StateReveal: Enabling Checkpointing of FPGA Designs with Buried State

Sameh Attia
University of Toronto
sameh.attia@mail.utoronto.ca

Vaughn Betz
University of Toronto
vaughn@eecg.utoronto.ca

2020 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-2302-1/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICFPT51103.2020.00036

***Abstract*—The ability to save and restore the entire state of an FPGA design is essential for hardware checkpointing, which enables live migration, fault recovery, and novel debugging flows. However, hard blocks such as block RAMs and DSP blocks have state elements that cannot be saved or restored using conventional checkpointing techniques such as readback. This results in an incomplete checkpoint, thereby preventing checkpointing of most FPGA designs. In this paper, we propose general techniques that allow a complete checkpoint to be captured and loaded even when a design contains inaccessible state elements. The proposed techniques include using multi-cycle capture and inserting capture registers that add observability and controllability of this buried state. Moreover, we present StateReveal, a tool that detects the presence of inaccessible state elements in a design and automatically inserts the required capture registers. This enables checkpointing of designs that contain fully registered block RAMs and pipelined DSP blocks, which was not previously feasible. We verify the ability to capture and load a complete checkpoint on several designs using an enhanced checkpointing framework that supports multi-cycle capture. StateReveal currently supports Xilinx FPGAs, and the added circuitry has a low timing overhead of 5% and an acceptable area overhead that averages 19 LUTs and 35 registers per hard block.**

***Index Terms*—checkpointing, readback, writeback, debugging, design state, task interruption**

I. Introduction

Hardware checkpointing, in which a snapshot of the current state of an FPGA task is taken, is becoming more crucial for various reasons. First, being able to save and restore the state of an FPGA task enables live migration of tasks in data centers from one server to another to balance loads or adapt to failures [1]. Second, hardware checkpointing 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 [2]. Third, creating and loading checkpoints also enable 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 [3].

Finally, hardware checkpointing enables new FPGA debugging flows, which are sorely needed as design verification typically occupies 50% of FPGA project time [4]. Checkpoints allow reproducible experiments and analyses. These are very beneficial for debugging intermittent faults, particularly in a data center where thousands of CPUs and FPGAs may be collaborating [5]. A checkpointing-based debugging framework for FPGAs was recently proposed in [6]. It can move the state of a task back and forth between an FPGA and a simulator, which enables a new FPGA debugging flow that combines simulation observability and hardware execution speed.

A checkpoint consists of the value of all state holders inside a task, including task registers and memories. There are two main techniques for hardware checkpointing. The first one is to add additional state access hardware to the task such as scan chains. However, the overhead of this technique is significant; the logic overhead of the *shadow scan chains* method is close to 100% and it only enables saving and restoring the state of soft logic registers [2]. Readback is an alternative commonly-used technique for accessing the on-chip state. It is an attractive option as it does not require adding special access hardware to the design; instead it uses the FPGA configuration port to read back the current values of on-chip state elements. It allows read out of state elements that can be initialized in the configuration bitstream: CLB/ALM registers and block RAM contents. Readback is available on Xilinx FPGAs and Intel's Stratix 10 FPGAs. Bitstream manipulation can be used in conjunction with readback to restore a checkpoint [7], [8].

Unfortunately, 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 Stratix 10 hyper registers. Consequently, attempting to read back the state of a task that uses a fully registered BRAM, a pipelined DSP block, or a Stratix 10 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. As the use of these inaccessible registers is pervasive in FPGA designs, this poses 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. We verify the ability to capture and load a complete checkpoint in hardware on

978-1-6654-2302-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICFPT51103.2020.00036

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.

This paper is organized as follows. Section II reviews hardware checkpointing methods. Section III discusses the general approaches to handle inaccessible state elements. Section IV presents StateReveal and provides techniques to save and restore the state of a design with BRAMs and pipelined DSP blocks. The proposed techniques are evaluated on several designs in Section V. Section VI concludes the paper and presents some recommendations for future FPGA architectures.

