Zoomie: A Software-like Debugging Tool for FPGAs

Tianrui Wei
tianruiwei@eecs.berkeley.edu
UC Berkeley
Berkeley, USA

Kevin Laeufer
laeufer@eecs.berkeley.edu
UC Berkeley
Berkeley, USA

Katie Lim
katielim@cs.washington.edu
University of Washington
Seattle, USA

Jerry Zhao
jerryz123@eecs.berkeley.edu
UC Berkeley
Berkeley, USA

Koushik Sen
ksen@eecs.berkeley.edu
UC Berkeley
Berkeley, USA

Jonathan Balkind
jbalkind@ucsb.edu
UC Santa Barbara
Santa Barbara, USA

Krste Asanović
krste@berkeley.edu
UC Berkeley
Berkeley, USA

Abstract

FPGA prototyping has long been an indispensable technique in pre-silicon verification as well as enabling early-stage software development. FPGAs themselves have also gained popularity as hardware accelerators deployed in datacenters. However, FPGA development brings a plethora of problems. These issues constitute a high barrier towards mass adoption of agile development surrounding FPGA-based projects.

To address these problems, we have built Zoomie for fast incremental compilation, reusing verification infrastructure, and a software-inspired approach towards open-source emulation. We show that Zoomie achieves 18× speedup over the vendor toolchain in incremental compilation time for million-gate designs. At the same time, Zoomie also provides a software-like debugging experience with breakpoints, stepping the design, and forcing values in a running design.

ACM Reference Format:
Tianrui Wei, Kevin Laeufer, Katie Lim, Jerry Zhao, Koushik Sen, Jonathan Balkind, and Krste Asanović. 2024. Zoomie: A Software-like Debugging Tool for FPGAs. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651356

1 Introduction

FPGA-based prototyping refers to compiling hardware designs down to bitstreams run on an FPGA and using the FPGA to test the behavior of the hardware design interfacing with different physical inputs and outputs (IO), such as DDR or Ethernet. Its unique ability to interface with real IOs, coupled with its four orders of magnitude speedup over cycle-accurate software RTL simulation [27], makes it indispensable for hardware verification and pre-silicon software development [26] However, despite its wide adoption in both academia and industry, FPGA prototyping has the grave drawback of lacking a software-like development environment. Compilation times are often on the order of hours if not days, and the vendor-provided debugger [10] can only observe a very limited set of signals and needs to be recompiled in order to observe different signals. Current FPGA prototyping focuses on finding bugs post-mortem, which makes finding out where the problem is (bug localization) and why it is happening much more challenging.

By contrast, the software world enjoys a much richer portfolio of debugging techniques, such as reverse debugging and full instrumentation with comprehensive debugger support [7, 17, 25, 40, 41]. These techniques enable developers to introspect and manipulate their program execution in a flexible manner. We argue that for agile hardware development to fully materialize, we must build debugging tools that offer similar functionalities. Although recent works [11–13, 34, 48, 51, 52, 59, 62] try to address some of these problems, studies [3] show that better debugging infrastructure is still needed for FPGA development. The aforementioned issues still persist if they are not addressed in an organic, coherent manner as a whole.

In this paper, we present Zoomie. Zoomie democratizes FPGA prototyping and debugging by providing a coherent set of components that give users the same abstraction as modern software debuggers for FPGA prototyping. Users can enjoy full visibility of their FPGA during execution, recompilation time in minutes, breakpoints, snapshots, and replaying FPGA execution for arbitrary RTL designs. Zoomie is also the first FPGA debugging platform to exploit specific features of state-of-the-art chiplet-based FPGAs [57], which are commonly used throughout the industry, such as inside Amazon EC2 F1.

We summarize the features of Zoomie as follows:

Blazing Fast Incremental Compilation In Vivado, a single line change can trigger hours of rebuild time, and the speedup for the vendor incremental compilation mode is usually around 10% over the baseline. We propose a new

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651356

paradigm of incremental compilation based on Dynamic Function Exchange (DFX). With Zoomie, users can easily iterate on the design with a turnaround time of minutes rather than hours. We argue that this is the most important part of providing an agile FPGA debugging experience, as the same debugging methods could be performed more than 100× in this context.

Reusing Verification Infrastructure We enable reuse of existing verification infrastructure in FPGA debugging. In particular, we exploit SystemVerilog Assertions, a key component in modern verification infrastructure which can encode the system's expected behavior in linear temporal logic (LTL). They are commonly used throughout the verification lifecycle, from simulation to formal verification.

Software-Like Debugger Lastly, we propose a software-inspired approach towards open-source emulation, implementing many features only available in commercial systems and software debuggers. We achieve this by deeply investigating the Xilinx FPGA architecture, bringing to light formerly opaque details surrounding the FPGA flow. We demonstrate how to use this knowledge to implement a software-like debugger with features like breakpoints, watchpoints, and single stepping. We also are the first work to support modern multi-chiplet FPGAs, with supported FPGA sizes exceeding 10× the prior work [57].

We show how we combine these features into Zoomie[1], an accessible, open-source platform that provides a software-like debugging experience on FPGAs. We target off-the-shelf FPGA boards and demonstrate an 18× speedup in compilation time.

2 Background

2.1 Challenges with FPGA prototyping

FPGA prototyping is a vital part of the hardware verification process. However, its vast advantage in speed comes at great costs. We classify the problems into three categories:

Incompatibility with existing verification infrastructure SystemVerilog [4] is split into synthesizable and unsynthesizable subsets [49]. However, commonly used verification infrastructure like SystemVerilog Assertions (SVA) or Universal Verification Methodology (UVM) [5] are not synthesizable. FPGA tools only support the synthesizable subset, thus making existing verification infrastructure incompatible with FPGA prototyping. As such, bugs are usually discovered post-mortem, with a software crash or return of the wrong result.

Ultra-long compilation time Compilation time for FPGAs can easily reach hours to days. This makes debugging iteration times long and hampers rapid development.

Lack of support for debugging tools Lastly, debugging instruments provided by FPGA vendors like Integrated Logic Analyzers (ILA) are print-style debugging [19]. These can only observe specific signals over time without mutating design state and need users to specify the signals to monitor ahead of compilation, which is further hindered by long compilation times. There is also a very limited subset of signals a user could observe with each compilation, as ILA adds a substantial hardware overhead. Finally, there are no off-the-shelf debugging tools to support breakpoints. With all these drawbacks, fixing bugs in FPGAs is extremely challenging.

[1] https://github.com/zoomie-project

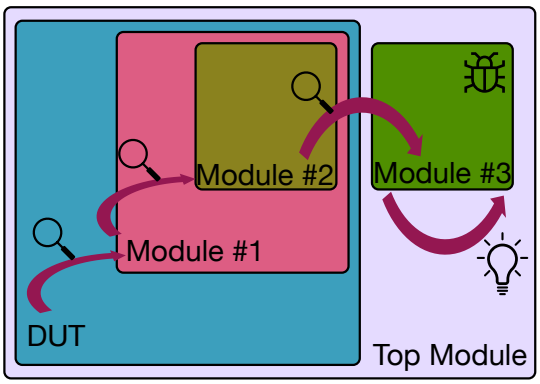

Figure 1. The traditional debugging process for an end user. Repeated recompilation and inspection localizes the bug before iterative bug fixes are tested.

