Cocotb-PYNQ-PR: From Co-Simulation to Deployment, A Unified DFX Framework for PYNQ

Emir Guevara, Chaitanya Sharma, Gavin Lusby
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
{e2guevar,chaitanya.sharma,gdplusby}@uwaterloo.ca

Nachiket Kapre
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
nachiket@uwaterloo.ca

***Abstract*—Dynamic Function eXchange (DFX) enables runtime reconfiguration of FPGA logic, yet its adoption on PYNQ is held back by three barriers: assembling a DFX design takes hundreds of lines of manual Tcl; PYNQ has no native way to simulate a runtime module swap while the static design keeps running, so each debug iteration costs a full ≈20–31 min Vivado build; and the default reconfiguration path loads partial bitstreams far below hardware capability. We present Cocotb-PYNQ-PR, a Python framework that, from a single YAML configuration, generates the complete Vivado DFX flow—block design, decouplers, per-partition DMA, AXI4-Stream routing, and partial bitstreams—for designs matching its AXI4-Stream/AXI4-Lite shell. Its central contribution is a swap-aware co-simulation environment: each reconfigurable module (RM) runs as a separate OS process behind a generated DPI bridge, so partial reconfiguration becomes process replacement—with an optional shared-library swap that cuts swap latency 18×—and the same Python test exercises an RM swap in simulation exactly as on hardware, turning a ≈20–31 min build-and-deploy round trip into a 69 s first-frame check. The same workflow then deploys through either the standard PYNQ PCAP path or an integrated high-throughput ICAP backend [1] without changing application code, reducing practical DFX development on PYNQ to RTL, a YAML configuration, and a Python test.**

***Index Terms*—partial reconfiguration, PYNQ, cocotb, hardware verification, dynamic function exchange, FPGA**

I. Introduction

Checking whether a single partial-reconfiguration swap behaves correctly in a PYNQ application today has no native simulation path that preserves a running static design: the developer must run a full Vivado build and program the board—a ≈20–31 min round trip—before observing even one result. That round trip is the price of Dynamic Function eXchange (DFX), the appealing feature of modern Field-Programmable Gate Arrays (FPGAs) that lets part of the programmable logic be reconfigured at runtime without disrupting the rest of the design. The PYNQ framework [2] extends this appeal to hardware developers by providing a Python API for the Xilinx Zynq and Kria platforms.

Despite its appeal, DFX adoption on PYNQ remains limited by three practical barriers: build complexity, verification, and runtime loading overhead. For verification, PYNQ has no native path to simulate runtime RM-swap behavior before hardware deployment, so a developer must complete the entire build flow and deploy to physical hardware before observing whether a reconfigurable module (RM) swap produces correct results. For runtime loading, while the Processor Configuration Access Port (PCAP) offers an AMD-reported non-secure throughput of 145 MB/s [3] (the theoretical 32-bit×100 MHz datapath bound is 400 MB/s), the observed throughput through the default PYNQ reconfiguration path is roughly 57× lower (2.56 MB/s measured; Table II)—and still ≈21× below the realizable `fpga_manager` path (53.93 MB/s)—due to Python overhead, memory allocation, and driver latency. For applications that swap accelerators frequently, this gap makes DFX on PYNQ impractical despite the hardware being capable of better.

In this work, we present Cocotb-PYNQ-PR, a Python framework that simplifies the development, verification, and deployment of DFX designs on PYNQ. It makes the following contributions:

- An automated DFX build flow that generates the required Vivado design files, floorplanning constraints, static bitstream, and partial bitstreams from a single high-level YAML configuration for designs matching the current shell interface.
- A multi-process co-simulation flow that models partial reconfiguration with per-RM DPI-bridged processes—plus an optional dynamic shared-library swap for fast iteration—running the same Python test body in simulation and on hardware.
- A deployment path that exposes both standard PYNQ PCAP loading and an integrated VERSATILE ICAP backend through the same Python workflow, measuring the end-to-end cost of each backend inside a PYNQ DFX application.

II. Prior Work

Prior work has addressed several important aspects of dynamic partial reconfiguration (DPR), including verification, runtime reconfiguration performance, and toolflow automation. However, these challenges have largely been tackled in isolation, particularly for PYNQ-based development.

Dynamic partial reconfiguration remains difficult to verify in simulation. Prior surveys note that vendor-supported flows generally verify specific configurations, while accurately

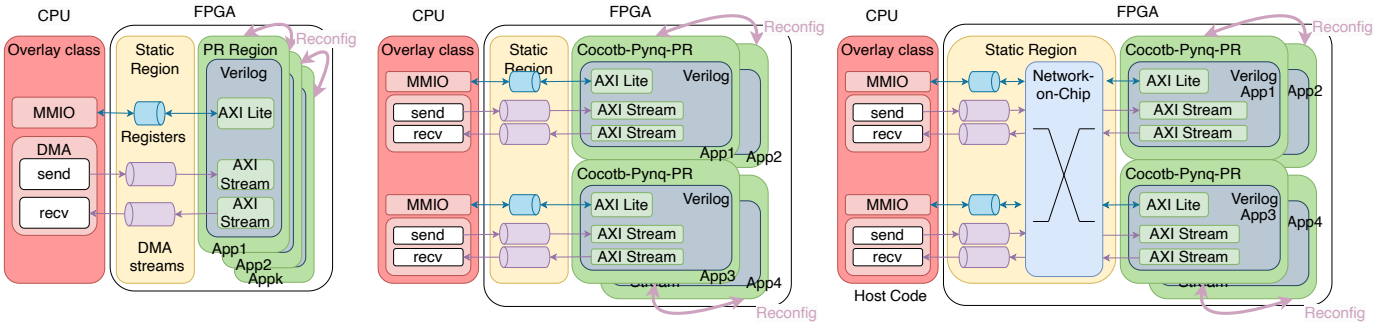


Fig. 1. Configuration options: (1) a single reconfigurable partition, (2) independent partitions with no inter-partition communication, (3) partitions chained through an AXI4-Stream switch.

modeling the reconfiguration event itself is much less supported [4]. In the AMD/Xilinx DFX flow, `pr_verify` is used to compare compatible routed configurations and confirm implementation consistency across checkpoints, rather than simulate the functional behavior of an RM swap [5], [6].