II. Related Work

Hardware checkpointing has been used to enable live migration [1], [9], context switching [3], [7], [10], and FPGA debugging [6], [11], [12]. Saving and restoring the design state can be performed by adding state access hardware (e.g. scan chains) [2], [10], [12], which has high area overhead. It can also be performed by using the existing configuration architecture (readback and bitstream manipulation) [1], [3], [6], [11].

However, hardware checkpointing has two challenges that, if not solved, limit the set of designs that can be checkpointed [13]. The first challenge is how to safely stop a design with multiple clocks and multi-cycle I/O transactions to be able to capture and load a consistent state. Fortunately, a set of design rules and design wrappers have been recently proposed that can achieve safe task interruption [14]. The second challenge, which we seek to address in this paper, is how to save and restore the state of a design with buried state elements. Previous works on live migration and context switching worked around these challenges by using OpenCL/HLS flows and only allowing the switch/migration at specific points where no or pre-determined state needs to be captured [10], [13], [15]. This still limits the set of designs that can be checkpointed, and it cannot be used for FPGA debugging. Thus, we seek to close this gap and enable the checkpointing of a wide range of designs.

III. General Approaches for Checkpointing a Task With Inaccessible State Elements

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 [14]. 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.

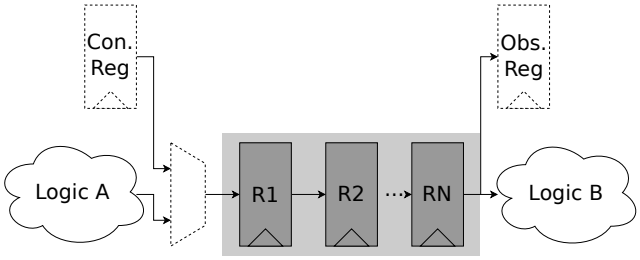

Fig. 1. Basic Approach. (Dark Gray: inaccessible registers; Dashed: added logic)

A. Basic Approach

Consider a task that has a chain of N registers whose state is inaccessible (e.g. hyper registers) as shown in Fig. 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 Fig. 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 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 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.

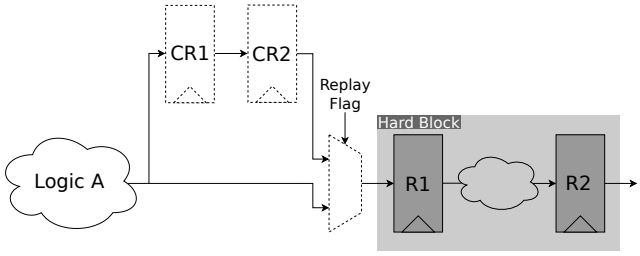

Fig. 2. Input Capture Approach. (N=2; Dark Gray: inaccessible registers; Dashed: added logic)

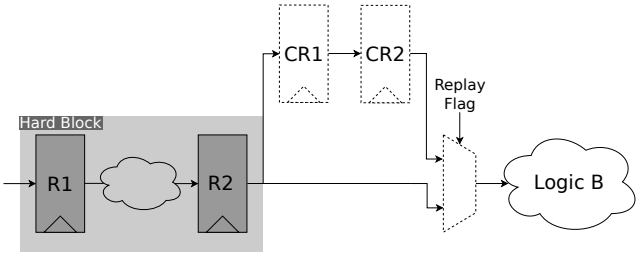

Fig. 3. Output Capture Approach. (N=2; Dark Gray: inaccessible registers; Dashed: added logic)

B. 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 Fig. 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.

C. 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 Fig. 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 extensions is discussed in Section IV-C1.

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.