2.2 Motivating Example

Here we use a simple running example based on MAPLE/Cohort [42, 55] and OpenPiton+Ariane/BYOC [14, 60] to demonstrate issues commonly encountered in FPGA prototyping. We find a bug described in Cohort's development logs, which caused the design to return the wrong TLB lookup in the wrong sequence. We highlight the piece of code we omit in pink.

```
assign ack = tlb_sel_r == i  & id == i ;
```

In this case, the users have an existing design that runs on an FPGA, and they would like to debug it with a software stack running on top. However, their only observation is that the software is not working. Following the official Xilinx Documentation, they marked some signals for debugging with an ILA [19]. The user had to iteratively mark signals and recompile to get to the bottom of the issue, where each attempt took around 2 hours. Lastly, to test the bug fix took another 2 hours. This process is depicted in Figure 1 and described further in Section 5.5. We summarize the problem with this process as follows:

First, **bug localization** requires iterative re-compilation. By contrast, there are many ways for users to localize bugs in software [7, 17, 25, 40, 41]. Here, the user could only guess where the bug might lie, then recompile the design iteratively. Despite the SystemVerilog specification's [4] rich verification constructs, most vendor tools do not support

them. This further undermines bug localization, as checks from simulation do not apply in FPGA.

In addition, there's little to no **incremental compilation** in commercial toolchains. Even with small changes, each run still takes almost as long as the initial run. Combined with the need to recompile the design for bug localization, debugging becomes a very slow and manual process.

Lastly, end users suffer from **poor introspection ability**. Unlike software debuggers with rich breakpoints, watchpoints, and an assortment of debugging facilities, the ILA is very limited in its functionality. It can only observe the design over a short window of cycles rather than interactively explore different execution paths.

These three drawbacks are detrimental to providing the kind of agile, productive debugging experience on FPGAs that we work toward. Zoomie provides users with a software-like interface for FPGA debugging. Users can insert custom breakpoints or watchpoints on the fly, which pause the design when they are triggered. From there, users can choose to continue executing the design, or to manipulate the state of the snapshot to test different execution scenarios.

For existing verification infrastructure, Zoomie synthesizes SVAs efficiently and converts them into breakpoints. Any violation of the assertions pauses the design, where the user can then investigate its state. Later, when users would like to change their design, Zoomie offers a way to incrementally compile the design with more than 18× speedup.

Zoomie's abilities, in sum, offer a software-like debugging experience on FPGAs. By contrast, traditional FPGA debugging is much more primitive and inefficient.

3 Zoomie architecture

Zoomie combines three components to enable fast and interactive debugging as illustrated in Figure 2. Pausing, resuming, and inspecting the FPGA state is achieved by the **Debug Controller**, which provides the user with the interface to inspect and manipulate the design's state. It consists of a hardware component inserted into the user design and communicates with software on the host computer to control and debug the design running on FPGA. The **Assertion Synthesis** compiler generates breakpoint triggers from SVAs by turning them into synthesizable state machines. Once a bug is found, our **VTI** (Vendor Tool Incrementalizer) component enables fast incremental synthesis and place and route, drastically reducing the time it takes to go from RTL changes to an updated design executing on the FPGA.

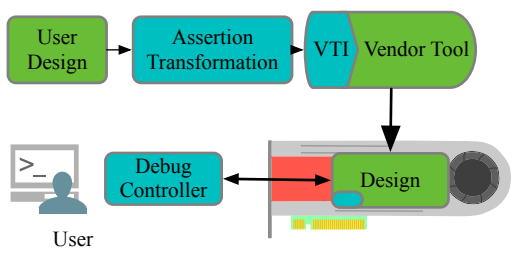

Figure 2. Design overview of Zoomie. Elements in blue denote Zoomie components.

3.1 Pausing the design with the Debug Controller

Zoomie provides the ability to halt the design during execution, similar to software breakpoints. Halting the design can be initiated using an internal trigger signal like a failed assertion. It can also be initiated by the user from the host computer, for example, when they observe unexpected behavior or design hangs. Pausing part of the design for debugging faces two major challenges: (1) the pause needs to be timing precise, meaning that we can stop the design in the exact cycle that a trigger is activated and (2) we must be able to resume execution of a paused design.

Both problems are addressed by the Debug Controller. This RTL component is placed as a wrapper around the module under test (MUT), i.e., the part of the design the user is currently testing. The MUT can be fairly large; we selected a complete processor tile in one of our case studies. This works best (i.e. requires least designer effort) when the MUT communicates via decoupled interfaces. These are regularly used in hardware designs today, providing benefits in scalability, simulation speed, and formal reasoning [20, 22, 37].

Timing-Precise Pausing. The Debug Controller contains a clock gate enabling it to stop the MUT's clock. To ensure a glitch-free and precise pause, Zoomie makes use of FPGA vendor-provided clocking primitives. Once a design is paused, we can inspect and update the state of all its registers and memories. We detail several uses of this functionality in sections 3.2 and 3.3.

Resuming a Paused Design. While merely clock-gating the MUT is enough to stop its execution, it will not allow us to resume execution safely. The modules outside the MUT will continue executing and could end up in an invalid state in which the execution of the complete design becomes impossible. We thus designed a novel **pause buffer** which interposes a (preferably decoupled) interface to enable safe pause and resume.

We illustrate the issue of incorrectly pausing the design with the waveform in Figure 3. The valid and ready signals form a latency-insensitive interface within the same clock domain. However, the clock for the valid signal can be gated, whereas the ready signal is driven by ext_clk, which is never gated. In the diagram, the interface successfully performed a handshake during cycle 2 (t1) and 4 (t2). However, because the clock for valid is gated at cycle 5 (t3), it never gets deasserted. The other module driving the ready signal does not know that the valid signal asserted in cycle 5 is due to clock gating rather than another transaction. This violates the semantics of the original RTL and can cause system hangs.

To combat this issue, Zoomie provides a set of formally verified pause buffers as part of the Debug Controller. These modules ensure the following:

1. If the requester initiates a transaction and is paused, the pause buffer continues to perform that transaction and passes the value to the responder when it is resumed.
2. If either the requester or responder is paused at the cycle of a transaction, the pause buffer restarts the transaction for the paused interface after it resumes.
3. If the requester has no pending transaction, the pause buffer does not incur a one-cycle latency between.

These modules are timing exact and ensure that pausing and resuming the design will not violate the interface protocols. With these, the Debug Controller supports pausing across module boundaries. It also supports different flavors of latency-insensitive interfaces, such as irrevocable interfaces, which dictate that a valid signal must stay high until the ready/ack signal is high. For other forms of timing-dependent interfaces, the designer can consider the Wire Sorts [22] and protocol behaviour to correctly apply the pause buffer.

3.2 State Extraction

Once a design is paused, Zoomie uses the readback functionality provided by the FPGA vendor to retrieve the values of all state elements in the MUT. Zoomie then parses the binary data and matches it up with names of registers and memories in the RTL description of the design. Metadata generated by the vendor toolchain makes this capability possible, but a custom program is required to match the readback with the RTL names and provide output helpful to a user.

3.3 Manipulating Design State

Zoomie also enables the user to manipulate all register and memory values in their design. Similar to a software debugger, this enables a designer to explore different runtime behaviors without the need to restart or recompile a design.

This can be used to deliberately hide known bugs that are difficult to hit in order to preserve emulation progress. Given a bug may take trillions of cycles to detect, if a user fixes the bug, they then have to restart their simulation and potentially wait trillions of cycles again to evaluate their fix. The preferable flow would instead be that the user can resume from a snapshot with the correct values in respective signals and replay the snapshot back on FPGA. If the bug is difficult to trigger, this releases the potential of losing emulation progress and enables extremely long-running benchmarks without interruption.

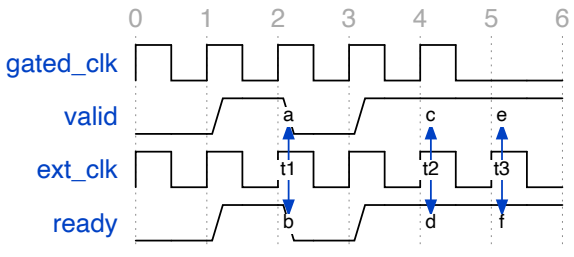

Figure 3. Example of a protocol violation when pausing the design incorrectly.

