ZynqParrot: A Scale-Down Approach to Cycle-Accurate, FPGA-Accelerated Co-Emulation

Daniel Ruelas-Petrisko (University of Washington) <petrisko@cs.washington.edu>
Farzam Gilani (University of Washington) <farzamgl@uw.edu>
Anoop Mysore Nataraja (University of Washington) <mysanoop@uw.edu>
Zoe Taylor (Seattle Country Day School) <zoe.nguyen.taylor@gmail.com>
Michael Taylor (University of Washington) <prof.taylor@gmail.com>

***Abstract*—As processors increase in complexity, costs grow even more rapidly, both for functional verification and performance validation. Additionally, performance models become ever more sensitive to slight microarchitecture inaccuracies. Runtime measurements of key workloads are an essential part of the performance debugging process. Most often, silicon characterizations comprise simple performance counters, which are aggregated and separated to tell a story. Based on these inferences, performance engineers employ microarchitectural simulation to inspect deeply into the core. Unfortunately, dramatically longer runtimes make simulation infeasible for long workloads.**

Traditionally, architects have bridged this gap by performing early prototyping on FPGA. Yet, the scale of modern designs is impractical to implement on a single emulation board. Large companies use Scale-Up solutions such as commercial emulation platforms, but these are unaffordable to academics, hobbyists and startups. Others have proposed Scale-Out solutions, leveraging cloud FPGA clusters to emulate large System-On-Chips. However, this approach prescribes certain I/O and memory system architectures instead of the native interface timings of any given subsystem.

Instead, we propose a *Scale-Down* approach to modelling and validation. Rather than up-sizing a prototyping platform to fit large and complex system designs, we show that it can be more accurate, faster, and more economical to decompose a system into manageable sub-components that can be prototyped independently. By carefully designing the prototyping interface, it is possible to adhere to strict non-interference of the Device Under Test (DUT). This allows architects to have the best of both worlds: the speed of FPGA acceleration while eliminating the inaccuracies of Scale-Out and the inherent costs of Scale-Up.

In this work, we present ZynqParrot: a Scale-Down FPGA-based modelling platform, capable of executing non-interfering, cycle-accurate co-emulations of arbitrary RTL designs. ZynqParrot is capable of verifying functionality and performance with arbitrary granularity. We also provide case studies using ZynqParrot to analyse the full-stack performance of an open-source RISC-V processor.

I. Introduction

As processors increase in complexity, verification costs grow even more rapidly, both for functionality and performance. The end of Dennard Scaling [13] has led to a Cambrian explosion of domain-specific accelerators which present unique, full-stack verification challenges. Besides functional correctness, performance validation is critical to achieving worthwhile gains from accelerator integration. The higher performance the design, the more sensitive it is to subtle disturbances in the microarchitecture. When characterizing full-system performance in silicon, software engineers use simple performance counters which must be decided upon early in the design process, before problematic subsystems have been identified. These counters are aggregated and separated to divine reasons for the performance of a target application. On the other end of the spectrum, architects can leverage the deep verification capabilities of pre-silicon waveform inspection to identify subsystem bottlenecks before tapeout, when fixes are much cheaper. However, cycle-accurate simulations are painfully slow, so architects must settle for sampling applications [29] to complete in a reasonable timescale.

Traditionally, architects have bridged this gap by performing early prototyping in FPGA. By doing so, RTL similar to tapeout designs can be emulated with cycle accuracy at 10-100x faster than simulation alone. However, modern designs are too large to economically fit on a single emulation board. Large companies can Scale-Up their prototyping systems using commercial emulation platforms [16], [38], [44], but these are unaffordable to academics, hobbyists and startups. Other academics have proposed Scale-Out solutions that leverage cloud FPGA clusters [1], [3] to emulate large System-On-Chips. However, this approach generally relies on regularity in the design, prescribes certain standardized I/O and memory system architectures and couples platforms to proprietary vendor IP which may interfere with the native interface timings of any given system and provide no insight or ability to adapt for system needs.

Inspired by biotechnological process modelling [39], we propose a *Scale-Down* approach for architectural prototyping, shown in Figure 1. Instead of unilaterally scaling up a chip design (process) from an FPGA prototype (lab-scale experiment) to a full tapeout (industrial manufacturing process), it is more economical to iteratively Scale-Up and Scale-Down the design to identify subsystem issues at scale and precisely debug performance in a smaller, more tightly instrumented system. By closely correlating the two environments, Scale-Down results can give deep debugging insights into process deviations as well as accurately predict future deviations for orders of magnitude lower cost than a full production run.

In this work, we show that decomposing and recomposing the system is more accurate, faster, and more economical than

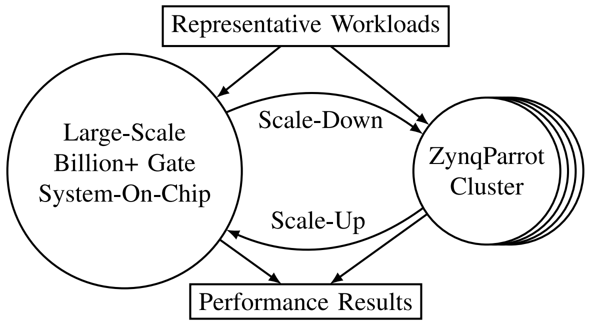


Fig. 1. A Scale-Up/Scale-Down cycle transforms industrial manufacturing processes into laboratory experiments. Experiments accept identical inputs as the full process. Results are extrapolated to predict impacts on the modified process before committing to costly change orders. Iterating over the process lifecycle leads to continuous improvement.

state of-the-art alternatives. By carefully designing the platform interfaces, it is possible to provide flexible environments that represent the interactions of real systems while maintaining system timings at the component interfaces. This allows architects to have the best of both words: the speed of FPGA acceleration while eliminating the inaccuracies of Scale-Out and the inherent costs of Scale-Up. Because iteration time is much faster than monolithic prototypes, small design teams can quickly bootstrap new Scale-Down subsystems, run large simulations on abstracted Scale-Up models and return to Scale-Down to rapidly debug and enhance components.

When Scaling-Down, great care must be taken to accurately transform the design and exactly mimic the environment between the full design and the subsystem. In particular, I/O interface timings must adhere to accurate timing models. Previous works [19], [32] have focused on instrumenting FPGA timing for common interfaces such as DRAM and Ethernet rather than the fully custom models required for subsystem partitioning. Others [5] have focused on providing full-system emulation for software development and too high a level of abstraction for microarchitectural debugging. Instead, this work aims to provide more precise and more flexible interface emulation to allow for finer-grained partitioning without sacrificing accuracy.

Previous FPGA emulation platforms [44] [16] [32] [19] are expensive, dependent on vendor IP, or cumbersome and prone to lock-up. In contrast, ZynqParrot builds upon the BaseJump STL [46] library to provide generic and completely open bridges to commonly available AXI and UART interfaces. When using Zynq-based FPGAs [55], the only requirement to use ZynqParrot is an SSH-capable machine running Vivado. For non-Zynq FPGAs, ZynqParrot requires a UART connection to the FPGA as well as a JTAG connection for bitstream programming, although the platform architecture easily supports plugins for additional host functionality. Contrast this to traditional solutions which require expensive PCIe-capable accelerators. These setups are hard to maintain, built on top of proprietary PCIe IP and software layers such as Xilinx XDMA [53]. Failure to interface correctly to PCIe can lockup

TABLE I
ON A PER-FPGA BASIS SCALE-DOWN SYSTEMS REQUIRE A MUCH SMALLER INVESTMENT, ALLOWING TEAMS TO INCREMENTALLY BUILD UP THEIR VERIFICATION INFRASTRUCTURE.