IV. 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 we adapt the general approaches proposed in

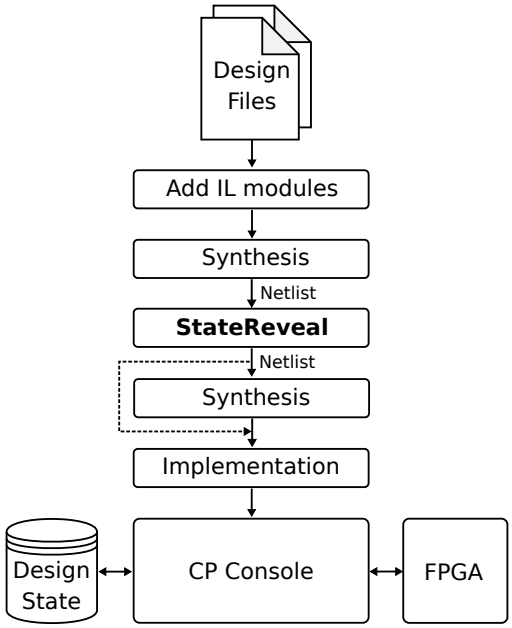

Fig. 4. StateReveal Tool Flow. (IL: Interruption Logic; CP: Checkpointing)

Section III to support the various configurations of Xilinx's BRAMs and DSP blocks.

A. Overview

StateReveal is written in python. The tool flow is shown in Fig. 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 assumes that the logic needed for safely stopping the design, denoted by *interruption logic* (IL), is already added to the design [6]. The interruption logic is based on the techniques proposed in [14].

StateReveal uses Pyverilog to parse the synthesized netlist and generate an Abstract Syntax Tree (AST) [16]. 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. For each hard block, StateReveal examines its configuration 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 can then be re-synthesized in Vivado to allow further synthesis-level optimizations or it can be implemented directly using the post-synthesis (implementation) steps of Xilinx's Vivado flow.

After that, the checkpointing (CP) console, which is the user interface of the checkpointing framework discussed in Section V, can then 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.

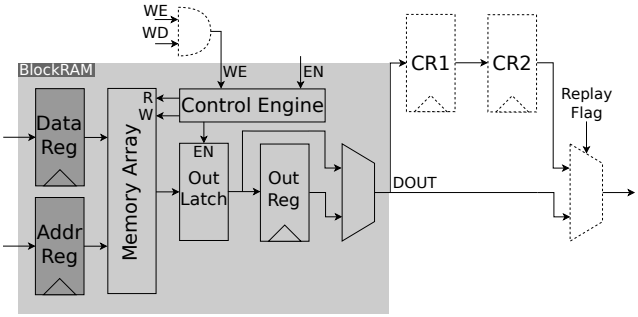

Fig. 5. A BRAM with output capture approach. (Dark Gray: inaccessible registers; Dashed: added logic)

B. Block RAMs

The block diagram of a Xilinx BRAM is shown in Fig. 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 [17]; 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 Section V, 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 III-C rather than the input capture approach

of Section III-B. 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 Fig. 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. However, we found 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 after the two additional cycles. 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. StateReveal connects the WD signal to the *interruption logic* that we add to the top module to stop the design.

After the second readback, our checkpointing framework creates a complete checkpoint 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.

C. DSP Blocks

The block diagram of Xilinx's UltraScale DSP block is shown in Fig. 6 [18]. 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

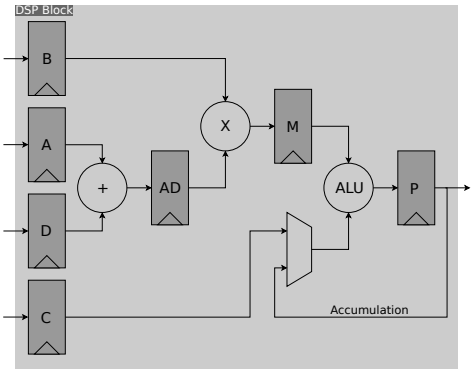

Fig. 6. Simplified view of Xilinx's DSP block. (Dark Gray: inaccessible registers)

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 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 Fig. 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.

1) 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.

