
No
State 1: Computing
State 2: Stalled
State 3: Flushed
State 4: Drained
Evenly distribute the cycle across instruction(s).
Attribute the cycle to the instruction that stalls.
Attribute the cycle to the instruction that emptied the ROB.

Figure 3.6: Oracle profiler clock cycle attribution overview. Each cycle is attributed to an instruction based on the state of the processors reorder buffer (*ROB*). On an empty ROB, the cycle is attributed to the instruction that flushed the ROB or is waiting to be scheduled. Otherwise, the cycle is attributed to the instruction being committed or stalling the pipeline. Figure from TIP [62].

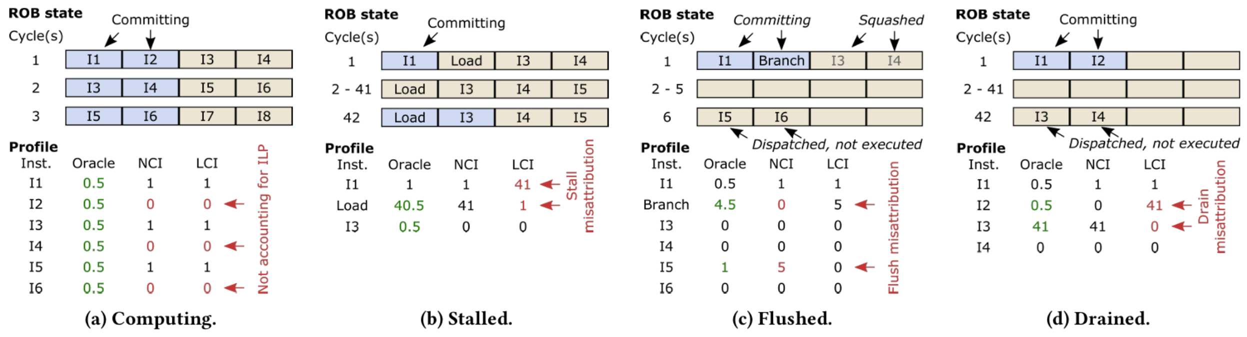


Figure 3.7: Example illustrating the Oracle, NCI, and LCI cycle-attribution on a 2-wide out-of-order processor. Rather than attributing execution cycles staticly, by dynamically identifying the offending instruction based on the state of processor pipeline, Oracle provides realistic insight of performance bottlenecks within the program. Figure from TIP [62].

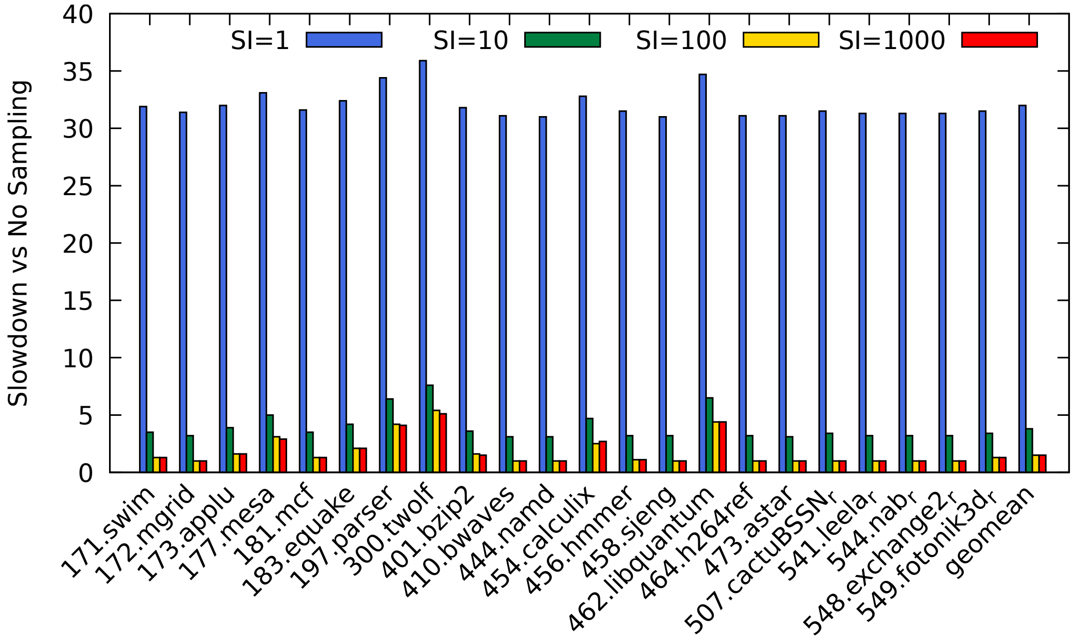


Figure 3.8: Condominium can be leveraged to dynamically switch co-emulation speed for sample rate. As sampling granularity decreases down to single step, there is a 32x slowdown. Therefore, best practice is to identify regions-of-interest and change sampling frequency to match importance.

of the processor pipeline can be in one of the four possible states in each cycle, and based on that state a different instruction is attributed to the cycle. Static attribution strategies, as shown in Figure 3.7, are biased towards current or previously committed instructions that leads to over-representing and under-representing instruction performance impact based on the current state of the processor pipeline. This cycle attribution is further broken down and categorized into various stall reasons that can cause a delay in instruction execution. By instrumenting the RTL with a time-proportional profiler, like Oracle, users can accurately identify how many cycles the processor spends on each instruction and how those cycles breakdown in terms of different sources of stalling in execution. While a golden profiler can provides accurate (PC, stall) pairs for each cycle that can be aggregated into instruction cycle stacks, traditional FPGA prototyping lacks the bandwidth, backpressure, and non-intrusion guaranties to maintain the cycle-accuracy needed to store and process these pairs. In this section, we explore how to leverage Condominium's sampling infrastructure to characterize BlackParrot through time-proportional performance profiling.

Condominium users can instantiate an Oracle-like performance profiler tightly coupled with

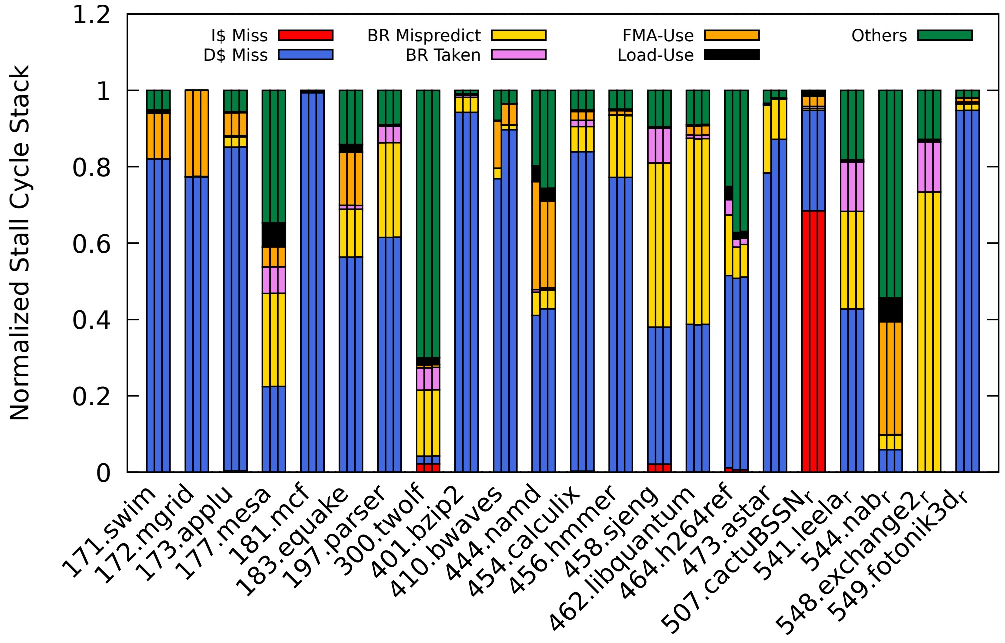


Figure 3.9: Benchmark aggregated stall stacks for sampling intervals 1, 10, and 100. Due to time-proportionality, stall stacks do not generally vary across sampling intervals. However, a few benchmarks such as 454.calculix and 464.h264ref have variances as high as 6.2%. Oracular sampling through Condominium is able to accurately identify these stall sources.

the core pipeline by pulling control wires from their corresponding execution pipe through hierarchical references. This profiler annotates each cycle of execution with a PC and event classification (stall type or commit), attributing at the commit stage to maintain time-proportionality. Then Condominium can be used to extract this information with different levels of granularity. If the user is simply interested in collecting aggregate data on the breakdown of stall classification for the entire benchmark, counters can be instantiated for each stall type and then connected to PL-Shell CSRs to be read on benchmark completion. At this level of granularity, the user can obtain a cycle-accurate breakdown of how different sources are contributing to the overall benchmark performance without any slowdown but cannot attribute those cycles to specific instructions. This limits the capability of designers to identify how specific instruction sequences can create performance bottlenecks in the core and how to optimize for them.

Previous works like TEA [61] and TIP [62] have demonstrated the benefits of time proportional instruction profiling, but concluded that the bandwidth overhead is impractical. To circumvent this, they have resorted to a coarse-grained sampling strategy for FPGA prototyping that relies on software interrupts to sample attributed (PC, stall) pairs from the profiler and extrapolate the results based on the sampling frequency(4kHz in [62] to match Linux *perf*'s default frequency). This approach is prone to low-frequency sampling errors, where it has a higher probability of detecting long latency stalls such as page table walks and L1 cache misses, but can often miss ultra fine-grained stall sources such as irregular dependency bubbles. Also, leveraging software interrupts as a method for sampling performance information disturbs the natural behavior of the benchmark by diverting the normal program flow to the sampling interrupt routine. The constant context switching between the benchmark code and the sampling routine can break data and timing dependencies in benchmark instructions and present a non-realistic view of the performance bottlenecks in the benchmark.

To achieve fine-grained performance evaluation, users can leverage Condominium to stream cycle-accurate (PC, stall) pairs from the core to perform instruction-level performance profiling. To extract precise microarchitectural information from RTL, Condominium leverages the same clock-gating mechanism used for ISA cosimulation. The DUT streams samples to PS across asynchronous FIFOs at a configurable sample rate, if necessary clock-gating identically to how Condominium manages emulation of interface timings. Critically, due to the backpressure mechanism, tuning profiling granularity becomes a simple trade-off between slowdown and precision. Figure 3.8 illustrates that with Condominium, an Oracle-like profiler incurs only moderate performance overhead and enables unprecedented insight for performance debugging.

The PS post-processes the stall information asynchronous to the DUT. Based on this information and profiler runtimes, users may choose to manipulate the sampling rate to gain further insights for design optimization. Figure 3.8 shows the emulation slowdown for performance sampling of the core with error-free per-cycle sampling or with different sampling frequencies that results from DUT clock gating. Note that due to other clock gating factors, such as maintaining a memory timing model, with increasing sampling interval, the slowdown curve saturates to a different value based on the running benchmark. As shown in Figure 3.9, due to the time-proportional nature of the profiler, aggregated stall stacks for the entire benchmark do not vary a lot across sampling rates, however, if users need insight into the attribution of stall cycles to instructions for localizing performance bottlenecks, sampling interval can greatly influence the correct attribution. As shown in Figure 3.10, once the attribution space has been expanded from a few stall bins by a factor of program size, the cycle attribution error rate rapidly grows with sampling interval. This dichotomy suggest a reasonable method for users using Condominium for performance evaluation: running coarse-grained regressions to gain a sense of the breakdown of important stall categories, and then running fine-grained analysis on interesting benchmarks to produce time-proportional stall attributions to individual PCs for deeper performance analysis.

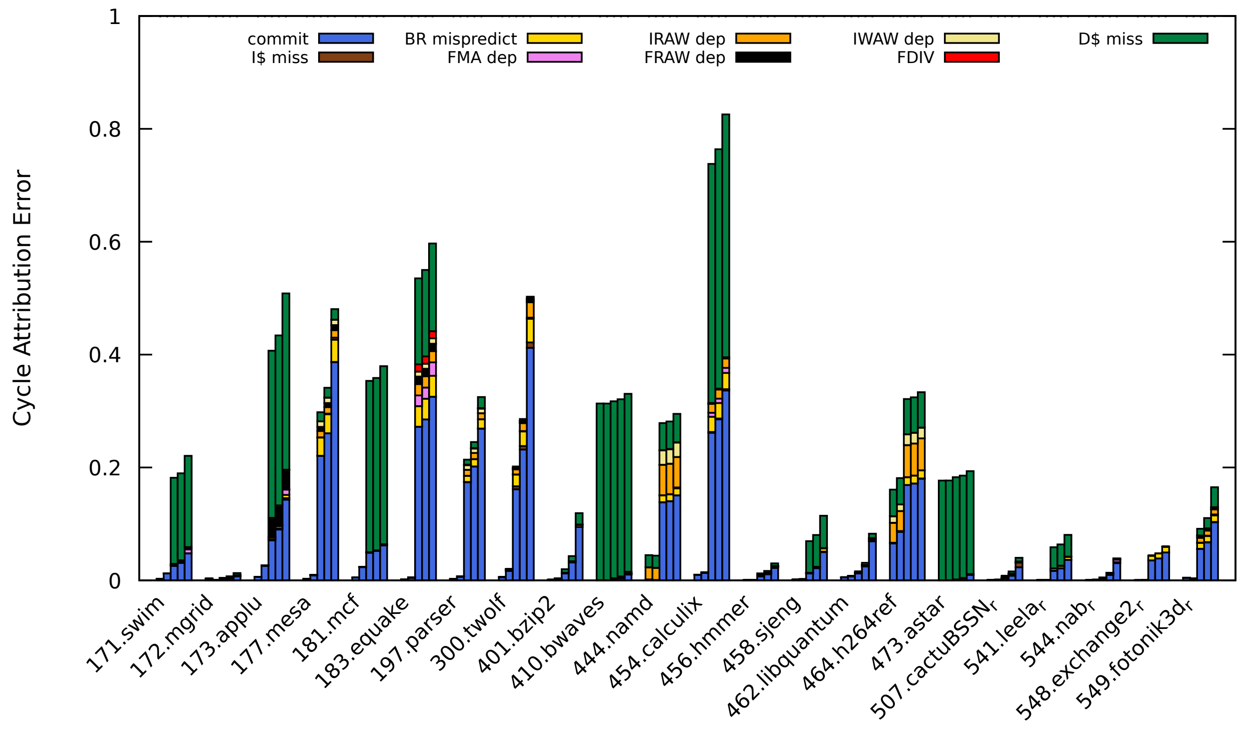
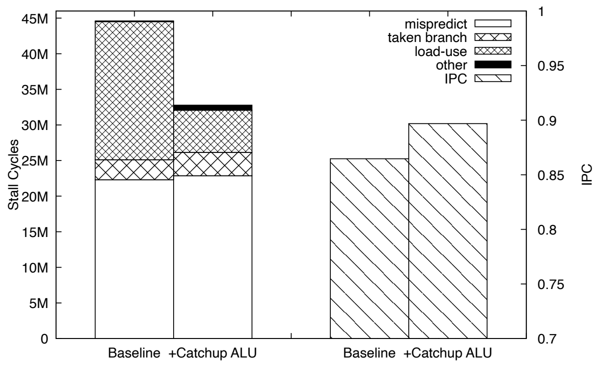


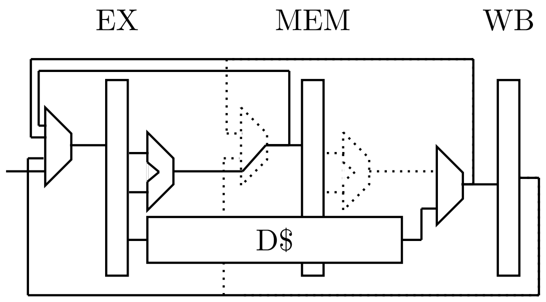
Figure 3.10: Stall cycle attribution errors for sampling intervals 10, 100, 1k, 10k, and 100k. Once attribution space is expanded to pair each cycle to a stall category and a PC, sampling errors have a noticeable impact on user's ability to identify segments of benchmark that act as performance bottlenecks.

3.4.3 Case Study: Catch-up ALU

In this section, we inspect how Condominium's performance profiling capabilities have been used to identify subtle performance bottlenecks in BlackParrot and measure the effect of solutions to mitigate those bottlenecks. After integrating a time-proportional performance profiler to the PL-Shell, Figure 3.11(a) shows the cycle-stack breakdown of stalls during execution of CoreMark [55]. While CoreMark is a flawed benchmark for full-system characterization, it is widely used as proof of microarchitectural optimization. Additionally, it is an ideal demonstration of performance optimization frameworks since there is so little low-hanging fruit remaining. Because BlackParrot is an in-order pipeline with large L1 caches, load-use stalls are a primary performance bottleneck, accounting for 18% of stalls in CoreMark. Load-use stalls have two subtypes: load-arithmetic and load-control operations. For number crunching applications, load-arithmetic stalls prevent optimal operation of tight loops. For pointer chasing segments, load-control stalls add extra delays on every null check.



(a) After basic core optimization, remaining low latency stalls (1-5 cycles) are difficult to detect via coarse-grained sampling. Tailored event counters can identify problematic categories, but lose PC association during aggregation. Condominium allows PS software to monitor stalls at a per-PC, per-cycle granularity.



(b) A second Catch-up ALU and set of bypass multiplexers allows the Catch-up ALU to execute pipelined instructions. However, a dependent non-integer instruction following a Catch-up operation will cause a bubble.

Figure 3.11: Condominium is used for stall categorization during CoreMark run, and using a Cath-up ALU to reduce the share of load-use stalls in the performance stack and improve the overall performance.

To reduce load-use stalls, we add a Catch-up ALU which is a secondary ALU located serially after the first ALU. Catch-up ALUs are a common way to improve performance in in-order cores. Out-of-order execution is often able to tolerate L1 hit latencies, so extra resources are better spent on more parallel ALUs for wider issue. For in-order cores, however, single threaded performance is sensitive to head-of-line blocking and so Catch-up ALUs can provide a substantial benefit. After justifying the idea in a high-level simulation model, we implement an RTL version of the idea in Condominium to evaluate marginal performance gains.