Runtime reconfiguration performance has also been studied extensively, showing that merely exposing the ICAP interface is not enough for fast partial reconfiguration. The vendor-provided AXI HWICAP core gives an embedded processor ICAP access but incurs substantial control and data-movement overhead, yielding low effective throughput [7]. At the other extreme, the UPaRC controller reaches up to 1.433 GB/s for uncompressed bitstreams, but only by storing them in on-chip block RAM (BRAM), whose capacity is far more limited than external DDR [8]. Practical high-performance reconfiguration thus needs not just ICAP access but an efficient controller and memory path; ICAP-based controllers such as ZyCAP [9] and, more recently, VERSATILE [1] strike a better balance using external memory.

Toolflow automation has likewise been explored, but without fully addressing the needs of PYNQ users. ZyCAP [9] demonstrated that routing bitstreams through the Internal Configuration Access Port (ICAP) from within the programmable logic yields significantly higher reconfiguration throughput than the PS-driven PCAP interface. ZyPR [10] extended this direction into an end-to-end build tool that automates DFX infrastructure generation for bare-metal and Linux environments, while HiPR and ExHiPR [11], [12] generate a design-specific DFX overlay directly from a high-level (HLS) description, using post-synthesis resource estimates to drive an automatic floorplanner. More recently, Coyote v2 [13] provides an open, layered DFX shell with software abstractions and fast runtime reconfiguration for datacenter (Alveo) FPGAs. However, none of these frameworks integrates with PYNQ or offers a path to simulate an RM swap before deployment—the two gaps this work targets.

PYNQ-oriented support for partial reconfiguration has also been explored. Early work extended the PYNQ software stack with support for partial reconfigurable overlays, exposing partial bitstreams through Python wrapper classes and hybrid software/hardware abstractions [14]. However, that approach focused on Python-side integration of pre-designed DPR overlays and explicitly did not address the design of DPR systems themselves. Native PYNQ partial reconfiguration support similarly loads prepared full and partial bitstreams through the overlay API [2], but leaves DFX design generation, floorplanning, partial-bitstream creation, and swap-aware simulation outside the PYNQ runtime. More recent efforts such as the Composable Pipeline [15] expose reconfigurable regions and AXI4-Stream routing through a PYNQ Python API, allowing users to compose and swap supported video-processing accelerators from Python without directly interacting with Vivado. The Composable Pipeline provides substantial flexibility within its composable overlay workflow, but it addresses a different use case from a unified flow for user-supplied RTL, automated DFX generation, and simulation-based verification of RM swaps prior to deployment.

III. Framework Overview

A. Design Entry and Build Automation

Cocotb-PYNQ-PR uses a shared project description to drive both simulation and hardware generation. The user provides RTL sources, a Python test, and a YAML configuration describing the target board, reconfigurable partitions, module variants, and optional runtime routing support. User-supplied RTL follows a fixed shell interface consisting of one AXI4-Stream [16] input, one AXI4-Stream output, and an AXI4-Lite slave interface for configuration.

During simulation, the framework generates the configuration needed to run the design through cocotb-pynq. This configuration includes the Direct Programming Interface (DPI) bridge setup that connects the persistent static-region simulation to independently simulated reconfigurable modules. Cocotb-PYNQ-PR wraps the lower-level cocotb-pynq execution flow behind the `pynq-pr sim` command, so users can launch simulation from the same project description used for implementation.

After generating the simulation-specific `sim_config.yaml` file from the user-provided DFX configuration, the framework instantiates cocotb-pynq's `PRSystem` and launches a `PRCocotbRunner` for the selected Python test, as shown in Listing 1. This wrapper hides details such as configuration-file management, test-

module resolution, and cocotb-pynq runtime setup, while preserving cocotb-pynq's underlying execution model.

```
from cocotbpynq._pr_engine import PRSystem
from cocotbpynq.pr import PRCocotbRunner

with PRSystem(config=str(config_path)) as system:
    print("Building RM binaries (cocotb mode)...")
    system.build(cocotb_mode=True)

    runner = PRCocotbRunner(
        pr_system=system,
        test_module=test_module,
        test_dir=test_dir,
    )
    runner.run()
```

Listing 1: Cocotb-PYNQ-PR simulation launcher built on cocotb-pynq's `PRSystem` and `PRCocotbRunner`.

B. Generated Overlay Architecture

From the YAML configuration, Cocotb-PYNQ-PR generates a DFX overlay in which each reconfigurable partition is paired with its own AXI DMA and DFX decoupler. The per-partition DMA provides a dedicated path for moving data between programmable logic and memory, allowing each reconfigurable module to be exercised and controlled independently at runtime. The decoupler is placed between the static logic and the reconfigurable partition so that signal interfaces can be safely isolated during partial reconfiguration, preventing invalid transactions or transient communication while a new module is being loaded. Fig. 1 illustrates the supported connection configurations: a single reconfigurable partition, multiple partitions with no inter-partition communication, and data exchange between partitions through an AXI4-Stream switch.

C. Runtime Reconfiguration Backend

Cocotb-PYNQ-PR supports two runtime backends for loading partial bitstreams. The default option uses the processor-driven PCAP path and follows the standard PYNQ reconfiguration flow [2]. While this provides straightforward compatibility with existing PYNQ usage, its effective throughput is limited by software overhead and driver latency, making it less suitable for applications that require frequent module swaps. To improve runtime performance, the framework can also generate an ICAP-based reconfiguration path using the VERSATILE DPR controller [1].