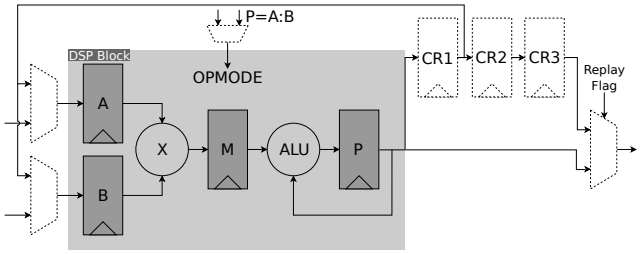

Fig. 7. A DSP block with accumulation and output capture approach. (Dark Gray: inaccessible registers; Dashed: added logic)

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 Fig. 7. When the checkpoint is loaded and before resuming the design, the added MUXes pass the captured P value to the A:B 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.

2) 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 III-B 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

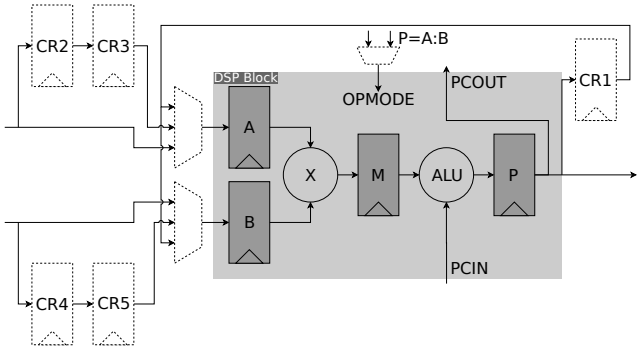

Fig. 8. A DSP block in cascade mode ($PCOUT = A * B + PCIN$). (Dark Gray: inaccessible registers; Dashed: added logic)

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.

3) 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 Fig. 8. This mode allows efficient implementations of filters and dot products. PCIN/PCOUT ports are not accessible via general-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 IV-C1 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 Fig. 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)

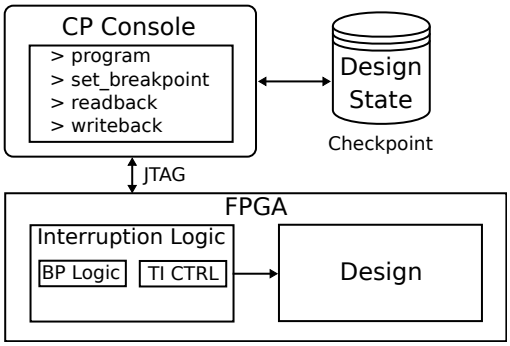

Fig. 9. The Checkpointing Framewrok

and a superset of Xilinx Virtex-5 DSP block (DSP48E).

V. Methodology and Evaluation

In this section, we first detail a checkpointing framework and use it to 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.

A. Checkpointing Framework

The checkpointing (CP) framework is based on the StateMover infrastructure [6] with some enhancements to enable multi-cycle capture and to provide the additional clocking which is required in the input and output capture approaches.

The CP framework supports Xilinx's UltraScale family, and it consists of the interruption logic and the CP console as shown in Fig. 9. The interruption logic contains a breakpoint (BP) logic and a task interruption (TI) controller which safely stops the design when a breakpoint is reached based on the techniques of [14]. The CP console is the user interface of the framework, and it interfaces with the interruption logic through Xilinx virtual I/Os [19] over JTAG. The console provides commands to set a breakpoint, read back the design on-chip state, and write back a checkpoint.

When a readback is invoked, the configuration frames are read out from the FPGA through JTAG and written to a readback file. The design state is then extracted from the readback file and written into a checkpoint file. The checkpoint file is a readable text file that contains the name and value of each state element in the design. When a writeback is invoked, the design state is read from the checkpoint file and embedded into either a full or partial bitstream file (both are supported). The FPGA is then programmed with the bitstream file, thereby restoring the design state.

The StateMover infrastructure supports capturing and loading on-chip BRAM memory contents, distributed memories, SRLs, and fabric (CLB) registers. We extend it to support capturing and restoring the output latch and output register of BRAMs. The logic location file, which contains information about the location of state elements inside the configuration frames, generated by Vivado 18.x or earlier does not contain information about the location of the BRAM output latch/register. Thus, we had to reverse engineer the location of these latches/registers to be able to capture and load them. This information has been added to the logic location file in newer Vivado versions (Vivado 19.x and later).