Strategy	**\$/year/FPGA**[0]	**Logic Unit**	**Required I/O**
Scale-Up[1]	∼\$5000	Full Design	Native PCIe
Scale-Out[2]	∼\$2000	Tile	PCIe Tunnel
Scale-Down[2]	**∼\$100**	**IP Block**	**SSH/Serial**

[0] 2000 hours is equivalent to a year of 8-hour regressions.
[1] \$1.4167 per AWS f1.16xlarge hour [4].
[2] \$0.6744 per AWS f1.4xlarge hour [4].
[3] \$300 per Avnet Ultra96v2 board [9], with a replacement rate of once per three years. Cluster MTBF is 100+ years [52].

not only the DUT but also the host server machine, requiring remote restart capabilities. A cluster of such host machines and FPGAs can easily exceed 10s of thousands of dollars and require full-time system administration (see Table I). In contrast, ZynqParrot provides verification teams the ability to begin with the minimal possible Total Cost of Ownership (*TCO*) and scale costs alongside the design progress.

ZynqParrot 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. Each Device-Under-Test (DUT) execution cycle can be verified against a emulated model, taking into account functional correctness along with verification of internal and external performance. Additionally, ZynqParrot is able to synthesize complex performance counters without interfering with mature or frozen RTL. Such a deep dissection of the subsystem can allow an architect to design experiments to identify subsystem bottlenecks and quickly iterate without requiring microarchitectural changes.

We provide case studies of how to analyse a complex RISC-V processor, *BlackParrot* [42], using ZynqParrot. We identify useful software and hardware enhancements to quickly verify functionality and performance and how they easily fit into the ZynqParrot framework. These non-invasive, instrumented measurements and host software abstraction layers were written for use in BlackParrot; however, they are generally applicable to a wide range of RISC-V projects. Additionally, we demonstrate real-world uses of ZynqParrot: first to host the term projects of an undergraduate/graduate architecture class of 20 students, second to bring up first-batch silicon during a commercial tapeout, and third to analyse a bottleneck and access the improvement of a new microarchitectural widget in BlackParrot.

Our major contributions are:

1) We present ZynqParrot[1], the first Scale-Down open-source prototyping platform for deterministic, cycle-accurate local FPGA co-emulation.
2) We provide a co-simulation capable prototyping library that eschews vendor dependencies and is cycle-identical

[1]The hardware and software code for ZynqParrot is open-source under a permissive BSD-3 License. (https://github.com/black-parrot-hdk/zynq-parrot)

to ZynqParrot co-emulation.

3) We explore the diversity of ZynqParrotusage via a series of case studies, from the classroom to academic research to commercial bring-up.
4) We introduce PanicRoom[2]: an ultra-portable library that leverages DRAM to run POSIX benchmarks on bare-metal systems.

II. ZynqParrot Architecture

Complex IP designs often require specific Vivado versions to ensure reproducible builds. However, each Vivado version only supports a small subset of operating systems, often directly conflicting with strict ASIC EDA vendor compatibility guides. Although Vivado-based build systems can be broken up into bitstream generation and programming machines, this requires supplemental system administration.

Additionally, many FPGAs are only available in PCIe-attached variants, resulting in complex and fragile bridge solutions. These connections require proprietary vendor solutions such as Xilinx XDMA [53], non-portable RTL and software layers and are prone to full-system lockup due to subtle bugs in configuration or execution. Lockup can cause long-running experiments to be lost, force costly manual debugging and complicate systems with fallback-recovery mechanisms.

Considering these limitations, emulation management software is often tailored to each specific host-FPGA pair and disparate from simulation-based testbench infrastructure, which must be maintained independently. As a result, to normalize environments across designs for powerful prototyping boards, groups must maintain costly heterogeneous server infrastructures with multiple x86 environments customized to each design.

Traditional FPGA prototyping systems require powerful discrete host servers to:

1) Compile a netlist and generate a bitstream, often requiring hours for large designs.
2) Program the bitstream, typically over a USB/JTAG connection.
3) Manage emulation execution, running software to load test vectors, monitor performance or verify results.

In this section, we describe the ZynqParrot architecture and how it addresses these challenges cost-effectively and with lower maintenance than previous solutions.

A. Hardware Architecture

Zynq FPGA boards couple hardened ARM cores (*VPS*) 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 [6] interfaces. The VPS master ports are called *GP* ports and cover a small address space. VPS client ports (*HP*) are larger and higher performance, allowing the PL to indirectly access DRAM and peripherals.

[2]The software code for PanicRoom is open-source and as of submission time being actively upstreamed to the Newlib project (https://github.com/black-parrot-sdk/libgloss-dramfs)

ZynqParrot leverages the Zynq architecture to decompose prototyping systems into these orthogonal functionalities. 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, program the overlay and DUT on the PL.

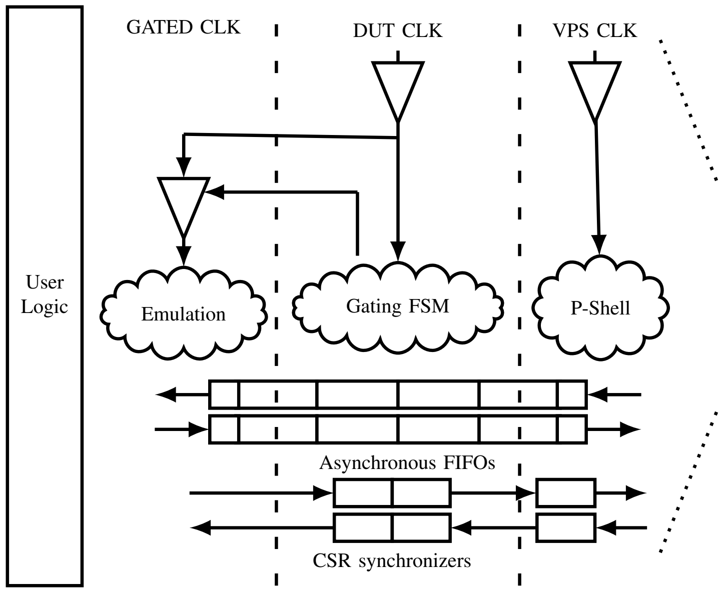
ZynqParrot provides an overlay (shown in Figure 2) that includes the *P-Shell*, the main interface between the host emulation and the DUT user logic. The P-Shell provides a parameterizable array of input/output Control and Status Registers *CSRs*, as well as an array of semi-blocking *SB-FIFOs*. An SB-FIFO exposes blocking ready/valid [46] interfaces to the PL side to support latency-insensitive interfaces, while the PS interacts with a non-blocking credit/valid 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.

While ZynqParrot fits seamlessly into the Zynq PS-PL paradigm, there are many FPGA architectures which lack hardened CPU cores. For these boards, ZynqParrot provides hardware bridges (shown in Figure 3) which can convert C++ P-Shell requests to a pseudo-GP master bus via a transparent software translation. We refer to this combination of C++ co-simulation code, host transport layer and GP master bus as a *VPS*. The VPS abstraction supports the flexibility of arbitrary MMIO interaction with the DUT while switching out transport layers optimized for the specific execution environment, all while maintaining lockup safety and reasonable performance.