IV. Simulation Architecture

Verifying dynamic partial reconfiguration in-simulation is a fundamental and known gap in existing FPGA tooling. Vivado's DFX flow includes `pr_verify`, but this only checks interface compatibility between static and partial netlists, and does not simulate the functional behavior of an RM swap. Running the design in XSIM or ModelSim verifies a single elaborated netlist; to test a different RM, the designer must re-elaborate and restart the simulation from cycle zero. Because each configuration is baked in at elaboration time, the cost of conventional simulation scales with the number of configurations a developer exercises—each one a fresh elaboration of several minutes—rather than with the $N \cdot M$ distinct RMs that actually exist. Iterating across configurations is therefore impractical with conventional tools, even though the underlying logic need only be built once. Prior simulation-only libraries such as ReSim [17] address part of this gap by instantiating all $N \cdot M$ variants together with a simulation-only reconfiguration layer in a single elaboration, avoiding a fresh elaboration per configuration. However, all variants are compiled into one netlist—there is no per-RM isolation, and changing any variant re-elaborates the whole model—and the swap itself is a simulation-only-layer activation of pre-elaborated RM instances rather than process or library replacement with independently built RM models. Our approach instead compiles each RM into a separate binary that executes its real logic and is swapped at runtime, so variants share no elaboration and adding or changing one never re-elaborates the others; the $N \cdot M$ binaries then compose into any system configuration through a runtime swap rather than a fresh elaboration.

A. Background: cocotb-pynq

cocotb-pynq [18] is a co-simulation layer that turns a cocotb/Verilator testbench into a drop-in substitute for the PYNQ runtime. It re-exports PYNQ's `Overlay`, `MMIO` (memory-mapped I/O), `DMA`, and `allocate` as Python classes that drive a Verilator model through cocotb's Verilog Procedural Interface (VPI) bridge, so the same Python test runs over either a physical board or a simulated processing-system↔programmable-logic boundary without code changes. cocotb-pynq targets a *single* elaborated design: clocking, reset, signal-handle resolution, and the cocotb scheduler all bind once at simulation start. Module instantiations are part of that elaboration and cannot be substituted mid-simulation, so DFX's defining behavior—replacing a partition's contents at runtime—falls outside what cocotb-pynq alone can model.

The contribution of this work is to relax exactly that restriction. We introduce a multi-process layer on top of cocotb-pynq in which each RM executes in its own OS process and reconfiguration becomes process replacement at the partition boundary. cocotb-pynq's static-region elaboration, VPI handle table, scheduler, and user-facing Python API are all preserved; only the *external* RM processes are swapped. The remainder of this section describes how that boundary is structured (Sections IV-B–IV-D), how the $N{+}1$ processes are kept causally coherent every cycle (Section IV-E), how an actual `pr_download` executes (Section IV-F), and a lower-latency dynamic-library swap variant (Section IV-G).

B. Design Space

The simulation must mirror the hardware asymmetry: the Python test emulates PS code, the static region is persistent, and RMs are ephemeral—swappable behind fixed partition boundaries.

We considered three alternatives. Hot-swapping the compiled RM inside a running Verilator process fails because cocotb's VPI handle table is frozen at elaboration time, so replacing a module corrupts its cached handles. Compiling

each RM as a shared library and loading it into the VPI-bearing cocotb process would likewise desynchronize that handle table—though once the RM is isolated from all cocotb/VPI state (Section IV-G), this becomes our fastest swap mechanism. Running the static region and each RM in independent simulator instances behind a Python façade would require reimplementing cocotb's asynchronous scheduling across processes. Distributed co-simulation frameworks [19], [20] attain high throughput precisely by communicating only across latency-insensitive boundaries; the static↔RM register interface here is not latency-insensitive, so preserving cocotb's per-cycle handshake across separate simulators would reintroduce prohibitive synchronization overhead. These failures motivate the boundary-level approach: a generated DPI bridge at the partition boundary, with each RM in a separate OS process over shared memory.

C. Multi-Process Architecture

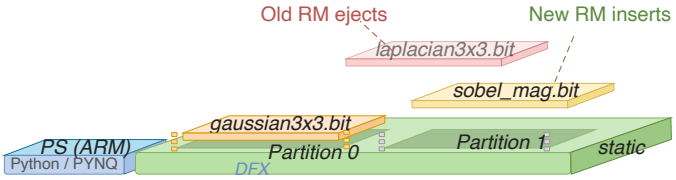
Our approach operates at the *boundary* level. On hardware the static region sees an RM only through its fixed boundary ports, so the static Verilator model need not contain the RM at all: a generated DPI bridge module with an identical port signature occupies the RM's slot in the static netlist, while the real RM logic runs in a separate process over shared memory. The static region's elaboration, cocotb's VPI handle table, and Python-side state stay undisturbed—only the external RM process changes—so killing and restarting an RM binary is a sub-millisecond OS operation, versus seconds to minutes to re-elaborate. Because DPI-C is standardized by IEEE 1800 [21] and shared memory is an OS primitive, the bridge is simulator-agnostic in principle, though our implementation targets Verilator.

The simulation consists of $N{+}1$ cooperating OS processes for a design with N reconfigurable partitions, as illustrated in Fig. 2:

- **Cocotb/static process:** Hosts the static region inside cocotb's Verilator instance, driven by the Python test through cocotb's VPI interface exactly as in a non-PR cocotb-pynq simulation. Each partition slot is replaced at build time by a *DPI bridge module* with an identical port signature that calls DPI-C functions to exchange data with an external process.
- **RM processes ($\times N$):** Each variant is independently compiled by Verilator into a self-contained binary; exactly one runs per partition, wrapping the real RM RTL in a generated DPI wrapper that reads and writes a per-partition shared-memory channel.

All $N{+}1$ processes synchronize at every simulated clock cycle via a shared-memory sense-reversing barrier [22]. When the Python test calls `overlay.pr_download('rp0', 'new_rm')`, the framework terminates the current RM process for partition `rp0`, starts the new variant's binary, and resumes barrier cycling—all within a few simulated clock cycles and without disturbing the static region or any other partition's RM process.

