Hardware Checkpointing and Productive Debugging Flows for FPGAs

by

Sameh Attia

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

Hardware Checkpointing and Productive Debugging Flows for FPGAs

Sameh Attia
Doctor of Philosophy
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2022

Abstract

As FPGAs become larger and more complex, productive debugging is becoming more challenging. In this work, we detail a new debugging flow based on hardware checkpointing that provides full visibility and controllability while maintaining reasonable execution speed. Hardware checkpointing is useful not only for debugging but also enables several other capabilities such as live migration, fault recovery, and context switching; however, it has been difficult to achieve for FPGA applications. In this thesis, we overcome the challenges of checkpointing FPGA designs and realize the proposed checkpoint-based debugging flow. First, we propose techniques and wrappers that can safely interrupt a running design to create a consistent (restartable) checkpoint while avoiding hazards such as data loss or deadlock. We also develop approaches and tools that can access buried on-chip state that cannot be directly captured to create complete checkpoints. We next propose a checkpoint-based debugging framework, StateMover, that can seamlessly move the design state back and forth between an FPGA and a simulator, achieving the best of both worlds: speed and observability. Finally, we build a transaction-based co-simulation framework, StateLink, to extend the functionality of the proposed debugging flow to systems that cannot be entirely moved to a simulator such as CPU+FPGA accelerators or datacenter-scale applications. The combination of these tools enables new and productive debugging flows. StateMover allows designs to run at full hardware speed until a region of interest is approached. Then, a checkpoint can be loaded and its execution observed and controlled in a simulator. StateLink allows a designer-selected portion of the system to be moved into a simulation, enabling simulation speedups of up to 25x versus simulating the entire design. We demonstrate on several designs that StateMover and StateLink support can be added to a design with low resource and timing overhead, and illustrate the utility of the flow by debugging a complete Memcached system.

Acknowledgements

First, I would like to thank my supervisor, Vaughn Betz. I am deeply appreciative of the time and effort he spent mentoring me and editing our papers. He provided very helpful feedback, which has significantly improved this work and has greatly developed my academic, writing, teaching and coding skills. Vaughn is a true mentor on both academic and personal levels. He made my PhD journey enjoyable and not stressful. I will be forever indebted to him.

I would like to thank my fellow lab mates and friends. Our fun breaks and discussions made my graduate school experience more enjoyable and fruitful. I would particularly like to thank Abed, Adel, Amin, Andrew, Dafrawy, Ibrahim, Kevin, Kimia, Linda, Marius, Mathew, Mohamed Ibrahim, Mustafa, Sadegh, and Tanner.

I would also like to thank my PhD committee members, Paul Chow and Jason Anderson for their helpful feedback and discussions over the years.

During this work, I have also been fortunate to receive funding from the Connaught scholarship, the Right Track CAD scholarship, and the University of Toronto.

Moreover, I am very grateful to my parents and my sister for providing me with continuous support and encouragement. Thank you for always having my back.

Finally, this work would not have been possible without the care and support of my wife, Marwa. Thank you for the unconditional love and for taking care of our little ones: Laila and Yassin.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Debugging productivity is critical for FPGA projects as currently 51% of FPGA project time, on average, is spent in verification and debugging [1]. Since FPGAs enable realization of hardware design simply by reprogramming a device without a lengthy wait for custom manufacturing, this long debug cycle is particularly unfortunate as it undermines the time-to-market advantage that FPGA programmability enables. Current FPGA debugging flows are based on either simulation or on-chip debugging that captures hardware information for later examination. Simulation allows an engineer to observe the entire state of a design and to control (change) this state; this makes it easier to find the root cause of bugs. However, simulation is extremely slow; the simulator speed is typically between 10 Hz and 10 kHz [2, 3, 4], which means simulation can be run only on smaller parts of a system or for brief execution periods. On-chip debugging, on the other hand, allows the design to run at full hardware speed, and hence is 100k×-10M× faster than RTL simulation, enabling execution of vastly more test-cases. However, it provides much less observability and controllability than simulation, which makes it harder to find the root cause of bugs. Both the slowness of simulation and the limited visibility of on-chip debugging significantly reduce debugging productivity.

FPGA debugging flows are markedly less productive than those of software debugging [5]. Software programs can run with full visibility without much speed loss (typically less than 10×). Also, recently software debuggers have appeared that automatically create several checkpoints of a running program. This not only allows examination of how state changes as the program runs forward in time, but also allows going backward to see what caused a bug [6]. For software crashes, data centers also rely on checkpointing of software processes to be able to trace the root cause of intermittent faults/bugs. Hence, similar capabilities for FPGAs that can achieve the software-like combination of speed and observability, and can create checkpoints for debugging purposes are highly desirable.

Hardware checkpointing, in which a snapshot of the current state of an FPGA task is taken, is useful not only for debugging but also several other desirable flows. First, hardware checkpoints enable live migration of tasks in data centers from one server to another to balance loads or adapt to failures [7]. Second, they can be used for fault recovery in which a fault-free checkpoint is loaded back when a fault occurs instead of having to start from the initial state of the task [8]. Third, creating and loading checkpoints also enables context switching between tasks in a manner similar

to CPUs. Context switching allows multiple, possibly long-running, tasks to time-share a physical FPGA device in a data center, and can also be used for efficiently running tasks that are used sporadically [9].

However, hardware checkpointing has two major challenges. The first challenge is how to safely stop a design with multiple clocks and multi-cycle I/O transactions in a manner that allows the capture and restoration of a *consistent* state from which the design can resume execution without data loss. The second challenge is how to save and restore the entire state of a design to be able to create and load a *complete* checkpoint. This is not trivial since hard blocks, such as block RAMs and DSP blocks, in current FPGAs have inaccessible *buried* state elements that cannot be directly saved or restored with either the FPGA readback circuitry or designer-inserted scan chains. Currently, these two challenges greatly reduce the subset of designs that can be checkpointed.

In this work, we enable hardware checkpointing for a wide variety of FPGA designs by creating safe task interruption rules and flows and by developing techniques to access the buried state inside FPGA hard blocks. These new methods allow us to capture consistent (restartable) and complete checkpoints, which enable live migration and context switching for FPGAs. Moreover, we leverage these hardware checkpointing capabilities to create a novel FPGA debugging flow that combines simulation and hardware execution in a seamless way to achieve the best of the two worlds – full observability and controllability with high execution speed.

1.2 Contributions

This thesis is based in part on three peer-reviewed journal publications [10, 11, 12] and three refereed conference proceedings [13, 14, 15]. The main contributions of this thesis are as follows:

- Identifying and simulating the hazards that can arise due to task interruption. We also derive rules that should be followed to avoid these hazards.
- Designing wrappers that can be placed around an FPGA task to achieve safe task interruption. The wrappers have a small area overhead and typically no timing overhead.
- Proposing general techniques that allow a complete checkpoint to be captured and loaded when a design contains buried state elements. The proposed techniques include modifying the design by adding readable capture registers and modifying the readback process using multi-cycle capture.
- Developing StateReveal, an open-source tool which detects buried state elements and automatically inserts the required capture registers and control logic to add observability and controllability of the inaccessible state inside these hard blocks. StateReveal enables creating complete checkpoints using automatic and low-cost design modifications.
- Creating an open-source checkpoint-based debugging framework, StateMover, which can move the design state back and forth between an FPGA and a simulator. StateMover can safely interrupt a running design at a user-defined breakpoint. It can read out the entire on-chip design state and the off-chip memory used by the design. This checkpoint can be transferred from an FPGA to a host computer, and can be loaded into a simulator, on which the design can continue running from this checkpoint. Moreover, StateMover can extract the state of a

design from a simulator, write it back into an FPGA and resume hardware execution. These capabilities are sufficient to realize the aforementioned software-like debugging flow, in which the design is run on the FPGA most of the time to benefit from the ∼1M× speedup over simulation and is only moved to the simulator when visibility is needed. To our knowledge, this is the first work that can move a design state back and forth between a simulator and an FPGA. It is also the first work to enable in-system debugging with full visibility for complex designs that have multiple clock domains and multi-cycle I/O interfaces.

- Building an open-source transaction-based co-simulation framework, StateLink, which enables a more fine-grained control on what to move between the FPGA and the simulator. It allows only part of the design (the task) to be moved to a simulator while still being connected to and active in the overall system. StateLink extends the benefits of checkpoint-based debugging frameworks like StateMover to tasks interfacing with other system parts that cannot be checkpointed, including external I/O streaming interfaces, large off-chip storage, and modules without simulation models. It also significantly speeds up the simulation of tasks that are part of a larger system by up to 25×.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and reviews related work on hardware checkpointing and current FPGA debugging techniques. Chapter 3 discusses the hazards that arise when a running design is stopped at an arbitrary point, and presents design rules and wrappers to achieve safe task interruption. It also quantifies the cost of using these wrappers. Techniques to access the buried state and the StateReveal tool are presented in Chapter 4. Chapter 5 proposes the new software-like debugging flow and shows the usage of StateMover and its implementation details. Chapter 6 presents the StateLink framework, and highlights the benefits of the proposed debugging flow on a complete Memcached design. Finally, Chapter 7 concludes this thesis and summarizes potential directions for future work.

Chapter 2

Background and Related Work

2.1 Hardware Checkpointing

Hardware checkpointing is a technique in which a snapshot of the current state of a task is taken and execution can later be resumed from this checkpoint on the same device or another device, as shown in Fig 2.1. Hardware checkpointing on FPGAs can enable several capabilities that are widely available and used with CPUs. In this section, we discuss capabilities that are enabled by checkpointing, and discuss the two main challenges that have to be solved to make checkpointing available for FPGAs.

2.1.1 Checkpointing Applications

Live Migration

Hardware checkpointing is essential to enable live migration, which allows moving tasks from one server to another to balance loads or adapt to failures. Live migration has been widely used in data centers for moving virtual machines (VMs) over a network as shown in Figure 2.2. VM live migration is used in Microsoft Azure [16] and Google Compute Engine [17] to allow a physical node to be shut down for maintenance and to move VMs off of a failing node. This process should be transparent for the VM user. For example, in the Google Compute Engine, the disruption time of live migration is usually well under one second [17]. As FPGAs are being deployed in data centers as in Microsoft Catapult [18], Amazon EC2 F1 [19] and Alibaba Cloud F3 [20], live migration for FPGAs has been recently explored [7, 21, 22, 23]. FPGA live migration requires the FPGA design to be stopped and its internal state and that of its external memory to be captured from one FPGA-based server node and restored on the target node.

Fault Recovery

Hardware checkpointing can also be used for fault recovery in which a fault-free checkpoint is loaded back when a fault occurs instead of having to start from the initial state of the task [8]. This is achieved by taking periodic snapshots of the running task as shown in Figure 2.3. Periodic checkpointing has been used in embedded processors to enable fault recovery [24]. Once a fault is detected, the processor is halted and the last checkpoint is loaded so that a fault-free execution can

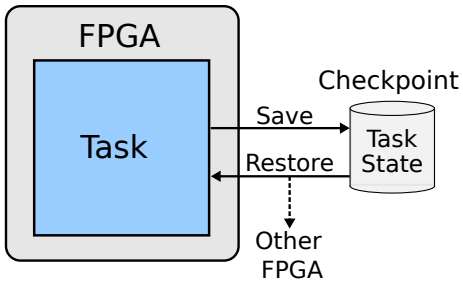


Figure 2.1: Hardware Checkpointing on FPGAs.

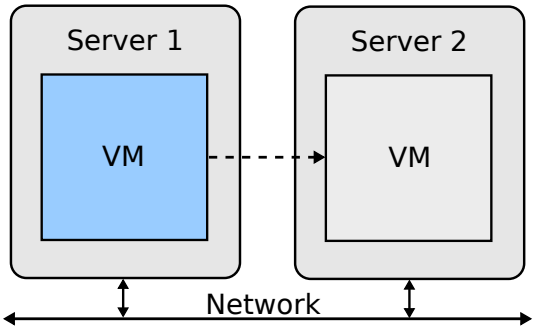


Figure 2.2: Live migration.

continue. Periodic checkpointing has also been used in energy-harvesting devices to be able to make progress despite power failures by saving the program state into a non-volatile memory [25, 26]. It has been leveraged also in distributed systems to make them fault-tolerant [27]. Fault recovery is needed in high-reliability FPGA-based real-time systems [28] and in FPGA-based scientific applications which run for long periods [29] to improve their vulnerability to transient faults and to recover from soft or hard errors.

Context Switching

Being able to create and load checkpoints also enables context switching between tasks, in a manner similar to CPUs, enabling time-sharing of hardware resources. Context switching has been an essential part of CPUs to allow multiple programs to share the CPU and to maximize system performance by swapping out idle programs [30]. It requires having precise interrupts (described in Section 2.1.2) so that the program can resume properly [31], and saving the program state which includes CPU registers and some memory contents. Several studies have proposed using the same technique on FPGAs to allow multiple, possibly long-running, tasks to time-share a physical FPGA device in the data center, and to enable more efficient execution of tasks that run sporadically [9, 32, 33, 34, 35]. It requires 1) interrupting a running task in a way similar to CPU precise interrupts, 2) saving the internal state of the task represented by the values of the state holders like task registers and memories, and 3) loading the new task and restoring its internal state by re-configuring the FPGA as shown in Figure 2.4 [36]. Partial reconfiguration (discussed in Section 2.1.4) is used so that only a portion of the FPGA design (the task) is swapped in/out without affecting the rest of the design [9, 32, 33].

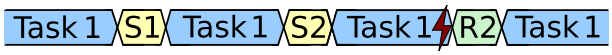


Figure 2.3: Fault recovery: after a fault occurs while running Task 1, the latest checkpoint (captured at S2) is restored (R2).

Figure 2.4: Context switching: to switch from Task 1 to Task 2, the state of Task 1 is saved (S1) and then the previous state of Task 2 is restored (R2).

FPGA Debugging

FPGA debugging, which is the application that we focus on in this thesis, can also benefit from hardware checkpointing. Checkpoints can be saved periodically or when predefined logic monitoring a breakpoint condition triggers [37]. By analyzing the state of a task in multiple subsequent snapshots, one can check whether the task is operating correctly and debug task faults. Checkpointing is particularly valuable in a data center where thousands of CPUs and FPGAs may be collaborating, and hence debugging intermittent faults can be extremely difficult without the ability to perform reproducible experiments and analyses that rely on checkpoints. Recently software debuggers have appeared that automatically create many checkpoints of a running program to allow programmers to not only examine how state changes as the program runs forward in time, but also as they go backward to see what caused a fault [6]. Similar checkpoint-based flows are possible for FPGAs, *if* FPGA tasks can be safely interrupted and checkpointed.

2.1.2 Safely Interrupting a Running Task

One of the main challenges for supporting hardware checkpointing is safely stopping a running task so that when the saved state of the stopped task is reloaded, the task can continue from exactly where it left off and no data loss or deadlocks occur. In CPUs, stopping a running process is performed using precise interrupts so that the execution can be restarted at the interrupt point properly [38]. An interrupt is considered precise if there is an interrupt point for which all instructions before it have been completed and no following instructions have modified the program state as shown in Figure 2.5. Precise interrupts are complex to implement in pipelined CPUs with out-of-order execution [31] and are hard to achieve in FPGAs since FPGA tasks are not controlled by a single instruction stream and usually do not have a single clock.

Stopping a hardware task at an arbitrary time can cause several hazards: data loss, data corruption, task deadlock, and system lockup. These hazards are particularly likely with tasks that have multiple clocks or multi-cycle I/O interfaces [36], which is the case in virtually all modern FPGA designs. For example, in data centers FPGA tasks usually communicate with DDR, Ethernet, and PCIe, and stopping a task in the middle of sending/receiving a transaction on these interfaces is definitely hazardous as shown in Figure 2.6.

Previous work on context switching and hardware checkpointing assumed that tasks have one clock [36, 9, 40], and did not handle multi-cycle I/O transactions [33]. Some prior work has achieved safe task interruption only in certain frameworks like ReconOS [40], or only for tasks generated by specific flows like OpenCL [41, 21]. In ReconOS, tasks have FIFO-based interfaces, and a task is

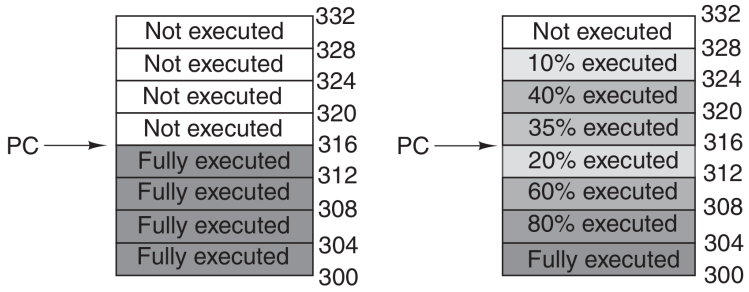


Figure 2.5: CPU precise interrupts (left) vs imprecise interrupts (right) [39].

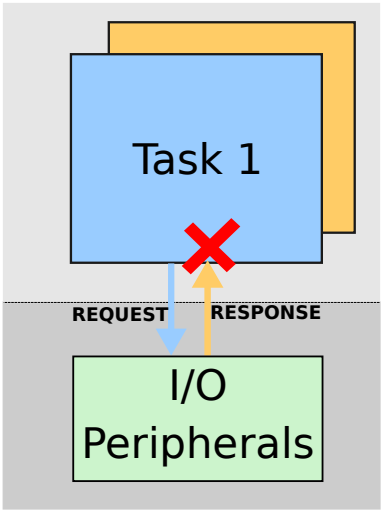
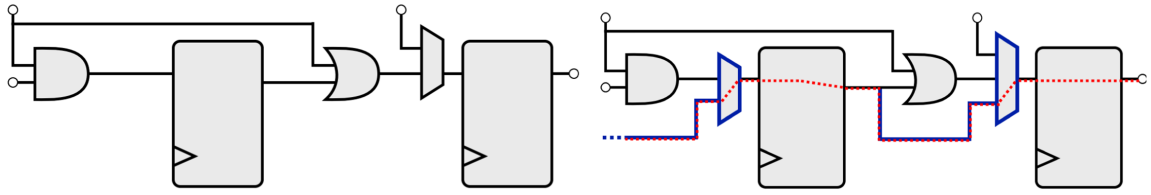


Figure 2.6: Example interruption hazard: a task is interrupted (and swapped out) just after sending a request to I/O peripherals. The response will not be received by the task, leading to data loss and/or task deadlock when the task is resumed later.

not stopped until all its interface FIFOs are empty. However, waiting for the FIFOs to completely drain is not practical if a task keeps sending/receiving data over these interfaces. As well, most prior work allowed task interruption only at specific, designer-identified points [42, 41, 43, 21]. One of the thesis goals is to develop the FPGA equivalent of the precise interrupts supported by all modern processors: the design may take many clock cycles to stop, but once it stops its state must be consistent, i.e., can be restarted from exactly that point (e.g., no in-flight I/O transactions where some of the design state is in the I/O interfaces).

2.1.3 Saving and Restoring the Entire Hardware State of a Task

The second challenge and the second step for checkpointing a task after interrupting it is to capture all its state. A task state is represented by the value of all state holders inside the task, including task registers and memories. There are two techniques to save and restore the on-chip state of a task.



(a) Original Circuit (b) Modified Circuit

Figure 2.7: Scan chain insertion [33] (blue represents added hardware, and red represents the state restoration path).

Scan-based Checkpointing

The first technique is to add additional state access hardware to the task such as scan chains [8, 33, 3, 44, 35]. For checkpointing, the access hardware should be able to read out the entire design state and also restore the state to all the state elements inside the task. An example of a simple 1-bit scan chain that can read and write the state is shown in Figure 2.7 in which FFs are connected into a chain by adding MUXes. However, this technique has significant area overhead to access all the task registers and to read out memory contents; Koch [8] reported an 80% logic overhead for example. Moreover, the process of adding the state access hardware to all registers and memories and connecting them in a specific scan order requires several iterations and timing analysis to select the optimal implementation and avoid a significant degradation in performance. This process can be done on the HDL level [23], netlist level [8], or on the C level for HLS designs [33]. To reduce area overhead, the work of [44] proposed a checkpointing architecture in which the HDL description of the design is studied to create a reduced set of essential state elements on which state access hardware is added, but still the area overhead of some designs is approximately 100%. The user can also find execution points that have a minimal set of essential state (live variables) and only add the access hardware on these states [8, 33], but this restricts the user to taking checkpoints at only these points. Scan-based checkpointing requires stopping the design clocks to read out and write back the design state. The shadow scan chain approach instead duplicates all registers so that a state copy can be performed in one cycle without stopping the clock [8]. However, this less disruptive checkpointing comes at a massive area overhead (more than 100% increase in LUTs and FFs) and does not work for designs with BRAMs.

Readback-based Checkpointing

Readback is an alternative technique for accessing the FPGA internal state which does not require any special access hardware to be added to a design [36, 45, 9, 7, 46]. Readback is a hardware utility that allows read out of the current values of the registers and memories that can be initialized in the bitstream configuration (logic block registers and BRAM contents) [47]. This is achieved by reading the configuration memory of the FPGA using a configuration interface like JTAG and generating a readback file as shown in Figure 2.8. The configuration memory contains the information of all programmable aspects of the FPGA including LUTs, logic block registers, BRAMs, and routing resources. During FPGA configuration, the configuration memory is updated with the bitstream's configuration data which contains the initial values of the logic block registers , distributed memories and BRAMs. The initial purpose of the readback hardware was to verify that the FPGA has been

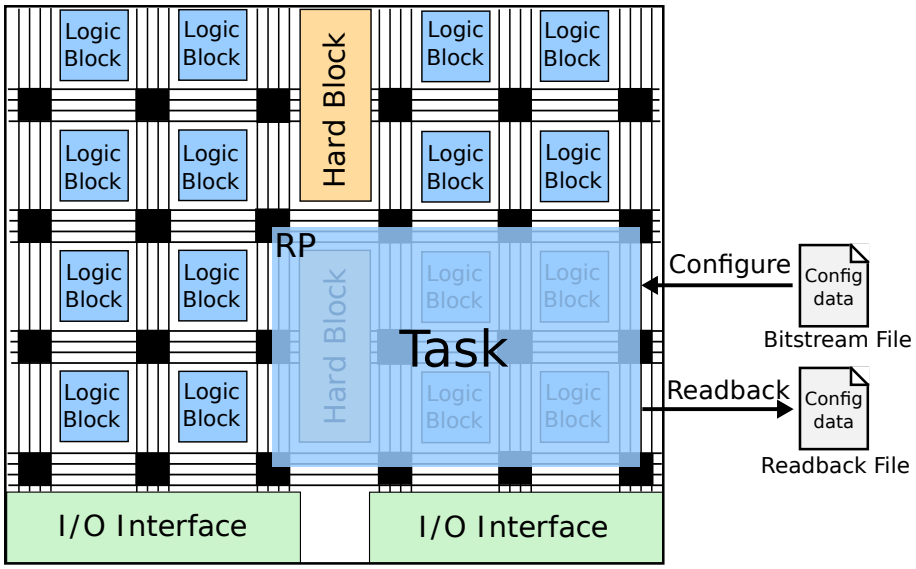


Figure 2.8: Saving and restoring the state of a task using readback and partial reconfiguration.

configured correctly by reading the configuration memory and comparing it with the bitstream and ignoring the components that have dynamically changing values in the user design. However, since the generated readback file also contains the current values of these user state elements, the readback hardware can also be used for checkpointing and debugging. Readback is available on Xilinx FPGAs and the latest Intel FPGAs[1], and enables most state elements to be retrieved from the device. A similar capability (i.e., SmartDebug) is also available on recent Microchip FPGAs [48].

To restore a task state, bitstream manipulation is required; the task state read using readback has to be embedded inside the initial full/partial bitstream by overwriting the initial values of task registers and memories. The FPGA is then programmed with the updated bitstream using full/partial reconfiguration as shown in Figure 2.8. Using a partial bitstream (i.e., the bitstream of the reconfigurable partition (RP) where the task resides) and partial reconfiguration (discussed in Section 2.1.4, in which only a part of the FPGA (the RP) is reconfigured, is recommended so that the rest of the design and external I/Os (DRAM, PCIe, Ethernet) are not affected by the writeback.

Unfortunately, some state elements, such as registers embedded within the FPGA hard blocks such as DSP pipeline registers and BRAM input registers, and registers embedded within the routing architecture such as the hyper-registers within the Stratix 10 interconnect, cannot be read out nor written back [41]. Capturing the state of a design that has these items results in an incomplete checkpoint. As the use of these registers is pervasive in FPGA designs [49], this poses an additional challenge for enabling hardware checkpointing on FPGAs.

Capturing the I/O state is another major challenge for checkpointing. I/Os such as Ethernet, DRAM, and PCIe can contain hard cores and buried state that cannot be captured. In addition, they can be shared among several tasks in multi-tenant FPGAs, and hence, stopping them to capture/load their state will affect tasks not involved in checkpointing.

Previous works on live migration and context switching worked around these challenges by using HLS/OpenCL flows and only allowing the switch/migration at specific points where either no state

[1] According to the release notes of Intel's Startix 10 and Agilex FPGAs

or pre-determined state needs to be captured [33, 41, 21]. For example, in [33], checkpointing of HLS designs is allowed only at points where there are minimum live variables on which state access hardware is added. In [41] and [21], state saving/restoring is allowed only at the end of a completed OpenCL work-group at which no internal state needs to be saved. This limits the set of designs that can be checkpointed, and it cannot be used for debugging or checkpointing the internal operation of the design. Thus, we seek to close this gap in this thesis and enable the checkpointing of a wide range of designs.

Readback-based checkpointing also requires task interruption before the state can be safely read out as the readback hardware generally requires all task clocks to be stopped in order to read out a consistent state [47]. On Xilinx Versal FPGAs, the state of designs with an operating frequency of 50 MHz or less can be read out without stopping the clock [50] but this is slower than almost all modern FPGA applications.

2.1.4 Partial Reconfiguration

Partial Reconfiguration (PR), in which a part of the FPGA (a reconfigurable partition) is reprogrammed without affecting the other parts of the FPGA, is required for context switching and can be used in readback-based checkpointing to efficiently restore the task state. Partial reconfiguration is supported on Xilinx FPGAs and recent Intel FPGAs [51, 52].

There are some design considerations when using PR in a design. First, the outputs of a reconfigurable partition should be tied to a constant value during the PR process. This is important to prevent spurious signals from the reconfigurable partition from propagating to the static (non-reconfigurable) region during the PR process and until the reconfigurable partition is reset. Thus, a *decoupler* should be inserted in the static region on the reconfigurable partition interfaces, especially on the control signals driven from the reconfigurable partition, to hold these interfaces in their inactive state. Xilinx PR Decoupler IP [53] and Intel Freeze Bridge IP [52] provide this functionality for AXI and Avalon interfaces, respectively.

Second, the PR process can result in a system lockup if the reconfigurable module is not stopped properly, mainly because of in-flight transactions on the module interfaces as we will discuss in Section 3.3. A PR controller such as the Xilinx PR controller [54] can be configured to wait until the reconfigurable module acknowledges that it is stopped and ready to be removed. However, the actual stopping mechanism has to be provided by the module itself. Xilinx provides PR AXI Shutdown Manager IP [55] which can be used on AXI memory-mapped interfaces of a reconfigurable partition to prevent the issuing of new transactions when a shutdown is requested while allowing on-going ones to complete. This IP can prevent system lockup for tasks with AXI memory-mapped interfaces but there is no support for other types of interfaces. As we will show in Section 3.3, even for AXI memory-mapped interfaces it also does not guarantee a safe task interruption (the task may not work properly when it is resumed). In this thesis, we develop additional task wrappers and design rules to achieve safe task interruption, making not only PR but also checkpointing possible.

2.2 FPGA Debugging

Productive FPGA debugging is becoming more crucial as the size and the complexity of FPGA systems increase. As shown in Figure 2.9, 51% of FPGA project time is spent, on average, in veri-

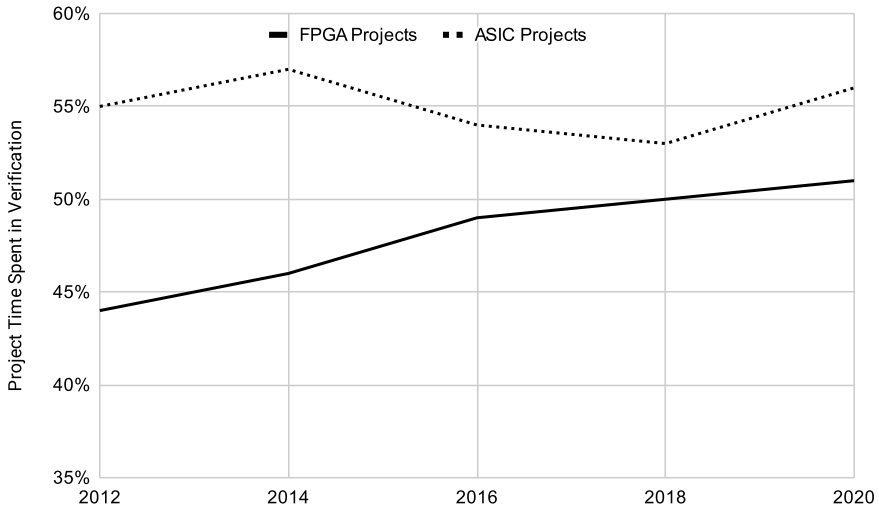


Figure 2.9: Percentage of FPGA and ASIC project time that is spent in verification and debugging.

fication and this percentage is increasing every year [1]. Despite the reprogrammability of FPGAs, this percentage is approaching that of ASIC projects as shown in Figure 2.9 [56]. Moreover, FPGA verification engineers spend 46% of their time on debugging [1]. This shows the need for new debugging flows that make use of FPGA features like reprogrammability and readback to help the user find the root cause of bugs more efficiently.

To debug an FPGA design, the user currently has two options: either use simulation or on-chip debugging tools. HDL simulation is the mainstay of FPGA debugging since it provides full visibility and controllability at no area cost. It is needed in the initial phases of design to perform logic testing of modules. Simulators are easy to use; they provide a design hierarchy where the user can select any module to observe its entire state at any time instance, making it easier to find the root cause of bugs. Users can even modify signal values to perform "what if" tests or to prove or disprove their hypothesis of what caused the bug. However, simulation is extremely slow compared to hardware execution. The simulation speed is usually below 10 kHz for mid-size designs [3]. It can be as slow as 10 Hz for very complex designs such as out-of-order or multi-core processors [2, 4] and for designs with high switching activity such as NoCs [57]. Thus, simulating a complex design can be $\sim$1M$\times$ slower than running it on an FPGA. Besides the speed disadvantage, RTL simulation usually runs without delays and certainly not with the spectrum of different possible signal delays on different chips. Thus, it does not fully capture everything that could happen on hardware; many timing-related bugs often appear only in hardware execution such as data races between clocks that were missed in the timing constraints, meta-stability, and clock-domain crossing issues. Moreover, even very large testbenches will not capture all sequences of events that occur in a live system.

For the aforementioned reasons, on-chip debugging that allows the design to run at hardware speed is an important part of FPGA debugging flows. It is essential for debugging large complex systems. Moreover, the speed of on-chip debugging enables running vastly more test-cases, allowing detection of rare bugs such as unusual input combinations, error states and unsafe resets. On-chip debugging techniques can be divided into three broad categories: 1) trace-based techniques, 2)

scan-based techniques, 3) readback-based techniques.

Before diving into on-chip debugging techniques, it is worth mentioning hardware emulation engines. They are known for their high debug capabilities that can be equivalent to simulators. They can be divided into processor-based and FPGA-based emulators. Processor-based emulators such as Cadence Palladium [58] and Synopsys ZeBu EP1 [59] provide full visibility of the system model. FPGA-based emulators such as Cadence Protium [60], Mentor Veloce Primo [61] and Synopsys ZeBu Server 4 [62] use several Xilinx UltraScale+ FPGAs to emulate the design-under-test. They provide high (not full) visibility of the design by using the trace-based techniques described in Section 2.2.1 augmented with off-chip storage and high-speed transceivers to allow capturing signal values during run-time. To provide full visibility, Mentor Veloce Strato [63] has a customized programmable logic chip named Crystal3 which contains an FPGA-like fabric that has dedicated resources to capture the necessary information to determine the value of all design signals. However, hardware emulation engines are extremely expensive, and hence, are not cost-efficient in most design projects. The second main drawback of emulators is that they can typically run at a speed of only 2 MHz, which is faster than simulation but around 100× slower than hardware execution.

2.2.1 Trace-based Debugging

Trace-based debugging is the most common on-chip debugging method. FPGA vendors rely on it to provide users with some on-chip visibility for their designs. For example, Xilinx and Intel offer Integrated Logic Analyzer (ILA) [64] (previously named Chipscope [65]) and SignalTap [66], respectively, as part of their commercial tools.

Trace-based techniques insert trace buffers into the design to record the values of on-chip signals as shown in Figure 2.10. The recording is done during the circuit operation, which means that the design can run at its full speed during debugging. The observed signal values are stored in on-chip memories, and then transferred to a workstation on which the values can be shown in a simulation-style waveform. Due to the limited on-chip memories, only a small subset of hardware signals can be observed and for a limited time. Hence, these approaches require the designer to create appropriate *trigger* logic that hopefully fires shortly before the error state is reached to capture relevant state data. Besides limited visibility, trace-based techniques have other disadvantages. They require design recompilation every time the list of observed signals or the trigger conditions are modified; since a full compile can take many hours, this is a major productivity bottleneck. These techniques also modify the design netlist to insert trace buffers, which could hide some timing-related bugs as the design timing changes after recompilation. Finally, the trace buffers are generally large and increase the usage of on-chip memories that are often in high demand by the design under test.

Previous academic work has tackled these problems using various approaches. The first approach is to intelligently select the most effective signals to observe in generic designs [67], high level synthesis (HLS) designs [68, 69, 70], and domain-specific (e.g., machine learning and soft processors) designs [71, 72]. For domain-specific designs, previous work has proposed to even perform some computations on the values of observed signals over time to provide the user with more useful information, instead of just raw data. For example, for machine learning applications, Noronha et al. [71, 73] propose compressing the values of the selected signals over time and providing the user with histograms that provide more insight into the behavior of the design without tracking all the changes. For HLS designs, the selected hardware signals can be mapped to the original software code, enabling more

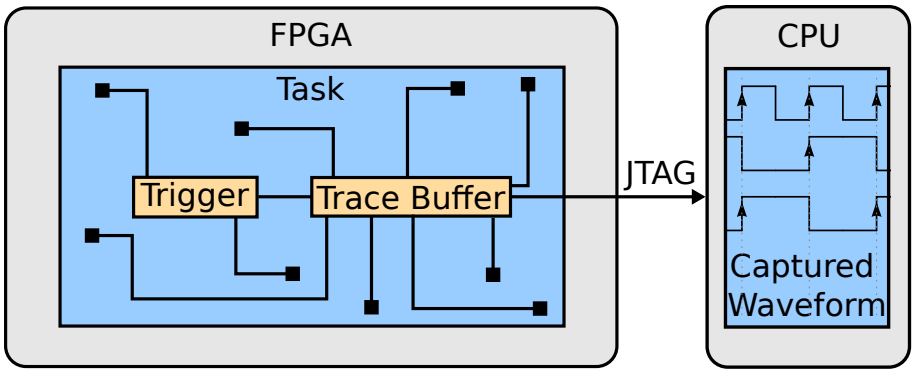


Figure 2.10: Trace-based debugging (yellow represents added hardware).

intuitive debugging of the HLS source code instead of signals synthesized from it [68].

Second, previous work has proposed incrementally inserting the trace buffers and the trigger logic after placing and routing the design, which not only eliminates the need for full design re-compilation but also preserves the design mapping and timing [74, 75, 76, 77, 78]. Unfortunately, incremental trace insertion is infeasible in highly utilized (70% or above resource utilization) FPGAs [79]. Incrementally inserting trace logic and preserving the previous design implementation as much as possible reduces compile time by 40% [78], but this still leads to fairly lengthy (minutes to hours) compiles for each trace insertion.

The third approach is to use partial reconfiguration to dynamically select the observed signals and change the trigger conditions during run-time. This is achieved by inserting multiplexers [80, 77] or by using debug overlays which can be generic [81, 82, 83, 84, 85] or domain-specific [86]. This can reduce the time between debug iterations, but adds restrictions on how many signals can be observed.

These trace-based approaches still have less visibility than simulators which have full observability in space and time as well as full controllability. Moreover, these approaches require the presence of spare on-chip resources.

2.2.2 Scan-based Debugging

Similar to scan-based checkpointing described in Section 2.1.3, scan-based debugging techniques add additional state access hardware to the design such as scan chains to provide state visibility [87, 88, 89, 90, 3]. The area and timing overhead of this approach is very high as mentioned in Section 2.1.3. Moreover, the provided visibility is available for only one cycle at a time since the design has to be stopped to read out the state. For a better understanding of the design behavior, the design can be single-stepped using clock control logic.

In [89], instead of single stepping the design, a snapshot is taken every specific number of cycles and saved in an external memory. Then, these saved values are set in the HDL file of the design and a new simulation is started to reconstruct the state between the snapshots. The primary inputs of the design have to be captured every cycle to be used in the state reconstruction.

DESSERT [3], an FPGA-accelerated method for RTL simulation, uses scan chains to provide full visibility of a certain region in the design. The RTL design is first translated by the FIRRTL compiler [91] which automatically adds scan chains into the design and transforms assert and print

statements in the RTL into error-checking and log generator circuits. DESSERT supports error replays by running two hardware instances of the design spaced apart in simulation time. Once an error is detected on the lead instance, a state snapshot is taken from the second instance and replayed in RTL simulation. DESSERT is similar to the debugging framework proposed in Chapter 5 in that it can move the state from hardware to a simulator. However, since it relies on scan chains it has a logic overhead of up to 80% and reduces the design frequency by more than 100%, and it completely changes the mapping of the original circuit which can hide time-related bugs. In addition, it assumes that the hardware execution is deterministic which is not the case for complex designs that may have data races. It also does not support writing back the state.