The Catch-up ALU resides in EX2, parallel with the second stage of the data cache access. When an integer or branch instruction has all dependencies met during issue, it is dispatched as normal to the Early ALU. Alternatively, when those dependencies are anticipated to be produced in EX2, the instruction is dispatched to the Catch-up ALU, which adds an additional cycle of latency, although fully-pipelined. In addition to arithmetic operations, the Catch-up ALU also processes control flow instructions. Because RISC-V branch comparisons are easily transformed from existing subtraction and comparison operations, this support is cheap to add. However, this feature adds complexity to the handling of branch mispredictions. The BlackParrot pipeline resolves branches early in EX1 to reduce the misprediction penalty. In order for load-branch operations to take advantage of the Catch-up ALU, the pipeline must suppress PC mismatches in EX1. Now, when the Catch-up ALU detects a PC mismatch, the pipeline must be flushed in addition to redirecting the front-end. Therefore in

BlackParrot, Catch-up ALU mispredictions are treated as synchronous exceptions, reusing their mechanism for replaying and recovering state. Figure 3.11(b) illustrates Catch-up ALU modifications to a sample five-stage pipeline.

As shown in Figure 3.11(a), the Catch-up ALU reduces load-use stalls from 43% of stalls to 18% of stalls, resulting in an overall 4% performance increase. There are additional stalls from dependencies on Catch-up ALU instructions, which now have an additional cycle of latency. However, these extra stalls do not diminish the gains from optimizing the more common load-use case. Interestingly, branch-related stalls increase by 1.04x, as deeper speculation past EX1 triggers additional mispredictions. A further optimization could restrict speculation only to branches which are predicted strongly taken which would increase load-branch stalls but should reduce Catch-up mispredictions. Leveraging cycle-accurate profiling with Condominium allows architects to easily identify potential bottlenecks as well as confirm both the positive and negative effects of their proposed improvements.

3.4.4 System-Call Abstraction

Due to the complexity of benchmarking experimental processor designs, architects use standardized benchmarks [27, 32, 55, 73, 74] to get a normalized performance insight into the design. However, the scale of commercial benchmarks are incompatible with the slowdowns of RTL simulators. Furthermore, conventional benchmarks also rely on a range of functions from the C standard library for I/O capability, file-system operations and memory management. While it is interesting to evaluate the performance of a full Linux distribution running a benchmark in user-space, during deep microarchitectural optimization architects often wish to observe bare-metal behavior. Yet, without operating system, it is impossible to run all but the most simple hand-crafted programs. Instead of glibc [53], embedded systems typically rely on smaller stdlib implementations, but these lack necessary system call compatibility.

To this end, software engineers have offered offloading solutions, such as RISCV-PK [52], that rely on issuing proxy kernel calls to a host machine for offloading POSIX system-call execution and sharing the result through a shared-memory and MMIO interfacing. Condominium supports this kind of proxy system-call benchmarking by hosting the underlying software library on the PS and using the PL-shell and the PS memory for interfacing system-call data back-and-forth between the program and PS. However, as shown in Table 3.2, system-calls can access many virtual and physical devices in a system, which makes it difficult to standardize their processing time when relying on running proxy-kernels on a completely different x86 machine. As a result, benchmarks that rely on these system-calls, like file-system operations, can have their performance heavily skewed by the native execution of the system-calls on the x86 processor which might behave completely differently from a RISC-V processor under test. To address this challenge and further simplify bare-metal benchmarking, software libraries like PanicRoom [25] aim to offer simple implementation of POSIX system-calls that

Table 3.2: POSIX system-calls [118] can access a range of system devices depending on their functionality and input arguments. By providing native bare-metal implementation of system-calls and standardizing their downstream device access time, users can expect a realistic performance behavior in bare-metal benchmarks.

syscall	**Function**	**Memory**	**Network**	**Other IO**
open/openat/close	obtain/release file descriptor	✓	✗	✓
read/pread/write/pwrite	read/write to file/device	✓	✓	✓
lseek	change file offset	✗	✗	✗
stat/fstat/lstat	access file metadata	✓	✗	✓
brk/sbrk	grow program heap	✗	✗	✗
nanosleep	suspend thread	✗	✗	✗
getpid/getuid/clock_gettime	misc system information	✗	✗	✗
mmap/munmap	map file/device to VM	✓	✗	✗
bind/connect	initialize socket	✗	✓	✗
send/recv/writev/readv	transmit/receive over socket	✗	✓	✗

break the reliance on proxy host offloading and create a standalone bare-metal benchmark binary. PanicRoom implements this functionality by using ARM LittleFS [11], an open-source, lightweight, DRAM-based file-system designed for embedded flash memories. But running bare-metal benchmarks built upon PanicRoom, Condominium can easily rely on the timing models for common peripherals, like DRAM, to get a more realistic processing time for system-calls that access said peripherals. This approach guaranties that a RISC-V system's performance is solely determined by running RISC-V instructions that access common endpoint IOs with controlled timing guaranties.

3.5 Related Work

While Condominium shares similarities with many FPGA-accelerated prototyping platforms, its Scale-Down focus and aggressive portability make it uniquely cost and effort effective. In this section, we compare to existing projects which offer subsets of the features in Condominium.

3.5.1 Gate-Level Accelerated Emulation

Teams desiring a out-of-the-box solution to RTL emulation employ commercial tools for FPGA-accelerated design modeling, such as *Cadence Palladium* [33], *Synopsys Zebu* [134] and *Mentor Veloce* [105]. Unfortunately this convenience is costly, with obfuscated pricing up to millions of dollars. In contrast Condominium is free and open-source, provided cycle-accuracy guaranties and design transparency, and requires an initial investment up to hundreds of dollars.

Table 3.3: Compared to other FPGA emulation platforms, Condominium has minimal TCO for the smallest designs. Aside from enabling the use of low-end FPGAs, the lack of a dedicated host server reduces hardware cost and eliminates sysadmin overheads. Additionally, by eliminating dependencies on vendor IP, Condominium is able to provide cycle-accurate co-simulation using open-source tools at no licensing cost to the user.

Platform	**Cost Model**	**Design Ratio** [0]	**Host**	**Cycle-Accurate**	**Co-Simulation** [1]	**Vendor-Agnostic**	**Open-Source**
Commercial [2]	High-End Cluster	1:1	Proprietary	✓	✓		
FireSim [92]	Cloud Rental	N:N	AWS F1	✓			✓
SMAPPIC [41]	Cloud Rental	N:1	AWS F1				✓
FreezeTime [106]	High-End Board	1=1	PCIe Server	✓		✓	
Condominium	**Low-End Board/ Low-End Cluster**	**N=N**	**None**	✓	✓	✓	✓

[0] The ratio of FPGAs:Designs in a single system emulation. Commercial tools map large hierarchies into large clusters. Firesim is able to emulate arbitrarily large systems using cloud auto-scaling. SMAPPIC is able to split large designs across FPGAs. FreezeTime maps a single design to a single FPGA. Finally, Condominium maps a number of designs to a fixed-sized local cluster.

[1] Co-simulation refers to the ability to reproduce the cycle-exact output of a system emulation on an RTL simulator such as Verilator [131] (albeit at significant slowdown). This ability is essential in emulation-system debugging.

[2] We combine Synopsys Zebu [134], Cadence Palladium [33] and Mentor Veloce [105] with similar features and limitations.

3.5.2 Emulating Large Systems with FPGAs

FireSim [92], *DIABLO* [136] and *SMAPPIC* [41] focus on scaling out the emulation to analyze large-scale designs such as datacenter-scale systems. They work by partitioning the system design over multiple FPGAs and using Ethernet-based token-passing systems to capture inter-node timing. Because they are based on AWS F1 [8] infrastructure, the emulation model relies on proprietary vendor libraries for the hardened AWS shell as well as PCIe DMA interfaces. Furthermore, relying exclusively on PCIe for token networking means any sub-microsecond access latency should be modeled by pausing emulation. As Condominium focuses on single-node systems, it allows for local execution with open-source simulations, resulting in a much lower recurring cost. In contrast, a local version for a comparable F1 FPGA setup, may cost tens of thousands of dollars. Condominium also allows for flexible and low-latency peripheral access via AXI bus.

3.5.3 Decomposed FPGA emulation

Similar to a Scale-Down methodology, *Protoflex* [43] and *FAST* [40] accelerate performance analysis using FPGAs. However, they focus on acceleration of large, slow, cycle-accurate models, attempting to gain performance insights into systems too large to simulate in a reasonable time frame. In contrast, Condominium allows for cycle-accurate emulation of arbitrary RTL so that architects can easily validate and debug performance with the deep introspection that RTL provides. *FreezeTime* [106] time-multiplexes the FPGA fabric so one board can time-share multiple RTL partitions. Similar to Condominium, *FreezeTime* leverages BUFGCE FPGA primitives to stall emulated blocks while virtualized blocks process cycle-accurate timing models. However, once multiple virtualized accelerators are accessed in the same time slice, *FreezeTime* needs to iteratively load and store accelerator accelerator state space into the memory, which greatly affects real emulation time on memory-bounded systems. On the other hand, Condominium aims for greater flexibility and lower resource overheads by standardizing C++ timing models in the PS and PL-shell interfaces and gaining fine-grained insight into emulated subsystems.

3.5.4 FPGA-Accelerated Performance Analysis

While custom cycle-level simulators and silicon performance counters are state-of-art for commercial performance validation, researchers have also proposed using accelerated sampling for microarchitectural debugging. *FirePerf* [91] provides two categories of microarchitectural analysis: commit tracing via TraceRV and out-of-band hardware profiling via AutoCounters. Condominium supports not only commit tracing and out-of-band event counters via PL-Shell CSRs but also time-proportional instruction profiling, allowing for cycle-attribution of performance bottlenecks and instruction. Additionally, Condominium is written in standard SystemVerilog rather than Chisel [22], making it more familiar to hardware designers. *TEA* [61] and *TIP* [62] propose time-proportional event analysis by creating Per-Instruction Cycle Stacks (PICS) to unify performance profiling and performance event analysis. While TEA and TIP are able to accurately ascribe microarchitectural events on average, they rely on statistical sampling by periodically interrupting the program that disrupts non-interference. Because Condominium combines commit-stage cycle attribution with cycle-accurate performance data streaming, it is able to accurately attribute cycles without any sampling errors, as well as trade co-emulation speed for sampling accuracy.

Chapter 4

High-Fidelity Coverage

4.1 Acknowledgment

Research in computer architecture is an intensive collaborative effort, and the work on High-Fidelity Coverage has relied very heavily on contributions from fellow BSG members: Anoop Mysore Nataraja, Paul Gao, Dan Petrisko, and Prof. Michael Taylor. I would like to specially thank Anoop Mysore Nataraja for the design of the coverage instrumentation algorithm and collaboration on related work analysis, and Paul Gao for the help on the bring-up of the ZC706 board which was used for coverage evaluations.

4.2 Motivation

The emergence of open-source hardware, alongside the growing need for more energy-efficient and high-performance computers, has resulted in an explosion of increasingly complex processor and accelerator designs. As open-source tools and hardware libraries [131, 142] continue to streamline the design process to cater to more agile chip-design practices, verification of complex designs has remained a costly and time-consuming exercise. Unlike software development, the hardware development process requires engineers to extensively verify their design before manufacturing since mitigating bugs after tape-out would be impractical, if not impossible [117]. Understandably, functional and performance validation consume a significant portion of the hardware development cycle, as hardware engineers, on-average, spend around 49% of their time on verification [50]. This, in addition to the growing complexity of modern processors and accelerators, motivates the use of diverse and efficient verification methodologies to ensure design correctness and reduce time-to-market. Among popular verification solutions, formal verification allows for the testing of complex design state space using formal methods and guarantees. However, formal methods have high computational cost and limited scalability. Dynamic verification and fuzzing [129] do not rely on formalization of the models; instead, they allow for a practical exploration of the design's functional state space for faster verification with significantly lower computational and human effort. This, of course, comes at the cost of strong formal guarantees which lowers confidence. In reality, due to the complex and modular nature of RTL design, developers usually opt for a combination of directed unit-testing, ISA compliance testing, formal verification, and hardware fuzzing to verify the functionality of their design at various stages of the chip development process.

Fuzz-testing, or Fuzzing, as a verification solution has evolved from constrained random verification. Initially developed for software verification, a fuzzer aims to find unexpected faults and vulnerabilities in software by triggering the various corner cases and behaviors it can exert using randomly generated inputs to the program. Modern software fuzzers, such as *AFL* [180], combine a brute-force program generator with a smart guidance algorithm based on instrumenting the code. AFL, for example, uses a modified form of software edge coverage to effortlessly pick up subtle, local-scale changes to the C++ program control flow. This coverage then is used as an input feedback to the learning-based algorithm which in turn guides the test generator towards selecting more interesting seeds for future test generation. The ultimate goal of the guidance algorithm is to use the history of the relation between generated tests and their corresponding program coverage to decisions on future test generation, so in the long run, it achieve a higher chance of triggering hard to reach states of the C++ program.

Inspired by prevalence of fuzzing in the software domain, hardware designers have attempted to migrate the same methodology for hardware verification. Modern hardware fuzzers, similar to their software counterparts, are usually composed as a verification loop including a test generator, a simulator for the design under test (DUT), a golden ISA model for bug discovery through differential cosimulation, and an RTL feedback, typically in form of coverage, for guiding future test generation towards exploring new DUT states. The use of coverage as a guiding feedback is called, based on the level of design transparency, Gray-box or White-box fuzzing, while some hardware fuzzers, called Black-box fuzzers, have opted out of guiding the fuzzer based on RTL feedback, and in turn have focused on improving the test-generation itself. Recent advancements in hardware fuzzing have focused on improving various aspects of the fuzzing loop. This includes the work on test generation and mutation algorithms for generating high-quality and self-verifying input programs [34, 93, 132, 179]. Others apply promising machine-learning and decision-making algorithms [30, 37, 60, 123] to guide the test generation aimed at maximizing design exploration in the long run. Other focus on application specific verification aimed at verifying certain ISA functionalities or security concerns [30, 35, 93].

The migration from the software to the RTL domain, however, faces challenges due to the different natures of the two domains and how similar concepts can be interpreted differently in them. To attempt a blind migration by overlooking inherent differences such as the sequential nature of software as opposed to the inherent concurrency of hardware elements in

RTL design, may prevent hardware fuzzing to reach its true potential efficacy in hardware verification. Notably, the choice of the coverage metric is a critical and under-evaluated decision in the design of a fuzzer. Not only does the quality of the metric provides a clear representation on how exhaustively the fuzzer has explored the design, but it also strongly influences subsequent test generation in coverage-guided fuzzing. Analysis on contemporary fuzzers [29] has shown that their proposed coverage metrics offer little improvement in fuzzing efficacy due to their nonalignment with the goal of RTL exploration and bug detection. Driven by the software inspired definition of coverage and prevalence of simulator enabled coverage metrics, conventional fuzzers often receive coverage feedback that ignores the inherent concurrency of RTL design, therefore steering the fuzzer away from the goal of maximal RTL exploration. Also, many of these coverage metrics either rely on built-in features of RTL simulators, or cannot be effectively synthesized on FPGA with acceptable scalability, significantly slowing down the fuzzing loop. With the continuous emergence of increasingly intelligent fuzzers that incorporate coverage as the sole source of RTL feedback for input generation, choosing a high-fidelity coverage metric that accurately represents the degree of DUT exploration can significantly impact the performance of fuzzing. Alternatively, a low-quality coverage metric can quickly misguide an input generation algorithm by providing inaccurate feedback on what type of inputs trigger a newly explored space in the RTL.

In this work, we focus on two important aspects of the coverage metric: *Fidelity* and *Feasibility*. Fidelity in a coverage metric means that it provides an accurate representation of the degree of RTL exploration, one that directly corresponds to triggering faulty datapaths. In a hardware unit, the output is driven by a tree of computational datapaths, with the activated path selected through a combination of cascaded control signals operating in different pipeline stages. It is only through the careful coupling of these control signals with their relative latencies that one can deterministically identify the activated datapath in the current cycle. By designing a coverage metric that incorporates this information, fuzzers can keep track of activated RTL datapaths, therefore providing an accurate metric for design exploration that can lead to bug discovery.

Feasibility indicates that the coverage collection should not impede the use of FPGA acceleration for DUT emulation, leaving iterative fuzzing experiments bound to the slow performance of software RTL simulators. Fuzzing is an iterative process where the design state space is incrementally explored through many iterations. Software RTL simulators have the advantage of full design visibility, but are impractically slow for exhaustive testing. FPGA acceleration therefore is a crucial method for reducing the turnaround time for each iteration and significantly improving the performance of the overall fuzzing methodology. This rises an important consideration in RTL coverage instrumentation – the tradeoff between the emulation performance and the complexity of the coverage metric. Processors are complex RTL designs composed of various computation pipes, state machines, memory systems, and controllers. A coverage metric that can adequately represent the influence of a test program on the DUT incurs a significant instrumentation overhead that can sometimes mirror the complexity of the DUT itself. While a well-defined group coverage metric can provide this needed complexity, the number of possible combinations grows exponentially with the group width, so storing and processing covergroups can be an expensive exercise since the storage overhead grows exponentially with the group size [100]. Moreover, FPGA emulations of instrumented DUTs allow limited visibility due to resource and interface limitations. By designing the coverage collection peripherals in a way that can be synthesized alongside the DUT with minimal area and performance overhead, we can break the reliance on software RTL simulators and enable massive speedups in hardware fuzzing. To enable high performance coverage guided fuzzing for RTL designs, this work aims to provide:

- A latency-aligned group-coverage metric that is designed to serve as a high-fidelity representation of RTL datapath activation within DUT for the purpose of guiding modern fuzzers towards effective design exploration.
- A synthesizable, low-overhead coverage collection engine that integrates into Condominium and enables FPGA acceleration of coverage guided fuzzing by tracking newly covered paths without losing cycle-accuracy.
- An automated instrumentation algorithm that can parse any given SystemVerilog design and extract latency information needed for generating the proposed high-fidelity coverage.

In section 4.3, we introduce the fundamental building blocks in a coverage-guided fuzzing loop and provide an overview of conventional hardware coverage metrics and their efficacy for guiding RTL fuzzer algorithms. In section 4.4, we introduce the high-fidelity hardware coverage metric that aims disambiguate the mapping between coverage values and activated RTL datapaths within the design. We also propose an automated instrumentation algorithm for extracting coverage latency information needed for generating the high-fidelity coverage from SystemVerilog design descriptions. In section 4.5, we intro specialized coverage engines, that by integrating into Condominium, collect unique coverage combinations and transmit to a host system for post-processing. These engines enable FPGA-acceleration of group-coverage metrics by eliminating the exponential state explosion problem of wider covergroups. In section 4.6, we evaluate the FPGA slowdown and utilization tradeoff of using high-fidelity coverage engines, present BlackParrot case studies where coverage latency-alignment highlights buggy behavior, and compare the guidance efficacy of different coverage metrics when guiding a simple coverage-guided fuzzer algorithm.