(a) Physical DFX (partition slots on Zynq PL fabric)



(b) Simulation mapping (DPI-bridge slots and RM processes)

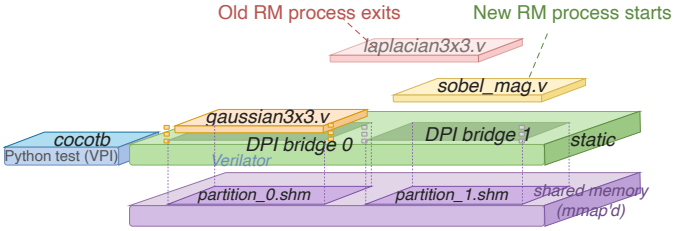


Fig. 2. Physical DFX and its simulation mapping, drawn as the same picture. **(a)** On hardware, the PS runs the Python/PYNQ application while reconfigurable modules eject and insert at fixed partition slots on the decoupled static fabric. **(b)** In simulation the structure is mirrored: the fabric becomes the single persistent cocotb/static Verilator process (the Python test drives it in-process via VPI), each partition slot becomes a DPI-bridge stub, and each RM becomes an OS process (`gaussian3x3.v`, `sobel_mag.v`) that the framework kills and restarts behind the bridge—so an RM swap is a process swap while the static region keeps running (mechanism detailed in Sections IV-D–IV-F).

D. DPI Bridge Architecture

The DPI bridge is the mechanism by which the static region communicates with external RM processes. Conceptually it plays the role of a co-emulation transactor in the spirit of the Accellera SCE-MI interface [23], which standardizes message-passing between a software model and a separate hardware model. Here the two "sides" are the static-region Verilator process and the per-partition RM process, connected through shared memory rather than an emulator link. The bridge exploits the fact that Verilator supports two foreign-function interfaces simultaneously: VPI (Verilog Procedural Interface, used by cocotb to drive signals from Python) and DPI-C (Direct Programming Interface, used for C++ function calls from SystemVerilog). These interfaces operate independently—cocotb's VPI callbacks and the bridge's DPI function calls execute in the same Verilator process without interference.

1) Static-Side Bridge: For each partition, the framework parses the RM's interface with slang [24] and generates a bridge module with the same module name, port list, and port widths as the RM it replaces. This module is a drop-in substitution in the static region's RTL source. The bridge contains no user logic; instead, it implements two clocked blocks: a *send* block that, for each to-RM port, writes the port's current value into the shared-memory outbox via a DPI-C call, and a *receive* block that reads each from-RM port's latest value from the inbox.

2) Shared Memory Channel Layout: Each partition's shared-memory channel is a page-aligned memory-mapped file with a fixed header plus cache-line-aligned to-RM and from-RM port slots (ports wider than 64 bits span multiple

slots). The header carries the flags that coordinate module termination and readiness during reconfiguration. To-RM slots are triple-buffered so the static and RM processes never write the same cache line in the same barrier phase; the leader copies outbox to inbox during phase B2 (Section IV-E), introducing exactly one cycle of boundary latency—matching a registered interface in hardware.

E. Cross-Process Synchronization

Correct simulation requires that all $N+1$ processes observe a consistent, causally ordered view of port values at each clock edge; letting processes run freely and poll shared memory would race the static region's read of a from-RM value against the RM's evaluation of the current cycle. We enforce ordering with an `mmap`'d sense-reversing barrier that divides each simulated clock cycle into three synchronization points. At **B1** all processes finish their negative-edge evaluation. At **B2** the cocotb process, acting as leader, copies every partition's outbox into its inbox—making the previous cycle's writes visible to the current cycle's reads—and services any pending Python-side MMIO or reconfiguration commands. At **B3** all processes evaluate their positive-edge logic and the DPI bridges exchange port values through shared memory. The B2 copy is what yields the one-cycle boundary latency: the static side writes outboxes at B3 of one cycle and reads inboxes at B3 of the next, exactly as a registered hardware boundary behaves.

F. Reconfiguration Protocol

Because the cocotb process is embedded inside Verilator and cannot itself manage OS processes, `overlay.pr_download()` uses a cross-process control plane: the Python coroutine posts a request into a dedicated control mailbox, and a control thread in the parent Python process drives the swap while the coroutine keeps the simulation clock advancing. Fig. 3 shows the resulting timeline in three steps:

① *Request:* the coroutine posts the `pr_download()` command into the control mailbox and the clock keeps advancing.

② *Swap:* the old RM exits cleanly at a barrier boundary while the static region spin-waits, and the new binary starts and resets.

③ *Ready:* the new RM signals that it is ready and barrier cycling resumes.

Uninvolved partitions continue cycling throughout; the barrier participant count is not modified.

G. Dynamic-Library RM Swap and Polling Latency

The process-swap path forks a new RM per swap, so each transition is dominated by OS work—process exit, `execve`, dynamic-linker startup, Verilator-runtime init—that has no analog in hardware DFX. To isolate the swap mechanism's inherent cost, the build also emits each RM as a position-independent shared library with a small C ABI (initialize, step, destroy). A persistent per-partition host—holding the

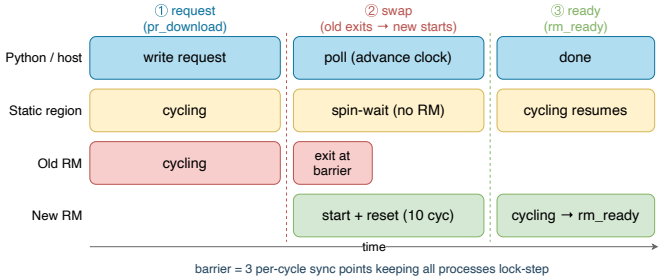


Fig. 3. Reconfiguration timeline. The old reconfigurable module (RM) keeps cycling until it exits at a barrier boundary; the static region spin-waits while no RM is present; the new RM then resets and resumes once ready.

DPI wrapper and shared-memory channel but no cocotb/VPI state—then swaps by `dlclose`/`dlopen`, with no `fork`, `execve`, or runtime re-init. Unlike the option rejected in Section IV-B (which loads into the VPI-bearing cocotb process), this loads only into the VPI-free host, so no handle-table conflict arises (Section V-D quantifies the gap).