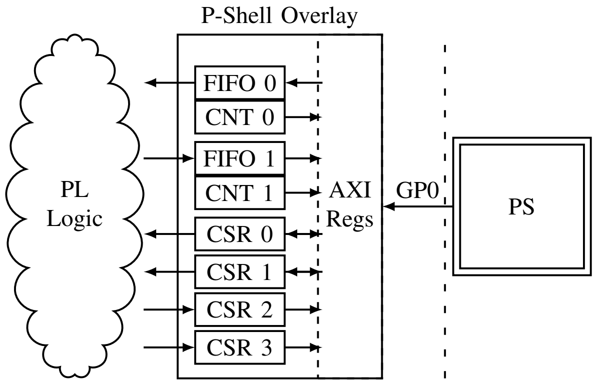
When prototyping ASICs, the DUT clock is often limited by poor mapping of standard cells to FPGA primitives [51], limiting the overall emulation performance. While some efficiency may be regained by explicit manual remapping of problematic primitives (CAMs, large muxes, heavily retimed modules), this duplicates design efforts and forces dependencies between FPGA and ASIC teams. Unfortunately, even the best mapping efforts cannot solve this problem for aggressive submicron designs. To alleviate performance bottlenecks and decouple the prototype and emulation, asynchronous FIFOs and CSR synchronizers bridge the VPS and DUT clock domains. This decoupling allows the VPS to run ahead of PL and averts complex emulation models slowing the DUT execution.

B. Emulation Layer

During deep performance profiling, the VPS may need to process monitoring information or system-call emulation every DUT cycle while it is also handling context switching, network bridging or other asynchronous processing. If the VPS is not ready to accept a new packet and an asynchronous FIFO fills, either the FIFO must backpressure such that cycle-accuracy is lost, or the packet is dropped. Most systems using latency-insensitive I/O constructs use ready/valid handshakes to pause the DUT operation upon backpressure. However, doing so perturbs the system and eliminates cycle-accuracy, making the



(a) ZynqParrot subcomponents interface with the P-Shell through a parameterizable set of SB-FIFOs and CSRs. The P-Shell logic is run asynchronously to the DUT, allowing for decoupled co-emulation. Clock gating logic on the VPS side 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 VPS lockup. In ZynqParrot, the P-Shell prevents lockup regardless of DUT state, by lifting generic DUT interfaces to a set of nonblocking FIFO and read/write CSRs.

Fig. 2. The ZynqParrot system provides system architects with full co-emulation capabilities through a simple C++ MMIO drivers, identically accessible from simulation, co-emulation or on deployed systems. Users parameterize the P-Shell to for control or monitor execution, while the VPS runs any necessary software functional models.

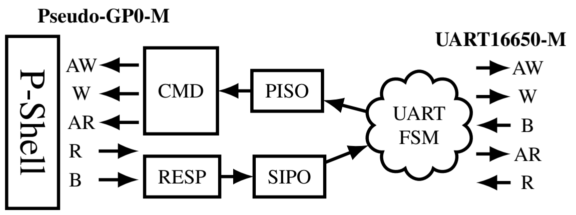


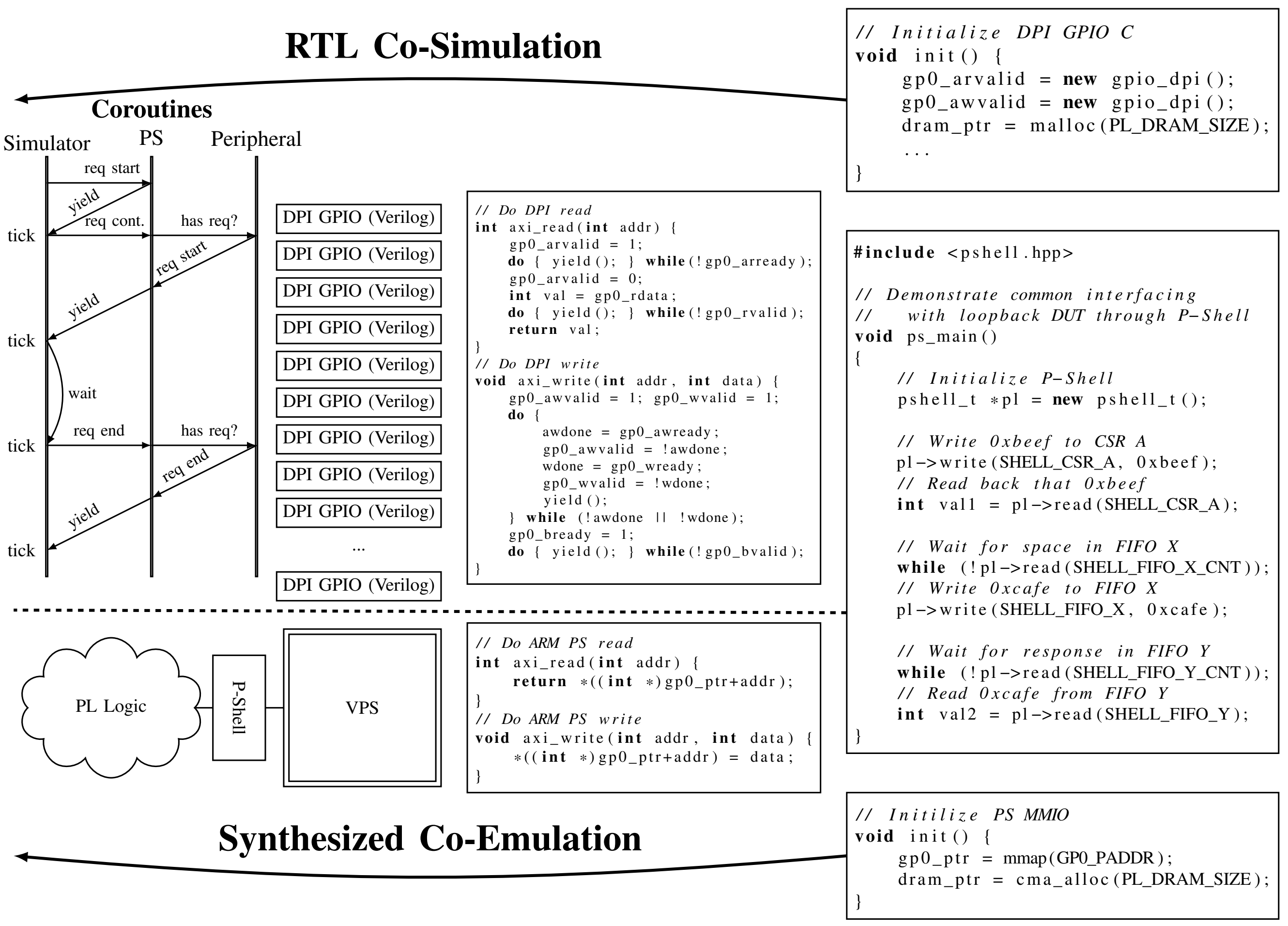
Fig. 3. When prototyping on non-Zynq FPGAs, ZynqParrot transparently tunnels requests into a "pseudo-GP0" accessing the P-Shell. For instance, GP0 writes are deserialized from UART RX while GP0 reads are deserialized from RX and then reserialized into UART TX.

emulation non-reproducible. Figure 4 shows an abstraction of the VPS responsibilities.