3.4 Pausing the Design with Internal Triggers

The Debug Controller contains three main sources of internal triggers: (1) value breakpoints that become active when a user-selected signal takes on a certain value, (2) a cycle breakpoint that becomes active after a user-specified number of clock cycles and (3) assertion breakpoints that become active when an SVA in the design fails.

Trigger Composition. All three sources of triggers can be arbitrarily combined through Algorithm 1. State manipulation capabilities introduced in Section 3.3 are used to reconfigure the trigger selection on the fly.

Algorithm 1 Debugger breakpoint algorithm

```
1:  procedure BREAKPOINT(signal_list)  ▷ a list of signals to use as inputs
2:      for sig_i ∈ signal_list do
3:          Debugger ← Andmask_i
4:          Debugger ← Ormask_i
5:          Debugger ← Refval_i
6:          And_i ← (sig_i == Refval_i) ∧ And_maski
7:          Or_i ← (sig_i == Refval_i) ∨ Or_maski
8:      end for
9:      And_stop ← ⋀^i And_i
10:     Or_stop ← ⋁^i Or_i
11:     Stop ← (And_stop ∧ And_sel) ∧ (Or_stop ∧ Or_sel)
12: end procedure
```

In Algorithm 1, each signal with a subscript is associated with a signal in the signal list. These intermediate signals are generated to control the mask generation process, so users can freely specify their triggers.

The overall stop signal is always assigned as follows.

$$Stop = (And_{sel} \wedge (\forall i.(sig_i == Refval_i) \wedge And\_mask_i)) \quad (1)$$

Stepping the Design. Similarly to gdb's 'until' commands, Zoomie's cycle breakpoint executes the design for a programmable number of cycles. This enables the user to do interesting things, such as stepping the design for a few cycles to observe its state or to skip over a specific part of the execution. Users can also dump the design periodically to snapshot different states of execution and replay those snapshots later. This is implemented by a special register inside the Debug Controller that counts the number of cycles the design should execute until pausing.

Assertion Breakpoints. We enable the user to use SVAs, which are commonly used to detect failing designs in simulation and for proving properties of designs through formal verification. The following snippet shows a simple example:

```
ack_valid: assert property
  (@(posedge clk) disable iff (!resetn)
    valid |-> #1 ack);
```

This assertion checks that whenever the `valid` signal is high, the `ack` signal should be high one cycle later. The values of the signals are sampled on the rising edge of the clock, and the assertion is disabled when the `resetn` signal is low.

Zoomie can automatically pause a design when an SVA property is violated. This feature is enabled by our **Assertion Synthesis** compiler which extends Yosys [56] with support for synthesizing SVAs to finite state machines. These are executed on the FPGA alongside the MUT. Assertion breakpoints can be dynamically disabled and combined with other breakpoint types as needed.

3.5 Smart Compilation for Debugging with VTI

Compiling a design for an FPGA often takes a few hours or even days [59]. In comparison, compilation for software development typically takes seconds to minutes, even for very large projects. We observe that existing FPGA toolchains, even with vendor-provided incremental compilation, suffer from long compilation times because compilation runs do not reuse any (or enough) results from prior compilation.

A summary of the compilation processes' differences in software, Vivado, and VTI is shown in Table 1. In software, compilations are mostly performed locally, on a per-file basis. The results are linked together in the end into a single executable. In comparison, Vivado treats designs in a monolithic manner. During the synthesis stage, it always produces a large netlist with optimizations across module boundaries. Subsequently, the netlist is placed and routed in the same monolithic fashion. In software terms, this is like inlining all functions across all the files into the main function and performing optimization on the monolithic program. Toolchains have the opportunity to perform aggressive optimizations across modules, but small changes in RTL could have a big impact on the optimized netlist.

Shown in Figure 4, VTI takes inspiration from the traditional software compilation process and takes a point in the middle by splitting the design into multiple partitions based on designer input. Subsequent compilations are done in parallel within each partition, and the linking happens in the end for all partitions together. Each partition can also be loaded onto the FPGA independently, which can drastically reduce the time it takes to load an updated design on an FPGA.

The user partitioning takes the form of a list of modules. VTI then guides Vivado to perform parallel synthesis on different partitions of RTL independently. Each partition can then be independently placed and routed in its respective physical region by the FPGA toolchain.

Table 1. Comparison of compilation processes

	Compilation unit	Optimization	Linking
Software	function	local	after compilation
Vivado	whole design	global	not required
VTI	partition	partition-local	after routing

Our formula for FPGA resource provisioning balances the trade-off between resource utilization and compilation time. For resource estimation of a single type of resource, we use the following formula, where F_R is the estimated resource usage within each partition, *resource* is the resource usage from the synthesized netlist, c is the over-provision coefficient to trade-off area for timing.

$$F_R = resource * (1 + c)$$

To estimate the overall required area for each partition, we use the following formula, where F_{total} represents the physical resources available within this area. We ensure that the each type of resource available on the partition is always greater than of the corresponding estimated usage.

$$F_{total} \geq \max_{resource} F_R$$

We use this estimate to provision FPGA resources to enable partial static linking of FPGA design fragments.

VTI guides Vivado to place all modules being debugged inside one FPGA chiplet to minimize cross-chiplet communication within debugged modules. Given debugging is typically focused on specific subcomponents of the design, we think it is reasonable to assume a user's region of interest can fit inside a chiplet. Most Alveo UltraScale+ FPGAs feature three or four chiplets, meaning the designer can focus their

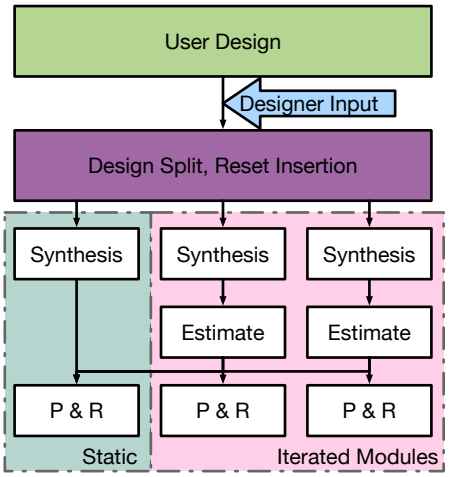

Figure 4. High-level design of our incremental flow

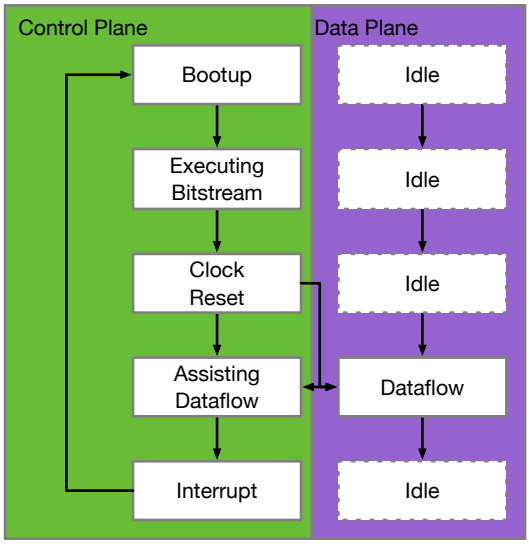

Figure 5. FPGA configuration flow recap. The left portion shows the steps that are run by the microcontroller, and the right shows what is executed in the FPGA fabric.

debugging effort on a reasonably large portion of their entire design.

4 Xilinx Features to Implement Zoomie

In this section, we show how Xilinx FPGAs work under the hood and demonstrate implementing features in Zoomie with these underlying capabilities. Zoomie utilizes these low-level capabilities to achieve rich functionality with little overhead.

4.1 Xilinx Configuration Flow

During configuration, the host machine running Vivado transmits the bitstream to a microcontroller (μc), which interprets it as a program. Broadly speaking, there are two types of instructions: the first type writes to special registers of the μc, and the second type writes to a memory-mapped address, which corresponds to a specific LUT or BRAM in the design. The process of configuration is summarized in Figure 5.