2.2.3 Readback-based Debugging

Readback-based debugging uses the readback hardware described in Section 2.1.3 to provide on-chip visibility without adding extra hardware (as in scan-based debugging) or requiring design recompilations (as in trace-based debugging). Several prior works have leveraged readback to build debugging frameworks for FPGAs [92, 93, 94, 95].

In [92], Angepat et al. present NIFD, a non-intrusive FPGA debugger, which provides a gdb-like interface that supports single-stepping and breakpoints. NIFD leverages readback to allow users to inspect the value of any state element in the design by typing its name in the console. NIFD adds minimal hardware to the design such as a clock controller to stop/resume the design, and a breakpoint controller. The major drawback of this framework is that it does not provide a way to visualize the readback data. It is almost impossible to debug a fault in a complex design with millions of signals by inspecting the signal values through a console with no signal history. In addition, NIFD supports only the relatively old Xilinx Virtex-II FPGAs.

In [93], Khan et al. present gNOSIS, an automated verification tool based on readback which verifies the correctness of hardware execution. The design is first simulated with the VCD (value change dump) option enabled to dump all the simulation output. Then, the design runs on the FPGA for some interval. After that, gNOSIS performs a readback and compares the values of registers in the FPGA with their expected values in the VCD file. If they match, gNOSIS resumes the design for another interval, and so on. Otherwise, the location and the time of the error is reported. The main problem with this flow is its need for a complete RTL simulation to be performed; this means it does not benefit from the speed of hardware execution. Moreover, gNOSIS can only read out the values of on-chip registers; it has no support for distributed or block memories. gNOSIS supports Xilinx Virtex-5 FPGAs.

In [37], Iskander et al. propose a readback-based low-level debug (LLD) framework. It consists of an on-chip processor which controls the design execution and communicates with the host over a serial port. Like NIFD, the serial console provides a gdb-like interface which allows the user to inspect register values using their design names. The framework has also on-chip condition-based breakpoint logic which allows the design to work at full speed until it reaches this breakpoint. The breakpoint logic is implemented in a separate reconfigurable region so it can be dynamically modified as long as the breakpoint condition is formed from signals that are already connected to this region. This framework shares the same major drawback of NIFD: it shows only signal values on the console without a time history which makes it impossible to debug complex designs. Moreover, register value inspection is significantly slow in LLD mainly because readback is controlled through the on-chip

processor which initiates a configuration frame readback for each register bit inquiry; a single bit value is retrieved in half a second making examination of full design state impractical. LLD supports Xilinx Virtex-5 FPGAs.

In [94], Li et al. propose the AMIDAR debugging framework for debugging software and hardware problems of soft-core processors. The framework provides a user interface through Eclipse which is used for software debugging and also for inspecting the value of on-chip state elements through readback. The framework supports Xilinx series 7 FPGAs. The framework does not freeze the design during readback to avoid data loss from the DRAM controller. However, this introduces some problems. First, since the BRAMs cannot be read out while they are being accessed by the design, AMIDAR has to force the processor not to execute any BRAM accesses during readback. Second, since the design is not frozen, the readback values of various state elements could be obtained during different clock cycles (i.e., inconsistent state). Third, modifying the contents of state elements (i.e., writeback) is not possible. Moreover, in recent FPGAs such as Xilinx UltraScale reading back the state without stopping all the design clocks is not supported [47, 95].

The debugging framework presented in [95] extends previous readback-based debugging frameworks by supporting applications with multiple asynchronous clocks. This is accomplished by using a clock-stop-restart-controller (CSRC) block which provides deterministic clock stopping and restarting. Once all the clocks of a multi-clock design are stopped, the framework issues a readback and constructs a waveform of the readback values for visualization. The framework has a Tcl interface and supports the latest Xilinx UltraScale FPGAs.

These readback-based frameworks do achieve full visibility but still have the following major drawbacks. First, most of these techniques give the user a gdb-like interface to inspect signal values; this could be acceptable in small designs in which the user knows what to look for, but for complex hardware designs with thousands of on-chip registers and memories they will be very difficult to use. Second, readback, unlike trace buffers, offers no signal history and hence it provides the user with only a single value for each on-chip signal, making it harder to debug the root cause of a fault. It is almost impossible to debug a complex design with millions of signals by inspecting the signal values through a console with no signal history.

Moreover, to retrieve signal values over time in order to provide the user with a simulation-style waveform similar to what trace-based techniques offer, readback-based frameworks provide clock-stepping circuits which run the design for one cycle and then stop it so that a readback can be performed again, and so on. By performing very frequent readbacks, all of these prior works lose much or all of the speed advantage of hardware execution compared to simulation. As well, stopping a design that has multi-cycle I/O interfaces at an arbitrary time can cause several hazards such as data loss and deadlocks as discussed in Section 2.1.2. Thus, these readback-based frameworks are not suitable for in-system debugging of complex designs involving multi-cycle I/O interfaces. They also cannot be used for designs with buried state or external memories.

The debugging framework proposed in Chapter 5 extends readback to *fully checkpoint* designs by safely interrupting the design and capturing the entire state, allowing it to support in-system debugging of complex designs. It also overcomes the observability challenges (no visualization, no signal history) of prior frameworks by allowing the user to move design checkpoints back and forth between an FPGA and a simulator. The idea of moving the design state from an FPGA to a simulator using readback was first proposed in [45], but requires the designer to use JHDL (i.e., a

CAD system in which designs are described in Java), and does not support moving the state the other way around.

Chapter 3

Safe Interruption

3.1 Introduction

A major challenge for supporting hardware checkpointing is achieving safe task interruption, which we define as the ability to stop a task so that when the saved state of the stopped task is reloaded: 1) the task can continue from exactly where it left off, and 2) no data loss or deadlocks occur. Our goal is to develop the FPGA equivalent of the precise interrupts supported by all modern processors: the design may take many clock cycles to stop, but once it stops its state must be consistent and confined inside the task borders (i.e., there is no essential state in the I/O controllers, data center shell or other interfaces to the task) so that it can be restarted from exactly that point.

Unfortunately, stopping a hardware task at an arbitrary time can cause several hazards: data loss, data corruption, task deadlock, and system lockup. These hazards are particularly likely with tasks that have multiple clocks or multi-cycle I/O interfaces [36], which is the case in virtually all modern FPGA designs. For example, in data centers, FPGA tasks usually communicate with DDR, Ethernet, and PCIe, and stopping a task in the middle of sending/receiving a transaction on these interfaces is definitely hazardous.

Some prior work has achieved safe task interruption only in certain frameworks [40], or only for tasks generated by specific flows like OpenCL [41] as discussed in Section 2.1.2. In this thesis, we propose a more general approach by developing a set of rules that tasks must follow to enable safe interruption and by creating task wrappers that can be placed around a task to enforce these rules.

Our approach can be applied to any FPGA task including, but not limited to, tasks that have multiple clocks and tasks that communicate with external interfaces such as DDR and Ethernet. The only limitation is that task interfaces should either be memory-mapped interfaces (which we refer to as request-response interfaces) or streaming interfaces. The task wrappers we develop support the industry standard AXI and Avalon memory-mapped and streaming interfaces. The approaches we develop could be applied to other interfaces as well, but given the high and increasing use of these standards (in IP cores, within designs and in system integration tools) we believe supporting standard interfaces will be sufficient for most designs. In addition, in a virtualized FPGA, the static region (shell) is usually connected to the reconfigurable partitions (RPs) implementing the tasks through standard interconnects/interfaces so that any compliant task (role) can be plugged in these RPs [96, 97].

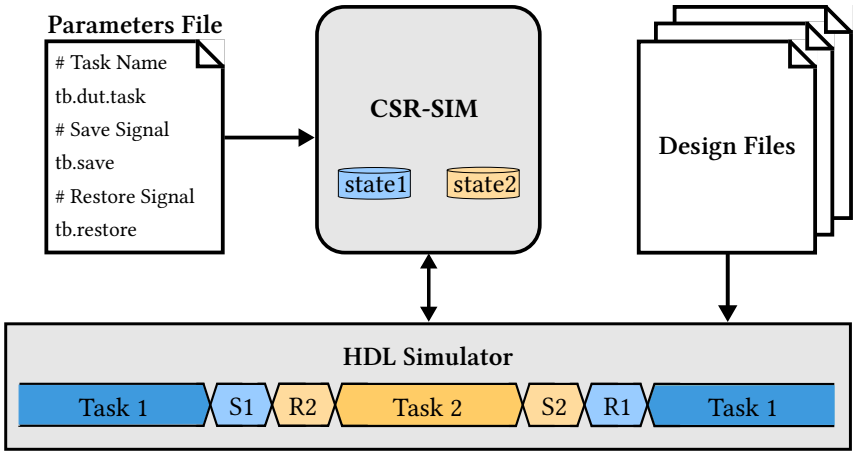


Figure 3.1: CSR-SIM simulator.

This chapter's contributions include:

- Building a context saving and restoring simulator (Section 3.2).
- Identifying and simulating the hazards that can arise due to task interruption (Section 3.3).
- Deriving rules that should be followed to avoid these hazards (Section 3.4).
- Designing task wrappers along with a task interruption controller to enforce these rules (Section 3.5).
- Quantifying the overhead added by these wrappers and demonstrating their use in a complex Memcached design (Section 3.6).

3.2 CSR-SIM

In this section, we introduce CSR-SIM, a Context Saving and Restoring SIMulator. CSR-SIM can read, write and modify the full context of a specific task in a design. It can be used for simulating hardware checkpointing and context switching, and can simulate the hazards that arise from task interruption and context change. We use CSR-SIM to identify the interruption hazards and to derive and test rules and wrappers to achieve safe task interruption. We also build on CSR-SIM to enable a new debugging flow presented in Chapter 5. CSR-SIM is written in C++ and uses PLI/VPI [98] to interface with the HDL simulation of the design. We are using Modelsim for HDL simulation; however, CSR-SIM is compatible with most commercial HDL simulators.

CSR-SIM, as shown in Figure 3.1, takes as an input 1) the name of the HDL module (i.e., task) for which the user wants to save/restore state, and 2) the name of the signals that trigger the save/restore events. These signals can be inside the design (e.g., from a controller) or test-bench signals that are triggered at specific times. At the beginning of the HDL simulation, CSR-SIM traverses the design simulation model created by Modelsim and creates a list of all the state elements, including registers and memories, inside the specified module and all its sub-modules recursively. It also stores the initial value of those state elements.

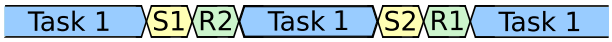


Figure 3.2: A convenient method for simulating context switching hazards in which CSR-SIM stores two different states (1 and 2) of the same task and switches between them (S denotes saving the state and R denotes restoring the state).

To simulate hardware checkpointing, CSR-SIM stores the current values of all these state elements when a save event occurs. Those values represent a task checkpoint, which is overwritten each time the save signal is asserted. Then, upon a restore event, CSR-SIM restores the latest checkpoint by restoring the previously saved values to the state elements of the task.

Saving the state of a task in hardware requires stopping the task (for thousands of cycles). In CSR-SIM, we can also simulate stopping and resuming a task. This is done by saving the task state, then ensuring that the task is inactive by setting all its state elements to zero or by using a decoupler [53] as discussed in Section 3.3.2, and finally restoring the saved state to resume the task. By changing the timing of the save/restore events, we can simulate all the hazards that can happen due to task interruption.

Context switching is simulated by adding MUXes on the interface between the static part of the design and the reconfigurable part to select which task is currently active. Upon a save event, CSR-SIM saves the state of the current task, and then corrupts its state elements to mimic the partial reconfiguration (PR) process (i.e., task swap-out) which clears out the state elements of the current task. To swap in a task, the MUXes are configured to activate that task and then the task saved state is restored by CSR-SIM. This flow is shown as part of Figure 3.1.

CSR-SIM provides another convenient method to simulate the hazards that arise from context switching that does not require any modification to the design (i.e., no added MUXes). In this method, context switching is performed between two instances of the same module. CSR-SIM stores two different states of the same module. These two states are initialized by the state of the module at the beginning of the simulation. To simulate context switching in this method, CSR-SIM saves the state of the current active task, corrupts its state elements to mimic the PR process, then loads the other saved state into the state elements, as shown in Figure 3.2.

CSR-SIM saves/restores the values of state elements inside the task; however, in some cases part of the task state may actually be outside the task. For example, if the task is communicating with an off-chip memory (DDR), that off-chip storage may be considered part of the task state. There are two basic approaches that can be used to enable checkpointing and context switching in this scenario. The first approach is to use memory isolation, in which each task is given a separate region of the DDR memory, so its DDR state persists when it is swapped in and out. This can be achieved by a small amount of memory isolation logic that adds a task base segment address to the requested address for each read and write, and also checks that a task stays within its legal address range. This approach is usually used in virtualized FPGAs [99]. The second approach is to add logic to the design so that once a task is safely stopped, its DDR state is read out and transferred to the host. This approach has the advantage of allowing a task to use the entire DDR memory, but the disadvantage of a longer task swap time [22].

To our knowledge, this is the first work to build a generic state saving/restoring and context switching simulator. PR simulators such as ReSim [100] have been proposed in the literature; they focus on simulating the partial reconfiguration procedure and do not try to save or restore the state

of the design being replaced. In [101], Gong et al. added state saving and restoring functionality to ReSim, and this work is the closest in functionality to CSR-SIM. ReSim simulates the underlying hardware used in the state saving/restoring process including the Xilinx ICAP and readback process. It targets specific Xilinx FPGAs, and is useful for debugging issues related to the sequencing and control of the actual state readback/writeback process. In contrast, CSR-SIM is device-generic and simulates the design at a higher level, which provides a fast and easy way to simulate context switching and checkpointing and to analyze how tasks are interrupted, which is more suitable for our work.

3.3 Task Interruption Hazards

Stopping a task at an arbitrary time can cause several functionality problems, which we refer to as *task interruption hazards*. These hazards affect the ability to resume the task from where it left off and can also affect other tasks running in the system. In this section, we identify these hazards and their causes.

3.3.1 Methodology

Task interruption hazards arise from two basic sources: in-flight transactions on task interfaces and multiple clock domains inside the task. Task interfaces are divided into request-response interfaces (e.g., AXI/Avalon memory-mapped) and streaming interfaces (e.g., AXI/Avalon stream). To identify the hazards that are caused by in-flight transactions, we examine the different types of transactions (e.g., single/burst reads/writes) that could happen on the four widely used FPGA standard interfaces: AXI/Avalon memory-mapped and streaming interfaces. Having multiple clock domains inside the task can also cause hazards when the task is interrupted. Virtually all recent FPGA applications have multiple clocks and in general contain both related clock domains (phase-related clocks with frequencies that are rational multiples) and unrelated (asynchronous) clock domains. We study the hazards that could occur due to the interruption of a task with multiple related and multiple unrelated clocks in order to find methods to support safe task interruption for a wide range of applications.

To identify the task interruption hazards, we create several basic designs that contain each type of the aforementioned interfaces in isolation, and both types of clock relationship. Then, we use CSR-SIM to simulate and identify the task interruption hazards in each of these created designs. The task interruption is performed *exhaustively* to ensure that we have identified all the possible hazards and to test our solutions for these hazards: we check all possible types of transactions on each of the interfaces, and we stop and resume on every clock cycle during these transactions. Since we are testing all possible types of transactions on each interface, our methodology for identifying task interruption hazards does not depend on what is located on the other side of the task interface; it could be an another task, Ethernet interface, off-chip memory, or a processor subsystem. We define a hazard to be present if the interrupted hardware (for any choice of interruption clock cycle) does not produce the same output it would have produced if we never interrupted the design.

We use CSR-SIM to simulate two scenarios of task interruption in which hazards could occur. In the first scenario, which we denote as *stop-resume*, we stop the task, save its state, and then resume the task again. This scenario shows the hazards that could occur if the task is interrupted at an

arbitrary point and resumed again without modifying its state – it occurs in hardware checkpointing when a snapshot of the design is taken, as that requires stopping the task for some time. In the more complex scenario, which we denote as *stop-switch-resume*, we stop a task, save its state, replace it with another task, run the new task for some time, and then later switch back to the original task and restore its state. This scenario shows the hazards that occur when a task is interrupted at an arbitrary point and then replaced by another task, which occurs in context switching between tasks. This scenario also simulates a superset of the events that occur when restoring a previously saved state (to the same task) in hardware checkpointing, so if no hazards occur in the *stop-switch-resume* flow, hardware checkpoint restores are also safe. As we mentioned earlier, the simulation of these two scenarios are repeated exhaustively for many *interruption points* (i.e., the clock cycle in which the task is interrupted) which are selected to cover all different types of transactions on the task interfaces, and to occur at each cycle from the beginning of a certain transaction to its completion.

This exhaustive simulation reveals many task interruption hazards, which we divide into four categories: data loss (D.loss) in which a task or its output destination does not receive some requested data, corruption of the task's data (D.corrupt), task deadlock (T.deadlock) in which it cannot continue after being resumed, and system lockup (S.lockup) in which stopping a task locks up other tasks so they cannot progress.

3.3.2 Request-Response Interfaces

In this part, we discuss the hazards arising from interrupting a task that has an AXI memory-mapped (AXI-MM) interface or Avalon memory-mapped (AV-MM) interface. Due to the similarities between these two interfaces, we will go through the hazards of AXI-MM in detail and then highlight the main differences for AV-MM. AXI-MM is the standard request-response interface adopted by Xilinx. It has five separate channels: read address (AR), read data (R), write address (AW), write data (W), and write response (B). Each channel has ready-valid handshaking between the master and slave of the channel. We create a design with two tasks communicating with a memory controller through a shared AXI4 interconnect, as shown in Figure 3.3, and simulate it using CSR-SIM. The purpose of having a second task is to test for system lockup hazards where stopping a task affects the entire system. Each task is placed in a separate RP while the memory controller and the shared AXI4 interconnect are placed in the static region (SR). A decoupler [53] is placed at the RP/SR boundary to prevent spurious signals by holding signals in the inactive state during PR. The two tasks write several words into separate regions of the memory, and then read them back and check for any mismatch in the read data. Both single and burst transactions are simulated.

Due to AXI handshaking signals, no data loss occurs if a task is swapped out at an arbitrary time and then swapped back again without being replaced by another task, given that the decoupler holds the valid and ready signals coming from the swapped-out task low. However, the entire design could lock up in various scenarios. For example, if a task, say Task A, is stopped after sending a request (write or read) and before receiving its response (write response or read data), the interconnect will hold the response since Task A is not ready to receive it. Then after the interconnect pipeline registers, if any, are full, the interconnect will send a back-pressure to the memory controller preventing any further responses from proceeding to Task B, which results in stalling the write/read channels while Task A is swapped out and a new task is swapped in. No other task can access the memory controller during this significant time, leading to system lockup

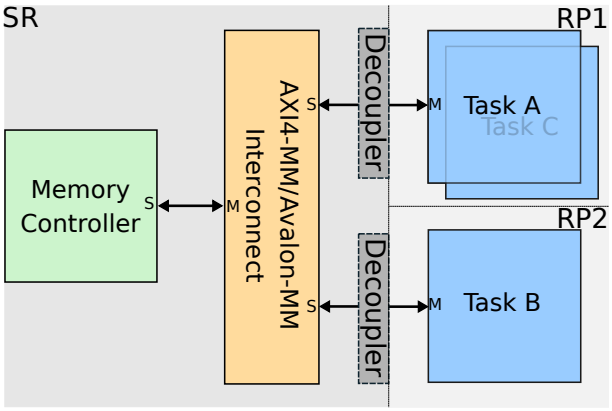


Figure 3.3: A request-response interface design example.

Table 3.1: Interruption hazards of the AXI memory-mapped interface.

Interruption Point	Stop-Resume	Stop-Switch-Resume
Read: before receiving the response	S.lockup	S.lockup, D.loss, D.corrupt, T.deadlock
Write: before receiving the response	S.lockup	S.lockup, D.loss, T.deadlock
Write: before sending all the data	S.lockup	S.lockup, D.loss, D.corrupt, T.deadlock

(S.lockup).

When Task A is replaced by another task, say Task C, that uses the same interface, the latter incorrectly receives the response of Task A. Eventually, when Task A is swapped back again, it keeps waiting for the lost response (T.deadlock and D.loss). Burst writes present another issue. In burst writes, a task blocks other writes until it writes all the data; if the task is swapped out before that, the write channel will be stalled (S.lockup). If the task is replaced by another task and the AXI interconnect chosen does not support outstanding transactions, all the tasks in the system including the new one cannot issue any writes (S.lockup). If outstanding transactions are supported, the new task can issue another write by writing an address on the write address channel; however, the written data will be considered a part of the previous burst write (D.loss and D.corrupt). A summary for the interruption hazards of the AXI memory-mapped interface is shown in Table 3.1. Hazards occur if a task with an AXI memory-mapped interface is stopped after initiating a transaction but before receiving the read/write response or before sending all the write data for a burst write.

To see if the AXI shutdown manager IP [55] provided by Xilinx can resolve these issues, we instantiate it on the AXI4 interface between the tasks and the interconnect, and run another CSR-SIM simulation. During the simulation, CSR-SIM waits until the AXI shutdown manager grants the shutdown request before saving/restoring the state. The main goal of the AXI shutdown manager is to prevent system lockup by ensuring that there are no ongoing AXI transactions at an SR/RP boundary when the RP is reconfigured. Thus, it allows all the transactions issued before the shutdown request to complete, and drops all the transactions issued after assertion of the shutdown request. The simulation shows that the Xilinx AXI shutdown manager does resolve the hazards shown in Table 3.1 for the case of a single interface and single transaction (read/write) in flight at a time. However, it is insufficient to achieve safe task interruption for some more complex cases as it does not guarantee that the task can resume when it is swapped back in. That is because the AXI shutdown manager is intended only for stopping the interfaces of tasks before swapping them out

Table 3.2: Interruption hazards of the AXI memory-mapped interface with the AXI shutdown manager.

Interruption Point	Stop-Resume	Stop-Switch-Resume
Multiple Reads: before receiving the first response	T.deadlock	T.deadlock
Multiple Writes: before receiving the first response	T.deadlock	T.deadlock
Multiple Writes: before sending all the first request data	T.deadlock	T.deadlock

Table 3.3: Interruption hazards of the Avalon memory-mapped interface.

Interruption Point	Stop-Resume	Stop-Switch-Resume
Read: before receiving the response	D.loss, T.deadlock	D.loss, D.corrupt, T.deadlock
Write: before receiving the response	N/A	N/A
Write: before sending all the data	S.lockup	S.lockup, D.loss, D.corrupt

when partial reconfiguration is used but not for checkpointing or context switching purposes.

For example, if a task issues two subsequent read requests, and a shutdown request is asserted right after the first request, the AXI shutdown manager allows the first request to proceed to the interconnect, drops the second one, then waits until the response of the first one comes back, and finally grants the shutdown request. From the task view, it has sent two requests, and it does not know that one of them was dropped since the master is notified about the dropped transaction (with SLV-ERR response) only if there is no active transaction of the same type. Thus, when the task is swapped back, it keeps waiting for the response of the second request forever (T.deadlock). A summary for the interruption hazards of the AXI memory-mapped interface with the AXI shutdown manager is shown in Table 3.2. Moreover, the AXI shutdown manager is not capable of handling dependent interfaces; in some scenarios, the AXI shutdown manager may never grant the shutdown request. For instance, consider a task that has two interfaces and to complete a transaction on the first interface a transaction on the second interface needs to be issued. If the shutdown request occurs before the second transaction is issued, the AXI shutdown manager will drop the second transaction and wait for the first transaction to complete, which will never happen (T.deadlock). An example of this issue is a DMA between two memory interfaces where a large read is initiated to the first interface, but it cannot complete without writing the data to the second interface.

Avalon memory-mapped interfaces lead to task hazards that are largely similar to those of AXI, but some differences arise. Avalon masters do not have a ready signal (i.e., no back-pressure) so data loss occurs if the task is swapped out during single reads or pipelined/burst reads since there is no way to make the interconnect hold the data (D.loss). In single reads, the valid signal indicating the validity of the read data is not required in Avalon (i.e., fixed-latency read), so incorrect data is read after resuming the task causing data corruption (D.corrupt). If the read data valid signal is used, the task goes into deadlock since it keeps waiting for the missed valid data as in the AXI case (T.deadlock). Another major difference between AXI and Avalon is that Avalon does not have a separate channel for write address; the address is sent with the first unit of the write data. Thus, no system lockup occurs in single writes, nor in any type of reads due to the absence of the master back-pressure. Swapping out a task during a burst write is the only cause of system lockup. In addition, the write response is not required in Avalon interfaces. Table 3.3 summarizes the hazards that occur for an Avalon memory-mapped interface that is stopped at different points during a read or write transaction.

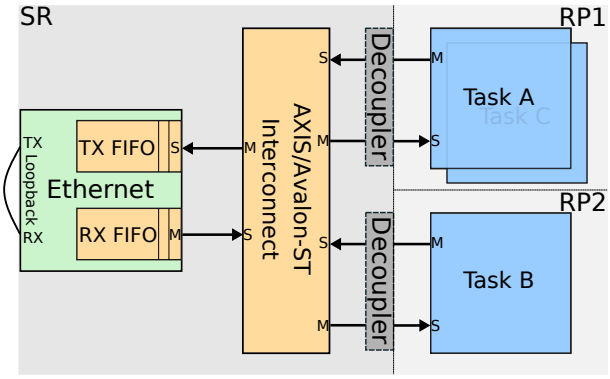


Figure 3.4: A streaming interface design example.

3.3.3 Streaming Interfaces

The streaming interfaces used by Xilinx (AXIS) and Intel (Avalon-ST) are similar. They both have back-pressure and support packet mode. Packet boundaries are detected by the *last*/*EOP* signal in AXIS/Avalon-ST. To simulate the hazards that could happen due to streaming interfaces, we create a design with two tasks communicating with an Ethernet core with internal buffers that has a loopback at the physical interface, as shown in Figure 3.4, and simulate it with CSR-SIM. The tasks generate packets, send them over Ethernet, and then receive the packets and compare them to the sent data. Each task has a dedicated Ethernet address and a routing table is used on the RX path to direct packets to the correct task according to the destination address in the Ethernet frame.

If a task, say Task A, is swapped out while sending a packet, different hazards can occur according to the interconnect configuration. In the first case, where the interconnect arbitration is performed only at packet boundaries, the interconnect will lock, and since Ethernet internal buffers do not send the data to the Ethernet MAC until a complete frame is received, the TX buffer and the entire TX path will stall until the *EOP* signal is asserted (S.lockup). If Task A is replaced by another, say Task C, the frame sent from Task C will be considered the continued data of Task A frame (D.corrupt), and when Task A is swapped back, it will continue sending data from the middle of a frame (and without asserting the *SOP* signal in the case of Avalon), so this data will also be lost (D.loss). In the second case, where the interconnect is allowed to perform another round of arbitration when the *valid* signal of the current source (Task A) goes low for multiple cycles, no deadlock will happen since the interconnect will forward the packets sent by other tasks in the system (e.g., Task B in Figure 3.4). However, those packets will be considered the continued data of Task A frame (D.corrupt). This is similar to the previous case except Task B that was not even part of the context switching will not receive the frame it sent (D.loss).

Another important task interruption point is when Task A is swapped out while receiving a packet. In this case, the interconnect will hold the data as the *ready* signal is deasserted by the decoupler. However, this stops the Ethernet RX buffer from draining the received packets which eventually causes a buffer overflow (S.lockup and D.loss). When Task C is swapped in and asserts the *ready* signal, it will receive the data destined for Task A (D.corrupt), and when Task A is swapped back, it will keep waiting for an *EOP* signal that will never come (T.deadlock). In addition to these hazards, receiving a new packet destined for a stopped task also locks up the RX path, which can cause buffer overflow and data loss for the entire system.

Table 3.4: Interruption hazards of the AXI and Avalon streaming interfaces.

Interruption Point	Stop-Resume	Stop-Switch-Resume
TX: before sending the entire packet	S.lockup	S.lockup, D.loss, D.corrupt
RX: before receiving the entire packet	S.lockup	S.lockup, D.loss, D.corrupt, T.deadlock
RX: receiving a packet for stopped task	S.lockup	S.lockup

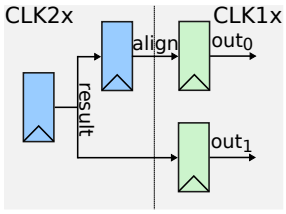
A summary for the interruption hazards of the AXI/Avalon streaming interface is shown in Table 3.4. Hazards occur if a task with an streaming interface is stopped before receiving or sending an entire packet. Note that the Xilinx AXI shutdown manager does not support streaming interfaces so it does not affect these hazards.

3.3.4 Multiple Clocks

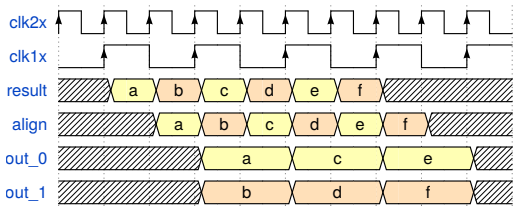
All the aforementioned hazards are caused by task interfaces; however, some hazards can arise internal to a task. Since stopping and resuming the task requires disabling and enabling the task clock to take a snapshot of task state elements and restore it back, having multiple clocks inside a task can cause several hazards. Task clocks can be related clocks (i.e., phase-related multiples) or unrelated clocks which require clock domain crossing (CDC) circuitry.

First, to simulate the hazards arising from context switching a task that has multiple related clocks, we create a design that uses the double-pumping technique to leverage the full throughput of a DSP block by multiplexing and outputting two data streams [102], as shown in Figure 3.5(a). Double-pumping is a widely used technique in which a block is operated at a higher clock frequency than its surrounding system. The two clock domains are working in lockstep, and the DSP block sees two clock edges and generates two results per slower (clk1x) cycle; these two results are then captured by the slower clk1x to feed the two output data streams. Figure 3.5(b) shows the normal behavior of the circuit. If the design is stopped and resumed at arbitrary clock ticks, this sequence can be disrupted. For example, suppose that the design is stopped, and a snapshot of state elements is taken, in the period between the positive edge of the faster clock (i.e., launch clock tick) and the positive edge of the slower clock (i.e., capture clock tick). If the design is resumed and the values of the state elements are restored in the period between the positive edge of the faster clock and the negative edge of the slower clock, the faster clock domain runs one extra edge before the capturing clock domain edge, overwriting one of the two results and causing data loss as shown in Figure 3.5(c). If the opposite occurs, the capturing clock domain will run before the two results are generated which results in data corruption.

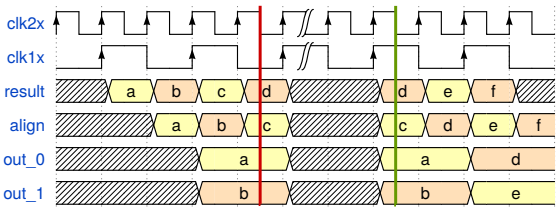
For the second case, unrelated clocks, hazards mainly depend on the type of synchronizers used. The commonly used FIFO synchronizers make the two clock domains nearly independent. Thus, stopping and resuming the two clock domains at an arbitrary time should be safe as long as no FIFO overflow or underflow happen. However, FIFO synchronizers still need to share some information between the two domains to generate the full and empty flags. Since disabling or enabling two unrelated clocks may not occur at the same time due to sampling edges and metastability resolution of the *clock-enable* signal, extra care should be taken for synchronizing this information. For instance, if a flag is generated in a slow clock domain, and needs to propagate to a relatively faster clock domain (e.g., <2x) using a conventional multi-FF synchronizer, holding the flag constant for two cycles of the destination clock should be enough to ensure the safe propagation of the flag. However, stopping



(a) Circuit



(b) Normal operation



(c) data loss/corruption due to task stoppage (red line) and resumption (green line)

Figure 3.5: A double-pumping design.

and resuming the task at an arbitrary time, and metastability resolution of the *clock-enable* signal can cause the slow clock domain to overwrite the flag before it is captured by the faster domain.

3.4 Design Rules for Hazard Avoidance

In this section, we derive a set of design rules that should be followed to avoid the hazards discussed in Section 3.3, and hence allow safe task interruption.

3.4.1 Atomic Transactions

As illustrated in Section 3.3, stopping a task in the middle of sending a request (e.g., a burst write) or sending a packet may cause system lockup and data corruption. Failing to wait for an entire response (e.g., read response) or an entire packet to be received may also cause data loss and task deadlock. Thus, we have to guarantee that all transactions that occur on task interfaces are atomic. That is, a transaction should either be fully completed or not be issued. To achieve that, all in-flight transactions at the I/O interfaces of the task should be allowed to finish before stopping the task. Completing a transaction includes receiving the entire response in request-response interfaces. For streaming interfaces, a transaction is considered complete when an entire packet is sent or received, when packet mode is supported, which is indicated by the *EOP* signal in Avalon streaming interfaces and the *last* signal in AXI streaming interfaces.

3.4.2 Stoppable Interface

Ensuring transaction atomicity is insufficient to guarantee safe interruption of a task. Since a task is not allowed to stop until all in-flight transactions are completed, we cannot stop the task if it keeps issuing new transactions while waiting for the ongoing ones to complete. Thus, we require task interfaces to be *stoppable*: we can prevent the issuing of new outgoing transactions without

stopping in-flight ones. Fortunately, all the standard request-response and streaming interfaces used by Xilinx and Intel can be stopped from issuing new transactions as we will discuss in the next section. It is worth noting that dropping transactions is different from stopping the interface from issuing transactions. If the new transactions issued by a task interface are dropped, the task will be stopped after all the in-flight transactions are completed; however, the task may lock up after resuming as it waits for a response that will never arrive, as detailed in Section 3.3.

3.4.3 Buffering, Dropping or Rejecting Incoming Transactions

The previous rule is about stopping the outgoing transactions in order to be able to safely stop the task, but one cannot control the incoming transactions that are issued somewhere else. For example, if a task has a slave streaming interface, packets can be sent to this interface at any time, and if the task is swapped out, those packets will either be incorrectly received by the newly swapped-in task or lock up the system. Thus, to be able to safely stop a task without causing system lockup or data corruption, incoming transactions to a swapped-out task should either be buffered, dropped or rejected, and should be isolated from the swapped-in task. The isolation is performed by identifying whether the incoming transaction is intended to the swapped-in task or the swapped-out task based on the address or the destination field of the transaction. The decision to buffer, drop or reject depends on the interface type. For streaming interfaces, incoming transactions (i.e., packets) to a swapped-out task should be buffered somewhere and then be fed to the task when it later resumes in order to avoid data loss. However, some streaming interfaces can tolerate data loss, so in this case, buffering is not required, and a dropping mechanism can be implemented instead to avoid system lockup. For example, Ethernet frames/packets can be safely dropped by relying on the transport layer to detect the packet loss and re-transmit the packets. On the other hand, if a task has a slave request-response interface, requests issued to the task after being swapped-out will cause a system lockup even if they are dropped or buffered at the task end. In this case, requests should be rejected by sending an error response notifying the sender of the unavailability of the slave like *SLV-ERR* response supported by AXI memory-mapped interfaces.

3.4.4 Stop Sequence For Dependent Interfaces

As proposed in the first and second design rules, in order to stop a task, all new transactions are blocked and all in-flight ones are allowed to complete. However, as shown in the AXI shutdown manager case in Section 3.3, a task can have dependent interfaces in which blocking a transaction on one interface could stop another transaction from proceeding. Consequently, stopping a task may cause system lockup if the ongoing transactions depend on the blocked ones. A trivial solution for this is to prevent all types of dependency between task interfaces. First, a transaction should not be issued until all the data needed to complete this transaction is available. For example, a task should not issue a write request until all the data to be written are available. Second, a task should not have to issue a request to be able to respond to another request; each request-response pair should be independent. Instead of preventing all dependent transactions, another solution is to force dependent transactions to be issued at the same time. By forcing this, dependent transactions will either be allowed to proceed, or all of them will be blocked. However, these two solutions limit the space of context-switch capable tasks. Instead, a general solution is to create a dependency

matrix indicating all the dependencies between task interfaces. Then, the proposed design rule is to sequentially stop task interfaces according to the dependency matrix. That is, an interface is stopped only after all dependent interfaces have been stopped. For example, even if an interface has no in-flight transactions, new transactions will be allowed to be issued on this interface until all the interfaces that depend on it have been completely stopped (i.e., no in-flight transactions). It is worth noting that it is not possible to have loops in the dependency matrix (e.g., A-B-C-A) since this case will cause system deadlock during the normal operation of the design. Also, dependent interfaces are not common in FPGA designs, so this rule is mainly for completeness.

3.4.5 Handling Multiple Clocks

As shown before, having multiple clocks in a task can cause several hazards when a task is interrupted. The reason for the hazards in a design with related clocks is that the related clock domains usually work in lockstep, which is disrupted by stopping and resuming a task at arbitrary clock ticks. Thus, the clock ticks relation between these clock domains must be preserved to avoid these hazards. To achieve this, the task should be stopped and resumed only at the edges of the slowest clock in the related group by having a centralized *clock-enable* signal that is controlled by the slowest clock, and fed to all the related clock domains. This ensures that all the related clock domains are stopped at the same time and the clock ticks relation is preserved.