Another approach to avoid degradation is to run a Real-Time Operating System (RTOS) on the VPS. 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 profiling the information bandwidth needed varies dramatically based on the specific performance aspect being monitored. These guarantees will need to be retuned for each monitoring mode, as well as relaxed for running in a simulation mode that has dramatically different (wall-clock) timing characteristics. In addition, compiling arbitrary programs is much more difficult on a specialized RTOS compared to a full POSIX operation system.

ZynqParrot leverages the VPS-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 VPS, which may be running a full PetaLinux [54] operating system. Clock gating in the P-Shell means that both VPS software and DUT logic can be completely unaware of the other side of the interface, operating in an ideal environment. Clearly defined boundaries between VPS and DUT domains simplify necessary timing constraints during synthesis and standardized, validated asynchronous primitives shield users from the subtle gotchas of multi-clock systems.

On the other hand, modelling exact I/O timing is an essential functionality in a Scale-Down system. In ZynqParrot, hardware model timers exist in the DUT clock domain, but timing information is stored in the VPS 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. The VPS receives the request, calculates the predicted timing of the specific HBM model, and programs the expected timing through P-Shell CSRs. The DUT clock then resume, waiting for the DRAM request to return but executing any other parallel tasks. If the DRAM request returns before the hardware model timer finishes, it will be paused until the correct cycle. 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.



```
// 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 <pshell.hpp>

// Demonstrate common interfacing
//   with loopback DUT through P-Shell
void ps_main()
{
    // Initialize P-Shell
    pshell_t *pl = new pshell_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);
}
```

Fig. 4. ZynqParrot enables designs to run identical C++ code on the VPS 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, ZynqParrot exposes pins on the P-Shell through a DPI-C interface. The result is fine-grained control over DUT execution, enabling software flow-control and thorough verification. As multithreading is disallowed by many commerical simulators, C++ coroutines are used to co-simulate the DUT with blocking transactions such as AXI requests, providing parallelism and deadlock avoidance.

III. ZYNQPARROT DECOMPOSITION

A key element of Scale-Down is shrinking the design size while maintaining the integrity of inputs and outputs of the system. As illustrated in Figure 5, even for a single design hierarchy there are various decomposition strategies. The best strategy will depend on the size of the design, verification team and FPGA supply. At later stages in the verification cycle it may be advantageous to hyperfocus on smaller modules, attempting to expose subtle optimizations that may be clouded by full-system effects. Small, cost-effective FPGAs are especially advantageous at this stage. Firstly, smaller designs will benefit from shorter compilation times and faster emulation speeds, leading to quicker overall turnaround time. Secondly, when running a large number of tests (for example running a benchmark suite on a processor design), each independent run requires system resources such as a DRAM storage and bandwidth to support it. Allocating a cheap board for each test scales testing throughput linearly, while investing in larger FPGAs quickly becomes cost prohibitive.

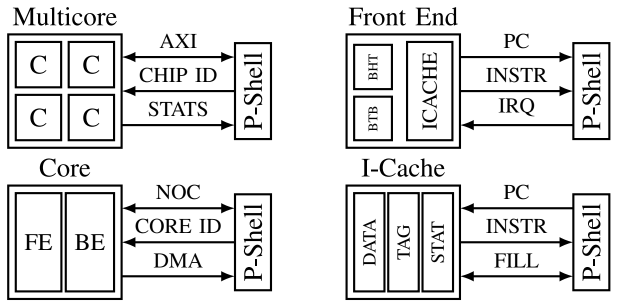


Fig. 5. Decomposition into smaller components leverage smaller FPGAs to verify modules in parallel as well as hyperfocus on specific performance or verification targets. In the above decomposition, verification engineers can tune their analysis for either a full multicore, a single core, the front end or just the instruction cache. Because the P-Shell supports both unidirectional and latency-insensitive links, any hardware interface can be exposed. Rather than cumbersome hardened RTL models, VPS software mimics the interface timings.

IV. Case Studies: ZynqParrot in the Wild

We first developed ZynqParrot for undergraduate/graduate architecture class performing full-stack analysis and optimizations to an open-source Linux-capable RISC-V multicore. Our main goals were:

1) Provide a cheap, flexible platform for designing and analyzing microarchitectural modifications to the core.
2) Avoid supporting all laptop to FPGA mappings by standardizing the host system (the Zynq PS).
3) Synchronize co-simulation and co-emulation execution to minimize FPGA debugging.
4) Make the platform robust to fatal RTL bugs by construction, impossible to hang the host system.

However, we quickly realized that the abstract capability of interacting out-of-band with arbitrary devices was applicable in a large number of diverse settings. In this section we describe a few use-cases that we have found for ZynqParrot since its inception.

A. A Scale-Down ZP-Farm Cluster

We built the first ZynqParrot on TUL Z2 [48], inexpensive educational boards available at an academic discount. Z2 boards are out-of-box compatible with the open-source Xilinx Pynq [56] SDK, providing a Python-based interface for bitstream programming, peripheral management and VPS 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. Most FPGA development boards, including the Z2, feature a watchdog timer which forces a reset upon hanging the board. In academia, this feature is invaluable to ensure that inexperienced students can always access their board.

Needing a more structured approach to coherently integrate a large number of boards, we designed the (*ZP-Farm*), a scalable cluster of network-attached FPGAs running ZynqParrot. All ZynqParrot data is stored on a host system and shared with the boards through a network-based distributed filesystem. 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 6(b) is built with commodity components: USB-Ethernet controllers, a network switch, and hand-cut plexiglass shelves. Comprising 20 Ultra96v2 boards, this ZP-Farm cost around $4500 and supports multiple projects and Continuous Integration (*CI*) runners for a modestly sized research group. Based on Table I, we estimate that a ZP-Farm outperforms in TCO after less than a full year of usage.

Maintaining a ZP-Farm is typically as simple as ensuring the central network switch is remotely accessible. If the watchdog timers are properly configured, any temporary glitch with the board has a fail-safe backup and connections can generally be restored after reboot. For further robustness, a remote network-attached reset switch (or Raspberry Pi [21]) 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 6(a), ZynqParrot easily supports heterogeneous ZP-Farms 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.

To ensure the correctness of the BlackParrot execution and the observed performance data, user needs to be able to verify the correct execution of a benchmark during its runtime. For example, ZynqParrot's infrastructure can also be used to extract instruction commit information of a RISC-V core and cross-verify it with Dromajo [18] abstract software-based golden model of DUT in VPS. Similarly, on each instruction commit and register write, corresponding information is written to asynchronous FIFOs that can gate the DUT clock while VPS drains and verifies the data. Using this feature, ZynqParrot can be integrated into the CI as part of the chip development cycle. On a major change, FPGAs can be used to accelerate design verification using longer benchmarks that are impractical for RTL simulation.