When the entire bitstream has finished execution on the μc, the bitstream programmer (Vivado, generally) then writes to another special register that starts the clock and raises the global set-reset (GSR) signal for all LUTs. After a certain period, the GSR is lowered to finish the reset process, and all logic begins its normal execution.

4.2 Controlling Clock and Reset in FPGA

Clocking is an indispensable part of digital design. Xilinx provides primitives to correctly handle different kinds of clocking topology. Further, clock gating/mux cells provide Zoomie with the ability to pause and unpause areas of the design mapped to the FPGA.

For reset, Xilinx provides two different ways to distribute reset into the entire system. There is a global reset signal that could distribute the reset signal to the entire FPGA, and there is high-speed backbone circuitry to distribute the reset signal faster. In addition, users could use masks that Xilinx provides to further manipulate the distribution of reset in a fine-grained manner.

Both of these mechanisms can be controlled via writes to global registers through the configuration μc.

4.3 Chiplet Connections between FPGAs

FPGA vendors use chiplet technology to expand the capacity and reduce costs of their FPGAs [1, 2, 21]. Chiplet-based FPGAs consist of multiple dies connected by an interposer [46, 57]. Although each die in these FPGAs is almost identical to monolithic chips, our mental model of understanding needs to be adapted to the newer generation of FPGAs. In Xilinx's terminology, each die is called an SLR (Super Logic Region). For multi-SLR FPGAs, there is a primary SLR that commands the other - secondary - SLRs. Xilinx has not documented the mechanism through which the primary SLR communicates with the secondary SLRs. However, understanding this mechanism is essential for reading back state from the FPGA. In a monolithic Xilinx FPGA, the state can be read by writing a special "readback" command to a command register, which specifies where to read from [6]. However, the means of specifying which SLR to read from was an open question.

In Bitfiltrator [28], the authors hypothesized that different SLRs are distinguished from one another by having separate `IDCODE`s, because instruction sequences inside the bitstream do contain writes of different `IDCODE`s for each chiplet. However, this is not the case. To show that, we instantiate 3 registers that initialize to different constants at reset, constrain each one to reside on a different chiplet using Vivado Tcl commands, and turn off optimizations. Now, when we attempt to read the register values, we follow the readback method used for monolithic FPGAs, except we inject the same `IDCODE` values that appeared in the writes in the original bitstream. In practice, we observed that no matter what `IDCODE` value is written to the register, the value we get from readback is always the constant of the register constrained to the primary SLR. Thus the writes into the `IDCODE` field yield no actual effect in this situation.

4.4 Hidden Instructions to Select SLRs

Since the `IDCODE` method hypothesized by the authors of Bitfiltrator clearly does not work, we started looking for undocumented bitfields and register names in the bitstream specification. In particular, we paid close attention to repetitions of **0xFFFFFFFF** and **0xAA995566**. The former is a dummy padding that is used to compensate for the wait time to avoid the μc being busy. The latter is used to synchronize the start of a command sequence.

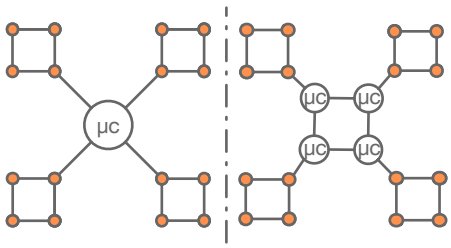

Figure 6. Potential views of a Xilinx multi-chiplet FPGA. **Left**: Logical view; **Right**: Our understanding.

During this process, we found writes to an undocumented "BOUT" register. However, all the writes contained empty data. For a U200 with three SLRs, the bitstream can be divided into three roughly equivalent chunks. This write does not occur when the bitstream is configuring the primary SLR. However, it appears once before configuring the first secondary SLR and appears twice before configuring the second secondary SLR. We extrapolate that empty writes to this special register, followed by appropriate padding, act as a switch to direct the JTAG operations to different SLRs. Once directed, all the JTAG operations only work for that particular SLR until these writes appear again.

In addition, because all the writes to other configuration registers need to be duplicated for each SLR, we conclude that each SLR is just a complete FPGA on a chiplet, the JTAG bus between microcontrollers has a ring topology, and each operation only affects a single SLR.

4.5 Hypothesis Validation

With the hypothesis that this special write, instead of device ID (`IDCODE`), controls the operation within the bitstream, we proceed with the following experiments to show that our hypothesis is correct.

Reading Back from Different SLRs. Using the same pattern and the testcase shown above, we are able to successfully retrieve logic values stored in registers of each of the SLRs.

Verifying Repetition Pattern. To verify that the SLRs are indeed connected via a ring topology, we also test our technique on an Alveo U250, which consists of 4 SLRs instead of 3. We note that the final SLR can be reached by pulsing the "BOUT" write 3 times. This validates that the pattern of repetition is simply incremented by one.

Mutating Device ID in Bitstream. We change the device ID of the secondary SLRs the bitstream writes to and observe the readback result (following Bitfiltrator). We notice that although this device ID is required by monolithic FPGAs for verification purposes, the ID value of secondary SLRs does not impact the readback result, as only writing to "BOUT" selects the SLR targeted by control commands.

4.6 Controlling FPGA Chiplets

With this in mind, we conclude that JTAG could only control one chiplet at a time. However, the chiplets are also connected through many wires on the interposer. We formulate the following two ways of controlling logic on different chiplets as a foundation for building Zoomie. Controlling SLRs synchronously means multiple SLRs will need to be accessed on the same cycle in order to retrieve the contents in the FPGA, while asynchronously means the logic that needs pausing will be restricted within a single SLR.

Controlling Different SLRs Locally/Asynchronously. In the case where we would like to inject values or pause a part of the logic, we would try to limit that region to be within a single physical SLR in order to minimize logic delay across different dies. In this case, different SLRs would be controlled by their local μc only, and we could select the appropriate μc through JTAG.

Controlling Different SLRs Globally/Synchronously. When we have big designs or logic blocks that will span multiple SLRs, we would control the stepping of the entire design through the primary SLR only. However, scanning out values in the design is still the job of the SLR-local controller.

4.7 Composing Xilinx features into Zoomie features

Given these primitives, we show below how we compose them into Zoomie's features.

Design Pausing. Zoomie leverages the glitch-less clock mux in Xilinx FPGAs as well as the Pause Buffer from Section 3.1. When Zoomie tries to pause the design in the next cycle, it will lower the clock enable signal going into the design and Pause Buffer, and resume when the pause finishes.

Design Stepping. To single-step the design for a prespecified number of cycles, Zoomie uses a 64-bit hardware counter to keep track of remaining cycles to execute as well as if there is any single-step ahead. Before resuming the design, Zoomie uses JTAG to set the counter value to be the number of cycles to step over. The design is paused again when the number reaches 0.

Design Readback. Instead of scanning out all the SLRs naïvely, Zoomie analyzes the MUT to determine whether it spreads over multiple SLRs. When the MUT is contained within a single SLR, Zoomie would simply switch control to that SLR utilizing the process described above to scan out the relevant logic region. When the MUT is split across multiple SLRs, Zoomie will scan each SLR only once. For each SLR, it only scans the regions that contain the MUT, as indicated by Vivado.

Combining Readback with Partial Reconfiguration. During partial reconfiguration, the FPGA uses the mask register to set restrict GSR to the dynamic region. However, it

does not restore the mask register by default. In order to complete the readback function correctly, Zoomie always tries to clear the mask bit prior to issuing readback commands.

Resuming from Snapshot Data. To resume from a snapshot, Zoomie first partially programs the FPGA with the snapshot data while leaving untouched regions intact. This is equivalent to only writing to some of the tiles with new data, but not all tiles. After that, Zoomie ungates the clock to unpause the design. However, before the MUT resumes execution, the pause buffers first try to complete all outstanding transactions during pausing; the design will then work normally.