It is important to draw a portability boundary here. The boundary DPI bridge of Section IV—the multi-process, shared-memory, process-swap architecture—rests only on DPI-C and OS primitives and is therefore portable in principle to any DPI-capable simulator. The dynamic-library fast path is not: it relies on Verilator compiling each RM into a position-independent C++ model that can be linked as a shared object and advanced through a plain `init`/`step`/`destroy` C ABI. Closed-source event-driven simulators such as XSIM or ModelSim do not expose their compiled model as a `dlopen`-able, single-step library, so they cannot use this mechanism and would be confined to the portable process-swap path. The $18\times$ shared-library speedup is thus a Verilator-specific optimization layered on top of the simulator-agnostic bridge, not a generic property of the approach.

Because the swap request and the swap itself live in separate processes, the cocotb side waits by polling the control mailbox. We instrumented this path with a lightweight structured tracer (JSONL span/event records keyed by partition and sequence number). It showed that a single conservative 10 ms `time.sleep` dominated the pre-tightening swap latency—roughly 90% of the median wall-clock—masking an underlying state transition that, on the library path, completes in tens of microseconds. Tightening the poll to 100 µs exposed that hidden work at negligible CPU cost (the poll is a `time.sleep`, not a busy-wait), cutting the library-path median swap latency by an order of magnitude; on the process path it uncovers the `fork`/`execve`/Verilator-init cost that has no analog in hardware DFX. The same tracer drives the per-subphase breakdown in Section V-D.

H. Limitations and Verification Boundaries

Cocotb-PYNQ-PR imposes three structural constraints beyond those of plain cocotb-pynq:

1. each RM is restricted to a single AXI4-Stream input and output, plus an AXI4-Lite control interface;
2. the static region is fixed—user logic cannot be added outside the reconfigurable partitions (a planned extension);

3. a design supports at most four reconfigurable partitions on the Z1: the automated floorplanner divides the fixed reconfigurable area into horizontal clock-region strips, and the supported boards expose at most four such strips within the RP region.

Cocotb-PYNQ-PR is a functional co-simulation framework: it verifies post-swap functional correctness rather than the electrical behavior of the reconfiguration event, so transient effects such as decoupler glitches or bus contention during configuration are out of scope. Its per-cycle cross-process synchronization makes it best suited to interface and swap-sequencing verification over modest frames.

V. Evaluation: Vision Processing Pipeline

We evaluate Cocotb-PYNQ-PR on a two-stage image processing pipeline that chains a smoothing filter into an edge detector across two reconfigurable partitions. This pipeline exercises the full framework path: YAML-driven build automation, multi-process simulation with repeated partial reconfiguration, runtime chaining through the AXI4-Stream switch, and deployment on hardware through the same Python-facing PCAP/ICAP reconfiguration API.

A. Pipeline Architecture

The pipeline assigns partition `rp0` to a smoothing stage and `rp1` to an edge detection stage. Each partition has two swappable modules: `rp0` alternates between a 3×3 Gaussian blur and a 3×3 median filter, while `rp1` alternates between a 3×3 Laplacian operator and a Sobel magnitude filter.

We generated all four filter kernels in Vitis HLS (high-level synthesis) from the Vitis Vision Library (xfOpenCV). Each HLS export produces a Verilog module with AXI4-Stream data ports and an AXI4-Lite control interface following the fixed shell interface required by Cocotb-PYNQ-PR. Pixels are encoded as 8-bit grayscale values in the low byte of each 32-bit AXI4-Stream word; frame dimensions are programmed at runtime through AXI-Lite registers.

The YAML configuration is shown below.

```yaml
axis_switch: true
sources: [rtl/wrappers/*.v,
          rtl/generated/**/*.v]
reconfigurable_partitions:
  - partition_name: rp0
    modules:
      - cell_name: gaussian3x3
        top: gaussian3x3_top
      - cell_name: median3x3
        top: median3x3_top
  - partition_name: rp1
    modules:
      - cell_name: laplacian3x3
        top: laplacian3x3_top
      - cell_name: sobel_mag
        top: sobel_mag_top
```

The `axis_switch` flag tells the framework to generate the AXI4-Stream switch (Fig. 1), enabling `ol.chain()` to route `rp0`'s output into `rp1` at runtime.

Listing 2 shows the test body used in both simulation and hardware. The dual-mode import block selects the cocotb-pynq backend or the PYNQ hardware backend depending on the execution context; all subsequent logic—bitstream loading, AXI-Lite configuration, DMA transfer, and frame-by-frame verification against Python reference filters—is shared. The test runs all four chained configurations over a video clip, reloading both partitions between configurations and verifying every processed frame.

```python
COCOTB_IS_RUNNING = "COCOTB_SYS_ARGV" in os.environ
if not COCOTB_IS_RUNNING:
    from pynq import allocate, MMIO
    from overlay import Overlay
else:
    import cocotbpynq
    from cocotbpynq import MMIO
    from pynq_pr.overlay_sim import Overlay, allocate

def configure_partition(ol, name, rows, cols): ...

def transfer_frame(dma, xb, yb, frame):
    np.copyto(xb, frame.astype(np.int32).reshape(-1))
    dma.recvchannel.transfer(yb)
    dma.sendchannel.transfer(xb)
    dma.sendchannel.wait(); dma.recvchannel.wait()
    return (np.array(yb) & 0xFF).astype(np.uint8)

def main(dut=None):
    ol = Overlay("vision_z1.bit")
    for rp0_mod, rp1_mod in CHAIN_CONFIGS:
        ol.pr_download("rp0", f"{rp0_mod}.bin", icap=True)
        ol.pr_download("rp1", f"{rp1_mod}.bin", icap=True)
        configure_partition(ol, "rp0", rows, cols)
        configure_partition(ol, "rp1", rows, cols)
        dma = ol.chain(["rp0", "rp1"])
        for frame in video_source(...):
            result = transfer_frame(dma, xb, yb, frame)
            expected = REFS[rp1_mod](REFS[rp0_mod](frame))
            check_frame(result, expected)
```

Listing 2: Unified video test body shared by simulation and hardware. The import block selects the backend; filter chaining, DMA transfers, and per-frame verification are shared across both paths.