Handling unrelated clocks is more complex since you cannot stop the different clock domains at exactly the same time. That is because even if there is a centralized *clock-enable* signal, it cannot reach the clock domains at the same time due to different sampling edges and metastability resolution. Thus, one clock domain can continue working one (or more) cycle(s) later than the other domain or can be resumed one cycle (or more) earlier. If the circuit makes no assumptions regarding the number of clock ticks on clock A versus the number of clock ticks on clock B which is usually the case for FPGA designs, we can safely stop the task with no extra work.

If it assumes some limits on the number of edges of B versus A as in the case discussed in Section 3.3, we propose to divide the task to smaller subtasks where each subtask has a separate clock domain (i.e., the clock domain crossing circuitry are located between the subtasks), and then apply the proposed design rules to stop each subtask individually. Then, since all the subtasks are stopped, which means that no in-flight transactions on all subtask interfaces, no clock domain crossing will be occurring on the synchronization circuits between the subtasks. Thus, the entire task is now safely stopped.

3.4.6 Safely Resuming with Multi-cycle Combinational Paths

Multi-cycle paths within a task are often viewed as an issue for task preemption [41]. A multi-cycle path is a register-to-register combinational path in which data launched from the source register takes more than one cycle to arrive at the destination register. For example, assume a task that contains a multi-cycle path of five is stopped one cycle before the capturing edge of the destination register, and then the task state is saved and later restored. When the task is resumed, the destination register will capture the data after one cycle, which seems like a possible issue – perhaps the data has not been given time to propagate through the five-cycle path, leading to a D.loss task interruption hazard. However, this hazard can be removed by the state restoration process for a stopped design.

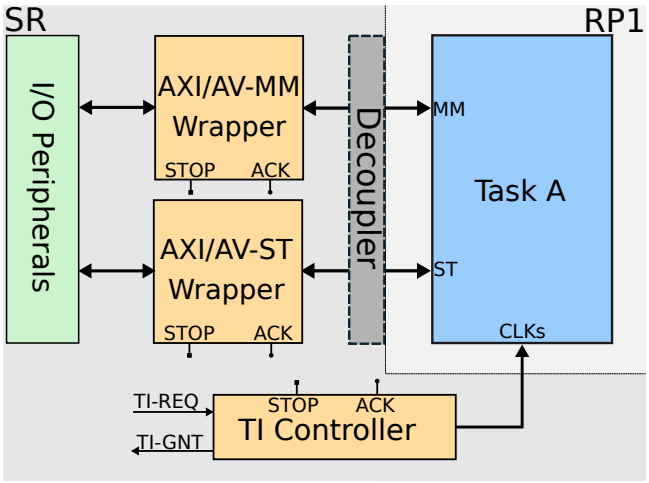


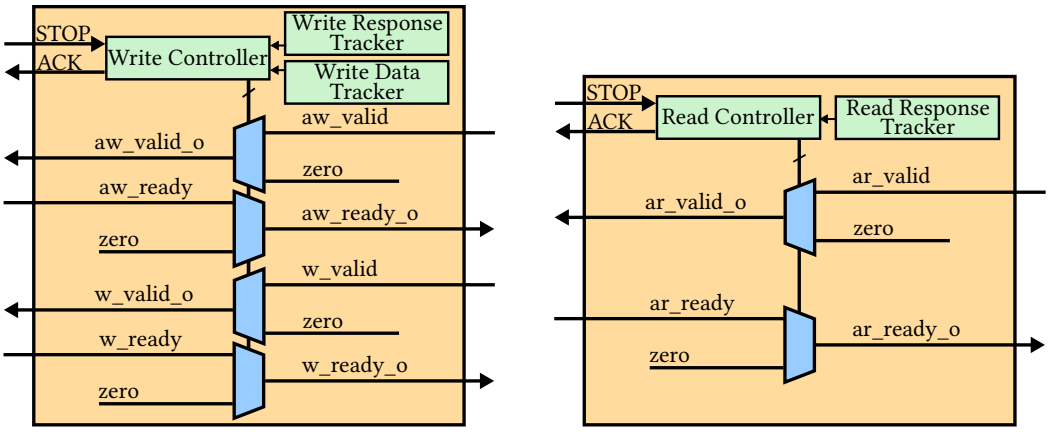
Figure 3.6: TI wrappers and TI controller in the proposed system.

It must guarantee that the design state is stable for a number of clock cycles equal to or greater than the largest multi-cycle value on any register in the clock domain. This ensures the restored data has propagated through any combinational logic (possibly requiring multiple cycles) and is ready for capture at the destination register when the design resumes. As the actual state restoration process takes thousands of cycles, adding a small number of extra cycles at the end of the process has a negligible impact on the time to restore a state in hardware. It is worth noting that these paths are not common in modern FPGA designs.

3.5 Task Wrappers for Hazard Avoidance

3.5.1 Overview

In this section, we propose task interruption (TI) wrappers that can be placed around a task to make it safely stoppable, along with a task interruption (TI) controller that orchestrates their behavior. TI wrappers together with the TI controller provide an implementation for the design rules proposed in Section 3.4. Thus, by using the proposed system shown in Figure 3.6, all hazards arising from interrupting any task are avoided, and designs can be safely checkpointed or context switched. TI wrappers are interface-dependent and each wrapper is responsible for stopping the associated task interface. They support the industry-standard request-response and streaming interfaces used by Xilinx and Intel: AXI/Avalon memory-mapped, and AXI/Avalon streaming interfaces. The TI wrappers proposed in this section can be used only for designs that have these standard interfaces. Given the high and increasing use of these standards (in IP cores, within designs and in system integration tools), we believe supporting these standard interfaces will be sufficient for most designs. Should designs require other interfaces, the general wrapper techniques we develop could be applied to those interfaces as well. TI wrappers have three states: ACTIVE, PAUSE, and SHUTDOWN. In the ACTIVE state, the wrappers are totally transparent; all transactions pass through the wrappers seamlessly. Once a stop request is asserted, the wrappers go to the PAUSE state in which they apply the design rules to safely stop the associated task interfaces. Once stopped, the wrappers go into the SHUTDOWN state and acknowledge the stop request.



(a) Write Channel (b) Read Channel

Figure 3.7: A simplified view of the AXI memory-mapped wrapper.

The TI controller also has three states: IDLE, WAIT, and SHUTDOWN. It takes a TI request as an input, and once a TI request is asserted, it goes to the WAIT state. During the WAIT state, it generates stop requests to all TI wrappers and waits for the ACKs of these requests. Once all stop ACKs are received, indicating all task interfaces are stopped, the TI controller stops the entire task by disabling the task clocks (using clock-enables) according to the procedure described in design rule (3.4.5). Once the task is completely stopped, the TI controller goes into the SHUTDOWN state. In this state, the TI controller activates the decoupler and grants the TI request. The task state can now be safely read out, or overwritten, or the task can be replaced by a new task via partial reconfiguration. The TI controller disables the decoupler once the state readout, writeback or PR process is completed.

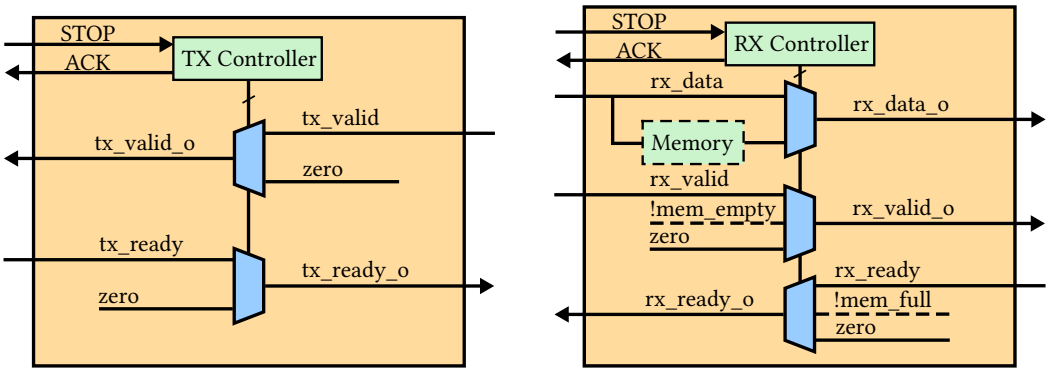
To enable safe task interruption, a designer should add the TI controller and TI wrappers to the design as shown in Figure 3.6. For each interface to/from the task, the designer should instantiate the corresponding (AXI/AValon memory-mapped/streaming) wrapper and connect the task side of the wrapper to the task and the other side to the module interfacing with the task, as shown in Figure 3.6. Since the wrappers have standard interfaces, they can also be easily inserted in the system using Xilinx Vivado IP Integrator or Intel Platform Designer. In virtualized FPGAs, the TI wrappers and the TI controller should be part of the shell, and hence the shell user does not have to do any additional work to make their tasks safely interruptible.

3.5.2 Request-Response Interface Wrappers

We design two request-response interface wrappers: an AXI memory-mapped (AXI-MM) wrapper and an Avalon memory-mapped (AV-MM) wrapper. The AXI-MM wrapper is a wrapper placed around the AXI memory-mapped interface to safely stop the interface. It supports outstanding read and write requests. It is divided into two independent wrappers: one for the write channel and one for the read channel as shown in Figure 3.7. The figure shows a simplified view of the AXI-MM wrapper illustrating its main components and relevant interface signals (i.e., signals that are not shown are passed through the wrapper without modification). To safely stop the write channels, according to design rule (3.4.1) all in-flight transactions should be allowed to complete, and according to design

rule (3.4.2) the issuing of new transactions (i.e., write requests) should be prevented. Thus, once the wrapper goes into the PAUSE state, it stops the write address (AW) channel in order to stop issuing new requests. To stop an AXI channel, the wrapper overrides the *valid* and *ready* signals of the channel. This is done by using a MUX that, instead of passing these signals, forces them to zero as shown in Figure 3.7, so that in the next cycle, the master of the channel sees that the slave is not ready to receive data (i.e., write request), and the slave sees that the master has no valid data on the channel. The write data (W) channel is allowed to send the data of the in-flight write requests. Since AXI supports transmission of non-aligned address and data, the W channel should be prevented from writing excess data (i.e., data of upcoming write requests). Otherwise, the excess data would be stored to the address of the first write request made by the replacing task. To prevent that, the wrapper contains a tracker (Write Data Tracker) that counts the number of pending data beats to be written. The tracker increments when a write request is issued and decrements when the last data beat of a write request is sent. Once the counter reaches zero, the W channel is also stopped. The task may have already sent some excess data before the stop request was asserted, which means that the counter is negative; in this case, the AW channel is not immediately stopped, and extra write requests are allowed to be issued to consume the already sent excess data. Since a transaction is considered complete only after the entire response is received, another tracker (Write Response Tracker) is used to count the number of write requests and responses. Once the responses of all issued requests are received, the wrapper acknowledges the write stop request indicating that the interface write channels have been safely stopped. To stop the read channels, the wrapper stops the read address (AR) channel once a read stop request is asserted by overriding the AR *valid* and *ready* signals. The wrapper also tracks the read requests and read responses (Read Response Tracker), waits until all read responses come back, and then acknowledges the read stop request.

The AV-MM wrapper is placed around the Avalon memory-mapped interface, and is similar to the AXI-MM wrapper shown in Figure 3.7 with some differences we detail below. Unlike AXI, no outstanding write transactions are allowed on an Avalon interface, so stopping the Avalon write channel is simpler. The wrapper tracks the total number of data beats to be written (i.e., burst length) and the number of completed write data beats. When the wrapper is in the PAUSE state, it waits until all the data beats are written, and then stops the write channel. The write channel is stopped by overriding the *wait-request* signal and the *write* signal. The *wait-request* is forced high so that the master sees that the slave is not ready to receive another write request, and the *write* signal is forced low so that the slave sees that there is no active write request. However, if the *write* and *wait-request* signals are both high, the wrapper waits until the *wait-request* goes low, even for one cycle, before overriding the *write* signal low. This is crucial since the Avalon standard states that no change on the channel signals should occur during an asserted *wait-request*. The write response is optional in Avalon interfaces. Thus, if the interface does not require a write response, the wrapper acknowledges the write stop request immediately after the write channel is stopped. The AV-MM wrapper supports pipelined and burst reads. To prevent the issuing of new read requests, the wrapper, once in PAUSE state, overrides the *read* signal by forcing it low, and the *wait-request* signal by forcing it high. The wrapper waits for all burst data of all the pipelined read requests to arrive and then goes into the SHUTDOWN state.



(a) TX Path (b) RX Path

Figure 3.8: A simplified view of the AXI/Avalon streaming wrapper. Dashed signals and blocks are needed only if the task does not tolerate packet loss.

3.5.3 Streaming Interface Wrappers

The AXI/Avalon streaming wrapper, shown in Figure 3.8, is used to stop the AXI/Avalon TX and RX streaming interfaces. When a TX stop request is asserted, the TX wrapper goes into the PAUSE state. If there is a transmission in progress, the wrapper follows design rule (3.4.1) by waiting until the entire packet is sent by monitoring the *last* (AXI) or *EOP* (Avalon) signal, and then stopping the TX channel. The TX channel is stopped by overriding the TX *ready* and *valid* signals, similarly to the AXI-MM case. Finally, the wrapper acknowledges the TX stop request and goes into the SHUTDOWN state. Similarly, during the PAUSE state the RX wrapper waits until an entire packet is received if there is an active reception. It then stops the RX channel by overriding the RX *ready* and *valid* signals, and acknowledges the RX stop request.

According to design rule (3.4.3), ingress packets to a swapped-out task should either be dropped or buffered. Thus, during the ACTIVE state, the wrapper compares the destination of ingress packets with the currently active task; if they match, the wrapper will be transparent, and the packets pass seamlessly. If the destination matches one of the swapped-out tasks, the wrapper will either drop the packet or buffer it into an internal memory as shown in dashed in Figure 3.8 based on its configuration. The wrapper should be configured to drop packets for tasks that tolerate packet loss (i.e., non-guaranteed delivery) or to buffer packets for tasks that do not tolerate packet loss (i.e., guaranteed delivery). Dropping the packet is done by overriding the *valid* signal by forcing it low so that the active task does not receive the packet, and overriding the *ready* signal by forcing it high so that the sender sees that the packet is processed in order to avoid system lockup. Buffering the packet is done by overriding the *valid* signal by forcing it low so that the active task does not receive the packet, and connecting the *ready* signal going to the sender to the *!full* signal of the internal memory. The internal memory stores the incoming packets along with any other important signals in the streaming interface.

If we wish only to checkpoint and restore the state of a single task, or context switch between only two tasks, then a FIFO can be used as an internal memory. To support context switching between several tasks in the same reconfigurable partition, a FIFO for each task would be needed. Instead, we can use a single shared memory, and keep track of the start address and the count of buffered data for each task. If this memory ever becomes full, an exception can be raised in a manner similar

Table 3.5: Simulated Hazard Summary: original basic test designs.

		Data loss	Data corrupt	Task deadlock	System lockup
AXI-MM design	read	✓	✓	✓	✓
	write	✓	✓	✓	✓
AV-MM design	read	✓	✓	✓	-
	write	✓	✓	-	✓
AXI-ST design	send	✓	✓	-	✓
	receive	✓	✓	✓	✓
AV-ST design	send	✓	✓	-	✓
	receive	✓	✓	✓	✓
Double-pumped DSP design		✓	✓	-	-

to CPU exceptions (e.g., page faults), where the task is stopped using the same stop procedure and then a memory transfer is performed from the internal memory to the external memory or a disk. The size of the internal memory is user-defined as discussed in Section 3.6. Generally, there should not be a need for a large internal memory, since an actual task should not be receiving a large amount of data without sending a request (e.g., HTTP request) or sending a response (e.g., acknowledgment of the data being received).

Draining the stored packets is performed immediately after switching to another task; once a task is swapped-in, the wrapper checks if there are buffered packets destined for this task and drains these packets. During draining, the RX channel is back-pressured by deasserting the *ready* signal, and the streaming interface of the task is driven by the wrapper. Once all the packets of the active task are drained, the wrapper again becomes transparent, and the task can receive packets directly from the RX path.

3.6 Results

3.6.1 Basic Test Designs

We test the proposed system that consists of task wrappers and a TI controller by incorporating it in the designs described in Section 3.3: 1) two designs with reconfigurable tasks that read and write to a memory controller through (a) AXI memory-mapped interfaces or (b) Avalon memory-mapped interfaces; 2) two designs with reconfigurable tasks that send and receive packets over Ethernet through (a) AXI streaming interfaces or (b) Avalon streaming interfaces; and 3) a design with reconfigurable tasks that have multiple related clocks. These designs are simulated again using CSR-SIM to see whether the hazards are resolved. In the simulation, we used the same context switching event timings that caused the aforementioned hazards. After adding the TI controller and task wrappers, the simulation shows that all the hazards are resolved. To further verify the elimination of hazards, the designs are simulated again with extensive test cases in which the context switching events are simulated to occur at all the different combinations of the critical signals of the interface (e.g., *valid*, *ready*), and at different scenarios (e.g., single/burst reads/writes) for the replaced and the replacing tasks. The simulation shows no hazards, and the tasks work normally regardless of the timing in which a context switching request occurs. Table 3.5 summarizes the hazards that originally occurred in each design (a ✓indicates a hazard presence in that design); all the hazards are resolved by using the proposed system.

Table 3.6: The area and the maximum frequency in MHz of the 32-bit memory-mapped and 64-bit streaming TI wrappers.

		LUTs/ALMs	FFs	FMAX
AXI-MM Wrapper	Transparent	32	18	931
	Registered	60	222	902
AV-MM Wrapper	Transparent	57	37	451
	Registered	198	271	455
AXI-ST Wrapper	Transparent	155	155	569
	Registered	161	306	544
AV-ST Wrapper	Transparent	81	29	747
	Registered	141	180	614

3.6.2 Wrapper Overhead

To quantify task wrapper cost, we implemented AXI-MM and AXI-ST wrappers on Xilinx Kintex UltraScale (XCKU040-2), and implemented AV-MM and AV-ST wrappers on Intel Arria 10 (10AX115S2). We implemented two versions of each wrapper, one that is truly transparent to the interconnect when in the ACTIVE state (i.e., it does not add any latency to transactions) but has combinational logic which adds some delay, and a registered version in which the transaction is registered inside the wrapper before proceeding. The latter version adds one-cycle of latency with no bubble cycles (i.e., up to 100% channel bandwidth) by using skid buffers which allow the input data one additional cycle to stop when the output side is not ready [103].

Table 3.6 shows the area, in terms of LUTs and FFs, and the maximum operating frequency of the task wrappers. The AXI-MM and AV-MM wrappers support the maximum burst length and the maximum number of outstanding transactions allowed by the AXI/Avalon standards. The reported results of registered memory-mapped wrappers are for a 32-bit interface. The area overhead of transparent wrappers is independent of interface width. The size of the internal replay memory used in the AXI-ST and AV-ST wrappers is parameterized. The reported results are for streaming wrappers with an internal memory that holds 64 packets of 64 bit-width and utilizes two BRAMs.

The wrappers achieve a high frequency which for some wrappers exceeds the restricted FMAX of the FPGA, and occupy less than 200 LUTs. The transparent memory-mapped wrappers do not have any sequential elements; the reported frequencies are based on the critical path delay from the inputs to the outputs of the wrappers. Registered wrappers have nearly equal or slightly lower maximum operating frequency compared to transparent wrappers due to the added logic (e.g., skid buffers) that eliminates bubble cycles. However, since registered wrappers add a pipeline stage to the interconnects, they have lower timing overhead on the path from/to the wrappers than transparent wrappers. We also implemented the Xilinx AXI shutdown manager (XASM), which can avoid some hazards on AXI memory-mapped interfaces, to compare it with our AXI-MM wrapper. XASM occupies 653 LUTs and 1133 FFs, and achieves a maximum operating frequency of 577 MHz. It is 5-10x bigger than the registered AXI-MM wrapper, and as previously discussed is also insufficient for safe task interruption since it is not intended for checkpointing and context switching purposes. The number of task wrappers that the TI controller manages is also parameterized. A TI controller that manages two task wrappers utilizes less than 10 LUTs and FFs.

We calculated the critical path delay overhead involved in adding TI wrappers to the aforementioned basic test designs. Figure 3.9 shows the timing overhead for the transparent version.

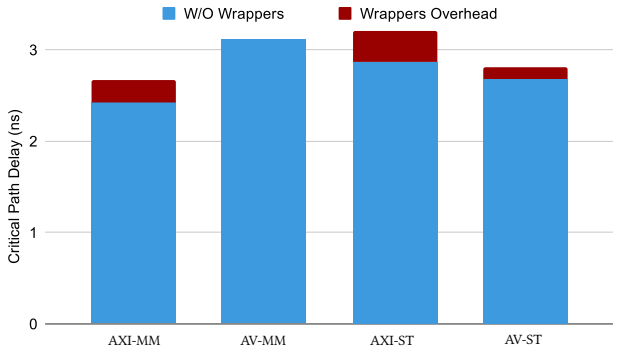


Figure 3.9: The critical path delay in (ns) for the basic test designs showing the timing overhead of the transparent TI wrappers.

Registered task wrappers have no effect on critical path delay but add one-cycle of latency. The transparent wrappers add 0-10% overhead to the critical path delay of the designs. It is worth noting that the interconnects used in the designs have no internal pipeline registers. If pipelined interconnects are used, the critical path delay will no longer be inside the interconnect. In this case, the transparent version also have no effect on the critical path delay.

3.6.3 System Test: Memcached

In the earlier sections of this chapter, we have exhaustively tested all standard interfaces and types of clock relationships, and found solutions to avoid hazards in each case. By composing these solutions, complex systems with many interfaces and many clocks will be safely interruptible by construction. To demonstrate that this is the case on a complex design with many interfaces and clocks, we deploy our solution on a Memcached accelerator system. Memcached is an in-memory key-value store caching system that is widely used in data centers to reduce database load [104]; it is used in the cloud solutions of Facebook, Twitter, Wikipedia and many other companies [105, 104]. We leveraged Xilinx's open-source HLS implementation of the Memcached core in our test system [106]. The Memcached core consists of a request parser, hash table, value store and response formatter. It has an AXI streaming interface connected internally to the request parser and the response formatter for receiving/sending Memcached requests/responses to and from the network. It also has two AXI memory-mapped interfaces connected internally to the hash table and value store (through an AXI data mover IP [107] and a read/write converter) for reading and writing into DDR memory.

To construct the test system which is shown in Figure 3.10, we built a shell that contains a DDR controller and an Ethernet system. The shell has two reconfigurable partitions (RPs) (i.e., role slots) where each RP has an AXI streaming interface connected to the Ethernet System, and two AXI memory-mapped interfaces connected to the DDR controller. Each RP is occupied by a Memcached core: RP1 is occupied by MCD-A and RP2 is occupied by MCD-B. The reason for having two Memcached cores (i.e., tasks) in our test system is to check for system lockup in which stopping one task locks up the entire system. The shell performs network and memory isolation, in which each task has a specific network address, and can access a certain region in the external memory. The shell provides the slots with two main asynchronous clocks: the Ethernet (RX transceiver) clock and the DDR clock. The Memcached core has a clock domain crossing circuit between the AXI data

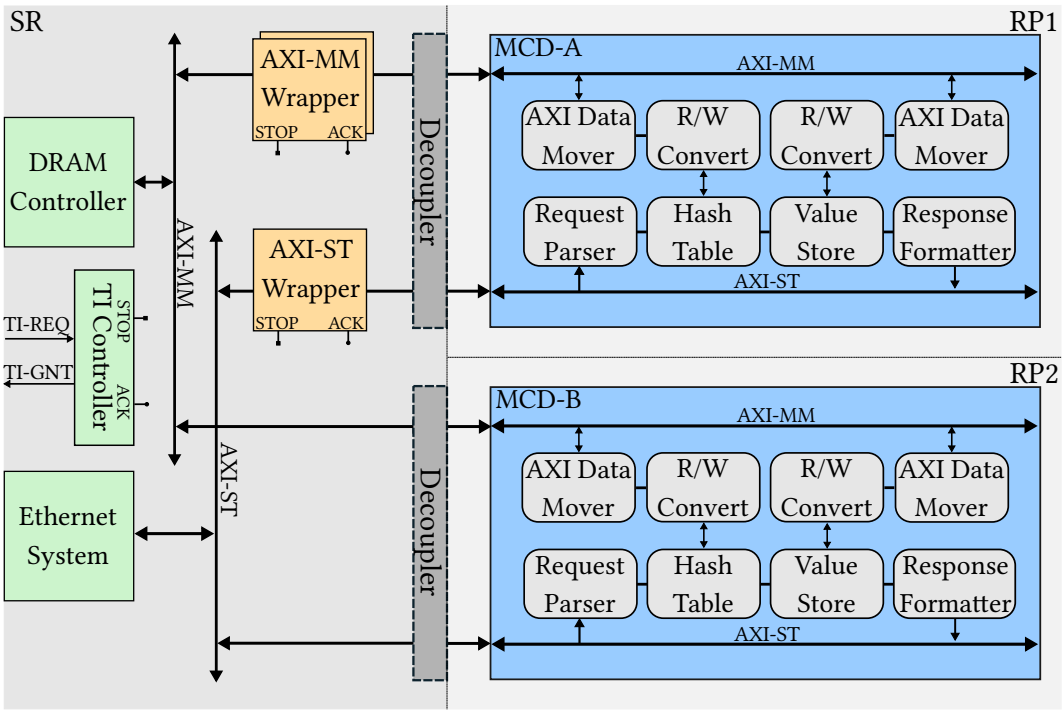


Figure 3.10: Memcached Accelerator System.

mover IP and the hash table and value store blocks. We expanded the Memcached HDL test-bench provided by Xilinx to simulate and test our entire system. The expanded test-bench has two packet drivers (PD-A and PD-B) that send Memcached requests over Ethernet to the two Memcached cores, and two packet monitors (PM-A and PM-B) that compare the received Memcached responses with the golden output. The Memcached requests are combinations of set (store a value in the cache) and get (retrieve a stored value) commands which are the fundamental Memcached commands; the requests have a variety of different lengths.

After testing the system without any task interruption, we used CSR-SIM to simulate the two task interruption scenarios discussed in Section 3.3.1: the stop-resume and the stop-switch-resume scenarios. First we tested the stop-resume scenario. CSR-SIM shows that if one of the Memcached cores is stopped at any time, the system will lock up until the stopped core is resumed again. If the stopping period is long enough, the Ethernet RX buffer overflows, causing data loss for both cores and discrepancy in their output compared to the golden output. The cause of the system lockup is dependent on the interruption point; stopping the core during burst memory reads or writes on the two AXI memory-mapped interfaces or in the middle of sending/receiving a packet over the AXI streaming interface immediately locks up the system. Even if the core is stopped when it is idle, the system also locks up once a request packet destined to that core is received. System lockup is a critical issue as live migrating one of the tasks would stop the other one permanently.

Secondly, we tested the stop-switch-resume scenario by using CSR-SIM to switch one of the Memcached cores (MCD-A) with a third Memcached core (MCD-C) after saving the state of the current core. The third Memcached core has a different network address and accesses a separate region in the DDR. The shell is programmed to pass packets destined to either MCD-A or MCD-C to the streaming interface of RP1. A third packet driver (PD-C) and a packet monitor (PM-C) are

added to the test-bench. When the context switching is requested, the packet driver PD-C starts sending requests, and the packet driver PD-A that sends requests to MCD-A is stopped. After running the system for some time, the MCD-A is switched back, and its saved state is restored. Obviously, more hazards occur in this scenario than in the stop-resume scenario. In fact, without our safe task interruption logic the system never generates a complete correct output no matter what the interruption point was, for the following reasons. First, Memcached requests that had already been sent from the packet driver PD-A are incorrectly received by MCD-C. This results in incorrect Memcached responses for these requests. Moreover, when MCD-A is swapped back, it also generates incorrect output due to the lost requests (i.e., lost set commands). It is worth noting that if the MCD-A requests are back-pressured rather than passing them to MCD-C, a system lockup and RX overflow will occur. On the other hand, if those requests are dropped, MCD-A output will also be incorrect and incomplete. Second, stopping MCD-A during any in-flight transaction on any of its three interfaces causes several problems as we have shown in Section 3.3 including incorrect/incomplete output and even task deadlock in some cases. For example, MCD-A could not resume if it was stopped after sending a read request and before receiving the read response on any one of its two AXI memory mapped interfaces.

To deploy our solution, we added two AXI-MM wrappers on the two AXI memory-mapped interfaces of the RP and the AXI-ST wrapper on the AXI streaming interface of the RP. As shown in Figure 3.10, the wrappers are added only on RP1 interfaces since we are switching MCD-A in/out. The TI controller is also added to the system to control the three added wrappers. Since the Memcached core does not have any related clocks, and the clock domain crossing circuit used for the two asynchronous clocks inside the core does not make any assumptions regarding the clock cycles in one clock domain versus the other, it is safe to stop the two clock domains once the wrappers stop the interfaces. The effort to add these wrappers to the system is quite modest; instantiating the wrappers and connecting them to the Memcached core and the interconnect took a developer less than one hour, and added ∼100 lines of HDL code. In addition, since the TI wrappers and the TI controller are added outside the RP, the wrappers could be part of the shell design, so the shell user does not have to do any additional work to make their tasks safely interruptible. We exhaustively test the system with the wrappers for the two task interruption scenarios according to the methodology discussed in Section 3.3; the simulation is repeated many times (∼20000 simulations, representing over 200 CPU days of simulation time) for all different interruption points. In all simulations, the two Memcached cores generated a complete and correct output compared to the golden output, and no system lockup occurred. Figure 3.11 displays a histogram of the *interruption latency*, which we define as the number of clock cycles from the assertion of the TI request (i.e., interruption point) to the assertion of the TI grant (i.e., the cycle at which the task is completely stopped), over all the simulations. The median interruption latency is 10 cycles with a minimum of 3 cycles and a maximum of 467 cycles. A high interruption latency occurs when a task is interrupted during a long burst read/write or during sending/receiving a long packet. The AXI memory-mapped standard supports burst reads/writes of up to 256 data beats, while the Ethernet system used in the shell supports packets of up to 512 flits.

The area and timing overhead of adding the TI wrappers to the Memcached system is shown in Table 3.7. The transparent wrappers have a negligible area overhead of 0.3%, but a timing overhead of 9% as the critical path is in the multi-master AXI-MM logic and hence passes through the AXI-

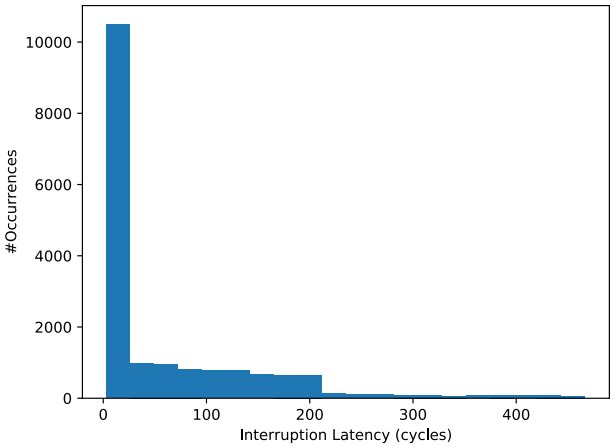


Figure 3.11: Interruption latency histogram of the Memcached accelerator system.

Table 3.7: Overheads to add safe task interruption to the Memcached accelerator system (FMAX in MHz).

	LUTs	FFs	FMAX	Area Overhead	Timing Overhead
Original design	93.2k	141.7k	300	-	-
With transparent wrappers	93.6k	141.9k	272	0.3%	9%
With registered wrappers	94.8k	144.4k	300	1.8%	0%

MM wrappers. This makes the registered AXI-MM wrappers a better choice for this design. We replaced the transparent AXI-MM wrappers with their registered version; this increases the area overhead to a (still small) 1.8%, but brings the timing overhead down to 0%.

3.6.4 System Test: AES

We deploy our proposed solution on another complex system: an Advanced Encryption Standard (AES) accelerator system. As shown in Figure 3.12, the system consists of an AES core that interfaces with a memory controller through an AXI memory-mapped interconnect. The AES core reads data from the memory behind the memory controller, encrypts it, and writes the encrypted data back to a separate region of the memory. The AXI memory-mapped interconnect is shared with another core, AES Check, which reads the encrypted data back from the memory, compares it to the expected data, and counts the number of matches.

After testing the system without any task interruption, we used CSR-SIM to simulate hardware checkpointing. There are two scenarios of interest with hardware checkpointing: stop-resume (useful for generating a checkpoint) and stop-switch-resume (useful for restoring a prior checkpoint to the hardware), as discussed in Section 3.3. To simulate both these scenarios, we first interrupt the AES core to save its state and create a checkpoint. We then resume the AES core and after running it for some time, we interrupt the AES core again and restore the saved checkpoint. This procedure is repeated multiple times to cover all possible interruption points during the simulation time (40000 simulations).

CSR-SIM shows that without using the wrappers, multiple task interruption hazards occur. Task interruption hazards that occur during the first interruption result in a checkpoint that cannot be resumed – no matter when the checkpoint is restored, the AES core does not resume properly.

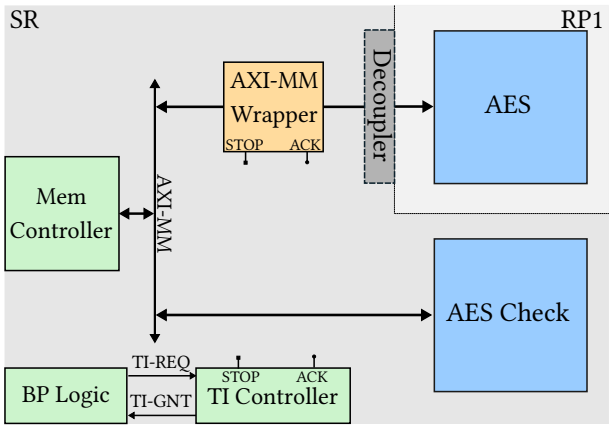


Figure 3.12: AES Accelerator System.

Table 3.8: Overheads to add safe task interruption to the AES accelerator system (FMAX in MHz).

	LUTs	FFs	FMAX	Area Overhead	Timing Overhead
Original design	6.08k	6.77k	250	-	-
With transparent wrappers	6.1k	6.79k	250	0.3%	0%

Moreover, hazards that occur during the second interruption prevent the AES core from resuming properly even if a proper (hazard-free) checkpoint is restored. In addition, in several simulations the AES check core was not able to run, due to a system lockup caused by an unsafe interruption. Due to these various hazards, the system produced incomplete/incorrect output or locked up in 7348 simulations (a failure rate of 18%).

To deploy our solution, we added an AXI-MM wrapper along with the TI controller to the AES system as shown in Figure 3.12. In all 40,000 simulations, which together exhaustively cover the possible interruption points, the AES core was able to resume after restoring the saved checkpoint, and the AES check core was also able to run and all checks passed. The average interruption latency is 4.5 cycles with a minimum of 4 cycles and a maximum of 14 cycles. The logic utilization overhead of adding the transparent AXI-MM wrapper and the TI controller to the AES system is 0.3%, and there is no timing overhead as shown in Table 3.8.

3.7 Summary

As FPGAs move to data centers, support for checkpointing and context switching is highly desirable. However, these features require task interruption, and as we have shown, stopping a task at an arbitrary time causes several hazards. To simulate and identify these hazards, we built CSR-SIM, a context saving and restoring simulator. We then derived design rules that should be followed to achieve safe task interruption. Next, we proposed two versions of task wrappers along with a TI controller that implement these rules. Transparent wrappers have a small footprint (<160 LUTs) but slightly reduce the maximum operating frequency by 0-10%. Registered wrappers add one-cycle of latency with full throughput and have no timing overhead. These wrappers can be placed around any FPGA task including those with multi-cycle I/O transactions and multiple clocks. We showed that deploying these wrappers on a full Memcached design enables safe task interruption regardless

of when the interrupt request occurs, and leads to only a 1.8% area overhead and no timing overhead. These wrappers and rules enable safe context switching and safe hardware checkpointing for a wide variety of FPGA tasks, providing a key underpinning to allow more efficient time-sharing of FPGAs and new debugging flows for both individual FPGAs and cloud-scale deployments.

Chapter 4

StateReveal

4.1 Introduction

In this chapter, we tackle the second challenge of hardware checkpointing: how to create and load *complete* checkpoints. A complete checkpoint consists of the value of *all* state holders inside a task, including task registers and memories. Readback, which is discussed in Section 2.1.3, is a commonly-used technique for accessing the on-chip state of a task that does not require adding special access hardware to the design. It uses the FPGA configuration port to read out of state elements that can be initialized in the configuration bitstream: CLB/ALM registers and BRAM contents. Bitstream manipulation is used in conjunction with readback to restore a checkpoint [32, 108, 7].

As mentioned in Section 2.1.3, the on-chip state of some registers cannot be retrieved nor restored (i.e., initialized) with FPGA readback hardware because they are not accessible to the configuration port. These registers are embedded within hard blocks, such as Xilinx's DSP48 pipeline registers and block RAM (BRAM) input registers, or are within the routing architecture, such as Intel's hyper registers. Hence, saving and restoring the state of these buried registers cannot be accomplished using readback or traditional state access hardware. Consequently, attempting to read back the state of a task that uses a fully registered BRAM, a pipelined DSP block, or a hyper register results in an incomplete checkpoint. This checkpoint cannot be used for restoring the full task state, and hence, the task will not work properly after the checkpoint is loaded, posing a challenge for enabling hardware checkpointing on FPGAs.