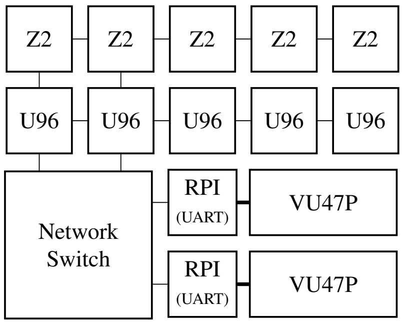
B. Microarchitectural Optimization: Catch-up ALU

In addition to modelling interface timings, the P-Shell can be used for verification of BlackParrot logic by

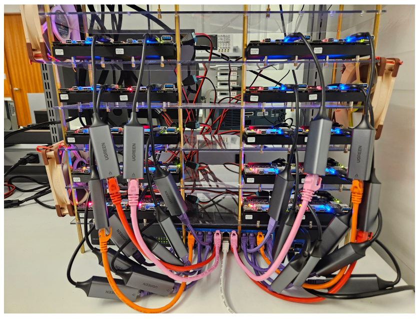
extracting commit information and cross-verifying it with Dromajo [18], an abstract software-based golden model. Upon each commit PC, instruction metadata, and writeback information are written to asynchronous FIFOs. VPS backpressure gates the DUT clock as VPS drains the commits. With cycle-accurate co-emulation, ZynqParrot can be integrated into the CI as part of the chip development cycle.

Because ZynqParrot is able to maintain cycle-accuracy with arbitrary bandwidth instrumentation, it enables deep insight into subtle microarchitectural bottlenecks. Previous works have proposed sampling-based architectures that accurately detect long latency stalls such as page table walks and L1 cache misses, but cannot diagnose ultra fine-grained stall sources such as irregular dependency bubbles. Section VI more thoroughly explores the trade-off between emulation speed and attribution accuracy, demonstrating the need for ultra fine-grained sampling to detect certain types of microarchitectural bottlenecks.

After adding synthesizable stall counters to the P-Shell, Figure 7 shows the cycle-stack breakdown of stalls during execution of CoreMark [25]. 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



(a) ZynqParrot 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 VPS 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.

Fig. 6. ZP-Farm can be configured for a variety of Pareto frontiers along cost, capacity and design parallelism. For a design space exploration of heterogenous 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.

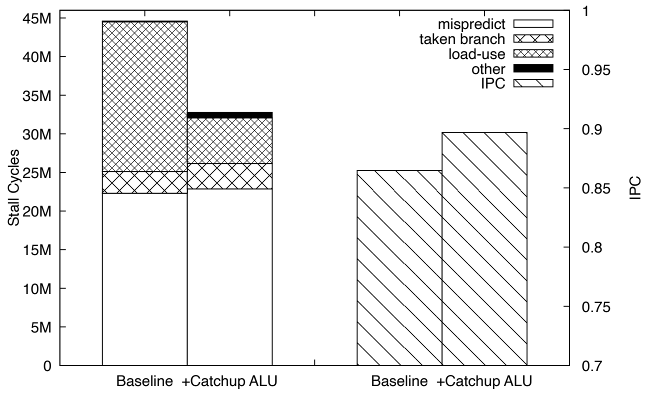


Fig. 7. After basic core optimization, remaining low latency stalls (1-5 cycles) are difficult to detect via coarse-grained sampling. Tailored event counters can identify problematic categories, but lose PC association during aggregation. ZynqParrot allows VPS software to monitor stalls at a per-PC, per-cycle granularity.

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.

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 (Scale-Up) simulation model, we implement an RTL version of the idea in ZynqParrot to evaluate marginal performance gains.

The Catch-up ALU resides in EX2, parallel with the second stage of the D$ access. When an integer or branch instruction

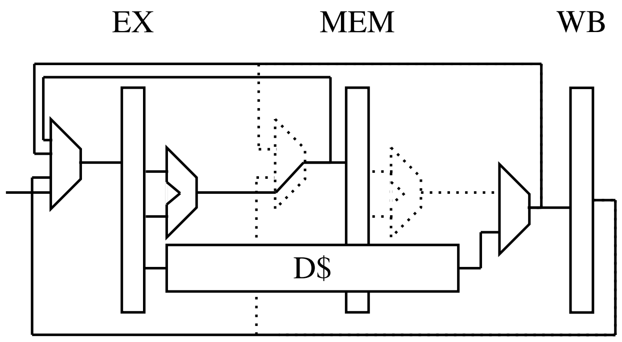


Fig. 8. A second Catch-up ALU and set of bypass multiplexers allows the Catch-up ALU to execute pipelined instructions. However, a dependent non-integer instruction following a Catch-up operation will cause a bubble.

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 8 illustrates Catch-up ALU modifications to a sample five-stage pipeline.

As shown in Figure 8, the Catch-up ALU reduces load-use stalls from 43% of stalls to 18% of stalls, resulting in an

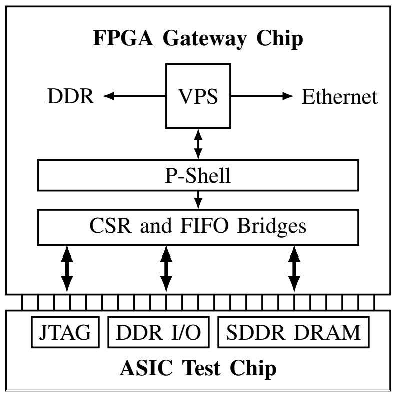
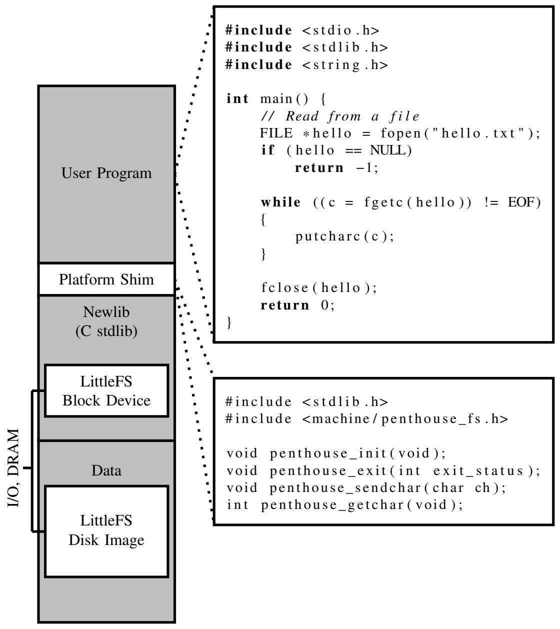


Fig. 9. When taping out a chip verification is scoped sequentially, starting with C++ and RTL models and finishing with annotated gate-level simulations. Differences in simulation and bring-up environments prevent sharing infrastructure between pre- and post-silicon environments. ZynqParrot unifies tape-in and tape-out infrastructure, reducing maintenance times and accelerating bring-up.