B. Experimental Setup

We ran simulation on Verilator 5.036 [25], hosted on an Intel Xeon W-2295 workstation (18 cores at 3.0 GHz base, 24.75 MB L3 cache). The synthetic video workload consists of a scrolling horizontal bar pattern at 320×480 resolution (153,600 pixels per frame), with each pixel streamed as one 32-bit AXI4-Stream word through the DMA→filter→DMA path. We run 10 frames per chained configuration (4 configurations), totaling 40 frames per simulation run. Each frame is verified element-wise against a Python reference implementation of the corresponding filter chain.

To measure reconfiguration throughput on hardware, we generated overlays with one to four reconfigurable partitions on both the PYNQ-Z1 and Kria KV260. All hardware bitstreams and overlays were generated using AMD Vivado and Vitis HLS 2023.2 under a cold-cache clean build environment with `PR_BUILD_JOBS=4`. Reconfiguration measurements were performed on physical boards: the PYNQ-Z1 (xc7z020clg400-1) and Kria KV260 (xck26-sfvc784-2LV-c). The software stack ran PYNQ image v3.0.1 (Ubuntu 22.04 LTS, Linux kernel version 5.15.0). Since partial bitstream size is determined by the physical floorplan region rather than the module's logic complexity, varying the partition count changes how the reconfigurable area is divided and thus the per-partition bitstream size. We evaluated three runtime backends:

the default PYNQ PCAP path, PCAP through the Linux `fpga_manager` driver, and the ICAP-based backend. For each method, we preceded repeated timed trials with warmup runs and recorded the average reconfiguration time. We tested the ICAP backend at nominal and overclocked frequencies: 100 MHz and 200 MHz on the Z1, 200 MHz and 400 MHz on the KV260.

C. Simulation vs. Build Time

Fig. 4 compares the wall-clock time for a single Verilator simulation frame against the full Vivado DFX build for both target boards. The Verilator binary compiles in 40 s (a one-time cost), and each 320×480 frame takes an average of 29 s to simulate through the two-stage filter chain. A first-frame result is therefore available in about 69 s. By contrast, the Vivado build—including synthesis of all module variants, implementation of the base design and each child configuration, and bitstream export—takes 19.6 min on the Z1 and 30.9 min on the KV260. Against the time to first result (the one-time 40 s Verilator compile plus one 29 s frame, ≈69 s) this is a 17× speedup on the Z1 and 27× on the KV260; measured against the marginal per-frame cost alone (29 s) the ratio rises to 40.6× and 63.9×.

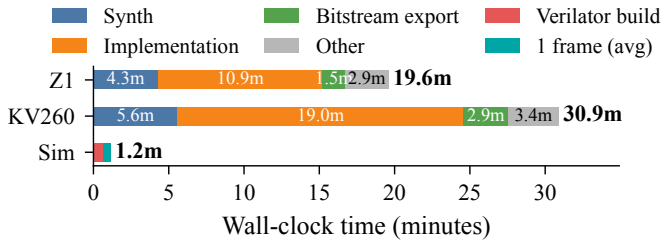


Fig. 4. Full Vivado DFX build (with phase breakdown) vs. a single-frame Verilator simulation (40 s compile + one 29 s frame at 320×480).

Because simulation cost scales linearly with pixel count while build cost is fixed, the number of frames at which cumulative simulation time equals the Vivado build is 39 frames for the Z1 and 63 frames for the KV260. In practice, a developer iterating on filter logic or verifying a new chained configuration needs only a frame or two to confirm correctness. The simulation path lets this happen in under two minutes (or about five frames in roughly three minutes) rather than waiting ≈20–31 min for a new bitstream.

D. Simulation Performance

The single-frame number in Fig. 4 is an aggregate of three underlying costs: the one-time Verilator compilation of the static region and RM binaries, the per-cycle DPI boundary traffic during the frame itself, and the cost of each `pr_download` when the chain changes mid-test. We characterise each below on four bundled example designs (Table I) that instantiate the connection configurations of Fig. 1: `axi_pr` and `switch_pr` are single-partition designs, while `dual_dma_pr` realises the no-inter-communication config and `stream_pr` chains two partitions over AXI-Stream. The vision pipeline (Fig. 4) is the same switch configuration at

TABLE I
BUNDLED EXAMPLE DESIGNS SPANNING THE FIG. 1 TOPOLOGIES, WITH PARTITION COUNT N AND VARIANTS-PER-PARTITION M.

Design	N	M	Topology
`axi_pr`	1	2	single RP, direct DMA
`switch_pr`	1	3	single RP via AXIS switch
`stream_pr`	2	2	two RPs chained (stream)
`dual_dma_pr`	2	3	two RPs, independent DMAs

N=2, M=2 carrying real image-filter kernels. Together they span $N \in \{1, 2\}$ partitions and $M \in \{2, 3\}$ variants across the full range of datapath topologies rather than a single point. We begin with the swap mechanism, the only component without an analog in real-hardware DFX.

Swap mechanism cost: The default process path repeats four OS-level phases at every swap: the old RM exits, `fork+execve` launches the new one, the Verilator runtime initializes, and the new process re-maps the shared-memory channel and DPI bindings. The library path keeps the persistent RM host process—and with it the shared-memory channel, the DPI bridge, and the barrier participants—in place; only the RM symbols are unbound and re-bound via `dlclose+dlopen`. Over 6,000 simulated swaps across four designs (Fig. 5), all measured with the tightened 100 µs poll, the gap is consistent per-example—process swaps cluster near 5.7 ms, library swaps near 0.3 ms—and the pooled distributions are fully separated, with median wall-clock falling from **5.70 ms** to **0.31 ms**, an **18**× reduction. The structured tracer of Section IV-G revealed that the original 10 ms control-mailbox poll dominated the pre-tightening median (≈90%) on both paths; tightening it to 100 µs exposed the true mechanism cost, leaving the process path at its irreducible `fork`/`execve` floor and the library path within the simulated-cycle floor of the protocol.