5 Evaluation

We evaluate our VTI algorithm for incremental compilation detailed in Section 3.5, our optimized state readback implementation, and our Assertion Synthesis compiler. We also present two case studies exploring the debugging speedups enabled by Zoomie.

5.1 Evaluation Environment

We conducted all of our compilations on a computer with an Intel Xeon Gold 6354 CPU and 512GB of main memory running Ubuntu 20.04 LTS. A separate computer with an Intel i9-10850K CPU running Debian Linux with kernel version 5.15.0 and 64 GB of memory hosts the FPGA, a Xilinx Alveo U200 FPGA card. We used Xilinx Vivado 2022.2.

5.2 VTI Incremental Compilation for a Manycore RISC-V SoC

Our VTI algorithm is designed to minimize the time it takes between updating the RTL source code of a design and running the updated design on an FPGA at the cost of some area inefficiency. To properly evaluate the magnitude of this speedup, we chose an open-source manycore RISC-V SoC [31] made of award-winning SERV cores [32] and configured the SoC to include a total of **5400** RISC-V cores.

We first look at the area usage. 5400 cores use most of our Alveo U200's resources. Table 2 shows the exact resource usage. This example demonstrates that VTI works even for designs that fill up almost the whole FPGA, despite the fact that VTI relies on reserving extra physical resources for design partitions to allow for fast incremental compilation.

Table 2. Resource usage of an SoC with **5400** RISC-V cores [31] on an Alveo U200 FPGA card.

	Utilization	Percentage
LUT	1103572	95.32
LUTRAM	54128	8.96
FF	12894858	53.42
BRAM	2120	98.19

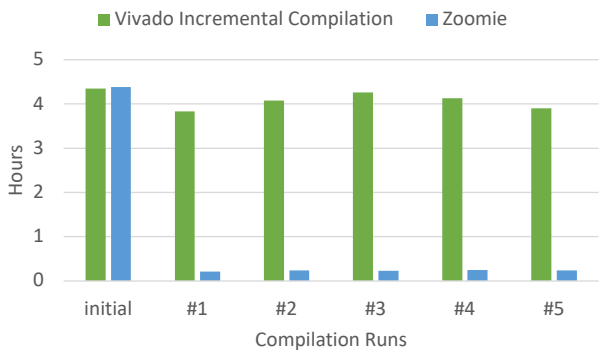

Figure 7. Comparison in Compilation Speed between Vivado Incremental and Zoomie

We chose one of the 5400 RISC-V cores in our SoC as our MUT. We then ran synthesis and place-and-route (from here on referred to as *compile*) for the complete design, once with the default Xilinx Vivado incremental flow and once with our VTI algorithm. The results of this are shown in Figure 7. We observe that although VTI requires additional steps when compiling the design from scratch, this overhead is negligible.

To quantify how long the process of modification to runtime debugging takes, we introduced minor changes to our RISC-V core to expose signals for debugging and measured the time it took to recompile the design. For the Vivado version, we also added ILA probes, which are not needed when using the Debug Controller provided by Zoomie.

Compared to the initial run, incremental compilation with VTI consistently provides around 18× speedup, reducing the time it takes by around 95%. This is because our approach enables Vivado to only recompile a small portion of the design separate from the entire SoC. At the same time, we reuse the results from the initial compilation run for all other parts of the design.

Vivado's incremental mode, on the other hand, shows little gain. While Zoomie enables the user to declare in advance which module they want to recompile later, Vivado has no way to utilize this information. We thus hypothesize that when Vivado is unable restrict the changes to a single FPGA tile, it will try to place a much larger part of the design in a much larger area, thus resulting in a larger region of recompilation. Vivado's incremental mode also often struggles to provide speedups for larger designs with tighter timing. This hypothesis is also supported by SMatch [44], which demonstrates that Vivado can perform placement and routing very quickly if changes are restricted to a single tile.

Resource Usage Tradeoffs. The design successfully met timing closure at the design's original, default frequency of 50MHz, with Zoomie's additions included, thus there was no timing impact. We then tried to increase the design's

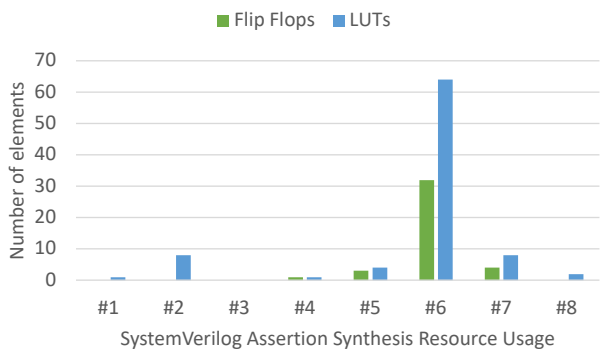

Figure 8. FPGA Resource Usage for Synthesizing SystemVerilog Assertions

frequency to 100MHz. This version failed to meet timing through Vivado's compilation. However, through further inspection, we found that none of the top 10 timing paths were in Zoomie-introduced code. With reference to the overprovisioning coefficient introduced in section 3.5, we used the default area overhead of 30%. However, the design also successfully reached timing closure at both 20% and 15% overheads.

5.3 SLR-Aware Readback Speed Comparison

In this section, we evaluate the speedup of our SLR-aware state readback mechanism detailed in Section 4.7. We use the same 5400 RISC-V core SoC design discussed in the previous section and measure the time it takes to retrieve the state from each of the three SLRs on our Alveo U200 FPGA card with and without our optimization. We repeat each measurement five times and show the average time in Table 3. Our optimization provides an average speedup of around 80×, which enables inspecting the design state at interactive speeds. The fact that reading from SLR 1, which controls the other two SLRs on our FPGA, takes slightly less time also confirms our model of how the chiplets are connected introduced in Section 4.7.

5.4 Assertion Synthesis Area Overhead

Our Assertion Synthesis compiler enables designers to reuse their SVAs for on-FPGA debugging. We picked Ariane [61], an open-source, industrial-strength 6-stage RISC-V CPU with extensive verification support, to evaluate which assertions we can successfully compile into synthesizable circuits and the resource utilization of these circuits.

We randomly select eight SVAs across different hardware modules found in Ariane. Zoomie successfully synthesized seven of the eight assertions we picked. We cannot synthesize assertion #3 because it contains SystemVerilog operator `$isunknown`, which only makes sense in a four-state software simulation since it checks for the presence of an X value, which is unsynthesizable for FPGA.

Figure 8 shows the resource usage for the seven assertions we synthesized. In total, all assertions combined utilize 40 flip-flops and 88 LUTS – a negligible amount compared to the 5k flip-flops and 42k LUTS taken up by a single Ariane core. We thus find that Zoomie can convert most common SystemVerilog Assertions into hardware monitors with very little overhead.

Table 3. Comparison in Readback Time between Unoptimized Version and Zoomie measured in seconds.

	SLR 0	SLR 1	SLR 2
Zoomie	0.397s	0.384s	0.392s
Unoptimized Zoomie	33.594s	33.560s	33.593s

5.5 Case Study #1: Debugging Heterogenous RISC-V SoC with Multi-million Gates

In our first case study, we demonstrate Zoomie debugging a **multi-million gate** heterogeneous Cohort RISC-V SoC[55]. We first recount how we used traditional debugging with ILAs and the FPGA vendor tools to fix a natural bug they encountered during development. We then discuss how Zoomie would have dramatically sped up this process had it been available at the time this bug was discovered.