In this work, we propose general approaches that allow a complete checkpoint to be captured and loaded when a design contains inaccessible (i.e., buried) state elements. The proposed techniques include modifying the design by adding readable *capture registers* and modifying the readback process using *multi-cycle capture*. We also develop the *StateReveal* tool, which detects inaccessible state elements and automatically inserts the required capture registers. The tool currently supports Xilinx FPGAs and provides an estimation of added hardware cost. We verify the ability to capture and load a complete checkpoint in hardware on different designs that include a variety of buried registers. We show that these techniques have a small timing overhead and moderate area overhead on several benchmark designs.

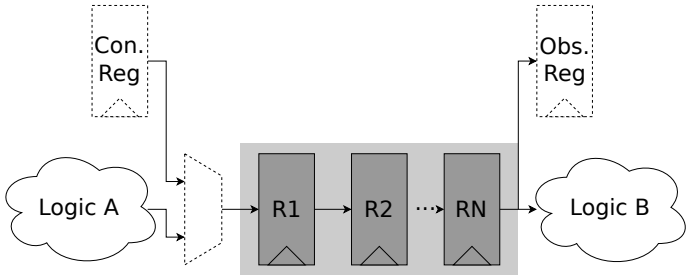


Figure 4.1: Basic Approach. (Dark Gray: inaccessible registers; Dashed: added logic)

4.2 Approaches for Checkpointing a Task with Buried State

Hardware checkpointing involves two steps: creating and loading a checkpoint. To create a checkpoint, a task is stopped, and its current state is read out and saved. To load a checkpoint, a saved state is restored to the task, and the task is resumed. The task must be able to work after the checkpoint is loaded in the same way as if it was never stopped. This is guaranteed if the checkpoint (1) contains the values of *all* state holders within the task and (2) the task is *safely* stopped such that the state is consistent across different clock domains and has no partially complete external I/O transactions, as discussed in Chapter 3. As mentioned earlier, (1) is not currently achievable for the vast majority of designs in current FPGAs since some state holders cannot be saved or restored. In this section, we propose general techniques that guarantee that hardware checkpointing is performed properly in the presence of inaccessible state elements.

4.2.1 Basic Approach

Consider a task that has a chain of N registers whose state is inaccessible (e.g., hyper registers) as shown in Figure 4.1. The output of the chain is connected to some logic denoted by Logic B. When a checkpoint is loaded, the state of these N registers are not restored, and hence, Logic B will be fed by incorrect data for N cycles. This problem can be tackled in different ways. A naive workaround is to change the compilation flow to limit the task to only use registers whose state is accessible (e.g., fabric registers). However, this would reduce both design speed and density significantly.

Instead, the basic approach that we propose is to add some logic to be able to extract the state of the N registers and push the captured state back into them. The added logic can be as simple as one register at the output to add observability and a register with a MUX at the input to add controllability as shown in Figure 4.1. To create a checkpoint, a multi-cycle capture is used in which (1) the task is stopped and its state is read back, and then (2) the task is clocked for one cycle and the observability register is read back. The last step is repeated N times, where N is the number of registers in the chain, until we have extracted all the buried state. To load a checkpoint, a multi-cycle restore is used in which (1) the captured value of the observability register is written back to the controllability register and the design is clocked once to load the first of the N registers. This step is repeated N times until the state of the N registers are restored, and then (2) the remainder of the task state is written back and the task is resumed.

This approach can only be used if the inaccessible state elements are a feed-forward chain of registers with no logic in between, such as interconnect registers, so that the captured state at the

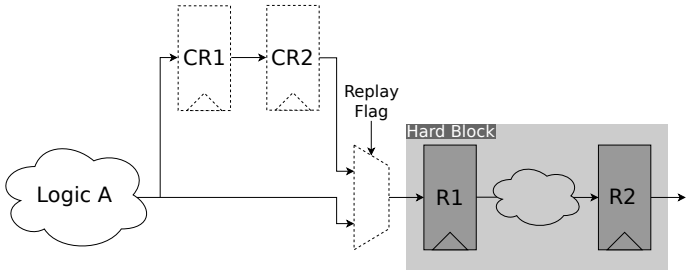


Figure 4.2: Input Capture Approach. (N=2; Dark Gray: inaccessible registers; Dashed: added logic)

output can be fed back at the input. Otherwise, offline processing is required to predict the data that should be fed into the input of the set of inaccessible state elements to restore their inferred state based on the captured output. For complex blocks, inferring the internal state could be infeasible from a limited sequence of captured outputs. To avoid this problematic state inference, the following sections detail variations on this basic approach that can be used with complex hard blocks such as BRAMs and DSP blocks.

4.2.2 Input Capture Approach

In the input capture approach, instead of extracting the buried state, we capture the last N inputs going into the hard block that contains inaccessible state elements. N is the number of inputs that were responsible for forming the current buried state, and it is determined based on the number of buried registers and the connections between them. For hard blocks with only feed-forward paths such as a DSP block with no accumulation, N is the depth of the longest pipeline within the hard block (e.g., the number of pipeline stages enabled by the designer).

To capture the last N inputs, N soft logic capture registers (CR) and a MUX are added at each input port of the hard block as shown in Figure 4.2. When the design is stopped, these registers contain the last N inputs to the block. Thus, this approach requires only one readback. When the checkpoint is loaded, the last N inputs are replayed to "warm up" the internal registers of the hard block (restore their state), and the design is then resumed.

To perform the warming up process, the replay flag that controls the added MUX is set in the checkpoint before loading it. Control logic is also added to the design which resets the replay flag after N cycles. Hence, when the checkpoint is loaded, the input to the hard block will be coming from the capture registers for N cycles. In order not to lose any inputs coming from the conventional input path of the hard block (Logic A), the warming process has to be completed before resuming the design. Hence, the capture registers and the hard block have to be clocked for N cycles before resuming the design. This can be performed by two methods. The first method is to use clock enables to selectively clock the required registers. In the second method, the entire design is clocked for N cycles to warm up the internal state of the hard block, and then a second writeback is performed to reload the checkpoint; the replay flag is not set in the second writeback. The second method can only be used if the writeback hardware in the target FPGA does not reset the buried registers. Unfortunately, the Xilinx UltraScale family we target does reset inaccessible state during writeback, so we must use the first method.

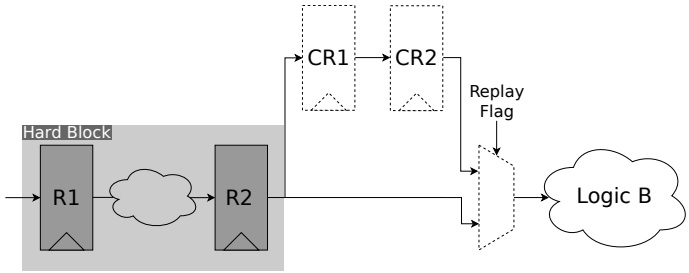


Figure 4.3: Output Capture Approach. (N=2; Dark Gray: inaccessible registers; Dashed: added logic)

4.2.3 Output Capture Approach

In the output capture approach, we do not restore the state of the buried registers. Instead, we ensure that the logic fed by the hard block (Logic B) will receive correct data until the values of the buried registers are overwritten by the upstream logic, which will take N cycles. This is accomplished by capturing the N future outputs that the hard block will generate after the design is stopped, and then outputting them to Logic B when the design is resumed. For hard blocks whose buried state depends only on the last N inputs (e.g., a DSP block with no accumulation), there is no need to restore the state of the buried registers – after N cycles, the buried registers will have correct values regardless their initial state.

To capture the N future outputs, N capture registers (CR) and a MUX are added at each output port of the hard block as shown in Figure 4.3. After stopping the design and reading out its state, the design is clocked for N cycles to fill the capture registers with the N future outputs of the hard block. Then, another readback is performed to read back the capture registers. The checkpoint is then updated with the new values of the capture registers. The replay flag that controls the added MUX is also set in the checkpoint before loading it. Control logic that was added to the design resets the flag after N cycles. When the checkpoint is loaded, the design resumes execution. The capture registers ensure that Logic B receives correct data regardless the current values of the internal registers of the hard block for N cycles. After N cycles, the buried state of the hard block is correct, replay ends, and the output of the hard block is used by downstream Logic B as usual.

For blocks whose buried state depends on more than the last N inputs, extra logic must be added to restore the state of some internal registers such as the DSP accumulator; this extension is discussed in Section 4.3.3.

This output capture approach is only used with hard blocks that have a single pipeline or multiple pipelines with the same depth. This is to ensure that the N outputs that are captured by clocking the design N times after it has been safely stopped depend only on the state of the hard block at the stopping time – new input data (which may be affected by stopping the design interfaces) will not affect the extracted state.

4.3 StateReveal: Automatic Insertion of Capture Registers

In this section, we present StateReveal, a tool that detects the presence of inaccessible state elements in a design and automatically inserts the required capture registers. This section also details how

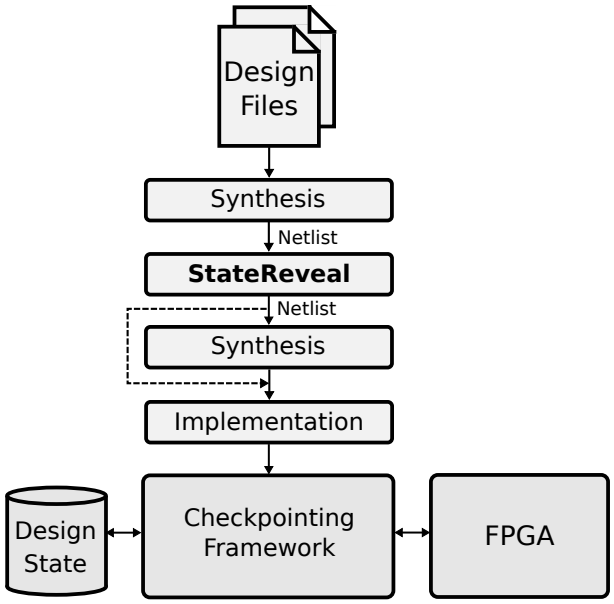


Figure 4.4: StateReveal tool flow. Dashed line represents the direct implementation flow which can be used if only device primitives are added.

we adapt the general approaches proposed in Section 4.2 to support the various configurations of Xilinx's BRAMs and DSP blocks.

4.3.1 Overview

StateReveal[1] is written in python. The tool flow is shown in Figure 4.4. The input of StateReveal is a synthesized netlist that can be generated after Xilinx's Vivado synthesis step with the *write_verilog* command. StateReveal uses Pyverilog to parse the synthesized netlist and generate an Abstract Syntax Tree (AST) [109]. StateReveal then traverses the AST and creates an in-memory hierarchical netlist in which some modules correspond to user-defined modules, but the lowest-level (leaf) modules are all device primitives such as registers and BRAMs. StateReveal then searches for modules that contain hard blocks such as BRAMs and DSP blocks. StateReveal examines the configuration of each hard block and determines if there are inaccessible internal registers. Then, StateReveal selects the appropriate capture approach and adapts it based on the configuration of the hard block as discussed in the following sections. It then modifies the synthesized netlist to add the capture registers and *dont_touch* directives to ensure they will be retained. The modified netlist is then be re-synthesized in Vivado to synthesize the added logic and to allow further synthesis-level optimizations. To avoid the re-synthesis step, StateReveal can be modified to only add device primitives so that the modified netlist can be implemented directly using the post-synthesis (implementation) steps of Xilinx's Vivado flow, as shown in dashed in Figure 4.4.

After that, the checkpointing framework described in Chapter 5 can be used to read out the design state at a user-defined breakpoint. This creates a complete checkpoint, which can then be written back to the FPGA. This enables hardware checkpointing for designs that contain fully registered BRAMs and pipelined DSP blocks, which was not previously feasible.

[1] Available at https://github.com/samehattia/StateMover/tree/master/StateReveal

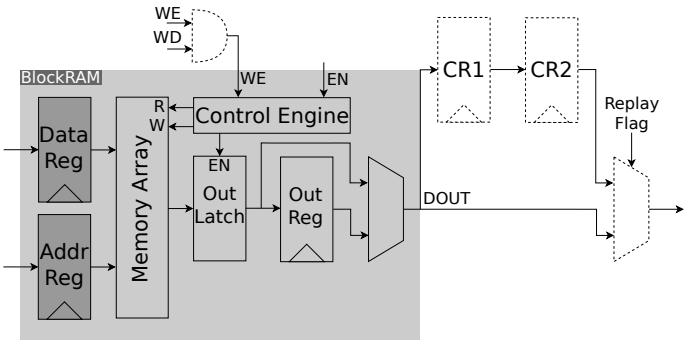


Figure 4.5: A BRAM with output capture approach. (Dark Gray: inaccessible registers; Dashed: added logic)

4.3.2 Block RAMs

The block diagram of a Xilinx BRAM is shown in Figure 4.5. Each BRAM port contains input registers for data and address, and an optional output register. When the output register is used the BRAM is in *register* mode [110]; otherwise it is in *latch* mode and the BRAM output comes directly from the output latch on the same cycle in which the read occurs.

The output latch and output register can be initialized in the bitstream. Hence, their state can be saved and restored, and our checkpointing framework, discussed in Chapter 5, supports that. However, as mentioned earlier, BRAM input registers cannot be saved or restored using readback. Saving and restoring the design state without capturing the BRAM internal registers does not affect the writing operation – the memory contents are always correct. However, it does affect read operations – downstream logic would receive incorrect data for one or two cycles for latch and register mode BRAMs, respectively, if we ignored this issue.

For BRAMs in latch mode, saving and restoring the output latch value obviates the need for capturing the input registers. This is because the output latch stores the last data value read out from the RAM, and it is updated in the same clock cycle the input data and address are registered. Therefore, even though the input registers are not restored, this has no impact as they will be overwritten on the next read request anyway, and we have the correct last read data until that happens.

This workaround cannot be used with BRAMs in register mode because the output latch and the output register share the same location in the configuration architecture. The UltraScale architecture does not support initializing the output latch and the output register to two different values, and only the output register can be read back if the BRAM is in register mode. Hence, StateReveal checks the BRAM mode, and adds soft logic capture registers if the BRAM is in register mode.

For BRAMs in register mode, we use the output capture approach of Section 4.2.3 rather than the input capture approach of Section 4.2.2. The main reason is that the input capture approach can cause a problem if the last two operations before the design is stopped were a read operation followed by a write operation to the same address. In this case, when we warm up the internal registers, the read operation will be replayed, but since the memory content was captured at the checkpoint time (after the write operation was performed), incorrect data will be read into the output register.

Thus, StateReveal adds two capture registers at the output of BRAMs in register mode as shown in Figure 4.5. After the design is stopped and its state is read back, we clock the design for two

additional cycles. After these two cycles, both the output register and the output latch values should be captured in the added capture registers. An additional complexity is that the readback operation (which is reading every row in the BRAM) destroys the output latch value (as implicit reads are occurring). To overcome this problem, we do not read back the BRAM contents in the first readback. Instead, the BRAM contents are read out in the second readback, after the two additional clock edges have moved the BRAM latch and register values to the observable fabric registers. To ensure that the BRAM contents are not changed due to the additional clocking, the BRAM write-enable (WE) signal is ANDed with an added write-disable (WD) signal to disable write operations during the additional clocking. The WD signal should be asserted after the design is safely stopped for the first readback. In the checkpointing framework discussed in Chapter 5, we add interruption logic to the top module of the design which contains the TI controller, presented in Chapter 3, that safely stops the design. Thus, we configure StateReveal to connect the WD signal to the interruption logic so that it can be asserted automatically once the design is stopped.

After the second readback, a complete checkpoint is created by combining the soft logic design state from the first readback with the BRAM contents and capture register values from the second readback. The replay flag is also set in this checkpoint. When the checkpoint is loaded, the captured values of the output latch and register will be fed to the downstream logic until the uninitialized input register (which updates the output latch) and the output register are overwritten by the upstream logic.

StateReveal supports various BRAM configurations such as single-port, simple dual-port (SDP), and true dual-port (TDP). It also supports both 18 Kb and 36 Kb RAM configurations. StateReveal infers the used configuration to know which port is used for reading (A, B, or both) and whether the output register is used in that port in order to determine where to add the capture registers. It also detects the width of the output register and its clock and reset signals in order to correctly generate the added logic. Moreover, if the BRAM is used as a ROM, the write disable logic is not added. StateReveal does not currently support BRAMs in FIFO mode, but the user can still use Xilinx FIFO IPs in soft mode (instead of built-in FIFO mode) in which the FIFO logic (read and write pointers and flags) is implemented on fabric.

4.3.3 DSP Blocks

The block diagram of Xilinx's UltraScale DSP block is shown in Figure 4.6 [111]. The DSP block has more internal registers and modes than BRAMs, which makes proper hardware checkpointing more challenging. The DSP block has four input data ports (A, B, C, and D) and one output data port P, in addition to other I/O data ports connected to adjacent DSP blocks to support DSP cascading. The DSP block has optional pipeline registers at the input ports (A, B, C, D registers), after the pre-adder (AD register), after the multiplier (M register), and after the ALU (P register). The DSP block can be configured to work with no pipeline registers or almost any combination of the internal registers. As mentioned earlier, none of the internal registers of the DSP block can be saved or restored using readback.

Both input and output capture approaches can be used with DSP blocks. The output capture approach is used by default in StateReveal since it is less complex (one output port compared to four possible input ports) and also does not require the usage of clock enables to selectively clock certain registers during the warming up process. However, the input capture approach must be used

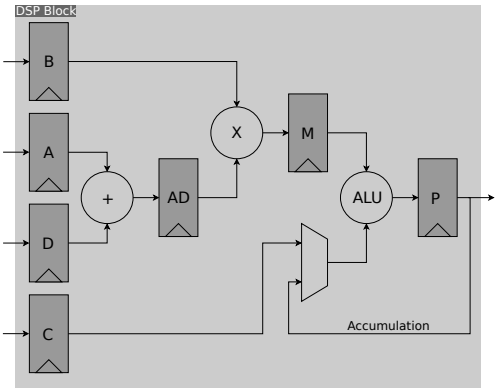


Figure 4.6: Simplified view of Xilinx's DSP block. (Dark Gray: inaccessible registers)

in certain DSP configurations such as when the DSP is configured in cascade mode or when the internal registers have different clock enables, as explained later in this section.

For DSP blocks that do not use the accumulation path, have constant clock enables (tied to VDD or GND), and do not use the I/O cascade ports, StateReveal adds capture registers at the DSP output according to the output capture approach shown in Figure 4.3. To determine the number of required capture registers (N), StateReveal detects the pipeline depth of the DSP block based on the configuration of the internal registers and internal MUXes. To checkpoint a design, the design is stopped and its state is read back, and then we clock the design for N additional cycles. Then, the capture register values are read out and combined with the soft logic design state to create a complete checkpoint. When the checkpoint is loaded, the capture registers will feed the downstream logic with correct outputs for N cycles until the uninitialized internal registers are overwritten by the upstream logic.

DSP Blocks With Accumulation

If the accumulation path is used, using the output capture approach without restoring the value of the accumulator register P is not sufficient; the DSP block will not output correct values after the N cycles with stored outputs. Hence we have to extract the value of P register and restore it, regardless of the used approach. In the output capture approach, the P register value is already captured in the capture registers, so we only need to add extra logic to load the P register with the captured value. If the input capture approach is used, a capture register has to be added at the output to extract the P value in addition to the added capture registers at the input.

There are different methods to load the P register with a certain value. First, we can change the DSP operation mode dynamically during run-time and then use the C input port or the A and B input ports concatenated (A:B) to directly load the P register. Alternatively, we can generate a set of input values that can be fed into the DSP block to set the P register to the required value without changing the operation mode. The latter requires multiple cycles (around 5 cycles) since the width of the accumulator register (48 bits) is larger than the multiplier output (45 bits). This should be performed before resuming the design. In StateReveal, we use the former method in conjunction with the output capture approach; the last value of the P register, which is captured in CR1, is fed back to the A:B ports through a pair of MUXes as shown in Figure 4.7. When the checkpoint is loaded and before resuming the design, the added MUXes pass the captured P value to the A:B

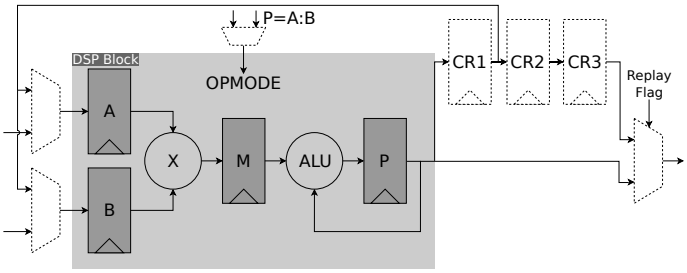


Figure 4.7: A DSP block with accumulation and output capture approach. (Dark Gray: inaccessible registers; Dashed: added logic)

ports. Then, the operation mode is changed temporarily to load the P register with that value and retain it until the other internal registers are overwritten by the upstream logic.

DSP Blocks With Clock Enables

DSP blocks whose internal registers have variable (not tied to VDD or GND) clock enables pose a challenge for proper checkpointing using the output capture approach for the following reasons. The output capture approach depends on capturing N future outputs, where N is the DSP pipeline depth, and then replaying them for N cycles. However, if the clock enables are low at the checkpoint time, the same output will be captured N times, which means that the internal state is not fully captured. Even if we override the clock enables during the capture stage, we still have a problem since we do not know when to output these captured values to the downstream logic. Moreover, if the design sets the clock enables in a certain sequence, overriding them all at the same time can produce incorrect outputs. In addition, we only replay N outputs, and if the DSP clock enables are still low after N cycles, the DSP block will output incorrect values since the internal registers are not yet overwritten by the upstream logic.

Therefore, if the clock enables of the internal registers are not constant, we use the input capture approach of Section 4.2.2 and adapt it to work with clock enables. We assume that all values clocked into the input registers will eventually be used by the later DSP block stages (i.e., those stages will be enabled before the input data is overwritten); otherwise the circuit is both poorly designed (sending data for no purpose) and more difficult to capture. StateReveal adds capture registers at the input and control logic that captures and controls the DSP clock enables. The capture registers have the same clock enables as the DSP input registers (A, B, C, and D). The control logic captures the behavior of the clock enables of other internal registers; for each clock enable (other than the clock enables of the input registers), we capture the number of cycles in which the clock enable is high after the input is registered. This information allows us to properly replay the inputs during the warming up process. We know how much further each input should be propagated down the pipeline because we counted the number of enabled clock edges; this allows us to restore the state of DSP internal registers properly.

DSP Blocks With Cascade Ports

A DSP block in cascade mode uses the PCOUT port to output the value of its P register to an adjacent DSP block through its PCIN port as shown in Figure 4.8. This mode allows efficient implementations of filters and dot products. PCIN/PCOUT ports are not accessible via general-

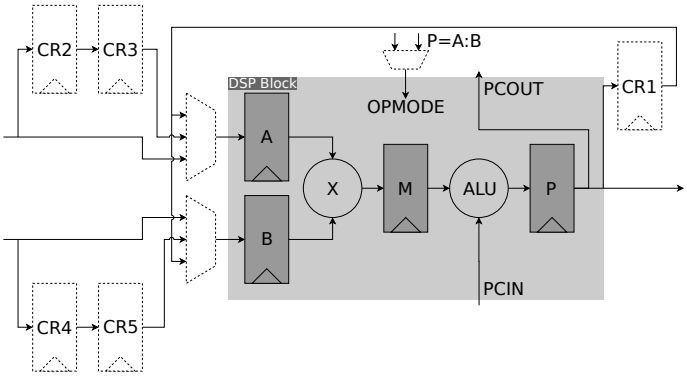


Figure 4.8: A DSP block in cascade mode ($PCOUT = A * B + PCIN$). (Dark Gray: inaccessible registers; Dashed: added logic)

purpose routing resources, which means that we cannot add capture registers at these ports. Thus, we have to treat the cascaded DSP blocks as a single large hard block and then use the proposed input/output capture approaches. However, DSP blocks in cascade mode often form a very long chain. In this case, to use the input/output capture approaches, we must capture a long trace of inputs/outputs, which increases the hardware cost of these approaches.

Instead, we use a hybrid approach in this case. We extract the last value of the P register (a limited output capture) by adding a capture register at the P port, and then use the techniques described in Section 4.3.3 to restore the P register (changing the operation mode to directly load the P register). In addition, we use the input capture approach to restore the other internal registers while retaining the restored P value as shown in Figure 4.8. This is performed for each DSP block in the cascade chain, thereby restoring the entire internal state of the cascaded DSP blocks.

The proposed approaches are also applicable to other DSP block features such as INMODE and OPMODE registers. They can also be applied on DSP blocks of other Xilinx families since the UltraScale DSP block (DSP48E2) is backward compatible with Xilinx's 6 and 7 series DSP block (DSP48E1) and is a superset of Xilinx Virtex-5 DSP block (DSP48E).

4.4 Results

In this section, we verify that we can capture and load a complete checkpoint for a variety of designs with buried hard state using StateReveal. We then report the area and timing overhead of the circuitry added by StateReveal to capture and restore this inaccessible state.

We first use a set of basic designs to verify the ability to capture and restore a complete checkpoint when the checkpointed design contains a BRAM or a DSP block with buried state. The basic designs consist of a hard block (BRAM or DSP block) that outputs data every cycle and circuitry that checks that the output of the hard block is always as expected. The designs are implemented on the Xilinx Kintex UltraScale KCU105 board.

For each design, we have an *original* version, a version that uses *StateReveal* to add the circuitry required to capture and restore inaccessible state, and a *Fabric* version where we manually added *dont_touch* directives on some registers in the original HDL code to stop the synthesis tool from absorbing RTL registers into the hard blocks, thereby forcing the design to use fabric registers instead of hard block internal registers whenever possible (the naive workaround mentioned in Section 4.2).

To verify the StateReveal ability of creating and loading complete checkpoints, we use the checkpointing framework, StateMover, presented in the next chapter. The modifications on the readback and the writeback process to enable multi-cycle capture and to provide the additional clocking which is required in the input and output capture approaches are described in Section 5.4.2.

For each version, we use the checkpointing framework to stop the design, checkpoint it, then reload the checkpoint, and resume the design. The check circuity always flags an error when the original versions of the designs are resumed, which means that the hard blocks output incorrect data for one or more cycles after the resumption as expected. The other two versions work properly when the execution is resumed, indicating that a complete checkpoint is successfully captured and restored.

The basic designs are listed in Table 4.1. The BRAM design contains a fully registered BRAM (input and output registers). The first and second DSP designs contain a DSP block that performs multiplication and has two and four pipeline stages, respectively. The DSP block in DSP design C is used as a multiply-and-accumulate unit and has three pipeline stages. The area overhead (the ratio between the current and the original *area*, which we define as the sum of the number of LUTs and FFs), and the timing overhead (the ratio between the current and the original FMAX) of the StateReveal and Fabric methods are reported in Table 4.1.

The area overhead depends on the number of pipeline stages of the hard block and the data width of its inputs/outputs. For DSP designs, the DSP block is configured with the maximum data width. The StateReveal and Fabric methods add a nearly equal number of FFs except for the BRAM design, but StateReveal adds slightly more LUTs because of the added MUXes. Note that the BRAM in the Fabric version of the BRAM design still uses the BRAM input registers as their use is mandatory; the reported overhead is only for forcing the BRAM output data to be registered using fabric registers instead of the BRAM output registers. That is why the number of added FFs in the StateReveal version of the BRAM design is more than double that of the Fabric version. However, we could leverage the ability to read back the BRAM output latch/register to cut the overhead of StateReveal by half (adding one-level of capture registers instead of two) for this case in the future.

The timing overhead of StateReveal for basic designs is low, from 5% to 9% with a geometric average of 5.5%. The Fabric method results in a much more significant 33% geometric average FMAX reduction; the largest impact is on designs with fully registered DSP blocks, whose performance is cut nearly in half. This result makes sense as the operating frequency of Kintex UltraScale DSP blocks without internal registers is 312 MHz compared to 661 MHz for fully registered ones [112]. This is the main advantage of StateReveal over the Fabric method that limits the design to only use fabric registers. In addition, DSP features such as accumulation cannot be leveraged with the Fabric method – moving the accumulation registers to the fabric forces the addition to use fabric carry chains. Finally, it should be noted that the StateReveal results are generated automatically, while the Fabric method required manual setting of *dont_touch* constraints. The Fabric method is difficult to automate since the directives have to be added before the BRAMs and DSP blocks are inferred in the synthesis step, so automating it would require an iterative method to determine which registers are mapped to hard blocks and hence should be marked as *dont_touch* before a resynthesis of the design.

To further verify StateReveal's ability to properly checkpoint designs with various hard blocks,

we deploy it on complete designs from Xilinx HLS examples [113] and Spiral Hardware [114]. The designs are listed in Table 4.2. We chose these designs because they have several hard blocks used in various configurations. The network sorter and the Reed-Solomon designs have fully registered BRAMs. The BRAMs in the latter have clock enables. The three other designs have both BRAMs and DSP blocks. The hard blocks of the digital-up converter designs have clock enables. The DFT designs have FIR filters that perform multiply-and-accumulate operations in DSP blocks. The floating-point DFT design has DSP blocks cascaded using the dedicated cascade connections. We wrapped the HLS designs (the Reed-Solomon and digital-up converter designs) with test circuitry that generates input data and compares the output with the golden one. The network sorter and the DFT designs are wrapped with a built-in-self-test (BIST) structure which is described in Section 5.6.2. When checkpointing these designs without using StateReveal, the output did not match after the resumption. With StateReveal, the designs work properly after loading the checkpoint.

The area and timing overhead of StateReveal are reported in Table 4.2. The timing overhead is low with a geometric mean of 4%. The area overhead depends on the number of hard blocks in the design. On average, StateReveal adds 25 LUTs and 39 FFs per hard block. The area overhead varies from a 1% to 33%, with a geometric mean of 18%.

4.5 Architecture Recommendation

While StateReveal allows checkpointing of buried state, it adds some timing and area overhead. Given the increasing importance of hardware checkpointing, we recommend a change to the configuration architecture of future FPGAs to remove this cost. As shown in this chapter, designs with fully registered BRAMs and pipelined DSPs cannot be checkpointed without adding registers to capture and restore their buried state. Instead, future FPGAs could change, as much as possible, the configuration architecture to allow reading out and writing back BRAM input registers and DSP pipeline registers via the configuration port.

An UltraScale DSP contains ∼290 pipeline register bits, and a BRAM contains ∼100 input register bits. For a medium-sized FPGA such as UltraScale XCKU040 with 1920 DSPs and 600 BRAMs, this change would add 616K bits to the 128M configuration bits, an increase of less than 0.5%. We believe the small area overhead necessary to add 0.5% more bits to the configuration circuitry is worthwhile to make checkpointing simpler and cheaper in future FPGAs.

However, there might be methodology issues in adding these bits since memories and DSPs are created with ASIC memory compilers and standard cell flows. Moreover, interconnect registers are more numerous, which makes it more expensive and perhaps impractical to include them in the configuration architecture. Thus, an alternative work-around is to harden the capture logic that we proposed in this chapter. This will also reduce the timing overhead and will not require the user to add any additional logic to capture the buried state.

4.6 Summary

FPGAs contain state elements that cannot be read out or written back via the configuration port, such as BRAM input registers, DSP registers, and interconnect registers. As most FPGA designs contain these inaccessible state elements, this poses a challenge for enabling hardware checkpointing,

which is crucial for live migration and FPGA debugging. To widen the set of designs that can be checkpointed, we proposed general approaches that enable capturing and loading a complete checkpoint even when a design contains buried state. We developed StateReveal to automate the design modifications necessary to checkpoint designs with fully registered BRAMs and pipelined DSP blocks. StateReveal adds, on average, 25 LUTs and 39 FFs per hard block to make buried state accessible, and its average timing overhead is less than 5%.

While this work targeted Xilinx FPGAs, the proposed techniques can also be used for BRAMs and DSP blocks in Intel FPGAs. Morevoer, the approach we developed in Section 4.2.1 can be used to extract state from the HyperFlex interconnect registers in Intel's Stratix 10 family with a low area overhead. It would add only two FFs per chain of interconnect registers.

Table 4.1: The area and timing overhead of the StateReveal and Fabric methods for basic designs (FMAX in MHz).

	Original Design			StateReveal Overhead				Fabric Method Overhead			
	LUTs	FFs	FMAX	LUTs	FFs	Norm. Area	Norm. FMAX	LUTs	FFs	Norm. Area	Norm. FMAX
BRAM Design	865	1680	583	4	26	1.01	0.94	0	8	1	0.93
DSP Design A (Multiply)	934	1715	397	22	74	1.04	0.91	0	72	1.03	0.78
DSP Design B (Multiply)	917	1715	607	73	115	1.07	0.93	0	144	1.05	0.51
DSP Design C (Mult-Acc)	993	1966	531	93	187	1.09	0.95	44	120	1.06	0.55
Geomean						**1.05**	**0.945**			**1.035**	**0.67**

Table 4.2: The area and timing overhead of the StateReveal for complete designs (FMAX in MHz).

	Hard Blocks		Original Design			StateReveal Overhead					
	BRAMs	DSPs	LUTs	FFs	FMAX	LUTs	FFs	LUTs/HB	FFs/HB	Norm. Area	Norm. FMAX
Network Sorter	8	0	13319	17003	353	124	275	16	34	1.01	0.93
Reed-Solomon Codec	55	0	3228	4227	351	421	1122	8	20	1.21	1
Digital Up Converter	24	32	4427	8775	323	1752	2006	31	36	1.28	0.92
DFT-64	38	36	6406	9799	382	1642	3683	22	50	1.33	0.96
DFT-64 FP	38	72	50202	71381	346	5268	6043	48	55	1.09	1
Average								**25**	**39**	**1.18**	**0.961**

Chapter 5

StateMover

5.1 Introduction

In this chapter, we seek to combine simulation and hardware execution in a seamless way to enable a novel FPGA debugging flow that has the software-like combination of speed and observability. We present StateMover: an FPGA debugging framework based on hardware checkpointing that allows moving the design state back and forth between an FPGA and simulator, and can create several checkpoints similarly to novel software debuggers [6]. StateMover can safely interrupt a running design and is capable of creating and loading complete design checkpoints even for designs with buried state in registered BRAMs or pipelined DSPs, and with external memories.

StateMover leverages the speed of hardware execution by allowing the design to run at full speed on hardware until the design reaches a point of interest (a user-defined breakpoint or a detected fault). Then, the entire on-chip state of the design is read out and loaded into a simulator, providing not only full observability, but also the ability to simulate the design starting from this checkpoint for further debugging, thereby fast forwarding hours of simulation. By taking periodic design snapshots, the user can load a previous checkpoint into the simulator to trace back the root cause of a bug. In addition, having the on-chip design state loaded on a simulator augments hardware debugging with the ease-of-use of a simulator where the user can navigate through the hierarchy of the design and not only inspect signal values but also modify the value of any state element to perform "what if" tests. The writeback capability of StateMover allows the modified design state to be pushed to the FPGA which enables the implementation of state changes at no cost (i.e., full controllability) and also allows fast forwarding the simulation in uninteresting periods. StateMover can also be used to perform debugging-unrelated tasks such as error injection and correction for soft error mitigation and fault recovery in which a fault-free checkpoint is written back.

This chapter is organized as follows. Section 5.2 introduces the StateMover checkpointing and debugging framework. We discuss how we safely stop a design based on the techniques of Chapter 3 in Section 5.3. The implementation details of how we read out and write back the design state from the hardware and the simulator are given in Sections 5.4 and 5.5, respectively. StateMover is evaluated and tested on various designs to demonstrate its utility and quantify its performance in Section 5.6. Section 5.7 summarizes the chapter.

5.2 System Overview

StateMover[1] is an FPGA debugging framework that combines simulation and hardware execution in a seamless way. It is based on hardware checkpointing, which is a technique in which a snapshot of the state of a design is taken and stored so that when loaded back, the design can continue from this state. StateMover allows moving the design state back and forth between a simulator and an FPGA, which we believe enables a novel debugging flow that has a software-like combination of observability and speed. Users are no longer required to sacrifice speed for full visibility or sacrifice on-chip resources to increase visibility.

StateMover has the following features. First, it can safely interrupt a design by bringing it into a state such that when loaded again, the design can continue from exactly where it left off, without any data loss or deadlocks. This includes designs that have multiple clocks and multi-cycle I/O interfaces. Second, StateMover can read out the entire on-chip state of a design and load it into a simulator in which the user can inspect all the signal values and simulate the design starting from this state. Being able to simulate the design starting from an on-chip state shows how the design behaves and is more powerful than readback, which only retrieves a single value for each signal, on its own; the simulation provides the user with a waveform that is similar to the signal history waveform provided by trace-based debugging techniques but with full visibility and without the hardware cost of trace buffers. Simulating the design to construct the waveform is much faster than the traditional method used by previous readback-based techniques: single-stepping the on-chip design and initiating a readback multiple times. Third, StateMover can extract the design state from a simulator and write it back to an FPGA, which not only speeds up the simulation by fast forwarding uninteresting periods, but also allows full controllability of the on-chip state of the design allowing state changes and "what if" tests.

The tool flow of StateMover is shown in Figure 5.1. First, the designer has to add interruption logic (IL) modules that we provide to allow safe interruption of the task based on the rules of Chapter 3. These IL modules, which are discussed in Section 5.3, are added *outside* the task, have a small area and do not affect the operating frequency of the design. We also provide two Verilog procedures (ILC) that are added into the designer's test-bench to control the IL during simulation. Next, the design files are passed through Xilinx's Vivado implementation flow. For designs that contain fully registered BRAMs and pipelined DSPs, extra logic has to be inserted in the design to add observability and controllability of the inaccessible state inside these hard blocks. The insertion process is completely automated by the StateReveal tool, as discussed in Section 5.4.2, and is performed after the design is synthesized. After the implementation step is complete, an additional script (SM Extract) we wrote as part of StateMover is run to extract additional information from the design and generate the bitstream files. The design is now ready to run in simulation and in hardware. The designer starts a simulation augmented with StateMover's Context Save and Restore Simulator (CSR-SIM), and opens the StateMover (SM) console, which programs and controls the FPGA. The simulator (augmented with CSR-SIM) and the SM console together form the user interface of StateMover. They provide commands that allow the design state to be extracted from the simulator and from the hardware, and similarly loaded into the hardware and simulator. The designer can run the simulation to a certain point and then continue the execution on hardware.