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    // Read from a file
    FILE *hello = fopen("hello.txt");
    if (hello == NULL)
        return -1;

    while ((c = fgetc(hello)) != EOF)
    {
        putcharc(c);
    }

    fclose(hello);
    return 0;
}
```

```
#include <stdlib.h>
#include <machine/penthouse_fs.h>

void penthouse_init(void);
void penthouse_exit(int exit_status);
void penthouse_sendchar(char ch);
int penthouse_getchar(void);
```

Fig. 10. PanicRoom provides filesystem and I/O operations to POSIX applications. Platform support needs 4 non-portable syscalls: init, exit, sendchar and getchar. All other syscall functionality is platform-independently provided by the PanicRoom libgloss implementation. Programs cannot differentiate between PanicRoom or a full OS, simply running benchmarks which otherwise require esoteric environments.

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 1.04×, 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 ZynqParrot allows architects to easily identify potential bottlenecks as well as confirm both the positive and negative effects of their proposed improvements.

C. ZynqParrot ASIC Bring-up Boards

During its initial evolution, ZynqParrot was also used as the design, prototype and bring-up infrastructure for a 14M gate, 28 nm ASIC developed by a boutique FPGA-based research and development firm. As a first generation test chip, operational mode fallbacks were essential. For instance, an experimental open-source LPDDR controller was taped out to accelerate applications. However, it was essential that the chip function enough to bring up all components independently. Additionally, it was not only possible but expected that during experiments on the chip to bring-up new subsystems, debug systemic issues and generate Shmoo plots [11] for operational PPA, the chip will fail to respond, generate illegal traffic or otherwise operate out of specification.

As shown in Figure 9, ZynqParrot leverages the P-Shell to allow bring-up software to interact with subsections of the DUT for large and slow gate-level simulations. Test chips are able to offload their memory controllers and I/O devices to the VPS, which allows pre-silicon bring-up to verify fallback functionality much earlier in the life-cycle. In the other direction, the flexible P-Shell can connect to a wide variety of bitbanged configuration devices (JTAG, SPI, I2C), higher performance I/O links (CAN, UART, custom SERDES), and DRAM (LPDDR, SDDR, HBM). In addition to bringing up the actual chip with ZynqParrot, users are able to substitute out fully detailed simulated execution models for faster-simulating smoke tests.

Using ZynqParrot, the ASIC was able to pass initial smoke tests on the first day of bring-up despite power and packaging problems that prevented it from operating in normal voltage operations. By using non-blocking registers to bitbang and configure the attached test chip, the team was able to quickly iterate and explore modes without hanging the system, needing to re-flash the FPGA for different tests or debug the infrastructure itself.

V. PanicRoom: Ultra-Portable Baremetal Benchmarking

Due to the complexity of benchmarking experimental processor designs, architects normalize performance across a wide range of applications such as [12], [15], [30], [31]. There are several significant challenges associated with reusing the same applications for fabricated chips as for pre-silicon designs: notably, the scale of commercial benchmarks are incompatible with the massive slowdowns of RTL simulation. Prior works have proposed reducing input set size [35], [47] and reducing instruction count through statistical sampling [29], [50]. Other approaches is creating targeted benchmarks which are intrinsically small and portable [25], [28], [41], but these have questionable correlation to high-performance microarchitectures.

TABLE II
PANICROOM SUPPORTS HOST FUNCTIONALITY MORE PORTABLY THAN COMPARABLE PROXY SOLUTIONS. WHILE A FULL DEBUGGING SOLUTION SUCH AS ARM SEMI-HOSTING PROVIDES AN END-TO-END EXPERIENCE, PANICROOM IS ABLE TO RUN IDENTICALLY IN SIMULATION AND HARDWARE, INCREASING VERIFICATION CORRELATION.

Proxy Solution	**Hardware Needed**[0]	**LoC (Userspace)**[1]	**Open-Source**
RISCV-PK	Host core	14157	✓
RAW Interface	Host core	6999	✓
ARM Semi-Hosting	Debugger	-	
PanicRoom	DRAM	**20**	✓

[0] While ZynqParrot provides a VPS host, PanicRoom can run fully untethered.
[1] We refer to non-benchmark, bare-metal code as "Userspace".

In addition to scale, commercial benchmarks also suffer from oversized scope. The most commonly evaluated suites rely on functions from the C standard library for I/O capability, filesystem operations and memory management. While it is interesting to evaluate the performance of an end-to-end system, 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 intentionally portable applications. Instead of glibc [24], embedded systems typically rely on smaller stdlib implementations, but these lack necessary system call compatibility.

To bridge this gap, we introduce *PanicRoom*: a minimal, mostly-platform-agnostic C standard library implementation that enables running POSIX applications on bare-metal systems. PanicRoom is built as a Board Support Package (BSP) on top of the light-weight C standard library newlib [23]. newlib elegantly separates system-specific functionality into an easily portable portion called libgloss. PanicRoom implements the libgloss functionality using an open-source, lightweight, DRAM-based filesystem designed for embedded flash memories, ARM LittleFS [7].

As shown in Figure 10, PanicRoom implements file I/O system calls by translating them to LFS function calls, which in turn operate on memory. In contrast, proxy-based solutions such as RAW [49] and RISCV-PK [22] work by packaging I/O calls and tunneling to a host core, which executes the actual filesystem functionality.

PanicRoom eschews platform-dependent syscalls, whereas previous works require porting syscalls to open-source simulators, commercial simulators, FPGA emulation frameworks, ASIC test boards and PCIe hosted chips (see Table 2). A more subtle benefit is that transforming the I/O emulation from an asynchronous host interaction to a synchronous function, which makes execution deterministic and easily reproducible.

VI. HIGH FIDELITY SAMPLING WITH ZYNQPARROT

While ZynqParrot accelerates design emulation, validating and optimizing performance at a Scale-Down granularity additionally requires deep introspection. Unfortunately, FPGA-enabled acceleration of designs is famously opaque and extracting microarchitecural information is unintuitive.

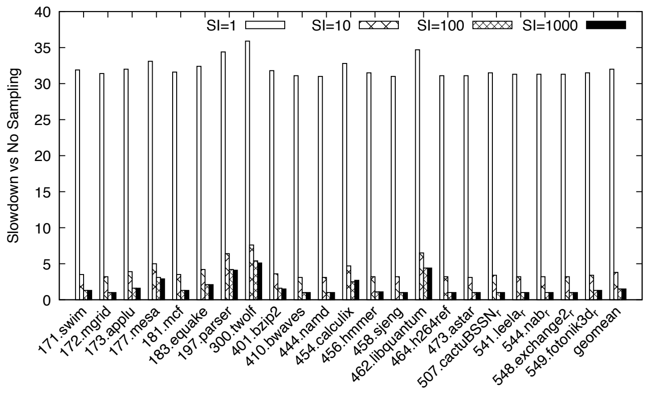


Fig. 11. ZynqParrot is able to dynamically switch co-emulation speed for sample rate. As sampling granularity decreases down to single step, there is a 32× slowdown. Therefore, best practice is to identify regions-of-interest and change sampling frequency to match importance.

Traditional solutions include using vendor IP such as JTAG scan-chains or Xilinx ILA [8] to extract signals. However, these solutions are slow, proprietary, have a high area overhead, and operate out-of-band, therefore lacking capability to maintain cycle-accuracy at full bandwidth. In this section, we explore how to leverage ZynqParrot's sampling infrastructure to characterize BlackParrot through time-proportional [26], [27] performance profiling.

To extract arbitrarily precise microarchitectural information from RTL, ZynqParrot leverages the same clock-gating mechanism used for I/O co-emulation. Users instantiate a parameterizable number of synthesizable performance counters in the P-Shell. These counters can be explicitly instantiated in the RTL, automatically generated by tools like FirePerf [34], or by hierarchically connecting PL counters to internal DUT signals. The latter does not require any modification to the DUT RTL and so is the simplest and least invasive solution.