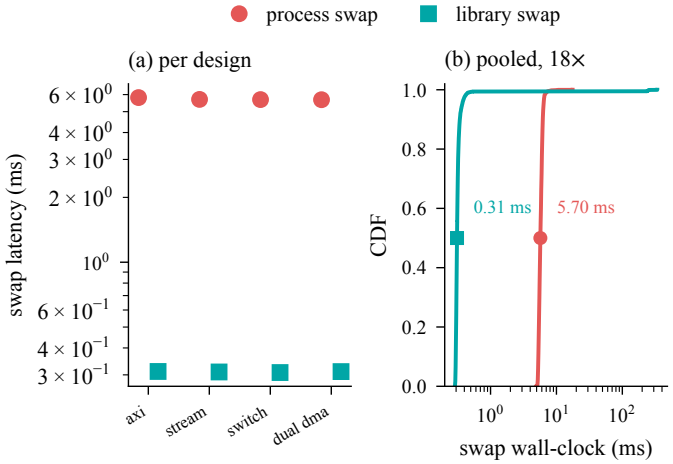


Fig. 5. Per-swap latency: process (●) vs. library (■) swap. (a) per-design medians; the ≈18× gap holds on all four. (b) pooled CDFs over 6,000 swaps; medians 5.70 ms (process) and 0.31 ms (library).

Build cost of supporting both swap mechanisms: Emitting both build variants (Fig. 6) adds only 10–15% wall-clock over the binary build—a small fixed cost for the order-of-magnitude latency reduction above. The dominant Verilator

TABLE II
MEASURED RECONFIGURATION THROUGHPUT. EFFICIENCY IS MEASURED/THEORETICAL (ICAP ROWS). †AMD-REPORTED NON-SECURE PCAP FIGURE, NOT A DATAPATH BOUND (400 MB/S).

Board	Port	Path	Freq. (MHz)	Throughput (MB/s) Theory	Meas.	Eff. (%)
Z1	PCAP	PYNQ	100	145.00†	2.56	–
		fpga_manager	100	145.00†	53.93	–
	ICAP	VERSATILE	100	400.00	203.59	50.9
			200	800.00	372.55	46.6
KV260	PCAP	PYNQ	200	–	10.57	–
		fpga_manager	200	–	237.49	–
	ICAP	VERSATILE	200	800.00	523.58	65.4
			400	1600.00	713.53	44.6

pass (SystemVerilog to C++ plus model compilation) is *identical* across variants and never repeated; the library variant adds only a position-independent re-compile of the per-RM driver and DPI sources plus a small shared-library link.

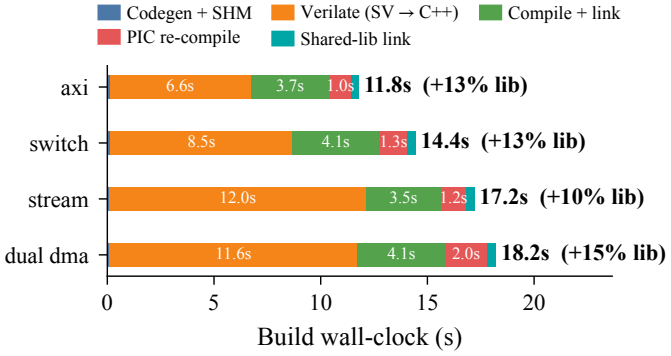


Fig. 6. Phase-decomposed build wall-clock per design (median, `PR_BUILD_JOBS=4`). The first three phases are the binary build used by the process-swap path; supporting the dynamic-library swap adds only the PIC re-compile and shared-library link (a +10–15% tail).

E. Reconfiguration Throughput

Table II reports end-to-end reconfiguration throughput through the runtime software stack. We do not claim a new controller microarchitecture; we quantify the cost of exposing standard PCAP loading and the VERSATILE ICAP controller through a common PYNQ interface, so the values include Python-side invocation, decoupling/recoupling, and per-backend control. The `PCAP (PYNQ)` row is the default PYNQ path; `PCAP (fpga_manager)` writes the bitstream directly to the Linux `fpga_manager` sysfs interface, separating software overhead from the PCAP interface itself.

The ICAP theoretical bound is datapath width × clock (32 bits/cycle): 400/800 MB/s at 100/200 MHz on the Z1 and 800/1600 MB/s at 200/400 MHz on the KV260. For Zynq-7000 PCAP, AMD reports an achievable non-secure 145 MB/s [3]—a vendor-measured figure, not a datapath limit (daggered in Table II).

Fig. 7 plots throughput against partial bitstream size for all methods on both boards (solid lines Z1, dashed KV260). Measured throughput rises with bitstream size and then flattens, as fixed runtime overheads are amortized over larger regions. The PCAP (PYNQ) path is consistently the slowest and most variable, confirming that software overhead, not the interface, dominates it. The integrated ICAP path provides a practical high-throughput deployment option within the same PYNQ workflow, and raising its clock frequency helps further, though sublinearly—reflecting residual control and data-movement overhead beyond the raw configuration clock.

The high-frequency ICAP points (200 MHz on the Z1, 400 MHz on the KV260) exceed the datasheet limits of 100/200 MHz [26], [27]. They are timing-clean in Vivado for the controller logic but drive the ICAP hard macro beyond its nominal rating. Across thousands of on-hardware reconfiguration cycles we observed no configuration errors, corruption, or lockups, and `ICAP_ERR` stayed unasserted—suggesting headroom—though commercial use should keep to nominal limits.