[1] Available at https://github.com/samehattia/StateMover

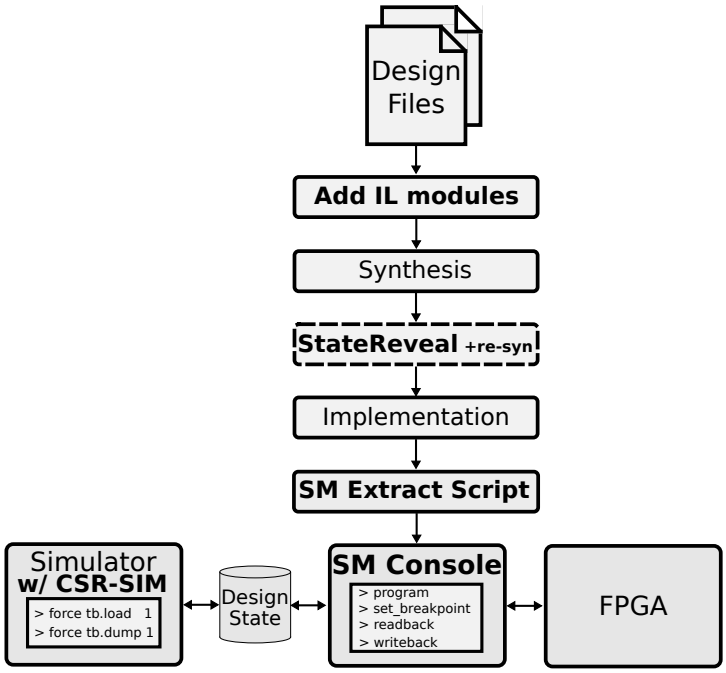


Figure 5.1: StateMover Flow. Dashed: only needed if the task contains fully registered BRAMs or pipelined DSPs.

This is performed by triggering the *dump* signal in the simulator using the *force* command at that point, and then executing the *writeback* command on the SM console. The designer can also stop the hardware execution at a certain point, then move the on-chip state of the task to the simulator, and continue the execution on the simulator with full visibility. This is performed by first using the *set_breakpoint* command in the SM console to stop the design. After the design is stopped, the designer executes the *readback* command in the console to retrieve the state from the FPGA, and then triggers the *load* signal in the simulator to load the extracted state.

5.3 Safe Interruption

To be able to capture and restore a consistent state, we provide interruption logic (IL) that has to be added to the design to safely stop it. The IL consists of breakpoint (BP) logic and a task interruption (TI) controller as shown in Figure 5.2. It interacts with the SM console through Xilinx virtual I/Os (VIOs) over JTAG.

The BP logic controls when the design is going to be interrupted. The design is run normally at full speed until a breakpoint is reached. Depending on the use case, various breakpoint logic can be implemented. Since StateMover supports partial reconfiguration, the breakpoint logic can also be implemented in a separate reconfigurable region so that it can be dynamically reconfigured [37]. In the current setup, we use a counter that starts counting after the design is reset, and the value of the counter is compared to the value set by the user from the SM console using the *set_breakpoint* command. If they match, the breakpoint logic interrupts the design through the TI controller. This provides a simple but effective way to checkpoint the design at any point of interest. The provided BP logic is useful for creating periodic checkpoints. Users can also write more complex BP logic

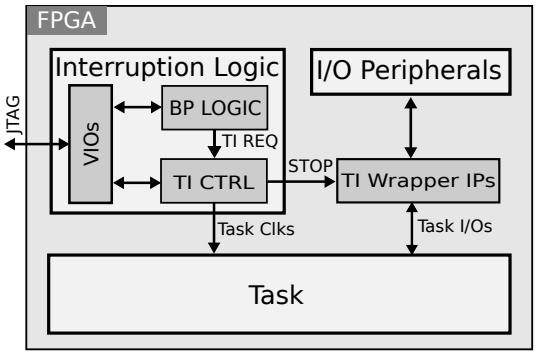


Figure 5.2: Interruption Logic

(e.g., conditional breakpoint logic) based on the targeted design.

The TI controller, which is presented in Section 3.5, takes a TI request as an input, and after it safely stops the task, it grants that request. In simple designs that have one clock domain and do not have multi-cycle I/O interfaces, task interruption is performed by deasserting the *clk_en* of the BUFG that drives the clock to the design. This is the method used in most prior readback-based debugging frameworks for stopping the design. However, StateMover supports debugging of complex designs that have multiple clocks and multi-cycle I/O interfaces, and stopping these designs at an arbitrary point can cause several hazards such as data loss and deadlocks. StateMover also supports debugging a specific task in a complex system, so we have to guarantee that other parts of the system are not affected by stopping that task. To achieve this, *safe* task interruption is performed according to the techniques in Chapter 3, by using TI wrappers which are controlled by the TI controller, and together the controller and wrappers provide an implementation for a set of design rules that should be followed to achieve safe task interruption.

TI wrappers are placed on the multi-cycle I/O interfaces of a task, and when a TI request is asserted, the TI controller sends a stop request to all the TI wrappers. Once the TI wrapper receives the stop request, it prevents any new transactions from being issued, and waits for the in-flight transactions to complete. This ensures that the task state is confined inside the task borders (i.e., there is no essential state in the I/O controllers or other interfaces to the task). TI wrappers support the industry-standard interfaces used by Xilinx and Intel: AXI/Avalon memory-mapped, and AXI/Avalon streaming interfaces.

We integrated the TI wrappers with the decoupler (shown in Figure 3.6), which is needed if partial configuration is used, into *parameterized* IPs (denoted as TI wrapper IPs). These IPs can be easily inserted in the system using Xilinx Vivado IP integrator. They add very small area overhead as detailed in Section 3.6 and Section 5.6. Once the TI controller sees that all TI wrapper IPs have acknowledged the stop request (indicating all interfaces have safely stopped), it disables the clocks of the task so that the task state can be read out, as detailed in Section 5.4.

For designs with multi-cycle interfaces, it can take a few cycles to stop the design after the breakpoint is reached if there are in-flight transactions on the multi-cycle interfaces. If the user needs to examine the design state at an exact cycle, they can load, using StateMover, an earlier checkpoint into a simulator and simulate the design until this cycle. This shows another advantage for frameworks that can move the design state between an FPGA and a simulator compared to other readback-based frameworks that just use readback for signal inspection at certain points.

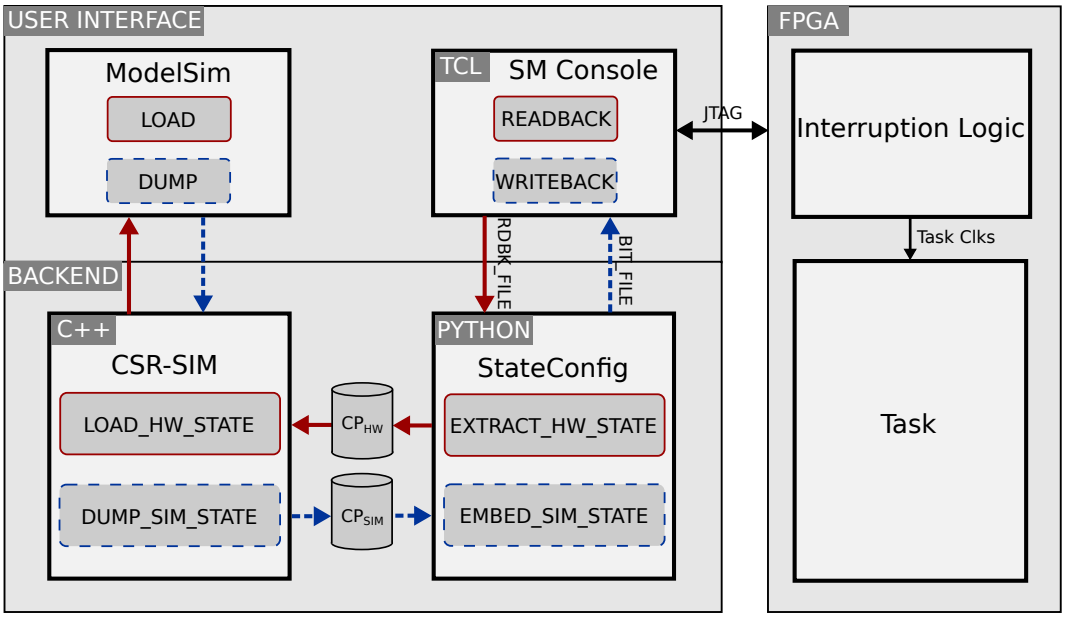


Figure 5.3: StateMover Infrastructure. (Dashed blue: ModelSim to FPGA path; Solid red: FPGA to ModelSim path)

5.4 Reading and Writing Hardware State

In this section, we discuss how StateMover reads out the hardware state of a stopped design, and how it writes a design checkpoint back to an FPGA. Reading and writing the hardware state is performed internally using StateConfig (shown in Figure 5.3) which is written in Python. We divide the hardware state into three categories: accessible state, inaccessible buried state, and external state. Accessible state can be read out and written back through the configuration port, and hence, does not require the insertion of any additional state access hardware. It includes CLB registers, logic blocks used as memories (LUTRAM), and BRAM contents. Designs with pipelined DSPs and fully registered BRAMs contain buried state that is not accessible through the configuration port, and thus require the addition of extra logic to be able to capture and load the *entire* design state as detailed in Chapter 4. The contents of the off-chip memory addressed by the design should also be considered as part of the design state, and hence this external state should be read out and written back as well. The SM console provides the *readback* and *writeback* commands to read out and write back the design state. These commands always capture the accessible state, and can be invoked with options to save and restore the buried state and the external state as well, as detailed in the following sections.

5.4.1 Accessible State

Reading the Hardware State

When the *readback* command is invoked, StateMover reads out the device configuration frames and writes them to a readback file. To speed up the readback process, StateMover supports partial readback in which only configuration frames related to the device region in which the design-under-test (task) is placed are read out. The design state is then extracted from the configuration frames

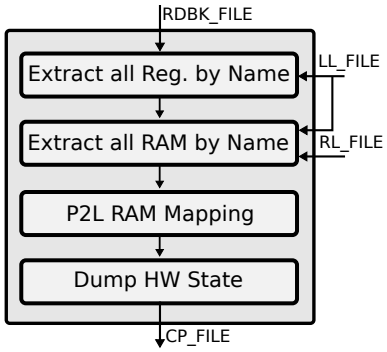


Figure 5.4: Extracting the hardware state.

by using the information in the logic location (LL) and RAM location (RL) files, which are parsed during SM console initialization. The LL file, which is generated by Vivado during the *write bitstream* phase, contains information about the location of state element bits in the configuration frames. It also contains the names of the design nets associated with these state elements, but the net name is shown only for CLB registers. The RL file, which is generated by our SM Extract script, contains the names and the placement information of logic blocks used as memories (LUTRAMs and SRLs) and block RAMs. StateMover uses the information in the LL and RL files to extract the value of each state element from the readback file as shown in Figure 5.4. Then, it dumps the name of each state element along with its extracted value in the checkpoint (CP) file. The resulting CP file can be loaded into a simulator using CSR-SIM as described in Section 5.5, or could be loaded back into hardware later to restore state, as detailed below.

Writing the Hardware State

When the designer invokes the *writeback* command in the SM console, StateMover embeds the design state in the configuration frames of the bitstream file so it can be moved to hardware. This design state could have been checkpointed earlier from hardware, or generated from simulation by CSR-SIM as described in Section 5.5. Figure 5.5 shows the design state embedding flow. First, StateMover reads the CP file, to extract state element names and their values. It then uses the information in the LL and RL files to find each state element's location in the bitstream. Next, it performs bitstream manipulation to overwrite the initial value of the state element with the value written in the CP file. StateMover supports embedding the design state into both full and partial bitstreams. A partial/full reconfiguration is then performed to write this checkpoint back to the FPGA.

Embedding the state in a partial bitstream, which is necessary if we wish to write back the task state without affecting other parts of the system, is more complex and required some reverse engineering. That is because the information in the LL file does not directly specify the location of state elements in the partial bitstream. The location information consists of three parts: the *bit offset* which is an offset calculated from the start of the first configuration frame; the *frame address*; and the *frame offset* which is an offset calculated from the start of this frame. For embedding the state in a full bitstream, StateMover skips the bitstream header and the configuration commands, and then uses the *bit offset* to jump to the location of the state element. In a partial bitstream, we cannot use the *bit offset* to jump to the appropriate configuration bit, as the *bit offset* is valid only

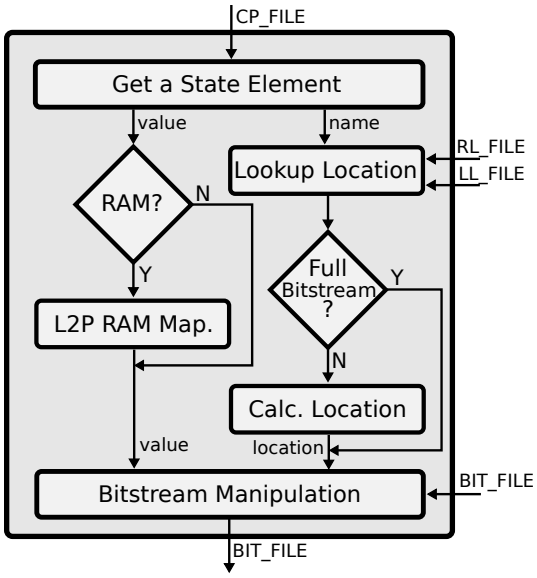


Figure 5.5: Embedding the design state.

for a full bitstream, not a partial one where only the relevant configuration frames are included. Instead, we must compute the bitstream location of the state elements from the *frame address* and *frame offset*. The configuration frame addresses are not continuous, and there is no documentation for how the *frame address* is incremented, and hence, the *frame address* cannot be directly used. Instead, we convert the discontinuous frame address to another continuous form, which we refer to as *frame index*. This conversion is device-dependent because the incrementation of the frame address depends on the number of columns and rows in the device, the number of frames allocated for each resource (e.g., CLB column, DSP column, ...), and the number of hard blocks and their distribution inside the device.

StateMover implements a function that performs the conversion from a *frame address* to a *frame index* on-the-fly by reading a device-specific database which we refer to as the *frame count* database. To know the location of a certain state element in the partial bitstream, StateMover uses this function to convert the frame address of the first configuration frame in the partial bitstream, and the frame address of that state element into frame indices (FI). We then use equation 5.1 to calculate the state element location in the partial bitstream and then modify its state as shown in Figure 5.5.

$$loc(SE) = (FI(SE) - \ FI(start)) * \ bitsPerFrame + frame\_offset \tag{5.1}$$

To create the *frame count* database for a specific FPGA, we implemented a function that takes the number of rows and columns in that FPGA, and an LL file of an FPGA design implemented on that FPGA that has at least a register at each CLB column (e.g., a huge shift register). It then uses the information in the LL file with some reverse engineered information about the number of frames allocated for each resource, which are published in [108], to create the *frame count* database which contains the number of frames in each column. It supports any FPGA from Xilinx's latest UltraScale family and could be extended to other families using the same procedure.

Reading and Writing LUTRAM State

Reading out and writing back the state of LUTRAM require additional steps so that we can create a checkpoint suitable for loading into either FPGA hardware or a simulator. We refer to these steps as physical to logical (P2L) RAM mapping and logical to physical (L2P) RAM mapping, respectively. P2L RAM mapping reconstructs the logical (simulator) representation of the LUTRAM state from the relevant physical bits read out from the FPGA. This is complicated by the fact that the simulator (logical) view of the design primitives does not perfectly match the hardware. The extracted physical bits from the readback file represent the memory content of 6-LUTs but these bits can be used in many ways, so we need to map these bits to the same format expected by the LUTRAM primitives used in simulation. For example, a LUTRAM primitive can describe multiple physical LUTs (CLB-wide) for wide memories. Logic blocks can also be used as shift register LUTs (SRLs), and two SRLs can be packed inside the same physical 6-LUT. Thus, we had to analyze how each type of LUTRAM is mapped to the FPGA and how its value is stored in the primitive model. Then, depending on the primitive type, we reconstruct the LUTRAM state and write it to the CP file. For example, to reconstruct the 16-bit value of an SRL, we first get the 32-bit value of the 5-LUT, which implements that SRL, out of the 64 physical bits representing the entire 6-LUT, and then discard every second bit of that 5-LUT, as the SRL state is stored only in odd bits. Since our CP file stores design state in a logical (simulator-friendly) format, L2P RAM mapping, which is exactly the inverse of the P2L RAM mapping, is performed to reconstruct the memory content of the 6-LUTs when we need to construct a bitstream file to restore hardware state.

Reading and Writing BRAM State

StateMover also saves and restores the data stored in BRAM memories, which are accessible through the configuration port. BRAMs are more than just an SRAM array; however, they also include input registers on the address, data and control signals, and designs can optionally include registers on the data outputs. To completely capture the state of a design, these additional BRAM state elements must be read back. The BRAM input registers are unfortunately not accessible through the configuration port, but the RAM output latch (if output registers are not used) or output registers (if they are used) can be read through the configuration port. If output registers are not used, we can completely capture the state of the BRAM by reading the output latch (i.e., we do not need the input register state). However, if the output registers are used we cannot fully extract nor restore the BRAM state using only configuration readback/writeback. For these more complex BRAM uses, we must use additional techniques to extract the input register *buried state* as detailed in Chapter 4.

Enabling Memory Readback

When partial reconfiguration is enabled in Vivado, we found that the LUTRAMs and BRAMs are no longer readable; the changing LUTRAM and BRAM bits are masked out (i.e., readout values are always zeros). [2] By using some bitstream hacking techniques, we determined that when partial reconfiguration is enabled, specific bits in each configuration frame are set to mask the BRAMs and

[2] UltraScale devices have a global control signal (GLUT_MASK_B) that enables and disables configuration memory masking; however, changing its value does not help since UltraScale devices implement fine grain masking at a resource level [115].

the LUT memory bits if the LUT is configured for dynamic operation (i.e., used as LUTRAM or SRL). To work around this problem, if partial reconfiguration is enabled, StateMover manipulates the bitstream before programming the FPGA to turn the masking bits off in the frames that are associated with the LUTRAMs and the BRAMs that are used by the design. This step enables BRAM and LUTRAM readback and is only required if partial reconfiguration is used.

5.4.2 Buried State

Attempting to read back the state of a task that uses a fully registered BRAM or a pipelined DSP block results in an incomplete design checkpoint due to the buried state in these hard blocks as discussed in Chapter 4. This checkpoint cannot be used for restoring the full task state, and hence, the task will not work properly after the checkpoint is loaded. As the use of these hard blocks is pervasive in FPGA designs, this would limit the set of designs that could be checkpointed and debugged using StateMover if it relied solely on the configuration port. To support a wide range of designs, we integrate our StateReveal tool, presented in Section 4.3 to make buried state accessible; Figure 5.1 shows the entire flow. We also enhance StateMover's readback and writeback processes to support both the multi-cycle capture and the additional clocking that are required by StateReveal.

StateMover's readback process is enhanced to support the multi-cycle capture and the additional clocking required by the output capture approach discussed in Section 4.2.3. The capture registers added by the output capture approach have to be read out after the output capture process is completed. Since (1) this capture process takes N cycles where N is the pipeline depth of the hard block, and (2) each hard block can have different pipeline depths, a variable number of readbacks are required. Since the maximum number of pipeline stages is two for BRAMs, and four for DSP blocks, a maximum of five readbacks is needed: the initial readback (@t0) and four extra readbacks (@t1-@t4). Based on the pipeline depth of the used hard blocks, which is reported by StateReveal, StateMover performs the required number of extra readbacks.

After each readback, StateMover clocks the design for one cycle. For designs with multiple clocks, all clocks that control any buried state are toggled once before each readback. StateReveal appends the pipeline depth of the hard block to the name of the associated capture registers. Based on their name, StateMover knows which capture registers to read at each extra readback. Then, StateMover generates the complete checkpoint file which contains the design state from the first readback and the capture register values from the following readbacks. The replay flags are also set in this checkpoint file. This entire process is performed automatically when the user invokes the readback command in the SM console with the *buried_state* option.

StateMover's writeback process is also modified to provide the additional clocking required by the input capture approach discussed in Section 4.2.2. When the user invokes the writeback command, StateMover writes back the checkpoint, and then toggles the clocks while appropriately setting the clock enables (inserted automatically by StateReveal) on hard blocks and input capture registers to recreate the necessary hard block buried state. After this warming up process is completed, the design is resumed.

5.4.3 External State

Designs that interface with external memories such as DRAM can be safely stopped using the interruption logic discussed in Section 5.3. However, if these designs are checkpointed by saving and restoring only their on-chip state, they may not behave as expected when their checkpoints are loaded. For example, if a design is reading some data from DRAM and this data is written during hardware execution, the design will read incorrect data when the design checkpoint is loaded into a simulator since this written data is not available in the simulation. Thus, we propose that off-chip memory contents accessible by a design should be considered part of the design state, and should be saved and restored along with the on-chip state. In this section, we discuss how StateMover reads out and writes back the contents of off-chip memories.

Memory contents can be transferred between an FPGA and a host over JTAG. JTAG is more convenient for standalone FPGAs, but it is slow. The default JTAG frequency for Xilinx FPGAs is 15 MHz, and hence, a full memory transfer of a 1 GB memory would take more than nine minutes (1.875 MB/s). However, most FPGA designs do not use the entire address space of the external memory. Thus, reading out and writing back only the memory content range accessed by the design can significantly reduce the transfer time. To further speed up the memory transfer, Ethernet or PCIe can be used. Ethernet is more universal since some data centers such as Microsoft Catapult do not provide PCIe connections.

StateMover currently supports external memory transfer over both JTAG and Ethernet. It provides IPs that can be placed at the I/O interface of a task that communicates with external memory. They can be easily inserted in the design using Xilinx Vivado IP integrator in the same way the TI wrapper IPs of Section 5.3 are added. Alternatively, if the task is placed in a partially reconfigurable region, these IPs can be programmed into that region to read or write a checkpoint to DRAM after the design is stopped, thereby avoiding any design size increase. In this case, the DRAM-readback IP is partially reconfigured in after reading back the on-chip design state for a checkpoint, and the DRAM-writeback IP is partially reconfigured in before writing back a checkpoint's on-chip design state.

For memory transfer over JTAG, StateMover uses Xilinx's JTAG-to-AXI IP. This IP allows a connected host to issue AXI transactions over JTAG. When the user invokes the the readback command with the *external_state* option, StateMover issues read transactions to read out the external memory contents accessed by the task. The read out contents are saved to the CP file. Similarly, when the writeback command is invoked with the external_state option, the external memory contents in the CP file are restored to the external memory by issuing write transactions. Each transaction reads or writes a memory chunk of 2 KB. The number of memory chunks and the start address can be configured in the SM console. The user can use these parameters to read and write only the memory contents that are accessible by the design instead of reading/writing the entire memory contents.

For memory transfer over Ethernet, StateMover uses a new IP we wrote: *DRAMMover*. It communicates with the SM console over Ethernet and is inserted in the design as an additional AXI master of both the external memory and Ethernet. To read out the external memory, StateMover sends a packet to the DRAMMover IP that contains the start address and number of memory chunks to read. DRAMMover then reads the specified memory contents and sends them over Ethernet to the SM console. The memory contents are then dumped in the checkpoint file. Similarly, the memory contents are restored by sending them over Ethernet to DRAMMover, which writes them back to

the off-chip memory. A memory transfer of a 1 GB DRAM takes less than 20 seconds.

5.5 Reading and Writing Simulator State

In this section, we discuss how StateMover reads out the simulator state of a stopped design, and how it writes a design checkpoint into a simulator. First, the task that the designer wants to debug using StateMover has to have been implemented through Vivado, and the implemented netlist, which is generated by the SM Extract script, is used for simulation. We simulate a post-implementation netlist so that the hardware and simulation state elements match; this allows register values, LUTRAM contents, and block RAM memory contents to be loaded into the simulator or written back to the device properly, no matter what optimizations were made by Vivado. It is worth noting that the post-implementation netlist may not be as familiar to the designer as the original HDL code due to optimization techniques such as retiming. However, tools can be developed to partially help in mapping back the signal values from the netlist to the original HDL code, but due to optimizations like forward retiming, the mapping might not always have a unique solution. Note that the task does not have to occupy the entire design; StateMover supports debugging of reconfigurable modules inside a larger design.

Next, the designer should start their simulator (e.g., ModelSim or VCS) augmented with CSR-SIM, which is done by specifying it as a PLI/VPI executable that can attached to any Verilog simulator [98]. CSR-SIM is a Context Saving and Restoring Simulator that can read, write, and modify the entire state (context) of a specific task in a design during simulation. CSR-SIM is enhanced (compared to the one presented in Section 3.2) to be aware of the physical (hardware) state as discussed in this section. It is written in C++ and uses PLI/VPI to interface with the HDL simulation of the design. CSR-SIM takes as input the name of the HDL module (i.e., task) for which the user wants to dump (save) and load (restore) state. At the beginning of the HDL simulation, CSR-SIM traverses the design simulation model created by the simulator and creates a list of all the state elements, including registers and memories, inside the specified module and all its sub-modules recursively.

To create this list in a way that enables moving state between the simulator and hardware, we must match each state element in the simulation model with the corresponding hardware element; as a large design contains millions of state elements, this must be done automatically. Algorithm 1 shows the overall method. First, CSR-SIM traverses the simulation netlist by querying the simulator through PLI/VPI in order to find all the registers and memories. However, this actually creates a superset of state elements because some of these registers and memories do not map to physical FFs or memories, even when an implemented netlist is used. The is because the simulation model of the hardware primitives used in the netlist could have some variables defined as registers but they do not map to physical registers. Thus, we perform some extra filtering to precisely identify physical state elements. CSR-SIM has a small database of Xilinx's UltraScale register primitives (e.g., FDRE, FDSE) and memory primitives (e.g., RAM32M, SRL16E, RAMB36E); while we created this list for UltraScale, it will mostly be compatible with other Xilinx families as well, so support for them could be added with minor updates. Before inserting a state element into the state element list, CSR-SIM checks if the parent module of this state element is a register primitive, a LUTRAM or SRL primitive or a block RAM primitive. The algorithm also checks if this state element is the actual

Algorithm 1 CSR-SIM: Create State Element List

```
1: let task_name: Name of the top-level module of the task
2: let reg_prim: Arch block types that map to registers
3: let mem_prim: Arch block types that map to memories
4: state_elements ← φ // Map of (name, simulation node) pairs
5: m ← get_module_by_name(task_name)
6: while m != null do // traverse down the hierarchy
7:    while (reg ← next_register(m)) != null do
8:       if type(m) ∈ reg_prim then
9:          if is_state_holder(reg, type(m)) then
10:            port ← get_output_port(m)
11:            signal ← get_signal_connected(port)
12:            state_elements.insert(name(signal), reg)
13:         end if
14:      else if type(m) ∈ mem_prim then
15:         if is_state_holder(reg, type(m)) then
16:            state_elements.insert(name(reg), reg)
17:         end if
18:      end if
19:   end while
20:   m = next_sub_module(m)
21: end while
```

register/memory that holds the state inside that primitive. For example, in the simulation model of some Xilinx primitives, some parameters that are only used for simulation are defined with the *reg* data type; we exclude these elements. Finally, to match state element names between simulation and hardware, CSR-SIM employs the naming rules used by Vivado as it creates the state element list. The naming depends on the primitive type. For example, the FF name in Xilinx's logic location file, which is used to get the location of FFs in the configuration frames, is actually the name of the net that is connected to the output of the register primitive, rather than the name of the primitive instance. These names are stored in the state element list, and are used for dumping and searching for state elements, and hence allowing a complete hardware state to be loaded into the simulator or written back to the device properly.

The designer can then dump the simulation state or load the hardware state into the simulator by triggering the *dump* and *load* signals, respectively. These signals are defined in the ILC procedures that are added to the designer's test-bench. When the dump signal is triggered, CSR-SIM retrieves the values of the task's state elements from the simulation, and dumps the name and the value of each state element into the CP_{sim} file in a readable format. When the load signal is triggered, CSR-SIM reads the CP_{hw} file which contains state element names along with their on-chip values, and then searches in the state element list for those state elements and overwrites their values in the simulation. The CP_{sim} and CP_{hw} (checkpoint) files have the same format, which allows comparison of the on-chip and simulation design state at a specific point using any *diff* tool. A snippet of the checkpoint file of one of the test designs is shown in Listing 5.1, which shows the value of a register, a LUTRAM, and a shift register LUT (SRL).

Although the value of buried state such as the internal registers of DSP blocks and BRAMs could be directly dumped and loaded by CSR-SIM during simulation, we instead use the techniques in Section 5.4.2 such as multi-cycle capture in order to be able to move the extracted state from

Listing 5.1: Checkpoint file format.

```
ns_bist/ns/.../golden[0] 1
ns_bist/ns/.../mem_reg_14_14/mem aaaaffff
ns_bist/ns/.../X0_reg[14]_srl2/data c3c3
```

the simulator to the FPGA and vice versa. The ILC procedures, that are invoked when *dump* or *load* of the design simulation state is requested, implement these multi-cycle capture and restore techniques, so no designer effort is needed.

For designs that use an external memory such as DRAM, StateMover also dumps and loads the memory contents accessible by the design. However, this process is more complex than accessing the contents of on-chip memories such as BRAMs. This is because the simulation model of a DRAM memory is usually encrypted, which prevents CSR-SIM from directly retrieving or restoring the memory contents. Instead, we leverage the functions that the DRAM model provides to access the memory array. These functions use the DRAM physical address which consists of group, bank, row and column. Thus, we have to perform a logical to physical address translation. When the dump signal is asserted, the memory contents accessible by the design are read out word-by-word and written to the CP_{sim} file. When the load signal is asserted, the external memory contents in the CP_{hw} file are restored in a similar manner.

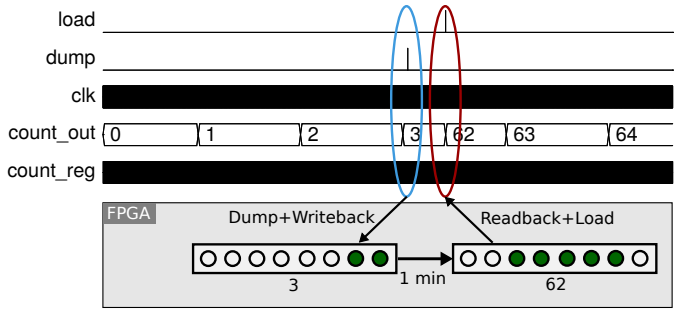
5.6 Results

In this section, we first show a StateMover example session, and then evaluate StateMover on various sets of designs. We report the speed of moving the state of these designs back and forth between a simulator and an FPGA, and quantify the area and performance overhead added to designs. A Xilinx UltraScale KCU105 board is used for all the tests in this section.

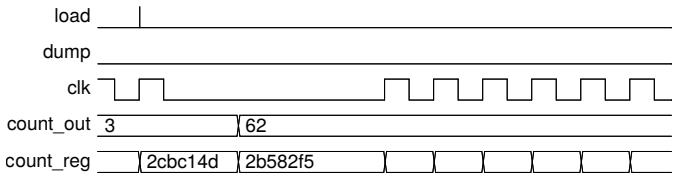
5.6.1 Example Session

In this example session, we use a simple counter design to show StateMover in action. This design contains a counter (*count_out*) that increments every second by using another counter (*count_reg*) as a rate divider, and runs on a 125 MHz clock. We start a ModelSim simulation of this design, and after the counter is incremented by three, which took around 12 minutes in simulation, we trigger the *dump* signal in ModelSim. Next, we invoke the *writeback* command in the SM console. This moves the design state dumped from the simulator to the FPGA; the hardware counter state is connected to the LEDs on the FPGA board so the updated count state is visible as shown in Figure 5.6(a).

After that, we set a breakpoint using the SM console to interrupt the design running on the FPGA after one minute, and after the design is stopped, we invoke the *readback* command and trigger the *load* signal in ModelSim. This seamlessly moves the on-chip state to the simulator, and the simulation can continue starting from this loaded checkpoint as displayed in ModelSim's waveform shown in Figure 5.6(a). The zoomed-in waveform, shown in Figure 5.6(b), shows that whenever the load or dump signal is triggered, the interruption logic (IL), controlled by the ILC procedures, stops the design clocks so the state can be loaded or dumped properly, and then enables the clock again.



(a) Simulation Waveform (top) and Physical LEDs (bottom)



(b) Zoomed-in Simulation Waveform

Figure 5.6: StateMover Example Session: A simple counter.

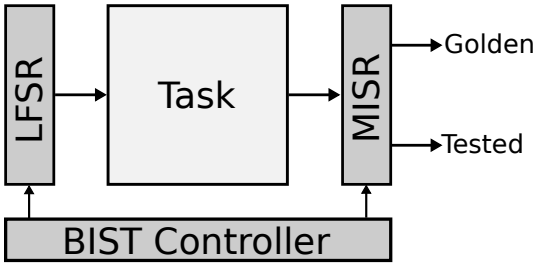


Figure 5.7: The BIST structure used for testing.

This example session shows one of the advantages of StateMover which is the ability to fast forward the simulation in uninteresting periods. For this simple design, four hours of simulation have been saved. This is more crucial for debugging complex designs in which simulation can be 10 million times slower that hardware execution as shown in Section 5.6.2.

5.6.2 BIST Designs

We tested StateMover on several test designs: a crossbar, a FIR filter, and a sorting network. These designs have different *accessible* design elements: the crossbar design uses logic only, the FIR filter uses 64 (unpipelined) hard DSP blocks, and the network sorter uses LUTRAMs. We wrapped these designs with a built-in-self-test (BIST) structure [116] which is shown in Figure 5.7. The BIST structure consists of a linear feedback shift register (LFSR) that feeds the design with pseudo-random inputs and a multiple input signature register (MISR) that is fed by the design outputs.

For each design, we first run the design on the FPGA to generate the golden signature. Then in the second run, we begin running in hardware, interrupt the design and move its state to the simulator, and simulate it to complete generation of the test signature. We also test the other way around, where we generate the golden signature and begin test signature generation in the simulator, interrupt the design in the simulator, and then continue the execution on the FPGA to complete

Table 5.1: The area, checkpoint size and hardware speedup of BIST designs.

	LUTs	FFs	Checkpoint Size	HW/SIM Speed
Crossbar	14,215	11,763	461 KB	1,000,000×
FIR Filter	4,251	6,433	230 KB	700,000×
Network Sorter	13,156	15,506	1.4 MB	570,000×
Crossbar-big	108,841	84,801	4.2 MB	11,000,000×

Table 5.2: The time to read/write hardware state of BIST designs.

	Readback Time	Extract Time	Embed Time	Writeback Time
Crossbar	1.2 s	0.1 s	0.1 s	1 s
FIR Filter	0.4 s	0.05 s	0.03 s	0.4 s
Network Sorter	1.3 s	0.3 s	0.6 s	1 s
Crossbar-big	10 s	0.75 s	0.3 s	9 s

the test signature generation. The test signature matches the golden signature for all the designs, which means that StateMover is able to move the entire design state between the simulator and the FPGA without state corruption, deadlock or other issues.

For this set of designs, since there is no buried state (fully registered BRAMs or pipelined DSPs) nor external interfaces, the area and performance overhead of StateMover is negligible. The area of these designs in terms of number of LUTs and registers are reported in Table 5.1. The interruption logic added to these designs has a small fixed area of 142 LUTs and 362 FFs, which is less than 0.07% of the FPGA resources. Most (90%) of this area actually comes from the Xilinx VIOs that we are using to control the interruption logic (e.g., setting breakpoints) from the SM console. There is no timing overhead since the interruption logic only controls the design clocks (there are no changes within the design task itself) as shown in Figure 5.2.

We measured the speed of moving the design state back and forth between a simulator and an FPGA for these designs and for a larger version of one of the designs to show StateMover's scalability. Reading and writing simulator state takes less than a tenth of a second even for the largest design. Reading the hardware state involves two steps: the configuration *readback* and the *extraction* of the design state out of the readback file. Similarly, writing the hardware state involves *embedding* the design state into a bitstream file and the configuration *writeback* (i.e., re-configuring the FPGA with the modified bitstream). Table 5.2 reports the time required to perform each of these steps.

A full readback of the entire FPGA of an UltraScale KCU105 board, which is a large FPGA that contains 530K logic cells, takes 10 seconds over the JTAG interface. StateMover can perform a partial readback, in which only the relevant configuration frames are read out, which reduces this time significantly as shown in Table 5.2. Similarly, a full writeback takes 9 seconds, and this time can also be reduced by performing a partial writeback (re-configuring the FPGA with a partial bitstream). However, a partial writeback can only be performed if partial reconfiguration is enabled and the task is placed in a reconfigurable region. Otherwise, a full writeback has to performed. Note that partial reconfiguration is not needed for a partial readback. The time of readback and writeback can also be improved by using a high bandwidth configuration port instead of the 15 MHz JTAG interface (1.875 MB/s); using PCIe instead would saturate the 200 MHz 32-bit ICAP (internal configuration access port) [117], and hence would theoretically be 400× faster. However, the current

Table 5.3: The time to read/write hardware state of BIST designs with buried state.