When profiling BlackParrot, the 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. The DUT streams samples to VPS across asynchronous FIFOs at a configurable sample rate, if necessary clock-gating identically to how ZynqParrot manages emulation of interface timings. Critically, due to the backpressure mechanism, tuning profiling granularity becomes a simple tradeoff between slowdown and precision. TEA [27] and TIP [26] have demonstrated the benefits of Oracular stall classification, but concluded that the bandwidth overhead is impractical. Figure 11 illustrates that with ZynqParrot, an Oracle incurs only moderate overhead and enables unprecedented insight for performance debugging.

The VPS post-processes the stall information asynchronous to the DUT. Based on this information, profiler runtimes may chose to manipulate the sampling rate, perturb the DUT with emulated I/O traffic or monitor the runtime execution. Figure 11 shows the emulation slowdown for performance sampling of the core with error-free per-cycle sampling or

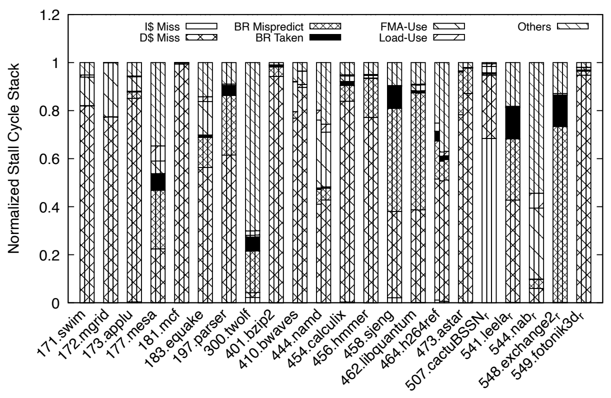


Fig. 12. Stall stacks for sampling intervals 1, 10, and 100. Due to time-proportionality, stall stacks do not generally vary across sampling intervals. However, a few benchmarks such as 454.calculix and 464.h264ref have variances as high as 6.2%. Oracular sampling through ZynqParrot is able to accurately identify these stall sources.

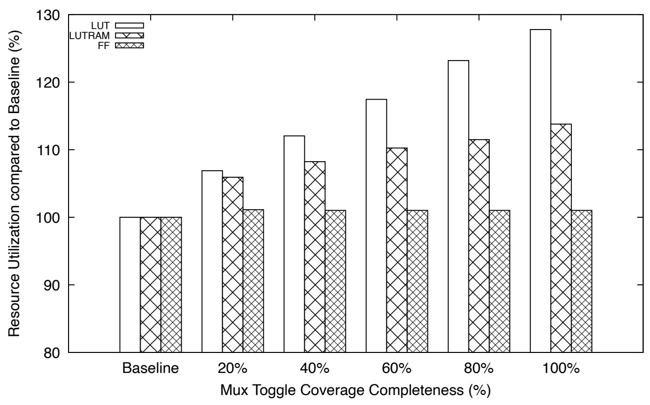


Fig. 13. All 3284 Mux Toggle coverpoints for BlackParrot consume an extra 13% LUTs and 27% FFs. If the overhead is unacceptable, designers may use ZP-Farm clusters to parallelize coverpoints and lower per-board overheads.

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. Due to the time-proportional nature of the profiler, stall stacks do not vary across sampling rates, as shown in Figure 12. There are two resulting modalities for performance profiling in ZynqParrot: running a coarse-grained regression suite to gain a sense of important stall categories, and running a fine-grained analysis to produce Oracular stall attributions to individual PCs.

VII. Scalably Acclerating Coverage Collection with ZynqParrot

Code coverage is paramount in pre-silicon verification [14]. Because of the enormous complexity of a large design, any simulation-based approach is merely sampling its state space. Still, state-of-art functional verification flows rely on coverpoints and covergroups to identify testing gaps. Recently, slow RTL simulation speeds have motivated verification teams to accelerate coverage collection using FPGAs. ZynqParrot supports automated, low-overhead coverage extraction through synthesizable coverpoints. In this section, we explore the design space and methodology for monitoring and extracting Mux Toggle coverage introduced by RFUZZ [37] using ZynqParrot[3].

ZynqParrot coverpoints comprise all mux select signals, based on the expectation that the cumulative select-signal toggles correlate with the exercising of all individual control paths in the hardware. To automate identification of the coverpoints from the design RTL, we employ a SystemVerilog parser, Surelog [2], to first generate an abstract syntax tree representation of the elaborated design, and then execute a customizable visitor at the root of the tree. Our visitor accumulates SystemVerilog structural elements that are likely to translate into Mux select-signals: the conditions in branch statements and ternary operations, while intelligently ignoring static expressions. The visitor also identifies case statement conditions for extracting FSM coverage for a more holistic coverage assessment.

Once identified, coverpoints are implemented as single-bit registers, unlike previous hardware coverpoint implementations [36] which use saturating counters to track toggles. Figure 13 shows the incremental overheads of instrumenting the BlackParrot processor, a relatively control-heavy design. As a result, ZynqParrot is able to obtain finer-grained coverage increments efficiently at the cost of under-representing coverage, thus enabling coverage-guided fuzzing. In most automated verification environments, under-representation is preferable to over-representation, as additional runtime can fill in the omitted toggles. Moreover, incremental coverage can be profiled dynamically when tracking coverage throughout program execution. Future fuzzers may take advantage of this interactivity in ZynqParrot through early termination and directed mutation of test programs.

[3]We exclude full cross-covergroup tracking which would assess *all* control paths in the hardware, as the state space quickly becomes unmanageable.

VIII. Related Work

While ZynqParrotshares 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 ZynqParrot.

A. Gate-Level Accelerated Emulation

Teams desiring a turnkey solution to RTL emulation employ commercial tools for FPGA-accelerated design modeling, such as Cadence Palladium [16], Synopsys Zebu [44] and Mentor Veloce [38]. Unfortunately this convenience is costly, with obfuscated pricing up to millions of dollars. In contrast ZynqParrotis free and open-source, with an initial required investment up to hundreds of dollars.

TABLE III

COMPARED TO OTHER FPGA EMULATION PLATFORMS, ZYNQPARROT 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, ZYNQPARROT 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]	**Coverage**	**Vendor-Agnostic**	**Open-Source**
Commercial [2]	High-End Cluster	1:1	Proprietary	✓	✓	✓		
FireSim [33]	Cloud Rental	N:N	AWS F1	✓		✓		✓
SMAPPIC [19]	Cloud Rental	N:1	AWS F1					✓
FreezeTime [40]	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 [43] (albeit at significant slowdown). This ability is essential in emulation-system debugging.

[2] We combine Synopsys Zebu [44], Cadence Palladium [16] and Mentor Veloce [38] with similar features and limitations.

B. Emulating Large Systems with FPGAs

FireSim [33], DIABLO [45] and SMAPPIC [19] focus on scaling up emulations 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 [3] infrastructure, the emulation model relies on proprietary vendor libraries for the hardened AWS shell as well as PCIe DMA interfaces. As ZynqParrotfocuses 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.

C. Decomposed FPGA emulation

Similar to a Scale-Down methodology, Protoflex [20] and FAST [17] 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, ZynqParrotallows for cycle-accurate emulation of arbitrary RTL so that architects can easily validate and debug performance with the deep introspection that RTL provides.

FreezeTime [40] uses time multiplexing for architectural virtualization of system components. Similar to ZynqParrot, Freezetime leverages BUFGCE FPGA primitives to stall emulated blocks while virtualized blocks process cycle-accurate timing models. However, ZynqParrotachieves greater flexibility and lower resource overheads by executing standard C++ timing models in the P-Shellrather than custom control logic per virtualized interface.

D. 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 [34] provides two categories of microarchitectural analysis: commit tracing via TraceRV and out-of-band hardware profiling via AutoCounters. ZynqParrotsupports not only commit tracing and out-of-band event counters via P-ShellCSRs but also dispatch-time stall tracing, allowing for deeper debugging insights. Additionally, ZynqParrotis written in standard SystemVerilog rather than Chisel [10], making it more familiar to hardware designers.