During FPGA prototyping, we found that one of our accelerators would hang in the middle of execution. In particular, for certain inputs, it could only return part of the result before hanging indefinitely. We thus went about debugging this issue using only traditional FPGA debugging tools as follows:

1. **Recompile** the design with two ILAs, one for the datapath of the accelerator and one for the load-store unit of the accelerator.
2. Observe that in the design, the datapath is computing the correct result, but the load store unit stops providing inputs to the datapath.
3. **Recompile** the design with two ILAs, one for the load-store unit and one for the main system bus *to localize the position of the bug between the load-store unit and the system bus.*
4. Observe that in the design, the load store unit stops issuing requests to the system bus while the system bus successfully responds to all requests made by the load store unit.
5. **Recompile** the design with two ILAs, one to the memory management unit, and one to the load store queues *to localize the position of the bug between memory management unit and load store unit.*
6. Observe that the memory management unit stops responding to requests made by the load store unit while

the load store unit successfully issues requests to the memory management unit.
7. **Recompile** the design with a big ILA that probes all major control functions of the memory management unit to **debug the issue within the memory management unit**.
8. Observe that the ready-valid interface was not properly implemented in the memory management unit, thus finding the bug.
9. Fix the bug, and recompile the design.
10. The design now works as expected!

We see that the problem here is the fact that ILAs can only observe a minimal subset of signals and costly re-compilation is necessary every time a new signal is selected for inspection. Zoomie, on the other hand, allows the user to inspect the values of all signals in the design once it is paused. This enabled us to find the same bug using Zoomie in **less than 20 minutes**, while we originally spent **more than 2 hours** to debug it with traditional FPGA prototyping tools.

5.6 Case Study #2: Debugging Software and Hardware Co-Design Issues

We used Zoomie to debug an issue with the Ariane RISC-V core [61] introduced in Section 5.4. In this case, the core was hanging, but it was unclear whether this was caused by the software running on it or an RTL bug. RISC-V has a flexible mechanism that allows multiple nested exceptions to occur. However, more deeply nested (such as 5-level nested) exceptions are difficult to distinguish from single-level or two-level exceptions. This can confuse users if they encounter bugs around exception handlers, because the CPU can show errors unrelated to the original issue.

In this example, we demonstrate how we use Zoomie to mitigate this issue and successfully distinguish software and hardware errors. On the hardware side, we use Zoomie to insert a breakpoint on condition `mcause[63] == 0 && MIE == 0 && MPIE == 0`. When this particular condition occurs, the core will have undergone two levels of exceptions and will respond to another exception. On the software side, we intentionally set the base address for the exception handler to an invalid address range. This will cause the CPU core to loop indefinitely without any meaningful error information.

During debugging, we can observe immediately that the flip-flop carrying the current PC value is equal to `mepc`, with the exception flag set to high. This indicates that we are in an infinite loop where the CPU keeps having exceptions on the current address. Combined with the fact that we are in the nested exception, we can conclude that this is indeed legal behavior in hardware due to software misconfiguration. This example demonstrates how Zoomie enables users to distinguish between hardware and software errors efficiently without recompiling the design to insert ILAs.

5.7 Case Study #3: Debugging With a High-Speed Network Interface

We also demonstrate Zoomie's ability to aid in debugging high-speed network processing by integrating with a 100Gbps FPGA-based hardware network stack called Beehive [36]. We successfully integrated Zoomie into Beehive without introducing timing violations with respect to the design's 250MHz clock. As such, we are able to insert breakpoints into AXI transactions and gain full visibility of the entire stack.

Networking bugs are tricky because they often manifest some time after the error as opposed to right when the error occurs, and the erroneous behavior can surface in regions of code that are not the actual root cause. There are two traditional approaches to debugging problems like this on FPGAs, but they are both ill-suited to network debugging: ILA and record and replay of packets. Xilinx's ILA is hard to use because it requires recompilation every time we want to inspect new signals, making iteration times long, and in practice, having a large number of signals with probes will require running the design at lower frequencies. In order to use record and replay for certain networking bugs, we need to capture packet traces with accurate timing information. When replaying in software simulation, this can mean that we have to simulate packets arriving over several seconds in real-time. Simulating this length of time takes on the scale of hours versus the seconds of FPGA prototype runtime. Zoomie's ability to allow in-situ readback of a large number of signals without hurting timing fills a unique niche that is especially friendly to this use case.

6 Limitations

Zoomie has the following limitations, some of which are fundamental to FPGA architecture.

6.1 Precise Stepping over Multiple Asynchronous Clock Domains

Precise stepping over multiple asynchronous clock domains is only possible under certain conditions. It requires that each clock domain's clock gating signal does not violate hold/setup timing in the other clock domain. It is generally only possible when clocks are phase-aligned, and frequencies are multiples of each other.

6.2 Precise Stepping with High-Speed Interfaces

Zoomie does not support cycle exact stepping at high-speed interfaces, e.g., with Ethernet PHY interfaces. Fundamentally, this is because the high-speed interface (GTX/GTY) signals provided by Xilinx do not support clock gating. However, users can extend their physical layer protocol implementations to achieve precise stepping, such as using the err signal in XGMII (the protocol used for Ethernet MAC-PHY communication) to achieve cycle exact stepping at the high-speed interfaces. As a specific end-to-end example, our use

Table 4. SystemVerilog Assertion Support in Zoomie

Feature	Example	Support
Immediate	assert (A == B);	full
System Functions	$past(signal, 2)	full
Clocking	@(posedge clk)	single clock
Implication	a |->b	full
Fixed Delay	a ##2 b	full
Delay Range	a ##[1:2] b	finite
Repetition	(a ## b)[*2]	only consecutive
Sequence Operator	a and b	finite a and b
Local Variable		unsupported
Asynchronous Reset		unsupported
First Match		unsupported

of Zoomie with a hardware network protocol implementation is beyond the high-speed interface but still at 250 MHz, though there is a queue that runs in the same clock domain as the MAC-PHY. This queue is used to drop full Ethernet frames if the consuming hardware is not ready to process them, which is necessary for the correct function of the design regardless of Zoomie. Zoomie can debug any portion of the design processing traffic after this queue. Pausing or stepping manually here will give you debugging with correct performance relative to the pausing or stepping frequency, but not truly transparent debugging. Note that debugging networked software socket applications with GDB runs into the same challenges; if an application is paused under GDB, the network stack may drop packets, and the performance or code paths exercised under these conditions may be different than when not pausing.

6.3 Limitations on SystemVerilog Assertion Synthesis

We do not support the full grammar of SVAs for synthesis. However, we do support most of the common assertions that are prevalent in open-source projects such as CVA6 [61]. Table 4 lists the common types of operators that Zoomie supports.

7 Related Work

7.1 FPGA Prototyping and FPGA Accelerated Simulation

FPGA accelerated simulation has recently become popular, with Ramp [33, 50, 51, 54] and Firesim [27] as examples. Rather than mapping logic elements directly into corresponding FPGA primitives, they instead map the logic elements into a timing accurate model and map the model onto an FPGA. This enables decoupling of the design's timing from the FPGA's [38] and thus cycle accurate performance modelling at high speed. However, these benefits come with a lower simulation speed, as simulating one cycle of the design requires multiple cycles of the FPGA. In addition, the FPGA must communicate with the host machine, which controls the simulation. This overhead is especially high for debugging, as the user needs to frequently pause the simulation to inspect the design's state.

7.2 Readback-based Debugging

Readback-based debugging is popular due to its minimal overhead and wide availability across vendors. With its ability to read out almost the entire FPGA configuration, it is also widely used in safety-critical applications, fault-injection, and detection. Prior research [9, 11, 29, 35, 48] suffers from the same problem: inability to debug large designs. In these works, several simplifying assumptions are made such as a single clock, a synchronous design, a monolithic FPGA, and/or the use of a single FPGA chiplet. These assumptions are reasonable for small designs, but quickly break down at scale. Notably, our evaluated designs are on the order of 100× larger and much more complex, with multiple clocks domains spread across multiple dies. In addition, previous work cannot mutate hardware state on the fly, which we consider to be critical for productive debugging.