	Readback Time	Extract Time	Embed Time	Writeback Time
Network Sorter	2.3 s	0.7 s	0.6 s	1 s
Reed-Solomon	2.7 s	1.6 s	1.8 s	1 s
Digital Up Conv.	4.5 s	1.3 s	0.8 s	1 s
DFT-64	3.2 s	1.1 s	0.7 s	1 s
DFT-64 FP	20 s	2.6 s	1.7 s	3 s

setup is already fast enough to allow both interactive debugging and regular checkpointing of a design in case later debugging is needed. The time to extract and embed the design state is typically less than a second even for a large design as shown in Table 5.2.

One of the features of StateMover is creating checkpoints of a design, which can be loaded into a simulator, resumed on hardware, or used for simulation/hardware discrepancy detection. The sizes of CP files of the test designs are reported in Table 5.1. The checkpoint size is small (4.2 MB for the largest design) when compared to the 132 MB size of the readback file.

We also measured the *functional* simulation time of these designs and compare them to the speed of hardware execution. The simulation is performed using the full version of ModelSim running on an Intel Xeon (E5-1620v3, 3.5GHz) CPU. We used the functional post-implementation netlist for simulating the task. As shown in Table 5.1, hardware execution has enormous speedups vs. simulation; from 500,000× to 11,000,000×, with the speedup increasing with design size. It would take ModelSim ∼127 days to simulate one second of hardware execution for the crossbar-big design, highlighting the productivity gains possible by using StateMover to run a design in hardware and move to simulation only when full visibility and controllability is necessary for debugging. When we used the behavioral representation of the task, the behavioral simulation of these designs is ∼11× faster that the functional post-implementation simulation, but still up to 1,000,000× slower than hardware execution.

5.6.3 BIST Designs with Buried State

We tested StateMover on another set of designs with buried state, introduced in Section 4.4, from Xilinx HLS examples [113] and Spiral Hardware [114]. These designs, which are listed in Table 4.2, have several hard blocks used in various configurations. The network sorter and the Reed-Solomon designs have fully registered BRAMs. The four other designs have both fully registered BRAMs and pipelined DSP blocks. The DFT designs contain FIR filters that perform pipelined multiply-and-accumulate operations in DSP blocks. The floating-point DFT design has DSP blocks cascaded using the dedicated cascade connections. We wrapped the HLS designs (the Reed-Solomon and digital-up converter designs) with test circuitry that generates input data and compares the output with the golden one. The other designs are wrapped with a BIST structure similar to the one described in section 5.6.2.

For each design, we tested two versions: the original one and the one that is modified by StateReveal (see Figure 5.1) to insert extra logic to make the buried state accessible. As expected, when we checkpointed the original version of these designs, the output did not match after execution resumption. When using the StateReveal version, these designs generated the correct outputs after moving

Table 5.4: The area and area overhead of System designs.

	LUTs	FFs	IL Overhead		DRAMMover Overhead	
			LUTs	FFs	LUTs	FFs
AES System	5,802	6,137	157	377	-	-
DDR Checker	26,432	35,622	157	377	908	1389
MicroBlaze Sys	29,405	37,870	157	377	908	1389

their state from the FPGA to the simulator and vice versa. Reading and writing the hardware state is performed using the *readback* and *writeback* commands with the *buried_state* option.

For designs with buried state, the area overhead has two components: the interruption logic overhead and the StateReveal overhead. The interruption logic added to these designs has a small fixed area of 148 LUTs and 380 FFs. The area overhead of StateReveal is reported in Table 4.2. On average, StateReveal adds 25 LUTs and 39 FFs per hard block with buried state. The area overhead is acceptable with a geometric mean of 18% and the timing overhead is low with a geometric mean of less than 5%.

The time to read back and extract the hardware state of these designs is reported in Table 5.3. This time includes the extra readbacks (multi-cycle capture) needed for retrieving the buried state. Based on the pipeline depth of the used hard blocks (containing buried state), which is reported by StateReveal, StateMover performs the required number of extra readbacks. The number of extra readbacks can vary between zero and four as discussed in section 5.4.2. The network sorter and the Reed-Solomn designs have fully registered BRAMs (two pipeline stages) and no pipelined DSPs. Therefore, only one extra readback (@t2) is needed which is performed after clocking the design for two cycles after the initial readback (@t0). The other four designs have hard blocks with two, three and four pipeline stages. Hence, a total of four readbacks is needed: the initial readback (@t0), and three extra readbacks (@t1-t4). The time to embed and write back the hardware state, again using JTAG, is also reported in Table 5.3. The time to extract and embed the BRAM state usually takes longer than that of FFs and LUTRAMs. However, the total time is still within 3 seconds.

5.6.4 System Designs

We tested StateMover on three system-level designs, listed in Table 5.4, in which the task we debug is part of a larger system and communicates with other system modules such as DDR controllers. These designs show the ability of StateMover to interrupt and read back the design state of a certain task without affecting the operation of other tasks in the system. This is extremely useful for debugging/checkpointing tasks in multi-tenant FPGAs, (e.g., in data centers). It is also crucial for tasks that interface with external memories, as the DDR controller must remain functional so StateMover can retrieve the DDR contents once a task is stopped. Note that partial reconfiguration is required to be able to write back the task state without affecting other parts of the system.

AES System

The first design, AES system, consists of two tasks (AES core and AES Check core) that interface with a memory controller through an AXI interconnect. The AES core, which is the task we want to debug, reads the contents of the memory, encrypts them and writes them back. The AES Check core reads this encrypted data, compares them to the expected data, and outputs the number of

Table 5.5: The time to read/write hardware state of System designs. DRAM transfer time over Ethernet is shown in brackets.

	Readback Time	Extract Time	Embed Time	Writeback Time
AES System	0.6 s	0.08 s	0.06 s	0.6 s
DDR Checker	0.32 (+0.005) s	0.07 s	0.04 s	0.6 (+0.005) s
MicroBlaze Sys	0.6 (+4.9) s	0.1 s	0.08 s	0.6 (+4.9) s

matches. To enable safe interruption of the AES Core, we add interruption logic to the design and a TI wrapper on the AXI interface of the AES core as shown in Figure 5.2.

To test this design, we begin running in hardware, interrupt the AES core (by setting a breakpoint) and read out its state. Next, we start a new simulation of the entire system and load the captured state into the simulator. After the execution is completed in the simulator, we check the number of matches reported by both the AES Check core in the simulator and the one running on the FPGA. The sum of the two numbers is equal to the number of memory contents, which means the AES core was able to resume correctly in the simulator and the AES Check core was able to run normally on the FPGA after the AES core was interrupted and checkpointed.

This design shows the importance of the added TI wrapper. Without the TI wrapper, if we just stop the clock of the AES core at an arbitrary time, several problems occur. For example, if the AES core is stopped in the middle of a transaction, the AXI interconnect will be locked, and hence the AES Check core cannot run. Moreover, the resulting checkpoint cannot be used to resume the execution properly because of the incomplete transaction; some relevant state is in the AXI interconnect and the memory controller, and it is not captured in this checkpoint. The TI wrapper ensures that the core state is confined inside the task borders by stopping the core from issuing any new transactions once the interrupt signal is raised, while allowing in-flight transactions to complete. Thus, the AES core can be safely interrupted and its execution can be resumed later.

Table 5.5 shows the time required to read and write the hardware state of the AES core. The area of the added interruption logic is 157 LUTs and 377 FFs, while the area of the added TI wrapper is 28 LUTs and 20 FFs. The StateMover added logic has no effect on the FMAX of the design.

DDR Checker

The DDR checker design shown in Figure 5.8 is used to test the ability of StateMover to checkpoint/debug designs that interface with an external memory. The Mem_check core is written in HLS and tests the memory by filling a memory region with a test pattern and then comparing every word in that region with its expected value. This core interfaces with a DDR controller through an AXI interconnect.

As discussed in Section 5.4.3, DDR memory contents accessible by a design should be saved and restored along with the on-chip state in order to have a complete checkpoint that can be used to resume the execution later. Hence, we added the JTAG-to-AXI IP to enable memory transfer over JTAG and the DRAMMover IP to enable memory transfer over Ethernet, as shown in Figure 5.8. The user can choose which IP to add to their design based on the size of the memory region accessible by the design and the availability of an Ethernet connection. These IPs are easily added using the graphical user interface of Vivado IP Integrator.

We test the design by running it on the FPGA, and after interrupting the Mem_Check core, we

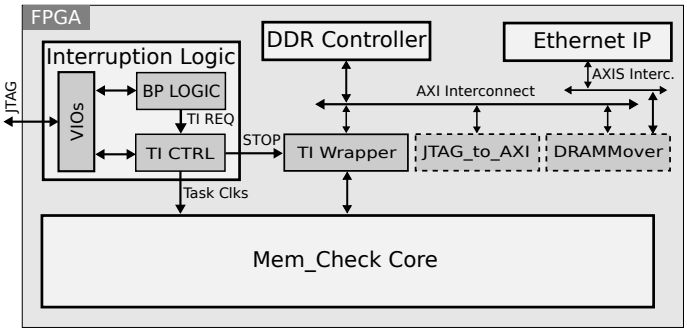


Figure 5.8: DDR Checker System. (Dark Grey: StateMover's added logic. Dashed: only one of the two IPs should be added)

read out its on-chip state along with the contents of the DRAM memory region that is being tested. This is performed using the *readback* command with the *external_state* option. The user can specify whether to use JTAG or Ethernet for DRAM transfer based on which IP was added to the design. The design was able to run properly when the captured state is loaded into the simulator. We also tested the other way around, in which we use the *writeback* command with the *external_state* option to move the dumped state from the simulator to the FPGA. The TI wrapper, placed on the task AXI interface, plays an important role by ensuring that there are no in-flight transactions when the Mem_Check core is stopped. Otherwise, we would not be able to read out the DRAM contents.

Figure 5.9 shows the speed of external memory transfer using the JTAG-to-AXI IP and the DRAMMover IP. Table 5.5 shows the time required to read and write the hardware state of the Mem_Check core. We configured the Mem_Check core to test a memory region of 256 KB which took 0.5 s over JTAG and 5 ms over Ethernet to read out or write back. The JTAG-to-AXI IP consumes 867 LUTs and 1962 FFs, while the DRAMMover IP consumes 908 LUTs and 1389 FFs. The total area overhead of StateMover includes the interruption logic, TI wrapper, as well as these DDR access IPs: the total overhead for this design is 1052 LUTs and 2359 FFs if we use JTAG to transfer the external memory, or 1093 LUTs and 1786 FFs if Ethernet is used for DDR memory transfer instead. If area utilization is critical, the JTAG to AXI or DRAMMover IPs can be reconfigured in place of the stopped module (Mem_Check in this case) after reading out its state and before writing back its state as discussed in Section 5.4.3.

MicroBlaze System

The MicroBlaze system design is similar to the DDR Checker design of Figure 5.8, but the Mem_Check core is replaced with a MicroBlaze processor that runs a program to test the DDR memory. In addition to interfacing with a DDR Controller, the MicroBlaze processor interfaces with a GPIO controller connected to the board LEDs. After performing the memory test, the processor outputs the test status on the LEDs.

We test the design by interrupting the processor and moving its state (including the instruction and data BRAMs, and the contents of the external memory region that is being tested) from the FPGA to the Simulator. We configured the MicroBlaze to test a memory region of 256 MB, which took 8.5 minutes over JTAG and 4.9 seconds over Ethernet to read out or write back. We also test the other way around, where the design is started in the Simulator and we move its state to the

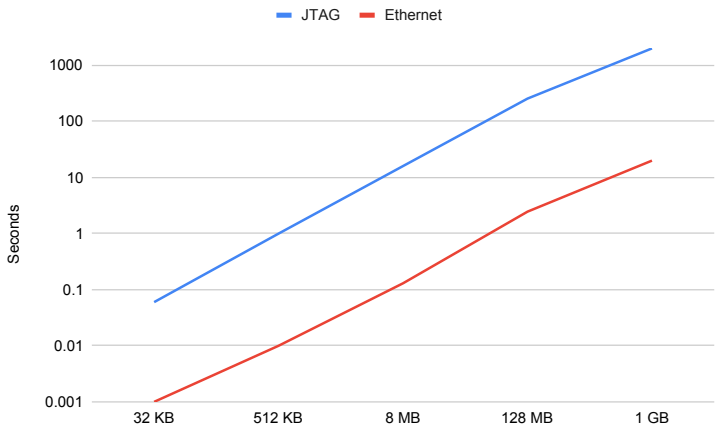


Figure 5.9: The measured speed of external memory transfer over JTAG and Ethernet.

FPGA. In both cases, the processor outputs the correct success status.

5.7 Summary

In this chapter, we proposed StateMover, an FPGA debugging framework based on hardware checkpointing, which can safely interrupt a running design and create *complete* design checkpoints. These design checkpoints can be used to resume the execution later on either an FPGA or a simulator. This enables live migration, fault recovery, and most importantly, a debugging flow that combines the speed of hardware execution and the full visibility and controllability of a simulator. Users can run their designs at full speed on an FPGA and only use the simulator when full visibility is required. This allows fast forwarding the simulation in uninteresting periods, which can save hours and days of simulation given the ∼10M× speedup of hardware execution for large designs. Users can also perform "what-if" tests by modifying the value of any state element in the simulator and then continue running the design on an FPGA.

StateMover supports any Xilinx UltraScale FPGA, and can be extended to any FPGA that supports readback. Compared to previous readback-based frameworks, this work is the first to be able to fully checkpoint designs that have multi-cycle I/O interfaces or have pipelined hard blocks. StateMover can also checkpoint the external memory, allowing it fully capture the state of a wide variety of designs.

Using StateMover does not add much design effort or hardware overhead. To enable StateMover, the user adds the interruption logic that we provide to their design. This added logic has a very small area of ∼150 LUTs and ∼350 FFs and does not affect the operating frequency of the design. Next, if the design contains hard blocks with buried state (i.e., not accessible by the configuration architecture) such as fully registered BRAMs or pipelined DSPs, the user invokes our StateReveal tool, which automatically inserts extra logic to add observability and controllability of the buried state. On average, StateReveal adds 25 LUTs and 39 FFs per hard block with buried state. For designs that use the off-chip memory, we also provide memory access IPs that can be easily added to the design or can be reconfigured in place of the stopped design to be able to save and restore the off-chip design state as well.

StateMover commands make it easy to checkpoint and transfer design state; they achieve reasonable speed using JTAG, and much higher speeds can be obtained if higher bandwidth I/O interfaces are used. Reading and writing back the entire state of a large FPGA takes 9-10 seconds over JTAG; moving the transfer to PCIe would increase bandwidth by 400× and greatly reduce this time. StateMover supports partial readback and writeback where only the relevant design state is read or written back, which reduces this time significantly if a task occupying only part of the FPGA is checkpointed. Extracting the state out of the readback data and loading it into the simulator takes less than a second for a large design with no buried state, and less than 3 seconds for a large design with fully registered BRAMs and pipelined DSPs. Moving the entire memory contents of a 1 GB DDR between an FPGA and a simulator takes less than 20 seconds over Ethernet. Reading and writing only the memory region accessible by the design cuts this time significantly.

We believe the ability of StateMover to capture and load complete FPGA checkpoints in seconds, with little designer effort or hardware overhead, opens the door to more efficient debugging, as well as new flows for context switching and live migration in FPGAs.

Chapter 6

StateLink

6.1 Introduction

In the previous chapter, we proposed, StateMover, a checkpoint-based debugging flow that can move the design state back and forth between an FPGA and a simulator. This allows the design to run at full hardware speed until a point of interest is reached. Then, the design is safely stopped and its entire on-chip state is moved to the simulator, where the user can resume it with full visibility and high controllability. When full visibility is no longer needed, StateMover can move the new design state back to the FPGA. This allows fast-forwarding hours of simulation in uninteresting periods and running more test-cases and "what-if" tests at full speed.

However, StateMover does not fully support designs with external I/Os. It can be used with designs that interface with external memories such as DRAM; the off-chip memory contents, accessible by the design, can be read out and moved to a simulator as discussed in Section 5.4.3. However, it cannot be used for debugging designs that have external I/O streaming interfaces (e.g., a network interface) as the "read out" approach cannot be used. These interfaces are very common in FPGA systems and they are infeasible to checkpoint as the data arrive as a stream. StateMover is also not suitable for designs that interface with a large storage media (e.g., a hard drive) in which moving the off-chip state to the simulator would be impractical.

One solution that has been previously proposed is to use I/O traces [3], in which the design's inputs are recorded for N cycles after the checkpoint is captured. The design is replayed in the simulator for N cycles using these recorded inputs. This method, while useful, has a number of limitations. In simulation, the design can only run for N cycles, and N must be determined by the user before the design can be implemented. A higher N value allows for a longer simulation, but it also increases the area overhead to capture I/O traces. In addition, the design state must be captured twice: the logic state must be captured at the desired moment and the I/O trace buffers must be captured again N cycles later. This method also limits signal controllability in the simulator and hardware writeback; to continue execution after simulation, the user must move the design state back to the FPGA after exactly N cycles of simulation, and not force any signal state changes in the simulator.

Instead, we propose a novel flow in which only part of the design (the task) is moved to a simulator while still being connected to and active in the overall system. This flow is similar to co-simulation

in the sense that part of the design is running in a simulator and part in hardware. However, it is fundamentally different than conventional co-simulation tools where the task runs in hardware and is fed by inputs from the software test-bench. In this chapter, we present StateLink, a transaction-based co-simulation framework, that realizes this flow and allows checkpoint-based debugging for designs with off-chip I/Os. StateLink allows the task, running in a simulator, to interact with arbitrary design parts that are still running in hardware. These parts can be other design modules on the FPGA chip, on-board storage such as DDR interfaces, and other system components such as network interfaces to other FPGAs or CPUs in a data center. StateLink connects the simulator's task interfaces with their hardware counterparts. It currently supports both AXI memory-mapped and AXI streaming standard interfaces, allowing any hardware to be linked to the simulator as long as it meets one of these standards.

StateLink brings all of the advantages of checkpoint-based debugging frameworks like StateMover to tasks interfacing with (1) external I/O streaming interfaces, (2) large off-chip storage, and (3) modules without simulation models. It also significantly cuts the simulation time for tasks that are part of a larger system. To put it another way, the user no longer needs to simulate the full design or use simulation models for other system parts including DRAM and Ethernet, which speeds up simulation and enhances debugging productivity.

This chapter is organized as follows. Section 6.2 provides a review on related co-simulation frameworks. The StateLink framework and its infrastructure are presented in Section 6.3. Section 6.4 and Section 6.5 provide details on how StateLink works for AXI memory-mapped and AXI streaming interfaces, and they also show how StateLink matches the hardware timing of transactions. StateLink is evaluated on several designs in Section 6.6 which shows the modest area overhead of StateLink and the benefits of the debugging flow enabled by StateLink and StateMover. Section 6.7 presents a comparison between the proposed debugging flow and trace-based debugging tools, and Section 6.9 summarizes the chapter.

6.2 Co-Simulation Related Work

Hardware/software co-simulation allows parts of a design to run in a simulator while others run in hardware. Xilinx's System Generator allows users to co-simulate an FPGA-based version of a Simulink block with the rest of a Simulink system to accelerate Simulink simulations [118]. In this flow, the hardware block runs on the FPGA and its inputs can only be driven from the Simulink simulation, in contrast to StateLink which allows multiple blocks to run on the FPGA and be connected to any I/O interface. The usage of co-simulation for debugging is proposed in [119, 120, 121]. In these frameworks, the task runs on an FPGA and the test-bench runs in an HDL simulator. The visibility of task internal signals is achieved by inserting scan chains in [119] and by connecting observed signals to a PCIe bus in [120]. Both systems have a significant area overhead; they increase logic utilization by 113% and 275%, respectively. Furthermore, as the number of observed signals grows, the co-simulation speed decreases considerably since each observed signal must be read out from hardware each cycle. These systems, unlike StateLink, always run in co-simulation; the user cannot transfer the entire design to an FPGA when visibility is not needed to benefit from hardware speed.

A co-simulation-based debug framework is presented in [122] that allows a design to run on an

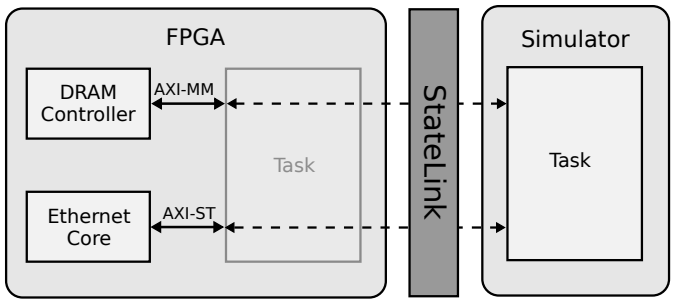


Figure 6.1: StateLink Overview.

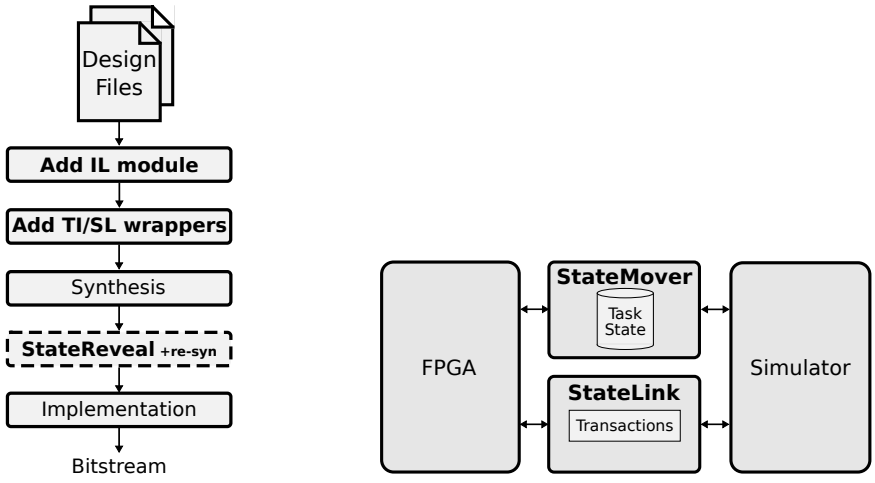
emulator, and then when an error is detected, the design is stopped and the user can select a block from the design to run in the simulator while the rest of the design continues to run on the emulator. Although this system is similar to our proposed flow in that it can run the design at hardware speed and send parts of the design to the simulator as needed, there are significant differences. First, this framework uses an emulator that runs at frequency of less than 2 MHz, which is far below the expected frequency of standalone FPGAs. Instead, we use the actual FPGA system during checkpointing and co-simulation, allowing full speed execution and avoiding the cost of an emulator. The second major difference is that in this framework's co-simulation mode, the simulator drives the entire design's inputs and clocks using captured input traces, whereas in StateLink, the design's inputs are unaffected, and the design can still be connected to external interfaces like DRAM and Ethernet. Furthermore, controlling the clock from the simulator can affect the operation of DRAM and Ethernet cores. For example, DRAM controllers have minimum as well as maximum clock rates, and use several internal clocks that must have precise frequency and phase relationships. Thus, this approach is not suitable for debugging tasks that interface with real external I/O interfaces such as Ethernet.

6.3 StateLink Framework

StateLink[1] is a transaction-based co-simulation framework in which transactions flow between the hardware and simulator. It allows a task to run in a simulator with full visibility and to interact with other system parts running in hardware including on-board I/Os, as illustrated in Figure 6.1. This allows in-system simulation of only a portion of a design (the task) which increases simulation speed and also enables checkpoint-based debugging for designs that interface with external I/Os or that include components with no simulation models. In contrast to cycle-accurate co-simulation in which the hardware and the simulator are in lockstep, in transaction-based co-simulation, the simulator and the hardware are synchronized at transaction boundaries by using interface handshaking and backpressure. This allows the on-board I/Os to run at their original frequency without the need to slow them down to match the simulator speed, and overcomes the synchronization problems of lockstep execution.

StateLink creates links between the interfaces of the task running in a simulator (referred to as the simulation task) and the interfaces of the on-chip task (referred to as the hardware task). StateLink currently supports tasks with AXI memory-mapped and AXI streaming interfaces and

[1] Available at https://github.com/samehattia/StateLink



(a) Compile flow (b) Run-time flow

Figure 6.2: StateMover Tool Flow with StateLink Integration.

can be easily extended to other standard interfaces such as Avalon. When StateLink is activated, the hardware task is safely stopped and its interfaces are *overridden*. Any incoming transactions to the hardware task are redirected to the simulation task. As well, any outgoing transactions from the interfaces of the simulation task are detected by StateLink and inserted in the hardware interface as if they were issued from the hardware task.

StateLink can be used on its own or as part of the StateMover flow discussed in Section 5.2. To integrate StateLink into the StateMover flow, the only modification is to use StateLink (SL) wrappers on the task I/O interfaces instead of TI wrappers as shown in Figure 6.2(a). Both TI and SL wrappers support standard interfaces (e.g., AXI) which allows the user to easily insert them in the design using Xilinx Vivado IP integrator. The SL wrappers provide a superset of the TI wrapper functionality: they can not only safely stop the design but also support transaction-based co-simulation by overriding the task interfaces when StateLink is active.

To use StateLink on its own, the design is implemented on an FPGA and the hardware task is not allowed to run by disabling its clocks using the IL module. Then, the user opens up an HDL simulator (e.g., ModelSim) and starts a simulation augmented with the $\text{StateLink}_{\text{SIM}}$ program, which overrides the interfaces of the simulation task. The user also starts the $\text{StateLink}_{\text{HW}}$ program to communicate with the hardware. Once StateLink is activated, transactions start to flow between the interfaces of the simulation task and their equivalents on the hardware in both directions.

When StateLink is used as part of the StateMover flow as shown in Figure 6.2(b), the user can stop the hardware execution of the task at a certain point and read out its entire on-chip state using the StateMover console. Once the on-chip state of the task is loaded in the simulator using StateMover, the simulation task can interact with other system parts that reside on the FPGA using StateLink. The user can also stop the simulation at a certain point and move the simulation state of the task back to the FPGA. Once the task state is written back to the FPGA, StateLink is deactivated and the hardware task can resume.

The rest of this section describes the infrastructure of StateLink which consists of three components: $\text{StateLink}_{\text{SIM}}$, $\text{StateLink}_{\text{HW}}$, and SL wrappers, as shown in Figure 6.3.

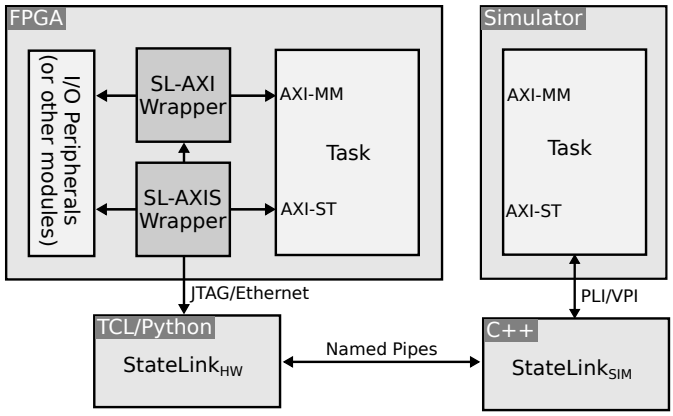


Figure 6.3: StateLink Infrastructure.

6.3.1 StateLink$_{\text{SIM}}$

The StateLink$_{\text{SIM}}$ program is written in C++ and uses PLI/VPI to interface with the HDL simulation. At the beginning of the simulation, StateLink$_{\text{SIM}}$ reads a parameter file that contains the name of the task and scans the netlist to detect the number of AXI-MM and AXI-ST interfaces to/from the task and their associated clocks. For each interface, a callback function is registered and is called at every active edge of the associated interface clock. The callback function acts as a sniffer for active outgoing transactions from the simulation task and for available incoming transactions from the hardware.

StateLink$_{\text{SIM}}$ also creates named pipes (FIFOs) for inter-process communication with StateLink$_{\text{HW}}$. These pipes are created for each interface so that transactions can flow independently between the two programs. For input interfaces, StateLink$_{\text{SIM}}$ also creates threads to listen for incoming transactions from the hardware on the pipes associated with these interfaces. Incoming transactions (AXI-ST RX transactions or AXI-MM read or write responses) are fed to the simulation task by StateLink$_{\text{SIM}}$ following the rules of the AXI standard. For example, StateLink$_{\text{SIM}}$ fully supports AXI-MM burst transactions and AXI-ST multi-flit packets.

6.3.2 StateLink$_{\text{HW}}$

The StateLink$_{\text{HW}}$ program is written in both Tcl and Python and communicates with StateLink$_{\text{SIM}}$ through named pipes, as previously mentioned. To ensure StateLink is usable with almost any FPGA design, it supports both JTAG and Ethernet connectivity with Ethernet-based StateLink providing a faster link.

StateLink$_{\text{HW}}$ forwards transactions coming from StateLink$_{\text{SIM}}$ to SL wrappers and vice versa. StateLink$_{\text{HW}}$ uses Tcl-based event handlers and Python-based asynchronous I/O to make sure transactions coming on named pipes are forwarded independently. Once the HDL simulation is closed, StateLink will be deactivated and StateLink$_{\text{HW}}$ will exit and print the number of transactions that passed through the links between the simulation task interfaces and the hardware task interfaces.

The communication between StateLink$_{\text{HW}}$ and JTAG-based SL wrappers is performed using Xilinx's JTAG-to-AXI IP commands [123], which can create AXI-MM read/write transactions on the board using Tcl commands from the host. This means that StateLink$_{\text{HW}}$ converts all transactions

to AXI-MM transactions and then the JTAG-based SL wrappers convert them back to their original type, as discussed in Section 6.5.

For Ethernet-based StateLink, $\text{StateLink}_{\text{HW}}$ uses sockets to communicate with Ethernet SL wrappers. The transactions are formatted into packets (referred to as Ethernet-AXI packets) and sent over Ethernet. Each Ethernet-AXI packet has an Ethernet-AXI ID which identifies the SL wrapper (task interface) to which the packets are sent or from which they are received.

6.3.3 SL Wrappers

SL wrappers are parameterized IPs that are placed on the task interfaces, as shown in Figure 6.3. They are available as packaged IPs and can be easily added to the design using the graphical user interface of Vivado IP integrator. They are interface-dependent; the SL-AXI wrapper is for AXI memory-mapped interfaces, while the SL-AXIS wrapper is for AXI streaming interfaces. There are two versions of each wrapper: JTAG-based for communicating over JTAG with $\text{StateLink}_{\text{HW}}$ and Ethernet-based for communicating over Ethernet with $\text{StateLink}_{\text{HW}}$.

SL wrappers are responsible for safely stopping the task interfaces at a user-defined breakpoint or at the beginning of the hardware execution if StateMover is not used. When StateLink is active, they override the task interfaces. They receive transactions from the simulation task through $\text{StateLink}_{\text{HW}}$ and replay them on hardware as if they are generated from the hardware task. For incoming transactions that were intended to the hardware task, SL wrappers redirect them to the simulation task by sending them over to $\text{StateLink}_{\text{HW}}$.

6.4 AXI Memory-Mapped Interfaces

In this section, we describe how StateLink works for AXI memory-mapped (AXI-MM) interfaces.

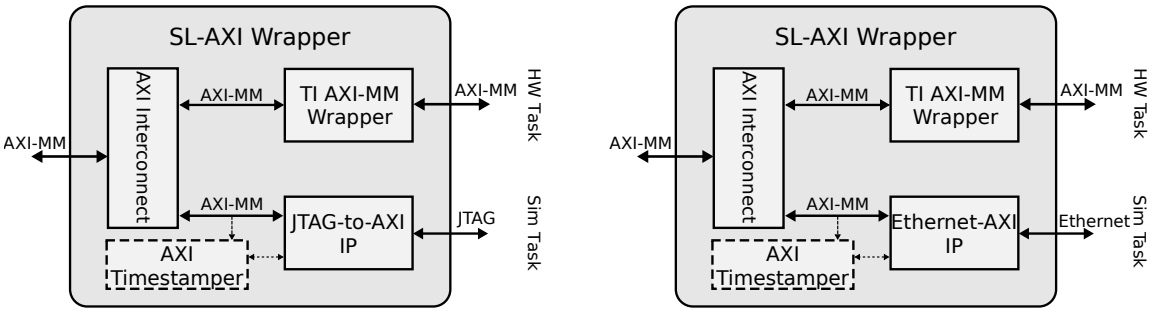
6.4.1 SL-AXI Wrapper

The SL-AXI wrapper, which is shown in Figure 6.4, is placed on AXI-MM interfaces. It consists of a task interruption (TI) AXI memory-mapped (AXI-MM) wrapper and a Xilinx JTAG-to-AXI IP in JTAG-based wrappers or an Ethernet-AXI IP in Ethernet-based wrappers.

The TI AXI-MM wrapper safely stops the task AXI-MM interfaces at a user-defined breakpoint when StateLink is used as part of the StateMover flow. It is controlled by the IL module which sends a stop request to the wrapper once the breakpoint is reached. At a stop request, the TI AXI-MM wrapper prevents the hardware task from issuing new transactions, while allowing in-flight transactions to complete. Once the task AXI-MM interface is stopped, the JTAG-to-AXI/Ethernet-AXI IP becomes the only master of the AXI interconnect, and it issues and receives transactions on behalf of the simulation task. During normal operation, both the TI wrapper and the entire SL-AXI wrapper are fully transparent; transactions pass through the wrapper without any extra latency or modifications.

6.4.2 Transaction Flow

An AXI-MM interface consists of five channels: write address, read address, write data, read data, and write response. For each interface, $\text{StateLink}_{\text{SIM}}$ checks for active AXI-MM transactions issued



(a) JTAG-based (b) Ethernet-based

Figure 6.4: The SL-AXI Wrapper.

by the simulation task every clock cycle.

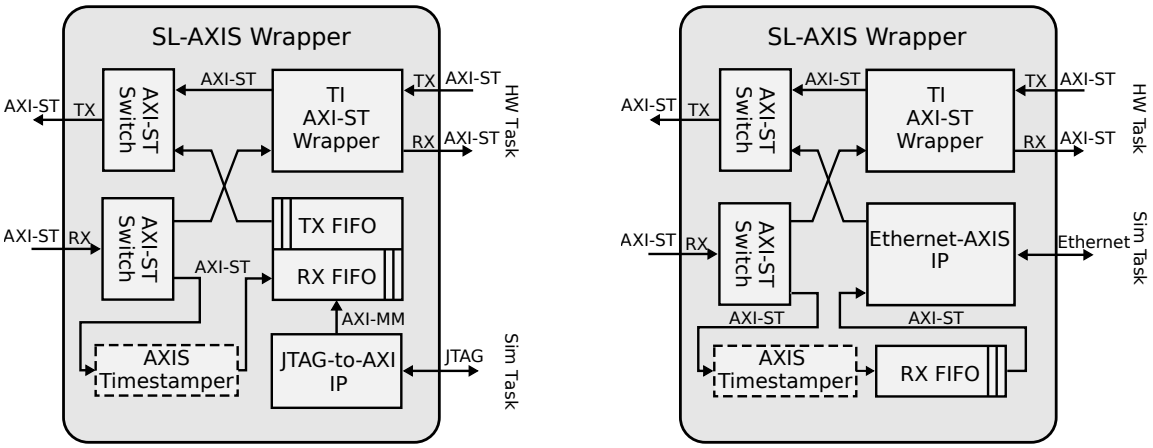
For each write transaction, $\text{StateLink}_{\text{SIM}}$ sends the write address, write data and the burst length to $\text{StateLink}_{\text{HW}}$, and then blocks the simulation until $\text{StateLink}_{\text{HW}}$ responds back. $\text{StateLink}_{\text{HW}}$ re-creates the write transaction and sends it to the SL-AXI wrapper over JTAG or Ethernet. Then, the JTAG-to-AXI/Ethernet-AXI IP inserts the write transaction on the AXI bus as if it was issued by the hardware task. If the on-chip write response indicates a successful write transaction, $\text{StateLink}_{\text{HW}}$ responds back to the $\text{StateLink}_{\text{SIM}}$ which passes the write response to the simulation task.

Similarly, for each read transaction, $\text{StateLink}_{\text{SIM}}$ sends the read address and the burst length to $\text{StateLink}_{\text{HW}}$, and then waits for the read data/response to come back. When the read data/response is received, $\text{StateLink}_{\text{SIM}}$ feeds the data and the response to the simulation task.

6.4.3 Hardware Time-Matching

A consequence of moving to transaction-based debugging is that we are not reproducing the exact cycle-by-cycle behavior at task interfaces. While most bugs will persist if we maintain the same transactions but with different timing, some timing or race condition dependent bugs may change their behavior. To assist in reproducing this latter class of bugs, StateLink provides options to mimic or reproduce the hardware interface timing.

StateLink provides two options to mimic the hardware behavior of the read latency (i.e., the number of cycles it takes for the read data to come back) and the write latency (i.e., the number of cycles it takes for the write response to come back). The first option is for the user to specify a fixed value for the read and write latencies; StateLink will wait the specified number of clock cycles to feed the received read data and write response to the simulation task. If accurate read/write latency is crucial, StateLink can instead use its *timestamp* mode (at a small area cost) to accurately capture the hardware read and write latency and use them in the simulation. In this mode, an AXI Timestamper IP is used in the SL-AXI wrapper, as shown in dashed lines in Figure 6.4. This IP sniffs transactions issued on behalf of the simulation task from the JTAG-to-AXI/Ethernet-AXI IP, and counts the number of cycles it takes for the read-data/write-response to come back after a read/write request is issued. This captured read and write latency is sent along with the read data and the write response to $\text{StateLink}_{\text{HW}}$ in one Ethernet-AXI packet in Ethernet-based StateLink or on a separate AXI-MM transaction in JTAG-based StateLink. $\text{StateLink}_{\text{HW}}$ passes these values to $\text{StateLink}_{\text{SIM}}$ which uses them to know when to feed the read data and write response to the



(a) JTAG-based (b) Ethernet-based

Figure 6.5: The SL-AXIS Wrapper.

simulation task.