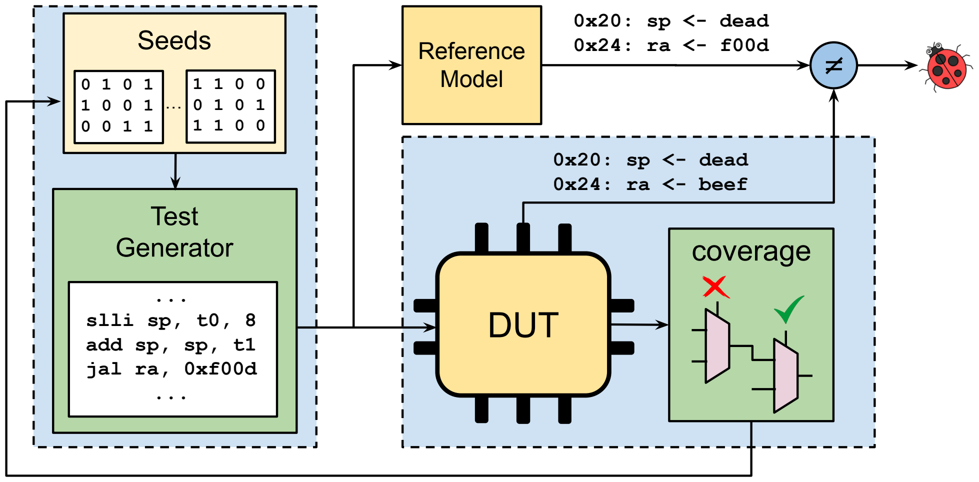


Figure 4.1: Coverage-guided hardware fuzzing loop including a test generator, DUT emulation environment instrumented with coverage engines. Collected coverage completes the feedback loop for future test generation while, in parallel, a reference model is used for bug discovery.

4.3 Background

4.3.1 Coverage Guided Fuzzing

Fuzzing is the practice of testing a design using automatically generated inputs to trigger and reveal underlying bugs and vulnerabilities in a design under test. As shown in Figure 4.1, in the context of hardware design, fuzzing is a closed loop verification methodology for DUT design space exploration with three main steps:

Test Generation: Input tests are automatically and randomly generated for the purpose of exploring the design and revealing bugs and vulnerabilities. There are many different flavors of test generators depending on the type of hardware under test, and the goal that the fuzzer is operating towards, such as searching for generic RTL bugs, or more application-specific purposes, such as revealing timing side-channels [30]. Tests can be generated either from a carefully chosen pool of templates which can be mutated and combined over time [37, 60, 179], or from the scratch by carefully arranging randomly generated instructions in a program [132]. Optionally coverage feedback is used for accumulating information on the relation between previous test generation and mutation choices and the resulting coverage, which in turn can be used to decide on interesting test seeds and mutations to generate in the future iterations [37, 60, 123].

Emulation and Coverage Collection: Input tests are then fed into the DUT for verification and coverage collection. For the purposes of DUT emulation, either software RTL simulators [131, 135], or FPGA environments [92] are used. Here, the DUT is carefully instrumented to incorporate chosen coverage metrics along with the necessary scaffolding and wrappers to methodically collect and compile coverage data, which will be provided as a feedback to the test generator for future iterations. The coverage is supposed to act as a proxy metric for evaluate the effectiveness of the input test in exploring the DUT state space. While using FPGA prototyping for this step significantly speeds up the fuzzing turnaround time, reliance on simulator-provided coverage reports, and non-scalable and non-synthesizable coverage collection logic has tied many fuzzers to software RTL simulation.

Verification: The main purpose of hardware fuzzing is to reveal bugs and vulnerabilities, so in after or in parallel to the DUT emulation, the test execution should be checked to ensure functional correctness. This can be done either by relying roughly on design assertions or crashes to reveal fault execution, or by using a golden reference model of the DUT to perform cross-reference co-simulation for run-time verification and bug localization. Some fuzzers [132] also generate self-verifying programs where bugs or deviations can surface through runtime errors, however their reliability is limited because the self-verifying mechanism built into the program itself is prone to RTL bugs and can behave unpredictably. While the verification step has no effect on steering the fuzzing experiment, it is vital for recording and reproducing the issues found throughout the experiment.

The test generation step is largely DUT-agnostic since it relies on leveraging the ISA to generate instructions that can increase the achieved coverage. There is an implicit assumption here: that an increase in coverage metric correlates to an increase in the DUT state space explored by the fuzzer. Since the test generator cannot directly observe activated RTL datapaths, it relies on an coverage metric as a proxy for that and expects an increase in coverage to represent a newly explored design space. So the RTL coverage serves as the sole microarchitectural feedback guiding the fuzzer towards better design exploration. This paradigm reflects a reinforcement-learning problem where an agent (test generator) chooses actions based on its previous interactions with the environment it is embedded in (emulated RTL) to maximize the cumulative reward (coverage metric). Researchers have attempted to apply various reinforcement-learning solutions to test-generators to fit the exploration-exploitation nature of hardware fuzzing [37, 60, 123]. While optimizing the test generation can lead to a better utilization of the ISA and result in more diverse programs, leading to a faster increase in coverage, it does not necessarily result in a better exploration of the RTL. A poorly designed reward function can result in suboptimal agent performance and prevent effective learning by misguiding the fuzzer from the goal of better RTL exploration.

This reinforces the importance of defining a high-fidelity coverage metric that accurately represents RTL exploration, as the sole DUT-dependent reward function in the loop.

4.3.2 Contemporary Coverage Metrics

In the context of hardware verification, there are a few popular coverage metric that vary in their representation of design exploration, being generic or directed at certain functionality, and their feasibility to implemented for FPGA prototyping in an scalable manner. These metrics can be derived either directly from the hardware description language (HDL) for different types of code coverage, or from the inferred logic for tracking various microarchitectural functionalities, or from the ISA for ensuring the correct functionality of different types of instructions supported by the ISA [75].

- **HDL line coverage:** The coverage over individual executable lines of the HDL. This metric is not portable across languages for the same design and can over-represent complex expressions. These metrics are typically implemented in software RTL simulators and is not easily scalable for FPGA acceleration.
- **Toggle coverage:** The coverage over all individual nets within the design over the two possible value transitions (0-1-0 and 1-0-1). A well implemented toggle coverage metric would consider even intermediate nets from the netlist representation that are not explicitly declared in the HDL. This is also typically implemented in software RTL simulators and not immediately scalable for FPGA acceleration.
- **FSM and interface coverage:** Tracks the transitional states of various Finite-State-Machines (FSM) or handshake interfaces within the design. This metric is useful for tracking the high-level control state of the design for preliminary testing, but fails to capture all microarchitectural transitions within the DUT. This is used alongside other metrics since the coverage information from the metric is typically orthogonal to other metrics.
- **CSR coverage:** Tracks the possible value transitions in the ISA CRSs implemented in a processor. This metric can be used to ensure different execution modes are compliant with the ISA requirements and can be integrated into the ISA compliance test suite. While this is a good check for maintaining high-level compatibility between the DUT and ISA, it does not capture microarchitectural states within the design.
- **Verilog Assertions:** Using a set of coverage primitives and formally-composed assertions for expected behavior of signals and complex functional properties, a metric of coverage can be defined to be the extent of coverage over the assertions. The advantage is that higher-level functionalities can be covered with explicit specification. Typically, however, stronger and higher-level functionalities require proportionally complex design insight. While assertions provide the guaranty the previously known issues do

not resurface, they do not provide a metric for exploring previously unknown bugs occurring in the RTL.

- **MUX toggle coverage:** The coverage over the toggle status of individual multiplexer select signals. This metric is inspired by software branch coverage, where, due to the sequential nature of software programs, only one branch is active at a time, and a nested branch activation implicitly confirms the activation of parent branches, so this coverage information would be useful for tracking various control flows of the software. However, unlike software branch coverage, due to the parallel nature of hardware, this metric as is would be incapable of accounting for the interaction of cascaded multiplexers within a datapath. For example, while toggling the select signal of a certain MUX provides information on which MUX input is connected to its output, the data might still be lost and not used due to downstream MUXes not selecting the said MUX output.
- **MUX group coverage:** This metric builds upon MUX toggle coverage by grouping the identified select signals into covergroup vectors and tracking the resulting combination values. By bundling individual coverpoints together in the covergroup, the metric aims to provide an abstract low-level state of the RTL unit corresponding to the single datapath trace that was active among all the possible datapath tree made possible by the cascade of multiplexers. This bundling mechanism is not exclusive to multiplexers and can be used to track the interaction of many different control signals within a module.

While proposed group coverage metrics offer a more complete feedback that toggle coverage on design exploration, it still fails to take into account the relative timing latencies that these coverpoints operate in the RTL. By ignoring the relative latencies of the grouped MUX control signals that operate in different pipeline stages, the resulting covergroup fails to accurately map newly activated routes in the RTL datapath to a distinct combination value in the coverage metric. In section 4.4 we propose a *high-fidelity coverage* metric that aims to create a direct mapping from coverage value to activated RTL datapaths. Furthermore, the exponential space complexity of group coverage makes it an expensive candidate for FPGA acceleration [100]. To mitigate this scalability problem and enable group-coverage for complex designs, mechanisms like hashing have been proposed to reduce the effective widths of covergroups [77]. However, this approach can introduce inaccuracies in the coverage feedback due to hashing collisions masking real progress in design exploration. In section 4.5 we propose a *CAM-based coverage engine* aimed at enabling group coverage collection on FPGA resorting to hashing or losing cycle-accuracy.

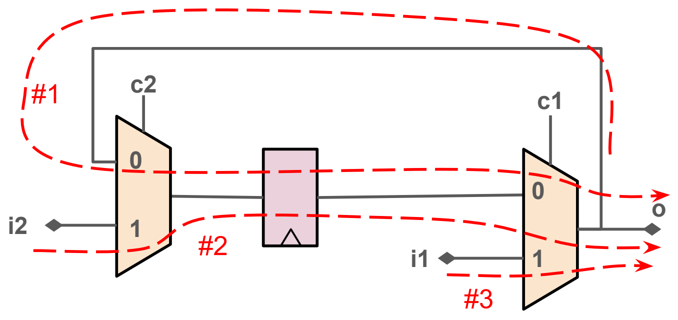
4.4 High-Fidelity Coverage

This section introduces the proposed latency-aligned group coverage metric. We argue that ignoring the relative latencies of control signals governing the datapath leads to misrepresentation of design exploration and provides inaccurate feedback for coverage guided fuzzing. We highlight the effect of latency alignment with an example design and showcase how unaligned group-coverage can both under-represent and over-represent design exploration, whereas the latency-aligned metric deterministically provides a reliable feedback for the same purpose. Also, to facilitate the process of instrumenting the design, we introduce the instrumentation algorithm for automatically inferring coverpoints and their relative latencies from the SystemVerilog representation of the DUT for the purpose of composing high-fidelity covergroups for instrumented fuzz testing.

4.4.1 Latency-Aligned Group-Coverage

The migration of fuzzing from the software domain to hardware, has carried over certain technical assumptions that do not hold true in this new environment. One of these assumptions is the sequential nature of software, and specifically branches. Once a branch is resolved as taken or not taken, one can imminently infer the result of its parent branches. Also, branches are resolved in the order they are executed and the delay between their execution does not influence the flow of the program. This delay is a factor of the non-branching instructions executed in the between them, compiler configurations, and the CPU performance, and the vast majority of well written programs are designed agnostic to these details. Non of these assumptions hold true when dealing with hardware, and while MUXes, like branches, do represent a decision tree in the design, their results are a factor of sequential interplay and latency sensitive data. By proposing a latency-aligned group coverage metric, we aim to resolve these unfounded assumptions and provide a coverage metric that faithfully represents the RTL environment. Therefore, to accurately represent the timing complexity and the intertwined nature of control signals and sequential elements in hardware designs, we present a latency-aligned group-coverage metric that builds on the previously used methods and provides a coverage feedback that's a more accurate representation of explored RTL.

Grouping cascaded coverpoints without recognizing their relative latencies results in producing an ambagious coverage metric that cannot provide reliable information about activated RTL paths to the fuzzer. For example, consider a circuit, as shown in Figure 4.2, with output *o* and inputs *i1, i2*. Control signals *c1, c2* govern the activation of the highlighted datapaths through the 2 multiplexers and a register. To fully test this circuit, all 3 datapaths #1, #2 and #3 must be exercised to reveal potential bugs that can manifest due to wrongfully designed logical units along those paths. For the latency-agnostic covergroup *{c1, c2}*, only 2 of the 4 possible combinations directly map to a specific activated datapath, while the other 2 combinations do not offer any reliable information on which datapath is activated in the current cycle. For achieving a deterministic direct mapping from covergroup values to acti-



c1	c2	o	Path
0	0	?	?
0	1	?	?
1	0	i1	#3
1	1	i1	#3

$\xrightarrow{\text{align}}$

c1	c2_r	o	Path
0	0	o_r	#1
0	1	i2_r	#2
1	0	i1	#3
1	1	i1	#3

Figure 4.2: A simple data-path controlled by two MUXes and a pipeline register. The latency-agnostic covergroup table(left) shows an unclear correlation from covergroup value to activated RTL path. The latency-aligned covergroup table(right) however shows a direct mapping from covergroup value to the activated RTL path.

vated paths, the relative latencies of the control conditions to the output must be considered. As a result, coverpoint *c2* must be delayed by one cycle. By defining the latency-aligned covergroup to be *{c1, c2_r}*, every possible coverage combination value maps to a specific activated datapath. The difference between the two metrics becomes more apparent when the circuit is subjected to the control streams depicted in Table 4.1. The latency-agnostic metric is both prone to false-positives, showing a 100% coverage despite not having covered all the possible datapaths, and also false-negatives, not showing 100% coverage even if all paths are activated. On the other hand, the latency-aligned values provide an accurate representation of the corresponding activated paths, providing a reliable feedback on RTL exploration for fuzzing experiments.

By incorporating the latency information into our coverage metric, we are essentially defining how these individual control signals interact with each other in different stages of the RTL pipeline. Instead of looking at a snapshot of all control signals in a single cycle, as with the latency-agnostic metric, we are following along RTL datapaths and looking at control signals in their effective cycle latencies that they contribute to the datapath output relative to the current cycle. Also, it should be noted, that designers can choose any type of control signals they decide to be of testing importance as coverpoints that can be realigned and bundled into covergroups. Here, the MUX select signals were chosen as coverpoints because

Table 4.1: When providing different MUX control signal streams to Figure 4.2, the latency-agnostic metric can both over-represent and under-represent explored RTL paths while the latency-aligned metric provides an accurate exploration feedback.

cycle	0	1	2	3	4	5	
c1	0	0	1	1	0	0	
c2	1	1	0	0	1	1	
path	-	#2	#3	#3	#1	#2	
{c1, c2}	01	01	10	10	01	01	$\rightarrow$ 50%
{c1, c2_r}	-	01	11	10	00	01	$\rightarrow$ 100%
c1	1	1	0	1	1	0	
c2	1	1	1	0	1	0	
path	-	#3	#2	#3	#3	#2	
{c1, c2}	11	11	01	10	11	00	$\rightarrow$ 100%
{c1, c2_r}	-	11	01	11	10	01	$\rightarrow$ 75%

they serve as generic RTL signals acting as decision points within the design, however, many other control signals driving various functional units can be added to the metic with the discretion of the hardware developers.

4.4.2 Automated Coverage Instrumentation

For inferring the latency information between coverpoints, we propose an automated instrumentation algorithm that leverages an open-source SystemVerilog compiler to generate the casual tree of drivers for each net in the design. This tree can be methodically traversed to generate the timing information necessary for the instrumentation. After that, each coverpoint can be realigned for composing the latency-aligned covergroup. The instrumentation algorithm for enabling latency-aligned coverage metric takes a given design description in SystemVerilog, extracts information about coverpoints and their relative latency information, and adds non-functional and synthesizable statements non-intrusively to instrument the DUT for coverage collection. The identification of candidate coverpoints is done using a configurable C++ visitor routine that parses an intermediate representation (IR) of the input design. The IR is generated by Surelog [4], an open-source SystemVerilog pre-processor, parser, and elaborator. Given an RTL design in SystemVerilog, Surelog produces an abstract syntax tree-like representation of the design, called Universal Hardware Data Model (UHDM), compliant with the Verilog Object Model. UHDM object then can be parsed through Python or C++ using Verilog Procedural Interface (VPI) APIs [45]. We use the design's UHDM object reference as an input to the C++ visitor routine which in turn traverses the data structure to extract coverage information. The instrumentation algorithm then generates corresponding hierarchical references in dot notation [122] to bundle the aligned candidate coverpoints after cycle-shifting them their relative latencies into covergroups. The latency-aligned covergroups then can be fed into coverage engines for the purposes of efficient storage and streaming out of FPGA to be used in verification.

To enable greater flexibility and debuggability through isolation, the algorithm processes the UHDM object in the granularity of functional units. The DUT is broken into various modules controlling different aspects of the its functionality, and each module is instrumented and represented by one covergroup. Isolating the exportability of different functional units into different covergroups can enable more mature fuzzers, like ML-based test generators, to associate coverage on specific subsections of the designs to the generated instruction sequences, and use that learned association to drive up coverage in the corresponding unit. For each instrumented module, the proposed visitor routine, develops a netlist-level association between individual nets and their corresponding drivers. Subsequently, using this association, it derives all possible datapaths terminating at each of the output ports of the top module starting, by generating the causal graph of said output port ending in input ports as leaves and internal nets as intermediate nodes. Each distinct datapath driving the output is modeled as a route from the root to the leaves and the MUX control signals on that route represent the coverpoints activating that datapath. The visitor also tracks the logical depth increments, equivalent to the number of registers passed throughout the route, for each datapath to help align the coverpoints by their relative latency to the output port. The latency-alignment is done through DFF chain instances before assembling the covergroup.

The visitor routine has two main stages: *Parse* stage and *Traverse* stage. The algorithm begins in the Parse stage where a UHDM model of the top module generated by Surelog is parsed to capture the relationships between its ports, nets, and sub-module IO connections. This relationship is transparent to the boundaries of the language features such as generate blocks, always blocks, macro declarations, etc, and populates a collection of data structures containing causal information about RTL nets and their corresponding drivers. At this stage, for each module instance, several data structures are populated with related netlist information. This includes instance parameters, ports and their high-connections, internal nets and their driver information, and sub-module instances and pointers to their data structures. For the purposes of this stage, only the information about right-hand-side drivers of a net is stored and logical operands are discarded, however, information about a net being driven as a DFF output is stored for the purposes of latency calculation.