7.3 Incremental Compilation for FPGA

Incremental compilation is a technique that enables the user to quickly recompile only the parts of the design that have changed. LiveHD [43, 44, 53] uses a graph-based design representation to quickly identify modifications and only resynthesize the necessary parts. It also proposes the novel technique of utilizing the fixed resource structure of LUTs on FPGAs to minimize the impact synthesis has on global results. We consider this to be complementary to our work, as it only addresses the synthesis phase of compilation. From their speedup breakdown, we could see LiveHD sped up the synthesis phase greatly over the commercial tool (by up to 20×), but end-to-end bitstream generation time is roughly the same, as over 80% of the time is still spent in place and route. By contrast, by leveraging insights into the FPGA's internal workings and designer aid, we achieve a uniform speedup across all phases of the compilation.

7.4 Just In Time Compilation for FPGAs

JIT based compilation is a technique that enables the user to modify the design on the fly and observe the result of the modification immediately. Like incremental compilation, JIT compilation is widely used in software development, but unlike incremental compilation, it is not widely used in FPGA development. Although this technique has been well demonstrated [16, 47], it has not been widely adopted in mainstream FPGA development as yet. These works decouple the logical clock for the design from the physical clock of the FPGA. Although some performance penalty is incurred, it allows the user to quickly iterate and supports some of the non-synthesizable features of Verilog, which are critical for debugging and not supported by most FPGA

accelerated simulation tools. Both of these works suffer from the fact efficient JIT compilation requires regular (at a user-configurable frequency) communication between the design on the FPGA and the host machine. Although this could be partly alleviated by using a high-speed interface such as PCIe and optimizing the communication algorithm, it still incurs an overhead compared to Zoomie.

7.5 Synthesizing SystemVerilog Assertions

In academia [8, 15, 18, 23, 45], the synthesis of Linear Temporal Logic (LTL) into Finite State Machines (FSM) is well-explored, yet there is an absence of readily available compilers that integrate with SVAs. Moreover, SVA synthesis demands consideration of specific semantics, including clock and disable signals, distinguishing it from conventional LTL synthesis.

7.6 Comparison to State of the Art Incremental Compilation Techniques

We classify incremental compilation techniques for FPGAs into three broad categories and compare Zoomie with existing works in each category.

Accelerator-Specific Techniques. Works like PLD and HiPR [58, 59] partition an FPGA into coarse-grained regions called pages which communicate through a common interconnect. The design is then mapped into smaller partitions that could fit within a page. During the mapping, the cycle-exact timing relationship between different pages is lost. Instead, this decomposition leverages latency-insensitive interfaces. This is best suited for accelerator designs where the system is made up of identical smaller modules that are replicated. In addition, the design needs to be mapped from a higher level specification, where transformations could easily decompose and recompose modules into equivalent yet different representations. By contrast, RTL represents designs at a much lower level. Gathering designer intent from RTL-based designs is much more challenging, making these transformations prohibitive. These techniques cannot be used for incrementally compiling general hardware modules, which our approach aims to serve. This is because they require injecting delays between latency-insensitive interfaces to work. However, in generic RTL designs, different modules could have latency-sensitive dependencies which delays would break. Zoomie can handle interfaces beyond latency-insensitive ones, with the guidance of the designer who is familiar with the particular interface protocol.

Techniques for combinatorial logic only. SMatch [43, 44, 53] achieves incremental compilation by checking if changes in the design are contained within the smallest unit (slice) on an FPGA. It then tries to directly manipulate the wiring of the LUTs instead of performing place and route to speed up the recompilation process. Its fast speed comes from directly manipulating the FPGA backend, but only small, local, combinatorial changes are allowed. Logic mapped across slices could not be handled, as well as changes larger than a slice (of 4 LUTs). Zoomie supports arbitrary (non-combinatorial/local) logic changes within the dynamic region and increasing/decreasing resource usage. Where SMatch is applicable, it would be faster than Zoomie as it avoids place and route altogether. This technique could be integrated into Zoomie for even faster compilation in the cases where changes are small enough to work.

Virtually-timed designs on FPGA. For works like Cascade and Manticore [24, 47], the DUT is not mapped directly onto an FPGA. Instead, the simulated behavior of the DUT is mapped onto a hardware engine running on the FPGA, and the simulation happens on top of this layer. This approach requires multiple cycles (though potentially as few as 2 for Cascade) to simulate a single cycle of the DUT. As a result, the DUT generally cannot directly use the high speed interfaces of the FPGA, and the simulated design has a lower effective frequency. On the other hand, although our approach introduces a small overhead in resource usage, users can still exploit native FPGA features like high-speed interfaces without lowering simulation speed.

7.7 Comparison to Debug Governor and DESSERT

Debug Governor [39] proposed a mechanism of intercepting latency-insensitive interfaces to pause, log, and inject data into these interfaces. First and foremost, Debug Governor only pauses the latency-insensitive interface without pausing the MUT, which would not preserve the precise state of design like Zoomie. While it shares some functionality with our Pause Buffer in Section 3.1, it cannot provide visibility outside the high-level protocol interface. By contrast, Zoomie provides full visibility of the DUT by default. Zoomie could also manipulate any point in the MUT instead of being limited to modules' protocol interfaces. Debug Governor still suffers from the need for recompilation when trying to observe new signals and a lack of software-debugger-like functionalities.

DESSERT [30] proposes leveraging partial state snapshots on FPGA to recreate CPU microarchitectural state in simulation. Zoomie imposes fewer restrictions than DESSERT and makes fewer assumptions about the design than DESSERT. For starters, DESSERT is CPU microarchitecture specific. It relies on scanning a selected subset of signals at runtime through a custom scan chain and replaying those signal values in an RTL simulation environment with a hand-written Verilog model for playing back the design. Compared to Zoomie, it requires much more user intervention. Zoomie provides a scalable alternative approach without imposing the overhead of a custom scan chain and would make it easier and more efficient to implement a DESSERT-like technique. For assertion synthesis, DESSERT only supports synthesizing combinatorial assertions, which means it can only reason

about circuit behavior within a single cycle. By contrast, Zoomie supports a more generic selection of concurrent assertions that can reason over multiple cycles and overlapping signals. These features are required for most assertions used in industry. DESSERT only supports Chisel-based designs (which accounts for a minor fraction of designs), while Zoomie is HDL agnostic. In terms of overhead, DESSERT introduces up to 85% logic overhead to perform partial value scan-out, whereas Zoomie requires very low logic overhead for scanning out signals from FPGA. Lastly, Zoomie could achieve the printing of arbitrary signals at run time by single stepping without recompiling the design, while adding new prints in DESSERT requires recompiling the whole design. These fundamental limitations make debugging in DESSERT much more difficult and time-consuming.

8 Conclusion

In this paper, we present Zoomie, a software-like debugging tool for FPGAs. We also provide an in-depth understanding of Xilinx multi-chiplet FPGAs, and show how we could leverage this understanding to provide a solution that has high visibility, low overhead, and scales to huge designs. We evaluate our approach on multiple designs with sizes more than 100× greater than prior evaluations and detail our comparison. With Zoomie, users can now debug their designs with a much higher productivity and debug much larger designs than before.

9 Acknowledgements

We thank all the reviewers and our shepherd, Eric Schkufza, for their very valuable feedback. This work was partially funded by SLICE Lab industrial sponsors and affiliates Amazon, AMD, Apple, Google, Intel, and Qualcomm. This work was also supported by NSF grant CNS-210458, NSF grant CCF-1900968, VMware, Cisco Systems, NSF GRFP. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. We also thank Stephen Neuendorffer, Kehui Liu, Shangyin Tan, Chris Batten and Sagar Karandikar for their feedback and comments.

References

[1] Intel Agilex® 9 FPGA Direct RF-Series Whitepaper. https://www.intel.com/content/www/us/en/products/docs/programmable/direct-rf-series-fpga-white-paper.html.

[2] Intel® Stratix® 10 FPGAs Overview - High Performance Intel® FPGA. https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html.

[3] 'My FPGA debug and verification flow should be improved…' | Exostiv Labs, September 2015.

[4] IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. page 1315, 2018.

[5] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pages 1–458, September 2020.