6.5 AXI Streaming Interfaces

In this section, we describe how StateLink works for AXI streaming (AXI-ST) interfaces.

6.5.1 SL-AXIS Wrapper

The SL-AXIS wrapper, which is shown in Figure 6.5, is placed on AXI-ST interfaces. The wrapper is bidirectional; it can be connected to the task's AXI-ST master (TX) interface and the task's AXI-ST slave (RX) interface at the same time.

The JTAG-based SL-AXIS wrapper contains a TI AXI-ST wrapper for safely stopping the task AXI-ST interfaces at a user defined breakpoint, and a Xilinx JTAG-to-AXI IP for communicating with $\text{StateLink}_{\text{HW}}$ over JTAG. Since the JTAG-to-AXI IP can only issue AXI-MM transactions, the wrapper also contains an RX FIFO and a TX FIFO that convert these AXI-MM transactions into AXI-ST transactions. The RX/TX FIFO sends and receives AXI-ST transactions on behalf of the simulation task. When StateLink is active, one AXI-ST switch is used to direct incoming packets to the RX FIFO and another AXI-ST switch is used to drive the TX interface from the TX FIFO instead of the hardware task.

The Ethernet-based SL-AXIS wrapper uses our Ethernet-AXIS IP for communicating with $\text{StateLink}_{\text{HW}}$ over Ethernet. The main difference is that Ethernet-AXIS IP can issue and receive AXI-ST transactions directly, in contrast to JTAG-to-AXI IP. This obviates the need for the TX and RX FIFOs. However, the wrapper still contains a simple AXI-ST FIFO (with only AXI-ST interfaces) for storing AXI-ST RX packets before sending them over to the simulation task.

6.5.2 Transaction Flow

$\text{StateLink}_{\text{SIM}}$ checks for active outgoing AXI-ST transactions issued by the simulation task on its AXI-ST TX interfaces every cycle, and also checks for available incoming AXI-ST transactions from

hardware intended for AXI-ST RX interfaces of the simulation task.

TX Transactions

For outgoing transactions, $\text{StateLink}_{\text{SIM}}$ sends the data and length of an AXI-ST packet (a packet can consist of a single or multiple transactions) to $\text{StateLink}_{\text{HW}}$.

When using the JTAG-based wrappers, $\text{StateLink}_{\text{HW}}$ first checks that the SL-AXIS's TX FIFO has available space. Then, it creates AXI-MM transactions to write the packet data and the packet length into this FIFO through the JTAG-to-AXI IP. Once the packet and its length are written to the TX FIFO, the FIFO inserts the AXI-ST transactions on the TX interface as if they were sent from the hardware task.

When using the Ethernet-based wrappers, $\text{StateLink}_{\text{HW}}$ creates an Ethernet-AXI packet that contains the packet data and packet length and sends it to the Ethernet-AXIS IP over Ethernet. The Ethernet-AXIS IP inserts the AXI-ST transactions on the TX interface, and then after the transaction is successfully inserted, it notifies $\text{StateLink}_{\text{HW}}$ that it is ready to receive another packet.

RX Transactions

Incoming transactions to the hardware task are directed to the SL-AXIS's RX FIFO once the task interface is safely stopped.

When using the JTAG-based wrappers, $\text{StateLink}_{\text{HW}}$ communicates during its startup with the RX FIFO to check for any packets that were received before the program was started. The RX FIFO has a configurable interrupt signal, which can be read out from $\text{StateLink}_{\text{HW}}$ through Xilinx VIOs over JTAG. During startup, $\text{StateLink}_{\text{HW}}$ also configures and enables the RX FIFO's interrupt signal to go high when an RX packet is received. It then creates a thread to wait for the interrupt signal to go high. When the interrupt signal is set, $\text{StateLink}_{\text{HW}}$ creates AXI-MM transactions to continuously read RX packets from the FIFO until the FIFO is empty. These read packets and their packet lengths are transferred to $\text{StateLink}_{\text{SIM}}$.

When using the Ethernet-based wrappers, $\text{StateLink}_{\text{HW}}$ sends an Ethernet-AXI packet at the beginning to inform the Ethernet-AXIS IP that it is up and is accepting packets. It then creates a socket for each SL-AXIS wrapper, and listens on these sockets. Once an RX packet is received in the RX FIFO, the Ethernet-AXIS IP reads it, then embeds it along with its length in an Ethernet-AXI packet, and sends it to $\text{StateLink}_{\text{HW}}$. The latter extracts the RX packet and its length and passes them forward to $\text{StateLink}_{\text{SIM}}$.

$\text{StateLink}_{\text{SIM}}$ stores the incoming packets in a software queue. It then feeds these packets to the task following the AXI-ST standard. The $\text{StateLink}_{\text{SIM}}$'s software queue acts as a storage extension for the RX FIFO. This extra storage is large and is limited only by the host memory. It is helpful for accepting bursts of streaming data that arrive faster than the simulator can process them.

However, JTAG-based StateLink is still limited by the transfer rate over JTAG which is $\sim$2 MB/s. When the on-chip RX FIFO is full because packets are arriving faster than JTAG transfer speed, the FIFO will backpressure the sender, thereby avoiding any data loss. In case of Ethernet packets, if the internal FIFO of the Ethernet core is full due to the backpressure, the Ethernet core will start dropping packets, which can also be tolerated if a reliable communication protocol such as TCP is used. On the other hand, Ethernet-based StateLink does not have this limitation and can support data arriving at Ethernet speed.

Note that the extra buffer space added by the RX FIFO and the software queue affects the *fullness* of the existing buffers in the design. This does not affect the functionality of designs with AXI interfaces due to the AXI handshaking/backpressure mechanisms, but it might hide bugs related to a buffer overflow. Accordingly, StateLink allows the user to configure the size of both the RX FIFO and the software queue. Using small buffer sizes will expose bugs related to backpressure and buffer fullness. However, as discussed before, prolonged backpressure due to the slow simulation speed can cause data loss for systems with no end-to-end backpressure, and therefore, a large software buffer should be specified for such systems.

6.5.3 Hardware Time-Matching

As in Section 6.4.3, StateLink provides two options to match the hardware timing to allow even timing-dependent bugs to be investigated with StateLink. The first option is for the user to specify a minimum cycle delay between consequent packets. This is useful if the packets are expected to arrive with a specific rate. $\text{StateLink}_{\text{SIM}}$ will feed a new packet to the task only if the specified number of clock cycles has passed.

To mimic the hardware behavior more accurately, StateLink provides a timestamp mode in which a timestamp is added to each incoming packet before inserting it in the RX FIFO. This allows mimicking the actual data rate of which the AXI-ST packets are arriving. It is also beneficial for dependent interfaces in which the order of transactions is important; it ensures the incoming packets on different interfaces are fed to the simulation task in their original order of arrival. In the timestamp mode, the AXIS timestamper IP is used inside the SL-AXIS wrapper, as shown in dashed lines in Figure 6.5. The AXIS timestamper IP inserts an extra flit that contains a timestamp (number of clock cycles since the task is stopped) into the RX FIFO with each incoming packet. The incoming packet along with its timestamp are sent to $\text{StateLink}_{\text{SIM}}$, which extracts the timestamp and uses it to calculate when to feed the packet to the task. This is done by comparing the timestamp with an internal counter that counts the number of clock cycles since the start of the simulation. To speed up the simulation when using the timestamp mode, $\text{StateLink}_{\text{SIM}}$ advances the internal counter to skip over long idle cycles in which there are no incoming packets. This is performed by comparing the timestamps of the incoming packets with the current value of the counter. If the difference exceeds the maximum number of idle cycles (specified by the user), the counter will be advanced.

6.6 Results

In this section, we quantify the area and timing overhead of StateLink. We then demonstrate the use of StateLink and the StateMover debugging flow on a complete Memcached design. We also report the total overhead and speedup of the entire debugging flow that integrates StateLink and StateMover.

6.6.1 StateLink Overhead

The hardware cost of using StateLink has two components: the interruption logic (IL) overhead and the SL wrappers overhead. The IL module overhead is small, and the exact overhead depends

Table 6.1: Area and FMAX (in MHz) of the SL-AXI wrappers vs. data width.

	Width	LUTs	FFs	BRAMs	FMAX
JTAG-based SL-AXI	32-bit	916	1751	2.5	506
	64-bit	1143	2016	3	501
	128-bit	1822	2654	3	418
	256-bit	2252	3068	3	405
	512-bit	3018	3884	3	371
Ethernet-based SL-AXI	32-bit	1160	2823	1	425
	64-bit	1214	3049	2	420
	128-bit	1767	3633	2	413
	256-bit	2222	4047	2	380
	512-bit	2986	4867	2	380

on the breakpoints set. For example, the IL we use in Section 6.6.2 allows breakpoints at any desired clock cycle count or transaction count; it requires 157 LUTs and 377 FFs and has no timing overhead. The overhead of the SL wrappers depends on the number, width, and type of task interfaces. To quantify the hardware cost of SL wrappers, we implemented the Ethernet/JTAG-based SL-AXI and SL-AXIS wrappers on a Xilinx UltraScale (XCKU040-2) FPGA. To calculate the maximum operating frequency of the wrappers, we inserted registers on the inputs and outputs of the wrappers (with *dont_touch* directives) and used a timing constraint of 500 MHz.

SL-AXI Wrapper

SL-AXI IP is placed on each AXI-MM interface of a task. Table 6.1 shows the LUT, FF, and 36Kb BRAM resource use and the maximum operating frequency of the JTAG-based and Ethernet-based SL-AXI wrappers at different data widths.

The JTAG-based SL-AXI wrapper has an area overhead ranging from 0.9 kLUTs and 1.8 kFFs (32-bit interfaces) to 3 kLUTs and 3.9 kFFs (512-bit interfaces). It can run at frequencies between 506 MHz and 371 MHz, which exceed typical design frequencies in this device family. The JTAG-to-AXI IP is the largest area block in the JTAG-based SL-AXI wrapper. The 64-bit JTAG-to-AXI IP uses 856 LUTs, 1906 FFs, and three 36Kb BRAMs. It accounts for 87% to 40% of the logic area and 100% of the BRAM usage of 64 to 512-bit SL-AXI wrappers, respectively. The JTAG-to-AXI IP can be shared between the SL wrappers for tasks that have several interfaces (including those with different data widths), thereby reducing the area overhead significantly.

The Ethernet-based SL-AXI wrapper has an area overhead ranging from 1.1 kLUTs and 2.8 kFFs (32-bit interfaces) to 3 kLUTs and 4.9 kFFs (512-bit interfaces). It can run at frequencies between 425 MHz and 380 MHz. The Ethernet-AXI IP is the largest block in terms of area inside the Ethernet-based SL-AXI wrapper. The 64-bit Ethernet-AXI IP uses 915 LUTs, 2932 FFs, and two 36Kb BRAMs, and as in JTAG this component can be shared across multiple interfaces to save area.

To use the timestamp mode, described in Section 6.4.3, the AXI Timestamper IP is inserted inside the SL-AXI wrapper and it increases the wrapper area by 55 LUTs and 232 FFs. The SL-AXI wrappers do not add any cycle latency on transactions during normal operation (when StateLink is inactive); there are no pipeline stages on the path from the hardware task to the output.

Table 6.2: Area and FMAX (in MHz) of SL-AXIS wrappers vs. data width.

	Width	LUTs	FFs	BRAMs	FMAX
JTAG-based SL-AXIS	32-bit	1423	2337	4.5	366
	64-bit	1500	2639	4.5	361
	128-bit	1641	2935	4.5	354
	256-bit	1869	3521	4.5	368
	512-bit	2333	4691	4.5	355
Ethernet-based SL-AXIS	32-bit	626	886	1	402
	64-bit	907	1263	1.5	343
	128-bit	1037	1855	1.5	327
	256-bit	1309	2438	1.5	337
	512-bit	1749	3600	1.5	343

SL-AXIS Wrapper

The SL-AXIS IP can be connected to two AXI-ST interfaces of a task (a master and a slave). Table 6.2 shows the LUT, FF, and 36Kb BRAM resource use and the maximum operating frequency of the JTAG-based and Ethernet-based SL-AXIS wrappers at different data widths.

The JTAG-based SL-AXIS wrapper has an area overhead ranging from 1.4 kLUTs and 2.3 kFFs (32-bit interfaces) to 2.3 kLUTs and 4.7 kFFs (512-bit interfaces). It can run at frequencies between 366 MHz and 354 MHz with the FIFO logic being the frequency limiting block. The JTAG-based SL-AXIS wrappers contain a 32-bit JTAG-to-AXI IP that uses 620 LUTs, 1406 FFs, and 2.5 36Kb BRAMs. This comprises 54% to 29% of the logic area and more than 50% of the BRAMs of 32 to 512-bit SL-AXIS wrappers, respectively; this block can again be shared between several SL-AXIS wrappers to reduce the area overhead.

The Ethernet-based SL-AXIS wrapper is slightly smaller and ranges from 0.6 kLUTs and 0.9 kFFs (32-bit interfaces) to 1.7 kLUTs and 3.6 kFFs (512-bit interfaces). It can run at frequencies between 402 MHz and 327 MHz. The Ethernet-based wrapper is smaller than its JTAG equivalent because it does not need to convert the AXI-ST transactions to AXI-MM transactions to send them to the host. This leads to the removal of the TX FIFO and the use of a simplified RX FIFO (with no conversion logic). The Ethernet-AXIS IP is also smaller than the JTAG-to-AXI IP; for example, the 32-bit Ethernet-AXIS IP uses only 395 LUTs and 597 FFs, and 0 BRAMs.

To use the timestamp mode, described in Section 6.5.3, the AXIS Timestamper IP is inserted inside the SL-AXIS wrapper and it increases the wrapper area by 107 LUTs and 65 FFs. The SL-AXIS wrappers do not add any pipeline stages between the hardware task and their output, so they do not affect the cycle latency of transactions when StateLink is inactive.

6.6.2 StateLink Evaluation Using a Memcached Design

To evaluate StateLink and the entire proposed debugging flow, we use a Memcached system. This is a complex design with multiple interfaces and multiple clocks, and hence exercises most StateLink and StateMover features.

Overview

The Memcached system, which is shown in Figure 6.6, consists of a Memcached server, a DRAM controller and an Ethernet sub-system, and has two major clocks (DRAM and Ethernet clocks). The Memcached server consists of a Memcached pipeline, two read/write converters and two AXI data mover IPs. The Memcached pipeline is an HLS IP from Xilinx [106] that consists of a request parser, hash table, value store, and response formatter. The Memcached server has an AXI-MM interface to communicate with DDR memory, which stores the hash table and the values. The server has also two AXI-ST interfaces to receive Memcached requests and to send Memcached responses from/to the client. In our setup, the client is running on an Intel CPU host and is connected to the FPGA board over 1 Gigabit Ethernet.

Test Setup

To use StateLink, we first added the interruption logic to control the clocks of the task (Memcached server) and added the SL wrappers. An SL-AXI wrapper is inserted on the AXI-MM interface and an SL-AXIS wrapper is connected to the two AXI-ST interfaces. To be able to move the task state back and forth between the FPGA and the simulator, we ran StateReveal on the synthesized netlist (see Figure 6.2) so we can create complete checkpoints; this step is needed only if the user wants to move the task state between the FPGA and the simulator. Then, we implemented the Memcached system on a Xilinx UltraScale KCU105 Board.

Test Flow

We first tested StateLink on its own, in which we disable the clocks of the Memcached server (hardware task) at start up time. Then, we run an HDL simulation of the Memcached server (task) and use StateLink to connect the interfaces of the simulation task to their equivalents in hardware. In other words, the Memcached server is running with full visibility in a simulator while still being able to read/write to the DDR memory, and send/receive packets over Ethernet. We also start the Memcached client which sends Memcached requests to the server. In this test the Memcached server running in the simulator was able to receive all the Memcached requests sent from the Memcached client running on the host, and the client received all the Memcached responses correctly.

We also tested the entire proposed flow, in which we start running the Memcached server on the FPGA, and then we use StateMover to set a breakpoint at which the server is safely stopped. Next we use StateMover to read out its state and load it into the simulator. Then, we continue running the Memcached server in the simulation with full visibility. The Memcached client received all the responses correctly, where some of the responses were sent from the hardware task before stopping it, and the rest of responses were sent from the simulation task after the task state was moved to the simulator.

Moving Different System Parts

Instead of simulating the entire Memcached server, we did another test in which the simulation task is the Memcached pipeline, as shown in Figure 6.6 as task 2. This test shows that we can move different system parts between the FPGA and the simulator, not only those which interface with external I/Os such as Ethernet or DDR. It also shows the scalability of StateLink and that it

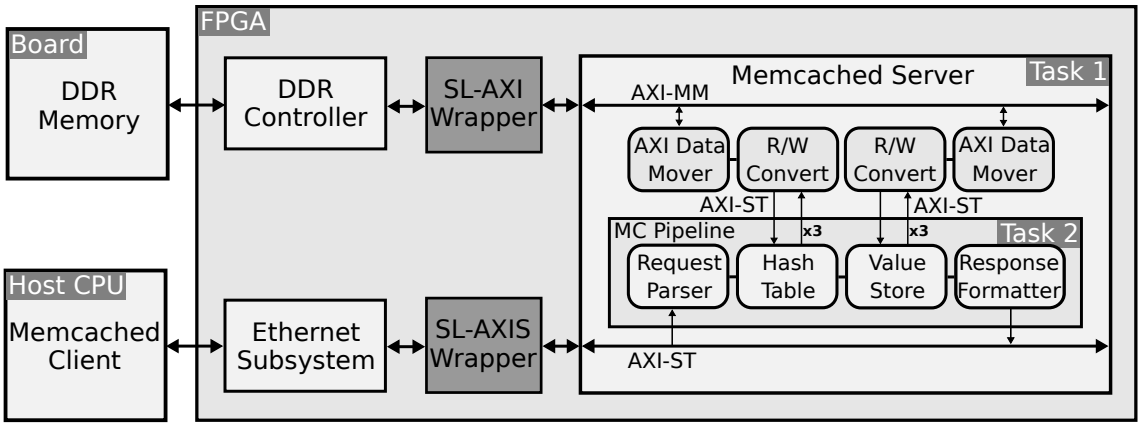


Figure 6.6: Memcached system.

can work with tasks with multiple interfaces; the Memcached pipeline has ten AXI-ST interfaces as shown in Figure 6.6.

In this test, we first added SL-AXIS wrappers on the interfaces of the Memcached pipeline using Vivado IP Integrator. Then, we implemented the design on the FPGA, where only the Memcached pipeline is stopped by disabling its clock. After that, we ran an HDL simulation of the Memcached pipeline only, and we used StateLink to connect its ten interfaces to the read/write converters and the Ethernet subsystem running on the board. The Memcached client was also able to receive all the Memcached responses correctly. The achieved speedup and the overhead is discussed in Section 6.6.3.

Debugging Session

To highlight the benefits of the proposed debugging flow, in this section we walk through the process we used to find and fix a real bug in our implementation of the Memcached system.

Before implementing the Memcached system on the FPGA board, we simulated the design by writing a testbench that feeds the system with 1024 Memcached requests. The requests are combinations of Memcached SET and GET commands that store and retrieve key-value pairs, respectively, and they have variable key and value lengths. The simulation took ~25 minutes, and all Memcached responses are received correctly. We then implemented the system on the board and tested it using the Memcached client, which allows testing the system with vastly more requests in a fraction of a second. However, after sending more than 1800 requests and receiving their responses correctly, the client received a SET response that has a non-zero status code which indicates an error. A SET command fails if the key does not exist and there are no empty slots in the hash table to add a new key. When we checked the corresponding Memcached request (request #1838), we found that it has a key that had been already set before and had not been subsequently deleted. Hence the SET command should have overwritten the corresponding value, and should never fail due to a capacity limit.

The debugging steps are shown in Figure 6.7. First, we used the *set_breakpoint* command of the StateMover (SM) Console to stop the Memcached core just before receiving request #1838. The default breakpoint logic of StateMover interrupts the design when the value of its internal counter is equal to the value set by the *set_breakpoint* command. This counter can be incremented based on a user-defined condition; in this case the condition is reception of a new packet (i.e., request)

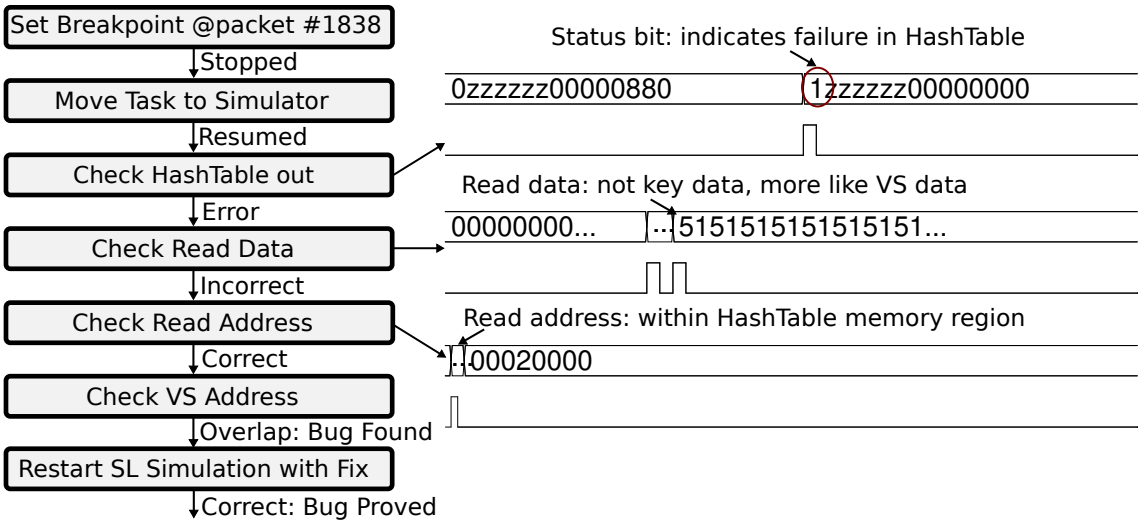


Figure 6.7: Debugging session steps (left) and simulation waveform (right).

allowing *set_breakpoint* to stop on a specific request number. In general, the user can provide any breakpoint logic that is suitable to the application in a manner analogous to the trigger condition in trace-based debugging.

After the design stopped at the breakpoint, we used the *readback* command of the SM Console to read out the on-chip state of the Memcached core and move it to the simulator. Moving the state took ∼15 seconds. Next, we resumed execution of the Memcached core in the simulator where we have full visibility of all signals, while the core is still connected to on-board Ethernet and DDR using StateLink. Since the error was a failed SET command we suspected a bug in the Hash Table and first checked the signals of the Hash Table IP when request #1838 was processed. The Hash Table IP takes a key, hashes it, and then uses the hashed value to access the hash table in DDR memory. It then compares the words read from the memory with the given key and checks for empty slots. As shown in Figure 6.7, the status code output of the Hash Table IP indicates a failed operation, which means the key was not found and there are no empty slots. We next checked the data words read from the DDR memory, and found that they did not appear to be the expected key data from the hash table, but instead appeared to be Value Store (VS) data, which should have been stored in a different DDR region. This led us to believe either the address provided to the DDR memory was not correct or the Hash Table had been overwritten at this location. Thus, we checked the address generated by the AXI Data Mover connected to the Hash Table IP; by examining signals in the simulation we found it was correct. We continued running the Memcached core in simulation and now checked the addresses generated by the AXI Data Mover IP connected to the Value Store and found that the addresses overlap with the memory region that contains the Hash Table (0x000010000-0x00001FFFF), suggesting that memory corruption was the underlying issue.

Next, we checked the code that creates the address pool for Value Store data. The code creates a pool of 64 addresses and each address corresponds to a memory block of 1024 words (not bytes), so it can generate addresses from 0x000000000 to 0x00007FFFF. This suggested that the memory region of the Hash Table should start from 0x000080000 (not 0x000010000). To quickly prove our

hypothesis, we leveraged the high controllability of the simulator. First, we restarted the execution of the Memcached core on the simulator from the beginning with StateLink enabled. Second, we forced the 19th bit of the Hash Table read and write addresses that are going to the AXI Data Mover IP to one, thereby removing the overlap between the Hash Table and Value Store memory regions. Then, we reran the Memcached client, and this time, all the responses of the 3000 sent requests were received correctly. This StateLink simulation took ∼8 minutes, and confirmed that we had found the root cause of the bug.

This case study illustrates how the StateLink flow speeds up the debugging process of intermittent bugs such as this memory overlap bug. Without the simulation fast-forwarding capability of this flow, it would have taken nearly an hour to reach the point-of-interest in the simulator, as opposed to ∼15 seconds in this flow. Moreover, finding the root cause of the bug required the examination of several signals; without the full visibility of task signals at the point-of-interest, it would have been difficult and time-consuming to find the root cause. The controllability of the simulator combined with the speed benefits of StateLink also allowed us to quickly verify the correctness of the proposed bug fix by forcing a signal change without an FPGA recompile.

6.6.3 Design Overheads and Speedup

In this section, we report the total area overhead, timing impact and simulation speedup for four designs: two versions of the Memcached system discussed in the previous section, and two other complete designs: an AES system and a DDR system. The task in the first version of the Memcached design is the entire Memcached server shown as task 1 in Figure 6.6, and for the second version it is the Memcached pipeline shown as task 2 in Figure 6.6.

We run the same two tests described in Section 6.6.2 on the AES system and the DDR system: testing StateLink on its own where we start a simulation of the task and use StateLink to connect the task to the rest of the system running in hardware; and testing the entire debugging flow where we start the design on the hardware and then stop the task and move it to the simulator using StateMover while keeping the task connected to the rest of the system using StateLink.

The AES system consists of an AES core (the task) that encrypts a memory array of 128k elements and an AES check core that reads the encrypted data and outputs the number of correctly encrypted data items. The AES core has an AXI-MM interface to communicate with the memory array, on which an SL-AXI wrapper is placed. In both tests, after the simulation of the AES core was completed, the AES check core running on the FPGA showed that all the memory elements had been encrypted correctly.

The DDR system consists of a memory-check core (the task), written in HLS, which fills a memory region with a test pattern and then reads every word in that region and outputs the number of correct words. The core interfaces with a DRAM controller using an AXI-MM interface, on which the SL-AXI wrapper is placed. In both tests, after the simulation of the memory-check core was completed, the core reported that all the read words matched the test pattern written.

Table 6.3 shows the total area in terms of LUTs and FFs of these four designs, including area added by StateLink and StateMover, as well as the percentage area overhead added. StateLink area overhead is the percentage of the design logic (LUTs and FFs) used by SL wrappers. For the AES and DDR systems, we used JTAG-based SL wrappers and for the two versions of the Memcached system, we used Ethernet-based SL wrappers since the Memcached system already uses Ethernet

Table 6.3: StateLink overhead and total area overhead for the full designs.

	Design Area		SL Area	Total Area
	LUTs	FFs	Overhead	Overhead
AES System	8,746	11,021	13%	16%
DDR System	21,753	29,901	5%	6%
Memcached with task 1	64,586	97,082	8%	13%
Memcached with task 2	66,392	96,986	13%	13%

for communicating with the host. The timestamp mode (i.e., hardware time-matching) is enabled in the two versions of the Memcached system to mimic the data rate of the Ethernet packets and to ensure that incoming transactions are fed in the correct order to the multiple AXI-ST interfaces of the simulation task 2. Table 6.3 also reports the total area overhead which includes not only SL wrappers but also StateMover's interruption logic. For the Memcached system, the total area also includes the area overhead of StateReveal which we use to enable capture of the buried state inside its registered BRAMs and DSPs.

For these four designs, the StateLink LUT + FF area overhead ranges from 5% to 13%, and the total area overhead added ranges from 6% to 16%. Since some of this overhead is fixed and some is per interface, we expect the percentage area overhead to be lower still for larger designs. Interestingly, for the second version of Memcached the StateLink area overhead was higher (because SL wrappers are added on the ten interfaces of Memcached pipeline), but the total area overhead remained almost the same. That is because the Memcached pipeline task does not have buried state (except one pipelined DSP) so StateReveal overhead is minimal, while the entire Memcached server has around 40 registered BRAMs.

The operating frequency of each of these designs is unaffected by the addition of StateLink and the interruption logic. The AES system and the DDR system run at 300 MHz. The Memcached system has two major clocks, a DDR clock (which also drives the AXI Data Mover blocks) of 300 MHz and an Ethernet clock (which drives the other major blocks) of 125 MHz.

One of the benefits of StateLink is that it enables simulation of only the part of a design being debugged (the task) while still being connected to the entire (hardware) system. We compare the speed of such a StateLink task simulation with the speed of *functional* simulation of the entire system. In both simulations, we use the functional post-implementation netlist for simulating the task. Using a behavioral representation of the task leads to a similar simulation time due to the complexity of the system. The simulation speed in each case is measured on an Intel Xeon E5-1620 v3 CPU running the full version of ModelSim 10.7a and is shown in Table 6.4. As shown in Table 6.4, StateLink never slows down the simulation; sending a transaction to hardware and getting the response back is usually much faster than performing the transaction in the simulator. As expected, StateLink speeds up the simulation since only a portion of the design is simulated (i.e., the task).

For the AES system, the simulation of the AES core encrypting the entire data set is sped up by 2.2×. When debugging the AES encryption core we run not only encryption but also the check core. In this case, StateLink's speedup increases to 4× as it can run the AES check core in hardware instead of simulation. For the DDR system, StateLink can speed up the simulation by 25× by simulating only the task and leaving the DDR transactions in hardware. It is worth noting that we are using a behavioral model for the DDR controller in the (traditional) entire system simulation. Performing a functional post-implementation simulation (which would include calibration) or a timing simulation

Table 6.4: Simulation and hardware speedup for the four full designs.

	Simulation Time	Simulation Speed	SL Simulation Time	StateLink Speedup	Hardware Speedup
AES System	31 min	2.8 kHz	14 min	2.2×	0.1 M×
DDR System	18 min	0.5 kHz	0.7 min	25×	0.5 M×
Memcached with task 1	66 min	0.4 kHz	8.5 min	7.8×	0.8 M×
Memcached with task 2	66 min	0.4 kHz	5.5 min	12×	0.8 M×

for the DDR core (which would add still more detail and timing models) instead would increase the simulation time of the entire system significantly, leading to higher speedup gains for using StateLink.

For Memcached with task 1, even though the task running in simulation is about 65% of the design logic, the StateLink simulation speedup is approximately 8×. For Memcached with task 2, the task is about 32% of the design logic, increasing the speedup to 12×. In general, the speedup of StateLink task simulation over traditional full system simulation depends on both the percentage of the design logic in the task and the complexity of the I/O and other IP blocks used in the design.

By combining StateLink with StateMover, we can further speed up debugging by running the *entire* design in hardware most of the time, and moving the task to StateLink simulation only for interesting (i.e., near a suspected bug) execution epochs. To show the speedup of this flow, we compared the full system simulation time of these designs to the hardware execution time. As Table 6.4 shows, hardware execution is 0.1-0.8 million times (M×) faster than simulation.

For the Memcached system, full system simulation uses an RTL test-bench to send 3000 Memcached requests and takes 66 minutes. The hardware can process the same 3000 requests in only 5 ms, for a 0.8M× speedup. With StateLink, we do not need to pre-compute the Memcached requests; instead, we can directly connect the hardware to the host CPU and have it send live Memcached requests as they are required by the larger application. In this case, 3000 Memcached requests were issued in 0.5 s by the host CPU (∼5 Mbps) and processed by the hardware, leading to a lower but still large 8k× speedup, despite the more flexible and realistic input stimulus.

The overall speedup experienced by an engineer performing debugging tasks will be a mix of the StateLink Speedup and the Hardware Speedup columns in Table 6.4. The Hardware Speedup of 0.1M× to 0.8M× will apply until the execution area of interest is approached, and then the StateLink speedup of 2.2× to 25× will apply.

6.7 Comparisons to Other Debugging and Verification Flows

In this section, we compare our proposed checkpoint-based debugging flow with previous scan-based and readback-based frameworks, and available trace-based vendor tools. First, we compare the StateMover flow to tools that can provide the user with full visibility. Table 6.5 summarizes the comparison in terms of visibility, controllability (ability to perform state changes), debugging speed, and area/timing overhead.

HDL simulation is the mainstay of FPGA debugging since it provides full visibility and controllability at no area or timing cost. It is crucial in initial design phases for logic testing of modules. However, as Table 6.5 shows, it is very slow, so for large complex systems, hardware debugging techniques such as trace-based and readback-based are also essential as they run at hardware speed.

Table 6.5: Comparison of different debugging frameworks that provide full visibility.

	HDL Simulation	[93, 92, 37, 94]	DESSERT [3]	This Work
Category	Simulation	Readback-based	Scan-based	Checkpoint-based
Visibility	Full	Full	Full	Full
Controllability	Full	Low	Moderate	High
Speed	Slowest	Moderate	Fast	Fast
Area Overhead	None	Low	High	Low
Timing Overhead	None	Low	High	Low

Readback-based frameworks (column 2 in Table 6.5) generally have minimal area overhead and no timing overhead since they use the readback hardware, which does not require additional resources. The added logic is mainly for stopping the design and allowing user interaction and is small for both prior work and StateMover. For example, gNOSIS [93], LLD [37], and AMIDAR [94] add 201 FFs, 816 FFs (204 logic slices), and 1340 FFs, respectively, to a design. StateMover has a comparable area overhead of ∼350 FFs for designs with no buried state. StateReveal allows StateMover to checkpoint and observe designs with registered BRAMs and pipelined DSP blocks, a capability that prior frameworks lacked. This comes at a moderate area overhead of ∼39 FFs per hard block with buried state.

In terms of debugging speed, StateMover is faster than previous readback-based frameworks in retrieving a snapshot of signal values. For example, LLD takes 0.5 seconds to retrieve a single bit's value, while StateMover can retrieve the 84k signal values of the Crossbar-big design (of Section 5.6.2) in less than 11 seconds. Moreover, to create a simulation-style waveform for debugging, previous readback-based frameworks have to stop the design every clock cycle, thereby losing the speed advantage of hardware execution. Instead of performing frequent readbacks, StateMover fully checkpoints the design and moves its state to the simulator where the design can be resumed with full visibility.

DESSERT [3] also proposed moving the design state from an FPGA to a simulator. It is a scan-based framework, and thus, has a logic overhead of 80% and reduces the design frequency by more than half. It uses the FIRRTL hardware compiler that performs several transformations on the design's RTL, thereby completely changing the mapping of the original circuit. Unlike DESSERT, StateMover can also move the design state from a simulator to an FPGA, which enables fast-forwarding the simulation in uninteresting periods and provides high controllability that allows running "what-if" tests in *hardware*. In addition, StateMover can create and load *complete* (by including buried state and external memory) and *consistent* (by safely stopping the design) checkpoints that can be used in areas other than debugging, such as live migration, context switching, and fault recovery.

Second, we compare our proposed debugging flow (StateMover + StateLink) with available trace-based vendor tools such Xilinx's ILA (Integrated Logic Analyzer) and Intel's SignalTap. Table 6.6 summarizes this comparison.

In trace-based tools, the user has to carefully select signals to monitor. The number of signals and the duration of their capture are limited by on-chip resources, and any change in these signals requires recompilation of the design. For example, in the debugging session described in Section 6.6.2, we would have needed to add Xilinx ILAs at the output of the Hash Table IP, inside the Hash Table IP, at the address and data buses of the AXI Data Mover connected to the Hash Table IP, and at

Table 6.6: Comparison of the trace-based flow and our proposed debugging flow.

	Trace-based	This Work
User Plan	Signal & Trigger Selection	Task & Breakpoint Selection
User Action	Add Traces on signals	Add TI/SL wrappers on interfaces
Design Perturbation	High	Low*
Area/Timing Overhead	High	Low
Speed	Fastest	Fast
Visibility	Limited	Full
Controllability	Very low	High
Cycle-Accurate at Interfaces	Yes	No

*To access buried state, StateReveal causes some design alteration; this step could be omitted in future FPGAs if readback is extended to all state.

the address bus of the AXI Data Mover connected to the Value Store IP, at least. The user has to connect each signal to the trace buffers by marking each signal/bus for debugging in the vendor tools, and then perform an FPGA compile. To evaluate the various hypotheses of 6.6.2 would have required multiple ILA insertions and time-consuming FPGA compiles.

On the other hand, in our checkpoint-based flow, the user has to decide only which part of the design to debug (the task), and it can be the entire design. Then, the user just needs to add the TI/SL wrappers on the interfaces of the task and connect them to the interruption logic. This can be done using the graphical user interface of vendor tools (Xilinx's IP Integrator).

Another concern in debugging is how much the debugging hardware itself perturbs the design (which might stop a bug from manifesting) and how much area and timing overhead is added. Trace-based tools generally perturb a design, as capturing new signals requires a recompile. The area and timing overhead is variable, depending on the number of signals captured and how large the trace window is. Being more selective in signals captured reduces area and timing overhead, but makes it more likely that multiple FPGA recompiles are necessary to observe sufficient debugging data. For example, we instrumented one of our test designs using Xilinx's ILA flow: the DFT-FP design (of Section 5.6.3) which is a mid-sized design with 50 kLUTs. In this experiment, we randomly selected 895 multi-bit signals (a total of ∼17k 1-bit signals) to monitor with a minimal sample depth of 1024. The selected 17k signals represent only 12% of the subset of signals that can be monitored using the ILA flow; this subset does not include BRAM contents nor BRAM/DSP buried registers as the ILA flow cannot directly access them. This instrumentation utilizes 64 kLUTs, 111 kFFs and 485 36Kb BRAMs. This represents a logic area overhead of 142% and a BRAM overhead of 94%; in total 85% of the device BRAM is used. Moreover, the FMAX is reduced by 65%. Changing the subset of monitored signals in the ILA flow would require re-running the place and route steps, which take 80 minutes to complete.