Following this, the Traverse stage identifies all the output ports of the top module and traverses each port's driver or source independently. The *traverse* routine, shown in Algorithm 1, starts at each module output with a zero *depth* latency, and begins traversing upstream recursively through their driver tree until it reaches the module boundaries at the input ports. For each net, to avoid infinite recursion caused by RTL loops, the traversal routine checks and returns if the current net has been already visited in the current recursion path. If the current net contains a coverpoint, such as MUX selects, it records the coverpoint alongside its current depth in relation to the output. Then the traversal routine is invoked on all of the current net's drivers, which can be internal signals in the current module, or ports in

Algorithm 1 Coverage Instrumentation Algorithm

```
1: procedure INSTRUMENT(module)
2:     PARSE(module)                              ▷ populate module data-structures
3:     covs ← {}
4:     for port in OUTPUTPORTS(module) do         ▷ start traversing output ports
5:         TRAVERSE(port, 0, covs)
6:     return covs
7: end procedure
8:
9: procedure TRAVERSE(net, depth, covs)
10:    if ISTOP(module) and ISINPUT(net) then     ▷ return on an input endpoint
11:        return
12:
13:    if net in visited then                     ▷ return on a visited net on current branch
14:        return
15:    visited.insert(net)
16:
17:    if ISMUXOUT(net) then                      ▷ record cover-point relative latency
18:        covs.insert({muxSel, depth})
19:
20:    for driver in drivers[net] do              ▷ traverse RHS net drivers
21:        if ISREG(net) then                     ▷ increase latency upon traversing registers
22:            TRAVERSE(driver, depth + 1)
23:        else
24:            TRAVERSE(driver, depth)
25:    visited.pop()                              ▷ update branch visited nets
26: end procedure
```

a sub-module, or the super-module containing the current module instance. To keep track of the current net's latency relative to the output, the depth number is incremented when passing through a DFF output during traversal. Once the traversal ends, the latency-aligned covergroups can be generated by running the coverpoints through DFF chains matching the length of their corresponding output latency.

While instrumenting the DUT a one-time task and not part of the fuzzing loop, the algorithm parses SystemVerilog designs quickly. Also, the visitor routine is highly conducive to parallel programming. When traversing a net's driver sub-tree, each branch can be independently traversed. This presents opportunities to parallelize parse and traverse instances dynamically as threads or work-items. In large chip designs, a parallelized visitor routine can highly benefit incremental design iterations. For the experiments we conducted the instrumenta-

Table 4.2: Coverage instrumentation time for various BlackParrot sub-modules. Even without parallelization, covergroup information including the constituent coverpoints and their relative latencies can be inferred in reasonable time.

BlackParrot	ZC706 Utilization			Covergroup	Visitor
Submodule	LUT	BRAM	FF	Width	Time
Backend	23611	16	7067	909	1:55
Frontend	3239	12	2333	246	1:28
Core	26847	28	9400	1171	2:03

tion of the BlackParrot core complex on an Intel Core i9-7940X CPU running at 3.10GHz, Table 4.2 records the instrumentation time for BlackParrot's core units for an estimation of the algorithm speed with the logic size.

4.5 Accelerated Coverage Collection

4.5.1 CAM-based Group Coverage Engine (CCE)

With the development of smarter and more efficient test-generators [5, 132] most time-consuming parts of the hardware fuzzing loop are DUT simulation and coverage collection. FPGA prototyping is already a popular checkpoint in design verification exercises, and can be adapted to fuzzing to considerably improve the iteration times. However, as coverage metrics mature to increase fuzzing accuracy, the corresponding processing load increases and can create a bottleneck in the fuzzing loop. While toggle-coverage can be implemented with almost linear scalability, as we move to group-coverage, we face an exponential explosion of possible covergroup values that need to be collected and processed [100]. Previous implementations have opted to convert N-bit covergroups to 2^N-bit toggle-maps, each bit representing one possible combination of the N-bit number. These toggle-maps aggregate the activated values of the covergroup during emulation and are post-processed after running the test. Due to the exponential growth of these toggle-maps, various methods have been employed to reduce the number of instrumented coverpoints within a group. These efforts range from reducing the total number of coverpoints by filtering-out the ones that are not explicitly marked as a control signal to hashing coverpoints together to create smaller covergroups[77]. While these methods enable an imperfect implementation of group-coverage which is an upgrade on simple toggle-coverage, they still suffer from scalability issues, and diminish the coverage quality by throwing out important coverage information by either ignoring coverpoints or due to hashing collisions when encoding the covergroup.

To avoid sacrificing coverage accuracy due to hashing collisions, and to significantly improve the scalability of covergroups, we propose specialized CAM-based Coverage Engines (CCE). As shown in Figure 4.3, CCEs sample values of latency-aligned covergroups, record only

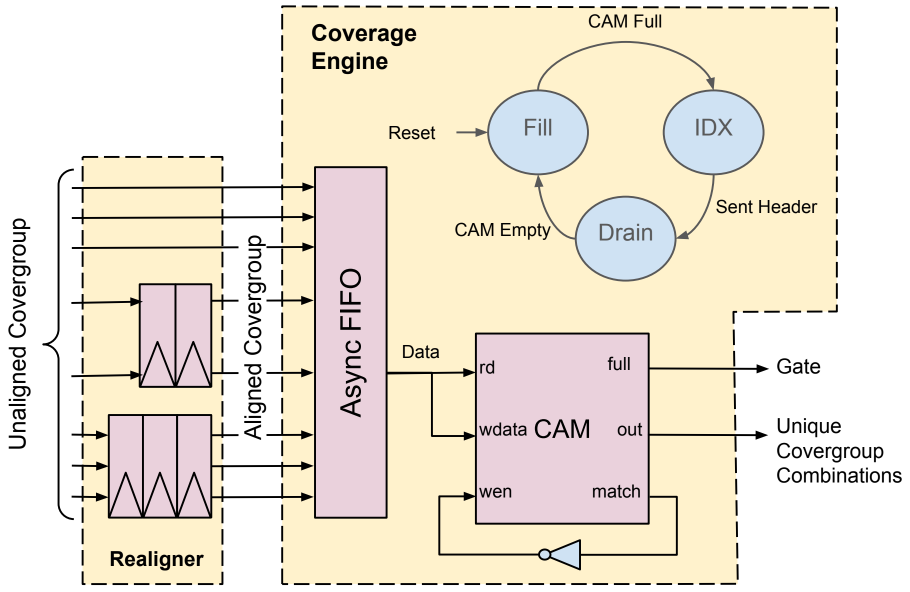


Figure 4.3: To assemble the latency-aligned covergroups, coverpoints are fed into a series of DFF chains corresponding to their interfered instrumentation depth. Once aligned, the covergroup is processed by a CAM-based group coverage engine where only unique combinations are stored into the CAM. Once the CAM is fully drained, the DUT clock is ungated and the emulation will resume. The use of CAMs alongside clock-gating enables the runtime processing of covergroups and eliminates the need for 2^N-bit toggle-maps.

unique values in their Content-Addressable Memory (CAM), and transfer them to the host control program after necessary clock domain crossings. By frequently draining the recorded covergroup values during emulation runtime, we eliminate the need to use 2^N-bit sized toggle-maps to hold the coverage information in RTL until program termination. To ensure cycle-accuracy and avoid perturbing DUT emulation when coverage streaming causes backpressure, the CCE employs a clock-gating mechanism to gate DUT execution while it streams the covergroup data to the host, and when the CCE is ready to sample again, it deasserts the gate to the DUT clock to resume emulation. By using clock-gating we ensure the DUT will not experience any extra emulation cycles due to the coverage collection overhead. This way, CCEs enable cycle-accurate runtime collection of covergroup values with minimal overhead to the emulation system, opening up FPGA acceleration of fuzzing for more complex coverage and larger designs. Using CAMs in place of toggle-maps allows for a much smaller storage overhead since the CAMs are also continuously draining throughout execution. Moreover, when an adequately deep CAM is used (suitably for the chosen width of the covergroup), the likelihood of the CAM filling up can be reduced. This is because even though there exists 2^N possible combinations for each covergroup, while the goal of fuzzing is to achieve a high coverage over many iterations, it's very rare that a single test generated by the

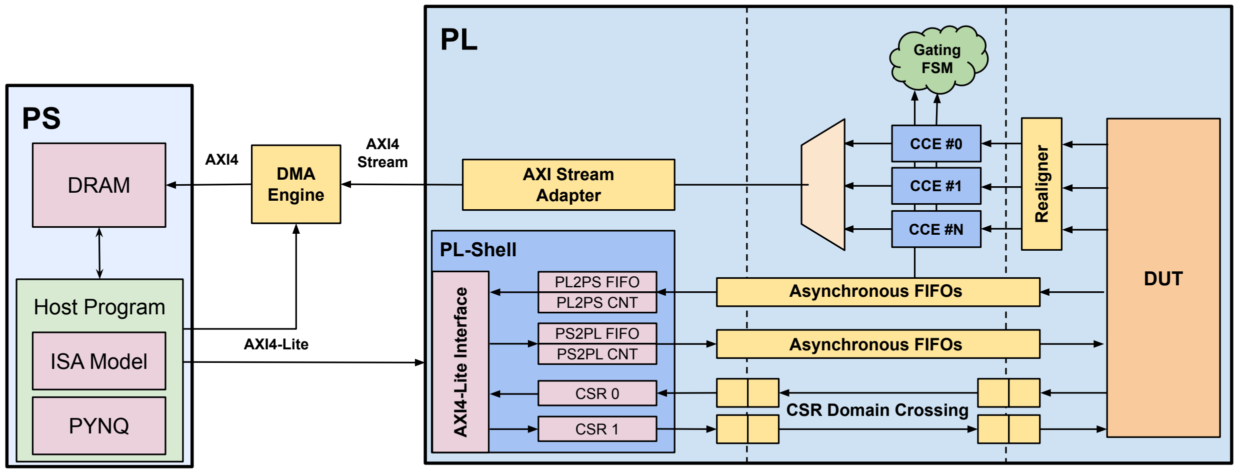


Figure 4.4: The integration of CCEs into Condominium system enables accelerated and non-intrusive collection of latency-aligned group coverage from an instrumented DUT. CCEs collect unique coverage data and stream them to the PS DRAM using a high-throughput AXI-STREAM DMA engine to create an agile verification loop. By plugging the CCEs into Condominium's clock-gating logic, coverage data can be collected and processed without perturbing the DUT emulation.

fuzzer will achieve a number close to that 2^N possibilities, but they will usually be hit over many iterations of fuzzer generated programs. This enables users to have a dynamic storage capacity for covergroups that can be tuned based on their performance, utilization, and streaming bandwidth considerations.

4.5.2 Condominium Integration

To enable accelerated and non-intrusive collection of latency-aligned group coverage data, the CCEs are integrated into Condominium's emulation infrastructure. Each CCE takes in the aligned covergroup representing on of DUT's instrumented units and records unique combinations during DUT execution cycles. When a CCE is full, it signals the Condominium clock-gating logic to pause DUT execution. During the draining phase, an arbiter picks a CCE to be drained and streams the coverage data alongside a header packer indicating the CCE information to the PS. When all CCEs are drained, Condominium un-gates DUT clock to resume emulation. Figure 4.4 shows the integration of instrumented DUT into Condominium for accelerated coverage collection.

While the PL-shell provides general PS-PL handshake, and AXI4 ports provide direct DRAM access, there needs to be a separate route for high throughput streaming of cycle-accurate

RTL information to the PS to reduce the gated DUT cycles gating caused by the CCEs and execution data backpressure. To this end, the covergroup data stream is routed to an AXI-DMA controller [177] through an AXI-STREAM interface [9]. The AXI-DMA engine is controlled by the PS through a GP-AXI4-LITE interface and is connected to the a DRAM allocated buffer by a HP-AXI4 client port. During emulation, the PS initializes the AXI-DMA engine and allocates a CMA buffer for data collection, then the PS continuously issues transfer commands to the DMA engine to write the information stream to the CMA buffer. Once the buffer is full, PS gets notified by an interrupt or by polling the DMA engine's status registers, reads and processes the buffer information, and issues another transfer command. By using the AXI-DMA controller we achieve an 18x speedup in emulation speed compared to streaming the coverage information over a simple FIFO interface. Also, to assist with runtime verification, we can transmit the runtime instruction commit information through the AXI-DMA for cross-comparison with a golden ISA model hosted on the ARM core.

4.6 Evaluation

This section evaluates the practical application of employing high-fidelity coverage for accelerated hardware fuzzing. We inspect slowdown-utilization trade-offs for proposed coverage engines and compare implementation feasibility with previously proposed logic. We inspect case studies on BlackParrot where the high-fidelity metric shows a unique capability to uniquely identify activated buggy states of the hardware. Finally, we leverage conventional test-generation and decision-making algorithms to implement a simple coverage-guided fuzzing loop and inspect the effect of guiding the loop with high-fidelity coverage compared to previously proposed coverage metrics.

4.6.1 FPGA Implementation

This section evaluates the FPGA-acceleration of the proposed coverage metric. We use the BlackParrot RISC-V core [115] as the DUT, and instrument its major submodules for targeted coverage feedback. Figure 4.5 depicts a stacked bar-graph of the relative latency distribution among the coverpoints within the module's instrumented covergroup. As evident from the figure, most covergroups have a significant portion of their constituent coverpoints with a relative latency of 1 or more cycles. This observation is important, because in an unaligned covergroup, those coverpoints will result in false-positive and false-negative changes in coverage that do not correlate with a change in newly activated RTL datapaths. The instrumented BlackParrot core is implemented on a Zynq 7000 SoC ZC706 Evaluation Kit [174]. The ZC706 uses a python-based PYNQ environment to provide the APIs for memory allocation, writing the bitstream to the PL, and configuring PL peripherals such as the AXI-DMA controller. The PL hosts the BlackParrot core alongside the CCEs, the gating logic, the AXI-Stream DMA adapter, and the PL-shell. For the purposes of these experiments, the AXI domain is clocked at 160MHz and the DUT domain is clocked at 40MHz.

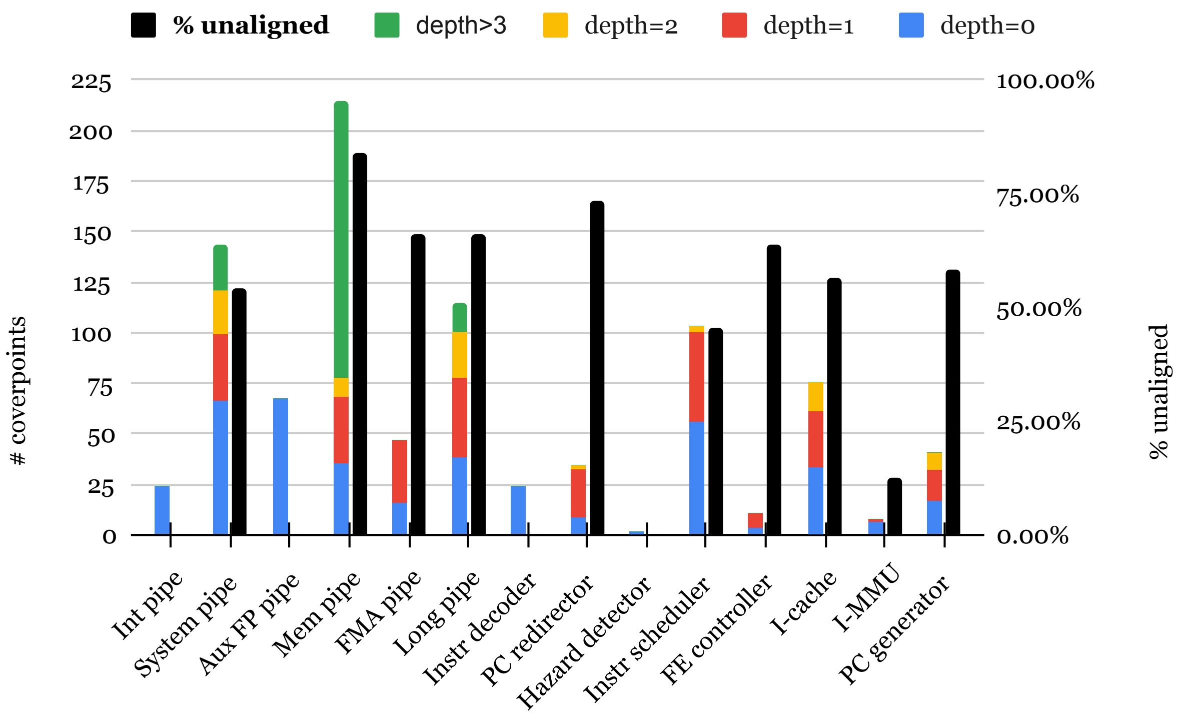


Figure 4.5: A breakdown of coverpoints at recorded relative latencies within each covergroup corresponding to BlackParrot modules. The Y-axis depicts both the number of coverpoints, categorized by their corresponding latency depth of 0, 1, 2, or 3+, and also the percentage of unaligned coverpoints within each covergroup before realignment.

Figure 4.6 showcases how as covergroups grow in width for more complex designs, the contemporary method of using toggle-maps [77] for group coverage processing quickly burns through FPGA utilization while CCEs offer a much more practical overhead. Toggle-map group coverage is implemented using simple dual-port 64-bit wide block RAM(BRAM) units. This is because most Zynq-7000 series, like most FPGAs, do not natively support bit-wise write operations in BRAMs, so single bit updates can be effected at a throughput of 1 update per cycle by simultaneous reading and writing of a dual-port BRAM. Since toggle-maps grow exponentially with covergroup width, so does their BRAM utilization, quickly outgrowing the ZC706 utilization past 24-bits rendering them impractical to be used for group-coverage collection in more complex designs. CAMs on the other hand utilize the FPGA LUT resources and grow linearly with covergroup width, allowing them to easily be coupled with the DUT for coverage collection.