[6] UltraScale Architecture Configuration User Guide, 2023.

[7] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, October 2009. Association for Computing Machinery.

[8] Omar Amin, Youssef Ramzy, Omar Ibrahem, Ahmed Fouad, Khaled Mohamed, and Mohamed Abdelsalam. System Verilog Assertions Synthesis Based Compiler. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 65–70, December 2016.

[9] Hari Angepat, Gage Eads, Christopher Craik, and Derek Chiou. NIFD: Non-intrusive FPGA Debugger – Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping. In *2010 International Conference on Field Programmable Logic and Applications*, pages 356–359, August 2010.

[10] Khalil Arshak, Essa Jafer, and Christian Ibala. Testing FPGA based digital system using XILINX ChipScope logic analyzer. In *2006 29th International Spring Seminar on Electronics Technology*, pages 355–360, May 2006.

[11] Sameh Attia and Vaughn Betz. StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, pages 175–185, New York, NY, USA, February 2020. Association for Computing Machinery.

[12] Sameh Attia and Vaughn Betz. StateLink: FPGA System Debugging via Flexible Simulation/Hardware Integration. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10, December 2021.

[13] Sameh Attia and Vaughn Betz. Stop and Look: A Novel Checkpointing and Debugging Flow for FPGAs. *IEEE Transactions on Computers*, 71(10):2513–2526, October 2022.

[14] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 699–714, Lausanne Switzerland, March 2020. ACM.

[15] Suguman Bansal, Giuseppe De Giacomo, Antonio Di Stasio, Yong Li, Moshe Y. Vardi, and Shufang Zhu. Compositional Safety LTL Synthesis. In *Verified Software. Theories, Tools and Experiments.: 14th International Conference, VSTTE 2022, Trento, Italy, October 17–18, 2022, Revised Selected Papers*, pages 1–19, Berlin, Heidelberg, February 2023. Springer-Verlag.

[16] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs. In Laurie Hendren, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 178–192, Berlin, Heidelberg, 2008. Springer.

[17] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 473–484, New York, NY, USA, October 2013. Association for Computing Machinery.

[18] Alberto Camacho, Jorge Baier, Christian Muise, and Sheila McIlraith. Finite LTL Synthesis as Planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 28:29–38, June 2018.

[19] Kevin Camera and Robert W. Brodersen. An integrated debugging environment for FPGA computing platforms. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, page 260, New York, NY, USA, February 2008. Association for Computing Machinery.

[20] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept./2001.

[21] Raghunandan Chaware, Ganesh Hariharan, Jeff Lin, Inderjit Singh, Glenn O'Rourke, Kenny Ng, S. Y. Pai, Chien-Chen. Li, Zill Huang, and S. K. Cheng. Assembly challenges in developing 3D IC package with ultra high yield and high reliability. In *2015 IEEE 65th Electronic Components and Technology Conference (ECTC)*, pages 1447–1451, San Diego, CA, May 2015. IEEE.

[22] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. Wire sorts: A language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 175–189, New York, NY, USA, 2021. Association for Computing Machinery.

[23] S. Das, R. Mohanty, P. Dasgupta, and P.P. Chakrabarti. Synthesis of System Verilog Assertions. In *Proceedings of the Design Automation &Amp; Test in Europe Conference*, pages 1–6, Munich, Germany, 2006. IEEE.

[24] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism, May 2023.

[25] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, September 2012.

[26] Harry Foster. 2022 Wilson Research Group IC/ASIC functional verification trends. 2022.

[27] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 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)*, pages 29–42, Los Angeles, CA, June 2018. IEEE.

[28] Sahand Kashani, Mahyar Emami, and James R. Larus. Bitfiltrator: A general approach for reverse-engineering Xilinx bitstream formats. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 01–08, August 2022.

[29] Ashfaquzzaman Khan, Richard Neil Pittman, and Alessandro Forin. gNOSIS: A Board-Level Debugging and Verification Tool. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 43–48, December 2010.

[30] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 76–764, August 2018.

[31] Olof Kindgren. CoreScore, June 2023.

[32] Olof Kindgren. Five years of SERVing. https://riscv.org/blog/2023/12/five-years-of-serving/, 2023.

[33] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. RAMP Blue: A Message-Passing Manycore System in FPGAs. In *2007 International Conference on Field Programmable Logic and Applications*, pages 54–61, Amsterdam, Netherlands, August 2007. IEEE.

[34] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 818–831, New York, NY, USA, April 2021. Association for Computing Machinery.

[35] Changgong Li, Alexander Schwarz, and Christian Hochberger. A readback based general debugging framework for soft-core processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 568–575, October 2016.

[36] Katie Lim, Matthew Giordano, Theano Stavrinos, Baris Kasikci, and Tom Anderson. Beehive: A flexible network stack for direct-attached accelerators, 2024.

[37] Guillem López-Paradís, Brian Li, Adriá Armejach, Stefan Wallentowitz, Miquel Moretó, and Jonathan Balkind. Fast behavioural RTL simulation of 10B transistor SoC designs with metro-mpi. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[38] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanovic. Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Westminster, CO, USA, November 2019. IEEE.

[39] Marco Antonio Merlini, Isamu Poy, and Paul Chow. Interactive Debugging at IP Block Interfaces in FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 138–144, Virtual Event USA, February 2021. ACM.

[40] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Ef?ciently. *ACM SIGARCH Computer Architecture News*, 36(3):289–300, June 2008.

[41] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record And Replay For Deployability: Extended Technical Report, May 2017.

[42] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. Tiny but mighty: Designing and realizing scalable latency tolerance for many-core SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 817–830, New York, NY, USA, June 2022. Association for Computing Machinery.

[43] Rafael Trapani Possignolo and Jose Renau. LiveSynth: Towards an interactive synthesis flow. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[44] Rafael Trapani Possignolo and Jose Renau. SMatch: Structural Matching for Fast Resynthesis in FPGAs. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 1–6, New York, NY, USA, June 2019. Association for Computing Machinery.

[45] D.L. Rosenband and Arvind. Hardware synthesis from guarded atomic actions with performance specifications. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 784–791, San Jose, CA, 2005. IEEE.

[46] Kirk Saban. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency. *Xilinx, White Paper*, 1(1):1–10, 2011.

[47] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–286, Providence RI USA, April 2019. ACM.

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

[49] Stuart Sutherland and Don Mills. Can My Synthesis Compiler Do That? In *DVCon 2014*, San Jose, CA, USA, DVCon 2014.

[50] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanovic´. RAMP gold: An FPGA-based architecture simulator for multiprocessors. In *Design Automation Conference*, pages 463–468, June 2010.

[51] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A case for FAME: FPGA architecture model execution. *ACM SIGARCH Computer Architecture News*, 38(3):290–301, June 2010.

[52] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: A virtual machine for programming modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 756–771, New York, NY, USA, June 2021. Association for Computing Machinery.

[53] Sheng-Hong Wang, Rafael Trapani Possignolo, Haven Blake Skinner, and Jose Renau. LiveHD: A Productive Live Hardware Development Flow. *IEEE Micro*, 40(4):67–75, July 2020.

[54] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, March 2007.

[55] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pages 105–117, New York, NY, USA, March 2023. Association for Computing Machinery.

[56] Claire Wolf and Johann Glaser. Yosys - A Free Verilog Synthesis Suite. In *21st Austrian Workshop on Microelectronics*, Linz, Austria, October 2013.

[57] Xin Wu. 3D-IC technologies and 3D FPGA. In *2015 International 3D Systems Integration Conference (3DIC)*, pages KN1–1. IEEE, 2015.

[58] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and André DeHon. HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 70–78, August 2022.

[59] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 933–945, New York, NY, USA, February 2022. Association for Computing Machinery.

[60] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, November 2019.

[61] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, November 2019.

[62] Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. Vidi: Record Replay for Reconfigurable Hardware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 806–820, Vancouver BC Canada, March 2023. ACM.