We modify the readback process to support multi-cycle capture as it is required by the output capture approach. The capture registers added by the output capture approach have to be read 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, the CP framework performs the required number of extra readbacks.

After each readback, the interruption logic clocks the design for an extra 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 so that the CP framework knows which capture registers to read at each cycle. The CP framework then 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 CP console.

We also modify the writeback process to provide the additional clocking required by the input capture approach. When the user invokes the *writeback* command, the CP framework writes back the checkpoint, and then toggles the clocks while appropriately setting the clock enables on hard blocks and input capture registers as detailed in III-B to recreate the necessary hard block buried state. After this warming up process is completed, the design can be resumed.

B. Results

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

TABLE I
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 II
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
								19	**35**	**1.18**	**0.95**

registers whenever possible (the naive workaround mentioned in Section III).

For each version, we use the CP 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, which means that a complete checkpoint is successfully captured and restored.

The basic designs are listed in Table I. 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 I.

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 [20]. 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 [21] and Spiral Hardware [22]. The designs are listed in Table II. 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 these designs with test circuitry that generates input data and compares the output with the golden one. 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 II. The timing overhead is low with a geometric mean of 5%. The area overhead depends on the number of hard blocks in the design. On average, StateReveal adds 19 LUTs and 35 FFs per hard block. The area overhead varies from a 1% to 33%, with a geometric mean of 18%.

VI. Conclusion and Recommendations

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. 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, 19 LUTs and 35 FFs per hard block to make buried state accessible, and its average timing overhead is only 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 III-A 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.

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 paper, 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 $\sim$290 pipeline register bits, and a BRAM contains $\sim$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 impractical to include them in the configuration architecture. Thus, StateReveal techniques will probably be necessary even in future FPGA families.

Acknowledgment

This work was supported by the NSERC/Intel Industrial Research Chair in Programmable Silicon, a Connaught Scholarship, and the Canadian Foundation for Innovation.

References

[1] O. Knodel, P. R. Genssler, and R. G. Spallek, "Migration of long-running tasks between reconfigurable resources using virtualization," *SIGARCH Computing Architecture News*, vol. 44, no. 4, pp. 56–61, 2017.

[2] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2007.

[3] 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.

[4] H. D. Foster, "2018 FPGA functional verification trends," in *International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2018, pp. 40–45.

[5] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *International Symposium on Computer Architecuture (ISCA)*, 2014.

[6] S. Attia and V. Betz, "StateMover: Combining simulation and hardware execution for efficient FPGA debugging," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2020, pp. 175–185.

[7] 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.

[8] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 894–897.

[9] D. Ly-Ma, "Live migration of FPGA applications," Master's thesis, University of Toronto, 2019.

[10] A. Bourge, O. Muller, and F. Rousseau, "Generating efficient context-switch capable circuits through autonomous design flow," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 1, pp. 9:1–9:23, 2016.

[11] B. L. Hutchings and B. E. Nelson, "Unifying simulation and execution in a design environment for FPGA systems," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 9, no. 1, pp. 201–205, 2001.

[12] D. Kim, C. Celio, S. Karandikar, D. Biancolin, J. Bachrach, and K. Asanovic, "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.

[13] A. Vaishnav, K. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for FPGAs using OpenCL," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2018.

[14] S. Attia and V. Betz, "Feel free to interrupt: Safe task stopping to enable FPGA checkpointing and context switching," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 1, 2020.

[15] A. Vaishnav, K. Pham, and D. Koch, "Live migration for OpenCL FPGA accelerators," in *International Conference on Field-Programmable Technology (FPT)*, 2018.

[16] S. 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.

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

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

[19] *PG159: Virtual Input/Output*, Xilinx, Inc., 2018.

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

[21] Xilinx. (2016) Open source HLx examples. [Online]. Available: https://github.com/Xilinx/HLx_Examples/

[22] F. Franchetti, T. Low, D. Popovici, R. Veras, D. Spampinato, J. Johnson, M. Püschel, J. C. Hoe, and J. Moura, "SPIRAL: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, 2018.