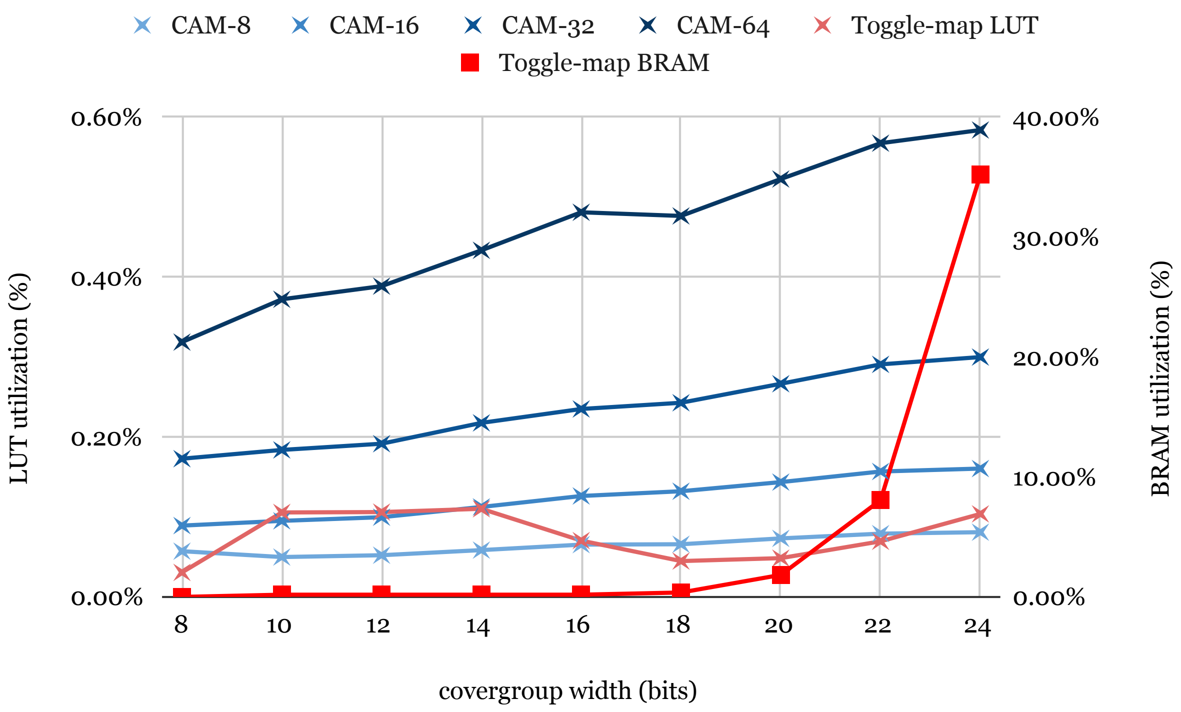


Figure 4.6: Representation of the tradeoff between the use of CCEs with varying CAM depths vs. use of BRAM-based toggle-maps for various covergroup widths (on X-axis). The left Y-axis represents LUT utilization on ZC706 for both CCEs and toggle-maps, while the right Y-axis represents BRAM utilization for toggle-maps. Note that CCEs do not consume BRAM resources on the FPGA.

Figure 4.7 showcases the tradeoff between FPGA emulation slowdown caused by CCE clock gating and their area utilization share when changing CAM depths used for coverage collection of the instrumented BlackParrot. It is evident that as CAM depths increase from very small sizes, the likelihood of the program hitting more unique coverage values than the depth supports reduces, resulting in a significant decreases in emulation slowdown for bigger depths. On the other hand, the utilization overhead for CCEs and the entire emulation system grows linearly with CAM depth. This results in a sweet spot where by using 24 entry CAMs we can conduct FPGA accelerated coverage collection at only 1.75x gating slowdown with CCEs consuming a share of 10% and 22% of entire system's LUTs and FFs respectively. Designers can optimize this tradeoff by preliminary testing of their target DUT with an initial CAM configuration, recording utilization and gating slowdowns resulting from CCE gating, and tuning each CCE CAM based on their covergroup widths and how often

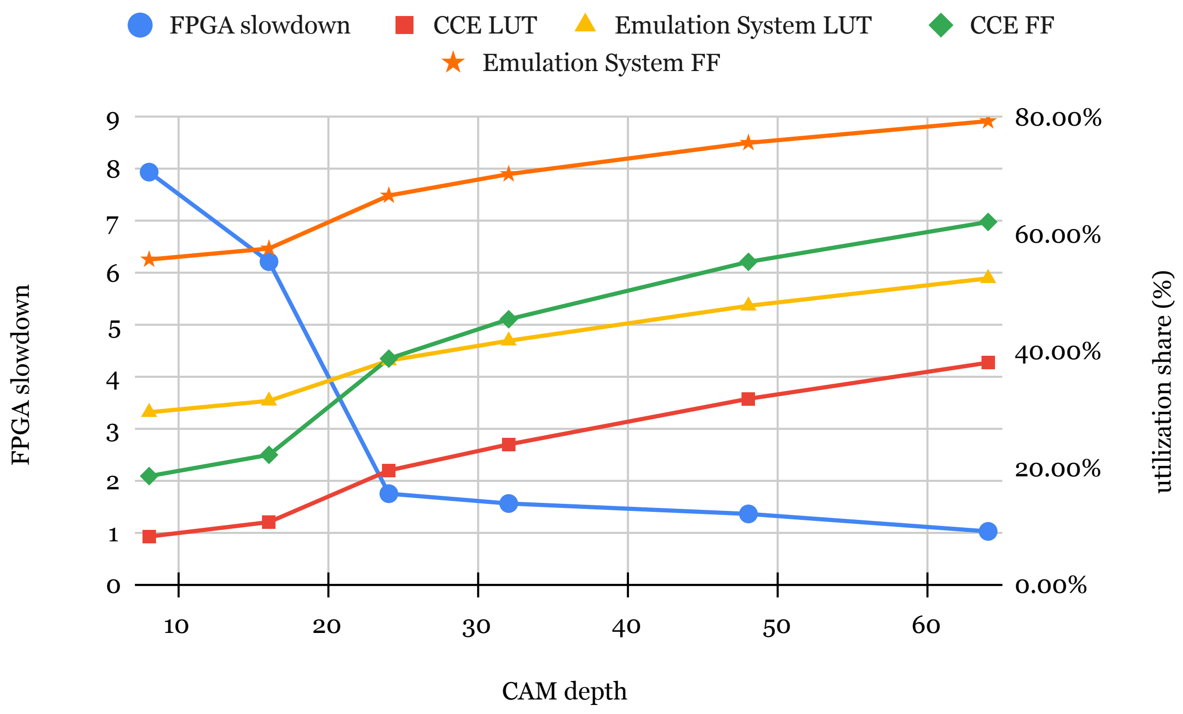


Figure 4.7: A representation of the FPGA slowdown-utilization tradeoff for varying CAM depth sizes for BlackParrot coverage instrumentation on ZC706. The left Y-axis depicts FPGA emulation slowdown due to DUT clock gating compared to the baseline of FPGA emulation without coverage collection. The right Y-axis shows the share of CCE LUT and FF utilization in the emulation system.

they contribute to emulation clock gating.

4.6.2 Case Study: BlackParrot Pipeline

In this case study we inspect the coverage instrumentation of BlackParrot's pipeline and showcase how latency-aligned group-coverage provides the necessary accuracy in identifying newly activated, potentially buggy datapaths that otherwise could not be distinguished through unaligned coverage metrics. BlackParrot's back-end (*BE*), depicted in Figure 4.8, is responsible for non-speculative execution of RISC-V instructions. It receives a speculative PC-instruction pair from the front-end, decodes it, feeds it to various execution pipes, and picks the correct result from the corresponding pipe and writes it back to the register-file. Once decoded and dispatched for execution, the instruction is fed to all the execution pipes with various latencies and the result is picked from the correct pipe in the corresponding pipeline stage based on its decoded information. The BlackParrot pipeline consists of 6 main pipes:

- **Integer pipe:** For 1-cycle simple ALU operations.
- **System pipe:** For 1-cycle CSR operations and trap handling.
- **Auxiliary floating-point pipe:** For 2-cycle floating-point conversion operations.
- **Memory pipe:** For 2/3-cycle integer and floating-point memory operations, data-cache handling, and cache-coherence communication.
- **FMA pipe:** For 4-cycle integer multiplication and 5-cycle floating-point arithmetic operations.
- **Long pipe** For dynamic latency integer and floating-point division and square-root operations.

This case study presents a realistic, artificially injected bug. The bug is due to a floating-point load (*FLD*) instruction, a memory operation, getting marked as both a memory and a floating-point instruction. This causes the outputs of both memory and FMA pipe to be valid for *FLD* in different cycles, and since the FMA pipe has a higher latency, its invalid result will override the correct result from the memory pipe. Looking at the coverage instrumentation of the part of pipeline responsible for generating the floating-point write-back data, we see that there are 5 successive MUXes at every pipe's output in successive pipeline stages. When the covergroup is assembled, the control signals of each MUX is latency-aligned relative to its distance to the write-back data. For the case study, we investigate the effect of latency alignment of this covergroup by running Core-Mark[55] on both the buggy and bug-free version of the BlackParrot pipeline and observing both coverage metrics. Figure 4.9 shows the progression of this covergroup activation during the Core-Mark execution. We can see that the latency-aligned metric shows an increment in coverage in the cycle the corrupt *FLD* result is being written back in the buggy pipeline. However, this increment does not happen during the execution of the bug-free pipeline. Since each possible combination in the floating-point write-back covergroup corresponds to a specific data-flow path, a combination that contains more than one set bit indicates that a pipe is overwriting the data from another pipe, and this results in the unique jump in coverage when the *FLD* bug is triggered. On the other hand, the latency-agnostic metric does not only behave completely different than the latency-aligned metric throughout the execution, but also does not show any contrast between the buggy and clean pipelines when *FLD* result is written-back. This case study shows how, as opposed to other coverage metrics, an accurate encoding of pipeline writeback paths can easily map erroneous pipe overwrites as unique covergroup values that can uniquely be identified when triggered in the RTL.

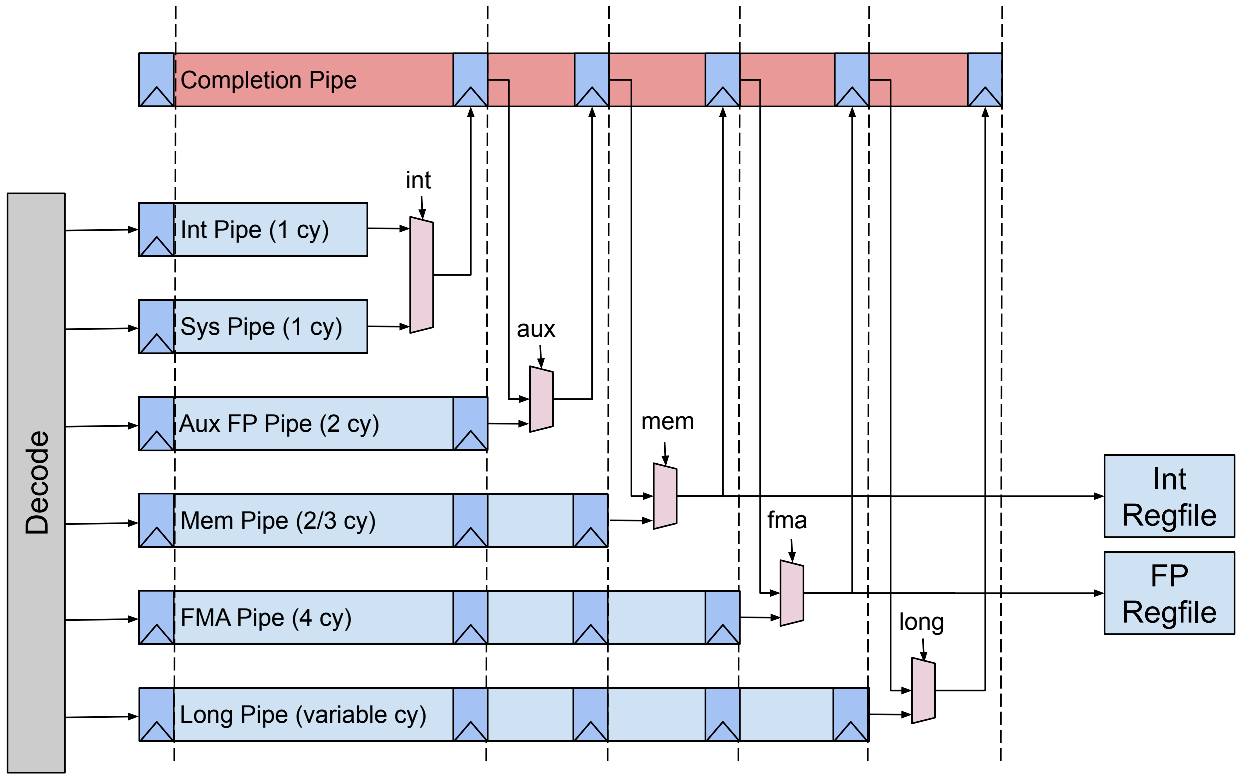


Figure 4.8: BlackParrot's execution pipeline. Each pipe is responsible for computing a RISC-V instruction type and the corresponding result is picked through a MUX chain for writeback. If due to wrong decoding, an instruction is fed into multiple pipes, the correct result can be overwritten by the misattributed and longer pipe.

4.6.3 Case Study: BlackParrot PC-Generator

In this case study we inspect the coverage instrumentation of the BlackParrot PC-generator and observe how the latency-aligned group coverage can identify unique, potentially buggy paths being triggered as a result of the alignment of asynchronous events. The BlackParrot front-end (*FE*) is responsible for speculatively fetching instructions from the memory and providing the BE with a stream of speculative PC-instruction pairs. The BE then inspects these pairs and can logically redirect the FE upon a PC misprediction or trap handling. The PC-generator, depicted in Figure 4.10, is implemented as a three stage pipeline. In the first stage, the PC-generator computes a speculative PC based on previous cycles or a BE redirection command, and starts fetching the instruction from the instruction cache. The instruction cache has a two cycle latency, so in the third stage, the FE can pass the PC-instruction pair to the BE. The default RISC-V instruction size is 32-bits, so instruction

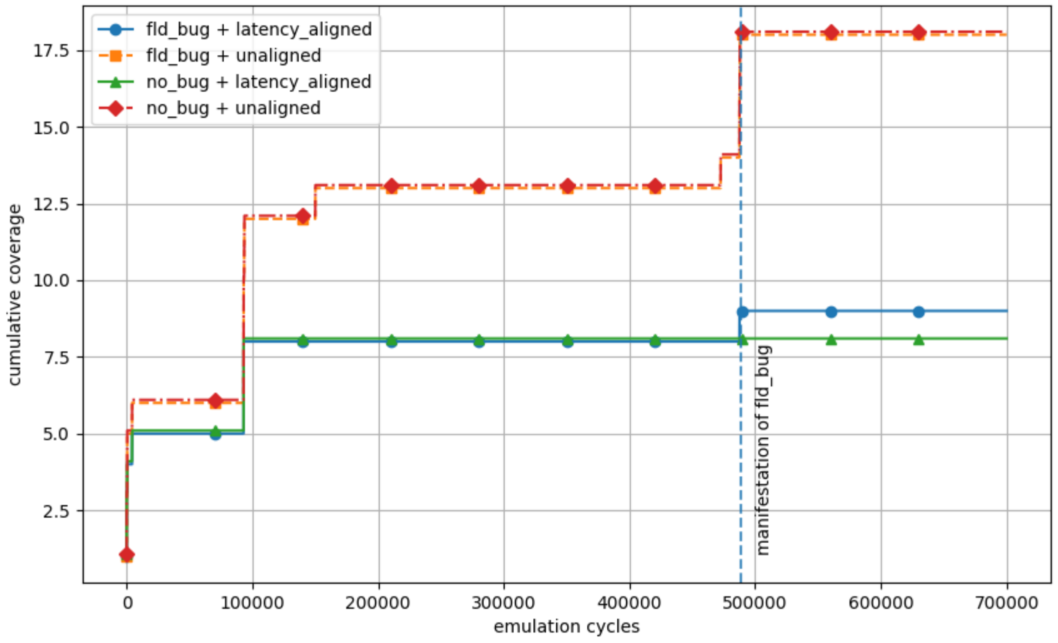
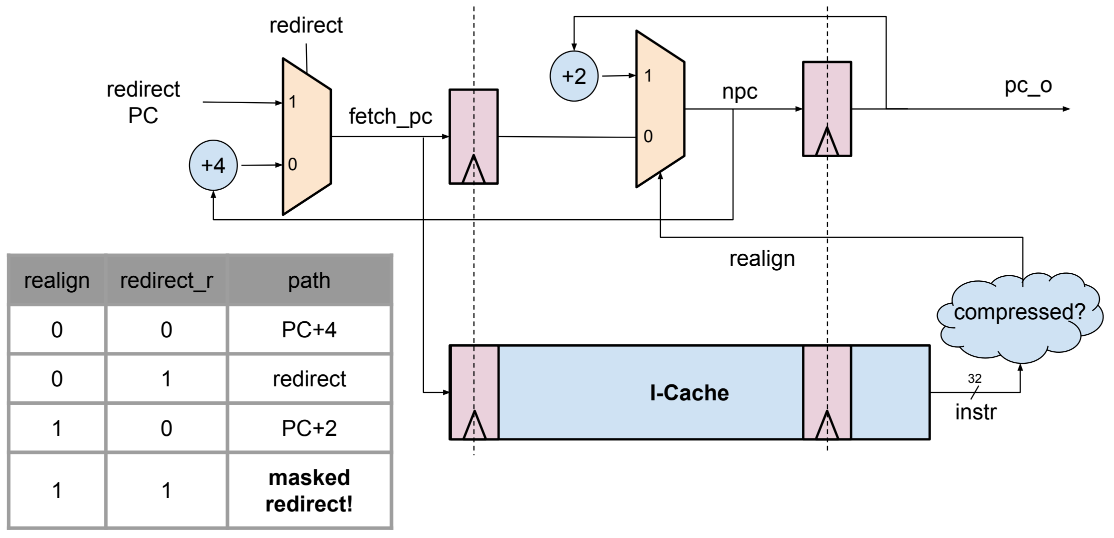


Figure 4.9: A representation of the cumulative coverage of the latency-agnostic and latency-aligned metrics for the BlackParrot pipeline's floating-point writeback result running CoreMark on both the buggy and fixed versions of the core. The latency-aligned metric shows a distinct increase in the coverage when program reaches the buggy *FLD* instruction while the latency-agnostic metric shows no divergence between the buggy and fixed *FLD* cycle.

cache also fetches 32-bit chunks from the memory. To optimize the memory footprint for program storage, RISC-V also supports 16-bit compressed instructions. To enable compressed instruction fetching in FE and also avoid bubbles, an instruction re-aligner is introduced to convert the 32-bit chunks of instruction cache fetch data into two 16-bit compressed instructions. The re-aligner sits at the third stage of the pipeline, and based on a rudimentary instruction decoding, decides if the fetched 32-bit chunk actually contains two compressed instructions. If so, it outputs the first instruction at stage three and sets next the PC at the second stage to point to the higher second half the 32-bit chunk, so it can output the second instruction in the next cycle.