In our flow, design alteration is low since the wrappers are added only on the task interfaces. The netlist of the task itself is not changed unless the task contains buried state. In this case, we use StateReveal to add capture logic around each hard block that contains buried state (registered BRAMs and pipelined DSPs) to access it. As suggested in Section 4.5, a change to the FPGA configuration architecture that makes the internal registers of hard blocks accessible would have a small cost (0.5% more configuration bits) and would make checkpointing simpler and cheaper in future FPGAs. The total area overhead of our flow is low: 6-16% as shown in Table 6.3. There is no timing overhead for designs with no buried state, and a low timing overhead of less than 5% for

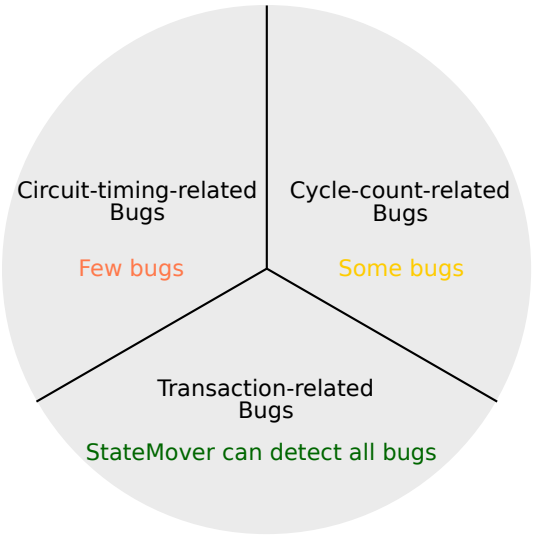


Figure 6.8: Bug Taxonomy and suitability for our checkpoint-based debugging flow.

designs with buried state.

In terms of speed, trace-based tools do not use simulation at all which make them the fastest debugging tool for FPGAs. However, when considering the multiple time-consuming FPGA compiles that might be needed, the overall debugging speed is adversely affected. In contrast to our flow, they also provide limited visibility and very low controllability, which significantly hurts debugging productivity.

Overall, StateMover and StateLink enables a software-like debugging flow where execution and signal examination are fast enough that the limiting factor is usually thinking about the next hypothesis, rather than waiting for FPGA recompiles or long simulation runs.

While StateMover and StateLink let us find many bugs faster, the relative slowness of simulation versus hardware affects the rate/timing of transactions on task interfaces, which means that the StateLink simulation is not cycle-accurate at these interfaces. This can hide (or expose) buffer overflow and cycle-count-dependent bugs. This limitation is mitigated by providing hardware time-matching discussed in Section 6.4.3 and Section 6.5.3. However, trace-based tools that do not change any event timings will still remain important for debugging these types of bugs. Hence, we believe the new checkpoint and co-simulation based debugging flow we have developed is an important complement to, rather than a replacement for, trace-based debugging techniques.

6.8 Debugging Flow Applicability

In this section, we discuss the types of bugs that can be uncovered using our proposed checkpoint-based debugging flow and those for which trace-based tools are more suitable. We classify hardware bugs into three categories as shown in Figure 6.8: circuit-timing-related, cycle-count-related and transaction-related.

Circuit-timing-related bugs are caused by timing issues such as metastability, clock-domain crossing issues, unsafe resets, and data races between clocks that were missed in the timing constraints. Activating these bugs depends on the precise delays within a circuit. Hence, they are hard to discover

during simulation, which usually runs without delays, and on-chip debugging such as trace-based tools are needed to uncover these bugs. Even with trace-based debugging, the inserted trace buffers alter the design netlist and can make some timing bugs disappear. Since our checkpoint-based debugging flow uses simulation, it is not ideally suited for debugging this bug type. However, there are some workarounds to use our flow for debugging these timing-related bugs. First, the user can perform a post-implementation *timing* simulation after the design state is moved to the simulator. Timing simulation can uncover some timing-related bugs, but since it does not fully capture the spectrum of different possible signal delays that occur in real chips with process, temperature, voltage and crosstalk variations, some bugs will remain undetected. The user can also dump both the hardware state and the simulation state at the same breakpoint and compare them to detect any discrepancies between hardware and simulation behavior as these mismatches are usually caused by timing-related issues.

Cycle-count-related bugs are design flaws that do not depend on the exact delays in a circuit, but do depend on the clock-cycle-level timing and/or rate of transactions at the inputs and outputs of a task. They include buffer overflow (an at-load effect), interface protocol issues, and mismatches in pipeline latencies. StateMover can help uncover these bugs as long as the entire design can be moved to the simulator. If only part of the design is moved and StateLink is used, the relative slowness of the simulation versus hardware can hide (or expose) cycle-dependent bugs as previously mentioned in Section 6.7. Thus, trace-based debugging which does not affect the timing of transactions/events is more suitable for these bugs. There are also two workarounds to use our flow for debugging cycle-related bugs. First, the user can use the hardware time-matching feature of StateLink to mimic the actual rate/timing of transactions on the interfaces of the simulation task. However, the rate/timing of transactions seen by the rest of the system will still be affected. The user can also use I/O traces in conjunction with StateMover to capture both the design state and its inputs to allow cycle-accurate simulation at the interfaces but for a limited number of cycles as discussed in Section 6.1.

The remaining design flaws can be grouped into transaction-based bugs which do not depend on circuit delays or the cycle-count. They include logic flaws either inside the task or in the connected system outside the FPGA, missing cases, and transaction ordering issues. They also include interactions of the entire system that the designer may have not thought about (i.e., conceptual bugs). Our proposed debugging flow is well suited for uncovering this bug type. With the combination of the speed of hardware execution and the visibility of simulation, it provides a more productive debugging flow than trace-based tools as discussed in Section 6.7.

While debugging flows are mainly designed for uncovering design flaws, they can also assist in debugging intermittent hardware faults like single-event upsets (SEUs). When a fault occurs, the user can leverage the checkpointing feature of our debugging flow by rolling back to a latest checkpoint and resuming the execution to see whether the error is persistent.

6.9 Summary

In this chapter, we developed StateLink, a transaction-based co-simulation framework which enables a checkpoint-based system debugging flow for FPGAs. It allows part of the system (task) to be moved from hardware to a simulator while remaining connected to other system components running in hardware. This realizes a promising debugging flow, where the design runs most of the time in

hardware, and when full visibility is needed only the task has to be moved to a simulator. Thus, the user benefits from both the hardware speedup of ∼1M× over simulation and the StateLink simulation speedup of up to 25× over full system simulation. StateLink currently supports tasks with AXI memory-mapped and AXI streaming interfaces and can communicate with FPGA boards using either JTAG or Ethernet. It can also mimic hardware timing by timestamping the hardware transactions, which is useful for capturing cycle-dependent bugs. It has a modest area overhead of 5-13% and typically no timing overhead. StateLink and StateMover enable fast diagnosis of faults and testing of hypotheses, enabling a work flow that has many of the desirable features of software debugging.

Chapter 7

Conclusion and Future Work

Hardware checkpointing is crucial for enabling live migration, fault recovery and context switching. In this thesis, we have tackled the major challenges that face hardware checkpointing on FPGAs. We have provided rules and IPs that can achieve safe task interruption, which enables the creation of *consistent* checkpoints. We have also developed techniques and an open-source tool (StateReveal) that can capture buried on-chip state, which enables loading and creating *complete* checkpoints.

In addition, we have leveraged hardware checkpointing to build an open-source checkpoint-based debugging framework (StateMover) that can move the design state back and forth between an FPGA and a simulator. This enables a new debugging flow for FPGAs that combines the speed of hardware execution and the visibility of simulation. The resulting improvement in FPGA debugging productivity is sorely needed as currently 51% of FPGA project time is spent on verification and debugging. Moreover, we have presented an open-source transaction-based co-simulation framework (StateLink) that extends the checkpoint-based debugging flow to designs with external I/Os. It allows only a part of the design to be moved to the simulator while remaining connected to the rest of the system. This enables a software-like debugging flow for any FPGA design with AXI interfaces and can be extended to other interfaces with valid/ready signals.

In this chapter, we discuss the key conclusions of this thesis and possible directions for future work.

7.1 Safe Interruption

We have identified the hazards that could arise from stopping a running task. We have classified them into data loss, data corruption, task deadlock, and system lockup hazards. The main reasons for these interruption hazards are in-flight transactions and multiple clock domains. After identifying the possible hazards, we derived design rules to achieve safe task interruption. In addition, we designed task wrappers that can be placed around a task to safely stop it based on the design rules. The wrappers support both AXI and Avalon standard interfaces and have a small area: a complete Memcached design can be safely stopped with 1.8% area overhead and no timing overhead.

While we have listed all the task interruption hazards of which we are aware and verified solutions for each using extensive simulations and some hardware designs, a useful future work would be to use formal techniques to verify the completeness and sufficiency of our proposed rules and wrappers.

While the wrappers can be easily inserted in the design, creating a tool that automatically analyzes a task and adds the relevant wrappers would streamline the process.

7.2 Accessing Buried State

DSP pipeline registers, BRAM input registers and interconnect registers cannot be read out or written back using the configuration port. Checkpointing a design that uses any of these elements leads to an incomplete checkpoint. In this thesis, we have proposed techniques that can create and load complete checkpoints even when the design contains buried state. We have also developed StateReveal, a tool that detects buried state in the design and automatically inserts the capture registers and the required logic to add controllability and visibility to the buried state. StateReveal has a timing overhead of less than 5%. The area overhead depends on the number of hard blocks that contain buried state and on their data width. On average, StateReveal adds 25 LUTs and 39 FFs per hard block.

Currently, StateReveal inserts RTL code in the synthesized netlist to add the capture logic. The added code needs to be mapped to hardware primitives and thus, the netlist is then re-synthesized. Modifying StateReveal to directly insert hardware primitives (i.e., mapped logic) would omit this step, and the modified netlist can be directly implemented. Another future work is to explore the usage of RapidWright [124] in StateReveal to be able to insert the capture logic in the placed design, rather than the synthesized netlist. This approach would cause less perturbation to the design, but could also cause routing or timing failures. To further reduce the area overhead of StateReveal, the inserted capture logic can be modified to capture only a subset of the buried state and then, data reconstruction techniques [125] can be leveraged to regenerate the missing buried state from the captured one.

7.3 Checkpoint-based Debugging

In this thesis, we have proposed a new debugging flow that achieves the best of two worlds: the visibility of a simulator and the speed of hardware execution. We have built StateMover, a checkpoint-based debugging framework that realizes this flow. StateMover can safely stop a running design and seamlessly move its state back and forth between an FPGA and a simulator. It allows fast forwarding uninteresting periods of the simulation given the $\sim$10M$\times$ speedup of hardware execution. StateMover can create complete design checkpoints even for designs that have multi-cycle interfaces, contain buried state, and use external memories. StateMover allows a designer to quickly make a design checkpointable with a small area overhead. Moving the state from/to an FPGA to/from a simulator can be performed in a few seconds for large Xilinx UltraScale FPGAs.

Using PCIe instead of JTAG to access the configuration architecture for reading and writing back the design state would greatly reduce the time required to capture or restore the state and would allow taking more frequent checkpoints. It would also extend StateMover to FPGAs in data centers which do not have JTAG access. Another possible future work is to support FPGA designs that use a hard processor in which the program running on the processor side is stopped and checkpointed along with the programmable logic side. It would also be interesting to build a graphical user interface that allows the user to navigate between the checkpoints that are taken during hardware

execution, going forward and backward between them as in the record and replay flow provided by recent software debuggers. In addition, a promising future work is to port this flow to Intel FPGAs that have readback capability.

7.4 Transaction-based Co-simulation

In this thesis, we have shown that checkpoint-based debugging can be extended to designs with external I/Os including DRAM and Ethernet and designs that cannot be moved entirely to the simulator such as CPU+FPGA accelerators or datacenter-scale applications. This can be done using StateLink, a transaction-based co-simulation framework that allows a task running in a simulator to interact with other design elements that reside in hardware. This enables a promising debugging flow, where the design runs most of the time on hardware, and when full visibility is needed, only part of the design has to be moved to a simulator. Thus, the user benefits from both the hardware speedup of ∼1M× over simulation and the StateLink simulation speedup of up to 25× over full system simulation. Incorporating StateLink in a design typically adds no timing overhead and a modest hardware area overhead of 5-13%.

StateLink currently supports tasks with AXI memory-mapped and streaming interfaces. Adding support for additional standard interfaces with ready/valid signals such as Avalon should be straightforward. Another promising future work is to leverage the transaction-based links created by StateLink to enable multi-chip designs for ASIC prototyping.

7.5 Future FPGA Debugging

FPGAs are becoming ever more heterogeneous and increasingly are entire systems-on-chip or system-in-package with not only FPGA fabrics, but also processor subsystems, massively parallel processor arrays and embedded networks-on-chip. A key challenge is debugging these entire systems, and for this thesis a key question is how the checkpoint-based debugging framework we have created can be used in or enhanced for such systems.

Hard processors are becoming more common in FPGA systems. For example, Intel provides system-on-chip (SoC) FPGA devices that contain an ARM processor, and are available with Arria 10 and Agilex FPGAs; the Xilinx Zynq series provides similar capabilities. The proposed checkpoint-based debugging flow can significantly help in debugging designs that use a hard processor. All or part of the design that is implemented on the FPGA can be moved to the simulator and by using transaction-based co-simulation (StateLink), we can keep the program running in the hard processor connected to the simulation task. This allows the user to use the software debugger available with the hard processor while having full visibility of either a key fabric task or the entire portion of the design in the FPGA fabric. This is similar to the Memcached system described in Section 6.6 where the Memcached server running in the simulator is kept connected to the Memcached client on another computer.

In addition to hard processors, complex hard acceleration blocks such as the AI engines in Xilinx Versal FPGAs are also getting more attention. Similarly to the hard processor case, transaction-based co-simulation also works well for designs that use these blocks since we can keep the accelerator blocks running on the FPGA while having full visibility of the rest of the design. We expect one or

both of hardware debugging hooks (e.g. the ability to single step or set breakpoints in the accelerator blocks) and bespoke simulators for these blocks will be necessary to debug them productively. In either case, StateLink and StateMover will allow the accelerator complex to be connected to our debugging framework for the FPGA fabric resources in order to debug the entire system.

Several recent FPGAs, including the Xilinx Versal and Achronix Speedster 7t families, incorporate a hard network-on-chip (NoC) as a system-level interconnect to connect hard I/O interfaces, accelerator complexes and many points in the FPGA fabric [49]. This poses a new challenge for FPGA debugging. To debug a task connected to the NoC with real-world stimuli in the simulator, we need to simulate the entire system including the NoC, which will further strain already slow system-level simulation speeds. Fortunately, connection to the NoC uses standard AXI interfaces, making every connection to the NoC an appropriate boundary for a StateLink/StateMover task. Hence, movement of tasks into StateLink simulation while the NoC and the rest of the system remain in hardware is straightforward. Moreover, the timestamp feature described in Section 6.4.3 and Section 6.5.3 can be leveraged to vary the transaction timing, enabling testing and debugging the task at different traffic loads. However, at-load testing and NoC livelock and deadlock debugging are harder to perform using this flow due to the relative slowness of simulation versus hardware – the NoC traffic will be impacted if a major task is moved to simulation. For these type of tests/bugs, trace-based debugging remains more suitable.

In general, the debugging flow proposed in this work provides the user with two options to debug these heterogeneous systems. First, the user can move only the fabric-based portion of the design to the simulator and then use StateLink to connect the simulation with the hardware execution of the hard cores (e.g., hard processors). This provides the user with full visibility of the fabric-based portion of the design and the hard cores can be treated as black boxes. The user can also leverage the software debuggers available with these hard cores to achieve full visibility of the entire system. Second, the user can use StateMover to capture the state of the entire system, if possible, and then use simulation models to simulate the entire system including the hard cores.

In order to better quantify the gains of our proposed debugging flow versus existing flows, a potential future work is to carry out a user study to measure the debugging speed and productivity for the different flows on a benchmark suite of designs. Ideally, this study would also be performed on different FPGA SoCs like those previously mentioned in this section. It should also test the debugging productivity of combining different flows together since a specific debugging flow can be a better fit for uncovering a certain bug type as we have shown in Section 6.8 and the combination of multiple tools helps in finding the root cause of bugs faster. For example, the user can combine our checkpoint-based flow with the trace-based flow to obtain not only complete visibility of the design state starting from the captured checkpoint but also partial visibility of the design state prior to that checkpoint by leveraging the signal history feature of trace-based tools.

Bibliography

[1] Harry Foster. *2020 Wilson Research Group functional verification study: FPGA functional verification trend report*. White Paper. Wilson Research Group and Mentor, A Siemens Business, 2020.

[2] Donggyu Kim et al. "Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL". In: *International Symposium on Computer Architecture (ISCA)*. 2016, pp. 128–139. DOI: 10.1109/ISCA.2016.21.

[3] Donggyu Kim et al. "DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 76–79. DOI: 10.1109/FPL.2018.00021.

[4] Sameh Asaad et al. "A Cycle-accurate, Cycle-reproducible multi-FPGA System for Accelerating Multi-core Processor Simulation". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. 2012, pp. 153–162. DOI: 10.1145/2145694.2145720.

[5] Jiacheng Ma et al. "Debugging in the Brave New World of Reconfigurable Hardware". In: *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2022, pp. 946–962. DOI: 10.1145/3503222.3507701.

[6] Robert O'Callahan et al. "Engineering Record and Replay for Deployability". In: *USENIX Annual Technical Conference*. 2017, pp. 377–389.

[7] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. "Migration of Long-running Tasks Between Reconfigurable Resources Using Virtualization". In: *ACM SIGARCH Computer Architecture News* 44.4 (2017), pp. 56–61. DOI: 10.1145/3039902.3039913.

[8] Dirk Koch, Christian Haubelt, and Jürgen Teich. "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. 2007. DOI: 10.1145/1216919.1216950.

[9] H. Kalte and M. Porrmann. "Context Saving and Restoring for Multitasking in Reconfigurable Systems". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2005, pp. 223–228. DOI: 10.1109/FPL.2005.1515726.

[10] Sameh Attia and Vaughn Betz. "Feel Free to Interrupt: Safe Task Stopping to Enable FPGA Checkpointing and Context Switching". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.1 (2020).

[11] Sameh Attia and Vaughn Betz. "Stop and Look: A Novel Checkpointing and Debugging Flow for FPGAs". In: *IEEE Transactions on Computers* (2022), pp. 1–1.

[12] Sameh Attia and Vaughn Betz. "Toward Software-Like Debugging for FPGAs via Checkpointing and Transaction-Based Co-Simulation". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2022). Under Review.

[13] Sameh Attia and Vaughn Betz. "StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. 2020, pp. 175–185.

[14] Sameh Attia and Vaughn Betz. "StateReveal: Enabling Checkpointing of FPGA Designs with Buried State". In: *International Conference on Field-Programmable Technologies (FPT)*. 2020.

[15] Sameh Attia and Vaughn Betz. "StateLink: FPGA System Debugging via Flexible Simulation/Hardware Integration". In: *International Conference on Field-Programmable Technologies (FPT)*. 2021.

[16] Microsoft. *Improving Azure Virtual Machine resiliency with predictive ML and live migration*. 2018. URL: https://azure.microsoft.com/en-us/blog/improving-azure-virtual-machine-resiliency-with-predictive-ml-and-live-migration/.

[17] Google. *Live Migration*. 2022. URL: https://cloud.google.com/compute/docs/instances/live-migration.

[18] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services". In: *International Symposium on Computer Architecuture (ISCA)*. 2014. DOI: 10.1145/2678373.2665678.

[19] Amazon Web Services. *Amazon EC2 F1 Instances*. 2017. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[20] Alibaba Cloud ECS. *Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances*. 2018. URL: https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057.

[21] Anuj Vaishnav, Khoa Pham, and Dirk Koch. "Live Migration for OpenCL FPGA Accelerators". In: *International Conference on Field-Programmable Technology (FPT)*. 2018. DOI: 10.1109/FPT.2018.00017.

[22] Daniel Ly-Ma. "Live Migration of FPGA Applications". MA thesis. University of Toronto, 2019.

[23] Hoang-Gia Vu, Takashi Nakada, and Yasuhiko Nakashima. "Efficient hardware task migration for heterogeneous FPGA computing using HDL-based checkpointing". In: *Integration* 77 (2021), pp. 180–192. DOI: https://doi.org/10.1016/j.vlsi.2020.11.011.

[24] S. Punnekkat and A. Burns. "Analysis of checkpointing for schedulability of real-time systems". In: *International Workshop on Real-Time Computing Systems and Applications*. 1997, pp. 198–205. DOI: 10.1109/RTCSA.1997.629219.

[25] Kiwan Maeng and Brandon Lucia. "Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 129–144.

[26] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. "NvMR: Non-Volatile Memory Renaming for Intermittent Computing". In: *International Symposium on Computer Architecture (ISCA)*. 2022. DOI: 10.1145/3470496.3527413.

[27] Maria Chtepen et al. "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids". In: *IEEE Transactions on Parallel and Distributed Systems* 20.2 (2009), pp. 180–190. DOI: 10.1109/TPDS.2008.93.

[28] Aitzan Sari, Mihalis Psarakis, and Dimitris Gizopoulos. "Combining checkpointing and scrubbing in FPGA-based real-time systems". In: *2013 IEEE 31st VLSI Test Symposium (VTS)*. 2013, pp. 1–6. DOI: 10.1109/VTS.2013.6548910.

[29] Marc Perelló Bacardit, Leonardo Bautista-Gomez, and Osman Unsal. "FPGA Checkpointing for Scientific Computing". In: *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2021, pp. 1–7. DOI: 10.1109/IOLTS52814.2021.9486693.

[30] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd. Addison-Wesley Professional, 2014.

[31] J.E. Smith and A.R. Pleszkun. "Implementing precise interrupts in pipelined processors". In: *IEEE Transactions on Computers* 37.5 (1988), pp. 562–573. DOI: 10.1109/12.4607.

[32] A. Morales-Villanueva and A. Gordon-Ross. "On-chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2013, pp. 61–64.

[33] Alban Bourge, Olivier Muller, and Frédéric Rousseau. "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10.1 (2016), 9:1–9:23. DOI: 10.1145/2996199.

[34] Kaan Kara and Gustavo Alonso. "PipeArch: Generic and Context-Switch Capable Data Processing on FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14.1 (2020). DOI: 10.1145/3418465.

[35] Arif Sasongko et al. "Hardware Context Switch-Based Cryptographic Accelerator for Handling Multiple Streams". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14.3 (2021). DOI: 10.1145/3460941.

[36] H. Simmler, L. Levinson, and Reinhard Männer. "Multitasking on FPGA Coprocessors". In: *International Workshop on Field-Programmable Logic and Applications (FPL)*. 2000, pp. 121–130.

[37] Yousef Iskander, Cameron Patterson, and Stephen Craven. "High-Level Abstractions and Modular Debugging for FPGA Design Validation". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.1 (2014), 2:1–2:22. DOI: 10.1145/2567662.

[38] M. Moudgill and S. Vassiiadis. "Precise interrupts". In: *IEEE Micro* 16.1 (1996), pp. 58–67. DOI: 10.1109/40.482313.

[39] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014.

[40] Markus Happe, Andreas Traber, and Ariane Keller. "Preemptive Hardware Multitasking in ReconOS". In: *International Symposium on Applied Reconfigurable Computing (ARC)*. 2015.

[41] Anuj Vaishnav et al. “Resource Elastic Virtualization for FPGAs using OpenCL”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2018. DOI: 10.1109/FPL.2018.00028.

[42] T. Xia, J. Prévotet, and F. Nouvel. “Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators”. In: *International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 44–52. DOI: 10.1109/FPT.2016.7929187.

[43] C. Huang et al. “Dynamically Swappable Hardware Design in Partially Reconfigurable Systems”. In: *International Symposium on Circuits and Systems (ISCAS)*. 2007, pp. 2742–2745. DOI: 10.1109/ISCAS.2007.378620.

[44] Hoang Gia Vu et al. “CPRtree: A Tree-Based Checkpointing Architecture for Heterogeneous FPGA Computing”. In: *International Symposium on Computing and Networking (CANDAR)*. 2016, pp. 57–66. DOI: 10.1109/CANDAR.2016.0024.

[45] B. L. Hutchings and B. E. Nelson. “Unifying simulation and execution in a design environment for FPGA systems”. In: *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 9.1 (2001), pp. 201–205.

[46] Aurelio Morales-Villanueva, Rohit Kumar, and Ann Gordon-Ross. “Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs”. In: *Design, Automation and Test in Europe Conference Exhibition (DATE)*. 2016, pp. 1505–1508.

[47] Stephanie Tapp. *XAPP1230: Configuration Readback Capture in UltraScale FPGAs*. Xilinx, Inc. 2015.

[48] *Libero Design Flow User Guide*. Microchip, Inc. 2022.

[49] Andrew Boutros and Vaughn Betz. “FPGA Architecture: Principles and Progression”. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[50] Brian Gaide et al. “Xilinx Adaptive Compute Acceleration Platform: Versal Architecture”. In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2019, pp. 84–93. DOI: 10.1145/3289602.3293906.

[51] *Vivado Design Suite User Guide: Partial Reconfiguration*. Xilinx, Inc. 2018.

[52] *Quartus Prime Pro Edition User Guide: Partial Reconfiguration*. Intel, Inc. 2019.

[53] *PG227: Partial Reconfiguration Decoupler*. Xilinx, Inc. 2016.

[54] *PG193: Partial Reconfiguration Controller*. Xilinx, Inc. 2018.

[55] *PG305: Partial Reconfiguration AXI Shutdown Manager*. Xilinx, Inc. 2018.

[56] Harry Foster. *2020 Wilson Research Group functional verification study: IC/ASIC functional verification trend report*. White Paper. Wilson Research Group and Mentor, A Siemens Business, 2020.

[57] Tobias Drewes, Jan Moritz Joseph, and Thilo Pionteck. “An FPGA-based prototyping framework for Networks-on-Chip”. In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2017, pp. 1–7. DOI: 10.1109/RECONFIG.2017.8279775.

[58] *Cadence Palladium Z1 Enterprise Emulation Platform*. Cadence Design Systems, Inc. 2015.

[59] Synopsys Inc. *ZeBu EP1*. 2022. URL: https://www.synopsys.com/verification/emulation/zebu-ep1.html.

[60] *Protium S1 FPGA-Based Prototyping Platform*. Cadence Design Systems, Inc. 2019.

[61] *Veloce Primo Enterprise Prototyping Solution*. Siemens Digital Industries Software. 2021.

[62] Synopsys Inc. *ZeBu Server 4*. 2022. URL: https://www.synopsys.com/verification/emulation/zebu-server.html.

[63] Charley Selvidge and Vijay Chobisa. *The Veloce Strato Platform: Unique core components create high value advantages*. White Paper. Siemens Digital Industries Software, 2020.

[64] *PG172: Integrated Logic Analyzer*. Xilinx, Inc. 2016.

[65] *Chipscope Pro Software and Cores: User Guide*. Xilinx, Inc. 2012.

[66] *Quartus Prime Handbook Volume 3: Verification*. Intel, Inc. 2015.

[67] Eddie Hung and Steven J. E. Wilton. "Scalable Signal Selection for Post-Silicon Debug". In: *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 21.6 (2013), pp. 1103–1115. DOI: 10.1109/TVLSI.2012.2202409.

[68] Jeffrey Goeders and Steven J. E. Wilton. "Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 36.1 (2017), pp. 83–96. DOI: 10.1109/TCAD.2016.2565204.

[69] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. "Source-level debugging for FPGA high-level synthesis". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927496.

[70] Jeffrey Goeders and Steven J.E. Wilton. "Effective FPGA debug for high-level synthesis generated circuits". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927498.

[71] Daniel Holanda Noronha et al. "On-chip FPGA Debug Instrumentation for Machine Learning Applications". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. 2019, pp. 110–115. DOI: 10.1145/3289602.3293922.

[72] David Sidler and Ken Eguro. "Debugging framework for FPGA-based soft processors". In: *International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 165–168. DOI: 10.1109/FPT.2016.7929524.

[73] Daniel Holanda Noronha et al. "Flexible Instrumentation for Live On-Chip Debug of Machine Learning Training on FPGAs". In: *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, pp. 20–28. DOI: 10.1109/FCCM51124.2021.00011.

[74] Eddie Hung and Steven J. E. Wilton. "Incremental Trace-Buffer Insertion for FPGA Debug". In: *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 22.4 (2014), pp. 850–863. DOI: 10.1109/TVLSI.2013.2255071.

[75] Brad L. Hutchings and Jared Keeley. "Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2014, pp. 72–79. DOI: 10.1109/FCCM.2014.29.

[76] Fatemeh Eslami and Steven J. E. Wilton. "Incremental distributed trigger insertion for efficient FPGA debug". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–4. DOI: 10.1109/FPL.2014.6927418.

[77] Alexandra Kourfali and Dirk Stroobandt. "Efficient Hardware Debugging Using Parameterized FPGA Reconfiguration". In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 277–282. DOI: 10.1109/IPDPSW.2016.95.

[78] Pavan K. Bussa, Jeffrey Goeders, and Steven J. E. Wilton. "Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056800.

[79] Robert Hale and Brad Hutchings. "Enabling Low Impact, Rapid Debug for Highly Utilized FPGA Designs". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 81–84. DOI: 10.1109/FPL.2018.00022.

[80] Zissis Poulos et al. "Leveraging reconfigurability to raise productivity in FPGA functional debug". In: *Design, Automation and Test in Europe Conference Exhibition (DATE)*. 2012, pp. 292–295. DOI: 10.1109/DATE.2012.6176481.

[81] Eddie Hung and Steven J.E. Wilton. "Towards Simulator-like Observability for FPGAs: A Virtual Overlay Network for Trace-Buffers". In: *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2013, pp. 19–28. DOI: 10.1145/2435264.2435272.

[82] Fatemeh Eslami and Steven J. E. Wilton. "An adaptive virtual overlay for fast trigger insertion for FPGA debug". In: *International Conference on Field Programmable Technology (FPT)*. 2015, pp. 32–39. DOI: 10.1109/FPT.2015.7393127.

[83] Fatemeh Eslami and Steven J.E. Wilton. "An Improved Overlay and Mapping Algorithm Supporting Rapid Triggering for FPGA Debug". In: *ACM SIGARCH Computer Architecture News* 44.4 (2017), pp. 20–25. DOI: 10.1145/3039902.3039907.

[84] Al-Shahna Jamal, Jeffrey Goeders, and Steven J. E. Wilton. "An FPGA Overlay Architecture Supporting Rapid Implementation of Functional Changes during On-Chip Debug". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 403–410. DOI: 10.1109/FPL.2018.00076.

[85] Al-Shahna Jamal, Jeffrey Goeders, and Steven J.E. Wilton. "Architecture Exploration for HLS-Oriented FPGA Debug Overlays". In: 2018, pp. 209–218. DOI: 10.1145/3174243.3174254.

[86] Daniel Holanda Noronha et al. "An Overlay for Rapid FPGA Debug of Machine Learning Applications". In: *International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 135–143. DOI: 10.1109/ICFPT47387.2019.00024.

[87] Timothy Wheeler et al. "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2001, pp. 483–492.

[88] Chin-Lung Chuang, Dong-Jung Lu, and C.-N.J. Liu. "A snapshot method to provide full visibility for functional debugging using FPGA". In: *Asian Test Symposium (ATS)*. 2004, pp. 164–169. DOI: 10.1109/ATS.2004.15.

[89] Chin-Lung Chuang et al. "Hybrid Approach to Faster Functional Verification with Full Visibility". In: *IEEE Design and Test of Computers* 24.2 (2007), pp. 154–162. DOI: 10.1109/MDT.2007.46.

[90] A. Tiwari and K.A. Tomko. "Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs". In: *IEEE Asia and South Pacific Design Automation Conference (ASPDAC)*. 2003, pp. 705–711. DOI: 10.1109/ASPDAC.2003.1195112.

[91] A. Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[92] Hari Angepat et al. "NIFD: Non-intrusive FPGA Debugger – Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2010, pp. 356–359. DOI: 10.1109/FPL.2010.77.

[93] Ashfaquzzaman Khan, Richard N. Pittman, and Alessandro Forin. "gNOSIS: A Board-Level Debugging and Verification Tool". In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2010, pp. 43–48. DOI: 10.1109/ReConFig.2010.71.

[94] Changgong Li, Alexander Schwarz, and Christian Hochberger. "A readback based general debugging framework for soft-core processors". In: *International Conference on Computer Design (ICCD)*. 2016, pp. 568–575. DOI: 10.1109/ICCD.2016.7753342.

[95] Georgios Tzimpragos et al. "Application debug in FPGAs in the presence of multiple asynchronous clocks". In: *International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 189–192. DOI: 10.1109/FPT.2016.7929530.

[96] Sadegh Yazdanshenas and Vaughn Betz. "Quantifying and Mitigating the Costs of FPGA Virtualization". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056807.

[97] Anuj Vaishnav, Khoa Pham, and Dirk Koch. "A Survey on FPGA Virtualization". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2018. DOI: 10.1109/FPL.2018.00031.

[98] Stuart Sutherland. *The Verilog PLI Handbook*. Kluwer Academic Publishers, 2002.

[99] Sadegh Yazdanshenas and Vaughn Betz. "Improving Confidentiality in Virtualized FPGAs". In: *International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 258–261.

[100] L. Gong and O. Diessel. "ReSim: A Reusable Library for RTL Simulation of Dynamic Partial Reconfiguration". In: *International Conference on Field-Programmable Technology (FPT)*. 2011, pp. 1–8. DOI: 10.1109/FPT.2011.6132709.

[101] Lingkan Gong and Oliver Diessel. "Functionally Verifying State Saving and Restoration in Dynamically Reconfigurable Systems". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. 2012, pp. 241–244. DOI: 10.1145/2145694.2145735.

[102] Reed P Tidwell. *Alpha Blending Two Data Streams Using a DSP48 DDR Technique*. Xilinx, Inc. 2005.

[103] *Advanced Synthesis Cookbook*. Altera/Intel. 2011.

[104] Dormando. *Memcached - A Distributed Memory Object Caching System*. 2018. URL: http://memcached.org/.

[105] Jongsok Choi et al. "Accelerating Memcached on AWS Cloud FPGAs". In: *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*. 2018, 2:1–2:8. DOI: 10.1145/3241793.3241795.

[106] Xilinx. *HLS implementation of Memcached pipeline*. 2016. URL: https://github.com/Xilinx/HLx_Examples/tree/master/Acceleration/memcached.

[107] *PG022: AXI DataMover v5.1*. Xilinx, Inc. 2017.

[108] Khoa Dang Pham, Edson Horta, and Dirk Koch. "BITMAN: A Tool and API for FPGA Bitstream Manipulations". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017, pp. 894–897. DOI: 10.23919/DATE.2017.7927114.

[109] Shinya Takamaeda-Yamazaki. "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL". In: *International Symposium on Applied Reconfigurable Computing (ARC)*. 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42.

[110] *UG573: UltraScale Architecture Memory Resources*. Xilinx, Inc. 2019.

[111] *UG579: UltraScale Architecture DSP48E2 Slice*. Xilinx, Inc. 2019.

[112] *Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics*. Xilinx, Inc. 2019.

[113] Xilinx. *Open Source HLx Examples*. 2016. URL: https://github.com/Xilinx/HLx_Examples/.

[114] F. Franchetti et al. "SPIRAL: Extreme Performance Portability". In: *Proceedings of the IEEE* 106.11 (2018).

[115] *PG187: UltraScale Architecture Soft Error Mitigation Controller*. Xilinx, Inc. 2019.

[116] Ibrahim Ahmed et al. "Measure twice and cut once: Robust dynamic voltage scaling for FPGAs". In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–11. DOI: 10.1109/FPL.2016.7577342.

[117] *Fast Partial Reconfiguration Over PCI Express*. Xilinx, Inc. 2019.

[118] *UG1483: Model Composer and System Generator User Guide*. Xilinx, Inc. 2020.

[119] Sangjun Yang et al. "A new RTL debugging methodology in FPGA-based verification platform". In: *IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (APASIC)*. 2004, pp. 180–183. DOI: 10.1109/APASIC.2004.1349442.

[120] X. Cheng et al. "A run-time RTL debugging methodology for FPGA-based co-simulation". In: *International Conference on Communications, Circuits and Systems (ICCCAS)*. 2010, pp. 891–895. DOI: 10.1109/ICCCAS.2010.5581847.

[121] Chung-Yang Huang et al. “SoC HW/SW verification and validation”. In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. 2011, pp. 297–300. DOI: 10.1109/ASPDAC.2011.5722202.

[122] Somnath Banerjee and Tushar Gupta. “Fast and scalable hybrid functional verification and debug with dynamically reconfigurable co-simulation”. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2012, pp. 115–122.

[123] *PG174: JTAG to AXI Master*. Xilinx, Inc. 2021.

[124] C. Lavin and A. Kaviani. “RapidWright: Enabling Custom Crafted Implementations for FPGAs”. In: *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 133–140.

[125] Ho Fai Ko and Nicola Nicolici. “Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 28.2 (2009), pp. 285–297. DOI: 10.1109/TCAD.2008.2009158.