TEA [27] and TIP [26] 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 ZynqParrotcombines commit-stage cycle attribution with cycle accurate tracing, it is able to accurately attribute stalls without any sampling error, as well as trade co-emulation speed for sampling accuracy.

E. FPGA-based Coverage Collection

FirePerf [34] injects synthesizable coverpoints and then extracts coverage through a scan-chain. Instead, ZynqParrotautomatically instruments designs and extracts stall data through the P-Shell, without dedicated scan hardware. Simulator Independent Coverage [36] introduces a flow for injecting hardware coverpoints into Chisel [10] designs through the introduction of a new *cover* keyword. In contrast, ZynqParrotcoverage instrumentation uses standard SystemVerilog primitives.

IX. CONCLUSION

We present ZynqParrot, a Scale-Down FPGA-based modelling platform capable of non-interfering, cycle-accurate co-emulations of arbitrary RTL designs. Vendor-agnostic and fully open-source, ZynqParrot provides architects with a accurate, convenient and low-cost infrastructure to prototype designs. ZynqParrot is a cheaper alternative to FPGA cloud infrastructures for small-scale experiments and provides vendor-agnosticism. An iterative Scale-Down/Scale-Up methodology allows architects to focus on subtle microarchitectural optimizations and avoid re-analysis of issues that only exist at either scale.

In this work we have explored diverse use-cases for ZynqParrot: distributed acceleration for a class of architecture students, performance debugging, functional verification and tapeout bringup for a complex 28 nm SoC. These studies are meant to illustrate ZynqParrot's fitness for teaching, research and industrial development. We examined instrastructure enablements: ultra fine-grained sampling and hybrid FPGA-software coverpoint strategies. We believe the ZynqParrot, the P-Shell interface and its open-source co-simulation libraries will accelerate RTL prototyping across teams and fields, helping develop better architectures faster.

References

[1] Alibaba, "https://www.alibabacloud.com/product/computing," 2023.

[2] C. Alliance, "https://github.com/chipsalliance/surelog," 2023.

[3] Amazon, "Amazon web services. 2022. amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/," 2023.

[4] Amazon, "https://aws.amazon.com/ec2/spot/pricing/," 2023.

[5] AntMicro, "https://github.com/renode/renode," 2023.

[6] ARM, "https://developer.arm.com/documentation/102202/0300/axi-protocol-overview," 2023.

[7] ARM, "https://github.com/littlefs-project/littlefs," 2023.

[8] K. Arshak, E. Jafer, and C. 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.

[9] AVnet, "https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/," 2023.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1216–1225. [Online]. Available: https://doi.org/10.1145/2228360.2228584

[11] K. Baker and J. Van Beers, "Shmoo plotting: The black art of ic testing," *IEEE Design & Test of Computers*, vol. 14, no. 3, pp. 90–97, 1997.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "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.

[13] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[14] Brahme and Abraham, "Functional testing of microprocessors," *IEEE transactions on Computers*, vol. 100, no. 6, pp. 475–485, 1984.

[15] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[16] Cadence, "https://www.cadence.com/en_us/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html," 2023.

[17] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "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.

[18] ChipsAlliance, "https://github.com/chipsalliance/dromajo," 2023.

[19] G. Chirkov and D. 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.

[20] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, pp. 1–32, 2009.

[21] R. P. Foundation, "https://www.raspberrypi.org," 2023.

[22] R.-V. Foundation, "https://github.com/riscv-software-src/riscv-pk," 2023.

[23] FSF, "https://sourceware.org/newlib/," 2023.

[24] FSF, "https://www.gnu.org/software/libc/," 2023.

[25] S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.

[26] B. Gottschall, L. Eeckhout, and M. Jahre, "Tip: Time-proportional instruction profiling," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 15–27.

[27] B. Gottschall, L. Eeckhout, and M. Jahre, "Tea: Time-proportional event analysis," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[29] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[30] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[31] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[32] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.

[33] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 29–42. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014

[34] S. Karandikar, A. Ou, A. Amid, H. Mao, R. Katz, B. Nikolić, and K. Asanović, "Fireperf: Fpga-accelerated full-system hardware/software performance profiling and co-design," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 715–731. [Online]. Available: https://doi.org/10.1145/3373376.3378455

[35] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the spec 2000 benchmark suite for simulation-based computer architecture research," *Workload characterization of emerging computer applications*, pp. 83–100, 2001.

[36] K. Laeufer, V. Iyer, D. Biancolin, J. Bachrach, B. Nikolić, and K. Sen, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 606–615. [Online]. Available: https://doi.org/10.1145/3582016.3582019

[37] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[38] Mentor, "https://eda.sw.siemens.com/en-us/ic/veloce/," 2023.

[39] F. Mirasol, "The principle of scaling down," *BioPharm International 32*, 2019.

[40] S. Mosanu, J. Fixelle, M. N. Sakib, K. Skadron, and M. Stan, "Freeze-time: Towards system emulation through architectural virtualization," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2023, pp. 129–136.

[41] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[42] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri *et al.*, "Blackparrot: An agile open-source risc-v multicore for accelerator socs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[43] W. Snyder, "https://github.com/verilator/verilator," 2024.

[44] Synopsys, "https://www.synopsys.com/verification/emulation/zebu-server.html," 2023.

[45] Z. Tan, Z. Qian, X. Chen, K. Asanovic, and D. Patterson, “Diablo: A warehouse-scale computer network simulator using fpgas,” *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 207–221, 2015.
[46] M. B. Taylor, “Basejump stl: Systemverilog needs a standard template library for hardware design,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.
[47] R. Todi, “Speclite: using representative samples to reduce spec cpu2000 workload,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 15–23.
[48] TUL, “https://www.tulembedded.com/fpga/productspynq-z2.html,” 2023.
[49] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua *et al.*, “Baring it all to software: Raw machines,” *Computer*, vol. 30, no. 9, pp. 86–93, 1997.
[50] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, “Simflex: statistical sampling of computer system simulation,” *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.
[51] H. T.-H. Wong, *A superscalar out-of-order x86 soft processor for fpga*. University of Toronto (Canada), 2017.
[52] Xilinx, “Device reliability report (ug116),” Xilinx, Tech. Rep., 2023.
[53] Xilinx, “https://docs.xilinx.com/r/en-us/pg195-pcie-dma,” 2023.
[54] Xilinx, “https://docs.xilinx.com/r/en-us/ug1144-petalinux-tools-reference-guide/introduction,” 2023.
[55] Xilinx, “https://www.xilinx.com/products/boards-and-kits/device-family/nav-zynq-7000.html,” 2023.
[56] Xilinx, “http://www.pynq.io/board.html,” 2023.