However, the addition of the re-aligner introduces a bug where a BE exception, or any other type of PC redirection, can be ignored by the FE. The bug manifests exactly one cycle after receiving a BE exception where the PC fetched at the third stage is identified as a compressed



realign	redirect_r	path
0	0	PC+4
0	1	redirect
1	0	PC+2
1	1	**masked redirect!**

Figure 4.10: A simplified representation of BlackParrot's front-end including a 3-stage PC-generator paired with a 2-cycle I-cache. The PC-generator can be redirected by the back-end due to causes like traps and branch mispredictions. In this case study, an instruction realigner, upon fetching a compressed RISC-V instruction will set the next PC, resulting in the masking of a previously received redirection. This asynchronous event can be distinctly identified by the latency-aligned coverage.

instruction. At this stage, the re-aligner overrides the exception PC from the second stage, thereby killing the instruction cache fetch, and replacing the corresponding instruction with the top half of the previously fetched chunk. As can be seen in Figure 4.10, the relative alignment of these two asynchronous events, compressed fetch and BE exception, can be uniquely identified by a specific combination of the latency-aligned covergroup representing the PC-generation pipeline. Whereas, the timing-agnostic metric does not provide a clear feedback on how these events can interact with each other.

4.6.4 Fuzzing Experiment

To evaluate the efficacy of the proposed latency-aligned group-coverage metric for guiding a fuzzer towards better RTL exploration, we design an experiment to compare fuzzers that are guided by different coverage metrics. For test generation we employ *Cascade* [132], a black-box RISC-V test generator that produces valid RISC-V programs of arbitrary length with randomized and interdependent control and data flows. By using ISA pre-simulation to entangle the data and control flow of generated programs, Cascade can produce programs without dead-code and a runtime self-verification mechanism, where a wrong computation value can result in early program termination. Cascade is also highly configurable and can take in a probably array of RISC-V instruction types in order to generate different flavors of programs. In this experiment we will use the probability array as a means to steer the fuzzing experiment towards better coverage.

To use the coverage feedback to guide the Cascade test generator, inspired by *MABFuzz* [60], we employ a multi-armed bandit (MAB) algorithm which is a reinforcement learning method where a decision maker iteratively selects one of multiple choices, *arms*, to maximize the cumulative reward received from pulling said arms. MAB algorithms aim to use previously gathered knowledge on arm rewards to strike a balance between exploitation of previously known fruitful arms, and exploration of other arms to get more information about their expected payoffs. The coverage-guided fuzzing experiment is a non-stationary environment with diminishing returns as iterative testing yields less newly covered states over time. So, as to let the MAB algorithm to reflectively react to this changing environment, we use a sliding-window EXP3 [17] configuration, shown in Algorithm 2, that gradually discards coverage rewards gathered in older iterations and puts a higher emphasis in more recent information. Each MAB arm is assigned to a specific RISC-V probability array configuration, and in each iteration based in previous coverage rewards, the MAB algorithm decides on an arm to pull, commanding Cascade to generate a test with the corresponding probability configuration. The new coverage resulting from running the test on DUT is fed back into the MAB algorithm as through a sigmoid reward function and updated the probabilities of pulling the arms. By coupling Cascade and a MAB agent, we have designed a simple fuzzer that aims to maximize the total cumulative coverage by choosing between different arms, different configurations of RISC-V programs, based on the coverage rewards that these arms have yielded in previous iterations.

We employ four instances of this fuzzer, each being driven by a different coverage metric: latency-aligned group-coverage, unaligned group-coverage, toggle-coverage, and no coverage. BlackParrot is used as DUT and its instruction scheduling unit is instrumented for coverage collection. Then, we launch each fuzzer independently and compare the cumulative activated RTL datapaths over 10k iterations. As shown in Figure 4.11, the proposed latency-aligned group-coverage outperforms the unaligned group-coverage by 5%. We can also see while the unaligned group-coverage offers a slight improvement over unguided fuzzing in the initial iterations, other metrics fail to offer any special guidance for the experiment. As observed by studies of efficacy of coverage-guided fuzzers [29], previously introduced coverage metrics do not offer improvements on the bug finding ability of said fuzzers. This is confirmed by our experiment where fuzzers guided by latency-agnostic toggle and group coverage metrics do not produce an excess of explored datapaths compared to black-box fuzzing. On the other hand, the proposed latency-aligned group-coverage has improved the design exploration capability of an existing black-box fuzzer, Cascade, by steering it with a metric that establishes a clear mapping to explored RTL datapaths. This experiment demonstrates that future efforts on developing more mature and complex fuzzers can leverage high-fidelity coverage to fully flourish the effectiveness of their exploration algorithms for the purposes of hardware

Algorithm 2 Sliding-Window EXP3 Multi-Armed Bandit

```
Require:
    Number of iterations $T$
    Number of arms $K$
    Window length $W$
    Exploration rate $\gamma \in (0, 1)$
    Sigmoid constant $s > 0$

 1: $\forall i \in \{1, \ldots, K\}: \; w_i \leftarrow ones(W)$        ▷ initialize arm weight windows
 2: for $t = 1$ to $T$ do
 3:     for $i = 1$ to $K$ do                                          ▷ update arm probabilities
 4:         $p_i \leftarrow (1-\gamma)\frac{\Pi_{k=1}^{W} w_{i,k}}{\sum_{j=1}^{K} \Pi_{k=1}^{W} w_{j,k}} + \frac{\gamma}{K}$
 5:     end for
 6:     $A_t \sim choice(p_1, \ldots, p_K)$                            ▷ sample arm based on probabilities
 7:     $r_t \leftarrow pull(A_t)$                                     ▷ pull arm and observe reward
 8:     $r_t \leftarrow \frac{1 - exp(-sr_t)}{1 + exp(-sr_t)}$           ▷ normalize reward $\in [0, 1)$
 9:     $\hat{x}_i \leftarrow \begin{cases} \frac{r_t}{p_i} & \textbf{if } i = A_t \\ 0 & \textbf{otherwise} \end{cases}$   ▷ EXP3 update
10:     for $i = 1$ to $K$ do
11:         $w_i \leftarrow \{w_{i,2}, \ldots, w_{i,W}\}$               ▷ remove oldest weight
12:         $w_i \leftarrow append\,(w_{i,W} \,.\, exp(\frac{\gamma \hat{x}_i}{K}))$   ▷ append new weight
13:     end for
14: end for
```

verification.

4.7 Related Work

A variety of research has been conducted in the area of hardware fuzzing, verification, and hardware coverage metrics. The initial work on verification through hardware fuzzing was motivated by the success of software fuzzers, such as *AFL* [180], and also the development of random program generators, such as *RISCV-DV*, that have opened the door for random test verification in hardware design process. To this end, a range of research has been conducted on improving and innovating on different parts of the iterative fuzzing process. These efforts include, but are not limited to, designing better random program generators, developing better test mutation techniques, leveraging various learning-based methods for guiding test generation, specializing fuzzing for finding specific types of data and timing

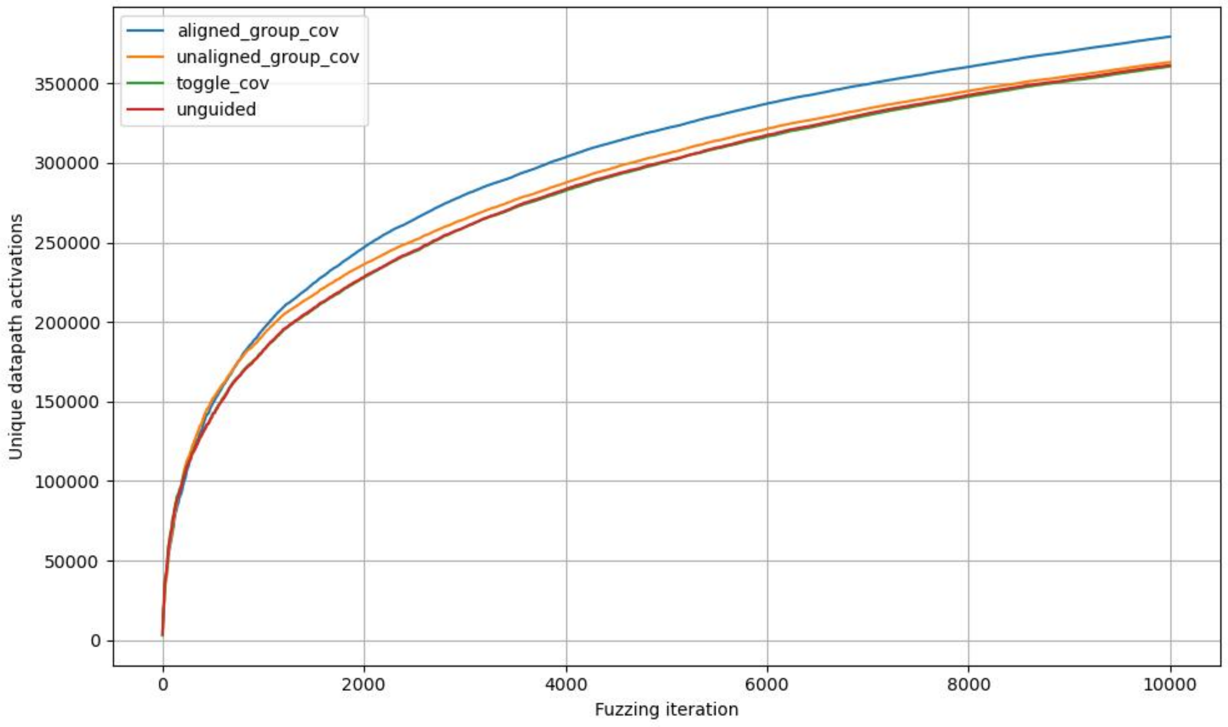


Figure 4.11: A comparison of four fuzzing experiment driven by different coverage metrics based on BlackParrot's scheduler unit. While the fuzzer guided by the proposed latency-aligned group-coverage achieves a higher number of unique datapath activations over 10k tests, the fuzzers guided by unaligned group-coverage and toggle-coverage struggle to over-perform the unguided fuzzer.

vulnerabilities, and fusing formal verification methods with fuzzing. While many of these works have proposed a variety of coverage-guided fuzzing methods, few have focused on the effectiveness and performance tradeoffs of used coverage metrics for verification. While some works have proposed using more generic microarchitectural coverage metrics for fuzzing, such as MUX-based coverage, others have relied solely on RTL simulator provided metrics as a rough estimation of design exploration. Others have focus on application-specific coverage for purposes of ensuring ISA compatibility, ex. RISC-V CSRs, or custom designed coverage for tracking certain timing and dataflow behaviors, ex. interface and timing side-channel custom coverage.

In this work, we have focused on a microarchitectural coverage metric that aims to provide a generic representation of RTL datapath exploration. As previous surveys on hardware bugs [29] have concluded, most designer errors, in their final form, manifest as a faulty

Table 4.3: This work aims to enable high-fidelity group coverage metric that is both Reliable and Feasible. Previously proposed metrics lack the incorporation of RTL latency information in group-coverage definitions. Moreover, this work aims to solve the scalability issue of FPGA acceleration for group-coverage instrumentation, enabling high-accuracy, high-performance verification.

Work	Coverage Metric	Group-Coverage	Latency-Aligned	FPGA-Capable	HDL	Application-Agnostic
RFUZZ [99]	MUX Control Toggle	✗	✗	✓	FIRRTL	✓
DifuzzRTL [77]	MUX Control Register	✓	✗	✓ [0]	Verilog/FIRRTL	✓
TheHuzz [90]	VCS Coverage	✓	✗	✗	Verilog/FIRRTL	✓
Laeufer et al. [100]	Branch/Line/Toggle/FSM	✓	✗	✓ [1]	FIRRTL	✓
ProcessorFuzz [35]	RISC-V CSR transition	✓	✗	✗	Verilog	✗
WhisperFuzz [30]	Micro-Event Graph	✓	✗	✗	Verilog/FIRRTL	✗
This Work [30]	**Latency-aligned Group**	✓	✓	✓	**Verilog**	✓

[0] Covergroup hashing is used to make FPGA prototyping possible to mitigate the scalability issue of exponentially growing toggle-maps, also embedding inaccuracies in the metric due to hashing collisions.
[1] Group coverage is mentioned to be a limitation of this work due to inefficiency of instantiating exponential number of counters for each possible value.

logical section in a certain datapath. So we believe that by proposing a generic coverage metric that encourages maximizing datapath activations, we can provide a better guidance feedback for future fuzzing efforts. Furthermore, we believe when it comes to iterative verification, speed can be one of the main factors contributing to verification efficacy. The more testing iterations a fuzzer can conduct in a certain time-frame, the more likely it is to trigger a bug. Since prior work on fuzzing has been mostly focused on developing better fuzzing algorithms, there has not been much thought poured into optimizing their iteration speed, and FPGA acceleration has remained mostly an afterthought. While, outside the scope of fuzzing research has been conducted accelerating coverage collection using FPGAs [100], they have focused on conventional simulator metrics and avoided group-coverage that scales exponentially. To ensure our proposed high-fidelity coverage can be used in an high-performance fuzzing environment, we focused on enabling FPGA acceleration by designing the specialized CAM-based engines and their leverage of DUT clock gating.

4.7.1 FPGA Acceleration of Coverage

Laeufer et al. [100] proposes automated synthesizable coverage metrics that include line, toggle, and finite state machine (FSM) coverage for Chisel-based [22] designs. They leverage FIRRTL [80] compiler passes to insert *cover* primitives for capturing branch, line, toggle, FSM, and ready/valid coverage. They synthesize their coverage metric through the FireSim [92] infrastructure and post-process the information captured from *cover* primitives alongside the static analysis of the RTL to generate said coverage metrics. Since they convert all forms of group-coverage, like FSM transitions, to exponential number of counters covering all possible combinations, they report limitations in implementing *cover-value* primitives, equivalent to group-coverage, due to state explosion for wider covergroups.

4.7.2 Coverage Metrics enabling Verification

Many projects have introduced coverage metrics to represent RTL exploration and guide the fuzzing effort. *RFUZZ* [99] introduced MUX toggle-coverage that formulates coverpoints as the sum total of MUX select signals and activations to be individual toggles. This approach, while easy to implement, treats hardware coverage like software branch coverage and ignores the complex concurrent relationships between various control signals in RTL. *DifuzzRTL* [77] proposes a type of the MUX group-coverage metric by instead instrumenting the upstream registers driving said MUXes. The key insight here is to reduce the number of coverpoints over those identified by *RFUZZ*. *DifuzzRTL* also proposed using techniques like hashing coverpoints together to downsize multiple covergroups to overcome the exponential scalablity issue. While *DifuzzRTL* improves on *RFUZZ* by highlighting the interplay of coverpoints by bundling them in covergroups, it introduces blind spots due to hashing collisions. Also, using exponentially sized toggle-maps for group-coverage implementation is not scalable during FPGA acceleration. Furthermore, because *DifuzzRTL* bundles coverpoints in different pipeline stages into a single covergroup without latency-alignment, the covergroups do not accurately represent the true testing progress.

DirectFuzz [34] proposed targeted gray-box testing with focus on specific design units to increase overall fuzzing performance. *ProcessorFuzz* [35] focuses on ensuring ISA compliancy of RISC-V processors by assessing the coverage over key RISC-V CSR value transitions and exploring different execution modes and associated state changes in RISC-V implementations. *TheHuzz* [90] aims to over-perform previous fuzzers by extracting a variety of VCS coverage metrics including statement, toggle, branch, expression, condition, and FSM coverage for guiding test-generation. Aside from relying on a commercial RTL simulator for coverage instrumentation, the sheer amount of the processed coverage data makes *TheHuzz* non-eligible for FPGA acceleration.

4.7.3 Coverage-guided Fuzzing

Other projects have focused on proposing decision-making algorithms for effectively utilizing the coverage feedback for test-generation to achieve faster and more efficient state space exploration. *MABFuzz* [60] introduces an algorithm based on Multi-Armed Bandit decision making, and *PSOFuzz* [37] proposes a Particle Swarm Optimization solution for dynamic test generation and mutation aimed at improving the effectiveness of state exploration. Both build upon *TheHuzz* by employing an algorithms that takes in the achieved coverage by independent fuzzing threads, and based on a reward function, accumulate information about expected rewards to make decisions on future test generation for maximizing the overall coverage. *WhisperFuzz* [30] is specialized for identifying timing security vulnerabilities through 2-stage fuzzing. In the first stage, *HyPFuzz* is utilized for generating a set of program seeds for diversifying the program bank. In the second stage, a data fields of instructions are mutated to reveal possible timing differences. Then another fuzzer, driven by a custom coverage metric designed to identity different timing patterns of RTL, analyzes the mutated seeds to find timing vulnerabilities. *Rostami et al.* [123] use a combination of LLM-based ISA learning, and RL-based learning of valid and interesting instruction generation based on a coverage-driven reward function. It uses the same VCS coverage metrics as *TheHuzz* but improves on it because in the terms of speed. *MorFuzz* [179] proposes runtime dynamic mutation of instructions in the processor pipeline aimed at generating more diverse instructions in simulation time. While it has the potential of generating more divers mutations, it's inherently slow and intrusive since it injects the fuzzer in the fetch to dispatch path of the RTL.

4.7.4 Fuzzing Hardware like Software

Trippel et al. [157] propose exploiting the well-explored practices in software fuzzing by targeting the generated software functional model of a given hardware design, through *AFL*, a software fuzzing infrastructure. While this approach relies on well-established software fuzzers and therefore has good potential for finding corner cases, the process is equivalent to performing fuzzing aimed at maximizing HDL line coverage which is not suitable for acceleration.

4.7.5 Black-box Fuzzing

As mentioned before, the ineffectiveness and overhead of coverage-guided fuzzing has led many researchers to solely focus on developing high-quality random test generation techniques, also known as black-box fuzzing. *Cascade* [132] focuses on generating high-quality and dead-code free test programs by carefully controlling the control flow in the generated programs. One existing challenge with proposed techniques for test generation and mutation is that the while fuzzer may expect to explore a new design space based on a new addition or mutation to an existing test, if said change happens in parts of the test program which are never executed, then the fuzzer will not see a corresponding coverage reward for said change and will miss on learning valuable information for future exploration. *Cascade* addresses this issue by creating by fusing program's data and control flow. By running a preliminary ISA simulation on a initial skeleton of the test program, it can fill-in immediate address and values to ensure a deterministic execution flow of the program and create self-verification mechanisms where a wrongly computed value can result in an early termination.