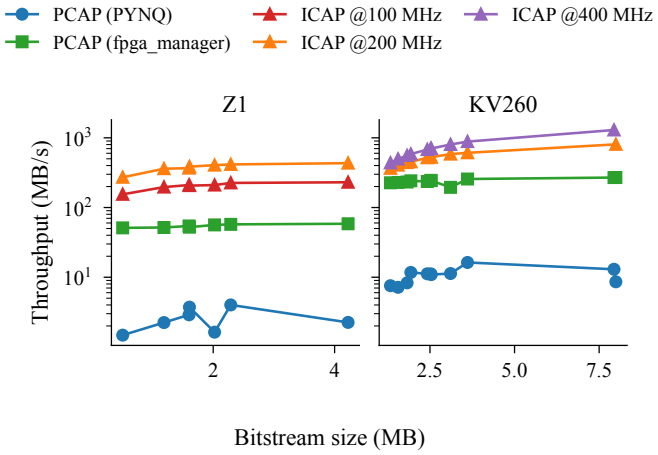


Fig. 7. End-to-end reconfiguration throughput vs. partial-bitstream size (solid: Z1, dashed: KV260). Throughput rises with bitstream size as fixed overheads amortize; the integrated ICAP path is the high-throughput option.

VI. CONCLUSION

This work presented Cocotb-PYNQ-PR, a unified framework that generates the full Vivado DFX flow, a matching swap-aware co-simulation, and a deployment path from a single YAML description. Its central contribution is the co-simulation environment: each reconfigurable module runs as a separate process behind a DPI bridge, so the same Python test exercises an RM swap in simulation exactly as on hardware—turning a ≈20–31 min build-and-deploy round trip into a 69 s first-frame check, with a VPI-free shared-library swap cutting median swap latency 18×. The generated overlay then deploys through either the standard PCAP path or an integrated ICAP backend without changing application code.

REFERENCES

[1] M. Ibrahim, S. Pillement, A. Pinna, and S. Le Nours, "VERSATILE: Very fast partial reconfiguration controller," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 3, Sep. 2025.

[2] Advanced Micro Devices, Inc., "PYNQ: Python productivity for Zynq," Version 3.1.2. [Online]. Available: https://pynq.io, 2024.

[3] AMD/Xilinx, *Zynq-7000 SoC Technical Reference Manual*, AMD/Xilinx, 2023, uG585. [Online]. Available: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM.

[4] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 72:1–72:39, 2018.

[5] Advanced Micro Devices, Inc., *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*, AMD, 2025, version 2025.2. [Online]. Available: https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration.

[6] ——, *Vivado Design Suite Tutorial: Dynamic Function eXchange (UG947)*, AMD, 2026, version 2026.1. [Online]. Available: https://docs.amd.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial.

[7] ——, *AXI HWICAP v3.0 LogiCORE IP Product Guide (PG134)*, AMD, 2025, version 3.0. [Online]. Available: https://docs.amd.com/r/en-US/pg134-axi-hwicap.

[8] R. Bonamy, H.-M. Pham, S. Pillement, and D. Chillet, "UPaRC—Ultra-fast Power-aware Reconfiguration Controller," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 1373–1378.

[9] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, Sep. 2014.

[10] A. R. Bucknall and S. A. Fahmy, "ZyPR: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 3, Jun. 2023.

[11] Y. Xiao, A. Hota, D. Park, and A. DeHon, "HiPR: High-level partial reconfiguration for fast incremental FPGA compilation," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 70–78.

[12] Y. Xiao, D. Park, Z. J. Niu, A. Hota, and A. DeHon, "ExHiPR: Extended high-level partial reconfiguration for fast incremental FPGA compilation," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 2, 2024.

[13] B. Ramhorst, D. Korolija, M. J. Heer, J. Dann, L. Liu, and G. Alonso, "Coyote v2: Raising the level of abstraction for data center FPGAs," in *Proceedings of the 31st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, 2025, pp. 639–654, arXiv:2504.21538.

[14] B. Janßen, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for pynq," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[15] Xilinx, "PYNQ composable pipeline," Version 1.0.0. [Online]. Available: https://github.com/Xilinx/PYNQ_Composable_Pipeline, 2022.

[16] Xilinx, Inc., *Vivado Design Suite: AXI Reference Guide (UG1037)*, Xilinx, 2017, version 4.0. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[17] L. Gong and O. Diessel, "ReSim: A reusable library for RTL simulation of dynamic partial reconfiguration," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. IEEE, 2011, pp. 1–8.

[18] G. Lusby and N. Kapre, "Cocotb-pynq: Co-simulating Python+RTL applications targeting Pynq platforms with cocotb," in *Proceedings of the 35th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2025, pp. 222–226.

[19] S. Herbst, N. Moroze, E. Iglesias, and A. Olofsson, "Switchboard: An open-source framework for modular simulation of large hardware systems," arXiv preprint arXiv:2407.20537, 2024, [Online]. Available: https://arxiv.org/abs/2407.20537.

[20] G. López-Paradís, B. Li, A. Armejach, S. Wallentowitz, M. Moretó, and J. Balkind, "Fast behavioural RTL simulation of 10b transistor SoC designs with Metro-MPI," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[21] IEEE, "IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2017, 2018.

[22] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 1st ed. Morgan Kaufmann, 2012.

[23] Accellera Systems Initiative, *Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual, Version 2.4*, Accellera, 2016.

[24] M. Popoloski, "slang: SystemVerilog compiler and language services," [Online]. Available: https://github.com/MikePopoloski/slang, 2025, IEEE 1800-2017 compliant.

[25] W. Snyder and contributors, "Verilator: Open-source SystemVerilog simulator and lint system," Version 5.036. [Online]. Available: https://www.veripool.org/verilator/, 2025.

[26] Xilinx, Inc., *7 Series FPGAs Configuration User Guide (UG470)*, Xilinx, 2023, version 1.17. [Online]. Available: https://docs.amd.com/v/u/en-US/ug470_7Series_Config.

[27] ——, *UltraScale Architecture Configuration User Guide (UG570)*, Xilinx, 2023, [Online]. Available: https://docs.amd.com/r/en-US/ug570-ultrascale-configuration.