4.7.6 Targeted Coverage Verification

BugsBunny [119], using the same coverage metric as *DifuzzRTL*, proposes a directed fuzzing approach where a dependency tree is generated for a target signal and guides the fuzzer based on the coverage metric and distance to the target. *BugsBunny* uses module-trimming to directly manipulate the target by controlling the unit's input ports. While this approach is effective for targeting certain coverpoints, it's inherently intrusive as it directly asserts RTL wires instead of driving them using input programs. *HyPFuzz* [36] uses a combination of coverage-guided fuzzing, *TheHuzz* in this case, and formal verification tools, JasperGold [70], to balance the explorability power of formal alongside the state exploitation power of fuzzers. A scheduler is designed for picking coverpoints to target and constantly switching between fuzzing and formal based on progression rate. *Design2Vec* [159] proposes an convolution based neural architecture that embeds the RTL and provides a model for providing the coverage achieved for a set of input instructions. Such models have great potential to be used for fuzzing as, with a high-fidelity coverage metric, they can be used to predict instructions to efficiently explore the state corresponding to the possible values of the coverage metric. By learning the semantic abstractions of the RTL, the next generation of fuzzers can mostly abandon random testing and instead develop a loop that iteratively goes through target covergroup values, corresponding to RTL datapath activations, and based on acquired RTL knowledge generate inputs to trigger and verify them.

4.7.7 Bug Injection and Fuzzer Evaluation

Encarsia [29] is an automated RTL-level bug-injection tool aimed at evaluating the efficacy of recent plethora of developed hardware fuzzers by offering a standardized way of creating buggy hardware benchmarks through methodological bug injection. To design the tool, the researchers conducted a survey of several previously reported bugs on various RISC-V processors and concluded that, in terms of manifestation in RTL, almost all of those bugs reduce to two categories: *signal mix-ups* and *broken conditionals*. Bugs in both these categories can be effectively triggered by activation of their corresponding RTL datapath and capturing the wrongly computed output in verification, which is why we argue a coverage metric that incentivizes fuzzers to maximize new datapath activations can help with revealing these categories of hidden hardware bugs. Furthermore, they conducted evaluations of both the performance bottleneck of coverage collection and the ineffectiveness of conventional coverage metrics in guided contemporary fuzzers.

4.8 Discussion and Future Work

4.8.1 Evaluation of Latency-Aligned Coverage

The latency-aligned group coverage metric is a high-fidelity coverage metric which has only recently seen feasibility with the advent in synthesizable coverpoints which enables orders of magnitude speedups. The true value of a high-accuracy, and resource-efficient coverage metric for verification can be unleashed when coupled with intelligent fuzzers with program generators that can quickly adapt to the changing coverage utilization to target the fraction of the state space that are yet to be exercised. Such a fuzzer must be sensitive to what instruction sequences influence what coverpoints and be able to align itself to maximize coverage over subsequent iterations. With the advent of more sophisticated machine-learning based fuzzers and models predicting instruction sequences targeting certain coverpoints [159], this intuition can be learned and used to generate precise instruction streams targeted at quickly maximizing accumulated coverage. To this end, more research is needed to couple more advanced fuzzing algorithms with high-fidelity coverage metrics.

4.8.2 One-to-One Coverage Mapping

While latency-alignment in group-coverage metrics guaranties a deterministic mapping from coverage value to activated RTL datapaths, depending on the type of chosen coverpoints, multiple values may point represent the same activated path. This is specially apparent when a majority of coverpoints consist of MUX select signals, given the fact that MUXes effectively render coverpoints upstream of the discarded input irrelevant to the output. This results in a coverage combination state space that is bigger than all the possible activated paths, with multiple combinations pointing to the same path. In this work we have focused on the general effect of latency-alignment in providing deterministic and reliable guidance for fuzzing assuming generic coverpoints that represent any interesting control net within the design. More research is needed to inspect the effect of creating a one-to-one mapping for covergroups consisting exclusionary coverpoints, such as MUX selects, on the fuzzing performance. While establishing a one-to-one mapping can eliminate some false-positives in the experiment, it also incentivizes multiple activation of a same datapath, providing more chances for manifesting the proper inputs for triggering a bug on the same path.

4.8.3 Progressive Coverage

While many coverage metrics concern themselves on whether certain selected coverpoints have toggled, they do not provide any information on how close the fuzzer has come to activate them. This binary situation makes the job of fuzzers trying to explore new states more difficult as they cannot learn on what actions have the potential of increasing the likelihood of toggling a certain coverpoint. By incorporating information on the progress towards toggling coverpoints through incorporating precursor intermediate nets affecting said coverpoints we can provide a smooth sense of progress to the fuzzer, potentially increasing fuzzing performance. We aim to further investigate the efficacy of implementing such progressive coverage for better coverage-guided fuzzing. Also, this approach can potentially bias the fuzzer to not only care about whether a coverpoint toggles, but if it toggles correctly by exhausting combinations of operand signals that the coverpoint is derived from.

4.8.4 BRAM-based Alternative to CCEs

The use of CAMs is motivated from the need to reduce duplicate covergroup values which can significantly lower the bandwidth consumption on Condominium. CAMs can be inefficient when synthesized on FPGAs, so the alternative to CAMs is to employ block-RAMs (BRAMs). Wider covergroups have little advantage in reducing duplicates when CAMs are used, since the probability of duplicate data reduces significantly with a bigger state space. BRAMs are then the natural alternatives. Of course, with BRAMs, their storage window, before data needs to be drained, is equal to the BRAM depth which could lead to considerable and emulation slowdowns. This slowdown can be potentially offset by instantiating a larger coverage buffer that can be simultaneously written by DUT and drained through the DMA route, resulting in some mitigation of potential slowdowns.

4.8.5 Extending to other Cores and HDLs

Although initially this work was motivated by verification of the BlackParrot core, the proposed methods are DUT and HDL agnostic. To demonstrate the flexibility of high-fidelity coverage and provide more insights for improving fuzzing efficacy, more experimentation is needed to extend this work for other RISC-V cores, such as CVA6 [181] and Rocket [14], and languages like Chisel [22].

Chapter 5

Conclusion

With the continued growth of open-source hardware development ecosystem, introduction of new design tools and libraries, and development of a diverse plethora of cores and application-specific accelerators, the need for delivering a clean design on a reduced time-to-tapeout cycle can be determinative in adoption of new designs. This dissertation explores challenges facing hardware engineers as design verification needs evolve throughout different stages of the development process. We present solutions that streamline these challenges by offering frameworks and techniques for agile functional verification and performance optimization.

In chapter 2, we inspect sources of bottlenecks in hardware development. We argue as designs mature and move to continuous regression testing using long and complex real-life benchmarks, the slow nature of software RTL simulators acts a pinch-point causing long iteration times and wasted engineering hours waiting for simulation results. The move to FPGA prototyping for accelerated emulation has the potential of reducing iteration times but suffers from very limited visibility into the design, and with a lack of emulation reproducibility, debugging FPGA issues can quickly become a whack-a-mole leading to repeated bitstream generations. We observe how, first time silicon success rates remain low, functional verification consumes a significant portion of engineering time. While formal verification suffers from scalability issues, longer benchmarks can be slow to debug and engineers have to rely on manually browsing waveforms to find silent bugs, highlighting the need to automated bug localization methods. Similarly, as conventional benchmarks provide standardized programs for stress testing different functional aspects of hardware, aggregated high-level metrics, like IPC or built-in ISA event counters, fail to provide any indication on the RTL and instruction sources contributing to benchmark performance bottlenecks. Modern performance profilers have been proposed by offering fine-grained attribution of execution cycles to stall sources and program instructions. However, these profilers process a high-volume of data that has to be processes by the emulation environment without perturbing the benchmark execution flow. Both of the proposed solutions for functional verification and performance optimization suffer from the same limitations of the performance-transparency tradeoff when migrating between software RTL simulation and FPGA emulation. We also introduce BlackParrot, an open-source Linux-capable RISC-V multicore. Much of the work in this dissertation was inspired by the need to develop agile design analysis during various stages of designing and maintaining BlackParrot in a large team with parallel threads of feature design.

In chapter 3, we present Condominium, a cycle-accurate FPGA emulation infrastructure that aims to provide both the acceleration and design visibility needed for leveraging the aforementioned functional verification and performance analysis methods. Built upon Zynq FPGA boards, Condominium decomposes design subsystems to be emulated on the PL fabric, while hosting the rest of the abstracted system and emulation control software on the PS ARM core. The two interface using a flexible PL-shell wrapper connected to the physical Zynq AXI ports and offering bi-directional memory-mapped CSRs and SB-FIFOs for communication. To enable the cycle-accuracy needed for reproducibility, the AXI, PL, and DUT clock domains are decoupled and a clock-gating state-machine is implemented for pausing DUT emulation upon PS backpressure. Also, the same clock-gating logic is used to provide access time guaranties between DUT and abstracted peripheral models hosted on the PS. Similarly, clock gating can be used to standardize system-call access time by enforcing the same timing models on the downstream peripherals they access. Using Condominium, we can easily perform cycle-accurate ISA cosimulation using PL-shell FIFOs to transmit execution metadata to a PS-hosted golden reference model for cross-comparison. This enables accelerate, reproducible bug localization for long benchmarks which results in significant reduction of manual debugging efforts, making Condominium a suitable candidate for integration into design's CI pipeline. Furthermore, Condominium was shown to enable unprecedented insight into microarchitectural performance by hosting fine-grained performance profilers. As opposed to relying on high-level metrics which do can only confirm pre-existing knowledge of RTL bottlenecks, Condominium enables per-cycle instruction and stall source attribution that gives users an accurate breakdown of when, in program, and where, in RTL, the design is wasting cycles waiting for data or a dependency to be resolved. We demonstrate this technique with a case study on adding a catch-up ALU to the BlackParrot pipeline upon discovering a high percentage of load-use dependencies.

In chapter 4, we present a high-fidelity hardware coverage metric for accelerated hardware fuzzing. As designs evolve into their final stages of development, random fuzz testing is employed to reveal corner-case bugs and vulnerabilities that can often be missed by conventional hand-crafted benchmark. Coverage-guided fuzzers rely on a hardware coverage metric as the sole microarchitectural feedback on newly activated logic and guide it towards better design exploration. We explain limitation of previously employed coverage metrics and how they often provide an ambiguous feedback on design exploration and are hard to prototype for accelerated fuzzing. We propose high-fidelity coverage, a latency-aligned group-coverage metric that present a deterministic mapping from coverage values to activated RTL datapaths. We provide an automated instrumentation algorithm that parses SystemVerilog designs and extracts coverpoints and their corresponding relative latencies used for assembling the high-fidelity coverage. We design specialized CAM-based coverage engines, that by integrating into Condominium, collect unique covergroup combinations from the DUT and stream them to the host for to be used as feedback for guiding the fuzzer. The CCEs enable cycle-accurate DUT coverage collection by resolving the exponential scalability issues of naively implementing covergroups and avoiding usage of remedies that introduce inaccuracies into the system such as covergroup hashing and pruning. We evaluate high-fidelity coverage by inspecting the FPGA slowdown and resource utilization tradeoff of instrumenting BlackParrot. We present case studies where the high-fidelity coverage metric uniquely identifies a buggy state of the hardware. Finally, we compare the high-fidelity coverage with previously proposed metrics by integrating them into different instances of a simple coverage-guided fuzzing loop and observe how the proposed metric guides the fuzzer towards better design exploration.

In conclusion, this dissertation identifies, tackles, and improves upon common challenges faced by hardware engineers when employing various techniques for hardware prototyping, functional verification, and performance optimization. By streamlining design evaluation, the research presented in this dissertation opens promising avenues for development of future research. Possible directions include open-source core and accelerator designs, improved FPGA emulation frameworks, better design verification and performance profiling techniques, and development of advanced LLM-based fuzzers for targeted design exploration.

Bibliography

[1] A. Agarwal et al. “The RAW compiler project”. In: *Proceedings of the Second SUIF Compiler Workshop*. 1997, pp. 21–23.

[2] Tutu Ajayi et al. “Celerity: An Open Source RISC-V Tiered Accelerator Fabric”. In: *HOTCHIPS*. Aug. 2017.

[3] Alibaba. 2023. URL: `https://www.alibabacloud.com/product/computing`.

[4] Chips Alliance. *https://github.com/chipsalliance/Surelog*. 2023.

[5] Chips Alliance. *RISCV-DV*. `https://github.com/chipsalliance/riscv-dv`.

[6] Alric Althoff et al. “Hiding Intermittant Information Leakage with Architectural Support for Blinking”. In: *International Symposium on Computer Architecture (ISCA)*. 2018.

[7] Amazon. 2023. URL: `https://aws.amazon.com/ec2/spot/pricing/`.

[8] Amazon. *Amazon Web Services. 2022. Amazon EC2 F1 Instances*. 2023. URL: `https://aws.amazon.com/ec2/instance-types/f1/`.

[9] AMBA. *AXI Protocol Specification*. `https://developer.arm.com/documentation/ihi0022/latest/`.

[10] ARM. 2023. URL: `https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview`.

[11] ARM. *https://github.com/littlefs-project/littlefs*. 2023.

[12] Manish Arora et al. “Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems”. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2011.

[13] Khalil Arshak, Essa Jafer, and Christian Ibala. “Testing FPGA based digital system using XILINX ChipScope logic analyzer”. In: *2006 29th International Spring Seminar on Electronics Technology*. IEEE. 2006, pp. 355–360.

[14] Krste Asanovic et al. “The rocket chip generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[15] Saahil Athrij. “Vectorizing the Hamerblade Compiler”. MA thesis. University of Washington, 2024.

[16] Sameh Attia and Vaughn Betz. “StateLink: FPGA system debugging via flexible simulation/hardware integration”. In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2021, pp. 1–10.

[17] Peter Auer et al. “The Nonstochastic Multiarmed Bandit Problem”. In: *SIAM Journal on Computing* 32.1 (2002), pp. 48–77. DOI: `10.1137/S0097539701398375`.

[18] AVnet. 2023. URL: `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/`.

[19] Z. Azad et al. “RACE: RISC-V SoC for En/decryption ACceleration on the Edge for Homomorphic Computation.” In: *ISLPED*. 2022.

[20] Zahra Azad et al. “RISE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Encryption”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2023.

[21] Jonathan Babb et al. “The Raw Benchmark Suite: Computation Structures for General Purpose Computing”. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 1997.

[22] Jonathan Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: `10.1145/2228360.2228584`. URL: `https://doi.org/10.1145/2228360.2228584`.

[23] Scott Beamer and David Donofrio. “Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218632`.

[24] B. Beresini, S. Ricketts, and M.B. Taylor. “Unifying manycore and FPGA processing with the RUSH architecture”. In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*. 2011, pp. 22–28.

[25] Bespoke Silicon Group. *PanicRoom: Newlib Port with DRAM-Based File System*. `https://github.com/bespoke-silicon-group/bsg_newlib_dramfs`. 2021.

[26] Vikram Bhatt et al. “ Sichrome: Mobile web browsing in Hardware to save Energy ”. In: *Dark Silicon Workshop, ISCA*. 2012.

[27] Christian Bienia et al. “The PARSEC benchmark suite: Characterization and architectural implications”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[28] Mark Bohr. “A 30 Year Retrospective on Dennard’s MOSFET Scaling Paper”. In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13. DOI: `10.1109/N-SSC.2007.4785534`.

[29] Matej Bölcskei et al. “Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection”. In: *34th USENIX Security*. 2025.

[30] Pallavi Borkar et al. “Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors”. In: *arXiv preprint arXiv:2402.03704* (2024).

[31] Ajay Brahmakshatriya et al. “Taming the Zoo: A Unified Graph Compiler Framework for Novel Architectures”. In: *ISCA*. 2021.

[32] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. “SPEC CPU2017: Next-generation compute benchmark”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 41–42.

[33] Cadence. 2023. URL: `https://www.cadence.com/en%5C_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html`.

[34] Sadullah Canakci et al. “DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing”. In: *DAC*. 2021.

[35] Sadullah Canakci et al. “ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance”. In: *HOST*. 2023.

[36] Chen Chen et al. “HyPFuzz: Formal-Assisted Processor Fuzzing”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1361–1378. ISBN: 978-1-939133-37-3. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/chen-chen`.

[37] Chen Chen et al. “PSOFuzz: Fuzzing processors with particle swarm optimization”. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2023, pp. 1–9.

[38] Lin Cheng et al. “A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), pp. 1620–1635.

[39] Lin Cheng et al. “Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories”. In: *ASPLOS*. 2023.

[40] Derek Chiou et al. “Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 249–261.

[41] Grigory Chirkov and David Wentzlaff. “SMAPPIC: Scalable Multi-FPGA Architecture Prototype Platform in the Cloud”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 733–746.

[42] Yuan-Mao Chueh. “A Complete Open Source Network Stack For BlackParrot”. MA thesis. University of Washington, 2022.

[43] Eric S Chung et al. "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2.2 (2009), pp. 1–32.

[44] Scott Davidson et al. "The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric". In: *Micro, IEEE* (Mar. 2018).

[45] C. Dawson, S.K. Pattanam, and D. Roberts. "The Verilog Procedural Interface for the Verilog Hardware Description Language". In: *Proceedings. IEEE International Verilog HDL Conference.* 1996, pp. 17–23. DOI: `10.1109/IVC.1996.496013`.

[46] Nekija Dzemaili. "A reliable booting system for Zynq Ultrascale+ MPSoC devices". PhD thesis. CERN, 2021.

[47] Mahyar Emami et al. "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*. ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 219–237. ISBN: 9798400703942. DOI: `10.1145/3623278.3624750`. URL: `https://doi.org/10.1145/3623278.3624750`.

[48] Hadi Esmaeilzadeh and Michael Bedford Taylor. "Open Source Hardware: Stone Soups and Not Stone Satues, Please". In: *SIGARCH Computer Architecture Today.* Dec. 2017.

[49] Jeffrey Fairbanks, Akshharaa Tharigonda, and Nasir U. Eisty. "Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab". In: *arXiv preprint arXiv:2303.16393* (2023).

[50] Harry D. Foster. "White Paper - 2024 Wilson Research Group IC/ASIC functional verification trend report". In: *Wilson Research Group and Mentor, A Siemens Business.* 2024.

[51] Raspberry Pi Foundation. *https://www.raspberrypi.org.* 2023.

[52] RISC-V Foundation. *https://github.com/riscv-software-src/riscv-pk.* 2023.

[53] FSF. *https://www.gnu.org/software/libc/.* 2023.

[54] Emily Furst. "Code Generation and Optimization of Graph Programs on a Manycore Architecture". PhD thesis. University of Washington, 2021.

[55] Shay Gal-On and Markus Levy. "Exploring coremark a benchmark maximizing simplicity and efficacy". In: *The Embedded Microprocessor Benchmark Consortium* (2012).

[56] S. Garcia et al. "The Kremlin Oracle for Sequential Code Parallelization". In: *Micro, IEEE* 32.4 (July 2012), pp. 42–53.

[57] Saturnino Garcia et al. "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning". In: *USENIX Workshop on Hot Topics in Parallelism (HOT-PAR).* 2010.

[58] Saturnino Garcia et al. "Kremlin: Rethinking and Rebooting gprof for the Multicore Age". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2011.

[59] GDB Developers and Free Software Foundation. *Debugging with GDB: The GNU Source-Level Debugger*. Tenth Edition, for GDB 16.3. Free Software Foundation. 2025.

[60] Vasudev Gohil et al. "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[61] Bjorn Gottschall, Lieven Eeckhout, and Magnus Jahre. "TEA: Time-Proportional Event Analysis". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–13.

[62] Bjorn Gottschall, Lieven Eeckhout, and Magnus Jahre. "Tip: Time-proportional instruction profiling". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 15–27.

[63] Nathan Goulding et al. "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon". In: *HOTCHIPS*. 2010.

[64] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future". In: *Micro, IEEE* (Mar. 2011), pp. 86–95.

[65] Nathan Goulding-Hotta. "Specialization as a Candle in the Dark Silicon Regime". PhD thesis. University of California, San Diego, 2020.

[66] Nathan Goulding-Hotta et al. "GreenDroid: An Architecture for the Dark Silicon Age". In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. 2012.

[67] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache". In: *International Conference on Computer Design (ICCD)*. 2013.

[68] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "QualityTime: A Simple Online Technique for Quantifying Multicore Execution Efficiency". In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014.

[69] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking". In: *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*. 2013.

[70] Ziyad Hanna and Jamil R. Mazzawi. "Formal Analysis of Security Data Paths in RTL Design". In: *Proceedings of the Haifa Verification Conference (HVC)*. Demonstrates the JasperGold formal-verification flow. Cadence Design Systems. 2012.

[71] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. "A Runtime Approach to Security and Privacy". In: *European Security and Privacy*. 2016.

[72] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. " BlackBox: Lightweight Security Monitoring for COTS Binaries". In: *Code Generation and Optimization*. 2016.

[73] John L Henning. "SPEC CPU2000: Measuring CPU performance in the new millennium". In: *Computer* 33.7 (2000), pp. 28–35.

[74] John L Henning. "SPEC CPU2006 benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[75] Vladimir Herdt et al. "Verifying Instruction Set Simulators using Coverage-guided Fuzzing". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019, pp. 360–365. DOI: `10.23919/DATE.2019.8714912`.

[76] Hu et al. "FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach ". In: *ICCD*. 2007.

[77] Jaewon Hur et al. "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1286–1303. DOI: `10.1109/SP40001.2021.00103`.

[78] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. IEEE Computer Society, 2018. DOI: `10.1109/IEEESTD.2018.8299595`.

[79] *ISO/IEC TS 22277:2017 — Technical Specification:* C++ Extensions for Coroutines. Tech. rep. International Organization for Standardization, 2017. URL: `https://www.iso.org/standard/73008.html`.

[80] Adam Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 209–216.

[81] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. "Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis". In: *Conference on Code Generation and Optimization (CGO)*. 2013.

[82] Donghwan Jeon et al. "Kismet: Parallel Speedup Estimates for Serial Programs". In: *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*. 2011.

[83] Donghwan Jeon et al. "Kremlin: Like gprof, but for Parallelization". In: *Principles and Practice of Parallel Programming (PPoPP)*. 2011.

[84] Donghwan Jeon et al. "Parkour: Parallel Speedup Estimates from Serial Code". In: *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*. 2011.

[85] Dai Cheol Jung. "Caches for Complex Open Source System-on-Chip Designs". MA thesis. University of Washington, 2019.

[86] Dai Cheol Jung et al. "Ruche Networks: Wire-Maximal, No-Fuss NoCs". In: *NOCS*. 2020.

[87] Dai Cheol Jung et al. "Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore". In: *ISCA*. 2024.

[88] Nursultan Kabylkas et al. "Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 667–678. ISBN: 9781450385572. DOI: `10.1145/3466752.3480092`. URL: `https://doi.org/10.1145/3466752.3480092`.

[89] Nursultan Kabylkas et al. "Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 667–678. ISBN: 9781450385572. DOI: `10.1145/3466752.3480092`. URL: `https://doi.org/10.1145/3466752.3480092`.

[90] Rahul Kande et al. "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/kande`.

[91] Sagar Karandikar et al. "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025. DOI: `10.1145/3373376.3378455`. URL: `https://doi.org/10.1145/3373376.3378455`.

[92] Sagar Karandikar et al. "FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: `10.1109/ISCA.2018.00014`. URL: `https://doi.org/10.1109/ISCA.2018.00014`.

[93] Nikos Karystinos et al. "Harpocrates: Breaking the Silence of CPU Faults through Hardware-in-the-Loop Program Generation". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 516–531. DOI: `10.1109/ISCA59077.2024.00045`.

[94] Moein Khazraee. "Reducing the development cost of customized cloud infrastructure". PhD thesis. University of California, San Diego, 2020.

[95] Moein Khazraee et al. "Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[96] Moein Khazraee et al. "Specializing a Planet's Computation: ASIC Clouds". In: *IEEE Micro* (May 2017).

[97] Jason Kim et al. "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. Aug. 2003.

[98] Sravanthi Kota Venkata et al. "SD-VBS: The San Diego Vision Benchmark Suite". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[99] Kevin Laeufer et al. "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: `10.1145/3240765.3240842`.

[100] Kevin Laeufer et al. "Simulator Independent Coverage for RTL Hardware Languages". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 606–615. ISBN: 9781450399180. DOI: `10.1145/3582016.3582019`. URL: `https://doi.org/10.1145/3582016.3582019`.

[101] Kangli Li. "An Open Source Non-Blocking Manycore L2 Cache". MA thesis. University of Washington, 2024.

[102] Ryan Lund. "Design and Application of a Co-Simulation Framework for Chisel". PhD thesis. MA thesis. EECS Department, University of California, Berkeley, 2021.

[103] Ikuo Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[104] Sergio Mazzola et al. "Data-Driven Power Modeling and Monitoring via Hardware Performance Counters Tracking". In: *arXiv preprint arXiv:2401.01826* (2024).

[105] Mentor. 2023. URL: `https://eda.sw.siemens.com/en-US/ic/veloce/`.

[106] Sergiu Mosanu et al. "FreezeTime: Towards System Emulation through Architectural Virtualization". In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2023, pp. 129–136.

[107] Sripathi Muralitharan. "TinyParrot: An Integration-Optimized Linux-Capable Host Multicore". MA thesis. University of Washington, 2021.

[108] Anoop Mysore Nataraja. "A Research-Fertile Co-Emulation Framework for RISC-V Processor Verification". English. PhD thesis. University of Washington, 2023, p. 93. ISBN: 9798380328562. URL: `https://www.proquest.com/dissertations-theses/research-fertile-co-emulation-framework-risc-v/docview/2863720086/se-2`.

[109] S. Pal et al. "A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm". In: *Symposium on VLSI Circuits*. 2019, pp. C150–C151.

[110] Scott Beamer Thomas Nijssen Krishna Pandian and Kyle Zhang. “ESSENT: A High-Performance RTL Simulator”. In: *Workshop on Open-Source EDA Technology (WOSET), at International Conference on Computer-Aided Design (ICCAD)* (2021).

[111] D. Park et al. “A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator”. In: *IEEE Journal of Solid-State Circuits* (Apr. 2020), pp. 933–944.

[112] Huwan Peng. “Methodologies and Architectures for AI Inference Hardware: From Foundational Networks to Large Language Models”. PhD thesis. University of Washington, 2025.

[113] Huwan Peng et al. “Chiplet Cloud: Building AI Supercomputers for Serving Large Generative Language Models”. In: *arXiv:2307.02666 [cs]* (2024). arXiv: `2307.02666`.

[114] Huwan Peng et al. “ReaLLM: A Trace-Driven Framework for Rapid Simulation of Large-Scale LLM Inference”. In: *ASAP*. 2025.

[115] D. Petrisko et al. “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs”. In: *IEEE Micro* (July 2020), pp. 93–102.

[116] Daniel Petrisko et al. “NoC Symbiosis”. In: *NOCS*. 2020.

[117] Vaughan Pratt. “Anatomy of the Pentium bug”. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1995, pp. 97–107.

[118] Linux man-pages project. *syscalls(2) – Linux system calls*. Version 6.10. Linux man-pages. Nov. 17, 2024. URL: `https://man7.org/linux/man-pages/man2/syscalls.2.html`.

[119] Hany Ragab et al. “BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer”. In: *SILM* (2022).

[120] Robert "Max" Ramstad. “Enabling Vector Load and Store instructions on HammerBlade Architecture”. MA thesis. University of Washington, 2024.

[121] Shashank Vijaya Ranga. “ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor”. MA thesis. University of Washington, 2021.

[122] David Rich. “The missing link: the Testbench to DUT connection”. In: *Fremont, CA: Design and Verification Technologies Mentor Graphics* 9 (2013).

[123] Mohamadreza Rostami et al. “Beyond random inputs: A novel ml-based hardware fuzzing”. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[124] A. Rovinski et al. “A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS”. In: *2019 Symposium on VLSI Circuits*. 2019, pp. C30–C31.

[125] A. Rovinski et al. “Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL”. In: *IEEE Solid-State Circuits Letters* 2.12 (2019), pp. 289–292.

[126] Jack Sampson et al. “An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors”. In: *Conference on Field Programmable Logic and Applications (FPL)*. 2011.

[127] Jack Sampson et al. “Efficient Complex Operators for Irregular Codes”. In: *High Performance Computing Architecture (HPCA)*. 2011.

[128] Andreas Sandberg et al. “Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed”. In: *2015 IEEE International Symposium on Workload Characterization*. 2015, pp. 183–192. DOI: `10.1109/IISWC.2015.29`.

[129] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. “The Emergence of Hardware Fuzzing: A Critical Review of its Significance”. In: *arXiv preprint arXiv:2403.12812* (2024).

[130] Debendra Das Sharma et al. “PCI Express 6.0 Specification: A Low-Latency, High-Bandwidth, High-Reliability, and Cost-Effective Interconnect with 64 GT/s pam-4 Signaling”. In: *IEEE Micro* 41.1 (2021), pp. 23–29. DOI: `10.1109/MM.2020.3039925`.

[131] Wilson Snyder. 2024. URL: `https://github.com/verilator/verilator`.

[132] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. “Cascade: CPU fuzzing via intricate program generation”. In: *Proc. 33rd USENIX Secur. Symp.* 2024, pp. 1–18.

[133] Steven Swanson and Michael Taylor. “GreenDroid: Exploring the next evolution for smartphone application processors”. In: *IEEE Communications Magazine*. Mar. 2011.

[134] Synopsys. 2023. URL: `https://www.synopsys.com/verification/emulation/zebu-server.html`.

[135] Synopsys, Inc. *VCS Functional Verification Solution*. Datasheet, release 10/26/2021. Synopsys, Inc. 2021.

[136] Zhangxi Tan et al. “Diablo: A warehouse-scale computer network simulator using fpgas”. In: *ACM SIGPLAN Notices* 50.4 (2015), pp. 207–221.

[137] MB Taylor et al. “The Raw processor-a scalable 32-bit fabric for embedded and general purpose computing”. In: *Proceedings of Hot Chips XIII*. 2001.

[138] Michael Taylor. “A Landscape of the New Dark Silicon Design Regime”. In: *Micro, IEEE* (Sept. 2013).

[139] Michael Taylor. “A Landscape of the New Dark Silicon Design Regime”. In: *Design Automation and Test in Europe*. Apr. 2014.

[140] Michael Taylor. “The Evolution of Bitcoin Hardware”. In: *Computer, IEEE* (Sept. 2017).

[141] Michael Taylor. “Tiled Microprocessors”. PhD thesis. Massachusetts Institute of Technology, 2007.

[142] Michael B. Taylor. “BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design”. In: *Design Automation Conference*. June 2018.

[143] Michael B. Taylor. “Bitcoin and the Age of Bespoke Silicon”. In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 2013.

[144] Michael B. Taylor. “Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse”. In: *Design Automation Conference (DAC)*. 2012.

[145] Michael B. Taylor et al. “A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network”. In: *IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2003.

[146] Michael B. Taylor et al. “Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams”. In: *International Symposium on Computer Architecture (ISCA)*. June 2004.

[147] Michael B. Taylor et al. “Scalar Operand Networks”. In: *IEEE Transactions on Parallel and Distributed Systems*. Feb. 2005.

[148] Michael B. Taylor et al. “Scalar Operand Networks”. In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. Feb. 2005.

[149] Michael B. Taylor et al. “Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2003.

[150] Michael B. Taylor et al. “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs”. In: *IEEE Micro*. Mar. 2002.

[151] Michael Bedford Taylor. “Geocomputers and the Commercial Borg”. In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[152] Michael Bedford Taylor. “Your agile open source HW stinks (because it is not a system)”. In: *ICCAD*. 2020.

[153] Michael Bedford Taylor et al. “ASIC Clouds: Specializing the Datacenter for Planet-Scale Applications”. In: *CACM* (2020), pp. 103–109.

[154] Fabian Thomas et al. *RISCVuzz: Discovering architectural CPU vulnerabilities via differential hardware fuzzing*.

[155] Shelby Thomas et al. “CortexSuite: A Synthetic Brain Benchmark Suite”. In: *International Symposium on Workload Characterization (IISWC)*. Oct. 2014.

[156] Linus Torvalds. “The Linux edge”. In: *Commun. ACM* 42.4 (Apr. 1999), pp. 38–39. ISSN: 0001-0782. DOI: `10.1145/299157.299165`. URL: `https://doi.org/10.1145/299157.299165`.

[157] Timothy Trippel et al. “Fuzzing hardware like software”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3237–3254.

[158] TUL. 2023. URL: `https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html`.

[159] Shobha Vasudevan et al. "Learning semantic representations to verify hardware designs". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 23491–23504.

[160] Luis Vega and Michael Bedford Taylor. " RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization ". In: *CARRV*. 2017.

[161] Bandhav Veluri et al. "NeuriCam: Low-Power Video Acquisition using Dual-Mode IoT Cameras". In: *MobiCom*. 2023.

[162] Ganesh Venkatesh et al. "Conservation cores: reducing the energy of mature computations". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010.

[163] Ganesh Venkatesh et al. "QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner". In: *International Symposium on Microarchitecture (MICRO)*. 2011.

[164] Elliot Waingold et al. "Baring it all to Software: Raw Machines". In: *IEEE Computer*. Sept. 1997.

[165] Vincent M. Weaver. "Self-Monitoring Overhead of the Linux perf_event Performance Counter Interface". In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 159–168. DOI: `10.1109/ISPASS.2015.7095792`.

[166] Vincent M. Weaver and Sally A. McKee. "Can hardware performance counters be trusted?" In: *2008 IEEE International Symposium on Workload Characterization*. 2008, pp. 141–150. DOI: `10.1109/IISWC.2008.4636099`.

[167] Western Digital. *OpenSBI: RISC-V Open-Source Supervisor Binary Interface*. `https://github.com/riscv-software-src/opensbi`. Release v1.6. 2024.

[168] Xilinx Wiki. *Cadence WDT Driver*. URL: `https://xilinx-wiki.atlassian.net/wiki/x/x4EfAQ`.

[169] Henry Ting-Hei Wong. *A superscalar out-of-order x86 soft processor for fpga*. University of Toronto (Canada), 2017.

[170] Mark Wyse et al. "The BlackParrot BedRock Cache Coherence System". In: *arXiv preprint arXiv:2211.06390* (2022).

[171] Chenhao Xie et al. "Q-VR: System-Level Design for Future Mobile Collaborative Virtual Reality". In: *ASPLOS*. 2021.

[172] Shaolin Xie et al. "Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds". In: *ACM Sigops Operating System Review*. 2018.

[173] Xilinx. 2023. URL: `https://docs.xilinx.com/r/en-US/pg195-pcie-dma`.

[174] Xilinx. 2023. URL: `https://www.xilinx.com/products/boards-and-kits/device-family/nav-zynq-7000.html`.

[175] Xilinx. 2023. URL: `http://www.pynq.io/board.html`.

[176] Xilinx. 2023. URL: `https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide/Introduction`.

[177] Xilinx. *AXI DMA Controller IP*. URL: `https://www.xilinx.com/products/intellectual-property/axi_dma.html`.

[178] Xilinx. *Device Reliability Report (UG116)*. Tech. rep. Xilinx, 2023.

[179] Jinyan Xu et al. "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 1307–1324.

[180] Michal Zalewski. *AFL*. `https://github.com/google/AFL`.

[181] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. ISSN: 1557-9999. DOI: `10.1109/TVLSI.2019.2926114`.

[182] Karen Zee et al. "Runtime Checking for Program Verification". In: *RV*. 2007.

[183] Xingyao Zhang et al. "η-LSTM: Co-Designing Highly-Efficient Large LSTM Training via Exploiting Memory-Saving and Architectural Design Opportunities". In: *ISCA*. 2021.

[184] Ritchie Zhao et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *7th RISC-V Workshop*. 2017.

[185] Qiaoshi Zheng et al. "Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon". In: *Transactions on Embedded Computing Systems (TECS)* (Mar. 2014).

[186] Yi Zhu et al. "Advancing supercomputer performance through interconnection topology synthesis". In: *International Conference on Computer-Aided Design (ICCAD)*. 2008, pp. 555–558.

[187] Yi Zhu et al. "Energy and Switch Area Optimizations for FPGA Global Routing Architectures". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. Jan. 2009.

Appendix A

Source-Code Repositories

At the time of writing this dissertation, all the source-code for the discussed projects are hosted in the following open-source GitHub repositories:

- The design code for BlackParrot alongside its software development kit can be found at the following repositories:
  - `https://github.com/black-parrot/black-parrot`
  - `https://github.com/black-parrot-sdk/black-parrot-sdk`
- The source-code for Condominium alongside the algorithm for instrumenting RTL designs with High-Fidelity Coverage can be found at:
  - `https://github.com/black-parrot-hdk/zynq-parrot`
- The source-code for managing the Condominium heterogeneous cluster alongside its custom PYNQ image generation flow can be found at:
  - `https://github.com/black-parrot-hdk/zynq-farm`
  - `https://github.com/black-parrot-hdk/pynq-image`
- The scripts for running experiments for High-Fidelity Coverage based on a MAB-Cascade fuzzer can be found at:
  - `https://github.com/farzamgl/hfcov`